CSCI 235, Programming Languages, Assignment 3

Rules for Assignments:

- You are expected to read the complete assignment and to complete every part of it. 'I did not see this part' will never be a valid excuse for not completing a part of the assignment.
- Submitted code will be checked for correctness, readability, style, and layout. We apply high quality standards during grading.
- If you need help, please post your question on Piazza. Questions that do not contain code can be asked publicly. Questions that contain code must be posted as private question to all instructors. If you are in doubt, post as private question, because instructors can always make the question public later. Don't mail questions directly to an instructor, because individual instructors are not always available. Piazza is watched almost full time, so you will get a quicker answer.
- You must write all submitted code by yourself! We check code for similarity using automatic tools and also by visual inspection. See the syllabus for additional rules. You may be invited for live grading, or for discussion about the quality of your code.

This assignment is the same as Assignment 2, but using Python instead of Java. The task is still to create a program that recursively walks through a directory tree, and constructs an index of all words that occur in the files in the visited directories. Suppose that we have the following directory structure:

```
dir1/
   f1
   dir2/
      f2
      f3
```

Suppose that file `f1` contains

```
have a
   great day !!!
```

File `f2` contains

```
   Day and Night
```

File `f3` contains

```
   Dies Irae,
   Dies Illa
```

Then starting the program with `dir1` should result in the following index:

```
"a" has 1 occurrences(s):
   in file dir1/f1
      at line 1, column 6
"and" has 1 occurrences(s):
   in file dir1/dir2/f2
      at line 1, column 5
"day" has 2 occurrences(s):
   in file dir1/dir2/f2
      at line 1, column 1
   in file dir1/f1
      at line 2, column 10
"dies" has 2 occurrences(s):
   in file dir1/dir2/f3
      at line 1, column 1
      at line 2, column 1
"great" has 1 occurrences(s):
   in file dir1/f1
      at line 2, column 4
"have" has 1 occurrences(s):
   in file dir1/f1
      at line 1, column 1
"illa" has 1 occurrences(s):
   in file dir1/dir2/f3
      at line 2, column 6
"irae" has 1 occurrences(s):
   in file dir1/dir2/f3
      at line 1, column 6
"night" has 1 occurrences(s):
   in file dir1/dir2/f2
      at line 1, column 9
```

As you can see, the words are sorted alphabetically, and for each word the occurrences are grouped by file name which are also sorted. Within a single file, the occurrences are sorted, first by line, then by column. It is important that your implementation will sort its output the same way. Words are always converted to lower case. Because of this, `Day` and `day` are treated as the same word. This does not apply to filenames: In filenames, we keep the distinction between upper and lower case, because the operating system may make this distinction.

In order to decide what letters are allowed in words, use the function `inWord(ch)` in file `syntax.py`, which is given. This function returns `True` if `ch` is allowed in a word. It allows letters, digits, single quotes, and the minus sign. A word is a non-empty, connected sequence of characters that are allowed by `inWord`. A line containing `abc123@#%PytHonIsCraZy!#$%'21-12'` will result in the words `abc123`, `pythoniscrazy` and `'21-12'`.

In order to make a word lower case, use the built-in method `lower( )` of `str`.

In addition to the function `inWord(ch)`, the file `syntax.py` contains a function `isNewLine(ch)`. Don't makes changes in the file `syntax.py`.

The file `assignment03.zip` contains `syntax.py` and a file `buildindex.py` containing a test function (which you should treat as **const**), test directory `dir1` (same as above), a bigger test directory `test_dir` with expected output `expected_output.txt` (same as in previous assignment), and starter files `fileposition.py`, `occurrences.py`, `filewalker.py`. After completing the assignment, you must submit these three files.

In the starter code, functions and classes have been annotated with type hints. They can be used by `mypy` to check type correctness of your code. We recommend that you use `mypy` frequently, because it detects many problems in code without running it. You are not allowed to change the type hints.

Tasks

1. First the class `FilePosition` needs to be completed.

   Since we are going to store file positions in a set, we need a hash method and an equality operator. Write the class methods `__hash__( self ) -> int` and `__eq__( self, other ) -> bool`.

   If everything goes well, function `__eq__` will be called very rarely. (Why?) If you want, you can put a print function in it to check that it is indeed called rarely. Don't forget to remove it before submitting.

2. Since we want to print file positions to the user in sorted order, we will also need ordered comparison on `FilePosition`. Write `__lt__( self, other ) -> bool`.

3. Next we will complete the class `Occurrences`. It has a single field

   ```
   occs : Dict[ str, Dict[ str, Set[ FilePosition ]]]
   ```

   which is similar to the field

   ```
   TreeMap<String, TreeMap<String, TreeSet<FilePosition>>> occMap
   ```

   in the Java assignment. Its structure was explained there. The main difference here is that Python has no built-in, tree-based dictionaries. This means that we will have to sort entries during printing.

   Create a method

```
def add( self, word : str, filepath : str, pos : FilePosition ) -> None
```

If a word has no occurrences yet, `add` must first associate an empty dictionary to `word`, and after that do the insertion.

Similarly, if `add` sees a word for the first time in a file, it must first create a new set for the given combination of word and filepath.

Add

```
def __repr__( self ) -> str :
   return str( self. occs )
```

so that you can see the result.

4. Python distinguishes between printing for the developer (ugly, using method `__repr__`) and printing for the user (pretty, using method `__str__`). Ugly print was easy to implement, because Python dictionaries and sets are printable.

   Now we will implement pretty-print `__str__( self ) -> str`. We are sure that our user wants to see the words, the filenames, and the occurrences sorted.

   In order to do this, collect the keys of the dictionary in a list, use the `sorted( )` function to sort it, and after that, iterate through the sorted list.

   Printing a set in sorted fashion is similar. Collect the elements in a list, sort the list, and iterate through it.

   Make sure that the output looks similar to the example above and `expected_output.txt`.

5. Create a class method `distinctWords( self ) -> int` that returns the number of distinct words in the given `Occurrences` object.

6. Create a class method

   ```
   def totalOccurrences( self, word : Optional[str] = None,
                               fname : Optional[str] = None ) -> int :
   ```

   that can be called in three different ways:

   (a) `totalOccurrences( )`: Returns the total number of word occurrences. The difference with `distinctWords( )` is that this method counts each encountered word once, while `totalOccurrences( )` counts the separate occurrences of each word.

   (b) `totalOccurrences( word )` : Returns the total number of occurrences of `word`. The method must return 0 when there is no occurrence of `word`.

(c) `totalOccurrences( word, fname )` : Returns the total number of occurrences of `word` in the file `fname`. The method must return 0 when there are no occurrences of word `word` in `fname`.

That completes the class `Occurrences`. Test your implementation carefully. Also use `mypy occurrences.py`, because it detects many potential problems.

7. In class `FileWalker`, complete the method

```
@staticmethod
def fileIterator( f : TextIO ) -> Generator[
                    Tuple[ str, FilePosition ], None, None ] :
```

This method defines an iterator that, when called with an open (for reading) file `f`, generates the tuples of word occurrences in the file `f`. The tuples must have form `( word, position )`, where `word` is a string, and `position` is a `FilePosition`.

Don't forget to make the word lower case before yielding it. If you want to test the iterator, use

```
import filewalker
for occ in filewalker.FileWalker.fileIterator(
                     open( "dir1/f1", "r", encoding = "utf8" )) :
   print( occ )
```

Here `dir1/f1` is a file that you want to test with. If it is the file from the example above, the output should look like:

```
('have', ( 1, 1 ))
('a', ( 1, 6 ))
('great', ( 2, 4 ))
('day', ( 2, 10 ))
```

The field `encoding = "utf8"` is needed because the files contain Cyrillic and Polish characters. You can call `fileIterator` without object, because it is a static method.

Since Python has no private fields, you can mess with the fields of `FilePosition`, but you should not do that. You should construct a `FilePosition` at the beginning and keep it up to date by calling its `advance(i)` and `nextLine( )` methods. Don't try to classify characters by yourself, use `inWord(ch)` in file `syntax.py`. You can use the `readlines( )` method to iterate through the lines. (So you won't need `isNewLine(ch)` in file `syntax.py`) Check the line numbers and the columns carefully. (We will also do that.)

8. It remains to complete method

```
   def recDirIterator( self ) -> Generator[
                        Tuple[ str, str, FilePosition ], None, None ] :
```

of class `FileWalker`.

It defines an iterator that generates all word occurrences as triples of form `( filename, word, position )`. For example:

```
import filewalker
wlk = filewalker.FileWalker( "dir" )
for occ in wlk. recDirIterator( ) :
   print( occ )

# generates
('dir1/f1', 'have', ( 1, 1 ))
('dir1/f1', 'a', ( 1, 6 ))
# etc.
```

First check that the field `topdir` exists and is a directory. Use `os.path.exists( )` and `os.path.isdir( )`. If not, bail out by raising a suitable exception. You can look on `https://docs.python.org/3/library/exceptions.html` for a complete list.

In order to walk through all files and subdirectories of `topdir`, use `os.walk( self.topdir )`. It creates an iterator that generates triples `( dir, subdirs, files )` where `dir` is a (possibly nested) subdirectory of `topdir`, `subdirs` is a list of the direct subdirectories of `dir`, and `files` is a list of the files in `dir`.

Since `os.walk` already recurses by itself, there is no need to use `subdirs`. Just write another loop that iterates through `files`. In this loop, create the complete filename by joining `dir` with the current filename. Use `os.path.join( dir, filename )` to do this. Any attempt at self-doctoring will result in non-portable code. Once you have the complete file name, you can open it (don't forget to include `encoding = "utf8"`) and iterate through the word occurrences generated by `fileIterator(f)`. These are pairs to which you have to add the filename (so that they become triples).

When your implementation is finished, you can test it on **test_dir**, and your output should look similar to **expected_output.txt**. In order to test your implementation, use `index( topdir : str )` in file `buildindex.py`. This function also creates a file `output_for_topdir.txt`, so that you can check the output. Make sure that you have no important file with this name, because it will be overwritten.

Submission Guidelines: Submit your code to Moodle before the deadline. In order to do this

1. find your project directory on your computer,

2. zip your three files **fileposition.py**, **occurrences.py** and **filewalker.py** into a single file named **assignment03.zip**.
3. Upload this file into Moodle.
4. If possible, double check. Download your zipped file from Moodle into another directory. Unzip it, and check if the resulting files appear correct.

We will likely be using an automatic checker, hence you must upload exactly the files as specified above. Don't use another archiver than zip.