

CYBR 4423

Linux/Unix Administration

BASH SCRIPTING

Overview

Bash scripting language basics

- Script files

- Language elements

 - Variable

 - Operators

 - Control flow

 - Function

Scripting apps

- Text and number processing

- File processing

Scripting

What is a script?

Why should system administrators know how to write scripts?

Script File

A set of commands grouped together in a file

Bash script file

.sh (suffix not required)

Add “#!/bin/bash” as a directive in the first line

```
#!/bin/bash  
var1="morning" #this is a variable  
echo "Good $var1"
```

One statement per line

; is optional

Use ; to separate statements if multiple commands are on the same line

Use # for comments

Variables

Variable naming

Variable names are **case sensitive**

All-caps names typically suggest environment variables or variables read from global configuration files

Local variables are all-lowercase with components separated by underscores

```
var1="today"
```

! **NO** space around the = symbol

Assignment and data type

All variables are of the string data type when assigned

```
var1=1000;  
printf "User input: $var1\n"
```

Referencing variables

Use the "\$"+variable name

Use {} around variable name optionally

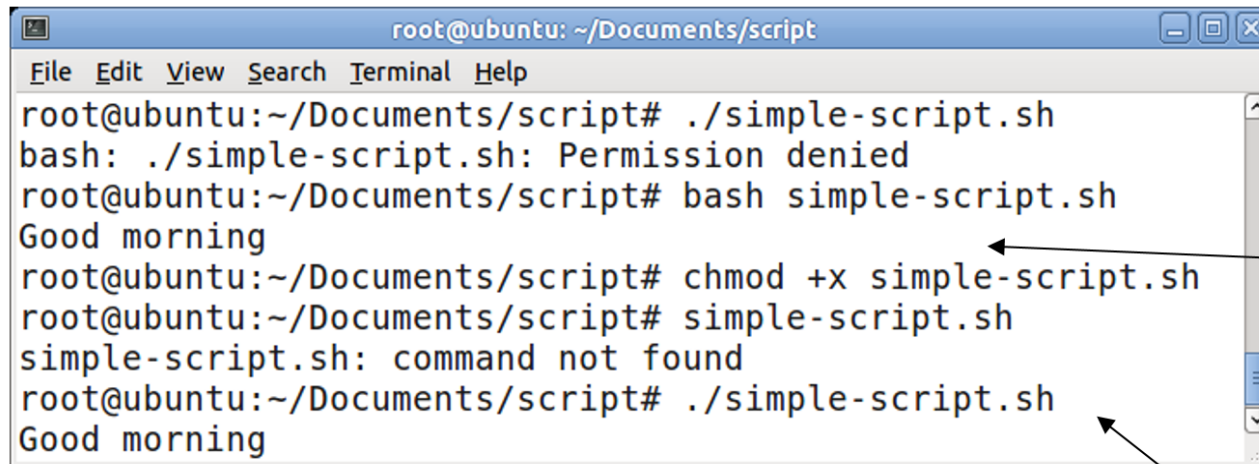
Script Execution Methods

Use the bash process to execute scripts

`bash [script file name]`

Directly run a script file as a separate process

Use “`chmod +x`” to allow the execution permission



```
root@ubuntu: ~/Documents/script
File Edit View Search Terminal Help
root@ubuntu:~/Documents/script# ./simple-script.sh
bash: ./simple-script.sh: Permission denied
root@ubuntu:~/Documents/script# bash simple-script.sh
Good morning
root@ubuntu:~/Documents/script# chmod +x simple-script.sh
root@ubuntu:~/Documents/script# simple-script.sh
simple-script.sh: command not found
root@ubuntu:~/Documents/script# ./simple-script.sh
Good morning
```

Initially, the file does not have execution permission. Use "bash" to execute it or add execution permission

!! For security reasons, the current directory is not in the search path. Add `./` to force search the current directory

Script Arguments

Script arguments are supplied at the command line after command name and separated by a space

Arguments are stored in order as \$1, \$2

\$0 refers to the command/script name

\$# is the number of arguments

```
#!/bin/bash
echo "Number of arguments: $#"
```

```
root@ubuntu: ~/Documents/script
File Edit View Search Terminal Help
root@ubuntu:~/Documents/script# ./script-args.sh
Number of arguments: 0
./script-args.sh

root@ubuntu:~/Documents/script# ./script-args.sh it4423 linux
Number of arguments: 2
./script-args.sh
it4423
linux
root@ubuntu:~/Documents/script#
```

Arithmetic Operations

Basic operators

+, -, *, /, %

++, --, +=, etc.

Arithmetic operations must be enforced by using

let

\$[...]

\$((...))

expr or bc command

More information

Pay attention to when to use \$ and when not to

```
#!/bin/bash
```

```
var1=5
```

```
var2=10
```

```
var3=var1+var2
```

```
echo "$var1+$var2=$var3"
```

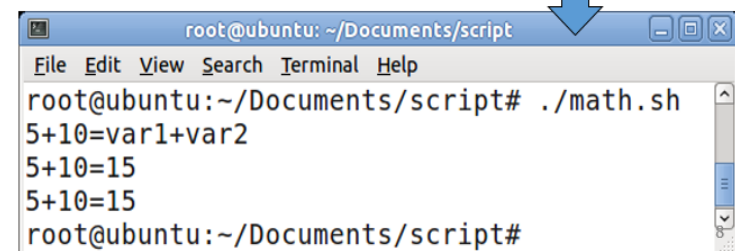
```
let var3=var1+var2
```

```
echo "$var1+$var2=$var3"
```

```
var3=$((var1+var2))
```

```
echo "$var1+$var2=$var3"
```

This is not a calculation in bash!!



```
root@ubuntu: ~/Documents/script
File Edit View Search Terminal Help
root@ubuntu:~/Documents/script# ./math.sh
5+10=var1+var2
5+10=15
5+10=15
root@ubuntu:~/Documents/script#
```


Comparison Operations

Table 2.2 Elementary bash comparison operators

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

The operators are different to compare numbers and strings.

Use \ before > and <

!!
Leave a space around =
otherwise it is variable
assignment

These are extremely
useful for file and
directory operations

Table 2.3 bash file evaluation operators

Operator	True if
<code>-d file</code>	<i>file</i> exists and is a directory
<code>-e file</code>	<i>file</i> exists
<code>-f file</code>	<i>file</i> exists and is a regular file
<code>-r file</code>	You have read permission on <i>file</i>
<code>-s file</code>	<i>file</i> exists and is not empty
<code>-w file</code>	You have write permission on <i>file</i>
<code>file1 -nt file2</code>	<i>file1</i> is newer than <i>file2</i>
<code>file1 -ot file2</code>	<i>file1</i> is older than <i>file2</i>

Flow Control – If

- Basic structure

```
if [ ... ]; then
...
elif [ ... ]; then
...
else
...
fi
```

!!
Leave a space
after [and
before]

Use [] for
test
conditions

```
#!/bin/bash

var1=5
var2=5

if [ $var1 = $var2 ]; then
    echo "Same"
else
    echo "Different"
fi
```

!!
Leave a space
before and
after =

Nested If

```
#!/bin/bash
```

```
if [ $# -eq 1 ]  
then
```

Check
number of
arguments

This script checks if
the provided filename
exists and is a file or a
directory.

!!
Leave a space
after [and
before]

```
    if [ -d $1 ]; then  
        echo "$1 exists and is a directory!"  
    elif [ -f $1 ]; then  
        echo "$1 exists and is a file!"  
    else  
        echo "$1 does not exist!"  
    fi  
else  
    if [ -d $PWD ]; then  
        echo "$PWD is a directory"  
    fi  
fi
```

Compound Conditions

Use `[[...]]` to evaluate compound conditions

`&&`: logical AND

`||`: logical OR

Other ways

`[...] && [...]`

`[...] || [...]`

`[... -a ...]`

`[... -o ...]`

```
#!/bin/bash
```

```
var1=5
```

```
var2=10
```

```
if [[ $var1 -lt 10 && $var1 -gt 4 ]]; then  
    echo "Within range"
```

```
else  
    echo "Out of range"
```

```
fi
```

```
if [[ $var2 -eq 10 || $var2 -eq 20 ]]; then  
    echo "Multiples of 10"
```

```
else  
    echo "Not multiples of 10"
```

```
fi
```

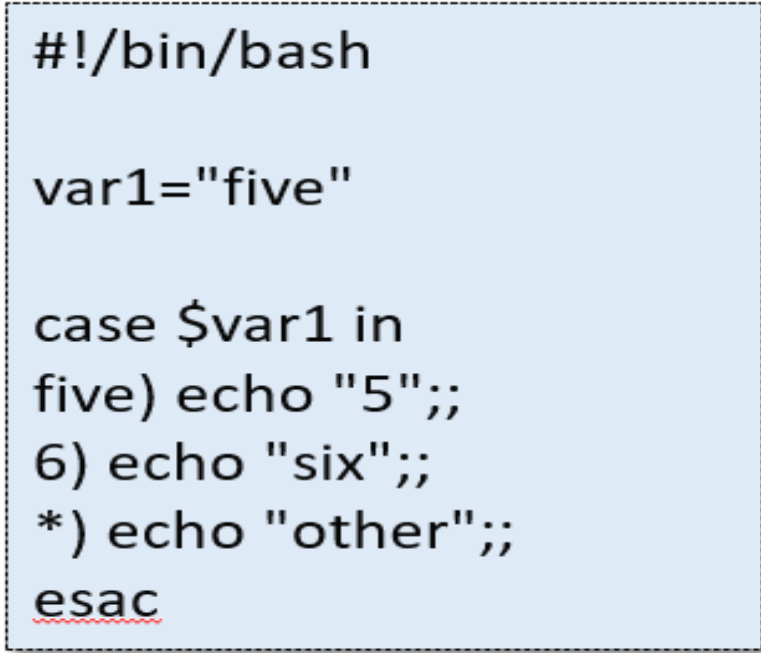
!!
Leave a space
after `[[` and
before `]]`



Flow Control – Case

Basic structure

```
case $variable in  
    condition1) ...;;  
    condition2) ...;;  
    *) ... ;;  
esac
```



```
#!/bin/bash  
  
var1="five"  
  
case $var1 in  
    five) echo "5";;  
    6) echo "six";;  
    *) echo "other";;  
esac
```

The case structure is very flexible on cases; case values can be numbers and strings – in fact, they are all strings.

Flow Control – For Loop

Basic structure

```
for (( i=0 ; i < $limit ; i++ ))  
do  
    ...  
done
```

No need to use
\$ for the control
variable

```
#!/bin/bash  
  
for (( i=0 ; i < 3 ; i++ ));  
do  
    echo "$i"  
done
```

Flow Control – For ... In Loop

Basic structure

```
for var1 in ... #arguments separated by space
do
    ...
done
```

```
#!/bin/bash

for var1 in 1 2 3 4
do
    echo "Number: $var1"
done
```

Flow Control – While Loop

Basic structure

```
counter=0
while [ loop condition ]
do
    ...
    $((counter++))
done
```

!!
Leave a space after [and before]

```
#!/bin/bash

counter=0
while [ $counter -lt 10 ]
do
    echo "Count: $counter"
    let counter++
done
```


Flow Control – Until Loop

Basic structure

```
counter=0
until [ loop stop condition ]
do
    ...
    ${counter++}
done
```

!!
Leave a space after [
and before]

```
#!/bin/bash

counter=0
until [ $counter -ge 10 ]
do
    echo "Count: $counter"
    let counter++
done
```

Functions and Parameters

Basics

Functions in bash do not return values

Functions must be declared before they can be called

Parameters do not need to be declared; they are referred to as \$1, \$2, etc. in the function

```
#!/bin/bash
function add_int
{
    echo "Add numbers:" ${1+$2};
}
get_diff()
{
    let r=${1-$2}
}
var1=8
var2=5
add_int var1 var2
get_diff var1 var2
echo "Difference: " $r
```

Function parameters

An alternative way to define a function

Pass parameters

A workaround to have functions return values

Summary

Key terms

Scripting mode

Script file

Argument

Function

Bash operators and symbols

Arithmetical operations: `$((...))` `$[...]` `let` `+` `-` `*` `/` `**` `%`

Comparison operators: `-eq` `-ne` `-gt` `-lt` `-d` `-e` `-f` `&&` `||`

Other operators: `;` `\` `#` `$`

Control flow structures: `if`, `case`, `for`, `for...in`, `while`, `until`
function, `{ }`, argument (`$#`, `$1`, `$2`, etc.)

Shell Initiation Files

Initiation files are script files that are executed when the system starts

System wide

/etc/profile

/etc/bashrc

User specific

~/.bash_profile

~/.bashrc

More reference

[Shell Initialization Files](#)



Good Readings and Resources

Quick guides to write scripts using the bash shell

[A quick guide to writing scripts using the bash shell](#)

[Shell Programming](#)

[Introduction to Linux Shell and Shell Scripting](#)

[Bash Script Examples](#)

References for further advanced study

[The bash shell](#)

[Bash Scripting Tutorial](#)

[Linux Shell Scripting Tutorial](#)

[Bash Guide for Beginners](#)

[Bash Programming – Introduction How-To](#)

[Advanced Bash-Scripting Guide](#)



Overview

Shell

- Shell builtin commands
- Command line scripting

Bash scripting basics

- Input/output
- Piping
- Variables and data types

Shell

A Unix/Linux shell is a piece of software (command-line interpreter) that provides an interface for users to access the services of the OS kernel

Major shells

Bourne shell compatibles

Bourne shell: /bin/sh

Bash (Bourne-Again shell): /bin/bash

C Shell (csh)

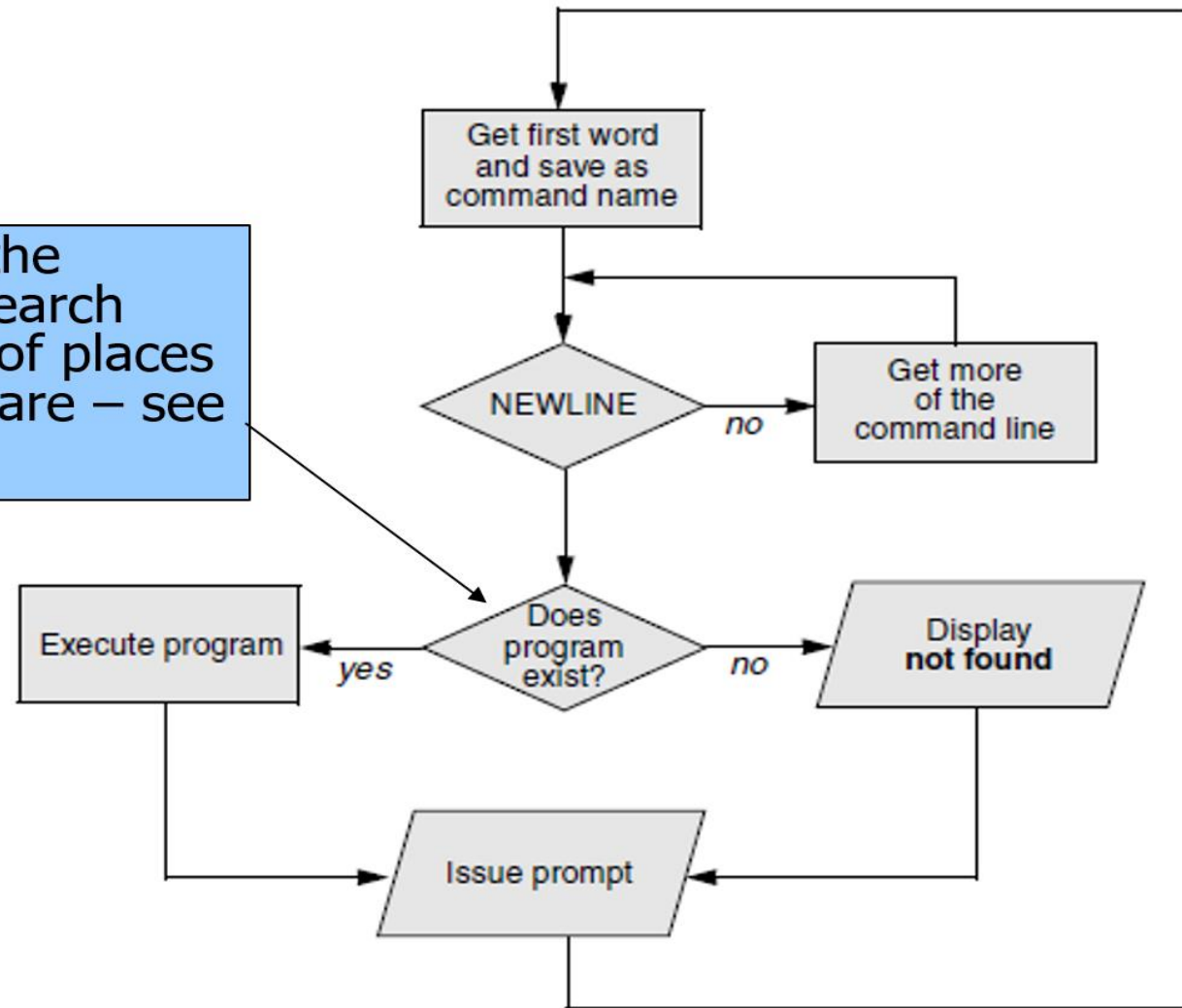
See the /etc/shells file for valid shells (not installed shells)

Ubuntu (and most Linux distros) uses bash as the default shell

You can switch to a different shell at any time use the "chsh" command

Command Execution

Shell will look for the command from "Search Path" – a number of places where commands are – see slides later



Get Command Reference

man

View command manuals (press "q" to exit)

type

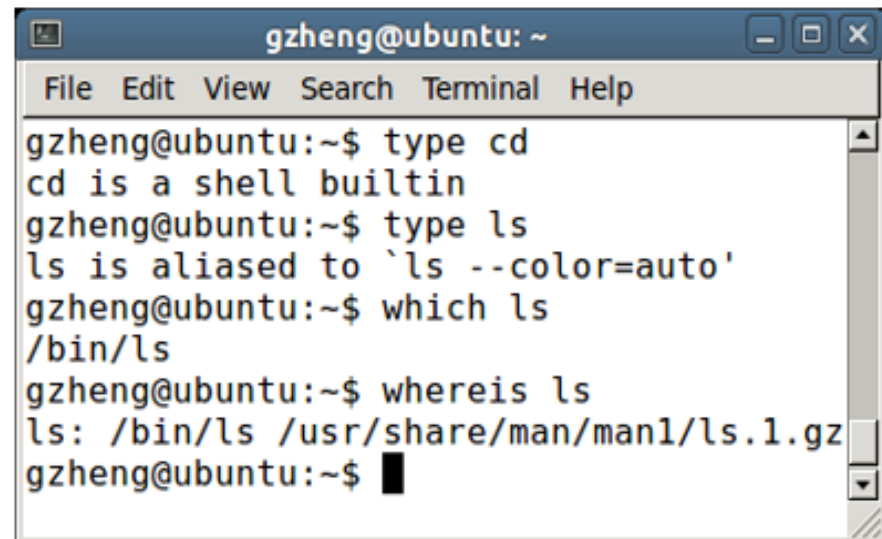
Show the type of a command

which

Show the file location of a command.

whereis

Locate binary, source, manual, configuration, and other files related to a command.



```
gzheng@ubuntu: ~  
File Edit View Search Terminal Help  
gzheng@ubuntu:~$ type cd  
cd is a shell builtin  
gzheng@ubuntu:~$ type ls  
ls is aliased to `ls --color=auto'  
gzheng@ubuntu:~$ which ls  
/bin/ls  
gzheng@ubuntu:~$ whereis ls  
ls: /bin/ls /usr/share/man/man1/ls.1.gz  
gzheng@ubuntu:~$
```

Basic Command Operations

Auto completion: use tab key

No need to type all characters!

This applies to commands and directory/file names

Command history

Use up/down arrow key to navigate through commands entered before.

Use "history" command and "!" operator

[Linux C and history command](#)

[Linux Command Line History](#)

Command editing

Use left/right arrow keys, del, backspace to edit a command

Scripting

Scripting vs. Programming

Scripting language in Linux

Shell (Bash)

Perl

Python

Two scripting modes

Command line scripting

Script file execution

Command Line Scripting

Write scripts directly in the shell at the command prompt

Command editing

Use up arrow key to get previous commands

Ctrl + A to go to the beginning of the command

Ctrl + E to go to the end of the command

Multi line commands

Use “\” to indicate a soft return and continue on the next line

Multiple commands on one line

Use “;” to separate commands

Shell Built-in Commands

Shell built-ins are commands interpreted by the shell directly (no separate executable files). Examples:

- cd
- pwd
- type
- echo
- alias

Use "type" command to see if a command is a builtin

[Reference](#)

Command Alias

An alias is a (usually short) name that the shell translates into another (usually longer) name or (complex) command.

Aliases allow you to define new commands.

Example

Enter "alias" without any argument to check the current aliases defined

```
#> alias ls='ls -l'  
#> alias cp='cp -i'  
#> alias dir='ls -l'
```

When an alias has a space in its name

Use "" (quotation marks)

and

avoid calling an alias

Remove alias

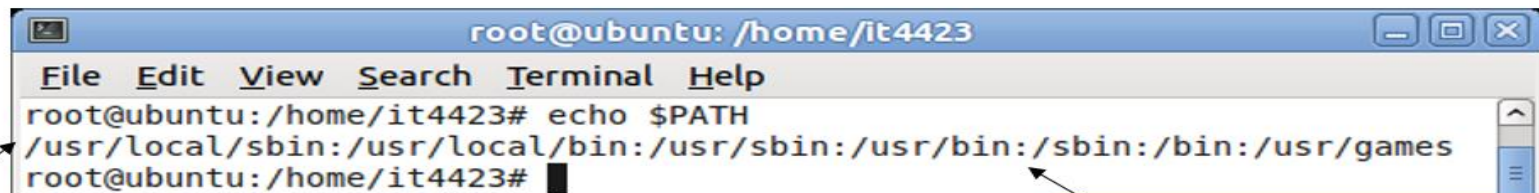
Use the "unalias" command followed by the alias name



Command Search Path

Search path

These are the directories to search when a command is entered without its path.
Paths are stored in the \$PATH environment variable

A terminal window titled 'root@ubuntu: /home/it4423' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the command 'echo \$PATH' being executed, resulting in the output: '/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games'. The prompt 'root@ubuntu:/home/it4423#' is visible at the bottom.

```
root@ubuntu: /home/it4423
File Edit View Search Terminal Help
root@ubuntu:/home/it4423# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
root@ubuntu:/home/it4423#
```

Note that the current directory (represented by a dot `.`) is not in search path. Why?

Each directory is separated by :

```
#>PATH=$PATH:/dir/path; export PATH
```

This is a second command to make PATH global.

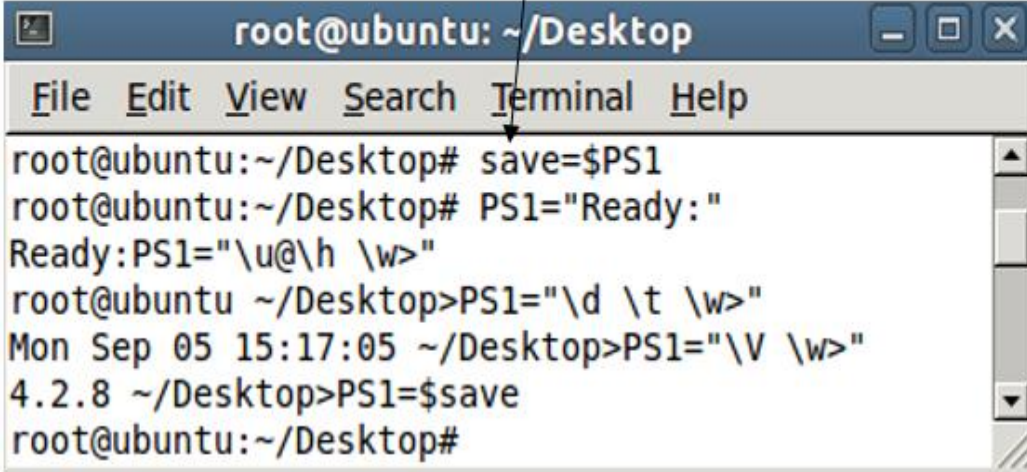
Changing Command Prompt

Command prompt is usually used to display useful environment information such as

Current user, directory, date, etc.

Change the “PS1” variable to change command prompt

Save the original PS1 to a temporary variable "save"



```
root@ubuntu: ~/Desktop
File Edit View Search Terminal Help
root@ubuntu:~/Desktop# save=$PS1
root@ubuntu:~/Desktop# PS1="Ready:"
Ready:PS1="\u@\h \w>"
root@ubuntu ~/Desktop>PS1="\d \t \w>"
Mon Sep 05 15:17:05 ~/Desktop>PS1="\V \w>"
4.2.8 ~/Desktop>PS1=$save
root@ubuntu:~/Desktop#
```

[Reference](#)

Other Environmental Variables

Prompt statement

\$PS1 - command prompt

\$PS2 - command continuation prompt

Other commonly used ones

\$SHELL - current shell

\$HOME - home directory

\$PWD - current working directory

\$PATH - command search path

Use "printenv" command to show all environmental variables

Input and Output

Every process has at least 3 communication channels available

"standard input"	(STDIN)
"standard output"	(STDOUT)
"standard error"	(STDERR)

These channels are setup by the kernel, so the process itself doesn't necessarily know them

Most commands accept their input from STDIN and write their output to STDOUT. They write error messages to STDERR

In the context of an interactive terminal window

STDIN normally reads from the keyboard

STDOUT and STDERR write their output to the screen

Reroute from/to Files

Use “>” to redirect screen output to files

Use “>>” to append to a file rather than to overwrite it

```
#>echo "hello, world" > file1  
#>echo "hello, world, again" >> file1
```

Use “<” to get input from a file

```
#>read variable1 < file1
```

Use “2>” to redirect errors to a file

Use “/dev/null” if errors should be ignored

```
#>badcommand 2> file1  
#>badcommand 2> /dev/null
```

Pipe

Pipe operator: |

To connect the STDOUT of one command to the STDIN of another

```
#>ls -lat | head -2
```

Filter

Any well-behaved command that reads STDIN and writes STDOUT can be used as a filter (that is, a component of a pipeline) to process data.

Common filter commands:

grep, wc, head, tail, tee, cut, sort, tr, cat

[Unix Filter](#)

More Pipe Examples

If file list is too long, use "less" to see them in pages.

```
#>ls -l | less
```

Count how many files

```
#>ls -l | wc -l
```

Look for subdirectories – the first letter in each file is "d" for directories

```
#>ls -l | grep ^d
```

Provide a value to a program reads from user put

grep

grep is used to find text within files

grep [options] PATTERN [FILE...]

Options

- i ignore case
- w whole word
- v reverse results
- o resulted phrase only

Pattern

Strings or regular expressions

More grep Examples

File name ending with ".jpg"

```
#>ls -l | grep .jpg$
```

Get only the match part, not the whole line (-o option)

```
#>ls -l | [a-z0-9-]*.jpg -o
```

[More examples](#)

[grep reference](#)

[Complete Regex reference](#)

Variables

Variable naming

Variable names are case sensitive

All-caps names typically suggest environment variables or variables read from global configuration files

Local variables are all-lowercase with components separated by underscores

Assignment

All variables are of the string data type when assigned

Referencing variables

Use the "\$"+variable name

Use {} around variable name optionally

```
#> var1="today"
```

no space around the = symbol

```
#>var1=1000; printf "User input: $var1\n"  
User input: 1000
```


I/O Command: echo

Use echo command to send results to “STDOUT”

```
#>echo "hello, world"  
hello, world
```

One argument

Multiple arguments followed

```
#>echo "hello," "world"  
hello, world
```

Two arguments

I/O Command: printf

Similar to “echo”, but with some formatting

\t tab
\n new line

Format controls reference: use man or visit

[The printf command](#)

```
#>printf "first name\tlast name\njack\t\tzheng\n"
```

```
first name    last name
```

```
jack          zheng
```

I/O Command: read

Accept user input from the keyboard and save it to a variable

Use -p option for input prompt

```
#> read -p "Enter something:" var1; echo "User input: $var1"  
Enter something: 1000  
User input: 1000
```

; to separate two commands.

User input will be saved to this variable

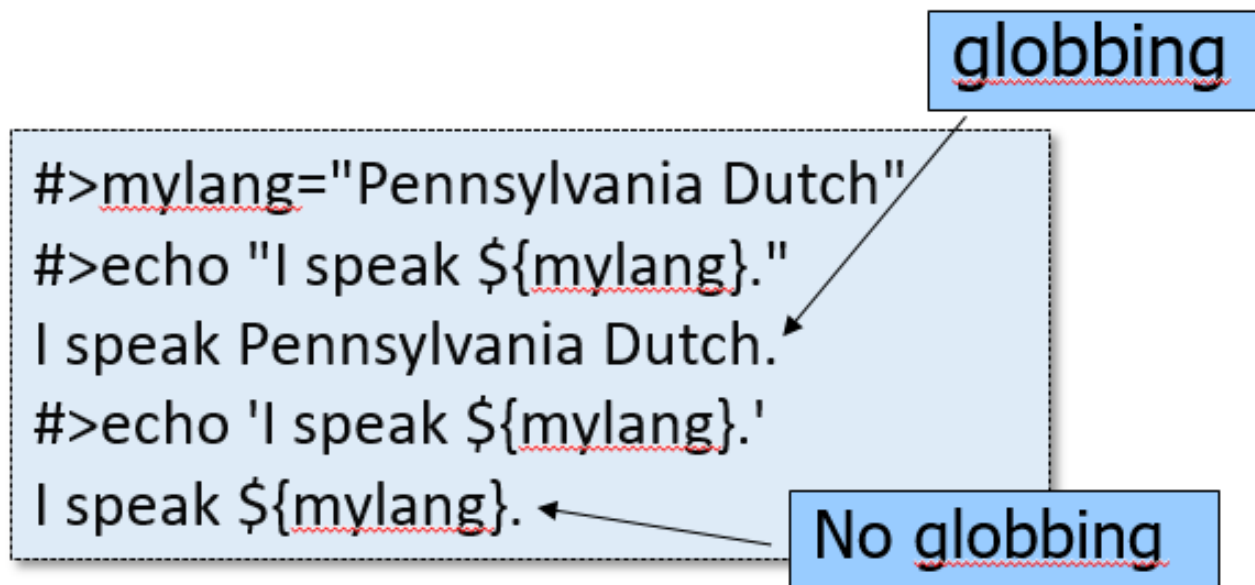
Double and Single Quotes

The shell treats strings enclosed in single and double quotes similarly, except that double-quoted strings are subject to globbing – a pattern matching behavior like:

The expansion of filename-matching metacharacters such as * and ?

Variable expansion \$

Command history !



Capture Command Output

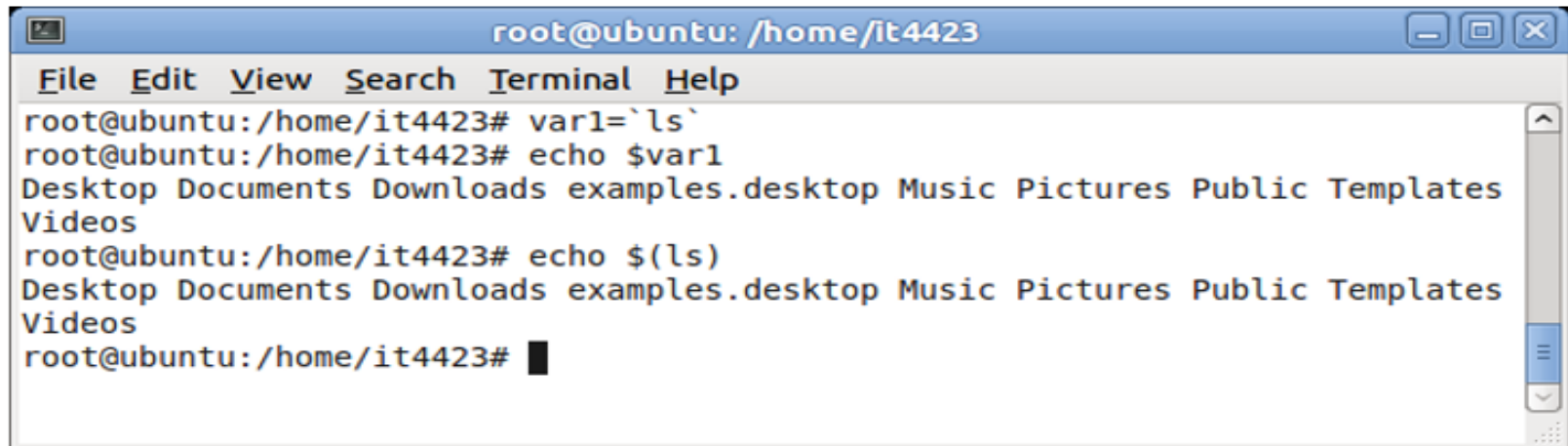
``

Back quotes (or back-ticks), are similar to double quotes, but they have the additional effect of executing the contents of the string as a shell command and replacing the string

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

\$()

Another form of command substitution



```
root@ubuntu: /home/it4423  
File Edit View Search Terminal Help  
root@ubuntu:/home/it4423# var1=`ls`  
root@ubuntu:/home/it4423# echo $var1  
Desktop Documents Downloads examples.desktop Music Pictures Public Templates  
Videos  
root@ubuntu:/home/it4423# echo $(ls)  
Desktop Documents Downloads examples.desktop Music Pictures Public Templates  
Videos  
root@ubuntu:/home/it4423#
```

Summary

Key concepts

Command-line scripting

I/O channels: stdin, stdout, stderr

Pipe

Environmental variable

Key skills: write simple command-line shell scripts utilizing the following elements

Channel operators: > < >> 2> |

Variables

Shell builtin commands

echo, printf, read, bash, alias

Commands: man, type, which, whereis

Good Readings and Resources

[Bash Guide for Beginners](#)

[15 Useful Bash Shell Built-in Commands](#)

Regular Expression for Searching

Regular Expression (Regex)

Looking for text patterns

Commonly used for string search

Use with “grep”

R`#> grep "http.*com" index.html`

Regex Quick Guide

Metacharacter

Meaning

[^]

^, \$

The ^ (circumflex or caret) **outside square brackets** means look only at the beginning of the target string.

The \$ (dollar) means look only at the end of the target string, for example, fox\$ will find a match in 'silver **fox**' since it appears at the end of the string but not in 'the fox jumped over the moon'.

.

The . (period) means any character(s) in this position, for example, **ton.** will find **tons**, **tone** and tonneau but not **wanton** because it has no following character.

[], a-z, 0-9, ^

range

|

* ? +

