Bash Basics 2: Expansions & Metachars

These slides introduce the basics of working with **Bash**:

1. Intro & Commands
2. **Expansions & Metachars**
   - **shell expansions**
   - **metacharacters for expansions**
   - **parameters**
   - **filename expansion**
3. Variables & Quoting
4. Interactive Use
5. Redirections & Locale

Expansions and Metacharcters

A key aspect of a shell “interpreting” a command line prior to executing it is the application of the various **shell expansions**.

Expansions make use of a number of **metacharacters**: characters that have special interpretations.

Expansions are a key *productivity feature*, since they save typing.

The main **expansion phases** are (in order):

1. **brace expansion**
2. **tilde expansion**
3. **parameter expansion**
4. **command substitution**
5. **arithmetic expansion**
6. **filename expansion** (also known as **globbing**)

Brace Expansion

Brace expansion ({str_1,str_2,etc} or {int_1..int_2}) allows *multiple words* containing *alternative* substrings to be generated:

- `echo a{bc,def,ghi}j` produces:
  `abcj adefj aghij` (a list of three words)
- `mv file.{txt,text}` produces:
  `mv file.txt file.text`
- `rm test.save{01..04}` produces:
  `rm test.save01 test.save02 test.save03 test.save04`
- `cp {a..g}* ~/save` produces:
  `cp a* b* ... f* g* ~/save`
  (Single characters effectively interpreted as ASCII integers.)

(Note that **concatenation** happens automatically in Bash via the adjacent placement of words.)

Tilde Expansion

Tilde expansion (~*username*) allows a users home directory to be compactly specified as part of a path:

- `~smith/Docs/foo` is replaced by `/home/smith/Docs/foo`
- `~` alone is replaced by the current user's home directory: e.g., `~/.bashrc` is replaced by `/home/carver/.bashrc`
- `~` is often used to indicate any user's home directory

Parameters

Before we can discuss **parameter expansion**, we must be clear on just what "**parameters**" are.

In Bash, a **parameter** is an *"entity that stores values."*

There are three types of parameters:

- **variables** – denoted by *names*
- **positional parameters** – denoted by *numbers*
- **special parameters** – denoted by *special characters*

Parameters (contd.)

Parameter examples:

- variables (possible examples): `x`, `file_ext`, etc.
- environment variables: `PATH`, `PWD`, etc.
- positional parameters: `1`, `2`, etc.
- special parameters: `*`, `@`, `?`, `#`, etc.

The positional parameters represent the arguments given to a program/command (or a function) on the command line, with `1` representing the first argument, `2` the second, and so forth.

Parameters (contd.)

Special parameters represent various things, such as:

- `*` – all the CL arguments (`"*"` gives args as one word)
- `@` – all the CL arguments (`"@"` gives args as separate words)
- `#` – the number of CL arguments
- `?` – **exit status** of the last command
- `$` – shell PID

Parameter Expansion

In most programming languages, when a variable is used in an expression or on the RHS of an assignment, the *variable name is replaced with its value*—i.e., the variable is *implicitly* **evaluated** and its value substituted.

This is *not* the case for Bash: variables (and other parameters) must be *explicity evaluated* if their values are desired.

Parameter expansion retrieves a parameter's value:

- `${param}` is replaced by the value stored in parameter `param`
- `$param` can also be used if end of parameter word is clear

Parameter Expansion (contd.)

The parameter expansion syntax also accepts a number of *operators* that can be used to manipulate a retrieved value.

Here are some examples showing the use of select operators: (assume `file` is a variable with value "`test.text.save`")

- `${file%.*}` `file`'s value minus the *final* extension: `test.text`
- `${file%%.*}` `file`'s value minus *all* extensions: `test`
- `${file#*.}` `file`'s value minus everything up to the *first* ".": `text.save`
- `${file##*.}` `file`'s value minus everything up to the *final* ".": `save`

Command Substitution

Command substitution (`$(`*cmd*`)`) executes a command and substitutes its result (output to *standard output*) in its place:

- `$(ls *.txt)` is replaced by a list of all the "`.txt`" files in the CWD
- `` `ls *.txt` `` is alternative syntax ("**backticks**")

Most commonly used in shell scripts to avoid having to use *temporary variables* to store command results.

Can be useful when working interactively to avoid need to do cutting-and-pasting or retyping results:

```
  ls -l $(which mv)
```

Arithmetic Expansion

Arithmentic expansion (`$((`*cmd*`))`) evaluates an arithmetic expression and substitutes its result in its place:

- `$((1 + 2))`
- `$((x + 2))`   (note: do not need to write `$x`)
- `$((x++))`

Note: an artihmetic expansion like "`$((x++))`" must be used differently in Bash than the expression "`x++;`" in C-family languages.

While "`x++;`" is a fine expression in C, having "`$((x++))`" on a line will cause the resulting number to be interpreted as a command: e.g., `bash: 1: command not found`

Filename Expansion

Filename expansion uses several metacharacters to do *pattern matching on filenames*, allowing *multiple files* to be specified compactly.

Note that the filename expansion metachars appear similar to those used in **regular expressions**, but they are *not* the same!

Filename expansion metacharacters:

- `*` – matches any string, including the null/empty string
- `?` – matches any single character (but not none)
- `[...]` – matches any single enclosed character:
  `[yY]` `[abcd]` `[0123456789]`

Filename Expansion (contd.)

`[...]` accepts additional notation:

- a *dash* can be used to specify a range of characters:
  - `[0-9]` for `[0123456789]`
  - `[A-Z]` for upper case alphabetic chars (C locale)
  - `[a-zA-Z]` for all alphabetic chars (C locale)
- `[:`*class*`:]` character class notation:
  `[[:alpha:]]  [[:upper:]]  [[:digit:]]`

Can combine notations:

- `[[:upper:][:digit:]]`
- `[ab[:upper:]0-3]`

Filename Expansion (contd.)

Filename pattern example:

- `cp i*-[1-5]??.{txt,text} ~/tmp`
- will copy these files:
  - `i-123.text` (* matches empty string, ?'s 23)
  - `iabc-5ab.txt` (* matches abc, ?'s ab)
- will not copy these files:
  - `i123.txt` (no dash)
  - `abc-123.txt` (doesn't start with i)
  - `ia-923.txt` (9 is not in range 1-5)
  - `ia-12.txt` (second ? has no match)
  - `i-123.text.save` (does not end in "`txt`" or "`text`")