10 - Bash Scripting III, Git Merging and Diffs

CS 2043: Unix Tools and Scripting, Spring 2016 [1]

Stephen McDowell

February 19th, 2016

Cornell University

Table of contents

Some Logistics

- (poll) are you confused about how to access the various resources in the class?

Some Logistics

- (poll) are you confused about how to access the various resources in the class?
- Review of variables.

Some Logistics

- (poll) are you confused about how to access the various resources in the class?
- Review of variables.
- Sorry about today...

Some Logistics

- (poll) are you confused about how to access the various resources in the class?
- Review of variables.
- Sorry about today...
- ...I wanted to get your HW to you. That will happen tonight.

More on Conditions

Case

- Just like a switch statement in other languages, only better.

Case

- Just like a switch statement in other languages, only better.
- Does not carry on to all cases if you forget that `break` keyword.

Case

- Just like a switch statement in other languages, only better.
- Does not carry on to all cases if you forget that `break` keyword.

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

Case

- Just like a switch statement in other languages, only better.
- Does not carry on to all cases if you forget that `break` keyword.

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

- Basically just shorthand for `if-elif-else`...

Case

- Just like a switch statement in other languages, only better.
- Does not carry on to all cases if you forget that `break` keyword.

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

- Basically just shorthand for `if-elif-else`...
- ...only not!

Simple If and Case Examples

- Suppose we wanted to make a simple program to print between 0 and 2 `blargh`s.

Simple If and Case Examples

- Suppose we wanted to make a simple program to print between 0 and 2 `blargh`s.
- Assume that the input to the script is `$1`.

Simple If and Case Examples

- Suppose we wanted to make a simple program to print between 0 and 2 `blargh`s.
- Assume that the input to the script is `$1`.
- We don't need to check because it will just not match.

Simple If and Case Examples

- Suppose we wanted to make a simple program to print between 0 and 2 `blargh`s.
- Assume that the input to the script is `$1`.
- We don't need to check because it will just not match.

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

Simple If and Case Examples

- Suppose we wanted to make a simple program to print between 0 and 2 `blargh`s.
- Assume that the input to the script is `$1`.
- We don't need to check because it will just not match.

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

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

Case and If Comparisons

- The matching strategy is different for `case` than `if`.

Case and If Comparisons

- The matching strategy is different for `case` than `if`.
    - By default, `case` statements are comparing *patterns*.

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.

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 [2].

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 [2].
  - By default, `if` statements are comparing values.

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 [2].
  - By default, `if` statements are comparing values.
    - To use *extended regular expressions* in `if` statements, you need to use the `=~` operator.

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 [2].
  - By default, `if` statements are comparing values.
    - To use *extended regular expressions* in `if` statements, you need to use the `=~` operator.
    - In most `bash`, the expression on the *right* is treated as an *extended* regular expression.

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 [2].
  - By default, `if` statements are comparing values.
    - To use *extended regular expressions* in `if` statements, you need to use the `=~` operator.
    - In most `bash`, the expression on the *right* is treated as an *extended* regular expression.
    - Not for all pre-`4.0`, pull up `man bash` and search for `=~`.

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 [2].
  - By default, `if` statements are comparing values.
    - To use *extended regular expressions* in `if` statements, you need to use the `=~` operator.
    - In most `bash`, the expression on the *right* is treated as an *extended* regular expression.
    - Not for all pre-`4.0`, pull up `man bash` and search for `=~`.
    - Remember to search in the man page type `/expr to search` and hit `enter`.

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 [2].
  - By default, `if` statements are comparing values.
    - To use *extended regular expressions* in `if` statements, you need to use the `=~` operator.
    - In most `bash`, the expression on the *right* is treated as an *extended* regular expression.
    - Not for all pre-`4.0`, pull up `man bash` and search for `=~`.
    - Remember to search in the man page type `/expr to search` and hit `enter`.
    - Cycle through the results with `n` for next search result.

Using Sets with **case**

- `case` with the set `[0-9]`:

Using Sets with **case**

- `case` with the set `[0-9]`:

```
#! /bin/bash
case "$1" in
  [[:digit:]] )
    echo "$1 blargh echoes..."
    for (( i = 1; i <= $1; i++ )); do
        echo "  [$i] blargh"
    done
    ;;
  * )
    echo "Blarghs only come in [0-9]."
    exit 1
    ;;
esac
```

Using Sets with **case**

- `case` with the set `[0-9]`:

```
#! /bin/bash
case "$1" in
  [[:digit:]] )
    echo "$1 blargh echoes..."
    for (( i = 1; i <= $1; i++ )); do
        echo "  [$i] blargh"
    done
    ;;
  * )
    echo "Blarghs only come in [0-9]."
    exit 1
    ;;
esac
```

- This will work on inputs `0-9`, as well as exit for everything else.

Using Sets with **case**

- `case` with the set `[0-9]`:

```
#! /bin/bash
case "$1" in
  [[:digit:]] )
    echo "$1 blargh echoes..."
    for (( i = 1; i <= $1; i++ )); do
        echo "  [$i] blargh"
    done
    ;;
  * )
    echo "Blarghs only come in [0-9]."
    exit 1
    ;;
esac
```

- This will work on inputs `0-9`, as well as exit for everything else.
- It will not match `11`, because that is not in the `set`.

Using Sets with **case**

- `case` with the set `[0-9]`:

```
#! /bin/bash
case "$1" in
  [[:digit:]] )
    echo "$1 blargh echoes..."
    for (( i = 1; i <= $1; i++ )); do
        echo "  [$i] blargh"
    done
    ;;
  * )
    echo "Blarghs only come in [0-9]."
    exit 1
    ;;
esac
```

- This will work on inputs `0-9`, as well as exit for everything else.
- It will not match `11`, because that is not in the `set`.
- Refer to [2] for the extent of what you can do with `case`.

Using Sets with **case**

- `case` with the set `[0-9]`:

```
#! /bin/bash
case "$1" in
  [[:digit:]] )
    echo "$1 blargh echoes..."
    for (( i = 1; i <= $1; i++ )); do
        echo "  [$i] blargh"
    done
    ;;
  * )
    echo "Blarghs only come in [0-9]."
    exit 1
    ;;
esac
```

- This will work on inputs `0-9`, as well as exit for everything else.
- It will not match `11`, because that is not in the `set`.
- Refer to [2] for the extent of what you can do with `case`.
- It should now make more sense why * being last is equivalent to `default`.

Using Sets with **case**

- `case` with the set `[0-9]`:

```
#! /bin/bash
case "$1" in
  [[:digit:]] )
    echo "$1 blargh echoes..."
    for (( i = 1; i <= $1; i++ )); do
        echo "  [$i] blargh"
    done
    ;;
  * )
    echo "Blarghs only come in [0-9]."
    exit 1
    ;;
esac
```

- This will work on inputs `0-9`, as well as exit for everything else.
- It will not match `11`, because that is not in the `set`.
- Refer to [2] for the extent of what you can do with `case`.
- It should now make more sense why * being last is equivalent to `default`.
  - Careful it actually is last!

Using Sets with **`if`**

- Lets use the same example:

Using Sets with `if`

- Lets use the same example:

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

Using Sets with `if`

- Lets use the same example:

```
#! /bin/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]`.

Using Sets with `if`

- Lets use the same example:

```
#! /bin/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.

Using Sets with `if`

- Lets use the same example:

```
#! /bin/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.

The =~ Operator

- Option 1 - negate a negation:

The =~ Operator

- Option 1 - negate a negation:

```
#           +------------+
#           | invert set |
#           +------------+
#                 |
if [[ ! "$1" =~ [^[:digit:]] ]]; then
```

The =~ Operator

- Option 1 - negate a negation:

```
#           +------------+
#           | invert set |
#           +------------+
#                 |
if [[ ! "$1" =~ [^[:digit:]] ]]; then
```

- Option 2 - use extended regular expressions:

The =~ Operator

- Option 1 - negate a negation:

```
#          +------------+
#          | invert set |
#          +------------+
#                |
if [[ ! "$1" =~ [^[:digit:]] ]]; then
```

- Option 2 - use extended regular expressions:

```
#   +-------------------+
#   | beginning of line |
#   +-------------------+
#             |
if [[ "$1" =~ ^[[:digit:]]+$ ]]; then
#                         ||
# +-----------------+     ||
# | 1 or more digit |-----+|
# +-----------------+      |
#  +-----------------------+
#  | $ matches end of line |
#  +-----------------------+
```

Git Tools

What is a Merge?

- What is a merge?

What is a Merge?

- What is a merge?
    - When `git` combines code bases that are divergent.

What is a Merge?

- What is a merge?
    - When `git` combines code bases that are divergent.
- When does it happen?

What is a Merge?

- What is a merge?
    - When `git` combines code bases that are divergent.
- When does it happen?
    - When `git` is merging two separate `commit`s, either across branches *or* across forks.

What is a Merge?

- What is a merge?
    - When `git` combines code bases that are divergent.
- When does it happen?
    - When `git` is merging two separate `commit`s, either across branches *or* across forks.
- Why does this matter?

What is a Merge?

- What is a merge?
  - When `git` combines code bases that are divergent.
- When does it happen?
  - When `git` is merging two separate `commit`s, either across branches *or* across forks.
- Why does this matter?
  - `git` may know how to automatically merge (fast-forward)...

What is a Merge?

- What is a merge?
    - When `git` combines code bases that are divergent.
- When does it happen?
    - When `git` is merging two separate `commit`s, either across branches *or* across forks.
- Why does this matter?
    - `git` may know how to automatically merge (fast-forward)...
    - ...or it won't (merge conflict).

What is a Merge?

- What is a merge?
  - When `git` combines code bases that are divergent.
- When does it happen?
  - When `git` is merging two separate `commit`s, either across branches *or* across forks.
- Why does this matter?
  - `git` may know how to automatically merge (fast-forward)...
  - ...or it won't (merge conflict).
- Lets go ahead and do one.

Status and Differences

- What does `git status` do?

Status and Differences

- What does `git status` do?
    - Informs us of changes in code, untracked files, etc.

Status and Differences

- What does `git status` do?
  - Informs us of changes in code, untracked files, etc.
- Can we get more information when there are differences?

Status and Differences

- What does `git status` do?
    - Informs us of changes in code, untracked files, etc.
- Can we get more information when there are differences?

```
>>> git diff
```

Status and Differences

- What does `git status` do?
    - Informs us of changes in code, untracked files, etc.
- Can we get more information when there are differences?

```
>>> git diff
```

- Can we get some useful / readable information?

Status and Differences

- What does `git status` do?
    - Informs us of changes in code, untracked files, etc.
- Can we get more information when there are differences?

```
>>> git diff
```

- Can we get some useful / readable information?

```
>>> git config --global diff.tool vimdiff
>>> git config --global alias.d difftool
# now 'git d' aliases to 'git difftool'
```

Status and Differences

- What does `git status` do?
    - Informs us of changes in code, untracked files, etc.
- Can we get more information when there are differences?

```
>>> git diff
```

- Can we get some useful / readable information?

```
>>> git config --global diff.tool vimdiff
>>> git config --global alias.d difftool
# now 'git d' aliases to 'git difftool'
```

- Time for a forced merge conflict!

References I

[1] B. Abrahao, H. Abu-Libdeh, N. Savva, D. Slater, and others over the years.
Previous cornell cs 2043 course slides.

[2] gnu.org.
Bash reference manual: Pattern matching.
`http://www.gnu.org/software/bash/manual/bashref.html#Pattern-Matching`.