**Data-Oriented Design**

"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming." - Rob Pike

**Design Philosophy:**

* If you don't understand the data, you don't understand the problem.
* All problems are unique and specific to the data you are working with.
* Data transformations are at the heart of solving problems. Each function, method and work-flow must focus on implementing the specific data transformations required to solve the problems.
* If your data is changing, your problems are changing. When your problems are changing, the data transformations needs to change with it.
* Uncertainty about the data is not a license to guess but a directive to STOP and learn more.
* Solving problems you don't have, creates more problems you now do.
* If performance matters, you must have mechanical sympathy for how the hardware and operating system work.
* Minimize, simplify and REDUCE the amount of code required to solve each problem. Do less work by not wasting effort.
* Code that can be reasoned about and does not hide execution costs can be better understood, debugged and performance tuned.
* Coupling data together and writing code that produces predictable access patterns to the data will be the most performant.
* Changing data layouts can yield more significant performance improvements than changing just the algorithms.
* Efficiency is obtained through algorithms but performance is obtained through data structures and layouts.

**Arrays**

Arrays are a special data structure in Go that allow us to allocate contiguous blocks of fixed size memory. Arrays have some special features in Go related to how they are declared and viewed as types.

**Notes**

* If you don't understand the data, you don't understand the problem.
* If you don't understand the cost of solving the problem, you can't reason about the problem.
* If you don't understand the hardware, you can't reason about the cost of solving the problem.
* Arrays are fixed length data structures that can't change.
* Arrays of different sizes are considered to be of different types.
* Memory is allocated as a contiguous block.
* Go gives you control over spacial locality.

**Slices**

Slices are an incredibly important data structure in Go. They form the basis for how we manage and manipulate data in a flexible, performant and dynamic way. It is incredibly important for all Go programmers to learn how to uses slices.

**Notes**

* Slices are like dynamic arrays with special and built-in functionality.
* There is a difference between a slices length and capacity and they each service a purpose.
* Slices allow for multiple "views" of the same underlying array.
* Slices can grow through the use of the built-in function append.

[Understanding Slices in Go Programming](https://www.ardanlabs.com/blog/2013/08/understanding-slices-in-go-programming.html)  
[Collections Of Unknown Length in Go](https://www.ardanlabs.com/blog/2013/08/collections-of-unknown-length-in-go.html)  
[Iterating Over Slices In Go](https://www.ardanlabs.com/blog/2013/09/iterating-over-slices-in-go.html)  
[Slices of Slices of Slices in Go](https://www.ardanlabs.com/blog/2013/09/slices-of-slices-of-slices-in-go.html)  
[Three-Index Slices in Go 1.2](https://www.ardanlabs.com/blog/2013/12/three-index-slices-in-go-12.html)  
[SliceTricks](https://github.com/golang/go/wiki/SliceTricks)

**Maps**

Maps provide a data structure that allow for the storage and management of key/value pair data.

**Notes**

* Maps provide a way to store and retrieve key/value pairs.
* Reading an absent key returns the zero value for the map's value type.
* Iterating over a map is always random.
* The map key must be a value that is comparable.
* Elements in a map are not addressable.
* Maps are a reference type.

[Macro View of Map Internals In Go](https://www.ardanlabs.com/blog/2013/12/macro-view-of-map-internals-in-go.html)