Introduction to Mojo Programming Language 🔥

Mojo represents a revolutionary leap forward in programming languages, seamlessly blending the elegance of Python's syntax with the power of systems programming and metaprogramming. This unique amalgamation empowers developers to craft high-performance, portable code that surpasses traditional languages like C, all while seamlessly integrating with the extensive Python ecosystem.

History and Background

Mojo emerged from the visionary mind of Chris Lattner, CEO, and co-founder of Modular AI. With a storied history of groundbreaking contributions to Swift, Clang, and LLVM, Lattner's influence permeates Mojo's DNA.

The Genesis of Mojo

Rooted in Modular AI's mission to democratize AI programming, Mojo addresses the dual challenges of performance and accessibility. By marrying the efficiency of C with the accessibility of Python, Mojo aims to usher in a new era of AI development, where innovation knows no bounds.

Performance Prowess

Mojo's performance is nothing short of astonishing, boasting a remarkable 35,000x speed improvement over Python. Leveraging advanced techniques such as Multi-Level Intermediate Representation (MLIR) and the LLVM toolchain, Mojo redefines the boundaries of what's possible, setting new standards for performance-driven programming.

Domain of Mojo

AI Development

In the realm of AI development, Mojo shines as a beacon of simplicity and power. By abstracting away the complexities of languages like C++, Mojo empowers developers to focus on what truly matters: innovation. With MLIR under the hood, Mojo enables developers to push the boundaries of AI without being bogged down by cumbersome syntax.

Microservices and Cloud Computing

Mojo's lightweight syntax and built-in concurrency support make it the ideal choice for microservices development in modern cloud architectures. Its seamless scalability and fault tolerance ensure that Mojo-powered applications can handle anything thrown their way, making it a cornerstone of resilient distributed systems.

Emerging Technologies

In the rapidly evolving landscape of emerging technologies, Mojo stands tall as a versatile and adaptable solution. From AI and machine learning to the Internet of Things (IoT), Mojo's performance, scalability, and interoperability make it the go-to choice for developers tackling the challenges of tomorrow.

Paradigms

Mojo's multi-paradigm approach empowers developers to tackle any problem with ease, offering the flexibility of imperative, functional, and object-oriented programming paradigms.

Imperative Programming

Mojo supports imperative programming, allowing developers to write code that specifies the sequence of operations to be performed. This paradigm is well-suited for tasks that require step-by-step execution. For example, the following code snippet demonstrates an imperative approach to calculating the sum of an array of numbers using the Kahan summation algorithm:

```
# Source: https://github.com/modularml/mojo/blob/main/examples/reduce.mojo

from tensor import Tensor

# Simple summation of the array elements
fn naive_reduce_sum[size: Int](array: Tensor[type]) -> Float32:
    var A = array
    var my_sum = array[0]
    var c: Float32 = 0.0
    for i in range(array.dim(0)):
        var y = array[i] - c
        var t = my_sum + y
        c = (t - my_sum) - y
        my_sum = t
    return my_sum
```

Functional Programming

Mojo's functional programming support enables developers to write concise, expressive code focused on immutability and pure functions. While Mojo's functional programming capabilities are still evolving and lack features like higher-order functions, currying, or monads, developers can still apply functional programming principles effectively. For example, Mojo provides `map` functions, allowing developers to apply a function over a range from 0 to a specified size.

```
from algorithm import map

fn double(x: Int) -> Int:
    return x * 2

fn main():
    var arr = InlinedFixedVector[Int, 9](9)
    var doubled = map(arr, double) # Pass the function `double` to `map`
as an argument
    print(doubled)
```

Object-Oriented Programming

Mojo's support for object-oriented programming elevates the language to new heights, empowering developers to conceptualize real-world entities as objects with attributes and methods. This paradigm fosters code organization, reusability, and modularity, facilitating the creation of scalable and maintainable software solutions.

Struct

A cornerstone of Mojo's object-oriented capabilities is the `struct`, a versatile data structure that encapsulates both fields and methods operating on an abstraction, such as a data type or an object. Fields represent variables holding data relevant to the `struct`, while methods, functions housed within a `struct`, typically act upon this field data. For instance, consider the following illustrative code snippet, showcasing an object-oriented approach to defining a Planet class with attributes and methods:

```
@value
struct Planet:
    var pos: SIMD[DType.float64, 4]
    var velocity: SIMD[DType.float64, 4]
    var mass: Float64

    fn __init__(
        inout self,
        pos: SIMD[DType.float64, 4],
        velocity: SIMD[DType.float64, 4],
        mass: Float64,
    ):
        self.pos = pos
        self.velocity = velocity
        self.mass = mass

# Source: https://github.com/modularml/mojo/blob/main/examples/nbody.mojo
```

Trait

In addition to structs, Mojo boasts robust support for inheritance through traits. A trait, akin to a contract, delineates a set of requirements that a type must fulfill. Comparable to Java interfaces, C++ concepts, Swift protocols, and Rust traits, Mojo's traits offer a flexible mechanism for defining shared behavior among types. Consider the following example of a trait in Mojo:

```
trait Quackable:
    fn quack(self):
        ...
```

The `Quackable` trait consist of method signatures, each denoted by three dots (...) to indicate their unimplemented status. Leveraging traits, developers can create reusable components that adhere to a defined set of behaviors. For instance, Mojo allows the creation of structs conforming to the Quackable trait:

```
@value
struct Duck(Quackable):
    fn quack(self):
        print("Quack")

@value
struct StealthCow(Quackable):
    fn quack(self):
        print("Moo!")
```

Moreover, Mojo facilitates trait inheritance, enabling the creation of hierarchical trait structures. Traits can inherit from other traits, thereby inheriting all requirements declared by their parent traits:

```
trait Animal:
    fn make_sound(self):
        ...

# Bird inherits from Animal
trait Bird(Animal):
    fn fly(self):
        ...

# Inherit from multiple traits
trait Duck(Bird, Quackable):
    ...
```

This hierarchical trait system empowers developers to create expressive and composable abstractions, fostering code that is both flexible and maintainable.

Explore Mojo's Standard Library

Mojo's standard library serves as a comprehensive toolkit, encompassing a wide array of modules and utilities aimed at enhancing development efficiency. From fundamental mathematical operations to intricate file I/O and networking tasks, the standard library furnishes developers with indispensable resources for crafting robust and high-performing applications. Here are some notable modules within Mojo's standard library:

- `tensor`: A module for tensor operations, enabling efficient manipulation of multi-dimensional arrays.
- `python`: A module for interacting with Python code and data structures, facilitating seamless integration with the Python ecosystem.
- `algorithm`: A module containing common algorithms and data structures, such as sorting and functional programming utilities.
- `collections`: A module for working with collections, including lists, sets, and dictionaries.
- `math`: A module providing mathematical functions such as polynomial evaluation and bit manipulation.

Currently, Mojo's standard library comprises approximately 20 modules, each meticulously designed to meet diverse programming needs. Expect ongoing expansion and refinement in future releases, as Mojo evolves to cater to emerging requirements and industry trends.

Value Life Cycle and Ownership

Life of a Value

Mojo has no built-in data types with special privileges. All data types in the standard library (such as `Bool`, `Int`, and `String`) are implemented as `structs`. You can actually write your own replacements for these types by using low-level primitives provided by MLIR dialects.

The life of a value in Mojo begins when a variable is initialized and continues up until the value is last used, at which point Mojo destroys it. Mojo destroys every value/object as soon as it's no longer used, using an “as soon as possible” (ASAP) destruction policy that runs after every sub-expression.

For each `struct`, Mojo equips it with copy constructors, move constructors, and destructors. The copy constructor is invoked when passing or returning a `struct` by value. Below illustrates a `HeapArray` struct showcasing copy and move constructors alongside a destructor:

```
struct HeapArray:
    var data: Pointer[Int]
    var size: Int

    fn __init__(inout self, size: Int, val: Int):
        self.size = size
        self.data = Pointer[Int].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, val)

    fn __copyinit__(inout self, existing: Self):
        # Deep-copy the existing value
        self.size = existing.size
        self.data = Pointer[Int].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, existing.data.load(i))

    fn __moveinit__(inout self, owned existing: Self):
        print("move")
        # Shallow copy the existing value
        self.size = existing.size
        self.data = existing.data
        # Then the lifetime of `existing` ends here, but
        # Mojo does NOT call its destructor

    fn __del__(owned self):
        self.data.free()

    fn dump(self):
        print("[", end="")
        for i in range(self.size):
```

```
            if i > 0:
                print(", ", end="")
            print(self.data.load(i), end="")
        print("]")

# Source: https://docs.modular.com/mojo/manual/lifecycle/life
```

Ownership

Mojo helps avoid side effects and memory leaks by enforcing a strict ownership model. The ownership model in Mojo utilizes `arguments conventions` to determine the ownership of a value. This specifies whether an argument is mutable or immutable ensure every value has only one owner at a time.

- `borrowed`: The function receives an **immutable reference**. This means the function can read the original value (it is *not* a copy), but it cannot mutate (modify) it. For example, the following code snippet demonstrates a function that prints a 3x3 Tic-Tac-Toe board:

```
fn print_board(borrowed board: InlinedFixedVector[Int, 9]):
    for i in range(3):
        for j in range(3):
            print(board[i * 3 + j], end=" ")
        print()
```

- `inout`: The function receives a **mutable reference**. This means the function can read and mutate the original value (it is *not* a copy). For example, the following code snippet demonstrates a function that fills a 9-element array with zeros (so it mutates the original array):

```
fn fill_with_0(inout board: InlinedFixedVector[Int, 9]):
    for i in range(9):
        board.append(0)
```

- `owned`: The function takes **ownership**. This means the function has exclusive mutable access to the argument—the function caller does not have access to this value (anymore). Often, this also implies that the caller should transfer ownership to this function, but that's not always what happens and this might instead be a copy (as you'll learn below). For example, the following code works by making a copy of the string, because—although `take_text()` uses the owned convention—the caller does not include the transfer operator:

```
fn take_text(owned text: String):
    text += "!"
    print(text)

fn my_function():
    var message: String = "Hello"
    take_text(message)
```

```
    print(message)

my_function()
```

Modules and Package

Mojo offers a sophisticated packaging system designed to streamline the organization and compilation of code libraries into easily importable files. This guide serves as an introduction to the fundamental concepts of structuring your code into modules and packages, akin to the approach used in Python. Furthermore, it provides detailed instructions on crafting a packaged binary using the mojo package command.

Modules

A Mojo module is a single Mojo source file that includes code suitable for use by other files that import it. For example, you can create a module to define a struct such as this one:

```
struct MyPair:
    var first: Int
    var second: Int

    fn __init__(inout self, first: Int, second: Int):
        self.first = first
        self.second = second

    fn dump(self):
        print(self.first, self.second)

# Source: https://docs.modular.com/mojo/manual/packages
```

Notice that this code has no `main()` function, so you can't execute `mymodule.mojo`. However, you can import this into another file with a `main()` function and use it there.

Package

A Mojo package is essentially a collection of Mojo modules within a directory structure, accompanied by an `__init__.mojo` file. This systematic arrangement facilitates the importation of modules either as a whole or individually. Moreover, there's the option to compile the package into a more portable `.mojopkg` or `.📦` file format, which ensures compatibility across various system architectures.

Importing Package and Modules

Importing a Mojo package and its constituent modules can be accomplished in two ways:

1. **From Source Files:**
    - When importing directly from source files, the directory name is utilized as the package name.
2. **From Compiled Package Files (.mojopkg/.📦):**

    - Importing from a compiled package involves using the filename as the package name, as specified during the compilation process using the mojo package command. This filename might differ from the directory name.

Regardless of the method chosen, Mojo treats both approaches equally, ensuring flexibility in the importation process.

References

- Modular AI
- Mojo Documentation
- Mojo GitHub Repository
- Mojo - A New Programming Language for AI
- LLVM
- MLIR

Appendix

I implemented a Tic-Tac-Toe game in Mojo. You can find the source code here. This is a simple example to demonstrate how to use basic syntax and data structures in Mojo. Feel free to check it out and play around with it!