Python Development Test

Silva Haberl

1 Task number 1

Task#1.py

First we create two files **file1.txt** and **file2.txt**. We open them for reading, extract the values, line by line, in the string. For easier manipulation of data, we use the dictionary in which we store data. New values that are stored in a new dictionary in the form of the *name surname id*, we print them to new file we call **fileNew.py**. The methods we use to create dictionaries from the list of data, creating dictionaries based on the same ids of the previous dictionaries, sorting the strings by id and printing the dictionary values into a new file are: **makeDict, resultDict, sortingDict, printToFile**.

```
def makeDict(dataList):
        """Create dictionary from list of data """
        dictionary = {}

        for i in range(0,len(dataList)):
                name, idNum = dataList[i].split('␣')
                dictionary[idNum] = name

        return dictionary

def resultDict(dict1, dict2):
        """Create new dictionary with name and surname
        that have equal id's as in previous two """
        dictionary = {}

        # big O, O(n) coplexity, where n is the number of elements in dict1
        for key,value in dict1.items():
                if dict2[key]:
                        nameSurname = value + '␣' + dict2[key]
                        dictionary[key] = nameSurname
                        nameSurname = ''

        return dictionary

def sortingDict(d):
        """Sort dictionary using sorted list, by id"""
```

```
        dictionary = OrderedDict()
        sortedList = sorted(d)

        for i in range(0,len(sortedList)):
                dictionary[sortedList[i]] = d[sortedList[i]]

        return dictionary

def printToFile(d, new):
        """Print key and values from last
        dictionary (sorted) to new file"""

        for key,value in d.items():
                line = value + ' ' + key + '\n'
                new.write(line)
                line = ''
```

From external library we use OrderedDict from collections.

Task#1e2.py

In the extension two, we have two larger data files. Each of them will be split into the same number of smaller files (n is the number of partitioned files). We use the **grouper** method for partitioning.

```
def grouper(n, iterable, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper(3, 'ABCDEFG', 'x') --> ABC DEF Gxx
    args = [iter(iterable)] * n
    return zip_longest(fillvalue=fillvalue, *args)
```

External library itertools gives us method *zip longest* for grouping data in partition files. Names of files in partition are stored in the *l1* list and in *l2* list.
In double *for loop* we compare each file from list l1 to each file in list l2. That is, we apply the resultDict method n * n times, where n is the length l1 and l2. If the sorted dictionary is not empty, it means that there is at least one id that matches two files and can be printed in a new document.

```
dictResult = resultDict(dictFirst, dictSecond)
sortedDictResult = sortingDict(dictResult)

if bool(sortedDictResult) != False:
        #write to new file
        i += 1
        new_path = 'fileNew{0}.txt'.format(i)
        newList.append(new_path)

        try:
                file_new = open(new_path, 'w')
        except IOError:
```

```
                print("Could not write to file:",
                    new_path)
                sys.exit()
        else:
                printToFile(sortedDictResult, file_new)
                file_new.close()

    joinFiles(newList)
```

After a series of new documents, we want to link them together into a unique document. This is done using the **joinFiles** method.

2 Task number 2

Task#2.py

The main method **balancedBraces** which as an argument receives the given string (text) consists of one *for loop* passing through all the characters of that argument (text). If the input char was a open parenthesis, we use the stack to which we put a corresponding closed parenthesis sign. If the input character was a letter, we put nothing on the stack. If the input character is a kid of closed parenthesis, function will return *True*. Closed parenthesis must match the element from the top of the stack, therefore obtained by the method *stack.pop()*. Otherwise, the method returns *False*.

```
parensAll = iter('(){}[]')
parensDict = dict(zip(parensAll, parensAll))
closingValues = parensDict.values()

def balancedBraces(givenString):
    stack = []
    for char in givenString:
        charClosed = parensDict.get(char, None)
        if charClosed:
            stack.append(charClosed)
        elif char in closingValues:
            if not stack or char != stack.pop():
                return False
    return not stack
```

3 Task number 3

Task#3.py

For the recursive **Fibonacci** function that computes the n-ti Fibonacci number, the values grow very rapidly and computing slows down if all the values have to be counted from the start at each step. To accelerate, we can memorize the values of the function that is already calculated, let's call it *cache*. Method we call *decorator* which contains the *wrapper* method will store values in *cache*. More specifically in this case the names are: **decorator function, wrapper function**. To use decorator function when computing Fibonacci, we need to make a mark *decorator function* before the Fibonacci method.

```
@decorator_function
def Fibonacci(n):
```

In dictionary *dictionaryFunction* are stored already calculated Fibonacci function values. The function argument in wrapper serves as the key to the current count of the natural Fibonacci number. In new dictionary we store data corresponding to current key: its Fibonacci number (*cache*), a counter that goes up to 10 and the last time we marked. From external libraries, we use *datetime* to check if it has passed 300 seconds (5 minutes) to calculate the current Fibo number.

```
import datetime

def decorator_function(original_function):
    """ Decorator that caches result of function calls"""

    dictionaryFunc = original_function.__globals__

    def wrapper_function(*arg, **kwarg):

        if not arg in dictionaryFunc:
            dictionaryFunc[arg] = {}

        dictionary = dictionaryFunc[arg]

        dictionary['counter'] = (dictionary['counter'] + 1) % 10 if 'counter'
           in dictionary else 0

        if (dictionary['counter'] == 0 or (datetime.datetime.now() -
           dictionary['last_update_time']).total_seconds() > 300):

            dictionary['cache'] = original_function(*arg, **kwarg)
            dictionary['counter'] = 0
            dictionary['last_update_time'] = datetime.datetime.now()

            print ("Cache updated :", dictionary['cache'])
        else:
            print ("From cache     :", dictionary['cache'])

        return dictionary['cache']

    return wrapper_function
```