# Metacharacters

The shell recognises a number of other characters as special. The most commonly used is the asterisk `*` which can be used to match filenames. For example, the following command echos the name of all files in the current directory which start with j.

```
$ echo j*
junk
```

Characters like `*` which have special properties are known as *metacharacters*. There are a lot of them. The following table gives the complete list.

| | |
|---|---|
| `>` | `cmd > file` direct standard output to `file` |
| `>>` | `cmd >> file` append standard output to `file` |
| `<` | `cmd < file` take standard output from `file` |
| `\|` | `cmd1 \| cmd2` connect standard output of `cmd1` to standard input of `cmd2` |
| `<< str` | *here document*: standard input follows, up to next `str` on a line by itself |
| `*` | match any string of zero or more characters in filenames |
| `?` | match any single character in filenames |
| `[ccc]` | match any character from `ccc` in filenames ranges like `0-9` or `a-z` are legal |
| `;` | command terminator: `cmd1 ; cmd2` does `cmd1` then `cmd2` |
| `&` | like `;` but doesn't wait for `cmd1` to finish |
| `` `...` `` | run command(s) in `...`; output replaces `` `...` `` |
| `(...)` | run command(s) in `...` in a sub-shell |
| `{...}` | run command(s) in `...` in current shell |
| `$1`, `$2` etc. | `$0` ··· `$9` replaced by arguments to shell script |
| `$var` | value of shell variable *var* |
| `${var}` | the value of *var*; avoids confusion when concatenated with text |
| `\` | `\c` take character `c` literally, `\newline` discarded |

| | |
|---|---|
| `'...'` | take `...` literally |
| `"..."` | take `...` literally after `$`, `` `...` `` and `\` interpreted |
| `#` | if `#` starts word, rest of line is a comment |
| `var=value` | assign to variable *`var`* |
| `cmd1 && cmd2` | run `cmd1`; if successful run `cmd2` |
| `cmd1 \|\| cmd2` | run `cmd1`; if unsuccessful run `cmd2` |

Given the number of shell metacharacters, there needs to be some way to tell the shell ``leave it alone". The simplest way to protect characters from being interpreted is to enclose them in single quote characters. For obvious reasons this is known as *quoting*.

```
$ echo '***
***
```

It is also possible to use double quotes `"..."`, but the shell will process any `$`, `` `...` `` and `\` it finds in `"..."`, so don't use this form of quoting unless you want this kind of metacharacter expansion to take place.

Another possibility is to put a backslash `\` in front of *each* character you want to protect from the shell, as in

```
$ echo \*\*\*
```

Quotes of one kind will protect quotes of another.

```
$ echo "Isn't this fun?"
Isn't this fun
```

and they don't need to surround the whole argument.

```
$ echo A'* ?'
A* ?
```

Note that in this last case, there is only one argument to `echo`. The space between `*` and `?` is part of the argument because its special function as a delimiter was removed by the quoting.

Quoted strings can contain newlines:

```
$ echo 'hi
> there'
hi
there
$
```

The string ``> " is the *secondary prompt* printed by the shell when it expects more input to complete a command.

In all these examples, the quoting of metacharacters prevents the shell from trying to interpret them. The command

```
$ echo j*
```

echos the all filenames beginning with `j`. The command `echo` knows nothing about files or shell metacharacters; the interpretation of `*` and any other metacharacters is supplied by the shell. (This is in contrast to where it is the individual programs which are responsible for handling metacharacter expansion (making programming much harder).)

A backslash at the end of a line causes the line to continued; this is the way to type a very long line to the shell.

```
$ echo abc\
> def\
> ghi
abcdefg
$
```

Notice that the newline is discarded when preceded by a backslash, but retained if is included in quotes.

The metacharacter `#` is used for shell comments. If a shell word begins with `#`, it and the rest of the line are ignored.

```
$ echo hi # there
hi
$ echo hi#there
hi#there
```

Next Up Previous
: Creating New Commands : Shell Programming : Command Structure

*Ross Ihaka ´��17ǯ11��26��*