Python による科学技術計算の概要

神嶌 敏弘
www.kamishima.net
2022-04-27 更新

このチュートリアルについて

プログラミング言語 Python

- Perl，Ruby，PHP と同様のスクリプト言語
  - 1991年2月に Guido van Rossum が0.9.0を発表
- 数値関連パッケージが充実しており，データ分析でよく使われている

チュートリアルの前提と目的

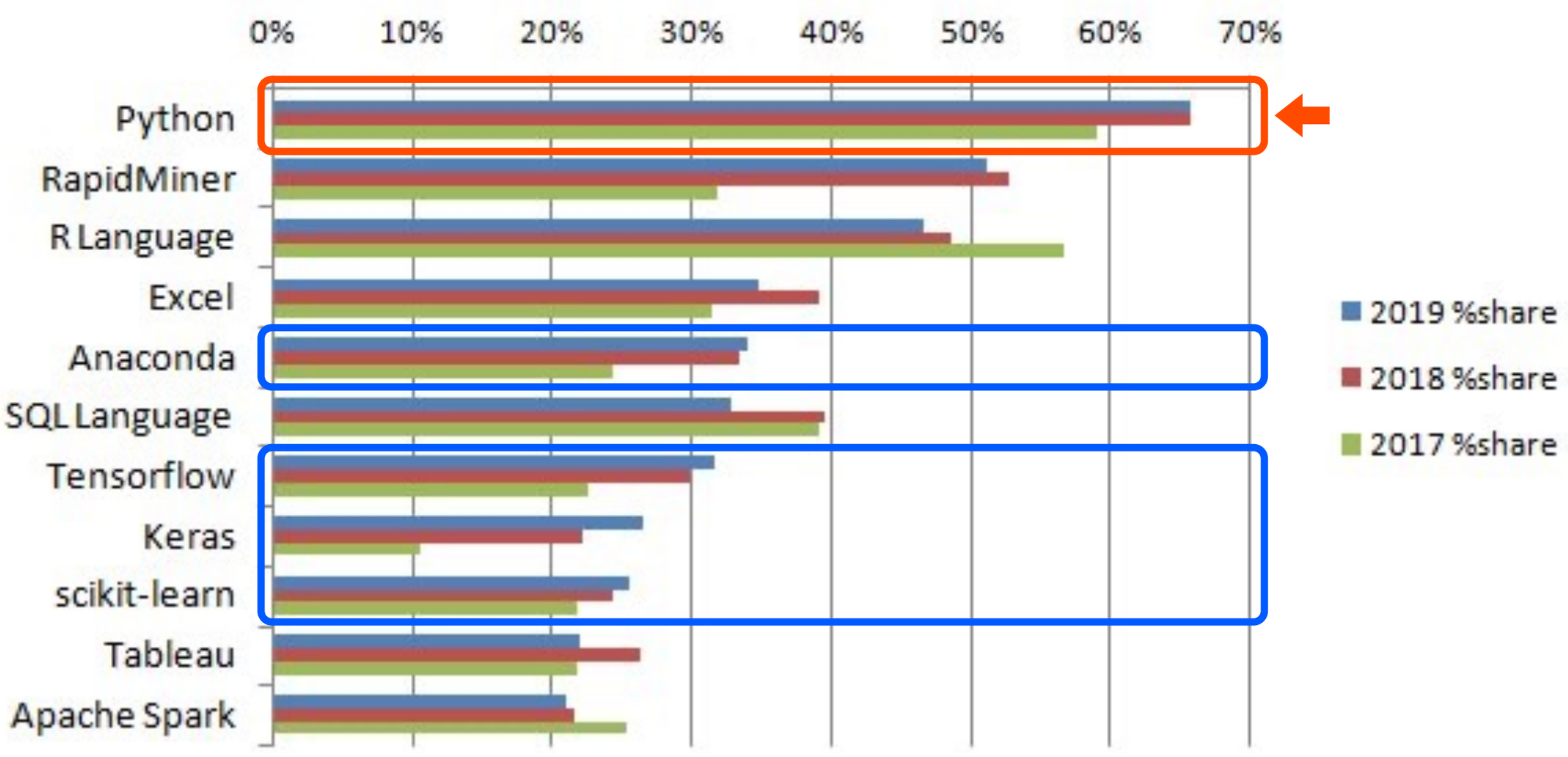
- 「Pythonの文法」と「統計・機械学習」の知識を前提
- ライブラリを用いたデータ分析用アルゴリズム実装の概要

チュートリアルのトピック

- 主要パッケージと実行環境
- 機械学習アルゴリズムの実装
- 関連パッケージと情報源

データ分析分野での Python

Kdnuggets による利用ソフトウェアの投票結果 2019年版
https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html



[参考] Top 10 Data Science Tools Over Time : https://youtu.be/pKPaHH7hnv8

Python はデータ分析ソフトとして重要な位置を占めている

データ分析における Python

他のデータ分析ツールと比較したときの長所と短所

長所

- インタープリタだが主要な計算はネイティブで高速に実行
- 普通のプログラミング言語 ➡ データ分析以外のAPIとの連携が容易，データ構造やメモリ管理の自由度が大きい，デバッガ・テストなどの機能が利用できる
- 数値計算系のライブラリや，他の数値計算ソフトへのライブラリが充実している

短所

- 普通のプログラミング言語 ➡ クラスなどプログラミングに関わる宣言の記述などが必要になる
- ライブラリは充実しているが R には及ばない

目次

第 I 部：主要パッケージと実行環境

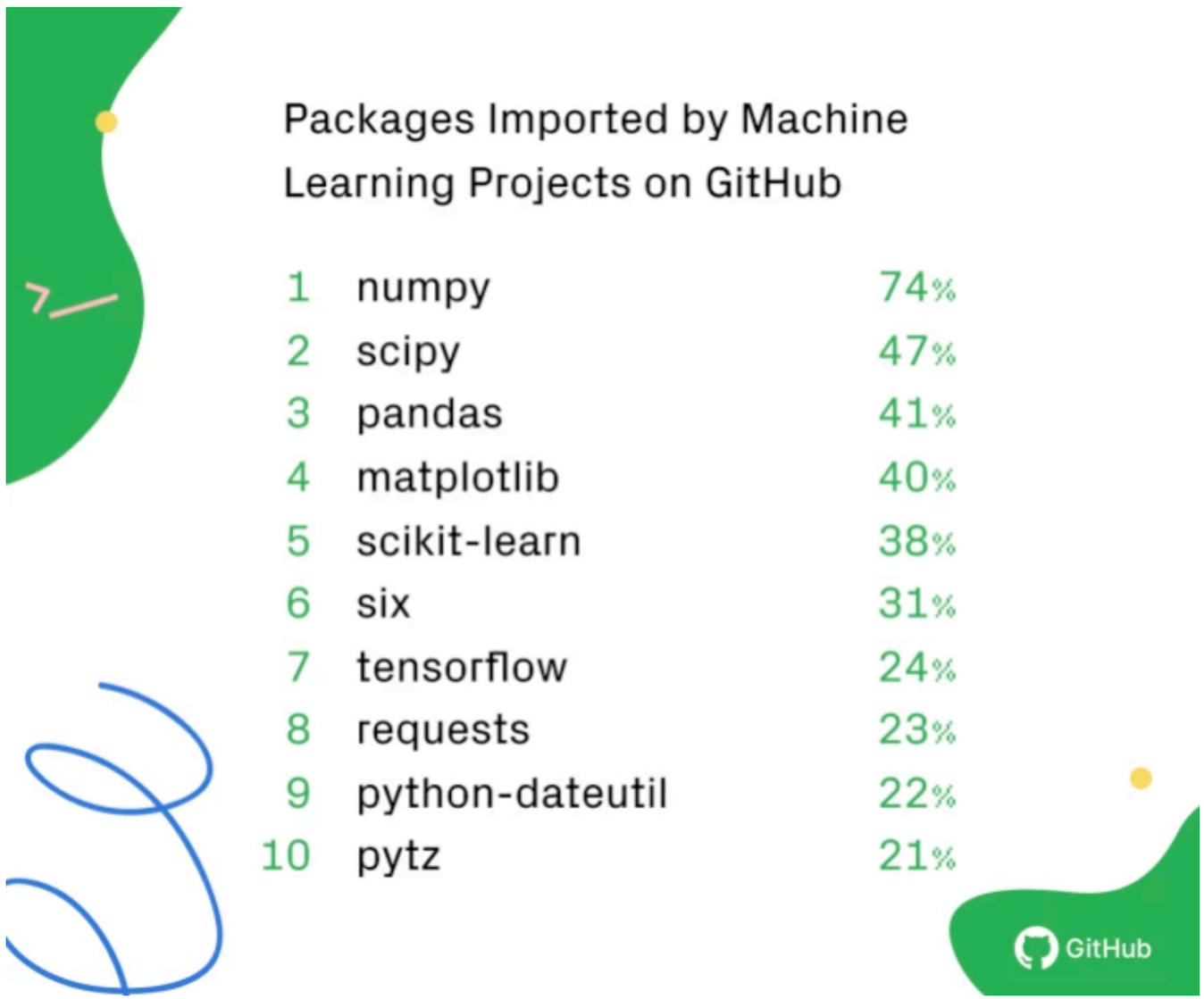
主要パッケージ

NumPy/SciPy と scikit-learn

利用されているPythonパッケージ

GitHub: Numpy and Scipy are the most popular packages for machine learning projects

2019–01–24



https://venturebeat.com/2019/01/24/github-numpy-and-scipy-are-the-most-popular-packages-for-machine-learning-projects/

NumPy

NumPy：多次元配列を効率よく，簡単に扱うためのライブラリ

- ホームページ： http://www.numpy.org/
- 内部的にはネイティブコードで実行されるので**演算は高速**
- **柔軟な要素の参照**（fancy indexing）
- **データの一括処理**（ユニバーサル関数，ブロードキャスティング）
  - 数値・論理演算，線形代数，初等関数，集約演算，乱数生成など
  - フーリエ変換や関数あてはめなどのやや高度な演算も可能
- **歴史**：Matrix Object (1994)，Numeric (1995)，Numarray (2001) などのプロジェクトを2005年に Travis Oliphant が統合して誕生
  - 詳しい歴史： http://www.slideshare.net/shoheihido/sci-pyhistory

SciPy

SciPy：NumPy より高度な数値演算処理

- ビルドに Fortran も用いているので，SciPy がインストールされていれば，NumPy のフーリエ変換なども高速になる
- **低次の演算**：物理定数，疎行列，特殊関数，確率分布など
- **高次の計算**：非線形最適化，補間，数値積分，計算幾何など
- **歴史**：2001年に複数のライブラリを統合して誕生

NumPy / SciPy を読み込むときの一般的な省略名

```
import numpy as np
import scipy as sp
```

scikit-learn

scikit-learn：非深層学習系の機械学習ライブラリの代表

- ホームページ： http://scikit-learn.org/stable/
- 多くのアルゴリズムを統一された API で利用可能

- **歴史**：2007に Google Summer of Code のプロジェクトとして始まり，2010年以降 INRIA のメンバーが加わって発展した

- **主要な機能**
  - **教師あり学習**：一般化線形モデル，SVM，アンサンブル学習など
  - **教師なし学習**：クラスタリング，異常検出，密度推定など
  - **モデル選択と評価**：交差確認，評価指標，超パラメータ探索など
  - **データ変換**：次元削減，標準化など
  - **その他**：ベンチマークデータ読み込み，テストデータ生成など

scikit-learn：線形回帰の例

線形回帰の利用例

サンプルファイル：linear_regression.ipynb

パッケージの読み込み

In [2]:
```
import numpy as np
from sklearn import linear_model
```

区間 [0, 10] と [0, 5] 上の一様分布に従う，2次元の独立変数のサンプルを100個生成

In [3]:
```
X = np.array([np.random.uniform(0, 10, 100), np.random.uniform(0, 5, 1
00)]).T
```

回帰式 $y = 1\,x_1 + 3\,x_2 + 10$で従属変数を生成

In [4]:
```
y = np.dot(np.array([[1, 3]]), X.T).ravel() + 10.0 + np.random.normal(
0, 0.1, 100)
```

線形回帰用のクラスを生成する；データに依存しないパラメータを指定

ここでは切片項を使う指定を行う

In [5]:
```
clr = linear_model.LinearRegression(fit_intercept=True)
```

scikit-learn：線形回帰の例

fitメソッドでデータをあてはめ

```
In [6]: clr.fit(X, y)
```

```
Out[6]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normaliz
        e=False)
```

従属変数の係数を調べると，ほぼ元の回帰式の 1 と 3 になっている

```
In [7]: clr.coef_
```

```
Out[7]: array([ 0.99932678,  2.9981098 ])
```

predictメソッドで推定したモデルに基づく予測

入力 $\mathbf{x} = (1, 2)$ に対する予測値は，ほぼ元の回帰式から得られる値 17 になっている

```
In [8]: clr.predict([[1, 2]])
```

```
Out[8]: array([ 16.99000072])
```

https://scikit-learn.org/stable/auto_examples/index.html
このExamplesの他に，User Guideにも多数のサンプル

第Ⅰ部：主要パッケージと実行環境

実行環境

環境構築とクラウド

実行環境の構築

Python バージョン3を使うこと

- Pythonバージョン2のサポートは2020年1月1日でサポート終了
- 2019年以降リリースのNumPyはバージョン3のみサポート

  https://github.com/numpy/numpy/blob/master/doc/neps/dropping-python2.7-proposal.rst

- **Python 2 の資産がある場合**：six などの移行ライブラリなどを使ったり，2to3などのツールを使ってPython3移行
- **Python 3**：統計量パッケージの標準化や行列演算子（3.5以降）やデータクラス（3.7以降）など数値計算に有用な機能

Python による科学技術計算の実行環境構築

- pip コマンドを使ってパッケージをインストールするのは問題も多かったが，最近は改善されてきた
- 各種のパッケージをまとめたインストーラは容易に使えるが，pip でインストールしたものと整合性を損なう場合もある

インストーラ

- フリーのものと商用のものがあるが，商用でもパッケージ数やサポートに制限のある無料版が提供されている
  - PyMC などインストーラによっては含まれないものも

フリー

- **apt / yum / port / brew**（Linux / Mac / BSD などUNIX系）

商用

- **Continuum Analytics Anaconda**
  - https://store.continuum.io/cshop/anaconda/
- **Enthought Canopy**
  - https://www.enthought.com/

その他 http://ibisforest.org/index.php?python%2Fnumpy を参照

第 II 部：機械学習アルゴリズムの実装

クラスの実装

scikit-learn形式のクラス と unittest

アルゴリズムのクラスによる実装

機械学習のアルゴリズムは，関数でも実装できるが，クラスを定義して実装すると以下の利点がある

- **Pythonの適性**：Python はオブジェクト指向型言語なので，クラスによる実装に適している
- **クラスの継承 や Mixin の利用**：一部だけが異なる学習アルゴリズムを容易に実装できる
- **学習結果の保存**：cPickle といったオブジェクトのシリアライズを利用して，学習したモデルのオブジェクトを保存可能
- **scikit-learnとの連携**：scikit-learn のクラスの作成規則に沿って実装することで，その機能を利用できる

scikit-learnのAPI仕様

- scikit-learn のクラス設計の基本仕様
  - Coding Guidelines：http://scikit-learn.org/stable/developers/
- **コンストラクタ**：クラスの初期化
  - 引数：データに依存しないアルゴリズムのパラメータ
- **fit()メソッド**：あてはめ・学習
  - 引数：訓練データと，データに依存したパラメータ
- **predict()メソッド**：学習済みデータを用いた予測
  - 引数：予測対象の新規の入力データ
- **score()メソッド**：モデルのデータへのあてはめの良さの評価
  - 評価対象のデータを，このメソッドの引数で指定する.
- **transform()メソッド**：次元削減などのデータ変換

MixIn を用いたクラス

Mixin（メソッドのみのクラス）による一部だけが異なる手法の実装

LogFuncMixin
object
funcメソッドは
対数関数

LinearFuncMixin
object
funcメソッドは線
形関数

FuncFitMixin
object
funcメソッドで，あてはめを
するfitと予測をするpredict

LogFit
BaseEstimator, RegressorMixin,
LogFuncMixin, FuncFitMixin
対数関数でのあてはめ

LinearFit
BaseEstimator, RegressorMixin,
LinerFuncMixin, FuncFitMixin
線形関数でのあてはめ

- 緑色は親クラス
- BaseEstimator と RegressorMixin は scikit-learnのクラス
- **必要な機能を組み合わせて部分的に異なるクラスを容易に実装可能**

Mixin を用いたクラス

サンプルファイル：class_mixin.ipynb

```
class LogFuncMixin(object):
    def func(self, x, a, b):
        return a * np.log(x) + b
```

対数関数のMixin

```
class LinearFuncMixin(object):
    def func(self, x, a, b):
        return a * x + b
```

線形関数のMixin

```
class FuncFitMixin(object):
    def fit(self, x, y):
        popt, pcov = curve_fit(self.func, x, y)
        self.coef_ = popt

    def predict(self, x):
        return self.func(x, *self.coef_)
```

あてはめと予測のMixin

```
class LogFit(BaseEstimator, RegressorMixin,
             LogFuncMixin, FuncFitMixin):
    pass
```

対数関数へのあてはめ

```
class LinearFit(BaseEstimator, RegressorMixin,
                LinearFuncMixin, FuncFitMixin):
    pass
```

線形関数へのあてはめ

scikit-learnとの連携

- scikit-learn のクラスの規則に従うことで，その機能を利用する
  - fit() や predict() などのメソッドの規則を踏襲する
  - sklearn.base.BaseEstimator クラスを親とし，score() メソッドを含む sklearn.base.ClassifierMixin なども継承する
- **交差確認による評価や，超パラメータ最適化が容易になる**

サンプルファイル：cross_validation.ipynb

IRISデータのロードと SVM 分類器の作成

In [5]:
```
iris = datasets.load_iris()
clf = svm.SVC(kernel='linear', C=1)
```

交差確認による汎化誤差の評価. `n_jobs=-1` とすると全CPUを使って並列計算

In [6]:
```
scores = model_selection.cross_val_score(clf, iris.data, iris.target,
cv=5, n_jobs=5)
np.mean(scores)
```

Out[6]: 0.9800000000000001

自動テストを書こう

- **unittest**：プログラムの最小単位ごとに，その入出力を検証する
  - 厳密に最小単位でなくても，リファクタリングや効率化で書き換えたときに自動で検証できるようになる
  - **一度信頼できる結果を得たときに，その結果と一致するかどうかだけの簡単なものでも有用なので，テストを書くことを薦める**
- Python の場合は nosetests というコマンドによりテストを実行できる
  - パッケージ全体について，全てのテストをまとめて実行できる

```
% nosetests -v test_svc.py
test_fit (test_svc.Test_SVC) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.742s

OK
```

unittestには何を書いたらいいのか？

- 本来は，エラー処理を含めて全てのコードが実行されるように書く
  - ➡ あくまで理想，変更が頻繁な実験コードなどでは無理がある

外部APIの仕様確認

- 機械学習のアルゴリズムを実装し，クラスを作って，fit() メソッドで学習し，predict() メソッドで予測する場合
- **小さなテストデータを読み込んで，学習後の損失関数値やパラメータ値，また，予測結果が一致するかを検証**
  - デバッグのときに書いたコードに，実行結果を加えて簡単に作っておく
- 特徴や目的関数の値が全て同じなどの特殊な境界条件のデータも追加で検証しておくと安全

外部APIの動作検証

サンプルファイル：test_svc.py

```
class Test_SVC(unittest.TestCase):

    def test_fit(self):

        # 小さなテストデータから，テストする分類器でモデルを学習
        from sklearn import svm
        from sklearn import datasets
        iris = datasets.load_iris()
        clf = svm.SVC(kernel='linear', C=1)
        clf.fit(iris.data[50:150, :], iris.target[50:150])

        # 係数が一致しているかを検証
        # 実数値は厳密な一致ではなく3桁か5桁ぐらいの精度で一致を検証
        np.testing.assert_allclose(clf.coef_,
            [[-0.59549776, -0.9739003 ,  2.03099958,  2.00630267]],
            rtol=1e-5)

        # 分類結果が一致しているかを検証
        # クラスのような離散値は厳密な一致を検証
        classes = clf.predict(iris.data[[50, 51, 100, 101], :])
        np.testing.assert_array_equal(classes, [1, 1, 2, 2])
```

unittestと乱数

- 乱数で初期化するアルゴリズムでは，実行するたびに結果が変わるため結果が一致するかどうかを検証できない
  - **乱数のシードを与え，クラス専用の疑似乱数生成器を作成する**
  - scikit-learn のユーティリティ関数 check_random_state() が便利

サンプルファイル：random_state.py

```
class RandomStateSample(BaseEstimator, TransformerMixin):

    # シードの初期化を行わない None が random_state のデフォルト値
    # unittest や再現性が必要な実験では適当な整数値を与える
    def __init__(self, random_state=None):
        self.random_state = random_state

    # fit メソッドでは乱数生成器を獲得
    def fit(self, X=None, y=None):
        self._rng = check_random_state(random_state)
        return self._rng.randn(10)

m = RandomStateSample(random_state=1234)
print("seed = 1234 @ init\n", m.fit())
```

unittestには何を書いたらいいのか？

内部メソッドの検証

- 最適化する目的関数（損失関数）やその勾配関数などの値を検証
- 単純な別コードや手計算で求めた結果と比較できる数10個の非常に小さなデータを使って，計算結果を信頼できるように
- 確率なら総和が 1 になるなど，結果が当然満たすべき条件も有用
- パラメータの値が全て 0 や全て 1，関数値が 0 になる点など，境界条件や特異値を検証コードに入れておく
- パラメータや潜在変数の初期値は，やはり境界条件や一様といった特別な条件で試しておく

境界条件の例

- pLSA アルゴリズムは，EMアルゴリズムという交互最適化法の一種を用いて解を求める
  - 教科書などでは初期値について，あまり記述はないが一様な値で初期化すると動かない
  - データが偶然に一様だったりすると動かないといった場合も

パラメータを一様に初期化

一様な定数

```
n = 100
k = 2
d = gen_data(n)
# print(d)
pXgZ = np.tile(1 / n, (n, k))
pYgZ = np.tile(1 / n, (n, k))
pZ = np.tile(1 / k, k)
for i in range(5):
    pXgZ, pYgZ, pZ = pLSAstep(d, pXgZ, pYgZ, pZ)
    print(log_loss(d, pXgZ, pYgZ, pZ))
```

```
-44014.18824
-44014.18824
-44014.18824
-44014.18824
-44014.18824
```

対数尤度は不変

サンプルファイル：pLSA_initialization.ipynb

パラメータを乱数で初期化

Dirichlet分布に従う乱数

```
n = 10
k = 2
d = gen_data(n)
# print(d)
pXgZ = np.random.dirichlet(alpha=np.repeat(1 / n, n), size=k).T
pYgZ = np.random.dirichlet(alpha=np.repeat(1 / n, n), size=k).T
pZ = np.random.dirichlet(alpha=np.repeat(1 / k, k))
for i in range(5):
    pXgZ, pYgZ, pZ = pLSAstep(d, pXgZ, pYgZ, pZ)
    print(log_loss(d, pXgZ, pYgZ, pZ))
```

```
-209.233267896
-202.685252472
-198.022879607
-192.019675469
-186.921832146
```

対数尤度は増加

第II部：機械学習アルゴリズムの実装

数値計算tips

NumPyの基本

NumPy 配列と多重リスト配列の違い

np.ndarray：N-d Array すなわち，N次元配列を扱うためのクラス

1. **メモリ上での保持**
   - **多重リスト**：リンクでセルを結合した形式でメモリ上に保持
     - 多重リストは動的に変更可能
   - **np.ndarray**：メモリの連続領域上に保持
     - 形状変更には全体の削除・再生成が必要.
2. **要素の型**
   - **多重リスト**：リスト内でその要素の型が異なることが許される
   - **np.ndarray**：基本的に全て同じ型の要素

NumPy 配列と多重リスト配列の違い

3. **配列の形状**
   - **多重リスト**：次元ごとに要素数が違ってもよい
   - **np.ndarray**：各次元ごとの要素数が等しい
     - 行ごとに列数が異なるような2次元配列などは扱えない

4. **高度で高速な演算操作**
   - **np.ndarray**：行や列を対象とした多くの高度な数学的操作を，多重リストより容易かつ高速に適用可能 (**fancy indexing**)
     - 配列中の全要素，もしくは一部の要素に対してまとめて演算や関数を適用することで，高速な処理が可能 (**ブロードキャスト**，**ユニバーサル関数**)

NumPy 配列の生成

np.array を使った生成

`np.array(object, dtype=None)`

- object には，配列の内容を，array_like という型で与える
  - array_like型：配列を np.ndarray の他，（多重）リストや（多重）タプルで表現したもの
- 要素が 1, 2, 3 である長さ 3 のベクトルの例

```
In [10]: a = np.array([1, 2, 3])
In [11]: a
Out[11]: array([1, 2, 3])
```

- タプルを使った表現も可能です

```
In [12]: a = np.array((10, 20, 30))
In [13]: a
Out[13]: array([10, 20, 30])
```

NumPy 配列の生成

- 2重にネストしたリストで表した配列の例

```
In [10]: a = np.array([[1.5, 0], [0, 3.0]])
In [11]: a
Out[11]:
array([[ 1.5,  0. ],
       [ 0. ,  3. ]])
```

- リストの要素に np.ndarray タプルを含むことも可能

```
In [12]: a = np.array([1.0, 2.0, 3.0])
In [13]: b = np.array([a, (10, 20, 30)])
In [14]: b
Out[14]:
array([[  1.,   2.,   3.],
       [ 10.,  20.,  30.]])
```

0行列と1行列

0行列（要素が全て 0 の配列）の生成

```
np.zeros(shape, dtype=None)
```

1行列（要素が全て 1 の配列）の生成

```
np.ones(shape, dtype=None)
```

- shape：スカラーや，タプルによって配列の各次元の長さを表現
- 長さが 3 の 0 ベクトルの例

```
In [10]: np.zeros(3)
Out[10]: array([ 0.,  0.,  0.])
```

- 3×4の1行列の例（引数をタプルにすることを忘れないように）

```
In [11]: np.ones((3, 4))
Out[11]:
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

初期化なし配列

要素の初期化なしで指定した大きさの配列を生成

```
np.empty(shape, dtype=None)
```

- 配列の生成後，その内容をすぐ後で書き換える場合には，配列の要素を 0 や 1 で初期化するのは無駄 ➡ メモリだけ確保・初期化しない

単位行列

単位行列を生成

```
np.identity(n, dtype=None)
```

- n は行列の大きさを表す
- 例：4 と指定＝大きさ4×4の行列（単位行列は正方行列なので）

```
In [10]: np.identity(4)
Out[10]:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

NumPy 配列の属性と要素の参照

np.ndarray クラスの主な属性

`class np.ndarray`

- **`dtype`**：配列要素の型
  - 後ほど，詳細を述べる
- **`ndim`**：配列の次元数
  - ベクトルでは 1 に，配列では 2
- **`shape`**：配列の各次元の大きさ＝配列の形状
  - 各次元ごとの配列の大きさをまとめたタプルで指定
  - 例：長さが 5 のベクトルは `(5,)`

    ※ Python のタプルは1要素のときは “`,`” が必要
  - 例：2×3行列では `(2, 3)`

NumPy の dtype

- **Python のビルトイン型に対応する以下の型はそのまま用いて大きな問題はない**

  真理値型＝ `bool`，整数型＝ `int`，浮動小数点型＝ `float`，

  複素数型＝ `complex`

- これらの型に対する正式な NumPy の型は，`np.` を前に付けるだけ

  `np.bool`, `np.int`, `np.float`, `np.complex`

※ さらに最後に “_” を最後につけた `np.int_` などが，Pure Python と厳密には互換の型

- **メモリのビット数を明示的に表す型**：`np.int32` や `np.float64` が定義されていて，次の目的で利用する
  - メモリを特に節約したい場合（10までなら8bitで十分）
  - C や Fortran で書いた関数とリンクする
    （32bit・64bitのどちらの環境で実行しても暴走しない）

NumPy の dtype（文字列）

- ビルトイン型の `str` / `unicode` と NumPy の`dtype` の相違点
  - np.ndarray の要素の大きさが同じである必要 ➡ 文字列は固定長
- NumPyでの文字列型：NumPy の型を返す関数 `np.dtype()` を利用
  - `np.dtype('S<the length of string>')`
- 例：最大長が16である文字列

```
np.dtype("S16")
```

- Unicode文字列の場合は，この S が U に置き換わります

```
np.dtype("U16")
```

※ Python3 の文字列は Unicode型 として指定

配列のdtypeの指定方法

- np.array() などの配列生成関数の dtype 引数で指定する方法

```
In [10]: a = np.array([1, 2, 3])
In [11]: a.dtype
Out[11]: dtype('int64')
In [12]: a = np.array([1, 2, 3], dtype=float)
In [13]: a.dtype
Out[13]: dtype('float64')
```

- np.ndarray の np.ndarray.astype() メソッドを使う方法

```
In [14]: a = np.array([1, 2, 3])
In [15]: a.dtype
Out[15]: dtype('int64')
In [16]: a = a.astype(float)
In [17]: a.dtype
Out[17]: dtype('float64')
In [18]: a
Out[18]: array([ 1.,  2.,  3.])
```

np.ndarrayの要素の参照

- 多様な要素の参照方法がありますが，ここでは基本的な方法のみ
- 各次元ごとに何番目の要素を参照するかを指定する方法
- 例：1次元配列であるベクトル a の要素 3 を a[3] 参照

```
In [10]: a = np.array([1, 2, 3, 4, 5], dtype=float)
In [11]: a[3]
Out[11]: 4.0
```

※ 添え字の範囲は，1 からではなく 0 から数える

※ a.shape[0] で，第1次元の要素の長さとして 5 が得られたとき添え字の範囲はそれより一つ前の 4 まで

```
In [12]: a = np.array([[11, 12, 13], [21, 22, 23]])
In [12]: a.shape
Out[12]: (2, 3)
In [13]: a[1,2]
Out[13]: 23
```

1次元配列のスライス

スライス：リストや文字列などのスライスと同様の方法により，配列の一部分をまとめて参照する方法

◆ 1次元配列：リストのスライス表記と同様の **開始:終了:増分** の形式

※ 負値は最後からの個数を示す

```
In [10]: x = np.array([0, 1, 2, 3, 4])
In [11]: x[1:3]
Out[11]: array([1, 2])
In [12]: x[0:5:2]
Out[12]: array([0, 2, 4])
In [13]: x[::-1]
Out[13]: array([4, 3, 2, 1, 0])
In [14]: x[-3:-1]
Out[14]: array([2, 3])
```

複数要素の同時指定

- 配列やリストを使って複数の要素を指定し，それらをまとめた配列を作る
- 11行目の例：リストを使って，0番目，2番目，1番目，2番目の要素を繋げた配列
- 12行目の例：配列を使って要素を取り出して，まとめた例

```
In [10]: x = np.array([10, 20, 30, 40, 50])
In [11]: x[[0, 2, 1, 2]]
Out[11]: array([10, 30, 20, 30])
In [12]: x[np.array([3, 3, 1, 1, 0, 0])]
Out[12]: array([40, 40, 20, 20, 10, 10])
```

2次元配列のスライス

- スライスは，2次元以上の配列でも同様の操作が可能
- 特に，“:”のみを使って，行や列全体を取り出す操作は頻繁に利用

```
In [10]: x = np.array([[11, 12, 13], [21, 22, 23]])
In [11]: x
Out[11]:
array([[11, 12, 13],
       [21, 22, 23]])
In [12]: x[0, :]
Out[12]: array([11, 12, 13])
In [13]: x[:, 1]
Out[13]: array([12, 22])
In [14]: x[:, 1:3]
Out[14]:
array([[12, 13],
       [22, 23]])
```

np.ndarrayと数学の行列

※ 1次元の np.ndarray 配列には，線形代数でいう縦ベクトルや横ベクトルという区別はなく，1次元配列は転置できない

- 縦ベクトルや横ベクトルを区別して表現するには，それぞれ列数が1である2次元の配列と，行数が1である2次元配列を利用
- 縦ベクトルの例：

```
In [10]: np.array([[1], [2], [3]])
Out[10]:
array([[1],
       [2],
       [3]])
```

- 横ベクトルの例：

```
In [11]: np.array([[1, 2, 3]])
Out[11]: array([[1, 2, 3]])
```

配列の次元数の追加

- `np.newaxis` を利用して1次元のベクトルを2次元の行列に変換

```
In [10]: a = np.array([1, 2, 3])
In [11]: a[:, np.newaxis]
Out[11]:
array([[1],
       [2],
       [3]])
In [12]: a[np.newaxis, :]
Out[13]: array([[1, 2, 3]])
```

- `reshape` メソッドをしてベクトルを配列に変換

※ `-1` を指定すると，全体の要素数が不変となるように大きさを計算

```
In [20]: a = np.array([1, 2, 3])
In [21]: a.reshape(-1, 1)
Out[21]:
array([[1],
       [2],
       [3]])
In [22]: a.reshape(1, -1)
Out[22]: array([[1, 2, 3]])
```

第II部：機械学習アルゴリズムの実装

数値計算tips

NumPy/SciPyの便利な機能

ユニバーサル関数

ユニバーサル関数：入力した配列の各要素に関数を適用し，その結果を入力と同じ形の配列して返す

- ユニバーサル関数の機能を利用するには，mathパッケージの `math.log()` などではなく，NumPy の `np.log()` を用いる

```
In [10]: a
Out[10]: array([1, 2, 3])
In [11]: np.log(a)
Out[11]: array([ 0.        ,  0.69314718,  1.09861229])
```

※ 論理関数のユニバーサル関数は and などではなく np.logical_and() などを用いる

ユニバーサル関数の作成

- ユニバーサル関数ではないユーザ関数をユニバーサル化するには
  `np.vectorize()` を用いる

`np.vectorize(pyfunc)`

- `pyfunc` で指定したユーザ関数をユニバーサル化した関数を返す

※ 関数を引数にとり，関数を出力するのでデコレータとして利用可能

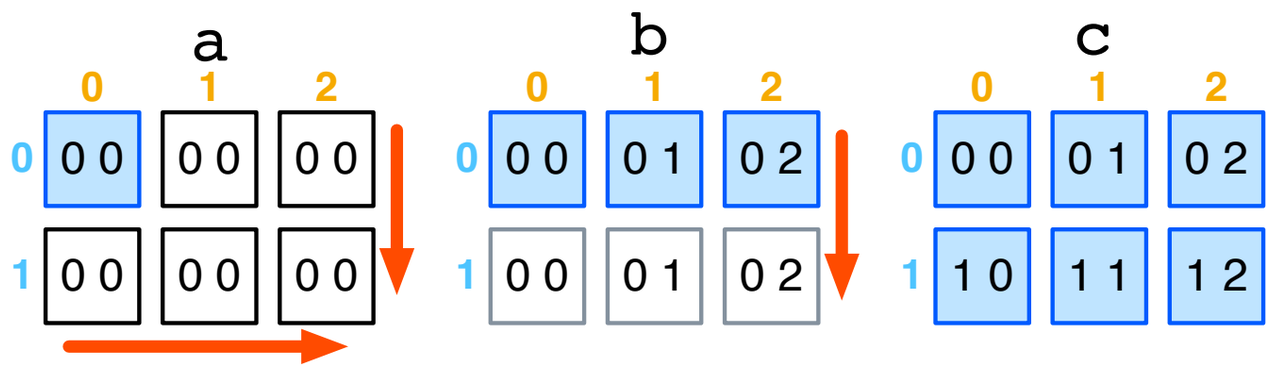
```
In [1]: @np.vectorize
   ...: def squre_plus_one(x):
   ...:         return x**2 + 1.0
In [2]: squre_plus_one(np.arange(3))
Out[2]: array([ 1.,  2.,  5.])
```

※ 出力が複数ある関数をユニバーサル化する `np.frompyfunc()` も

ブロードキャスト

ブロードキャスト：要素ごとの演算を行うときに，配列の大きさが異なっていれば自動的に要素をコピーして大きさを揃える機能

- 例：大きさがそれぞれ 0次元配列 a，長さ3の1次元ベクトル b，2×3 の行列 c の要素積を求める



- 大きさが1の次元は，必要に応じて要素の値を自動的にコピーする
- **np.newaxis により次元数を合わせてから利用することを薦める**

```
a[np.newaxis, np.newaxis] * b[np.newaxis, :] * c[:, :]
```

行列演算の関数プログラミング的実装

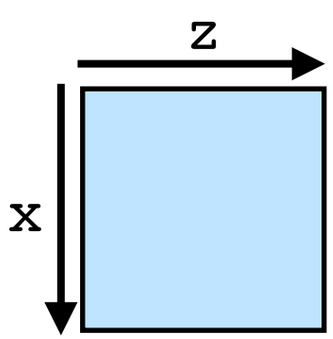
- `map()` と `reduce()` を使った関数プログラミング的な実装に基づくと，NumPy による計算は理解しやすいだろう
- リストではなく，共通の配列に `map()` して要素ごとの計算をしたあと，`sum()` や `max()` などの集約演算で `reduce()` すると考える

例：pLSAモデル

$$\Pr[X, Y] = \sum_Z \Pr[X \mid Z] \Pr[Z \mid Y] \Pr[Y]$$

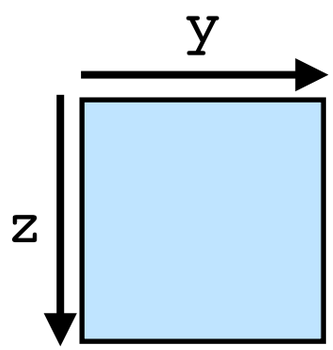
$\Pr[X \mid Z]$

`pXgZ[x, z]`



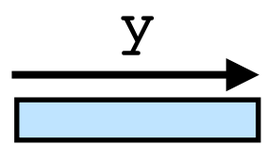
$\Pr[Z \mid Y]$

`pZgY[z, y]`



$\Pr[Y]$

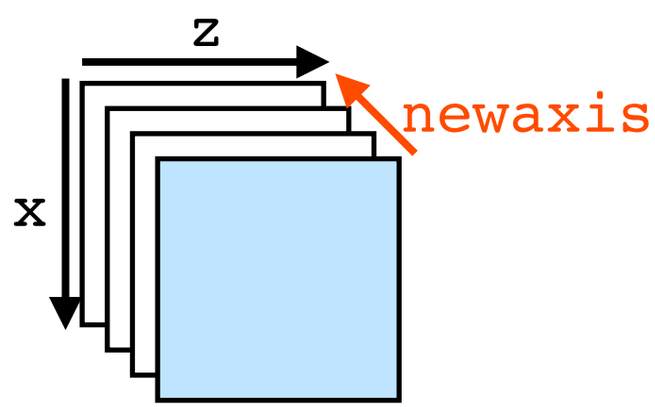
`pY[y]`



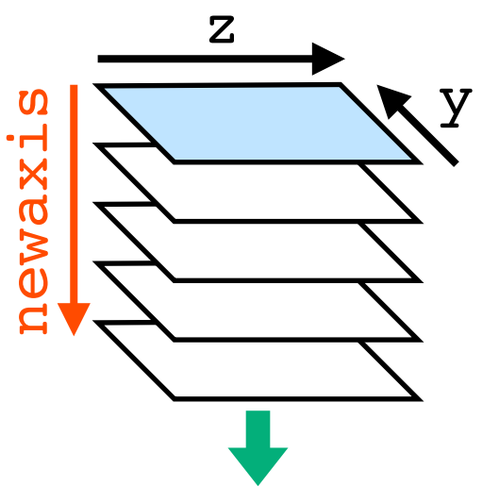
行列演算の関数プログラミング的実装

np.newaxisやブロードキャストで共通の配列に揃える

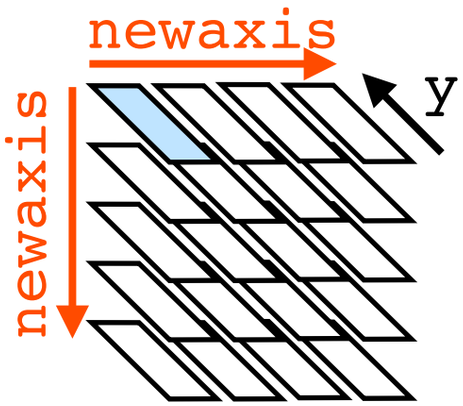
```
pXgZ[x,z,np.newaxis]
```




```
pZgY[np.newaxis,z,y]
```




```
pY[np.newaxis,np.newaxis,y]
```

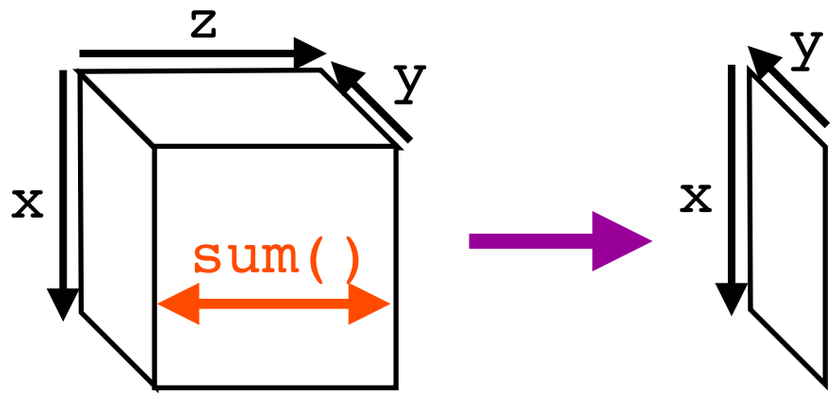


要素ごとの計算（map のイメージ）

```
pXgZ[x,z,np.newaxis] * pZgY[np.newaxis,z,y] * pY[np.newaxis,np.newaxis,y]
```

集約演算（reduceのイメージ）

```
np.sum(pXgZ[x,z,np.newaxis] * pZgY[np.newaxis,z,y]
       * pY[np.newaxis,np.newaxis,y], axis=1)
```



for文のブロードキャストへの変換

- for文を用いた計算を，ブロードキャストを使ったものと置き換える
  - n 個のデータを含む配列 x と y は，それぞれ {0, ..., n_x - 1} と {0, ..., n_y - 1} の値をとる
  - x と y の対の値を数え上げ分割表を作る

for文による基本的な実装

- n 個のデータを走査し，一つずつ数え上げる

サンプルファイル：contingency_table_example.ipynb

```
ct = np.zeros((n_x, n_y), dtype=int)
for i in range(n):
    ct[x[i], y[i]] += 1
```

※ 参考： http://www.kamishima.net/mlmpyja/nbayes2/distclass.html

for文のブロードキャストへの変換

配列を参照する変数が，for文の変数になるようにする

- 配列 ct のインデックスの x[i] などを，for文の変数 xi に置き換え
  - x や y の値を番号ではなく 0/1 ベクトルの形式で表す目的
  - x や y の値が，for文の変数 xi や yi と一致したとき数え上げる

x の値のために新たなfor変数 xi を導入

```
ct = np.zeros((n_x, n_y), dtype=int)
for i in range(n):
    for xi in range(n_x):
        for yi in range(n_y):
            if (x[i] == xi) and (y[i] == yi):
                ct[xi, yi] += 1
```

for文のブロードキャストへの変換

ブロードキャストを用いた実装

- 各for変数を，配列の各次元に割り当てる
  - for変数 i，xi，yi をそれぞれ配列の次元0, 1, 2に割り当てる
- for文の変動範囲を np.arange で作った配列と置き換える
  - さらに，3次元（for文の段数）の設定した次元に割り当てる
- if 文の判定はユニバーサル関数でまとめて実行（map演算）
- 最後に次元0の変数 i について総和をとる（reduce演算）

for変数 i は次元0に割り当てた

```
i = np.arange(n)[:, np.newaxis, np.newaxis]
xi = np.arange(n_x)[np.newaxis, :, np.newaxis]
yi = np.arange(n_y)[np.newaxis, np.newaxis, :]
ct = np.logical_and(x[i] == xi, y[i] == yi)
ct = ct.sum(axis=0)
```

欠損値や無限値

欠損値 np.nan や無限大・無限小 np.inf などを含むデータの処理

- **配列の要素が全て有限値かどうかを検査する**
  - np.all(np.isfinite(x)) の代わりに np.isfinite(x.sum())
- **np.nan を除外した集約演算**
  - 総和 np.nansum() や最大値 np.nanargmax() などがある
  - この場合に特化した bottleneck というパッケージも存在
- **欠損値を他の値で置き換える**
  - ユニバーサルな3項演算子にあたる np.where() が便利

```
In [10]: a
Out[10]: array([  0.,  nan,   1.,  inf,   2.])
In [11]: np.where(np.isfinite(a), a, -1.0)
Out[11]: array([ 0., -1.,  1., -1.,  2.])
```

欠損値や無限値

- scikit-learn で開発者用に提供されている欠損値などの検査関数
  - http://scikit-learn.org/stable/developers/utilities.html
  - sklern.utils には下記の関数の他，数値演算やテスト用のものが

assert_all_finite

- 配列に無限値や欠損値が含まれていると ValueError を発生

check_array

- 配列の要素の型の変換，次元数や大きさの検査，無限値・欠損値の有無などをまとめて実行できる

check_X_y

- check_array に加え，特徴用配列 `X` と教示情報用配列 `y` の大きさの一致を検査

演算エラー処理

- np.log(0) など有限にならない値を求めると，警告メッセージが表示される
- このような**演算例外に対する挙動を変更するのが np.seterr()**
  - **状況の指定**：全般 all，割り算 divide，オーバーフロー over，アンダーフロー under，不正演算 invalid
  - **動作の指定**：無視 ignore，警告表示 warn/print，例外送出 raise，関数呼び出し call，ログの記録 log
- **np.seterr(all='ignore') とすると警告メッセージを全て抑制できる**

scipy.optimize

scipy.optimize：非線形最適化のパッケージ，機械学習ではよく利用

minimize_scolar()：有限区間上のスカラー関数の最適化

- **brent** brent法，**golden** 黄金分割法

minimize()：多次元の非線形最小化

- **勾配を使わない最適化**：勾配を求める必要はないが，代わりに目的関数の評価回数が増えるので非常に遅い
  - **Nelder-Mead** ネルダー・ミード法，**Powell** パウエル法
- **勾配を使う最適化**：収束は早いが，勾配やヘシアンも与える必要
  - **CG** 共役勾配法，**BFGS** BFGS法，**Newton-CG** ニュートン法

※ BFGSやニュートンは収束は早いがヘシアン計算で遅くなることも

- **制約付きの最適化**：パラメータが有限区間にある多次元最適化
  - **L-BFGS-B**，**TNC** 切断ニュートン法，**COBYLA**

第 II 部：機械学習アルゴリズムの実装

数値計算プログラミングの注意点

Python や NumPy の浮動小数点数

- 数値計算プログラミングでは，数学上の概念である無限の精度がある実数ではなく，精度が有限の浮動小数点数で計算する

浮動小数点数：IEEE 754

$$[\text{符号}] \times 2^{\text{指数}} \times (1+\text{仮数})$$

  - 符号部 (sign)：正負の符号を表す
  - 指数部 (exponent)：桁数を表す
  - 仮数部 (mantissa)：有限桁の整数を表す

数値誤差：統計学では観測の過程で生じるものだが，**計算機科学としては浮動小数点を用いることによる誤差もある**

参考文献

- 伊理正夫，藤野和建『数値計算の常識』共立出版 (1985)
- 幸谷智紀『Python数値計算プログラミング』講談社 (2021)

Python と NumPy の整数型

- Python の整数型は任意の桁数の数値（メモリが許す限り）を扱える
- NumPy の数値は有限のbit数（int64 や uint32 など）なので，表現できる数の桁数は有限になる
  - ➡ 大きすぎる桁数の数を代入すれば，桁あふれ (overflow) となる
  - ➡ 浮動小数点に代入すれば有限精度の数になる

サンプルファイル：python_numpy_integer.ipynb

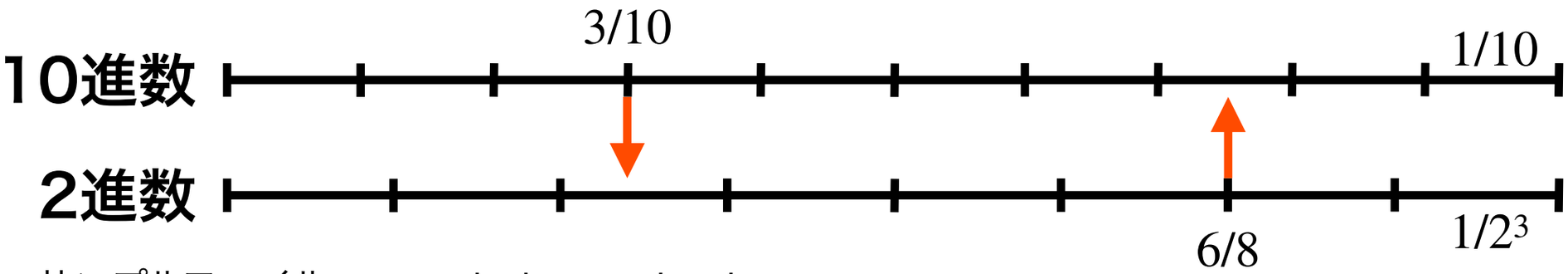
```
a = 1
for i in range(1, 50):
    a = a * i
print(a) # factorial 50
b = np.empty(1, dtype=np.int64)
b[:] = a
print(b)

608281864034267560872252163321295376887552831379210240000000000
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-8-d206455b3e0e> in <module>()
      4 print(a)
      5 b = np.empty(1, dtype=np.int64)
----> 6 b[:] = a
      7 print(b)
OverflowError: Python int too large to convert to C long
```

608那由多 2818阿僧祇 6403恒河沙 …

10進数と2進数

人が収集したデータ＝**10進数** ⇔ 計算機が処理するデータ＝**2進数**

- **整数**：10進数と2進数は相互に変換可能
- **小数**：**必ずしも相互に変換可能ではない** ＆ 有限桁ではさらに制限
  - **10進数**：10^{-n} 刻みの小数 ⇔ **2進数**：2^{-n} 刻みの小数を表現可能



サンプルファイル：numerical_errors.ipynb

```
a = np.empty(1, dtype=np.float64)
a[:] = 0.2
np.set_printoptions(precision=8) # default
print(a)
np.set_printoptions(precision=30)
print(a)
```

有効数字8桁では違いが分からない

```
[ 0.2]
[ 0.200000000000000011102230246252]
```

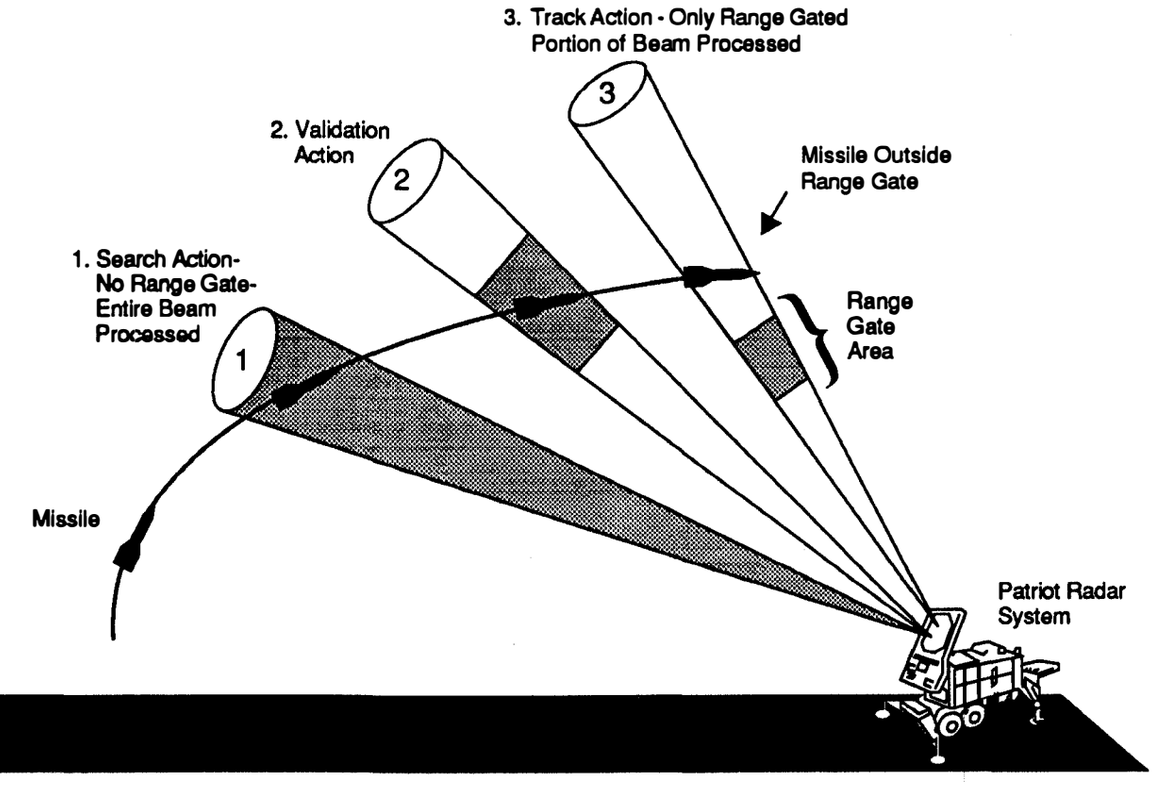
有効数字30桁では近似だと分かる

まるめ誤差による失敗事例

Patriot Missile Defense:
Software Problem Led to System Failure at Dhahran, Saudi Arabia

パトリオットシステムが，1991年に，サウジアラビアで，イランのミサイルの迎撃に失敗した

時間のカウンタを24bit精度の浮動小数点に変換



時間が経つと誤差が蓄積して，時間の精度を維持できない

https://www.gao.gov/assets/220/215614.pdf

数値誤差

- 量 x の測定値・浮動小数点による表現値 a に誤差 $\Delta a > 0$ がある

$x \in (a - \Delta a, a + \Delta a)$

- 量 y についても同様に観測値・表現値 b に誤差 $\Delta b > 0$ がある
- x と y についての演算で z を計算，z の表現値は c その誤差 Δc

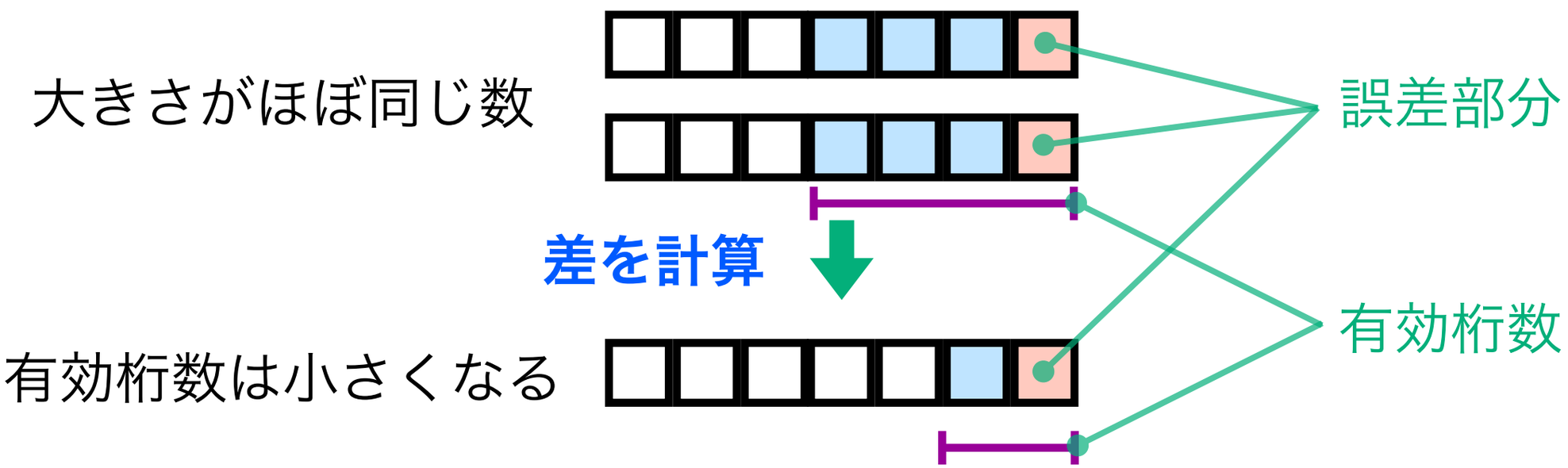
加減算：絶対誤差が和になる

$$z = x \pm y \quad \Rightarrow \quad \Delta c = \Delta a + \Delta b$$

乗除算：相対誤差が和になる

$$z = xy \text{ or } z = x/y \quad \Rightarrow \quad \left|\frac{\Delta c}{c}\right| = \left|\frac{\Delta a}{a}\right| + \left|\frac{\Delta b}{b}\right|$$

桁落ち



桁落ち

有効桁数に対して誤差部分
の影響が相対的に拡大する

大きさがほぼ同じの大きな数の差は誤差を大きくする

誤差に特に注意が必要な場合

- **加減算**：絶対誤差が和になる
  - **大きな数の加減算で，解の絶対値が小さい**（同じくらいの数の差）
    - ➡ 解に対して，絶対誤差の影響が相対的に大きい
    - ➡ 桁落ちを生じやすい
  - **大きな数と小さな数の加減算**
    - ➡ 小さい方の値に対して，解の誤差の影響が相対的に大きい
- **乗除算**：相対誤差が和になる
  - **小さな数と大きな数の乗除算**
    - ➡ 相対誤差の大きくなりやすい小さな数が，解の相対誤差を大きくする

桁落ち

大きな数同士の差による桁落ちの例

本来は負にならない（二乗平均）−（平均の二乗）が負になる

サンプルファイル：numerical_errors.ipynb

```
n = 1000
x = 1000 + np.random.randn(n) * 10**-5
print(np.mean(x**2) - np.mean(x)**2)

-1.16415321827e-10
```

1000 + 小さな乱数

大きな絶対値を引く

➡ 誤差に対する有効桁数を相対的に大きくして対処

```
y = x - 1000
print(np.mean(y**2) - np.mean(y)**2)

9.50575571567e-11
```

大きな絶対値を引く

大きな数と小さな数の和

大きな数と小さな数の加減算 ➡ 大きな数の絶対誤差＞小さな数

小さな数の総和：最後の方は大きな数と小さな数の和は，大きな数にとって誤差となる

サンプルファイル：numerical_errors.ipynb

```
k = 5
a = 0
for i in range(10**k):
    a += 10**-k
print(1 - a)

1.9162494050302e-12
```

1/10000 を 10000回 足す

```
a = 0.
for i1 in range(10):
    b = 0.
    for i2 in range(10):
skip…
   a += b
print(1 - a)

-6.661338147750939e-16
```

小さな数同士を先に足して大きな数にする

分けて計算すれば影響は小さい

桁あふれ

シグモイド関数など指数を使う関数は容易に桁あふれを起こす

$$\mathrm{sig}(x) = \frac{1}{1 + \exp(-x)}$$

サンプルファイル：numerical_errors.ipynb

```
np.set_printoptions(precision=30)
for x in range(20, 50, 5):
    print(x, sig(x))
print(sig(50) == 1.0)
```

```
20 0.999999979388463
25 0.999999999986112
30 0.999999999999065
35 0.999999999999993
40 1.0
45 1.0
True
```

本来は 1 にならないはずだが，1 になる

↓

x の入力の範囲をクリップしてしまう

```
x = np.clip(x, -34.538776394910684, 34.538776394910684)
```

小さな数の割り算

大きな数 / 小さな数 ➡ 小さな数の相対誤差が結果の相対誤差

特異値分解　$\mathbf{A} = \mathbf{U}\,\mathrm{diag}[\lambda_1, \lambda_2, \ldots, \lambda_{p-1}, \lambda_p]\,\mathbf{V}^\top$

⬇

逆行列　$\mathbf{A}^{-1} = \mathbf{V}\,\mathrm{diag}[1/\lambda_1, 1/\lambda_2, \ldots, 1/\lambda_{p-1}, 1/\lambda_p]\,\mathbf{U}^\top$

割り算

逆行列の計算では固有値の除算により相対誤差が大きくなりやすい

⬇

小さな固有値は無視して行列のランクを小さくして回避

条件数 (conditional number) $\|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|$

- 連立方程式 $\mathbf{Ax} = \mathbf{y}$ で $\mathbf{x}$ の誤差は $\mathbf{A}$ や $\mathbf{y}$ の誤差のたかだか条件数倍
- np.linalg.cond ：いろいろなノルムの条件数

数値誤差への対策

誤差があることを前提とした実装

- **数値誤差のために，非負などの条件が成立しない場合がある**
  - 例：分散の公式（二乗平均 – 平均の二乗）が負になりうる
  - 非負を保証するように max 関数などを用いる

```
np.sqrt(np.max(0, np.mean(x**2) - np.mean(x)**2))
```

- **浮動小数点の計算では等号演算子は使わない**
  - 代入だけでも，値が変わることがあり，成立しない
  - np.isclose(a, b) という微少な変動を許した等号判定関数がある

数値誤差への対策

- 明示的に比較演算をしない場合でも，暗黙的な比較がある場合
  - 加算を繰り返すことの誤差によって，大小比較の結果が変わるため，指定した終了値で終わらない場合がある

```
In [1]: np.arange(1, 1.5, 0.1)
Out[1]: array([1. , 1.1, 1.2, 1.3, 1.4])
In [2]: np.arange(1, 1.6, 0.1)
Out[2]: array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6])
```

- この場合は指定した区間を区切る np.linspce() で対応できる

```
In [3]: np.linspace(1, 1.5, 6)
Out[3]: array([1. , 1.1, 1.2, 1.3, 1.4, 1.5])
In [4]: np.linspace(1, 1.6, 7)
Out[4]: array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6])
```

桁あふれ対策

- 最後の計算結果の最大化するだけなら，対数などをとる
- 桁あふれが生じそうな入力に対しては，定数などで置き換える
- Π記号の計算で対数をとってから最後に指数関数を適用する
  - sp.misc.logsumexp() などの専用関数もある
- 途中の結果が大きく，最後に小さくなる場合
  - 例：ユークリッド距離

```
np.sqrt((x**2).sum())
```

  - 大きな x で正規化しておく

```
x_max = np.max(np.abs(x))
x_max * np.sqrt(((x / x_max)**2).sum())
```

第III部：関連パッケージと情報源

関連パッケージ

Jupyter Notebook，scikit-statmodels，最適化など

Jupyter Notebook/Lab

便利な対話的な実行環境

- コマンドの補完（tabキー），ヒストリの管理など
- notebook は，計算結果をそのままメモとして残すことができる
- ipyparallel で複数のマシン上でのジョブ管理ができる

- **ドキュメントの参照**
  - オブジェクト名 + “?” で，そのオブジェクトの説明を参照できる

- **マジックコマンド**：% (単一行)や %% (複数行) で始まるコマンド群
  - **%quickref**：コマンドのリファレンス
  - **%cd**, **%ls**：shell としてのコマンド
  - **%timeit**：関数の実行時間の計測
  - **%run**：ファイルの実行
  - **%pycat**：カラーリング付きのソース表示

Jupyter Notebook

分析コードとメモを一つにまとめておける

- Jupyter は ipython と独立して，Python の他，R / Julia など様々なスクリプト言語が利用できるようになった.

- **起動**：コマンドラインでは “`jupyter-notebook`”
  - **ヘルプの keyboard shortcuts を最初に読むとよい**
- **メモの記述**：Markdown 記法と，LaTeX の数式記法が利用可能
- **カスタマイズ**：“ipython profile create” を実行すると，“ipython profile locate” で表示される場所に“ipython_default” が作成され，カスタマイズ可能に
  - その中の startup ディレクトリに起動ファイル .ipy を設置できる
- **分析結果の保存**：notebook は `.ipynb` ファイルに保存できる
- **ノートブックの公開**：**GitHub** や **nbviewer** https://nbviewer.ipython.org で公開できる

SymPy

- **数式処理を行うためのパッケージ**
- 最初に sympy.init_session() を実行しておくと便利
  - x, y, z, t を実数変数，k, m, n を整数変数と認識するようになる
- **主な機能**
  - 微分：diff，積分：integrate，極限：limit
  - 展開：expand，単純化：simplify，テイラー展開：series
  - 方程式の解：solve

サンプルファイル：sympy_demo.ipynb

sympy.init_session() 実行後に行う

微分

```
In [3]: diff((x ** 2 + log(x)) / x, x)
```

Out[3]: $\frac{1}{x}\left(2x + \frac{1}{x}\right) - \frac{1}{x^2}\left(x^2 + \log(x)\right)$

積分

```
In [4]: integrate(x ** 3 + sin(x) ** 2, x)
```

Out[4]: $\frac{x^4}{4} + \frac{x}{2} - \frac{1}{2}\sin(x)\cos(x)$

展開

```
In [5]: expand((x + 1)**2)
```

Out[5]: $x^2 + 2x + 1$

scikit-statmodels

- scikit-learn と同様に使える，scikit-learn に対する特色は
  - ARMA などの時系列分析モデル
  - R言語のように線形モデルを記述する記法を備える
  - 多重検定など検定周辺の機能

サンプルファイル：statsmodels_demo.ipynb

パッケージの読み込み

In [1]:
```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
```

データの読み込み

In [2]:
```
url = 'http://vincentarelbundock.github.io/Rdatasets/csv/HistData/Guerry.csv'
dat = pd.read_csv(url)
```

回帰モデルのあてはめ

In [3]:
```
results = smf.ols('Lottery ~ Literacy + np.log(Pop1831)', data=dat).fit()
```

結果の表示

In [4]:
```
print results.summary()
```

自動微分

- TensorFlowなどの深層学習やAutograd などのパッケージが提供する
- 数値微分とは異なり，解析的に微分し，数値を代入した結果を得る
- 他にも数式を直接的に定義可能
- 非線形最適化など各種の数理計画法で結果を利用可能

サンプルファイル：tensorflow_auto_diff.ipynb

Tensorflow 関連のパッケージの読み込み

In [2]:
```
import tensorflow as tf
import numpy as np
```

スカラー変数のシンボルの定義

In [3]:
```
x = tf.placeholder(tf.float64) # double
```

関数 $f(x) = \sin(x)$ を定義

In [4]:
```
y = tf.sin(x)
```

変数 x に，値 π を代入して関数 $f(x)$ 評価する：$\sin(\pi) = 0$

In [5]:
```
feed_dict = {x: np.pi}
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  print(sess.run([y], feed_dict))
```

```
[1.2246467991473532e-16]
```

In [6]:
```
dy = tf.gradients(y, x)
```

関数 $f(x)$ を微分して，値 π で評価する：$\sin'(\pi) = \cos(\pi) = -1.0$

In [7]:
```
feed_dict = {x: np.pi}
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  print(sess.run(dy, feed_dict))
```

```
[-1.0]
```

ベイズ推定

- Python でのベイズ推定では PyMC と PyStan が著名
  - どちらも，生成モデルを定義すると MCMC により，データを与えたときの変数の事後分布を推定する

PyMC3 https://docs.pymc.io/

- Python でのベイズ推定パッケージとしては開発が長く続いている
- Python のオブジェクトを使ってモデルを構築できる
- Tensorflow Probability を基盤にしたPyMC4は開発頓挫

PyStan https://pystan.readthedocs.org

- 専用の記述言語でモデルを構築し，それを Python からでも R からでも利用できる

超パラメータ調整

- 機械学習手法は適切な超パラメータの設定の必要
- scikit-learn には GridSearchCV や RandomizedSearchCV がある

AutoML https://www.ml4aad.org/automl/

Optuna https://www.preferred.jp/en/projects/optuna/

- 性能指標の分布予測を利用した高効率な超パラメータの探索

※ **より多くの手法や超パラメータの中から選択するには，それに応じた十分なデータ数が必要なことには注意**

サンプルファイル：

auto_sklearn_demo.py

http://www.ml4aad.org/automl/auto-sklearn/

```
import autosklearn.classification
import sklearn.model_selection
import sklearn.datasets
import sklearn.metrics

# load data
X, y = sklearn.datasets.load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = (
        sklearn.model_selection.train_test_split(X, y, random_state=1))

# search best method and hyper-parameters
automl = autosklearn.classification.AutoSklearnClassifier()
automl.fit(X_train, y_train)
```

最適化関連パッケージ

scipy その他の最適化関連

- **scipy.optimize.root** 関数値を0にする入力値である根を求める
- **scipy.optimize.linprog** 線形計画法

cvxopt http://cvxopt.org

- 機械学習では必要になる凸最適化に特化
- 凸最適化で著名な Stephen Boyd のチームが作成

TensorFlow https://www.tensorflow.org/

- 自動微分と確率的勾配降下による最適化
- 計算モデルの可視化，分散環境，ユーザコミュニティが大規模

autograd https://github.com/HIPS/autograd

- TensorFlow のようにシンボルを定義せず，Python 関数そのままで自動微分ができる

機械学習の総合パッケージ

scikit-learn https://scikit-learn.org/

- よく利用され，安定している
- contrib https://github.com/scikit-learn-contrib などの拡張も

MLlib https://spark.apache.org/mllib/

- 大規模並列データ処理環境Spark上で動作する
- 分類・相関・クラスタリングに加え，頻出パターンマイニングも

Orange https://orange.biolab.si

- 分析モデルを GUI を用いて対話的に構築できる統合環境
- 頻出パターンマイニングなどの拡張プラグインも

Vowpal Wabbit https://github.com/VowpalWabbit/vowpal_wabbit

- オンライン最適化を用いた大規模・高速対応

深層学習

Pythonから利用できる深層学習フレームワークは多数

PyTorch http://pytorch.org/

- 動的に生成する計算グラフに強い

Keras https://keras.io/

- TensorFlow を演算基盤に，モデルを簡易に定義可能

※ 他に **MXNet** https://mxnet.incubator.apache.org/ や **Caffe** https://caffe.berkeleyvision.org/

高速化

numexpr https://github.com/pydata/numexpr

- 要素ごとの演算をキャッシュミスが起きないようにする

bottleneck http://berkeleyanalytics.com/bottleneck/

- 欠損値 np.nan がある場合に高速化

Cython http://cython.org

- 制限はあるが Python をネイティブコードに変換できる
- 型宣言をコードに追加するとさらに高速化できる

高速な実行環境

- **PyPy**：文法を制限した Python を実行
- **Numba**：NumPy用のコンパイラ
- **PySton**：clang などの仮想マシン LLVM 上で実行

並列実行環境

joblib https://pythonhosted.org/joblib/

- jobの並列パイプライン実行
- scikit-learn で利用されており，パッケージ内に含まれている
- 実際のバックエンドに dask や ipyparallel を利用可能

dask https://dask.pydata.org

- Numpy や Pandas の操作を並列化できる
- バックエンドにipyparallel を利用可能

ipyparallel https://ipyparallel.readthedocs.io

- クラスタノード間の通信にいろいろな手段が使える
- ローカル, SSH, MPI, PBS (SGE, Torque, LSF), Windows HPC
- SSHトンネリングで，クラウド上でjobをセキュアに実行可能
- jupyter notebook から操作でき，便利な %コマンド がある

その他のパッケージ

matplotlib https://matplotlib.org

- グラフの描画

Pandas https://pandas.pydata.org

- R言語のデータフレームのような表データに対する多様な操作

csvkit https://csvkit.readthedocs.org

- csvファイルを操作する便利なコマンド群

Sphinx https://www.sphinx-doc.org/

- ReStructuredText 形式で記述するドキュメント作成ソフト

OpenCV https://opencv.org

- 画像処理で著名なライブラリ．Pythonインタフェースあり

NLTK https://www.nltk.org

- 自然言語処理のPythonパッケージ

第III部：関連パッケージと情報源

情報源

Pythonによるデータ分析についての資料や勉強会

国際会議・勉強会

SciPy Conferences https://conference.scipy.org/

- 科学技術計算での Python 利用に関する国際会議
- US，欧州，各国の会議情報があり，スライド・ビデオを公開

PyCon JP https://www.pycon.jp

- 毎年9月に開催．アジア太平洋の PyCon APAC として開催されることも
- Python全般の国内会議．科学技術計算系の講演が増えている

PyData Tokyo https://pydatatokyo.connpass.com

- データ分析でのPython利用の勉強会

Start Python Club https://startpython.connpass.com

- Pythonでスタートする人たちの集い

拙著のSciPy関連資料

拙著のPythonプログラミング関係を含む資料

- http://www.kamishima.net/jp/kaisetsu/

機械学習のPythonとの出会い (Machine Learning Meets Python)

- 機械学習とPythonの基礎を知っている人向けの，NumPy / SciPy を使った数値計算プログラムのチュートリアルです
- 具体的なアルゴリズムを実装する過程を通じて，パッケージをどう使うかを知ることができるように工夫しました
- HTML版の他PDF版とePub版も配布しています

Python による科学技術計算の概要

- Pythonを使った科学技術計算の講演資料をまとめたものです

朱鷺の杜Wiki http://ibisforest.org/index.php?python

- 関連情報をいろいろ集めてまとめています