CS1026: Assignment 4: Country Classes

Due: *Monday*, December 6th 2021, 9:00pm
Weight: 12%

Learning Outcomes:
By completing this assignment, you will gain skills relating to

- Strings and text files
- Writing and using your own classes
- Using complex data structures (i.e., lists, sets and dictionaries)
- Testing code and developing test cases; adhering to specifications

Background:
Data files are very common in many disciplines. They form the input for a variety of different kinds of processing, analysis, summaries, etc. An associated problem is to maintain these data files – to add information, update or correct information. Manually updating data files can be tedious and can lead to errors. A common approach is to use a program – an "update" program that takes a data file and a file of "updates" and then creates a new version of the data file with the updates applied.

In this assignment you will create a complete program that will update an existing data file containing information about countries. The tasks are outlined below.

Tasks:
You will create a program called **processUpdates.py,** that will use the information in one file (the "updates" file) to update information in a master file (data.txt) and produce a new file with updated information (output.txt). Your program will make use of a that has a function **processUpdates** that does all the updates (see below). Your program **processUpdates.py** will make use of two classes:

- A class **Country** that holds the information about a single country and is defined in the file **country.py**, and
- A class called **CountryCatalogue** that is defined in the file **catalogue.py .**

Details on updates, **processUpdates.py,** the classes **Country** and **CountryCatalogue** can be found in the following

Task 1: Implement a class **Country** that holds the information about a single country; the file must be named **country.py.**

The class **Country** should have the following **Instance Variables**:

- name: The name of the country (string); country names consist of a sequence of letters and underscores beginning with a capital letter; underscores are used to separate words for countries made up of multiple words, e.g. South_Korea, United_States_of_America.
- population: The population in the country (string)
- area: The area of the country (string)
- continent: The name of the continent to which the country belongs (string); continent names consist of a sequence of letters and underscores; underscores are used to separate words for continents made up of multiple words, e.g. South_America, North_America.

Note 1*: you may include other instance variables as needed.*
Note 2*: the instance variable* population *and* area *can be left as strings for this assignment as there are not specific computations involving either. The update program just needs to update these values and save them to a file.*

The class **Country** should have the following **Methods** with the shown headers:

- Constructor, **__init__(self, name, pop, area, continent)**
  The constructor method will take four parameters (besides "self") with information about the county: name of country, population, area and continent.
- Getter Methods: **getName**, **getPopulation**, **getArea**, **getContinent**
- Setter Methods: **setPopulation**, **setArea**, **setContinent**
- **def __repr__(self)**: generates a string representation for class objects.

  Example of output from **__repr__**
  Name (pop: population value, size: area value) in Continent
  e.g., Nigeria (pop: 11723456, size: 324935) in Africa

  Note*: you may include other methods as needed.*

Test all your classes and methods before moving to the next section. Feel free to create other helper methods if necessary.

Task 2: Implement a class **CountryCatalogue** that holds the information about all the countries; the file must be named **catalogue.py.**

This class will use a file to build the data structures to hold the information about countries. The file **data.txt** contains information about a number of countries. An example of the data file is:

```
Country|Continent|Population|Area
Brazil|South_America|193,364,000|8,511,965
Canada|North_America|34,207,000|9,976,140
China|Asia|1,339,190,000|9,596,960
Colombia||48,654,392|1,090,909
Egypt|Africa|93,383,574|
```

Notice that:

- There is a header line: "`Country|Continent|Population|Area`". This is really for information purpose for the data file; it is a reminder of the format of the data in the file.
- The file consists of "records" for each country – one per line with the information as described in the header line.
- The different "fields" of each "record" are separated by vertical bars, "|"; each line has exactly 3 vertical bars.
- There are no spaces in any of the records.
- A field (**except for country name**) may be missing (see Colombia and Egypt) but the record will still have the three vertical bars; **you may assume that there is always a country name.**

The class **CountryCatalogue** should have the following **Instance Variable**:

- **countryCat:** This is a collection of countries, i.e., each member is an object of the class **Country**. The collection could be a set, dictionary or list. **It is important that the collection store objects of type Country.**
  Note: you may include other instance variables as needed.

The class **CountryCatalogue** should have the following **Methods** with the shown headers:

1. Constructor, **__init__(self, countryFile)**
   The constructor method will take a single parameter (besides "self") that is the name of the file that contains the information about each country. The constructor will use the data in the file to construct the data structure **countryCat.** Sample data files has been provided: **data.txt** and **data2.txt**. [***Note*: Both files have headers and these files are meant for you to test your program; they may NOT be the same used to evaluate your solution.]**

2. Setter Methods: **setPopulationOfCountry**, **setAreaOfCountry**, **setContinentOfCountry**

3. **findCountry(self,country)**
   Given a country object (that is, an object of the **Country** class), this methods checks to see if that country object is in **countryCat** . If it is, it just returns the country object, but if it is not, it returns the null object **NONE**.

4. **addCountry(self,countryName,pop,area,cont)**
   Given the name, population, area and continent of a country, this method adds a new country to **countryCat**; it should only be added if the country does not exist in **countryCat** . The method should return **True** if the operation is successful (country successfully added), or **False** if it is not, e.g. the country they entered already exists.

5. **printCountryCatalogue(self)**
   This method prints a list of the countries and their information to the screen; it should make use of the **repr** method of the **Country** class.

6. **saveCountryCatalogue(self,fname)**
   This method will enable all the country information in the catalogue to be saved to a file. The method takes as a parameter the name of the file. All the countries should be sorted alphabetically by name (A – Z) prior to saving. If the operation is successful, the method should return the number of items written, otherwise it should return a value of -1. ***The file should appear in the exact same format as the original file, namely:***

```
Country|Continent|Population|Area
Brazil|South_America|193,364,000|8,511,965
Canada|North_America|34,207,000|9,976,140
China|Asia|1,339,190,000|9,596,960
Egypt||93,384,001|1,000,000
Indonesia||260,581,345|
```

Notice that:

- The output file has <u>**NO**</u> spaces; there should be no spaces in names, numbers or between the fields and vertical bars.
- If a field has no value, then nothing should be printed, e.g. see Egypt and Indonesia with no continent value and Indonesia with no area given; the vertical bars are printed.
- Each record of the file should have 3 vertical bars.

<u>Note</u>*: you may include other methods as needed.*
Test all your classes and methods before moving to the next section. Feel free to create other helper methods if necessary.

<u>Task 3</u>: Implement a Python module, called **processUpdates.py**, that has a function **processUpdates(cntryFileName,updateFileName,badUpdateFile)**. This function takes three parameters: the **<u>first</u>** parameter will be the name of a file containing country data (e.g. data.txt). The **<u>second</u>** parameter is the name of a file containing updates. The **third** parameter is the name of a file which will contain any bad update records that are found. The function will process the updates in the file **updateFileName** to update the records in **cntryFileName** and will output the updated records to a file "output.txt". Any invalid update records will be written to the file **badUpdateFile** .

Each record in the update file specifies updates for a single country. The format of a record in the update file is as follows:

```
Country;update1;update2;update3
```

Where an "update" is of the form "L=value" and L can be **P**, for population, **A** for area and **C** for continent. A **<u>valid update record</u>**:

- May have spaces, but all spaces should be removed before processing the update record.
- Begins with a name – it is treated as the name of a country. This will be the string before the first semicolon or the only string on the line if there are no semicolons. Country names consist of a sequence of letters and underscores beginning with a capital letter; underscores are used to separate words for countries made up of multiple words, e.g. South_Korea, United_States_of_America. If the string for the country name contains any other characters, the update record is invalid. Country names should be taken exactly as given, e.g. "France" is a valid country name, "france" is not. Your program should not try to correct names.
- Has 0, 1, 2 or 3 semicolons; the semicolons separate the country name and the "updates".
- Has at most one update of the form "P=", "A=", "C="; if there is more than one of these, the update record is considered invalid.
- Valid values in an update are as follows:
  a) For **Population** and **Area**, a valid value is a string of digits with commas separating groups of 3. Examples of valid values are "123,456", "1,333,456", "23,333,444". Examples of invalid values are "123456", 123444555", "123.344.444".
  b) For **Continent**, a **valid value is one of the continents**: "Africa", "Antarctica", "Arctic", "Asia", "Europe", "North_America", "South_America" ***<u>exactly as given</u>***. **<u>These are the only acceptable continent names.</u>** If an update to a continent name has any other string value, then the update is simply ignored. For example, if the update is "C=NorthAmerica" or "C=North_america", then it is an invalid continent update and the update should be ignored. Continent names are treated exactly as entered; there should be no conversion to

upper or lower case: e.g., “asia” and “EUROPE” are both treated as invalid; your program should not try to correct names.

The following are **examples of valid update records:**

`Canada;P=37,591,023`
`Germany;A=213,000;P=83,022,544`
`Germany ; A=213 , 000 ; P=83, 022, 544` ***(valid after spaces removed)***
`Vietnam;P=94,469,067`
`Mexico`
`Mexico;` ***(country, though no updates)***
`France;;A=213,000;P = 67,067,112` ***(3 semicolons, empty update ignored)***
`France; A=213,000 ; P = 67,067,112 ; C=Europe` ***(allowable spaces)***
`United_States_of_America;P=328,566,312` ***(valid country name)***
`United_states_of_america;P=328,566,312` ***(valid country name, but different than*** **`United_States_of_America`*****)***

The following are **examples of invalid update records**:

`Canada ; P=37591023` **(*invalid population)***
`Canada P=37,591,023` **(*country name is “CanadaP=37,591,023” after removing spaces, which is invalid)***
`Indonesia;P 260,581,345` **(*no equal sign, invalid update)***
`germany;A=213,000;P=83,022,544` **(*invalid country name; does not begin with a capital)***
`;A=213,000;P=83,022,544` **(*no country; nothing preceding first semicolon)***
`Mexico;A=213,000;P=83,022,544;A=213,000;P=83,022,544` ***(too many semicolons)***
`France;A=213,000;A=213,067` ***(two area updates)***
`Vietnam;P=94,469,067;C=asia` **(*invalid continent)***

An example of an update file is:

```
Brazil;P=193,364,111;A=8,511,966
Indonesia;P=260,581,345
Japan;P=127,380,555;A=377,800
Sweden;P=9,995,345;A=450,295;C = europe
Italy; P = 59,801,999
Canada;C=North America
Egypt;A=1,000,000 ; P=93,384,001
France;P = 64,668,129;P=64,668,150;A=541,666;C=Europe
```

Notice that:

- The updates can be in different orders, e.g. the updates for Egypt are area first, then population.
- If a country is specified and that country is ***NOT*** already in the catalogue ***then that country and its attributes (at least the ones specified; not all have to be specified) should be added***

to the catalogue (again, if there are more than 3 attributes, the update is invalid and the country is not added).

- There can be spaces; spaces should be removed before processing the update.
- If there is a blank line in the file, it should be skipped.
- In the updates to `France`, for example, there are 4 updates; there are also two updates for population. This update record has two problems; you do not have to detect all the problems with an update record. Once the record is determined to be invalid, it can be rejected.

The function **processUpdates** gets three files as parameters. The first parameter is the name of the file with the country data, the second parameter is the name of file with the update data, and the third parameter is the name of a file that will contain any invalid updates that are found. The function **processUpdates** should return a tuple **(result, catlog)** where **result** is either **True** or **False** depending on whether the update was successful or not and **catlog** is an object of class **countryCatalogue**. If the result of **processUpdates** is **True** then **main.py** will print the catalog **catlog** to the screen using the method **printCountryCatalog**. If the result of **processUpdates** is **False** then **processUpdates** should return a tuple **(False, None)**.

<u>The country file should be processed first</u>. The function **processUpdates** should ensure that it exists.

- ***<u>If it does exist</u>***, it should then proceed to process the data file using the class **CountryCatalogue**.
- ***<u>If the country file DOES NOT not exist</u>***, then **processUpdates** should give the user the option to quit or not (prompts for "Y" (yes) or "N" (no)). If the user does not wish to quit (responds with "N"), then the function should prompt the user for the name of a new file. If the user wishes to quit (responds with a "Y" or anything other than a "N", then the method should exit and return **(False, None)**. The method should continue to prompt for file names until a valid file is found or the user quits.
- ***If the user chooses to quit with no country file processed***, then the function **processUpdates** should **NOT** try to process the updates file and should write to the file "**output.txt**" the message **"Update Unsuccessful\n"** – just the single line. It should then exit (i.e.,) and return **(False, None)**.

If the function **processUpdates** has successfully processed the country file, then it should proceed to process the file of updates.

- It should prompt the user in a similar manner to how the country file was input, i.e., using a loop to continuously prompt the user until a file was found that existed or until the user quits.
- ***If the user quits without an update file being selected***, then the function **processUpdates** should write to the file "**output.txt**" the message **"Update Unsuccessful\n"** – just the single line. It should then exit and return **(False, None)**.

If an update file is found, then the function **processUpdates** should process the updates, i.e., update the information about the countries and then output the new updated list of countries in alphabetical order to the file "**output.txt**". The output should use the method **saveCountryCatalogue(self,fname)** from the class **CountryCatalogue** (see Task 2). It should then exit and return **(True, catlog)**, where **catlog** is the name of the **CountryCatalogue** in your function **processUpdates**.

In processing the update records from the update file, invalid updates may be found (see above). **If an update record is invalid, <u>then the entire line should be skipped and none of the updates processed</u>**. Any invalid update should be written to the bad update file (third parameter) one per line and in the order in which they were found. This file should have the name **"badupdates.txt"** (it is defined in **main.py** and passed to **processUpdates** so you should not have to do anything if you use **main.py**). If there are ***<u>NO</u>*** invalid updates, then this file is left empty.

The Python program **main.py** file is provided as the interface to your program. It has been designed to prompt for the names of a country file and a file of updates and then call the function **processUpdates** with a country file, file of updates and the bad update file; ***<u>make sure that you use these names of functions and methods as described above</u>***. You may use the **main.py** to test your program in other ways. Two additional data sets of countries have been provided, **data2.txt** and **data3.txt**, as well as two different files of updates, **upd2.txt** and **upd3.txt**. You may use these to test your program. ***<u>NOTE:</u>*** **while main.py will test some aspects of your program, it does not do a complete and thorough testing – this is left up to you. A program similar to, but different than, *main.py* will be used to test your program - it will include other tests.**

The following may be helpful in sorting data structures
http://pythoncentral.io/how-to-sort-a-list-tuple-or-object-with-sorted-in-python/
http://pythoncentral.io/how-to-sort-python-dictionaries-by-key-or-value/

<u>Functional Specifications</u>:

1. Your module module, **processUpdates.py**, ***<u>must include</u>*** a function **processUpdates(cntryFileName,updateFileName,badUpdateFile)** that has three parameters: the **<u>first</u>** parameter will be the name of a file containing country data (e.g. data.txt). The **<u>second</u>** parameter is the name of a file containing updates. The **<u>third</u>** parameter is the name of a file which will contain any bad update records that are found. The updated country data is to be written to a **new file**. For this assignment, ***<u>you must use</u>*** the following file names:
   a. The file with the updated country data should be "**output.txt**".
   b. The file of bad update records should be "**badupdates.txt**".

   Not adhering to these constraints will cause the automated testing program to fail.

2. Since the function **processUpdates** takes file names as parameters, these should be input in **main.py** and then **main.py** would call **processUpdates**. The function **processUpdates** should try opening the country data file first. If it is not valid, **processUpdates** should prompt the user until a valid file name is entered or the user quits. Once a valid country data file is found, **it should then** open the updates file. Again, if it is not valid, **processUpdates** should prompt the user until a valid file name is entered or the user quits. urn `True` or `False` depending on whether the function was successful in opening and processing the files.

3. The function **processUpdates** should return a tuple **(result, catlog)** where **result** is either **True** or **False** depending on whether the update was successful or not and **catlog** is an object of class **countryCatalogue**.

4. You are given a program **main.py** that prompts the user for file names and calls **processUpdates**. You should use this to call your function **processUpdates.**

5. Data files with country data are provided (**data2.txt**, **data2.txt** and **data3.txt**), as well as different files of updates (**upd.txt upd2.txt** and **upd3.txt)**. You may use these to test your program. ***NOTE:*** **while main.py will test some aspects of your program, it does not do a complete and thorough testing – this is left up to you. A program similar to, but different than, *main.py* will be used to test your program - it will include other tests.**

Non-functional Specifications:

1. The program should strictly adhere to the input and output requirements and parameters for the function **processUpdates**, **particularly the order of the parameters.**

2. Include brief comments in your code identifying yourself, describing the program, and describing key portions of the code.

3. Assignments are to be done individually and must **be your own work**. Software may be used to detect academic dishonesty (cheating).

4. Use Python coding conventions and good programming techniques, for example:
   - Meaningful variable names
   - Conventions for naming variables and constants
   - Use of constants where appropriate
   - Readability: indentation, white space, consistency

5. You should submit the files:
   - **country.py** containing the class **Country**;
   - **catalogue.py** containing the class **CountryCatalogue;**
   - **processUpdates.py** contain the function **processUpdates**.

   Make sure you upload your Python file to your assignment; ***DO NOT*** put the code inline in the textbox.

Marking of the Assignment:

1. **Your program will be executed by an automated testing program**. This testing program assumes that:
   - The modules are named **main.py, procesUpdates.py, catalogue.py** and **country.py**.
   - That you are using Python Python 3.9 and that everything executes in PyCharm Edu.
   - That you have submitted it via OWL by uploading it.

   Failure to adhere to these constraints will likely cause the testing program to fail. This may require a remarking of your program which will include a 20% penalty.

2. ***Functional specifications are met***; as described above:
   - Are the modules, classes and functions named correctly for testing?
   - Is there a program **processUpdates.py** which has a function **processUpdates**?
   - Does the program behave according to specifications? Does it work on with the test

program, **main.py** ?

- Is there an effective use of functions and methods?
- Is the output according to specifications?

2. ***Non-functional specifications are met***: as described above.
3. ***Assignment submission***: via the OWL, though the assignment submission in OWL.