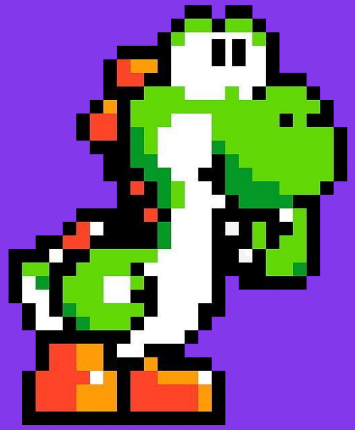
Bash Scripts #3

Input/Output + Oneliners

By: Laura & Yosef

(Pixel) Pet Tax



Yoshi

Psyduck

Flappy Bird

Final Exam Logistics

- Week after Thanksgiving during lecture time (**Dec 2nd**)
- Will be multiple choice and short answer
- Cover all material including this lecture (Bash Pt. 3)
- Paper + pen/pencil
- Materials Allowed:
  - Man pages will be provided
  - 1 pg (8.5 x 11) handwritten cheat sheet (front and back)
  - You can write it out digitally but you must have it printed
  - Name at the top of the sheet
  - We will collect your cheat sheets at the end!
  - It will be heavily curved so don't worry!

Final Grade Logistics

- All labs turned in by Dec 2nd (hard deadline!)
- Double check autolab!
  - Make sure you have done the pull request for collab on Github

How to Become a Bash Expert

Bash as a full-fledged scripting language

STEP 3

STEP 2

Bash as a data processing language

Learn to browse files and run programs

STEP 1

01

Unix Process Communication

Everything is a file!

02

Scripting w. I/O Redirect

How programs talk to each other!

03

More Scripting

How to use pipes and more to stream info!

04

Oneliners

One line programs to make Bash so easy!

01.

Unix Process Comm.

(it's I/Opening)

Unix as a 2-layered API

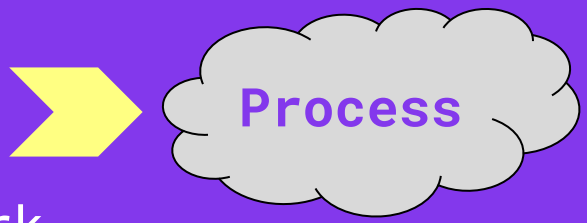
- C functions
  - for “real” system programming
- Shell commands
  - subset of C functionality
  - for scripting and interactive use

Unix as a 2-layered API

- C functions **(15-213)**
  - for “real” system programming
- Shell commands **(GPI)**
  - subset of C functionality
  - for scripting and interactive use

Unix process interacts with the world

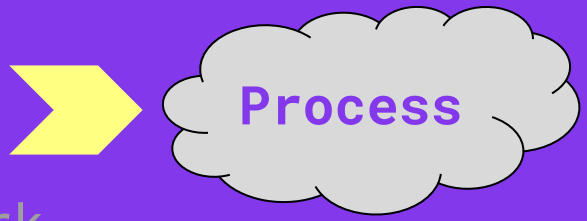
- stdin
- args
- env
- fs
- network



- stdout
- stderr
- exitcode
- fs
- network

We can script some of these

- stdin
- args
- env
- fs
- network



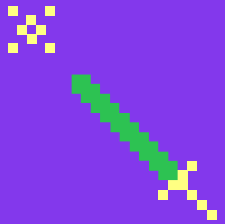
- stdout
- stderr
- exitcode
- fs
- network

Input & Output (streams)

- stdin - standard input (file descriptor = 0)
  - `raw_input, scanf`
- stdout - standard output (file descriptor = 1)
  - `print, printf`
- stderr - standard error (file descriptor = 2)
  - `fprint(stderr)`

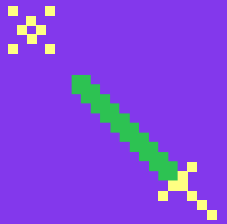
Arguments

- args - command line arguments
- Scripts can access arguments with
  - $# = number of arguments given to the script (different from “argc” in C, which includes program name)
  - $1 = first argument, $2 = second argument, ...
- e.g. $ echo Hello World
  - $# = 2
  - $1 = Hello
  - $2 = World
  - $0 = echo



DEMO

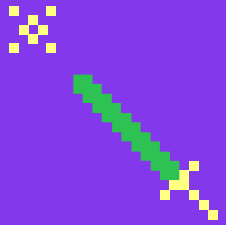
Environment Variables



- A list of key-value pairs
- Essentially the shell's global variables
- Any program can use these variables
  - Access a VAR by $VAR
- `printenv` - prints currently set environment variables

Exit Code

- All programs exit with some code
- This is determined by the programmer
- In general, exit 0 means success
- Anything else indicates some error/failure
- Process can access last executed program’s exit code



DEMO

02.

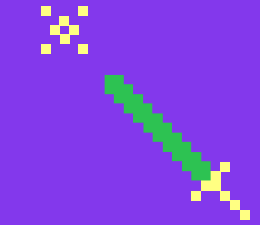
Scripting I/O (Redirect)

(or knowing your computer will always listen to you)

Redirection

	Input	Output	Error
Append	*	`[cmd]>>[file]`	`[cmd]2>>[file]`
Read/ Overwrite	`[cmd]<[file]`	`[cmd]>[file]`	`[cmd]2>[file]`

Redirection Tricks



DEMO

- Redirect one stream to another
  - `[cmd] 2>&1`
- Ignore a stream - redirect stdout to the “null device”
  - `[cmd] > /dev/null`
- Ignore any output from a program (both stdout and stderr)
  - `[cmd] > /dev/null 2>&1`
  - (alternatively) `[cmd] 2> /dev/null 1>&2`
  - (alternatively) `[cmd] > /dev/null 2> /dev/null`

03.

Scripting More

(playing with words is fun!)

Unix Pipes - Intro

- Pipes connect processes by linking `stdout` of first process to `stdin` of second
- Think of it like function composition (if it's not too traumatic):

`g(f(x))` ↔ `x | f | g`

Unix Pipes - Syntax

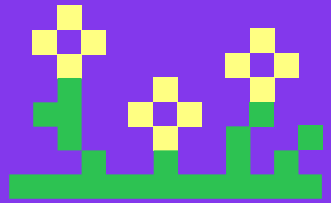
```
<cmd> [ARGS] [REDIRECTS] | <cmd> [ARGS] [REDIRECTS]
```

- The pipe character is ⇧ Shift + \ (i.e. the character above ↵ Return)

Unix Pipes - Warnings

- A few things to keep in mind using pipes:
  - Programs in a series of pipes are run in PARALLEL
    - i.e. if your future programs are dependent on the previous running to completion before starting, don't use pipes
  - At pipe boundaries, results are buffered
    - i.e. if your future programs cannot handle buffered input, don't use pipes

DEMO TIME



(Commands in Demo)

```
fortune | cowsay -n (and optionally -p or -s or -d)
```

```
find . -name "*pdf" | grep -Ev ".*hw[0-9]*.pdf"
```

```
echo “some text” > tmp.txt | cat tmp.txt > tmp2.txt | cat tmp2.txt
```

Scripting More Things

- You can easily change the scripting environment by setting environment variables before a command:

```
VAR1=value1  VAR2=value2 <cmd> [args]
```

- You can get the exit code of a program easily too:

```
$?
```

- Plus, Bash has all the features of a scripting language, including conditionals, functions, loops, and processing tools
  - But, it's hard to write correct code easily (see: pitfalls)
  - Bash is great for automation and it can make your life (a lot!) easier

Command Substitution

- You've already seen how to match file arguments easily with globs
- It's pretty easy to use command substitution to get the output of a command as a single argument: use $(command)

An example:

```
touch myfile-$(date +%s).txt
```

This creates a file with the current timestamp inputted in the name!

An Aside:
The Parable of Knuth and McIlroy

- Jon Bentley, a famous person who improved the speed of quicksort, challenged Donald Knuth to write a program guided by documentation (documentation is 🔑) and asked Doug McIlroy to critique it
- Knuth is famous -- he wrote The Art of Computer Programming, among other things
- McIlroy is also famous -- he literally invented pipes
- Knuth wrote a 10+ page Pascal program -- McIlroy wrote a well-explained 6-line Bash script

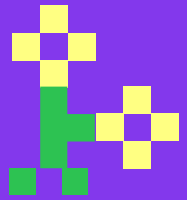
Find the original story here

The Parable of Knuth and McIlory (Continued)

See if you can figure out what the Bash script does!

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Aside 2: More Shells



Fish

The friendly interactive shell

Xonsh

The Python shell

Oil

An upgrade from Bash

ZSH

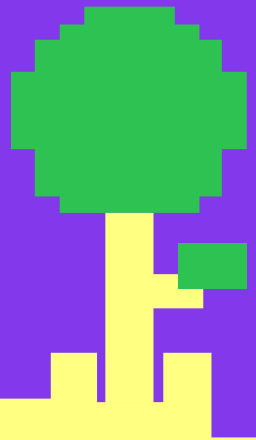
ZSH is super customizable! Check out the Dotfiles extratation slides to get set up!

PowerShell

The object-oriented shell

ZSH

Yea it's on here twice! It's super cool - strongly recommend. (I'm using it rn)



PIPELAB

It's midterm season and with the thought of studying for daunting tests looming over your thoughts, you decide to make a wise(?) decision and play some classic **Super Mario Bros**.

In fact, you've just finished the final boss fight with Bowser and Princess Peach is about to be released when the game seems to glitch... You've managed to unlock the secret hidden level!

"I'm still trapped, Mario, help!" cries Peach from in game. Eager to trounce this level as handily as the others, you notice a green pipe has appeared before you, and you jump right in.

04.

Oneliners

(I got into a fight with the number 1 once.
Its friends 3, 5, 7, and 9 showed up.
The odds were against me)

Examples

- I'm reading a book about anti-gravity. It's impossible to put down.
- Don't spell part backwards. It's a trap.
- The past, the present, and the future walk into a bar. It was tense!
- How did the picture end up in jail? It was framed!

find

- We use **`grep`** to search through file *contents*
- **`find`** does the same thing for file *names* for deep recursive file system searches
  - Walk a file hierarchy and do something for each things that matches

```
find <directory> -regex '<regex>'

find <directory> -name '<glob>'
```

xargs

- Read input from **`stdin`** and execute argument command with arguments constructed from **`stdin`**

```
xargs <command>
```

Examples (For Real)

- Find all my uses of **`find`**

```
history | grep -E "find .*"
```

- Find all my shell scripts, add permissions, and execute them:
  - -t flag is to also print the commands run
  - -n1 flag specifies run command per line of input (one at a time)

```
find . -name '*.sh' | xargs -t chmod +rwx
find . -name '*.sh' | xargs -t -n1 bash
```

Examples (For Real)

- Find all my non-writeup files and reads them:
  - -E treats the expression as extended regex
  - -v all lines not matching the specified pattern

```
find . -name "*txt" | grep -Ev ".*hw[0-9]*.txt" | xargs cat
```

curl

- Make a network request to return a file or webpage located at the argument URL

```
curl <URL>
```

sed

- A Vim-related streaming editor used for scripting that supports many of the same commands

```
sed '<sed_script>' <files>
```

- This is the familiar substitute command which has similar syntax

```
sed 's/<original>/<replacement>/g' <files>
```

Examples (For Real)

- Rename all occurrences of ‘google‘ in output to ‘duckduckgo’
  - -s is silent mode, doesn’t show progress bar or error messages
  - -v is verbose mode, displays some extra info (good for debugging)
  - Look! This also redirects stderr in the first command to stdout!

```
curl -v -s google.com 2>&1 | sed 's/google/duckduckgo/g'
```

Oneliner Tips

- The best way to get a fully working oneliner is to keep building iteratively
  - Try each step one at a time and see what happens when it runs
- Figure out what you think the steps to do what you want should be, and *then* try to write the script
- You stand on the shoulders of all the programmers before you
  - Use Google/StackOverflow as resources to try and figure out if there's an easy way to do what you want
  - Use man pages, they're made to teach people how to use a tool

Useful Resources

Bash One-Liners Explained

A multi-part guide to various bash oneliners explained in detail! (Also has different articles on sed, awk, Perl, among others!)

Bash-Oneliner

A gigantic list of oneliners that will probably have what you want to do with Bash!

Bash Scripting How-To

An introductory article to a wide list of features Bash offers.

Reminders

- **hauntlab** and **zombielab** due 11:59 pm ET tonight
- Extratation this Saturday 1-2 pm at GHC 4211: **Exam Review**
- Course Feedback on tinyurl.com/f21-gpi-feedback

Lab Pro Tips

Helpful commands for pipelab:

- **Curl** - pulls content from an url
- **Sed** - Edits text (stream editing) (input can be supplied through stdin)
- **Xargs** <command> - Transformed newline separated text in stdin to arguments for the given command
- **Test locally first! Construct iteratively!**

Small secret:

- ./driver/driver is a bash script
- Wow! (you can hack it if you want
- But it's probably easier to do the lab...)