Miscellaneous Utilities

`man`

This is one command you need to know. It gives you access to the manual pages ("man pages" or "manpages", for short) for programs. When you don't know what a program does, or what options it has, `man program` should be the first thing you try.

`kill` and `killall`

When a program is out of control, or if it's running in the background, you will probably need to fall back on the `kill` command to terminate it. This takes one or more process IDs (PIDs) as arguments, and optionally the signal to use. SIGTERM is the default, and is usually what you want, though sometimes you want SIGKILL:

```
kill 1234     # kill PID 1234 with TERM signal
kill -9 1234  # kill PID 1234 with KILL signal
```

Note that the TERM signal can be caught by the process being killed, allowing it to clean up after itself. The KILL signal cannot be caught, and causes the process to terminate immediately.

The `killall` program matches command names, rather than PIDs. It is potentially error-prone, but sometimes very useful.

`true` and `false`

These are very useful in scripts. `true` exits with status 0, and does nothing else. `false` exits with a non-0 status (often -1), and does nothing else. These can be used as nops, or to create infinite loops:

```
while true
do
    # ...
done

until false
do
    # ...
done
```

`yes`

This program is similar to the file `/dev/zero`, in that it will keep providing output as long as you read it. Rather than producing nulls, it produces an infinite stream of lines containing the character "y". This can be useful for scripting with tools that require confirmation.

`seq`

This produces a sequence of numbers, optionally with a starting point and increment. Compare the following:

```
seq 5
seq 1 5
seq 1 2 5
seq 5 1
seq 5 -2 1
```

See the manpage for other options, including more complex formatting.

This is useful in scripts to provide a loop over indices:

```
for a in $(seq 0 5)
do
    # ...
done
```

`tar`

On Unix systems, `tar` (Tape Archive) is used more frequently than `zip`, so it's worth learning to use.

Command	*Meaning*
`tar cf foo.tar foo/`	Create a tar file named `foo.tar` from `foo/`
`tar zcf foo.tgz foo/`	As above, but the file will be gzipped
`tar jcf foo.tbz foo/`	As above, but the file will be bzipped
`tar xf foo.tar`	Extract the contacts of `foo.tar`
	Also works on gzipped and bzipped files
`tar tf foo.tar`	Read the table of contents of `foo.tar`

There are many other options, but these will get you far.

`cut`

This is a workhorse for splitting lines of text.

```
cut -d, -f2 foo.csv      # get column 2 from a comma-separated list
cut -d, -f2,4-7 foo.csv  # get columns 2, 4, 5, 6, and 7
ifconfig | grep flags | cut -d\< -f2 | cut -d\> -f1
```

`awk`

cut is somewhat limited, so a more powerful tool is frequently useful. awk has a full programming language, but you'll typically only need a few pieces of it.

By default, awk splits on whitespace, but you can change this with the `-F` option, which takes a regex, rather than a single character. A typical invocation would look like:

```
awk '{ print $1,$3 }' foo.txt
```

to print columns 1 and 3 from foo.txt.

You can also do math in awk, which makes it a useful supplement to bash's integer math. For example:

```
total=$(echo ${total} ${s} | awk '{ print $1 + $2 }')
```

This allows us to sum potentially floating-point numbers. We could also do this by assigning values to variables:

```
total=$(echo | awk -v a=${total} b=${s} '{print a + b }')
```

We still have to pass a file to awk, because it's expecting to operate on a file. Fortunately, echo is fairly light-weight.

Here's an example from a script that updates a single column in a CSV, re-sums the values, and dumps the results. It also strips off a trailing comma, using another utility called `sed` (see the manpage).

```
echo $LINE | awk -v s=${score} -F\, '{
        $5=s
        for (i=3; i<=7; i++) SUM+=$i;
        for (i=1; i<=NF; i++){
                if(i == 2) $i=SUM
                printf "%s,",$i
        }
        print ""
}' | sed 's/,$//g'
```

This overwrites one of the input fields in the line

```
        $5=s
```

The first time we add to the variable `SUM`, it's initialized to 0. The `printf` command works pretty much the same as in C.