CSE 391, Winter 2020
Extra Credit Assignment: More Advanced Shell Scripting
Due Friday, March 20, 2020, 5:00 PM

This extra credit assignment is a culmination of most of what you have learned this quarter. You will use your knowledge of Bash shell scripting at a more advanced level, as well as a few regular expressions. With that being said, we expect the assignment to be doable for everyone in the class (as long as you look at the hints we provide you at the end of this specification). Electronically turn in file `canvassing_mapper.sh` as described in this document. You will also want the support files from the Homework section of the course web site.

***

IMPORTANT: Since this assignment is extra credit and is due very close to when grades must be submitted, we will be grading the assignment coarsely and purely based on external correctness. If your script follows the specification below exactly, you will get two points of extra credit, which will count towards the 14 points you need to pass the class. In order to receive credit, your script must work for any directory that meets the specification; you should NOT hardcode any specific student or file names.

If it does not work for any reason (e.g. has a syntax error and fails to run, produces incorrect output, does not modify the files as described, etc.), we will not spend any time trying to debug it for you and you will not get any credit. Please use `diff` to compare your output against the expected output and make sure that the file structure after running your script is correct.

Debugging help via email will be limited since this is an extra credit assignment, but please ask questions on the discussion board for clarification or if you think there are mistakes in the assignment (it is a new assignment) – we will be happy to look into these.

***

Background:

You have decided that you want to TA for the hypothetical CSE 392 (Unix Tools) course at the hypothetical University of Dryington. As part of your TA duties, you are expected to grade some student assignment submissions, and luckily have some grading scripts to help you with your grading. Unfortunately, the grading scripts were set up to expect student submissions in a different format than the format in which you download the student submissions.

For example, if students Joe Smith, Ann Allison, Henry Harrison, and Victoria Velociraptor are each submitting the files `part1.sh` and `part2.sh`, your grading scripts expect that you have a directory called `users`, with four subdirectories: `smithjoe`, `allisonann`, `harrisonhenry`, and `velociraptorvictoria`, each with up to two files (see below for more on that), called `part1.sh` and `part2.sh` (*exactly* matching that directory structure and file/directory naming and casing) in them. We want something like this:

```
users/
      allisonann/
            part1.sh
            part2.sh
      harrisonhenry/
            part1.sh
            part2.sh
      smithjoe/
            part1.sh
      velociraptorvictoria/
            part1.sh
            part2.sh
```

Meanwhile, your hypothetical Canvassing LMS (the LMS is short for “Learning Mismanagement System”) only allows you to download student submissions as a single zip archive, `submissions.zip`, with all submissions from all students mixed together at the top level of the zip. The structure of the unzipped submissions directory looks something like this:

```
submissions/
      allisonann_2929292_92812432_part1
      allisonann_2929292_92812433_part2
      harrisonhenry_2989128_10233005_part1-3
      harrisonhenry_2989128_10233006_mypart2
      smithjoe_late_3458993_29204892_part1-1
      velociraptorvictoria_0289482_12340840_part1
      velociraptorvictoria_0289482_12340841_my part2
      velociraptorvictoria_0289482_12340842_part3-2
```

Pay attention to a few things about this naming format:

- Student submissions are renamed to be the full student name, then an underscore, then a sequence of numbers, then an underscore, then another sequence of numbers, then an underscore, then finally a (possibly shortened version of) the filename.
- Students may not name the files exactly the way they were asked to (including by adding spaces into the filename!) – see the hints section below for tips on dealing with that.
- Resubmissions (when a student submits multiple times) cause a number (e.g. a "-3" for the third resubmission) to be appended to the end of the filename, and only the latest submission is included in the zip.
- If a student submits a file late, the string "_late" will be inserted into the filename after the student name but before the first underscore.
- Students may only submit *some* of the required files for the assignment (if they submit none, that is okay to ignore them for this assignment).
- Students may submit extra files that weren't expected (or the filename may not be close enough to any expected file), in which case you should print an error message and then ignore the file (leave it where it is).

Implementation Details:

For this assignment, you will write a *robust* script `canvassing_mapper.sh` that converts the zip file downloaded from Canvassing into the directory format expected by the grading script. Note that we are not running student submissions as we did in Homework 7. We are just renaming and reorganizing them.

Your script should be able to handle all the quirks mentioned above. Your script should not be particular to this set of students or this particular assignment (with the exact names `part1.sh` and `part2.sh`). You should also print a brief message about each student that submits an assignment late, a message to indicate improperly named/unexpected files, and a message to indicate that the script has finished executing.

To get started, download the following support files from the course web site:

- **`submissions.zip`** : This file contains a set of fake student solutions that we want to rename and reorganize. Extract this file with the command `unzip submissions.zip` . It will output something like this:

```
Archive:  submissions.zip
   creating: submissions/
  inflating: submissions/smithjoe_late_3458993_29204892_part1-1
 extracting: submissions/velociraptorvictoria_0289482_12340840_part1
  inflating: submissions/velociraptorvictoria_0289482_12340842_part3-2
  inflating: submissions/velociraptorvictoria_0289482_12340841_my part2
  inflating: submissions/harrisonhenry_2989128_10233006_mypart2
  inflating: submissions/allisonann_2929292_92812433_part2
 extracting: submissions/allisonann_2929292_92812432_part1
 extracting: submissions/harrisonhenry_2989128_10233005_part1-3
```

  This should create a submissions directory, with these eight student submission files in it. Note that you are not targeting the specific students shown above; your code should not specifically mention names like `smithjoe` or `allisonann`. Your code should process *all* submissions in the `submissions` directory.

  - We have also provided the same files in tar format as `submissions.tar.gz` if you prefer.

- **`expected_output.txt`** : This file contains the expected homework output. You should diff the output of running your script against this file.

You must write a script `canvassing_mapper.sh` that runs from a directory containing a folder called `submissions` and **changes** that submissions directory into a directory `users` that meets the format the grading script expects (specified above). We recommend you keep the submissions zip archive handy so you can unzip again and replace the submissions directory if you accidentally mess it up.

Your script should accept pairs of command line arguments, with one pair for each file we expect the students to submit. The pair consists of a short string that we should use to recognize the file, followed by the full, "target name" of the file. This system is to allow students (or Canvassing) to misname the file but have our script still accept the file.

For example, if we want to collect the file `part2.sh` but could plausibly expect it to be misnamed, we might tell the canvassing_mapper.sh script that any file a student submits that contains "part2" should be put in their directory and renamed to `part2.sh`. So in this case, the appropriate pairs list is: `part1 part1.sh part2 part2.sh`. You should not assume that there will be exactly two pairs, or that the naming of a short or long name follows any particular convention (e.g. "long names end in .sh")

The information contained in the pairs list is the *only* file misnaming we expect you to handle. The pairs arguments are saying that your script should only recognize a file and rename it to `part1.sh` if it contains the exact string "part1", and similar for part2. If someone were to submit a file called `Part1.sh`, your script should print an error message for the bad file, but not recognize it (because the 'p' is capitalized improperly). This restriction is here for simplicity.

You do not need to check the pairs arguments for validity. If your script is given more than one command-line argument, you can assume it is a valid pairs list. If your script gets less than two command-line arguments, you should print out a usage message (up to you what that looks like) and exit with the appropriate exit status.

For each file in the submissions directory, you should loop through all pairs given in the pairs list to try to match it to one of the short names we were expecting students to submit for this assignment. If you get a match, you rename the file to the second element of the pair (the long name) and put it in a directory named after the student. If you cannot match the filename to any pair in the pairs lists, you should leave that file where it is (at the top level of what will become your users directory) and print out an error message: `Bad file <full filename>, ignoring...`

You should also print out the message `Late file <filename>` for any late file in the directory (although make sure to still move and rename the file as appropriate!), and you should print out `Done!` when the script is done executing.

Here is sample output from a run of `canvassing_mapper.sh` :

```
$ ./canvassing_mapper.sh part1 part1.sh part2 part2.sh
Bad file ./velociraptorvictoria_0289482_12340842_part3-2, ignoring...
Late file ./smithjoe_late_3458993_29204892_part1-1
Done!
```

It is okay for your `canvassing_mapper.sh` script to create temporary files while doing its work, as long as you eventually remove them. It is okay if the order of lines output is switched (e.g. find and ls would traverse the files in a different order).

Development Strategy (suggested):

- Make your script able to simply output the names of all of the files to be processed.
- Test your more complicated commands one at a time on the command-line and make sure they work before putting them in your script! If you put a bunch of commands into a script and then run it, it most likely will not work, and the error messages will likely not be helpful. But if you can narrow down what could be the problem by increasing confidence in the correctness of individual commands, it might save you time. The same thing goes for any commands that you put inside of `$( )`.
- On a similar note, remember to use `echo` statements to output partial results, computations, commands, etc. to verify them.

Your script should work if placed in any directory containing a **`submissions/`** directory with submitted files named as specified above. **Do not hard code in a specific directory on your machine or your home directory on attu. This is important for grading!!**

Each Linux/Unix box can be slightly different; for full credit, your commands must be able to work properly either on the CSE virtual machine image, `attu`, or on the CSE basement lab computers.

Hints:

Certain part of this assignment can be tricky. Since our goal is not to have you stuck on details of things not covered in this course, we are going to give you some specific tips here (if you like a challenge, feel free to figure these out yourself before looking):

- There are a few different ways you can loop through values as pairs. Here are a few hints that you may find helpful:
    - Read the man pages for the `seq` command, in particular to generate relevant sequence of numbers you want.
    - You can convert command line arguments to an array with `ARGS=( "$@" )`
    - You can refer to dynamically created variable names, as in:

      ```
      eval variable2=\$$variable1
      echo "$variable2"
      ```

      where `variable1` is a variable and its value is used to refer to another variable of that name. For example, if the value of `variable1` was "hello", `variable2` would store the same contents as another variable called `hello`. You may find this useful for dynamically referring to command-line arguments.
- If you get to a point where you would like to do an increment operation (such as `i++`) or use the value of something like `i+1`, you use the command `((i+=1))`. The double parentheses are saying that bash should do an arithmetic evaluation on the expression inside the parens. You can also use variables that are declared with "let" which force arithmetic evaluation.
- You may assume that no file name has a space in it.
- To get the name of a student from a filename, you can assume that the student name is anything before the first underscore in the filename.
- To tell if a file is submitted late, see if it has the substring "_late_" in it. It's best to include the underscores so you don't accidentally assume somebody like Ted Tesla (tes**late**d) submitted an assignment late just because of their name.
- In this class, we have used grep to search a file for a regex but have not really searched a variable for a regex before. This is possible through a clever use of the pipe operator.
- Accessing `${ARGS[$i]}` can be tricky if you try to use a while loop for the inner loop. Better to use a for loop.