What is the Shell?

These days, computers come with all sorts of interfaces for commands—fancy graphical ones, voice commands, and even AR/VR options. While these are user-friendly for most situations, they often have limitations. You can't press a button that doesn't exist or say a command that hasn't been programmed. To really unlock the potential of your computer, you need to go back to basics and use a text-based interface: the Shell.

Most platforms offer some version of a shell, and many even have multiple options. Although they differ in specifics, they all share a common function: letting you run programs, provide inputs, and view outputs in a structured way.

In this session, we'll look at the Bourne Again SHell (bash), which is one of the most popular shells and has syntax similar to many others. To start using a shell, you need to open a terminal, which is likely pre-installed on your device or easy to install.

Using the Shell

When you open your terminal, you'll see a prompt that looks something like this:

```
missing:~$
```

This indicates you're on a machine named "missing" and your current working directory is "~" (short for your home directory). The "$" shows you're not logged in as the root user. At this prompt, you can type commands, and the shell will interpret them. For instance, typing:

```
missing:~$ date
```

will display the current date and time. The shell will then prompt you for another command. You can also run commands with additional arguments:

```
missing:~$ echo hello
```

```
hello
```

Here, the `echo` command outputs its argument, "hello". The shell breaks down the command by spaces and runs the program mentioned first, using the other words as arguments. If you have an argument with spaces (like "My Photos"), you can either quote it (`"My Photos"`) or escape the spaces (`My\ Photos`).

So, how does the shell know where to find commands like `date` or `echo`? The shell operates like a programming environment with variables and functions. When you enter a command, if it doesn't match a known keyword, it checks the `$PATH` variable, which lists directories to search for the requested programs:

```
missing:~$ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

The `which` command helps you find the specific file that corresponds to a command name:

```
missing:~$ which echo

/bin/echo
```

You can also directly provide a path to bypass `$PATH`.

Navigating in the Shell

Paths in the shell are lists of directories, separated by slashes ("/" on Linux and macOS, "\" on Windows). On Linux and macOS, the path "/" represents the root of the file system. A path that starts with "/" is an absolute path; otherwise, it's a relative path. You can see your current directory with `pwd` and change it with `cd`. In a path, `.` refers to the current directory, while `..` refers to the parent directory:

```
missing:~$ pwd
/home/missing
missing:~$ cd /home
missing:/home$ pwd
/home
missing:/home$ cd ..
missing:/$ pwd
/
```

When you run a program, it usually works in your current directory. To view what's in a directory, use the `ls` command:

```
missing:~$ ls
```

If no directory is specified, `ls` shows the contents of the current one. Most commands accept flags that modify their behavior. For example:

```
missing:~$ ls -l /home
```

The `-l` flag gives a detailed list of files and directories, showing permissions and other attributes. Each line starts with a character indicating if it's a directory (`d`) or a file. The next sets of characters represent the permissions for the owner, group, and others.

Some useful commands to know include `mv` (move/rename), `cp` (copy), and `mkdir` (create a directory).

To learn more about a command's options, use the `man` command followed by the program name:

```
```

```
missing:~$ man ls
```

Connecting Programs

In the shell, programs have two main streams: input and output. Typically, these streams are your terminal (keyboard input and screen output). However, you can redirect these streams.

Basic redirection uses `<` and `>` to send input to a program or output to a file:

```
missing:~$ echo hello > hello.txt

missing:~$ cat hello.txt

hello
```

You can also append to a file with `>>`. A powerful feature is piping (`|`), which allows you to chain programs so that the output of one feeds into the input of another:

```
missing:~$ ls -l / | tail -n1
```

A Versatile and Powerful Tool

Most Unix-like systems have a special user known as the "root" user, which has unrestricted access to all files. However, it's best not to operate as root all the time to avoid accidental issues. Instead, use `sudo` to perform commands with root privileges when needed.

Sometimes, certain actions, like adjusting system parameters via `/sys`, require root access. For example, changing screen brightness can be done through:

```
$ echo 3 | sudo tee brightness
```

This command uses `tee` to write the value into a system file while running with root permissions.

Overall, the shell provides a powerful way to interact with your system, manage files, and run programs efficiently.