Names:

Erin Green, Keiran Berry, Mary Moore, Oliver Schwab, & Robert Book

---

Language:

Mojo 🔥

Coding environment:

MacOS Visual Studio Code and Ubuntu 22.4 for WSL on Windows 11 Visual Studio Code
Mojo v0.6.0 is available to download for both Linux & MacOS here: developer.modular.com/download
Visual Studio Code is available for Windows, Linux, and MacOS here: code.visualstudio.com/download
The Mojo syntax highlighting & code completion extension for Visual Studio Code is available here: marketplace.visualstudio.com
Mojo is compiled with the Mojo CLI

Distinct pieces of functionality (that the team personally wrote for the project part):

Menu - Oliver

menu.🔥

History - Oliver

history.🔥

Output: Terminal & File - Mary

commandLine.🔥
fileOutput.🔥

AI Easy - Erin

basicAI.🔥

AI Medium - Robert

avgAI.🔥

AI Hard - Erin

minmaxAI.🔥

Data - Keiran	AI v.s. AI - Robert
array.🔥 board.🔥	AI_vs_AI.🔥

Section 1: Comparison to C++, Java, Python, and/or Scala

What is the language's philosophy?

Mojo aims to combine the usability of Python with the performance of C in order to gain greater programmability of AI. Its goal is to be a language with powerful compile-time metaprogramming, integration of adaptive compilation techniques, caching throughout the compilation flow, and other features that are not supported by existing languages. Mojo was designed to be a superset of Python, meaning it would be compatible with all existing Python programs and libraries, but that developers could use their Python code as a base and add Mojo to gain performance that would otherwise require C or C++. However, Mojo is still a first-class, standalone language that is not dependent on Python. Additionally, Mojo allows the programmer to decide when to use static or dynamic.

Compare and contrast your language in terms of the location it is used.

Since Mojo is still in development, it is not being used in production anywhere yet; however, it is designed to be used for artificial intelligence and machine learning. Python also has the ability to be used for these purposes through the use of many specialized libraries and is already the dominant force in these fields. As stated previously, since Mojo is a superset of Python & will be fully compatible with everything Python, it can be used to solve any sort of AI/ML problems Python solves. While Mojo can be used for many purposes, like Python, it is doubtful that it will be implemented for many other purposes (than AI/ML) because it is so specialized and there would be better choices.

Compare and contrast your language in terms of where it excels and where it fails

Excels

One of Mojo's strengths is that it is a very easy language to learn and start coding in. The syntax is simple and straightforward, and feels just like an extension to python should.

Fails

To be frank, Mojo has a lot of areas where it fails. The first place that it fails is because of how few resources are out there for Mojo. There are a total of three places where one can go to get help for this language. There's also the issue of how little support there is for the language. Mojo can only be run in one version of Ubuntu Linux, there's little IDE support (a singular extension on Visual Studio Code), and

there is no proper debugger, so the support is vastly inefficient for anything outside of pet projects. Finally, Mojo itself is very underdeveloped. There are no arrays, no classes, no inheritance, no proper typecasting, and so much more. For now, all of these issues are due to the fact that Mojo is so new. It is an unstable language that needs a few more years of development before it can be properly evaluated.

Compare and contrast your language in terms of portability, simplicity, orthogonality, AND reliability.

Portability

Mojo is a compiled language, but has the option to be either Ahead-of-Time (AOT) or Just-in-Time (JIT) compiled. Along with that, Mojo is designed to be a superset of Python, so in the future there is a high hope for it being highly portable. Unfortunately, Mojo is far from portable at the moment. It can only be run in a specific version of Ubuntu Linux, so it is very limited on where it can run.

Simplicity

Mojo is pretty simple to learn, as its syntax is based on Python. The changes from Python's syntax are straightforward and easy to remember. Mojo being a very new language also greatly contributes to this fact.

Orthogonality

Mojo is really orthogonal at this point in time. This is mainly due to two reasons. The first is because its syntax is heavily based on Python, which is an orthogonal language, and the second is because Mojo is so new. At its current point in development, Mojo only handles the basics, so really isn't anything that can't be orthogonal at this point. There are no arrays, lists, vectors, or other unique data types that could create unexpected syntax and decrease orthogonality.

Reliability

Mojo's reliability highly depends on how the programmer uses the language. Since static typing is an option (but not a requirement), we can declare and check data types at compile time, which increases reliability, but if the programmer opts to use dynamic typing, then the checking is up to them as well, and reliability decreases. At the moment, when things go wrong, Mojo simply crashes, decreasing its reliability.

Section 2: Syntax, OOP

Write an example of one type of assignment expression in the language.

```
var age: Int = 42
```

```
var name: String = "Harry Potter"
```

Then write the generic format of an expression. E.g., Pre-fix, post-fix, curly brackets, indentation requirements, etc. The goal here is the general appearance of a line or block of code.

There are indentation and line break requirements identical to Python, no curly braces are used. Assignments and expressions use Infix notation.

```
age = 1
fn increaseAge() -> None:
      age = age + 1
```

How does the language support extension, etc. (single inheritance, interfaces, root object, class OOP, prototype OOP, other OOP, file importing, file extension, plugins, piping, module linking, etc.)?

Currently there are no classes or interfaces in Mojo, only static structs which can act as pseudo-classes. Structs cannot be inherited, so currently there is no inheritance; but structs can be imported from one file to another. Additionally, unlike Python, all code in Mojo must be enclosed in a function or struct. Top-level, single statements only work in the REPL.

> **Give an example.**
>
> ```
> from [fileName] import [structName]
> ```

How does the language handle modules/namespace/packages/etc.?

Mojo allows programmers to create their own modules and packages. Modules are included as:

```
from moduleName import structName
import moduleName
import moduleName as myModule
```

Modules can only be imported if they are in the same directory as main. In order to import a file from a different directory, a package must be created.

Packages in Mojo work almost identically as they do in Python. Within the package, there must be an `__init__.mojo` file, without this it is not recognized as a package and the module cannot be imported. Packages are included as follows:

```
from myPackage.moduleName import myModule
```

Additionally, packages can either be within the same directory as main or the source code can exist elsewhere and a compiled version of the package can be in the directory instead, similar to Python. Namespaces are not a functionality of Mojo.

What is the scope operator(s)? Alternatively, explain how to pick which variable if two code courses contain the same name?

The only scope operator Mojo currently has is '.'.

Does the language allow function overloading (name repetition), function redefinition, and/or function overriding?

Mojo allows function overloading and for parameter overloading. Since there is currently no inheritance, there is no function overriding or redefinition.

Give example syntax if it does.

```
struct Complex:
    var re: Float32
    var im: Float32
    fn __init__(inout self, x: Float32):
    """Construct a complex number given a real
number."""
    self.re = x
    self.im = 0.0

    fn __init__(inout self, r: Float32, i: Float32):
    """Construct a complex number given its real and
imaginary components."""
    self.re = r
    self.im = i
```

Section 3: Binding, type system, and data type range

Is the language static or dynamically typed? Give example syntax in code.

Mojo allows for both static and dynamic typing, as per Mojo's design philosophy that both forms of typing are important and good for applications, so it should be up to the programmer on what to use. Thus, Mojo supports both types of typing, and it's up to you to read the documentation to know what is statically typed and what is not. For example, structs and fn are both always statically typed, while defs are dynamic. Examples of static typing and dynamic typing in mojo can be seen below:

Example Code for Static Typing

```
struct Triangle:
    var base: Float32
```

```
    var height: Float32

    fn __init__(inout self, b: Float32, h: Float32):
    """Construct a triangle."""
    self.base = b
    self.height = h

    fn area(self) -> Float32:
    """Calculate the area of a triangle."""
    return self.base * self.height / 2
```

Example Code for Dynamic Typing

```
def triangleArea(base, height):
    """Calculate the area of a triangle."""
    return base*height/2.0
```

Is the language static or dynamically scoped? Give an example in code.

Mojo is statically scoped. Example code can be seen below:

Example Code

```
fn greetings(n: Int32) -> Int32:
    """Prints out hello n times"""
    var sum = 2024

    for i in range(0,n):
        var sum = i
        print("Hello! ", sum) #sum = 0, 1, 2, ..n

    return sum #will return 2024
```

Describe the type system (equivalence, compatibility, etc.).

Mojo's type equivalence is currently non-existent. You cannot compare different types to see if they are equivalent. Mojo's type compatibility is pretty lacking as well. The only real datatype conversion available at the moment is converting to an integer.

Mojo does have type inference. When statically typing in Mojo, you can choose not to declare the type of a variable on initialization, and the Mojo compiler will assign a type for you. It also has type checking.

Mojo is a strongly typed language and has strict type checking. Finally, Mojo has type definitions for a lot of their data types, but not all of them. Datatypes that don't have specific definitions in Mojo default to

Python's definitions. This goes for generic data types such as Int and Float, while Mojo has also added defined datatypes such as Int16, and Float32, which specify the sizes.

What are the built-in data types and their ranges? (List 4-10, or send me a note if you believe that there are less than 4)

All of the data types listed below do not have specified ranges:

- Int
- Float
- String
- Pointer[T] (where T can be any type)
- AnyType

Some data types with specified ranges can be seen below:

- Int8 → 8 bit signed integer
- UInt64 → 62 bit unsigned integer
- Float32 → 32 bit floating point number
- Float64 → 64 bit floating point number

Note that there are more data types out there.

Section 4: Control flow, functions, specialties

What are the selection and repetition structures of the language, and what are their syntax?

Mojo has selection and repetition structures identical to Python:

```
for x in range(9, 0, -3):
      print(x)

if c != b:
      print(b)
elif c == b:
      print(c)
else:
      print("uh oh")

while y < 10:
      print("y: ", y)
      y -= 1
```

Are functions pass-by-value, pass-by reference, etc.? Give example syntax in code.

All Mojo **`def`** functions are pass-by-value (as compared to Python **`def`** which are pass-by-reference). By default, functions in Mojo that use **`fn`** are immutable references. Mojo has this by default because they want to avoid surprise changes to the data while also limiting making copies of data that would slow down the program. However, it is possible to make parameters pass-by-reference where the data is mutable by using the **`inout`** modifier. It is also possible to make a parameter **`owned`**, meaning it is both mutable within the function, but will not affect anything outside of the function: pass-by-value.

```
# default: pass-by-immutable reference
fn add(x: Int, y: Int) -> Int:
      return x + y

# inout: pass-by-reference
fn add_inout(inout x: Int, inout y: Int) -> Int:
      x += 1
      y += 1
      return x + y
var a = 1
var b = 2
c = add_inout(a, b)
print(a, b, c) # 2 3 5

# owned: pass-by-value
fn set_fire(owned text: String) -> String:
      text += "🔥"
      return text
fn mojo():
      let a: String = "mojo"
      let b = set_fire(a)
      print(a)
      print(b)
mojo()
''' prints:
mojo
mojo🔥
'''
```

Describe at least two of the language specialties.

Specialty one:

The language is geared towards writing high performance AI code. It directly interfaces with MLIR to perform high speed operations. MLIR is the modern standard for high performance machine learning accelerators. This means it gives high-performance support for a wide variety of accelerators.

Specialty two:

Mojo can directly interface with Python, its built-in packages, and 3rd party libraries. This is because it's built using Python's syntax with the goal of becoming a Python superset. This will enable Mojo to run normal Python programs natively.
