Bash Basics 3: Variables & Quoting

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

1. Intro & Commands
2. Expansions & Metachars
3. **Variables & Quoting**
   - **variables vs. parameters**
   - **assignment and evaluation**
   - **values**
   - **environment variables and exporting**
   - **quoting**
   - **word splitting**
4. Interactive Use
5. Redirections & Locale

Variables

Variables are one type of Bash **parameter** (*"entity that stores values"*).

Variables are used more in **shell scripts** than interactively, but are occasionally useful on the CL to avoid retyping arguments.

Users may need access to **environment variables** on the CL, and environment variables are accessible as shell variables.

Variables (contd.)

Variables in Bash are usually created via an **assignment**:

- `var=value` (*lack of whitespace is required!*)
- `var=` or `var=""` (assigns empty string value)

A variable's value is retrieved by explicity **evaluating it** (via **parameter expansion**):

- `${var}`
- `$var` (if end of variable name is clear)

Values are effectively stored as *strings*, which may be interpreted as other things (e.g., numbers) by using appropriate syntax.

Variables (contd.)

Variables can be declared using the `declare` builtin.

Variables are generally `declare`'d only if they are to be *arrays*.

They can also be `declare`'d to hold *integers*, which can simplify use of variables in arithmetic expressions.

Can "delete" a variable with the `unset` command.

Environment Variables

When Bash starts, all **environment variables** are automatically added as **shell variables**.

This means you can obtain the values of environment variables by evaluating the corresponding shell variables: `echo $PATH`

Regular shell variables are *not inherited* by any commands run in the shell (recall commands are normally run in subprocesses).

The `export` builtin can be used to cause regular shell variables to be accessible as *environment variables* in subsequent commands.

Quoting

Quoting can be used to disable the special treatment of characters or words, including:

- expansion **metacharacters**
- **word delimiter/separator** characters (e.g., whitespace)
- Bash **reserved words**

There are three quoting mechanisms:

- double quotes: `"filename with spaces"`
- single quotes: `'filename with spaces'`
- escapes: `filename\ with\ spaces`

Quoting (contd.)

Quoting has two main interactive uses:

- avoiding filenames *containing whitespace* from being misinterpreted as *multiple arguments*:
  - e.g., `ls "Linux Talk 1"`
  - e.g., `ls 'Linux Talk 1'`
  - e.g., `ls Linux\ Talk\ 1`
- allowing *metacharacters* to be *passed* into a program (without being removed by the shell):
  - e.g., `myrename "*.txt"`
  - (want program to receive "`*.txt`" as its argument, not a list of `.txt` files)

Quoting (contd.)

The different types of quoting behave differently:

- *single quotes* preserve the literal value of characters (but a single quoted string cannot contain a single quote, even if it is escaped)
- *double quotes* preserve the literal value of most characters, but maintain the *special meaning of*: "$", "`", "\", and "!"
- can get the *literal value* of these characters by *escaping them*:
  `"micro\$oft"`
- can include double quote(s) inside a double-quoted string by *escaping them*:
  `echo "type the command: \"test\" next"`

Quoting (contd.)

It is virtually always a good idea to *enclose variable evaluations inside double quotes*:

- e.g., `"$var"`
- otherwise there can be problems if `var` contains whitespace or is unset

It can sometimes be fairly difficult to understand the result of combining multiple quoting mechanisms, or to figure out how to ensure metachars get passed as arguments to programs.

Word Splitting

As mentioned above, one use of quoting is to suppress filenames from being split into multiple "words."

Bash *tokenizes* its input as a first step in its processing, and does similar"**word splitting**" during the expansion phases, just *prior* to *filename expansion*.

Sometimes it is critical to understand when/where word splitting will occur in order to know if quoting is required.

For example, in a variable **assignment** statement, `name=[value]`, `value` is not subject to word splitting (or filename expansion).

Thus, there is no need to write `x="$y"` rather than `x=$y`, since the result of parameter expansion on `y` will not be word split.

Word Splitting (contd.)

The shell variable `IFS` contains a string of the characters that can serve as word **delimiters/separators** in command lines.

By default, `IFS` is "$\langle space \rangle \langle tab \rangle \langle newline \rangle$" meaning that any sequence of spaces/tabs/newlines serves to separate words.

It is sometimes desirable to change `IFS`.

For example, if one wants to work with lists of filenames that can contain spaces, it might be desirable to split only on newlines.

This can be done with: `IFS=$'\n'`
(note special "C-string" notation to get a $\langle newline \rangle$)