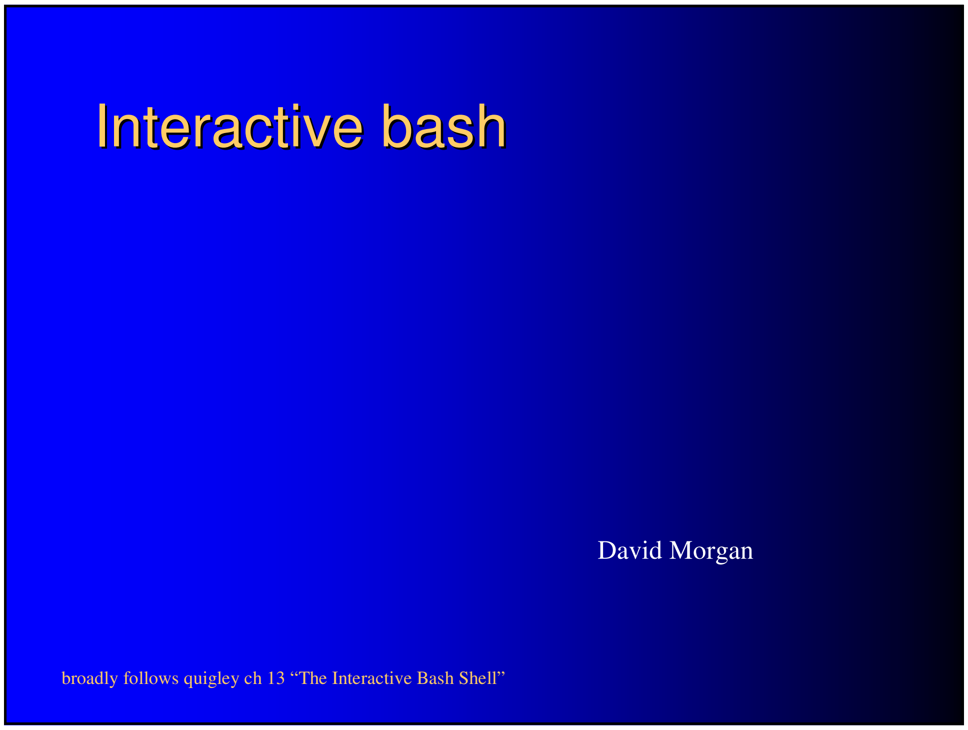
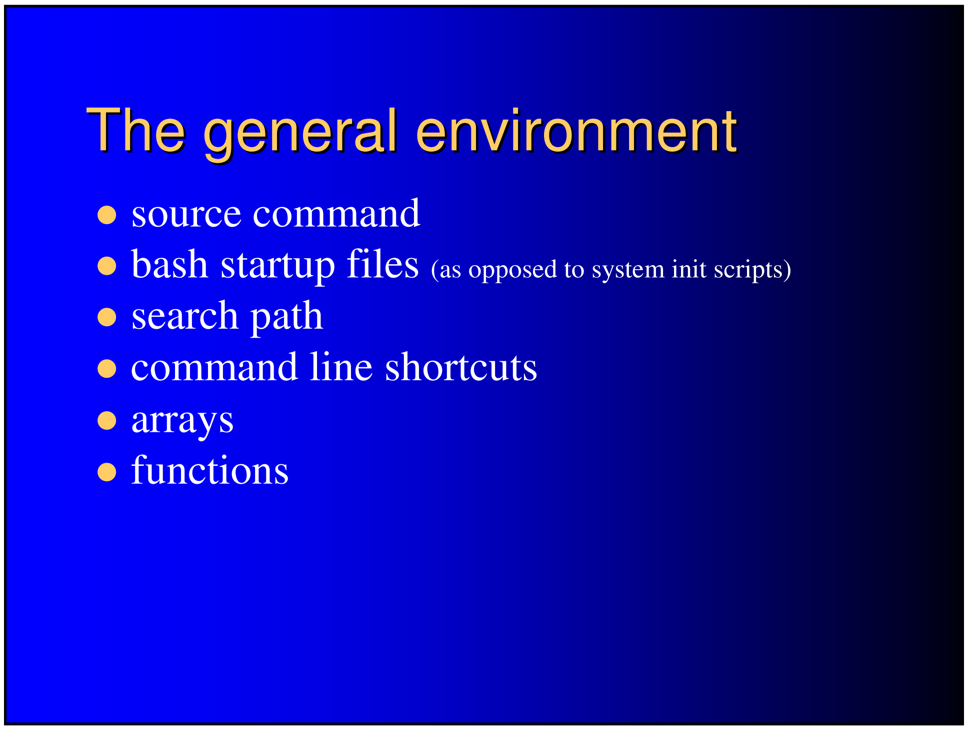


Interactive bash

David Morgan

broadly follows quigley ch 13 “The Interactive Bash Shell”



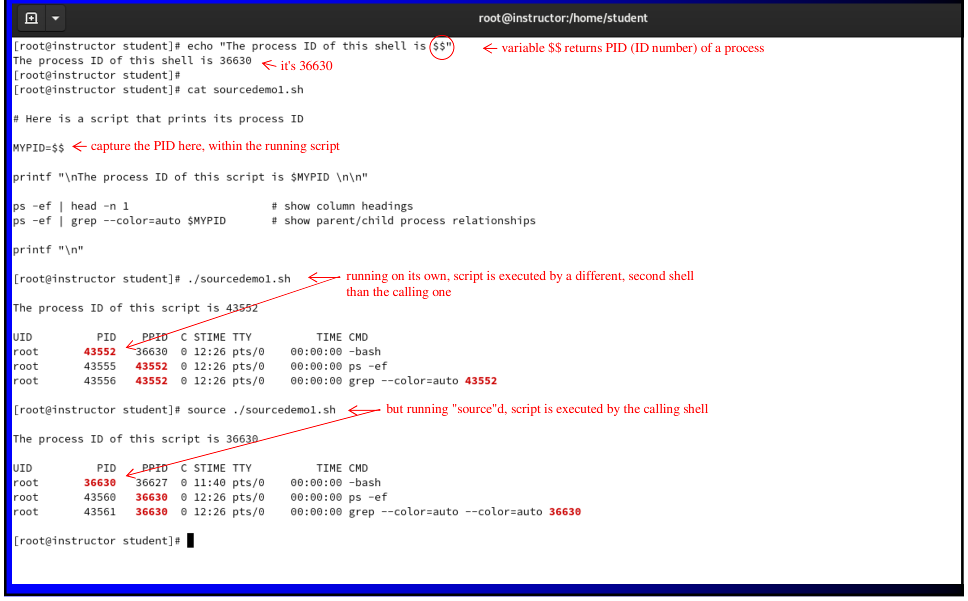
The general environment

- source command
- bash startup files (as opposed to system init scripts)
- search path
- command line shortcuts
- arrays
- functions

The source command

- takes a filename as argument
- operates differently than when called directly
  - executes commands in the file, in the current shell (compare C #include)
  - any #! interpreter script first line is a comment
  - variables are in common with current shell
  - file need not have execute permissions

source demo 1 - current shell



```
[root@instructor student]# echo "The process ID of this shell is $$"
The process ID of this shell is 36630
[root@instructor student]#
[root@instructor student]# cat sourcedemo1.sh

# Here is a script that prints its process ID

MYPID=$$

printf "\nThe process ID of this script is $MYPID \n\n"

ps -ef | head -n 1                    # show column headings
ps -ef | grep --color=auto $MYPID     # show parent/child process relationships

printf "\n"

[root@instructor student]# ./sourcedemo1.sh

The process ID of this script is 43552

UID          PID    PPID  C STIME TTY          TIME CMD
root       43552   36630  0 12:26 pts/0    00:00:00 -bash
root       43555   43552  0 12:26 pts/0    00:00:00 ps -ef
root       43556   43552  0 12:26 pts/0    00:00:00 grep --color=auto 43552

[root@instructor student]# source ./sourcedemo1.sh

The process ID of this script is 36630

UID          PID    PPID  C STIME TTY          TIME CMD
root       36630   36627  0 11:40 pts/0    00:00:00 -bash
root       43560   36630  0 12:26 pts/0    00:00:00 ps -ef
root       43561   36630  0 12:26 pts/0    00:00:00 grep --color=auto --color=auto 36630

[root@instructor student]#
```

← variable $$ returns PID (ID number) of a process

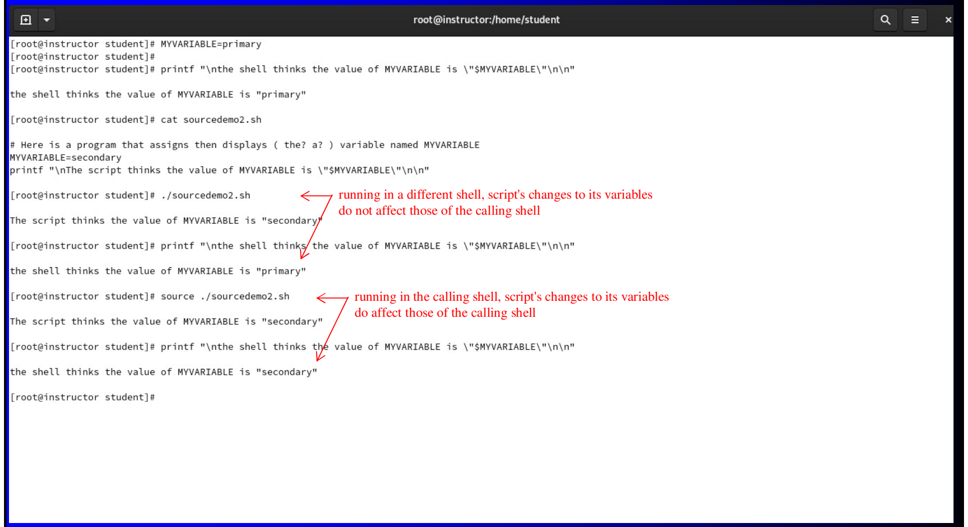
← it's 36630

← capture the PID here, within the running script

← running on its own, script is executed by a different, second shell than the calling one

← but running "source"d, script is executed by the calling shell

source demo 2 - common variables

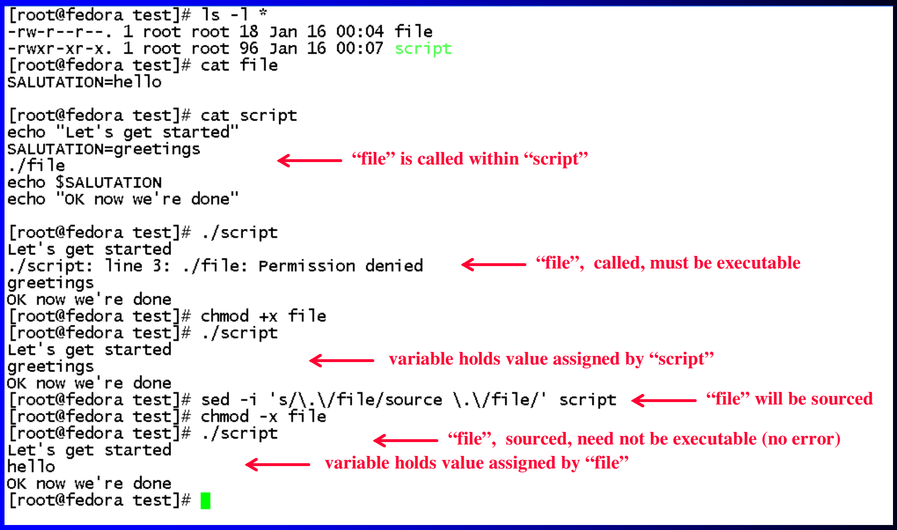


```
[root@instructor student]# MYVARIABLE=primary
[root@instructor student]#
[root@instructor student]# printf "\nthe shell thinks the value of MYVARIABLE is \"$MYVARIABLE\"\n\n"

the shell thinks the value of MYVARIABLE is "primary"

[root@instructor student]# cat sourcedemo2.sh

# Here is a program that assigns then displays ( the? a? ) variable named MYVARIABLE
MYVARIABLE=secondary
printf "\nThe script thinks the value of MYVARIABLE is \"$MYVARIABLE\"\n\n"

[root@instructor student]# ./sourcedemo2.sh

The script thinks the value of MYVARIABLE is "secondary"

[root@instructor student]# printf "\nthe shell thinks the value of MYVARIABLE is \"$MYVARIABLE\"\n\n"

the shell thinks the value of MYVARIABLE is "primary"

[root@instructor student]# source ./sourcedemo2.sh

The script thinks the value of MYVARIABLE is "secondary"

[root@instructor student]# printf "\nthe shell thinks the value of MYVARIABLE is \"$MYVARIABLE\"\n\n"

the shell thinks the value of MYVARIABLE is "secondary"

[root@instructor student]#
```

running in a different shell, script's changes to its variables do not affect those of the calling shell

running in the calling shell, script's changes to its variables do affect those of the calling shell

source demo 3 - execute perm unneeded



```
[root@fedora test]# ls -l *
-rw-r--r--. 1 root root 18 Jan 16 00:04 file
-rwxr-xr-x. 1 root root 96 Jan 16 00:07 script
[root@fedora test]# cat file
SALUTATION=hello

[root@fedora test]# cat script
echo "Let's get started"
SALUTATION=greetings
./file
echo $SALUTATION
echo "OK now we're done"

[root@fedora test]# ./script
Let's get started
./script: line 3: ./file: Permission denied
greetings
OK now we're done
[root@fedora test]# chmod +x file
[root@fedora test]# ./script
Let's get started
greetings
OK now we're done
[root@fedora test]# sed -i 's/\.\/file/source \.\/file/' script
[root@fedora test]# chmod -x file
[root@fedora test]# ./script
Let's get started
hello
OK now we're done
[root@fedora test]#
```

"file" is called within "script"

"file", called, must be executable

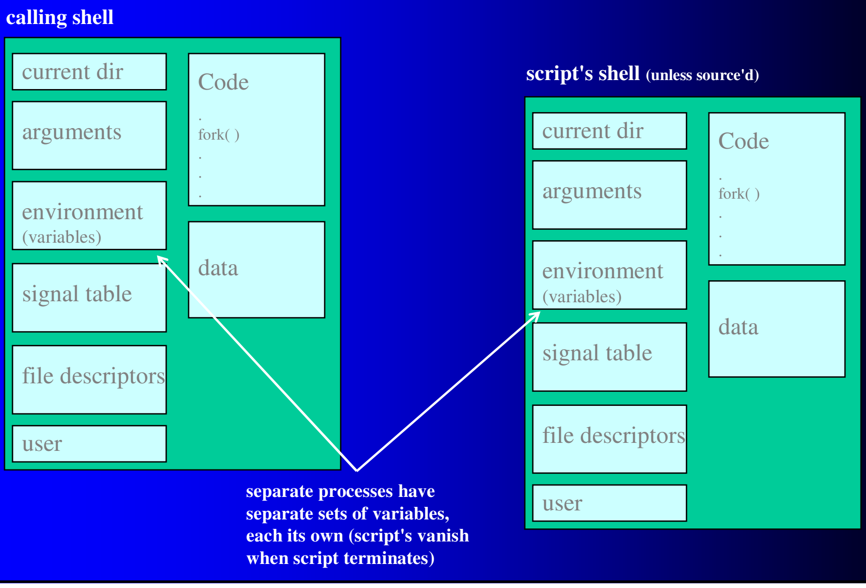
variable holds value assigned by "script"

"file" will be sourced

"file", sourced, need not be executable (no error)

variable holds value assigned by "file"

Why don't variable changes "work"?



bash startup files

- scripts that run when bash starts
- which ones depends on shell type, whether
  – login shell or not, and whether
  – interactive shell or not

Shell types

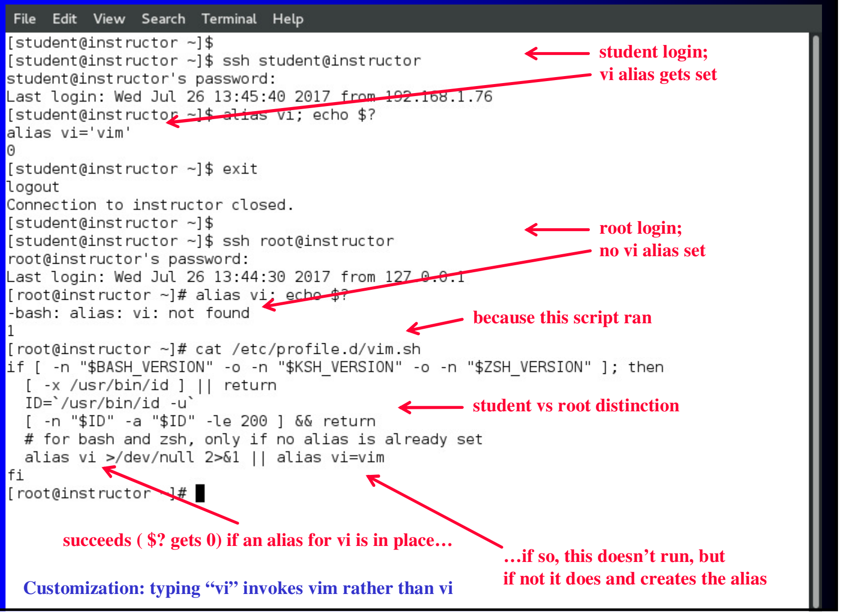
	interactive	non-interactive
login	initial login shells ssh/telnet shells	n/a
non-login	GUI terminal windows’ shells	shell scripts’ shells

Startup files per shell type

	interactive	non-interactive
login	/etc/profile read & executed by bash /etc/profile.d/*.sh sourced by /etc/profile ~/.bash_profile ~/.bash_login one, read & executed by bash ~/.profile ~/.bashrc sourced by .bash_profile /etc/bashrc sourced by .bashrc	n/a
non-login	~/.bashrc called by bash /etc/bashrc sourced by .bashrc	file named in BASH_ENV

Example /etc/profile.d customization script

(vim.sh)



```
[student@instructor ~]$
[student@instructor ~]$ ssh student@instructor
student@instructor's password:
Last login: Wed Jul 26 13:45:40 2017 from 192.168.1.76
[student@instructor ~]$ alias vi; echo $?
alias vi='vim'
0
[student@instructor ~]$ exit
logout
Connection to instructor closed.
[student@instructor ~]$
[student@instructor ~]$ ssh root@instructor
root@instructor's password:
Last login: Wed Jul 26 13:44:30 2017 from 127.0.0.1
[root@instructor ~]# alias vi; echo $?
-bash: alias: vi: not found
1
[root@instructor ~]# cat /etc/profile.d/vim.sh
if [ -n "$BASH_VERSION" -o -n "$KSH_VERSION" -o -n "$ZSH_VERSION" ]; then
  [ -x /usr/bin/id ] || return
  ID=`/usr/bin/id -u`
  [ -n "$ID" -a "$ID" -le 200 ] && return
  # for bash and zsh, only if no alias is already set
  alias vi >/dev/null 2>&1 || alias vi=vim
fi
[root@instructor ~]# █
```

student login; vi alias gets set

root login; no vi alias set

because this script ran

student vs root distinction

succeeds ($? gets 0) if an alias for vi is in place…

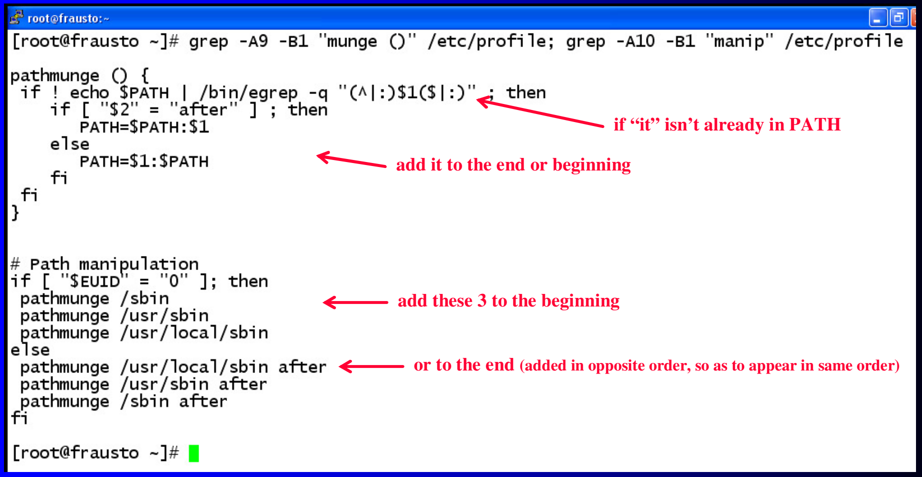
…if so, this doesn't run, but if not it does and creates the alias

Customization: typing "vi" invokes vim rather than vi

The search path

- "The default path is system-dependent, and is set by the administrator who installs bash." –bash man page
  (I cannot figure out how from bash's README/INSTALL.)
- manipulated by some startup files
  - /etc/profile
  - some /etc/profile.d/ scripts
    - krb5-devel.sh, krb5-workstation.sh, ccache.sh, qt.sh
  - others may
- customize in ~/.bash_profile

Search path, excerpts from /etc/profile



```
[root@frausto ~]# grep -A9 -B1 "munge ()" /etc/profile; grep -A10 -B1 "manip" /etc/profile

pathmunge () {
 if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
    if [ "$2" = "after" ] ; then
       PATH=$PATH:$1
    else
       PATH=$1:$PATH
    fi
 fi
}


# Path manipulation
if [ "$EUID" = "0" ]; then
 pathmunge /sbin
 pathmunge /usr/sbin
 pathmunge /usr/local/sbin
else
 pathmunge /usr/local/sbin after
 pathmunge /usr/sbin after
 pathmunge /sbin after
fi

[root@frausto ~]#
```

if “it” isn’t already in PATH

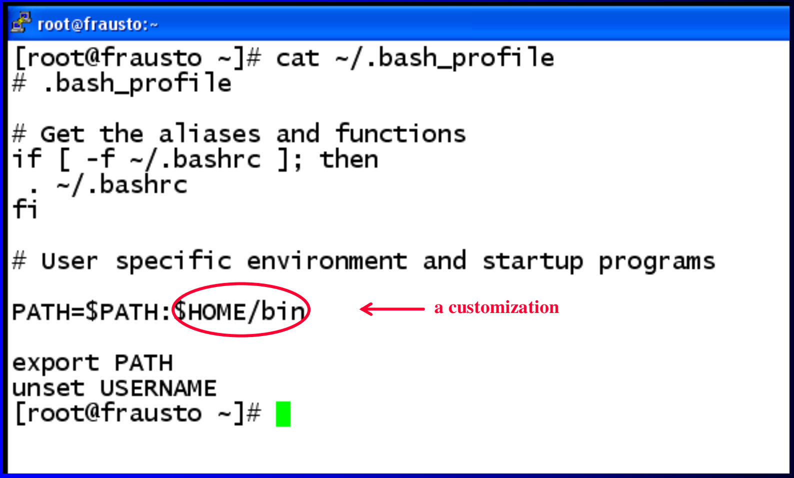
add it to the end or beginning

add these 3 to the beginning

or to the end (added in opposite order, so as to appear in same order)

from a Fedora 10 installation

Search path in ~/.bash_profile



```
[root@frausto ~]# cat ~/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
unset USERNAME
[root@frausto ~]#
```

a customization

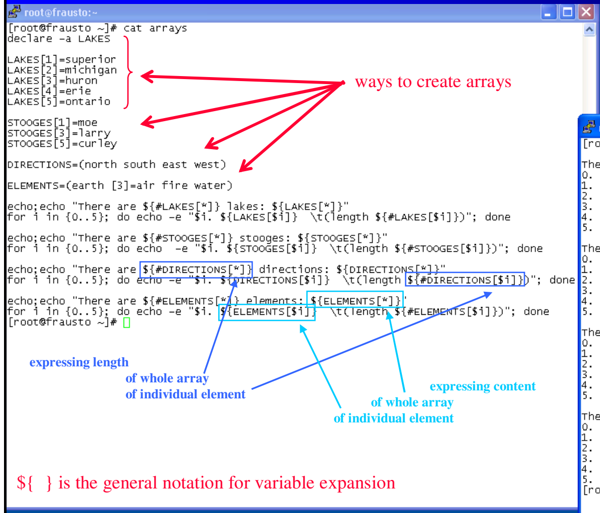
Importantly…

- startup scripts source one another
  - so their PATH alterations, being in a single shell, accumulate and persist
- export PATH
  - so later calls from bash get the changed PATH

Command line shortcuts

- filename completion
  - press tab in midstream while typing filename
  - scope is fileset in current directory
- command completion
  - press tab in midstream while typing command name
- command history
  - latest command recall
    - uparrow - recall most recent command, uparrow again command before that,...
  - reverse incremental history search
    - ctrl-r - then type a substring you remember in a past command you wish to recall

Arrays



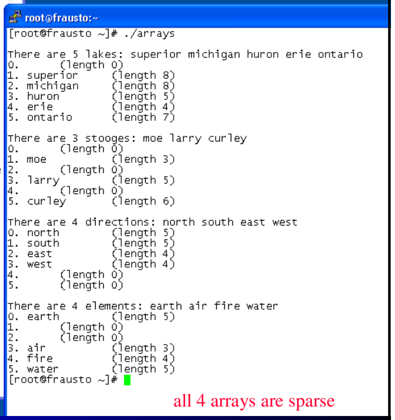
```
[root@frausto ~]# cat arrays
declare -a LAKES

LAKES[1]=superior
LAKES[2]=michigan
LAKES[3]=huron
LAKES[4]=erie
LAKES[5]=ontario

STOOGES[1]=moe
STOOGES[3]=larry
STOOGES[5]=curley

DIRECTIONS=(north south east west)

ELEMENTS=(earth [3]=air fire water)

echo;echo "There are ${#LAKES[*]} lakes: ${LAKES[*]}"
for i in {0..5}; do echo -e "$i. ${LAKES[$i]}  \t(length ${#LAKES[$i]})"; done

echo;echo "There are ${#STOOGES[*]} stooges: ${STOOGES[*]}"
for i in {0..5}; do echo  -e "$i. ${STOOGES[$i]}  \t(length ${#STOOGES[$i]})"; done

echo;echo "There are ${#DIRECTIONS[*]} directions: ${DIRECTIONS[*]}"
for i in {0..5}; do echo -e "$i. ${DIRECTIONS[$i]}  \t(length ${#DIRECTIONS[$i]})"; done

echo;echo "There are ${#ELEMENTS[*]} elements: ${ELEMENTS[*]}"
for i in {0..5}; do echo -e "$i. ${ELEMENTS[$i]}  \t(length ${#ELEMENTS[$i]})"; done
[root@frausto ~]#
```

ways to create arrays

expressing length
of whole array
of individual element

expressing content
of whole array
of individual element

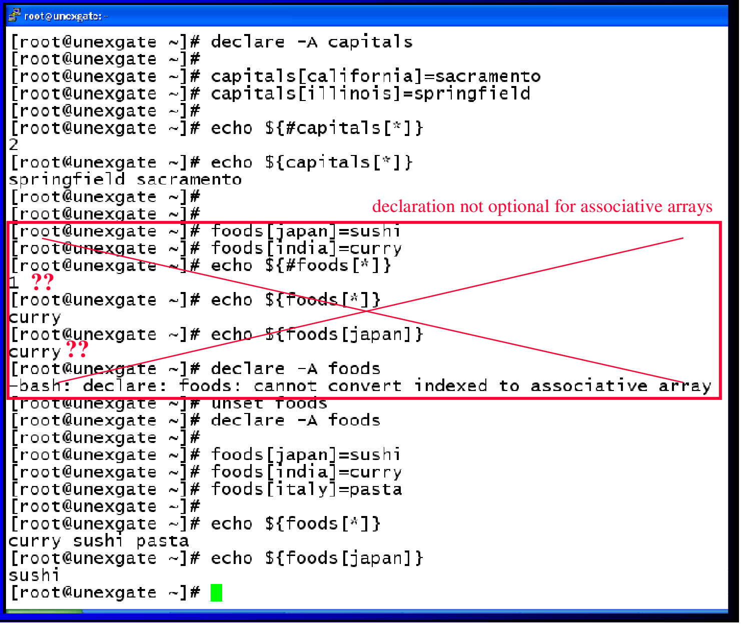
${ } is the general notation for variable expansion



```
[root@frausto ~]# ./arrays

There are 5 lakes: superior michigan huron erie ontario
0.       (length 0)
1. superior     (length 8)
2. michigan     (length 8)
3. huron        (length 5)
4. erie         (length 4)
5. ontario      (length 7)

There are 3 stooges: moe larry curley
0.       (length 0)
1. moe          (length 3)
2.       (length 0)
3. larry        (length 5)
4.       (length 0)
5. curley       (length 6)

There are 4 directions: north south east west
0. north        (length 5)
1. south        (length 5)
2. east         (length 4)
3. west         (length 4)
4.       (length 0)
5.       (length 0)

There are 4 elements: earth air fire water
0. earth        (length 5)
1.       (length 0)
2.       (length 0)
3. air          (length 3)
4. fire         (length 4)
5. water        (length 5)
[root@frausto ~]#
```

all 4 arrays are sparse

New (bash 4) associative array type



```
[root@unexgate ~]# declare -A capitals
[root@unexgate ~]#
[root@unexgate ~]# capitals[california]=sacramento
[root@unexgate ~]# capitals[illinois]=springfield
[root@unexgate ~]#
[root@unexgate ~]# echo ${#capitals[*]}
2
[root@unexgate ~]# echo ${capitals[*]}
springfield sacramento
[root@unexgate ~]#
[root@unexgate ~]#
[root@unexgate ~]# foods[japan]=sushi
[root@unexgate ~]# foods[india]=curry
[root@unexgate ~]# echo ${#foods[*]}
1 ??
[root@unexgate ~]# echo ${foods[*]}
curry
[root@unexgate ~]# echo ${foods[japan]}
curry ??
[root@unexgate ~]# declare -A foods
-bash: declare: foods: cannot convert indexed to associative array
[root@unexgate ~]# unset foods
[root@unexgate ~]# declare -A foods
[root@unexgate ~]#
[root@unexgate ~]# foods[japan]=sushi
[root@unexgate ~]# foods[india]=curry
[root@unexgate ~]# foods[italy]=pasta
[root@unexgate ~]#
[root@unexgate ~]# echo ${foods[*]}
curry sushi pasta
[root@unexgate ~]# echo ${foods[japan]}
sushi
[root@unexgate ~]#
```

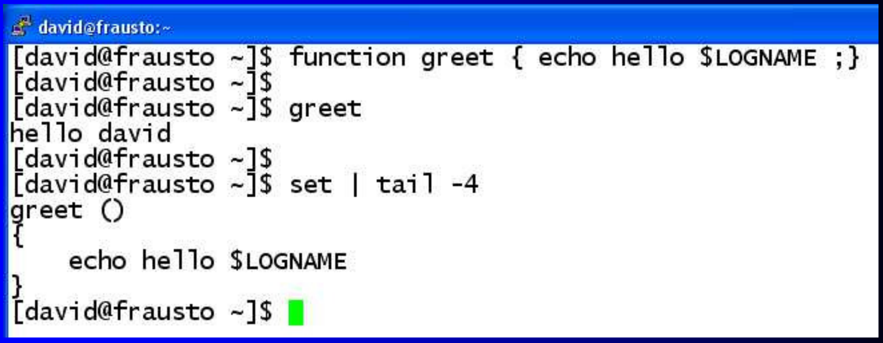
declaration not optional for associative arrays

Functions

- install runnable code unit in memory
- under a callable name

"A shell function… stores a series of commands for later execution.
When the name of a shell function is used as a simple command name,
the list of commands associated with that function name is executed.
Functions are executed in the context of the current shell; no new process
is created to interpret them (contrast this with the execution of a shell script).
bash man page

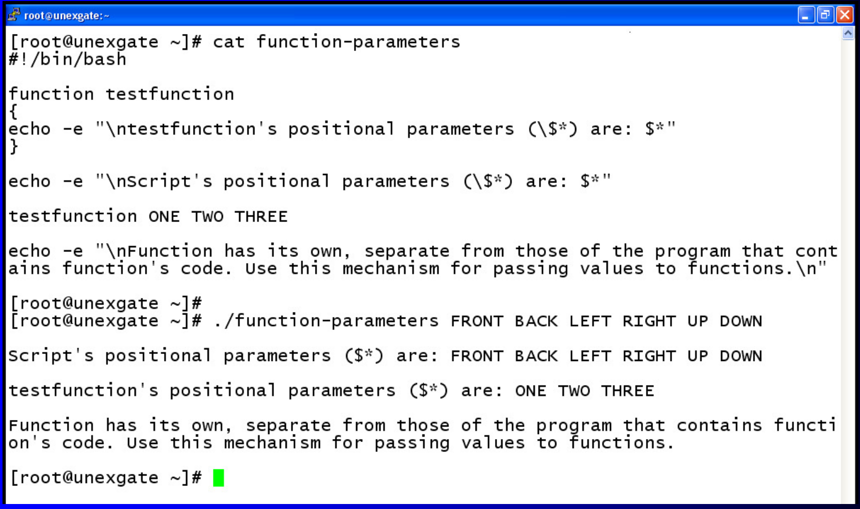
Functions



```
[david@frausto ~]$ function greet { echo hello $LOGNAME ;}
[david@frausto ~]$
[david@frausto ~]$ greet
hello david
[david@frausto ~]$
[david@frausto ~]$ set | tail -4
greet ()
{
    echo hello $LOGNAME
}
[david@frausto ~]$ 
```

Functions – passing parameters

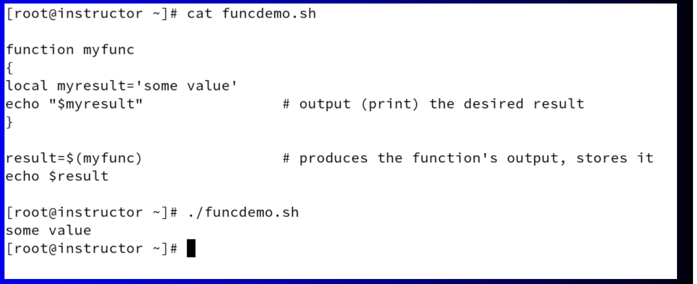
via positional parameters, like any command



```
[root@unexgate ~]# cat function-parameters
#!/bin/bash

function testfunction
{
echo -e "\ntestfunction's positional parameters (\$*) are: $*"
}

echo -e "\nScript's positional parameters (\$*) are: $*"

testfunction ONE TWO THREE

echo -e "\nFunction has its own, separate from those of the program that cont
ains function's code. Use this mechanism for passing values to functions.\n"

[root@unexgate ~]#
[root@unexgate ~]# ./function-parameters FRONT BACK LEFT RIGHT UP DOWN

Script's positional parameters ($*) are: FRONT BACK LEFT RIGHT UP DOWN

testfunction's positional parameters ($*) are: ONE TWO THREE

Function has its own, separate from those of the program that contains functi
on's code. Use this mechanism for passing values to functions.

[root@unexgate ~]#
```

Functions – returning values

- functions do not return values
  - only an exit status, like any command
    - exit status explicitly set in a "return" statement, or
    - that of the function's final command
- can set a global variable
- better:



```
[root@instructor ~]# cat funcdemo.sh

function myfunc
{
local myresult='some value'
echo "$myresult"                # output (print) the desired result
}

result=$(myfunc)                # produces the function's output, stores it
echo $result

[root@instructor ~]# ./funcdemo.sh
some value
[root@instructor ~]#
```