# ***DCU School of Computing Assignment Submission***

|  |  |
| --- | --- |
| Student Name(s): | Conor Berns |
| Student Number(s): | 17456886 |
| Programme: | BSc in Computer Applications |
| Project Title: | CA341 - Imperative vs OOP |
| Module code: | CA341 |
| Lecturer: | Dr. David Sinclair |
| Project Due Date: | 25/11/19 |

|  |
| --- |
| Declaration  I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.  I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.  This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found at [http://www.library.dcu.ie/citing&refguide08.pdf](https://websvc.dcu.ie/"http://www.library.dcu.ie/citing&refguide08.pdf/") and/or recommended in the assignment guidelines.  I understand that I may be required to discuss with the module lecturer/s the contents of this submission.  I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.  Signed: Conor Berns |

***Introduction***

For this assignment I chose to implement both my program styles using Python 3. Python is a universal scripting language which supports programming Imperatively and Objectively **.** Python is also a very readable language, which leads to a much better analysis between the two styles, as seen in this report.

For both of my styles, I chose to make an interactive interface. The user runs the program and inputs a set of commands which are shown to them at the start of the execution.

The Imperative style uses no Objects while the Object Oriented Style makes heavy uses of both built in and defined objects.

Both styles have their pros and cons which I will discuss in the report.

Throughout the report I will refer to Object Oriented as OO.

***Imperative***

Imperative programming is a paradigm of programming that executes a sequence of steps in a guided manner to achieve a desired outcome. This leads to the creation, mutation and addressing of variables in a program which change its state. The Imperative paradigm relies heavily on the order in which the steps are executed. This is due to each step having an effect on the state of the program, thus the order of the steps is extremely important. Code and data are treated separately, and a code-operates-on-data relationship is prevalent.

Imperative programming is often used to write small programs and/or prototypes because of the reduced complexity of the syntax and reduced lines of code needed to reach the end result. This is a major advantage of the imperative paradigm over OO. However, Imperative programming is often not scalable. The approach of using a sequence of ordered steps to reach an end goal often means the program is very sensitive to change, meaning an imperative program is *generally* only used for the original purpose for which it was written. It is harder to alter an imperative program to add more functionality by definition.

Imperative: code and data are treated separately (independently), and a basic code-operates-on-data-relationship exists.

***Object Oriented***

OO programming is a paradigm of programming where everything is represented by an Object. An object can contain data (known as the object’s attributes) and methods (which access and modify the data associated with the object). OO programming allows data and the code that manipulates data to exist in the same space or “Object”. This leads to a much safer environment where modifying one Object’s data (if the objects are designed correctly) should not have a knock-on effect throughout the program.

OO programming allows many Objects to be defined which represent important aspects of a program. These objects are then bundled together into a large module which can interact with each object independently (or sometimes dependently, which is known as coupling) to achieve a desired end result.

OO programming is often used for bigger programs, as it allows the author to break a big problem into smaller problems, which intuitively can be represented by the use of objects. OO programming introduces a great deal of re-usability and scalability over plain Imperative programming. If a new change is needed in the program, a new object or method can be introduced to achieve this. Due to OO programming not relying as heavily on guided steps, the removing/insertion of new/old Objects should allow a program to gain extended functionality/reach a different end result. This isn’t as easily possible in the Imperative style due to a heavy dependency of executing guided steps to modify/create data.

Code and data are bundled together in OOP.

***Brief Description of the approach for both styles***

Imperative:

* Global variable of type string exists to represent the Queue.
* Input is read in from stdin and parsed
* Parsed input causes different steps of the program to be run in a non-changing order
* Commands exists to view and modify the Queue

OO:

* The queue is represented as a Queue object
* The Task and Event are represented by objects which are **children** of a To-Do Item object.
* A parser object exists to solicit input from the user and modify the Queue (which is an attribute of the Parser object) based on commands (which call methods of the Queue object) existing in the users input

***Comparison:***

I will break the comparison down into various categories.

1. Representing the Queue

2. Representing the Tasks and Events

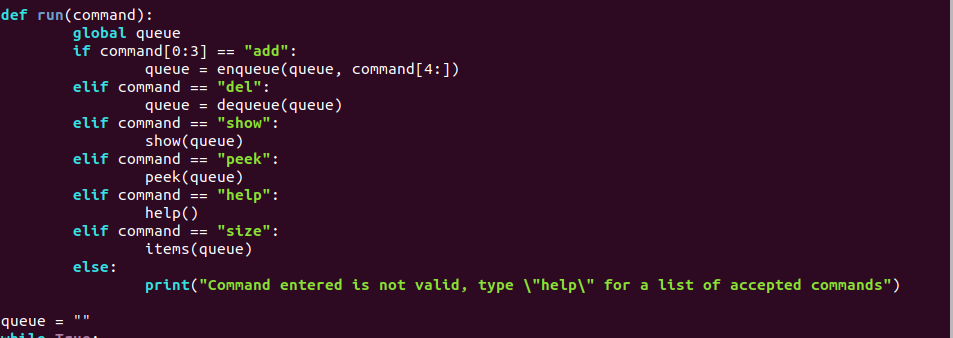
3. Adding and Deleting from the Queue

4. Accessing the Queue

***Representing the Queue:***

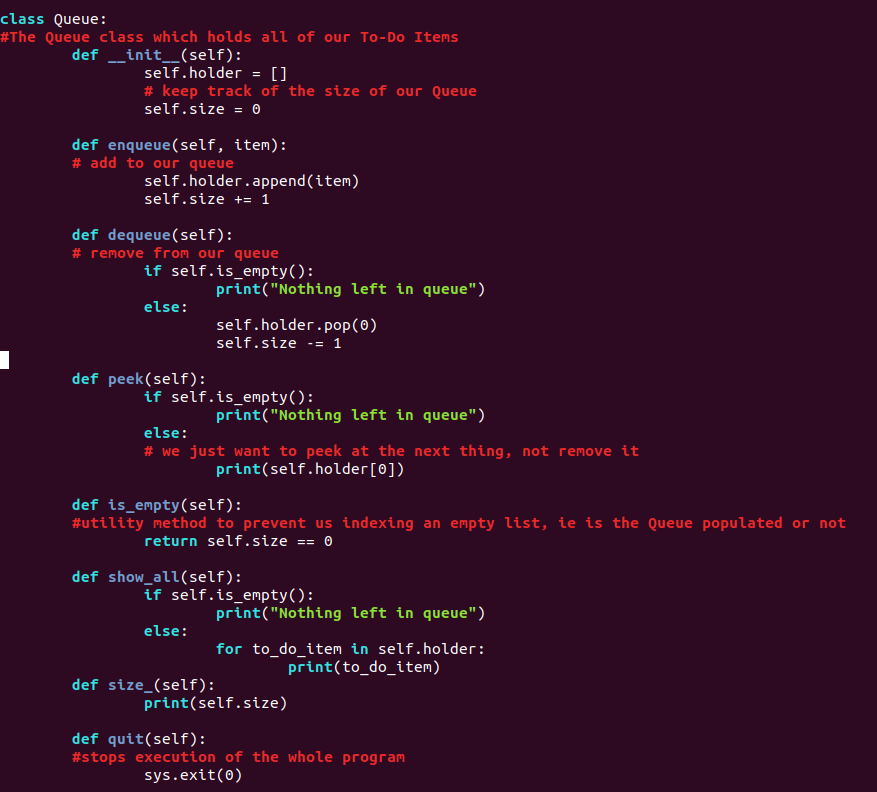
**Imperative:**

The Queue is represented as a globally accessible String Variable. This variable is passed as a parameter to any function which modifies/accesses the Queue. In the below example we can see the creation of the Global Queue variable.

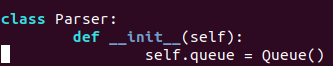
******

**OO:**

The Queue is represented as a Queue object which has two attributes (a list to hold the data, an Integer data type to keep track of the size of the Queue) and various methods. This Queue object is encapsulated in the Parser class and is not globally available. This ensures the only way to modify the Queue is to use the Parser Object to call the relevant Queue methods.



Encapsulation of the Queue



**Comparison:**

In the Imperative approach the Queue being global follows the Imperative style of accessing/modifying/creating variables which causes a change in state of the program. It also highlights the lack of data-security in the Imperative Paradigm. A global variable is easily accessible by ALL parts of a program which can mutate the Queue (either intentionally or accidentally). Here we see the clear need for guided steps as, if a wrong step were to be executed, the Queue and its contents could be completely lost.

In the OO approach the Queue being encapsulated in the Parser Object ensures that the Queue is manipulated safely. These direct method calls (In the Parser) can only modify the Queue and its contents by calling methods of the Queue object. This provides very secure data storage and manipulation.

The OO representation of the Queue ensures a more secure storage space for the data where it isn’t globally available/modifiable, whereas the Imperative representation provides a carefree fire and forget data storage facility.

***Representing the Tasks and Events:***

**Imperative:**

In the imperative implementation this is one of the downfalls of the approach. In the conquest of not using any objects, the above approach has no pre-defined representation of a Task or an Event. The program simply takes the input from the user as a string and inserts it into the Queue. There are no checks present to ensure the input is validated, and thus we can see another example of the lack of extensibility and security using an imperative approach. It is also worth noting that, while the data is not checked, it is considerably easier and quicker (in terms of lines of code and syntax) to write it.

The imperative approach simply represents either object (without checking for correctness) as a String which is added to the Queue.

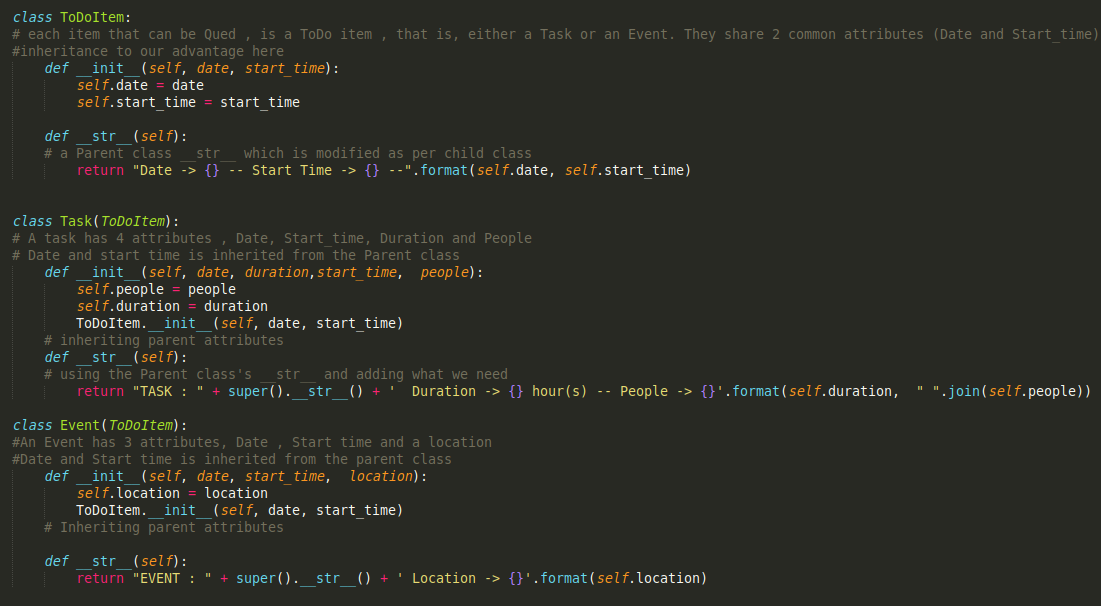
**OO:**

In the OO approach, the representing of Tasks and Events is handled by the following three Objects (using inheritance)

* To-Do Item ← Parent
* Task ← Child of To-Do Item
* Event ← Child of To-Do Item

The Task has 3 methods (Date, Start Time, Duration and a list of people assigned). The Event has 3 methods (Date, Start Time and location).

The Task and Event follow a **is-a** relationship of a To-Do Item object (Inheritance). Both objects have 2 common attributes (Date and Start time). This allows us to use Inheritance here to reduce code complexity and repetition by inheriting the parent class’s attributes. This inheritance is also extended to the \_\_str\_\_() method which is merely modified slightly for each child.

****

**Comparison:**

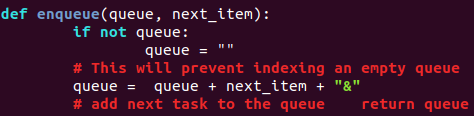
The imperative approach has no pre-defined way of representing the Task or Event. The OO approach does. This leads to OO having more secure addition to the Queue than Imperative, while the imperative approach is quicker/faster and less complicated (syntax and code wise).

***Adding/Deleting from the Queue:***

**Imperative:**

**Adding:**

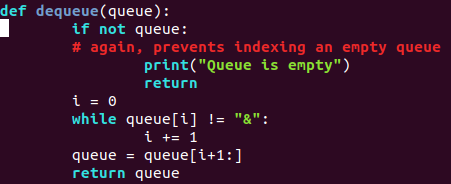
The program firstly takes a String (the next item to be added). It then concatenates the next item onto the end of the Queue String (modifying variables). When doing this, a placeholder symbol (&) is added onto the very end of the input string to signify a break between tasks/events. This follows imperative convention and creates/modifies variables (seen below)



**Deleting:**

The program firstly checks that the Queue isn't already empty. If it is, a message is displayed to the user to notify them.

If the Queue has data in it, the de-queue function starts at the beginning of the Queue string and keeps iterating until it sees the first placeholder symbol, while keeping a count of the letters seen in the form of an Integer data type. It then modifies the Queue global variable to display from this count onwards (i.e. removing the first item). However, if a placeholder symbol is inserted mistakenly by the user, this will cause the dequeue method to dequeue from where-ever the place holder is found.



**OO:**

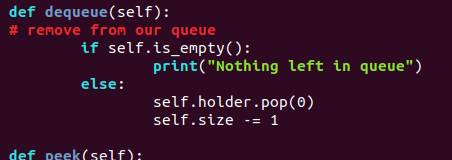
**Adding:**

The program firstly takes a String (the next item to be added) into the Parsers parse\_input method. This string is then passed through the following methods:

* choose\_item → Verifies whether it is a Task or Event that needs to be added to the Queue
* check\_date and check\_time → Verifies the time and date attributes are of the correct format
* make\_task → Instantiate a Task to be added to the Queue
* make\_event → Instantiate an Event to be added to the Queue
* enqueue → Add the instantiated child object (Task or Event) to the Queue

**Deleting:**

Deleting is extremely straightforward in the OO approach. A simple call to the Queue method via the Parser objects dictionary of commands causes the Queues dequeue() method to be called. This removes the first item that was inserted into the Queue (FIFO)



**Comparison:**

Adding in the Imperative approach is much more simplistic and requires little to no computation at all. Simply concatenate two strings with a placeholder character. This is extremely quick and very readable.

Adding in the OO way however is much more complex but the overhead has its advantages. When we add the finished instantiated Object (Task or Event) to the Queue, we know it is of the correct format and has all relevant information stored correctly.

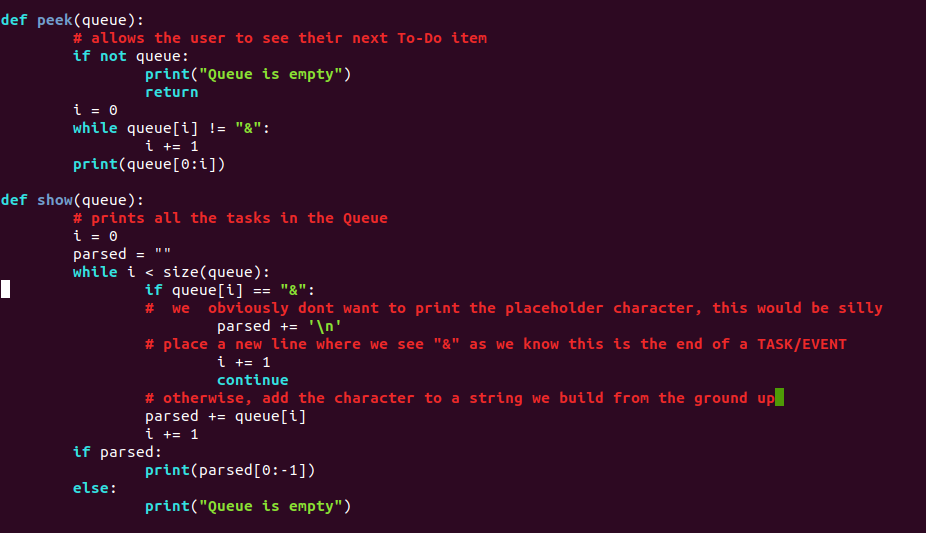
Deleting in the imperative way is much more complex than deleting in the OO. Good OO design allowed us to ensure, regardless of the objects contents/attributes, that de-queue() would perform the intended task each time. This can’t be said for the imperative approach: as outlined above, a placeholder inserted by the user would cause the queue to become corrupt.

***Viewing The Queue:***

**Imperative:**

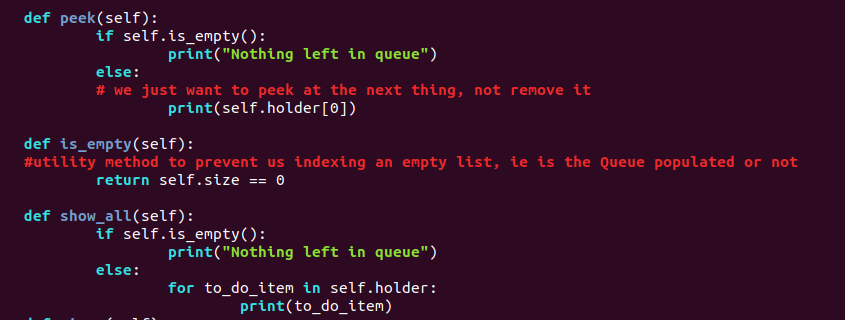
Viewing the Queue is simple as it is parsing a string. If the peek() or show() functions are called, the program simply parses the Queue until a placeholder object is met. While the placeholder object *hasn’t* been seen, an intermediate variable is created with all of the seen characters of the string. This variable is then subsequently printed. Here we see again the Imperative style of changing variables and state.

* Peek() → Show next item
* Show() → Show all items in the queue



**OO:**

Viewing is also very simple in the OO approach. Each To-Do Item has its own \_\_str\_\_() method to allow us to see a printable version of the object. Our Task and Event both inherit the base \_\_str\_\_() from the parent class and slightly modify it (with the non-inherited data) when it is invoked. Thus peek() indexes the Queues array at 0 and prints the To-Do Item there, while show() prints the whole list of ToDoItems.



**Comparison:**

The imperative style is as simple as the OO style however the Imperative is more susceptible to incorrect viewing due to placeholder characters being present in the string.

The use of objects here shows the power of bundling code AND data together and not just using a code acts on data approach (as the imperative style uses).

***Conclusion:***

From the comparison of the two paradigms and use of them I can safely say they both have their pros and cons for this task.

While the Imperative runs on very simple principles, is quick to write and runs quite quickly – It is evident that this task is much more suited to an OO style. The OO style was easier to write (in terms of error checking etc) and works much better.

In general however, other tasks may NOT be suited to an OO approach and would suit an Imperative design much better.