Shell Introduction

Very basic shell is “sh”. I begin with it to emphasize fundamental understanding.

Docs in `man sh`. Hard to follow for beginners, hopefully much better after these notes.

Workplace uses feature-rich shells, e.g., bash (Ubuntu default), tcsh, fish, zsh. I will cover a bit of bash later.

Homework, test/exam questions specify which shell to code for, e.g., “write an `sh` script for this”, “write a `bash` script for that”.

For daily work you may use any shell.

Toy Example Script

`toyscript` demos many constructs. Designed behaviour:

For each command line argument: Consider as filename, and:

- If Python file (filename ends with .py), delete.
  Unless `dryrun` is set to a non-empty string, then don't delete, just print "would have deleted" filename.
  If `verbose` is set to a non-empty string, print "deleting" filename.
- If not, print "hello" and filename.
  "hello" can be changed by setting `msg`.
  If `verbose` is set to a non-empty string, print "not deleting" filename.

Usage example: `sh toyscript x.py y.c toyscript`

Comments

A comment begins with '#' and extends until end of line. Can be whole line or begin from middle of line.

```
# whole line comment
ls -l       # comment
```

echo: The Print Command

To print stuff to stdout:

```
echo xxx yyy zzz
```

By default has newline at the end. To omit:

```
echo -n xxx yyy zzz
```

echo: The Print Command

To print stuff to stdout:

```
echo xxx yyy zzz
```

By default has newline at the end. To omit:

```
echo -n xxx yyy zzz
```

What if you want 4 spaces between xxx and yyy?

This won't work. (Exercise: Why?)

```
echo xxx    yyy zzz
```

Solutions:

```
echo xxx\ \ \ \ yyy zzz
echo 'xxx    yyy zzz'
echo "xxx    yyy zzz"
echo xxx'    'yyy zzz
```

etc.

Variables

Type is string.

Set value: `var=value`

Tricky: No space around "="
Exercise: What if you add spaces? `var = value`

Read value: `$var` or `${var}` (two more, next slide)
(If uninit: get empty string.)

Why sometimes you must write `${var}`:

```
v=xxx
v0=yyy
echo $v0          # yyy
echo ${v}0        # xxx0
```

Variables

Suppose `v='Sales Receipt.txt'`

What if you do this: `ls $v`
Answer: 2 arguments “Sales”, “Receipt.txt”
Splitting happens after variable substitution.

Variables

Suppose `v='Sales Receipt.txt'`

What if you do this: `ls $v`
Answer: 2 arguments “Sales”, “Receipt.txt”
Splitting happens after variable substitution.

How to disable splitting: `ls "$v"`
Good idea to always write like that.

Can also: `echo "The file $v is OK"`
the whole string is 1 argument

Similarly for `${v}`:

```
ls "${v}"
echo "The file ${v} is OK"
```

Want your $ back?

If you want the string $v itself, not the variable:

```
echo \$v
echo '$v'
echo '$'v
echo "\$v"
```

Arithmetic Expressions

`$((` *expression* `))`

Examples:

```
x=$((4 + 1))
y=$(($x * 2))       # $((x * 2)) also OK
echo $(($y + 3))    # $((y + 3)) also OK
```

Command Substitution

Run a command, capture its stdout, insert output data in-place:
`$(` *command* `)`

The data is split into words.
`./print-args $(echo ' aaa  bbb  ccc ')`
⇒ 3 arguments, spaces stripped.

If inside double-quotes, not splitted.
`./print-args "$(echo ' aaa  bbb  ccc ')"`
⇒ 1 argument, spaces preserved.
But tricky details for newlines, not shown.

More use cases:
```
echo "Time: $(date)"
x="$(date)"
```

Shell Scripts

Put your commands in a file, call it “myscript” say. You can run it with

```
sh myscript
```

More savvy users go one step further:

- Put as first line: `#!/bin/sh`
- Set executable flag on the file:
  `chmod u+x myscript`
- Run it with `./myscript`

Example: `print-things`

Command Line Arguments: Positional Parameters

If I run your script with arguments:
`sh myscript foo bar xyz`

- `$#` is 3, number of arguments
- `$0` is myscript, name of script
- `$1` is foo
- `$2` is bar
- `$3` is xyz
- `$*` is foo bar xyz
  `"$*"` expands to one single word “foo bar xyz”
- `$@` is foo bar xyz
  `"$@"` expands to 3 words “foo”, “bar”, “xyz”

Demo: `print-3-args`

shift

Shift positional parameters. E.g., starting from the previous slide, one `shift` causes:

- `$#` is 2
- `$1` is bar
- `$2` is xyz
- `$*` is bar xyz
  "`$*`" expands to one single word “bar xyz”
- `$@` is “bar xyz”
  "`$@`" expands to 2 words “bar”, “xyz”

Demo: `print-args`

Empty-string argument and argument containing spaces:

```
sh print-args "" " " "hello world"
```

Command Grammar: “Simple” Commands

“simple command” = command name, arguments, optionally [file] redirection (next slide).

Example (without redirection): `tr -d 123`

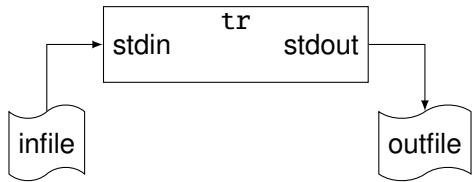
Command name has 4 cases, not apparent from syntax:

- Shell built-in command, e.g., ‘`cd`’
- Shell function (user-defined).
- Shell alias (user-defined) (not covered, but dead simple).
- Program, e.g., ‘`tr`’, ‘`./print-args`’

How to tell: The `type` command, e.g., `type cd`, `type tr`

[File] Redirection

```
tr -d 123 < infile > outfile
tr -d 123 0< infile 1> outfile
```



‘>’ erases and overwrites. To append: ‘>>’

Redirect stderr:
`command 2> file`

Redirect both stdout and stderr to the same file:
`command > file 2>&1`

Shell Grammar: Compound Commands Overview

Next slides explain constructs for compound commands.

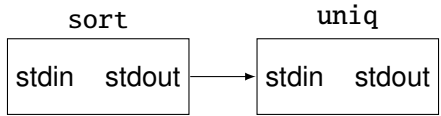
Operators from highest to lowest precedence:

description	operator
grouping	{ } ( )
redirection	< > >>
pipeline	|
not	!
and, or	&& ||
command list	; newline

Also if-then-else, loops.

Pipeline

E.g., `sort | uniq`



[Command] List—Sequential Composition

Multiple commands can be separated by newlines (especially in shell script files). Example:

```
cd B09
ls -l
cd ..
```

Or, a single line but separated by semicolons. Example:

```
cd B09 ; ls -l ; cd ..
```

Either way, known as “list” or “command list”, sequentially executed: wait for one to finish before running the next.

[Command] List—Sequential Composition

Multiple commands can be separated by newlines (especially in shell script files). Example:

```
cd B09
ls -l
cd ..
```

Or, a single line but separated by semicolons. Example:

```
cd B09 ; ls -l ; cd ..
```

Either way, known as “list” or “command list”, sequentially executed: wait for one to finish before running the next.

One command but you want to split into multiple lines: Need to escape the newlines:

```
echo hello B09 \
     students
```

Exit Code. True And False

Commands give an exit code when done.

Special variable `$?` is most recent command's exit code.

In C, recall "`int main(...)`". Return value is exit code!
Demo: `ret.c`

Interpreted as true/false by logical operators, if-then-else, while-loops. Often also success/error.
0 means true, success
non-0 means false, error, e.g.,

- File commands use 1 for file not found.
- String search programs use 1 for string not found.

Exercise: Run `echo $?` twice. Second time prints 0. Why?

Logical AND, OR, NOT, true, false

`mkdir foo && cp myfile foo`
Sequential execution, but stop upon first false.

`mkdir foo1 || mkdir foo2 || mkdir foo3`
Sequential execution, but stop upon first true.

(So they are short-circuiting.)

`! mkdir foo`
Logical not: turn 0 to 1, non-0 to 0.

Operator precedence:
'&&', '||' same precedence (tricky!)
both lower than '!'

`true`: Always true.
`false`: Always false.

Test Commands

A whole suite of “[expression]” commands for useful tests giving exit codes for booleans.

File tests (more on sh man page, search for “test expression”):

- `[ -e path ]`: exists
- `[ -f path ]`: exists and regular file
- `[ -d path ]`: exists and directory
- `[ -r path ]`: exists and readable
- `[ -w path ]`: exists and writable
- `[ -x path ]`: exists and executable
- `[ path1 -nt path2 ]`: both exist and path1 is newer
- `[ path1 -ot path2 ]`: both exist and path1 is older

Test Commands

String comparisons:

- `[ s1 = s2 ]`: string equality
  Also `!=`, `<`, `>` (need escaping/quoting)
- `[ -n string ]`: string not empty
- `[ -z string ]`: string empty

Recall `$v` vs `"$v"`. You want:

```
[ "$v" = xxx ]
[ -n "$v" ]
[ -z "$v" ]
```

Number comparisons (parsing strings to numbers):

- `[ n1 -eq n2 ]`: integer equality
  Also `-ne`, `-gt`, `-ge`, `-lt`, `-le`

Test Commands

Logical connectives, by example:

- `[ ! -e path ]`: not
- `[ "$x" -eq 5 -a "$y" -eq 6 ]`: and
- `[ "$x" -eq 5 -o "$y" -eq 6 ]`: or

`-a` higher precedence than `-o`

Parentheses also supported, but need escaping or quoting.

```
[ -d dir1 -a '(' -d dir2 -o -d dir3 ')' ]
```

Test Commands

Why need quoting (or backslahses) and spaces in
`[ ! '(' "$x" '>' "$y" ')' ]`
and why these are misinterpreted
`['!'('"$x"'>'"$y"')']`
`[ ! ( "$x" > "$y" ) ]`

- Command name is `[`
- Expression represented as arguments.
  One argument per operand/operator, separately.
- Last argument must be `]`
- Grammar clashes with shell grammar. Need quoting to tell shell “not for you; pass-thru to the command”.

Grouping 1/2

When operator precedence doesn't work for you, write

```
{ list ; }
```

for explicit grouping. (Recall "[command] list".)

Example:

```
{ grep foo file1 ; ls ; } > file2
```

Again, may use newline instead of `;`

Easy to miss: This looks right but is wrong, tricky!

```
{ grep foo file1 ; ls } > file2
```

Missing one last newline or `;` before `}`

Grouping 2/2: Subshell

`( )` also does grouping, plus one more thing.

```
( list ; )
```

Difference from `{}` by example:

```
{ x=hello ; cd / ; }
```
Effects retained afterwards. Faster.

```
( x=hello ; cd / ; )
```
Effects lost afterwards. Slower, in fact new shell process.
Hence known as “run in subshell”.

Operators Summary And Precedence

From highest to lowest precedence:

description	operator
grouping	{} ()
redirection	< > >>
pipeline	|
not	!
and, or	&& ||
command list	; newline

Conditional Branching

Demo: `if-demo`

```
if list1 ; then
    list2
elif list3 ; then
    list4
else
   list5
fi
```

Easy to miss:

- before "`then`", need `;` or newline
- "`elif`", not "`else if`"

Exercise: "`else if`" is not wrong, but what is annoying about it?

While-Loop

Demo: `while-demo`, `print-args`

```
while list1 ; do
    list2
done
```

```
while list1 ; do list2 ; done
```

May use ‘`break`’ and ‘`continue`’.

Easy to miss: before “`do`”, need `;` or newline.
These look right but are wrong:

```
while list1 do
    list2
done
```

```
while list1 ; do list2 done
```

Test Commands in if/while

```
if [ $x = $y ] ; then
  ...
fi

while [ $x = $y ] ; do
  ...
done
```

Easy to miss: Still need `;` or newline, even though `]` ends the condition. `]` is the last argument of the test command.

For-loop

Demo: `for-demo`

```
for var in word1 word2 ... ; do
    list
done
```

Use `$var` to read the variable.

May use ‘`break`’ and ‘`continue`’.

Easy to miss: Need `;` or newline before `do` to mark end of words. Lest computer thinks your `do` is one of the words, like above.

for i=0 to 3

Integer range is delegated to the `seq` program.

`seq 0 3` outputs 0 to 3 to stdout.

Use command substitution to capture, give to for-loop.

```
for i in $(seq 0 3) ; do ... ; done
```

Demo: `for-demo`

See `seq --help` or `man seq` for variations.

Patterns (to match filenames)

- '*' matches any [part of] filename not containing /
  Example: `ls a2/*.py`
  All python files in directory a2
- '?' matches one character
- '`[ace]`' matches "a" or "c" or "e"
- '`[0-9]`' matches a digit
- '`[!0-9]`' matches a non-digit

Important: Expands to multiple pathnames before handing to command. `ls` never saw "a2/*.py"; it saw "a2/foo.py", "a2/bar.py", etc.

Important: If no match, the pattern stays as itself.

Good for for-loops too:

```
for i in *.py ; do echo $i ; done
```

Patterns (to match your string)

Pattern matching but on the string you want.

```
f=$1
case "$f" in
    *.py)
        echo "$f is pseudo-codoe"
        ;;
    *.c | *.sh | myscript)
        echo "$f is real code"
        ;;
    *)
        echo "$f is meh"
esac
```

(case-demo)

Exit

Command ‘`exit`’ terminates the whole shell script and the shell process.

Not required if your script just runs from start to finish normally.

But handy for:

Early termination (even inside loops, functions, etc.)

Controlling exit code, e.g., ‘`exit 1`’.

(Default exit code is whatever the last executed command gives.)

getopts: General Option Processing

Shell built-in `getopts` helps pick out those `-n`, `-v` options.

Suppose I want to support these options:

- `-M` followed by a string
- `-n`
- `-v`

and after options, arbitrarily many non-option arguments.

I also need to choose a variable name. I choose `myflag`.

Then I use one of these (they're equivalent):

```
getopts M:nv myflag
getopts vM:n myflag
getopts nvM: myflag
```

getopts Sample Run 1

If user runs my script (code: `getopts-trace`) with

```
sh getopts-trace -n -v -Mfoo -v -M bar abc def -n xyz
```

then when I call `getopts M:nv myflag` the *i*th time:

i	`$myflag`	`$OPTARG`	`$OPTIND`	exit code
1	n	(empty)	2	0
2	v	(empty)	3	0
3	M	foo	4	0
4	v	(empty)	5	0
5	M	bar	7	0
6	?	bar	7	1

Note that `$7` is abc, 1st non-option argument. I can use `shift 6` to get rid of options.

`getopts` does not pick out options after seeing 1st non-option argument.

getopts Sample Run 2

User can add `--` to explicitly mark end of options:

```
sh getopts-trace -n -v -Mfoo -v -M bar -- abc def -n xyz
```

then when I call `getopts M:nv myflag` the *i*th time:

i	`$myflag`	`$OPTARG`	`$OPTIND`	exit code
1	n	(empty)	2	0
2	v	(empty)	3	0
3	M	foo	4	0
4	v	(empty)	5	0
5	M	bar	7	0
6	?	bar	8	1

Note that `$8` is abc, 1st non-option argument. I can use `shift 7` to get rid of options.

`getopts` honours using `--` to mean end of options.

getopts Sample Run 3

If user gives unsupported option, e.g., `-k`:

```
sh getopts-trace -n -v -Mfoo -k -M bar abc def -n xyz
```

then when I call `getopts M:nv myflag` the *i*th time:

i	`$myflag`	`$OPTARG`	`$OPTIND`	exit code
1	n	(empty)	2	0
2	v	(empty)	3	0
3	M	foo	4	0
4	?	(empty)	5	0
	and “Illegal option -k” to stderr			
5	M	bar	7	0
6	?	bar	7	1

getopts Demo

`toyscript-opts` is like `toyscript` but uses `getopts` to set variables according to options.

`-h`: Print usage summary.
`-n`: Set `dryrun` to non-empty.
`-v`: Set `verbose` to non-empty
`-M` *foo*: Set `msg` to *foo*.

Usage example:

```
sh toyscript-opts -v -M welcome x.py y.c toyscript-opts
```

Functions

Example function definition:

```
myfunction() {
  echo "$1"
  echo "$@"
}
```

Example function call: `myfunction foo bar xyz`

Arguments are positional parameters e.g. `$1`, `$@`.

Exercise: May use `getopts`, but what should you reset first?

May return from function early, or specify return value, with `return` or e.g. `return 1`.

(Default return value is exit code of last executed command.)

Escaping And Quoting

Recall special-meaning characters in shell syntax:
< * $ # (& | ; space newline (and more)

Use escaping or quoting to get the character itself.

Example: print “<*; #” (2 spaces in between):

```
echo \<\*\;\ \ \#
echo '<*;  #'
echo "<*;  #"
```

Note: So ‘\’ is also special! Use ‘\\’ for backslash itself.

Example: store that string in a variable:

```
v='<*;  #'
```

Example: Many use cases of [:

```
[ "$v" '<' "$w" ]
```

Variables in Double Quotes

Common mistake when checking whether `$v` is non-empty:
`[ -n $v ]`
No!

- If `$v` is empty, shell sees `[ -n ]`, which makes no sense.
- If `$v` is purely spaces, shell still sees `[ -n ]`
- If `$v` is "x y", shell sees `[ -n x y ]`, which makes no sense.

Solution: `[ -n "$v" ]`

Exercise: Older generation used
`[ x != x$v ]`
When does it work? When does it break?

More Fun with echo

Why do I need $4n$ backslashes to get `echo` to print n backslashes?

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

(BTW: Odd number $\Rightarrow$ last backslash escapes newline, shell thinks I am splitting my command into two lines.)

If I use quoting, I still need $2n$ backslashes:

```
$ echo '\\\\\\'
\\\
```

More Fun with echo

Use `print-args-c.c` to verify how many backslashes actually seen by command.

```
$ ./print-args-c \\\\\\\\\\\\
argc = 2
argv[0] = "./print-args-c"
argv[1] = "\\\\\\"
```

No surprise, shell said it would translate 2 backslashes to 1.

```
$ ./print-args-c '\\\\\\'
argc = 2
argv[0] = "./print-args-c"
argv[1] = "\\\\\\"
```

No surprise, quoting works.

Oh so `echo` adds its own translation. . .

More Fun with echo

`sh` man page: `echo` also interprets backslash:

`\n`	newline
`\t`	tab
`\\`	1 backslash

etc.

More Fun with echo

`sh` man page: `echo` also interprets backslash:

`\n`	newline
`\t`	tab
`\\`	1 backslash

etc.

Moral of the story:

What you see is never what you get.

It's telephone games all the way down.

It's lasagna all the way down.

Unless you prefer desserts, in which case:
It's baklava all the way down.

Dot command: Execute stuff in current shell

This reads commands from cmds.sh, executes them in current shell:

```
. ./cmds.sh
```

The command name is a single dot "`.`"

Contrast: `sh cmds.sh` runs in newly spawned shell process.

Use case: If cmds.sh defines functions, set variables, or uses `cd`, then

- `. ./cmds.sh` does them in current shell.
- `sh cmds.sh` does them in new shell process, which then quits, much ado about nothing.
  `./cmds.sh` ditto.

Demo: `dot-demo`

Here Document

To feed multi-line hardcoded text into stdin of a command:

```
cat << EOF
Hello I'm Albert.
You can use variables too
E.g., \$x=$x
EOF
```

"EOF" is not a reserved word, you may choose any word to mark begnning and end. Just don't clash with your actual text!

Code files: here-doc, here-python

Here Document: One More Thing

If you declare your end-marker in quotes:

```
cat << 'EOF'
Hello I'm Albert.
Now $x is $x
EOF

cat << "EOF"
Hello I'm Albert.
And $x is still $x
EOF
```

then $ is no longer special.

Code file: here-doc

Environment Variables

Every process (shell or otherwise) has a collection of “environment variables”, as part of process state.

Names are strings, values are strings too. Convention: Names in all caps, e.g., PATH, HOME, TZ, LC_ALL (these are standard Unix ones), CLASSPATH (specific to Java).

Initialized by copying from launcher (done by kernel): If *p* launches *q*, *q* gets a copy of *p*’s. But independently changeable otherwise.

Program ‘`printenv`’ prints the environment variables you currently have. It works because at startup it gets a copy of yours! Now it just has to print what *it* has.

(In C programs: `getenv()`, `setenv()`, `unsetenv()`.)

Environment Variables in Shell

Shell *downplays* difference between shell variables and environment variables. Only convention: shell variable names are in lowercase.

Both read by same syntax: `$x`, `$LC_ALL`

Both changeable by same syntax:
`x=C`
`LC_ALL=C`

Both erasable by same ‘`unset`’ command.

How to mark a variable as environment variable:

```
export MYENVVAR=foo
```

or two commands:

```
MYENVVAR=foo
export MYENVVAR
```

Environment Variables in Shell

To run a program but give it different environment variables (existing or new) without changing your own:

```
LC_ALL=C MYNEWENV=foo printenv
```

This is why the following two commands mean different things:

```
x='foo bar'
x=foo bar
```

Some Standard Environment Variables

HOME: Home directory.

TZ: User timezone preference. (Can be absent.)

PATH: Colon-separated list of directories. Searched when you launch a program, if program name does not contain any slash.

Example: Assume
`PATH=/usr/local/cms/eclipse:/usr/bin`

`eclipse` found in `/usr/local/cms/eclipse`

`printenv` found in `/usr/bin`

Some Bash Features

Docs in `man bash` and Bash Ref. Manual.

Bash Feature: Local Variables in Functions

Sh has global variables only.
Bash supports local variables in functions. Use `local`

```
myfunc() {
  local x y=hello  # x,y local, y inited
  x=hi
  echo "$x" "$y"
}
```

Demo: bash-local-demo

But dynamic scoping, not lexical scoping. See demo.

Bash Feature: Arrays

```
crew=(kermit piggy fozzie)  # set
crew[3]='sam eagle'         # set by index
echo "${crew[1]}"           # get by index
crew+=(gonzo 'dr pepper')   # append
echo ${#crew[@]}            # number of elements
for c in "${crew[@]}"; do   # all elements, like $@
  ...
done
# no prepend feature, but you can always do:
crew=(scooter "${crew[@]}")
```

Demo: bash-array-demo

Bash Feature: Associative Arrays

Key-value dictionary. “Array” but string indexes.

```
declare -A mark
mark=([denise]=4 [bob]=9)           # preload
mark[charles]=3                     # insert one
mark+=([bob]=7 [alice]=5)           # insert many
for k in "${!mark[@]}"; do          # all keys
  echo "$k has ${mark[$k]} marks"   # lookup
done
```

`declare -A` required, lest bash assumes integer-indexed array.

Demo: `bash-array-demo`

Bash Feature: Process Substitution

Pipelining (`x | y | ...`) limitation: only 1 input src, only 1 output dst.

Bash's process substitution generalizes to multiple srcs and dsts.

Example (2 input srcs): `sort <(cmd1) <(cmd2)`

Behaviour: Two input srcs to `sort`: stdout of `cmd1`, stdout of `cmd2`.

Oversimplified theory:

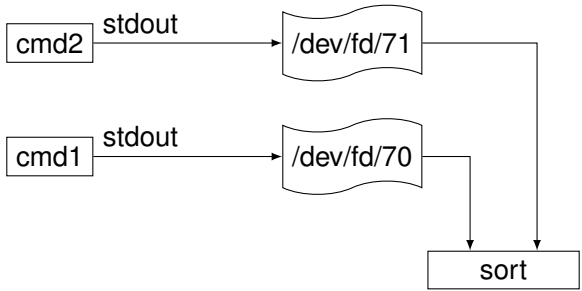
- Bash makes 2 fake files, say /dev/fd/70, /dev/fd/71.
- Bash runs these 3 commands in parallel:

```
cmd1 > /dev/fd/70
cmd2 > /dev/fd/71
sort /dev/fd/70 /dev/fd/71
```

Demo: `bash-procsub-demo`

Process Substition Example Picture

```
sort <(cmd1) <(cmd2)
```



Process Substition Example: Output

Example (1 output dst): `foo >(cmd1)`

Theory:

- Bash makes a fake file, say /dev/fd/70.
- Bash runs these 2 commands in parallel:
  `foo /dev/fd/70`
  `cmd1 < /dev/fd/70`

Picture: