7CCSMCMP: Coursework 1

Computer Programming for Data Scientists

The 7CCSMCMP instructors

2022

Contents

1 Activity 1 (35 points)

This activity requires you to use many of the concepts learned so far on the module. You will develop a simple eCommerce system. The program has to allow the user to:

- add products;
- remove products;
- show a summary of their shopping session;
- export the content of the shopping cart in *JSON* format.

The activity is divided in three parts that add up to 35 points. You need to submit all parts in a single file called `shopping.py`.

1.1 The domain classes (10 points)

1.1.1 Attributes

First create a class **Product** with the following attributes:

- `name`: a String value;
- `price`: a Float value;
- `quantity`: an Integer value;
- `unique identifier`: a String value that is a 13 digit sequence. Allowed digits in the sequence go from 0 to 9. The identifier must be unique for each product;
- `brand`: a String value.

1.1.2 to_json() method

The class **Product** should also offer a `to_json()` method that returns the *JSON*-formatted representation of the product. This should be a list of key-value pairs describing the product's attributes given in 1.1.1.

1.1.3 Subclasses

Then, create three subclasses of the class **Product**:

- **Clothing**, which has the following additional attributes: `size` and `material`;
- **Food**, with the additional attributes: `expiry_date`, `gluten_free`, and `suitable_for_vegans`; and
- one additional subclass of your choice. Think about the domain of your program and pick a product you are familiar with. Define at least 2 relevant additional attributes of that product.

The types of all these additional attributes are for you to decide, but they need to match the domain and they need to be consistent. For instance, if you defined `gluten_free` as a Boolean, you may not want to define `suitable_for_vegans` as a String. The same applies to the attributes of your subclass of choice.

Each subclass of **Product** must adapt the `to_json()` method of the superclass accordingly.

1.2 The shopping system (10 points)

Next, create the class **ShoppingCart**, which is a container of products in a shopping session. The **ShoppingCart** class offers the following methods:

- `addProduct(p)`, to add a product p to the cart;
- `removeProduct(p)`, to remove the product p from the cart;
- `getContents()`, to obtain the contents of the cart, which should be returned in alphabetical order of product name;
- `changeProductQuantity(p, q)`, to change the quantity of product p to the quantity q .

These methods should be the only way through which the cart's contents can be accessed or altered.

1.3 Doing some shopping (15 points)

Now that the classes are ready, and we have methods in place to change the content of shopping carts, we need some code to do the shopping. In the same Python file, write code that prompts the user to type in commands in a while loop (see Listing 1)

```
print('The program has started.')
print('Insert your next command (H for help):')
terminated = False
while not terminated:
    c = input("Type your next command:")
    ....
    ....
print( 'Goodbye.' )
```

Listing 1: Structure of the main while loop. The loop can terminate only when the users types the command **T**

The script should support the following commands:

- **A** - allows the user to add a product to the cart (see the example in Listing 2). Remember that identifiers are unique to each product;
- **R** - allows the user to remove an existing product from the shopping chart. This should let the user specify exactly what product to remove, without any ambiguity whatsoever. If no such product is present, no removal should take place;
- **S** - prints out a formatted text summary of the cart, with an easy-to-read list with product names, quantities, partial sums per product type and total sum (see example in Listing 3). The list should be ordered alphabetically by product name. Note that this command does not generate JSON, for that we have command E below;

- **Q** - the user can directly change the quantity of a product already present in the cart. This, like the R option, should let the user specify a product without ambiguity and, if no product is found, no changes should take place;
- **E** - generates a summary of the cart as a *JSON*-formatted data dump, printed to the console. This output is an array of *JSON* representations of each product. You are free to design it in any way you wish, using the data stored in the attributes of each type of product;
- **T** - the script terminates (exiting the while loop);
- **H** - a request for help from the user. The commands that the program recognises are printed out to the console (see Listing 4);

Any other command should print out the following message: *"Command not recognised. Please try again"*.

Remember that:

- users should be helped interact with your program. If an error occurs (like a product not found), users should be told what happened and what they should do next;
- product attributes should have their typing enforced; that is, you need to ensure that the user input for each attribute of each product is of the intended type.

```
>>> Insert the next command: A
>>> Adding a new product:
>>> Insert its type: Clothing
>>> Insert its name: Gloves
>>> Insert its price (£): 23.5
>>> Insert its quantity: 1
>>> Insert its brand: Moret and Spark
>>> Insert its EAN code: 1234567891234
>>> Insert its size: XL
>>> Insert its material: Silk
>>> The product Gloves has been added to the cart.
>>> The cart contains 4 products.
```

Listing 2: Adding a new product to the cart

```
>>> Insert the next command: S
>>> This is the total of the expenses:
>>>   1 - 6 * Eggs  = £2.4
>>>   2 - Gloves = £23.5
>>>   3 - Hat = £15
>>>   4 - 2 * Orange juice = £5
>>>   Total = £45.9
```

Listing 3: Summary of an ongoing shopping session

```
>>> Insert the next command: H
>>> The program supports the following commands:
>>>    [A] - Add a new product to the cart
>>>    [R] - Remove a product from the cart
>>>    [S] - Print a summary of the cart
>>>    [Q] - Change the quantity of a product
>>>    [E] - Export a JSON version of the cart
>>>    [T] - Terminate the program
>>>    [H] - List the supported commands
```

Listing 4: A user help request

2 Activity 2 (40 points)

In Week 4, we talked about a series of advanced data types such as trees. In particular, we mentioned **binary search trees** (or BST), which are trees with the following property: for any parent node, its data value must be higher than the data from any child node to its left, and lower than the data of any child node to its right. Creating and maintaining such a tree comes with a cost, as every time new nodes are added, removed or changed, the developer needs to ensure that this key property still holds. In return, the tree is fast to search through - finding a value in it takes `O(log n)`. For increasingly larger n this should be much faster than a regular linked list.

In this activity, you will implement a binary search tree and study its algorithmic complexity. You will also create a linked list and compare the complexities of both structures.

The activity is divided in three parts that add up to 40 points. You need to submit a zip file with three .py files, containing all the functionality described in the next sections.

2.1 The binary search tree class (4 points)

To begin with, create a Python file `binarysearchtree.py` and implement a **TreeNode** class. The class should include attributes and methods to change (set) and access (get) the following:

- A data value (or cargo) of the node;
- A link to its left child;
- A link to its right child;

Accessing and manipulating the attributes inside the nodes should be done only through these methods.

Now, create a class *BinarySearchTree*. It should have:

- The root of the tree as an attribute;
- An optional limit in size;
- A method `is_empty()` to check if the tree is empty;
- A method `is_full()` to check if the tree is full;

Like nodes, attributes on the tree should be accessed and altered only through getter and setter methods. Also, make sure to use the methods `is_full()` and `is_empty()`.

2.2 Operations on BSTs (8 points)

Expand your tree with operations. You should implement the following methods:

- A method `search()`, which receives a data value and searches the tree for it, returning a Boolean indicating whether or not the item is there.
- A method `insert()`, which receives a data value and inserts it as a new node in the tree. The tree may include duplicate values.
- A method `delete()`, which deletes a value from the tree, and mends the tree back together in case it breaks, keeping the tree's binary search property intact.

- A method `traverse()`, which returns the values stored across the nodes of the tree in ascending order.
- A method `print_tree()` to print a visual representation of the tree. Think about how you would prefer to visualise a tree if you could only use `print()` calls; it doesn't need to be the `__str__` method.

2.3 Random trees' simulation (8 points)

You should now have a functioning binary search tree. It is time to study the time complexity of its search. To achieve this, do the following:

1. Create a new script, called `complexity.py` and import your **BinarySearchTree** class;
2. In `complexity.py`, create a function `random_tree(n)`, which takes an input n and generates a tree of size n by populating it with n random integers from 1 to 1000;
3. Then create a list X of equally spaced numbers from 5 to 100 on step size 5 (so, 5, 10, 15 etc.)
4. For each value s in X, generate 1000 random trees of size s, and search them for the value 42, storing the average time spent by the `search` call into a list Y;

Notes:

- The `time` module might be useful here.
- In case your laptop does not have the capacity to run a simulation of this size, reduce either the size of the trees or the size of X or both. In this case, declare two constants in `complexity.py`, one called $TREE\_SIZE$ and the other one called $NUMBER\_OF\_TREES$ where you specify the numbers that work for you.

2.4 Complexity analysis for BSTs (8 points)

We can now see how the time spent to search in a BST varies according to its size n. In *complexity.py*, add more code to:

1. Plot X against Y. X stores the different sizes of trees. Y stores the times spent searching. Use the code in Listing 5 for that.
2. Add a comment in *complexity.py* that describes in your own words the complexity of BST search based on the shape of the graph you plotted. **Hint**: relationships could be e.g. linear, sub-linear, exponential, quadratic, logarithmic etc as explained in the lectures in Week 4. Start your comment with the words *Complexity analysis X vs Y"* so that we can find it easily in your code.
3. Create a new list *Y2* with estimates of average search time t under an ideal linear relationship to the size of the trees in X. **Hint**: In a linear relationship, $t = c * n + b$, where c and b are constants. Using the search time t for $n = 5$ and $n = 10$ from Y, you can find c and b and estimate the other values of t in $Y2$.
4. In the same way, create a third list $Y3$ with estimates of average search time under an ideal logarithmic relationship to the size of the trees in X. **Hint**: In a logarithmic relationship,

$t = c * log(n) + b$, where c and b are constants. Using the search time t for $n = 5$ and $n = 10$ from Y, you can find c and b and estimate the other values of t in $Y3$. Use $log(n) = log_2(n)$.

5. Plot the three curves (X against Y, $Y2$ and $Y3$) using the code found in Listing 6.
6. Add another comment in *complexity.py* that describes in your words how the initial graph compares to an ideal linear or logarithmic complexity. What could be the reason wny your Y line does not follow exactly the same line as e.g. $Y3$? Is there any issue with our class Binary Search Tree or with the way we created tree objects that we'd need to fix so that Y gets closer to an ideal complexity? Start your comment with the words *Complexity analysis X vs Y, Y2 and Y3"* so that we can find it easily in your code.

Note: The `time` module's `time()` function records time as seen by the processor. Try not to run any demanding processing on your laptop while executing the code - if you have other processes running in the background, the time recorded will be impacted by them, tampering with your results. This may mean your plots will get wobbly - if that happens, try to stop any other processes that might interfere and run the code a few times until the plotted lines become smoother.

```python
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.xlabel('Size of trees')
plt.ylabel('Search time')
plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0))
plt.show()
```

Listing 5: How to plot a simple X vs Y graph, where X and Y are number lists of the same length

```python
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.plot(X, Y2)
plt.plot(X, Y3)
plt.legend(['BST','Linear','Logarithmic'])
plt.xlabel('Size of trees')
plt.ylabel('Search time')
plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0))
plt.show()
```

Listing 6: How to plot a multi-line graph with line labels

2.5 The linked list class (4 points)

For a better understanding of the importance of complexity, let us now compare BSTs to standard linked lists.

First, create a Python file `linkedlist.py` and implement a **ListNode** class. It should have attributes and methods for:

- A data value (or cargo) stored in the node;
- A link to its next node;
- A way to print the data value in the node;

Now, create a class **LinkedList**. It should have:

- The first node as an attribute;
- An optional limit in size;
- A method `is_empty()` to verify if the list is empty;
- A method `is_full()` to verify if the list is full;
- A method `__str__` to print a visual representation of the list.

Just like the tree and its nodes, the list and its nodes should have their attributes manipulated and accessed only through getter and setter methods. Also, make sure to use the methods `is_full()` and `is_empty()`.

2.6 Operations on linked lists (4 points)

Expand your **LinkedList** with proper operations. You should implement the following methods:

- A method `search()`, which receives a data value and searches the list for it, returning a Boolean indicating whether or not the item is there.
- A method `insert()`, which receives a data value as a parameter and inserts it as a new node in the list. Duplicate values are allowed.
- A method `delete()`, which receives a data value as a parameter and deletes its first occurrence from the list.
- A method `traverse()`, which returns the values of the nodes in the linked order.

2.7 Comparing the two data types (4 points)

Repeat the steps from Section 2.3 using linked lists. Use the same list X to generate the lists and save the average search times in a new list $Y4$.

Then analyse the time complexity. To do so:

1. Plot X vs $Y4$ by executing the code found in Listing 7.
2. Add a new comment in your code to describe how linked lists compare to BSTs. Start your comment with the words *Complexity analysis X vs Y, Y2, Y3 and Y4”* so that we can find it easily in your code.

```python
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.plot(X, Y2)
plt.plot(X, Y3)
plt.plot(X, Y4)
plt.legend(['BST','Linear','Logarithmic','LL'])
plt.xlabel('Size of trees')
plt.ylabel('Search time')
plt.ticklabel_format(axis='both', style='sci', scilimits=(0,0))
plt.show()
```

Listing 7: Adding a new plot to the mix

3 Activity 3 (25 points)

This activity builds upon the recent concepts you learned about reading and writing files with Python.

For this, you will use a dataset that lists the number of medals awarded to countries at the Tokio 2021 Olympics games. This dataset is available on the following Kaggle page: `https://www.kaggle.com/arjunprasadsarkhel/2021-olympics-in-tokyo/version/7`.

The original file is available in XSLX format and we converted it for you into CSV: medals.csv.

Download the medals.csv file on your machine. Then create a Python script `medals.py` that:

- Stores each country's information as an individual object;
- Defines standardised functionalities for these objects;
- Creates an interface through which users can query the information in these objects.

Similar to Activity 1, the Python file you submit will have all classes included in the activity, plus the execution loop specified in the last section.

3.1 CountryMedals class and methods (5 points)

First, define a class named *CountryMedals* that represents a country's medal board and has the following attributes:

- *name*: the name of the country;
- *gold*: the number of gold medals awarded to the country;
- *silver*: the number of silver medals awarded to the country;
- *bronze*: the number of bronze medals awarded to the country.

The class should also contain the following methods:

- `to_json()`: returns a *JSON* representation of the object, with the object's attributes as key-value pairs, i.e. the keys *name*, *gold*, *silver*, and *bronze*, as well as an additional key *total* that carries the total number of medals received by the country;
- `get_medals(medal_type)`: takes the argument *medal_type*, a string carrying one of the values *"gold"*, *"silver"*, *"bronze"*, or *"total"*, and returns the corresponding number of medals. Note: this method must return *None* if *medal_type* is not one of the values listed above;
- `print_summary()`: prints a textual summary of the number of medals received by the country. E.g., for Poland, the summary should be:

  Poland received 14 *medals in total;* 4 *gold,* 5 *silver, and* 5 *bronze.*

- `compare(country_2)`: shows a comparison of the medals received by two countries. It requires a second CountryMedals object as the parameter *country_2*. E.g., this is how a comparison between the medals received by Italy and Germany should look like:

Medals comparison between 'Italy' and 'Germany':
- Both Italy and Germany received 10 *gold medal(s).*
- Italy received 10 *silver medal(s),* 1 *fewer than Germany, which received* 11*.*
- Italy received 20 *bronze medal(s),* 4 *more than Germany, which received* 16*.*
Overall, Italy received 40 *medal(s),* 3 *more than Germany, which received* 37 *medal(s).*

3.2 Loading the data (5 points)

Now, you need to transform each country's entry from *medals.csv* into its own CountryMedals object. Parse the *medals.csv* file into a dictionary named *countries*; its keys should be the countries' names, and its values the *CountryMedals* objects representing the corresponding countries. This dictionary will serve as a base for the rest of the exercise.

3.3 Defining useful functions (10 points)

Before working on the next section, you will need to implement some helper functions. The first three functions will manipulate the data in the *countries* dictionary:

- `get_sorted_list_of_country_names(countries)`: extracts from the dictionary *countries* a list of country names alphabetically sorted;
- `sort_countries_by_medal_type_ascending(countries, medal_type)`: returns a list of *CountryMedals* instances sorted in ascending order by the number of medals of the given *medal_type* argument.
- `sort_countries_by_medal_type_descending(countries, medal_type)`: similar to the previous function, but the instances are sorted in descending order.

You will also need to define some helper functions to deal with the user input:

- `read_positive_integer()`: prompts the user to input a positive integer (here, 0 is also allowed). If the value entered is not a positive integer, the function should prompt the user again;
- `read_country_name()`: prompts the user to input a country's name. If the given name is present in the keyset of the *countries* dictionary, the function returns the inputted name. Otherwise, the function prompts the user for a country's name again, showing a list of options;
- `read_medal_type()`: prompts the user to input a medal type, i.e. "gold", "silver", "bronze", or "total". If the medal type inputted is not one of those, the function prompts the user for a value again.

3.4 Execution loop (5 points)

The last step of this exercise is to write a loop that will read commands from the user and execute operations accordingly. The available command options are:

- **Help** (or **H**/**h**): provides help to the user by printing the available commands and what they do. An example of an execution of this command is shown in Listing 8;

```
Insert a command (Type 'H' for help ): h

List of commands:
- (H)elp shows the list of comments;
- (L)ist shows the list of countries present in the dataset;
- (S)ummary prints out a summary of the medals won by a single country;
- (C)ompare allows for a comparison of the medals won by two countries;
- (M)ore, given a medal type, lists all the countries that received more medals than a threshold;
- (F)ewer, given a medal type, lists all the countries that received fewer medals than a threshold;
- (E)xport, save the medals table as '.json' file;
- (Q)uit.
```

Listing 8: Execution of the **help** command

- **List** (or **L/l**): shows the list of countries present in the dataset. The list has to be taken from the `get_sorted_list_of_countries_names` function. A possible execution of the *list* command is shown in Listing 9;

```
Insert a command (Type 'H' for help ): l

The dataset contains 93 countries: Argentina, Armenia, Australia, Austria, Azerbaijan, Bahamas, Bahra
```

Listing 9: Execution of the **list** command

- **Summary** (or **S/s**): prints out a summary of the medals won by a given country. Listing 10 shows a possible execution of the *summary* command for Norway;

```
Insert a command (Type 'H' for help ): s
>> Insert a country name ('q' for quit): Norway
Norway received 8 medals in total; 4 gold, 2 silver, and 2 bronze.
```

Listing 10: Execution of the **summary** command for Norway

- **Compare** (or **C/c**): performs a comparison of the medals won by two countries. This should use the helper functions you defined previously in order to ask the user for two countries to be compared. Listing 11 shows a possible execution of the *compare* command for *Australia* and *Hungary*;
- **More** (or **M/m**): lists all the countries that received more medals than a threshold, according to specific medal type. To get this medal type, as well as the threshold, use the helper functions you defined previously. Listing 12 shows a possible execution of this command;
- **Fewer** (or **F/f**): lists all the countries that received fewer medals than a threshold, according to specific medal type. To get this medal type, as well as the threshold, use the helper functions

```
Insert a command (Type 'H' for help ): c

Compare two countries
>> Insert a country name ('q' for quit): Australia

Insert the name of the country you want to compare against 'Australia'
>> Insert a country name ('q' for quit): Hungary

Medals comparison between 'Australia' and 'Hungary':
- Australia received 17 gold medal(s), 11 more than Hungary, which reveived 6 of them.
- Both Australia and Hungary received 17 silver medals.
- Australia received 22 bronze medal(s), 15 more than Hungary, which reveived 7 of them.

Overall, Australia received 46 medals, 26 more than Hungary which received 20 medals.
```

Listing 11: Example of execution on the **compare** command between Australia and Hungary

```
Insert a command (Type 'H' for help ): m
Given a medal type, lists all the countries that received more medals than a threshold;
Insert a medal type (chose among 'gold', 'silver', 'bronze', or 'total'): silver
Enter the threshold (a positive integer):20
Countries that received more than 20 'silver' medals:

 - United States of America received 41
 - People's Republic of China received 32
 - ROC received 28
 - Great Britain received 21
```

Listing 12: Example of execution of the **more** command

you defined previously. Listing 13 shows a possible execution of this command;

- **Export** (or **E**/**e**): saves the data contained in the *countries* dictionary into a *JSON* file. The file name needs to be given by the user when executing this command. Listing 14 shows a possible execution of the *export* command where user entered the file name ‘Exported’. Listing 15 shows an example of how the exported *JSON* needs to be formatted;
- **Quit** (or **Q**/**q**): terminates the interface loop and finishes the program’s execution.

```
Insert a command (Type 'H' for help ): f
Given a medal type, lists all the countries that received fewer medals than a threshold;
Insert a medal type (chose among 'gold', 'silver', 'bronze', or 'total'): total
Enter the threshold (a positive integer):3
Countries that received fewer than 3 'total' medals:

 - Bermuda received 1
 - Morocco received 1
 - Puerto Rico received 1
 - Bahrain received 1
 - Saudi Arabia received 1
 - Lithuania received 1
 - North Macedonia received 1
 - Namibia received 1
 - Turkmenistan received 1
 - Botswana received 1
 - Burkina Faso received 1
 - Côte d'Ivoire received 1
 - Ghana received 1
 - Grenada received 1
 - Kuwait received 1
 - Republic of Moldova received 1
 - Syrian Arab Republic received 1
 - Bahamas received 2
 - Kosovo received 2
 - Tunisia received 2
 - Estonia received 2
 - Fiji received 2
 - Latvia received 2
 - Thailand received 2
 - Jordan received 2
 - Malaysia received 2
 - Nigeria received 2
 - Finland received 2
```

Listing 13: Example of execution of the **fewer** command

```
Insert a command (Type 'H' for help ): e
Enter the file name (.json)Exported
File 'Exported' correctly saved.
```

Listing 14: Example of the execution of the **export** command

```json
{
    "Germany" : {
            "gold": 10,
            "silver": 11,
            "bronze": 16,
            "total": 37
    },
    ...
}
```

Listing 15: Example of a portion of the exported *JSON* file including all the medals