PART 2 / 言語仕様

Mojo入門

言語仕様

Mojo Manual を体系的に — 関数・型・所有権・メタプログラミング・GPU まで

Hiromi / Mojo 0.26.2 対応

github.com/h3adeu/mojo-from-scratch

AGENDA

Part 2 の流れ — 6 セクション・10 章

01

言語基礎 (1)

概要・関数・変数・スコープ

02

型・演算子・制御

struct・SIMD・制御・エラー

03

struct・参照・package

ユーザー定義型と公開範囲

04

値・所有権・ライフ

value semantics と lifecycle

05

メタプログラミング

compile-time・trait・generics

06

GPU・Layout・interop

ポインタ・並列・Python連携

01

CHAPTER 01

言語基礎 (1)

概要・関数・変数 — Mojo の入口を整える。

FUNCTIONS

関数 — シグネチャが多くを語る

1. **def が基本**

   fn は非推奨。def で関数を定義する。

2. **() と [] は別物**

   () は実行時引数、[] はコンパイル時 parameter。

3. **オーバーロード可**

   同名でも引数型が違えば別関数として共存できる。

4. **raises がシグネチャに**

   エラーを投げる関数は raises を明示する。

functions.mojo

```
def show(a: Int):
    print("int", a)

def repeat[count: Int](msg: String):
    for _ in range(count): print(msg)

def pick_nonzero(x: Int) raises -> Int:
    if x == 0:
        raise Error("zero")
    return x
```

VARIABLES & SCOPE

変数 — var と暗黙宣言とスコープ

var ブロックスコープ

`var x = 1` は **宣言したブロック内** のみ有効。
if 内で var すると、外では使えない。

暗黙宣言は関数スコープ

`y = 10` のような **暗黙宣言** は
関数全体で見える。Python に近い感覚。

^ — 所有権の移動 (ownership transfer)

```
var msg = String("hi")
take(msg^)  # ここで msg は相手に渡し切られる
# このあと msg は使えない (compile error)
```

02

CHAPTER 02

型・演算子・制御・エラー

Mojo の足回りを揃える。

TYPES

型 — 名義的・SIMD・明示的変換

NOMINAL

struct は名前で区別

フィールドが同じでも別名なら別型。Point2D ≠ PointXY。

SIMD

要素数が型に含まれる

SIMD[DType.float64, 4] は固定長。+ や * は各要素に作用。

CAST

型は自分で揃える

Int と Float64 を混ぜるなら Float64(n) で明示的に変換。

よく使う組み込み型:

`Int` · `Float64` · `Bool` · `String` · `Tuple` · `List` · `Dict` · `Set` · `Optional`

演算子の落とし穴と制御フロー

には 2 つの意味がある

二項演算子: ビット XOR

```
5 ^ 3    # → 6 (XOR)
```

接尾辞: 値をムーブ

```
consume_list(data^)  # move
```

二項か接尾辞かで意味が変わる。コードを読むときは形に注目。

制御フローは Python に近い

`if / elif / else`	条件分岐
`for / while`	ループ
`for ref v in xs:`	要素の参照を借用、その場で更新
`for / while else`	break しなかったときだけ実行

ERROR HANDLING

エラー処理 — try / except / else / finally

`try / except`	失敗を捕まえる
`else`	成功したときだけ実行
`finally`	成功・失敗を問わず必ず実行
`raise e^`	受け取ったエラーを外へ投げ直す
型付きエラー	struct や Variant でエラー型を作れる

errors_try.mojo

```
def process(id: Int) raises:
    if id < 0:
        raise Error("bad id")

try:
    process(id)
except e:
    if id < 0: raise e^
else:  print("ok")
finally: print("done")
```

03

CHAPTER 03

struct・参照型・パッケージ

ユーザー定義型と公開範囲。

STRUCT

struct を軸に型を組み立てる

ユーザー定義型の中心

Mojo では struct が型の基本単位。

継承はない

trait と generics で振る舞いを共有する。

init と method

def __init__(out self, …) でフィールドを埋める。

可変メソッドは mut self

状態を変える操作は mut self を明示する。

```
struct Counter:
    var n: Int
    def __init__(out self, start: Int): self.n = start
    def bump(mut self): self.n += 1
```

REFERENCES & PACKAGES

参照型とパッケージ構成

REFERENCES

参照やポインタが必要になる場面

- 自己参照を含む型を組みたいとき
- 値の共有/借用を表したいとき
- メモリ配置を意識したいとき

→ ただの「生アドレス」ではなく、用途を型で分けて表す。

PACKAGES

ファイル構成と公開範囲

```
my_pkg/
├── __init__.mojo
├── core.mojo
├── utils.mojo
└── mojopkg.toml
```

__init__.mojo と *mojopkg.toml* が公開範囲・依存を整理する。

04

CHAPTER 04

値・所有権・ライフサイクル

持ち主・有効期間・一生をはっきり扱う。

OWNERSHIP

値セマンティクスと所有権 — 4 つの役割

`read`	`mut`	`var`	`ref`
共有して読む	**書き換えのため**	**所有して移動できる**	**借用を表す**
デフォルト。所有を奪わず借用するだけ。	破壊的更新を意図する引数に明示的に付ける。	受け取り側が所有権を持つ。^ で渡される。	for ref で要素の可変借用 → in-place 更新。

GC は無い。 値セマンティクスでは 「名前が指すオブジェクト」 より **「誰が唯一の所有者で、誰が借りているか」** で考える。

LIFETIMES & LIFECYCLE

ライフタイムとライフサイクル

LIFETIMES

参照の有効期間

- 参照は元の値より長くは生きられない
- 関数を抜けると借用は終わる
- ダングリング参照はコンパイラが拒否

LIFECYCLE

値の一生 (生成 → 使用 → 破棄)

- Copyable / @fieldwise_init で複製のしくみが決まる
- ムーブは別経路 (^) で扱う
- RAIIに近い: 必要な後始末を自動で安全に

05

CHAPTER 05

メタプログラミング

先に決められることは先に決める。

COMPILE-TIME

[] と () — コード生成に直接効く

[ ]

compile-time parameter

- コンパイル時に決まる値
- 型・値・別名を渡せる
- 使い方ごとにコードが特化される

( )

runtime argument

- 実行時に渡す普通の引数
- Python の引数と同じ感覚
- ふだんの値の受け渡しに使う

specialization.mojo

```
fn lanes[width: Int](scale: Int) -> Int:
    return width * scale

lanes[8](2)
  # 「幅 8 に特化した関数」
  # に実行時の 2 を渡す
```

comptime if / comptime for

選ばれなかった枝はバイナリに残らない。
ループはコンパイル時に展開される。

TRAITS / GENERICS / CONSTRAINTS

型の能力をコンパイラに伝える

TRAIT

振る舞いの契約

「この型はこの操作ができる」を表す。実装は struct 側に書く。

```
Copyable / Movable / Stringable / Equatable
```

GENERIC

型を後から差し替える

[T: Writable] のように trait 制約を付けて再利用する。

```
def byte_len[T: Writable](x: T) -> Int
```

CONSTRAINT

誤用を早期に弾く

where 句で更に細かく条件を絞る。コンパイル時に失敗できる。

```
struct Buf[size: Int where size > 0]
```

06

CHAPTER 06

ポインタ・GPU・レイアウト

安全な抽象から、必要なときだけ低水準へ。

POINTERS

ポインタも用途で型を分ける

型	用途	説明
`Pointer`	借用	値を読み書きする借用ポインタ。所有しない。
`OwnedPointer`	所有	領域の所有を持つ。スコープを抜けると解放。
`ArcPointer`	共有	参照カウントで共有する。アトミックに増減。
`UnsafePointer`	低水準操作	FFI / 低水準バッファ用。最後の手段。

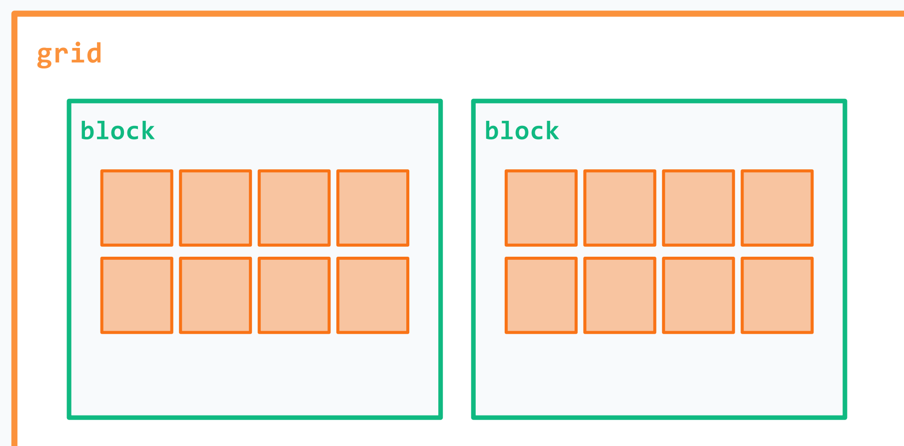
「便利だから *unsafe*」ではなく、安全な *API* で足りないときの最後の手段。

GPU & LAYOUT

GPU 階層と Layout

GPU 実行モデル

thread → block → grid



global_id = block_id × block_dim + thread_in_block

LAYOUT

shape × stride を型で表現

shape	形 (例: (3, 4))
stride	並び方の間隔
tile	部分レイアウトを切り出す

LayoutTensor = Layout + 実データの多次元ビュー

07

CHAPTER 07

Python 相互運用と導入判断

段階導入こそ、Mojo の現実的な使い方。

PYTHON INTEROP

Mojo ↔ Python の双方向呼び出し

Mojo → Python

```
from std.python import Python

def main() raises:
    np = Python.import_module("numpy")
    arr = np.array([1, 2, 3])
    print(arr)
```

→ *NumPy* 等の *Python* 資産をそのまま使える

Python → Mojo

拡張モジュールとして読み込む

- 1. Mojo で拡張モジュールをビルド
- 2. PYTHONPATH に配置する
- 3. Python 側から import mojo_module
- 4. Mojo の関数を Python の関数のように呼べる

→ ホットパスだけを *Mojo* で書き、*Python* から呼ぶ

CONCLUSION

導入判断と Part 2 まとめ

✓ Mojo 導入が向く場面

- Python で性能限界が見えている処理
- 数値計算・カーネル寄りのコード
- CPU / GPU / アクセラレータを見据える処理
- Python 資産を残したまま部分高速化したい

△ 慎重に見るべき場面

- 一般アプリ全体の全面置換
- すぐに強い安定性が必要な場合

NEXT

Part 3

MicroGPT 実装

最初は PoC や限定導入から。

hot path を 1 本だけ Mojo 化。

uv / pixi で環境を作る。

***Mojo* は *Python* の書きやすさを入口に、性能・安全性・ハードウェア制御へ段階的に進む言語。**