PART 1 / FREE PREVIEW

Mojo入門

Mojo とは何か

Python開発者のための、AIインフラ向けシステムプログラミング言語

Hiromi / Mojo 0.26.2 対応

github.com/h3adeu/mojo-from-scratch

AGENDA

本書 Part 1 の流れ

01 **Mojo とは何か・位置づけ**

用途・向き不向き・結論

02 **誕生の背景と MLIR**

Modular社・Chris Lattner・コンパイラ基盤

03 **設計思想 — 3つの柱**

ownership・compile-time・Python interop

04 **Python との比較・入口**

4つの違いと読み替えのコツ

05 **最小サンプルのアセンブリを読む**

print("Hello, Mojo") の裏側

INTRODUCTION

なぜ今、Mojo を学ぶのか

> 見た目は Python に親しみやすく、
> **中身はかなり本格的。**
>
> “Python の次にどこへ進むか” を
> 考えるための実践的な地図として。
>
> — 著者より

LLM 時代の学び方の変化

試し・調べ・検証する速度が上がった今、新しい言語の輪郭をつかみやすい。

Python だけでは届かない領域

AI の本番運用では、性能・メモリ・ハードウェア制御が同時に問われる。

Modular の挑戦

Python の書きやすさを保ったまま、システム言語の力を取り戻す試み。

AUDIENCE

対象読者

本書は次のような方を想定しています

01

Python 経験者

Python を使ったことがあり、次のステップを探しているプログラマー。

02

Mojo 興味あり

Mojo の名前は聞いたことがあるが、どこから始めればよいかわからない方。

03

LLM の中身

小さな GPT 実装を通して、LLMの仕組みを具体的に理解したい方。

注: サンプルコードは *Mojo 0.26.2* を対象。バージョンにより記法・*API* が変わる場合があります。

01

CHAPTER 01

Mojo とは何か・位置づけ

用途・向き不向き・結論を一気に整理する。

WHAT IS MOJO?

Mojo の一行定義

Mojo は、次の4点を同時に追求する言語として設計されている。

1

AIインフラのための高性能

システムプログラミング言語として位置づけられる

2

Python に近い書きやすさ

静的型・メモリ安全を重視しつつ、入口は親しみやすく

3

CPU・GPU・アクセラレータ

ハードウェアを意識した最適化を前提に書ける

4

Python ライブラリと相互運用

既存のPython資産をそのまま活かせる

***Mojo* は *Python* の方言ではなく、*Python* に似た入口を持つ高性能言語。**

USE CASES

向いている用途 / 慎重に見るべき用途

✓ 向いている用途

- 数値計算・テンソル処理・カーネル
- メモリ配置・GPU を意識する処理
- Python の性能ボトルネック部分の高速化

△ 慎重に見るべき用途

- 一般的な Web アプリの全置き換え
- 業務アプリの最初から最後までの実装
- 現時点では「Python の完全代替」は時期尚早

結論: Mojo は 「*Python*の完全な代わり」 ではなく **「Pythonを補う高性能な選択肢」** と捉える。

02

CHAPTER 02

誕生の背景と MLIR

なぜ今、新しい言語が必要なのか。

BACKGROUND

Modular・Chris Lattner と新言語の必要性

Modular Inc.

AIインフラを構築する企業

推論やデプロイを含む AI インフラ全体を扱う。
生産性と性能の両立を強く意識した会社。

Chris Lattner

LLVM・Clang・Swift の主要開発者

コンパイラと言語設計の経験が深い人物が、
Mojo の基盤づくりに関わっている。

AIインフラの本番環境で混ざりやすいもの

- Python
- C / C++
- CUDA
- 専用ランタイム
- 複数の中間表現（IR）

→ *Mojo* はこの混在を一つの流れに統合しようとする

MLIR

Mojo のコンパイラ基盤 — MLIR

MLIR

Multi-Level IR

いろいろな段階の中間表現を
まとめて扱うための仕組み。

抽象的な表現から低水準な表現まで、
段階を分けて変換していける `(lowering)`。

lowering の流れ

03

CHAPTER 03

設計思想 — 3つの柱

ownership・compile-time・Python interop。

THREE PILLARS

Mojo を支える 3 つの柱

01 Ownership と Value Semantics

一言定義

値の持ち主と寿命を明示する

嬉しいこと

GC なしでメモリ安全 / 不要なコピーを減らせる

02 Compile-time

一言定義

コンパイル時情報でコードを作り分ける

嬉しいこと

実行時の無駄を減らし、ハードウェアに特化できる

03 Python Interop

一言定義

Python の資産を Mojo から使える

嬉しいこと

既存コードを残しつつ、必要な部分だけ高速化

Part 2 では各柱を *read / mut / ^ / [] / from python import* など具体構文と一緒に確認する。

04

CHAPTER 04

Python との比較・入口

似ている入口、根本的に違う中身。

DIFFERENCES

Python と Mojo — 4 つの根本的な違い

観点	Python	Mojo
静的型・コンパイル時	実行時の動的型が基本。型ヒントは補助的。	型がはっきりし、[] の compile-time 情報がコード特化に関わる。
値の扱い (ownership)	オブジェクト参照として考えがち。	value semantics と ownership を重視する。
エラー (raises)	例外が中心。シグネチャにコストが見えない。	raises で「エラーを返す可能性」を明示する。
Python interop	外部連携として考えがち。	言語機能として組み込まれている。

見た目は *Python* に近いが、設計の中心はシステムプログラミング言語。

READING TIPS

Python に慣れた人の読み替えのコツ

1

「だれが値を持っているか」を意識

「名前がオブジェクトを指す」より、ownership を見る。

2

関数シグネチャをしっかり読む

read / mut / var / raises に関数の性質が出る。

3

[] と () を混同しない

[] は compile-time、() は runtime と区別する。

calculate_average.mojo

```
def calculate_average(temps: List[Float64])
raises -> Float64:
    if len(temps) == 0:
        raise Error("No temperature data")
    var total: Float64 = 0.0
    for i in range(len(temps)):
        total += temps[i]
    return total / Float64(len(temps))
```

raises がシグネチャに見える — Python との大きな違い

UNDER THE HOOD

アセンブリで見る print("Hello, Mojo")

単純な 1 行の裏で、Mojo は何をしているか ― 3 つのポイント

STACK

文字列はスタック上に組み立てられる

全てがヒープのオブジェクトになるのではなく、サイズ・ポインタ・フラグを並べた構造体としてスタックに組まれる。

ERROR

エラーは戻り値のビットフラグで返る

0x4000... のマスクで特定ビットを取り出し、エラーありなしを判定する。例外が機械語レベルでこう実装される一例。

ATOMIC

参照カウントは lock xaddq でアトミック

マルチスレッド下でも安全に refcount を操作。1行の print にもメモリ安全の仕組みが反映される。

```
lock xaddq %rax, (%rcx)   ; アトミックに refcount -= 1
```

KEY TAKEAWAYS

Part 1 のまとめ

- Mojo は Python に似た文法を持つ高性能言語
- 目的は書きやすさ + 性能・安全性・ハードウェア制御
- ownership・compile-time・Python interop が3本柱
- Python の完全置換ではなく、性能ボトルネックから段階導入
- 強いコンパイラ基盤 (MLIR) が書きやすさと性能を両立させる

NEXT

Part 2

言語仕様

Part 3

MicroGPT 実装

(有料販売)

Source code: github.com/h3adeu/mojo-from-scratch