CS 1410 Assignment 1: Bulk renaming tool

In this assignment, you will write a utility to rename all of the images in a single directory. You will sort them according to when each was last modified, then number them in order with a common prefix. For example, if you had a directory with these files:

```
a.jpg               (modified January 3)
b.jpg               (modified January 4)
c.jpg               (modified January 2)
```

After running the bulk renamer with prefix "picture", the directory would contain the same three files with new names:

```
picture1.jpg        (modified January 2)
picture2.jpg        (modified January 3)
picture3.jpg        (modified January 4)
```

This assignment will introduce the use of Python as a language for writing *scripts*. A script is just a program, but the label is typically applied to programs that are written for specialized tasks that might otherwise be done by hand. Many scripts are used only a few times and are less general-purpose than other programs. System administrators often write scripts to automate tasks.

In this project you will learn some basic file and directory manipulation techniques, practice string manipulation, use python dictionaries and lists, and solve a tricky algorithmic problem.

Before you begin

Before you begin writing code, take some time to study a few of the python standard library modules that you will use:

- http://docs.python.org/library/os.html
- http://docs.python.org/library/os.path.html
- http://docs.python.org/library/sys.html
- http://docs.python.org/library/stdtypes.html#string-methods

You will need to use several functions in each of these libraries, and only brief introductions to them will be given in this text. You are expected to research each one to learn how to use it properly. In addition to reading the documentation, open a python shell and try each function out.

Details

Start by creating the (very simple) user interface. This is a tool that will be used from the command line. When a program is launched from the command line, the operating system passes the name of the program (as it was typed on the command line) and all of the arguments to the program as a list of strings. For example, if you typed:

```
python bulkrename.py mypics vacation
```

the program would launch and have access to this list:

```
['bulkrename.py', 'mypics', 'vacation']
```

The list is available as `sys.argv`, which means you will need to import the "sys" module to use it. Your renaming tool will take one required argument and one optional argument. Note that the name of the script is always included as the first element of the list (element number zero). Some programs change their behavior based on the name they were launched with.

Define a function called "main" that starts by checking the command-line arguments that were passed to the script (in `sys.argv`). If the program was called with no arguments or with more than two arguments, print a *usage message* and quit by calling `sys.exit(1)`. This indicates that the program quit due to an error. An example usage message would be:

```
Usage: python bulkrename.py <directory> [<prefix>]
```

This reminds the user that a directory name is required, and a prefix is optional. The directory is where the files that

should be renamed are located. The final program will rename all of the files in that directory according to the rules described below. If the optional prefix is supplied, the new file names will all start with that prefix, e.g., if you called:

```
python bulkrename.py mypics vacation
```

All of the pictures in the "mypics" directory would be renamed as follows:

```
vacation1.jpg
vacation2.jpg
...
```

If the optional argument is omitted, the name of the directory is used as the prefix:

```
python bulkrename.py mypics
```

results in:

```
mypics1.jpg
mypics2.jpg
...
```

Start with a program that just tests for the right number of arguments. If the number is correct, print then out to the console for testing. For example:

```
python bulkrename.py mypics vacation
```

should result in:

```
Directory: mypics
Prefix: vacation
```

and:

```
python bulkrename.py mypics
```

should result in:

```
Directory: mypics
(no prefix)
```

If the user did not supply the optional prefix, take the directory name as the prefix. First, convert the directory the user supplied into an *absoulute path* using `os.path.abspath`. Read the documentation and try this function out a few times in the interactive shell to see how it works. When printing your test message to the console, start using the absolute path and try it out.

Given the absolute path of the directory, you can get the directory name using `os.path.basename`. Set this as your prefix if the user did not supply one, and print it to the console.

It is important that you use the absolute path, otherwise you may get bad results in some cases. For example, create a directory called "sub" in your project directory and change into it. Then run your script like this:

```
python ../bulkrename.py ..
```

(note: in Windows, this would be `python ..\bulkrename.py ..`)

Assuming your project directory is called "bulkrename", you should see the following:

```
Directory: /path/to/your/project/bulkrename
Prefix: bulkrename
```

If you do not use the absolute path, you will see:

```
Directory: ..
Prefix: ..
```

You probably do not want files with names like “..1.jpg”.

Getting the list of all files

The next step is to gather the list of all files in the directory that the user requested. Use `os.listdir` to do this, and store the result in a list called `allfiles`. For testing, print this list to the console. Try it on a few different directories.

Getting the list of image files

The purpose of this program is to rename image files from a camera, so it should ignore files in the directory that are not images. Create a list of extensions that image files have:

```
extensions = ['jpeg', 'jpg', 'png', 'gif']
```

You can add additional extensions if you like. Now write a function called `filterByExtension` that you will call like this:

```
matching = filterByExtension(root, allfiles, extensions)
```

Here `root` is the absolute path where the images are stored, `allfiles` is the list of files you gathered using `os.listdir`, and `extensions` is the list of acceptable extensions. When this is completed, `matching` should be a list of files that have one of the image extensions.

`filterByExtension` should start with an empty list of files. It should loop through `allfiles` and examine each candidate file. If you find a reason to disqualify it, use `continue` to skip to the next iteration of the loop. If it passes all of the tests, `append` it to your list of results as the final step in the loop.

First, check to see if the file has an extension. Use the `rfind` method of strings (the documentation for this is one of the links given at the beginning) to find where the last dot character is in the file name. If it does not have an extension, there will be no dot, so `rfind` will return -1. Skip the file (using `continue`) in this case.

Extract the extension using a string slice, and convert it to lower case (using the `lower` method of strings). Check if that extension appears in the list of valid extensions (use the `in` keyword to test this). If not, continue to the next file.

If the file has the correct extension, test to make sure it is a regular file (not a directory or something else). Use `os.path.join` to combine the directory name with the file name. Note that you should never join file names by simply putting a `/` character between them, as this only works on some operating systems. Windows requires a backslash (`\`), but this will not work on Linux. `os.path.join` will do the right thing on every platform. Now, use `os.path.isfile` to test if it is a regular file, and skip it if it is not. It would be strange to find a directory named “foo.jpg”, so print a warning message to the console if you find an entry with a matching file name that is not a regular file.

Note: if you are unsure how any of these library functions work, open a Python shell and try it out a few times. Try running through all the steps that `filterByExtension` is supposed to follow by hand a few times. This will help you be a lot more confident and will reduce the bugs that you introduce.

Finally, return the list of matching files that you found. You now know that each file in the list exists, is a regular file, and has an extensions suggesting it is an image file.

Sorting the list by modified time

Next, write a function called `sortByMTime` that will be called like this from `main`:

```
inorder = sortByMTime(root, matching)
```

It will sort the list according to the time the file was last modified. Note: after the assignment is complete, you may wish to modify this function to sort accorting to whatever rules your prefer so that you can use this tool for yourself.

Python has a built in `sort` function that can sort a list. Given a list of file names, it will sort them in *lexicographical order*, which is similar to alphabetical order (look the term up if you do not know what it means). Since we want to sort by a different order, we must do things a little differently. When given a list of tuples, `sort` compares the first item in each tuple. If they are different, it uses them to decide which element should come first. If they are the same, it checks the second element in each tuple as a tie-breaker, and so on.

`sortByMTime` will create a list of tuples based on the list of files. The first element in each tuple will be the time it was last modified (expressed as the number of seconds since January 1, 1970), and the second element will be the file name. You will sort this list, then make the file list by running through the sorted list in order and appending the file name from each tuple to your result list.

To get the modified time for a file, first get the complete path (use `os.path.join` as before), then call `os.path.getmtime` on that full path. Try this in a python shell a few times to see how it works. Then construct a tuple with the modified time first, and the file name second. This will use the existing file name as a tie-breaker. Optionally, you can make a tuple with three elements: the modified time, the file name converted to lower case, and the actual file name. Then, the first tie-breaker will compare lower-case versions of the file, resulting in a case-insensitive comparison.

After constructing the list of tuples, sort it using the `sort` method.

Finally, run through the list of sorted tuples and extract the original file name from each one. Store that in your result list (append each file name as you find it) and return that list.

For example, if you have these file names in your list:

```
['a.jpg', 'b.jpg', 'c.jpg']
```

Your list of tuples might look this this (with larger numbers):

```
[ (56, 'a.jpg'), (59, 'b.jpg'), (50, 'c.jpg') ]
```

When sorted, it would look like:

```
[ (50, 'c.jpg'), (56, 'a.jpg'), (59, 'b.jpg') ]
```

And the final result list would look like:

```
[ 'c.jpg', 'a.jpg', 'b.jpg' ]
```

This is what `sortByMTime` should return.

Assigning the new names

Write a function called `assignNames` that will be called like this from `main`:

```
newnames = assignNames(prefix, inorder)
```

When numbering the files, make sure that each name has the same number of digits. For example, if there are between 10–99 files, you should use a leading zero for the first 9 files, e.g., 01, 02, ...

Start by figuring out how many digits you will need. One way to do this is to take the total number of files and convert it to a string, then see how long that string is, e.g., if there are 143 files, convert that number to a string, yielding `'143'`, then take the length of that string. To convert a number into a string with leading zeros, use something like this:

```
n = 5
s = '%03d' % n
```

The `%03d` format string tells it to use at least 3 digits, and to pad it with leading zeros instead of leading spaces. Since you do not know in advance how many digits you will need, you can construct the format string, as in:

```
digits = 3
template = '%%0%dd' % digits
n = 5
s = template % n
```

Now you should have the tools you need to assign new names to all of the files. Create a dictionary where the key for each entry is the old name of the file, and the value is the new name, for example:

```
newnames = {
    'c.jpg': 'mypics1.jpg',
    'a.jpg': 'mypics2.jpg',
    'b.jpg': 'mypics3.jpg',
```

```
}
```

Make sure that the new names always have lower-case extensions as well. You can use the same techniques you used when checking the extension earlier to make sure that it is lower case.

Loop through the files in order, assigning each one a name of the form:

```
prefixNNN.ext
```

where `prefix` is the prefix passed in to `assignNames` (the one you figured out in `main`), `NNN` is the number with the correct number of leading digits (start your numbering at 1, not 0), and `.ext` is the original extension from the file in lower case.

Return the dictionary with the new name assignments.

Temporary name

Write a function called `makeTempName` that will be called like this from `main`:

```
tempname = makeTempName(allfiles)
```

It generates a temporary file name that does not already exist in the list of files it is given.

Do this by generating a random number between 0 and 1,000,000,000. Then generate a name like `__tempNNNN__` where `NNN` is the random number you generated. Check if that name appears anywhere in the list of all files. If it does, increment the random number until you can generate a name that does not already exist.

Return the new temporary name. Note that you will only need a single temporary name.

Generating the rename script

Write a function called `makeScript` that will be called like this from `main`:

```
script = makeScript(inorder, newnames, tempname)
```

This function is responsible for figuring out the precise sequence of renames that should take place, a sequence that we will call the "script".

The first parameter contains the complete list of files that need to be renamed. The second is the dictionary giving the new name for each file. As the rename for each file is added to the script, you will remove it from the `newnames` dictionary. Anything that is left in the dictionary still needs to be renamed.

Start with an empty script (just an empty list). Loop through the list of files (the `inorder` parameter) and consider each one. There are a few cases to handle:

1. The file does not appear in `newnames`. In this case, its rename action has already been added to the script and no further action is needed.
2. The new name for the file is the same as the old name. Remove it from the `newnames` dictionary (using the Python `del` keyword) and continue. No further action is required. To delete an element from a dictionary, use:

   ```
   del newnames[key]
   ```

3. The file can be renamed without causing any problems. Get the new name for the file, then check if that new name appears in the `newnames` dictionary as a key. If it does not, then the file can be renamed safely. Add a tuple to the script containing the old name and the new name. Then delete the element from the `newnames` dictionary.
4. The target name already exists, so you must work around the conflict. The basic idea here is to create a chain of renames that must occur before this one can. For example, if you are trying to rename a file called "a.jpg" and `newnames` contains these entries:

   ```
   {
       'a.jpg':            'mypics3.jpg',
       'mypics3.jpg':      'mypics4.jpg',
       'mypics4.jpg':      'mypics5.jpg',
   ```

```
}
```

Then your goal will be to create this chain of renames that must happen first, listed here in reverse order:

```
[
    ('a.jpg', 'mypics3.jpg'),
    ('mypics3.jpg', 'mypics4.jpg'),
    ('mypics4.jpg', 'mypics5.jpg'),
]
```

You can interpret this list as:

- "rename a.jpg to mypics3.jpg, but first..."
- "rename mypics3.jpg to mypics4.jpg, but first..."
- "rename mypics4.jpg to mypics5.jpg"

Write a `while` loop to generate this chain. It follows this pseudo-code:

- `chain` is an empty list, `inthechain` is an empty dictionary, and `link` is the old name of the current file.
- loop forever
    - get the target name from `newnames`
    - add `(link, targetname)` to the end of the chain
    - set `inthechain[link]` to True
    - set `link` to be the target name
    - break out of the loop if the new `link` is not in newnamess

This will build the chain, and it will also keep track of all of the files in the chain, which will be useful later.

After the loop ends, reverse the order of the chain (using `chain.reverse()`), and then add each entry to the script. Delete each file entry from `newnames` at the same time.

This will handle most cases, but not cases involving loops of renames. Test it on this data by loading it into an interactive shell and calling it directly:

```
inorder = ['mypics1.jpg', 'mypics2.jpg', 'a.jpg',
    'mypics3.jpg', 'mypics4.jpg']
newnames = {
    'mypics1.jpg': 'mypics1.jpg',
    'mypics2.jpg': 'mypics2.jpg',
    'a.jpg': 'mypics3.jpg',
    'mypics3.jpg': 'mypics4.jpg',
    'mypics4.jpg': 'mypics5.jpg',
}
makeScript(inorder, newnames, '__temp1234__')
```

This should return:

```
[
    ('mypics4.jpg', 'mypics5.jpg'),
    ('mypics3.jpg', 'mypics4.jpg'),
    ('a.jpg', 'mypics3.jpg')
]
```

and `newnames` should be empty after it returns. Note that there are no entries in the script for `mypics1.jpg` and `mypics2.jpg`, since no action is required for them. The order that the script specifies for the remaining items will ensure that no data is lost as the files are renamed.

We will come back to the case involving cycles.

Perform the renames

Write a function called `doRenames` that will be called from `main` as:

```
doRenames(root, script)
```

This function loops through the actions in the script and performs each one. Each entry in the script has an old name and a new name. Print a message before renaming for each one so the output will look like this:

```
mypics4.jpg -> mypics5.jpg
mypics3.jpg -> mypics4.jpg
a.jpg -> mypics3.jpg
```

Get the full path for each one by using `os.path.join` to merge the root (the directory) with the file names. Before actually performing the rename, verify that the new name does not exist using `os.path.exists`. If it does, print an error message and quit using `os.exit(1)`.

If everything is okay, us `os.rename` to rename each file.

Handling cycles

Go back to `makeScript` and enhance it to handle cycles in the renaming script. For example, with this input:

```
inorder = ['mypics1.jpg', 'mypics2.jpg', 'mypics3.jpg', 'mypics4.jpg']
newnames = {
    'mypics1.jpg': 'mypics2.jpg',
    'mypics2.jpg': 'mypics3.jpg',
    'mypics3.jpg': 'mypics4.jpg',
    'mypics4.jpg': 'mypics1.jpg',
}
makeScript(inorder, newnames, '__temp1234__')
```

The script that the existing code will find fails. You must break the cycle of rename dependencies. Here is how you can do it.

Within the loop that is building a chain of rename dependencies, before adding an element to the chain, check if the target name is already in the chain (it will appear in the `inthechain` dictionary). If so, you have found a cycle. To break it, perform the following steps:

1. Change the entry in `newnames` for this link so that it will be renamed to the temporary name.
2. Add an entry to `newnames` for the temporary name, setting it to be renamed to the actual target name. These actions effectively "pretend" that it was supposed to be renamed in two steps all along.
3. Add an entry to the chain changing the old name to the temporary name.
4. Insert an entry at the *beginning* of the chain changing the temporary name to the actual target name. Use the `insert` method of python lists to accomplish this.
5. Break out of the loop.

The existing code will reverse the chain and add the entries to the script as usual, with the effect that the first action will be to rename the entry to the temporary name, and the last action will be to rename the temporary name to the final target name. Everything else will work as usual.

Test it using the example data given earlier. Make sure you have not broken the normal cases; every case should work now.

Testing your code

You should test each function in isolation as you go. A few examples are given for the trickier functions, but you should test all of them as you go by calling them directly. It is much easier to test small functions in isolation than to test an entire program with many moving parts.

When you are satisfied that everything works, create some directories with test data. Test simple cases, cases with chains of renames that must happen in the right order, and cases that involve cycles.

It will be helpful to be able to change the time stamp on a file. Within Linux or Mac OS, you can use the `touch` command to set a file's timestamp to the current time:

```
touch mypics1.jpg
```

Within Windows, you can do the same using this command from the command line:

```
copy /b mypics1.jpg +,,
```

You can also rename files from within the GUI tools. Run your script to put everying in order by timestamp, then rename a few files to change the order. Basically do the reverse of what you renaming tool will have to do: rename a file to a temporary name, then rename others to fill in the gap, then rename the temporary name to its final name.

Using a GUI to do this will ultimately be more work. Learning to use a command-line tool normally takes more learning up front, but makes the job easier in the long run.