filelist

CMPE230 Project 2

Spring 2018

Alper Çakan
2016400000

1. Summary

In this document, we elaborate the design of a script, **filelist**, which is a file utility program that can traverse directories and report path names of files that satisfy some search criteria. The program is implemented in Python 2.7, and was tested with Python 2.7.14 on macOS 10.13.4. The program supports a variety of filtering options based on the size, name, and modification date and the content of the files. Also, the program supports grouping options such as "group by duplicate name" and "group by duplicate content". Furthermore, the program support operations such as zipping or deletion of the filtered files.

In the following sections, the details of usage, implementation and design choices will be explained.

2. Usage

To be able to use the program, you must have Python 2.7 on your system. Once it is ready, first cd to the directory in which the script filelist.py resides. Then, set the execution bit of this file. Now, you can simply run the program from your terminal. We said that you need Python 2.7 because it was only tested with it. However, the program might also work just fine with another version.

The syntax of the command line arguments for the program is as follows:

```
filelist [options] [directory list]
```

where both the options and the directory list is optional. The complete list of the possible options can be found the project description. However, you should consult the following bullet list for the details about the options which also explains some of the design choices made by us:

- -duplcont and -duplname cannot be used at the same time.
- Giving -nofilelist and one (or both) of -duplcont and -duplname is illegal (this is because -nofilelist requires no file listing but the others requires listing of files)
- When -duplcont and/or -duplname is given, no operational option (i.e., -zip and -delete) can be given. This is because -duplcont and -duplname are for listing purposes only and it is not clear which/how files should be operated on when these options are present.
- -zip and -delete cannot be given at the same time. This is for safety reasons so that you do not lost any of your files.
- When a selector option which expects an argument (such as -bigger, -after etc.) is given more than once, only the last one is considered (ex: "-bigger 7 -bigger 5" is the same as "-bigger 5").
- The values in -bigger and -smaller can be floating point values.
- The match option is full-match. (ex: the pattern "a" will not match "abc" even though it matches some part of it).
- When -zip option is used, all the zipped files will be in the archive root regardless of their original path. Also, the name collisions will be resolved by adding a number inside parenthesis to the beginning of colliding file names (ex: when there are two files name "myfile", one of the will be "myfile" and the other will be "(1) myfile").
- All bounds (modification date, size etc.) are inclusive.

- When one of the command line arguments is not a legal option, that arguments and all of the following arguments are assumed to be paths.

3. Implementation

The general logic of the program can be summarized as follow: Starting from the given traversal roots, the file system is recursively examined on each file encountered, a bunch of selectors is applied. The files which are selected are listed. Also, if some operational options are supplied, the selected files are acted on. Furthermore, if "-stats" option is given, some statistics about the traversal is printed. Although the comments and the documentation in the code is sufficient to explain most of the implementations, the following are some selected important points about the implementation:

- The command line arguments are parsed and processed at the beginning of the program and later on they are never used in their raw format. The parser return a quadruple which is composed of selectors, operations, output modes and paths. The paths is the list of the supplied paths which are to be the roots of the traversal. The selectors, operations and output modes are dictionaries. All of them are about the given command line arguments, but they are grouped so that the different parts of the program uses only the necessary parts. In all three of these dictionaries, the keys are the command line options and the values are the arguments of the options. For example, if "-bigger 3" is given, selector['-bigger'] will be 3. For options which do not expect an argument, the value in the dictionary is True on existence of the option and False when the options is not supplied. For example, if "-delete" is given, operations['-delete'] will be True. Also, we fill the dictionaries with all of the legal options so that rest of the program does not need to worry about the existence of keys. For example, when "-delete" option is not given, the key '-delete' still exists in the dictionary operations but now the value is False. Furthermore, the arguments are processed. For example, when "-bigger 3K" is given, we get selectors['-bigger'] = 3072 so that the selector function does not need to worry about strings and validations and so on.
- To make life easier in traversal visit table and so on, we always use absolute paths (so that we do not deal with ./../ etc.).
- The selection-traversal logic is as follows. We have a traversal method which accepts only one root. But this method also accepts a set which holds the paths of selected files and a dictionary which holds the visited files and directories. As a result, we can call this method with different traversal roots and still there will be no problem of visiting the same directory twice or so. This traversal method implements the usual BFS algorithms: the files and directories in the current "node" are obtained, the directories are marked visited and enqueued if they are not already visited, the file paths are printed if they are selected by the selectors. We have a separate method for checking if a file is selected by the selectors. That method takes as argument the dictionary of processed selectors (which we have explained above how we obtain it) and tests all selectors one by one on the given file. If a selection test fails, it returns False. At the end, it returns True. With this algorithm, we can easily traverse the file system and print the files which satisfy the given criterion. After all of the traversals are completed, we execute the methods which print the duplicates, if the corresponding options are given. And at the end, we execute the operational methods if the corresponding options are given (-zip, -delete).
- For zipping, we use the Python standard module zipfile.
- For file content duplication check, we use SHA1 hash.

- For duplication check, we use a bucket system. Since the Python's dictionary data structure is very efficient, we used it for the bucket system. Depending on the option -duplname or -dupcont, file name or file content hash, respectively, is used as the key of the bucket. Each bucket is a list of absolute paths. We used lists as buckets because the duplicate printer function already accepts a set, so we know that there are no duplicates. Hence, we do not need to use sets as buckets and waste our time (since list append is more efficient that set append)
- The match selector was one of the problems in the implementation. Since Python 2 does not fullmatch method unlike Python 3, our only choice was the partial match method. However, partial match is not the behavior that a user would expect. For example, the regular expression pattern "a" matches "abc" whereas a user would expect that it only matches "a". As a solution, we wrap the given regular expression pattern in parentheses and end-of-string "character", which is \Z. Since the partial match already matches only from the beginning, we did not need to add head-of-string "character". With this solution, the behavior is as expected now.

4. Documentation

For documentation, we used docstrings.