Contents

1 Basic Test Results

```
Starting tests...
Thu Jan 15 19:58:36 IST 2015
4056f36443af52b9040ef228e27db7b5c4009071  -


-rw-r--r-- ransha/stud    2380 2015-01-15 19:55 README
-rwxr--r-- ransha/stud    5091 2015-01-15 19:50 WordTracker.py
-rwxr--r-- ransha/stud    2774 2015-01-15 18:24 WordExtractor.py
-rwxr--r-- ransha/stud    5692 2015-01-15 18:15 PathScanner.py

Testing README...
Done testing README...

Running presubmit tests...
result_code    wtrck_example    4    1
result_code    ps_example    5    1
result_code    wext_example    3    1
Done running presubmit tests

Tests completed
```

2 README

```
ransha
203781000
Ran Shaham

=========================================
=  README for ex10: Iterators and files  =
=========================================

==================
=  Description:  =
==================

In this exercise we implemented 3 programs: one for reading all words in a
certain file, one that determines if a given word is in a certain list and the
last is for searching files containing a given words list in the file system.
All of those were implemented using iterator, classes, files i/o (actually
only i...) and recursion.

=============================
=  List of submitted files: =
=============================

README              This file

WordExtractor.py    An iterator that extracts every word from a file
                    that contains text.
WordTracker.py      A class that takes a 'dictionary' (word list) as and
                    contains methods that sort and search the list.
PathScanner.py      Contains methods that use the other files (from this
                    exercise) and search for file in the file system.


========
= NOTE =
========

 - Part 2 - WordTracker.py -

__contains__() method:
In order to use binary search in the __contains__ function the list of words
needs to be sorted. Therefore, I implemented 'Merge Sort' in order to
efficiently sort the list (_sort_dict() function that runs on __init__).
The instructions specified the implementation need to be efficient in terms
of runtime complexity, so the merge sort, in my opinion, is a good chioce.
Although it uses recursion and a relatively large amount of memory, it runs
in O(nlog(n)) which is better (or at least as good) than other sorts.

encounter() method:
this function uses the __contains__ method that runs in O(n) and returns its
value. In the __contains__ method the found words list is updated before it
returns True, so that the encountered_all() function will run fast.
Therefore, worst-case runtime complexity will still be O(n).

NOTE - I think binary search has runtime complexity of O(log(n)) not like
mentioned in the instructions (O(n)).


encountered_all() method:
Goes over every item in the found words list (max n items) to check if one of
```

```
the words wasn't found. therefore: worst-case runtime complexity of O(n).

reset() method:
Sets a new list of 'False' values, thus resetting the search for words.
Runtime complexity of O(n).
```

3 PathScanner.py

```python
import os
from WordExtractor import *
from WordTracker import *

class PathIterator:
    """
    An iterator which iterates over all the directories and files
    in a given path (note - in the path only, not in the
    full depth). There is no importance to the order of iteration.
    """
    def __init__(self, path):
        self.items = os.listdir(path)
        self.index = 0
        self.path = path
        self.SEPERATOR = '/'

    def __iter__(self):
        return self

    def __next__(self):
        '''
        Goes over the items list, as given by the os.listdir() function one
        item at a time.
        returns the path of a file/dir in the directory as string
        '''
        if self.index >= len(self.items): raise StopIteration()
        item = self.items[self.index]
        self.index += 1
        return self.path + self.SEPERATOR + item



def path_iterator(path):
    """
    Returns an iterator to the current path's filed and directories.
    Note - the iterator class is not outlined in the original
     version of this file - but rather is should be designed
     and implemented by you.
    :param path: A (relative or an absolute) path to a directory.
    It can be assumed that the path is valid and that indeed it
    leads to a directory (and not to a file).
    :return: An iterator which returns all the files and directories
    in the *current* path (but not in the *full depth* of the path).
    """
    return PathIterator(path)

def print_tree(path, sep='  '):
    """
    Print the full hierarchical tree of the given path.
    Recursively print the full depth of the given path such that
    only the files and directory names should be printed (and not
    their full path), each in its own line preceded by a number
    of separators (indicated by the sep parameter) that correlates
    to the hierarchical depth relative to the given path parameter.
    :param path: A (relative or an absolute) path to a directory.
    It can be assumed that the path is valid and that indeed it
    leads to a directory (and not to a file).
    :param sep: A string separator which indicates the depth of
     current hierarchy.
```

```python
    """

    def recursive_tree(path, sep, depth):
        '''
        The recursive function that goes over the path.
        Gets a path of a DIRECTORY, prints every file in it and recursively
        enters directories in it.
        :param path: A relative or an absolute path to a directory.
        :type path: string
        :param sep: a string which indicates the depth of current hierarchy.
        :type sep: string
        :param depth: The current directory depth (distance from input folder)
        :type depth: int
        '''
        # an index for the 'absolute' name of file (or dir) inside 'path'
        path_index = len(path) + 1
        items = path_iterator(path)
        for item in items:
            print(sep * depth + item[path_index:])
            if os.path.isdir(item):
                recursive_tree(item, sep, depth + 1)
    # starts printing with 0 depth.
    recursive_tree(path,sep,0)




def file_with_all_words(path, word_list):
    """
    Find a file in the full depth of the given path which contains
    all the words in word_list.
    Recursively go over  the files in the full depth of the given
    path. For each, check whether it contains all the words in
     word_list and if so return it.
    :param path: A (relative or an absolute) path to a directory.
    In the full path of this directory the search should take place.
    It can be assumed that the path is valid and that indeed it
    leads to a directory (and not to a file).
    :param word_list: A list of words (of strings). The search is for
    a file which contains this list of words.
    :return: The path to a single file which contains all the
    words in word_list if such exists, and None otherwise.
    If there exists more than one file which contains all the
    words in word_list in the full depth of the given path, just one
    of theses should be returned (does not matter which).
    """

    def recursive_search(path, word_tracker):
        '''
        Recursive search for the file containing all of the words in
        the input word_list, using the class WordTracker.
        :param path: Relative or absolute path of a directory. a string.
        :param word_tracker: a WordTracker object that holds the word list
        that we are looking for in the file system.
        '''
        items = path_iterator(path)

        for item in items:
            if os.path.isfile(item):

                try: # to avoid file system errors
                    words = WordExtractor(item)
                    for word in words: # checks every word in the file
                        word_tracker.encounter(word)
                    if word_tracker.encountered_all():
                        return item # if desired file was found stop searching
```

```
                    else:
                        word_tracker.reset()
                except: # will be reached if the file wasn't meant to be read.
                        # if so, resets the tracker and continues the search.
                    word_tracker.reset()

            elif os.path.isdir(item): # recursive search in directories
                recursive_search(item, word_tracker)

        return None # reached if no file was found.
    # let the search begin.
    word_tracker = WordTracker(word_list)
    return recursive_search(path, word_tracker)
```

4 WordExtractor.py

```python
#!/usr/bin/env python3


class WordExtractor(object):
    """
    This class should be used to iterate over words contained in files.
     The class should maintain space complexity of O(1); i.e, regardless
     of the size of the iterated file, the memory requirements ofa class
     instance should be bounded by some constant.
     To comply with the space requirement, the implementation may assume
     that all words and lines in the iterated file are bounded by some
     constant, so it is allowed to read words or lines from the
     iterated file (but not all of them at once).
    """

    def __init__(self, filename):
        """
        Initiate a new WordExtractor instance whose *source file* is
        indicated by filename.
        :param filename: A string representing the path to the instance's
        *source file*
        """

        self.NEW_LINE = '\n'
        self.file = open(filename)
        self.line = self.file.readline() # the line string
        self.splitted = self.line.split() # list of words in the line
        self.curr_index = 0
        self.end_of_file = False


    def __iter__(self):
        """
        Returns an iterator which iterates over the words in the
        *source file* (i.e - self)
        :return: An iterator which iterates over the words in the
        *source file*
        """

        return self

    def _next_line(self):
        '''
        Reads next line in the file and saves it to the class.
        '''
        self.line = self.file.readline()
        self.splitted = self.line.split()
        self.curr_index = 0

    def _bad_line(self,line):
        '''
        Checks if a certain line contains no readable words (string).
        :param line: a string that represents a line in the file.
        :return: >True if the line is empty or if the end of line is reached
                 >False if the line is good or it is the end of file.
        '''

        self.end_of_file = len(line) == 0
        empty_line = len(self.splitted) == 0
```

```
        end_of_line = self.curr_index >= len(self.splitted)

        # the end of file line isn't a bad line.
        return (empty_line or end_of_line) and (not self.end_of_file)

    def __next__(self):
        """
        Make a single word iteration over the source file.
        :return: A word from the file.
        """

        while self._bad_line(self.line):
            self._next_line() # only read good lines.

        if not self.end_of_file:
            word = self.splitted[self.curr_index]
            self.curr_index += 1
            return word
        else:
            self.file.close()
            raise StopIteration()


```

5 WordTracker.py

```python
#!/usr/bin/env python3


class WordTracker(object):
    """
    This class is used to track occurrences of words.
     The class uses a fixed list of words as its dictionary
     (note - 'dictionary' in this context is just a name and does
     not refer to the pythonic concept of dictionaries).
     The class maintains the occurrences of words in its
     dictionary as to be able to report if all dictionary's words
     were encountered.
    """

    def __init__(self, word_list):
        """
        Initiates a new WordTracker instance.
        :param word_list: The instance's dictionary.
        """
        self.dict = word_list[:]              # sets a NEW list
        self._sort_dict()                     # sorts it
        self.found = [False] * len(self.dict) # list of booleans that flags
#        self.found_idx = 0                    # found words.

    def _sort_dict(self):
        '''
        Uses merge sort algorithm and python string comparison to sort the
        given list of words. Uses recursion.
        :return: a sorted list of words.
        '''
        def merge_sort(lst):
            '''
            the recursive sort.
            '''
            size = len(lst)
            if size <= 1:
                return lst # sorted list (contains 1 or less elements)
            middle = size // 2
            left, right = merge_sort(lst[:middle]), merge_sort(lst[middle:])
            lst = merge(left, right) # the merging
            return lst # returns a sorted list

        def merge(left, right):
            '''
            Merges two lists according to its' items' values.
            Gets two parameters 'left','right' as lists and merges them.
            :return: merged list.
            '''
            lst = left + right
            size = len(lst)
            l, r = 0, 0
            for i in range(size):
                l_val = left[l] if l < len(left) else None
                r_val = right[r] if r < len(right) else None
                r_bigger = l_val and r_val and l_val < r_val
                l_bigger = l_val and r_val and l_val >= r_val
                if r_bigger or r_val == None:
                    lst[i] = l_val
                    l += 1
```

```
                elif l_bigger or l_val == None:
                    lst[i] = r_val
                    r += 1
            return lst
        # calls the recursive sort.
        self.dict = merge_sort(self.dict)

    def __contains__(self, word):
        """
        Check if the input word in contained within dictionary.
         For a dictionary with n entries, this method guarantees a
         worst-case running time of O(n) by implementing a
         binary-search.
        :param word: The word to be examined if contained in the
        dictionary.
        :return: True if word is contained in the dictionary,
        False otherwise.
        """

        is_in_dict = False
        lst = self.dict
        left = 0
        right = len(lst)

        while left <= right and right <= len(lst) and left >= 0:
            middle_i = (right+left) // 2
            middle = lst[middle_i]
            if middle > word:
                if middle_i <= 0: break # to avoid infinite loop
                right = middle_i - 1
            elif middle < word:
                if middle_i >= len(lst) -1: break # same as ^
                left = middle_i + 1
            else:
                is_in_dict = True
                self.found[middle_i] = is_in_dict
                break
            # if the word isn't in the dictionary the middle_i index can
            # reach the end of the list, and that can cause infinite search.
        return is_in_dict


    def encounter(self, word):
        """
        A "report" that the give word was encountered.
        The implementation changes the internal state of the object as
        to "remember" this encounter.
        :param word: The encountered word.
        :return: True if the given word is contained in the dictionary,
        False otherwise.
        """
        return word in self # uses the __contains__ method

    def encountered_all(self):
        """
        Checks whether all the words in the dictionary were
        already "encountered".
        :return: True if for each word in the dictionary,
        the encounter function was called with this word;
        False otherwise.
        """
        for found in self.found:
            if not found: return found
        return True

    def reset(self):
        """
        Changes the internal representation of the instance such
```

```
        that it "forget" all past encounters. One implication of
        such forgetfulness is that for encountered_all function
        to return True, all the dictionaries' entries should be
        called with the encounter function (regardless of whether
        they were previously encountered ot not).
        """
        self.found = [False] * len(self.dict)
```