**Comparison Document**

**Introduction:**

The three languages chosen for my given Slot Machine task is C++, Python and Go. The paradigms chosen for each language are as follows; C++: Object Oriented Programming, Python: Procedural, and Go: Functional. This wide range of choices enables a thorough investigation of the languages' capabilities and illustrates how several paradigms influence the execution of a shared job.

**C++ vs. Python:**

**Similarities**

In Python and C++, numerous similarities were evident despite the paradigm differences. The functional aspects of methods, even though associated with objects in C++, exhibited a consistent core execution, making it easily translatable between the two languages. There were resemblances in the game flow and execution, as both languages employed a loop structure, allowing users to continue playing if they had sufficient funds or opted to exit. Additionally, in both languages, the symbol count and value systems utilised some form of map variation to correlate counts with respective values.

**Differences**

One of the primary distinctions between Python and C++ lies in their approach to structuring code. Python predominantly employs functions, whereas C++ embraces methods tied to classes and objects. In C++, to utilize a method, an object of the corresponding class must be instantiated. Conversely, Python, following a procedural approach, lacks the concept of classes, enabling functions to be invoked independently at any point.

In the C++ code, an abstract class is employed, delineating fundamental methods and properties essential for a slot machine (Normal/JackPot Machine). This choice exemplifies both abstraction and inheritance, as it outlines a blueprint for common functionalities. In contrast, the Python code opts not to implement any form of inheritance. Each function is globally accessible, eliminating the need for explicit abstract classes; however, this also means modifying function names without formal declarations. The distinction in paradigm highlights the contrasting methodologies each language employs in achieving code organisation and reusability.

**C++ vs. GO:**

**Similarities:**

In C++ and Go they either have a Struct or a Class despite them meaning the same thing outlie a structure of data/principles that aggregates to a single unit, such as player or Slot Machine in this case.

Both structs in Go and classes in C++ embrace a level of abstraction, employing distinct mechanisms – Go opts for interfaces, while C++ employs abstract classes. In both paradigms, derived classes/structs implement specific methods/functions, fostering a more abstract and generalised code structure.

Furthermore, both implementations demonstrate encapsulation through methods and functions, effectively concealing internal details like the balance. This practice safeguards the integrity of the code, mitigating unintended interference.

**Differences**

In the provided code, there is a notable feature of the functional paradigm in Go – the use of function closures, allowing a function to be returned within another function. This enables a level of abstraction and flexibility not directly achievable in C++. While C++ lacks native support for function closures, it compensates by allowing functions to be split into multiple functions and used in correlation with each other.

Another distinction arises in the treatment of structures. In the Go code, after defining a struct, you can reference an object of this type within the parameters of a method, making it specific to the structure type. Conversely, in C++, these references would need to be declared within the class itself, potentially limiting the flexibility and reusability of code.

The Go code further demonstrates features of immutability. For instance, in the method 'deductBalance,' a new instance of the 'Player' struct is created when updating the balance, preserving the original instance unchanged. This aligns with the functional programming paradigm's emphasis on immutability. In contrast, the C++ code opts for mutability, where the player instance remains the same, with the method modifying the existing object.

**Python vs. GO:**

**Similarities**

In both Python and Go, player balance is managed through dedicated functions, and the game logic is encapsulated in a similar loop. While Python places the game loop directly in the main, Go encapsulates it within a separate method. Both implementations prompt users to spin or exit, directing them accordingly. Additionally, both codes feature separate functions for a NormalSpin and JackPot, accommodating variations in Symbol\_Count, Symbol\_Value, and the size of rows and columns. This modular approach enhances code readability and allows for easy adaptation to different game configurations.

**Differences**

The primary distinction in the code revolves around instance association. In Python, methods can be employed independently without requiring object association, emphasising the procedural programming approach. Conversely, Go showcases functional features, particularly first-class functions, intricately linked with a Struct. Unlike Python, the Go code associates these functions specifically with either JackPot or Normal. This distinction results in method repetition in Go, necessitated by class associations, whereas Python allows a more streamlined utilisation of methods, reducing redundancy across different classes. This comparison highlights differences in procedural and functional programming paradigms.

In the Go code it focuses on Immutability when creating functions dealing with Player associated variables. Whereas, if we look at the Python code it relies on these are declared as global variables meaning as we proceed though the code these variables will be and can be modified.

**Conclusion:**

In conclusion, several key findings emerged during the task. C++'s OOP approach, while providing clear hierarchies and class differentiation, the complicated boilerplate code could complicate function understanding. Python's paradigm choice facilitated an easy and concise implementation, questioning the necessity of extensive OOP structures in smaller tasks. Go's emphasis on immutability, although a good programming style, seemed excessive for this task with limited associations. Ultimately, Python's procedural approach proved optimal, given the task. Understanding paradigmatic differences is crucial, enabling efficient task execution and the selection of the most suitable language, ultimately streamlining development processes.