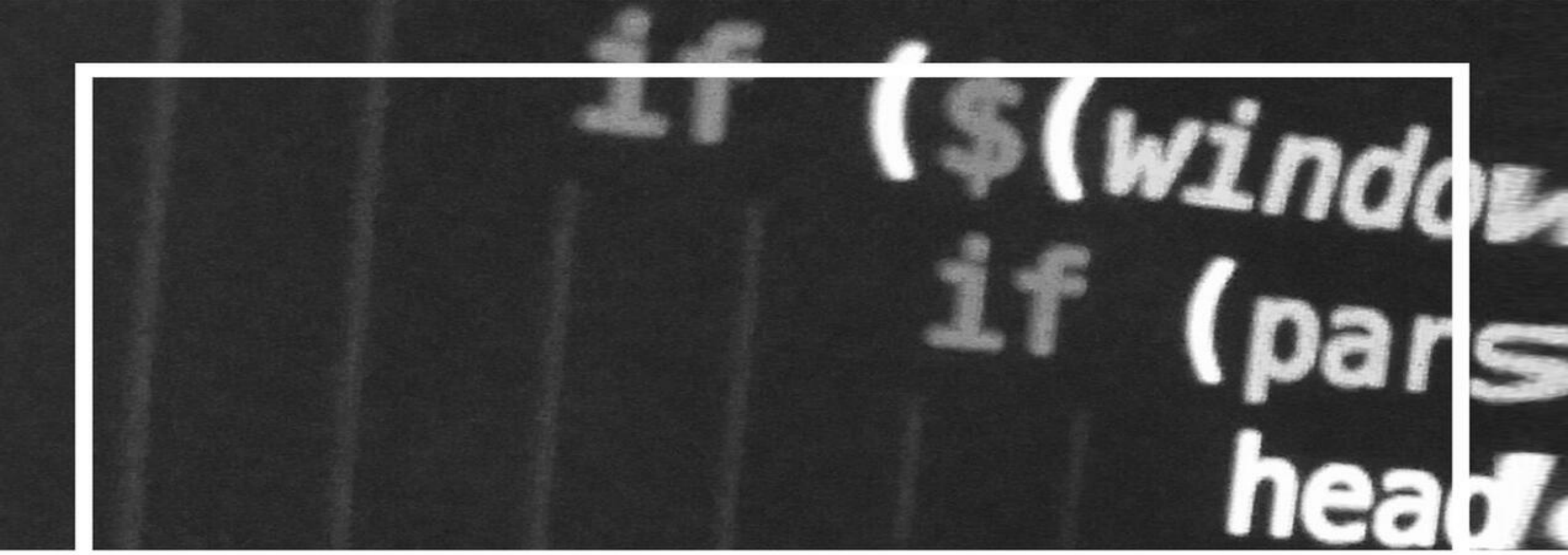
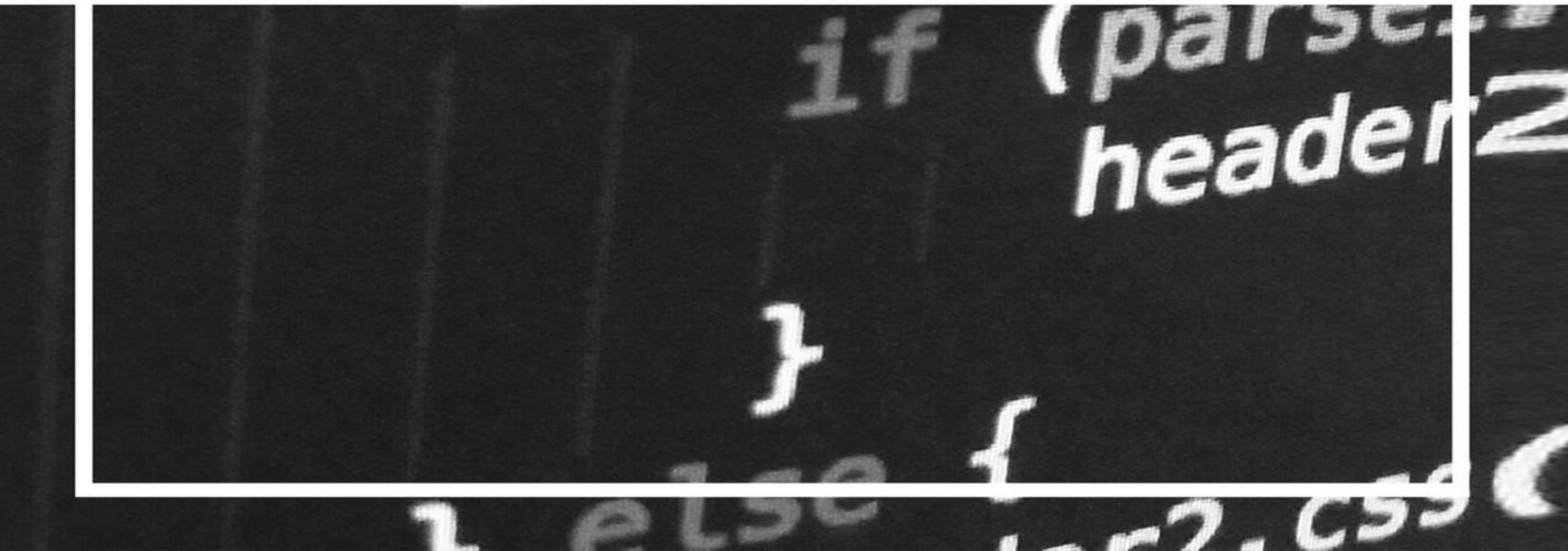
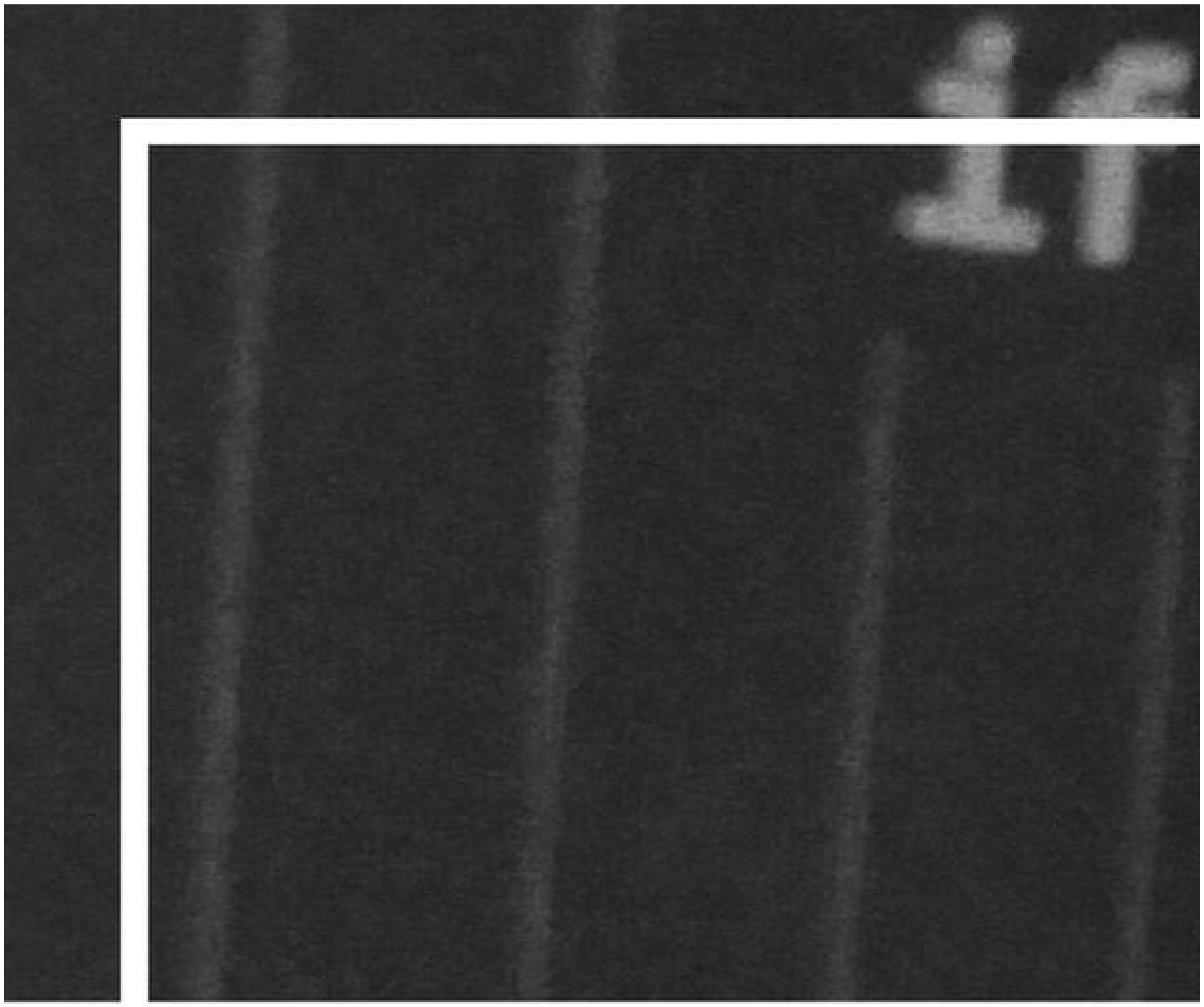


BASH GUIDE for Beginners

Everything You Need To Know About Bash Scripting





BA

GU

Introduction Motive

Why, when there are various coding assets as of now, did I compose this aide? The appropriate response is straightforward: models. I have consistently learned best by concentrating on models. This is an aide for those out there like me that need to see an assignment performed before they can appropriately copy it.

Other aides adopt various strategies. Best case scenario, the general language and absolutely broad cases utilized all through course book like assets are mind-numbingly dry. Even from a pessimistic standpoint, they hamper learning. On the flipside, the endeavor of certain advisers for give a novice well disposed prologue to the material take such an elevated view that any viable insights about the how are lost to stream graphs and extensive clarifications of the why.

This aide endeavors to overcome any barrier between the two; to give enough of the what and why so this present reality how has setting. Hence, clarifications are commonly short and followed by various examples.

Do not endeavor to remember everything about —minute subtleties are effortlessly neglected, and simply turned upward. Rather, endeavor to see how every one of the models capacities. All language structure tables and summed up cases have been duplicated toward the finish to make a “Fast Reference” that can be gone to when subtleties definitely fall by the wayside.

Audience background

No programming, coding, or prearranging information is essential for the perusing or comprehension of this aide. Actually, no information on the Linux working framework or terminal itself is required. All things considered, obviously any past dialects, experience, or specialized information will hurry the speed at which the data introduced is acquired.

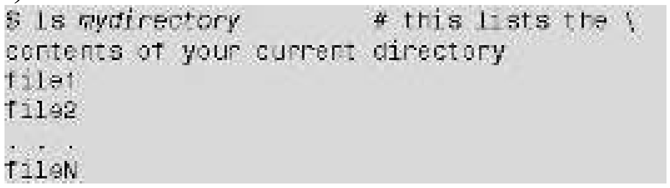
Formatting

The designing utilized all through the aide has been normalized. The principal utilization of significant ideas are in strong. Orders, contentions, boundaries, filenames, ways, and comparable phrasing are italicized.

Code, scripts, order line input, and the result for every one of them are held inside code blocks. Normally, code can be gone into your own terminal word for word. Exceptions are items such as *yourfile* , *my_domain* , and others, which should be replaced with the entry relevant to your system, working directory, input or data file, etc.

Anything continued by a hash mark, # , is a remark and ought not be entered (however you might enter it in the event that you decide; it will have no impact). An oblique punctuation line, \ , permits code to be forged ahead the following line without it being broken by a newline.

Output of in excess of a couple of lines will ordinarily be shortened by an ellipsis (. .
.).



```
$ ls mydirectory        # this lists the \
contents of your current directory
file1
file2
. . .
fileN
```

Chapter 1: History

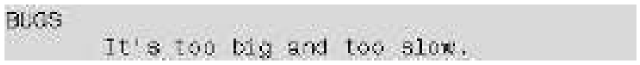
Early interfacing with PCs —in the post wire-attachment and punch-card era — was cultivated by entering text orders into an order line mediator. The capacity of this order line mediator, or shell, is to converse with the fundamental working framework. At the core of the shell is the part, which converses with actual gadgets like printers, consoles, mice, and screens, through their different drivers. Along these lines, using a shell, a client can handle, execute, and computerize practically any undertaking an advanced PC is fit for performing.

Shells

The main shell was the Thompson shell, composed by Ken Thompson. Because of the constraints of the frameworks at that point, it was essentially called sh. Later almost a time of utilization, it was removed by the more component rich Bourne shell, composed by Stephen Bourne. To make things mistaking for the nerds of that period, he kept a similar name, sh. It would take a touch more than one more decade before the Bourne shell was surpassed by the—hang tight for it—Bourne-again shell, or Bash.

Bash

Though initially delivered in 1989, Bash has experienced a long series of updates, redesigns, and corrections, every one of which gave it admittance to always valuable and incredible assets. Moreover, as utilities were created to deal with specific assignments, or other contending shells delivered new highlights, Bash coordinated a large number of them into itself. As of this composition, Bash is on rendition 4.3. Its man page (manual page), comprises of more than 57 hundred lines, and close to the end contains the accompanying (practically diverting) line.



```
BUGS
       It's too big and too slow.
```

Other shells

Although many other shells exist, such as csh, ksh, and zsh, this guide assumes you are using Bash. Note that you can utilize this aide without approaching a Bash shell. You will notwithstanding, should know that not all things describe in this book is compact; that is, a content that runs in Bash won’t really work effectively (or by any means) in another shell, or the other way around. *[[is a perfect representation of this* . Do not despair though! All the concepts, from variables and arrays to conditionals and loops, are transferrable—you may simply need to learn a little shell-specific syntax.

You can check which shell you are utilizing by giving the accompanying order, which prints the $SHELL climate variable.

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

If your PC is utilizing some different option from/canister/slam, you can change your default shell to Bash (gave obviously Bash is introduced on your framework) by giving one of the orders below.

The first sets the shell for a particular user (change *username* to reflect your username, i.e. the output of whoami) in the configuration file, */etc/passwd* . The second sets the shell for the currently logged-in user upon login through the use of that user's *~/.profile* configuration file.

```
$ usermod -s /bin/bash username
$ echo "SHELL=/bin/bash exec /bin/bash \
--login" >> ~/.profile
```

What a shell is (and isn't)

At its generally fundamental, a shell, in spite of near on forty years of improvement, is only an order mediator. Notwithstanding how complex the orders are,the means by which tangled the rationale, and how elusive the references, every one of the a shell permits you to do is issue orders intuitively or non-intelligently (more on this later).

A shell isn 't your PC, working framework, or show. Nor is it your proofreader, terminal, mouse, console, or kernel.
When you dispatch Bash, what you are really doing is dispatching a terminal emulator window with Bash running inside it. Like different projects, you can interface with the terminal emulator window (and in this way Bash) through utilization of your mouse and console. Typing and running commands allows you to speak to your system, be it parts of the operating system such as configuration files or window managers, or the kernel itself to say, start, stop, or restart a process.

Using Bash

As referenced above, Bash permits you to give orders intelligently or non-interactively.
Bash's intuitive mode is reasonable the one you are the most acquainted with. It comprises of a terminal emulator window with Bash running inside it. At the point when you type an order, it is written progressively (or close going) to your terminal and showed on your monitor.
Non-intuitive mode is just Bash running behind the scenes. This doesn't mean it can't (or won't) yield anything to the screen, compose or read to or from a document, or anything like that—it can assuming you tell it to! Nonintelligent mode basically reduces to a consecutive posting of orders to be performed.
This posting of orders is known as a Bash script. Those of you with a foundation in Microsoft Windows might perceive the terms DOS prearranging or bunch document. The two are one in the equivalent, however neither compares to Bash prearranging (which is the reason Microsoft in the end delivered the Powershell, and is as of now attempting to permit local execution of Bash on its OS).

Prompts
There are three significant kinds of prompts. Each starts with an alternate person. The ampersand image, and, shows you are running a Bourne, POSIX, or Korn shell. A percent image, %, implies you are running a csh, or zsh shell. What's more a hash, #, tells you are executing orders as root—be extra careful!

```
$
%
#
```

Without a doubt your brief is significantly longer than a solitary image. The default for a large number of the more famous GNU/Linux conveyances is your username at the machine name you are running on, trailed by the current working index and afterward the shell prompt.

Here is a model for the client aries, chipping away at the PC damages PC, which is running a Bash shell, and has a current working catalog of ~ (a tilde grows to/home/username, for this situation/home/aries).

```
mars-laptop@aries ~ $
```

For the purpose of clearness, usernames and machine names will be overlooked from the rest of this aide except if required.

Help

Before you go any further, make certain to familiarize yourself with orders that permit you to query help documentation.
Man, short for manual, shows the (who could have imagined) manual pages for the
contention (erring on this later) that quickly follows .

```
$ man bash
```

Note that for different orders there exist more than one man page. For instance, the order printf has two man pages (1 and 3).

```
$ man printf
$ man 3 printf
```

Help shows the man pages for Bash 's constructed ins and catchphrases. Here we access the assistance pages for the restrictive squares, [[. Both shell fabricated ins/catchphrases and restrictive squares will be examined inside and out further on.

```
$ man [[
No manual entry for [[
$ help [[
[[ ... ]]: [[ expression ]]
    Execute conditional command.

    Returns a status of 0 or 1 depending on
the evaluation of the conditional
. . .
```

If you want to figure out what type a specific order is, you use type.

```
$ type [[
[[ is a shell keyword
$ type [
[ is a shell builtin
$ type cat
cat is /bin/cat
$ type ls
ls is aliased to `ls --color=auto'
```

Whatis prints a short, single line depiction of the contention that follows. Adding a switch (an order explicit contention) to man permits it to play out the equivalent function.

```
$ whatis bash
bash (1)             - GNU Bourne-Again
Shell
$ man -f bash
bash (1)             - GNU Bourne-Again
Shell
```

Apropos scans a man page 's diminutive portrayal for the watchword that was passed to it. Once more, adding a change to man permits it to do the equivalent thing.

```
$ apropos bash
bash (1)             - GNU Bourne-Again
Shell
bash-builtins (7)    - bash built-in
commands, see bash(1)
. . .
$ man -k bash
bash (1)             - GNU Bourne-Again
Shell
bash-builtins (7)    - bash built-in
commands, see bash(1)
. . .
```

Chapter 2: Editors

There are a wide range of editors accessible, from barebones, terminaljust editors to undeniable programming interfaces complete with form control and mechanized compiling.

Unless you get into remote administration o f your (or other peoples') computers, rarely will you encounter a situation that forces you to use one editor or another. Beyond that, choosing a particular editor is largely a matter of personal preference. My suggestion is to try many different editors—installation of a new editor is rather straightforward thanks to modern day package managers—and pick one you think will work for you (in terms of features, appeal, etc.). Then stick with it! The longer you use an editor, the deeper understanding you get of its advantages and pitfalls, its shortcuts, and nuances (or perhaps nuisances. . .).

One basic thing before I plunge into the actual editors: don ’t utilize Microsoft Word, Microsoft WordPad, OpenOffice, Libreoffice, or whatever other WYSIWYG (what you see is the thing that you get) content tool for prearranging or programming. Behind the pretty façade lurks invisible formatting tags that can cause *major* headaches (such as Microsoft Office’s carriage returns). At the point when these documents are perused by a GNU/Linux order mediator, things will more often than not break undetectably. Use them at your own hazard! Assuming you are on a Windows-based machine and need another option, I profoundly propose downloading and introducing the free source code supervisor, Notepad++.

And presently, in no specific request, I present four editors and their essential usage.
As referenced previously, there are numerous others out there—joe, pico, jed,
gvim, NEdit, overshadow, the rundown continues. Assuming you don’t observe one you like immediately, don’t be discouraged.

Emacs

Boasting a graphical UI fit for showing numerous concurrent, constant cushions, emacs is a long-term client top choice. Though you can issue commands through its file menus, a never-ending series of keyboard shortcuts exist (nearly two thousand as of last count). Numerous a web comic makes fun of the sheer number of easy routes and the dark assignments they perform.

Installation is normally performed through an order line bundle director (conciliatory sentiments to non-Debian based clients out there, however for simplicities purpose I needed to pick one bundle chief; adept/fitness it is).

```
$ sudo apt-get update
$ sudo apt-get install emacs
```

Upon opening emacs interestingly, you will get a sprinkle screen. Essential use is pretty much as straightforward as composing what you need, then, at that point, utilizing the record menu to save.

VIM

Originally made as VI, this manager has since been moved up to VIM (Vi Improved). *VIM is the primary opponent of emacs in the long-standing Unix editorial manager wars* . No, it isn't graphical (except if obviously you introduce gvim), and negative, it isn't as lovely, however you would be unable to find a more enthusiastic client base.
Originally composed for the days when window supervisors didn't exist and modems were evaluated in twofold or triple-digit baud rates, *vim runs completely inside a terminal* . Like Dwarf Fortress, the ASCII-based Dwarven city the board game, it has an incredible expectation to absorb information. Try not to allow that to deter you from attempting it, nonetheless. The nuts and bolts are not difficult to learn, and in the possession of an expert, the outcomes aren't anything short of magical.

```
$ sudo apt-get update
$ sudo apt-get install vim
```

VIM begins in “ordinary mode.”
To compose orders, you should enter “embed mode” by hitting [i]. You would then be able to type utilizing your ordinary console schedules. At the point when prepared to save, reemerge “typical mode” by hitting [ESC]. A couple of fundamental orders are show in the table beneath. For further developed use, counsel the manual or an on the web “cheat sheet.”

VIM usage:
Command Description

i Enter “insert mode” [ESC] Enter “normal mode” :w Write file (save) ZZ Write file and quit :q Quit
:q! Quit without writing file

Nano

Nano is a decent, lightweight terminal manager. Introduced naturally (if not, just follow the past well-suited get order, however trade vim for nano) on various conveyances, it is very simple to utilize. All the order alternate routes are shown in that general area on the lower part of the screen for your convenience.

The caret (or cap) going before the order letter implies you ought to hit *[CTRL] first, trailed by the key of your decision, say [X], to quit .*

Gedit

The last manager covered is Gedit, which is additionally accessible for establishment on Windows-based PCs. It is a decent, broadly useful scratch pad, order and content supervisor, record watcher, and the sky is the limit from there. As in the past, in the event that it isn't introduced on your framework, you can undoubtedly introduce it with your bundle manager.

Keyboard easy routes exist, and are displayed in dark close to their order in the fitting document menus. Generally, the alternate ways are as old as ones you would find on a standard Windows machine—to be specific [CTRL][S] to save, *[CTRL][Q] to stop* , thus on.

Chapter 3: The Basics Comments

Anything went before by a hash (#) is a remark, and isn't executed by Bash. Thus, the accompanying produces no output.

```
$ # Hi there!
```

Comments are rarely printed, in any event, when included later an order or contention. Here, reverberation is utilized to parrot back a word that contains a following comment.

```
$ echo Hi!          # Print the text, "Hi!"
Hi!
```

Commands

At its heart, Bash, regardless of whether utilized intelligently or run by means of a non-intuitive content, is basically an order line translator. Each order demonstrates a guidance or errand to be conveyed out.

One of the most essential orders is ls . Its responsibility is to list the substance of an index. You can confirm that through utilization of a couple of orders addressed before, man, and whatis.

Incidentally, in light of the fact that man and whatis are themselves orders, composing the accompanying into a terminal is impeccably valid.

```
$ man man
$ whatis whatis
```

Arguments

Arguments mean the world (and whatever) follows a command. Arguments are isolated by whitespace (spaces, tabs, and newlines). To harden this idea, three orders are presented, mkdir, compact disc, and touch.

Mkdir is another way to say "make registry" and takes at least one contentions . Every contention isolated that is by a whitespace is deciphered by the order as the name of another index to make. This division is called word splitting.

```
$ mkdir test new_dir
$ ls
new dir test
```

Word parting is an idea urgent to Bash . To play out any errand on something containing any whitespace, you should consider word parting and either quote it or use get away from characters, as canvassed in the resulting sections.

In request to move into the test index we made over, the order compact disc, another way to say "change registry" is utilized. It takes precisely one contention—the name of the registry you which to move into.

```
$ cd test
$ ls
```

The explanation our ls order returns nothing is on the grounds that there are no records or indexes in it yet!
For those of you with some involvement in GNU/Linux frameworks, you're most likely saying, "However there are two secret things in this catalog!" And you would be correct. You can confirm this by passing a contention

(additionally called a banner) to ls. That banner is - (a dash followed by a lowercase a). It tells our ls order not to disregard passages that beginning with a spot (which are concealed by the working framework by default).

```
$ ls -a
.  ..
```

These “dab catalogs” are exceptional in that they allude to the current index itself (a solitary speck), and its parent registry (a twofold dab). Documents can likewise be “specked” and stowed away. They are then called “spot files.”

For instance, you can move from the current catalog to its parent registry with:

```
/home/mars-laptop/test $ cd ..
/home/mars-laptop $
```

In request to make a document (or three or twelve) you pass them as —you got it!— contentions to our third order, contact, whose work it is to change the last altered season of a record. Nonetheless, since out records don’t yet exist, *contact makes another document for each argument* .

```
$ touch one fish two fish red fish blue fish
$ ls
blue fish one red test two
```

If the result of this apparently straightforward assertion befuddles you, have no fear!
Several things are happening at once. When *touch* is executed, Bash performs
word splitting to find all the words in the line. They are then passed to *touch* and
processed one at a time. This means that the file *one* is created first, followedby
fish , then *two* , then *fish* again… wait a minute! Because the file *fish* was just
created a millisecond ago, instead of creating another file with the same name,
touch performs its duty faithfully and updates the last modified time of the file

fish . Next up is the file *red* , *fish* again, *blue* , then *fish* yet again.
Tricky,huh? It is critical to take note of that in spite of how much whitespace between
contentions, Bash just uses them to delimit one contention from another, nothing more.
First, eliminate the documents you made you recently made in the test catalog
with rm.

```
$ rm *
```

The mark (*), is known as a glob and has a unique implying that we will cover inside and out later on. For the time being, consider it an extraordinary “match everything” character. Hence, our past order is telling rm to play out its activity (eliminate records or registries) on all that it experiences in the current registry. As I’m certain you think, you ought to be extra cautious with this, as there is no fix button.

Let’s have a go at utilizing contact once more, this time with spaces.

```
$ touch one two     three               four
$ ls
four one three two
```

Notice that how much whitespace doesn’t make a lick of difference. I can hear you thinking, “Yet imagine a scenario where I need to make a document with spaces in it?” The appropriate response is to utilize quotes.

Quotes

To show how statements work without making a wreck of records and indexes, the reverberation order is especially useful. Reverberation works very much like a reverberation, in actuality, and reverberations back whatever contentions you pass to it.

```
$ echo It was a dark and stormy night
It was a dark and stormy night
```

Okay so that worked similarly as we needed it as well. *Reverberation took every one of our contentions each in turn and printed them each in turn* *with a solitary space in the middle* . But what if, as in our *touch* command a minute ago, there is more than once space?

```
$ echo It  was a  dark and   stormy night
It was a dark and stormy night
```

Though it understands right, it is feeling the loss of the additional areas, needed or not. I'm certain you can perceive how this can immediately turn into a migraine on the off chance that you are not anticipating it. Enter statements. Quotes transform everything inside them into a solitary argument. Now as opposed to perusing a word, tracking down a space and thinking the

following word is another contention, Bash regards the cited segment as one single, long argument.

```
$ echo "It  was a  dark and   stormy night"
It  was a  dark and   stormy night
```

That 's more similar to it!
Why might you at any point need this activity? We should take a look. The situation: you are simply putting the last addresses your last, finish ofthe-semester report. The telephone rings. It's your chief. He lets you know that except if you document your TPS Report right that moment, you're terminated! With a murmur, you hang up, compose the bothersome tps report, and email it on its joyful way. As any rational individual would do, you promptly erase it so it's not jumbling up your home PC and at the forefront of your thoughts for the remainder of the night.

```
$ ls
tps report  report
$ rm tps report
rm: cannot remove 'tps': No such file or
directory
```

You go to open your last semester report just to think that it is' no longer there, yet that terrible tps report is!

```
$ ls
tps report
```

The problem was that your tps report had a space in the filename. Because you did not quote it, Bash read in the arguments and performed word splitting: it separated the arguments by whitespace and operated on them one at a time. *tps was the first argument* . Since there was no file named tps in the directory, it predictably failed. When it went on to the next argument, *report* , it found the file present, and deleted it. Thus, you deleted what you wanted to save, and saved what you wanted to delete.

The proper way is to use quotes.

```
$ ls
tps report  report
$ rm "tps report"
$ ls
report
```

Much better!

Get into a habit of double quoting any string that does, will, or may contain *spaces, tabs, newlines, or special characters* . You'll thank yourself for it later.

Backslash escapes

An "alternative" to quoting is to backslash escape any c haracter than has special meaning (essentially all non-alphanumeric characters). Though it works, it is extremely tedious, causes problems if not handled correctly, and is not easily readable. The example below is one of the very few times you will see escape characters in this guide.

```
$ ls
tps report
$ rm tps\ report
$ ls

$ mkdir aries\'\ five\ space\ \ \ \ \ 
dir\!/
$ ls
aries' five space      dir!
```

Strings

Whether you realized it or not, virtually every thing you've seen thus far is a string. A string is simply a series of characters (letters, numbers, symbols, and punctuation marks). Unless explicitly told otherwise, Bash treats virtually everything as a string.

The name of the command is a string. The arguments that follow are each strings in their own right. A quoted sentence is a string. Each of the aforementioned strings can be thought of as a substring to the entire entered command, arguments and all. Heck, an entire file can be thought of as a single string.

This is an important concept to grasp, because as powerful as computers are, they cannot reason. They do not see things as we do.
When we see a sentence like, "Hi, how are you?" we understand it is a question, and one asking how we are doing.
On the other hand, Bash only sees a single string, "Hi, how are you?"
After word splitting, it sees sequence of strings, each separated by a whitespace, and made up of individual characters.
Bash's view
String 1
Hi, how are you?
Word 1 Word 2 Word 3 Word 4
Hi, how are you?
Note that two of the words, *Hi,* and you? are complete gibberish if taken literally—you will not find either in the dictionary. Bash does not care. It sees a quotation mark, and per the rules of its programming, takes all the following characters as part of the same word until it encounters a whitespace, at which point it begins the second word, and so on until the closing quotation mark. Note that quotation marks are one of the many special characters in Bash, and have a nonliteral meaning.
This means is that the onus is on you, the programmer, the scripter, the coder, to ensure that whatever you type not only makes sense to you, but more importantly, it makes perfect sense to Bash's rigid rules. If done incorrectly, the best result will be that Bash throws an error and ceases to execute your code. At worst, the syntax you entered will mean something completely different than what you intended and you will find yourself in deep trouble.

IFS

This section utilizes many concepts presented in later chapters of this guide. It is recommended you skip it and return once you have completed the guide. For now, simply be aware that IFS stands for Internal Field Separator. Its purpose is to figure out where and how to perform word splitting based on a list of delimiters. The default IFS value (delimiters) are space (' ')*, horizontal tab ('\t')* , and newline ('\n').

There are a couple ways to check the IFS settings. The first uses printf, the second od, and the third, set.

```
$ printf %q "$IFS"
$' \t\n'
$ echo -n "$IFS" | od -abc
0000000  sp  ht  nl
        040 011 012
             \t  \n
0000003
$ set | grep ^IFS
IFS=$' \t\n'
```

Setting the IFS value is a matter of passing the special characters you want as delimiters to IFS.
Here, the IFS value is set to the null string.

```
$ IFS=
```

This is useful if you want to preserve all whitespace, perhaps while reading from a file (both loops and the read command are covered in later chapters).

```
$ while IFS= read -r
> do
>    line=$REPLY
>    echo "$REPLY"
> done < somefile
```

Another example is to set IFS to a colon, commonly used to delimit columns in spreadsheets. The triple less-than sign is called a here string, and is covered near the end of the guide.

```
$ IFS=: read -r column1 column2
<<<"Budget:6000"; echo "$column1 - $column2"
Budget - 6000
```

Commas are also common delimiters. Here, the IFS value is set as such.

```
$ set | grep ^IFS
IFS=$' \t\n'
$ IFS=,
$ set | grep ^IFS
IFS=,
```

Notice how the IFS value is permanently modified after you set it? This is typically not useful. In most cases, you want to change the IFS value for a single command, such as reading input, but leave the value at its default for the remainder of the script; else your script will behave strange, as word splitting is no longer being performed the way you expect.

One option is to set a variable with the current IFS value, and then return it later.

```
$ ORIG_IFS=$IFS
$ IFS=,
$ perform your commands here
$ IFS=$ORIG_IFS
```

A second is to temporarily set a new IFS value by surrounding it with double quotation marks.

```
$ set | grep ^IFS
IFS=$' \t\n'
$ IFS="," read -r column1 column2
<<<"Budget,6000"; echo "$column1 ---
$column2"
Budget --- 6000
$ set | grep ^IFS
IFS=$' \t\n'
```

See how the IFS value only changed for that single command? That’s exactly what we wanted.
Note that an IFS value corresponds to its value and exactly its value. Thus, if two delimiters are encountered, it corresponds to an empty field between the two.

```
$ IFS="-" read -r column1 column2 column3
<<<"Budget--6000"; echo "Col1:$column1
Col2:$column2 Col3:$column3"
Col1:Budget Col2: Col3:6000
```

You always can return IFS to its default value by issuing:

```
$ IFS=$' \t\n'
```

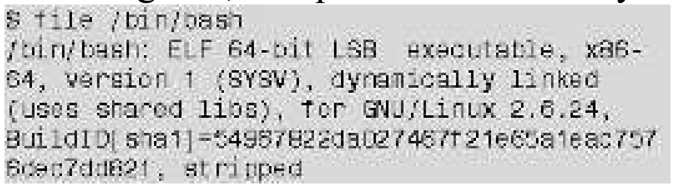
Chapter 4: Terminology Executables

If you have spent any time on a Windows machine you are likely familiar with the term executable. They are files that end with *.exe* . When double

clicked, a program or application runs. Executables are the result of a programmer writing code in a high level language such as C++, then compiling the code and turning it into machine code instructions (typically assembly language) for a processor to run. Once compiled, the program is no longer directly editable by a human.
Rather, the original high level code must be edited and the executable recompiled.

On a GNU/Linux based machine, an executable typically goes by the term, binary. Other terms used are applications and external commands. As with all executables, they are not directly editable. Inside a terminal running Bash, when a binary is called by name or path, it runs. You can determine if a something is a binary through use of the command, file.

For this guide, the quintessential binary is Bash itself.

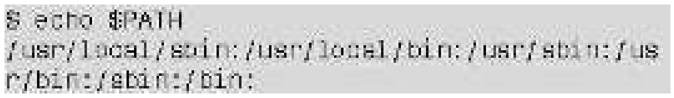


```
$ file /bin/bash
/bin/bash: ELF 64-bit LSB  executable, x86-
64, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=54957822da027467f21e65a1eac757
6dec7dd621, stripped
```

Other binaries include touch, ls, mkdir, emacs, and vim.

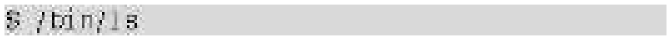
Paths

The way Bash knows where to find binaries and other items refers to by name is through paths.
Paths store a series of, well, paths. These paths lead to places where binaries are commonly stored. *$PATH is an environment variable (as evident by the dollar sign preceding a series of all capital characters; much more on this later)* . In Linux, all Bash has to do when you call, for example, *ls* , is look in a few pre- defined directories. You can see your full $PATH variable by issuing:



```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/us
r/bin:/sbin:/bin:
```

If not for $PATH, every time you wanted to call ls, you would need to specify exactly where it was stored:



```
$ /bin/ls
```

This is important to understand because when you start writing scripts, you will not be able to call your script by name like you call ls. The reason is because it will not be stored in one of the places named in your path!

Well, it should not be at any rate. Adding a random folder to your $PATH makes your system dramatically less secure. It introduces the ability for a malicious user (or a careless mistake) to wreck your system. Imagine if you added your script storage folder to the front end of $PATH, then accidently called a bulk, non-interactive script you named rm (thus clobbering, or colliding with the existing binary, *rm)* . Bash would search for the name you gave it (rm), find your script first (instead of the system binary, *rm*), then run it without a care for what it deleted. You can see the chance for misuse and abuse!

If, after all that, you are still bound and determined to add a directory to your path variable, at least use something consistent like /opt/bin or */home/username/bin and always place the new path at the end of the current path (because the PATH environment variable is searched in order, from left to right)* .

Builtins

Now that you know what binaries are, a builtin is a piece of cake. As the name suggests, a builtin is quite literally something built-in to Bash. Because they do not require searching of the PATH environment variable, then forking off a separate process to run the binary, they are typically execute quite a bit faster.

Examples of shell builtins are: alias, cd, echo, read, and type.

Scripts

Scripts are the focus of this guide. All a script is is a series of commands stored in a file. Unlike a binary, this file written in a high level language, is directly editable, and does not require compiling. You can think of Bash as the liaison or interpreter between your high level script and the low level code required by the system's processor.

Scripts all begin with the same line, called the shebang:

```
#!/bin/bash
```

When you call your script, the Linux kernel reads the first line, finds the *shebang, and is told that it should look for the liaison/interpreter at the location /bin/bash .*

Despite the many similarities between shells such and dash, sh, and bash, and zsh, ksh, and csh, there are differences. If you have written and tested your script using Bash, do not assume it will work in dash or sh (though many will).

Even though they are not binaries, scripts need to be given executable permissions before they are run. That is unless you want to call it as an argument to bash every time you run it.

```
$ /bin/bash your_script
```

or, since bash is in your PATH,

```
$ bash your_script
```

The way you give executable permission to a script is through use of the binary, chmod.

```
$ chmod +x your_script
```

You can now run the script by name and residing directory (because chances are your current working directory is not in the $PATH variable, right?).

```
$ ./your_script
```

Aliases

Aliases are shortened commands. That's it. Nothing fancy or dramatic. Aliases are simply a way to save time when having to type a long command over and over again, or save you from remembering all the switches and arguments a long command needs. Running alias without any arguments lists all aliases currently defined for your user.

```
$ alias
```

The same command also defines new aliases.

```
$ alias upgrade='sudo apt-get update \
&& sudo apt-get upgrade'
```

Now when upgrade is entered into a terminal, Bash replaces the string upgrade with the string “sudo apt-get update && sudo aptget upgrade”. It is that simple. Below are a few (hopefully) useful aliases.

Standard aliases:

```
$ alias egrep='egrep --color=auto'
$ alias fgrep='fgrep --color=auto'
$ alias grep='grep --color=auto'
$ alias l='ls -CF'
$ alias la='ls -A'
$ alias ll='ls -alF'
$ alias ls='ls --color=auto'
```

Update and upgrade:

```
$ alias uu='sudo apt-get update && sudo apt-
get upgrade'
```

Install a package:

```
$ alias ins='sudo apt-get install'
```

Remove a package:

```
$ alias rem='sudo apt-get remove'
```

Change to the parent directory(s):

```
$ alias ..='cd ../'
$ alias ...='cd ../../'
$ alias ....='cd ../../../'
$ alias .....='cd ../../../../'
```

Backup using rsync:

```
$ alias backup='sudo rsync -av --exclude-
from=/home/aries/rsync-excludes / \
/mnt/exthdd/backup'
```

Note that if you are using rsync, you should exclude temporary filesystems, removable storage and the like (/proc, /run, /sys, / dev, /media, /mnt, and /tmp).
Backup using tar:

```
$ alias backup='sudo tar cvpzf \
backup.tar.gz --exclude=/backup.tar.gz \  -
-exclude=/proc --exclude=/sys \          --
exclude=/dev/ --exclude=/run \          --
exclude=/media --exclude=/mnt /'
```

SSH:

```
$ alias connect='ssh -X -c blowfish \ -C
username@domain.net'
```

Shutdown:

```
$ alias sd='sudo shutdown -h now'
```

Reboot:

```
$ alias rb='sudo reboot'
```

Functions

Functions are basically just mini scripts. They can handle variables, commands, arguments, and much more. As your Bash coding skills progress, your scripts will typically take on the look of a chain of islands (the functions), each with nice little landing and takeoff strip for planes from the mainland (the script itself) to deliver, retrieve, and perform tasks on packages (data). Functions will be covered once script basics have been covered.

Standard streams

There are three standard streams, Each has a specific and distinct purpose. Stdin is the first, and stands for "standard input." By default it refers to input from a keyboard. Inside a terminal running Bash, all typed text commands are entered via the keyboard, and thus via standard input. The file
descriptor (more on this topic will be covered later on) for standard input is 0 (zero).
Later chapters of this guide cover other modes of input such as reading from a file, a process, or the output of a previous command. While not entered via the keyboard, they are redirected to standard input through use of a special character such as a pipe (|) or multiple less-than signs (<*, <<, <<<)* .
Stdout is the second stream and refers to "standout output." By default it outputs to the terminal running Bash. The file descriptor for standard output is 1 (one).
As with stdin, stdout can be redirected to another process or file through use of a special character such as a pipe (|) or one or more greater-than signs (>,

>>*)* .

Sterr means "standard error." Whenever a program generates an error, whether through misuse, mistakes, or even intentionally, any and all error codes or messages are written to standard error. It is important to note that standard error and standard output do not always output to the same location unless explicitly told to do so. While a potential point of confusion at first, this solves the issue of having to sift through many lines of output for an obfuscated error message. It has the file descriptor 2 (two).

As with the other standard streams, standard error stream can be, and a lot of times is (especially when debugging), redirected. Typically it will be redirected to the same destination as standard output. This is accomplished by pointing the stderr file descriptor to stdout’s file descriptor. Again, this will be covered much more thoroughly in later chapters.

```
2>$1
```

Chapter 5: Variables Variables

A variable is a storage location called by a name referred to as an identifier. A variable holds a piece of information, called a value.
As in high school math, a variable can be as simple as “X”. You will probably want them to be a bit more descriptive in your code however, say fname, short for file name, or fdogn and mdogt for female dog name and male dog type respectively. The reason for shortening the variables is solely because retyping surveyed_female_dog_name or observed_male_canine_type is a hassle and waste of time.
There are a few types of variables. In Bash, nearly all are strings. They are also by far the most common type. Other variable types include arrays, and integers.

Strings

String variables are assigned by passing the value to the identifier with an equals sign (=).

```
$ identifier=value
```

Variables are recalled by naming it with a dollar sign ($). It is important to note that the dollar sign is not part of the variable name, it just tells Bash to “expand the variable that follows.”

What Bash does is expand (replace) identifier with the value, then echo the result. You can see this in action by setting verbose debugging mode in a script with set -xv like so:

```
#!/bin/bash
set -xv
# var_test_script
identifier=value
$ ./var_test_script
echo $identifier
+ echo value
value
```

Remember a few pages ago when we talked about word splitting? If you recall, whenever Bash encounters a string that is not quoted, it splits it into separate words, *delimited by whitespace* . The first of those words is taken as the command, and the remaining words are arguments passed to that command. Thus Bash thinks this,

```
$ identifier = value
```

means search $PATH for the command identifier, then pass it the arguments = and value for execution one at a time. Of course there is not a command called *identifier, so the above code fails with an error that says as much* .

```
identifier: command not found
```

There is however, a command called identity, and few thousand others on your machine. This means if you forget to remove the spaces between an identifier and its value, you may actually send arguments to a legitimate command without realizing it!

```
$ test = dummy_value
bash: test: =: unary operator expected
```

Whoops! Test (typically referred to as [) is actually a Bash built-in. Be sure to remove whitespace when assigning variables.

```
$ test=dummy_value
$ echo $test
dummy_value
```

Since this type of variable holds a string, it is perfectly acceptable to assign as long of a string to it as you would like—properly quoted of course!

```
$ sentence="I like beaches!"
$ filename="beach picture 42.jpg"
```

To verify that we stored the string correctly, let's echo them back.

```
$ echo $sentence
I like beaches!
$ echo $filename
beach picture 42.jpg
```

Looks good! Only problem is, that particular beach picture is terrible, so let's delete it.

```
$ rm $filename
rm: cannot remove 'beach': No such file or
directory
rm: cannot remove 'picture': No such file or
directory
rm: cannot remove '42.jpg': No such file or
directory
```

Hmmm. Not quite what the result we were looking for. It all goes back to how Bash replaces the variable with its value, then performs word splitting. Again, if we turn on verbose debugging we can see what Bash is actually trying to do.

Our script:

```
#!/bin/bash
set -xv
filename='beach picture 42.jpg'
echo $filename
```

And the output:

```
$ ./beachpic

filename='beach picture 42.jpg'
+ filename='beach picture 42.jpg'

rm $filename
+ rm beach picture 42.jpg
rm: cannot remove 'beach': No such file or
directory
rm: cannot remove 'picture': No such file or
directory
rm: cannot remove '42.jpg': No such file or
directory
```

Notice how Bash replaced $filename with beach picture 42*.jpg, performed word splitting, then tried to delete the results? The proper way to do this is with quotes* .

Ahh, much better.
It is worth repeating a sentence from a few pages ago: get into a habit of double quoting any string that does, will, or may contain any whitespace

(spaces, tabs, newlines, or special characters). You’ll thank yourself for it later.

Integers

Integer variables are one of the few times you will encounter something other than a string in Bash. On the flip side, they are virtually never used, because arithmetic expansion (more on this later) is much easier to use and read.

To set an integer variable, you must explicitly set it as such with **declare -i** .

```
$ declare -i varname
```

To erase an integer variable, you do so with **unset.**

```
$ unset varname
```

A single line example of declaring and setting an integer variable, echoing back the variable, and erasing it afterword looks like this:

```
$ declare -i x=10; echo "$x"; unset x
10
```

Remember that you must explicitly declare an integer variable. If you do not, you will get a string!

```
$ x=10; x+=5; echo "$x"; unset x
105
```

Let’s try that again, this time properly declaring it and then adding five to the integer (the “+=5” part).

```
$ declare -i x=10; x+=5; echo $x; unset x
15
```

Much better!

Let also works.

```
$ x=10; let x+=5; echo "$x"; unset x
15
```

It does not work the other way around, however.

```
$ let x=10; x+=5; echo "$x"; unset x
105
```

Unless of course you perform the addition inside the let statement.

```
$ declare -i x=10+5; echo "$x"; unset x
15
$ let x=10+5; echo "$x"; unset x
15
```

Once you get into multiple

integer variables things get even more cumbersome.

```
$ declare -i x=10; declare -i y=5; \
z="$x+$y"; echo "$z"; unset x; unset y; \
unset z
10+5
$ declare -i x=10; declare -i y=5; \
declare -i z="$x+$y"; echo "$z"; \
unset x; unset y; unset z
15
```

Are you thoroughly confused yet? I know I am. Do not worry too much about it; this is why integer variables are rarely used in Bash. We'll cover arithmetic expansion in a bit, which is vastly easier to write, read, and use.

Read only

Read only variables are exactly the same as normal variables except, as the name implies, they are read only. You set a read only variable by passing the -r flag to declare.

```
$ declare -r varname
```

Here we set our ideal weather conditions:

```
$ declare -r weather='sunny'; echo \
"$weather"
sunny
```

Once set, read only variables are permanent. That is, unset has no effect. Read only variables stick around until the shell process exits or is killed.

```
$ unset weather
bash: unset: weather: cannot unset: readonly
variable
$ echo "$weather"
sunny
```

To replace your current instance of Bash with a new one and get rid of read only variables, you can issue the following:

```
$ exec bash
```

Shell variables

In addition to setting "normal" variables, Bash provides a few special variables for your convenience. Not all the shell variables are listed here, but the most commonly used are.

Common shell variables Variable
$BASH
$BASH_VERSION $BASH_VERSINFO $EDITOR
$EUID
$HOME
$HOSTNAME
$HOSTTYPE
$IFS
$MACHTYPE
$OLDPWD
$OSTYPE
$PATH
$PIPESTATUS

$PPID
$PS1, …, $PS4
$PWD
$RANDOM
$SECONDS $UID

Description
The path to Bash's binary.
Bash's version.
Bash's major version.
The environment's default editor. The current user's effective user ID. The current user's home directory path. The machine's hostname.
The machine's hosttype.
The IFS setting. Blank if set as default. The machine type.
The previous directory you were in. The operating system type.
The command search path.
The exist status of the last completed pipe. The process ID of the current shell's

parent process.
The shell's (prompts 1 through 4) format. The current working directory.
A pseudo random integer between 0

2^15.

How many seconds the script has been executing.
The current user’s ID.
There are a couple important things to notice about shell variables. First, you do not have to manually set, declare, unset, or modify them yourself. They exist whether you decide to use them or not. Second, shell variables are always fully uppercase. This helps you avoid clobbering one of them with one of your own. A good rule of thumb is to always name your variables in lowercase. That way, whenever you see one in uppercase, you know it is special. Finally, shell variables and environment variables are not the same. Environment variables are covered in chapter 13.
For now, the most in-depth example involving shell variables will be a simple listing of them and their stored values. Once we get into tests and conditionals they will be incredibly useful. Just think of how much information you can glean about a logged in user by seeing if their UID is 0, or about a system by parsing the output of MACHTYPE.

```
$ cat sh_vars
#!/bin/bash
echo "Bash: $BASH"
echo "Bash Version: $BASH_VERSION"
echo "Bash Versinfo: $BASH_VERSINFO"
echo "Editor: $EDITOR"
echo "EUID: $EUID"
echo "Home: $HOME"
echo "Hostname: $HOSTNAME"
echo "Hosttype: $HOSTTYPE"
echo "IFS: $IFS"
echo "Machtype: $MACHTYPE"
echo "Oldpwd: $OLDPWD"
echo "OSType$OSTYPE"
echo "Path: $PATH"
echo "Pipestatus: $PIPESTATUS"
echo "PPID: $PPID"
echo "PS1: $PS1"
echo "PWD: $PWD"
echo "Random: $RANDOM $RANDOM $RANDOM"
echo "Seconds: $SECONDS"
echo "UID: $UID"
```

```
$ ./sh_vars
Bash: /bin/bash
Bash Version: 4.3.11(1)-release
Bash Versinfo: 4
Editor: /usr/bin/vim
EUID: 1000
Home: /home/aries
Hostname: aries_laptop
Hosttype: x86_64
IFS:

Machtype: x86_64-pc-linux-gnu
Oldpwd:
OSTypelinux-gnu
Path:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/us
r/bin:/sbin:/bin
Pipestatus: 0
PPID: 4413
PS1:
PWD: /home/aries/bash_scripts
Random: 7522 18854 14864
Seconds: 0
UID: 1000
```

Chapter 6: Arrays

Arrays are used more often than integer variables, but they require significant accounting to ensure you are setting, reading, replacing, and recalling the correct array member. My suggestion is to skip this part for now and return once you have read about special characters, special parameters, and loops.

The way an array works is that it holds key to value mappings. Each indexed key-value pair is called an element.

Indexed arrays

In this type of array (far and away the most common type), each value is mapped to a key starting from zero (0). This is called an indexed array. Graphically, it looks like this:

array
key value
0 *value1*
1 *value2*
. . . . . .
N *valueN*

An array called animals is visualized below.
animals
0 Dog
1 Bear
2 Llama
3 Parrot
4 Otter

Array creation is similar to variable creation and assignation. Like everything in Bash, there are usually multiple ways of performing the same task. Arrays are no exception. They can be declared in a couple different manners.

Declaration Usage
array=()
array[0]=value

declare -a array Description
Create array
Declares an empty array called *array* . **Create & set array key**
Declares array, *array* , and sets its first value

(index 0) to *value* .
Create array
Declares an empty array, *array* .

Once the array has been created, with or without values, it can be modified using the following syntax.
Storing values
Usage

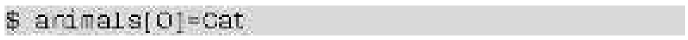
array[i]=value
array=(value1, value2, . . ., valueN)
array+=(value1, value2, . . ., valueN)

array=([i]=value1,
[j]=value2, . . ., [k]=valueN) Description
Set array value
Sets the *i* th key of *array* to *value* . **Set entire array**
Sets the entire array, *array* , to any

number of values, *value1, value2, . . .valueN* , which are indexed *in order, from zero* . Any previously set array values are lost unless the array is appended with +=

Append array

Appends the existing array, *array* , with the values, *value1, value2, . . .valueN* .
Compound set array values
Set the indexed keys, *i, j,* and *k* , to *value1, value2, . . .valueN* respectively. Any previously set array values are lost. Here we create a simple indexed



```
$ animals[0]=Cat
```

array. The array just created looks
animals
0 like this: Cat

Let’s add additional values to our existing array.

```
$ animals=("Dog" "Bear" "Llama" "Parrot"
"Otter")
```

Due to the way array creation and assignation works, our attempt to modify the array actually wiped (*unset)* our previous value, and reset the entire array as:**animals**

0 Dog
1 Bear
2 Llama
3 Parrot
4 Otter

If we want to add the value Cat back into the array, we can append it to the end.

```
$ animals+=('Cat')
```

The animals array now looks like this:
animals
0 Dog
1 Bear
2 Llama
3 Parrot
4 Otter
5 Cat

In the event that a singular cluster component should be altered, or one more added, the accompanying language structure allows us to do so.

```
$ animals[1]=Tiger; animals[7]=Chicken
```

There are two important things to note here. The first is that we have accidently created a **sparse** array. That is, the *animals* array has eight *keys* (remember array elements are indexed from zero), yet only *seven values* .

animals
0 Dog
1 Tiger
2 Llama
3 Parrot
4 Otter
5 Cat
6
7 Chicken
If a circle is utilized to count the exhibit, key six will be unfilled! This is normally not what you need to happen.
Secondly, assuming you endeavor to set the exhibit esteems utilizing this syntax,

```
$ animals=([1]="Tiger" [7]="Chicken")
```

you will unset the current cluster and make another one that resembles this:
animals
0
1 Tiger
2
3
4
5
6
7 Chicken
That isn't what we needed by any stretch of the imagination! You could attempt to affix the cluster all things considered and determine keys and their individual qualities, however attaching annexes everything.

```
# "reset" array if you need to
$ animals=("Dog" "Bear" "Llama" "Parrot"
"Otter" "Cat")
$ animals+=([1]="Tiger" [7]="Chicken")
```

The exhibit is now:
animals

0 Dog
1 BearTiger
2 Llama
3 Parrot
4 Otter
5 Cat
6
7 Chicken

As you can tell, it is basic to maintain cautious bookkeeping of each activity that is performed on a cluster to guarantee that it results in just, and precisely, the activities you implied it too.

Note that adding client inputted information is a straightforward matter of redirection. In the code underneath, a formerly entered client inputted sentence is diverted into a cluster by means of a here string. The sentence will be parted dependent on the default IFS settings except if in any case determined. *Here archives* , record redirection, and others work in a comparative manner.

```
$ sentence="I'm a user-inputted sentence!"
$ read -a words <<< $sentence
```

Let's "reset" the cluster to the extra exhibit we accidently made previously, and figure out how to peruse values from it.

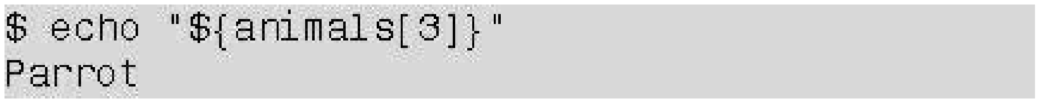
```
$ animals=("Dog" "Tiger" "Llama" "Parrot"
"Otter" "Cat" "" "Chicken")
```

Reading factors from a cluster is like performing boundary extension on that array.
Retrieving values Usage

${array[i]}
${array[@]} "${array[@]}"
${array[*]} "${array[*]}"
${array[@]:offset:num} "${array[@]:offset: num }"

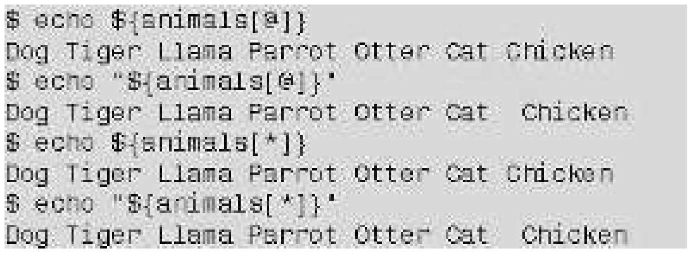
${array[*]:offset:num } “${array[*]:offset:num }” Description
Expand value
Expands to the *i* th value of indexed

array, *array* . Negative numbers count backwards from the highest indexed key. **Mass expand values**
Expands to all values in the array. If double quoted, it expands to all values in the array individually quoted.
Mass expand values
Expands to all values in the array. If double quoted, it expands to all values in the array quoted as a whole.
Subarray expansion
Expands to *num* of array values, beginning at *offset* . The same quoting rules as above apply.
Subarray expansion
Expands to *num* of array values, beginning at *offset* . The same quoting rules as above apply.
Reading a solitary exhibit component is like perusing a solitary person of a string, besides with an exhibit, you should determine the vital worth to be read.



```
$ echo "${animals[3]}"
Parrot
```

Parrot is shown on the grounds that it relates to key worth 3, not the third component of cluster animals!
Displaying every one of the components of creatures cluster should be possible in a couple of ways, each inconspicuously unique. This might be befuddling at first on the grounds that the repeated result of the strategies show up almost identical.



```
$ echo ${animals[@]}
Dog Tiger Llama Parrot Otter Cat Chicken
$ echo "${animals[@]}"
Dog Tiger Llama Parrot Otter Cat  Chicken
$ echo ${animals[*]}
Dog Tiger Llama Parrot Otter Cat Chicken
$ echo "${animals[*]}"
Dog Tiger Llama Parrot Otter Cat  Chicken
```

The main evident distinction above is the additional room between the qualities Cat and Chicken (the missing sixth component in our meager exhibit) when the cluster is quoted.
If we utilize a circle all things considered, perceive how the result changes:

```
$ for elem in ${animals[@]}; do echo
“$elem”; done
Dog
Tiger
Llama
Parrot
Otter
Cat
Chicken
# for elem in ${animals[*]}; . . .
# produces exactly the same output as above
```

Quoting the arrays shows us that there is indeed a difference. The *at* (@) method results in each individual value, word split and quoted itself. The *glob* (*) results in a single long string with any extra whitespace (i.e. the missingelements in our sparse array) truncated.

```
$ for elem in '${animals[@]}'; do echo
“$elem”; done
Dog
Tiger
Llama
Parrot
Otter
Cat

Chicken
$ for elem in '${animals[*]}'; do echo
“$elem”; done
Dog Tiger Llama Parrot Otter Cat Chicken
```

Expanding quite a few components creates a similarly result.

```
$ for elem in '${animals[@]:5:3}'; do echo
“$elem”; done
Cat

Chicken
$ for elem in '${animals[*]:5:3}'; do echo
'$elem'; done
Cat  Chicken
```

Like boundary development, exhibits additionally have metadata that compares to the quantity of qualities, records, and length. These are particularly helpful in loops.

Metadata
Usage
${#array[i]}

${#array[@]} ${#array[*]} ${!array[@]} ${!array[*]}

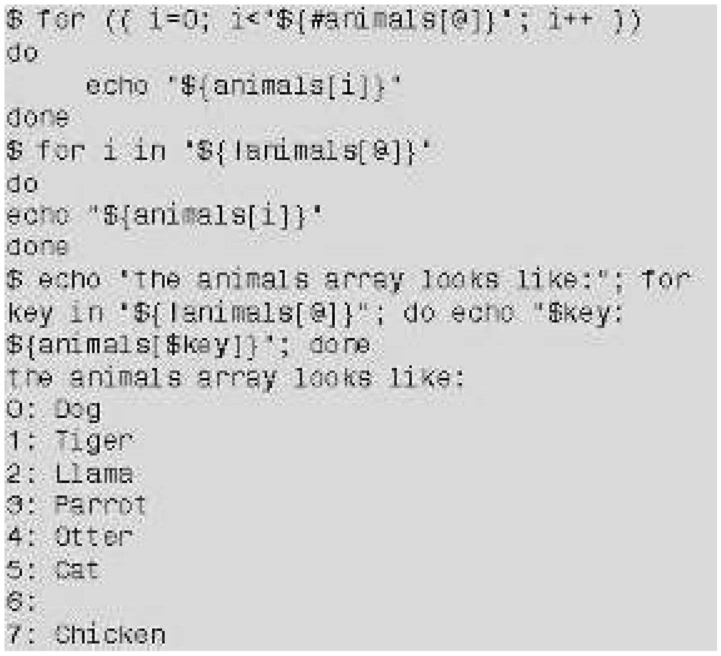
```
$ echo "${animals[3]} has ${#animals[3]}
letters."
Parrot has 6 letters.
```

Notice that Bash makes no extraordinary differentiation for scanty exhibits. That is, despite the fact that key 6 (comparing to component 7) in the creatures

Description
Value string length
Expands to the string length of the *i* th array value **Array values**
Expands to the number of values in the array. **Array indexes**
Expands to the indexes in the array.
exhibit is unfilled, it is as yet treated as a component, very much like all the others keys that contain real values.

```
$ echo ${#animals[@]}
8
$ echo ${!animals[@]}
0 1 2 3 4 5 6 7
```

Loops (for, while, and until) can be developed utilizing standard syntax:


```
$ for (( i=0; i<"${#animals[@]}"; i++ ))
do
    echo "${animals[i]}"
done
$ for i in "${!animals[@]}"
do
echo "${animals[i]}"
done
$ echo "the animals array looks like:"; for
key in "${!animals[@]}"; do echo "$key:
${animals[$key]}"; done
the animals array looks like:
0: Dog
1: Tiger
2: Llama
3: Parrot
4: Otter
5: Cat
6:
7: Chicken
```

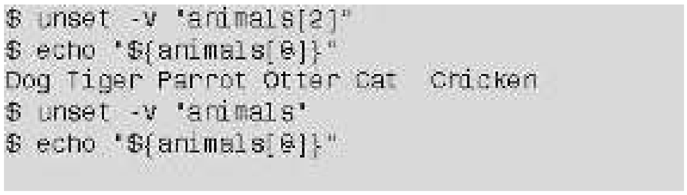
Arrays are eradicated by unsetting them.
Deletion
Usage Description
unset -v array
unset -v Erase an array
array[@] Completely erases the array, *array* .
unset -v array[*]

unset -v array[i]
Erase an array value
Erases the *i* th array value from *array* .

Associative arrays

Unlike *indexed arrays* , which use keys that begin at zero and increment upward by one, **associative arrays** use string labels to associate with eachvalue. **array**
string label
string1
otherstring2

. . .
anotherstring3



```
$ unset -v 'animals[2]'
$ echo "${animals[@]}"
Dog Tiger Parrot Otter Cat  Chicken
$ unset -v 'animals'
$ echo "${animals[@]}"
```

value
value1
value2
. . .
valueN
Because acquainted clusters are not mathematically listed, *they are dependably unordered* . This is vital! It implies that any endeavor to recover different qualities from the exhibit will return them in irregular request (except if each string name is unequivocally indicated and requested in and of itself).
The sentence structure to make, store, recover, and perform different activities on them is like recorded arrays.
Declaration
Usage

declare -A array

array[str]=value Description
Create array
Declares an empty array, *array* .
Create & set array key
Declares array, *array* , and sets its first *string*

label to *value* .
Storing values Usage

array[str]=value Description
Set array value
Sets the element indexed by *str* of *array* to

value .

array=([str1]=value1, [str2]=value2, . . ., [str3]=valueN)
Compound set array values
Set the elements indexed by strings, *str1,*

str2, and *str3* , to *value1, value2, . . .valueN* respectively. Any previously set array values are lost.

Retrieving values Usage
${array[str]}
${array[@]} "${array[@]}"

${array[*]} "${array[*]}" Description
Expand value
Expands to the element indexed by string

str of array, *array* .
Mass expand values
Expands to all values in the array. If double

quoted, it expands to all values in the array individually quoted. Output is in random order.

Mass expand values
Expands to all values in the array. If double quoted, it expands to all values in the array. quoted as a whole. Output is in random order. **Metadata Usage**

${#array[str]}
${#array[@]}
${#array[*]}
${!array[@]}
${!array[*]}

Deletion
Usage

unset -v array
unset -v array[@] unset -v array[*]

unset -v array[str]

unset -v array
unset -v array[@] unset -v array[*] Description
Value string length
Expands to the string length of the element

indexed by *str*
Array values
Expands to the number of values in the

array.
Array indexes
Expands to the string labels in the array.

Description
Erase an array
Completely erases the array, *array* . **Erase an array value**
Erases the element indexed by *str* from *array* .
Erase an array
Completely erases the array, *array* .

We start by pronouncing a cluster, stingarr, and setting a couple of qualities. Repeating the exhibit back shows us that the qualities are for sure put away. Notice how they returned in irregular request? They are not arranged alphabetically!

```
$ declare -A stringarr
$ stringarr=([weather]="sunny" [temp]="86"
[skies]="clear" [activity]="hiking")
$ echo "${stringarr[*]}"
86 sunny hiking clear
```

Next, we add a string name and esteem and check it has without a doubt been added.

```
$ stringarr[companion]="black lab"
$ for elem in "${stringarr[@]}"; do echo
"$elem"; done
86
sunny
black lab
hiking
clear
```

Unsetting and adding values shows how cooperative cluster metadata recovery works.

```
$ unset -v stringarr[temp]
$ echo "stringarr has ${#stringarr[@]}
elements."
stringarr has 4 elements.
$ stringarr[season]="autumn"
$ echo "now it has ${#stringarr[@]}
elements."
now it has 5 elements.
```

Just like filed exhibits, the string marks for affiliated clusters is effectively recovered and shown, first as an independent rundown, and second as a combined rundown with its put away values.

```
$ echo "${#stringarr[@]} preferences are
set: ${!stringarr[@]}"
5 preferences are set: weather companion
activity skies season
```

Chapter 7: Special Characters

A few of Bash's special characters— characters that have other than a literal meaning—have already been introduced, such as the double quotes and dollar sign, but many more exist. While not thorough, the larger part are recorded below.

Basic

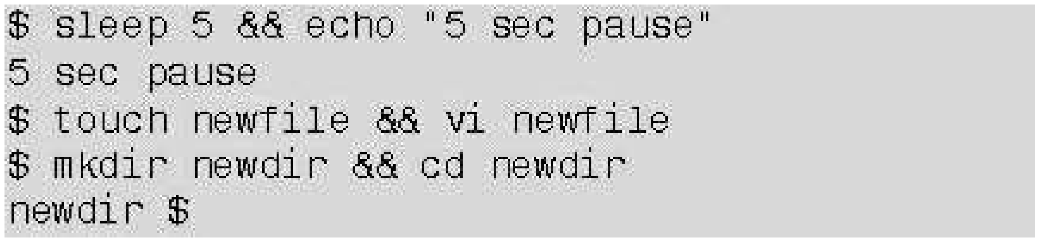
Character Description
#

Comment
Lines beginning with a hash will not be executed. **Whitespace**

“ “ Bash uses whitespace (e.g. spaces, tabs, and newlines) to perform word splitting.
Run in background
& Cause the preceding command to run in the background.
Command separator
; Allows for the placement of another command on the same line.

Logic

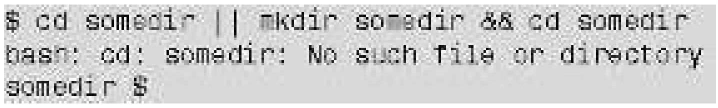
Character Description
And
&& Logical *and* operator. Returns a success if both of the conditions before *and* after the operator are true. **Or**
|| Logical *or* operator. Returns a success if either of the conditions before *or* after the operator are true. The legitimate AND administrator is very much like that learned in other programming dialects or gadgets/PC/rationale courses. It plays out a subsequent errand if and provided that the first effectively finishes. Slam knows whether an assignment effectively finished by inspecting the order’s leave code. A whole segment is given to leave codes later in the aide. For the time being, all you really want to know is that a leave code is 0 for valid, and whatever else (regularly 1) for false.



```
$ sleep 5 && echo "5 sec pause"
5 sec pause
$ touch newfile && vi newfile
$ mkdir newdir && cd newdir
newdir $
```

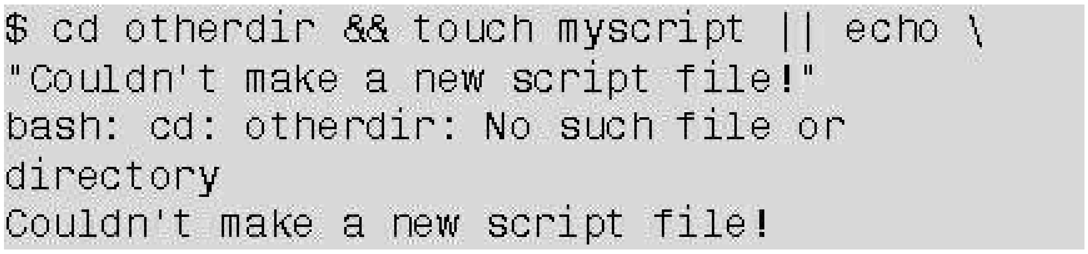
The logical OR operator works precisely the opposite of the AND operator. It performs a second task *if and only if* the first *does not successfully*

complete (i.e. its error code does not equal 0).
In the two cases, various ANDs and ORs can be hung together.



```
$ cd somedir || mkdir somedir && cd somedir
bash: cd: somedir: No such file or directory
somedir $
```

This isn't great practice notwithstanding, as the leave code read is consistently the last one. This can cause sudden behavior.

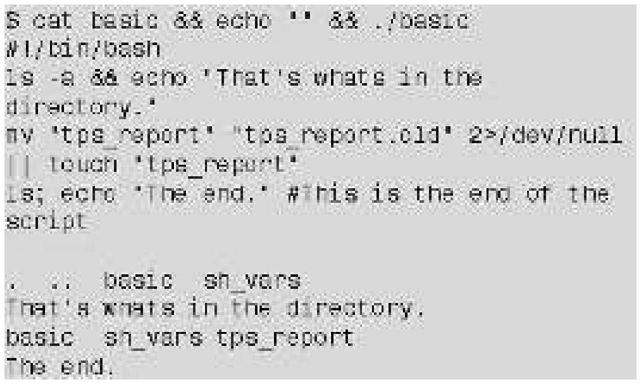


```
$ cd otherdir && touch myscript || echo \
"Couldn't make a new script file!"
bash: cd: otherdir: No such file or
directory
Couldn't make a new script file!
```

But stand by! Slam never at any point attempted to make the document, *myscript* . What it did was attempt to change to otherdir, which fizzled on the grounds that the index didn't exist. The leave code was set to 1 (a disappointment), which made it skirt the touch order. At the point when it saw the OR administrator, it executed the order there on the grounds that the leave code was as yet set to the final remaining one (the compact disc disappointment). It then told us it could not make the new script file even though it never even tried.

If you are not extremely cautious with your rationale administrators, the present circumstance will play out when you wouldn't dare hoping anymore, you will wind up with a regal migraine of sudden results.

The main time you should sting more than one legitimate administrator together is assuming they are all ANDs or all ORs. In any case, I exceptionally recommend that assuming you really want more than one intelligent administrator, you use control gatherings, as shown in a couple sections.



```
$ cat basic && echo '' && ./basic
#!/bin/bash
ls -a && echo 'That's whats in the
directory.'
mv 'tps_report' 'tps_report.old' 2>/dev/null
|| touch 'tps_report'
ls; echo 'The end.' #This is the end of the
script

.  ..  basic  sh_vars
That's whats in the directory.
basic  sh_vars tps_report
The end.
```

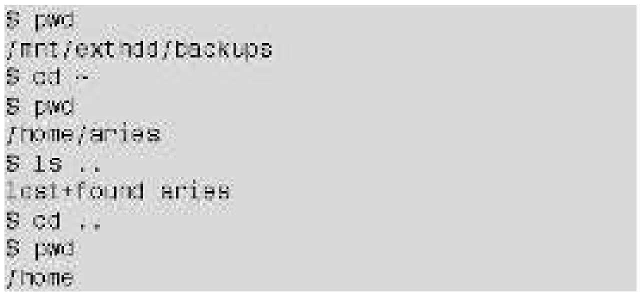
Directory traversal

Character Description
Home directory
~ Represents the home directory, and the current user's home directory when followed by a forward slash. **Current directory**
A dot in front of a filename makes it "hidden." Use *ls -a* to view.
. A dot directory represents the current working directory.
A dot separated from a filename by a space *sources* (loads) the file.

..
Parent directory
A double dot directory represents the parent directory. **Filename separator**

/

A forward slash separates the components of a filename.
Performs division when used arithmetically. These are essential, all inclusive orders in any GNU/Linux system.



```
$ pwd
/mnt/exthdd/backups
$ cd ~
$ pwd
/home/aries
$ ls ..
lost+found aries
$ cd ..
$ pwd
/home
```

Quoting

Character Description
Escape
\

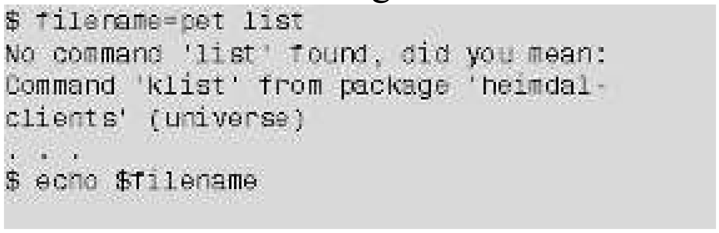
Escapes the following special character and causes it to be treated literally.
Allows for line splitting of a long command.
Full quoting

‘ ‘

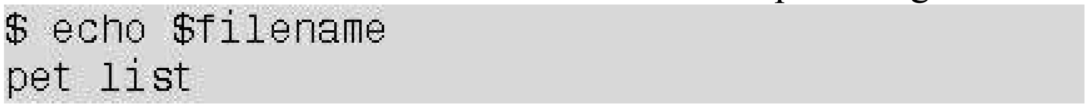
Special characters within single quotes lose their meaning and become literal.
Word splitting is not performed on the contents.
Partial quoting
Special characters within double quotes lose their

“ “

meaning, with the notable exception of parameter expansion, arithmetic expansion, and command substitution.

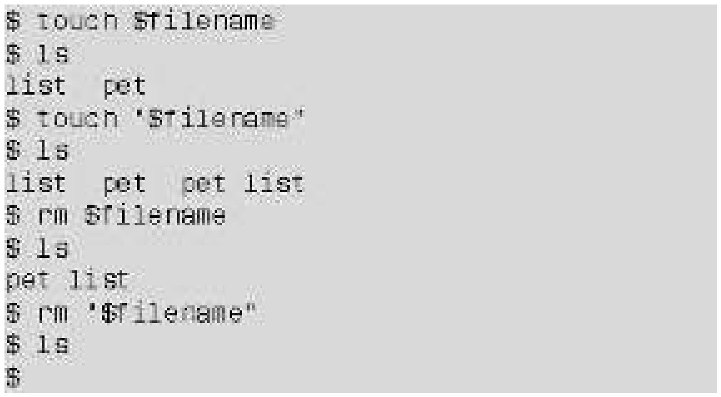
Word splitting is not performed on the contents. Though citing has been covered, it merits rehashing once more: if all else fails, in every case twofold statement strings!


```
$ filename=pet list
No command 'list' found, did you mean:
Command 'klist' from package 'heimdal-
clients' (universe)
. . .
$ echo $filename
```

Oops. Citing tackles this.


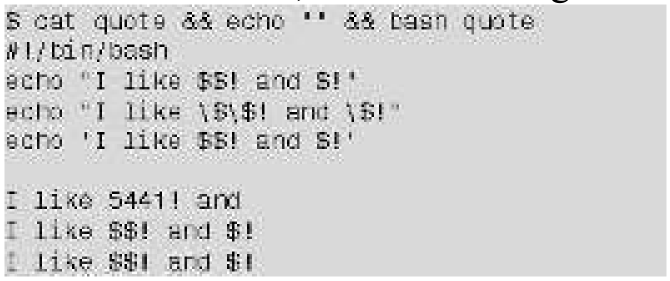
```
$ echo $filename
pet list
```

But recall that strings should be cited wherever word parting occurs.



```
$ touch $filename
$ ls
list   pet
$ touch '$filename"
$ ls
list   pet   pet list
$ rm $filename
$ ls
pet list
$ rm '$filename"
$ ls
$
```

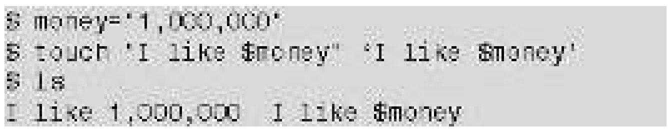
The contrast among halfway and full citing is that incomplete citing handles unique characters like variable extension, while full citing does not.



```
$ cat quote && echo '' && bash quote
#!/bin/bash
echo "I like $$! and $!'
echo "I like \$\$! and \$!"
echo 'I like $$! and $!'

I like 5441! and
I like $$! and $!
I like $$! and $!
```

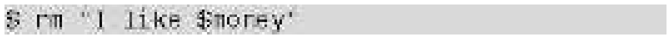
Notice how when to some degree cited (twofold quotes), to show dollar signs, they must be gotten away. With full citing (single quotes), that was superfluous. In the event that you were pondering, the result from the twofold dollar signs (&&) grows to the cycle id of the current shell.

Incidentally, this is an excellent motivation behind why filenames ought to never contain unique characters—they will generally break things except if extraordinary taking care of methods are coded in for them.



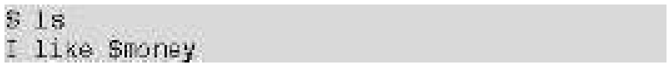
```
$ money='1,000,000'
$ touch 'I like $money" 'I like $money'
$ ls
I like 1,000,000  I like $money
```

Which record will be eliminated when you run this?



```
$ rm 'I like $money'
```

Did you surmise right?



```
$ ls
I like $money
```

Even assuming you just delayed for a negligible portion of a second prior noting effectively, envision how troublesome you could make life for yourself.
The overall principle of thumb in regards to strings in Bash is this: on the

off chance that you don’t need the string’s substance worked on in any capacity, utilize full citing. In any remaining cases, utilize incomplete quoting.

Redirection

Character Description
>
Output redirection
Redirects standard output, typically to a file.

>>
Output redirection
Redirects and *appends* standard output, typically to a file. **Input redirection**

< Redirects standard input, typically for reading in a file’s contents.
Here document
<< Reads in strings of data until a specified delimiter is encountered.

<<<
Here string
Reads in a single string of data immediately following. **Pipe**

| Redirects the standard output of a command to the standard input of another command.

Redirection is a long and muddled theme that has a committed section close to the furthest limit of the aide. An exceptionally fast (and unexplained) illustration of each is displayed underneath. See section 13 for definite explanations.

You can keep in touch with a record with a more noteworthy than sign, >.

```
$ route > routeinfo
$ cat routeinfo
Kernel IP routing table
Destination    Gateway       Genmask . . .
default        10.0.0.1      0.0.0.0 . . .
10.0.0.0       *             255.255.255 . . .
```

Writing to a document overwrites any current contents.

A twofold more prominent than sign, >>, attaches to a document without overwriting its contents.

```
$ ping -c 1 10.0.0.1 >> routeinfo
$ cat routeinfo
Hello!
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of
data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64
time=7.29 ms

--- 10.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet
loss, time 0ms
rtt min/avg/max/mdev =
7.297/7.297/7.297/0.000 ms
```

The not exactly sign, <, permits the substance of a document to be perused as standard input.

```
$ cat <coworkers
John, Dan, Mark Hughes
Nicole Fisher      Jack James Johnson
Eddie: Mr. Ice
$ grep -n -i 'ni' <coworkers
2:Nicole Fisher      Jack James Johnson
```

Here reports permit you to install squares of information into your script. Here strings, then again, accept just a solitary string as input.

```
$ sentence='this sentence becomes an array'
$ read -r -a words <<< "$sentence"
$ echo "${words[@]}"
this sentence becomes an array
```

A line permits you to pipe (send) the standard result of one order to the standard contribution of another.

```
$ cat /etc/ssh/sshd_config | grep -n -i port
4:# What ports, IPs and protocols we listen
for
5:Port 22
```

Groups

Character Description
Inline group

Commands within the curly braces are treated as a

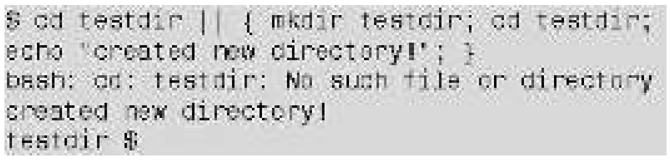
{ }

single command. Essentially a nameless function without the ability to assign or use local variables. The final command in an inline group must be terminated with a semicolon.

Command group
() Commands within are executed as a subshell. Variables inside the subshell are not visible to the rest of the script.
Arithmetic expression
Within the expression, mathematical operators (+ , - , * ,

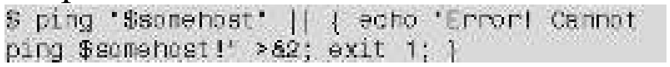
(()) */, >, <, etc.*) take on their mathematical meanings (addition, subtraction, multiplication, division). Used for assigning variable values and in tests.

Braces, or inline gatherings, permit all orders inside to be treated as a solitary order. Here we endeavor to change into a catalog. Assuming that the catalog exists, everything works and the inline bunch following the OR rationale is skipped altogether. If the directory does not exist, then it is created, *cd'd* into, and amessage is printed as such.



```
$ cd testdir || { mkdir testdir; cd testdir;
echo 'created new directory!'; }
bash: cd: testdir: No such file or directory
created new directory!
testdir $
```

Along these lines, inline bunches are particularly valuable for straightforward blunder taking care of in scripts.



```
$ ping '$somehost' || { echo 'Error! Cannot
ping $somehost!' >&2; exit 1; }
```

Command bunches permit a square of orders to be executed in a subscript. When utilizing order gatherings, it is critical to comprehend that each cycle happens in its own subshell. This implies that each time an order is called, Bash forks the parent interaction to make a kid cycle, and afterward executes the
order there. When executed, the kid cycle is killed. This presents extra

handling overhead that while not critical in more modest contents, can rapidly outgrow control in bigger. Furthermore, assuming the parent is killed before the youngster, the kid turns into a zombie cycle and is acquired by init (process ID 1). Traps are helpful for controlling this undesirable conduct and tidying up kid processes.

```
$ ( ping -c 1 10.0.0.1 ) | (grep -i packet)
1 packets transmitted, 1 received, 0% packet
loss, time 0ms
```

Along similar lines, to this end Bash constructed ins execute speedier than their outer partners. Constructed ins don't need the forking off of a sub-process, while their outer partners do.

Finally, factors in a subshell are not noticeable outside of that subshell — even to their parent interaction. This implies that any factors in a subshell are basically nearby variables.

```
$ guy='John'; girl='Jane'
$ echo "$guy and $girl"
John and Jane
$ (last='Doe'; echo "$guy and $girl $last";)
John and Jane Doe
$ echo "$guy and $girl $last"
John and Jane
$ (girl='Betty'; guy='Sam'; echo "$guy and
$girl $last")
Sam and Betty
$ echo "$guy and $girl $last"
John and Jane
```

Arithmetic articulations are helpful whenever a numerical activity should be performed, as they don't need the unequivocally announcement of a variable as an integer.

```
$ $ x=3;y=4;z=5;
$ ((result=x+y*z-10))
$ echo $result
13
```

Arithmetic articulations utilized without development are a type of test. Inside the twofold bracket, numerical administrators work as they would on a number cruncher. The articulation returns of worth of valid (0), or bogus (1) in light of the numerical articulation held inside. Section 11 covers tests.

```
$ (( $result == 13 )) && echo 'You are most
unlucky!'
You are most unlucky!
```

It is important to note that Bash only performs integer math. Bash does not know what *longs* , *doubles* , *tuples* , or anything like that are. Additionally, Bash’s built-in math functions are limited. If you need square roots, decimal division, etc., you must use an external utility like bc.

```
$ echo 'scale=2; sqrt(31^2+49^2)' | bc
57.98
```

Command gatherings and number-crunching articulations can both be extended. They then, at that point, become order replacement and math development individually. Both are canvassed exhaustively in part 10.

Chapter 8: Globs

Globs, or special cases in Bash, are a lifeline when matching characters and strings. Rather than physically entering strings that contain spaces, blended case letters, extraordinary characters, and others, globs permit you to enter one person and be done.

Globs
Character ?
*

[…] Description
Wildcard (character)
Serves as a match for a single character.
Wildcard (string)
Serves as a match-for any number of characters. **Wildcard (list)**
Serves as a match for a list of characters or ranges.

?

The question mark is the most straightforward to utilize. It subs for a solitary person, and signifies, “match any single character.”
The least difficult utilization of ? is going along with it with a few“standard” orders like ls, cd, rm, cp, and mv.
More useful is to use it to find all items of a known designation. Below we issue the *find* command to find all instances of “*sd?* ” on the system (*sda* stands for SATA drive A).

```
$ find / -name sd?
/run/sdp
/sys/devices/pci0000:00/0000:00:1f.2/ata1/ho
st0/target0:0:0/0:0:0:0/block/sda
/sys/block/sda
/sys/class/block/sda
/dev/sda
/usr/share/mlt/sdl
/usr/lib/virtualbox/sdk
```

A comparable order tracks down every one of the parcels on sda.

```
$ find / -name sda?
/proc/fs/ext4/sda1
/sys/fs/ext4/sda1
/sys/devices/pci0000:00/0000:00:1f.2/ata1/ho
st0/target0:0:0/0:0:0:0/block/sda/sda1
/sys/devices/pci0000:00/0000:00:1f.2/ata1/ho
st0/target0:0:0/0:0:0:0/block/sda/sda2
/sys/devices/virtual/block/dm-0/slaves/sda2
/sys/devices/virtual/block/dm-1/slaves/sda2
/sys/devices/virtual/block/dm-2/slaves/sda2
/sys/class/block/sda1
/sys/class/block/sda2
/dev/sda2
/dev/sda1
```

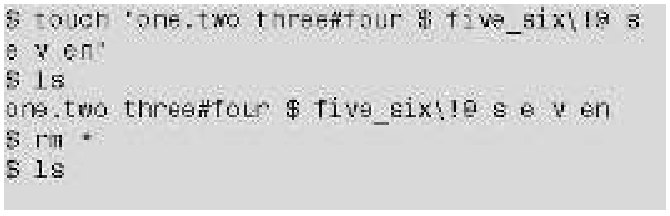
*

The reference bullet is by a wide margin the most helpful glob. That is to say, “match quite a few characters” in a string, including a totally unfilled string, called the invalid string.

The manner in which it works is that Bash executes the order, say,

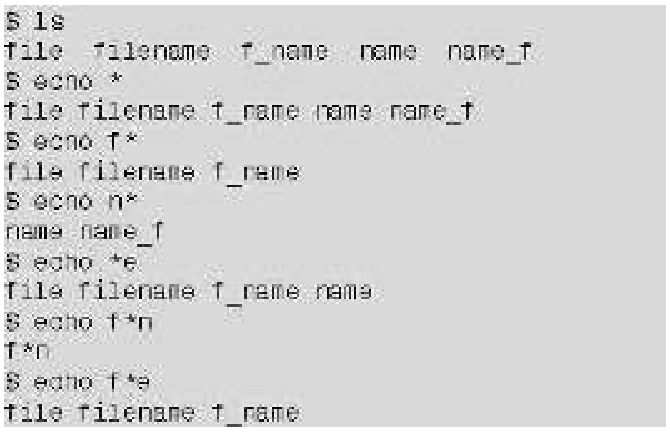
```
$ echo *
```

by perusing in reverberation as the order, performs word parting (according to IFS), and passes the glob, * , to the reverberation order. Then, Bash extends the glob and matches it against all records in the current registry. After all the matches are found, Bash sorts them alphanumerically, then replaces the glob (the asterisk) with the results one at a time. The request here is vital—did you see that Bash performs word parting before it extends the glob? This guarantees that matching glob examples won't be separated by your IFS setting and will forever be taken care of correctly!



```
$ touch 'one.two three#four $ five_six\!@ s
e v en'
$ ls
one.two three#four $ five_six\!@ s e v en
$ rm *
$ ls
```

Another significant point is that globs are certainly moored at the two closures. A basic model ought to show how this works.



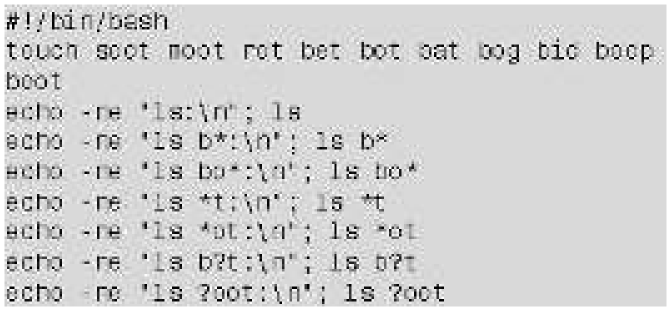
```
$ ls
file  filename  f_name  name  name_f
$ echo *
file filename f_name name name_f
$ echo f*
file filename f_name
$ echo n*
name name_f
$ echo *e
file filename f_name name
$ echo f*n
f*n
$ echo f*e
file filename f_name
```

See how n* didn 't match filename? The explanation is on the grounds that the person, *n* , is secured at the front. This is the very same motivation behind why *e neglected to match name_f regardless of the document containing the letter e—in light of the fact that the letter is moored at the end.

It is passable to utilize globbing in a filename, as in the last two models, yet once more, securing applies. Since no records started or ended with the characters, *f and n* , individually, *reverberation just reverberated back your assertion* . Changing this conduct is shrouded in the segment called invalid globs.

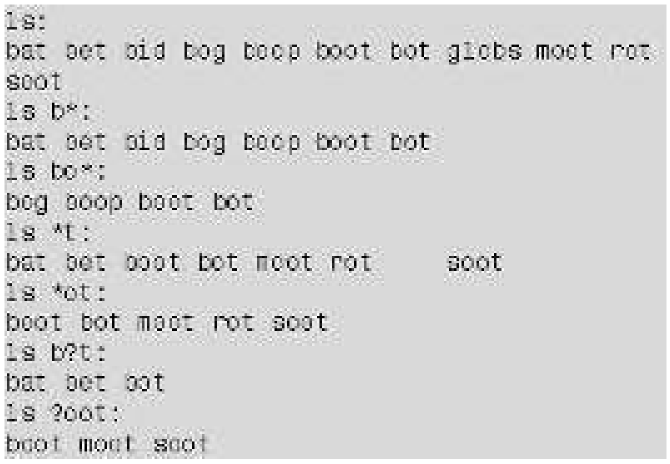
Note that neither trump card can match the forward cut (/) character. The purpose for this is on the grounds that the forward cut is the filename separator. On the off chance that globs could match this person, you would have zero influence over index recursion.

Here are a couple of more instances of globbing in real life. The switches passed to the reverberation order forestalls the default added following newline, and afterward permits us to physically indicate where a newline ought to pass by entering, *\n* .



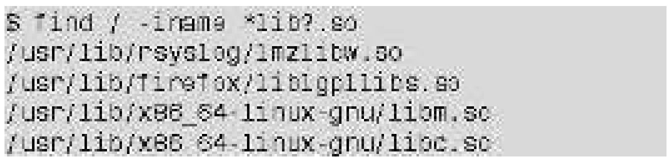
```
#!/bin/bash
touch soot moot rot bet bot bat bog bic boop
boot
echo -re 'ls:\n'; ls
echo -ne 'ls b*:\n'; ls b*
echo -re 'ls bo*:\n'; ls bo*
echo -ne 'ls *t:\n'; ls *t
echo -re 'ls *ot:\n'; ls *ot
echo -re 'ls b?t:\n'; ls b?t
echo -re 'ls ?oot:\n'; ls ?oot
```

Its result looks like this:



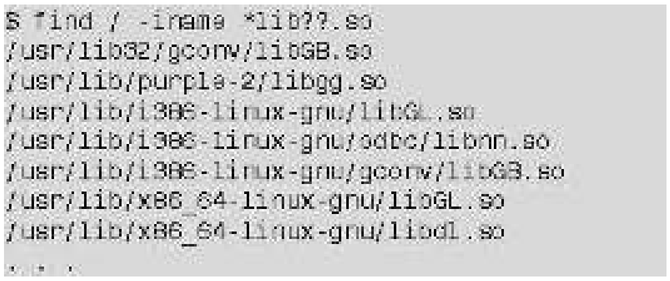
```
ls:
bat bet bid bog boop boot bot globs moot rot
soot
ls b*:
bat bet bid bog boop boot bot
ls bo*:
bog boop boot bot
ls *t:
bat bet boot bot moot rot     soot
ls *ot:
boot bot moot rot soot
ls b?t:
bat bet bot
ls ?oot:
boot moot soot
```

The question mark and reference bullet globs can be utilized together, in any request, and in any amount. In the first place, we utilize the track down order to track down any records on the framework that end in“lib” trailed by any single person in addition to“*.so”* .



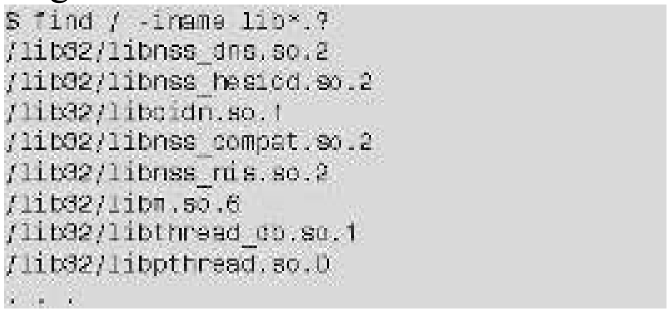
```
$ find / -iname *lib?.so
/usr/lib/rsyslog/lmzlibw.so
/usr/lib/firefox/liblgpllibs.so
/usr/lib/x86_64-linux-gnu/libm.so
/usr/lib/x86_64-linux-gnu/libc.so
```

Next, we permit“lib” to be prevailed by any two characters in addition to “*.so”* .



```
$ find / -iname *lib??.so
/usr/lib32/gconv/libGB.so
/usr/lib/purple-2/libgg.so
/usr/lib/i386-linux-gnu/libGL.so
/usr/lib/i386-linux-gnu/odbc/libnn.so
/usr/lib/i386-linux-gnu/gconv/libGB.so
/usr/lib/x86_64-linux-gnu/libGL.so
/usr/lib/x86_64-linux-gnu/libdl.so
. . .
```

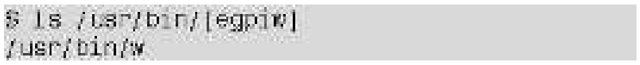
Finally, we track down just documents that start with "lib" and end with a period in addition to any single character.



```
$ find / -iname lib*.?
/lib32/libnss_dns.so.2
/lib32/libnss_hesiod.so.2
/lib32/libcidn.so.1
/lib32/libnss_compat.so.2
/lib32/libnss_nis.so.2
/lib32/libm.so.6
/lib32/libthread_db.so.1
/lib32/libpthread.so.0
. . .
```

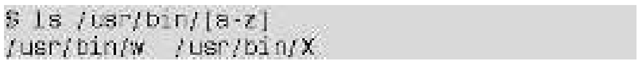
[…]

Square supports are to some degree a center ground between ? furthermore *. They take into account the matching of quite a few individual characters and additionally scopes of characters. Determining the characters you wish to grow for is finished by basic specifying them between the braces.



```
$ ls /usr/bin/[egpiw]
/usr/bin/w
```

While a dash, - , demonstrates an extension range.



```
$ ls /usr/bin/[a-z]
/usr/bin/w  /usr/bin/X
```

They can remain all alone, but except if you know the specific length, as well as blend of characters, it is ordinarily useful to add with the question mark or reference bullet glob. The model beneath tells us the number of pairs starting with the letters an or b are in the index/usr/bin.

```
$ ls /usr/bin/[ab]* | wc -l
127
```

You could likewise list every one of the parallels that are just two letters:

```
$ ls /usr/bin/[a-z]?
/usr/bin/ar  /usr/bin/ci  /usr/bin/ex
/usr/bin/lp  /usr/bin/od  /usr/bin/ul
/usr/bin/as  /usr/bin/co  /usr/bin/gs
/usr/bin/lz  /usr/bin/pg  /usr/bin/uz
/usr/bin/at  /usr/bin/dc  /usr/bin/hd
/usr/bin/mf  /usr/bin/pr  /usr/bin/vi
/usr/bin/bc  /usr/bin/dh  /usr/bin/id
/usr/bin/nl  /usr/bin/sg  /usr/bin/wc
/usr/bin/cc  /usr/bin/du  /usr/bin/ld
/usr/bin/nm  /usr/bin/tr  /usr/bin/xz
```

Adding more than one square support extension is OK. This order records all the two letter parallels in/usr/receptacle that beginning with the letters a through m, and end with the letters m through z.

```
$ ls /usr/bin/[a-m][m-z]
/usr/bin/ar  /usr/bin/at  /usr/bin/du
/usr/bin/gs  /usr/bin/lz  /usr/bin/as
/usr/bin/co  /usr/bin/ex  /usr/bin/lp
```

Adding a *caret,* ^*,* negates the expansion. Along these lines running,

```
$ ls /usr/bin/dpkg-[a-m]* | wc -l
12
```

produces a similar outcome as,

```
$ ls /usr/bin/dpkg-[^n-z]* | wc -l
12
```

Hold on for a moment! How about we analyze the distinction (the aftereffects of the orders have been physically altered to be next to each other for clarity).

```
$ ls /usr/bin/[a-m]* | head -n 8
$ ls /usr/bin/[^n-z]* | head -n 8
/usr/bin/a2p               /usr/bin/[
/usr/bin/aconnect          /usr/bin/2to3
/usr/bin/acpi_listen       /usr/bin/2to3-2.7
/usr/bin/adb               /usr/bin/2to3-3.4
/usr/bin/add-apt-repo…     /usr/bin/7z
/usr/bin/addpart           /usr/bin/7za
/usr/bin/addr2line         /usr/bin/7zr
/usr/bin/add-remove-…      /usr/bin/a2p
```

What ’s going on is we failed to remember that square supports can match numbers and unique characters notwithstanding letters!
Note that a similar order could be refined by diverting a cat
result of each square support venture into the utility, diff.

```
$ diff --suppress-common-lines <(ls
/usr/bin/[a-m]*) <(ls /usr/bin/[^n-z]*)
```

Redirection will be canvassed exhaustively in part 13.
Knowing that numbers are substantial permits orders, for example, this:

```
$ ls /usr/bin/[0-9]*
/usr/bin/2to3      /usr/bin/2to3-3.4
/usr/bin/7za
/usr/bin/2to3-2.7  /usr/bin/7z
/usr/bin/7zr
```

Special characters require escaping,

```
$ ls /usr/bin/[\[]
/usr/bin/[
```

and work related to other expansions,

```
$ ls /usr/bin/[\[0-9a-z]
/usr/bin/[  /usr/bin/w  /usr/bin/X
```

Null globs

Null globs are basically non-matching globs. At the end of the day, on the off chance that you utilize a glob to match filenames, yet no filenames match your measures, the invalid glob will in any case match.

The basic script,

```
#!/bin/bash
echo *.txt
```

would result in the output,

```
$ ./nullglob
*.txt
```

if there didn ’t exist a document whose name matched our special case expansion,

**.txt* .

Having the glob extending to itself can be especially maddening in scripts where filenames are concerned on the grounds that you would rather not perform assignments on the trump card extension—odds are your content will mistake out.

Additionally, counters break when the glob grows to itself. The model underneath adds generally matching records each in turn to the cluster, filelist, then, at that point, yields the quantity of components in the array.

```
#!/bin/bash
filelist=(*.txt)
echo ${#filelist[@]}
$ ./nullglob
1
```

Notice how the content says there is a document that matches the development, yet there truly isn't?
Loops (canvassed in part 12) don't proceed true to form either.

```
#!/bin/bash
for file in *.txt; do
     echo "$file"
done
$ ./nullglob
*.txt
```

The appropriate response is to utilize the invalid glob shell option.

```
shopt -s nullglob
```

Now, when we run our contents to repeat any matching filenames and count the quantity of matching documents, we get the right outputs.
Echo results:

```
#!/bin/bash
shopt -s nullglob
echo *.txt
$ ./nullglob
```

Array component counting:

```
#!/bin/bash
shopt -s nullglob
filelist=(*.txt)
echo ${#filelist[@]}
$ ./nullglob
0
```

Loops:

```
#!/bin/bash
shopt -s nullglob
for file in *.txt; do
     echo "$file"
done
$ ./nullglob
```

Extended globs

Beyond “ordinary” design matching is something many refer to as standard articulations. An easier form utilizing marginally unique punctuation is called broadened globs. Like propped records, *[...]* , broadened globs take into account the matching of example, not simply individual characters.

To empower expanded globs, do as such by means of the shell option.

```
shopt -s extglob
```

You presently can match patterns.

Extended glob pattern matching

Glob Description

?(list) Matches exactly zero or one of the listed pattern. ***(list)** Matches any number of the listed patterns.

@(list) Matches exactly one of the listed patterns.

+(list) Matches at least one of the listed patterns.

!(list) Matches anything *but* the listed patterns.

A couple of direct instances of example matching are displayed in the content below.

```
#!/bin/bash
shopt -s extglob
echo *
days='*(mon*|wed*|fri*)'
echo ""
echo "mon, wed, fri: " $days
ftype='*(*txt|*doc)'
echo ""
echo "txt, doc files: " $ftype
echo "jpg only: " *(*jpg)
echo "all other files:  " !($days|$ftype)
```

And its output:

```
$ ./extglob
extglob fri.jpg mon.txt sat.omp sun.tiff
thurs.doc tues.txt wed.doc

mon, wed, fri:  fri.jpg mon.txt wed.doc

txt, doc files:  mon.txt thurs.doc tues.txt
wed.doc
jpg only:  fri.jpg
all other files:   eglob sat.omp sun.tiff
```

Regular expressions

Though mind boggling accommodating in design coordinating, the utilization of standard articulations, commonly alluded to as regex, is past the extent of this aide. There are many books and different assets out there on the off chance that you are keen on learning the language of normal expressions.

To kick you off, examine the =~ administrator of the [[keyword.

Chapter 9: Parameters

What 's the distinction among boundaries and factors, you inquire? Not all that much, and yet, a ton. Factors are really a sort of boundary—a sort that can be named. Two different sorts exist: positional (boundaries that are numbered), and exceptional (boundaries that are signified by a unique character).

Positional parameters

Remember back in the start of the aide how we discussed orders and contentions? Positional boundaries are the manner by which you pass these to your content. Beginning from 0 and proceeding up the number line, they are constantly recorded something very similar. Make sure to cite them as needed!

Positional parameters

Parameter	**Description**
$0	The called name of the script.
$1, $2, … $9	Print arguments 1 through 9 passed to the script.

${10}, ${11} . . Print arguments 10 and higher.

```
#!/bin/bash
echo "Script called is: $0"
echo "Arguments passed are: $1, $2, $3,
etc."
$ bash pnames arg1 arg2 arg3
Script called is: pnames
Arguments passed are: arg1, arg2, arg3, etc.
```

Notice how the boundary, $0, is the called name of the content? Assuming that you set the executable piece on the content and call it like this,

```
$ ./pnames a b c
Script called is: ./pnames
Arguments passed are: a, b, c, etc.
```

then the boundary becomes *./pnames* . You can without much of a stretch eliminate the main dab cut utilizing sed.

```
#!/bin/bash
echo "$0"
echo "$0" | sed "s|^\./||"
$ ./pnames
./pnames
pnames
```

Where positional boundaries give data about the running content and the contentions passed to it, unique boundaries, give more ways of yielding them, the number that are set, and extra data identified with leave codes and processes.

If you need to access more than ten positional parameters (i.e. more than *$0* through *$9*), you need to put the positional parameter within an *inline group* (curly braces). Thus, positional parameter ten is *${10}* , eleven is *${11}* , and so on.

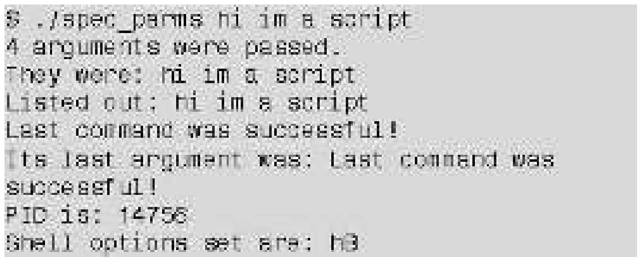
Special parameters

Special boundaries are boundaries that Bash treats in an extraordinary manner.
Helpful, I know. These are boundaries that must be referred to—you can't change them straightforwardly—and contain metadata identifying with things, for example, exit situations with, recognizable pieces of proof, shell choices, thus on.

Special parameters
Parameter Description
$#
The number of positional parameters passed to the script.
$*
All the positional parameters. If double quoted, results in a single *string* of them all.
$@
All the positional parameters. If double quoted, results in a single *list* of them all.
$?
The exit code of the last completed foreground command.
$!

The exit code of the last completed background command.
$$ The process ID of the current shell.
$_** The last argument of the last completed command. **$ The shell options that are set.

These boundaries are valuable in various ways not likely obvious at this phase of the guide.
Querying the quantity of positional boundaries is helpful when your content peruses them in and sets up its current circumstance/input/yield. As displayed in section 8, shell choices are vital to think about when utilizing globs, expanded globs, ordinary articulations, and others.
Outputting the positional boundaries in a rundown permits them to be flawlessly saved in an exhibit, while a string is more useful for human coherence. See part 6 for arrays.
Exit codes are critical with regards to restrictive squares and circles, as displayed in sections 11 and 12.
Finding a content or order's interaction ID is one of the great explanations behind utilizing traps, which is shrouded in part 15.
The result of the content resembles this:


```
$ ./spec_parms hi im a script
4 arguments were passed.
They were: hi im a script
Listed out: hi im a script
Last command was successful!
Its last argument was: Last command was
successful!
PID is: 14756
Shell options set are: h3
```

Parameter expansion

Utilization of boundary extension is remarkably useful. From changing over the instance of characters and appointing qualities to boundaries to shortening and looking and supplanting, most scripts (particularly ones that arrangement with filenames, ways, and long strings) use some type of boundary expansion.

A couple of models ought to show their power and strong convenience.
Simple usage Usage
${parameter}

${#parameter} Description
Append
Allows for additional characters beyond the end

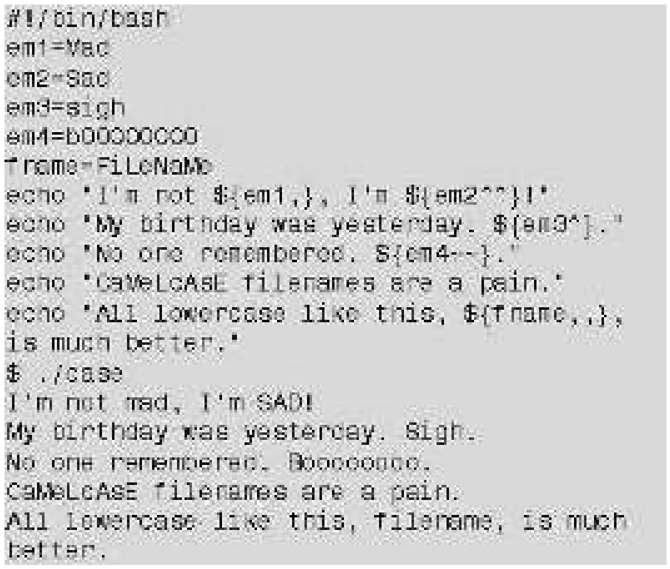
curly brace to be appended to *parameter* after substitution.
String length
The number of characters in *parameter.*

Modifying character case Usage
${parameter^}
${parameter^^}

${parameter,} Description
Uppercase
Converts the *first* character to uppercase. **Uppercase**
Converts *all* characters to uppercase. **Lowercase**
Converts the *first* character to lowercase.

${parameter,,}
${parameter~}

${parameter~~} Lowercase
Converts *all* characters to lowercase. **Reversecase**
Reverses the case of the *first* character. **Reversecase**
Reverses the case of *all* characters.



```
#!/bin/bash
em1=Mad
em2=Sad
em3=sigh
em4=bOOOOOOOO
fname=FiLeNaMe
echo 'I'm not ${em1,}, I'm ${em2^^}!'
echo 'My birthday was yesterday. ${em3^}.'
echo 'No one remembered. ${em4~~}.'
echo 'CaMeLcAsE filenames are a pain.'
echo 'All lowercase like this, ${fname,,},
is much better.'
$ ./case
I'm not mad, I'm SAD!
My birthday was yesterday. Sigh.
No one remembered. Boooooooo.
CaMeLcAsE filenames are a pain.
All lowercase like this, filename, is much
better.
```

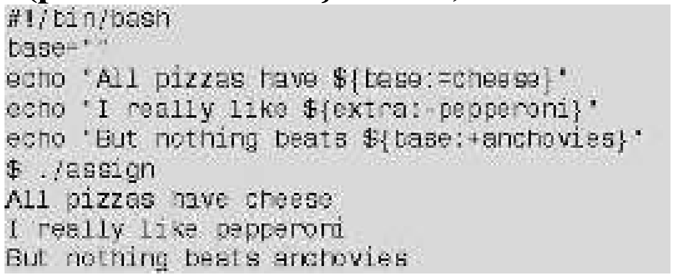
Assigning values Usage
${parameter:=value} ${parameter=value}
${parameter:+value} $parameter+word}

${parameter:-value} Description
Assign default value
Set *parameter* to *value* if *parameter* is not set,

then substitute *parameter* .
Use alternate value
Substitute nothing for *parameter* if *parameter*

is not set, else substitute *value* .
Use default value
Substitute *value* for *parameter* if *parameter* is

$(parameter-value} not set, else substitute *parameter* .



```
#!/bin/bash
base=""
echo 'All pizzas have ${base:=cheese}'
echo 'I really like ${extra:-pepperoni}'
echo 'But nothing beats ${base:+anchovies}'
$ ./assign
All pizzas have cheese
I really like pepperoni
But nothing beats anchovies
```

Substring removal Usage

${parameter:offset:length}
${parameter#pattern}
${parameter##pattern}
${parameter%pattern}

${parameter%%pattern} Description
Results in a string of *length*

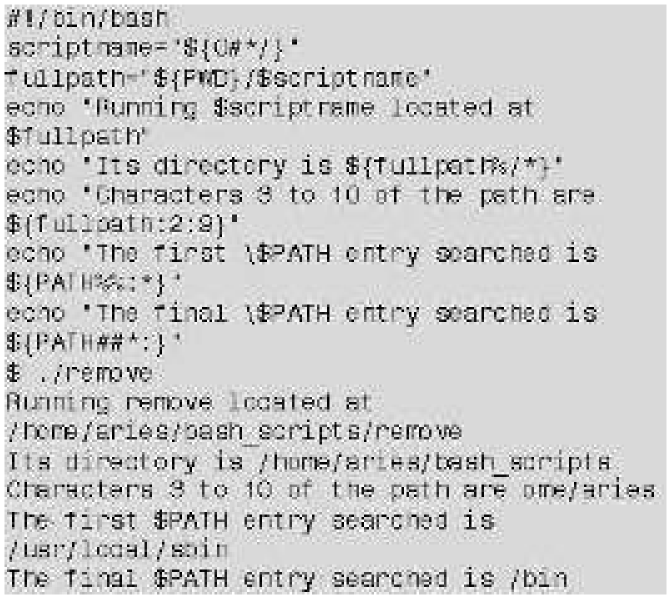
characters, starting at *offset* characters. If *length* is not specified, takes all characters after *offset* . If *length* is negative, count backwards.

Searches from front to back *parameter* until the *first* occurrence of *pattern* is found and deletes the result.

Searches from front to back of *parameter* until the *last* occurrence of *pattern* is found and deletes the result.

Searches from back to front of *parameter* until the *first* occurrence of *pattern* is found and deletes the result.

Searches from back to front of *parameter* until the *last* occurrence of *pattern* is found and deletes the result.



```
#!/bin/bash
scriptname="${0##*/}"
fullpath="${PWD}/$scriptname"
echo "Running $scriptname located at
$fullpath"
echo "Its directory is ${fullpath%/*}"
echo "Characters 3 to 10 of the path are
${fullpath:2:9}"
echo "The first \$PATH entry searched is
${PATH%%:*}"
echo "The final \$PATH entry searched is
${PATH##*:}"
$ ./remove
Running remove located at
/home/aries/bash_scripts/remove
Its directory is /home/aries/bash_scripts
Characters 3 to 10 of the path are ome/aries
The first $PATH entry searched is
/usr/local/sbin
The final $PATH entry searched is /bin
```

Search and replace Usage

${parameter/pattern/value}
${parameter//pattern/value}
${parameter/pattern}

${parameter//pattern} Description

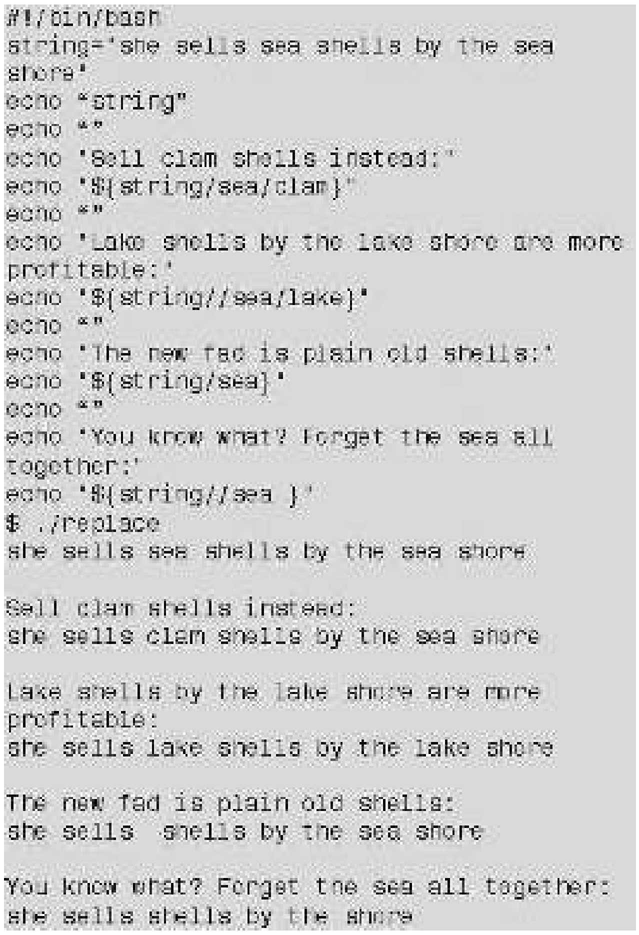
Searches from front to back of

parameter and replaces the *first* occurrence of *pattern* with *value* .

Searches from front to back of *parameter* and replaces *all* occurrences of *pattern* with *value* .

By leaving out *value* , the *first* instance of *pattern* in *parameter* is replaced with the null string.

By leaving out *value* , *all* instances of *pattern* in *parameter* are replaced with the null string.



```
#!/bin/bash
string='she sells sea shells by the sea
shore'
echo "string"
echo ""
echo 'Sell clam shells instead:'
echo "${string/sea/clam}"
echo ""
echo 'Lake shells by the lake shore are more
profitable:'
echo "${string//sea/lake}"
echo ""
echo 'The new fad is plain old shells:'
echo "${string/sea}"
echo ""
echo 'You know what? Forget the sea all
together:'
echo "${string//sea }"
$ ./replace
she sells sea shells by the sea shore

Sell clam shells instead:
she sells clam shells by the sea shore

Lake shells by the lake shore are more
profitable:
she sells lake shells by the lake shore

The new fad is plain old shells:
she sells  shells by the sea shore

You know what? Forget the sea all together:
she sells shells by the shore
```

Chapter 10: Command Substitution

Before we continue on to order replacement, there are two things to cover that are in somewhat of their very own classification. They are number juggling extension and support expansion.

First up, number-crunching development. Like any remaining types of development, number-crunching extension does precisely what its name suggests: extends and replaces itself with the numerical activity characterized within.

Arithmetic expansion

Character Description
Arithmetic expansion
$(( ))

Works exactly the same as arithmetic expression, except the entire expression is replaced with its mathematical result.

```
$ echo "$((34*52/2))"
884
```

Another model, this time utilizing client info and writing to a variable.

```
#!/bin/bash
fixed=13
result=$(($1*$2-$3+$fixed))
echo "$1 * $2 - $3 + $fixed = $result"
$ ./arithexp 8 5 99
8 * 5 - 99 + 13 = -46
```

Remember again that Bash performs whole number math. It doesn’t have the foggiest idea what floats, yearns, duplicates, or anything like that are.

```
$ echo "$((34%7))"
6
$ echo "$((34/7))"
4
```

Thus assuming your content requires an accuracy result, you should call an outer paired, for example, bc.

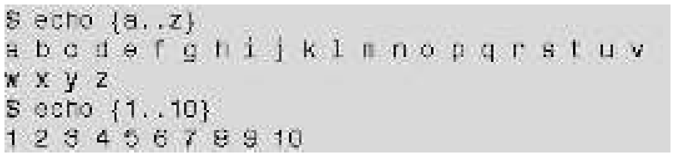
```
$ echo 'scale=2; 34/7' | bc -l
4.85
```

The scale choice sets the quantity of critical figures later the decimal and the - l switch characterizes the standard mathematical library to be utilized. Ideally this will save you from a couple headaches.

Next up is support extension, which is pretty much a method of making lists.

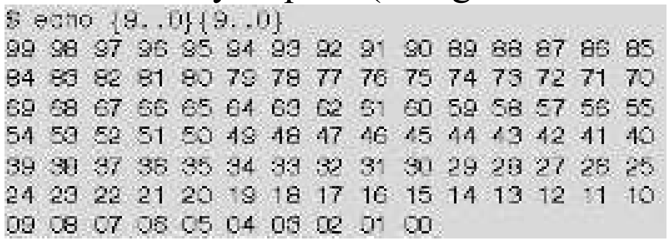
Brace expansion

Character Description
Brace expansion
{} Expands the character sequence within.
The most straightforward utilization of support development is to extend a rundown of letters or numbers.



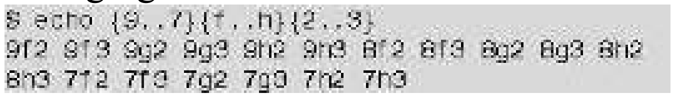
```
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v
w x y z
$ echo {1..10}
1 2 3 4 5 6 7 8 9 10
```

Brace development can be utilized in any request (rising or descending)



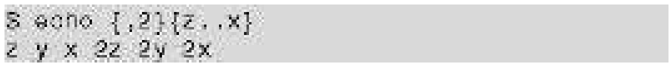
```
$ echo {9..0}{9..0}
99 98 97 96 95 94 93 92 91 90 89 88 87 86 85
84 83 82 81 80 79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
54 53 52 51 50 49 48 47 46 45 44 43 42 41 40
39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
24 23 22 21 20 19 18 17 16 15 14 13 12 11 10
09 08 07 06 05 04 03 02 01 00
```

and doesn't play out any arranging of the results.



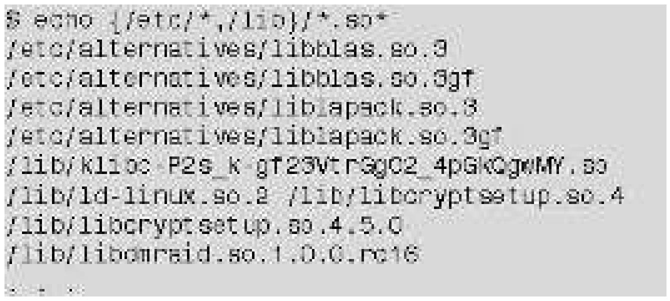
```
$ echo {9..7}{f..h}{2..3}
9f2 9f3 9g2 9g3 9h2 9h3 8f2 8f3 8g2 8g3 8h2
8h3 7f2 7f3 7g2 7g3 7h2 7h3
```

Brace extension acknowledges void data sources (notice the arrangement of the comma in the main support, which show the thing going before it was a void item).



```
$ echo {,2}{z..x}
z y x 2z 2y 2x
```

Finally, it can grow to pathnames as well.



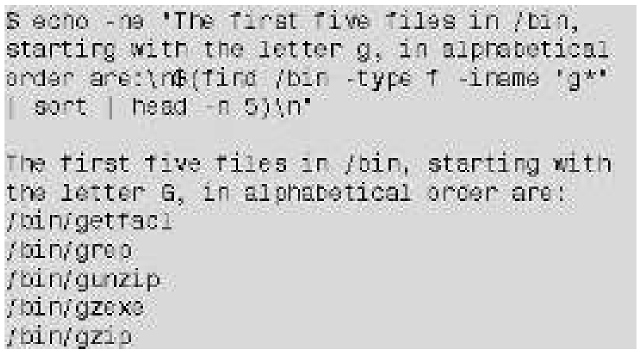
```
$ echo {/etc/*,/lib}/*.so*
/etc/alternatives/libblas.so.3
/etc/alternatives/libblas.so.3gf
/etc/alternatives/liblapack.so.3
/etc/alternatives/liblapack.so.3gf
/lib/klibc-P2s_k-gf23VtrGgO2_4pGkQgwMY.so
/lib/ld-linux.so.2 /lib/libcryptsetup.so.4
/lib/libcryptsetup.so.4.5.0
/lib/libdmraid.so.1.0.0.rc16
```

Command substitution

Now on to the genuine subject of this part, order replacement. What happens during command substitution is that Bash spawns a subshell, passes it the expression and evaluates (executes) it, then takes its standard output and substitutes it in for the original substitution syntax, *$(..)*

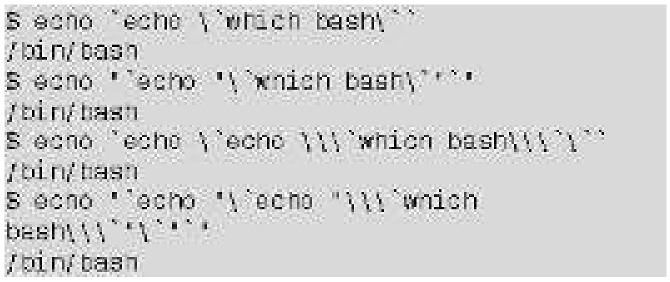
One point bears rehashing: order replacement performed inside a subshell. It is a vital qualification. It implies that the outcomes go through word parting and pathname expansion.

Character Description
Command substitution
\` \` The output of the command within the backticks are run and substituted in place of the entire backtick grouping. **Command substitution**
$( ) Exactly the same as backticks above, but provide nestability and better readability.



```
$ echo -ne 'The first five files in /bin,
starting with the letter g, in alphabetical
order are:\n$(find /bin -type f -iname 'g*'
| sort | head -n 5)\n'

The first five files in /bin, starting with
the letter G, in alphabetical order are:
/bin/getfacl
/bin/grep
/bin/gunzip
/bin/gzexe
/bin/gzip
```

Though backticks perform the same function as *$(..)* , they should **not** be used. They are essentially included in Bash only to support legacy code. Backticks don't settle without getting away from themselves and other exceptional characters, are hard to peruse, amazingly appalling, and are not POSIX compatible.



```
$ echo `echo \`which bash\``
/bin/bash
$ echo "`echo "\`which bash\`"`"
/bin/bash
$ echo `echo \`echo \\\`which bash\\\`\``
/bin/bash
$ echo "`echo "\`echo "\\\`which
bash\\\`"\`"`"
/bin/bash
```

By examination, $(*..) looks overall quite slick and doesn't need any escaping* .

```
$ echo $(echo $(which bash))
/bin/bash
$ echo "$(echo "$(which bash)")"
/bin/bash
$ echo $(echo $(echo $(which bash)))
/bin/bash
$ echo "$(echo "$(echo "$(which bash)")")"
/bin/bash
```

Here a few instances of setting factors and repeating yield that are considerably more valuable than those shown above.

```
$ today="$(date)"
$ right_now="$(date +'%Y%m%d-%H%M')"
$ line_cnt="$(wc -l flist | cut -d' ' -f1)"
$ echo -e 'Users online: $(w)'
$ cwd="$(pwd)"
$ echo "File list: $(ls)"
$ host_and_user="$(whoami; hostname) "
$ echo "User $(id -gn) has uid $(id -u) is
in groups $(id -Gn)"
```

Chapter 11: Conditional Blocks

A restrictive assertion is a method for changing the progression of a content dependent on a boolean condition—a condition that assesses to one or the other valid or bogus. In languages such as C++, the evaluation of a conditional statement becomes either 1 for true, or 0 for false In Bash however, a *0 is true* , and *any other number is false* (though it is typically a 1). The purpose for this is a result of something many refer to as exit codes.

Exit codes

Exit codes are exactly what they sound like: codes (an integer between 0 and or 255[i.e. 2^8 values]) that indicate the previous command's termination status. A 0 indicates successful completion (i.e. a true). A 1 is the default value for an unsuccessful completion (i.e. a false).

You can check the leave code produced by the last forefront order by utilizing the extraordinary boundary, $?.

```
$ echo $?
0
```

Two unique orders we can use to confirm this usefulness are valid and *bogus* . From their man
pages:

```
$ true          # do nothing, successfully
$ false         # do nothing, unsuccessfully
```

Thus, genuine consistently finishes with a 0 leave code, and bogus consistently finishes with a 1 as its exit code.

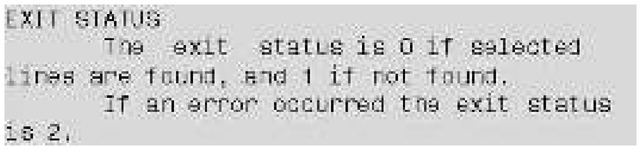
```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

Values over 1 all demonstrate fruitless fulfillment, however are commonly set inside the actual content. For instance, you could set it so inability to make a record exits with code 60, while inability to erase a document exits with code 80. Thusly, in light of on the content’s leave code, you can determine what occurred (or didn’t). This is cultivated with the exit fabricated in.

```
$ touch file || { echo 'Error creating file'
>&2; exit 60; }
$ rm file || { echo 'Error deleting file'
>&2; exit 80; }
```

In the above model, the twofold lines (||, the sensible OR administrator) cause the ensuing reverberation/leave order gathering to execute provided that the first order, either contact or rm, come up short. The redirection, *>&2* , causes the blunder message to be printed to sterr, rather than bold. Both of these ideas will be shrouded later in the guide.

Exit then, at that point, sets the leave code as a worth other than the catchall, 1, which makes investigating more straightforward and investigating easier .
Many projects utilize custom blunder codes as a method for giving more data concerning what turned out badly. Here is a model from grep’s man page:



```
EXIT STATUS
       The  exit  status is 0 if selected
lines are found, and 1 if not found.
       If an error occurred the exit status
is 2.
```

Standard blunder codes are recorded below.
Exit Code Description Example 0 Successful execution. true 1 Unsuccessful execution catchall. false 2 Incorrect use of a shell builtin ls — option 126 Command cannot execute /dev/null 127 Command not found (typically a touchtypo)

128 Incorrect exit code argument exit 2.718 kill -9 *pid* # num =

128+num
Fatal error signal "num"

137 kill -15 *pid* # num =

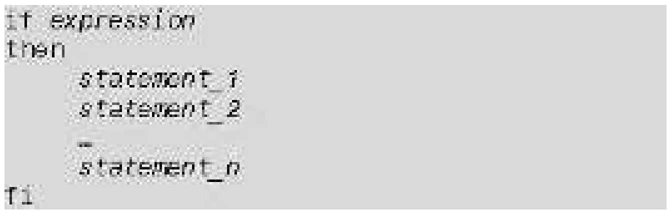
143 130 Script killed with Control-C [CTRL]-[c] 255* Exit code out of range exit 666

Now that we thoroughly understand leave codes, we can begin utilizing them for more than straightforward mistake handling.

If

The easiest sort of contingent square is the if block. *Assuming is a shell watchword that checks the leave code of an order and executes one order block assuming that it finished effectively, and another assuming it did not* .

The most essential utilization of this restrictive square is the straightforward if statement.



```
if expression
then
     statement_1
     statement_2
     …
     statement_n
fi
```

Depending on your own

inclinations, it is entirely adequate to arrange the if articulation in the accompanying ways:

Beginning with the main catchphrase, *if,* the contingent articulation is assessed. Ifit is true (its Boolean value is 0), the *statements* (commands) between the keywords *then* and *fi* are executed (in Bash, multi-line statements like *if* and *case* terminate with their character reverses, *fi* , and *esac*). Assuming that the articulation is bogus, every one of the ensuing orders are skipped and the content proceeds as ordinary later the fi keyword.

A minor example:

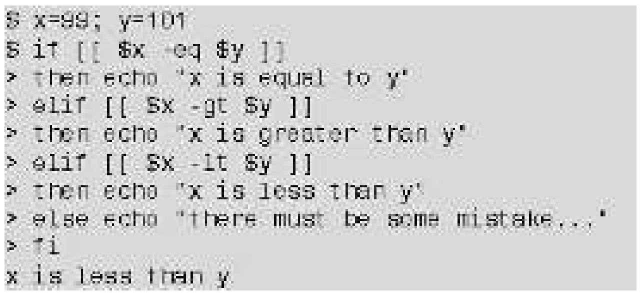
```
$ if true
> then echo 'Why hello there!'
> fi
Why hello there!
```

What to accomplish something dependent on the articulation being bogus? If
else articulations are the best approach . Everything you do is add another catchphrase, else, with the assertions to be executed in case of a bogus evaluation.

```
if expression
then
     statements
else
     other_statements
fi
$ if false
> then echo 'Why hello there!'
> else echo 'Goodbye!'
> fi
Goodbye!
```

Though the Boolean can just convey two qualities, valid or bogus, various discrete tests can be pursued one the other by utilizing an expanding if-elif*else statement* .

Borrowing test (twofold square sections) from a couple of pages further on, we can see this in real life. Here, test is verifying whether the variable x is equivalent (- eq), more prominent than (- gt), or not exactly (- lt) the variable y.

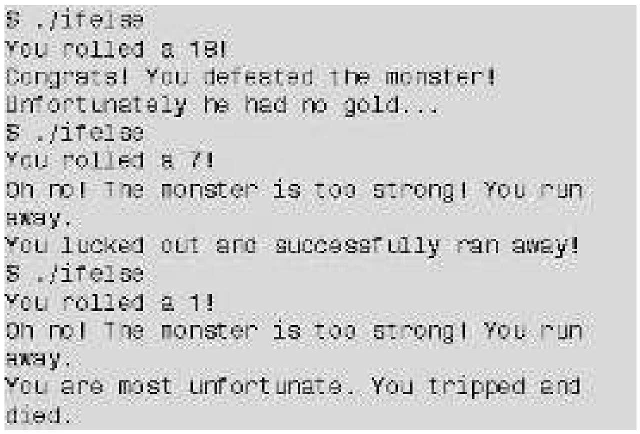


```
$ x=99; y=101
$ if [[ $x -eq $y ]]
> then echo 'x is equal to y'
> elif [[ $x -gt $y ]]
> then echo 'x is greater than y'
> elif [[ $x -lt $y ]]
> then echo 'x is less than y'
> else echo "there must be some mistake..."
> fi
x is less than y
```

Building on the expanding proclamation, various if-else articulations can be settled, permitting you to test further down the tree. This is cultivated with a settled if-else statement.

Here is a model that joins all we take care of about if-else statements.

And its output:



```
$ ./ifelse
You rolled a 18!
Congrats! You defeated the monster!
Unfortunately he had no gold...
$ ./ifelse
You rolled a 7!
Oh no! The monster is too strong! You run
away.
You lucked out and successfully ran away!
$ ./ifelse
You rolled a 1!
Oh no! The monster is too strong! You run
away.
You are most unfortunate. You tripped and
died.
```

Attention to arranging can incredibly decrease migraines when composing restrictive squares! Assuming you wind up requiring mutiple or two if proclamations, you should investigate utilizing the case statement.

Case

The catchphrase, case, is a method for making a move contingent upon the substance of a variable without utilizing an inconvenient series of expanding if explanations. It gives an overall quite perfect method for matching a variable against a few potential patterns.

```
case variable in
pattern1)
        statements1
        ;;
pattern2)
        statements2
        ;;
pattern3)
        statements3
        ;;
*)
        statements*
        ;;
esac
```

What happens when the content experiences the case catchphrase is that it look start to finish for the principal matching example. When found, it executes the assertions (orders) until it arrives at the twofold semi colon (;;) watchword. Assuming the variable doesn't match a particular example, the last case, **) fills in as a catch all . esac ends the case .*

It merits rehashing that case look all together and just executes the principal matching example. Once executed, case breaks (closures, and forges ahead to the principal line following esac).

Perhaps the most predominant use of the case proclamation is in a standard startup script.

```
case "$1" in
    start)
        start_service_name
    ;;
    stop)
        stop_service_name
    ;;
    force-reload|restart)
        stop_service_name
        start_service_name
    ;;
    *)
        echo "Usage: $0 {start|stop|force-
reload|restart}"
        exit 3
    ;;
esac
```

Notice that at least two examples can return a similar outcome by isolating them with a line, |. It is additionally conceivable to utilize globs for design matching inside a case statement.

```
case "$LANG" in
     de*)
          echo "Setting language to German."
          export LANG=de_DE.UTF-8
     ;;
en*)
          echo "Setting language to
English."
          export LANG=en_US.UTF-8
     ;;
     *)
          echo "Sorry, your language is not
supported."
     ;;
esac
```

You can utilize a case square to make a yes or no prompt:

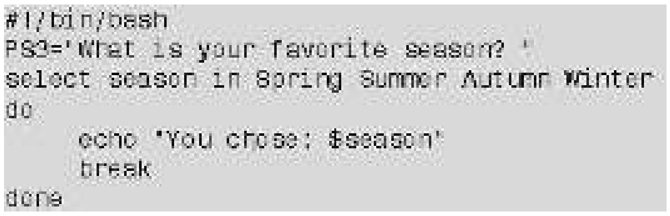
```
#!/bin/bash
read -p 'Are you sure you want to continue
(y/n)? ' choice
case '$choice' in
     Y|y) echo 'some statements';;
     N|n) echo 'other statements';;
     *) echo 'bad selection!';;
esac
```

Select

Select is to some degree like case in that a progression of variable choices are inspected, yet while case takes a formerly set variable and matches it against an example, *select prompts the client to make a choice (thus the name)* . Another distinction is that select behaves like a circle (more on circles to come), and just exits once it gets the break catchphrase (in any case it proceeds to redisplay the menu).

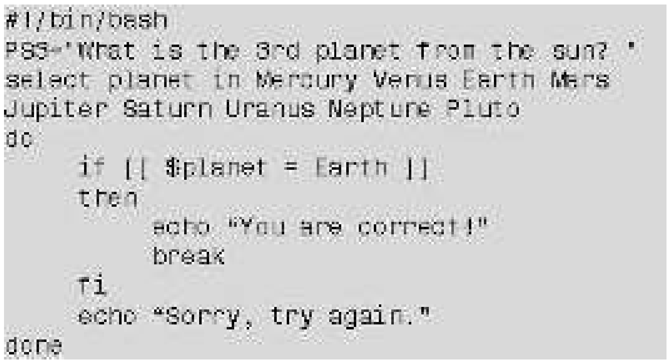
```
PS3="Prompt title goes here"
select variable in value1 value2 . . .
valueN
do
     statements1
     statements2
     . . .
     statementsN
     break
done
```

It is genuinely essential, however proves to be useful in number of various circumstances. You can utilize it to set a variable.



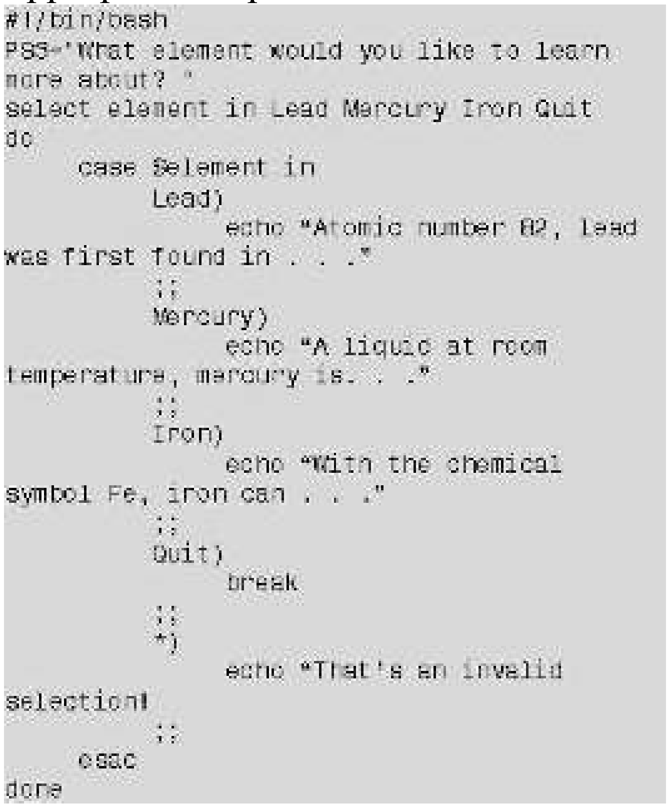
```
#!/bin/bash
PS3='What is your favorite season? '
select season in Spring Summer Autumn Winter
do
     echo 'You chose: $season'
     break
done
```

You can make a test.



```
#!/bin/bash
PS3='What is the 3rd planet from the sun? '
select planet in Mercury Venus Earth Mars
Jupiter Saturn Uranus Neptune Pluto
do
     if [[ $planet = Earth ]]
     then
          echo "You are correct!"
          break
     fi
     echo "Sorry, try again."
done
```

Obviously you ought to presumably randomize the appropriate responses! You can utilize it to give more data about a subject.



```
#!/bin/bash
PS3='What element would you like to learn
more about? '
select element in Lead Mercury Iron Quit
do
     case $element in
          Lead)
               echo "Atomic number 82, lead
was first found in . . ."
          ;;
          Mercury)
               echo "A liquid at room
temperature, mercury is. . ."
          ;;
          Iron)
               echo "With the chemical
symbol Fe, iron can . . ."
          ;;
          Quit)
               break
          ;;
          *)
               echo "That's an invalid
selection!
          ;;
     esac
done
```

And so on thus forth.

Conditionals

Conditional Description
Negate
!
Negates a test or exit status.

If iss ued from the command line, invokes Bash's history mechanism.
Test
The expression between the brackets is tested and **[ ]** results in a true/false output. This is a shell builtin. If not concerned with compatibility with the *sh* shell, the double square bracket test is much more flexible.

Test
Tests the expression between the brackets but

[[ ]] provides many more options, features, and flexibility than single square brackets. It is a shell keyword. Tests will be covered later.

[

[, or test, is a shell builtin. It is utilized to test and assess contingent squares as one or the other valid or false.
For the most part it has been made out of date by [[, which is a shell watchword. While by far most of tests are a similar whether utilizing [or [[(allude to the [[area for every one of the tests, their depictions and utilizations), there are a couple of outstanding contrasts. Except if you are keeping up with inheritance code or realize your content should be sh viable, you should utilize consistently utilize [[.

[vs [[
[[doesn't need citing of factors. [does.

```
$ report='TPS Report'; filename='My file'
$ [ $report = $filename ]
bash: [: too many arguments
$ echo $?
2
```

Though it shows up there are just two contentions in the above test, every factor is extended to its set worth before tried. [doesn’t have the foggiest idea how to deal with this without citing. Also, while [[is restricted to four contentions, [is overburdened with more than two.

[[doesn ’t need getting away from extraordinary characters, (for example, the more prominent than, >, and not as much as characters, <). With [, you should get away from all unique characters.

```
$ x=100; y=200
$ [ $x \> $y ]
$ echo $?
1
$ [ $x \< $y ]
$ echo $?
0
```

[[can use coherent AND (&&) and additionally (||) administrators. [requires the utilization of
- a for &&, and - o for ||) .
[[can perform design coordinating while [cannot:

```
$ [[ frame = *.jpg ]] || echo "$fname is not
a jpg file"
```

[[can perform ordinary articulation coordinating while [cannot:

```
$ [[ "$emailaddr" =~ '^[A-Za-z0-9._%+-
]+<b>@</b>[A-Za-z0-9.-]+<b>\.</b>[A-Za-
z]{2,4}$' ]] || echo "emailaddr is not a
properly formatted email address"
```

[[

[[is the “new” [. Very much like [, [[is utilized to test and assess contingent squares as one or the other valid or bogus. The least demanding method for seeing how it functions is by seeing what tests it can perform, and how they are performed.

Remember that

```
$ echo $?
```

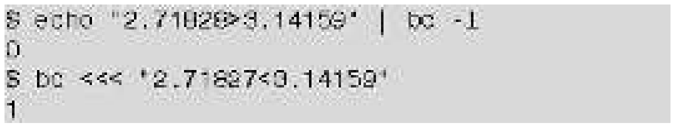
returns the leave code of the last

finished closer view order (genuine is 0, bogus is 1).

Arithmetic tests
Operator Description
X **-eq** *Y* True if *X* is **equal to** *Y*
X **-ne** *Y* True if *X* is **not equal to** *Y*
X **-gt** *Y* True if *X* is **greater than** *Y*
X **-lt** *Y* True if *X* is **less than** *Y*
X **-le** *Y* True if *X* is **less than or equal to** *Y*
X **-ge** *Y* True if *X* is **greater than or equal to** *Y*

All of these tests are self-explanatory.
It is essential to recall that Bash just gets whole numbers. Assuming you want to test drifting point numbers, you should utilize an outer program, for example, bc. It acknowledges the representative administrator equivalents.

[[operator bc equivalent
-eq ==
-ne !=
-gt >
-lt < *Y*
-le <=
-ge >=



```
$ echo "2.71828>3.14159" | bc -l
0
$ bc <<< '2.71827<3.14159'
1
```

Note that bc returns 0 assuming the connection is bogus and 1 assuming the connection is true!

String tests
Operator
-z *STRING*
-n *STRING*
STRING1 = STRING2 STRING1 == STRING2

STRING1 != STRING2
STRING1 < STRING2
STRING1 > STRING2

Description
True if *STRING* is **empty**
True if *STRING* is **not empty**
True if *STRING1* is **equal to** *STRING2* True if *STRING1* is **equal to** *STRING2* True if *STRING1* is **not equal to**

STRING2
True if *STRING1* is **sorts lexicographically before** *STRING2* (must be escaped if using [)
True if *STRING1* is **sorts lexicographically after** *STRING2* (must be escaped if using [)
These string tests can be utilized in various helpful ways. Maybe the most direct is verifying whether a variable is set or not.

```
$ [[ -z $name ]] && echo 'I don't know whose
name it is!'
I don't know whose name it is!
[[ -n $name ]] && echo 'You already have a
name set as $name. If you proceed it will be
overwritten. Are you sure?'
You already have a name set as John Smith.
If you proceed it will be overwritten. Are
you sure?
```

A powerful helpful follow-up is to utilize perused and brief the client for input, like the characters Y or y for indeed, and N or n for no. The utilization of read will be shrouded later in the aide. For a fast model, bounce down to regular articulation design matching.

Note that the = administrator isn't as old as eq administrator from the above number juggling test. The equivalents sign (=) is really a string test. Accordingly the test,

```
$ [[ 01 = 1 ]]
```

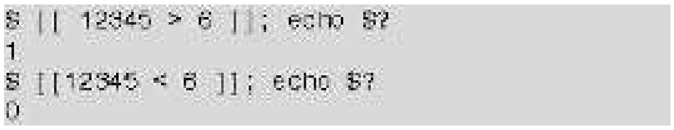
would actually return a value of false, while the test,

```
$ [[ 01 -eq 1 ]]
```

would return a value of true.

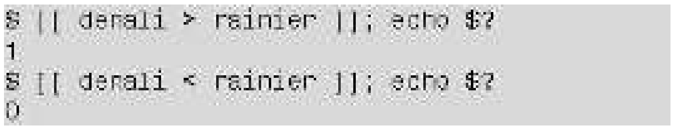
```
$ [[ 01 = 1 ]]
$ echo $?
1
$ [[ 01 -eq 1 ]]
$ echo $?
0
```

Again, I should rehash that the emblematic administrators are not numbercrunching administrators. They are string administrators. The word reference would sort 12345 preceding 6, not on the grounds that 12345 is under 6, but since the first person in quite a while before the principal character in 6.



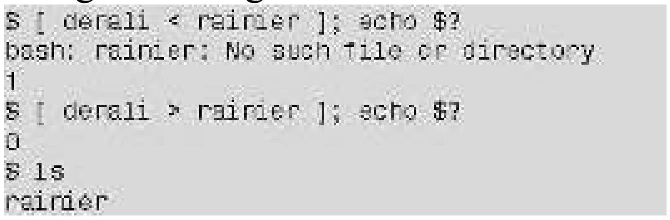
```
$ [[ 12345 > 6 ]]; echo $?
1
$ [[12345 < 6 ]]; echo $?
0
```

Working with letters makes more sense.



```
$ [[ denali > rainier ]]; echo $?
1
$ [[ denali < rainier ]]; echo $?
0
```

If you use [rather than [[, you will get startling behavior.



```
$ [ denali < rainier ]; echo $?
bash: rainier: No such file or directory
1
$ [ denali > rainier ]; echo $?
0
$ ls
rainier
```

This is on the grounds that the images < and > have exceptional importance in Bash. [thinks
< is attempting to divert the substance of document rainier to the record denali,
while > is endeavoring the inverse. The blunder message and record creation are
a direct result
of how those administrators work when the filename contentions don't exist.

File tests

All filenames should be cited if utilizing [.
Operator Description
-e *FILE* True if *FILE* exists
-f *FILE* True if *FILE* exists and is a **regular** file
-d *FILE* True if *FILE* exists and is a **directory**

-c
FILE
True if *FILE* exists and is a **special character** file
-b *FILE* True if *FILE* exists and is a **special block** file
-p
FILE

True if *FILE* exists and is a **named pipe** (FIFO —"first in, first out")
-S *FILE* True if *FILE* exists and is a **socket** file

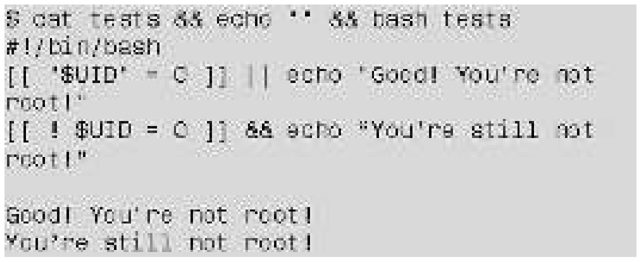
-h *FILE* True if *FILE* exists and is a **symbolic link**
-O
FILE
True if *FILE* exists and is **effectively owned by you**
-G
FILE
True if *FILE* exists and is **effectively owned by your group**

-g *FILE* True if *FILE* exists and has **SGID** set
-u *FILE* True if *FILE* exists and has **SUID** set
-r *FILE* True if *FILE* exists and is **readable** by you
-w *FILE* True if *FILE* exists and is **writable** by you
-x *FILE* True if *FILE* exists and is **executable** by you

-s
FILE
True if *FILE* exists and is **non-empty** (size is > 0)
-t
FD
True if *FD* (file descriptor) is opened on a terminal
FILE1 **-nt** True if *FILE1* is **newer than** *FILE2* *FILE2*
FILE1 **-ot** True if *FILE1* is **older than** *FILE2* *FILE2*
FILE1 **-ef** True if *FILE1* and *FILE2* refer to the **same**

FILE2 **device and inode numbers**
The use of a couple of the administrators are displayed underneath. The

rest, their capacities obvious and uses clear, are left as an activity for the reader.

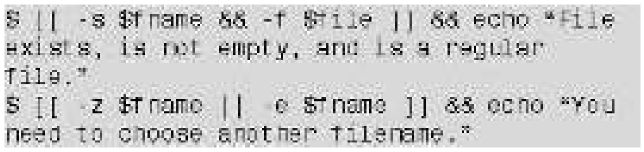


```
$ cat tests && echo '' && bash tests
#!/bin/bash
[[ '$UID' = 0 ]] || echo 'Good! You're not
root!'
[[ ! $UID = 0 ]] && echo "You're still not
root!"

Good! You're not root!
You're still not root!
```

Logical tests Operator
EXP1 **&&**

EXP2
EXP1 || *EXP2*
Description
True if **both** *EXP1* **and** *EXP2* are **true** (equivalent of **-a** if using [])
True if **either** *EXP1* **or** *EXP2* is **true** (equivalent of **-o** if using [])

Both of these administrators were covered already, but in an alternate setting. There are no progressions in the manner they work to mention.



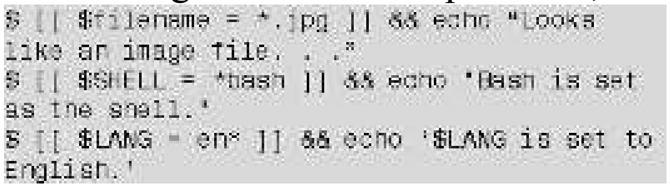
```
$ [[ -s $fname && -f $file ]] && echo "File
exists, is not empty, and is a regular
file."
$ [[ -z $fname || -e $fname ]] && echo "You
need to choose another filename."
```

Pattern tests
All example tests are [[only.
Operator
STRING = *PATTERN*
STRING == *PATTERN*

STRING =~ *PATTERN*

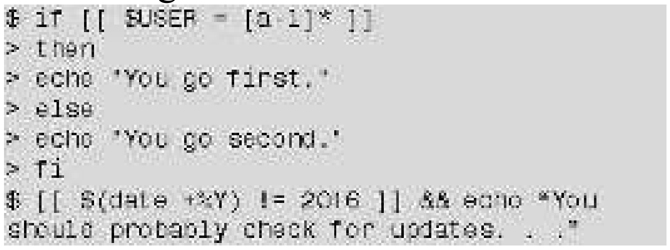
Description
True if *STRING* **matches** *PATTERN* True if *STRING* **matches** *PATTERN*
True if *STRING* **matches the regular**

expression pattern, *PATTERN*
Pattern matching is very advantageous for undertakings including strings which cling to a norm or explicit use, for example, filename extensions.



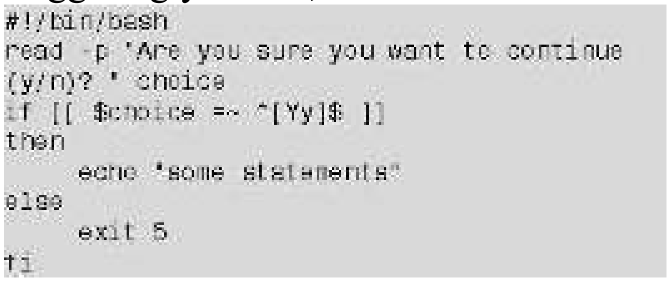
```
$ [[ $filename = *.jpg ]] && echo "Looks
like an image file. . ."
$ [[ $SHELL = *bash ]] && echo 'Bash is set
as the shell.'
$ [[ $LANG = en* ]] && echo '$LANG is set to
English.'
```

Obviously you can likewise do other things.



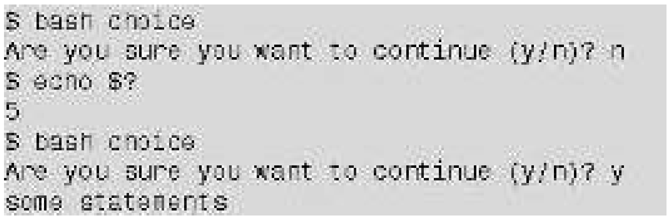
```
$ if [[ $USER = [a-l]* ]]
> then
> echo 'You go first.'
> else
> echo 'You go second.'
> fi
$ [[ $(date +%Y) != 2016 ]] && echo "You
should probably check for updates. . ."
```

Regular articulations are staggeringly useful, however their utilization is past the extent of this guide.



```
#!/bin/bash
read -p 'Are you sure you want to continue
(y/n)? ' choice
if [[ $choice =~ ^[Yy]$ ]]
then
    echo "some statements"
else
    exit 5
fi
```

The result would look like this:



```
$ bash choice
Are you sure you want to continue (y/n)? n
$ echo $?
5
$ bash choice
Are you sure you want to continue (y/n)? y
some statements
```

Pattern matching tests
Though this segment falls under customary articulations and is past the extent of this aide, Bash additionally upholds a couple of character classes of example coordinating. The classes upheld are the ones characterized in the POSIX standard.

Class Description
alnum

Alphanumeric characters. Equivalent to specifying both *alpha* and *digit* , or [0-9A-Za-z].

alpha Alphabetic characters. Equivalent to [A-Za-z]. *ascii* ASCII characters.
blank Space and tab.

cntrl

Control characters. These are the characters with octal codes 000 through 037, and 177.
digit Digits. Equivalent to [0-9].
graph Visible characters. Equivalent to *alnum* and *punct. lower* Lowercase characters. Equivalent to [a-z].

print
Visible characters and spaces. Equivalent to *alnum* , *punct* , and space.
punct
Punctuation characters.
! @ # $ % ^ & * ( ) { } [ ] _ + - , . : ; < > = \| / ' " ` ~ ? *space*

Whitespace characters. Tab, newline, vertical tab, newline, carriage return, space, and form feed.
upper Uppercase characters
word Alphanumeric characters plus "_"
xdigit Hexadecimal digits

The person classes are indicated with the syntax:
`[:class:]` For more data on normal articulation coordinating, counsel one of the numerous manuals on the subject.

Miscellaneous
Operator Description
! *TEST* **Inverts** the result of *TEST*
(*TEST* **Groups** a *TEST* (changes precedence of tests, must be

) escaped if using [])
-o *OPT* True if shell option, *OPT* , **is set**
-v *VAR* True if variable, *VAR* , **is set**

A typical shell choice to turn on for assuming you are repeating filenames is nullglob, which fixes things such that the glob, *, isn’t dealt with in a real sense assuming the catalog you are getting filenames from is empty.

```
if [[ ! -o nullglob ]]
then
    echo "Turning on shopt nullglob. . ."
    shopt -s nullglob
fi
```

Grouping proclamations works precisely as covered previously. A model is to check that possibly you or one of the gatherings you have a place with own a record prior to digging any further and say, checking whether the document has had a reinforcement created.

```
# say "$fname" == server_config
# expression is true if "server_config" is
owned by your user or users' groups, and a
file exists in the directory called
"server_config.backup"
$ [[ ( -O '$fname' || -G "$fname' ) && -e
'${fname}.backup" ]]
```

Chapter 12: Conditional Loops

As of recently, all the prearranging you have done, regardless of whether looking for a boundary and adjusting it, setting a variable, printing text to a terminal, or playing out a test, runs once and precisely once. A circle is consistent with its and completes your orders on different occasions in progression. Contingent upon the sort of circle you use and the models you indicate for it, it can run until an order fizzles (fruitlessly executes; some time circle), succeeds (effectively executes; an until circle), or a predetermined number of times (iterative; a for loop).

Before plunging into circle language structure, it is valuable to know (since botches bringing about endless circles occur!) that circles can be hindered by conveying it the SIGINT message through the console succession [CTRL][C]. The same order is:

```
$ kill -2 PID
```

Later in the aide we will cover traps, which are a method for getting signals (however not all are catchable) and perform custom activities dependent on which sign was received.

While

The while circle is quite basic. It assesses a condition and executes the assertions inside the circle until the conditions returns a non-genuine leave code (1 or higher).

```
while condition
do
     statement1
     statement2
     . . .
     statementN
done
```

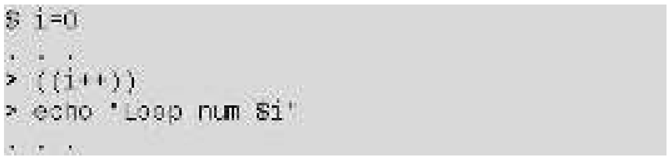
The least complex form is an endless loop.

```
$ i=0
$ while true
> do
> echo "iteration $((i++))"
> done
iteration 1
iteration 2
iteration 3
. . .
# and so on to infinity
```

Be ready to hit [CTRL][C]!
A more valuable adaptation goes through the circle a predetermined number of times.

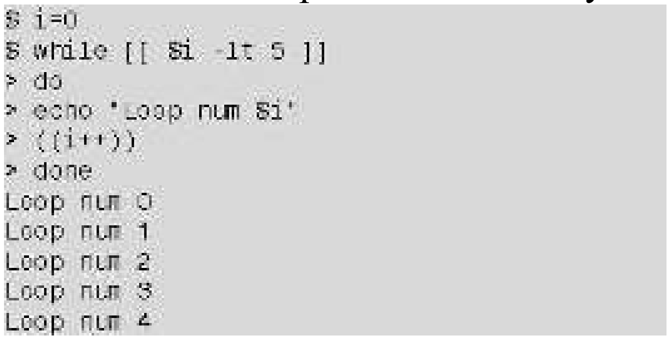
```
$ i=0
$ while [[ $i -lt 5 ]]
> do
> ((i++))
> echo "Loop num $i"
> done
Loop num 1
Loop num 2
Loop num 3
Loop num 4
Loop num 5
```

Note in the above model that the circle counter began at 0, then augmented to 1 preceding repeating the circle number.



```
$ i=0
. . .
> ((i++))
> echo 'Loop num $i'
. . .
```

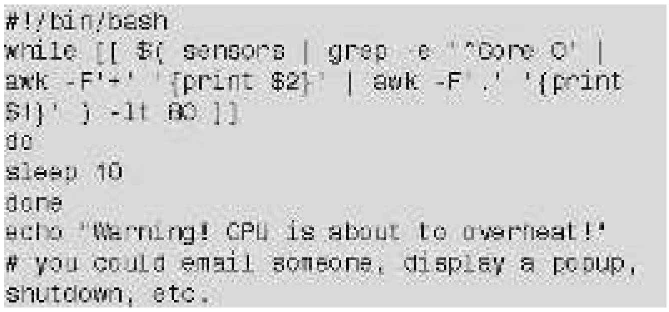
If those lines were turned around, the main circle emphasis would be cycle 0.



```
$ i=0
$ while [[ $i -lt 5 ]]
> do
> echo 'Loop num $i'
> ((i++))
> done
Loop num 0
Loop num 1
Loop num 2
Loop num 3
Loop num 4
```

Regardless the outcome is as yet five circles, with both ordered at 0 (which is the reason the content uses - lt 5 and not eq 5), but rather the showed numbers between the two are different.

Here is a convenient content that cautions you when your CPU temperature is getting excessively hot. What is does is really take a look at the CPU temperature of "Center 0" through the sensors order (from the lm_sensors bundle) and checks whether it is under 80 degrees Celsius. Assuming that it will be, it holds up ten seconds prior to attempting once more. Assuming the CPU temperature transcends 80, the circle breaks and the lines following are executed.



```
#!/bin/bash
while [[ $( sensors | grep -e '^Core 0' |
awk -F'+' '{print $2}' | awk -F '.' '{print
$1}' ) -lt 80 ]]
do
sleep 10
done
echo "Warning! CPU is about to overheat!"
# you could email someone, display a popup,
shutdown, etc.
```

The monstrous line of Bash code that snatches the temperature would be an ideal use
for a capacity (shrouded later on in the guide).
Another model is to utilize [[and shift to push through all the set positional

parameters.

```
$ cat shift; echo ''; bash shift abc 123 foo
bar

#!/bin/bash
i=0
while [[ ! -z "$1" ]]
do
((i++))
echo "Positional param $i is: $1"
shift
done

Positional param 1 is: abc
Positional param 2 is: 123
Positional param 3 is: foo
Positional param 4 is: bar
```

Until

The until circle is by and large something contrary to the while circle. It runs a test and executes the assertions inside the circle until the order returns a genuine leave code (0).

```
until condition
do
     statement1
     statement2
     . . .
     statementN
done
```

The relating boundless circle looks like this:
It is somewhat aggravating in the above setting. You would not be distant from everyone else in perusing it as, “until [the condition] is bogus, do. . .”. Be that as it may, since bogus consistently returns a leave code of 1, and until runs until it gets a leave code of 0, the right perusing of the above boundless circle would be, “until the assertion bogus turns out to be valid (which won’t ever occur), do. . .”

Here is a more reasonable example:

```
#!/bin/bash
echo -n 'waiting for host 10.0.0.209 to be
up. '
until ping -c 1 10.0.0.209 1> /dev/null
do
echo -n '. '
sleep 1
done
echo "host is up!"
```

This model runs ping one time each second and verifies whether it gets any reaction from have 10.0.0.209. When it gets a reaction, *ping returns a genuine leave code (0)* , and the until circle exits. The - n switch on reverberation stifles the following newline from being added, and the 1>/dev/invalid sends all standard result to/dev/invalid for a cleaner-looking screen.

The result will look something like this:

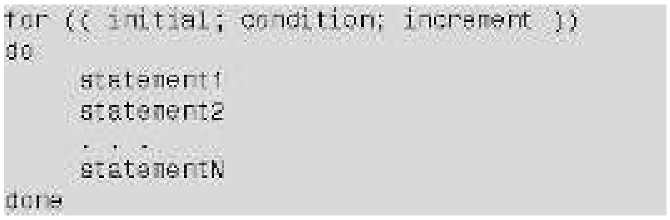
```
$ bash ping_host
waiting for host 10.0.0.209 to be up. . . . .
. . . . . . . . . host is up!
```

The until circle adaptation of the CPU temperature model requires changing the measures from not exactly to more noteworthy than.
Cycling through the positional boundaries utilizing an until circle looks like this:

```
#!/bin/bash
i=0
until [[ -z "$1" ]]
do
((i++))
echo "Positional param $i is: $1"
shift
done
```

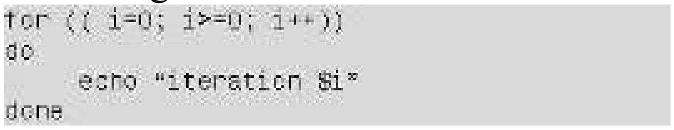
For

Arguably quite possibly the most widely recognized loop (as it has been around always—it was initially executed in C/C++ in the mid 1970s) is the for circle, which uses a variable, a condition, and an increment.



```
for (( initial; condition; increment ))
do
    statement1
    statement2
    . . .
    statementN
done
```

An endless circle would look something like this:

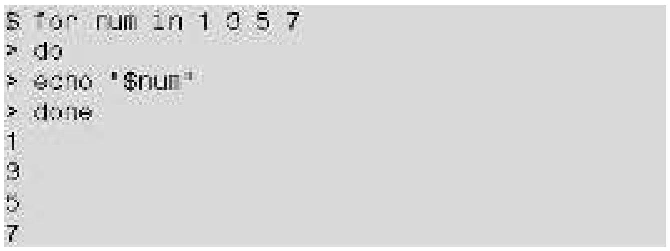


```
for (( i=0; i>=0; i++))
do
    echo "iteration $i"
done
```

Here is a straightforward model that circles multiple times, printing the current circle number each time:

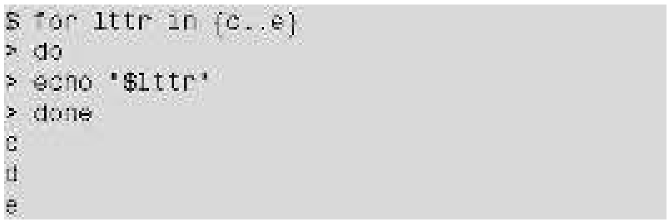
Note that this style of for circle utilizes twofold enclosure. This implies that every one of the administrators inside are number juggling administrators. In contrast to tests, where more noteworthy than or equivalent to is represented by
- ge, a for circle comprehends >= in its number-crunching setting. The equivalent goes for > and <, which are seen not as redirection (in [), or string arranging (in [[), however as more prominent than, and less-than.

For circles likewise work with recorded items,



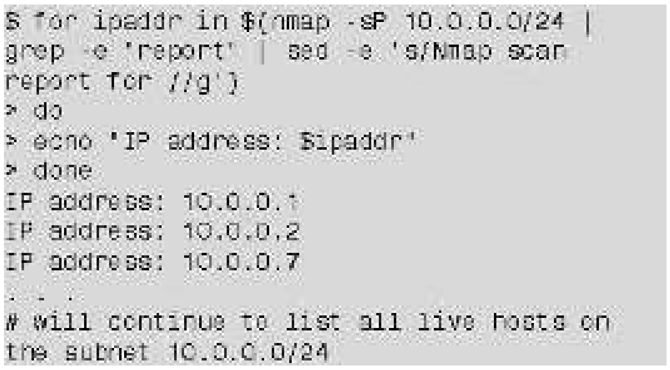
```
$ for num in 1 3 5 7
> do
> echo "$num"
> done
1
3
5
7
```

support expansions,



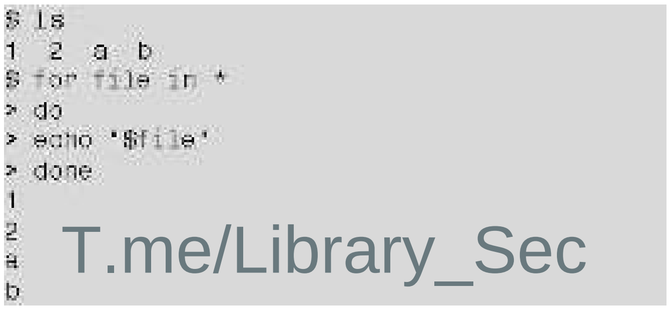
```
$ for lttr in {c..e}
> do
> echo "$lttr"
> done
c
d
e
```

subshell expansion,



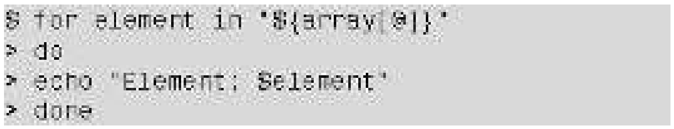
```
$ for ipaddr in $(nmap -sP 10.0.0.0/24 |
grep -e 'report' | sed -e 's/Nmap scan
report for //g')
> do
> echo "IP address: $ipaddr"
> done
IP address: 10.0.0.1
IP address: 10.0.0.2
IP address: 10.0.0.7
. . .
# will continue to list all live hosts on
the subnet 10.0.0.0/24
```

globbing,



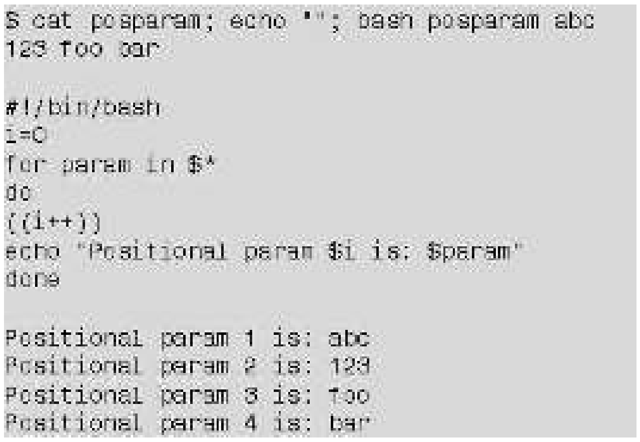
```
$ ls
1  2  a  b
$ for file in *
> do
> echo "$file"
> done
1
2
a
b
```

arrays,



```
$ for element in "${array[@]}"
> do
> echo "Element: $element"
> done
```

positional parameters,



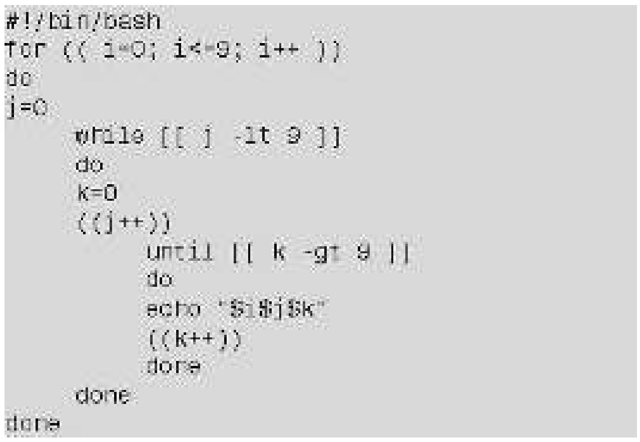
```
$ cat posparam; echo ""; bash posparam abc
123 foo bar

#!/bin/bash
i=0
for param in $*
do
((i++))
echo "Positional param $i is: $param"
done

Positional param 1 is: abc
Positional param 2 is: 123
Positional param 3 is: foo
Positional param 4 is: bar
```

and then some. As may be obvious, the conceivable outcomes are everything except endless.

Miscellaneous

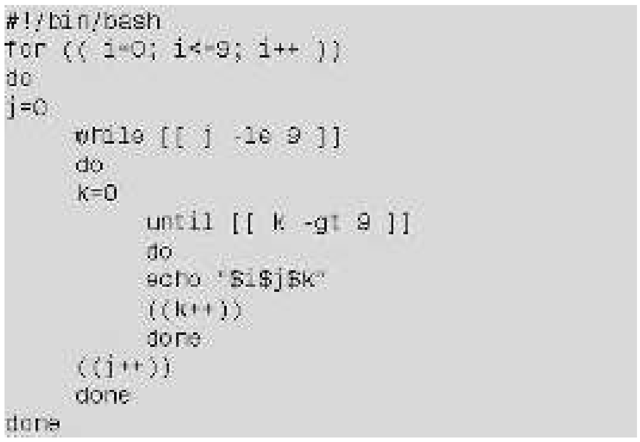
Nesting

Like restrictive articulations, everything circles can be settled, in any request, and utilizing any mix of while, until, and for loops.
This code utilizes each of the three circle types to print, all together, the numbers 000 through 999.



```
#!/bin/bash
for (( i=0; i<=9; i++ ))
do
j=0
    while [[ j -lt 9 ]]
    do
    k=0
    ((j++))
        until [[ k -gt 9 ]]
        do
        echo "$i$j$k"
        ((k++))
        done
    done
done
```

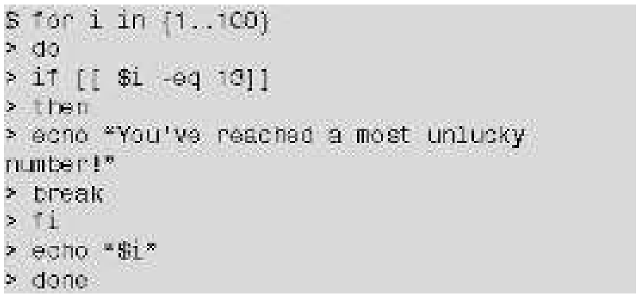
If you are asking why j accurately shows all digits 0 to 9 regardless of its test checking for a not as much as condition, the explanation is because of the position of the j incrementer. Since it is over the k circle, it augmentations to 9 preceding the k circle starts. Assuming the j incrementer were moved underneath the k circle, the contingent assertion would need to change to either
- gt or le.



```
#!/bin/bash
for (( i=0; i<=9; i++ ))
do
j=0
    while [[ j -le 9 ]]
    do
    k=0
        until [[ k -gt 9 ]]
        do
        echo "$i$j$k"
        ((k++))
        done
    ((j++))
    done
done
```

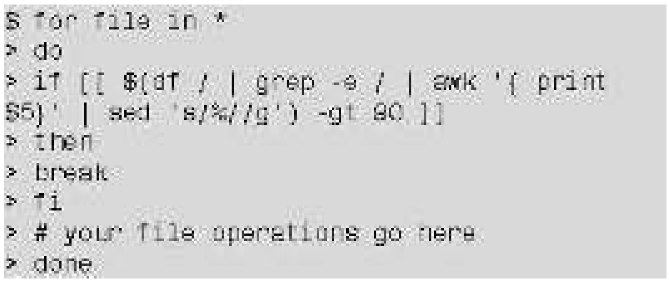
Break
All circles can be left by means of the break articulation, in which case the

content promptly following the circle is executed. This isn't to be mistaken for the leave explanation, which leaves the content (in addition to the circle) entirely.



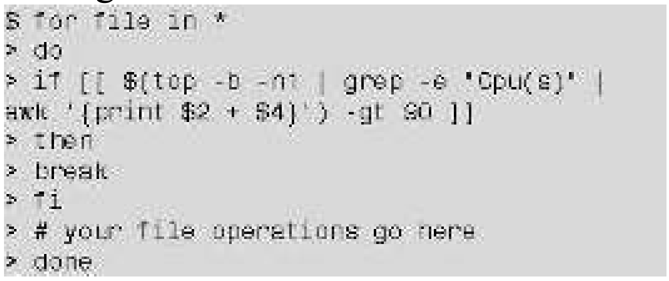
```
$ for i in {1..100}
> do
> if [[ $i -eq 13]]
> then
> echo "You've reached a most unlucky
number!"
> break
> fi
> echo "$i"
> done
```

Typically break is utilized to leap out of a circle when a "risky" condition is met. This could be just about as basic as attempting to adjust a document that isn't possessed by you,
halting assuming you come up short on plate space,



```
$ for file in *
> do
> if [[ $(df / | grep -e / | awk '{ print
$5}' | sed 's/%//g') -gt 90 ]]
> then
> break
> fi
> # your file operations go here
> done
```

or stopping your CPU-concentrated interaction assuming the processor's utilization turns out to be too high.

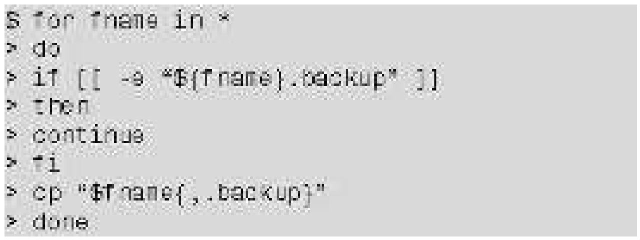


```
$ for file in *
> do
> if [[ $(top -b -n1 | grep -e 'Cpu(s)' |
awk '{print $2 + $4}') -gt 90 ]]
> then
> break
> fi
> # your file operations go here
> done
```

Continue

Usage of the proceed with proclamation permits the current emphasis of the circle to be skipped, and the close to be promptly started.
The accompanying model checks assuming any records in the current catalog contain a capitalized character. In the event that any do, the filename is changed over to all lowercase. Assuming no capitalized characters are available, the content has no work to do and the circle

proceeds to the following iteration.
Another model is to make reinforcement duplicates, everything being equal, skirting those which have effectively had reinforcements created.



```
$ for fname in *
> do
> if [[ -e “${fname}.backup” ]]
> then
> continue
> fi
> cp “$fname{,.backup}”
> done
```

Chapter 13: Input & Output

As you ’ve most likely seen, while prearranging in Bash there is ordinarily no single “right” method of getting things done. Rather, there are many right ways, with the determination exclusively dependent on close to home inclination. Information and result are no different.

Input is whatever is gotten or perused by a content. Input incorporates positional and exceptional boundaries, shell and climate factors, records, streams, lines, and that’s just the beginning.

Yield then again, is whatever is delivered or composed by a content. Shell and climate factors, records, streams, and lines are a couple examples.
Parameters
Positional and special parameters were covered earlier in the guide. The listing of these parameters is repeated here for convenience.
Parameter Description
$0 The called name of the script.
$1, $2, … Print the arguments passed to the script.

$#
The number of positional parameters passed to the script.
$*
All the positional parameters. If double quoted, results in a single *string* of them all.
Shell variables

Like boundaries, shell factors were shrouded before in the aide and are rehashed beneath for convenience.

Variable
$BASH
$BASH_VERSION $BASH_VERSINFO $EDITOR
$EUID
$HOME
$HOSTNAME
$HOSTTYPE
$IFS
$MACHTYPE
$OLDPWD
$OSTYPE
$PATH
$PIPESTATUS

$PPID
$PS1, …, $PS4
$PWD

$RANDOM
Description
The path to Bash's binary.
Bash's version.
Bash's major version.
The environment's default editor. The current user's effective user ID. The current user's home directory path. The machine's hostname.
The machine's hosttype.
The IFS setting. Blank if set as default. The machine type.
The previous directory you were in. The operating system type.
The command search path.
The exist status of the last completed pipe. The process ID of the current shell's

parent process.
The shell's (prompts 1 through 4) format. The current working directory.
A pseudo random integer between 0

2^15.
$SECONDS
How many seconds the script has been executing.
$UID The current user's ID.

Environment variables

Like shell factors, climate factors comprise of a name and worth matching. Nonetheless, dissimilar to shell factors, which are nearby and substantial in the current terminal example just, climate factors are framework wide.

The accompanying table contains a dense posting of ordinarily utilized climate variables. **Variable**
BASH_VERSION BROWSER

DISPLAY

EDITOR
HOSTNAME HISTFILE

HOME
IFS
LANG

LOGNAME MAIL
MANPATH PAGER

PATH

PS1
PWD
SHELL
TERM
TZ
Description
Bash's version number.

The system’s default web browser.
The hostname, display and screen number for

graphical application display.

The default file editor. VISUAL falls back to this if it fails.
The system’s name.
The path to the file were command history is saved.
The current user’s home directory.
The **I** nternal **F** ield **S** eparator’s global setting.
The language settings used by system applications.
The currently logged in user.
The storage location for system mail.
The storage location for system manpages.
The utility used to display long text output.
The colon separated search path for commands.
The prompt settings.
The current working directory.
The path to the current user’s shell program.
The current terminal emulation type.
The timezone used by the system clock. **USER** The currently logged in user.
VISUAL The default GUI file editor.
To get a full posting of the climate factors set on your framework, which will without a doubt have a lot more than recorded above, utilize the printenv command.

```
$ printenv
```

Though climate factors are a framework organization point, a fast introduction on climate factors follows. For additional data, counsel a fitting guide.

To show the at present set incentive for a specific climate variable, pass the variable to printenv as a boundary or utilize the standard dollar-sign boundary expansion.

```
$ printenv TERM
$ echo $TERM
```

To add or change climate factors, trade is utilized. Evacuation is cultivated by means of unset.

```
$ unset SHELL          # completely erases
the SHELL environment variable
$ export SHELL=/bin/bash # sets the SHELL
environment variable as /bin/bash
```

There are a couple significant things to note when setting and unsetting climate variables.
First, you can’t change the climate of a running system. Changing climate factors just influences its to-be-generated kid processes. A covering script is an ordinarily utilized piece of code that changes the climate here and there then executes another program.
Second, all progressions to the climate are transitory except if they are composed, regularly to one of the accompanying locations:

```
~/.pam_environment
~/.profile
~/.bashrc
/etc/environment
```

Because you can ’t change a running climate, you should relogin for your progressions to produce results. On the other hand, you can drive your framework to rehash the altered record (and hence change recently brought forth processes) with source.
Using executive is another choice that will supplant the current terminal example with a recently generated instance.

```
$ source /etc/environment
$ exec bash
```

Standard streams

The three standard streams, stdin, stdout, and stderr, were canvassed in the start of this aide with a guarantee to elucidate upon them later. That opportunity has arrived as record descriptors.

File descriptors

File descriptors (FDs), are handles (theoretical pointers) that are utilized to get to an information or result document, stream, pipe, attachment, gadget, network interface, thus on.

At a significant level, document descriptors work by giving a layer of deliberation between a genuine equipment gadget, like a console, screen, or cdrom drive, and an exceptional document made by the piece for the gadget, populated by udev, and put away in the/dev index. At the point when a call is made by a cycle to peruse or compose, record descriptors give a way to those calls to be steered to the right place.

The significant focuses to recollect are summed up in the accompanying table:
Name

Standard input
Standard output
Standard error

Stream
Default
File Descriptor stdin Keyboard 0
stdout Screen 1 stderr Screen 2

File redirection

A typical and fundamental type of redirection is document redirection. Typically, a program sends its result to stdout (FD 1). With document redirection, the FD for stdout is highlighted a record, rather than a screen (the default). Yield redirection is cultivated by utilizing a more noteworthy than sign, >.

Normally, reverberation shows a line of text on stdout. In any case, we can divert its result to a document as displayed here:

```
$ echo "This writes to a file." > myfile
```

When the above order is run, reverberation no longer shows its text onscreen in light of the fact that its FD is currently highlighting a record called myfile.

It is basic to take note of that > opens the document and keeps in touch with it without care whether it recently existed or contained anything. There was no such thing as assuming that the record before the order was run, it is made and written to. Assuming that the document existed, anything it contained is completely overwritten.

In request to abstain from overwriting a current record 's substance, Bash contains a multiplied adaptation of the result redirection administrator, >>. Its motivation is to attach information to the furthest limit of a current file.

```
$ echo "Line one." > myfile
$ echo "Still line one." > myfile
$ echo "Line one, yet again." > myfile
$ echo "Here is line two!" >> myfile
```

Input redirection is refined by utilizing <.

In request to pass a document, myfile, to a program as info and afterward show its substance on stdout, we utilize the utility cat.
From the feline manpage:

```
NAME
       cat - concatenate files and print on
the standard output

SYNOPSIS
       cat [OPTION]... [FILE]...

DESCRIPTION
       Concatenate FILE(s), or standard
input, to standard output.
```

Version 1:

```
$ cat
hi
hi
wait, what?
wait, what?
```

Version 2:

```
$ cat myfile
Line one, yet again.
Here is line two!
```

Version 3:

```
$ cat < myfile
Line one, yet again.
Here is line two!
```

Though the principal form seems not the same as the last two, which appear to be indistinguishable, they all work distinctively behind the scenes.
In the first version, *cat* is started, and begins to read from *stdin* . It then takes your input and displays it on *stdout* . The reason your text is displayed twice is because the same reason commands don't execute until you hit [ENTER]; your terminal is giving you the chance to edit (via backspace, delete, arrow keys, etc.) what you have typed. Once you hit [ENTER] however, your text becomes *stdin* for *cat* , which takes and (re)displays it on *stdout* . [CTRL][D] is the keyboard shortcut for the *End of File* (EOF) character, which makes *cat* think that *stdin* has stopped.
In the subsequent strategy, *myfile is passed as a contention to feline* . Very much like you need to choose how to manage, and how to deal with, positional boundaries in your content, *feline should do similarly—it has unlimited authority over what to do and how to approach doing it* . Since the purpose of *cat* is to print to *stdout* , it opens the file, then reads and prints its contents to *stdout* .
In the third strategy, *feline gets no contribution from your console* , nor from any contentions. All things being equal, a record descriptor is opened to the document, *myfile* , which makes the substance of the record accessible for feline as stdin. Consequently, when feline peruses from stdin, it is currently perusing from a document, and not your console. This implies that however the result of adaptation 3 is indistinguishable from that of rendition 2, its strategy for activity is really like variant 1!
The multiplied form of the info redirection administrator, <<, is known as a here
report, and is covered somewhat further on.

File descriptor redirection

In the past segment, we utilized "stripped" redirection administrators. That is, we utilized > and < without anyone else, without indicating a specific record descriptor. This worked for document redirection on the grounds that the default activities were by and large the thing we were searching for. You can be that as it may, physically indicate which record descriptor you wish to divert. This is cultivated by adding the mathematical prefix of the standard stream you wish to redirect.

Redirecting the stdout of order to document would resemble this (the two varieties work precisely the same):

```
$ command > file
$ command 1> file
```

Similarly, diverting record to the stdin of order would resemble this (once more, the two lines work precisely the same):

```
$ command < file
$ command 0< file
```

Why is adding the stream number valuable when the "exposed" redirector turns out totally great, you inquire? Since there is another record descriptor, stderr! Since the defaults don't indicate stderr, we really want to determine them manually.

To show this present, we should make a vacant registry, change to it, and attempt to eliminate a nonexistent file.

```
$ mkdir emptydir
$ cd emptydir
$ ls
$ rm myfile
rm: cannot remove 'myfile': No such file or
directory
```

As expected, rm created a mistake message. Contingent upon what you are doing, this may or not be useful. You could basically divert them to/dev/invalid, which is a unique filesystem object that disposes of everything written to it.

```
$ rm myfile 2>/dev/null
```

That disposed of the blunder message. However, essentially discarding things you would rather not see isn't dependably the best arrangement! Maybe a superior way is divert the message to a document you can counsel as needed.

```
$ rm myfile 2> err_msgs
$ cat err_msgs
rm: cannot remove 'myfile': No such file or
directory
```

While better, it is far from ideal when running a script and a few dozen (hundred? thousand?) errors or warnings flash by—they would each overwrite the last! The double redirection operator solves that problem by appending the error file with each new message.

```
$ rm myfile{,1,2,3} 2>> err_msgs
$ cat err_msgs
rm: cannot remove ‘myfile’: No such file or
directory
rm: cannot remove ‘myfile1’: No such file or
directory
rm: cannot remove ‘myfile2’: No such file or
directory
rm: cannot remove ‘myfile3’: No such file or
directory
```

That is much better.
If you can divert one document descriptor, why not two? As in send both

stdout and stderr to a document? The appropriate response is that you can do this, and regularly will, yet it doesn’t exactly work the manner in which you think it would.

First, how about we concoct an order to utilize. An ordinary time you would need to divert the two streams is the point at which you are ordering a program via,

```
$ ./configure && make && make install
```

For our demonstration however, we are looking for something that is easily reproducible and does not actually require modifying our system (i.e. installing a program).

```
$ find / -iname *config*
```

That will work pleasantly. Linux frameworks store their setups in config documents, not which are all open to a non-root client, in this manner producing many lines of both “typical” yield just as “mistake” yield. The following is a little piece of the result from this find command.

```
/var/cache/debconf/config.dat
/var/cache/pppconfig
/var/cache/ldconfig
find: `/var/cache/ldconfig': Permission
denied
find: `/var/spool/rsyslog': Permission
denied
find: `/var/spool/cups': Permission denied
```

The standard perspective to divert these results is guide them toward a document as you are utilized to doing.

```
$ find / -iname *config* 1> find.log 2>
find.log
```

The issue is that with both the stdout and stderr record descriptors highlighting a similar record, find.*log, results can be bludgeoned* . This implies that the result of one redirection will overwrite the result of the other.

A more straightforward method for seeing this in real life is to make our own little testbed. Here we make two records and populate them with a solitary line of information. We then change the owner and permissions of one of the files so that we no longer perform any operations on it (thus generating an error when we attempt a non-root operation on it).

```
$ mkdir clobber && cd clobber
$ date > datefile
$ cat /etc/timezone > tzfile
$ cat *
Mon Mar 21 12:48:54 PDT 2016
America/Los_Angeles
$ chown root:root tzfile
$ chmod 600 tzfile
$ cat *
Mon Mar 21 12:48:54 PDT 2016
cat: tzfile: Permission denied
```

Once the testbed is arrangement, we attempt to divert both result record descriptors.

```
$ cat * 1> cat.log 2> cat.log
$ cat cat.log
cat: tzfile: Permission denied
```

Not exactly what was generally anticipated, right? The issue is that the result from our stderr stream bludgeoned and overwrote that of our stdout stream. This was
in light of the fact that both document descriptors were highlighting a similar objective. Note that the document descriptors are perused from left to right, and that a similar issue will occur backward. That is, two record

descriptors highlighting a similar info source will forever pummel one another.

The right method for playing out this errand is to divert one document descriptor to one more by pointing our subsequent one (stderr) to our initial one (stdout). This is cultivated by utilizing the dollar sign administrator, and with our record descriptor redirection.

```
>&
```

Now, rather than opening two separate document descriptors, just one is opened for stdout (FD 1), and is consequently copied and set in stderr (FD 2).

```
2>&1
```

Trying the order in our proving ground yields,

```
$ cat * > cat.log 2>&1
$ cat cat.log
Mon Mar 21 12:48:54 PDT 2016
cat: tzfile: Permission denied
```

Exactly what we wanted! We would now be able to get back to our unique observe order, guaranteed that the substance of our log record will be correct.

```
$ find / -iname *config* > find.log 2>&1
```

It is vital to repeat that the record descriptors are perused from left to right. Thus attempting to copy FD 1 and spot it in FD 2 preceding FD 1 is diverted will
not work correctly .

```
$ find / -iname *config* 2>&1 > find.log
```

Here document

One principle in writing computer programs is that your code and information ought not consume a similar space. That is, info and result should come from documents that are being perused or written to. Consider it thusly: opening a content and altering squares of information each time you need to utilize an alternate arrangement of information is unwieldy, chaotic, and takes into account the incidental presentation of mistakes. The proviso is that utilizing records when the sum total of what you have are a couple of words or a line or two is over the top excess, also a problem. Enter here documents.

Here records are ideal for inserting modest quantities of information into a script.

```
command << delimiter
data
delimiter
```

What happens is the shell peruses the << administrator and comprehends it is to take the accompanying message/characters as contribution up to the predefined delimiter, which closes the info steam and sends it to stdin of the gave command.

```
$ wc -c <<EOF
> This is a short sentence.
> EOF
26
```

Note that the delimiter can’t contain whitespace. *EOF is usually utilized, and represents End Of File* .

```
$ fname=heredoc
$ cat > $fname <<iamtheend
> All this input will go via stdin
> to the command cat to be written to $fname
> iamtheend
$ cat $fname
All this input will go via stdin
to the command cat to be written to heredoc
```

Two significant things happened previously. To start with, did you see that you can remember other sidetracks for a similar line at here archives? That is on the grounds that here reports are diverts like some other. Second, did you see that here reports permit boundary replacement? You can forestall that by citing your delimiter.

```
$ cat > $fname <<'iamtheend'
> All this input will go via stdin
> to the command cat to be written to $fname
> iamtheend
$ cat $fname
All this input will go via stdin
to the command cat to be written to $fname
```

One last subtlety with here reports is that they protect tabs, and whenever utilized in a content, the delimiter should be toward the start of the line.

```
#!/bin/bash
echo "Testing here documents."
for (( i=0; i<1; i++ ))
do
     cat << EOF
          My nicely formatted block
          of data goes right here!
EOF
done
```

Its result will look like this:

```
Testing here documents.
          My nicely formatted block
          of data goes right here!
```

You can advise here archives to disregard tabs by adding a scramble,
- , to its operator.

```
#!/bin/bash
echo "Testing here documents."
for (( i=0; i<1; i++ ))
do
     cat <<- EOF
          My nicely formatted block
          of data goes right here!
          EOF
done
Testing here documents.
My nicely formatted block
of data goes right here!
```

Notice how it chipped away at both the delimiter and the information? It is a small

change however it makes a major difference.
By far most of the time here reports are utilized to dump basic use documentation to prearrange/program's user.

```
info() {
     cat <<-EOF
          usage: script [-v] [FILE]
          Script does THINGS to FILE
          -v turns on verbosity
          EOF
}
```

Here string

Here strings are basically the same as here archives in that they read input data and send it to an order. Not at all like here reports nonetheless, they just work on a solitary string that promptly continues the here string operator, <<<.

```
command <<< "string"
$ wc -c <<< 'I am a simple sentence.'
24
```

Remember to statement your string! Assuming you don't, everything past the first whitespace turns into a contention to your command.

```
$ wc -c <<<I am a sentence
wc: am: No such file or directory
wc: a: No such file or directory
wc: sentence: No such file or directory
0 total
```

Like here records, boundary replacement happens in here strings. Additionally, here strings and other sidetracks can work on the equivalent line.

```
$ cat <<<'$SHELL is running $TERM on
$DISPLAY' > info
$ cat info
/bin/bash is running xterm on :0
```

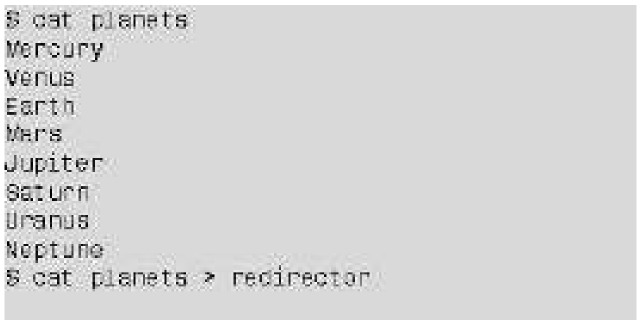
Because they are such a great deal less complex, and can just hold back a solitary string, here strings are utilized significantly more habitually than here documents.

FIFOs

Expanding on the information on record descriptor control to compose and peruse from documents, an extraordinary document called a FIFO is presented. FIFO represents First In, First Out, and dissimilar to a typical document, it stores no genuine information. Rather, it serves to divert information and result starting with one program then onto the next. FIFOs are made utilizing the mkfifo command.

```
$ mkfifo redirector
```

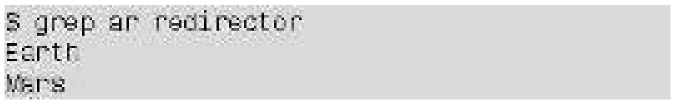
Since FIFO represents First In, First Out, we should keep in touch with certain information to it, and afterward play out a procedure on it at the other end.



```
$ cat planets
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
$ cat planets > redirector
```

Here I took a record containing the nine eight, planets, and diverted it to the FIFO, just to be locked out of my terminal! What happened was the FIFO hindered. Recollect how FIFOs don't store any information? They truly don't! Everything they do is give a passage—a named pipe—to carry it starting with one spot then onto the next. Until an order attempts to peruse from the FIFO, it will stay impeded forever.

Opening up one more terminal and perusing from the FIFO permits us to see this action.

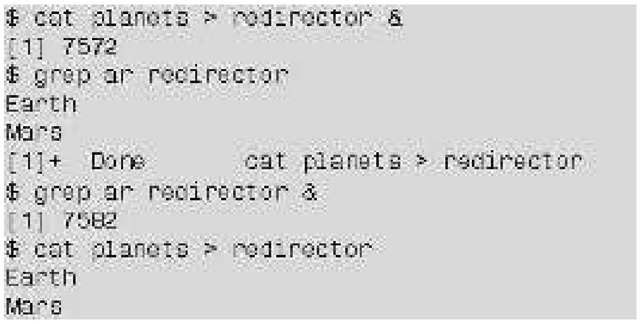


```
$ grep ar redirector
Earth
Mars
```

Did you see how the two orders finished simultaneously? That is on the grounds that the FIFO acknowledged it was being perused from so it unblocked the compose activity and transported the information to the read activity. Later it got done, it had no more work to do and closed.

If you would rather not utilize numerous terminals, you can in any case utilize FIFOs by essentially sending one of the cycles—it doesn't make any difference which—to the foundation by utilizing the dollar sign, and. Once behind the scenes, it blocks, delaying until the FIFO is gotten to by a comparing peruse or compose operation.

To illustrate, here are similar orders as above, acted in two distinct strategies: composing first and understanding first. Notice that the two of them work similarly well.



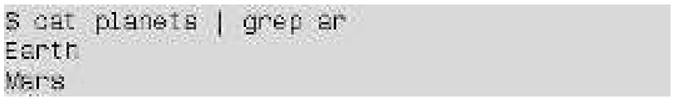
```
$ cat planets > redirector &
[1] 7572
$ grep ar redirector
Earth
Mars
[1]+  Done      cat planets > redirector
$ grep ar redirector &
[1] 7582
$ cat planets > redirector
Earth
Mars
```

Though they perform their task admirably, FIFOs are a pain to manage. If a FIFO is blocked by a read operation, it will remain blocked until it receives a write operation, or vice versa. You run the risk of having no tasks complete, and even then, potential task mismatch (perhaps because one failed due to a bug, bad syntax, etc., and thus skewed the ones following it). Additionally, FIFOs have file permissions and ownerships just like any other file, require creationand deletion, and a lot of typing to get them to work.

The considerably more easy to understand elective is a pipe.

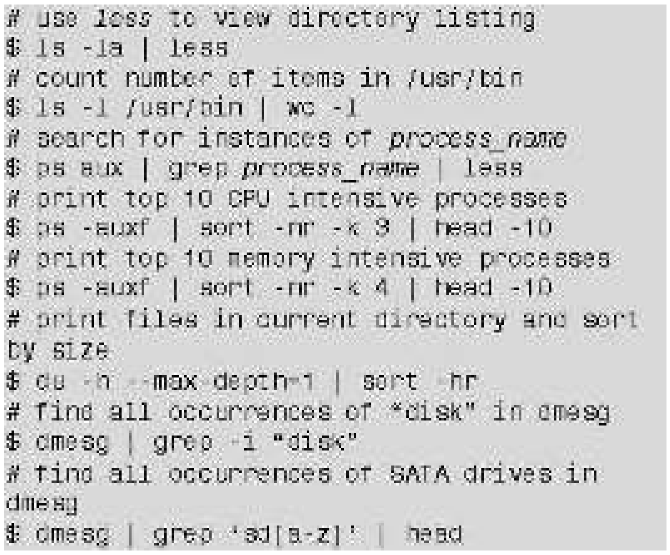
Pipes

Similar to FIFOs, pipes interface the stdout of one order to the stdin of another, and all without the issue of managing FIFOs. Pipes are made with an upward bar, |. Our equivalent planet looking through model from the past area is presently refined a lot more easily.



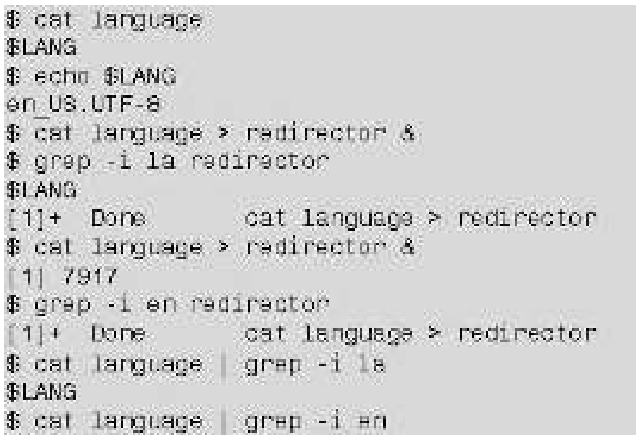
```
$ cat planets | grep ar
Earth
Mars
```

Here are a couple valuable examples:



```
# use less to view directory listing
$ ls -la | less
# count number of items in /usr/bin
$ ls -l /usr/bin | wc -l
# search for instances of process_name
$ ps aux | grep process_name | less
# print top 10 CPU intensive processes
$ ps -auxf | sort -nr -k 3 | head -10
# print top 10 memory intensive processes
$ ps -auxf | sort -nr -k 4 | head -10
# print files in current directory and sort
by size
$ du -h --max-depth=1 | sort -hr
# find all occurrences of "disk" in dmesg
$ dmesg | grep -i "disk"
# find all occurrences of SATA drives in
dmesg
$ dmesg | grep 'sd[a-z]' | head
```

Something last to note is that all FIFOs and lines do is transport information starting with one spot then onto the next. They don't follow up on it. Consequently, boundary replacement doesn't work.



```
$ cat language
$LANG
$ echo $LANG
en_US.UTF-8
$ cat language > redirector &
$ grep -i la redirector
$LANG
[1]+  Done       cat language > redirector
$ cat language > redirector &
[1] 7917
$ grep -i en redirector
[1]+  Done       cat language > redirector
$ cat language | grep -i la
$LANG
$ cat language | grep -i en
```

The main manner by which boundary extension works is in the event that you utilize a line and an inline bunch (recall those?). Note notwithstanding, that lines generate a kid shell each time called, so any boundary changes made inside the subshell have no impact on their parent.

```
$ home="123 Fake Street"
$ echo "I live at $home"
I live at 123 Fake Street
$ echo "321 Real Ave" | read home
$ echo "321 Real Ave " | { read home; echo
"I live at $home"; }
I live at 321 Real Ave
$ echo "I live at $home"
I live at 123 Fake Street
```

Process substitution

While hanging numerous channeled orders to each other straight unquestionably works, each new line makes it substantially more hard to follow and investigate. Enter process replacement. It basically permits you to pipe the stdout of numerous orders to another order. Input replacement is cultivated through the <() linguistic structure, while yield replacement is refined with >().

```
<()       # process input substitution
>()       # process output substitution
```

Way back in the aide, order bunches were covered. Orders inside an order bunch are executed in a subshell, where any factor tasks are temporary. A speedy model would be to change catalogs prior to running a solitary order and not have the remainder of your content affected.

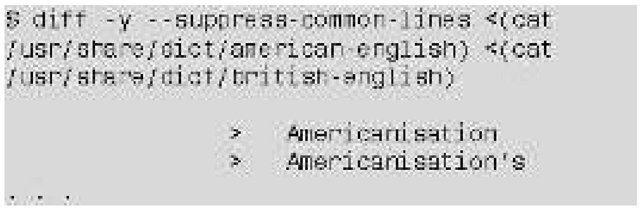
```
( cd /tmp || exit 1; stat $files )
```

Working a digit like FIFOs, and a bit like lines, process replacement utilizes Bash made and took care of named lines to divert information and result, saving you from dealing with the unique documents that accompany FIFOs.

```
$ wc -l <(cat /usr/share/dict/american-
english)
99171 /dev/fd/63
$ wc -l <(grep color
/usr/share/dict/american-english )
34 /dev/fd/63
```

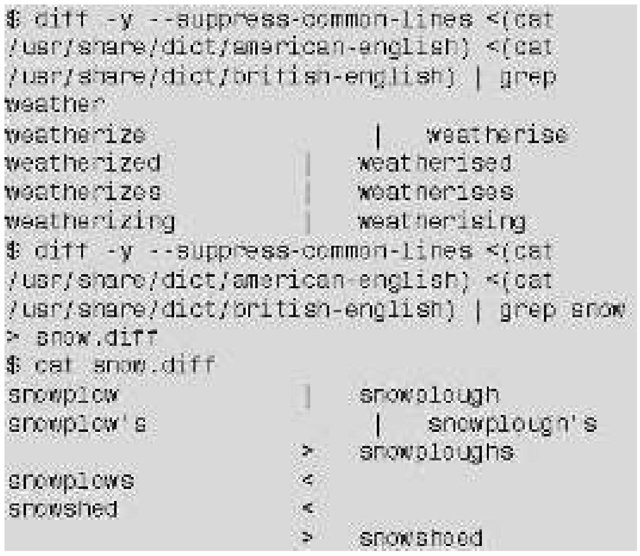
The main model prints the quantity of lines (which relates to words, for this situation) in the “American English” word reference. The second prints the quantity of lines (words) that contain the halfway word, “shading.” For each situation, the result is trailed by the way to the Bash-made document descriptor(s) utilized by the interaction replacement (for this situation, */dev/fd/63)* .

Process replacement isn 't restricted to one replacement. *diff requires two records as info, which is effortlessly cultivated in the accompanying model that thinks about the "American English" and "English" dictionaries* .



```
$ diff -y --suppress-common-lines <(cat
/usr/share/dict/american-english) <(cat
/usr/share/dict/british-english)

                  >   Americanisation
                  >   Americanisation's
. . .
```

Pipes and other redirections actually work like normal:



```
$ diff -y --suppress-common-lines <(cat
/usr/share/dict/american-english) <(cat
/usr/share/dict/british-english) | grep
weather
weatherize                  |   weatherise
weatherized             |   weatherised
weatherizes             |   weatherises
weatherizing            |   weatherising
$ diff -y --suppress-common-lines <(cat
/usr/share/dict/american-english) <(cat
/usr/share/dict/british-english) | grep snow
> snow.diff
$ cat snow.diff
snowplow                |   snowplough
snowplow's                  |   snowplough's
                        >   snowploughs
snowplows               <
snowshed                <
                        >   snowshoed
```

Output redirection works similarly. Apparently the most widely recognized use is with the utility tee, which copies stdout. Here the contents of */usr/bin* are listed, their output is sent to *stdout* , duplicated by *tee* , then*grep'ed* for *apt* and *dpkg* , which are both written to corresponding file lists.

```
$ ls /usr/bin | tee >(grep apt > aptlist)
>(grep dpkg > dpkglist)
$ cat aptlist
add-apt-repository
apt
apt-add-repository
apt-cache
. . .
$ cat dpkglist
dpkg
dpkg-architecture
dpkg-buildflags
dpkg-buildpackage
. . .
```

Another model is to tar an index (maybe containing an enormous number of little documents), send that tar file to another PC (venus.mydomain.com) by means of ssh where it is untarred in place.

```
$ tar -cf >(ssh venus.mydomain.com tar xf -)
.
```

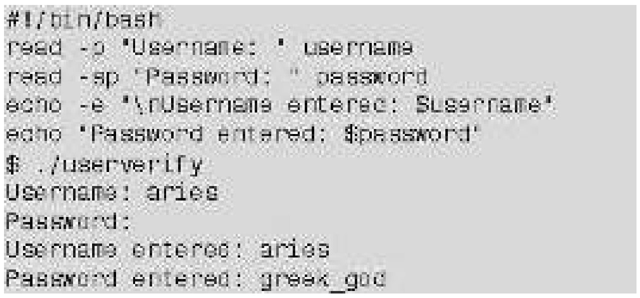
Read

Although not a document descriptor, redirector, line, or anything like that, the inherent read is a significant piece of perusing information from stdin, documents, or document descriptors into a variable.

There are a few standard utilizations. Getting client input is the first.

```
#!/bin/bash
echo 'Please enter your name.'
read name
echo 'Hi $name.'
$ ./readuser
Please enter your name.
Aries, the God of War
Hi Aries, the God of War.
```

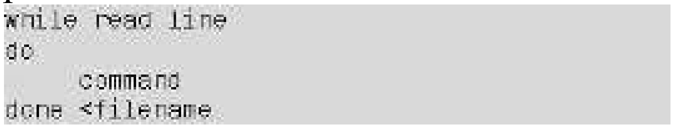
Two helpful changes to the read builtin are - p, which prompts the client for a passage, and - s, which stifles stdout for that section. It is basic to take note of that all factors are decoded, and therefor hazardous for any errand that contains in any way shape or form delicate material. Clearly you ought to never show a secret phrase as done in the beneath example.



```
#!/bin/bash
read -p "Username: " username
read -sp "Password: " password
echo -e "\rUsername entered: $username"
echo "Password entered: $password"
$ ./userverify
Username: aries
Password:
Username entered: aries
Password entered: greek_god
```

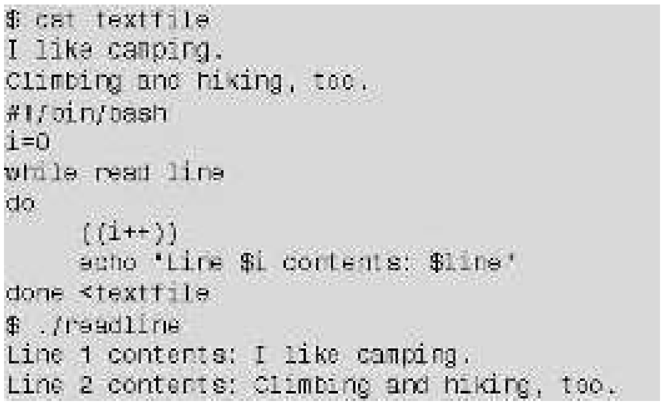
Since read takes in a string, it is feasible to part the string dependent on your IFS settings and store them in individual factors (or clusters, with the -an array_name option).

Read isn't restricted to getting client input . It can likewise be utilized to peruse in information from a file.



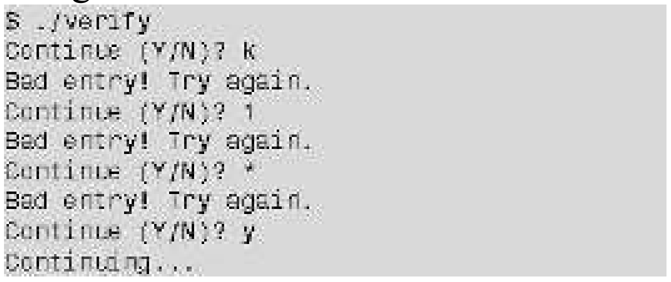
```
while read line
do
     command
done <filename
```

As an example:



```
$ cat textfile
I like camping.
Climbing and hiking, too.
#!/bin/bash
i=0
while read line
do
     ((i++))
     echo "Line $i contents: $line"
done <textfile
$ ./readline
Line 1 contents: I like camping.
Line 2 contents: Climbing and hiking, too.
```

The exemplary MS-DOS stop order can be reproduced utilizing read. Here is a model that confirms whether or not to continue.



```
$ ./verify
Continue (Y/N)? k
Bad entry! Try again.
Continue (Y/N)? 1
Bad entry! Try again.
Continue (Y/N)? *
Bad entry! Try again.
Continue (Y/N)? y
Continuing...
```

This content peruses in a way to a record, beginning with the client aries' home directory.

```
#!/bin/bash
read -e -p 'Where is the file? ' -i
'/home/aries/' fpath
echo "The file is at: $fpath"
$ bash path
Where is the file?
/home/aries/deity/greek/olympian/warplans
The file is at:
/home/aries/deity/greek/olympian/warplans
```

Chapter 14: Functions Functions

Functions are fundamentally scripts that dwell inside scripts. Since they live inside the running content itself, they don't have to produce extra cycles. Moreover, while youngster cycles and subshells can't adjust factors, capacities are totally fit for doing so.

Functions are characterized by name.

```
function function_name() {
commands
}
```

The capacity identifier is discretionary. Excluding it is common.

```
function_name() {
commands
}
```

The most fundamental capacity is one whose sole reason for existing is to illuminate you it is such.

```
fun_name() {
echo "I'm a function"
}
```

However assuming you place the above work in a content, nothing will happen.
That is on the grounds that capacities should be called by name
to execute.

```
#!/bin/bash
fun_name() {
echo "I'm a function"
}
fun_name
```

The result of the above content would look like this:

```
$ ./functiontest
I'm a function
```

It is critical to comprehend that Bash peruses start to finish. Assuming it goes over a call to a capacity that has not yet been characterized, the outcome will be a mistake message. Attempt it and see.

```
#!/bin/bash
fun_name
fun_name() {
echo "I'm a function"
}
$ ./functiontest
functiontest: line 2: fun_name: command not
found
```

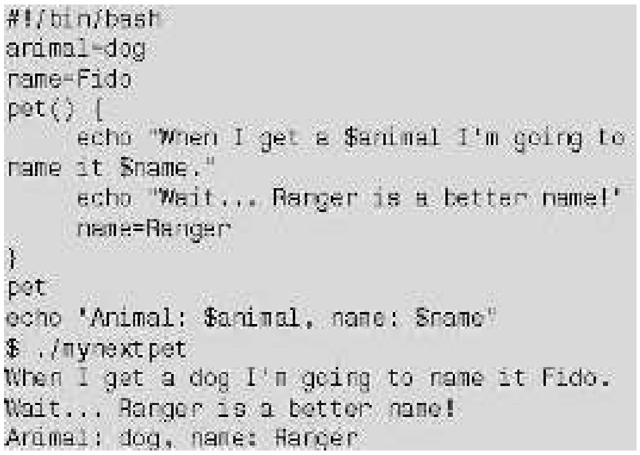
Be mindful that“placeholder” capacities will likewise bring about a mistake message.
Capacities should have substance and remarks don’t count.

```
#!/bin/bash
empty_fun() {
# there are no commands here
}
empty_fun
$ ./functiontest
functiontest: line 4: syntax error near
unexpected token `}'
functiontest: line 4: `}'
```

Though the capacity definition should go before its call, capacities can show up anyplace an order gathering could go. Moreover, when a capacity is characterized, its call is identical to a command.

```
#!/bin/bash
if [[ $UID -eq 0 ]]
then
    is_root() { echo 'Why hello, root!'; }
else
    exit
fi
is_root
$ sudo bash root_check
Why hello, root!
```

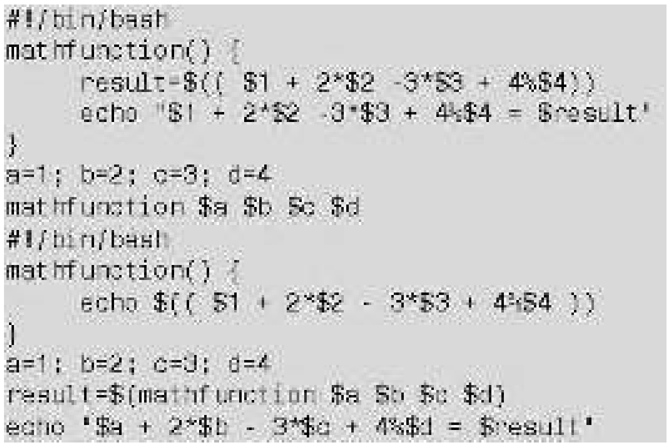
You could settle capacities, however I can ’t imagine a situation where that would be useful.
As referenced, capacities approach a content’s worldwide variables.



```
#!/bin/bash
animal=dog
name=Fido
pet() {
     echo "When I get a $animal I'm going to
name it $name."
     echo "Wait... Ranger is a better name!"
     name=Ranger
}
pet
echo "Animal: $animal, name: $name"
$ ./mynextpet
When I get a dog I'm going to name it Fido.
Wait... Ranger is a better name!
Animal: dog, name: Ranger
```

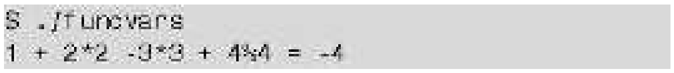
Local factors, then again, are just legitimate in the capacity, and won ’t overwrite factors of a similar name in the guest’s namespace. It is great practice not to have factors of a similar name in a content, however circle instated counters like I, *j,* and k are a typical exception.

Passing factors to and from capacities is very valuable. Factors are passed to a capacity by attaching the capacity call with the factors to be passed. Know that capacities work very much like contents, which implies that they take and record positional boundaries beginning from $1.



```
#!/bin/bash
mathfunction() {
     result=$(( $1 + 2*$2 -3*$3 + 4%$4))
     echo "$1 + 2*$2 -3*$3 + 4%$4 = $result"
}
a=1; b=2; c=3; d=4
mathfunction $a $b $c $d
#!/bin/bash
mathfunction() {
     echo $(( $1 + 2*$2 - 3*$3 + 4%$4 ))
}
a=1; b=2; c=3; d=4
result=$(mathfunction $a $b $c $d)
echo "$a + 2*$b - 3*$c + 4%$d = $result"
```

Because work calls are orders, both of the contents above would deliver the very same result shown below.



```
$ ./funcvars
1 + 2*2 -3*3 + 4%4 = -4
```

Note that capacities don’t approach positional boundaries passed to the content at run time except if expressly elapsed during the capacity call.

Finally, not at all like numerous other programming dialects, capacities in Bash don't uphold bring values back. In Bash, they essentially return a leave code (which can give an impromptu return worth would it be advisable for you so desire).

```
#!/bin/bash
returnfunc() {
     echo "Hi."
     return 13
}
returnfunc
echo "$?"
$ ./returnfunc
Hi.
13
```

Here are a couple (ideally) helpful functions.
No contention provided actually look at utilizing number of parameters:

```
arguments() {
if [[ $# -eq 0 || -z "$1"]]
then
     echo 'No arguments supplied'
fi
}
```

Check if user is running the script as root (i.e. checking their *euid* or *uid* to see if it is *0*):

```
rootcheck() {
if [[ "$EUID" -ne 0 || $UID -ne 0]]
then
     echo 'Please run as root'
fi
}
```

Check assuming every one of the characters in a string are alphanumeric: Check on the off chance that every one of the characters in a string are alphanumeric (this technique works by eliminating every single alphanumeric person and checking whether anything is left):

```
alphanumeric() {
case $string in
     *[^[:alnum:]]*) echo "Non \
alphanumeric character(s) present" ;;
                  *) echo "Only \
alphanumeric characters present" ;;
esac
}
```

Convert a string to all lowercase:

```
tolower() {
lower=$(echo "$@" | tr A-Z a-z)
}
```

Convert a string to all uppercase:

```
toupper() {
upper=$(echo "$@" | tr a-z A-Z)
}
```

Chapter 15: Traps Signals

Before we can get into traps, signals should be covered. Signals are a type of “between process correspondence.” What that implies is that the framework can send messages, “signals,” starting with one interaction then onto the next, not to move information, but rather to let the cycle know that some occasion has happened. Perhaps the most well-known signal is planned to the console alternate way [CTRL][C],and is one you’ve probably used to end buggy code, endless circles, etc. A couple of the more usually utilized signs are in the underneath table. The rest can be recorded by giving both of the commands,

```
$ kill -l
$ trap -l
```

Signals
Name
EXIT
SIGHUP

SIGINT
SIGQUIT

SIGKILL SIGTERM SIGCONT SIGSTOP

SIGTSTP Notes:

Value Effect Shortcut Trappable?
0 Exit
1 Hangup Yes

2
Interrupt
[CTRL]

[C] Yes
3 Quit[CTRL] Yes[\]

9 Kill No
15 Terminate Yes
18 Continue Yes
19 Stop No
20 Terminal [CTRL]YesStop [Z]

• A non-trappable sign can’t be gotten, hIndered, or disregarded. In different words, the bit kills the interaction, without allowing the cycle an opportunity to end gracefully.

• “EXIT” isn’t actually a sign. Rather, at whatever point a content ways out, for
any explanation, flagged or not, an EXIT trap is run.

Signals can be shipped off an interaction either by hitting the bound console easy route, or by utilizing the kill command.

```
$ kill -SIGNUM PID
```

To kill a cycle, you should begin with the least “hazardous” signal. That is, you should begin with the sign that, whenever upheld, permits the cycle to end itself nimbly and tidy up any brief documents, interaction, or streams it has open.

```
$ kill -2 PID
```

If that doesn’t work, attempt a more grounded signal, for example, one requesting that the program eliminate itself.

```
$ kill -15 PID
```

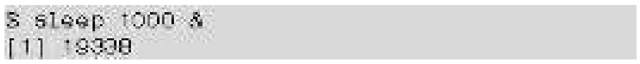
If that actually has no impact, continue on to a sign that kills the interaction through and through. Know that killing a cycle as such vagrants generally sub cycles (which are “embraced” by int, or PID 1), and leaves all open records, streams, or different things in an inadequate state.

```
$ kill -9 PID
```

Process management

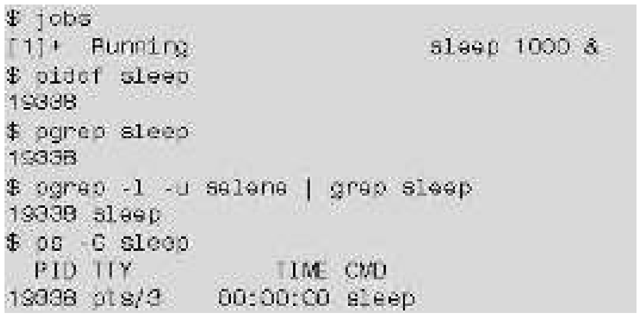
Process the executives is a frameworks organization theme and won ’t be shrouded top to bottom here. The reason it is mentioned at all is because you will need to know the basics of how to find a process ID, name, and kill said process when you start into the section on traps.

To observe processes, you can give the positions, pidof, pgrep, or ps commands.
First, we make a basic rest process and send it to the foundation to work on finding it.



```
$ sleep 1000 &
[1] 19338
```

Next, we observe it utilizing various distinctive methods.



```
$ jobs
[1]+  Running                 sleep 1000 &
$ pidof sleep
19338
$ pgrep sleep
19338
$ pgrep -l -u selene | grep sleep
19338 sleep
$ ps -C sleep
  PID TTY          TIME CMD
19338 pts/3    00:00:00 sleep
```

An intelligent option is top, and its brilliant partner, htop. Then, we kill the rest process.

```
$ kill -2 19338
$ pgrep sleep
[1]+  Interrupt                 sleep 1000
$ kill -9 19338
$ pgrep sleep
[1]+  Killed                    sleep 1000
$ kill -15 19435
$ pgrep sleep
[1]+  Terminated                sleep 1000
$ pkill sleep
[1]+  Terminated                sleep 1000
```

```
$ pkill -HUP sleep
[1]+  Hangup                    sleep 1000
$ killall sleep
[1]+  Terminated                sleep 1000
```

Traps

Now that you know what a sign is and how to oversee processes, we should continue on to flag controllers, or traps. Traps give a way to "get" signals. What you do after getting a sign is dependent upon you. You could disregard the sign (ordinarily an exceptionally poorly conceived notion which can prompt maximized CPU, plate, memory, process table cutoff points, and that's just the beginning). On the other hand, you could (and ought to) tidy up anything your content was sincerely busy (writing to records, shutting open documents, killing youngster processes, eliminating extraordinary as well as impermanent documents, FIFOs, the rundown continues) and permit it to be killed.

Just like in real life, a trap must be setup before it can be effective. This is refined by giving the Bash worked in, *trap* , trailed by the name of the signal(s) you wish to trap. Except if determined if not, a sign shipped off a cycle will have its default activity preformed. Along these lines, a non-caught SIGINT will kill a cycle. Recollect that not everything signs can be trapped.

```
trap COMMAND SIGNAL
```

Let 's beginning with a straightforward model. Here we make an endless circle that yields a period one time each second. Have a second terminal convenient before you start it and attempt to kill it utilizing the different techniques you learned in the past section.

```
#!/bin/bash
trap '' HUP INT QUIT KILL TERM
echo "My PID is $$"
while true
do
    echo -n '.'
    sleep 1
done
trap - HUP INT QUIT KILL TERM
```

There are various things to see in the model above. The first is that you expected to kill it by issuing,

```
$ kill -9 SCRIPT_PID
```

This is on the grounds that this line,

```
trap '' HUP INT QUIT KILL TERM
```

determines that the content is to trap the signs, HUP, INT, QUIT, KILL, and TERM, gotten while executing the code beneath the snare order, and complete

ignore them (the single quotation marks, ‘ ‘, i.e. run no command). This line could have also been written as:

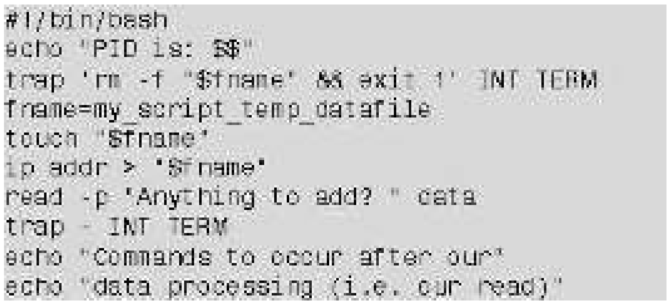
```
trap '' 1 2 3 9 15
```

Note that the endeavor to trap and overlook signal 9, SIGKILL, had no effect.
That is on the grounds that the sign is untrappable.
Finally, the last line in the script,

```
trap - HUP INT QUIT KILL TERM
```

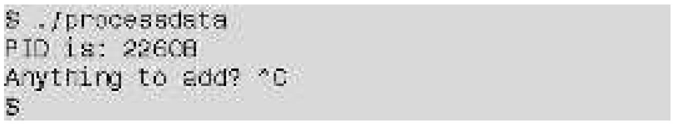
advises the content to return to its default treatment of the recorded signals. Returning to the default treatment of signs is critical to recollect! Assuming that excluded, all the code beneath the order where you determine the treatment of caught signs will react similarly whether you need them to or not.

Instead of overlooking signs, how about we catch and interaction them. In this case, we’ll create a temporary file, populate it with some data, then issue the command to *read* input from the user. In a genuine setting, *read would be appear as an order or series of orders that processes the information in the brief document* . (Since this is a learning script, as opposed to making a runaway interaction or a limitless circle that would impede at least one of your framework assets, *read is utilized instead* .)



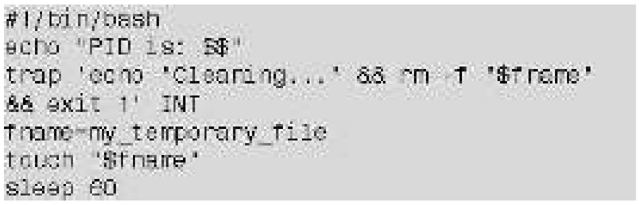
```
#!/bin/bash
echo "PID is: $$"
trap 'rm -f "$fname" && exit 1' INT TERM
fname=my_script_temp_datafile
touch "$fname"
ip addr > "$fname"
read -p "Anything to add? " data
trap - INT TERM
echo "Commands to occur after our"
echo "data processing (i.e. our read)"
```

Its result will look like this:



```
$ ./processdata
PID is: 22608
Anything to add? ^C
$
```

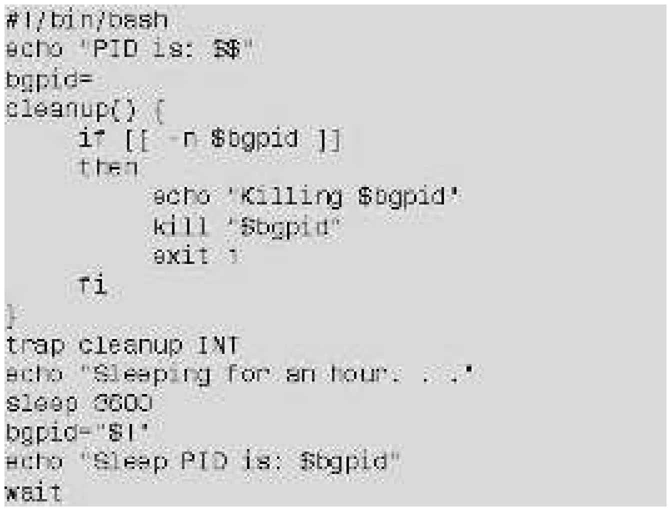
It is critical to comprehend the reason why we didn ’t just put the content to sleep.
That is on the grounds that signs are just dealt with once the current frontal area
process ends. This is a basic concept!
A model should make this more clear. Have a go at killing the content with the kill - 2
command.



```
#!/bin/bash
echo "PID is: $$"
trap 'echo "Clearing..." && rm -f "$fname"
&& exit 1' INT
fname=my_temporary_file
touch "$fname"
sleep 60
```

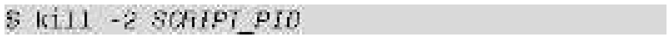
Did you perceive how the rest 60 order needed to finish before the snare executed?
Three significant things to specify here:
• First, the explanation the model two pages back didn’t appear to have this issue, regardless of having a rest order, is on the grounds that it just dozed briefly! Odds are you didn’t see that little a postponement between your kill order and when the snare kicked in.
• Second, trap - INT was excluded at the lower part of the script in light of the fact that there was no more content to execute, in this way no justifiable excuse to return the treatment of the SIGINT snare for the content back to

typical. Trap controllers are nearby to the content they are remembered for just; they are not framework wide.
• And third, [CTRL][C] and *kill, pkill, killall,* etc. workdifferently! While the order line kills convey the predefined message to the PID, the console SIGINT sends it to the cycle's gathering ID, which contains the rest command.
If you need your content to not need to sit tight for the closer view interaction to end prior to dealing with traps, the interaction should be put behind the scenes. An order to look out for the latest cycle shipped off the foundation is the manner by which this is cultivated (sit tight searches for an interaction status change, like an approaching sign). A snare would then be able to tidy up this foundation task prior to leaving—precisely what traps were intended to do!



```
#!/bin/bash
echo "PID is: $$"
bgpid=
cleanup() {
     if [[ -n $bgpid ]]
     then
          echo 'Killing $bgpid'
          kill "$bgpid"
          exit 1
     fi
}
trap cleanup INT
echo "Sleeping for an hour. . ."
sleep 3600
bgpid="$!"
echo "Sleep PID is: $bgpid"
wait
```

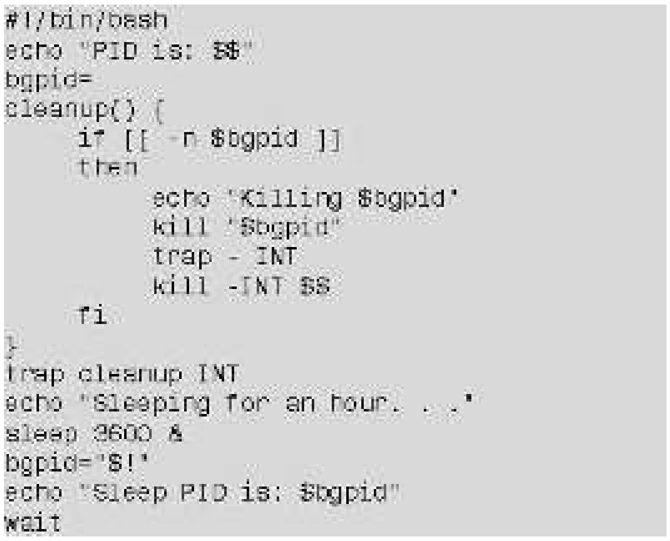
See how now when you issue the command,



```
$ kill -2 SCRIPT_PID
```

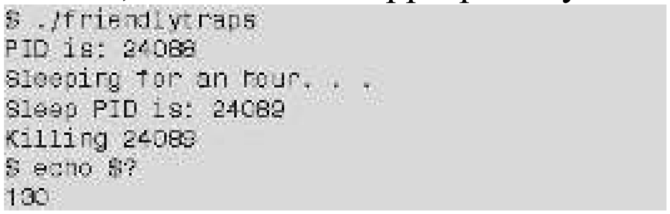
the content gets the sign, runs the snare work (review that capacity calls are orders, and consequently substantial snares!), and kills the foundation rest process before exiting?

One last note will help your projects play well with others ', particularly when managing circles. It is great practice to kill your content with a similar sign gave to it. Consequently as opposed to giving way out 1, what we ought to have been doing this entire time is resetting the snare overseers and killing the content with a similar sign we caught! This permits the guest to see that the content was killed, and didn't, say, exit on a blunder. Amending this misstep in the last model would yield the amicable code as displayed below.



```
#!/bin/bash
echo "PID is: $$"
bgpid=
cleanup() {
     if [[ -n $bgpid ]]
     then
          echo "Killing $bgpid"
          kill "$bgpid"
          trap - INT
          kill -INT $$
     fi
}
trap cleanup INT
echo "Sleeping for an hour. . ."
sleep 3600 &
bgpid="$!"
echo "Sleep PID is: $bgpid"
wait
```

Now when we run (and kill) the content, its leave code appropriately reflects how it terminated!



```
$ ./friendlytraps
PID is: 24088
Sleeping for an hour. . .
Sleep PID is: 24089
Killing 24089
$ echo $?
130
```

I 'll end this section with one very accommodating content for any Linux client that needs to cut down a help to play out an undertaking. Normally, you would have to notice that the script failed, and then manually restart it, right? What happens if you had to bring down the network to run a script while remoted in and your script did not successfully complete and bring it back up? Traps are your answer.

Example Scripts Common Programming Interview Scripts

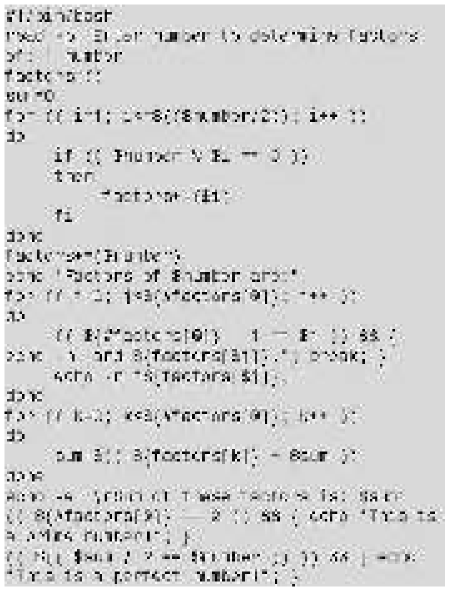
A typical "get rid of" inquiry in a programming meeting is to compose a content that plays out a certain, straightforward undertaking which shows information on the essential standards of programming. This part contains a few normal such programming inquiries questions. Except if noticed, all not really settled by means of an animal power search and are not upgraded for speed, proficiency, CPU use, etc.

Factor
Determine every one of the variables and different properties of a surrendered number.
First, read the number from the client. Then, using an iterative for loop,

determine the factors of the number and add them to an array. This is an animal power search, so by simply looking up to number/2, the program's run time is sliced down the middle (since there are no variables over this limit).

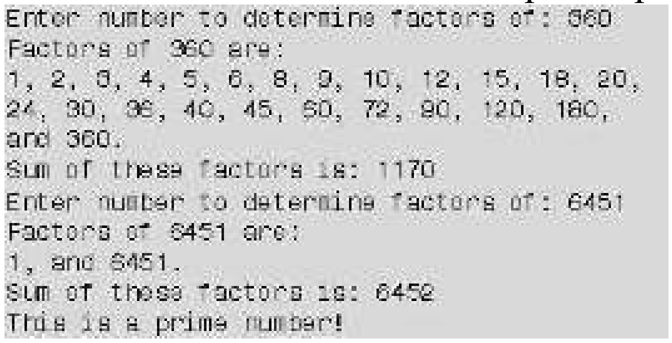
Next, print the properties of the number:

• Print the elements to the screen by going through the elements cluster one at a time.
• Calculate and show the amount of the factors.
• Check the quantity of elements to decide whether the number is prime.
• Check the amount of the elements to decide whether the number is perfect.

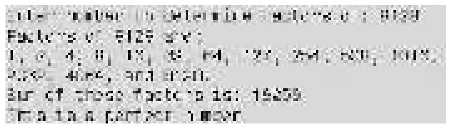


```
#!/bin/bash
read -p "Enter number to determine factors of: " number
factors=()
sum=0
for (( i=1; i<=$(($number/2)); i++ ))
do
    if (( $number % $i == 0 ))
    then
        factors+=($i)
    fi
done
factors+=($number)
echo "Factors of $number are:"
for (( j=0; j<${#factors[@]}; j++ ))
do
    (( ${#factors[@]} - 1 == $j )) && {
echo "and ${factors[$j]}."; break; }
    echo -n "${factors[$j]}, "
done
for (( k=0; k<${#factors[@]}; k++ ))
do
    sum=$(( ${factors[k]} + $sum ))
done
echo -e "\nSum of these factors is: $sum"
(( ${#factors[@]} == 2 )) && { echo "This is
a prime number!"; }
(( $(( $sum / 2 == $number )) )) && { echo
"This is a perfect number!"; }
```

Sample output:



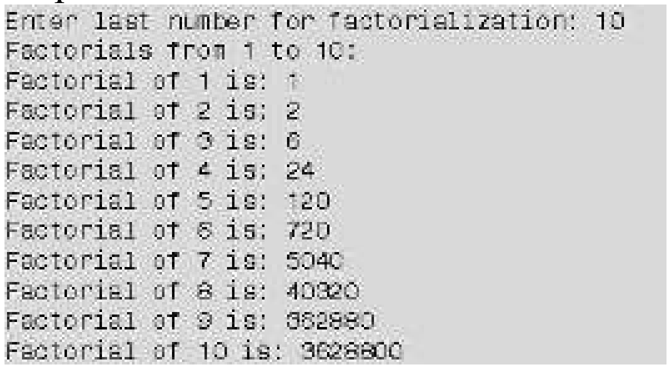
```
Enter number to determine factors of: 360
Factors of 360 are:
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 18, 20,
24, 30, 36, 40, 45, 60, 72, 90, 120, 180,
and 360.
Sum of these factors is: 1170
Enter number to determine factors of: 6451
Factors of 6451 are:
1, and 6451.
Sum of these factors is: 6452
This is a prime number!
```



```
Enter number to determine factors of: 8128
Factors of 8128 are:
1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016,
2032, 4064, and 8128.
Sum of these factors is: 16256
This is a perfect number!
```

Factorial

Calculate and show all factorials from 1 until a given number. Sample output:



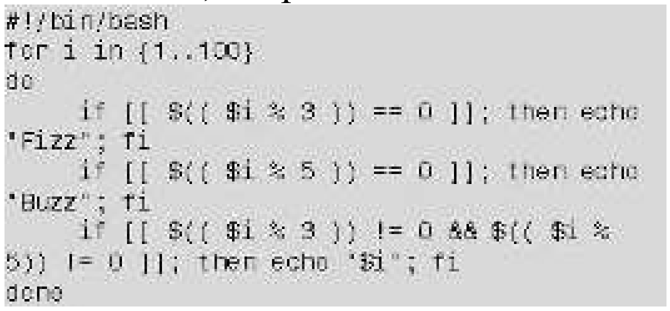
```
Enter last number for factorialization: 10
Factorials from 1 to 10:
Factorial of 1 is: 1
Factorial of 2 is: 2
Factorial of 3 is: 6
Factorial of 4 is: 24
Factorial of 5 is: 120
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
Factorial of 9 is: 362880
Factorial of 10 is: 3628800
```

Fizz Buzz

Arguable the most widely recognized inquiry question.
Given every one of the numbers 1 through 100:

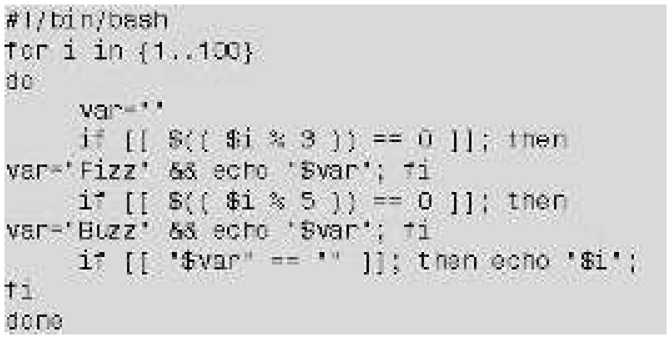
• If the number is a different of 3, print “Bubble” to the screen
• If the number is a various of 5, print “Buzz” to the screen
• If neither of the above is valid, print the number to the screen It is critical to understand that there are a couple of extraordinary cases:
• What happens when the number is a numerous of 3 and 5?
• What might occur assuming you remember 0 for the list?

Version 1: Conditional statement
Iterate from 1 to 100, then for each number, check for multiples using iftest statements. Note that the assertions can’t be settled, else for products of both 3 and 5, the proclamation will exit later the main articulation is true.



```
#!/bin/bash
for i in {1..100}
do
     if [[ $(( $i % 3 )) == 0 ]]; then echo
'Fizz'; fi
     if [[ $(( $i % 5 )) == 0 ]]; then echo
'Buzz'; fi
     if [[ $(( $i % 3 )) != 0 && $(( $i %
5)) != 0 ]]; then echo "$i"; fi
done
```

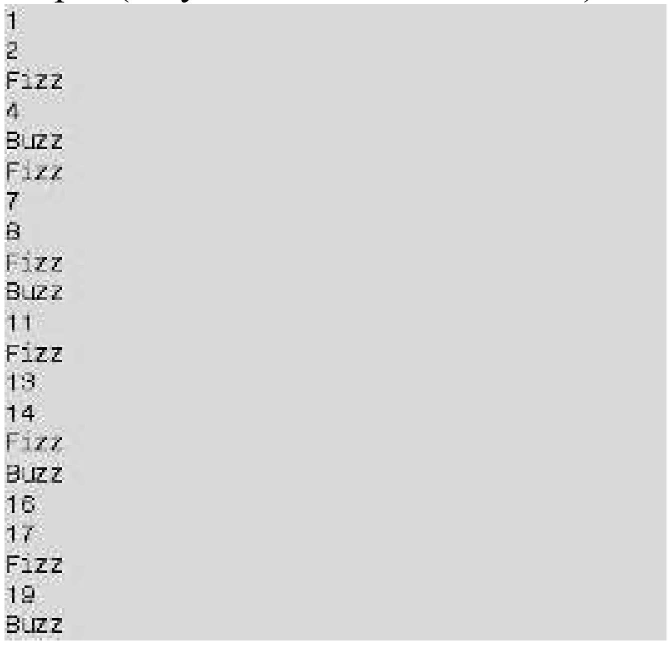
Version 2: Conditional explanation with overseer variable
In a substitute rendition, toward the beginning of the for-do-circle, you can utilize a “outsider” variable which takes out the requirement for the tremendous test in the third if statement.



```
#!/bin/bash
for i in {1..100}
do
     var=''
     if [[ $(( $i % 3 )) == 0 ]]; then
var='Fizz' && echo '$var'; fi
     if [[ $(( $i % 5 )) == 0 ]]; then
var='Buzz' && echo '$var'; fi
     if [[ "$var" == "" ]]; then echo '$i';
fi
done
```

Version 3: Case statement
Finally, utilizing a case articulation, which is monstrous since you need to accurately deal with when the number is a different of 3 and 5. Sample

output (only the first 20 lines shown):



```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
Fizz
Buzz
16
17
Fizz
19
Buzz
```

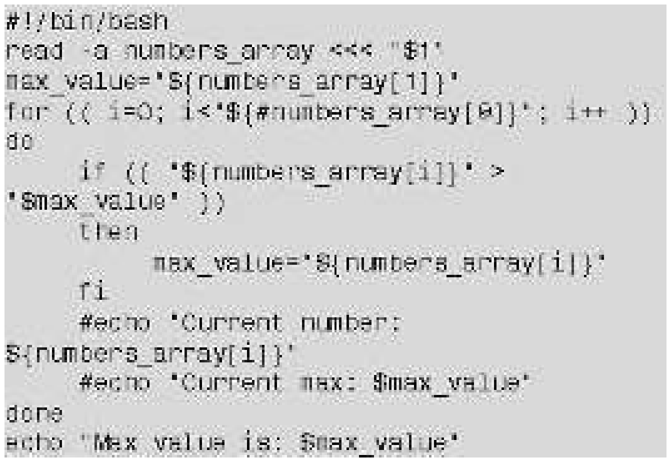
Maximum value
Given a client inputted series of number, find the greatest value. This inquiry is straightforward assuming you comprehend the idea driving

clusters (which you can see the fundamentals of in the “Invert a Sentence” post).

The essential thought is to peruse the client input into an exhibit with each number in the information isolated into its own component in the exhibit (the read - a numbers_array line).

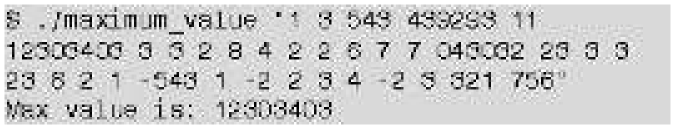
Once the numbers are in the exhibit, we need to repeat through every one of the components and check on the off chance that the current component is bigger than the current most extreme component. If it is, then we want to override the previous maximum with the new maximum. Note that it is vital that the underlying greatest worth is either not indicated or set as one of the components in the cluster toward the beginning. This forestalls mistakes assuming the client number rundown range doesn’t envelop the underlying most extreme value.

If you remark out the lines (#echo “Current number/max) close to the base, you can look as the content thought about the numbers and sets the greatest value.



```
#!/bin/bash
read -a numbers_array <<< "$1"
max_value="${numbers_array[1]}"
for (( i=0; i<"${#numbers_array[@]}"; i++ ))
do
     if (( "${numbers_array[i]}" >
"$max_value" ))
     then
          max_value="${numbers_array[i]}"
     fi
     #echo "Current number:
${numbers_array[i]}"
     #echo "Current max: $max_value"
done
echo "Max value is: $max_value"
```

Sample output:



```
$ ./maximum_value "1 3 543 499293 11
12303403 3 3 2 8 4 2 2 6 7 7 045002 23 3 3
23 6 2 1 -543 1 -2 2 3 4 -2 5 321 756"
Max value is: 12303403
```

Minimum value
Given a client inputted series of number, find the base value.
This inquiry is straightforward in the event that you comprehend the idea

driving clusters (which you can see the essentials of in the “Switch a Sentence” post).

The fundamental thought is to peruse the client input into an exhibit with each number in the information isolated into its own component in the exhibit (the read - a numbers_array line).

Once the numbers are in the exhibit, we need to emphasize through every one of the components and check on the off chance that the current component is more modest than the current least component. If it is, then we want to override the previous minimum with the new minimum. Note that it is vital that the underlying least worth is either not determined or set as one of the components in the cluster toward the beginning. This forestalls blunders in the event that the client number rundown range doesn’t incorporate the underlying least value.

If you remark out the lines (#echo “Current number/min) close to the base, you can look as the content analyzed the numbers and sets the base value.

```
#!/bin/bash
read -a numbers_array <<< "$1"
min_value="${numbers_array[1]}"
for (( i=0; i<"${#numbers_array[@]}"; i++ ))
do
     if (( "${numbers_array[i]}" <
"$min_value" ))
     then
          min_value="${numbers_array[i]}"
     fi
     #echo "Current number:
${numbers_array[i]}"
     #echo "Current min: $min_value"
done
echo "Min value is: $min_value"
```

Sample output:

```
$ ./minimum_value '1 3 543 439293 11
12303403 3 3 2 8 4 2 2 6 7 7 043032 23 3 3
23 6 2 1 -543 1 -2 2 3 4 -2 3 321 756'
Min value is: -543
```

Power towers

Calculate and show all numbers from a given base to its raised power.

```
#!/bin/bash
read -p "Enter base number: " base
read -p "Enter iterations: " iterations
number=$base
for (( i=0; i<$iterations; i++ ))
do
     echo "$number"
     number=$(($number*base))
done
```

Sample output:

```
Enter base number: 2
Enter iterations: 5
2
4
8
16
32
Enter base number: 6
Enter iterations: 5
6
36
216
1296
7776
```

Prime number generator

Generate a rundown of indivisible numbers. This isn’t a programming meeting question!

Sieve

I ’ll talk about how to make an indivisible number generator utilizing the Sieve of Eratosthenes.

The explanation I’m utilizing this specific strainer is a result of this line from the Wikipedia page: “The sifter of Eratosthenes is quite possibly the

most effective way to track down the more modest primes as a whole (under 10 million or somewhere in the vicinity).” Note that this script increases its pursuit at first, not utilizing last cycles of “checked off” numbers which makes it go through the rundown correlation a lot more occasions that necessary.
Again, replicated from the Wikipedia page, this is the way the strainer works: Create a rundown of sequential numbers from 2 to n: (2, 3, 4, ..., n).
Initially, let p equivalent 2, the primary prime number.
Starting from p, count up in increments of p and mark each of these numbers greater than p itself in the list. These will be multiples of p: 2p, 3p, 4p, etc.; note that some of them may have already been marked.
Find the main number more prominent than p in the rundown that isn’t checked. Assuming there was no such number, stop. In any case, let p currently equivalent this number (which is the following prime), and rehash from step 3.
To get everything rolling, lets separate the underlying strides from above into usable BASH code.

Step 1
Create a rundown of successive whole numbers from 2 to n:
(2, 3, 4, ..., n). Peruse the upper hunt limit, n, from the user.

```
read -p 'Enter upper limit of search: '
upper search limit
```

Then make the rundown of numbers utilizing seq (I use sort to guarantee later orders acknowledge the information correctly).

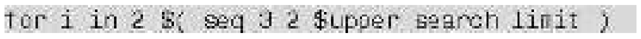
```
sieve="$( seq 2 $upper_search_limit | sort
)"
```

Step 2/3
Initially, let p equivalent 2, the principal indivisible number. Then, at that point, beginning at the number 2, include up in augmentations of 2 and imprint

those numbers off the list.

To do this, I began at 2, then, at that point, increased by 2 until the upper_search_limit was reached, making a rundown of every single odd

number while as yet including the number
2. I really didn't stamp them off the rundown yet, however will just actually look at these numbers to make new records later (hence trying not to make a rundown for every single much number, since no significantly number can be a prime).



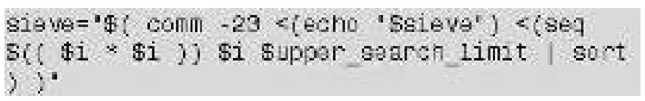
```
tor i in 2 $( seq 3 2 $upper search limit )
```

This is one of the significant wellsprings of shortcomings in my code, since it goes through and checks each odd number rather than just checking those that had
been separated the rundown from later iterations.

Step 4
Starting from p, count up in increments of p and mark each of these numbers greater than p itself in the list. These will be multiples of p: 2p, 3p, 4p, etc.; note that some of them may have already been marked.

Comparing the ebb and flow strainer (which at first contained all numbers) to the primary emphasis of the circle (what began at 2 and increased by 2, in this way containing a rundown of every single significantly number), and just holding the numbers novel to the first rundown, yields the new sifter displayed beneath (up to a hunt cutoff of 10, not separating through sort).

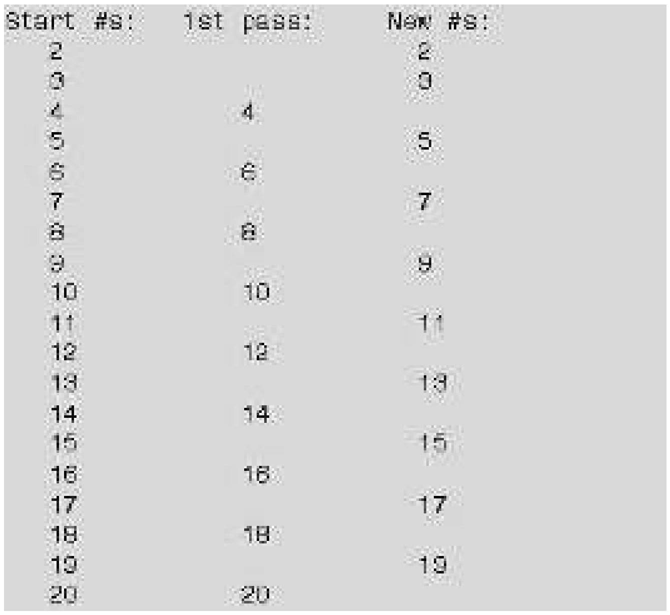
Here, the strainer is set (sieve= "$(..)") to the lines special to the numbers in the main rundown (the - 23 switches), where the principal list is the past cycle of the strainer (reverberation"$sieve"), and the subsequent rundown is the momentum cycle of the strainer, utilizing the odd number rundown created in sync 3 above (seq $(($i * $i)) $i
$upper_search_limit | sort)).



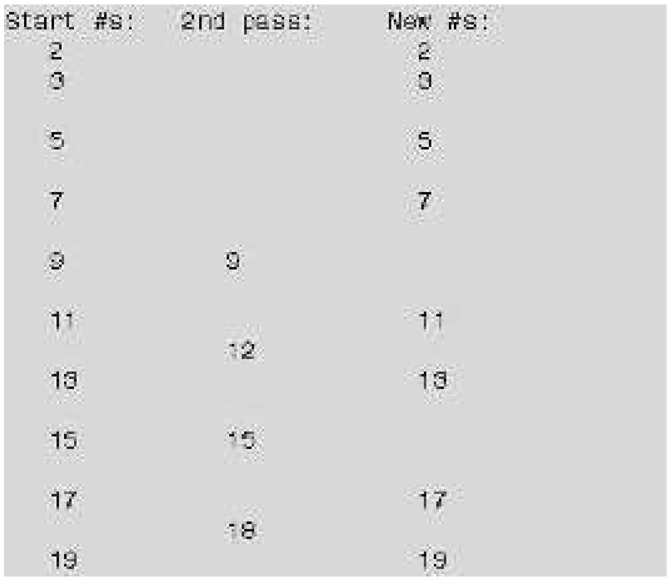
```
sieve="$( comm -23 <(echo '$sieve') <(seq
$(( $i * $i )) $i $upper_search_limit | sort
) )"
```

Remember that the main numbers checked off the rundown are the ones that beginning later the underlying "seed" number, henceforth 2 and 3 both stay in the passes underneath (and the justification for the square ($i * $i) in the code above — concentrate on the Wikipedia page, particularly the liveliness on the option to see that the principal number checked off for each new number is consistently the square of the “seed” number).

Also recall that I utilized sort to guarantee the rundowns were dealt with appropriately by comm.
What happens:



```
Start #s:   1st pass:     New #s:
   2                         2
   3                         3
   4            4
   5                         5
   6            6
   7                         7
   8            8
   9                         9
   10           10
   11                        11
   12           12
   13                        13
   14           14
   15                        15
   16           16
   17                        17
   18           18
   19                        19
   20           20
```

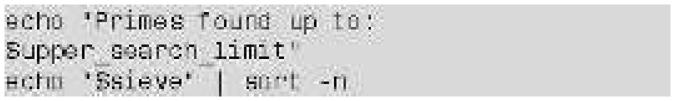


```
Start #s:   2nd pass:     New #s:
   2                         2
   3                         3

   5                         5

   7                         7

   9           9

   11                        11
               12
   13                        13

   15          15

   17                        17
               18
   19                        19
```

Note that while the code has observed all indivisible numbers from 2 to our upper inquiry limit in 2 passes, it actually will go through the circle 8 additional occasions for the “seed” numbers, 5, 7, 9, 11, 13, 15, 17, 19!

While this is awfully wasteful for such a little extension, guaranteeing, say, products of 5, 7 and 9 are checked off is fundamental for actually looking at primes up to 100, etc for higher pursuit limits.

Step 5
Display the results
This is self-explanatory.

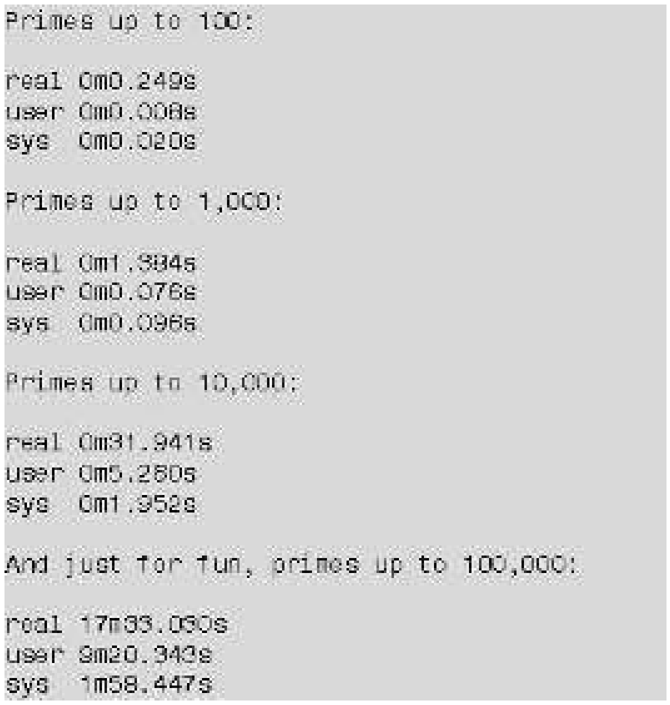


```
echo 'Primes found up to:
$upper_search_limit'
echo '$sieve' | sort -n
```

Script

Now for the last code itself.
Output
And, wallah!, every one of the primes up to as far as possible (100 in this case).

Processing time
If you envelop the content by a period order (time {..}), you can perceive what amount of time it requires to strainer for each excellent rundown to the upper hunt limit.

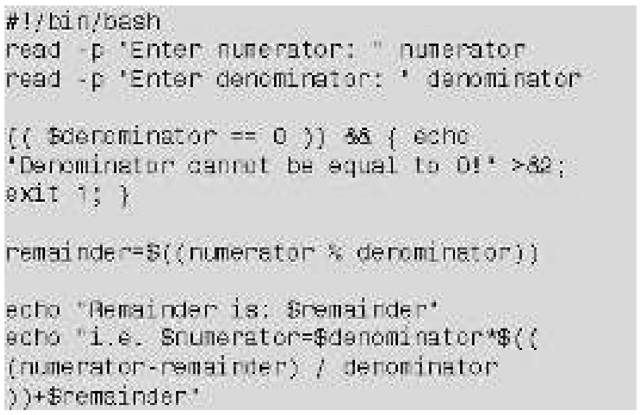


```
Primes up to 100:

real 0m0.249s
user 0m0.008s
sys  0m0.020s

Primes up to 1,000:

real 0m1.384s
user 0m0.076s
sys  0m0.096s

Primes up to 10,000:

real 0m31.941s
user 0m5.280s
sys  0m1.952s

And just for fun, primes up to 100,000:

real 17m33.090s
user 8m20.343s
sys  1m58.447s
```

Remainder
Given a numerator and denominator, ascertain the remainder.

Modulus method
This inquiry is minor assuming you utilize the modulus (%) administrator. Basically read the client input, compute and show the modulus and you are finished.

I added one just check to guarantee that the denominator isn ’t equivalent to zero so division can happen without error.
The final line simply shows the arithmetic expression (i.e. a = q*d + r).
One note is that while working out the rest of negative numbers, there will be two potential answers which I didn’t try to code.



```
#!/bin/bash
read -p 'Enter numerator: " numerator
read -p 'Enter denominator: ' denominator

(( $denominator == 0 )) && { echo
'Denominator cannot be equal to 0!' >&2;
exit 1; }

remainder=$((numerator % denominator))

echo "Remainder is: $remainder"
echo "i.e. $numerator=$denominator*$((
(numerator-remainder) / denominator
))+$remainder"
```

Formula method
If for reasons unknown you don’t (or can’t) utilize the modulus administrator, you can generally count on the number-crunching articulation to compute the remainder.

The strategy beneath takes advantage of the manner in which BASH does math: it just does number math except if in any case determined. In this manner, 9/5=1, not 1.8. Utilizing this, it is not difficult to compose an articulation that determined the rest of basically taking away the biggest entire different.

Once more, this code experiences a similar issue as the one above, in particular that it just works out one of the leftovers for negative numbers with the main blunder beware of the denominator not being 0.

```
#!/bin/bash
read -p 'Enter numerator: ' numerator
read -p 'Enter denominator: ' denominator

(( $denominator == 0 )) && { echo
'Denominator cannot be equal to 0!' >&2;
exit 1; }

# BASH only preforms integer math unless
otherwise specified so 9/5=1

remainder=$(( numerator- (
(numerator/denominator) * denominator) ))

echo "Remainder is: $remainder"
echo "i.e.
$numerator=$denominator*$((numerator/denomin
ator))+$remainder"
```

Sample output:

```
$ ./remainder
Enter numerator: 133419
Enter denominator: 234
Remainder is: 39
i.e. 133419=234*570+39
```

Test for a denominator equivalent to zero:

```
$ ./remainder
Enter numerator: 12345
Enter denominator: 0
Denominator cannot be equal to 0!
```

Reverse a sentence
Given a sentence as info, turn around the words however not their singular letters. Accordingly, the sentence “One two three” would bring about“three two One”.
Using a cluster is the main reasonable method for achieving this. Assuming

you don ’t, you’ll have a migraine of a period looking over for whitespace, putting away the characters until the following whitespace and rehashing until the finish of the sentence where you show it onscreen.

In my solution, I read in a string passed to the script into an array (the -a sentence_array part) as arguments separated by whitespace (i.e. a standard sentence). Note that I don't need to specify IFS=' ' since by default, IFS separates by whitespace (spaces, tabs and newlines).

Using a for loop, I simply iterate from the end to the beginning, one element at a time in the array. Since all whitespace is used as deliminators, the array will be full (i.e. not sparse), and display with only one space between the words when echoed back.

```
#!/bin/bash
read -r -a sentence_array <<< "$1"
for (( i="${#sentence_array[@]}"; i>=0; i--
))
do
     sentence_string="$sentence_string
${sentence_array[i]}"
done
echo "Reversed sentence is:$sentence_string"
```

Sample output:

```
$ ./reverse_sentence "This is a nice, long,
sentence with a number of words and spaces
that I wouldn't want to reverse manually."

Reversed sentence is: manually. reverse to
want wouldn't I that spaces and words of
number a with sentence long, nice, a is This
```

Reverse a string

Given an arbitrary string as information, compose a content which switches it character-for-character.

Cheating

```
$ echo 'This is a sentence' | rev
ecnetnes a si sihT
#!/bin/bash
echo "$1" | rev
$ ./reverse_string "Kinda long test sentence
using rev"
ver gnisu ecnetnes tset gnol adniK
```

For loop

While loop

```
#!/bin/bash
string="$1"
reverse_string=""
string_length="${#string}"
while (( "$string_length">=1 ))
do
     string_length=$(( $string_length - 1 ))

reverse_string="$reverse_string${string:$str
ing_length:1}"
done
echo "$reverse_string"
```

Sample output:

```
$ ./reverse_string 123456789
987654321
$ ./reverse_string
abcdefghijklmnoporstuvwxyz

zyxwvutsrqponmlkjihgfedcba
$ ./reverse_string 'this is a long, horrible
string to reverse'

esrever ot gnirts elbirroh ,gnol a si siht
```

Serpinski

Calculate and show all cycles of the Serpinski equation from 1 to the given iteration.

Sample output:

```
Enter number of iterations to perform: 20
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
987 1597 2584 4181 6765
```

```
Enter number of iterations to perform: 10
1 -
1 -
2 --
3 ---
5 -----
8 --------
13 -------------
21 ---------------------
34 ----------------------------------
55 -----------------------------------------
--------------
```

Quick Reference

The motivation behind this segment is to give a quick way query fundamental Bash punctuation canvassed in this guide.

Basics

Prompts

```
$    Bourne, POSIX, Korn
%    csh, ksh
#    root
```

Comments

```
$ command      # comments are not displayed
```

Arguments

The order and all that after are liable to word splitting.

```
$ command arg1 arg2 . . . argN
```

Aliases

```
$ alias aname='command arg1 arg2 . . .argN'
```

Functions

Help

```
function_name() {
commands
}
function_name
```

Man pages

```
$ man command
$ man 3 command
```

Help pages

```
$ help [[
[[ ... ]]: [[ expression ]]
    Execute conditional command.

    Returns a status of 0 or 1 depending on
the evaluation of the conditional
. . .
```

Whatis

```
$ whatis bash
bash (1)             - GNU Bourne-Again
Shell
$ man -f bash
bash (1)             - GNU Bourne-Again
Shell
```

Apropos

```
$ apropos bash
bash (1)             - GNU Bourne-Again
Shell
bash-builtins (7)    - bash built-in
commands, see bash(1)
. . .
$ man -k bash
bash (1)             - GNU Bourne-Again
Shell
bash-builtins (7)    - bash built-in
commands, see bash(1)
. . .
```

Variables

Strings

```
$ identifier=value
$ unset identifier
```

Expansion

```
$ echo $identifier
value
```

Integer

```
$ declare -i varname
$ unset varname
$ declare -i x=10; x+=5; echo $x; unset x
15
$ x=10; let x+=5; echo "$x"; unset x
15
```

Read only

```
$ declare -r varname
```

Shell
Variable Description
$PATH
$PIPESTATUS

$PPID
$PS1, …, $PS4 $PWD
$RANDOM

$SECONDS
$UID

Environment
Variable
BASH_VERSION BROWSER

DISPLAY

EDITOR
HOSTNAME HISTFILE

HOME
IFS
LANG

LOGNAME MAIL
MANPATH PAGER

PATH

The command search path.
The exist status of the last completed pipe. The process ID of the current

shell’s

parent process.
The shell’s (prompts 1 through 4) format. The current working directory.
A pseudo random integer between 0

2^15.

How many seconds the script has been executing.
The current user’s ID.

Description
Bash’s version number.
The system’s default web browser.
The hostname, display and screen number for

graphical application display.

The default file editor. VISUAL falls back to this if it fails.
The system’s name.
The path to the file were command history is saved.
The currentuser’s home directory.
The **I** nternal **F** ield **S** eparator’s global setting.
The language settings used by system applications.
The currently logged in user.
The storage location for system mail.
The storage location for system manpages.
The utility used to display long text output.
The colon separated search path for commands.
PS1 The prompt settings.
PWD The current working directory. **SHELL** The path to the current user’s shell program. **TERM** The current terminal emulation type. **TZ** The timezone used by the system clock. **USER** The currently logged in user. **VISUAL** The default GUI file editor.

Arrays, indexed

Structure
array
key value 0 *value1* **1** *value2* **. . . . . .** **N** *valueN*

Declaration Usage
array=()
array[0]=value

declare -a array Description
Create array
Declares an empty array called *array* . **Create & set array key**
Declares array, *array* , and sets its first value

(index 0) to *value* .
Create array
Declares an empty array, *array* .

Storing values Usage

array[i]=value Description
Set array value
Sets the *i* th key of *array* to *value* .

array=(value1, value2, . . ., valueN)

array+=(value1, value2, . . ., valueN)
Set entire array
Sets the entire array, *array* , to any

number of values, *value1, value2, . . .valueN* , which are indexed *in order, from zero* . Any previously set array values are lost unless the array is appended with +=

Append array
Appends the existing array, *array* , with the values, *value1, value2, . . .valueN* .

array=([i]=value1,
[j]=value2, . . ., [k]=valueN) Compound set array values Set the indexed keys, *i, j,* and *k* , to

value1, value2, . . .valueN respectively. Any previously set array values are lost. **Retrieving values Usage**
${array[i]}
${array[@]} "${array[@]}"
${array[*]} "${array[*]}"
${array[@]:offset:num} "${array[@]:offset: num }"
${array[*]:offset:num } "${array[*]:offset:num

}"
Description
Expand value
Expands to the *i* th value of indexed

array, *array* . Negative numbers count backwards from the highest indexed key. **Mass expand values**
Expands to all values in the array. If double quoted, it expands to all values in the array individually quoted.
Mass expand values
Expands to all values in the array. If double quoted, it expands to all values in the array quoted as a whole.
Subarray expansion
Expands to *num* of array values, beginning at *offset* . The same quoting rules as above apply.
Subarray expansion
Expands to *num* of array values, beginning at *offset* . The same quoting rules as above apply.

Metadata
Usage Description

${#array[i]}
Value string length
Expands to the string length of the *i* th array value

${#array[@]} Array values
${#array[*]}** Expands to the number of values in the array. **${!array[@]}
Array indexes

${!array[*]} Expands to the indexes in the array.

Deletion
Usage
unset -v array unset -v

array[@]
unset -v array[*] unset -v array[i] Description
Erase an array
Completely erases the array, *array* .
Erase an array value
Erases the *i* th array value from *array* .

Arrays, associative

Structure
array
string label value *string1 value1 otherstring2 value2* **. . . . . .**
anotherstring3 valueN

Declaration
Usage
declare -A array

array[str]=value Description
Create array
Declares an empty array, *array* .
Create & set array key
Declares array, *array* , and sets its first *string*

label to *value* . **Storing values**
Usage Description

Set array value
array[str]=value Sets the element indexed by *str* of *array* to *value* .

array=([str1]=value1, [str2]=value2, . . .,
[str3]=valueN)
Compound set array values
Set the elements indexed by strings, *str1,*

str2, and *str3* , to *value1, value2, . . .valueN* respectively. Any previously set array values are lost.

Retrieving values Usage

${array[str]} Description
Expand value
Expands to the element indexed by string

str of array, *array* .
${array[@]} "${array[@]}"

${array[*]} "${array[*]}" Mass expand values
Expands to all values in the array. If double

quoted, it expands to all values in the array individually quoted. Output is in random order.

Mass expand values
Expands to all values in the array. If double quoted, it expands to all values in the array. quoted as a whole. Output is in random order.

Metadata Usage
${#array[str]}
${#array[@]}
${#array[*]}
${!array[@]}
${!array[*]}

Deletion
Usage
unset -v array
unset -v array[@] unset -v array[*]

unset -v array[str]

unset -v array
unset -v array[@] unset -v array[*] Description
Value string length
Expands to the string length of the element

indexed by *str*
Array values
Expands to the number of values in the

array.
Array indexes
Expands to the string labels in the array.

Description
Erase an array
Completely erases the array, *array* . **Erase an array value**
Erases the element indexed by *str* from *array* .
Erase an array
Completely erases the array, *array* .

Special characters

Basic
Character Description

#
Comment
Lines beginning with a hash will not be executed. **Whitespace**

" " Bash uses whitespace (e.g. spaces, tabs, and newlines) to perform word splitting.
Run in background
& Cause the preceding command to run in the background.
Command separator
; Allows for the placement of another command on the same line.
Logic
Character Description
And
&& Logical *and* operator. Returns a success if both of the conditions before *and* after the operator are true. **Or**
|| Logical *or* operator. Returns a success if either of the conditions before *or* after the operator are true. **Directory traversal**
Character Description
Home directory

~ Represents the home directory, and the current user's home directory when followed by a forward slash. **Current directory**
A dot in front of a filename makes it "hidden." Use *ls -a*

to view.
. A dot directory represents the current working
..
/
directory.

A dot separated from a filename by a space *sources* (loads) the file.
Parent directory
A double dot directory represents the parent directory.
Filename separator
A forward slash separates the components of a filename.

Quoting Character
\
' '

" " Description
Escape

Escapes the following special character and causes it to

be treated literally.
Allows for line splitting of a long command. **Full quoting**
Special characters within single quotes lose their

meaning and become literal.
Word splitting is not performed on the contents. **Partial quoting**
Special characters within double quotes lose their

meaning, with the notable exception of parameter expansion, arithmetic expansion, and command substitution.

Word splitting is not performed on the contents. **Redirection Character >**
>>

< Description
Output redirection
Redirects standard output, typically to a file. **Output redirection**
Redirects and *appends* standard output, typically to a file. **Input redirection**
Redirects standard input, typically for reading in a file's

contents.
<<
<<<

| Here document
Reads in strings of data until a specified delimiter is

encountered.
Here string
Reads in a single string of data immediately following. **Pipe**
Redirects the standard output of a command to the

standard input of another command.
Groups
Character

{ }
()

(()) Description
Inline group
Commands within the curly braces are treated as a

single command. Essentially a nameless function without the ability to assign or use local variables. The final command in an inline group must be terminated with a semicolon.

Command group

Commands within are executed as a subshell. Variables inside the subshell are not visible to the rest of the script.
Arithmetic expression
Within the expression, mathematical operators (+ , - , * , */, >, <, etc.*) take on their mathematical meanings (addition, subtraction, multiplication, division). Used for assigning variable values and in tests.

Globs

Character ?
*

[…] Description
Wildcard (character)
Serves as a match for a single character.
Wildcard (string)
Serves as a match-for any number of characters. **Wildcard (list)**
Serves as a match for a list of characters or ranges.

Null globs

```
shopt -s nullglob        #set
shopt -u nullglob        #unset
```

Extended globs

```
shopt -s extglob         #set
shopt -u extglob         #unset
```

Glob Description
?(list) Matches exactly zero or one of the listed pattern. ***(list)** Matches any number of the listed patterns. **@(list)** Matches exactly one of the listed patterns. **+(list)** Matches at least one of the listed patterns. **!(list)** Matches anything *but* the listed patterns.

Parameters

Positional
Parameter Description
$0 The called name of the script.

$1, $2, … $9
Print the arguments 0 through 9 passed to the script.
${10}. ${11} . . Print the arguments 10 and higher.

Special
Parameter $#
$*
$@
$?
$!

$$
$_
$

Expansion Description
The number of positional parameters passed to the

script.

All the positional parameters. If double quoted, results in a single *string* of them all.
All the positional parameters. If double quoted, results in a single *list* of them all.
The exit code of the last completed foreground command.

The exit code of the last completed background command.
The process ID of the current shell.
The last argument of the last completed command.
The shell options that are set.

Usage Description
Append
${parameter}

Allows for additional characters beyond the end curly brace to be appended to *parameter* after substitution.

String length
${#parameter} The number of characters in *parameter.*
Modifying character case Usage
${parameter^}
${parameter^^}
${parameter,}
${parameter,,}
${parameter~}

${parameter~~} Description
Uppercase
Converts the *first* character to uppercase. **Uppercase**
Converts *all* characters to uppercase. **Lowercase**
Converts the *first* character to lowercase. **Lowercase**
Converts *all* characters to lowercase. **Reversecase**
Reverses the case of the *first* character. **Reversecase**
Reverses the case of *all* characters.

Assigning values Usage
${parameter:=value} ${parameter=value}
${parameter:+value} $parameter+word}

${parameter:-value} $(parameter-value} Description
Assign default value
Set *parameter* to *value* if *parameter* is not set,

then substitute *parameter* .
Use alternate value
Substitute nothing for *parameter* if *parameter*

is not set, else substitute *value* .
Use default value
Substitute *value* for *parameter* if *parameter* is

not set, else substitute *parameter* .
Substring removal Usage

${parameter:offset:length} Description
Results in a string of *length*

characters, starting at *offset* characters. If *length* is not specified, takes all characters
${parameter#pattern}
${parameter##pattern}
${parameter%pattern}
${parameter%%pattern}
after *offset* . If *length* is negative, count backwards.

Searches from front to back *parameter* until the *first* occurrence of *pattern* is found and deletes the result.

Searches from front to back of *parameter* until the *last* occurrence of *pattern* is found and deletes the result.

Searches from back to front of *parameter* until the *first* occurrence of *pattern* is found and deletes the result.

Searches from back to front of *parameter* until the *last* occurrence of *pattern* is found and deletes the result.

Search and replace Usage
${parameter/pattern/value}
${parameter//pattern/value}
${parameter/pattern} ${parameter//pattern}

Command substitution

Arithmetic expansion
Character Description Arithmetic expansion
$(( ))

Works exactly the same as arithmetic expression, except the entire expression is replaced with its mathematical result.

Description
Searches from front to back of *parameter* and replaces the *first* occurrence of *pattern* with *value* .
Searches from front to back of *parameter* and replaces *all* occurrences of *pattern* with *value* .
By leaving out *value* , the *first* instance of *pattern* in *parameter* is replaced with the null string.
By leaving out *value* , *all* instances of *pattern* in *parameter* are replaced with the null string.

Brace expansion
Character Description
Brace expansion
{} Expands the character sequence within.
Command substitution
Character Description
Command substitution
` ` The output of the command within the backticks are run and substituted in place of the entire backtick grouping. **Command substitution**
$( ) Exactly the same as backticks above, but provide nestability and better readability.

Conditional blocks

Exit codes Exit Code 0
1

2
126

127
128

Description
Successful execution.
Unsuccessful execution catchall. Incorrect use of a shell builtin Command cannot execute Command not found (typically a

typo)
Incorrect exit code argument
128+num Fatal error signal “num”

Example true
false
ls —option /dev/null

toucj

exit 2.718 kill -9 *pid* # num =

137
kill -15 *pid* # num =

143
130
255*

If

Script killed with Control-C Exit code out of range [CTRL]-[c] exit 666

```
if expression; then
     statement_1
     statement_2
     …
     statement_n
fi
```

If-else

```
if expression
then
     statements
else
     other_statements
fi
```

If-elif-else

```
if expression_1
then
     statements_1
elif expression_2
then
     statements_2
elif expression_3
then
     statements_3
else
     statements_4
fi
```

If-else nesting

```
if expression
then
     statements0
     if expression0
     then
          statements0-0
     else
          statements0-1
else
     statements1
     if expression1
     then
          statements1-0
     else
          statements1-1
fi
```

Case

```
case variable in
pattern1)
          statements1
          ;;
pattern2)
          statements2
          ;;
pattern3)
          statements3
          ;;
*)
          statements*
          ;;
esac
```

Select

Tests

Types
Conditional
!
[ ]

[[ ]] Description
Negate
Negates a test or exit status.
If issued from the command line, invokes Bash's

history mechanism.
Test
The expression between the brackets is tested and

results in a true/false output. This is a shell builtin. If not concerned with compatibility with the *sh* shell, the double square bracket test is much more flexible.

Test
Tests the expression between the brackets but provides many more options, features, and flexibility than single square brackets. It is a shell keyword. Tests will be covered later.

Arithmetic Operator *X* **-eq** *Y*
X **-ne** *Y*
X **-gt** *Y*
X **-lt** *Y*
X **-le** *Y*
X **-ge** *Y*

Description
True if *X* is **equal to** *Y*
True if *X* is **not equal to** *Y*
True if *X* is **greater than** *Y*
True if *X* is **less than** *Y*

True if *X* is **less than or equal to** *Y* True if *X* is **greater than or equal to** *Y*
bc:
[[operator bc equivalent
-eq ==
-ne !=
-gt >
-lt < *Y*
-le <=
-ge >=

Strings
Operator Description
-z *STRING* True if *STRING* is **empty**
-n *STRING* True if *STRING* is **not empty** *STRING1 = STRING2* True if *STRING1* is **equal to** *STRING2 STRING1 == STRING2* True if *STRING1* is **equal to** *STRING2*

STRING1
!
=
STRING2

True if *STRING1* is **not equal to** *STRING2*
True if *STRING1* is **sorts** *STRING1 < STRING2* **lexicographically before** *STRING2* (must be escaped if using [)
True if *STRING1* is **sorts** *STRING1 > STRING2* **lexicographically after** *STRING2* (must be escaped if using [)

Files
Operator Description
-e *FILE* True if *FILE* exists
-f *FILE* True if *FILE* exists and is a **regular** file
-d *FILE* True if *FILE* exists and is a **directory**

-c
FILE
True if *FILE* exists and is a **special character** file
-b *FILE* True if *FILE* exists and is a **special block** file

-p
FILE

True if *FILE* exists and is a **named pipe** (FIFO —“first in, first out”)
-S *FILE* True if *FILE* exists and is a **socket** file

-h *FILE* True if *FILE* exists and is a **symbolic link**
-O
FILE
True if *FILE* exists and is **effectively owned by you**
-G *FILE*

-g *FILE*
-u *FILE*
-r *FILE*
-w *FILE*
-x *FILE*

-s *FILE*
-t *FD*

FILE1 **-nt**
FILE2
FILE1 **-ot**
FILE2
FILE1 **-ef**
FILE2

Logical
Operator
EXP1 **&&**

EXP2
EXP1 || *EXP2*

Patterns Operator *STRING = PATTERN STRING == PATTERN*

STRING =~ PATTERN

True if *FILE* exists and is **effectively owned by your group**
True if *FILE* exists and has **SGID** set
True if *FILE* exists and has **SUID** set
True if *FILE* exists and is **readable** by you
True if *FILE* exists and is **writable** by you
True if *FILE* exists and is **executable** by you
True if *FILE* exists and is **non-empty** (size is > 0)
True if *FD* (file descriptor) is opened on a terminal

True if *FILE1* is **newer than** *FILE2*
True if *FILE1* is **older than** *FILE2* True if *FILE1* and *FILE2* refer to the **same device and inode numbers**
Description
True if **both** *EXP1* **and** *EXP2* are **true** (equivalent of **-a** if using [])
True if **either** *EXP1* **or** *EXP2* is **true** (equivalent of **-o** if using [])

Description
True if *STRING* **matches** *PATTERN* True if *STRING* **matches** *PATTERN*
True if *STRING* **matches the regular**

expression pattern, *PATTERN*
Character classes
Class Description
alnum

Alphanumeric characters. Equivalent to specifying both *alpha* and *digit* , or [0-9A-Za-z].
alpha Alphabetic characters. Equivalent to [A-Za-z]. *ascii* ASCII characters.
blank Space and tab.

cntrl

Control characters. These are the characters with octal codes 000 through 037, and 177.
digit Digits. Equivalent to [0-9].
graph Visible characters. Equivalent to *alnum* and *punct. lower* Lowercase characters. Equivalent to [a-z].

print
Visible characters and spaces. Equivalent to *alnum* , *punct* , and space.
punct
Punctuation characters.
! @ # $ % ^ & * ( ) { } [ ] _ + - , . : ; < > = \| / ' “ ` ~ ? *space*

Whitespace characters. Tab, newline, vertical tab, newline, carriage return, space, and form feed.
upper Uppercase characters
word Alphanumeric characters plus “_”
xdigit Hexadecimal digits

Miscellaneous
Operator Description
! *TEST* **Inverts** the result of *TEST*
(*TEST* **Groups** a *TEST* (changes precedence of tests, must be

) escaped if using [])
-o *OPT* True if shell option, *OPT* , **is set**
-v *VAR* True if variable, *VAR* , **is set**

Loops

While

```
while condition
do
     statement1
     statement2
     . . .
     statementN
done
```

Until

```
until condition
do
    statement1
    statement2
    . . .
    statementN
done
```

For

```
for (( initial; condition; increment ))
do
    statement1
    statement2
    . . .
    statementN
done
```

File descriptors

Standard streams
Name Stream Standard stdininput
Standard stdoutoutput
Standard stderrerror

Default
File Descriptor

Keyboard 0
Screen 1
Screen 2
Redirection *stdout of order to file:*

```
$ command > file
$ command 1> file
```

record to stdin of command:

```
$ command < file
$ command 0< file
```

stderr of order to location:

```
$ command args 2>/location
```

Duplication

stdout copied and set in stderr:

```
2>&1
```

Redirection

Write Append

```
$ echo "This writes to a file." > myfile
```

Read

```
$ echo "This appends to a file." >> myfile
```

Here document

```
$ cat < myfile
```

```
command << delimiter
data
delimiter
```

Here string FIFO

```
command <<< "string"
```

PIPE

```
$ mkfifo redirector
```

Process substitution

```
$ command1 args | command2 args
```

```
<()       # process input substitution
>()       # process output substitution
```

Traps

Signals
Name Value Effect Shortcut Trappable? EXIT 0 Exit
SIGHUP 1 Hangup Yes

Set

```
trap COMMAND SIGNAL
```

Reset

```
trap - SIGNAL
```

Ignore

```
trap '' SIGNAL
```