

Backup utility Problem decomposition

I. General behavior of a backup utility.

Choose a root in a directory tree.

If there exists a backup file previously created from this root, find all files below the chosen root that have been changed since the date of the most recent backup file created from this root. Write those files to a new backup file of type ZipFile.

If no previous backup file exists, write all files below this root to a backup file. Create a text log file that lists the paths to all of the files that were written to the backupfile.

Note: Most changes made to files are small. An even smarter backup utility would find the differences between old and new versions of the files that have been changed, and only backup the differences.

Thinking of this reminds us that making backup copies of files only helps us if we are able to retrieve the files later. If you were doing the backup utility for real, you would also want to write a program to restore files. Generally, the more complicated the procedure for saving files, the more complicated the procedure for retrieving them.

II. Pathnames, files, walking a directory.

Choose a root in a directory tree.

This is the specification of fromPath. You will need a way to do this. Simplest is to have the user enter it as an argument when the backup() function is called.

What does os.walk(path) return? What is the structure of the returned object?

the line of code:

```
for (root, dirs, files) in os.walk(path):
```

gives you access to the parts of the directory.

From these parts, you can build a list of the files in a directory.

```
for (root, dirs, files) in os.walk(path):  
    print files
```

gives you a list of the files below the root "path", then a list of files in each of the directories below root.

To get the full path name for each file, you need to join the name to the path name of the directory it is in. You can do this with the os.path.join() function:

```
fpath = os.path.join(root, fileName)
```

You have a list of names of files. You have now reconstructed full path names for each file. You can now construct a tuple for each file that consists of its full path name, its modification time, and its size.

Consult the documentation for the `os.path` module for functions that will return the modification time and size. Since you will also be using functions from the `time` module, do not forget to import that module as well.

Once you have replaced each item in the list of files with a tuple with structure: `(pathname, modTime, size)`, you can sort the new list.

III. Making comparisons, sorting, choosing the files, processing lists.

But how do you sort a list of tuples? And how do you sort the list of tuples when you want to sort on the basis of the second item in the tuple?

As Rik explained in class on 5/27, the `sort` method can be invoked on a list object and called with an argument that specifies the function to be used when making the comparisons inside the sorting procedure.

Imagine two tuples, `a` and `b`, each with the structure described above. `a[1]` will return the `modTime` of the `a` tuple, and `b[1]` will return the `modTime` of the `b` tuple.

If you write a function that returns 1 when `a[1]` is greater than `b[1]`, returns -1 when `a[1]` is less than `b[1]`, and returns 0 otherwise, you can then call the `sort` method with this function. For example, if you defined the function `tupleCompare` as described above, and if your list of file tuples was called `fileTups`, you could sort the list of file tuples according to `modTime` using this line of code:

```
fileTups.sort(tupleCompare)
```

Notice that this is **not** a function call. It is invoking the `sort` **method** on the `fileTups` object (which is a list), using the function `tupleCompare` inside the method to do the comparison. There is more on the difference between function calls and method invocation in the final section below.

There is also a built-in method for reversing a list (see the documentation of built-in functions and methods).

Since the `"for (root, dirs, files) in os.walk(path)"` construction returns a separate list for each directory in the tree below the root, you will want to collect the lists into a larger list.

Putting these pieces together, you should be able to create a `traverse()` function that behaves as specified in the assignment.

Designing the backup function.

The desired calling syntax is: `backup(buName, fromPath, buDir)`

The backup utility should only make copies of files that have been modified since the time of the last backup operation. Assume that the name of the backup file will indicate something about the root from which the backup is relative. That means that you will need to look in your backup directory to find the date of the most recent relevant backup file there. Finding a relevant file means you will have to consider only files that have the same `buName` as the one you propose to create.

Now that you have a `traverse()` function, you can run that on the backup directory, `buDir`, to return a list of tuples representing the files in the backup directory.

You can write a function to determine if the string you are using for the `buName` is present in the names of the files in the backup directory. If such a name is present, this means you already have a backup by this name (but a different timestamp), so you only want to copy files that are more recent than the most recent of these similarly named files. You can use some of the concepts from the `traverse()` function to write a function that looks at the list returned by `traverse(buDir)` and returns the `modTime` of the most recently modified backup file.

So now, you could check to see if there is already a file with the same backup name in the `buDir`? If so, then only copy files that have been modified since the `modTime` of the most recently created such backup file. If there is no such file, then write all files below the `fromPath` root to the backup file. If there is already a file with the same backup name in the `buDir`, but no files have been modified since the `modTime` of the most recently modified such file, then notify the user that no files need to be backed up.

IV. Creating files and zipfiles and writing to them.

The key concepts necessary to understand writing files are (1) that a file object must first be created with a call to the built-in function `open()`, or `file()`, and (2) then it can be written to using the `write` method. The syntax for the function call and the method invocation are different. For example, you can create a log file object with the `open()` function:

```
log = open(log_dest, 'w')
```

where `'log_dest'` is the path name for the log file that is being created and `'w'` tells the file that it is being opened in `'write'` mode.

To actually write to the file object, you use the `write` method. Method invocation

has a different syntax than function calls. To invoke the write method on the log file object, you use:

```
log.write("what you want to write")
```

Thus the object is named, followed by a period and the method name, then the arguments to the method in parenthesis.

Similarly, zip files are created with a function call from the zipfile module (don't forget to import this module). For example,

```
arch = zipfile.ZipFile(arch_dest, 'w')
```

will create an object of type ZipFile with the path 'arch_dest' (I've named it to remind me of the destination of the archive), in write mode. To actually put something in that file, one would invoke the write method on the object, thus:

```
arch.write("something to compress and write")
```

Of course in your programs, the arguments passed to the write methods will be variables that expand into file names. When you pass a string representing a path name to the log file object as in

```
log.write(pathName)
```

the string will be written to the file.

When you pass a similar string representing a path name to the arch ZipFile object as in

```
arch.write(pathName)
```

the contents of the file itself will be compressed and written to the file.

Once you are finished writing to (or reading from) files, they should be closed. For example,

```
log.close()
```

```
arch.close()
```

If you forget to do this, the data you write will not be saved.