
VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

# System & Network Administration

## Scripting & Shell

Le Hai Duong, PhD. (lhduong@hcmiu.edu.vn)

# Outline

- Scripting philosophy
- Shell basics
- sh scripting
- Regular expression

# Why scripting?

- Standardize administrative chores and free up admins' time
- Record the steps needed to complete a particular task → documentation
- Alternative to scripting is to use the configuration management systems

# Scripting philosophy

- Write microscripts
  - Most admins keep a selection of short scripts for personal use (aka scriptlets) in their **`~/bin`** directories
  - Can also define functions that live inside your shell configuration files (e.g., **`.bash_profile`**)
  - Aliases, e.g. **`alias ls='ls -Fh'`**
- Learn a few tools well (should know a shell, a text editor, and a scripting language thoroughly) ➔ increased productivity
- Automate all the things
- Don’t optimize prematurely ➔ administrative scripts should emphasize programmer efficiency and code clarity rather than computational efficiency
- Pick the right scripting language
  - **Python** and **Ruby** are modern, general-purpose programming languages that are both well suited for administrative work
- Follow best practices

## Scripting language cheat sheet

| Language | Designer | When to use it |
|---|---|---|
| Bourne shell | Stephen Bourne | Simple series of commands, portable scripts |
| **bash** | Brian Fox | Like Bourne shell; nicer but less portable |
| C shell | Bill Joy | Never for scripting; see footnote on page 189 |
| JavaScript | Brendan Eich | Web development, app scripting |
| Perl | Larry Wall | Quick hacks, one-liners, text processing |
| PHP | Rasmus Lerdorf | You've been bad and deserve punishment |
| Python | Guido van Rossum | General-purpose scripting, data wrangling |
| Ruby | "Matz" Matsumoto | General-purpose scripting, web |

# Best practices

- When run with inappropriate arguments, scripts should print a usage message and exit (implement `--help` this way, too)
- **Validate inputs** and **sanity-check** derived values
- Return a **meaningful exit code**: zero for success and nonzero for failure
- Use **appropriate naming conventions** for variables, scripts, and routines (use mixed case or underscores to make long names readable)
- Assign variable names that reflect the values they store, but keep them short
- Developing a style guide
- Start every script with a comment block that tells what the script does and what parameters it takes, etc.
- Comment at the level you yourself will find helpful when you return to the script after a month or two
- Don’t clutter code with useless comments
- It’s OK to run scripts as root, but avoid making them setuid
- Don’t script what you don’t understand
- Adapt code from existing scripts for your own needs but have to understand the code
- With `bash`, use `-x` to echo commands before they are executed and `-n` to check commands for syntax without executing them
-

# Tom Christiansen's five golden rules for useful error messages:

- Error messages should go to STDERR, not STDOUT
- Include the name of the program that's issuing the error
- State what function or operation failed
- If a system call fails, include the `perror` string
- Exit with some code other than 0

# Shell basics

# Shells

- Bourne shell (**`sh`**)
  - most commonly manifested in Almquist shell (**`ash`**, **`dash`**, or **`sh`**) or "Bourne-again" shell, **`bash`**
- **`bash`** is good for both scripting and interactive use
- **`ksh`** (the Korn shell)
- **`zsh`**: compatibility with **`sh`**, **`ksh`**, and **`bash`**, and has many interesting features of its own, including spelling correction and enhanced **globbing**
- **`csh`** (C shell): interactive shell → enhanced version called **`tcsh`**
- **`bash`** is pretty much the universal standard these days

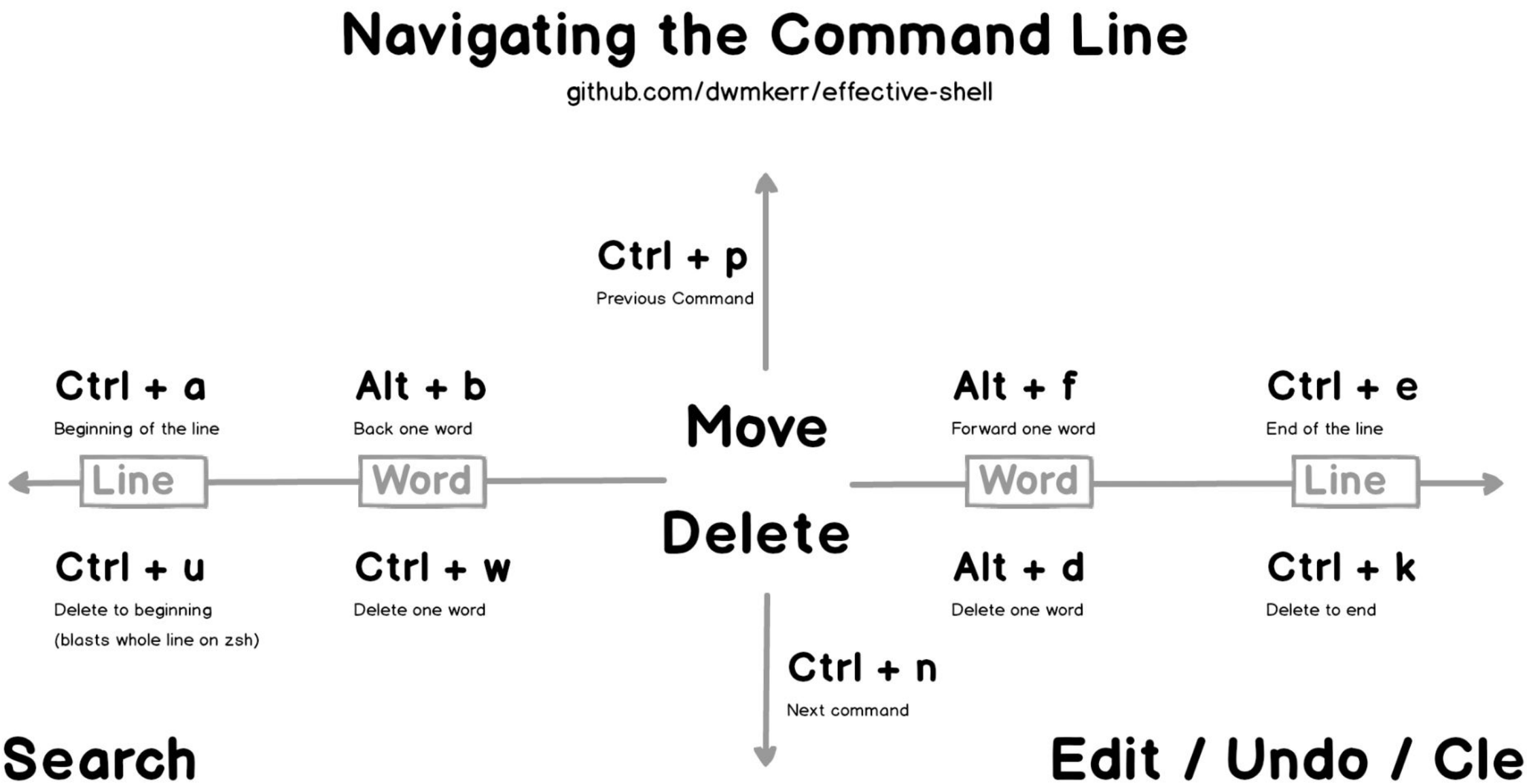
# Check which shell is running

- `ps -p $$` - Display your current shell name reliably
- `echo "$SHELL"` – Print the shell for the current user but not necessarily the shell that is running at the moment
- `echo $0` – Another reliable and simple method to get the current shell interpreter name on Linux or Unix-like systems
- `readlink /proc/$$/exe` – Another option to get the current shell name reliably on Linux operating systems
- `grep "^$USER" /etc/passwd` – Print the default shell name. The default shell runs when you open a terminal window

# Change shell

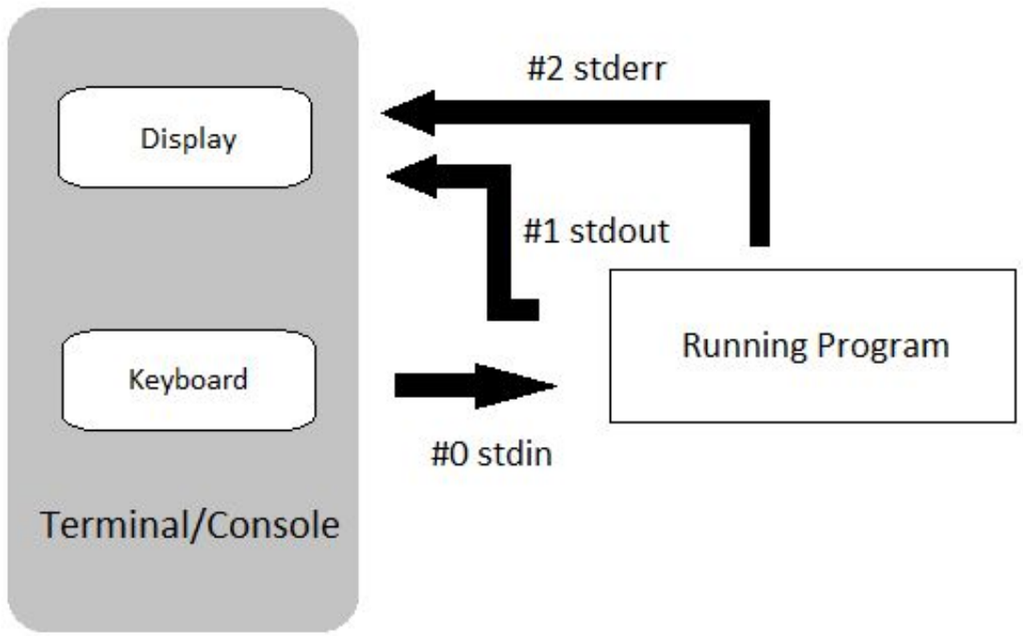
- `chsh -s /bin/zsh`

# Command editing



# Pipes and redirection

- Every process has at least three communication channels available to it:
  - standard input (**`STDIN`**) - **`0`**
  - standard output (**`STDOUT`**) - **`1`**
  - standard error (**`STDERR`**) - **`2`**




- They might connect to a terminal window, a file, a network connection, or a channel belonging to another process, etc.
- Each channel is named with a small integer called a **file descriptor**

# Pipes and redirection (conti.)

- Shell interprets the symbols `<`, `>`, and `>>` as instructions to reroute a command's input or output to or from a file
- To redirect both `STDOUT` and `STDERR` to the same place, use the `>&` symbol
- To redirect `STDERR` only, use `2>`

# Pipes and redirection (conti.)

- To connect the STDOUT of one command to the STDIN of another, use the `|` symbol, commonly known as a **`pipe`**
- To execute a second command only if its precursor completes successfully, you can separate the commands with an `&&` symbol
- The `||` symbol executes the following command only if the preceding command fails (that is, it produces a nonzero exit status)
- Use a **backslash** to break a command onto multiple lines
- Multiple commands combined onto one line—you can use a **semicolon** as a statement separator

# Pipes and redirection (conti.)

- `grep bash /etc/passwd > /tmp/bash-users`
- `sort < /tmp/bash-users`
- `find / -name core 2> /dev/null`
- `find / -name core > /tmp/corefiles 2> /dev/null`
- `find / -name core 2> /dev/null | less`
- `ps -ef | grep httpd`
- `cut -d: -f7 < /etc/passwd | sort -u`
- `mkdir foo && cd foo`
- `cp --preserve --recursive /etc/* /spare/backup \`

  `|| echo "Did NOT make backup"`

- `mkdir foo; cd foo; touch afile`

# Variables

- Variable names are **unmarked** in **assignments** but prefixed with a **dollar sign** (`$`) when their values are **referenced**

```
$ etcdir='/etc'
$ echo $etcdir
/etc
```

- When **referencing** a variable, surround its name with curly braces to clarify to the parser and to human readers where the variable name stops and other text begins; for example, `${etcdir}` instead of just `$etcdir`
- No standard convention for the naming of shell variables, but **all-caps names** typically suggest **environment variables** or variables read from global configuration files;
  - Often, local variables are all-lowercase with components separated by underscores
- Variable names are case sensitive

# Quoting

- Shell treats strings enclosed in single and double quotes similarly
- **Except: double-quoted** strings are subject to **globbing** (the expansion of filename-matching metacharacters such as * and ?) and **variable expansion**

```
$ mylang="Pennsylvania Dutch"
$ echo "I speak ${mylang}."
I speak Pennsylvania Dutch.
$ echo 'I speak ${mylang}.'
I speak ${mylang}.
```

# Quoting (conti.)

- **Backquotes**, also known as **backticks**, are treated similarly to double quotes
- **Additional effect**: executing the contents of the string as a shell command and replacing the string with the command's output

```
$ echo "There are `wc -l < /etc/passwd` lines in the passwd file."
There are 28 lines in the passwd file.
```

# Environment variables

- When a process starts up, it receives a list of command-line **arguments** and also a set of **environment variables**
- Show you the current environment: `printenv`
- By convention, environment variables have all-caps names
- `export varname` to promote a shell variable to an environment variable
- Commands for environment variables that you want to set up at login time should be included in your `~/.profile` or `~/.bash_profile` file

```
$ export EDITOR=nano
$ vipw
<starts the nano editor>
```

# Common filter commands

- `cut`: separate lines into fields
- `sort`: sort lines
- `uniq`: print unique lines
- `wc`: count lines, words, and characters
- `tee`: copy input to two places
- `head` and `tail`: read the beginning or end of a file
- `grep`: search its input text and prints the lines that match a given pattern

# sort

## sort options

| Opt | Meaning |
|---|---|
| -b | Ignore leading whitespace |
| -f | Sort case-insensitively |
| -h | Sort "human readable" numbers (e.g., 2MB) |
| -k | Specify the columns that form the sort key |
| -n | Compare fields as integer numbers |
| -r | Reverse sort order |
| -t | Set field separator (the default is whitespace) |
| -u | Output only unique records |

# sort (conti.)

- `sort -t: -k3 -n /etc/group`
- `sort -t: -k3 /etc/group`
- `du -sh /usr/* | sort -h`
- `du -sh /usr/* | sort -rh`

# uniq

- Similar in spirit to `sort -u`
- `-c` to count the number of instances of each line
- `-d` to show only duplicated lines
- `-u` to show only nonduplicated lines

E.g:

```
cut -d: -f7 /etc/passwd | sort | uniq -c
```

# wc

```
$ wc /etc/passwd
 32  77 2003 /etc/passwd
```

- Counting the number of lines, words, and characters in a file
- `-l` to count lines
- `-w` to count words
- `-c` to count characters

# tee

- Sends its standard input both to standard out and to a file that you specify on the command line

```
find / -name core 2>/dev/null | tee /dev/tty | wc -l
```

- The device `/dev/tty` is a synonym for the current terminal window

# head and tail

- Use `-n` numlines option to specify more or fewer
- `tail -f` waits for new lines to be added to the end of the file and prints them as they appear—great for monitoring log files

# grep

- Searches its input text and prints the lines that match a given **pattern**
- “Regular expressions” are text-matching patterns written in a standard and well-characterized pattern-matching language; universal standard

- `-c` to print a count of matching lines
- `-i` to ignore case when matching
- `-v` to print nonmatching
- `-l` (lower case L) to print only the names of matching files rather than printing each line that matches

```
$ sudo grep -l mdadm /var/log/*
/var/log/auth.log
/var/log/syslog.0
```

# grep (conti.)

- --line-buffered option to make sure you see each matching line as soon as it becomes available
- `tail -f /var/log/auth.log | grep --line-buffered root`

# sh scripting

# sh scripting

- `sh` is great for simple scripts that automate things you'd otherwise be typing on the command line
- `sh`-compatible shells often supplement this baseline with additional language features
- don't assume that the system's version of `sh` is always `bash`

# Execution

- `sh` comments start with a sharp (`#`) and continue to the end of the line
- Break a single logical line onto multiple physical lines by escaping the newline with a backslash (`\`)
- An `sh` script can consist of nothing but a series of command lines

**shebang** statement and declares the text file to be a script for interpretation by `/bin/sh`

```
#!/bin/sh
echo "Hello, world!"
```

# Execution (conti.)

- To prepare a script for running, just turn on its execute bit:

```
$ chmod +x helloworld
$ ./helloworld[10]
Hello, world!
```

- You can also invoke the shell as an interpreter directly:

```
$ sh helloworld
Hello, world!
$ source helloworld
Hello, world!
```

runs **helloworld** in a new instance of `sh`

makes your existing login shell read and execute the contents of the file

# From commands to scripts

For example, suppose you have log files named with the suffixes `.log` and `.LOG`

```
$ find . -name '*log'
.do-not-touch/important.log
admin.com-log/
foo.log
genius/spew.log
leather_flog
...
```

```
$ find . -type f -name '*.log'
.do-not-touch/important.log
foo.log
genius/spew.log
...
```

```
$ find . -type f -name '*.log' | grep -v .do-not-touch
foo.log
genius/spew.log
...
```

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname
> do
> echo mv $fname `echo $fname | sed s/.log/.LOG/`
> done
mv foo.log foo.LOG
mv genius/spew.log genius/spew.LOG
```

# From commands to scripts (conti.)

For example, suppose you have log files named with the suffixes `.log` and `.LOG`

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname;
    do echo mv $fname `echo $fname | sed s/.log/.LOG/`; done | sh -x
+ mv foo.log foo.LOG
+ mv genius/spew.log genius/spew.LOG
...
```

`-x` option to `sh` prints each command before executing it

# From commands to scripts (conti.)

To summarize this approach:

1. Develop the script (or script component )as a pipeline, one step at a time, entirely on the command line
2. Send output to standard output and check to be sure it looks right
3. At each step, use the shell's command history to recall pipelines and the shell's editing features to tweak them
4. Until the output looks right, you haven't actually done anything, so there's nothing to undo if the command is incorrect
5. Once the output is correct, execute the actual commands and verify that they worked as you intended
6. Use `fc` to capture your work, then clean it up and save it

# Input and output

- `printf` provides more control over your output

```
$ echo "\taa\tbb\tcc\n"
\taa\tbb\tcc\n
$ printf "\taa\tbb\tcc\n"
    aa   bb   cc
```

- `read` to prompt for input

```
#!/bin/sh

echo -n "Enter your name: "
read user_name

if [ -n "$user_name" ]; then
    echo "Hello $user_name!"
    exit 0
else
    echo "Greetings, nameless one!"
    exit 1
fi
```

`-n` in the `echo` command suppresses the usual newline

`-n` in the `if` statement evaluates to true if its string argument is **not null**

# Spaces in filenames

- Spaceful filenames can be quoted to keep their pieces together

```
$ less "My spacey file"
```

- or escape individual spaces with a backslash

```
$ less My\ spacey\ file
```

```
$ find /home -type f -size +1M -print0 | xargs -0 ls -l
```

# Command-line arguments and functions

- `$1` is the first command-line argument, `$2` is the second, and so on
- `$0` is the name by which the script was invoked
- `$#` contains the number of command-line arguments

```
#!/bin/sh

show_usage() {
    echo "Usage: $0 source_dir dest_dir" 1>&2
    exit 1
}

# Main program starts here

if [ $# -ne 2 ]; then
    show_usage
else # There are two arguments
    if [ -d $1 ]; then
        source_dir=$1
    else
        echo 'Invalid source directory' 1>&2
        show_usage
    fi
    if [ -d $2 ]; then
        dest_dir=$2
    else
        echo 'Invalid destination directory' 1>&2
        show_usage
    fi
fi

printf "Source directory is ${source_dir}\n"
printf "Destination directory is ${dest_dir}\n"
```

`1>&2` makes the output go to STDERR

# Control flow

`[]` syntax for comparisons; actually a shorthand way of invoking `test` and are not a syntactic requirement of the `if` statement

```
if [ $base -eq 1 ] && [ $dm -eq 1 ]; then
    installDMBase
elif [ $base -ne 1 ] && [ $dm -eq 1 ]; then
    installBase
elif [ $base -eq 1 ] && [ $dm -ne 1 ]; then
    installDM
else
    echo '==> Installing nothing'
fi
```

# Elementary sh comparison operators

| String | Numeric | True if |
|---|---|---|
| `x = y` | `x -eq y` | x is equal to y |
| `x != y` | `x -ne y` | x is not equal to y |
| `x <`[a] `y` | `x -lt y` | x is less than y |
| *n/a* | `x -le y` | x is less than or equal to y |
| `x >`[a] `y` | `x -gt y` | x is greater than y |
| *n/a* | `x -ge y` | x is greater than or equal to y |
| `-n x` | *n/a* | x is not null |
| `-z x` | *n/a* | x is null |

a. Must be backslash-escaped or double bracketed to prevent interpretation as an input or output redirection character.

# sh file evaluation operators

| Operator | True if |
|---|---|
| –d *file* | *file* exists and is a directory |
| –e *file* | *file* exists |
| –f *file* | *file* exists and is a regular file |
| –r *file* | User has read permission on *file* |
| –s *file* | *file* exists and is not empty |
| –w *file* | User has write permission on *file* |
| *file1* –nt *file2* | *file1* is newer than *file2* |
| *file1* –ot *file2* | *file1* is older than *file2* |

# case selection

```
# The log level is set in the global variable LOG_LEVEL. The choic
# are, from most to least severe, Error, Warning, Info, and Debug.

logMsg() {
    message_level=$1
    message_itself=$2
    if [ $message_level -le $LOG_LEVEL ]; then
        case $message_level in
            0) message_level_text="Error" ;;
            1) message_level_text="Warning" ;;
            2) message_level_text="Info" ;;
            3) message_level_text="Debug" ;;
            *) message_level_text="Other"
        esac
        echo "${message_level_text}: $message_itself"
    fi
}
```

# Loops

- `for...in` construct makes it easy to take some action for a group of **values** or **files**, especially when combined with **filename globbing** (the expansion of simple pattern-matching characters such as `*` and `?` to form filenames or lists of filenames)

```
#!/bin/sh

suffix=BACKUP--`date +%Y-%m-%d-%H%M`

for script in *.sh; do
    newname="$script.$suffix"
    echo "Copying $script to $newname..."
    cp -p $script $newname
done
```

# Loops (conti.)

- Any **whitespace-separated list** of things, including the contents of a variable. works as a target of `for...in`

```
for script in rhel.sh sles.sh; do
```

- Can also omit the list entirely, the loop implicitly iterates over the script's command-line arguments or the arguments passed to a function

```
#!/bin/sh

for file; do
    newname="${file}.backup"
    echo "Copying $file to $newname..."
    cp -p $file $newname
done
```

# while loop

- Useful for processing **command-line arguments** and for reading the **lines of a file**

```
#!/bin/sh

exec 0<$1
counter=1
while read line; do
    echo "$counter: $line"
    counter=$((counter + 1))
done
```

redefines the script's standard input to come from whatever file is named by the first command-line argument

`$((...))` notation forces numeric evaluation; works in the context of double quotes, too

# Arithmetic

- All `sh` variables are **string** valued, so `sh` does not distinguish between the number `1` and the character string `"1"` in assignments

```
#!/bin/sh

a=1
b=$((2))

c=$a+$b
d=$((a + b))

echo "$a + $b = $c \t(plus sign as string literal)"
echo "$a + $b = $d \t(plus sign as arithmetic addition)"
```

# Regular Expressions

# Regex

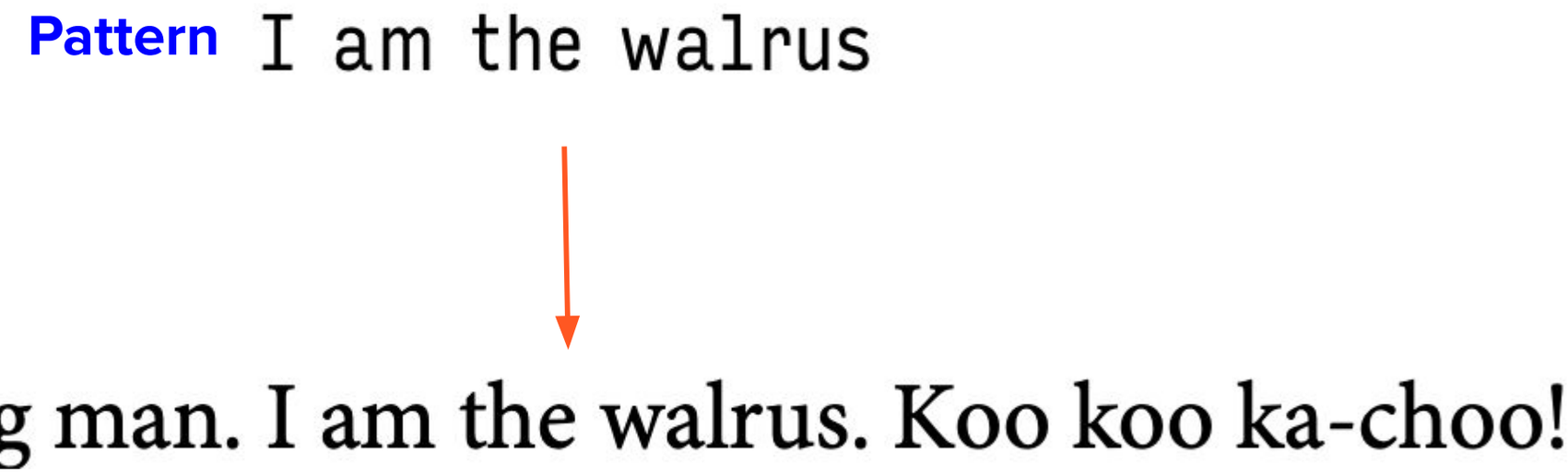
- Regular expressions are supported by most modern languages
- The filename matching and expansion performed by the shell **is not** a form of regular expression matching

# The matching process

- Code that evaluates a regular expression attempts to match a single given text string to a single given pattern
- For the matcher to declare success, the entire search pattern must match a contiguous section of the search text
- The evaluator returns the text of the match along with a list of matches

# Literal characters

- Matching is case sensitive



# Special characters

| Symbol | What it matches or does |
|---|---|
| . | Matches any character |
| [*chars*] | Matches any character from a given set |
| [^*chars*] | Matches any character not in a given set |
| ^ | Matches the beginning of a line |
| $ | Matches the end of a line |
| \w | Matches any "word" character (same as [A-Za-z0-9_]) |
| \s | Matches any whitespace character (same as [ \f\t\n\r])[a] |
| \d | Matches any digit (same as [0-9]) |
| \| | Matches either the element to its left or the one to its right |
| (*expr*) | Limits scope, groups elements, allows matches to be captured |
| ? | Allows zero or one match of the preceding element |
| * | Allows zero, one, or many matches of the preceding element |
| + | Allows one or more matches of the preceding element |
| { *n* } | Matches exactly *n* instances of the preceding element |
| { *min,* } | Matches at least *min* instances (note the comma) |
| { *min,max* } | Matches any number of instances from *min* to *max* |

a. That is, a space, a form feed, a tab, a newline, or a return

# Example regular expressions

- To match a regular zip code, you must match a five-digit number: `^\d{5}$`
- A five-digit zip code or an extended zip+4: `^\d{5}(-\d{4})?$`
- A classic demonstration of regex matching is the following expression:

```
M[ou]'?am+[ae]r ([AEae]l[- ])?[GKQ]h?[aeu]+([dtz][dhz]?){1,2}af[iy]
```

matches

- Muammar al-Kaddafi (BBC)
- Moammar Gadhafi (Associated Press)
- Muammar al-Qadhafi (Al-Jazeera)
- Mu'ammar Al-Qadhafi (U.S. Department of State)

# Captures

- When a match succeeds, every set of parentheses becomes a “capture group” that records the actual text that it matched
- The matches arrive in the same order as the opening parentheses

```
(I am the (walrus|egg man)\. ?){1,2}
```

matching the text

```
I am the egg man. I am the walrus.
```

there are two results, one for each set of parentheses:

```
I am the walrus.
walrus
```