Bash Shell Scripting for HPC

Dr. David Hoover, HPC @ NIH
staff@hpc.nih.gov

This Presentation Online

https://hpc.nih.gov/training

Guide To This Presentation

- These lines are narrative information
- Beige boxes are interactive terminal sessions:

```
$ echo Today is $(date)
```

- Gray boxes are file/script contents:

```
This is the first line of this file.
This is the second line.
```

- White boxes are commands:

```
echo Today is $(date)
```

Terminal Emulators

- PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/)
- iTerm2 (http://iterm2.com/)
- NoMachine (https://www.nomachine.com/)
- MobaXterm (https://mobaxterm.mobatek.net/)

Connect to Biowulf

- open terminal window on desktop, laptop

```
$ ssh user@biowulf.nih.gov
```

- DON'T USE 'user'. Replace with YOUR login.

Start Interactive Session

- Allocate 5 GB local scratch space
- Default 8 hours of walltime
- Default 2 CPUs and 1.5 GB RAM

```
sinteractive --gres=lscratch:5
```

Copy Examples

- Create a working directory

```
mkdir bash_class
cd bash_class
```

- Copy files recursively to your own directory

```
cp -R /data/classes/bash/* .
```

Copy Examples

- Not on Helix/Biowulf:

```
mkdir bash_class
cd bash_class

curl \
https://hpc.nih.gov/training/handouts/BashScripting.tgz \
> BashScripting.tgz

tar -x -z -f BashScripting.tgz
```

HISTORY AND INTRODUCTION

Bash

- Bash is a shell, like Bourne, Korn, and C
- Written and developed by the FSF in 1989
- Default shell for most Linux flavors

Definitive References

- http://www.gnu.org/software/bash/manual/
- http://www.tldp.org/
- http://wiki.bash-hackers.org/

Bash on HPC

- All machines within HPC run Bash v4.2.46

You might be using Bash already

```
$ ssh user@biowulf.nih.gov
...
Last login: Wed Aug 10 12:27:09 2018 from
123.456.78.90
$ echo $SHELL
/bin/bash
```

If not, just start a new shell:

```
$ bash
```

What shell are you running?

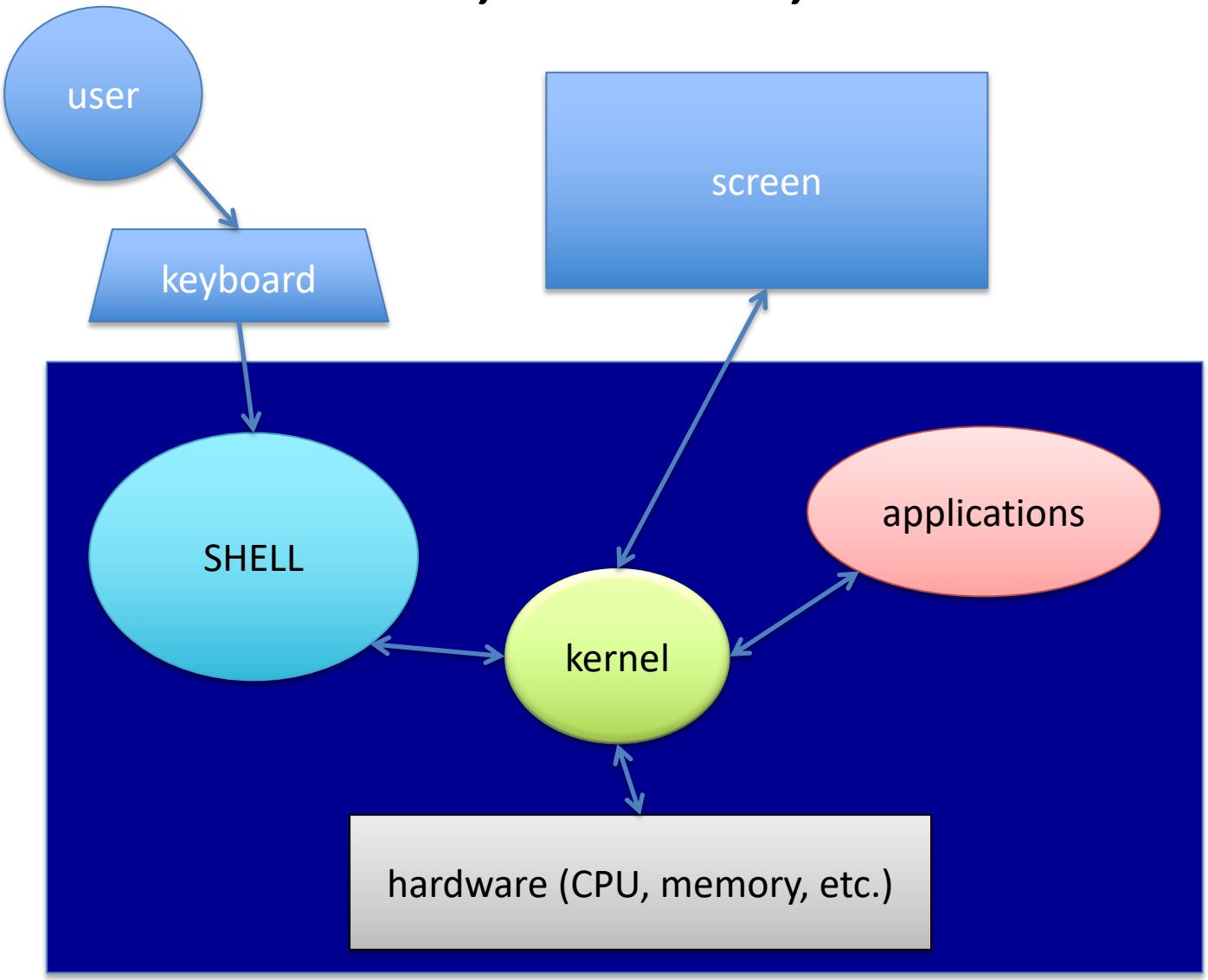
- You should be running bash:

```
$ echo $0
-bash
```

- Maybe you're running something else?

```
$ echo $0
-ksh
-csh
-tcsh
-zsh
```

Shell, Kernel, What?



ESSENTIALS

*nix Prerequisite

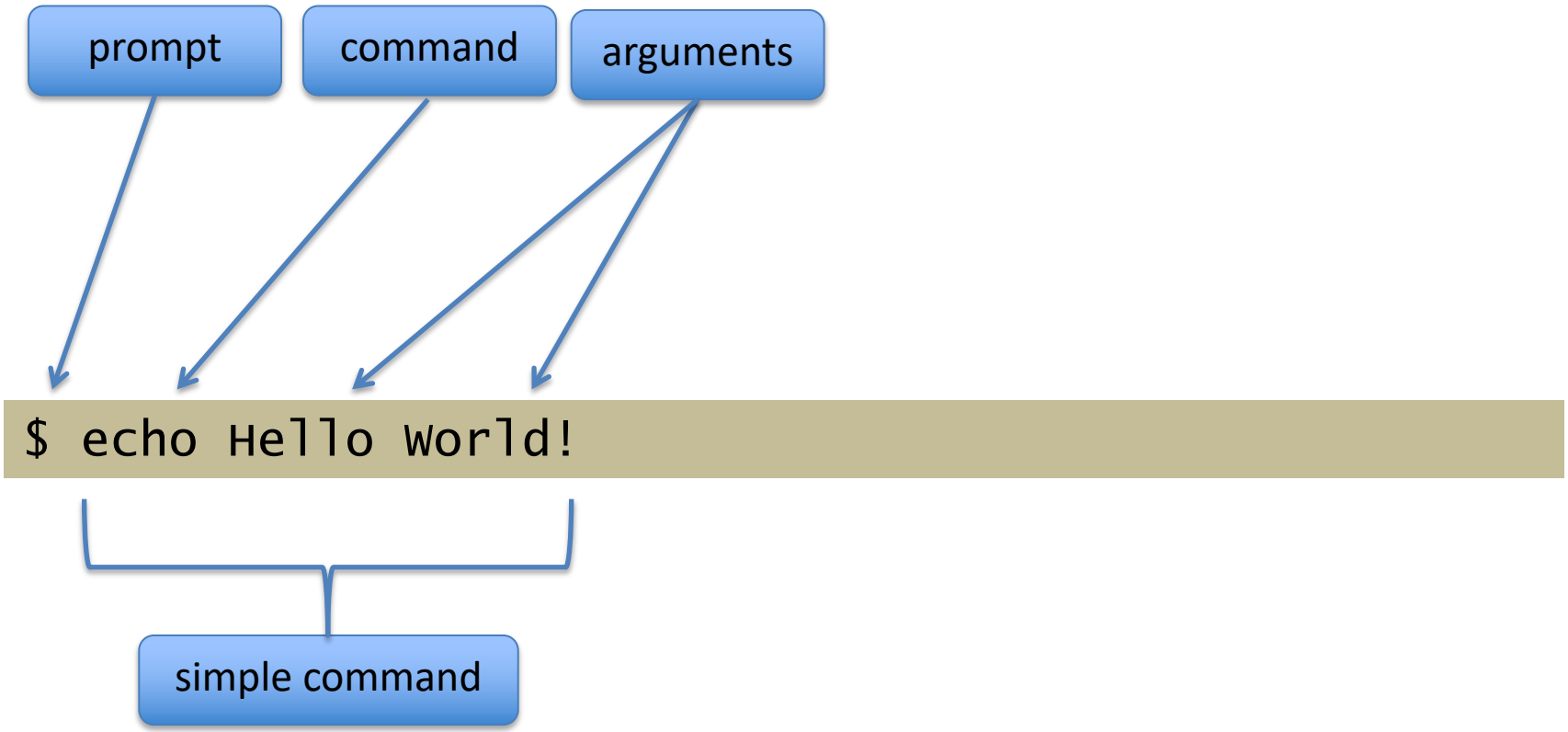
- It is essential that you know something about UNIX or Linux
- Introduction to Linux (Helix Systems)
- https://hpc.nih.gov/training
- Linux Tutorial at TACC
- Linux Tutorial at TLDP

Elemental Bash

- Bash is a command processor: interprets characters typed on the command line and tells the kernel what programs to use and how to run them
- AKA command line interpreter (CLI)

POTPOURRI OF COMMANDS

Simple Command

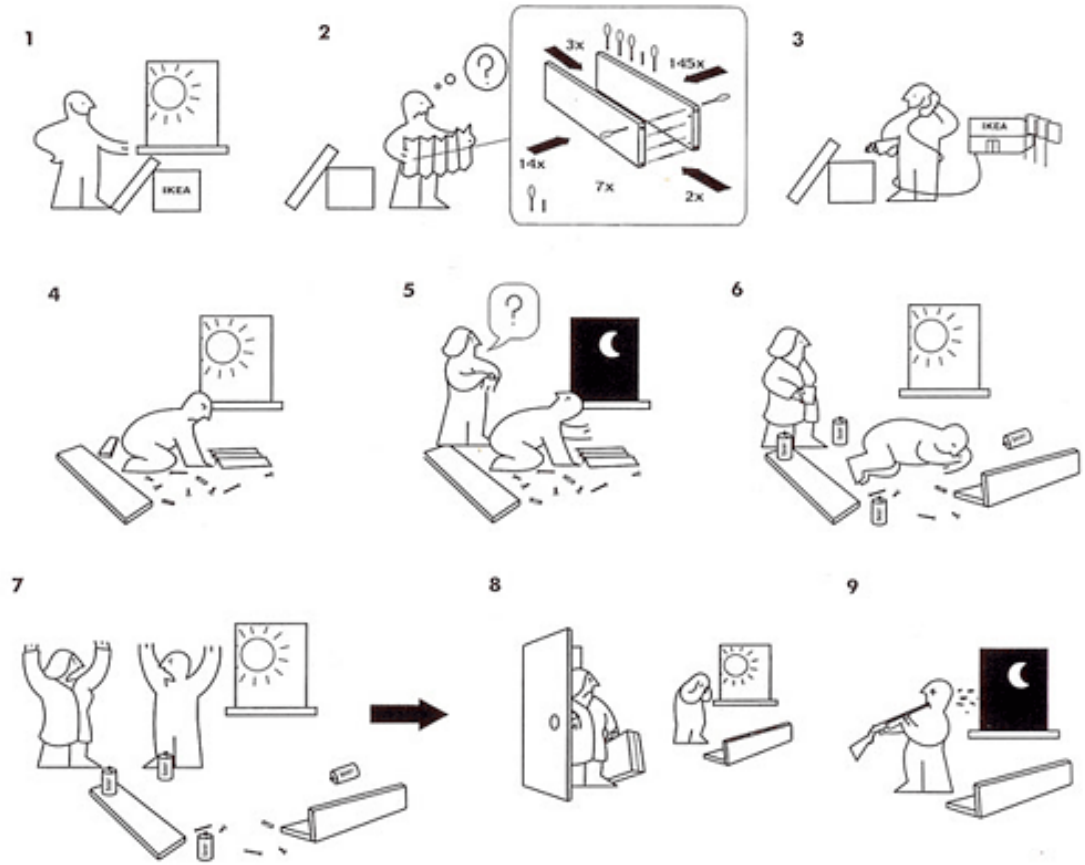


Essential *nix Commands

cd	ls	pwd	file	wc	find	du
chmod	touch	mv	cp	rm	cut	paste
sort	split	cat	grep	head	tail	less
more	sed	awk	diff	comm	ln	
mkdir	rmdir	df	pushd	popd		
date	exit	ssh	rsh	printenv	time	echo
ps	jobs	[CTRL-c]	[CTRL-z]	[CTRL-d]	top	kill
apropos	info	man	help	type		

Comprehensive List

- https://hpc.nih.gov/training/handouts/BashScripting_LinuxCommands.pdf



DOCUMENTATION

Documentation

- [google]
- `info`
- `man`
- `help`
- `type`
- `--help`

Documentation: `info`

- `info` is a comprehensive commandline documentation viewer
- displays various documentation formats

```
$ info kill
$ info diff
$ info read
$ info strftime
$ info info
```

Documentation: man

```
$ man pwd
$ man diff
$ man head
```

Documentation: `type`

- `type` is a `builtin` that displays the `type` of a word

```
$ type -t rmdir
file
$ type -t if
keyword
$ type -t echo
builtin
$ type -t module
function
$ type -t ls
alias
```

builtin

- Bash has built-in commands (`builtin`)
- `echo, exit, hash, printf`

Documentation: `builtin`

- Documentation can be seen with `help`

```
$ help
```

- Specifics with `help [builtin]`

```
$ help pwd
```

Non-Bash Commands

- Found in `/bin`, `/usr/bin`, and `/usr/local/bin`
- Some overlap between Bash `builtin` and external executables

```
$ help time
$ man time
```

```
$ help pwd
$ man pwd
```

--help, -h

- Many commands have (or should have) a help menu

```
$ find --help
```

- Sometimes -h works as well, but not always

```
$ sbatch -h
```

SCRIPTS

Why write a script?

- One-liners are not enough
- Quick and dirty prototypes
- Maintain library of functional tools
- Glue to string other apps together
- Batch system submissions

What is in a Script?

- Commands
- Variables
- Functions
- Loops
- Conditional statements
- Comments and documentation
- Options and settings

My Very First Script

- create a script and inspect its contents:

```
echo 'echo Hello World!' > hello_world.sh
cat hello_world.sh
```

- call it with bash:

```
bash hello_world.sh
```

- results:

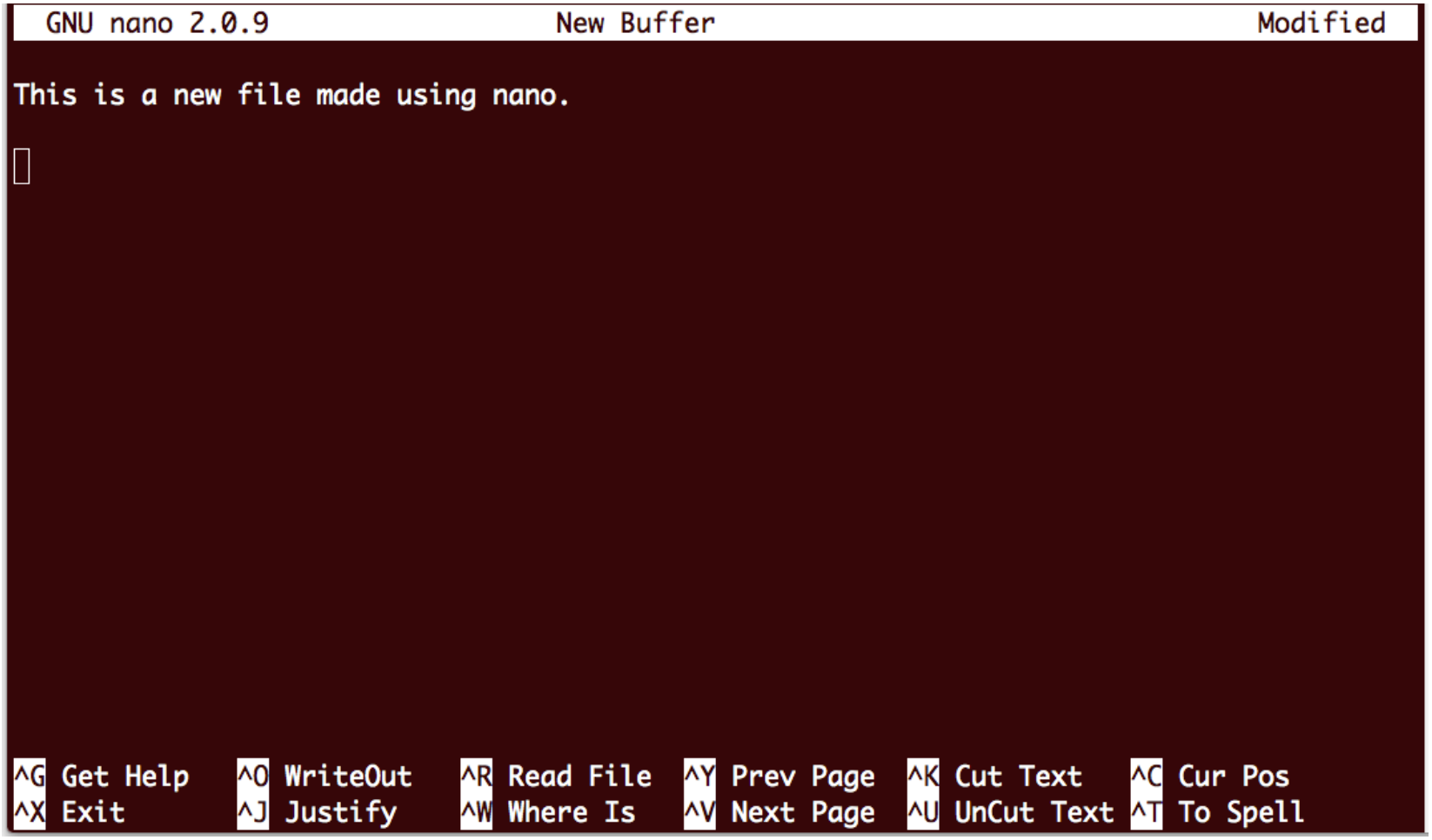
```
$ bash hello_world.sh
Hello World!
```

Multiline file creation

- create a multiline script:

```
$ cat << ALLDONE > hello_world_multiline.sh
> echo Hello World!
> echo Yet another line.
> echo This is getting boring.
> ALLDONE
$
$ bash hello_world_multiline.sh
Hello World!
Yet another line.
This is getting boring.
$
```

nano file editor



Other Editors

- `SciTE`
- `gedit`
- `nano` and `pico`
- `vi`, `vim`, and `gvim`
- `emacs`
- Use `dos2unix` if text file created on Windows machine

Execute With A Shebang!

- If bash is the default shell, making it executable allows it to be run directly

```
chmod +x hello_world.sh
./hello_world.sh
```

- Adding a shebang specifies the scripting language:

```
#!/bin/bash
echo Hello World!
```

Debugging

- Call bash with `-x -v`

```
bash -x -v hello_world.sh
```

```
#!/bin/bash -xv
echo Hello World!
```

- `-x` displays commands and their results
- `-v` displays everything, even comments and spaces
- `-xv` can be combined

A Word About The Shebang

- The shebang allows only one argument

```
#!/bin/bash -xv
echo Hello World!
```

- is ok

```
#!/bin/bash -x -v
echo Hello World!
```

- is not ok

examples/input/args.sh

```
echo You passed $# arguments to the script
echo The first three arguments are:
echo $1
echo $2
echo $3
```

Special Parameters - Positional

- Positional parameters are arguments passed to the shell when invoked
- Denoted by ${digit}, > 0

```
$ cat x.sh
#!/bin/bash
echo ${4} ${15} ${7} ${3} ${1} ${20}
$ bash x.sh {A..Z}
D O G C A T
```

- Normally used with scripts and functions

Shell Parameters - Special

Special parameters have a single character

`$*`	expands to the positional parameters
`$@`	the same as $*, but as array rather than string
`$#`	number of positional parameters
`$-`	current option flags when shell invoked
`$$`	process id of the shell
`$!`	process id of last executed background command
`$0`	name of the shell or shell script
`$_`	final argument of last executed foreground command
`$?`	exit status of last executed foreground command

SETTING THE ENVIRONMENT

Environment

- A set of variables and functions recognized by the kernel and used by most programs
- Not all variables are environment variables, must be `exported`
- Initially set by **startup files**
- `printenv` displays variables and values in the environment
- `set` displays ALL variable

Variables

- A *variable* is construct to hold information, assigned to a *word*
- Variables are initially *undefined*
- Variables have *scope*
- Variables are a key component of the shell's *environment*

Using Variables

- A simple script:

```
#!/bin/bash
cd /path/to/somewhere
mkdir /path/to/somewhere/test1
cd /path/to/somewhere/test1
echo 'this is boring' > /path/to/somewhere/test1/file1
```

- Variables simplify and make portable:

```
#!/bin/bash
dir=/path/to/somewhere
cd $dir
mkdir $dir/test1
cd $dir/test1
echo 'this is boring' > $dir/test1/file1
```

Set a variable

- Formally done using `declare`

```
declare myvar=100
```

- Very simple, no spaces

```
myvar=10
```

Set a variable

- Examine value with `echo` and `$`:

```
$ echo $myvar
10
```

- This is actually **parameter expansion**

export

- `export` is used to set an environment variable:

```
MYENVVAR=10
export MYENVVAR
printenv MYENVVAR
```

- You can do it one move

```
export MYENVVAR=10
```

`unset` a variable

- `unset` is used for this

```
unset myvar
```

- Can be used for environment variables

```
$ echo $HOME
/home/user
$ unset HOME
$ echo $HOME

$ export HOME=/home/user
```

Remove from environment

- `export -n`

```
$ declare myvar=10
$ echo $myvar
10
$ printenv myvar      # nothing - not in environment
$ export myvar
$ printenv myvar      # now it's set
10
$ echo $myvar
10
$ export -n myvar
$ printenv myvar      # now it's gone
$ echo $myvar
10
```

Bash Variables

- `$HOME` = `/home/[user]` = `~`
- `$PWD` = current working directory
- `$PATH` = list of filepaths to look for commands
- `$TMPDIR` = temporary directory (`/tmp`)
- `$RANDOM` = random number
- Many, many others…

printenv

```
$ printenv
HOSTNAME=biowulf.nih.gov
TERM=xterm
SHELL=/bin/bash
HISTSIZE=500
SSH_CLIENT=96.231.6.99 52018 22
SSH_TTY=/dev/pts/274
HISTFILESIZE=500
USER=student1
```

module

- module can set your environment

```
module load python/2.7
module unload python/2.7
```

- Can be used for environment variables

```
module avail
```

https://hpc.nih.gov/apps/modules.html

Setting Your Prompt

- The `$PS1` variable (primary prompt, `$PS2` and `$PS3` are for other things)
- Has its own format rules
- Example:

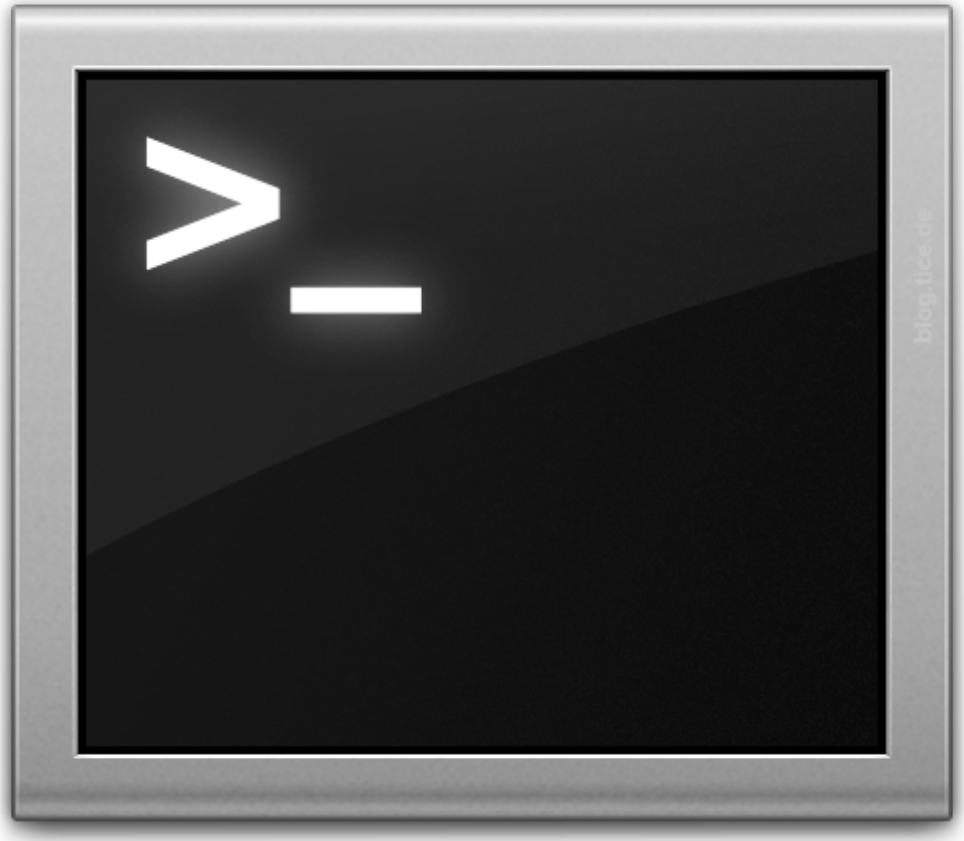
```
PS1="[\u@\h \W]$ "
```

```
[user@host myDir]$ echo Hello World!
Hello World!
[user@host myDir]$
```

Setting Your Prompt

- \d date ("Tue May 6")
- \h hostname ("helix")
- \j number of jobs
- \u username
- \W basename of $PWD
- \a bell character (why?)

http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/index.html
http://www.gnu.org/software/bash/manual/bashref.html#Printing-a-Prompt



COMMAND LINE INTERPRETER

Watch What You Write

- Bash interprets what you write *after* you hit return
- Patterns of characters can cause **expansion**

Parameter Expansion

- $ is placed outside for parameter expansion

```
name=monster
echo $name
```

```
monster
```

- Braces can be used to preserve variable

```
echo $name_silly
```

```

```

```
echo ${name}_silly
```

```
monster_silly
```

Parameter Expansion

- More than one in a row

```
$ var1=cookie
$ var3=_is_silly
$ echo ${var1}_${name}${var3}
```

```
cookie_monster_is_silly
```

Brace Expansions

- Brace expansion { , , }

```
echo {bilbo,frodo,gandalf}
```

```
bilbo frodo gandalf
```

```
echo {0,1,2,3,4,5,6,7,8,9}
```

```
0 1 2 3 4 5 6 7 8 9
```

Brace Expansions

- Brace expansion { .. }

```
echo {0..9}
```

```
0 1 2 3 4 5 6 7 8 9
```

```
$ echo {b..g}
```

```
b c d e f g
```

```
echo {bilbo..gandalf}
```

```
{bilbo..gandalf}
```

Brace Expansions

- Nested brace expansions

```
mkdir z{0,1,2,3,4,5,6,7,8,9}
ls
```

```
z0 z1  z2  z3  z4  z5  z6  z7  z8  z9
```

```
rmdir z{{1..4},7,8}
ls
```

```
z0 z5  z6  z9
```

Brace Expansions

- Distinct from parameter expansion $ or ${

```
echo {$var1,${name},brought,to,you,by,{1..3}}
```

```
cookie monster brought to you by 1 2 3
```

Arithmetic Expansion

- `(( ))` is used to evaluate math
- `$` is placed outside for parameter expansion

```
echo ((12-7))
```

```
-bash: syntax error near unexpected token `('
```

```
echo $((12-7))
```

```
5
```

Arithmetic Expansion

- Variables can be updated, not just evaluated

```
a=4
b=8
echo $a
```

```
4
```

```
echo $b
```

```
8
```

```
echo $((a+b))
```

```
12
```

Arithmetic Expansion

- Variables can be updated, not just evaluated

```
echo $a
```

```
4
```

```
echo $b
```

```
8
```

```
echo $((a=a+b))
```

```
12
```

```
echo $a
```

```
12
```

Arithmetic Expansion

- The ++ and -- operators only work on variables, and update the value

```
a=4
((a++))
echo $a
```

```
5
```

```
unset b
((b--))
echo $b
```

```
-1
```

Arithmetic – integers only

- Bash can only handle integers

```
a=4.5
((a=a/3))
```

```
-bash: ((: 4.5: syntax error: invalid arithmetic operator
(error token is ".5")
```

Arithmetic – integers only

- Bash can only do integer math

```
a=3
((a=a/7))
echo $a
```

```
0
```

- Division by zero is caught with exit status

```
((a=a/0))
```

```
-bash: let: a=a/0: division by 0 (error token is "0")
```

Arithmetic Expansion

- Math is done using `let` and '`(( ))`'

```
$ a=1
$ echo $a
1
$ let a++
$ echo $a
2
$ ((a++))
$ echo $a
3
$ let a=a+4
$ echo $a
7
```

Command Substitution

- ` = backtick
- better to use $()

```
echo uname -n
```

```
uname -n
```

```
echo `uname -n`
```

```
biowulf.nih.gov
```

```
echo $(uname -n)
```

```
biowulf.nih.gov
```

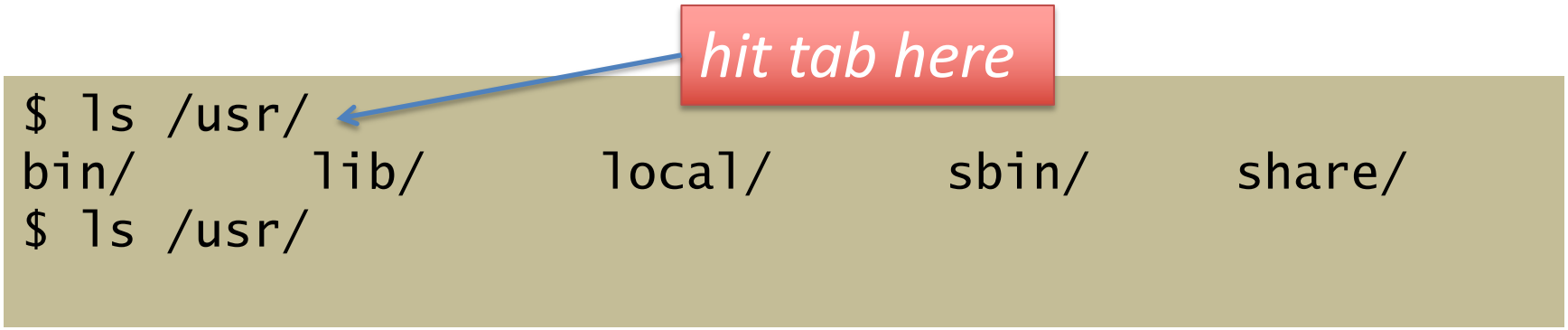
Command Substitution

- Nested processes

```
y=$(wc -l $(find scripts/ -type f) | tail -n 1)
echo $y
```

```
633 total
```

Tab Expansion



Tilde Expansion

```
$ echo ~
```

```
/home/user
```

Quotes

- Single quotes preserve literal values

```
echo 'cd $PWD $(uname -n)'
```

```
cd $PWD $(uname -n)
```

- Double quotes allow variable and shell expansions

```
echo "cd $PWD $(uname -n)"
```

```
cd /home/user biowulf.nih.gov
```

Quotes

- Double quotes also preserve blank characters

```
$ msg=$(echo Hi$'\t'there.$'\n'How are you?)
$ echo $msg
Hi there. How are you?
$ echo "$msg"
Hi      there.
How are you?
```

- use `$'\t'` to insert a tab
- use `$'\n'` to insert a newline

Escapes

- The escape character ‘\’ preserves literal value of following character

```
echo \$PWD is $PWD
```

```
$PWD is /home/user
```

- However, it treats newline as line continuation

```
echo \$PWD is $PWD \
something else
```

```
$PWD is /home/user something else
```

Escapes

- Funny non-printing characters
- $'char'

```
echo Hello World
```

```
Hello World
```

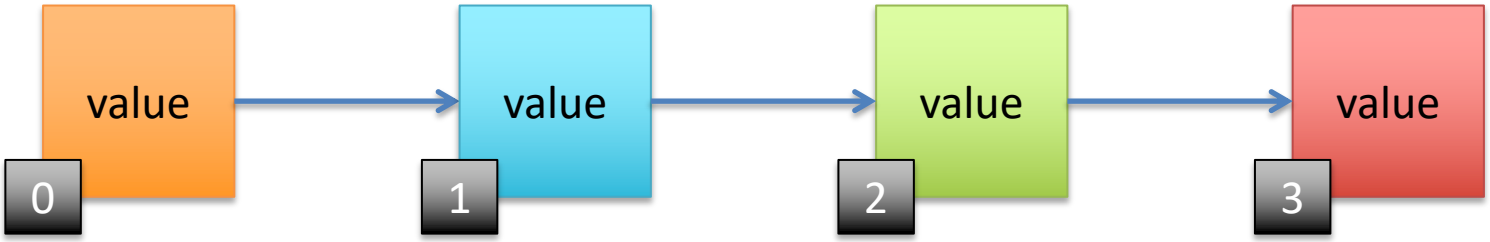
```
echo $'\n\n'Hello$'\t\t'World$'\n\n'
```

```


Hello		World


```

ARRAYS

What is an array

- An array is a linear, ordered set of values
- The values are indexed by integers



Arrays

- Arrays are indexed by integers (0,1,2,…)

```
array=(apple pear fig)
```

- Arrays are referenced with {} and []

```
$ echo ${array[*]}
apple pear fig
$ echo ${array[2]}
fig
$ echo ${#array[*]}
3
```

Arrays vs. Variables

- Arrays are actually just an extension of variables

```
$ var1=apple
$ echo $var1
apple
$ echo ${var1[0]}
apple
$ echo ${var1[1]}

$ echo ${#var1[*]}
1
$ var1[1]=pear
$ echo ${#var1[*]}
2
```

Arrays vs. Variables

- Unlike a variable, an array CAN NOT be exported to the environment
- An array CAN NOT be propagated to subshells

Arrays and Loops

- Arrays are very helpful with loops

```
for i in ${array[*]} ; do echo $i ; done
```

```
apple
pear
fig
```

Using Arrays

- The number of elements in an array

```
num=${#array[@]}
```

- Walk an array by index

```
$ for (( i=0 ; i < ${#array[@]} ; i++ ))
> do
> echo $i: ${array[$i]}
> done
0: apple
1: pear
2: fig
```

Arrays * vs @

- * concatenates all elements when quoted

```
$ for i in ${array[*]}; do echo $i ; done
apple
pear
fig
$ for i in ${array[@]}; do echo $i ; done
apple
pear
fig
$ for i in "${array[*]}"; do echo $i ; done
apple pear fig
$ for i in "${array[@]}"; do echo $i ; done
apple
pear
fig
```

Careful thinking

- Add one more tricky element

```
array+=("blood orange")
```

- See what happens

```
for i in ${array[*]}; do echo $i ; done
for i in ${array[@]}; do echo $i ; done
for i in "${array[*]}"; do echo $i ; done
for i in "${array[@]}"; do echo $i ; done
```

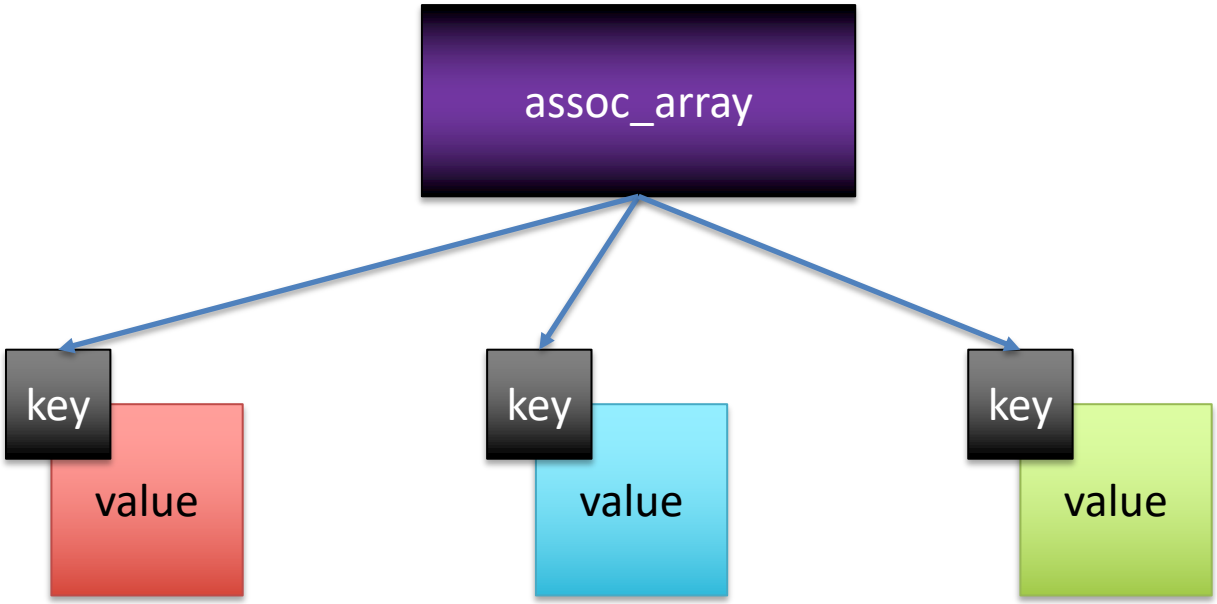
examples/arrays/arrays.sh

```
declare -a array
array=(apple pear fig)

echo There are ${#array[*]} elements in the array

echo Here are all the elements in one line:
${array[*]}

echo Here are the elements with their indices:

for (( i=0 ; i < ${#array[@]} ; i++ ))
do
  echo "  $i: ${array[$i]}"
done
```

ASSOCIATIVE ARRAYS

Associative Arrays

- Unordered, indexed collection of values



Associative Arrays

- Indexed by strings instead of integers, requires `declare -A`

```
declare -A assoc_array=([huey]=red [dewie]=blue
[louie]=green)
```

- Associative arrays have *keys* and *values*

```
$ echo ${assoc_array[huey]}
red
$ echo ${assoc_array[dewie]}
blue
$ echo ${#assoc_array[*]}
3
```

Associative Arrays

- Values are displayed by default

```
$ echo ${assoc_array[*]}
green blue red
```

- Keys require prefixing !

```
$ echo ${!assoc_array[*]}
louie dewie huey
```

- Ordering is **lost**

examples/arrays/assoc_array.sh

```
declare -A assoc_array=([huey]=red [dewie]=blue
[louie]=green)

for key in ${!assoc_array[@]}
do
  echo key: $key, value: ${assoc_array[$key]}
done
```

Associative Array

- If ordering is important, will need to maintain both regular and associative array

```
$ ordered_array=(huey dewie louie)
$ for i in ${ordered_array[@]} ; do
> echo key: $i, value: ${assoc_array[$i]}
> done
key: huey, value: red
key: dewie, value: blue
key: louie, value: green
```

ALIASES AND FUNCTIONS

Aliases

- A few aliases are set by default

```
$ alias
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias edit='nano'
```

- Can be added or deleted

```
$ unalias ls
$ alias ls='ls -CF'
```

- Remove all aliases

```
$ unalias -a
```

Aliases

- Aliases belong to the shell, but *NOT* the environment
- Aliases can *NOT* be propagated to subshells
- Limited use in scripts
- Only useful for interactive sessions

Aliases Within Scripts

- Aliases have limited use within a script
- Aliases are not expanded within a script by default, requires special option setting:

```
$ shopt -s expand_aliases
```

- Worse, aliases are not expanded within conditional statements, loops, or functions

Functions

- functions are a defined set of commands assigned to a word

```
function status { date; uptime; who | grep $USER; checkquota; }
```

```
$ status
Thu Aug 18 14:06:09 EDT 2016
 14:06:09 up 51 days,  7:54, 271 users,  load average: 1.12, 0.91, 0.86
user pts/128      2013-10-17 10:52 (128.231.77.30)
Mount                        Used      Quota  Percent    Files    Limit
/data:                    92.7 GB   100.0 GB   92.72%   233046  6225917
/home:                     2.2 GB     8.0 GB   27.48%     5510      n/a
```

Functions

- display what functions are set:

```
declare -F
```

- display code for function:

```
declare -f status
```

```
status ()
{
    date;
    uptime;
    who | grep --color $USER;
    checkquota
}
```

Functions

- functions can propagate to child shells using `export`

```
export -f status
```

Functions

- `unset` deletes function

```
unset status
```

Functions

- local variables can be set using `local`

```
$ export TMPDIR=/tmp
$ function processFile {
> local TMPDIR=/data/user/tmpdir
> echo $TMPDIR
> sort $1 | grep $2 > $2.out
> }
$ processFile /path/to/file string
/data/user/tmpdir
$ echo $TMPDIR
/tmp
```

Three ways to leave a function

- *nothing* (default)

```
function test1 { echo test1; }
```

- `return`

```
function test2 { echo test2; return; }
```

- `exit` (danger!)

```
function test3 { echo test3; exit; }
```

Location of function in script

- A function must be defined prior to call

```
function doSomething
{
  echo using $1
}

doSomething
```

examples/functions/function.sh

```
function throwError
{
  echo ERROR: $1
  exit 1
}

tag=$USER.$RANDOM
throwError "No can do!"
mkdir $tag
cd $tag
```

LOGIN

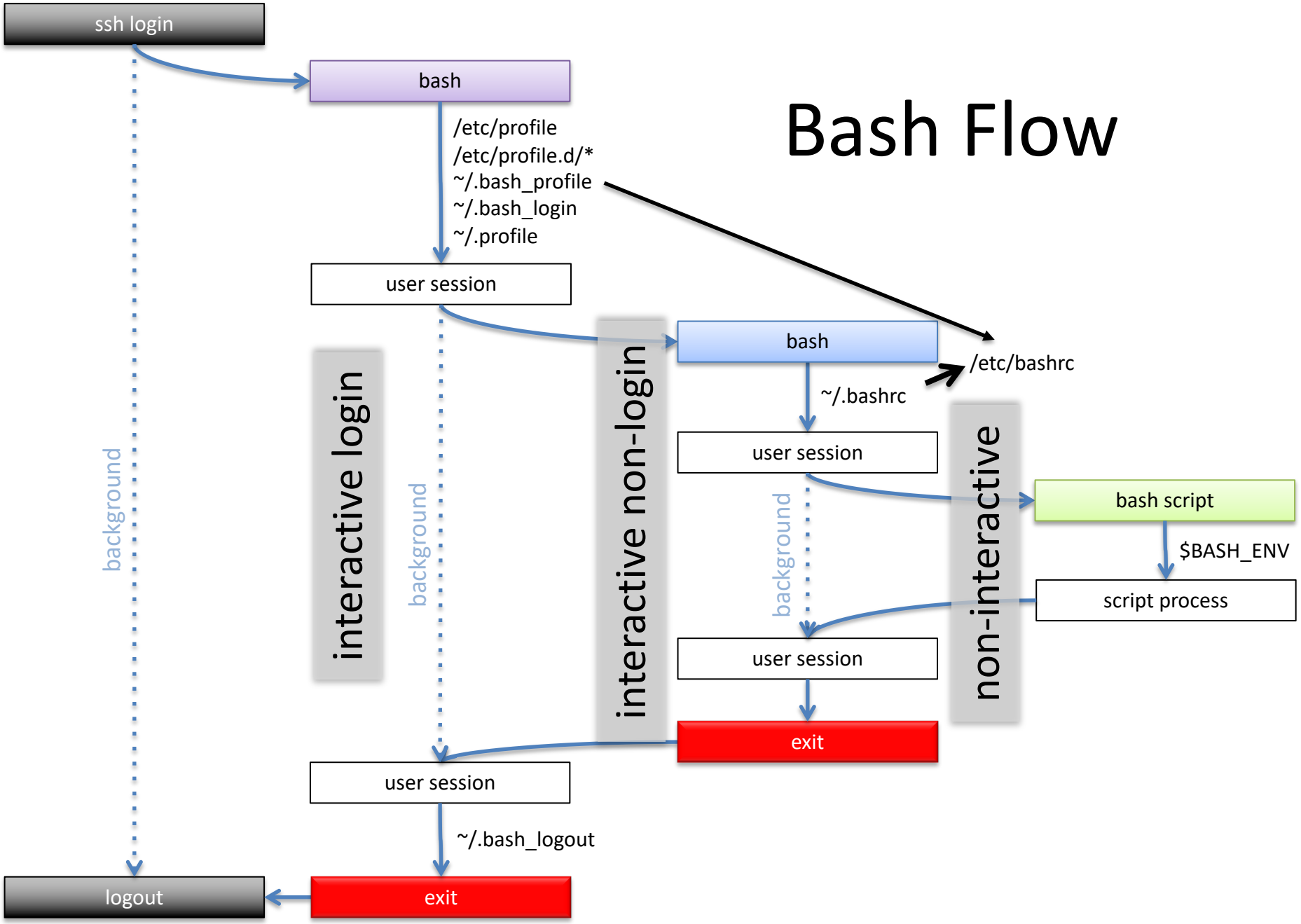
Logging In

- `ssh` is the default login client

```
$ ssh $USER@biowulf.nih.gov
```

- what happens next?

Bash Flow


ssh login
bash
/etc/profile
/etc/profile.d/*
~/.bash_profile
~/.bash_login
~/.profile
user session
bash
/etc/bashrc
~/.bashrc
user session
bash script
$BASH_ENV
script process
user session
exit
user session
~/.bash_logout
exit
logout
interactive login
interactive non-login
non-interactive
background
background
background

Logging In

- Interactive login shell (`ssh` from somewhere else)

`/etc/profile`
`~/.bash_profile`

`~/.bash_logout` (when exiting)

- The startup files are `sourced`, not executed

`source` and `.`

- `source` executes a file in the current shell and preserves changes to the environment
- `.` is the same as `source`
- Legacy from Bourne shell

~/.bash_profile

```
cat ~/.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
```

Non-Login Shell

- Interactive non-login shell (calling bash from the commandline)
- Retains environment from login shell

  `~/.bashrc`

- Shell levels seen with `$SHLVL`

```
$ echo $SHLVL
1
$ bash
$ echo $SHLVL
2
```

~/.bashrc

```
cat ~/.bashrc
```

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
   . /etc/bashrc
fi

# User specific aliases and functions
```

Non-Interactive Shell

- From a script
- Retains environment from login shell

  $BASH_ENV (if set)

- Set to a file like ~/.bashrc

Arbitrary Startup File

- User-defined (e.g. ~/.my_profile)

```
$ source ~/.my_profile
[dir user]!
```

- Can also use '.'

```
$ . ~/.my_profile
[dir user]!
```

examples/functions/function2.sh

```
source function_depot.sh
tag=$USER.$RANDOM
throwError "No can do!"
cd $tag
```

SIMPLE COMMANDS

Definitions

Command	word
Simple command / process	command arg1 arg2 …
Pipeline / Job	simple command 1 | simple command 2 |& …
List	pipeline 1 ; pipeline 2 ; …

Some command examples

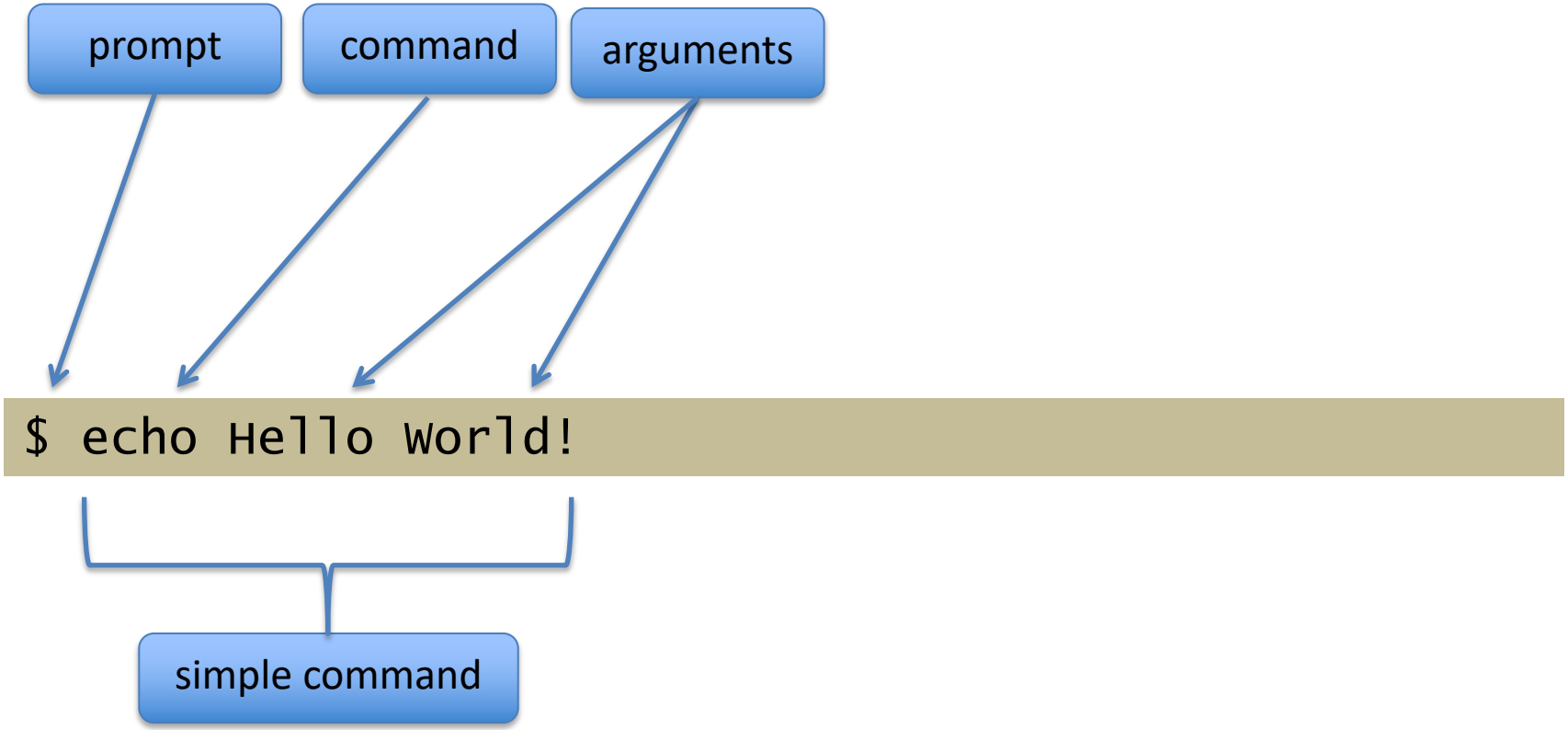
- What is the current time and date?

```
date
```

- Where are you?

```
pwd
```

Simple Command



Simple commands

- List the contents of your /home directory

```
ls -l -a $HOME
```

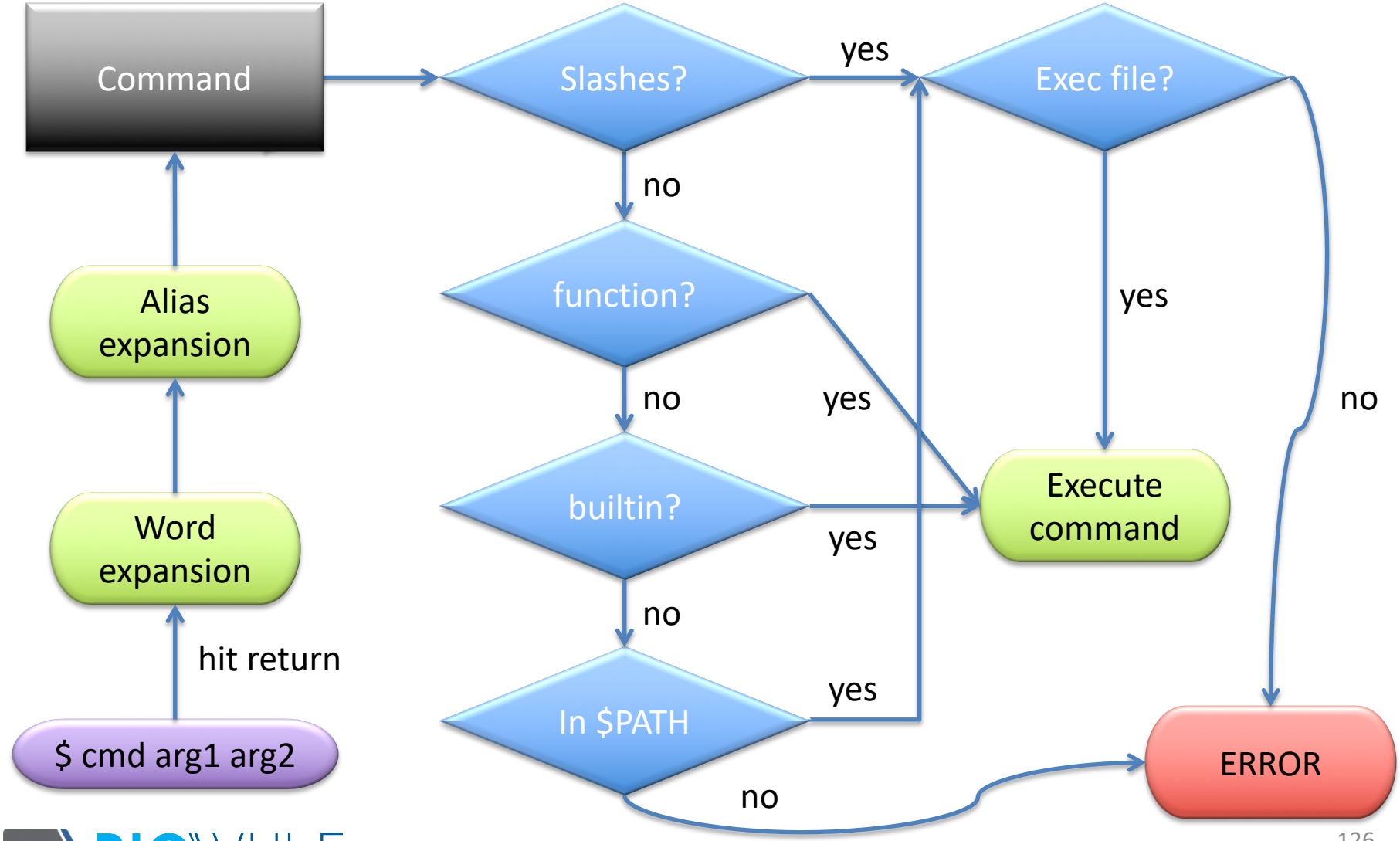
- How much disk space are you using?

```
du -h -s $HOME
```

- What are you up to?

```
ps -u $USER -o size,pcpu,etime,comm --forest
```

Command Search Tree



Process

- A *process* is an executing instance of a simple command
- Can be seen using `ps` command
- Has a unique id (*process id*, or *pid*)
- Belongs to a process group

top command

```
top - 15:51:30 up 5 days, 19:16, 240 users,  load average: 15.40, 14.51, 14.77
Tasks: 4930 total,  16 running, 4897 sleeping,  17 stopped,   0 zombie
Cpu(s):  5.0%us,  1.6%sy,  3.9%ni, 89.4%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  1058786896k total, 969744396k used, 89042500k free,    87800k buffers
Swap: 67108856k total,     2736k used, 67106120k free, 650786452k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
77108 wenxiao   39  19 63.8g  63g 1696 R 100.0  6.3 104:15.91 getAlignmentSta
98656 guptaas   39  19 4868m 1.2g  18m R 100.5  0.1   1287:38 MathKernel
254799 wenxiao  20   0 13.1g  13g 1664 R 100.0  1.3   3:48.60 getAlignmentSta
52986 wenxiao   39  19 77.0g  76g 1696 R 100.0  7.6 104:19.32 getAlignmentSta
59585 lobkovsa  39  19 24616  11m 1136 R 100.0  0.0 193:29.44 org_level_corr_
60824 hex2      39  19 28.9g  28g 4100 R 100.0  2.8   7483:20 R
245039 lobkovsa 20   0 24616  11m 1136 R 100.0  0.0  29:18.36 org_level_corr_
245092 lobkovsa 20   0 24616  11m 1136 R 100.0  0.0  29:06.75 org_level_corr_
254829 wenxiao  20   0 12.1g  11g 1664 R 100.0  1.1   3:39.88 getAlignmentSta
10184 rdmorris  20   0  531m 282m 4640 R 99.9  0.0   0:43.47 R
245080 lobkovsa 20   0 24616  11m 1136 R 99.6  0.0  29:13.30 org_level_corr_
252981 javiergc 20   0 9204m 465m  55m S 99.6  0.0   9:12.76 MATLAB
179412 sedavis  39  19 63624 7632 2876 S 11.3  0.0 442:10.21 ssh
```

ps command

```
[root@helix ~]# ps -u rdmorris -f --forest
UID         PID   PPID  C STIME TTY          TIME CMD
rdmorris 242197 242128  0 Dec12 ?        00:00:00 sshd: rdmorris@pts/107
rdmorris 242198 242197  0 Dec12 pts/107  00:00:00  \_ -bash
rdmorris 114343 114296  0 Dec12 ?        00:00:02 sshd: rdmorris@pts/251
rdmorris 114344 114343  0 Dec12 pts/251  00:00:00  \_ -bash
rdmorris  55737 114344  0 12:26 pts/251  00:00:00      \_ /bin/bash /home/rdmorris/MascotTools/M
rdmorris  55738  55737 10 12:26 pts/251  00:22:01          \_ /usr/local/R-2.13-64/lib64/R/bin/e
rdmorris  46893  46840  0 Dec12 ?        00:00:09 sshd: rdmorris@pts/112
rdmorris  46915  46893  0 Dec12 pts/112  00:00:00  \_ -bash
```

history

- The `history` command shows old commands run:

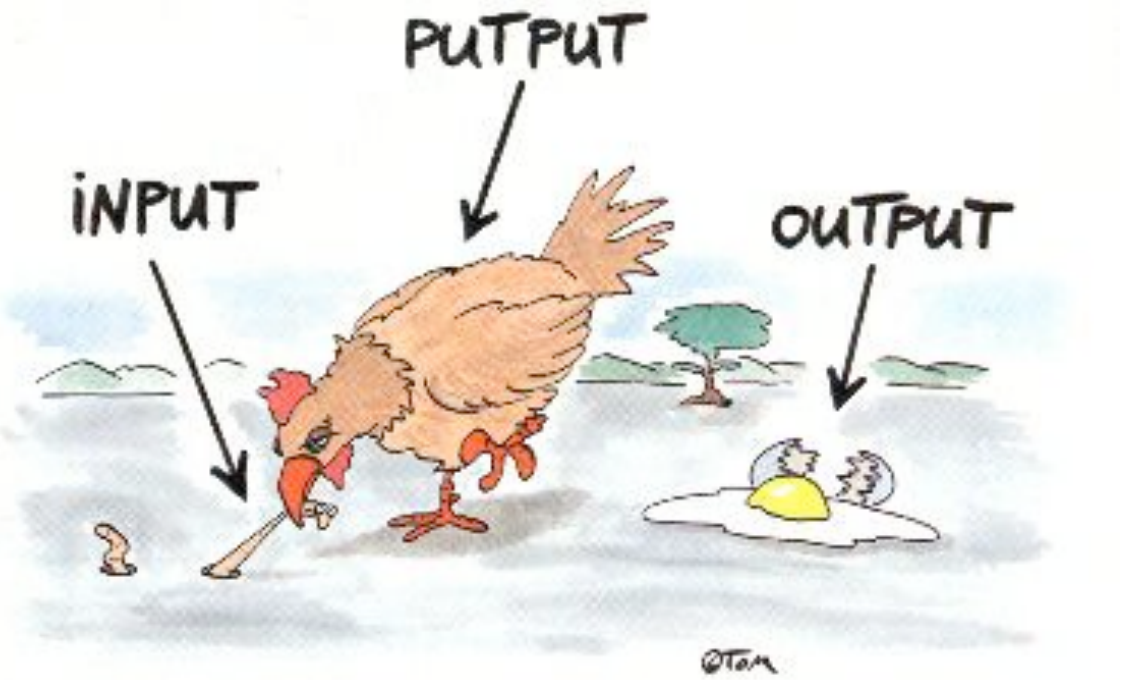
```
$ history 10
  594  12:04  pwd
  595  12:04  ll
  596  12:06  cat ~/.bash_script_exports
  597  12:06  vi ~/.bash_script_exports
  598  12:34  jobs
  599  12:34  ls -ltra
  600  12:34  tail run_tophat.out
  601  12:34  cat run_tophat.err
  602  12:54  history 10
```

history

- HISTTIMEFORMAT controls display format

```
$ export HISTTIMEFORMAT='%F %T  '
$ history 10
  596  2018-10-16 12:06:22  cat ~/.bash_script_exports
  597  2018-10-16 12:06:31  vi ~/.bash_script_exports
  598  2018-10-16 12:34:13  jobs
  599  2018-10-16 12:34:15  ls -ltra
  600  2018-10-16 12:34:21  tail run_tophat.out
  601  2018-10-16 12:34:24  cat run_tophat.err
  602  2018-10-16 12:54:38  history 10
  603  2018-10-16 12:56:14  export HISTTIMEFORMAT='%F %T  '
  604  2018-10-16 12:56:18  history 10
```

http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x1712.htm



INPUT AND OUTPUT

Redirection

- Every process has three file descriptors (file handles): STDIN (0), STDOUT (1), STDERR (2)
- Content can be redirected

```
cmd < x.in
```

Redirect file descriptor 0 from STDIN to x.in

```
cmd > x.out
```

Redirect file descriptor 1 from STDOUT to x.out

```
cmd 1> x.out 2> x.err
```

Redirect file descriptor 1 from STDOUT to x.out, file descriptor 2 from STDERR to x.err

Combine STDOUT and STDERR

```
cmd 2>&1
```

Redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (STDOUT)

- Ordering is important

Correct:

```
cmd > x.out 2>&1
```

Redirect file descriptor 1 from STDOUT to filename x.out, then redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (x.out)

Incorrect:

```
cmd 2>&1 > x.out
```

Redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (STDOUT), then redirect file descriptor 1 from STDOUT to filename x.out

Redirection to a File

- Use better syntax instead – these all do the same thing:

```
cmd > x.out 2>&1
```

~~`cmd 1> x.out 2> x.out`~~ WRONG!

```
cmd &> x.out
```

```
cmd >& x.out
```

Redirection

- Appending to a file

`cmd >> x.out` Append STDOUT to x.out

`cmd 1>> x.out 2>&1` Combine STDOUT and STDERR, append to x.out

`cmd &>> x.out` Alternative for concatenation and appendation

~~`cmd >>& x.out`~~ WRONG!

Named Pipes/FIFO

- Send STDOUT and/or STDERR into temporary file for commands that can't accept ordinary pipes

```
$ ls /home/$USER > file1_out
$ ls /home/$USER/.snapshot/Weekly.2018-12-30* >
file2_out
$ diff file1_out file2_out > diff.out
$ rm file1_out file2_out
$ cat diff.out
```

Named Pipes/FIFO

- FIFO special file can simplify this
- You typically need multiple sessions or shells to use named pipes

```
[sess1]$ mkfifo pipe1
[sess1]$ ls /home/$USER > pipe1
```

```
[sess2]$ cat pipe1
```

Named Pipes/FIFO

- Can be used to consolidate output without appending to a file

```
$ mkfifo pipe1 pipe2 pipe3
$ echo one > pipe1 &
$ echo two > pipe2 &
$ echo three > pipe3 &
```

```
$ cat pipe*
```

Process Substitution

- The operators <( ) and >() can be used to create transient named pipes
- AKA process substitution

```
$ diff <(ls /home/$USER) <(ls
/home/$USER/.snapshot/Weekly.2018-12-30*)
$ cat /usr/share/dict/words > >(grep zag)
```

- The operators '> >()' are equivalent to '|'

examples/pipes/fifo.sh

```
diff <(zcat examples/pipes/webfiles.new.gz | sort) <(zcat
examples/pipes/webfiles.old.gz | sort)

# equivalent to:
#diff <(zcat examples/pipes/webfiles.new.gz > >(sort) )
<(zcat examples/pipes/webfiles.old.gz > >(sort) )
```

PIPELINES AND JOBS

Pipeline

- STDOUT of each simple command is passed as STDIN of the next command

```
ps -ef | grep sh | head -n 3
```

Pipeline

- STDERR can be combined with STDOUT

```
$ ls emptydir | grep -c 'No such file'
ls: cannot access emptydir: No such file or
directory
0
$ ls emptydir 2>&1 | grep -c 'No such file'
1
$ ls emptydir |& grep -c 'No such file'
1
```

examples/pipes/pipes.sh

```
# The file genome_stuff.csv is a comma-delineated output
# file from an unamed application.  You want to pull out
# the data related to chrX, converting the delineator to
# a tab character, then sort based on the SNP id, if
# present.  Also, you want to maintain the column headers
# (the topmost line).

# Note the use of line continuation markers and named
# pipes.

input=examples/pipes/genome_stuff.csv
cat <(cut -d',' -f1,2,22-33 $input | head -1 \
  | tr ',' $'\t') <(cut -d',' -f1,2,22-33 $input \
  | grep chrX |  tr ',' $'\t' | sort -k3)
```

Job Control

- A *job* is another name for pipeline

```
echo Hello World! > x | cat x | grep o
```

- Each simple command is a process

Foreground and Background

- A job (pipeline) runs in the foreground by default
- *Asynchronous* jobs are run in background (in parallel, fire and forget)

```
sleep 5 &
```

- Requires single ‘&’ at the end
- Background jobs run in their own shell
- Exit status is not available

wait

- The `wait` command does just that

```
sleep 9 &
sleep 7 &
sleep 5 &
wait
```

- useful for consolidation or summary steps

Job Control

- The shell itemizes jobs by number

```
$ sleep 10 &
[1] 8683
$ sleep 10 &
[2] 8684
$ sleep 10 &
[3] 8686
$ jobs
[1]   Running                 sleep 10 &
[2]-  Running                 sleep 10 &
[3]+  Running                 sleep 10 &
$
[1]   Done                    sleep 10
[2]-  Done                    sleep 10
[3]+  Done                    sleep 10
```

Job Control

- A job can be moved from foreground to background with [CTRL-z] and bg

```
$ step1.sh | step2.sh | grep normal
[CTRL-z]
[1]+  Stopped                 step1.sh | step2.sh ..
bg
[1]+ step1.sh | step2.sh ..
```

- Can be brought back with fg

```
$ jobs
[1]+  Running                 step1.sh | step2.. &
$ fg
step1.sh step2.sh ...
```

SUBSHELLS

Shells and Subshells

- A shell is defined by a set of variables, options, and a current working directory (CWD)
- Subshells (child shells) inherit the environment and CWD
- Changes to the environment and CWD are ***not*** propagated back to the parent

Talking To Yourself

```
$ pwd
/home/user
$ export NUMBER=5
$ bash
$ ((NUMBER++))
$ echo $NUMBER
6
$ cd /scratch
$ exit
$ echo $NUMBER
5
$ pwd
/home/user
```

in a subshell

examples/subshells/subshell.sh

```
pwd
for i in {1..4} ; do
(
  BASE=/tmp/$i.$RANDOM
  mkdir $BASE ; echo -n "OLD DIR: " ; pwd
  cd $BASE ; echo -n "NEW DIR: " ; pwd
  sleep 2
  rmdir $BASE
)
done
pwd
```

examples/subshells/exit.sh

```
(
  echo this is running in a subshell
  echo exiting now
  exit
)
echo intermission
{
  echo this is running in the current shell
  echo exiting now
  exit
}
echo all done!
```

COMMAND LISTS

Command List

- Sequence of one or more jobs separated by ‘ ; ’, ‘ & ’, ‘ && ’, or ‘ | | ’.
- Simple commands/pipelines/jobs are run sequentially when separated by ‘ ; ’

```
echo 1 > x ; echo 2 > y ; echo 3 > z
```

```
date ; sleep 5 ; sleep 5 ; sleep 5 ; date
```

Command List

- Sequential command list is equivalent to

```
echo 1 > x
echo 2 > y
echo 3 > z
```

- or

```
date
sleep 5
sleep 5
sleep 5
date
```

Command List

- Asynchronously when separated by '&'

```
echo 1 > x & echo 2 > y & echo 3 > z &
```

- `wait` pauses until all background jobs complete

```
date ; sleep 5 & sleep 5 & sleep 5 & wait ; date
```

Command List

- Asynchronous command list is equivalent to

```
echo 1 > x &
echo 2 > y &
echo 3 > z &
```

- or

```
date
sleep 5 &
sleep 5 &
sleep 5 &
wait
date
```

Grouped Command List

- To execute a list of sequential pipelines in the background, or to pool STDOUT/STDERR, enclose with ‘ () ’ or ‘ {} ’

```
$ ( cmd 1 < input ; cmd 2 < input ) > output &
[1] 12345
$ { cmd 1 < input ; cmd 2 < input ; } > output &
[2] 12346
```

- STDIN is NOT pooled, but must be redirected with each command or pipeline.

Grouped Command List

- `'{ }'` runs in the current shell
- `'( )'` runs in a child/sub shell

Grouped Command List Details

```
(sleep 3 ; sleep 5 ; sleep 8)
```

```
user 6299  6283  0 09:29 ?        00:00:00 sshd: user@pts/170
user 6303  6299  0 09:29 pts/170  00:00:00  \_ -bash
user 5040  6303  0 16:33 pts/170  00:00:00      \_ -bash
user 5100  5040  0 16:33 pts/170  00:00:00          \_ sleep 8
```

```
{ sleep 3 ; sleep 5 ; sleep 8 ; }
```

```
user 6299  6283  0 09:29 ?        00:00:00 sshd: user@pts/170
user 6303  6299  0 09:29 pts/170  00:00:00  \_ -bash
user 6980  6303  0 16:35 pts/170  00:00:00      \_ sleep 8
```

Grouped Command List Details

- Grouped command list run in background are identical to ' ( ) '

```
(sleep 3 ; sleep 5 ; sleep 8) &
```

```
{ sleep 3 ; sleep 5 ; sleep 8 ; } &
```

```
user  6299  6283  0 09:29 ?        00:00:00 sshd: user@pts/170
user  6303  6299  0 09:29 pts/170  00:00:00  \_ -bash
user 17643  6303  0 16:50 pts/170  00:00:00      \_ -bash
user 17696 17643  0 16:50 pts/170  00:00:00          \_ sleep 8
```

Grouped Command List Details

```
(sleep 3 & sleep 5 & sleep 8 &)
```

```
user 6299  6283  0 09:29 ?        00:00:00 sshd: user@pts/170
user 6303  6299  0 09:29 pts/170  00:00:00  \_ -bash
user 7891     1  0 16:37 pts/170  00:00:00 sleep 8
user 7890     1  0 16:37 pts/170  00:00:00 sleep 5
user 7889     1  0 16:37 pts/170  00:00:00 sleep 3
```

```
{ sleep 3 & sleep 5 & sleep 8 & }
```

```
user 6299  6283  0 09:29 ?        00:00:00 sshd: user@pts/170
user 6303  6299  0 09:29 pts/170  00:00:00  \_ -bash
user 9165  6303  0 16:39 pts/170  00:00:00      \_ sleep 3
user 9166  6303  0 16:39 pts/170  00:00:00      \_ sleep 5
user 9167  6303  0 16:39 pts/170  00:00:00      \_ sleep 8
```

examples/parallel/geography_serial.sh

```
tag=$RANDOM
zcat geography.txt.gz > $tag.input

function reformat {
  cat <(grep "^$1" $tag.input | sort -nk2) <(echo "--------") >
$tag.file$2
}

reformat "Africa" 1
reformat "Asia" 2
reformat "Europe" 3
reformat "North America" 4
reformat "Oceania" 5
reformat "South America" 6
```

examples/parallel/geography_parallel.sh

```
tag=$RANDOM
zcat geography.txt.gz > $tag.input

function reformat {
  cat <(grep "^$1" $tag.input | sort -nk2) <(echo "--------") >
$tag.file$2
}

{
  reformat "Africa" 1 &
  reformat "Asia" 2 &
  reformat "Europe" 3 &
  reformat "North America" 4 &
  reformat "Oceania" 5 &
  reformat "South America" 6 &
}
wait
```

CONDITIONAL STATEMENTS

Exit Status

- A process returns an exit status (0-255)
- 0 = success (almost always)
- 1 = general error, 2-255 = specific error
- Stored in `$?` parameter

```
cat /var/audit
```

```
cat: /var/audit: Permission denied
```

```
echo $?
```

```
1
```

Exit Status

- Exit status value is not very helpful

```
ls /zzz
```

```
ls: cannot access /zzz: No such file or directory
```

```
echo $?
```

```
2
```

test

- `test` is a general conditional statement

```
ls ~/.bashrc
```

```
/home/user/.bashrc
```

```
echo $?
```

```
0
```

```
test -e ~/.bashrc
echo $?
```

```
0
```

test

- take action based on results of `test`

```
test -f ~/.bashrc && tail -n 1 ~/.bashrc
```

```
# User specific aliases and functions
```

Conditional Statement Run-on

```
$ ( ( test -e ~/.bashrc && echo "~/.bashrc exists" ) || ( test -e ~/.bash_profile && echo "~/.bash_profile exists" ) || ( echo "You may not be running bash" ) )
/home/user/.bashrc
```

- kind of hard to swallow

if .. elif .. else .. fi

```
if test-commands ; then
  consequent-commands
elif more-test-commands ; then
  more-consequents
else
  alternate-consequents
fi
```

if

```
if test -e ~/.bashrc
then echo "~/.bashrc exists"
elif test -e ~/.bash_profile
then echo "~/.bash_profile exists"
else echo "You may not be running bash"
fi
```

```
~/.bashrc exists
```

examples/conditionals/indent.sh

- `if` statements in scripts are made cleaner with indents and `then` on the same line

```
#!/bin/bash
if test -e ~/Desktop; then
  clear
  echo "You are expecting a desktop"
elif test -f ~/.bashrc; then
  cd ~
  echo "No desktop here"
fi
```

[and [[

- [is an executable file

```
which [
```

```
/usr/bin/[
```

- [[is a builtin
- both substitute for `test`

```
[[ -d /tmp ]] && echo "/tmp exists"
```

Conditionals: `test` and `[[ ]]`

- `test` has many different primaries and operators

```
help test
```

- Can be used for files (directories) or comparing strings and variables
- Both `test` and `[[ ]]` are builtins, `[` is a program, and `test` is also a program

```
help [[
man test
```

General `if` Statements

- `if` evaluates exit status, so it can be used with any command

```
if grep -q bashrc ~/.bash_profile ; then
  echo yes
fi
```

- The inverse is possible with ‘ ! ’

```
if ! cat /zzz &> /dev/null ; then
  echo empty
fi
```

General if Statements

- Identical to grouped command list

```
[[ -e /tmp ]] && echo "/tmp exists"
```

```
/tmp exists
```

```
[[ $PATH == $HOME ]] || echo "not equal"
```

```
not equal
```

Boolean Operators

- Multiple `if` statements in series

```
$ if [ -e file1 ] && [ -e file2 ] ; then echo both ; fi
```

Conditional Command List

- A list can execute sequence pipelines conditionally
- Execute cmd2 if cmd1 was successful

```
ls ~/.bashrc && tail ~/.bashrc
```

- Execute cmd2 if cmd1 was *NOT* successful

```
cd /bogus || echo "bogus is bogus"
```

- Conditional lists can be grouped using single parentheses:

```
( ls file.txt || ( echo 'F' && touch file.txt ) )
```

`if` Format

- Must be a space between `test` statement and brackets

```
if [[-e file]] ; then echo file exists ; fi
```

```
-bash: [[-e: command not found
```

```
if [[ -e file]] ; then echo file exists; fi
```

```
-bash: syntax error in conditional expression: unexpected
token `;'
-bash: syntax error near `;'
```

Booleans for Math

- Can use math as conditionals in multiple tests

```
a=4
if ((a==4)) ; then echo yes ; else echo no ; fi
```

```
yes
```

```
if (( (a-5) == 0 )) ; then echo yes ; else echo no ; fi
```

```
no
```

```
if (( a < 10 )) ; then echo yes; else echo no ; fi
```

```
yes
```

Booleans for Math

- Be careful with equal signs!

```
a=4
echo $a
```

```
4
```

```
if (( a = 5 )) ; then echo yes ; else echo no ; fi
```

```
yes
```

```
echo $a
```

```
5
```

PATTERN MATCHING

Pattern Matching

- There are three types of matches, standard glob, extended glob, and extended regular expression (regex)
- *, ?, [?-?],[^?],[[:CLASS:]] are standard glob
- ?(), *(), +(), @(), !() are extended glob
- +, {N}, (), \, ^…$ are allowed for regex

Pattern Matching -- Glob

- * : match any string
- ? : match any single character
- [?-?] : match range of characters
- [!?] or [^?] : not match character

```
mkdir trash && cd trash
touch {{1..9},{a..z}}
ls [a-e1-4]
```

```
1  2  3  4  a  b  c  d  e
```

Character Classes

- `[[:CLASS:]]` can be included with pattern matching

```
touch {1..9} y{1..9} z{1..9}
ls [[:alpha:]]4
```

```
y4 z4
```

- full list

`alnum`	`cntrl`	`print`	`word`
`alpha`	`digit`	`punct`	`xdigit`
`ascii`	`graph`	`space`	
`blank`	`lower`	`upper`	

Character Classes

- Using [[:CLASS:]]

```
touch %5 A3 x.9
ls [[:upper:]]*
```

```
A3
```

```
ls *[[:punct:]]*
```

```
%5 x.9
```

```
$ ls [![:lower:]]*
```

```
%5 A3
```

Extended Glob

- Enabled with `shopt -s extglob`

```
a=939
[[ $a == +([0-9]) ]] && echo is a number
```

```
is a number
```

Extended Glob Matching

- Multiples

```
touch apple pear fig pineapple plum orange
ls p+([a-z])
```

```
pear  pineapple  plum
```

```
ls ?(pine)apple
```

```
apple  pineapple
```

```
ls @(p|f)*([a-z])
```

```
fig  pear  pineapple  plum
```

Pattern Matching Conditionals

- == or != for glob

```
x=apple
[[ $x == apple ]] && echo this is an apple
[[ $x != pear ]] && echo this is NOT an apple
[[ $x == ?(pine)apple ]] && echo some kind of apple
```

Pattern Matching Conditional

- Test if a string matches a pattern
- Is a variable a number?

```
a=939
[[ $a == [0-9][0-9][0-9] ]] && echo is a number
```

```
is a number
```

- Does a string end with an alphabetical character?

```
a=939
if [[ $a == *[[:alpha:]] ]] ; then echo yes
else echo no ; fi
```

```
no
```

Regular Expressions

- Allow more precise matching
- Only enabled with =~ operator

```
x=fig
[[ $x =~ ^f[io]g$ ]] && echo this is probably fig
[[ ! $x =~ ^d[io]g$ ]] && echo huh?
```

- Allows capturing of submatches

```
y=123.456.7890
[[ $y =~ ^([0-9]+)\.[0-9]{3}\.[0-9]{4}$ ]] &&
  echo ${BASH_REMATCH[1]}
```

```
123
```

$BASH_REMATCH

- Can capture a submatch using () and $BASH_REMATCH array
- `examples/matching/rematch.sh`

```
str="The quick red fox jumped over the lazy brown dog"

if [[ $str =~ quick\ (.*)\ fox ]] ; then
  echo Total match: ${BASH_REMATCH[0]}
  echo Submatch:    ${BASH_REMATCH[1]}
fi
```

```
Total match: quick red fox
Submatch: red
```

examples/matching/matching_number.sh

```
echo -n "Type in something : "
read response
echo "You said: $response"

if [[ "$response" == [0-9][0-9][0-9] ]] ; then
  echo "This is a three-digit number!"
elif $( echo "$response" | grep -q "^[0-9]\+$" ) ; then
  echo "This is a integer!"
elif $( echo "$response" | grep -q "^[0-9]\+\.[0-9]*$" ) ; then
  echo "This is a floating point number!"
elif $( echo "$response" | egrep -q '^(\+|-)?[0-9]+\.?[0-9]+(e\+[0-9]+|e-
[0-9]+|e[0-9]+)?$' ) ; then
  echo "This is scientific notation!"
else
  echo "I don't know what to say..."
fi
```

examples/matching/matching_number2.sh

```
#!/bin/bash
shopt -s extglob
case $1 in
  [0-9][0-9][0-9]) echo three-digit number ;;
  *([-\+])+([0-9]) ) echo integer ;;
  *([-\+])*([0-9]).+([0-9]) ) echo floating point ;;
  *([-\+])*([0-9]).+([0-9])e*([-\+])+([0-9]))
    echo scientific notation ;;
  *) echo IDK ;;
esac
```

examples/pipes/pipes2.sh

```
cat <(zgrep ^chrX examples/pipes/ins.bed.gz | \
    awk '{if ($2 > 100000000 && $2 < 100500000) \
    print "INS\t" $0}') \
  <(zgrep ^chrX examples/pipes/del.bed.gz | \
    awk '{if ($2 > 100000000 && $2 < 100500000) \
    print "DEL\t" $0}') \
  <(zgrep ^chrX examples/pipes/jun.bed.gz | \
    awk '{if ($2 > 100000000 && $2 < 100500000) \
    print "JUN\t" $0}') | \
  sort -nk3,4
```

case … esac

- Fixed pattern matching, small selection

```
case word in
   pattern )
      commands ;;
   pattern )
      commands ;;
esac
```

examples/conditionals/case.sh

```
animal=$1
[[ -n $animal ]] || { echo What animal?; exit; }

case $animal in
  dog)
    echo "this is a dog” ;;
  cat)
    echo "this is a cat” ;;
  fish)
    echo "this is a fish” ;;
  *)
    echo "This is not on my list” ;;
esac
```

LOOPS

for .. do .. done

```
for name in words
do
   commands
done

for (( expr1 ; expr2 ; expr3 ))
do
   commands
done
```

Loops - for

- for is used to step through multiple words

```
for i in apple pear fig ; do echo $i ; done
```

```
apple
pear
fig
```

Loops - for

- With brace expansion:

```
for i in {1..5}; do echo $i ; done
```

```
1
2
3
4
5
```

Loops - for

- With arrays

```
array=(moe larry curly)
for i in "${array[@]}" ; do echo $i ; done
```

```
moe
larry
curly
```

Walk Through Directories

- Find all files in /home and count how many lines are in each file:

```
for file in $(ls -a ~/); do
[[ -f ~/$file ]] && wc -l ~/$file
done
```

```
2 .bash_logout
12 .bash_profile
8 .bashrc
20 .emacs
11 .kshrc
34 .zshrc
```

C-style `for` loop

- `for` can be used for integer traversal

```
for (( i=1 ; i < 1000 ; i=i+i ))
do
echo $i
done
```

```
1
2
4
8
16
32
64
128
256
512
```

while .. do .. done

```
while test-commands
do
   consequent-commands
done
```

until .. do .. done

```
until test-commands
do
   consequent-commands
done
```

Loops - `until`

- Until uses `test` commands
- Handy with math

```
a=0
until [[ $a -gt 10 ]] ; do echo $a ; ((a=a+3)) ; done
```

```
0
3
6
9
```

- Can be used with semaphore files

```
$ until [[ -e stop.file ]] ; do sleep 60 ; done
```

Loops - while

- while is the reverse of until

```
a=1
while [[ $a -le 5 ]] ; do echo $a ; ((a++))  ; done
```

```
1
2
3
4
5
```

continue

- Can be used to skip to next element

```
for a in {1..5}; do
  if [[ $a == 2 ]]; then
    continue;
  fi
  echo $a
done
```

```
1
3
4
5
```

break

- Can be used to end loops or skip sections

```
a=1
while [[ 1 ]] ; do
  echo $a
  if (( a > $(date +%s) )) ; then
    break
  fi
  (( a=a+a ))
done
```

```
1
2
4
…
2147483648
```

examples/loops/stepping.sh

```
iterations=5
iterations_remaining=$iterations
seconds_per_step=2
while (( $iterations_remaining > 0 ))
do
  if (( ($(date +%s) % $seconds_per_step) == 0 ))
  then
    echo -n "$iterations_remaining : "
    ( date ; sleep 10 ) &
    ((iterations_remaining--))
  fi
  sleep 1
done
```

examples/loops/genome_nonsense.sh

```
#!/bin/bash
BASE=gn.$RANDOM
for i in {1..22} X Y M ; do
  label=$i
  if [[ $i == [[:digit:]] ]]; then
    label=$(printf '%02d' $i)
  fi
  [[ -f $BASE/$label/trial.out ]] && break
  [[ -d $BASE/$label ]] || mkdir -p $BASE/$label
  pushd $BASE/$label 2>&1 > /dev/null
  echo Running chr${i}_out
  # actually do something, not this
  touch trial_chr${i}.out
  popd >& /dev/null
done
```

ACCESSORIZE

Interactive Script Input

- use the `read` command to interactively get a line of input:

```
echo -n "Type in a number: "
read response
echo "You said: $response"
```

```
$ bash script.sh
Type in a number: 4
You said: 4
$
```

Using `while` and `read` in a script

- `examples/loops/readfile.sh`

```
while read var
do
  if [[ $var == "exit" ]]
  then
    break
  fi
  echo $var
  # do something else with $var
done
```

Walk a file with read

- examples/loops/repeat_file.sh

```
#!/bin/bash
while IFS='' read -r line || [[ -n "$line" ]]; do
    echo "Text read from file: $line"
done < "$1"
```

```
bash examples/loops/repeat_file.sh input/loop_input.txt
```

```
Text read from file: These are fruits:
Text read from file:
Text read from file: apple
Text read from file: pear
Text read from file: fig
Text read from file: banana
```

examples/options/basic.sh

```
mem="1g"
tmpdir="/tmp"
usage="$0 [-m] [-t] [-h]

    -m mem (default 1g)
    -t tmpdir (default /tmp)
    -h show this help menu"

while getopts "m:t:h" flag; do
  case "$flag" in
    m) mem=$OPTARG ;;
    t) tmpdir=$OPTARG ;;
    h) echo "$usage" ; exit 0 ;;
    ?) echo "$usage" ; exit 1 ;;
  esac
done
```

examples/options/select.sh

```
#!/bin/bash

echo "Pick a fruit, any fruit"

select fname in apple pear fig plum pineapple orange
do
  break
done

echo "You picked $fname"
```

```
$ ./select.sh
Pick a fruit, any fruit
1) apple       3) fig         5) pineapple
2) pear        4) plum        6) orange
#? 4
You picked plum
```

STUPID PET TRICKS

Extended Math

- Use bc instead:

```
number=$(echo "scale=4; 17/7" | bc)
echo $number
```

```
2.4285
```

```
x=8
y=3
z=$(echo "scale = 3; $x/$y" | bc)
echo $z
```

```
2.666
```

- Might as well use perl or python instead…

Fun with substring expansion

```
$ string=01234567890abcdefgh
$ echo ${string:7}
7890abcdefgh
$ echo ${string:7:0}

$ echo ${string:7:2}
78
$ echo ${string:7:-2}
7890abcdef
$ echo ${string: -7}
bcdefgh
$ echo ${string: -7:0}

$ echo ${string: -7:2}
bc
$ echo ${string: -7:-2}
bcdef
```

Substring expansion with arrays

```
$ array=(0 1 2 3 4 5 6 7 8 9 0 a b c d e f g h)
$ echo ${array[@]:7}
7 8 9 0 a b c d e f g h
$ echo ${array[@]:7:2}
7 8
$ echo ${array[@]: -7:2}
b c
$ echo ${array[@]: -7:-2}
bash: -2: substring expression < 0
$ echo ${array[@]:0}
0 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
$ echo ${array[@]:0:2}
0 1
$ echo ${array[@]: -7:0}
```

Parse fasta file

```
$ cat zz.fasta
>gi|bogus
abcde
>gi|nonsense
xyz123
>gi|something else
ABCDEF123

$ while read -d '>' block ; do
array[${#array[@]}]="$block" ; done < <(cat zz.fasta
<(echo '>')  )

# This sets the zeroth element as blank.  Use this command
to remove the zeroth element

$ array=("${array[@]:1}")
```

grep

- Find only that which matched

```
ls -l /home | grep -o "Dec .[[:digit:]] ..:.."
```

- Recursive search

```
grep -R -l bash *
```

- Use a list of matches from a file

```
grep -f examples/grep/complex.grep -h examples/matching/*
```

egrep

- Extended `grep` allows additional metacharacters like `+`, `?`, `|`, `{}` and `()`

```
egrep
'ftp/phase3/data/[^\/]+/exome_alignment/[^\/]+\.mapped\.ILLUMINA\
.bwa\.[^\/]+\.exome\.[^\/]+\.bam'
/fdb/1000genomes/ftp/current.tree
```

- `{}` match number of times

```
ls -l /home | egrep --color ' [0-9]{7} [[:alpha:]]{3} [0-
9]{2}'
```

find

- Find large files

```
find . -type f -size +1M
```

- Find recently modified files

```
find ~ -mmin -100
```

find

- Find and update files and directories in a shared directory

```
$ find . -user [you] ! -group [grp]
$ find . -user [you] ! -group [grp] -exec chgrp
[yourgroup] {} \;

$ find . -user [you] -perm /u+r ! -perm /g+r
$ find . -user [you] -perm /u+r ! -perm /g+r -exec chmod
g+r {} \;

$ find . -user [you] -perm /u+x ! -perm /g+x
$ find . -user [you] -perm /u+x ! -perm /g+x -exec chmod
g+x {} \;
```

find

- Find newest and oldest files in a directory tree

```
find . -type f -printf '%Tc %p\n' | sort -nr | head
find . -type f -printf '%Tc %p\n' | sort -nr | tail
```

- Find biggest and smallest files

```
find . -type f -printf '%k %p\n' | sort -nr | head
find . -type f -printf '%k %p\n' | sort -nr | tail
```

- Find empty directories

```
find . -type d -empty -printf "%Tc %p\n"
```

sort

- input/file_for_sorting.txt

```
sort input/file_for_sorting.txt
```

- Sort by alternative column (key)

```
sort -k1 input/file_for_sorting.txt
```

- Multi-column sort

```
sort -k3 -k1 -k4 input/file_for_sorting.txt
```

sort

- Sort file, locking top line

```
cat input/file_for_sorting.txt | (read -r; \
  printf "%s\n" "$REPLY"; sort -k3 -k1 -k4)
```

- Sort as recognized data type

```
cat input/file_for_sorting.txt | (read -r; \
  printf "%s\n" "$REPLY"; sort -k1M -k2n -k3r)
```

sort

- Deal with missing values

```
cat input/file_for_sorting.txt | (read -r; \
  printf "%s\n" "$REPLY"; sort -t $'\t' -k6h)
```

- Sort very large file with memory limit

```
sort /fdb/annovar/current/hg19/hg19_ljb26_cadd.txt -n -k6
-S 25%
```

- `man sort`

Spaces in file names

- Replace newline with null character

```
for i in $(ls -1) ; do
  wc -l $i
done
find . -type f -print0 | xargs -0 wc -l
```

- Use IFS= and double quotes

```
while IFS= read -r line; do
  du -hs "$line"
done < <(ls -1)
```

Tab delimited files

- Rearrange order of columns using awk

```
awk 'BEGIN { FS = "\t" } ; {print $3FS$2FS$1}' \
  input/file_for_sorting.txt
```

- Rearrange order using `while … read`

```
while IFS=$'\t' read -r field1 field2 field3 others
do
  echo ${field2}$'\t'${field3}$'\t'${field1}
done < input/file_for_sorting.txt
```

Use of IFS

- Parse string into words

```
IFS=: p=($PATH)
for i in ${p[@]}; do echo $i ; done
```

```
/usr/bin
/bin
/usr/local/bin
/home/user/bin
```

complete

- Tab-complete possible arguments

```
function do_something { echo You picked $1; }
array=(pineapple apple pear fig banana)
complete -W "${array[*]}" do_something
```

```
$ do_something <tab><tab>
apple       banana      fig         pear        pineapple
$ do_something a<tab><tab>
apple
$ do_something p<tab><tab>
pear        pineapple
```

complete

- Specify file type allowed

```
complete -f -X '!*.tsv' do_something
```

```
$ do_something input/<tab><tab>
database.tsv  exp1.tsv      exp2.tsv      exp3.tsv
final.tsv
```

- man complete

hash

- Similar to history table
- Hard-code executable paths outside of $PATH
- Keep a count of executable calls

```
$ hash
$ hash -l
$ hash -p /path/to/command name
$ hash -d name
$ hash -r
```

- The hash table is NOT propagated to subshells

readarray

- Read a file into an array

```
$ cat file_to_be_mapped.txt
This is line one
The second line
Finally line 3
$ readarray zzz < file_to_be_mapped.txt
$ echo ${zzz[1]}
The second line
```

Sort elements of an array

- Use `xargs`

```
array=(apple pear fig plum pineapple orange)
echo ${array[@]} | xargs -n 1 | sort | xargs
```

```
apple fig orange pear pineapple plum
```

- Unique sorted elements

```
array=(3 0 4 7 0 8 4 5 9 6 3 1 9 0 2 3 4)
echo ${array[@]} | xargs -n 1 | sort -u | xargs
```

```
0 1 2 3 4 5 6 7 8 9
```

Other uses for `xargs`

- Print wc output neatly

```
find . -type f -print0 | xargs -0 wc
```

- Operate on contents of a file

```
apple
pear
fig
orange
```

```
$ xargs -a fruits mkdir
$ ls -F
apple/  fig/  fruits  orange/  pear/
```

awk and sed

- awk is a scripting language

```
awk '{sum += $2} END {print sum/NR}'  \
  input/file_for_sorting.txt
```

- sed is a file/stream manipulation language

```
sed 's/Feb/February/g' input/file_for_sorting.txt
sed -n '5,8p' input/file_for_sorting.txt
```

Funny Characters

- Windows text editors will add carriage returns

```
file input/notepad.txt
```

```
notepad.txt: ASCII text, with CRLF line terminators
```

- Certain text editors may insert unicode characters

```
file input/unicode.txt
```

```
unicode.txt: UTF-8 Unicode text
```

Funny Characters

- You can find non-printing characters using `sed` and `grep`:

```
sed 's/\r/\xFF/g' input/notepad.txt
```

```
1:This is a test file
2:created using Notepad
3:on a Windows machine.
4:
```

Funny Characters

- Unicode:

```
LANG=C grep -n --color=always \
  '[^[:cntrl:] -~]\+' input/unicode.txt
```

```
1:This is a file that contains unicode characters '소녀시대
' That snuck in somehow
```

- Even tabs:

```
sed 's/\t/\xe2\x9e\xa4/g' input/tab_delimit.txt | \
  grep --color=auto -P -n "[^\x00-\x7F]"
```

```
1:This➤file➤has➤tab delimited➤fields➤for➤searching
```

highlightTabs, highlightUnicode

```
$ highlightTabs.sh
examples/funny_characters/tab_delimit.txt
1:This➤file➤has➤tab delimited➤fields➤for➤searching
```

```
$ highlightUnicode.sh examples/funny_characters/unicode.txt
1:This is a file that contains unicode characters
'소녀시대' That snuck in somehow
2:Strange space character: [  ] em space U+2003
4:Another funny character '▉'
```

screen and tmux

- Allows login sessions to be saved, detached, and reattached
- Cryptic controls with [CTRL-a]

```
$ screen
```

- tmux is a little more sophisticated, and is not immediately available

```
$ ml tmux
$ tmux
```

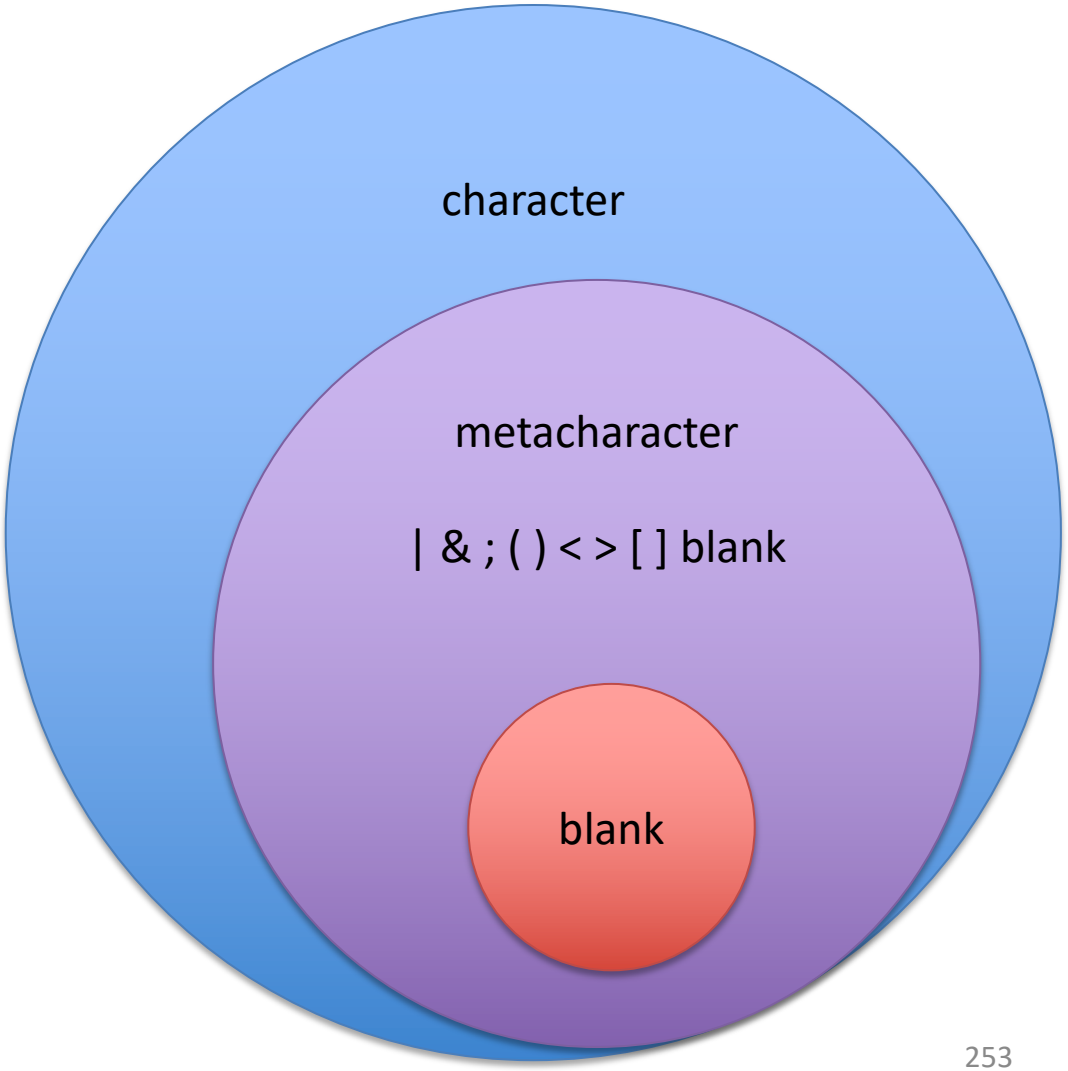
EXTRAS

Expanded List of Linux Commands

a2p
a2ps
acroread
alias
apropos
at
awk
basename
bash
batch
batchlim
bc
bg
bind
break
bunzip2
bzcat
bzcmp
bzdiff
bzgrep
bzip2
bzip2recover
bzless
bzmore
cal
cat
cd
checkquota
chfn
chgrp
chmod
chown
chsh
cksum
clear
cmp
column
combinediff
comm
continue
cp
crontab
csh
csplit
curl
cut
cvs
date
dc
dd
declare
df
diff
diff3
dir
dircolors
dirname
dirs
disown
display
dos2unix
du
echo
egrep
emacs
enable
enscript
env
eval
exec
exit
expand
export
expr
factor
false
fg
fgrep
file
find
finger
flock
fmt
fold
free
freen
ftp
funzip
gawk
gedit
getopts
ghostscript
glxinfo
gpg
grep
groups
gs
gunzip
gvim
gvimdiff
gzip
hash
head
help
history
host
hostname
iconv
id
info
interdiff
iostat
jobcheck
jobload
jobs
join
kill
ksh
less
let
lftp
lftpget
ln
local
locate
lockfile
login
logname
logout
look
ls
lsof
mac2unix
man
man2html
md5sum
merge
mkdir
mkfifo
more
mv
namei
nano
netstat
newer
newgrp
nice
nl
nohup
oclock
openssl
passwd
paste
patch
pathchk
pdf2ps
perl
pgrep
php
pico
pidof
ping
pinky
pkill
popd
pr
printenv
printf
ps
pushd
pwd
python
qdel
qselect
qstat
qsub
quota
rcp
rdist
read
readlink
readonly
rename
renice
reset
resize
return
rev
rexec
rlogin
rm
rmdir
rsh
rsync
scp
screen
script
sdiff
sed
sendmail
seq
set
setsid
sftp
sg
sh
shift
shopt
shred
sleep
sort
source
split
splitdiff
sqlite3
ssh
stat
strace
stream
strings
submitinfo
suspend
svn
swarm
swarmdel
tac
tail
tailf
tar
tcsh
tee
telnet
test
time
timeout3
times
top
touch
tr
trap
tree
true
tset
type
ulimit
umask
unalias
uname
unexpand
uniq
unix2dos
unlink
unset
unzip
updatedb
uptime
urlview
users
usleep
vdir
vi
view
vim
vimdiff
vimtutor
vmstat
zmore
wait
wall
watch
wc
wget
whatis
whereis
which
whiptail
who
whoami
whois
xargs
xclock
xdiff
xeyes
xterm
yes
zcat
zcmp
zdiff
zforce
zgrep
zip
zipgrep
zipinfo
zipnote
zipsplit
zless
zsh

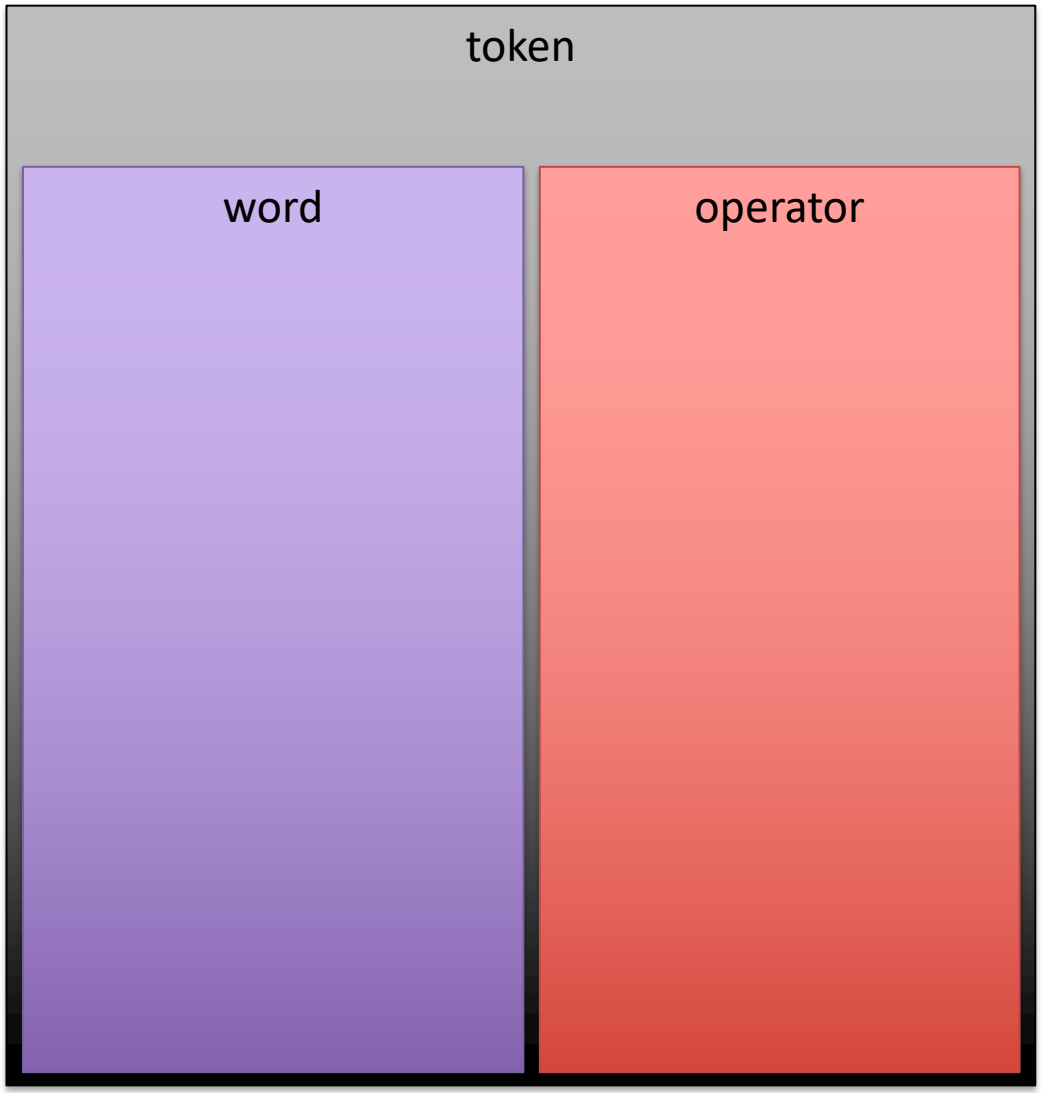
Definitions

- Character: result of a keystroke
- Metacharacter: separates words
- Blank: space, tab, or newline



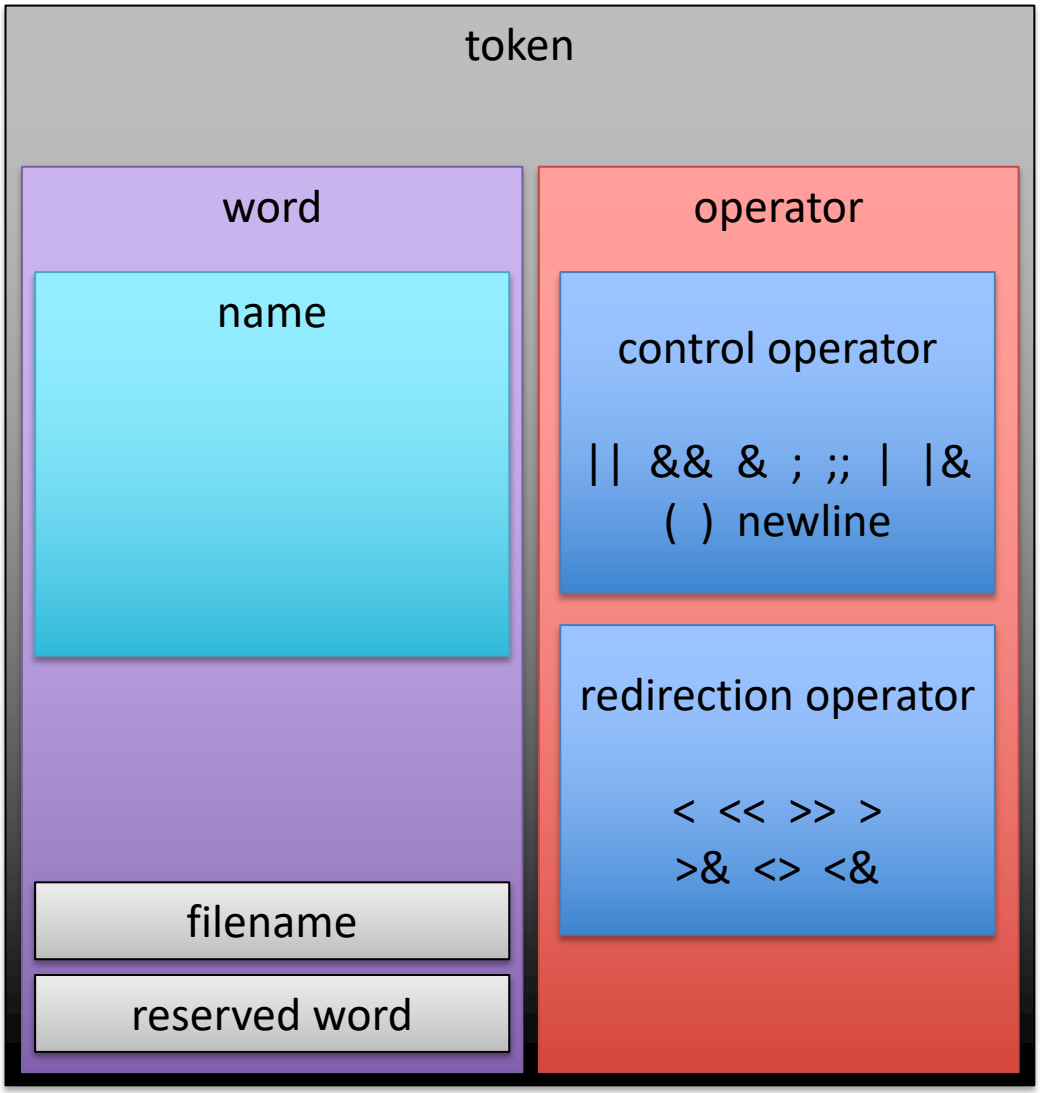
Definitions

- Token: one or more characters separated into fields
- Word: a token with no unquoted metacharacters
- Operator: a token with one or more metacharacters



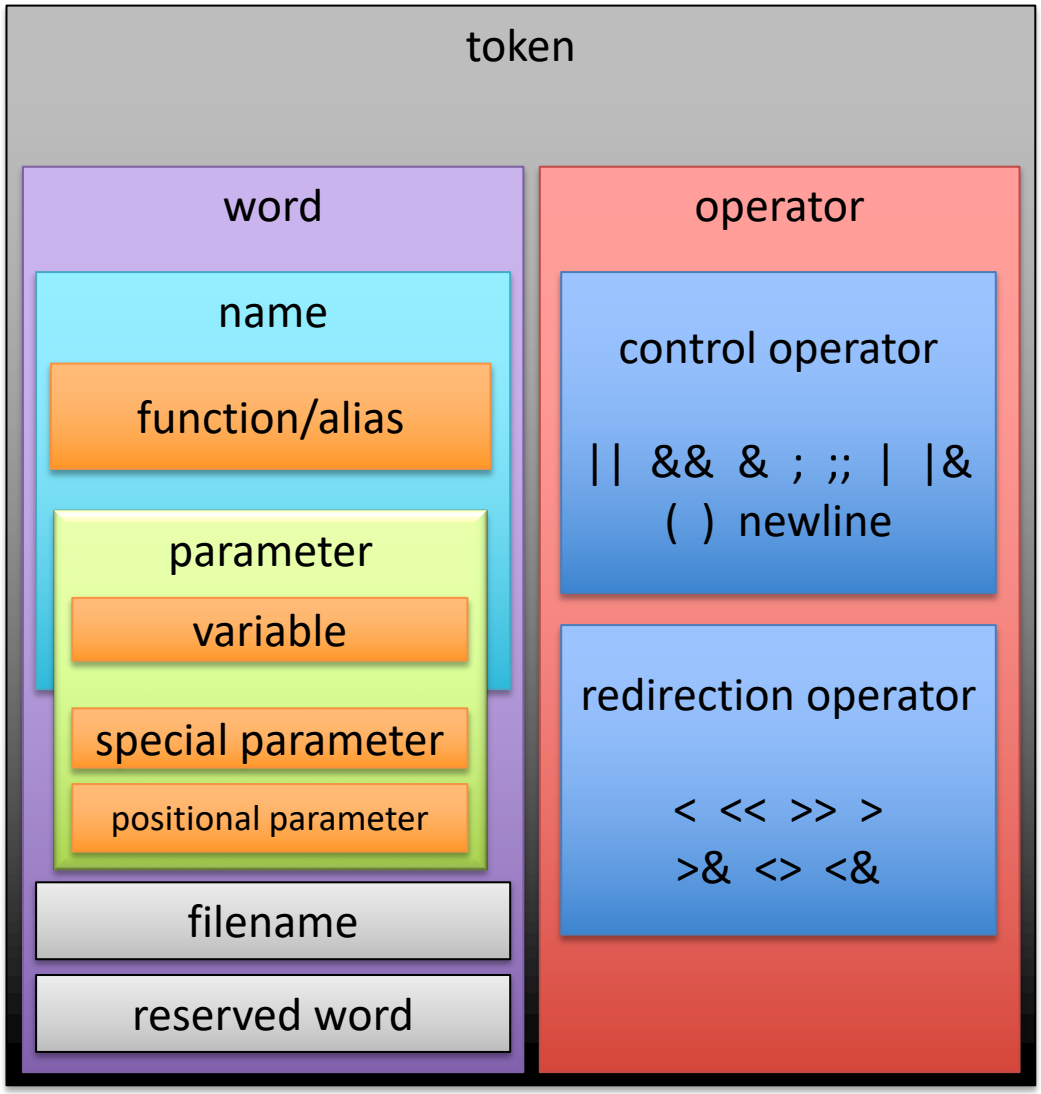
Definitions

- Name: a word with [a..z,A..Z,0..9,_] only
- Control operator: performs control functions
- Redirection operator: redirects file descriptors
- Filename: identifies a file
- Reserved word: special meaning , like `for` or `while`



Definitions

- Parameter: stores values
- Variable: name for storing values, must begin with letter
- Special parameter: can't be modified
- Positional parameters: for retrieving arguments



Shell Parameters - Special

Special parameters have a single character

`$*`	expands to the positional parameters
`$@`	the same as $*, but as array rather than string
`$#`	number of positional parameters
`$-`	current option flags when shell invoked
`$$`	process id of the shell
`$!`	process id of last executed background command
`$0`	name of the shell or shell script
`$_`	final argument of last executed foreground command
`$?`	exit status of last executed foreground command

Bash Variables SET

- $BASH_REMATCH
- $BASH_SUBSHELL
- $PWD
- $REPLY
- $SHLVL
- … type man bash, search for Shell Variables

Bash Variables USED

- $BASH_ENV
- $HISTFILE
- $HISTFILESIZE
- $IFS
- $PATH
- … type man bash, search for Shell Variables

Questions? Comments?

staff@hpc.nih.gov