1. More on Conditions
2. Bash Arrays
3. Bash functions and local variables
As always: Everybody! ssh to wash.cs.cornell.edu

- Quiz time! Everybody! run **quiz-02-15-19**
- You can just explain a concept from last class, doesn’t have to be a command this time.
- **NOTE:** demos for this lecture:
  /course/cs2043/demos/11-demos
  
  - the leading / is important!
More on Conditions
• Just like a switch statement in other languages, only better.
• Does not carry on to all cases if you forget that `break` keyword.

```bash
case "\$var" in
    "A"
    )
    cmds to execute for case "A"
    ;;
    "B"
    )
    cmds to execute for case "B"
    ;;
    *
    )
    cmds for DEFAULT (not matched) case
    ;;
```

• Sort of like shorthand for `if-elif-else` statements...
• ...only not quite the same!
Simple If and Case Examples

- Make a simple program to print between 0 and 2 `blarghs`
- Input is `$1`, explicit check not necessary (`else` or `*`) case

```bash
#!/usr/bin/env bash

# (empty to fill space in minted)
# (empty to fill space in minted)
# (empty to fill space in minted)

if [[ "$1" == "0" ]]; then
    echo "0 blargh echoes..."
elif [[ "$1" == "1" ]]; then
    echo "1 blargh echoes..."
    echo "[1] blargh"
# number or string
elif [[ "$1" -eq 2 ]]; then
    echo "2 blargh echoes..."
    echo "[1] blargh"
    echo "[2] blargh"
else
    echo "Blarghs come in [0-2]."
    exit 1
fi
```

Demo file `simple/if.sh`.

```bash
#!/usr/bin/env bash
case "$1" in
    "0")
        echo "0 blargh echoes..."
        ;;
    "1")
        echo "1 blargh echoes..."
        echo "[1] blargh"
    ;;
    2)
        echo "2 blargh echoes..."
        echo "[1] blargh"
        echo "[2] blargh"
    ;;
    *)
        echo "Blarghs come in [0-2]."
        exit 1
    ;;
esac
```

Demo file `simple/case.sh`.
Difference Between Case and If Comparisons

• The matching strategy is different for `case` than `if`.
• By default, `case` statements are comparing patterns.
  • Note that a single value e.g., "A" is just an explicit pattern.
  • Patterns are **NOT** regular expressions! Refer to [1].
• By default, `if` statements are comparing values.
  • To use *extended regular expressions* in `if` statements, you need to use the `=~` operator.
  • Use `[[ double bracket expressions ]]` for extended regular expressions in `if`
  • The `=~` operator not available for **all bash < 4.0**. Check `man bash` and search for `=~`.
    • Recall: after `man bash`, type `/expr` and hit `<enter>` to search.
    So type `/=~` and hit `<enter>`.
    • Cycle through results with `n` for next search result.
Using Sets with Case

• See demo file sets/case.sh.

```bash
#!/usr/bin/env bash

case "$1" in
    [[:digit:]]
        echo "$1 blargh echoes..."
        for (( i = 1; i <= $1; i++ ))
            echo " ["$i" blargh"
        done
    ;;
    *
        echo "Blarghs only come in [0-9]."
        exit 1
    ;;

esac
```

• Works on inputs 0-9, as well as exit for everything else.
• Will not match 11 (sets only match one character, see [1]).
• So *) being last is equivalent to default in other languages
  • But only if *) is actually last!
Using Sets with If Part 1

• See demo file `sets/if.sh`.

```bash
#!/usr/bin/env bash
if [[ "$1" =~ [:digit:] ]]; then
  echo "$1 blargh echoes..."
  for (( i = 1; i <= $1; i++ )); do
    echo " [i] blargh"
  done
else
  echo "Blarghs only come in [0-9]."
  exit 1
fi
```

• Works on `[0-9]`.
• Cool! Works on `99`.
• Whoops! Works on `208a` – the `for` loop crashes!
• Option 1: negate a negation (read: *if not “not a number”*):

```bash
# +-----------------+ +-----------------+
# | Negate if match | | Negate (invert) set |
# +-----------------+ +-----------------+
# if [[ ! "$1" =~ [[:digit:]] ]]; then
```

• Option 2: use a complete *extended regular expression pattern*:

```bash
# +---------------------+
# | ^: beginning of line |
# +---------------------+
# if [[ "$1" =~ ^[[:digit:]]+$ ]]; then
# +---------------------+ || +---------------------+
# | += 1 or more digit $ matches end of line |
# +---------------------+ +---------------------+
```
• The last example felt pretty bullet-proof, what can go wrong?

• Using demo file `eregex/if.sh`:

```bash
$ ./if.sh 08
./if.sh: line 4: ((): i <= 08: value too great for base
 (error token is "08")
```

• This is because of the leading 0 — `bash` treats this as octal:

```bash
$ ./if.sh 0111
0111 blargh echos...
  [1] blargh
  [2] blargh
  ...
  [72] blargh
  [73] blargh
```

• For now, we’ll happily ignore this.
Bash Arrays
Arrays in **bash** are extraordinarily flexible in some senses...
...and particularly fickle in other senses.
Short version:
```
arr=( use parentheses and separate items by spaces )
```

Mixed “types”: `my_arr=( "a string" 1 twelve "33" )`

Question: what are the types of `twelve` and `"33"`
  - `twelve` would be interpreted as a `string`.
  - `"33"` can be either a `string` or a number!
  - **bash** doesn’t really have a “type system”.

```
my_arr=( "a string" 1 twelve "33" )
echo "Index '3' with '44' added: $(( ${my_arr[3]} + 44 ))"
# Prints:
# Index '3' with '44' added: 77
```
• The majority of the remaining examples are either copied or modified from [2].
  • A truly excellent resource, worth reading on your own!
  • We do not have time to cover all of the cool and obscure things you can do with arrays.

• We’ll be going through chunks of demo file slide_arrays.sh.
Alternative Initialization

- `arr=( parentheses enumerations )` gives indices in range 0, up to *but not including* length of array.

- Custom indices are allowed!
  
  ```text
  arr[22]=22
  arr[33]=33
  arr[51]="a string value"
  arr[52]="different string value"
  ```

- Indices do **not** need to be integers:
  
  ```text
  some_array=( zero one two )  # Indices: 0, 1, 2
  some_array[11]=11           # Indices: 0, 1, 2, 11
  some_array["hi"]="there"   # Indices: 0, 1, 2, 11, "hi"
  ```

- You **cannot** have an `array` of `arrays`.
Array Functions

• You perform an **array** operation with `{expr}`
  • Works on non-arrays too; mandatory for arrays

• You use the name of the variable followed by the operation:
  
  ```
  echo "Index 11: ${arr[11]}"  # prints: Index 11: 11
  echo "Index 51: ${arr[51]}"  # prints: Index 51: a string value
  echo "Index 0: ${arr[0]}"    # DOES NOT EXIST! (aka nothing)
  ```

• Like loops, `@` and `*` expand differently:
  
  ```
  echo "Individual: ${arr[@]}"
  # Individual: 11 22 33 a string value different string value
  echo "Joined::: : ${arr[*]}"
  # Joined::: : 11 22 33 a string value different string value
  ```

• Differently how?
  
  ```
  echo "Length of Individual: ${#arr[@]}"
  # Length of Individual: 5
  echo "Length of Joined::: : ${#arr[*]}"
  # Length of Joined::: : 5
  ```
Differently HOW?!!!

- Easier to compare with loops
  - Remember that `;` allows you to continue on the same line.

- Individual expansion (@):
  ```bash
  for x in "${arr[@]}"; do echo "$x"; done
  # 11
  # 22
  # 33
  # a string value
  # different string value
  ```

- Joined expansion (*):
  ```bash
  for x in "${arr[*]}"; do echo "$x"; done
  # 11 22 33 a string value different string value
  ```

- The * loop only executes once (everything is *globbed* together).
- The @ loop iterates over each element in the array.
Even More Initialization Options

- Evaluate expressions and initialize at once:

```bash
echo "Index 44: ${arr[44]}"  # Index 44: 44
echo "Index 55: ${arr[55]}"  # Index 55: 55
```

- Alternative index specifications:

```bash
new_arr=([17]="seventeen" [24]="twenty-four")
new_arr[99]="ninety nine"  # may as well, not new
for x in "${new_arr[@]}"; do echo "$x"; done
# seventeen
# twenty-four
# ninety nine
```

- Get the list of indices:

```bash
for idx in "${!new_arr[@]}"; do echo "$idx"; done
# 17
# 24
# 99
```
• You can just as easily *slice* your arrays.

• Use `@` to get whole array, then specify indices to *slice*
  
  • Syntax: `{{array_var[@]:start_index:slice_size}}`
  
  • If `end_index` is not specified, takes until last index

```bash
zed=( zero one two three four )
echo "From start: ${zed[@]:0}"  
# From start: zero one two three four
echo "From 2: ${zed[@]:2}"  
# From 2: two three four
echo "Indices [2-4]: ${zed[@]:2:3}"  
# Indices [2-4]: two three four
for x in "${zed[@]:2:3}"; do echo "$x"; done  
# two
# three
# four
for x in "${zed[*]:2:3}"; do echo "$x"; done  
# two three four
```
• This was a *small subset* of what can be done with **bash** arrays.
• I highly suggest you go through the examples listed in [2] in.
  • Search for **Substring Removal** for some insanely cool tricks!
Bash functions and local variables
can define **functions** in bash

---

declare a new **function** (bash builtin)

```
function <name> {
body...
}
```

line breaks are essential!

```
function hello {
    echo "hello world!"
}
```

- functions take arguments, just like scripts!
  - arguments to script are hidden within the function

```
function print_an_arg {
    echo "$*"
}
```
Variables defined in functions

- Reminder: *environment* variables inherited by sub-scripts
- Reminder: *local* variables only in current script
- Variables defined in functions are visible outside!

```bash
function define_a_variable {
    x='words!'
}
define_a_variable
echo $x
#prints words!
```

- invoke a function just like a command
Very-local variables

- the `local` keyword keeps variables within the function only
- a terrible name; `local` variables and “local” (as in not environment) variables are different.
- *opposite* of `global` keyword in python

```python
function define_a_variable {
    local x='words!

}
define_a_variable
echo $x
#prints nothing
```
