pythonで become データマエショリスト

宮本 丈@第5回 WACODE勉強会

2016/8/6

自己紹介

- 国立癌センターで技術職員やってます。
- python歴は1年ほど
- プログラミングは2年ちょっと
- ダメ出し大歓迎です

今日の内容

1. 言語の背景、対象、目的
2. 型について
3. 基本文法
4. パッケージングの作法
5. 実技

今日の内容

1. 言語の背景、対象、目的
2. 型について
3. 基本文法
4. パッケージングの作法
5. 実技

1. 言語の背景、対象、目的

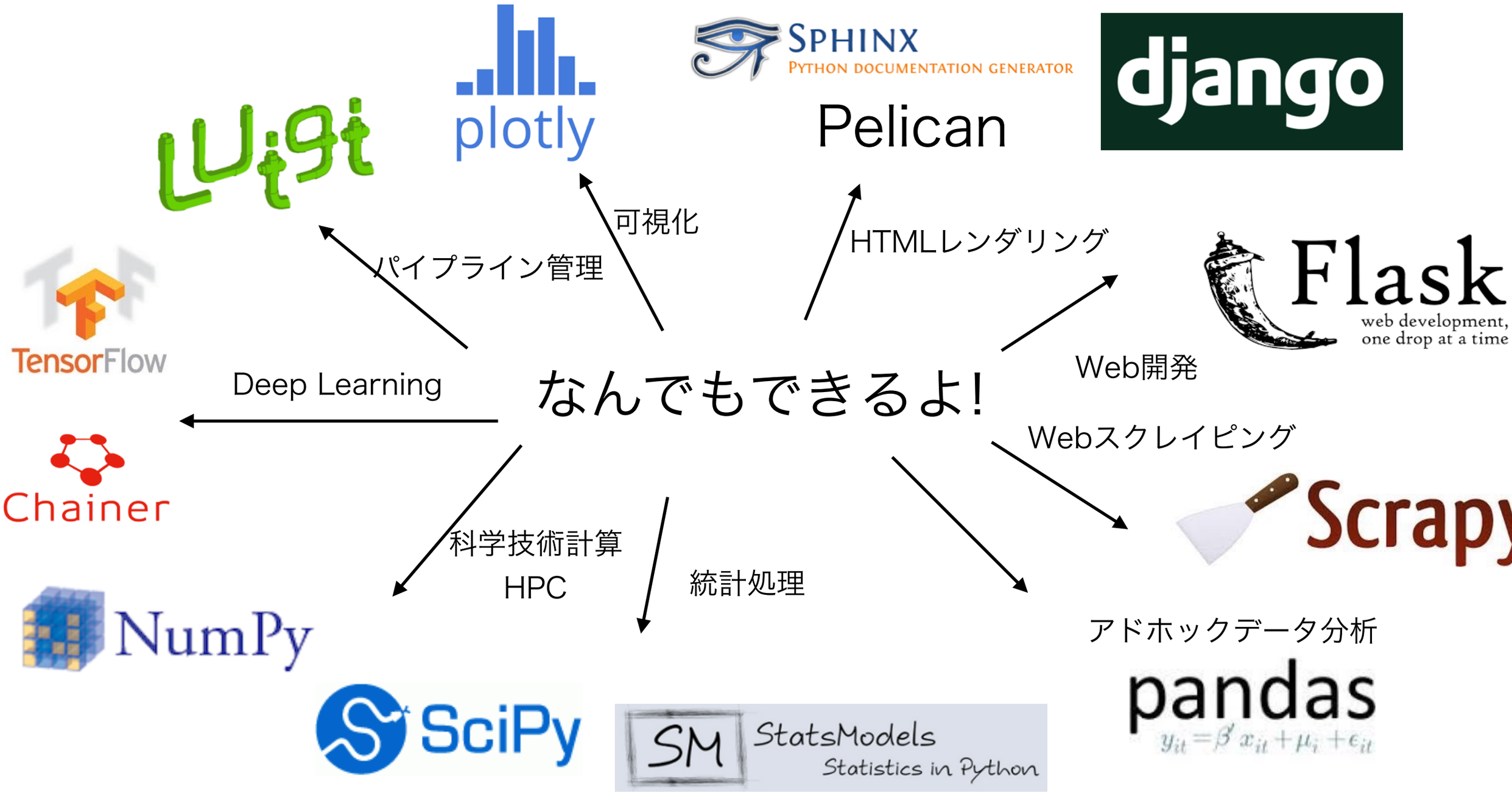
誰のための言語？

language for statistician

language for data scientist

language for child

pythonが得意なもの



pythonが苦手なもの

1. CPUバウンドの処理 -> numpy, Cythonでなんとかなる
2. 並列・並行処理 -> asyncio, multiprocessingでなんとかなる
3. 組み込み -> なんとかならない

Python is Glue language

python自体の優位性

・「ユーザーインターフェイス」が優れている

後発の言語の多くはpythonを踏襲している

ので長く使える知識

・ほとんどの場合に「正しいやり方」が用意されている

「the pythonic way」

・コミュニティがでかい

最初にならう言語として最適！

pythonの歴史とJulia登場の背景

Guido van Rossum (BDFL) が標準実装(Cpython)の開発開始	1989
google初のプロダクトをpythonで開発	1996

「pythonのパラドックス」	2004
Guidoがgoogleに移籍 (名実ともにGoogleの公用語に)	2005
python3発表	2008

pythonのパラドックス

最近の講演の中で、私は多くの人を怒らせるような発言をしてしまった。Javaを扱うプロジェクトよりもPythonを扱うプロジェクトの方が賢いプログラマを集められると言ったのだ。
Javaのプログラマがバカだと言いたかったのではない。Pythonのプログラマが賢いということなのだ。新たなプログラミング言語を学ぶには、やらなければならないことが山ほどある。Pythonを学ぶ人は、それで職を得ようとして学ぶのではない。本当にプログラミングが好きで、既に知っている言語では満足できないから学ぶのだ。
その気持ちが、彼らを企業が雇いたいと思うプログラマにさせるのだ。
そこで、もっといい名前があるかもしれないが、私は以下の論理を「Pythonのパラドックス」と呼ぶことにする。比較的難解な言語でソフトウェアを記述しようとする企業は、より良いプログラマを採用することができるだろう。学ぶことに意欲的な人材だけが、その企業を魅力的だと感じるからだ。そしてプログラマには、このパラドックスはさらに顕著に現れるだろう。職を得たいと思った時に学ぶべき言語は、普通の人が単に仕事を得るだけのために学ぼうとは思わないような言語だということだ。

ポール・グレアム著: 「ハッカーと画家」

pythonのパラドックス

最近の講演の中で、私は多くの人を怒らせるような発言をしてしまった。Javaを扱うプロジェクトよりもPythonを扱うプロジェクトの方が賢いプログラマを集められると言ったのだ。
Javaのプログラマがバカだと言いたかったのではない。Pythonのプログラマが賢いということなのだ。新たなプログラミング言語を学ぶには、やらなければならないことが山ほどある。Pythonを学ぶ人は、そこで、既に知っている言語では満足できないから学ぶのだ。
その気持ちが、彼らを企業が雇いたいと思うプログラマにさせるのだ。
そこで、もっといい名前があるかもしれないが、私は以下の論理を「Pythonのパラドックス」と呼ぶことにする。比較的難解な言語でソフトウェアを記述しようとする企業は、より良いプログラマを採用することができるだろう。学ぶことに意欲的な人材だけが、その企業を魅力的だと感じるからだ。そしてプログラマには、このパラドックスはさらに顕著に現れるだろう。職を得たいと思った時に学ぶべき言語は、普通の人が単に仕事を得るだけのために学ぼうとは思わないような言語だということだ。

要約: pythonistaはデキる奴が多い

注: 2004年時点の話です。

python3 … 後方互換性を捨てる

	2	3
prin文がstatementから関数に	print “hoge”	print(“hoge”)
イテレータ関係の関数に違い	xrange(iter), imap(iter)	range(iter), map(iter)
文字列の扱い	strとunicode	bytesとstrに統一
数値の扱い	int / int がintを返す	int / int がfloatを返す。

etc..

今日は3だけです。

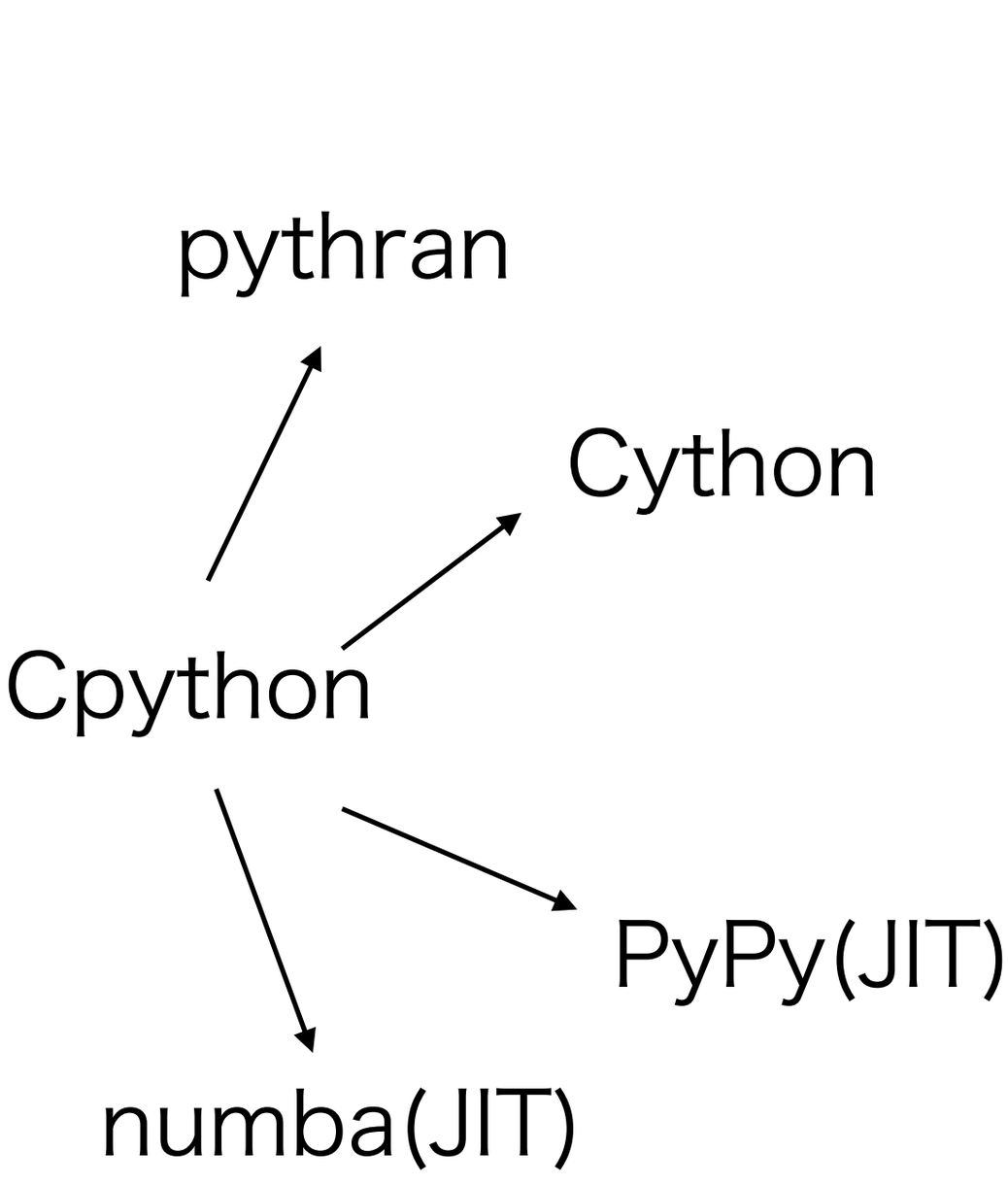
Global Interpreter Lock(GIL)

Cpythonでは同時にひとつのスレッド
しか実行できない。

マルチコアで処理を高速化したければ
インタプリタを別プロセスで起動(multiprocessing)し
それぞれでデータを共有させるしかない。

注: I/Oバウンドならマルチスレッドで高速化することもできます

処理系のカオス化



勝手に並列化されて
おそくなったのですが…

python3サポートしてない…

ドキュメントが不親切で
デバッグできない

コンパイラの気持ちとか
考えたくない…

numpyが動かない…

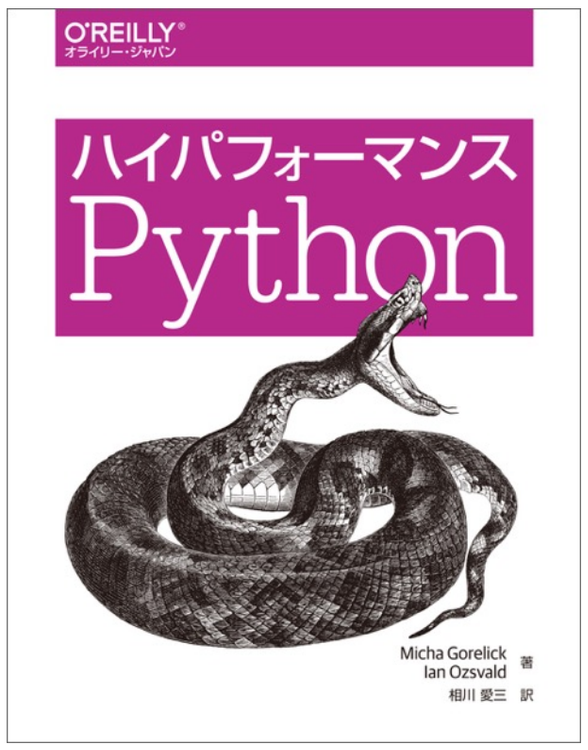
これもう別の言語では？

julia

なぜpythonで機械学習？

1. 豊富な高水準ライブラリ
2. numpy
3. Cython
4. multiprocessing

で99割は十分



実行速度を上げる前に
やるべきことがあるよね？

レポジトリ

デフォルトパッケージマネージャは
PyPIからダウンロード

githubから直接入れる場合は以下のコマンド

```
git clone <url>
python setup.py install
```

あるいは

```
pip install git+<url>
```

データサイエンス向けパッケージ全部入りPython

とりあえず入れておくと万能感に浸れる

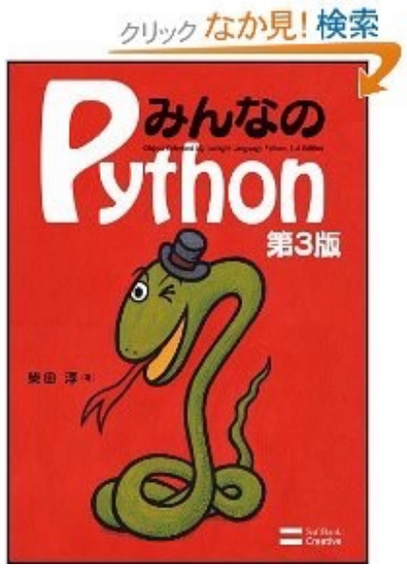
ダウンロードはこちら

https://www.continuum.io/downloads

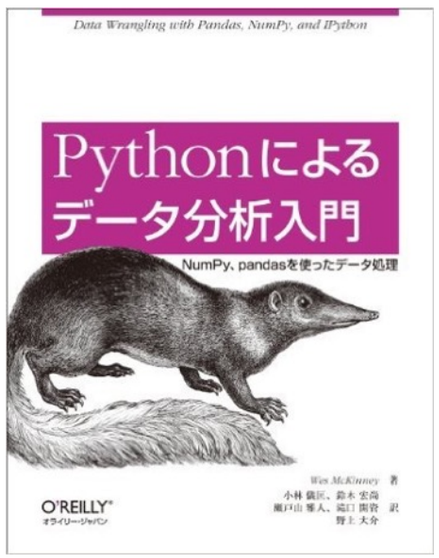
minicondaもあるよ！

参考図書

最初の1冊

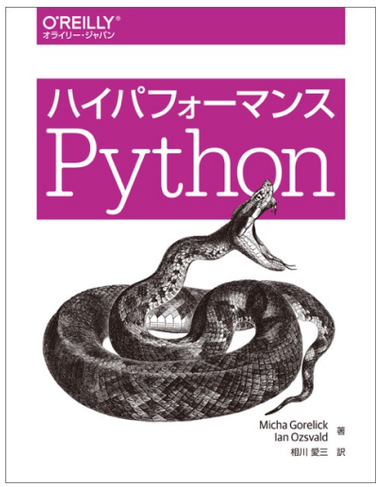


pandasとか

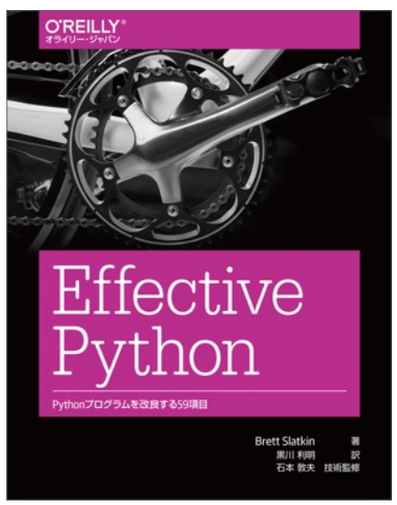


狭義のデータサイエンス

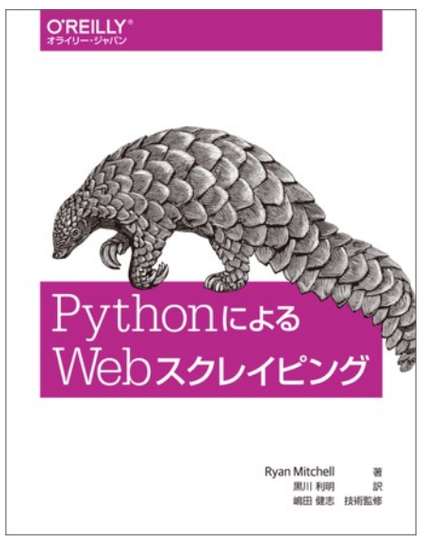
高速化の作法



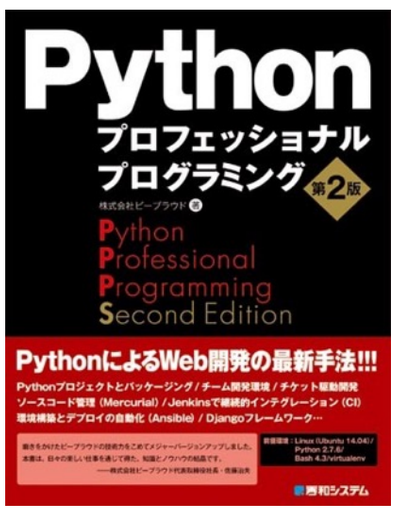
正しい書き方について



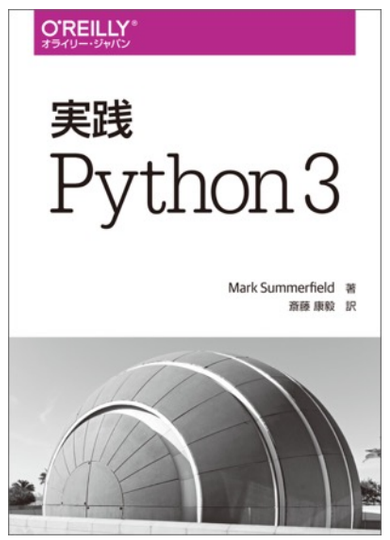
Webの仕組みを知りたい



Webアプリ作りたい



デザインパターン勉強したい



Dive into python3というサイトもチュートリアルとしてオススメ

統合開発環境

- pyCharm … web向けっぽい
- Jupyter … 言わずと知れた実行環境

ipython nbconvert —to script でスクリプトに変換できるので便利

- vim … いいぞ。

他にも色々あるけどvimとnotebookでいいんじゃないかな

今日の内容

1. 言語の背景、対象、目的

2. 型について

3. 基本文法

4. パッケージングの作法

5. 実技

2. データ型とデータ構造

基本データ型

動的型付けなので、覚えることは少ない

型名	リテラル	備考
文字列	r”hoge”, ‘fuga’	.joinや, .encodeなどの メソッドをよく使う
数値	1, 2.5, 3 + 4j	int, float, complexの3種
バイト列	b”hoge”	.decode(“utf-8”)で文字列に変換 外部プロセスとやり取りする際に用いる
論理値	True, False	Rと違って書き方に多様性はない
None	None	値がないことを示す

複合データ型(コンテナ型)

名前	内部実装	リテラル
リスト	ミュータブルな配列	[1, “hoge”, [2 , True]]
タプル	イミュータブルな配列	(1, “hoge”)
辞書(Dictionary)	ハッシュテーブル	{key: value}
集合型(set)	ハッシュ集合	{key}

複雑にネストした型を作る場合、辞書でも不可能ではないが
クラスを用いた方が保守性が高い

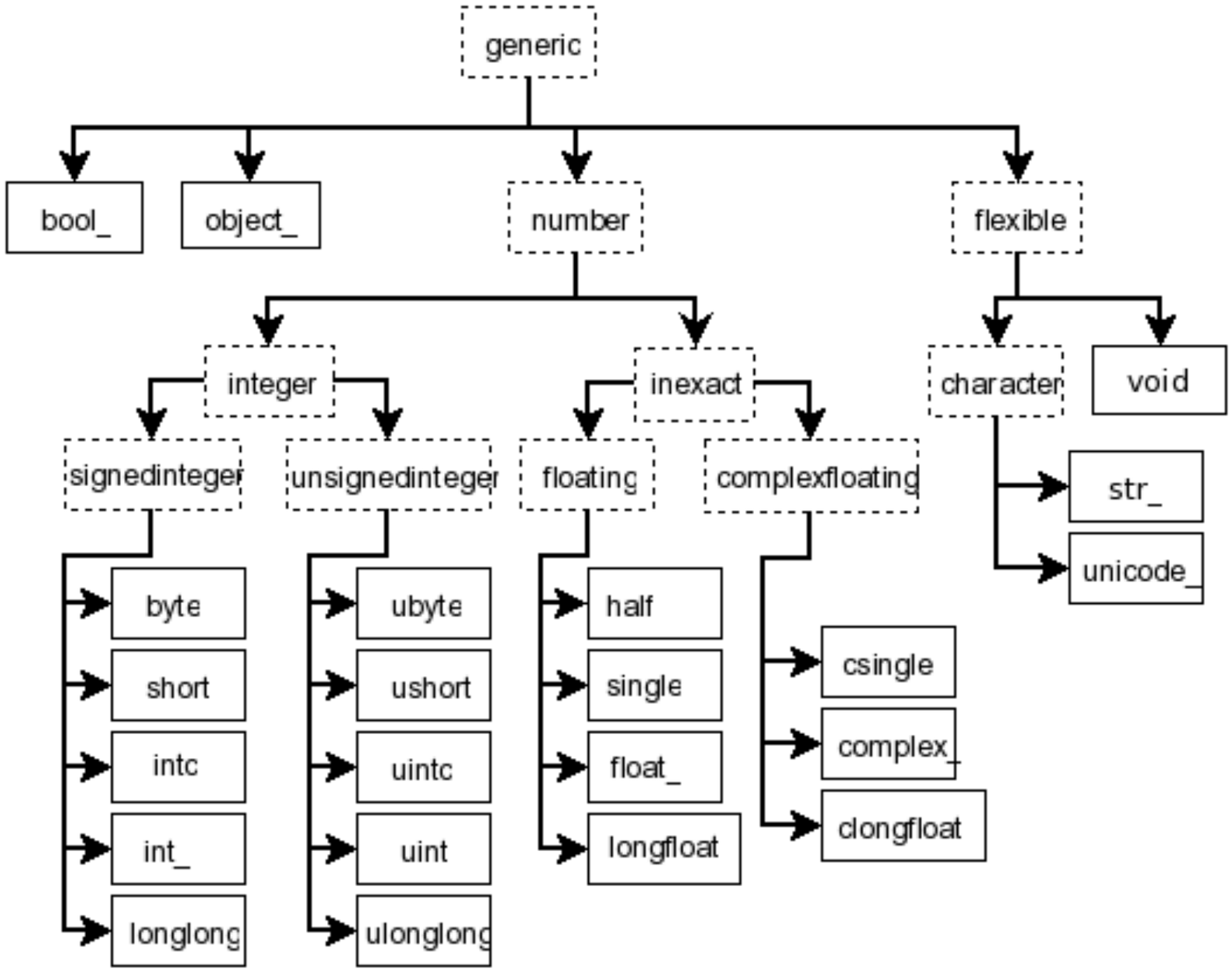
標準ライブラリの型

頻出のもののみ

名前	パッケージを含めた絶対名	使うタイミング
ヒープキュー	heapq.heapify	巨大なリストをソートしつつ扱いたい時
名前付きタプル	collections.namedtuple	タプルの可読性を上げたい時
デフォルト辞書	collections.defaultdict	辞書の値にデフォルトの型を指定したい時
キュー	multiprocessing.queue	プロセス間でデータをシェアしたい時

基本データ型

(numpy, scipy, pandas)



- 精度が固定
- 符号の有無がある

C互換性がある

pandasは因子型もある。

行列(numpy.ndarray)

行列中の型は統一されていなくてはならない

```
import numpy as np
int_array = np.array([1, 2, 3])
print(int_array.dtype) # 自動で型を決めてくれる
float_array = int_array.astype(np.float64)  # 浮動小数点型に変換
print(float_array.dtype)
x = np.array([[1,2,3], [4, 5, 6]], np.int32) # 型を指定した多次元配列
print(x)
```

```
int64
float64
[[1 2 3]
 [4 5 6]]
```

今日はnumpy, pandasは使いません

今日の内容

1. 言語の背景、対象、目的
2. 型について
3. 基本文法
4. パッケージングの作法
5. 実技

3. 基本文法

ipythonを起動

ipython notebook & … ノートブックを起動しブラウザからアクセス
　　jupyter notebook & … も大体一緒

ipython　　… コンソールからインタプリタを起動

引数なしのpythonコマンドでデフォルトインタプリタが起動するが、低機能なのでやめたほうがよい

Zen of python

```
import this
```

の出力を音読しましょう(任意)

条件分岐

```
if not "hoge"  == "fuga":
    print("hoge is not fuga !!")
else:
    print("hoge is fuga !!")
```

```
hoge is fuga !!
```

例外処理

```
mydict = {"key1": "value1", "key2": "value2"}
try:
    print(mydict["key3"])
except KeyError as e:
    print("there is no key3 so going to raise Error")
    raise
```

```
there is no key3 so going to raise Error
```

繰り返し文

for文の例

```
mydict = {} # 空の辞書を作成
for i, item in enumerate(["all", "you", "need", "is", "love"]): # iterableに対するindex付きループ
    mydict[item] = i

print(mydict)
```

```
{'is': 3, 'all': 0, 'need': 2, 'you': 1, 'love': 4}
```

リスト内包表記 (とても便利)

```
squared_odd_numbers = [ x**2 for x in range(1, 10) if x % 2 == 1]
print(squared_odd_numbers)
```

```
[1, 9, 25, 49, 81]
```

ジェネレータ内包や辞書内包も便利

繰り返し文

while文の例

```
import multiprocessing as mp
import queue
q = mp.Queue()
q.put("hoge")
q.put("fuga")

while True:
    try:
        val = q.get(timeout=1)
        print(val)
    except queue.Empty:
        break
```

```
hoge
fuga
```

プロデューサー
(キューに値を入れる関数)
と
ワーカー
(値を取る関数)

を分けて並行処理する際に使う

イテレータ

コンテナ型の要素を反復して参照するためのインターフェイス

特定ファイルの全行に対して何度も
イテレートするためのカスタム型

```
class FileContainer:
    def __init__(self, path):
        self.path = path

    def __iter__(self):
        with open(self.path) as fh
            for line in fh:
                yield line
```

イテレータプロトコル

__iter__

を実装している型(iterable)
はなんでもイテレータになりうる

標準ライブラリのitertools
を用いるとイテレータを結合したり
無限にイテレートしたりできて便利

関数

例

```
def my_funciton(myarg, mykwarg="world !"):
    """myargにkwargを結合して返す"""
    assert isinstance(myarg, str) # 型チェック
    return "¥t".join([myarg, mykwarg])

result = my_funciton("hello !")
print(result)
```

```
hello !¥tworld !
```

1. defで定義
2. snakecaseで書く

ディレクトリ操作

ipython上なら！ を行頭につけると
シェルコマンドが使えて便利

```
!pwd
```

```
/Users/miyamotojou/working/sandbox/python
```

スクリプト中ではosモジュールを使用する

```
import os
files_in_currentpath = os.listdir("./")
```

ファイル操作、システム操作

1行ずつ読み込んで逐次処理する例

```
with open("my_readonly_file.txt", "r") as fh: # 第二引数でモードを指定
    for line in fh: # 行ごとに逐次処理
        print(line)
```

CSVパッケージを使用して書き込む例

```
with open("outputfile.csv", "w") as fh:
    writer = csv.writer(fh, delimiter=",")
    writer.writerow([1, 2, 3])
```

ジェネレータ

```
def fib(n):
    a, b = 0, 1
    for _ in range(n): # ダミー変数にアンダーバーを使う慣習がある。
        yield a
        a, b = b, a + b

fib_generator = fib(10) # yieldを含む関数はgenerator objectを返す。
print(fib_generator)

val1 = next(fib_generator) # 値を一つずつ取得
val2 = next(fib_generator)
print("val1 is {} and val2 is {}".format(val1, val2))

print(list(fib_generator)) # 値をyieldした後、リストに変換
next(fib_generator) # 全てyieldしたので、StopIteration例外を生成する
```

```
<generator object fib at 0x106947d38>
val1 is 0 and val2 is 1
[1, 2, 3, 5, 8, 13, 21, 34]

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
```

・ 値を複数回にわたって返す関数

つまり

・ 実行を一時停止できる関数

つまり

・ 値を受け取らないコルーチン

デコレータ

関数を取り、関数を返す関数

```
def divtag_decorator(func):
    def wrapper(name):
        return "<div>{0}</div>".format(func(name))
    return wrapper

@divtag_decorator
def greeting(name):
    return "hi {} !! nice to meet you !".format(name)

greeting("Joe")
```

```
'<div>hi Joe !! nice to meet you !</div>'
```

実行時間を測ったり
型チェックしたり
といった使い方がある

詳しくはこの辺

http://qiita.com/_rdtr/items/d3bc1a8d4b7eb375c368

配列操作

```
mylist = [x for x in range(1, 20)]
first = mylist[0]
last = mylist[-1] # 最後の要素
odds = mylist[4:14:2] # 4 から14 まで2つごとに取得
odds_rev = odds[::-1] # 逆順にする

# 途中で改行してもOK
print("fitst is {} last is {} odds are {} and the one reversed is {} "
      .format(first, last, odds, odds_rev))

if 5 in mylist: # in演算子も使える
    print("there was 5")
```

```
fitst is 1 last is 19 odds are [5, 7, 9, 11, 13] and the one reversed is [13, 11, 9, 7, 5]
there was 5
```

文字列操作

文字列はイミュータブルなので、変更する操作は必ず
左辺で返り値を受け取る必要がある。

```
my_str = "草木も眠るウシミツ・アワー、荘厳なる鳥居は、壮絶な戦の 開始点と化す！"
my_str = my_str.replace("戦", "イクサ") # 文字列の置換
print(my_str)
my_str.endswith("!") # 最後の文字をチェック、Trueを返す
my_str.startswith("hoge") # 最初の文字をチェック、Falseを返す。
my_list = my_str.split("、")
print(my_list)

kusaki = my_str[0:2] # リストのようにスライシングもできる。
print(kusaki)
```

正規表現は標準ライブラリのreを用いる

utf-8で統一するためにシェバング以下にこう書いておくと良い

```
#!usr/bin/env python
# -*- coding: utf-8 -*-
```

オブジェクト指向（概念）

乱暴に要約すると
「インターフェイスを統一しましょう」
という話。

きちんと設計するのは難易度が高いが、
大規模なフレームワークならば必須。

大抵は関数とモジュールで十分なので
下手に手をだすのはやめておきましょう

オブジェクト指向（文法）

クラス(データ型に関数がくっついたもの)

```
class MyAbstractClass:
    def __init__(self, variable):
        self.variable = variable

    def __call__(self, arg):
        return NotImplemented # オーバーライドしてほしいメソッドはこうする。


class MyClass(MyAbstractClass):  # 継承の記法
    def __init__(self, variable, count):
        super().__init__(variable) # 親クラスをオーバーライド
        self.count = count

    def __call__(self, arg):
        self.count += 1
        return arg

    def _private_function(self, arg): # モジュールと同様、プライベートならばアンダーバーから
        return "hogehoge"  + arg


my_instance = MyClass("hoge", 0)
my_instance("fugafuga") # __call__が呼び出される
print(my_instance.__dict__) # 全フィールドを辞書形式で取得
```

```
{'variable': 'hoge', 'count': 1}
```

__init__が
デフォルトコンストラクタ

キャメルケースで書く

オブジェクトプロトコル

pythonはすでに様々なインターフェイス規約がある。
以下の様な場合のインスタンスの挙動を定義する

- __init__(self) … デフォルトコンストラクタ
- __add__(self) … + 演算子に渡した時
- __iter__(self) … イテレータとして評価した時
- __getattr__(self) … 未定義のメンバ変数にアクセスした時
- __call__(self) … インスタンスを関数として評価した時
- __repr__(self) … コンソールでの見え方
- __subclasshook__ … isinstance()で評価した時

etc …

デバッグに便利なTIPS

objという名のオブジェクトが合った時に

obj.__class__でオブジェクトのクラスがわかる
obj.__dict__ で属性の一覧
dir(obj)でメソッドと属性の一覧
obj?? でソースコードを見る(ipython)

可視化

matplotlib	探索的データ分析の時のデファクトスタンダード
Bokeh	D3.jsのラッパ。インタラクティブに可視化できる
Seaborn	matplotlibの見た目をブラッシュアップしたもの なのでpandasと相性がよい
ggplot2	文法がRのそれと似ているので慣れている方には便利
plotly	イケてる

matplotlibがお手軽だが、plotlyを使っておくと
他の言語に移行しても楽っぽいのでオトクかも

今日の内容

1. 言語の背景、対象、目的
2. 型について
3. 基本文法
4. パッケージングの作法
5. 実技

4. パッケージング

Rと比べて

ただのスクリプトとパッケージの差が少ない
従って覚えることが少なくて非常に楽
それだけに軽い気持ちで作り始めて泣きを見ることも

既存パッケージのインストール

pip install パッケージ名
conda install パッケージ名

condaはanaconda,miniconda付属の
パッケージマネージャ
コンパイル済みのものをゲットできるので楽

ライブラリのサーチパス

1. sys.pathにappend,insert

```
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
```

このファイル
の存在するディレクトリ
と”..”を結合
したものの絶対パス
をサーチパスの最初に追加

2. $PYTHONPATHをexport

3. ディレクトリを移動

のいずれかの方法で追加できる。

仮想環境の作成

pyvenv

python -m venv

conda create

pyenv-virtualenv

いろいろあって現状カオス

依存パッケージの正確なバージョン
が知りたかったら必須だが、そこまで神経質に
ならなくても良いのでは派です。

setup.py

メタデータを記述するスクリプト
setuptoolsとかdistutilsとか紆余曲折合ったが
今はsetuptoolsで落ち着いている

詳しくはgithubを見てください

docstring

ソースコード中の文字列が
そのままユーザリファレンス
となるすぐれもの。
モジュール・クラス・関数
に対して書く

1. 引数と返り値を書くこと
2. numpy styleとGoogle styleがある。
3. どちらを用いても良いが、統一すること。
4. ReST記法を使うと吉
5. 引数と、返り値を、書くこと。
6. 例も書くと吉

Google Styleの例

```
def add_2_numbers(arg1, arg2):
    """add 2 numbers for calculation afterwords

    long description if needed ...

    Args:
        arg1 : numeric type to be added
        arg2 : same as above

    Returns:
        float: result of addtion

    Examples:
        >>> add_2_numbers(1, 2)
        3.0
    """
    return float(arg1 + arg2)
```

doctestでお手軽TDD

モジュール最下部にこう書いておき、
スクリプトとして実行する。

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

先ほどの例のExamplesがそのままテストに!!

副作用のない関数を書く必要が出てくるので、設計にも好影響！

Sphinx、reST

reStructuredText

ちょっと高級なMarkdown
みたいなもの

ドキュメントをhtmlで
レンダリングしてくれる。
拡張モジュールでdocstring
も取り込める。

pip install sphinx && sphinx-quickstart
でテンプレートを作成してみよう！

unittest

JavaのJUnitっぽいテストスイート

testsディレクトリ以下に置くのが慣習

```
from .context import wiki_to_matrix

import unittest


class AdvancedTestSuite(unittest.TestCase):
    """Advanced test cases."""

    def test_crowl(self):
        wiki_to_matrix.crowl("http://url_for_test", "/tmp")


if __name__ == '__main__':
    unittest.main()
```

必須

python setup.py test

tests/以下のテストを実行

必須

python setup.py build

パッケージのビルド

egg-infoというpypi用のメタデータを作成する

必須

python setup.py install

site-packages以下にインストールする

Step.11 : 公開

- pypircに、tomlでメタデータを書く
- setup.cfgに配布に含めるドキュメントを指定
- python setup.py register -r <project_name>

で、審査待ちになるっぽいです。

今日の内容

1. 言語の背景、対象、目的
2. 型について
3. 基本文法
4. パッケージングの作法
5. 実技

5. 言語の特性を生かした解析例

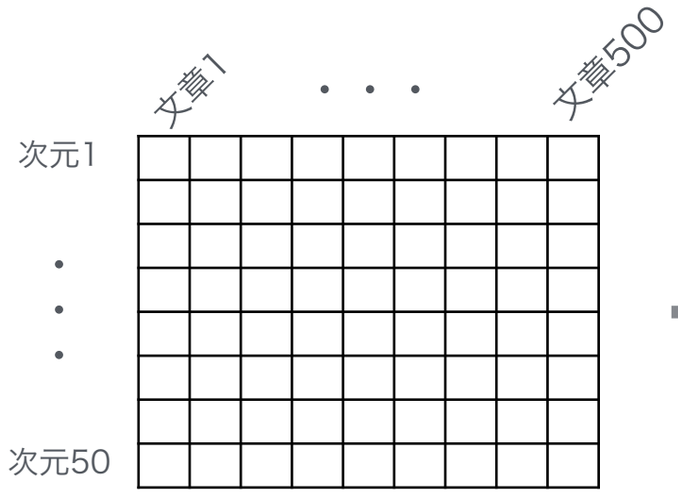
今日使うパッケージ

requests … httpでのやり取りを担当
lxml … xmlパーサ
BeautifulSoup … htmlパーサ。内部でlxmlを使用する
logger … 標準ライブラリ。ログ出力を行う

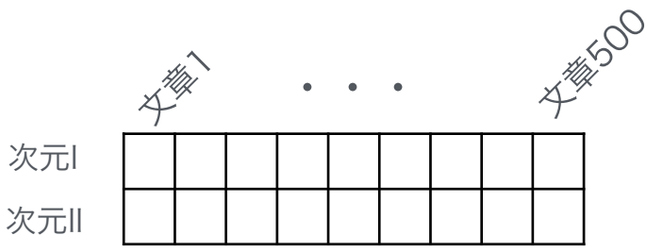
全体の流れ

Webスクレイピング (Wikipedia)

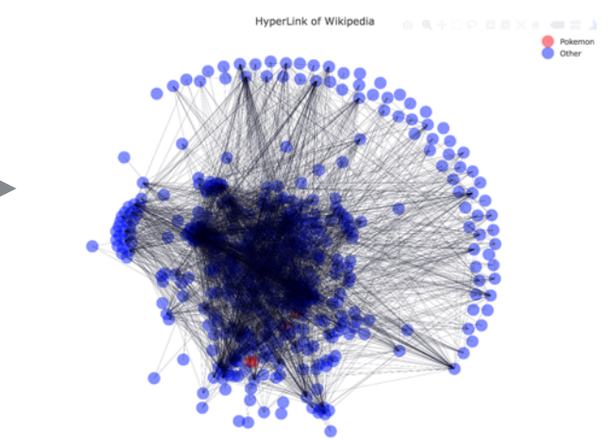
文章のベクトル化 (word2vec)



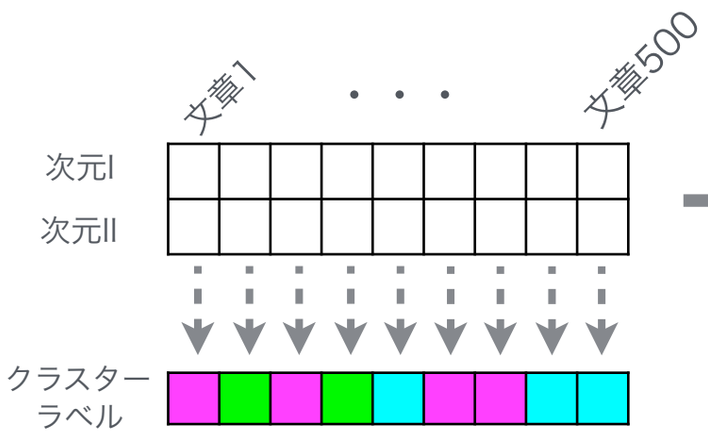
次元圧縮（t-SNE）



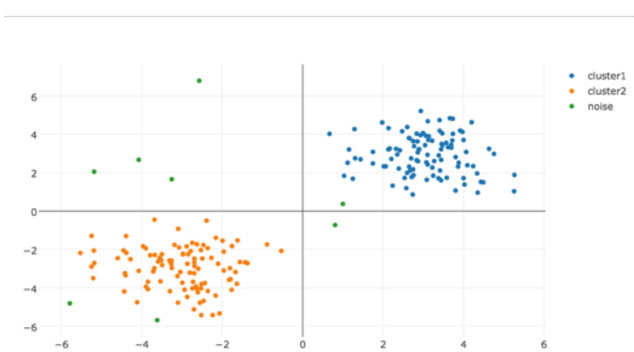
可視化 (plot, igraph)



アルゴリズム開発 (DBSCAN)



可視化（Plots.jl）

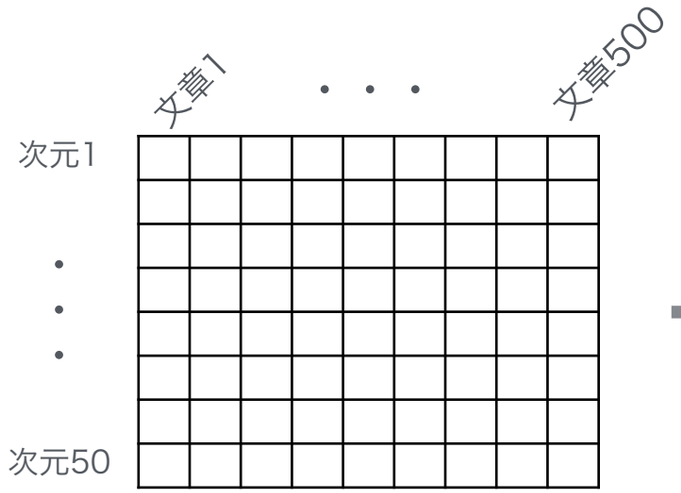


全体の流れ

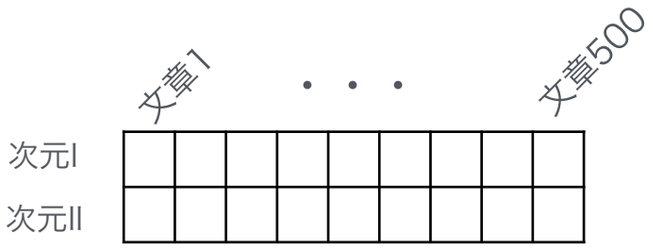
ここだけやります

Webスクレイピング (Wikipedia)

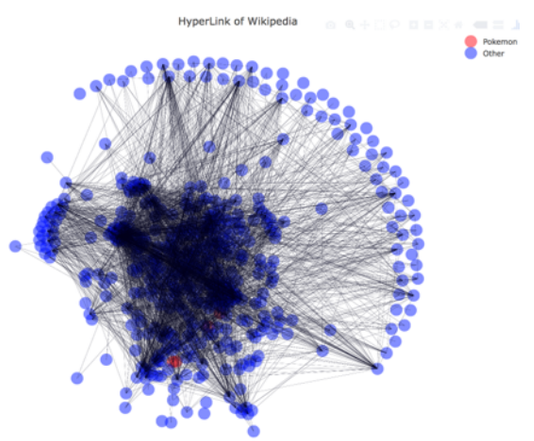
文章のベクトル化 (word2vec)



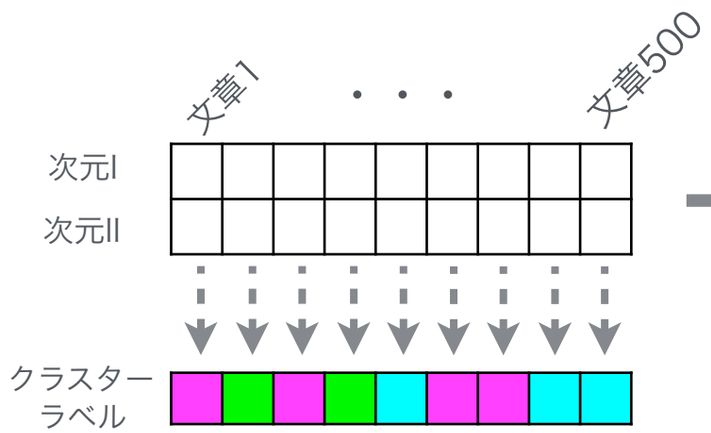
次元圧縮（t-SNE）



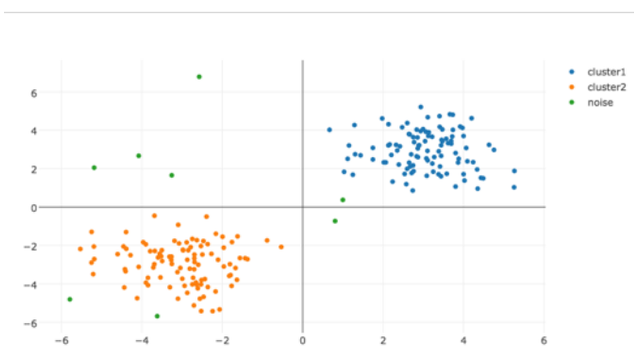
可視化 (plot, igraph)



アルゴリズム開発 (DBSCAN)



可視化（Plots.jl）



Python, Juliaと比較して（個人的な見解）

- 十徳ナイフのような言語
- **R**に比べて環境問題が面倒ではない
- データ分析の場合、OOPより関数型的な考え方の方が有用
- ググればたいてい情報が出てくる
- バイオはそんなに強くない（BioPythonは個人的にイマイチ）

解答例

リポジトリ内で
git checkout answer
すると解答例が見れます

https://github.com/wacode5/python1