


# 複数の言語からなる プロジェクトを 作るということ

#PyConJp 2016/09/21

@cocuh\_

github:cocuh/pyconjp2016-chimera

powered by index. 

# whoami

**Kosuke Kusano** (a.k.a. cocuh)

twitter: cocuh\_



**専門: 機械学習とプライバシー**

related to 差分プライバシー, k匿名性

using 離散数学, 確率論, 線形代数

劣モジュラとか深層学習とかもやってて専門迷子

**趣味: 技術の無駄使い(ゲテモノ好き)**

MacBookProにArchLinux

オタク機械学習

Pythonでワンライナー

# 闇Pythonista入門(Pythonワンライナーのテクニック集)

Python 3.3.4 7257

投稿を編集

cocuhが2014/04/06に投稿(2014/12/06に編集)・編集履歴(11)・問題がある投稿を報告する

192  
ストック

0  
コメント

14882  
Views

ストック



🕒 この記事は最終更新日から1年以上が経過しています。

## はじめに

世界には1行でプログラムを書くワンライナーという技巧的プログラミングの世界があります。

ワンライナーと言われる言語の多くはPerlやRubyなのですが、比較的委員長キャラのPythonでもワンライナーができます。

PEP8とZen of Pythonで綺麗になっている白Pythonの世界に

Pythonでも1行で書いたよ！楽しい！！

#!/usr/bin/env python

Tweet

14

0

Like 4

Pocket 33



cocuh

2263 Contribution

### 人気の投稿

- ・ [そこそこセキュアなlinuxサーバーを作る](#)
- ・ [闇Pythonista入門\(Pythonワンライナーのテクニック集\)](#)
- ・ [Cythonで連結リスト\(linked list\)をつくってみる](#)
- ・ [jupyter with pythonで信号解析するときには試聴したい](#)
- ・ [Pythonのリスト内包表現はdict,setでも使える](#)

# 注意

いろんな言語の話が出てくるので,  
Python原理主義の方ごめんなさい

もしかしたら: 上級者向け

間違いなどありましたら  
@cocuh\_ までご指摘お願いします.

## このトークのテーマ

複数のプログラミング言語を組み合わせる  
→「キメラ」と命名



## このトークのテーマ

複数のプログラミング言語を組み合わせる  
→「キメラ」と命名

## このトークの目的

- 「複数の言語を組み合わせること」に対し名前をつける
- ノウハウやツールが不足しているので先駆けとして
- この視点を共有し議論を活発化させる
- 私がキメラと戦って得た知見の共有(Python+C++/Rust)
- なぜPythonで書くのか, Pythonの長所短所は何かを考える

# おしながき

1. キメラの紹介と関連技術
3. CとPythonを組み合わせる方法3つ
4. 事例紹介(Rust+Python, etc...)
5. キメラをする上で考えるべきこと
6. 現状の問題点
7. キメラから見たPythonの今後

なぜプログラミング言語を  
組み合わせようとするか



プログラミング言語ごとに

言語仕様

開発/実行環境

文化/ライブラリ が異なり

それぞれの言語に**長所**/**短所**が存在するから

→ **適材適所**

→ **言語で役割分担**

たとえば？

# numpy: C+Python(+Fortran)

Pythonで使える行列演算ライブラリ

numpy / numpy

Watch 252

Star 3,320

Fork 1,681

Code

Issues 1,144

Pull requests 141

Wiki

Pulse

Graphs

Numpy main repository <http://www.numpy.org/>

● C 54.5%

● Python 44.9%

● Other 0.6%

# numpy: C+Python

巨大な行列計算を高速に計算させたい

## Python:

pros: 書きやすい

cons: 遅い

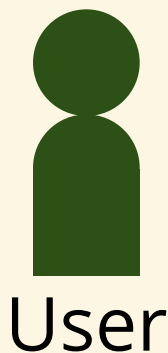
→ PythonへのAPI

## C言語:

pros: 速い

cons: 書きにくい

→ 処理部分はCで書く



Python

`y=W.dot(x)+b`

Pythonで簡単に書く

NumPy(Python)

```
class Matrix:
    def dot(self, x):
        return c_lang.dot(self, x)
```

```
class Vector:
    def __add__(self, b):
        return c_lang.add(b)
```

Cの構造体に変換  
Cの関数にbind

NumPy(C)

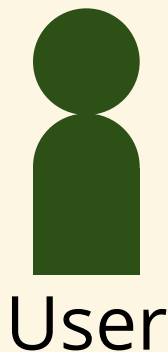
```
void dot(Matrix w, Vector x, Vector &y){
    for(int i = 0; i < m; i++) {
        y[i] = 0.;
        for(int j = 0; j < d; j++) {
            y[i] += w[i][j] * x[j];
        }
    }
}
```

Cで書かれた処理が  
走る

# numpy: C+Python

ユーザーは**Python**を書けばいいので**書きやすい**  
動いているコードは**C言語**で書いてあるから**速い**

→ 言語で役割分担



Python

`y=W.dot(x)+b`

Pythonで簡単に書く

NumPy(Python)

```
class Matrix:
    def dot(self, x):
        return c_lang.dot(self, x)

class Vector:
    def __add__(self, b):
        return c_lang.add(b)
```

Cの構造体に変換  
Cの関数にbind

NumPy(C)

```
void dot(Matrix w, Vector x, Vector &y){
    for(int i = 0; i < m; i++) {
        y[i] = 0.;
        for(int j = 0; j < d; j++) {
            y[i] += w[i][j] * x[j];
        }
    }
}
```

Cで書かれた処理が  
走る

キメラの目的

プログラミング言語の  
いいところ取りをしたい

**異なる言語を組み合わせる技術**  
**=language binding**

# 異なる言語を組み合わせる技術

- 基本的に手続き呼び出しに還元される  
=procedure call

手続き呼び出しの関連技術が転用可能である

## - Procedure Call

同一のホスト上で同一プロセス上で実行される  
FFI, ctypes(Python, OCaml)

本トークで主に取り上げる

## - Local Procedure Call(LPC)

同一のホスト上で異なるプロセス上で実行される  
D-Bus, RabbitMQ

## - Remote Procedure Call(RPC)

異なるホスト上で実行される  
CORBA, SOAP, xml-rpc,

※このあたりの定義は人/ライブラリによって違うので注意#



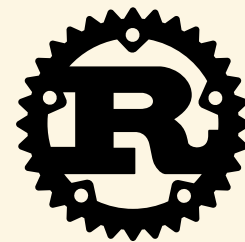
# 言語と言語を組み合わせる

- 基本的に**ググればだいたいある** binding, bridge, integrate, embedding, communication

[ 言語A 言語B binding ][検索]

基本的にbindingでググるとcffiでやるものが多い

Python + Rust → rust-cpython, python-rust-ext



Python + erlang → erlport

Python + Haskell → HaPy



# CとPythonを組み合わせる方法

Q.なぜC?

A.一番相性がいいから

# CとPythonを組み合わせる方法

**1. ctypes:**

**2. Cython:**

**3. C Extension**

# PythonからC/C++を呼び出す手法: ctypes

## ctypes:

- Pythonの標準ライブラリ
- C互換の関数呼び出し(cffi)
  - .dll/.soにできればだいたい使える
- 簡単に使えるが高度なことはできない
- build, pathの管理はしてくれない

資料: 「16.16. ctypes — Pythonのための外部関数ライブラリ」

```
import ctypes

libc = ctypes.CDLL('/usr/lib/libc.so.6')
libc.printf(b"Hello %s!\n", b"Youjo")
```

# Cython:C Extension for Python

- Python likeな構文で記述するとCを出力するライブラリ
- setup.pyに記述すればbuildまでしてくれる
- 実はIPythonでも叩ける
- 内部的にはC Extensionを生成している
- numpyをCythonを使って高速化

参考: Building Cython Codes

参考: Working with NumPy

```
cimport libc.stdio
```

```
def hello(name: str):  
    cdef bytes name_c = name.encode()  
    libc.stdio.printf(b"hello %s!\n", name_c)
```

```
from distutils.core import setup  
from Cython.Build import cythonize
```

```
setup(  
    name="hello",  
    ext_modules=cythonize("hello.pyx"),  
)
```



# PythonからC/C++を呼び出す手法: C Extension

## Python C Extension:

- CPythonの機能(たしか)
- かなりめんどくさいが高度なことができる
- Python2とPython3でかなり仕様が違う
- CPythonの指定する関数を.dll/.soに生やす
- setup.pyを叩くとbuildが走る

資料: 「C や C++ による Python の拡張」公式ドキュメント

資料: 「実践C拡張モジュール開発」pycon apac 2013のスライド

どのくらいめんどくさいかというと...

```
#include<Python.h>
```

```
static PyObject *  
hello(PyObject *self, PyObject *args)  
{  
    const char *name;  
    if (!PyArg_ParseTuple(args, "s", &name)){return NULL;}  
  
    printf("Hello %s!\n", name);  
    Py_RETURN_NONE;  
}
```

```
static PyMethodDef methods[] = {  
    {"hello", (PyCFunction)hello, METH_VARARGS, "hello function\n"},  
};
```

```
static struct PyModuleDef moduledef = {PyModuleDef_HEAD_INIT,"hello", NULL, 0, methods, NULL, NULL,
```

```
PyMODINIT_FUNC  
PyInit_hello(void)  
{  
    #if PY_MAJOR_VERSION >= 3  
        PyObject *module = PyModule_Create(&moduledef);  
    #else  
        PyObject *module = Py_initModule("hello", methods);  
    #endif  
    return module;  
}
```

...('A`)

# CとPythonを組み合わせる方法(まとめ)

## 1. ctypes:

簡単だけど低機能

## 2. Cython:

そこそこ手軽で高機能  
おすすめ

## 3. C Extension

物好き向け



# 私の事例紹介

# その1: Python + C++ with Cython

**目的:**既存のソルバーを拡張してPythonから使いたい  
(ex. SATソルバーを拡張してプライバシー評価で使いたい)

## 背景

あるリスクを評価することが充足可能性問題に帰着(ALLSAT)  
→既存のソルバーを使用したい

## 充足可能性問題

$$(a \vee \neg b) \wedge (b \vee c)$$

a: true, b:true, c:true

a: true, b:true, c:false

a: true, b:false, c:true

a:false, b:false, c:true

# その1: Python + C++ with Cython

**目的:**既存のソルバーを拡張してPythonから使いたい  
(ex. SATソルバーを拡張してプライバシー評価で使いたい)

## 背景

あるリスクを評価することが充足可能性問題に帰着(ALLSAT)  
→既存のソルバーを使用したい

### Python:

- これまでの実験コードがすべてPython
- Jupyter notebookで実験している

### C++:

- 既存のソルバーの実装はC++であることが多い
- 線形計画問題/充足可能性問題/
- 既存のソルバー(Minisat)の拡張が必要

**PythonからC++を呼び出せるようにしよう**

MiniSATSolver

solve()

既存

C++

所有

AllSATSolver

- solver: MiniSAT  
enumerate()

Cython

Python

```
solver = AllSATSolver()  
solver.enumerate()
```

# Cythonをつかう上で

- Cythonは基本Cが想定されているがC++も可能
  - ドキュメントも実装もあまり充実してない ref Cython libcpp
- Cythonにそれほど**多くのことを求めてはいけない**
  - **アロー演算子**がない
    - (\*ptr)するかcython.operator.dereferenceをつかう
  - テンプレートの引数に数字を渡せない
  - genericsが使えない
- **上手く行かない時が一番つらい**
  - Cython→cppだけでなくcpp→binaryもコケることある
    - 自動生成されたcppを読む, gdb
  - メモリーリーク

参考: 実験コードをCythonでゴリゴリ実装したら つらかった話

# Cythonで大きいプログラムを書くのはやめよう

C/C++で書きtestもC/C++で書く

CythonはInterface定義やPython型をstructに変換するだけにする

# その2:Python+Rust

**目的:**NP困難な最適化問題を解く  
(分枝限定法の並列実装)

## 背景

プライバシーを保護した機械学習予測器を設計することを最適化問題に帰着することを証明

### Python:

- GILがあるためmultithreadによる並列処理ができない
- subprocessはめんどくさい

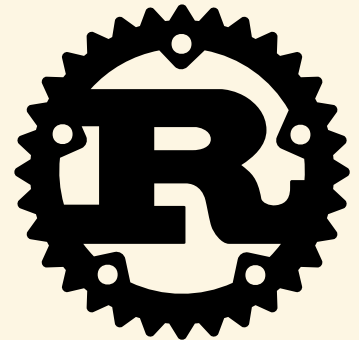
### Rust:

- 並列処理が書きやすい
- Cと疎遠ないぐらいの速さ
- 私が今一番好きな言語

# Rust

- **静的型付け**のコンパイラ言語(interpreter実装も一応ある)
- 非常に**高速**な言語(Cに匹敵)
- モダンな型システム**
  - 型的にthread safeであることを保証
  - 関数宣言に型を明示しなければならない(変数の型は省略可能)
- 基本heapでなくstackを使うように書く文化
  - 参照カウンタつきptrもあるstd::rc::Rc
- 並列処理**が得意
- C++が好きな人に向いてるイメージ

```
fn hello(name: &str) {  
    println!("Hello {}!", name)  
}  
fn main() {  
    hello("Youjo");  
}
```





## **dgrunwald/rust-cpython** ref: github

- dgrunwald氏作成
- rustとcpythonをくっつけるもの
- Python C Extensionで求められているmethodの生えた.soを出力
- rustからpythonを呼ぶのもpythonからrustを呼ぶのもできる
- 参照カウンタ管理はよしなにやってくれる

## **cocuh/python-rust-ext** ref: github

- cocuh作成
- python setup.py buildするとrustもbuildしてくれる

```
use std::thread;
use std::sync::{Arc, Mutex};
```

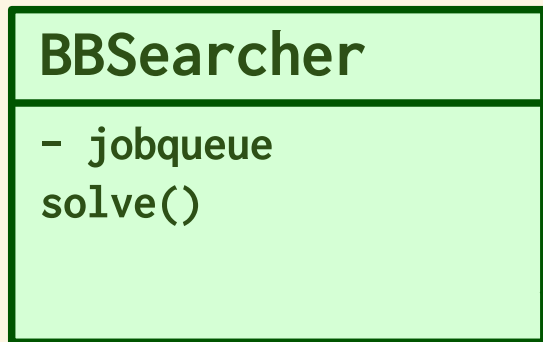
```
fn sleep_sort(py: Python, py_args: PyList) -> PyResult<PyList> {

    // convert Python object to Rust object
    let args: Vec<u32> = py_args.iter(py)
        .map(|x| x.extract(py))
        .filter(|x| x.is_ok())
        .map(|x| x.ok().unwrap())
        .collect::<Vec<_>>();

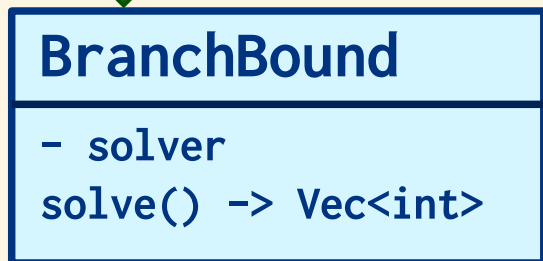
    // generate workers
    let result = Arc::new(Mutex::new(Vec::<u32>::new()));
    let workers = args.into_iter().map(|x| {
        let result = result.clone();
        thread::spawn(move || {
            thread::sleep_ms(x * 100);
            let mut result = result.lock().unwrap(); // Rust's COOL mutex!!
            result.push(x);
        })
    })
    .collect::<Vec<_>>();

    // join worker threads
    workers
        .into_iter()
        .map(|x| {
            x.join();
        })
        .collect::<Vec<_>>();

    // convert Rust object to Python object
    let res = result.lock().unwrap().to_py_object(py);
    Ok(res)
}
```



所有



Worker

Worker

Worker

Worker

Rust

Python

```
solver = BranchBound(~~~)
solver.solve()
```

# Rust+Pythonの懸念点

- Rust実行中にGILが開放されない(現時点のrust-cpython)
- linuxだと動くがMac OS Xだと動かない(windowsは知らない)
  - buildが通らない系: ld系だったが, 切り分けが難しかった
  - segfaる系: pyenvを使ってるときのlink先
    - virtualenvで今の所困ったことはない

```
[ ~/workspace/python-rust-ext/example ]
cocuh@cocuh-macmini% python
Python 3.5.1 (default, Apr 26 2016, 16:23:03)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import youjo.hello
Fatal Python error: take_gil: NULL tstate
zsh: segmentation fault  python
```

**キメラにおいて注意すべきこと**

# 設計において

## API/FFI設計

- どのような**手順**で呼び出すか
  - 引数は?状態は?
- どのように**serialize**するか
  - protobuf / json / csv / pickle / 独自
- なにで**通信**するか
  - zeromq/socket/shared memory
- 所有権

## 結合度

- 言語間でオブジェクトのやり取りは基本めんどい
- 言語間を跨ぐところは**結合度を低く**する

両方の言語の特徴を知る必要はあるが**基本は全く変わらない**

# 開発において

## build tool / package manager

- 言語それぞれがbuild toolを持っている
- 如何に**統合的に扱う**か
- どの言語がどの言語を管理するか
  - ex. PythonがCを管理する
    - setup.pyでCがbuildされる

## 開発環境

- 言語により**依存ライブラリ**が増える
- 環境構築のめんどくささ
- dockerによる環境の固定化

# 開発において

## テスト

- 実装か言語間の連携のバグかの切り分けが難しい
  - 環境が原因ということもありうる
  - 原因の切り分けにはtestを書くべき
- 言語ごとに機能が分割されているはず
- それぞれの言語においてテストを書く



# 保守において

## change log監視対象の増加

- 言語/依存ライブラリが増えるので追うのが大変
- 言語に依存しないライブラリが欲しくなる(cffiになる)

## 学習コスト

- 双方の言語をしらないと設計/実装できない
- 開発が分担されているならapiを定義して(ry
  - よくあるweb鯖/clientの図
  - それぞれのチームがそれぞれを保守

# 現状の問題点

議論しているひとがいない(と思ったらいらっしまった)

- 言語を組み合わせることに対し焦点をあて
- **開発ツール**や設計に関して議論している人
- **人柱不足**なので**ノウハウ不足**
- 開発コスト下げる指針もない

## 本当の意味でのglue言語の不在

- PythonはC/C++ friendlyだが**Rust friendly**ではない
  - **原因はsetup.py**周りの仕様が古代兵器だから
  - `setup(cmdclass={'install_lib': install_with_rust})`
- 異なる言語のコンポーネントを組み合わせることに  
特化した本当の意味でのglue言語が欲しい
  - pipeline言語

キメラなんて開発コスト高いし  
役に立たないよ

「（´Д`）」ヤレヤレ

人人  
— 同意 —  
> 同意 <  
Y^Y

- なぜ開発コストが高いのか
- どうすれば開発コストが減るのか

**なぜ我々はPythonを使うのか**

**Pythonは何が強い/弱いのか**

いまはまだないが必要なツールは何か  
Pythonに向いているツールとは何か

# 私なりのPythonの長所/短所

- + どの計算機にも入っている
  - + ライブラリの豊富さ(機械学習/セキュリティ/Web)
  - + 読みやすいコードを書く文化
  - + 付き合いやすいコミュニティ
  - Global Interpreter Lock
  - setup.pyの設計がモダンではない
  - Python2, Python3
  - 動的型言語とRuntimeError
  - 言語仕様が古風
  - ライブラリの代謝が進んでない(最終更新日が10年前のpypl package)
  - 遅い
- Pythonはbuild tool/glue言語...?

**Pythonは何に強いと思いますか？**

**人柱募集中！**