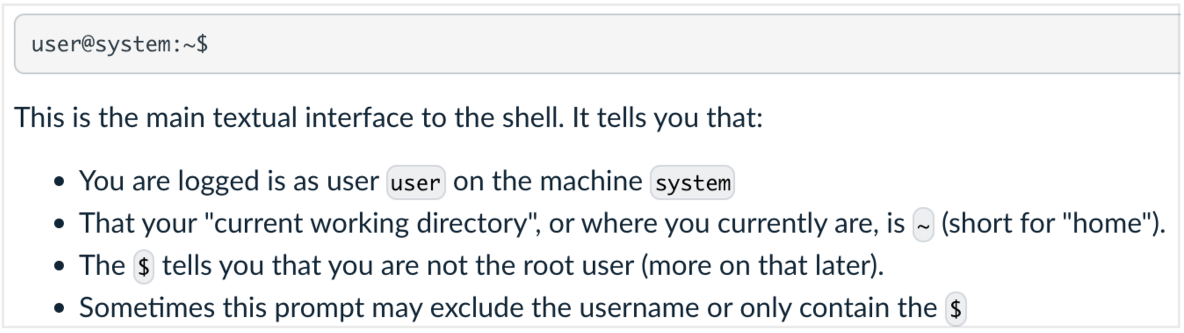
Section B1 Linux

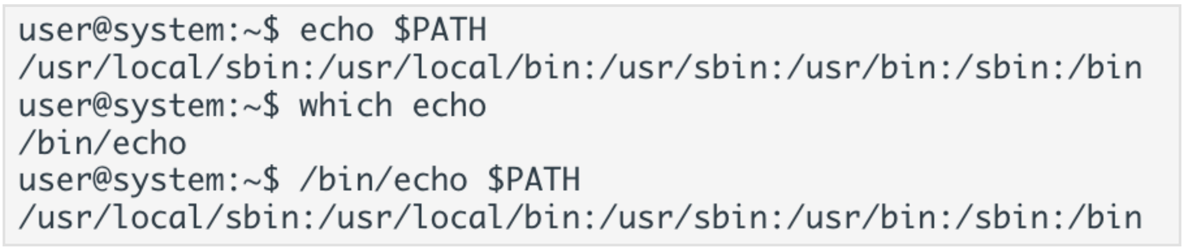


```
user@system:~$
```

This is the main textual interface to the shell. It tells you that:

- You are logged is as user `user` on the machine `system`
- That your "current working directory", or where you currently are, is `~` (short for "home").
- The `$` tells you that you are not the root user (more on that later).
- Sometimes this prompt may exclude the username or only contain the `$`

If the shell is asked to execute a command that doesn't match one of its programming keywords, it consults an *environment variable* called $PATH that lists which directories the shell should search for programs when it is given a command:



```
user@system:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
user@system:~$ which echo
/bin/echo
user@system:~$ /bin/echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

. refers to the current directory, and .. to its parent directory

commands

Create an empty file
The touch command modifies file properties.
Display the contents of a (plain text) file
The cat command displays the contents of a file.
Copy a file
The cp command copies a file.
It takes 2 arguments; the name/path of the file to copy, and the name/path of the destination:
Move a file
The mv command moves a file.
It takes 2 arguments; the name/path of the file to move, and the name/path of the destination:
Delete a file
The rm command deletes files.
It takes 1 arguments; the name/path of the file to delete:
Create a directory/folder
The mkdir command creates a directory.
Get more information about a program

give the man program a try. It takes as an argument the name of a program, and shows you its *manual page*. Press q to exit.
Ctrl+R will give you a search prompt that you can use to find previous commands:
Finding files (by name)
One of the most common repetitive tasks that every programmer faces is finding files or directories. All UNIX-like systems come packaged with **find** , a great shell tool to find files. find will recursively search for files matching some criteria. Some examples:

```
# Find all directories named src
find . -name src -type d

# Find all python files that have a folder named test in their path
find . -path '*/test/*.py' -type f

# Find all files modified in the last day
find . -mtime -1
# Find all zip files with size in range 500k to 10M
find . -size +500k -size -10M -name '*.tar.gz'
```

Beyond listing files, find can also perform actions over files that match your query. This property can be incredibly helpful to simplify what could be fairly monotonous tasks.

```
# Delete all files with .tmp extension
find . -name '*.tmp' -exec rm {} \;
# Find all PNG files and convert them to JPG
find . -name '*.png' -exec convert {} {}.jpg \;
```

Finding content in files
The **grep** command searches file contents.
It takes 2 arguments; the the pattern to search for, and the name/path of the destination:

```
user@system:~$ grep "thing to search for" path/to/file
This is the thing to search for in this file
```

Use the **-R** flag to search recursively (all subfolders and their files)
Arrange the content of files
The **sort** command arranges the lines of a file in alphabetical order and outputs to the Shell:

```
user@system:~$ cat foo
Some line
Another line
Last line
user@system:~$ sort foo
Another line
Last line
Some line
```

The sort command only outputs the sorted content without modifying the file.

Output only the start/end of a file

The **head** and **tail** commands can be used to output only the start/end of a file respectively:

Compare files

The **diff** command compares files.

It takes 2 arguments; the name/path of each file to compare, the generates an output that shows all the places where the files differ. If the files have the same contents, no output is produced.

Some common options are **-B** which ignores blank lines, **-i** which ignores case-sensitivity when comparing, and **-w** which ignores all differences in whitespace.

Redirecting Input & Output

The simplest form of redirection is **< file** and **> file**. These let you rewire the input and output streams of a program to a file respectively:

```
user@system:~$ echo hello > hello.txt
user@system:~$ cat hello.txt
hello
user@system:~$ cat < hello.txt
hello
user@system:~$ cat < hello.txt > hello2.txt
user@system:~$ cat hello2.txt
hello
```

It's worth noting that cat hello.txt and cat < hello.txt produce the same result, but they do it in slightly different ways. The former directly tells cat which file to read, while the latter redirects the file's content to cat.

cat < hello.txt > hello2.txt

- This combines both input (<) and output (>) redirection. The command reads the content of hello.txt and then writes it to hello2.txt.

You can also use **>>** to append to a file. Normally, the shell will happily overwrite any existing file when you redirect output to that filename. In contrast, to append content to an existing file, use >> like

```
user@system:~$ echo 'hello'  > hello.txt
user@system:~$ echo 'and hello again!'  >> hello.txt
user@system:~$ cat hello.txt
hello
and hello again!
```

the difference between > and >> is:

- > writes to a file, overwriting its content if it already exists.
- >> appends to a file, adding to its existing content (or creating a new file if it doesn't exist).
- < takes the input for a command from a file instead of the keyboard.

Pipes

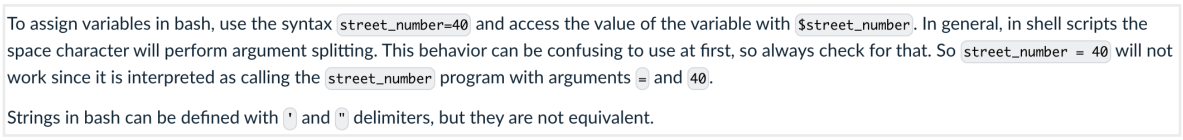
| : The pipe operator allows you to use the output of one command as the input to another command. This lets you "chain" commands together in powerful ways.
head $file | tail --lines=2
The head $file command will, by default, output the first 10 lines of the file. Piping this to tail --lines=2 will show the last two lines of those ten lines, which are lines 9 and 10.

Globbing

- Wildcards(通配符) - Whenever you want to perform some sort of wildcard matching, you can use **?** and ***** to match one or any amount of characters respectively.
- Curly braces **{}** - Whenever you have a common substring in a series of commands, you can use curly braces for bash to expand this automatically.

```
convert image.{png,jpg}
# Will expand to
convert image.png image.jpg
```

$Variables



To assign variables in bash, use the syntax `street_number=40` and access the value of the variable with `$street_number`. In general, in shell scripts the space character will perform argument splitting. This behavior can be confusing to use at first, so always check for that. So `street_number = 40` will not work since it is interpreted as calling the `street_number` program with arguments `=` and `40`.

Strings in bash can be defined with `'` and `"` delimiters, but they are not equivalent.

- If you want to include variables inside a string and have them replaced by their values, use double quotes (" ").
- If you want to treat a string as a literal string and ignore any special characters or variables, use single quotes (' ').

Environment Variables

we can use *environment variables* to make variables *global* to all of the commands invoked from the shell.
You can aslo define your own environment variables using the builtin bash command **export**

Executing a Script & Shebang

The directive to the shell has the name "*shebang*" or "*hash-bang*" from its identifying characters **#!**
we need to use this incantation **chmod u+x** "$filename" to allow ourselves to execute a file.

Arguments & Return Codes

- $0 - Name of the script
- $1 to $9 - Arguments to the script. $1 is the first argument and so on.
- $@ - All the arguments
- $# - Number of arguments
- $? - Return code of the previous command
- $$ - Process identification number (PID) for the current script
- !! - Entire last command, including arguments. A common pattern is to execute a command only for it to fail due to missing permissions; you can quickly re-execute the command with sudo by doing sudo !!
- $_ - Last argument from the last command. If you are in an interactive shell, you can also quickly get this value by typing Esc followed by . or Alt+.

Commands will often return output using **STDOUT**, errors through **STDERR**, and a Return Code to report errors in a more script-friendly manner. A value of 0 usually means everything went OK; anything different from 0 means an error occurred.
The **true** program will always have a 0 return code and the **false** command will always have a 1 return code.

Q

Write shell commands to create the following directory/file structure in the working directory:

```
.
└── dir9
    └── dir9_2
        └── dir9_3
            └── foo
```

mkdir -p dir9/dir9_2/dir9_3
touch dir9/dir9_2/dir9_3/foo