Assignment 2 (20 Marks)

Due: Friday 27 April 2018 at 6:00pm

Introduction

The second assignment is in the same Olympic sports domain as the first assignment. In the second assignment you will be processing the results data from a set of Olympic events. In this assignment you can assume that the data is valid, you just need to process it to determine the results for events.

Design

You will need to implement several classes for this program. There are two files for the assignment. One is **`entities.py`** that contains the class definitions for the data entities in the program. The classes that need to be defined in this file are:

- **Athlete** – data for one athlete competing at the games.
- **Country** – data for one country and it's delegation at the games.
- **Event** – data for one event at the games.
- **Result** – data that is one athlete's result in one event.
- **ManagedDictionary** – a wrapper class that provides an application specific interface to a dictionary of items. This is used to create collections that hold all of the athletes, countries and events in the application. These collection objects are defined globally and are to be used by your application and will be queried by the automated marking script.

The interfaces for these classes are provided in **`entities.py`** and you need to provide the implementations for these classes. Comments are provided to explain what the methods in these classes need to do. Do not change the provided interfaces, as they will be used by the automated marking script. You may add other attributes and methods to these classes as needed for your implementation.

There is a function **load_data** in **`entities.py`** that you need to implement to load the data from files. This function needs to load the data into the **all_athletes**, **all_events** and **all_countries ManagedDictionary** objects. The logic for loading data may be lengthy, so you may want to create functions that load a single file and which are called by **load_data**. As is indicated in the description of the data, implementation of this function can be delayed until after getting other basic functionality working.

The other file is **`processing.py`** that contains the class definitions for the logical processing of the application. The classes that need to be defined in this file are:

- **ProcessResults** – superclass that defines the logic processing interface.
- **AthleteResults** – provides details of one athlete's results for all of the events in which they competed.
- **EventResults** – provides details of the results of all athletes who competed in one event.
- **CountryResults** – provides a summary of the results of all athletes who competed for one country.
- **DeterminePlaces** – determines the place ranking of all athletes who competed in one event.

The **ProcessResults** superclass is provided in the **`processing.py`** file. The **ProcessResults** class has two abstract methods. The **process** method is to be overridden in the subclasses to perform the type of processing of results that are specific to that subclass. It also keeps track of how many times any results processing object executes the **process** method. The **get_results** method is to be overridden in the subclasses to return a list of the processed results generated by the **process**

method. The **get_results** methods in the subclasses is to raise a **ValueError** if it is called before the process method has been called.

You will need to define and implement the other four classes. They should all inherit from **ProcessResults** and override its abstract methods. Each of the subclasses that inherit from **ProcessResults** are to implement a static method called **usage_ratio**. This method is to return the ratio of how often the specific subclass was used to process results in comparison to all of the other subclasses of **ProcessResults**. An example is provided of doing this for **AthleteResults**.

AthleteResults is to obtain the results, for one athlete, in all of the events in which they competed. The required processing is to sort these results into best to worst order, based on the place the athlete obtained in each event. If more than one event has the same place, then these should be ordered by the event name, in ascending alphabetical order. The **__init__** method needs to take a single parameter which is the **Athlete** object on whom the processing is to occur. The **get_results** method is to return a list of **Result** objects that are ordered according to the logic implemented in the **process** method. A partial implementation of the **AthleteResults** class is provided to help clarify the details of what needs to be done in the subclasses.

EventResults is to obtain the results of all athletes who competed in the one event. The required processing is to sort these results into best to worst order, based on the places the athletes obtained. If there is a tie and multiple athletes have the same place, then these should be ordered by the athlete's full name, in ascending alphabetical order. The **__init__** method needs to take a single parameter which is the **Event** object on which the processing is to occur. The **get_results** method is to return a list of **Athlete** objects that are ordered according to the logic implemented in the **process** method.

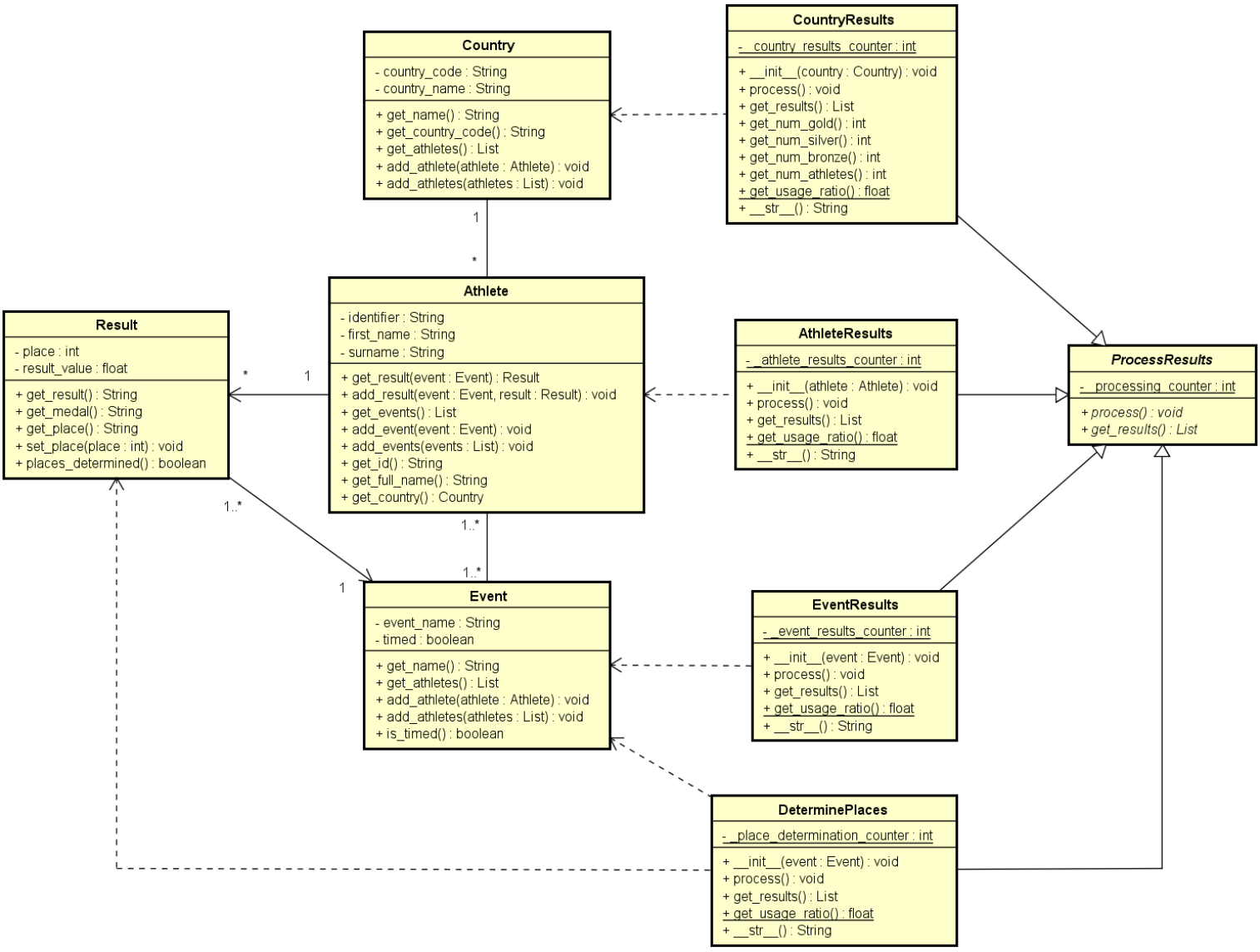
CountryResults is to obtain a summary of the results of one country's delegation. The required processing is to determine how many gold, silver and bronze medals were won by athletes who competed for the country. It should also be possible to find out how many athletes competed for the country. The **__init__** method needs to take a single parameter which is the **Country** object on which the processing is to occur. The **get_results** method is to return a list where the first list item is the number of gold medals won by the country, the second is the number of silver medals won, the third is the number of bronze medals won, and the fourth is the number of athletes who competed for the country. The **CountryResults** class also needs to have methods: **get_num_gold**, **get_num_silver**, **get_num_bronze** and **get_num_athletes** which return their specific values as ints.

DeterminePlaces is to process the results of all athletes who competed in the one event to determine their final places. The required processing is to sort the athlete results into best to worst order, based on the time or score obtained by the athlete. The **Event** class has an **is_timed** method that indicates whether the event results are determined by a time value (in seconds) or a score. Timed events are won by the lowest time. Scored events are won by the highest score. Once the results are ordered, the place obtained by each athlete needs to be set in the correct **Result** object, catering for potential ties in a place. If there is a tie, the place following the tie skips by the number of tied athletes. For example, if there are four athletes, and two tied for second place, there would be athletes who obtained first, second, second and fourth place. The **__init__** method needs to take a single parameter which is the **Event** object on which the processing is to occur. The **get_results** method is to return a list of **Athlete** objects that are ordered according to the logic implemented in the **process** method. If there is a tie and multiple athletes have the same place, then these should be ordered by the athlete's full name, in ascending alphabetical order.

You will need to create one or more functions that implement the main driving logic of your application. You will need to call these in the main block of code, where the demo functions are currently called.

Class Diagram

The following diagram provides an overview of the entity and logical processing classes in this design and their relationships. The diagram does not include the **ManagedDictionary** class.



Data

The data is stored in five comma-separated values (CSV) files. The athletes.csv file contains data about all the athletes. This file contains four columns:

Data	Data Type
Athlete Identifier	Integer
Athlete's First Name	String
Athlete's Surname	String
Country Code	String

The countries.csv file contains the name and code for each country.

Data	Data Type
Country Code	String
Country Name	String

The events.csv file contains names of the sporting events.

Data	Data Type
Event Name	String

The timed_event_results.csv file contains data about the results from the sporting events where the results are based on the time taken to complete the event.

Data	Data Type
Athlete Identifier	Integer
Event Name	String
Time Result	Floating Point

The scored_event_results.csv file contains data about the results from the sporting events where the results are based on the score obtained from the judging of the event.

Data	Data Type
Athlete Identifier	Integer
Event Name	String
Score	Floating Point

For the purposes of this assignment the results data files will not contain any results that are DNS, DNF or PEN. Similarly, you can assume that all athletes who compete in an event will obtain a result.

You will need to consider the dependencies between the objects in the design to correctly load the data from these five files. When creating an athlete object it needs to know the country for which the athlete is competing. This means you must load the country_codes.csv file before loading the athletes.csv file. But, countries need to know the athletes who belong to their delegation, so you need to add the athlete(s) to their country as you create them. An event also needs to know the athletes who compete in the event. The results files can only be loaded once you have loaded the athletes.csv and events.csv files.

Note: To get started on the assignment you can implement your entity and logical processing classes and test these with objects created in the program. (See the demo functions provided in **`processing.py`**, which provides an example of doing this.) You can get some the program logic working correctly without loading any data. Once part of your program is working you can then deal with loading the data.

User Interaction

You may implement a user interface that allows someone to query the program for results. You may find this convenient for visualising what is happening in your program. This user interface will not be assessed or contribute to your marks for the assignment. Your assignment will be marked based on correct functionality, as determined by the automated tests, and by the quality of your implementation.

Hints

The focus of this assignment is on demonstrating your ability to implement a simple object-oriented program using objects, classes and inheritance. It is possible to pass the assignment if you only implement some of the specified functionality. You should design and implement your program in stages, so that after the first stage you will always have a working previous partial implementation.

In thinking about the stages to implement the program, consider that some features are easier to implement than others. Focus on getting the easier features working first. You may find

implementing and testing the entity classes easier than the logical processing classes. The logical processing classes are based on the assumption that **DeterminePlaces** has already processed its result.

Dealing with athletes who tie for an event and athletes who have the same place result for different events is an additional complication to the processing. It is recommended that you get the basic processing working, assuming that these complications will not occur. Once the basic processing works you can then deal with the details of ties and athletes who have the same place in different events.

Submission

You must submit your assignment electronically through Blackboard. You need to submit both your `entities.py` and `processing.py` files (use these names – all lower case). You may submit your assignment multiple times before the deadline – only the **last** submission will be marked.

Late submission of the assignment will **not** be accepted. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension:
http://www.courses.uq.edu.au/student_section_loader.php?section=5&profileId=92705

Requests for extensions **must** be made no later than 48 hours prior to the submission deadline. The expectation is that with less than 48 hours before an assignment is due it should be substantially completed and submittable. Applications for extension, and any supporting documentation (e.g. medical certificate), **must** be submitted via my.UQ. You **must** retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so.

Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program,
6. apply techniques for testing and debugging

Criteria	Mark
Programming Constructs • Program is well structured and readable • Identifier names are meaningful and informative • Algorithmic logic is appropriate and uses appropriate constructs • Methods are well-designed, simple cohesive blocks of logic • Object usage demonstrates understanding of differences between classes and instances • Class implementation demonstrates understanding of encapsulation and inheritance	 1 1 1 1 1 1
Sub-Total	6

Functionality • Athlete implemented correctly • Event implemented correctly • Result implemented correctly • Country implemented correctly • AthleteResults implemented correctly • CountryResults implemented correctly • EventResults implemented correctly • DeterminePlaces implemented correctly	 1 0.5 0.5 0.5 1.5 1 2 3
Sub-Total	10
Documentation • All modules, classes, methods and functions have informative docstring comments • Comments are clear and concise, without excessive or extraneous text • Significant blocks of program logic are clearly explained by comments	 2 1 1
Sub-Total	4
Total	**/ 20**

Your mark will be limited to a maximum possible value if your program does not execute or crashes when executing. Your assignment will be limited to a maximum mark of:

- 4, if the program contains syntax or semantic errors that prevent it from executing;
- 5, if the program if none of the required functionality is implemented.
- 5, if the program crashes when executing using programmatically defined objects;
- 6, if the program crashes when executing using the provided data files;
- 7, if the program crashes when executing using other test data files;

It is your responsibility to ensure that you have adequately tested your program to ensure that it is working correctly. You are provided with sample data files. These do not cover all possible test cases. Your submitted assignment will be tested with different data files that will contain other combinations of data that your program should manage. You should create your own data files to test all features of your implementation.

In addition to providing a working solution to the assignment problem, the assessment will involve discussing your code submission with a tutor. This discussion will take place in week 9, in the practical session to which you have signed up. You **must** attend that session in order to obtain marks for the assignment.

In preparation for your discussion with a tutor you may wish to consider:

- any parts of the assignment that you found particularly difficult, and how you overcame them to arrive at a solution; or, if you did not overcome the difficulty, what you would like to ask the tutor about the problem;
- whether you considered any alternative ways of implementing a given function;
- where you have known errors in your code, their cause and possible solutions (if known).

It is also important that you can explain to the tutor the details of, and rationale for, your implementation.

Marks will be awarded based on a combination of the correctness of your code and on your understanding of the code that you have written. **A technically correct solution will not achieve a pass mark unless you can demonstrate that you understand its operation.**

A partial solution will be marked. If your partial solution causes problems in the Python interpreter please comment out the code causing the issue and we will mark the working code. Python 3.6.4 will be used to test your program. If your program works correctly with an earlier version of Python but does not work correctly with Python 3.6.4, you will lose **at least** all of the marks for the functionality criteria.

Please read the section in the course profile about plagiarism. Submitted assignments will be electronically checked for potential plagiarism.

Detailed Marking Criteria

Criteria	Mark		
Programming Constructs	**1**	**0.5**	**0**
• Program is well structured and readable	Code structure highlights logical blocks and is easy to understand. Code does not employ unnecessary global variables. Constants clarify code meaning.	Code structure corresponds to some logical intent and does not make the code too difficult to read. Code does not employ unnecessary global variables.	Code structure makes the code difficult to read.
• Identifier names are meaningful and informative	All identifier names are informative, clearly describing their purpose, and aiding code readability.	Most identifier names are informative, aiding code readability to some extent.	Some identifier names are not informative, detracting from code readability.
• Algorithmic logic is appropriate and uses appropriate constructs	Algorithm design is simple, appropriate, and has no logical errors. Control structures are good choices to implement the expected logic.	Algorithm design is not too complex or has minor logical errors. Most control structures are good choices to implement the expected logic, but a few may be a little convoluted.	Algorithm design is overly complex or has significant errors. Some control structures are used in a convoluted manner (e.g. unnecessary nesting, multiple looping, …).
• Methods are well-designed, simple cohesive blocks of logic	Methods have a single logical purpose. Parameters and return values are used well and do not break class encapsulation.	Most methods have a single logical purpose. Parameters and return values are appropriate and rarely break class encapsulation.	Some methods perform multiple functional tasks or are overly complex. Parameters and return values are not used appropriately or break class encapsulation.
• Object usage demonstrates understanding of differences between classes and instances	Methods, attributes and comments indicate each class is treated as a definition of a type and objects are used well in implementation. System behaviour is implemented in terms of objects sending messages to each other.	Methods and attributes are based on provided design and objects are mostly used appropriately in implementation. Most behaviour is implemented in terms of objects sending messages to each other.	Classes are treated as modules with functions, not as self-contained entities. Method implementations depend on external logic or variables.
• Class implementation demonstrates understanding of encapsulation and inheritance	Provided class interfaces have not been modified, aside from adding new attributes and methods that complement the design. Attributes are only used inside of their defining class. Subclasses of ProcessResults conform to provided definition and extend behaviour in a way that supports polymorphism.	Provided class interfaces have not been modified, aside from adding new attributes and methods. Attributes are only used inside of their defining class. Subclasses of ProcessResults conform to provided definition.	Provided class interfaces have been modified in ways detrimental to the design. Attributes are accessed directly from outside of their defining class. Subclasses of ProcessResults do not conform to provided definition.

Documentation	**2**	**1.5**	**1**	**0.5**	**0**
• All modules, classes, methods and functions have informative docstring comments	All docstrings are accurate, complete & unambiguous descriptions of how item is to be used. All parameters and return types, and expected values, are described clearly.	All docstrings are accurate and reasonably clear descriptions of how item is to be used. Almost all parameters and return types, and expected values, are described clearly.	Almost all docstrings are accurate and reasonably clear descriptions of how item is to be used. Almost all parameters and return types are described clearly.	Almost all docstrings are accurate descriptions of how item is to be used. Most parameters and return types are described clearly.	Some docstrings are inaccurate or unclear, or there are some items without docstrings. Some parameters and return types, or expected values, are not clear.
• Comments are clear and concise, without excessive or extraneous text			Comments provide useful information that clarifies the intent of the code, making it easier to understand. Comments do not repeat logic already clear in code.	Most comments provide useful information that clarifies the intent of the code. Comments rarely repeat logic already clear in code.	Some comments are irrelevant, not providing any detail beyond what is obvious in the code. Comment style, or excessive length, obscures some program logic.
• Significant blocks of program logic are clearly explained by comments			Important or complex blocks of logic (e.g. significant loops or conditionals) are clearly explained and summarised (i.e. stating a loop iterates over a list is not a useful explanation or summary).	Most important or complex blocks of logic are clearly explained and summarised; and/or, a few less important blocks of logic are described in too much detail.	Some important or complex blocks of logic are explained poorly, or are not summarised. Some unimportant blocks of logic are described in too much detail.

Functionality

Athlete implemented correctly

1	All methods in provided interface pass all tests.
0.75	At least 85% of tests from the full test suite pass, and all simple methods work correctly (get_id, get_full_name, get_country, get_events, add_event).
0.5	At least 70% of tests pass.
0.25	At least 40% of tests pass.
0	Less than 40% of tests pass.

Event implemented correctly

0.5	At least 80% of tests from the full test suite pass.
0.25	At least 65% of tests pass.
0	Less than 65% of tests pass.

Result implemented correctly

0.5	At least 80% of tests from the full test suite pass.
0.25	At least 65% of tests pass.
0	Less than 65% of tests pass.

Country implemented correctly

0.5	At least 80% of tests from the full test suite pass.
0.25	At least 60% of tests pass.
0	Less than 60% of tests pass.

AthleteResults implemented correctly

1.5	All tests from the full test suite pass.
1	All tests pass, aside from dealing with multiple events with the same place.
0.75	Processing logic fails tests but get_results correctly raises ValueError and get_usage_ratio passes tests, and AthleteResults correctly inherits and overrides methods from ProcessResults.
0.5	At least two of: get_results correctly raises ValueError, get_usage_ratio passes tests, or AthleteResults correctly inherits and overrides methods from ProcessResults.
0.25	At least one method passes its tests.
0	No tests pass.

CountryResults implemented correctly

1	All tests from the full test suite pass.
0.75	At least 75% of all tests pass, and all but one of the medal or athlete totals are correct.
0.5	At least 60% of all tests pass, and all but two of the medal or athlete totals are correct.
0.25	At least 35% of the tests pass.
0	Less than 35% of the tests pass.

EventResults implemented correctly

2	All tests from the full test suite pass.
1.5	All tests pass, aside from dealing with athletes tying for the same place.
1	Processing logic fails tests but get_results correctly raises ValueError and get_usage_ratio passes tests, and EventResults correctly inherits and overrides methods from ProcessResults.
0.5	At least two of: get_results correctly raises ValueError, get_usage_ratio passes tests, or EventResults correctly inherits and overrides methods from ProcessResults.
0.25	At least one method passes its tests.
0	No tests pass.

DeterminePlaces implemented correctly

3	All tests from the full test suite pass.
2.5	All tests pass, aside from dealing with multiple sets of athletes tying for the same place, for either timed or scored events.
2.25	All tests pass, aside from dealing with multiple sets of athletes tying for the same place, for both timed and scored events.
2	All tests pass, aside from dealing with athletes tying for the same place, for either timed or scored events.
1.75	All tests pass, aside from dealing with athletes tying for the same place, for both timed and scored events.
1	Processing logic fails tests but get_results correctly raises ValueError and get_usage_ratio passes tests, and EventResults correctly inherits and overrides methods from ProcessResults.
0.5	At least two of: get_results correctly raises ValueError, get_usage_ratio passes tests, or EventResults correctly inherits and overrides methods from ProcessResults.
0.25	At least one method passes its tests.
0	No tests pass.