Hy ショートイントロブック

日本 **Hy** ユーザー会　著

2019-09-20 版　　発行

前書き

この本は、プログラミング言語 Hy のイントロブックです。「Python であり Lisp であるプログラミング言語」Hy の紹介と、その導入のため判断材料の提供を目的に執筆しました。

想定読者

ペルソナとして、次のような読者を想定しています。

- 他言語でプログラミングの経験が十分ある
- Lisp の知識や経験はほとんどない
- Python とその利用にある程度なじみがある、もしくは Python を利用した何かの開発を検討している

本書はプログラミングが初めての読者向けの本ではありません。そのため、解説は（非 Lisp の）プログラミング言語と Hy の違いにフォーカスしています。

「短い記述量で、他言語の経験者に Hy の概要をつかんでもらう」

が本書の目的です。これが達成されることを祈っています。

その他の注意

執筆量の物理的負担を減らすため、Hy 自体以外の事項については説明を省いています。

- virtualenv, pip など Python の周辺ツールに関する内容
- Docker 自体の利用法など
- エディタとその拡張の管理方法 etc...

これらは実践的・自己完結的な本にするには一定の解説をするべきですが、本書の読者はこれらを自己解決していただけることを期待しています。

Hy 自体に関しても、本書は網羅にはほど遠い内容の解説しかしていません。もし本書で Hy に興味を持っていただいたら、実際に Hy で書いて楽しみましょう！

検証環境について

本書は Arch Linux / Python 3.7 (virtualenv) / Hy 0.17 で検証・執筆しています。開発環境の章を除いて OS などの環境依存はない内容になっているはずですが、検証環境以外・特に Mac や Windows での動作を保証するものではありません。

目次

目次

目次

第 1 章

Hy とは / 概要

完全に Python であり、Lisp である

プログラミング言語 Hy は、Python ランタイム上で動く Lisp 方言です。Hy のコードが実行される時に、まず Hy -> Python のコードにコンパイルされ、その上で Python と同様に実行されます。

Hy は Lisp の方言ではありますが、あくまで Python の上に構築されたものです。Python 上の Lisp 風 DSL、というのが一番実態に近いかもしれません。

Hy は完全に Python です。具体的には、次のような特性を備えています。

- 基本的なデータ型が Python と共通している。tuple, dict, list など。Hy の built in 機能のレベルで generator にも対応しています
- Python のモジュールをそのまま import できる。定義ファイルなどを書く必要はありません
- Python コードの生成機能がある。単に生成機能があるだけでなく、基本的に予測可能なコンパイル後コードを出力
- 上記機能を使って、Hy で書いた資産を Python 側で利用可能
- 上記機能で、Python ランタイム縛りのある環境（Cloud Functions

等）でも Hy で書いて実行可能

Hy を使う理由

Python は広く普及している言語です。また、数値計算や機械学習などいくつかの分野ではデファクトスタンダードになっており、各種ライブラリの充実度合いから Python が選択されることもあります。

一方で、Python はプログラミング初学者にとっつきやすいとされている一方、使い倒そうとすると不満が出てくる言語でもあります。元々は古い言語であるため、、、、、、、、、、、

「充実した Python のライブラリ群を使いたい」「もっと書きやすい言語を使いたい」「Lisp を使いたい」これらをまとめて満たす言語として、Hy は高いポテンシャルを持っています。

うれしいこと・かなしいこと

うれしいこと

何を置いても、まず Lisp であることです。Lisp と S 式は好き嫌いが分かれると思いますが、好きの側の人には Lisp であることが選択の理由になるでしょう。

Lisp でありながら、Python の資産を完全に利用できます。ランタイムも含めて Python なので、インピーダンスミスマッチに悩まされる可能性はほぼありません。Common Lisp や Clojure でなく、Hy を選ぶ理由はここにあるでしょう。

もう少し消極的な理由として、AltPython として使えるという点があります。モダンな言語で経験を積んだプログラマーは、Python の至らない点にストレスを感じていることが少なくありません。太古の姿のまま高い完成度を誇る Lisp の一族として、Hy はそこに光をもたらしてくれる希望があり

ます。

かなしいこと

Python は HPC の文脈では「遅い」とされている言語です。Hy はあくまで Python ランタイム上で動くため、この問題は解決されません。計算量が多い問題に対しては、そもそも Python 以外の回答が必要になるでしょう。

Hy は Python にコンパイル→実行をするので、その分コンパイル時のオーバーヘッドは大きくなります。

Hy は極めてマイナーな言語です。そのためエディタのサポートも弱く、落ちている情報の量も多くはありません。ユーザー数がオーダー単位で多い JavaScript の世界から Python の世界に来ると文明レベルの差に愕然としますが、Hy のユーザーは Python よりもさらに数桁少ないです。ここには未開拓の大地が広がっていて、DIY を求められるシーンは少なくありません。

DIY を求められるのですが、そのような状況で Lisp 系言語であることは強い武器になるでしょう。チェーンソーを使って林野を切り開きましょう。

第 2 章

言語ひとめぐり

本章では Hy の雰囲気を見ていきます。「だいたいこういう感じになるんだな」という感触と、「Python だとこうなるんだ」という相場をつかんでもらうのが本章の目的です。

雰囲気を見るためのロゼッタストーン

Hy のサンプルと、これを hy2py で Python として出力したコードを掲載します。

```
(import sys)

(defn printer [x]
  (print x))

(defn arg2str []
  (if (> (len sys.argv) 0) (get sys.argv 1)
      (get sys.argv 0)))

(printer (arg2str))
```

```
(defclass A [][a "Attr"]
  (defn __init__ [self]
    (setv self.x "member variable"))

  (defn printme [self]
    (print self.a self.x)))

(defclass B [A]
  [dict {"key" "Value" :hi "Hy!"}
   lst [1 4 "Yeah!"]]

  (defn $$?? [self]
    (, self.dict self.lst)))

(setv b (B))
(b.printme)
(print (b.$$??))
```

```
from hy import HyKeyword
import sys


def printer(x):
    return print(x)


def arg2str():
    return sys.argv[1] if len(sys.argv) > 0 else sys.argv[0]


printer(arg2str())


class A:
    a = 'Attr'
```

```
    def __init__(self):
        self.x = 'member variable'
        return None

    def printme(self):
        return print(self.a, self.x)


class B(A):
    dict = {'key': 'Value', HyKeyword('hi'): 'Hy!'}
    lst = [1, 4, 'Yeah!']

    def hyx_is_Xdollar_signXXdollar_signXXquestion_markX(self):
        return self.dict, self.lst


b = B()
b.printme()
print(b.hyx_is_Xdollar_signXXdollar_signXXquestion_markX())
```

S 式の読み方にさえ慣れていれば、Hy と Python がどういう風に対応するのか想像するのは難しくないと思います。

S 式の読み方

- Hy の構文はシンボルとフォームで構成される
- フォームはシンボルの並びを円カッコ () で囲んだものである
- フォームは値を持つ（値を返す）
- フォームの 1 番目のシンボルは呼び出しをされ、残りは引数になる

多少の例外はあるものの、以上のルールを頭に入れるだけで Hy のコードは人力パースできてしまいます。具体例を挙げてみましょう。

```
(+ 2 3 x)
```

上記のフォームでは、「+」「2」「3」「x」がシンボルです。このうち、+が先頭にあるため「+」が関数として呼び出され、残りの「2」「3」「x」が引数となります。

Python 風に書けば、以下のようになります。

```
plus_func(2, 3, x)
```

フォームは入れ子にもなります。

```
;;; Hy
(F "some" (G x))

# Python
F("some", G(x))
```

Lisp 族の構文の基本形はシンプルです。上記のように「カッコで囲まれた要素がある」「カッコで囲まれた先頭要素が呼び出され、残りは引数扱いになる」でほぼ全てです。この原則だけ頭に入れておけば、コードリーディングであまり困らないでしょう。

これを踏まえて、Hy の構文の要素を冒頭のロゼッタストーンを参照しながら解説します。

使える記号とデリミター

冒頭コードでは $$?? のようなケッタイな名前をつけている関数があります。Hy は一般的なプログラミング言語と異なり、名前への記号の使用にほとんど制約がありません。絵文字も使いたければ使えます。以下は合法な名前の例です。

```
$*
2dConv
your-favorite_hifn
,,,
++C++;
*->
CamelCase
snake_case
```

これらの Python では違法な文字を使った名前は、コンパイル時に一定の規則で Python 向けの名前に変換されます。

名前付けへの制約で大きな特徴は、Hy では使用可能な文字がホワイトリスト式ではなくブラックリスト式な点でしょう。そのため、上記のような自由な名前付けができます。

原則として名前に含められない、ブラックリスト入りの記号は下記のものです。

```
.（ドット）
 （半角スペース）
‘ （バッククオート）
’ （クオート）
```

先頭の " （ダブルクオート。文字列になる）

「原則として」と奥歯に挟まった言い方をしているのは、リーダーマクロなどの黒魔術を使えばあらゆる行為は合法にできるからです。本書では高度なマクロの解説はしませんが、必要な時に人はそこに導かれると思います。

関数定義

Built-in にある defn マクロを使うことで関数定義が可能です。{//footnote defun ではありません。Common Lisp とのキーワード衝突が回避されているため、これを利用して無理矢理 Common Lisp とインンターオペをするという手法も存在します//}

おおむね Python の def による関数定義と 1 対 1 に対応していますが、返り値の書き方だけは多少異なります。Hy では defn 内のコードパスで一番最後に評価された値が返り値になります。{//footnote Scala などと同様の方式です//} Python では明示的に return した値のみが返り値になりますが、Hy では全ての関数が値を返します。意図しない返り値を呼び出し側に渡さないためにも、全てのパターンで返り値を書いた方がいいでしょう。

代入

setv という代入用の built-in があります。

この setv はフォームの値としては None を返しますが、Python3.8 のセイウチ演算子に対応した値を返す版の setx もあります。

```
;;; Hy
(setv a (setx x "Hy, world!"))
```

```
# Python
a = (x := 'Hy, world!')
```

setx は Python3.8 上での実行が必要になります。

各種データの生成

tuple

タプルは , （カンマ）を使って生成します。

```
(, "str" 123)

-> ('str', 123)
```

タプルを使った代入にも対応しています。

```
(defn f [] (, "タプルの" "値"))

(setv (, a b) (f))

(print a b)
;;; 出力； タプルの 値
```

dictionary

{} で囲むと辞書型になります。

```
{"key" "Value" :key 1234}

-> {"key" : "Value, HySymbol("key") : 1234}
```

{key1 value1 key2 value2 ...} のように、key と value を交互に並べていきます。Python のようにコロン及びカンマでの区切りはありません。

{} 内の要素が奇数だとコンパイルエラーになります。

list

[] で囲むとリストになります。

```
[1 2 4 6]

-> [1, 2, 4, 6]
```

list, dictionary ともに値の取得は get を使います。

```
(get dict "key")
(get ary 2)
```

Python と異なり、x[i] のようにインデックス参照をして値を取得することはできません。

```
(setv x ary[2])
```

のように書いた場合、下記のようにパースされます。

```
(setv x ary [2])) ; ary と [2] が別々に扱われる
```

この場合は setv が 2 つの引数（代入先と値）しか取ることを許していないため、コンパイルエラーになってくれます。しかし、関数の返り値で ary[2] のようにした場合は、ary の 2 番目の要素「ではなく」、 [2] というリストが返り値になります。慣れてくるとそもそもこのような書き方はしなくなりますが、Python から来たばかりの人はハマることがあるかもしれません。

文字列

ダブルクオートで囲むと文字列になります。Python と異なり、シングルクオートには Lisp 特有の用途（quote）があるため必ずダブルクオートになります。quote / unquote はマクロで頻用されます。

```
"string literal"

-> 'string literal'
```

アトリビュートへのアクセス . (ドット)

ドットは Python と同じような使い方で、アトリビュートへのアクセスができます。冒頭掲載コードのように、記号でつけられたアトリビュートへも b.$$?? のようにアクセスできます。

このような書き方ができることか察せるように、Hy でドットはパース時に特別な扱いを受けています。

シンボル :symbol-name

(コロン) から始まる名前は symbol になります。

関数のキーワード呼び出しなどで使われます。また、定数値として必要だけど値自体には意味がない時などにも使えます。

これは Python には対応する機能がありません。コンパイル後には HySymbol('xxx') のように変換されます。

コメント

; (セミコロン) から始まる行はコメントになります。また、セミコロン以降は行末までコメントになります。

Python との相互利用

Hy のデータ型は Python のデータ型のスーパーセットになっています。そのため Python のモジュールを import した上で、全ての内容を利用可能です。

モジュールのインポートは import 文を使います。

```
(import torch)
(import [numpy :as np])

;;; from module_name import some_class には対応していません

(torch.randn 3 2) ;;; -> [[0.3134, 0.66134, 0.234265], [0.12312,0.6781,0.89138]]
```

クラス定義

defclass という built in でクラスを作成できます。

```
(defclass クラス名 [継承するクラス]
  [属性名 1 属性値 1 属性名 2 属性値 2 …]

  (defn __init__ [self]) …)
  以下メソッドの定義
  …
  ))
```

属性の扱いを除き、Python の class キーワードによる定義とほぼ同様です。

属性はクラス生成後に上書きができません。そのため、クラス定義時に決定できる値を設定する、定数値のような運用が主になると思います。変更しづらいという点を使って、データ型専用のクラスを作るのに使うという手もあるでしょう。

マクロ

Lisp と言えばマクロです。Hy も Lisp の一員なので、defmacro でマクロを定義できます。

Lisp のマクロについて解説することは、「ショートイントロブック」としての本書の範囲を超えてしまいます。加えて、Hy 公式ドキュメントでもマクロの解説は十分とは言い難いように思います。

幸い Hy のマクロは他の Lisp 族と似たようなもののようなので、Common Lisp などのマクロの解説を適用していけるでしょう。Lisp におけるマクロが詳述された書籍として、Paul Graham 著「On Lisp」を挙げて代わりとしておきます。

第 3 章

開発環境

開発環境はプログラミング言語そのものの内容ではありませんが、入門時に困りがちな情報として掲載します。

前提

著者の環境が Linux デスクトップなので、それを基本とした内容になっています。

Mac ユーザーの場合、本章内容の多くはほぼそのまま利用可能なはずです。筆者の宗教上の理由で Mac 機を保有していないため、実際の動作は一切検証していません。

Windows ユーザーの方には、WSL(Windows Subsystem on Linux) の利用を推奨します。WSL で neovim や Hy がある程度動かせることは確認済みです。

WSL は Docker が動くか怪しい、IO が絶望的に低速などの問題があります。それでもコマンドプロンプトや Cygwin の利用で発生する問題と天秤にかけるなら、こちらの方がマシだと筆者は考えています。WSL 自体は Microsoft が本気で改善を続けているため、現時点での不都合は将来的な解決も期待できます。

各エディタの現状

メジャーどころから Emacs, vim, VSCode の 3 つに加え、Emacs の拡張で別物になった Spacemacs を取り上げます。

Emacs

Lisp と言えば Emacs です。Hy はエディタのサポートが潤沢とは言い難い言語ですが、Emacs（及びその拡張の Spacemacs）は相対的には一番サポートが厚いです。

hy-mode は 2019 年 9 月現在もアクティブなようで、更新頻度が高いとは言えないものの当面は利用し続けることが期待できます。また、Emacs の伝統的な Lisp 拡張 paredit なども利用することができます。

Vim

vim での Hy に関しては、現状はかなり厳しいと言わざるを得ません。vim-hy という拡張はありますが、現在では unmentained な上に動作に問題が出ている状態です。

誰かが vim 向け拡張を作れば、それがデファクトになるオープンソースチャンスという見方はあります。

VSCode

VSCode のマーケットにある拡張は、現状シンタックスハイライトがつくものしかないようです。こちらも IDE とは程遠い環境にしかならないでしょう。

Spacemacs

Spacemacs は Emacs の拡張で、原理的に Emacs と同等の環境構築が可能です。Emacs でありながら、通常モード・挿入モードの使い分けなど vim 風のキーバインドがあるのが特長です。また Emacs 拡張の導入など、面倒なところの面倒を見てくれます。

特筆すべき点として、Spacemacs 公式から hy-lyaer が提供されています。そのためドットファイル（.spacemacs）に一行追加をするだけで hy-mode と関連する拡張が利用可能になります。結果として得られる文明レベルは生の Emacs と同等ですが、導入の容易さでは Spacemacs に軍配が上がるでしょう。

で、どのエディタがいいのか

大前提として、全体的に未整備な世界なので己の信じるエディタがある人はそれを貫いた方が幸せになるでしょう。

特にこだわりがない人には、Spacemacs を推奨します。

- vim, VSCode に比べて相対的には Hy 関係の拡張に恵まれている
- 導入が容易。Emacs や vim と異なり、拡張管理のための環境整備などが不要

何かのエディタのプロでない人には、これらの特長は初心者にやさしくメリットが大きいです。

Spacemacs での環境構築

ここでは Spacemacs の導入を簡単に解説しつつ、Hy 向けの設定を有効化する手順を説明します。

Spacemacs のしくみ

Spacemacs の実体は、.emacs.d に作られた Emacs の拡張です。そのため、Emacs のインストールは前提となります。そのため、既存の Emacs を乗っ取る形で動作します。通常の emacs との共存は困難なので、既存 Emacs ユーザーは注意が必要です。

Spacemacs の導入手順

前提として、Emacs をインストールします。
その後、Spacemacs のリポジトリを .emacs.d に clone します。

```
git clone https://github.com/syl20bnr/spacemacs ~/.emacs.d
```

master のバージョンはやや古いままアップデートされていないので、develop に追随します。筆者が検証した時点では、Spacemacs デフォルトバンドルの Hy レイヤーを利用したい場合は develop のバージョンでないと動作をしませんでした。

```
cd .emacs.d
git checkout develop
git pull origin develop
```

ここで Emacs を起動します。初回起動時の拡張が色々インストールされ、各種ファイルの生成も走ります。起動が終わったら、 ホームディレクトリに生成される .spacemacs を編集します。.spacemacs の dotspacemacs-

configuration-layers に、使用する「レイヤー」を追記していきます。

「レイヤー」をものすごく雑に説明すると、Spacemacs で拡張や設定をパッケージ化したようなものです。このあたりを掘り始めると Emacs Lisp の沼に沈み込んでいくので、詳細な解説は文献をあたってください。

```
dotspacemacs-configuration-layers
'( python
  hy
  auto-completion
  better-defaults
  emacs-lisp
  ...
```

"python" や "hy" といったレイヤーは Spacemacs 公式から提供されているため、上記の設定のみでユーザーは利用可能になります。

エディタを再起動するか、 SPC f e R (dotspacemacs/sync-configuration-layers) コマンドを実行すると hy-layer に必要な拡張がインストールされます。これで hy-mode が利用可能になりました！

その他の設定

拡張の導入は以上で完了するのですが、paredit など一部の拡張向けのキーバインドは自動では設定されません。そのため、ユーザーによる設定でキーバインドを追加する必要があります。

キーバインドの設定方法は複数ありますが、はじめのうちは.spacemacs の user-config にベタ書きしてしまうのが一番楽だと思います。もちろん、Spacemacs にはモジュール分割を使ったより高度な設定方法も存在します。

下記に筆者の追加したキーバインド例を挙げます。一般的でないキーバインドにしているためこの通りにするのは推奨しませんが、必要なキーバインド設定の例として眺めてみたください。

```
(defun dotspacemacs/user-config ()
  (define-key evil-normal-state-map (kbd "SPC (") 'sp-wrap-round)
  (define-key evil-normal-state-map (kbd "SPC )") 'sp-unwrap-sexp)
  (define-key evil-normal-state-map (kbd "SPC [") 'sp-wrap-square)
  (define-key evil-normal-state-map (kbd "SPC ]") 'sp-unwrap-sexp)
  (define-key evil-normal-state-map (kbd "spc >") 'sp-next-sexp)
  (define-key evil-normal-state-map (kbd "spc <") 'sp-previous-sexp)
   )
```

このキーバインドでは、例えば "SPC (" を入力すると選択部分を () で囲み、"SPC)" でカッコを外します。このあたりの設定は、みなさんの好みに設定してください。

実行環境の作り方

Hy は Python のランタイム上で動作します。そのため、実行環境づくりも Python での手法に準拠します。

素朴な手法としては pip install などでグローバルにあれこれインストールする手法がありますが、これは _絶対にやめましょう。_ Python は dependency hell を起こしやすく、さらに Python2 系と Python3 系が混在するのが当たり前のシステムが大半です。どんなに注意深くやっても事故は起こりますし、仮にあなたが達人でもその注意力を建設的な作業にふりむけられるようにするべきでしょう。

Python の経験が浅い読者は「大げさだなあ」と思われるかもしれません。読者が必要を感じられない場合は、一旦この節の内容は忘れて素朴な管理方法をしてみるのもいいでしょう。色々あって破綻して、OS 再インストールをしたくなった時に再読していただければ幸いです。

Docker

Docker に全てを閉じ込めてしまえば、その中では環境汚染し放題です。システムインストールではやる気にならないお行儀の悪いことも躊躇なくできます。

Docker を利用する場合、Dockerfile に pip install を書き連ねていくスタイルになります。

Docker 内で Anaconda, virtualenv などは不要です。ライブラリなどは Docker イメージ単位で管理ができますし、何かが壊れてもコンテナを破棄すればそれで済みます。一方で virtualenv 等のインストールはイメージサイズを肥大化させてしまいます。Docker のメリットを活かすためには不要でしょう。

Docker イメージについて、Hy 公式から提供されているものもありますが、ベースイメージや Python のバージョンなどで融通が効きにくいです。筆者は Python 公式が提供している Docker イメージをベースに、その上に Hy 及び必要なライブラリをインストールする Dockerfile を作成するスタイルにしています。

```
from python:3.7-alpine

RUN pip install hy && pip install 必要なパッケージ
```

Docker 利用での開発スタイル

hy コマンドで実行するだけなら、 docker run をするだけで通常のコマンド実行に近い感覚で実行できます。

```
docker run -rm your-hy-image -v /to/your/data:/data main.hy
```

コーディングを行う場合には、考慮する点が出てきます。

hy-mode を利用した自動補完や、REPL 駆動をする場合はエディタの動作している hy の環境が必要になります。Docker に実行環境を閉じ込めている場合、ホスト側で動作するエディタはこれらを利用できません。

解決方法の 1 つは、Docker イメージないにエディタをインストールしてしまい、コンテナ内で vim や Emacs を開いて編集するというものです。ただしこれは CUI 限定になりますし、ホスト側での動作と機能性が異なってくる場合があります。

その他の方法としては、VSCode preview に搭載されているようなコンテナ接続機能を使う方法です。

virtualenv

Python の環境管理ツールの標準です。過去には似たようなツールも出回っていましたが、virtualenv の利用におおむね収束したと言えます。virtualenv 自体の利用法は Python と変わりません。

REPL 駆動

Lisp での開発と言えば REPL 駆動です。REPL 駆動とは、書いているコードの一部を REPL で評価しながら、その動作を見て開発をしていく手法です。その場限りの書き捨て単体テストを連ねていくスタイルとも言えます。

実際には 1 行ずつ hy のコンソール上で評価するわけではなく、エディタの機能を使います。

Spacemacs での REPL 駆動

hy-mode が有効化されていると、hy-shell-eval-xxx といったコマンドが利用できるようになります。利用例として、

1. 評価したいコード部分をビジュアルモードで選択
2. , s r を入力（もしくは SPC SPC から、hy-shell-eval-region を入力）

これだけで、Spacemacs が内部で持っている Hy の REPL 上でコードが評価されます。数行程度書く→ REPL で評価する→結果を見てコーディング、のループを回すと気持ちよく開発できます。

デプロイ

Docker コンテナを実行できる環境へのデプロイなら、Docker イメージを作成してデプロイで済みます。これは他の言語やフレームワークと何も変わりません。

Hy 特有の例として、Cloud Functions へのデプロイを取り上げます。

Python への変換と Cloud Functions へのデプロイ例

Cloud Functions は、

- サーバーインスタンスの面倒を直接見る必要がなく、
- 実行時間の分のみの課金で済むので利用方法によっては大きくコストを削減できる

という利点の一方で、

- ランタイムが GCP 側で準備したものしか使えない

という制約があります。AWS Labmda も同様ですが、この手のサービス

で普段遣いしている言語がなくて悲しい思いをしがちです。

Hy のようなマイナー言語自体は、当然ながら Cloud Functions ではサポートされていません。しかし、Python ならサポートされています。そして Hy は Python と完全にコンパチかつ、Python へのトランスパイラも付属しています！

ここまで来れば、やることは非常に単純です。

- Cloud Functions で Python ランタイムのサービスを作成
- Hy で書いたコードを、hy2py で Python に変換
- 変換した Python コードを、通常の Python ファイルのデプロイと同様に処理

やっていることは非常に簡単ですが、これは Hy ならではの利点です。「困ったら Python コードを出力すれば何とかなる」という手法の一例として、Cloud Functions へのデプロイは典型と言えます。

第 4 章

少し洗練された書き方

第 2 章で、Hy の構文をごくごく簡単に紹介しました。Python や他のプログラミング言語の経験があれば、S 式のルールになじんでしまえばとりあえず書くだけならそこまで困らないでしょう。

本章では、ざっと構文を眺めただけだと気づきにくい機能・スタイルからピックアップして紹介をします。

メソッドチェーン

```
obj.f().g(x).h(y, z)
```

のような形のメソッドチェーンは、Hy の構文とあまり相性が良くありません。しかし、Python のライブラリではこのような形で API が提供されている場合もあり、一時変数を使うよりはメソッドチェーンを利用したい場合があります。

Hy にはメソッドチェーンを呼び出しやすくするための、doto という構文があります。

```
(doto obj (.f) (.g x) (.h y z))
```

これで、上記の Python と等価なコードに展開されます。

ドットの機能と利用法

attribute へのアクセスのために、ドット（.）が用意されています。ドットは Hy の構文上、特殊な扱いを受けています。

基本

```
(. obj attr subattr [(+ 1 2)] subsubattr)
```

上記のフォームは、Python で

```
obj.attr.subattr[1 + 2].subsubattr
```

のように展開されます。

(.) フォームは、最初の引数をオブジェクトとして、以下各引数で attribute への参照を行います。今回の例では、第一引数の obj がオブジェクトで、第 2 引数の attr という名前の obj.attr という attribute への参照を行い、以下の引数にも同様の attr への参照を繰り返します。

attribute へのアクセスで利用できるのは、シンボル（生の変数名）かリスト [] のみです。その他の要素、"str" のような文字列や、:hysymbol など

の即値を引数にするとコンパイルエラーになります。

(.) でのアクセスはあくまで attribute へのアクセスのため、メソッド呼び出しの場合はさらに呼び出しのフォームとして囲ってあげる必要があります。

```
((. obj F) "some argument")
```

簡易形

ドットによる attribute アクセスは、シンボル中でも動作しています。上記の関数呼び出しは、下記のようにも書けます。

```
(obj.F "some argument")
```

オブジェクト指向なメソッド呼び出しでは、こちらの方が読みやすいでしょう。本当は attribute へのアクセスが裏で行われているのですが、何となくこう書くと動いてしまうので雰囲気で attribute へのアクセスを含むコードは書けてしまいます。

Hy のスタイルガイドでは (obj.F) より (. obj F) のスタイルが推奨されています。ただ、筆者の意見としてはクラスベースオブジェクト指向なライブラリで毎回ドットから始まるフォームを書くのはあまりに冗長なので、適宜 Python ライクな obj.attr の attribute アクセスは使ってしまって問題ないと思います。

変数名の中でドットを使うスタイルは便利ですし、可読性を上げることもあります。しかし、そのためにドットは Hy の構文解釈上で特殊な扱いを受

けているため自由に変数名などに使うことが難しいです。

このあたりは言語の構文としては若干ダーティな感じにはなっていますが、Hy が Python であることを選択してる結果と言えるでしょう。

Threading macro の利用

Hy には threading macro（->, ->>）が標準で用意されています。これらを使うと、Python では一時変数と再代入が必要だったコードがそれらなしできれいに書けることがあります。

一般的な利用法

公式ドキュメントには下記のように説明されています。

```
The threading macro inserts each expression into the next expression'
s first argument place.
(訳：スレッディングマクロは、[訳注：スレッド内の] それぞれの式を次の式の最初
の引数に配置します。)
```

文字通りの意味なのですが、具体例を出した方が分かりやすいので下記に概念的なマクロ展開例を出してみます。

```
; (defn f [x y z] (+ x y z)) など何か適当な関数
(-> 10
  (f 1 2)
  (f 3 4))

(->> 10
  (f 1 2)
  (f 3 4))
```

```
↓↓↓

(f (f 10 1 2) 3 4)

(f 3 4 (f 1 2 10))
```

実際の使用例

挙動は分かったとして、これだけだと有用性がピンと来ないかもしれません。ここでは私が Python から Hy に移植したコードの抜とを比較してみます。

```
# 移植元の Python コード
# https://github.com/huggingface/pytorch-pretrained-BigGAN/blob/master/pytorch_pretraine
り

    def forward(self, cond_vector, truncation):
        z = self.gen_z(cond_vector)

        z = z.view(-1, 4, 4, 16 * self.config.channel_width)
        z = z.permute(0, 3, 1, 2).contiguous()

        for i, layer in enumerate(self.layers):
            if isinstance(layer, GenBlock):
                z = layer(z, cond_vector, truncation)
            else:
                z = layer(z)

        z = self.bn(z, truncation)
        z = self.relu(z)
        z = self.conv_to_rgb(z)
        z = z[:, :3, ...]
        z = self.tanh(z)
```

```
        return z
```

```
;;; 移植した Hy コード

(setv ,,, Ellipsis)

(defn forward [self z truncation]
    (with [(torch.no_grad)]
      (-> (self.G.gen_z z)
          (doto (.view -1 4 4 (* 16 self.g.config.channel_width))
            (. permute 0 3 1 2)
            (.contiguous))

          (layers-forward self.G.layers z truncation)

          (self.G.bn truncation)
          (self.G.relu)
          (self.G.conv_to_rgb)
          (get [(slice None None) (slice None 3) ,,,])
(self.G.tanh))))

(defn layers-forward [z* layers z truncation]
  (for [layer layers]
    (cond [(in "GenBlock" layer.__class__.__name__) (setv z* (layer z* z truncation))]
          [True (setv z* (layer z*))]))
  z*)
```

Python 版では z = … を毎行書いていたのが、Hy 版では一時変数と代入が forward 関数中からは撲滅できています。1 引数関数の呼び出しでチェインする部分は特に見通しが良くなっているのではないでしょうか。

一時変数を使わないとネストが深くなってコードが見づらくなる場合には、-> や ->> の使用は検討の価値があると思います。

第 5 章

深層学習への利用

Python は深層学習ライブラリが最も充実している言語です。これらの Python の資産を利用して、Hy で深層学習が可能です。

本章では、代表的な深層学習フレームワークのひとつ PyTorch を例に取り、Hy での深層学習ライブラリの利用方法を挙げていきます。例として用いたフレームワークは PyTorch ですが、同様の方法で TensorFlow / Keras や Chainer など Python 向け深層学習ライブラリは利用できるでしょう。

フレームワークの利用法

基本的な使い方は Python と変わりません。

```
(import torch)

(setv x (torch randn 5 3 100 120))


x.shape
;; Size(5, 3, 100, 120)
```

Hy は Python と完全な互換性があるので、Python で書かれた各種コードは機械的に Hy に置き換え可能です。

Python コードを置き換えただけの Hy コードでは、特に Hy を使っているメリットを享受できるわけではありません。しかし、最悪でも Python と同等の書き方ができるという事実はある種の保証になります。「Hy でなく Python でなら実装できていた」ということはありえないので、安心して Hy を採用可能です。

Hy で深層学習に取り組む時は、まずサンプルの Python を Hy に書き写してみて、そこから改善していくというのは一つの方法になるでしょう。

Hy のパフォーマンス

機械学習は多量のコンピューティング資源を消費します。Hy は元から早いとは言えない Python ランタイムの上で、さらに Hy の処理系を走らせているのでお世辞にも高速な部類ではないでしょう。そのような処理系で機械学習をして大丈夫でしょうか？

この点に関しては、深層学習フレームワークを適切に利用する限りは、Python に比べて大きな問題は起きません。そして世の深層学習は多くの場合は Python で回せています。

深層学習の速度で、でボトルネックになりそうな部分を挙げていくと、以下のようになるでしょう。

1. GPU 上での処理速度
2. データの IO の速度（GPU へのデータ転送も IO の一種とみなせます）
3. スクリプト上での処理

このうち、Hy が低速なことで影響を受けるのは 3 番目です。ところで、このあたりの事情は Python でも変わりません。そのため、深層学習フレームワークでは 3 番目の処理で重いものは無くて済むように設計されていま

す。Hy での深層学習フレームワークでも同様の利用をするため、Python と比べて多少低速であることは問題にならないでしょう。

パフォーマンスについては、第 6 章でベンチマークも混じえた考察を行っています。

Jupyter の利用について

機械学習では Jupyter の利用も一般的で、Python では Jupyter 上で全てを片付けてしまうスタイルも一般的です。Hy にも Jupyter のカーネルがありますが、残念ながら Python 並のサポートは受けられないのが現状です。

筆者が試した限り、Hy カーネルは

- 動作が不安定。Jupyter 自体の再起動が必要になる場合が少なくない
- GNUPlot などの描画機能に対応していない

など、Jupyter を敢えて使うメリットが少ない状態です。Hy カーネルの開発に比べて Jupyter 自体の進化が早いので、誰かが気合を入れて Hy カーネル開発にコミットしない限りこの状況は改善しないと見込まれます。

ではどうするかという話ですが、エディタ上での REPL 実行をする REPL 駆動開発で Jupyter のセル単位の実行に近いスタイルをすることはできます。（REPL 駆動の具体的な方法に関しては、3 章「REPL 駆動」の節を参照）元よりコーディング自体はブラウザ上でやるより Emacs 等のエディタ上でやる方が快適なので、コーディング＆コード実行のサイクルは慣れればブラウザ上でやるより楽にできます。

試論：マクロの利用による静的検査

本節では、深層学習フレームワークに Hy のマクロのラッパーを作成することで、コンパイル時に可能な検査をしてしまう手法を提示します。

本節の内容はまだ検証前の段階で、本格的な実用を行える段階には至って

いません。そのため「試論」とした上で、潜在的な Hy の価値を示すために敢えて掲載しています。

著者は現在実装中の深層学習タスクで本節の内容を実戦利用の予定があるため、本書のバージョンアップ時には「試論」が取れた内容でお届けをする予定です。

Conv 算

深層学習フレームワークの利用者なら、入力した配列の次元が期待される値と違ってエラーを起こしたことがあります（断言）。

例えば、PyTorch の 2 次元畳み込みは次のような API を持っています。

```
（PyTorch ドキュメントから引用すること）
class Conv2d(in_channels, out_channels_, stride, padding, ...)
```

第 1 引数に注目してください。ここでは入力チャンネル数をユーザーが明示的に指定する必要があります。もしここで指定したチャンネル数と異なる入力をした場合、実行時エラーが発生します。このような形で入力次元に制約のある API は、深層学習ライブラリでは一般的です。

この手の次元不一致はたいてい実行時になって判明します。アプリケーション制作やシステムプログラミングと違い、条件分岐を多用することはあまりない機械学習のコードでは埋まったまま見つかりづらいバグになることは少ないでしょう。しかし、深層学習の場合は一度の実行に長い時間がかかります。コードパスの後ろの方でこの手のバグがあった場合、実行開始から何時間も経ってから実行時エラーからのやり直しになることもあります。

つまり、静的解析の需要があります。そして、この問題は Python では解決困難です。ここで Hy のマクロを利用することで、実行前のエラー検出をする手法が考えられます。

前提を整理すると、

- 実行前に配列次元が固定されている。実行時に配列次元を可変にする必要性は、深層学習ではあまりありません
- コンパイル時に配列次元数の検査をしたい

以上の要件を、マクロの利用で解決していきます。

準備

深層学習に適用する前に、よりシンプルな状況でマクロを構成してみます。「ある関数が実行される前に、引数が前提を満たすか assert をかける」を目標にします。

次のような関数があったとします。

```
(defn f1 [n]
  (print "f1" n)
  (* n 2))

(defn f2 [n]
  (print "f2" n)
  (* n 2))


;;; 次のような呼び出しがをする予定

(setv x (f1 10))
(setv x (f2 x))
```

それぞれ引数を取って、その値を表示します。与えられた引数によって決まる返り値を返す、単純な関数です。print フォームは任意の副作用と考えれば、このような形式の関数は実践でも一般的です。各関数内の副作用ーこの場合は print ーが実行される前に、引数が特定の値であるかの検査をする

仕組みを作りましょう。

基本的なアイディアは、以下のようになります。

- f1, f2 関数それぞれのラッパーマクロ f1*, f2* を作り、利用側はそれらを呼び出す
- マクロ展開時に assertion をかける
- マクロの展開後は、通常の関数呼び出しと同様の展開形にしたい

これを踏まえて実装すると、下記のようなマクロができます。

```
(defmacro f1* [n]
  (assert (= 100 n))
  (, `(f1 ~n) n))

(defmacro f2* [t]
  (assert (odd? (second t)))
  (, `(f2 (first ~t)) (* (second t) 2)))

;;; 利用法

(f2* (f1* 100))

;;; Threading macro も使える

(-> (f1* 100)
    (f2*))
```

マクロ f1*は、引数 n に対して以下の 2 段階の処理をします。

1. n の値への assertion
2. n を使った関数 f1 の呼び出しと、n を含んだタプルを展開形にする

ここで、1 段目の処理はマクロ展開時に評価がされます。マクロの展開後にはマクロの「戻り値」だけが残るため、マクロ f1*内での assert は展開後

には残りません。結果として、(f1* 100) のような呼び出しは、マクロの展開後は (, ‘(f1 100) 100) のようになります。

このマクロ f1*は、引数 n が 100 であることを想定しています。n に 100 以外の値を入れて呼び出した場合に、何が起こるでしょうか？

仮に (f1* 2) のような呼び出しをしたとします。マクロ展開時に、マクロ f1*内のフォームが順番に評価され、(assert (= 100 2)) が評価されることになります。この時点で AssertError が発生し、コンパイラはエラー発生として中断します。これはマクロ展開時、つまりプログラム本体の実行前であることに注目です！ 実際、次のようなプログラム

```
(print "started!")
(f1* 2)
```

を実行すると、"started!" が表示される _前に_ AssertionError が発生して、コンパイラが停止します。

マクロ f2* は、マクロ f1* の展開後のタプル t を取り、以下の 2 段階の処理をします。

1. 引数タプル t の 2 番目の値で assertion
2. 引数タプル t の 1 番目の値での関数 f2 の呼び出しと、2 番目の値を使ってタプルの 2 番目の戻り値を構成

assertion と展開形に関しては、マクロ f1*と同様です。これらを合わせて、

```
(f2* (f1* 100))
```

とすると、概念的には下記のように評価と展開されます。

```
1. (f1\* 100) の展開が開始
   1. (assert (= 100 100)) が評価され、検査をパス
   2. 展開後の (, `(f1 100) 100) が残る
2. (f2\* (, `(f1 100) 100)) の展開が開始
   1. (assert (odd? (second (, `(f1 100) 100)))) が評価され、検査をパス
   2. 展開形が下記のように評価される


(, `(f2 (first (, `(f1 100) 100))) (* (second (, `(f1 100) 100)) 2))

; 展開型タプルの 2 番目はこの時点で評価されるので、結果的な展開型は

(, `(f2 (first (, `(f1 100) 100))) (* 100 2))
; ↓ (* 100 2) のフォームが評価され、最終形
(, `(f2 (first (, `(f1 100) 100))) 200)
```

f1* と f2* の返り値・引数は、関数の評価をクオートしたものと引数を評価した値のペアにしています。これにより、「f1* に与えられた引数の性質を引き継いで f2* でも検査を行う」ことが可能になります。これは f1* が単純に関数 f1 を評価するフォームだけを返しているとできないことです。関数の戻り値が自明なこの例だと意味を感じにくいですが、次節以降の深層学習 API を利用する時には活用できる性質です。

ニューラルネットモジュールへの適用

前節までの内容で、ニューラルネットの静的検査をするためのテクニックは揃っています。違いは適用する関数が複雑になっていることだけで、マクロの基本的な構成は同じです。

```
;;; nn.Conv2d のラッパー
(defmacro conv2d* [prev Cout kernel-size stride padding dilation]
  (setv (, N Cin Hin Win nets) (eval prev))

  (setv Hout (int (/ (+ Hin (* 2 padding) (- (* dilation (- kernel-size 1))) -1) stride)
  (setv Wout (int (/ (+ Win (* 2 padding) (- (* dilation (- kernel-size 1))) -1) stride)


  (,  N Cout Hout Wout `[~nets ``(nn.Conv2d ~~~Cin ~~~Cout ~~~kernel-size ~~~stride ~~~p

;;; nn.BatchNorm2d のラッパー
(defmacro bn2d* [prev]
  (setv (, N Cin Hin Win nets) (eval prev))
;  (print nets)

  (setv num-features (* N Cin Hin Win))

  (, N Cin Hin Win `[~nets ``(nn.BatchNorm2d ~~~num-features)]))


(setv net (-> (conv2d* (,  5 3 100 200 `[]) 30 1 1 0 1)
              (bn2d* )
              (conv2d* 35 1 1 0 1))

;; net に nn.Conv2d と nn.BatchNorm2d のリストが入る
;; リストがネストした汚い形になるので、利用時に適宜 falttten する必要はありま
す
```

上記は畳み込み（Conv2d）と Batch Normalization(BatchNorm2d) を挟む、畳み込みニューラルネットの一部でよく見るパターンです。マクロを利用して、コンパイル時に入力次元数を計算して引数にしています。

これでニューラルネットの静的検査をしながらモデルが組める！ 大勝利！！…と言いたいところですが、上記のプログラムはネットワークの深さが一定以上になると

```
hy.errors.HyCompileError: Internal Compiler Bug
```

なメッセージが出て死んでしまいます。本書初版執筆時点では時間切れで解決できませんでしたが、近日中に解決策を導いて改訂時には「試論：」を取れるようにする予定です。

第 6 章

パフォーマンスの考察

Hy は Python のフロントエンド「のようなもの」です。元より速くはない Python ランタイムの上で、さらに Hy の層が加わるとなるとパフォーマンスが気になるのは人情だと思います。

本章では、Hy のパフォーマンスとその実用上の意味について検討していきます。

Hy ランタイムの概要

Hy のコンパイラが呼び出された時、

1. Hy ソースのパース：Hy の AST を生成
2. Hy の AST を Python の AST に変換
3. Python AST を Python バイトコードに変換
4. Python バイトコードを Python ランタイムで実行

という処理が行われます。
以上から、次のような性質が予想されるでしょう。

- 実行速度のスループットは Python と同等にななる

- 立ち上げ時にコンパイルの分オーバーヘッドが発生する

本当に Python と同等の速度なのか

「原理的には Python と同等の実行速度になる」のですが、原理は現実の前に汚されがちです。hy2py で Python ソースコードに変換したものを読むと、潜在的に速度が落ちる要素が見えたりします。

例 1：マクロ展開形
```
hyxxxx_yyyyy(annovar(x))
```
のような式が生成されることがあります。

例 2：シンボル
```
:name
```
↓↓↓
```
HySymbol("name")
```

のように、オブジェクト生成を伴う式にコンパイルされています。

一方で、defn や derclass など Python と 1 対 1 に対応する機能を用いただけの場合は、極めて素直な Python コードが生成されます。生で Python を書いた場合と見分けがつかないでしょう。

```
(defn tarai [x y z]
  (if (<= x y) y
      (tarai (tarai (- x 1) y z) (tarai (- y 1) z x) (tarai (- z 1) x y))))

# hy2py で出力されるコード
def tarai(x, y, z):
    return y if x <= y else tarai(tarai(x - 1, y, z), tarai(y - 1, z, x),
```

```
        tarai(z - 1, x, y))
```

まとめると、

- Hy 特有の凝った機能を使うと（潜在的に）遅いコードが出力されがち。特にマクロの利用で顕著になります
- Python にある機能を使う限り、Python そのもののようなコードが出力される

ということになります。

現実に遅くなるのか

では Hy の実際の利用環境で、Python に比べて意味のある速度差が出るのでしょうか？ 筆者としては、答えはノーだと考えています。

HyExpression が入ることで速度差が出るようなケースは、次の 2 つが考えられます。

1. 重い計算処理でループの内側にあるような場合
2. レイテンシが重要な処理

1 番目に関しては、ここでは高速なコードが生成されるように気を使う必要があります。ただしこれは、Hy に限らずあらゆるプログラミング言語で同様です。「高速」とされているプログラミング言語、例えば C++ や Java であっても、ループの内側で不用意な処理をすると大幅に低速化することには代わりはありません。つまり、Hy の速度云々よりもプログラマーの責任の問題だと言えます。

2 番目に関しては、そのような処理を（生の）Python で書くこと自体がほとんど場合に間違った選択です。1 番目のケースも含めて、高度なチュー

ニングやレイテンシの削減が必要な場合は、Cython を導入する・部分的に C++ で書くなどの解決法を検討するべきです。

ところで、Python には numpy をはじめコンピューティングのための高度なライブラリが充実しています。これらのライブラリでは、「ループの外側」を Python で書くだけで高速に、また多くの場合並列化をした処理を行ってくれます。

Hy を、ひいては Python を利用するそもそもの動機はこれらのライブラリにあることが多いでしょう。つまり、現実の問題を解く場合には、上記のような高速化について考える必要がある場合はあまりないはずです。もしあったら、問題設定から見直した方がいいかもしれません。

総合して、Hy は Python で十分な場合には十分は速度を出すし、そうでない時には不足した速度しか出せません。Hy はあくまで Python 世界の住人です。能力も限界も、Python そのものであると考えればいいでしょう。

ベンチマーク

いくつかのケースについて、筆者の手元でベンチマークを取った結果を見てみます。Hy のコードと、それを hy2py した Python コードの実行で比較していきます。

立ち上げ速度

大量ののクラスがあるソースファイルの実行です。

```
(defclass ClassNameXXXX [][]
  (defn f [x] (+ x 0.123434))) ;; 生成時に加算する数値をランダム生成

;; このクラスを 10 万個並べている
;; hy2py で下記のようになる
```

```
class ClassNameXXXX:
    def f(x):
        return x + 0.12345
```

結果：

| クラス数 | Hy | Python | | -------- | ------- | ------- | | 100 | 1s 以下 | | | 1000 | 1s 以下 | 1s 以下 | | 10000 | 1s | 1s 以下 | | 100000 | 10s | 1s 以下 |

実用的な範囲では Hy と Python で大差はありませんが、1 万行を超えたあたりからは Python が有意に速くなります。ソースファイルにデータを埋め込むようなエッジケースでは、Hy であることで問題が発生しそうです。

CPU だけの計算

上で挙げた竹内関数（たらい回し関数）を計算してみます。

```
(tarai 14 6 0)

tarai(14, 6, 0)
```

実行結果：

| | Hy | Python | | ----- | ----- | ------ | | tarai | 39.75 | 39.7 |

有意差はないと言っていいでしょう。

メモリや IO を含む計算：深層学習

MNIST データセットに対して、（バニラ）GAN で画像生成をするモデルの学習をする深層学習のコードに対して、学習速度の計測をしてみます。コードは （引用元）の Python コードと、これを Hy に同内容で書き直し

たものを使っています。

このコードは、以下のような要素を含んでいます。

- CUDA の利用：GPU 上での演算と、GPU との間で転送が発生します
- メインメモリの読み書き
- ファイル IO（MNIST データの読み込み）

10epoch の実行時間は、HY 89.49 Python 89.28 でした。このベンチマークは epoch に対してのみ行っているので、Hy のコンパイル時間は含まれていません。実行時間で 0.2% ほど Python が速くなっていますが、有意な差はないと言えるでしょう。天気のいい日に実行したら Hy の方が速くなるかもしれない範囲です。

結論と未解決事項

白魔術を使う限り、Hy であることが理由でパフォーマンスの問題は起きません。Hy で問題になる場合は、Python でも問題が起きるはずです。

マクロをはじめ黒魔術を駆使した場合に何が起こるかについては、今回は未検証です。Hy のコンパイラをクラッシュさせるマクロくらいは頑張れば作れそうな気はするので、オモチャがどれくらいで壊れるか遊んでみたい方は挑戦してみてはいかがでしょうか？

参考文献

本章で解説した Hy コンパイラの動作は、概要に絞った説明になっています。Hy 公式ドキュメントでは、より詳細なコンパイラの動作の解説があります。

http://docs.hylang.org/en/stable/language/internals.html#hy-internal-theory

マクロの展開タイミングなど、より詳細な情報はこちらから読むことができます。

第 7 章

寄稿：Hylang と Python ライブラリをつかってロボットを動かしてみよう

城野遼太さんにロボット操作での Hy の使用レポートをいただきました。城野さんは Hy 自体はこの日初めて触っていますが、本稿のように支障なく使えたようです。Hy の利用範囲のポテンシャルを感じていただければ幸いです。

概要

巷で大人気の Python には豊富なライブラリ群があります。教材の類も充実してますし、会社や研究室の先行プロジェクト資産もあるでしょう。Hylang は幸いにして Python との連携機能が充実しているのでこういった先行資産を活用しながら、徐々に Lisp 色に染めていくというスタイルを現実的に実行可能なヤベぇ特徴を備えています。

今回はオープンソースなロボットアーム Open Manipulator X をシリアル通信経由で操作する単純なアプリケーションを考えます。

実際の利用

当該のロボットアームは Ubuntu 端末からシリアル通信経由で所定の指令文を送ることで操作します。Python でやると、こんな感じになります。

```
import serial
ser = serial.Serial('/dev/ttyACM0', 57600)
ser.write("task,up\n")
```

これを Hylang で書き直してみましょう。

Python ライブラリのインポートまず、ライブラリインポートですが、これはそのまんまの記法です。

```
(import serial)
```

次に変数設定。let は残念ながらない (欲しい時は自分で作りましょう) ので、Hy 公式のスタイルに則って setv で値を定義します。文字列はダブルクオートじゃないと受け取って貰えないので注意!

```
(setv ser ( serial.Serial "/dev/ttyACM0" 57600))
```

シリアルの書き込み

次に書き込みです。なんも考えないと、こんな感じの実装になるかと思います。

```
(ser.write (str.encode "task,back\n"))
```

Clojure 由来らしい魔界記法 ”スレッドマクロ"をつかうと..... 以下のように書き下せます。

```
(-> (str.encode "task,up\n")
  (ser.write))
```

指令文本体は単体の要素として先に切り出して渡してあげてもいいので.....

```
(-> "task,up\n" (str.encode)
  (ser.write) )
```

この例ですと旨味がないですが、"."記法を使う場合、スレッドテイル・マクロを使い末尾にくっつけるようにしてこう書くこともできますね。

```
(->> "task,down\n"
  (.encode str)
  (.write ser) )
```

スレッドテイル方式のものを、書き込み部だけ関数に切り分けましょう。defun じゃなくて defn です。

```
(defn robotTask [dist] (-> (+ "task," dist "\n") (str.encode) (ser.write)))

(robotTask "up")
```

繰り返し処理も見通しよくなりますね。

感想

データをつくって渡していく構造、シェルスクリプト っぽいですね。本格的に S 式で作り込んで行けば、Python や C だと目が回るような複雑な処理もスマートに記述できそうです。

執筆者情報

tackman

Android アプリ屋、ソシャゲの会社、Web などの受託を経て現在フリーランス。個人制作で深層学習に手を出した時に初めて本格的に Python を触り、この言語を作ったのは誰だあっと叫んでいる時に Hy に出会った。

個人の活動だと、百合のボードゲームの比重の方が高い気がする。

城野遼太

都内某大学でロボット屋を営む零細事業主。最近、暗号制御なる分野の研究をすすめているらしい……。

日本 Hy ユーザー会について

Hy 本を出すために技術書典への参加にあたり創設されたばかりの団体（もしくはサークル）です。本稿執筆時点での会員は tackman 一人です。Hy society への参加者は随時募集中。

Hy ショートイントロブック

2019 年 9 月 20 日　初版第 1 刷　発行

著　者　日本 Hy ユーザー会