J. Plasma Fusion Res. Vol.94, No.2 (2018) 86‐89

講座

今日からはじめる Python

You're One of Pythonistas from Today

1．はじめに

1. Preface

吉沼幹朗
YOSHINUMA Mikirou
核融合科学研究所

（原稿受付：2017年11月20日）

汎用プログラミング言語のひとつであるPythonは，多くの環境で誰でも利用でき，簡単な記述で様々なことができるため利用者が増えています．プラズマ・核融合分野においても利用者を見かけるようになり，Pythonを使ってみようと考えている人も多いのではないでしょうか．本章では，Pythonがどのような言語であるかを手短に説明します．

Keywords:
Python, set-up, guideline

1.1　Python とは

プログラミング言語 Python は Guido van Rossum 氏によって作られました．Python と書いて，「パイソン」と呼ばれています．現在はPythonソフトウェア財団で管理，開発されています．数年前，すでに海外において多くのユーザを獲得していたPythonですが，日本においては，和文の入門書や解説書の数も少なく，周囲で利用しているユーザも少ない状態でした．近年，大量に取得されたデータ（ビッグデータ）の統計解析やそれを用いた機械学習といった技術が広まるとともに，日本においても，その解析手法の開発環境として Python のユーザが増加してきたと思われます．最近では，書店のパソコン関連の棚にもタイトルに Python が入った書籍を見かけるようになり，入門書も数多く出版されています．また，インターネット経由でアクセスできる情報も充実しており，多くのサンプルコードや日本語による解説を見ることができます．

プラズマ・核融合分野においても，Python を利用している人を見かけるようになりました．そろそろ使ってみようと考えている人もいることでしょう．はじめようと思うけれど，これまで馴染んできた Fortran や C 言語といったコンパイラとは異なる点に不安を覚えているかもしれません．本講座は，はじめてみようかと思っている人の背中を押して，実際にはじめてもらうことをめざしています．Python という言語を短く説明すると「誰でも利用できる書きやすく読みやすい汎用のプログラミング言語，だけど簡単」となります．

1.2　Python は誰でも利用できます

Python は無料で利用できるので，どなたでもインストールしておくことができます．また，Windows，Linux，OS X といった主要なプラットホームで利用できます．これは，共同研究をする上で非常に重要なことです．ある解析プログラムを渡して実行をお願いしても，その実行環境のライセンスを相手が持っていない場合がしばしばあるためです．Python で記述されたスクリプトは，主要なプラットホーム上で誰でも実行できます．

1.3　Python は書きやすい言語です

Python は，実行時の値によって変数の型が決められる（動的型付け）言語です．変数の型を指定する必要がないため，処理内容に集中して書くことができます．また，インタラクティブな実行環境を持っているため，処理の記述（プログラム作成）と適用結果の表示を繰り返して解析を進めていくことが容易です．

National Institute for Fusion Science, Toki, GIFU 509-5292, Japan

author's e-mail: yoshinuma@nifs.ac.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research

1.4 Pythonは読みやすい言語です

Pythonでは，ブロック構造をインデントを用いて表現するため，読みやすい構造をもった書式が維持されます．ブロック構造とは，条件つきの処理，繰り返し処理や関数定義などの範囲のことです．自由形式の言語に慣れている人にとっては，書きづらいと感じるかもしれません．また，インデントのずれによって正常に動かなくなることを不安に感じる人も多いと思います．しかし，読み易さを損なわないためには，どの言語であっても一貫したインデントは必要です．また，インデントのずれについては，TABキーによって空白が入らないようにエディタを設定するだけで防ぐことができます．そして，Pythonでは技巧的な短い（パズル的で楽しい）記述よりも，わかりやすい記述が選ばれます．このような特徴から，インターネット上の解説やスクリプト，さらには付属のライブラリまで，読んで理解することが比較的容易です．

1.5 Pythonは汎用プログラミング言語です

Pythonは汎用のプログラミング言語なので，データ解析のみならず，テキストファイルの処理，グラフィカルユーザーインターフェース（GUI）の作成，ネットワークプログラミング，画像処理，ゲーム作成などパソコンで行われるほとんどのことに利用できます．また Pythonプログラムは，手続き的に記述することも，オブジェクト指向や関数型と呼ばれるような技法で記述することも可能であるため，そのようなプログラミング技法を理解することの助けにもなるでしょう．プログラミング入門用の言語としてもおすすめできます．

1.6 だけど簡単

Pythonが人気を得た理由のひとつとして，多くの有用なモジュールの存在があると思われます．モジュールは，パッケージと呼ばれる機能がつまったモジュールファイルの集まりで，それらを自分のPython環境にインストールすることによって，プログラムを開発しなくても必要な処理が行えるようになります．Pythonに標準で備わっているモジュールでも多くのことができますが，外部から提供されるモジュールによって，さらに簡単にできるようになります．たいていの場合，行いたい処理に応じたモジュールを利用することになります．プラズマ・核融合分野においては，数値計算や解析のための優れたモジュール（NumPy，SciPy，Matplotlib，pandas）や，それらのモジュールを便利に利用するための対話的な環境（jupyter）が利用されています．

1.7 Pythonの欠点

よいことばかりのように書いてしまいましたが，Pythonは万能ではありません．C言語やFortranといったコンパイラ言語を使っている方は，その実行時の速度に不満を持つかもしれません．大規模な数値計算やシミュレーションの主となる部分にPythonを利用することは考えるべきではないでしょう．コードに依存しますが，動的な型付けを行うインタプリタであるPythonは大雑把に言うならばC言語と比べて数十倍遅いでしょう．しかし，実験データの解析処理のうちインタラクティブに進められるような時間スケールの処理については，記述しやすいPythonを用いる方が素早く結果を得られることがあるでしょう．PythonからC言語やFortranのコードを利用することができますので，時間がかかる処理をC言語やFortranに置き換えることも可能です．Pythonは，それらの処理へのデータ入出力を担当することができるでしょう．そうすることで，C言語やFortranでは記述するのが煩わしい入出力の部分を，ユーザにとって便利なものにすることが容易になります．

1.8 Pythonを試してみよう

Pythonがどのような言語かイメージしていただけたでしょうか．Pythonは，以下のような場面で便利に利用できるかと思います．

- 電卓代りにちょっとした計算をしたい．
- シェルスクリプトで解析処理を制御しているけど，より複雑な分岐制御をしたい．
- 様々なデータファイルから必要なデータを取り出して，解析したい．
- 解析処理プログラムをFortranで書いたけど，入力データファイルの書式がバラバラだ．書式を合わせたい．
- ネットワーク経由で情報を取得したい．サーバー上の解析プログラムを自動で実行したい．
- 解析結果をグラフで描画しながら解析を進めたい．
- すばやく解析処理を書いて，手法の可能性を確認したい．

第2章にて，Pythonを使えるようにする方法が説明されますが，ここでは，環境を整える前にPythonを試していただきたいと思います．ウェブブラウザでhttps://try.jupyter.org/にアクセスしてください．**図1**のようなページが表示されると思います．このページは，第2章で紹介されるJupyter-notebookという対話型のPython開発環境を体験することのできるサービスです．右上のNewボタンからPython 3を選択すると，新しいノートブックファイルが作成され，図のような新しいノートブックページが開かれます．

Jupyter-notebookでは，セルと呼ばれるボックス内にスクリプトを入力します．まずは**図2**にあるように，ひとつ目のセルに以下を入力してみましょう．

```
print('Hello world!')
```

このセルを実行するためには，再生ボタンを押すか，キーボードのShift + Enterを同時に押下してください．

実際に実行すると，入力したセルの下に「Hello world！」が表示されると思います．C言語などのコンパイル言語でこのような命令を実行するためには，まずは完全なソースコードを作成し，コンパイルして実行ファイルを生成することが必要です．一方でインタープリタ言語である

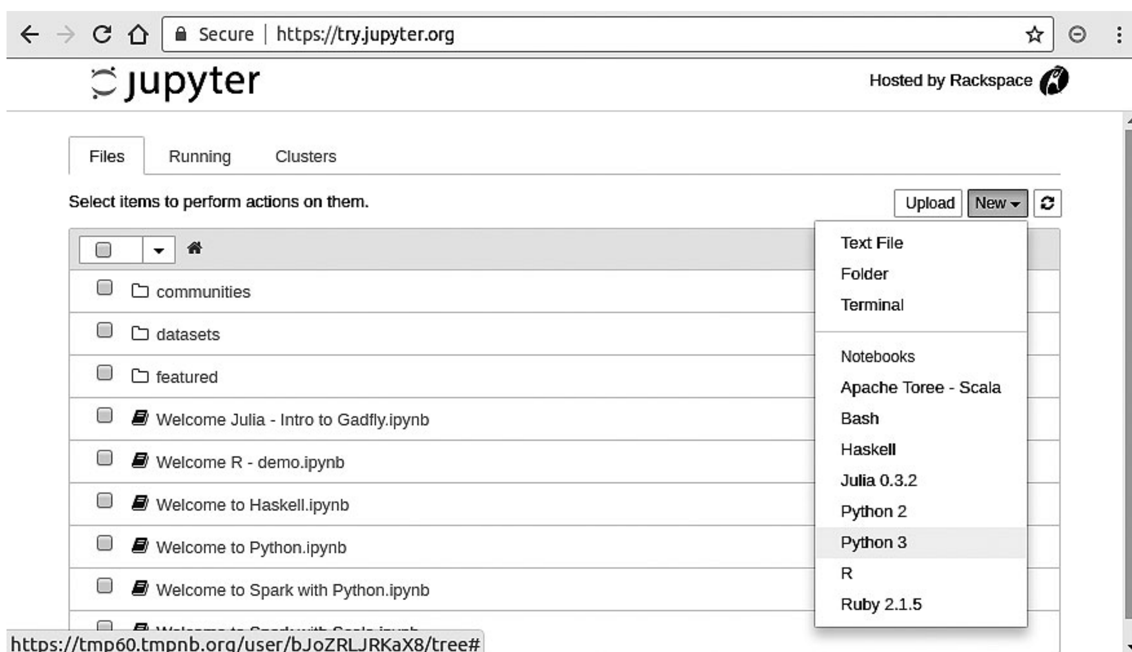


図1 http://try.jupyter.org のトップ画面．右上の New ボタンから新しいノートブックファイルを作成できます．

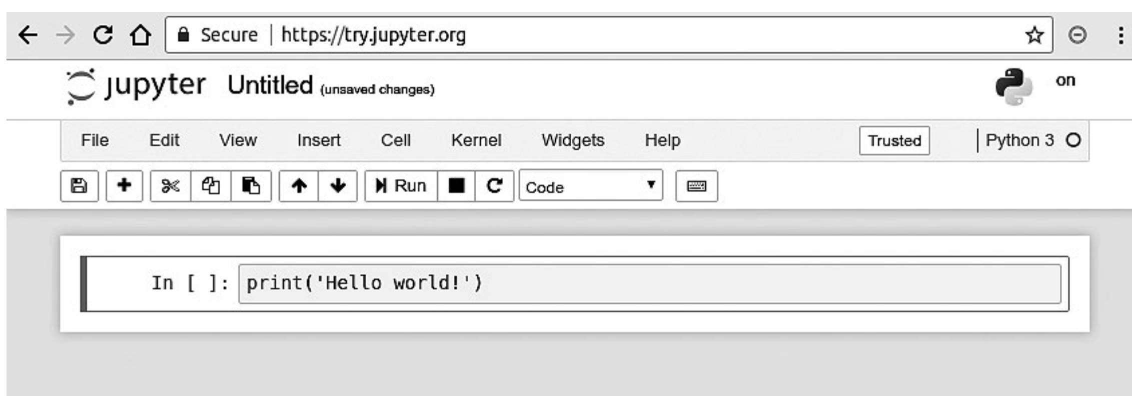


図2 新しいノートブックファイルを作成したときの様子．ひとつ目のセルに print（'Hello world!'）と入力しています．

Python ではコンパイル作業が不要であるほか，開発途中のソースコードでも途中まで実行できる点が特徴です．

次に**図3**のように以下の内容を入力してみてください．

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.linspace(-3, 3, 21)
x
```

```
y = np.exp(-x*x / 2)
y
```

```
plt.plot(x, y)
```

上記では，ベクトル変数 x と y を定義し，その内容を画面に表示したあと，グラフに描画しています．詳しくは，次号の NumPy や Matplotlib の説明によって明らかになるでしょう．

Python ではこのように，手早くコードを作成・実行し結果をすぐに可視化できます．このような特徴は，試行錯誤しながら進める研究活動を行うのに有用なだけでなく，プログラムスキルを身に付けるためにも非常に効果的だと思います．もちろん，このような対話的な環境ではなく，一般的な言語と同じようにコマンドでの実行もできます．ぜひ第2章を読んで，開発環境をあなたの PC にインストールしてください．

1.9 まとめ

Python をはじめてみようという気分になってきたでしょうか．Python をはじめたばかりの方からは，どのようなときにどのようなモジュールを利用したらよいか判断できない，モジュールの使い方がわからないという声を聞きます．そこで本講座では，プラズマ・核融合分野でよく利用されている環境やモジュールを，みなさんが使い始めることができるように紹介していきたいと思います．

本講座は，全3回の構成です．第1回の今回は「Python スタートアップガイド」として，Anaconda という Python

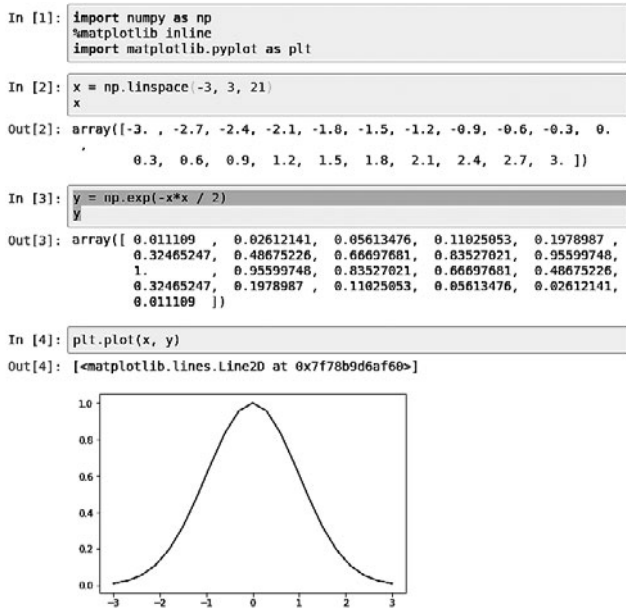


図3 ノートブックファイルの実行例.

の実行環境のインストール方法をWindows，Linux，OS Xを利用の方に向けて説明します．その後，Pythonスクリプトの作成方法，実行方法とJupyter-notebookという対話的な環境を紹介し，それを用いてPythonの基本的な文法を説明します．次号では，「Pythonによる科学技術計算」ということで，核融合分野に限らず，科学技術計算に利用されるモジュールである NumPy，SciPy，およびその結果をグラフとして描画するのに便利なMatplotlibの使い方が説明されます．また，機械学習を実行するモジュールscikit-learnについても紹介されます．そして，3回目では，「Pythonの活用事例」として，LHD実験やJT-60SA実験においてどのようにPythonが利用されているかを紹介していただきます．

なお，本記事で紹介するソースコードやデータは，http://purakaku-python.readthedocs.ioにもアップロードしています．各自のパソコンでぜひ実行しながらお読みください．

吉沼幹朗（よしぬまみきろう）

核融合科学研究所，大型ヘリカル研究部，助教．2000年，東北大学大学院工学研究科電気・通信工学専攻博士課程修了．LHDにおいて中性粒子ビームを用いたプラズマ計測（荷電交換分光，モーショナルシュタルク効果分光）を行っています．特にプラズマ中の流れや電場構造の形成に関心をもって研究を行っています．LHDの実験シーケンスに同期させた解析プログラムの実行や解析データ可視化にPythonを利用しています．

J. Plasma Fusion Res. Vol.94, No.2 (2018) 90‑101

講座 今日からはじめる Python

2．Python スタートアップガイド

2. Python Start-Up Guide

藤 井 恵 介
FUJII Keisuke
京都大学工学研究科

（原稿受付：2017年11月20日）

Python は簡単に始めることのできる言語です．これまでプログラミングにほとんど触れたことのない人でも数クリックで環境構築を完了し，数分でグラフ描画までできることが特徴です．さらに，多様なパッケージが公開されており，最先端のデータ操作を無料で行うことも可能です．ここでは Python 開発環境の構築法を紹介します．さらに，少し発展的な話題として，外部パッケージを新しくインストールする方法，Python 開発環境を一つの PC 内に複数構築する方法について述べることにします．

Keywords:
Python, Anaconda distribution, programming, open-source, virtual environment

2.1 Python のインストールとパッケージマネージングシステム

2.1.1 Python 開発環境の構築

これまで，プログラミング開発環境の構築に手間取って何時間もかけた記憶のある人も多いと思います．しかし，Python はそうではありません．本講座は「今日から始める Python」と題している通り，今日から Python プログラミングを始めることを趣旨としています．忙しい研究者の方々の時間を，環境構築などにかけてはいられません．セットアップは 5 分で終わらせましょう．

ここでは，Windows，Mac および Linux の各オペレーティングシステム（OS）上での Python 開発環境の構築法について述べます．開発環境をセットアップする方法も色々ありますが，ここでは非常に簡単で人気のある Anaconda distribution という統合パッケージを利用することにします．

なお，Anaconda distribution の特徴や詳細については環境構築法を述べたあとに紹介します．Python を触ったことのない初心者は，環境構築が完了すれば以降は飛ばして，次節の Jupyter-notebook を用いた Python 入門に進んでもよいでしょう．

2.1.1.1 Windows での開発環境構築

まずは Anaconda distribution ダウンロードページ https://www.anaconda.com/download にアクセスし，Windows 版 Anaconda をダウンロードします．なお，（残念ながら）Python にはバージョン 2 系（現在の最新は2.7）と 3 系（現在の最新は3.6）という大きく 2 つのバージョン系列があります．Python2.7 のサポートが2020年に打ち切られることが議論されていること[*1]，現在も開発続いているパッケージはほとんどが 3 系に対応していることから，これからはバージョン 3 系を利用することをお勧めします．

OS の bit 数などに合わせたバージョン（現在では64bit システムが一般的です）をダウンロードしてください．ダウンロードされたインストーラをクリックすることでインストールを開始できます．

詳細は後述しますが，Anaconda では Python 開発環境をユーザ固有のもにするか PC 内で共通にするか選ぶことができます．他のユーザと環境が衝突しないよう，個人ごとの環境を構築すること（**図 1** の Just Me を選択）を勧めます．インストーラが終了したあと，Anaconda Prompt や Jupyter Notebook というアプリケーションがスタートメニューに登録されていれば Python 開発環境の構築は終了です．

2.1.1.2 Mac での開発環境構築

https://www.anaconda.com/download/にアクセスします．執筆時点の Anaconda distribution の version は5.0.1です．ウェブサイトでリンゴの画像をクリックすると Python 3.6 と Python 2.7 のインストーラーが出てきますので迷わず，3.6 version をダウンロードします．この際，Cheat Sheet（Starter Guide）が必要かと聞かれます．必要な場合は e-mail address を入力します．Get して損はないでしょう．

その後はインストーラを起動し，基本的に続けるボタンを押し，最後にインストールボタンを押せばインストール完了です．ここで

```
which python
```

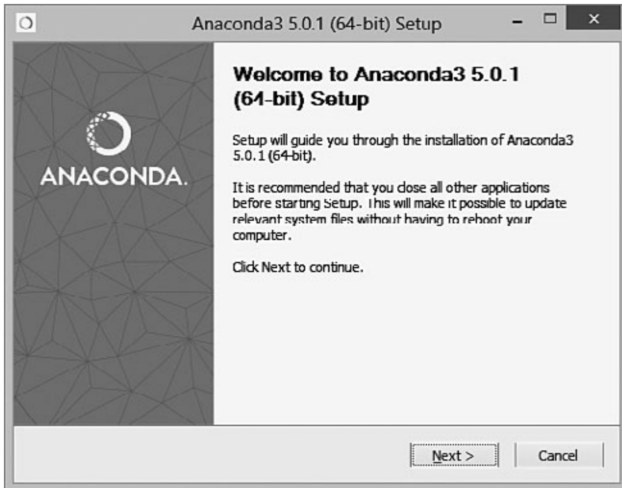
とすれば

＊1　PEP373 https://www.python.org/dev/peps/pep-0373/

Kyoto University, KYOTO 606-8501, Japan

author's e-mail: fujii@me.kyoto-u.ac.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research



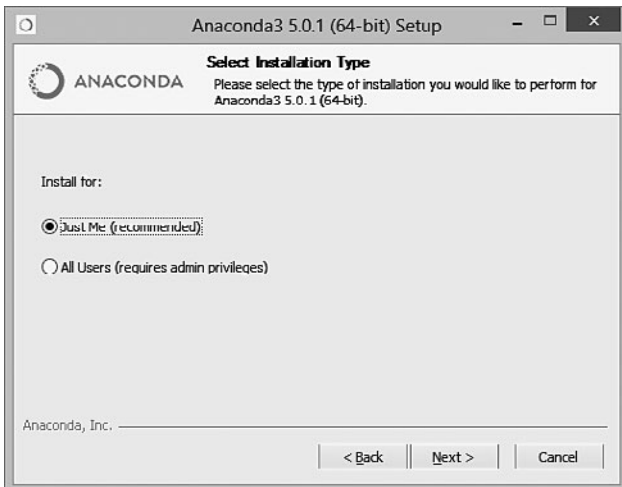


図1　ダウンロードしたインストーラを実行している様子.

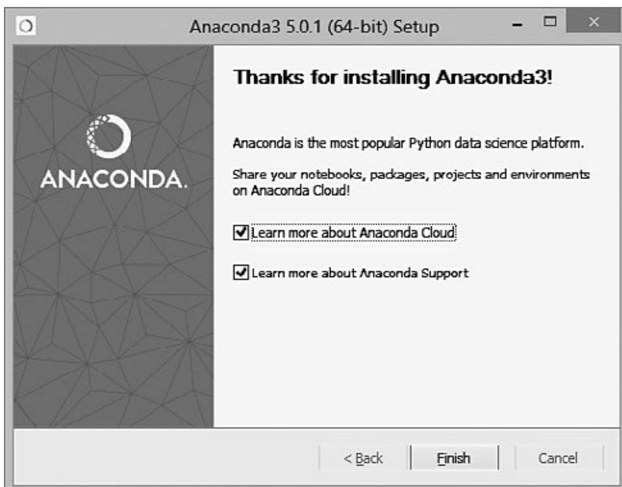


図2　インストール終了時のスクリーンショット.

```
/Users/[user_name]/anaconda/bin/python
```

となり，Default の python が MacOS の native の python（バージョン2.7）から Anaconda の Python へと変更されていることがわかります.

なお，既に homebrew をインストールしている場合，anaconda との衝突を避ける為に pyenv を先にインストールした方が良いようです.

```
brew install pyenv
pyenv install --list
```

でインストール可能な python の一覧が出てきますのでその中から anaconda3-..*を探し，

```
pyenv install anaconda3-*.*.*
```

とします.

2.1.1.3　Linux での開発環境構築

Anaconda distribution ダウンロードページ https://www.anaconda.com/download にアクセスし，Linux 版 Anaconda をダウンロードしてください.

Anaconda3-5.0.1-Linux-x86_64.sh というようなファイル名（5.0.1などの数字は Anaconda distribution のバージョン番号）のスクリプトファイルがダウンロードされます. 以下のようなコマンドを用いて，実行権限を付与して実行してください.

```
chmod +x Anaconda3-5.0.1-Linux-x86_64.sh
./Anaconda3-5.0.1-Linux-x86_64.sh
```

なお Linux 版 Anaconda でも，OS の Python 環境と切り離した環境を構築することが可能です. そのため上記コマンドは，管理者でなく一般ユーザの権限で実行することをおすすめします. ライセンスに同意すれば，インストールが始まります.

最後に Anaconda を Path に加えるか問われます. ここで yes を選択しておくとよいでしょう. これによりターミナルから Python を実行する際に Anaconda の Python が優先して選択されることになります. なお，ディストリビューションによっては一度ログインし直す必要があるかもしれないので注意してください.

以上で Linux における Python 開発環境の構築は終了です.

2.1.2　Anaconda distribution

上で紹介した Anaconda distribution は Anaconda Inc. が開発する Python および R 開発環境を提供するオープンソース・ソフトウェアです. 3-clause BSD License で提供されており，自由に利用することができます.

Anaconda distribution の主な特徴に

```
pycrypto
    A collection of both secure hash functions (such as SHA256 and RIPEMD160), and var
ious encryption algorithms (AES, DES, RSA, ElGamal, etc.).

pyopenssl
    A thin Python wrapper around (a subset of) the OpenSSL library.

kerberos (krb5, non-Windows platforms)
    A network authentication protocol designed to provide strong authentication for cl
ient/server applications by using secret-key cryptography.

cryptography
    A Python library which exposes cryptographic recipes and primitives.


Do you accept the license terms? [yes|no]
[no] >>>
```

図3　ターミナルからインストーラを実行している時の様子. ライセンス同意書に同意することでインストールが始まります.

図4 Anaconda を Path に加えるかを問われている画面．ここで yes を選択しておくとよいでしょう．

- 優れたパッケージ管理システム
- 簡単な仮想環境の構築

が挙げられるでしょう．これらの特徴のため，Anaconda distribution は Python の開発環境として非常に人気のあるものになっています．以下にその特徴を簡単に紹介します．

2.1.2.1 外部パッケージのインストール

Python では，言語の基本的な機能だけで実現できることは意外と少なく，実際にはほとんどの操作を外部のパッケージを用いて行うことになるでしょう．本講座でデータ解析を行う時も NumPy や Matplotlib など外部のパッケージを用いることになります．

様々なプログラミング言語のなかでも Python は特に外部パッケージが豊富であり，そのインストールも非常に簡単に行うことができます．現在10万種類を超える多種多様なパッケージが公開されており，NumPy，Matplotlib を含めたこれらパッケージのほとんどはオープンで開発が行われています．なお読者の方々も，プログラム開発に習熟すればこれらの活動に参加・貢献することも可能です．ぜひコミュニティに貢献しましょう．

上述の通りに Anaconda distribution をインストールすれば，NumPy，Matplotlib を含めた基本的なパッケージは自動的にインストールされます．しかし，Python に慣れてくれば，より専門的なパッケージを用いることも多くなることは間違いありません．そういった時には，新たにそれらのパッケージをインストールする必要があります．

ここでは例として，後の3章で紹介する多次元データ処理ツールである xarray を新たにインストールすることを考えます．なお少し詳細になりますが，Anaconda 環境でパッケージをインストールする方法は大きく2つあります．

- Python の持つパッケージインストールコマンド pip を用いる方法
- Anaconda の持つパッケージインストールコマンド conda を用いる方法

以下で少し触れるように conda の方が高機能であるため，こちらを用いるほうがよいでしょう．conda コマンドで新たなパッケージをインストールするためには，以下を実行してください．

```
conda install xarray
```

これにより Python 環境に xarray がインストールされます．なお，xarray は別のパッケージである Pandas を内部で用

図5 xarray を conda コマンドにより実行している様子．

いています（依存関係があります）．conda コマンドでは，そういった依存関係のあるパッケージも自動的にダウンロード・インストールされます．

インストールしたパッケージをバージョンアップするには

```
conda update xarray
```

アンインストールするには

```
conda uninstall xarray
```

を実行すればよいでしょう．また，現在の環境にインストールしているパッケージの一覧を確認するには，以下を実行してください．

```
conda list
```

その他のコマンドについては，Anaconda のマニュアルページ https://conda.io/docs/user-guide/tasks/manage-pkgs.html を参考にしてください．

2.1.2.2 Anaconda によるパッケージ管理

Python では他言語との連携が容易であり，それを前提としたパッケージも多数存在します．例えば，Python の最も基本的な数値計算パッケージである NumPy は，主に C 言語で書かれておりそれをパッケージ内部から呼び出しています．さらに NumPy は，Intel が提供する並列計算ライブラリ MKL と連携しており，行列計算などは自動的に並列化してくれます．他にも，データベースを操作する PostgreSQL など実際には別の言語で書かれているパッケージも数多くあります．

Python 自体はクロスプラットフォームな言語なので，OS 環境には依存しません．Windows で作成したスクリプトをほとんど何も改変せずに Mac で動かすことも可能です．しかし C 言語や Fortran などで OS のコンパイラを用いる場合や，並列化計算を実現するためにはその実装は OS に依存したものとなってしまいます．Anaconda は各プラットフォームに合わせたバイナリ・コンパイラを提供しており，conda コマンド一つで，それらパッケージのダウンロード・コンパイル等，必要なことを自動的に行ってくれる仕組みになっています．そのためユーザーは，OS の違いを気にすることなく，パッケージをインストールしたり，実行したりできるのです．

2.1.2.3 Anaconda による Python 仮想環境

Anaconda によって構築した Python 開発環境は，OS 内の環境とは独立した仮想環境になっています．例えば Anaconda 内でパッケージをインストールしても，OS の環境，他のユーザの環境に影響を与えません．そのため，管

理者権限を持たないコンピュータ上にもホストPCの環境を崩さずに開発環境をインストールすることができます．さらに，ユーザ各自が好きなパッケージをインストールすることができるため，個人のPCだけでなく，共同で用いる計算サーバでの利用にも適していると言えるでしょう．

もっと言うと，このような仮想環境を１つのPC内に複数構築することも可能です．例えば研究を進めていくと，あるパッケージの過去のバージョンでしか実行できないの古いプログラムを使いたいといった場合も出てくるでしょう．通常であれば，PC内のそのパッケージのバージョンを全て古いものに戻す必要がありますが，そうしてしまうとこれまで開発してきたスクリプトが動かなくなるなどトラブルが想定されます．

こういった場合には，これらのプログラムを動かす環境を普段使っている環境と隔離した仮想環境として構築することが有効でしょう．ある仮想環境でインストールしたパッケージは他の環境に影響を与えないため，その古いパッケージ専用の仮想環境を用意すれば，安全に利用することが可能です．

Anacondaでは，以下のコマンドを実行することでPythonの仮想環境を構築することができます．

```
conda create -n py27 python=2.7
```

このコマンドは，Python2.7が動くpy27という名前の仮想環境を作る，という意味です．このようにして作成した仮想環境py27をアクティブにするには，Windowsでは以下を

```
activate py27
```

Mac，Linuxでは以下を実行してください．

```
source activate py27
```

コマンドプロンプト・ターミナルに(py27)と表示されていると思います．これはpy27仮想環境がアクティブになっていることを示す表示です．

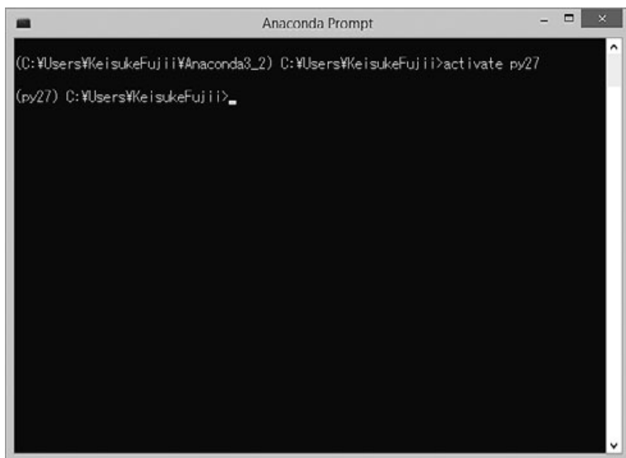


図6　Windowsで仮想環境py27をアクティブにする様子．

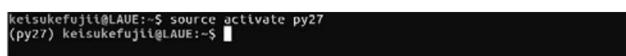


図7　Linuxで仮想環境py27をアクティブにする様子．

なお，上記コマンドで作成した仮想環境には，NumPyなどのパッケージはインストールされていません．以下に述べた方法によりパッケージをインストール・アンインストールすることが必要です．

なお，この仮想環境を非アクティブ化するにはWindowsでは以下を

```
deactivate py27
```

Mac，Linuxでは以下を実行してください．

```
source deactivate py27
```

2.2　対話的開発環境を用いたPython入門

2.2.1　対話的開発環境Jupyter-notebook

本節では，対話的プログラミング環境の１つであるJupyter-notebookを紹介します．Jupyter-notebookは

- 実行結果を可視化しながらデータ解析を進めることができる
- ソースコード，実行結果，グラフ，数式，文書を１つのファイルに保存することができる

などが特徴で，近年利用が大きく広まっています．

元々Jupyter-notebookはIPythonというPythonの対話的プログラム環境の一部のIPython-notebookとして開発が進められてきましたが，その有用性が認められ，現在はPythonに限らずJulia，R，rubyなど他のインタープリタ言語でも利用可能な汎用ソフトウェアとして開発が続けられています．

上記の特徴は，試行錯誤しながらすすめる研究活動を行うのに効果的なだけでなく，研究成果をデータ・解析コード・結果をまとめて公開するのにも有用です．例えば，2016年に重力波が初めて発見された時の計測データ・解析プログラムはhttps://losc.ligo.org/s/events/GW150914/GW150914_tutorial.htmlにJupyter-notebook形式で公開されており，誰でもその結果を追試することができるようになっています．

2.2.1.1　Jupyter-notebookのインストール

本講座で紹介したPython環境の１つであるAnacondaには，すでにJupyter-notebookが含まれています．インストールされていない別の環境などではターミナルから

```
conda install jupyter-notebook
```

もしくは

```
pip install jupyter-notebook
```

を実行することでインストールできます．

2.2.1.2　Jupyter-notebookの起動と終了

Windowsでの起動

Windowsから Jupyter-notebookを起動するための手順は，以下の通りです．

1．コマンドプロンプトを立ち上げる（必要に応じてディレクトリに移動する）
2．起動コマンド jupyter-notebook を入力する

これにより，自動的にブラウザが立ち上がり，Jupyter-notebook のホーム画面が表示されます．

Mac，Linux での起動

Mac，Linux の場合も同様に

1．ターミナルを立ち上げる（必要に応じて適宜ディレクトリを移動する）
2．起動コマンド jupyter-notebook を入力する

上記操作を行うことで，1章の**図1**と同様の画面がブラウザに表示されます．なお，ブラウザを誤って閉じてしまった場合も，http://localhost:8888 にアクセスすることで，Jupyer-notebook のホーム画面を再度表示できます．

ノートブックファイルの新規作成

Jupyter-notebook のホーム画面では，ディレクトリがツリー表示されています．フォルダ間の移動，ファイル・フォルダの名前の新規作成・名前の変更・削除など，シンプルなファイル操作はJupyter-notebook内で一通りできるようになっています．ディレクトリを適宜移動し，ノートブックファイルを新たに作成して学習を始めましょう．

前節で体験したように，右上の New から Python 3 を選択すると，新しいノートブックファイルが作成され新しいウィンドウとして開かれます．作成されるファイル名はデフォルトでは Untitled となっており，ページの最上部に表示されています．この部分をクリックするとファイル名を変更するダイアログが表示されますので，適宜わかりやすい名前に変更してください．

Jupyter-notebook の終了

Jupyter-notebook には，個別のノートブックファイル（カーネル）の終了と Jupyter-notebook 自体の終了の2つのレベルの終了があります．カーネルとは Python を実行しているコアの部分のことであり，オブジェクトの内容をメモリ格納していたり，何か計算をしていたりします．ブラウザ画面を閉じるだけではカーネルは終了されません．つまり，データはメモリ内に残されたままになっています．ツールバーの File > Close and halt を実行することでカーネルを終了してください．もしくは，Jupyter-notebookホーム画面のRunningタブには，カーネルが実行中であるノートブックファイル一覧が表示されています．そこで該当するノートブックファイルの shutdown ボタンを押すことでも，カーネルを停止することができます．

Jupyter-notebook 自体を終了するには，起動したコマンドプロンプトもしくはターミナルにて Ctrl+C を入力してください．

2.2.2 Python 入門

ここではJupyter-notebookを使って，Pythonの文法を学びます．

2.2.2.1 Python の基本変数

C 言語や Fortran のようなコンパイラ言語と異なり，変数（オブジェクト）を定義する際にその型を指定する必要がありません．

```
In [1]: x = 'Hello python!'

In [2]: y = 2

In [3]: x
Out[3]: 'Hello python!'

In [4]: y
Out[4]: 2
```

上記の場合 x は文字列(str)であり，y は整数(int)となります．オブジェクトの型を知りたい場合は，

```
In [5]: type(x)
Out[5]: str

In [6]: type(y)
Out[6]: int
```

を実行してください．なお，type()関数は，引数のオブジェクト型を返す関数です．

Python の基本的な型には主に以下のものがあります．

- 整数（int）
- 浮動小数点実数（float）
- 文字列（str）
- リスト（list）
- タプル（tuple）
- 辞書（dict，別名：連想配列）

Python はオブジェクト指向の言語であり，実際には後ほど説明する NumPy などのパッケージに含まれる上記とは別の型（オブジェクト）を多用することになりますが，ここではまず，上記の基本型を学びましょう．

整数・浮動小数点

Python は他のプログラム言語と同様，整数や浮動小数点といった型をサポートしており，四則演算やべき乗等は一般的な書き方で計算できます．

```
In [7]: 1 + 3
Out[7]: 4

In [8]: 3.0 ** 2.0 # ** はべき乗を表します.
Out[8]: 9.0

In [9]: (1.0 + 3.0j) * 2.j # 複素数は，数字の末尾に j を付けることで表すことができます.
Out[9]: (-6+2j)
```

文字列

str は文字列を格納するオブジェクトです．文字をシン

グルコーテーションもしくはダブルコーテーションで囲うことで文字列となります．なお，3連のシングルコーテーションで囲うと，改行を含めた文字列として扱うことができます．

strオブジェクトの各要素にアクセスするためには，[]演算子を利用してください．このように，連続的なオブジェクトに添字を用いてアクセスする方法をIndexing（インデクシング）と呼びます．

```
In [10]: x = 'Hello python!'

In [11]: x[0]
Out[11]: 'H'

In [12]: x[2]
Out[12]: 'l'

In [13]: x[-1]
Out[13]: '!'
```

Pythonのインデクシングには，以下の特徴があります．

- 一番最初の要素は0番目として数えられる（C言語と同様．Fortranと異なる）
- 負のインデクスを指定することで，末尾から数えることもできる．例えば，インデクスとして-1を指定すると，末尾の要素，-2を指定すると　末尾から2つ目の要素にアクセスでます．

さらに，同時に複数の要素にアクセスするスライシング(Slicing）も可能です．

```
In [14]: x[1:4]
Out[14]: 'ell'

In [15]: x[:5]
Out[15]: 'Hello'

In [16]: x[-3:]
Out[16]: 'on!'
```

1:4という表記は，（0から数えて）1～3番目の要素を示し，元のオブジェクトの部分要素が選択されることになります．また，片方を省略した:5などは，0:5と同じ意味です．

ここでxは文字列（str）ですが，文字列は以下のように+記号で結合できます．

```
In [17]: x2 = x + 'I love you !'

In [18]: x2
Out[18]: 'Hello python! I love you !'
```

リスト

リスト（list）は複数のオブジェクトを格納するオブジェクトです．[]内に複数の要素をコンマ区切りで記述することで，リストオブジェクトを作成することができます．なお，それぞれのオブジェクトは型が異なっても問題ありません．

```
In [19]: z_list = [x, 3.0, x2]

In [20]: z_list
Out[20]: ['Hello python!', 3.0, 'Hello python! I love you !']
```

リストも，文字列の場合と同様に，インデクシングやスライシングに対応しています．

```
In [21]: z_list[-1]
Out[21]: 'Hello python! I love you !'
```

.appendを用いることで，リストの末尾に新しいオブジェクトを追加することができます．

```
In [22]: z_list.append(5.0)

In [23]: z_list
Out[23]: ['Hello python!', 3.0, 'Hello python! I love you !', 5.0]
```

また，要素数はlen関数により知ることができます．

```
In [24]: len(z_list)
Out[24]: 4
```

タプル

タプル（tuple）はリストと似ていますが，要素数が後から変更できないという点でリストと異なります．()内に複数の要素を記述するか，あるいは単純にコンマ区切りで記述することで，タプルオブジェクトを作成することができます．

```
In [25]: t = (1, 3.0, x)

In [26]: t
Out[26]: (1, 3.0, 'Hello python!')

In [27]: a, b = 2.0, 3.0 # (a, b)に(2.0, 3.0)を代入しています
In [28]: a, b
Out[28]: (2.0, 3.0)
```

辞書

辞書は連想配列とも呼ばれ，リストと同様に複数の要素を格納できるオブジェクトです．ただし，引数（キーと呼ぶ）に任意の（より厳密には，ハッシュ可能な）オブジェクトを用いることができる点でリストと異なります．なお一般的には以下のように，文字列をキーにする場合が多いでしょう．

```
In [29]: d = {'a': 1.0, 'b': 3.0}
```

```
In [30]: d['a']
Out[30]: 1.0

In [31]: d['c'] = 5.0  # 新しい要素を追加するには，単に新しいキーを指定して値を代入してください．

In [32]: d
Out[32]: {'a': 1.0, 'b': 3.0, 'c': 5.0}
```

上記のように，{ }内に，キーと要素を：で対応させて記述することで辞書型のオブジェクトを作成できます．

2.2.2.2 Python の基本文法

この節では，Python の主な文法を簡単に紹介します．C 言語や Fortran などでは例えば if 文の及ぶ範囲を{ }や IF-END IF で囲って表記します．一方で Python ではそれらをインデントで表します．なおインデントには 4 つのスペースを用いることが一般的です．

if 文

Python の if 文は以下のように，if [条件]：と書き，条件が真の場合に実行する内容を次の行から新たなインデントを用いて記述することになります．

```
if a < b:
    print(a)
```

while ループ

while ループも同様に，繰り返し実行する内容をインデントにより区別して記述します．

```
while a < b:
    a += 1
    print(a)
```

for ループ

Python の for ループは，C 言語や Fortran の do ループと少し異なります．C 言語や Fortran では整数を 1 ずつ増やしながら実行することが多いですが，Python ではリスト（やタプル）を 1 つずつ選択しながら，全ての要素に対して操作を繰り返すことになります．

```
In [33]: for z in z_list:
    ....:     print(z)
    ....:
Hello python!
3.0
Hello python! I love you !
5.0
```

上記は，z_list の各要素を 1 つずつ z に代入してインデントで表されたコードブロックを実行する，という操作を z_list の最初の要素から最後の要素まで繰り返しています．

C 言語や Fortran の for ループと同様の操作は，0 から指定した値までの整数を順に並べたオブジェクトを返す range 関数を用いることで実現できます．

```
In [34]: for i in range(len(z_list)):
  ....  :     print(z_list[i])
  ....  :
Hello python!
3.0
Hello python! I love you !
5.0
```

内包表記

ソースコードは一般的に，短いほど可読性が上がります．Python には内包表記と呼ばれる記述方法があります．これは，ループ構造を簡略的に記述するもので，簡単な操作を行う時によく用いられます．

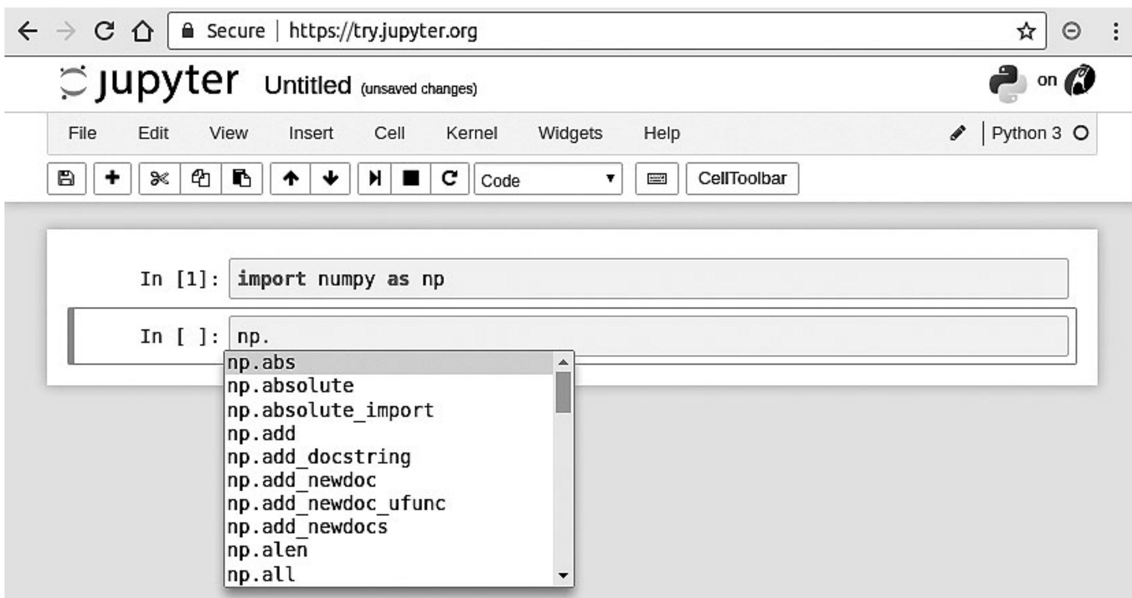


図 8　Jupyter-notebook のサジェスト機能の様子．np．まで入力してから，Tab を押すことで np パッケージ内でアクセスできるものが表示できます．

```
In [35]: x_list = [1.0, 2.0, 4.0]

In [36]: y_list = [x**2 for x in x_list]

In [37]: y_list
Out[37]: [1.0, 4.0, 16.0]
```

上記の例の2行目は，x_listから1つ要素を取ってきてxに代入し，それぞれの二乗の値を要素にもつリストをy_listに代入する，という意味であり，以下を一行で記述したものに相当します．

```
In [38]: y_list = []

In [39]: for x in x_list:
 ....   :     y_list.append(x**2)
 ....   :
```

2.2.3 外部パッケージの使用

前節でも述べたように，多様なパッケージ（いくつかの機能をまとめたライブラリ）が豊富に開発されていることがPythonの大きな特徴の1つです．

実際には生のPythonだけで用いることは少なく，基本的な数値計算機能を提供するNumPy*2やグラフ描画ライブラリであるMatplotlib*3を始めとして外部パッケージを多用することになると思います．

2.2.3.1 NumPyを利用する

NumPyやMatplotlibはデータ解析における最も基本的なパッケージです．前節の方法に従ってAnacondaをインストールした環境では，これらは既にインストールされているので，すぐに使い始めることができます．

こういったパッケージを　各自で開発するスクリプトで用いるためには，以下のようなimport文によりその使用を宣言する必要があります．

```
In [1]: import numpy as np
```

ここでimport numpy as npは，「NumPyパッケージをnpという名前で用いる」という宣言です．なお，asの後ろの名前はユーザが勝手に決めてよいものですが，混乱を避けるため，広く用いられている略称を用いることが望ましいでしょう．NumPyの場合，npが正式な略称です．このようにしてインポートした後は，その機能をnp.***という形で用いることになります．

NumPyの代表的な関数

NumPyを用いることで，多くの種類の算術演算を行うことができます．例えばsin関数は以下のようにして用います．

```
In [2]: x = np.sin(0.5 * np.pi)

In [3]: x
Out[3]: 1.0
```

NumPyでは非常に多くの種類の関数やクラスが用意されています．そのためどのような関数が用意されているかを把握することも難しく，それらの使用法をすべて暗記することはほとんど不可能でしょう．

NumPyなどのオープンソースソフトウェアでは，開発に際して，使用法などの文書を同時に残していく文化が形成されており，ユーザがある関数の使い方を知りたいと思った場合もすぐにその情報にアクセスできるようになっています．

Jupyter-notebookでは，以下のようにnp.まで記入してからTabを押下すると，np.内にある関数一覧が表示されるほか，np.sまで記入してからTabを押下するとそれに合う候補を表示してくれます．また，使用法がわからない関数でも，カーソルが括弧内にあるときにShift + Tabを押下することで，それぞれの関数の使い方に関する文書（docstringsと言う）を表示させることができます．これらを読むことで，新しい関数でもその使い方をすぐに理解することができるでしょう．

さらに，科学技術用途以外も含めPythonは広く用いられている汎用言語なので，インターネットで検索するだけでも多くの情報を見つけることができるのも特徴です．

多次元配列型 np.ndarray

NumPyは，多次元配列用のクラス（クラスについては後で少し紹介します）であるnp.ndarrayを提供しています．（なお，"nd" arrayは，n-dimensionalの略です）．np.ndarrayは配列の大きさを後から変更できない，全ての要素の型が同一なものに限られる，という点はリストと異なりますが，同様にインデクシング・スライシングに対応しています．

np.ndarrayは，多次元配列の基礎となるクラスで，線形演算を含む多くのNumPy関数で利用される他，pandasなど他のライブラリでも広く利用されている基本的なオブジェクト形式となっています．詳しくは3章で紹介しますが，ここではその利用法について簡単にだけ触れることにします．

np.ndarrayを定義するためには，np.ndarrayから用意するか，np.onesやnp.linspaceなどの関数を用いることになります．

```
# [5 x 3 x 2]の大きさの配列をxとして確保する．
In [4]: x = np.ndarray((5, 3, 2))
```

```
# [2 x 3] の大きさで，要素がすべて1のint型の配列をyとして確保する．
In [5]: y = np.ones((2, 3), dtype=int)

In [6]: y
```

＊2　NumPy, NumPy developers, http://www.NumPy.org/

＊3　Matplotlib, John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team, https://matplotlib.org/

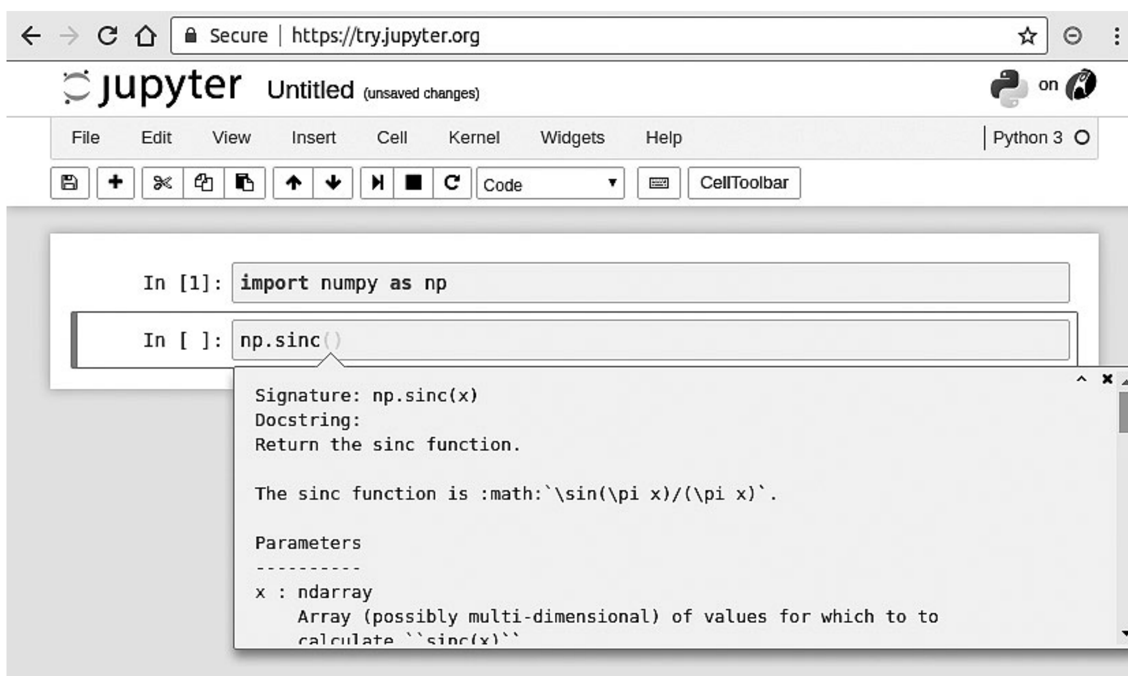


図 9　Jupyter-notebook による docstring 表示の様子．関数内部（括弧"()"の内側）で Shift + Tab を入力することで，関数名，引数，docstring を表示できます．

```
Out[6]:
array([[1, 1, 1],
       [1, 1, 1]])
```

np.ndarray とスカラー，np.ndarray 同士の計算は，要素ごとの計算として定義されています．（broadcast）と呼ばれています．

```
In [7]: y * 3
Out[7]:
array([[3, 3, 3],
       [3, 3, 3]])

In [8]: y + y
Out[8]:
array([[2, 2, 2],
       [2, 2, 2]])
```

また，np.abs() や np.square() などスカラーを引数に持つ関数に渡した場合は，要素ごとに該当する演算が行われた np.ndarray が返されます．

```
In [9]: np.sin(y)
Out[9]:
array([[0.84147098, 0.84147098, 0.84147098],}
       [0.84147098, 0.84147098, 0.84147098]])
```

2.2.3.2　Matplotlib を利用する

Matplotlib は，広く用いられているグラフ描画ライブラリです．Matlabのグラフ描画機能を参考にして開発されたようで，よく似た命名規則を持っています．Matplotlib の詳しい使い方自体も次章に譲ることにして，ここでは単純な描画方法についてのみ述べます．

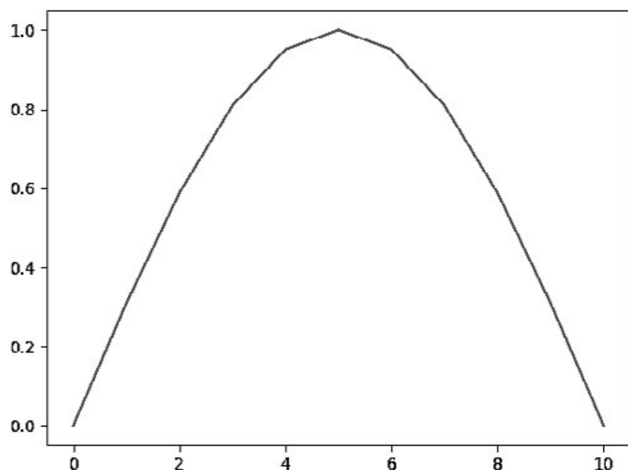
Jupyter-notebook 内で Matplotlib を表示するには，以下を宣言します．

```
import matplotlib.pyplot as plt
%matplotlib inline
```

import matplotlib.pyplot as plt は，matplotlib パッケージの中の pyplot モジュールを plt という名前で用いるという意味です（パッケージ，モジュールなどの厳密な定義は，後の「Python の階層構造」を参考にしてください）．また matplotlib inline は Jupyter-notebook 用のコマンドであり，コードセルのすぐ下に Matplotlib の図を表示させるためのものです．

以下のように，plt.plot の引数に 1 次元データを渡すことで，横軸が要素番号，縦軸が要素の値のグラフを描画できます．

```
In [10]: x = np.linspace(0,1,11) #0～1 を11等分した要素を持つ np.ndarray を返す関数

In [11]: y = np.sin(np.pi * x)
In [12]: plt.plot(y)
Out[12]: [<matplotlib.lines.Line2D at 0x7f6201e95128>]
```



2.2.4 関数を定義して使う

これまで，np.sinを始めとする関数を用いてきました．このように，何か操作をして値を返すもの（ファイルを保存するなど値を返さないものもある）を関数と呼びます．本節ではこのような関数を定義して用いる方法について述べます．

なお，もしNumPyなどの外部パッケージが同様の操作を行う関数を定義している場合はそちらを用いる方がよいことがほとんどです．既存のパッケージは，大勢で作成・動作確認しているため，個人が作成したものよりもバグが圧倒的に少なくなっています．さらにNumPyのように，CやFortranを裏で利用して高速化を行っていることも多いでしょう．

以下に例として，極座標変数(r, theta)から直行座標での値(x, y)への変換を行う関数を定義してみます．

```
In [1]: def get_xy(r, theta):
   ...:     """
   ...:     Returns x, y values from polar coordinate variable
            r (radial coordinate) and theta (angular coordinate).
   ...:     """
   ...:     x = r * np.cos(theta)
   ...:     y = r * np.sin(theta)
   ...:     return x, y
   ...:
```

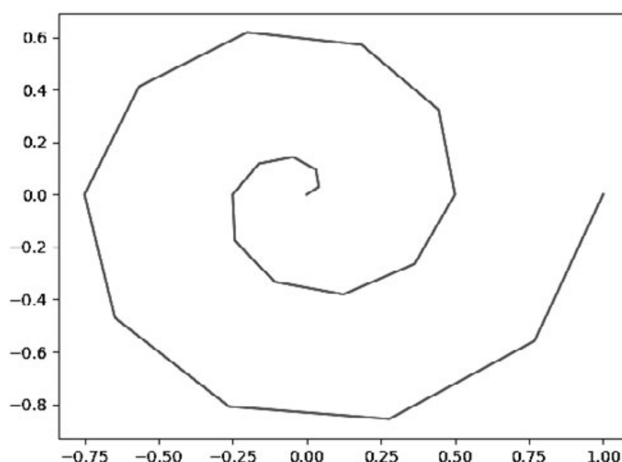
上記のように，関数の定義はdef文から始め，関数名（get_xy）の次にカッコ内に引数（r, theta）を指定し，コロンの後に改行します．関数の範囲はインデントにより示すことが必要です．なお，慣習として関数名は小文字で始めることになっています．

また動作上必須ではありませんが，def文の次の行にはその関数の説明，操作の内容や引数・戻り値の意味などを３つのダブルコーテーションで囲った文字列として書いておくことが推奨されています（この説明文はdocstringsと呼ばれます）．docstringsを記述しておくことで，メンテナンス時に関数の役割を思い出しやすいほか，Jupyter-notebookなどの開発環境ではその使用法をソースコードを読むことなく理解できるという利点があります．自身で使うだけのプログラムでも，簡単に記述しておくことが重要でしょう．

関数の内部（インデントで示される領域）では必要な計算を行い，必要であればreturn文で値を返します．なお，Pythonの関数は複数の値を返すことも可能です．その場合，戻り値はそれらを含むタプルとなります．

定義した関数は以下のように，引数にオブジェクトを渡すことで実行します．

```
In [2]: r = np.linspace(0, 1, 21)
        # 0 ～ 1 を 21 等分した点列

In [3]: theta = np.linspace(0, 4.0 * np.pi, 21)
        # 0 ～ 4(π) を 21 等分した点列

In [4]: x, y = get_xy(r, theta)

In [5]: plt.plot(x, y)
Out[5]: [<matplotlib.lines.Line2D at 0x7f6201e370b8>]
```



2.2.5 スクリプトファイルを読み込む

上記の関数など，作成したスクリプトを再利用するためには，スクリプトファイルとして作成しておくほうが良いでしょう．

スクリプトファイルと言っても，これまで学習した内容を.pyファイルに保存するだけです．なお，import文や変数の定義はファイルをまたいでは引き継がれないため，スクリプトファイルごとにそれらを適宜記入する必要があるでしょう．

前節で定義した関数をスクリプトファイルとして保存するには，以下のような内容をファイルとして保存してください．今回は，これをpolar.pyとして保存したとしましょう．

```
import numpy as np

def get_xy(r, theta):
    """
```

```
    Returns x, y values from polar coordinate variable r
    (radial coordinate) and theta (angular coordinate).
    """
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x, y
```

このスクリプトファイルを読み込むためには，以下のようにimport文を用いてください．

```
import polar

r = np.linspace(0, 1, 31)
theta = np.linspace(0, 4.0 * np.pi, 31)
x, y = polar.get_xy(r, theta)
```

なお，import文で読み込めるスクリプトファイルは，実行するスクリプトと同じディレクトリ内かパスの通ったディレクトリのみとなります．任意の場所にあるスクリプトファイルを読み込むためには

```
import sys
sys.path.append('path/to/script')
import polar
```

というように，Pythonのカーネルからパスを通す必要があることに注意してください．

2.2.6 Pythonスクリプトの階層構造

Pythonはオブジェクト指向の言語であり，いくつかの階層構造があります．例えば，NumPyなどのライブラリは，「パッケージ」と呼ばれる大きなコード群として提供されています．また各パッケージには，「モジュール」と呼ばれる小さなコード群が複数含まれることが多くなっています．例えば，NumPyの線形代数機能はlinalgという名前のモジュールで提供されています．

さらに，各モジュールには「クラス」が定義されていることがあり，その中に変数や関数（クラスに付属する関数はメソッドと呼ばれる）が含まれています．

クラスは，オブジェクト指向プログラミングに必要な重要な概念ですが，発展的な内容を含むため，ここでは詳しくは紹介しません．ただし多くのライブラリで用いられることが多いため，その使用法だけを簡単に紹介します．

2.2.6.1 パッケージ内・モジュール内へのアクセス

これまでNumPyのsin関数を実行する際，np.sin()を実行しました．これは，NumPyパッケージ内に定義されているsin関数を呼び出すための記法です．ここで「.」は，1つ下の階層への移動を意味するもので，この例の場合「パッケージ名.関数名」という形式でアクセスしていることになります．

同様に，NumPyパッケージ内のfftモジュール内の実数に対する高速フーリエ変換rfftを実行するためにはnp.fft.rfftというように「パッケージ名.モジュール名.関数名」と言う形式でアクセスする必要があります．

2.2.6.2 クラスの使い方 np.ndarray

クラスはオブジェクト指向の重要な概念です．その定義を説明する前に，ここではまずその一例であるnp.ndarrayを紹介します．

```
In [1]: x = np.linspace(0,1.1,12)

In [2]: x
Out[2]:
array([0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
       0.8, 0.9, 1., 1.1])
```

ここで，xは12個の要素からなる1次元ベクトルです．

言葉の定義を少し明確にしておきましょう．「クラス」は型名のことであり，ここでxのクラスはnp.ndarrayです．一方，xはnp.ndarray型を持つ変数です．オブジェクト指向の言語ではこの変数のことを「オブジェクト」や「インスタンス」と呼びます．

各オブジェクトは，内部に変数や関数などの別のオブジェクトを有していることが多々あります．例えばnp.ndarrayのオブジェクトは，アレイの大きさや次元数などを別のオブジェクトとして有しています．

```
In [3]: x.shape # xの形状
Out[3]: (12,)

In [4]: x.ndim # xの次元数
Out[4]: 1
```

このように，オブジェクト内部のオブジェクトには「オブジェクト名.オブジェクト名」というように「.」を使うことでアクセスできます．

クラスは，それ自身の値を操作する関数（メソッドと呼ぶ）を有していることも多いでしょう．例えばnp.ndarrayの場合，reshapeメソッドを実行することで12要素を持つ1次元のnp.ndarrayを3x4要素を持つ2次元行列に並び替えることができます．このようなメソッドにも「.」を用いてアクセスできます．

```
In [5]: y = x.reshape(4,3)
In [6]: y.shape
Out[6]: (4, 3)

In [7]: y
Out[7]:
array([[0. , 0.1, 0.2],
       [0.3, 0.4, 0.5],
       [0.6, 0.7, 0.8],
       [0.9, 1. , 1.1]])
```

このreshapeのように，一般的にメソッドは，自分自身（ここではオブジェクトx）と，それに加えて引数（ここでは4, 3）を受け取り，何か戻り値（ここでは別のnp.ndarray型のオブジェクトy）を返す関数になっています．

本講座では，クラスをどのようにして定義するかなどは

省略しますが，クラスはデータ（上の例では x に格納されている値）と機能（reshape などのメソッド）をひとまとめにしたものであると理解できれば，外部パッケージの使用には十分でしょう．

2.2.7 まとめ

本節では，Jupyter-notebookを用いることでPythonの基本的な文法と使用法を概観しました．Jupyter-notebookのような対話的な環境では書き間違いなどによるバグがすぐに見つかるため，特にプログラミングに不慣れな人にとって効果的な学習環境となるでしょう．

本節では，Python の基本的な文法を紹介したのみで，例えばクラスの定義法やオブジェクト指向プログラミングなどは詳しく述べませんでした．しかし Python には，Numpy や Matplotlib などの多様なパッケージが用意されており，一般的なデータ解析では，それらで定義されたオブジェクトを使用するだけで十分なことが多いでしょう．

データ解析では試行と可視化の繰り返しが重要です．特に，新しい実験データ・シミュレーションデータが得られた時のように確立した解析法がないデータから意味を抽出する必要がある場合には，精緻なプログラムを構築するというよりは，外部パッケージを使っていろいろな解析を様々に試してその結果を可視化する，というサイクルを高速に回す方が効果的であることが多いと思います．

これから Python を学ぶ読者も，その厳密な文法を学ぶ前に，以降の章で説明する外部パッケージを用いてまずは実際のデータを解析してみて，その結果を見ながら徐々に言語に慣れていくという方針をとるとよいように思います．

藤井恵介（ふじい けいすけ）

京都大学工学研究科専門は光計測，機械学習．2012年３月京都大学工学研究科博士後期課程修了．豊富に蓄積されているプラズマ実験データの有効利用を目指して，統計的手法・機械学習の導入に奮闘しています．
趣味ではオープンソースソフトウェアの開発にも参加しています？ https://github.com/fujiisoup.プラズマ・核融合分野でもオープンソース・ソフトウェアの文化が広まればいいですね．

J. Plasma Fusion Res. Vol.94, No.3 (2018) 141‑149

講座 今日からはじめる Python

3．Python による科学技術計算

3. Scientific Computing in Python

3.1　NumPy/SciPy によるデータ解析

3.1 NumPy/SciPy for Data Analysis

釼 持 尚 輝
KENMOCHI Naoki
東京大学大学院　新領域創成科学研究科

（原稿受付：2018年１月23日）

NumPyはFortranのような多次元配列と科学技術計算をサポートするライブラリです．これにより配列全体への演算が可能となります．本章では，実際にプラズマ実験のデータ解析を行いながら，配列の読み書き，作成，配列演算について解説します．さらに，高度な科学技術計算に関する様々な機能を提供するパッケージであるSciPyについても，使い方を紹介します．例として，簡単なシミュレーションもできるSciPyパッケージのodeintモジュールを用いた微分方程式の数値解法例を示します．

Keywords:
python, numpy, scipy, simulation

3.1.1　はじめに

本章までに，Python で簡単な解析やプログラムの構築はできるようになりました．次にみなさんが行いたいのは，実験データの解析やシミュレーションでしょう．そこで，Python における科学技術計算の基礎ライブラリである NumPy の使い方を，実際の実験データを解析しながら紹介します．更に，高度な数学的アルゴリズムを提供する SciPy に関する簡単な紹介とともに，シミュレーションの例としてPredator-Preyモデルの簡単な解析例を示します．本章で興味を持たれた方は，NumPy/SciPy の公式 HP [1,2]や，近年充実してきている日本語書籍[3,4]を参考にして，より理解を深めていただければと思います．

3.1.1.1　NumPy/SciPy とは

NumPy

NumPy は"Numerical Python"の略語で，科学技術計算やデータ分析のための基本的なパッケージです．Python は一般に，C や Fortran 等のコンパイラー型言語と比較して性能が犠牲になっています．そこで NumPy では多次元配列 ndarray を C 言語で実装することで，使い勝手の良いインターフェイスを提供しつつも高速な演算を実現しています．特に，配列のデータをシステムのメモリ（RAM）に隙間なく配置することで，以下のような理由により高速化することができています．

・データをCPUレジスタにまとめて効率的に読み出せる
・スライス，転置などの操作を実際にデータをコピーせずに実現できる
・CPU のベクトル化演算の恩恵を受けられる

SciPy

SciPy はNumPyを利用するパッケージであり，高速フーリエ変換，最適化，数値積分，信号処理などの科学技術計算に関する機能を提供しています．SciPy では，Fortran のプログラムで実装された多くの関数群を提供しており，Python のスクリプト言語としての機能を大幅に強化しています．このおかげで，Python が科学技術計算において MATLAB，IDL，Octave，及び Scilab に匹敵するシステムになっています．

SciPy は，以下の表に示すような多くのサブパッケージ群から構成されています．

cluster	ベクトル量子化／K 平均法
constants	物理／数学定数
fftpack	FFT の関数
integrate	積分と常微分方程式ソルバー
interpolate	内挿とスムージングスプライン
io	データ入出力
linalg	線形代数ルーチン
ndimage	N 次元画像パッケージ
odr	直交距離回帰（Orthogonal Distance Regression）
optimize	最適化及び解探索ルーチン
signal	信号処理
sparse	疎行列と関連する関数
spatial	空間データ構造とアルゴリズム

Graduate School of Frontier Sciences, The University of Tokyo, Kashiwa, CHIBA 277-8561, Japan
author's e-mail: kenmochi@ppl.k.u-tokyo.ac.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research

special	任意の数学特殊関数
stats	統計分布，統計関数
weave	C/C++統合

これらの機能を全て紹介することはできませんが，SciPyの公式ドキュメント[2]を閲覧するか，Pythonのインタラクティブシェルでヘルプを表示させることで詳細を調べることができます*1.

NumPy と SciPy の関係

SciPyのDocstringの冒頭には，「SciPyはNumPyの名前空間から全ての関数をimportし，加えて以下のサブパッケージを提供する」ということが書かれています．つまり，SciPyをimportすると，基本的に全てのNumPy関数を使えるようになります．ただし，SciPyの関数はNumPyの同一関数よりも最適化されていたり，機能が拡張されている場合が多いため，両者に関数が存在する場合には，SciPyの関数を用いるほうが計算速度の面で有利なことが多いです.

3.1.1.2 NumPy/SciPy の利用

それでは早速，NumPy/SciPyを使っていきましょう．第1章を参考にanacondaを使ってPythonをインストールした人は，既にNumPyは入っていると思います．そこで，まずはNumPy/SciPyの有無を確認してみます.

Python コンソールで

```
In [1]: import numpy

In [2]: import scipy
```

と打ってみてエラーがなければ無事にインストールされています．`No Module Named numpy`のようなエラーが出る場合は，ターミナルで以下のコマンドを入れてインストールしてください*2.

```
$ conda install numpy

$ conda install scipy
```

NumPyのインストールが完了したら，プログラム中で使用するためにimportを行います．外部パッケージの使用に関する詳細は第2章を参照してください．NumPyをimportするには，プログラム冒頭で以下のように宣言します.

```
In [3]: import numpy

In [4]: from numpy import *
```

`from` モジュール名 `import *`というコードは，既にスコープに存在する変数を知らない間に上書きしてしまう恐れがあります．そのため，本章ではNumPyの呼び出しは

```
In [5]: import numpy as np

In [6]: import scipy as sp
```

に統一してあります．読者の皆さんにも`np.`**関数名**での呼び出し記法を強く推奨します.

3.1.2 NumPy/SciPy を用いた実験データ解析

NumPy/SciPyを使う準備ができましたので，実際にプラズマ実験で得られたデータに対して解析をしてみましょう．ここでは，東京大学が所有する磁気圏型プラズマ装置RT-1[5]において得られた2視線のマイクロ波干渉計のデータを例にします．今回解析対象とする実験では，変化が分かりやすいように時刻$t=2.0$ secに5 msec間のガスパフ入射を行っています.

なお，今回の記事で紹介する計測データにはhttps://github.com/PlasmaLib/python_tutorial/tree/master/dataからアクセスできます．ぜひ自身のPCにダウンロードして，実際に手を動かして操作感を感じていただければと思います.

3.1.2.1 実験データの読み込み

まずは実験データを読み込んでNumPyの配列を生成します．NumPyではファイル形式にバイナリとテキストを選びファイルの読み書きを行うことができますが，ここでは`np.loadtxt`を使用してテキスト形式で保存されている実験データを読み込んでみます.

```
In [1]: IF = np.loadtxt(”data/IF_20170608_74_raw.txt”, delimiter=’,’)
```

NumPyにおけるテキスト形式での読み書きには，以下の特徴があります.

・他のアプリケーションと互換性のある.dat，.csv，.txt形式のファイルの読み書きができる
・保存できる配列の次元は2次元まで

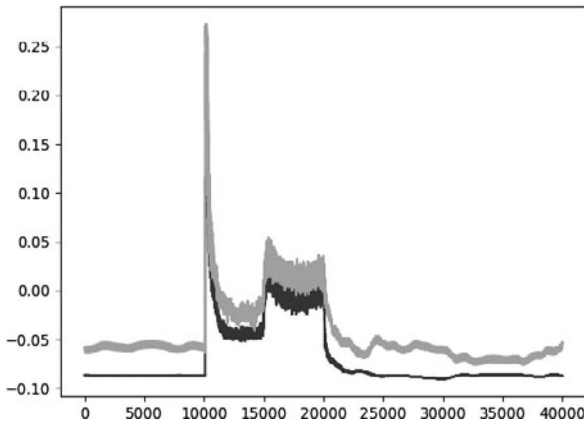
なお，バイナリ形式の読み書きには，`np.load`，`np.save`，`np.savez`，`np.savez_compressed`を使います．これらの関数は3次元以上のndarray配列も効率的にそのまま保存できますが，扱うファイル形式（.pickle，.npz，.npy）に他のアプリケーションとの互換性が殆どないことに注意が必要です．バイナリ形式での読み書きに関する詳細は，公式HP[6]を参照してください.

読み込んだデータの確認のため，Pythonで広く用いられるグラフ描写ライブラリであるMatplotlibを使って，グラフに表示してみます．Matplotlibの詳細は次章に譲るとして，ここでは以下のようにMatplotlibを読み込んでおきます.

```
In [2]: import matplotlib.pyplot as plt


In [3]: plt.plot(IF)
Out[3]:
[<matplotlib.lines.Line2D at 0x7f67d7503dd8>,
 <matplotlib.lines.Line2D at 0x7f67d74ff4a8>]
```

＊1 たとえばscipy.linalgのヘルプを表示させたい場合は，IPythonなどで`scipy.linalg?`と入力すればヘルプを参照することができます.

＊2 `$ pip install numpy`や`$ pip install scipy`でもインストールはできますが，condaを使うとIntel製の高性能行列ライブラリMKLが使えるようになるため，自動的に全てのコアを使って計算してくれるようになります.



3.1.2.2 配列の生成

次に，上図の時間軸を表示するための配列変数を作成します．時間軸のような等差数列の生成には，`np.arange` や `np.linespace` を使用します．なお，RT-1のマイクロ波干渉計では，時刻 $t = 0.5$ sec から $t = 4.5$ sec まで，サンプル周波数 10 kHz でデータ収集を行っています．

```
In [4]: sampling_time = 1.0e-4

In [5]: delay = 0.5

In [6]: time = npo.arange(len(IF)) * sampling_time
+ delay
```

ここで，時間軸などの生成によく利用する `np.arange` と `np.linspace` の使い方を簡単に紹介します．

numpy.arange

`np.arange` は，連番や等差数列を生成します．使い方は Python の組み込み関数 range と似ており，以下のように引数を取ります．なお，`[]` で囲んだ引数は省略できるということを意味します．

```
arange([start,] stop, [step,] dtype=None)
```

`start` で指定した数から `stop` で指定した数まで，`step` 間隔の数字列を生成します．第2引数 `stop` 以外は省略ができますが，第3引数 `step` を指定するときは同時に第1引数 `start` も設定する必要があります．なお，第2引数 `stop` だけを指定した場合は，初項0で交差1の等差数列を要素とする ndarray を生成します．

numpy.linspace

`np.linspace` は等差数列を生成する関数です．同様の関数として先程紹介した `np.arange` がありますが，`np.linspace` を使用すると指定した区間をN等分した配列を生成しているということが明確になります．

```
linspace(start, stop, num=50, endpoint=True,
retstep=False, dtype=None)
```

の形で使用し，生成する等差数列の始点と終点を `start` と `stop` で指定します．第3引数 `num` で配列の長さを，第4引数 `endpoint` で終点を配列の要素として含むかどうかを指定します．

3.1.2.3 配列の演算

データを読み込んで配列が生成できたところで，計測信号の較正値を適用して干渉計の位相信号を密度の値に変換し，そこからオフセットを差し引きます．

NumPy では ndarray で表現した行列に対して，行列の和・積，逆行列の計算，行列式の計算，固有値計算などさまざまな計算を行うメソッドや関数が用意されています．ここで，行列計算では ndarray の +（和），−（差），*（積），/（除算），**（べき乗），//（打ち切り除算），%（剰余）は要素同士の計算になるという点に注意が必要です．行列積を計算するには，dot メソッドを使うか，@演算子（Python3.5以上かつ NumPy1.10以上）を使う必要があります．

今回の例では，まず較正係数を適用して信号値を位相差の値に変換します．

```
In [7]: a1 = -0.005

In [8]: a2 = 0.000

In [9]: b1 = 0.135

In [10]: b2 = 0.300

In [11]: IF[:, 0] = np.arcsin((IF[:, 0]-a1)/b1)*
180/np.pi

In [12]: IF[:, 1] = np.arcsin((IF[:, 1]-a2)/b2)*
180/np.pi
```

次に，位相差を線積分密度の値に変換します．

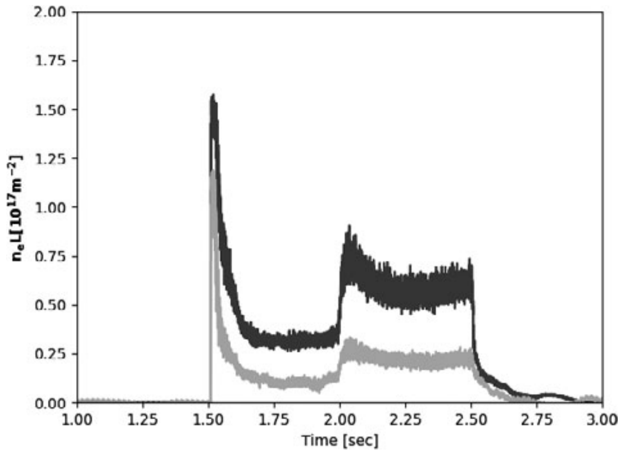
```
In [13]: IF = IF*5.58/360
```

最後に，プラズマのない時間帯の値をオフセットとして差し引きます．

```
In [14]: IF -= np.mean(IF[:5000], axis=0)
```

始めに作成した時間軸の配列とともにグラフに表示してみます．

```
In [15]: plt.plot(time, IF[:, 0]);

In [16]: plt.plot(time, IF[:, 1]);

In [17]: plt.xlim(1.0, 3.0);

In [18]: plt.ylim(0.0, 2.0);

In [19]: plt.xlabel('Time [sec]');

In [20]: plt.ylabel('$mathbf{n_eL [10^{17}m^{-2}]}$')
Out[20]: <matplotlib.text.Text at 0x7f67da09d5c0>
```



上記で用いた`IF[:5000]`は，プラズマがない時間帯（5000番目まで）のデータを切り出しています．このような処理を**インデキシング（Indexing）**と呼びます．`[]`の中身の`:5000`で配列 IF の第 0 軸（この場合は時間方向に相当）の先頭から5000番目までの部分を示しています．

切り出した配列に対し`np.mean`では，`axis`でどの軸（axis）に沿って平均を求めていくのかを決めています．今回は各視線ごとの平均値を求めることが目的のため，`axis = 0`として行方向，つまり列ごとの平均である 1 次元の 2 要素（視線 1，視線 2 のデータ）のベクトルを求めています．

`IF -= np.mean(IF[:5000], axis=0)`は，元のデータから上記で求めた平均を差し引く操作です．2 次元データである`IF`と，`np.mean`によって求めた 1 次元配列との引き算は，大きさが異なるため計算できないように思えます．その後の処理の，較正係数の引き算，除算も同様です．実は NumPy では，**ブロードキャスティング（Broadcasting）**と呼ばれる仕組みにより，大きさを揃える操作を自動的に行っています．

インデキシング

上の例のように NumPy では，インデキシングという処理により，配列の任意の要素・行・列を切り出すことができます．ただし，切り出し方によりコピーを生成するかビュー（参照）を生成するかという違いがありますので注意が必要です．本講座の第 2 章で紹介したように，Python のリストやタプルにも実装されているスライシング（Slicing）を ndarray に対して行うと，その部分配列がビューとして返ってきます．つまり，その部分配列はデータのコピーではなく，元の配列の一部を参照していることになります．そのため，部分配列に対する変更はオリジナルの ndarray を変更してしまいます．

試しに，1 列目の干渉計のプラズマ着火前の信号を抜き出してみます．

```
In [21]: IF_slice = IF[:5000, 0]
```

IF_slice の中身を 0 に変更してみます．

```
In [22]: IF_slice[:] = 0

In [23]: IF[:5000, 0]
Out[23]: array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

この例では，配列 IF_slice はビューですので，元の配列 IF に変更が反映されています．

他の配列指向の言語ではスライスのようなデータ片はコピーとして生成する仕様のものが多いため，このインデキシングの仕様に驚く方は多いと思います．NumPy は，大量のデータ処理を目的として開発されてきました．ビューを用いると元のデータのコピーがメモリ上に作成されないため，特に大きな配列の操作に適しています*3

ブロードキャスティング

＋－＊/等の四則演算や，ユニバーサル関数を使って ndarray 同士の演算を行う際に，異なるサイズの 2 つの ndarray を使って計算を行わなければならないことがあります．こういった処理を簡単・効率的に行うため，NumPy では配列演算の拡張ルールであるブロードキャスティングを採用しています．以下にブロードキャスティングの一例として，1 次元配列と 2 次元配列の配列演算を紹介します（**図 1**）．

```
#1 から 12 までの等差数列を作成し，形状を(4, 3)に変更する
In [24]: b = np.arange(1, 13, 1).reshape((4, 3))

In [25]: b
Out[25]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [26]: c = np.array([1, 2, 3])

In [27]: c.shape   # c の形状(shape)を確認する
Out[27]: (3,)

In [28]: b + c
Out[28]:
array([[ 2,  4,  6],
       [ 5,  7,  9],
       [ 8, 10, 12],
       [11, 13, 15]])
```

NumPy には，配列の全要素に対して要素ごとに演算処理を行う，ユニバーサル関数が組み込まれています．ユニバーサル関数は C や Fortran で実装されており，かつ線形演算では BLAS/LAPACK のおかげで C/C++ と遜色のないほど高速に動作します．例えば，`exp`関数に配列を渡すことで，全要素に指数関数を適用した配列を生成することができます．

```
In [29]: np.exp(c)
Out[29]: array([  2.71828183,   7.3890561 ,
```

＊3 スライスを ndarray の実コピーとして生成する場合には，明示的に`arr2d[1, 1:].copy()`のようにします．

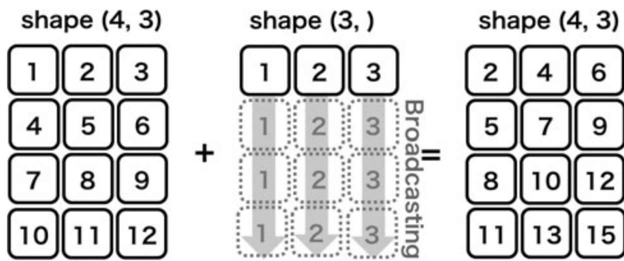


図1 ブロードキャスティングによる配列演算.

```
  20.08553692])
```

このように，NumPy では複数の配列要素に対して処理を一度に実行できます．こうすることで，ループ構造を用いるより圧倒的に高速に計算することができます．

Python のコードで良いパフォーマンスを得るには，以下の事が重要です．

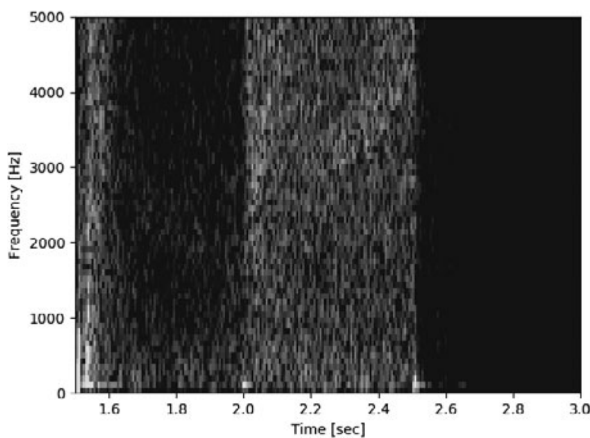
- Python のループと条件分岐のロジックを，配列操作と真偽値の配列の操作に変換する
- 可能なときは必ずブロードキャストする
- 配列のビュー（スライシング）を用いてデータのコピーを防ぐ
- ユニバーサル関数を活用する

特に，Pyhon の言語仕様に慣れないうちは for ループを多用しがちですが，これらに気をつけると Python でも高速で動作するプログラムを作ることができます．

3.1.2.4 SciPy を用いたデータ解析

時系列データの配列を作成することができたので，解析を行っていきましょう．今回の例では，SciPy の信号処理に関するサブモジュール scipy.signal の中の関数 spectrogram を用いて，上記のデータに短時間フーリエ変換を施し，プラズマの不安定性の有無を調べてみます．

```
In [30]: import scipy.signal as sig

In [31]: f, t, Pxx = sig.spectrogram(IF, axis=0, fs=1/sampling_time, window='hamming', nperseg=128, noverlap=64, mode='complex')

In [32]: plt.pcolormesh(t+0.5, f, np.log(np.abs(Pxx[:, 0]) + 1e-15));

In [33]: plt.xlim(1.5, 3.0);

In [34]: plt.xlabel('Time [sec]');

In [35]: plt.ylabel('Frequency [Hz]');

In [36]: plt.clim(-9, -6)
```



ここで，`sig.spectrogram` には，元のデータ `IF` のほか，どの次元に対してフーリエ変換を施すかを `axis` オプションで，サンプリング周波数や，窓関数を `fs`，`window` オプションで指定して渡しています．$t=2.2$ sec，周波数 3〜4 kHz あたりに何か構造があるような気もします．

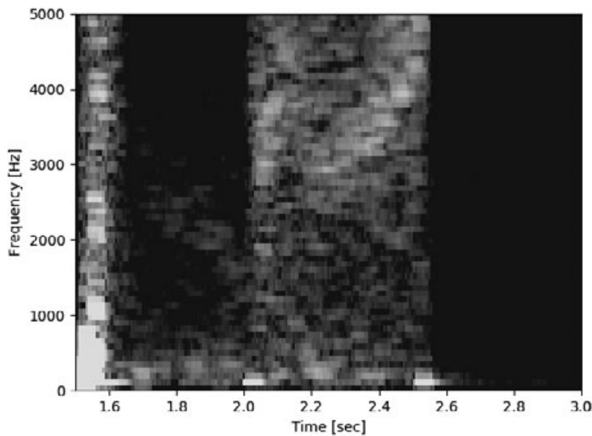
もう少しノイズを除去するために，２つの干渉計信号のクロススペクトルを計算してみましょう．クロススペクトルは，以下の式で計算される量です．

$$\langle f_1 f_2^* \rangle$$

$\langle \mathrm{x} \rangle$ はxに関するサンプル平均を表します．ここでは移動平均で代用することにしましょう．

```
In [37]: def moving_average(x, N):
   ....:     x = np.pad(x, ((0, 0), (N, 0)), mode='constant')
   ....:     cumsum = np.cumsum(x, axis=1)
   ....:     return (cumsum[:, N:] - cumsum[:, :-N]) / N
   ....:
```

```
# クロススペクトルを求める
In [38]: Pxx_run = moving_average(Pxx[:, 0] * np.conj(Pxx[:, 1]), 8)

In [39]: plt.pcolormesh(t+0.5, f, np.log(np.abs(Pxx_run)));

In [40]: plt.xlim(1.5, 3.0);

In [41]: plt.clim(-19, -15);

In [42]: plt.xlabel('Time [sec]');

In [43]: plt.ylabel('Frequency [Hz]')
Out[43]: <matplotlib.text.Text at 0x7f67cdf55dd8>
```



ここで，np.conj(x)は複素共役を求めるユニバーサル関数で，配列の要素ごとに適用されます．移動平均を取る関数moving_averageの説明は省略しますが，スライシングと累積和を用いることで効率よく計算しています．

上記操作により，3〜4 kHz付近の構造を可視化することができました．このように，NumPy/SciPyの既存のツールを用いることで，スペクトル解析を簡単・高速に行うことができます．Matplotlibで描画することで，その結果をすぐに可視化しながら高速に解析を進めることができるでしょう．

3.1.2.5 解析データの書き込み

最後に，物理量に変換した配列を時間軸と一緒にテキスト形式で保存します．

```
In [44]: np.savetxt('time_IF.txt', np.c_[time, IF], delimiter=',')
```

ここでは，配列の結合にnp.c_というオブジェクトを使用しています．np.c_はaxis=1の方向（2次元の場合は列方向）に，np.r_はaxis=0方向（2次元の場合は行方向）に配列を結合します．どちらも関数ではなくオブジェクトなので，全て[]の中に配列や値を入れて操作していきます．np.c_やnp.r_について更に詳しく知りたい場合は，docstring等を参照してください*4．なお他にも，np.concatenate，np.hstack，np.vstackなどの関数を用いても配列の結合を行うことができます．

3.1.3 SciPyを用いたPredator-Preyモデルのシミュレーション

本章の最後に，SciPyを用いた微分方程式の解法例として，Predator-Preyモデルのシミュレーションについて紹介します．

帯状流と乱流の相互作用は，捕食者−被食者（Predator-Prey）モデルで記述されることが知られており[7]，このモデルは1階の連立微分方程式の形をしています．SciPyパッケージのodeintモジュールを使うと，1階の常微分方程式の数値解を簡単に得ることができます*5．odeintはLSODA（Livermore Solver for Ordinary Differential equations with Automatic switching for stiff and non-stiff problems）法を利用した汎用的な積分器ですが，詳しくはODEPACK Fortran library[8]を参照してください．

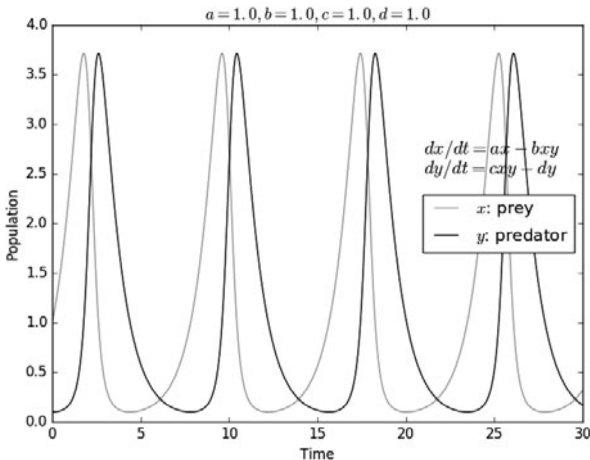
まずは，ソースコードを見てみましょう．

```
1  #!/usr/bin/env python
2  ¥PYGZdq¥PYGZdq¥PYGZdq
3      Sample Code
```

＊4　IPythonなどでnp.r_?と呼び出してdocstringを確認することができます．
＊5　なお，高階の微分方程式でも，1階の微分方程式に変換することでodeintを用いて計算することができます．

```
4
5       Status
6       ------
7       Version 1.0
8
9       Authour
10      -------
11      Shigeru Inagaki
12      Research Institute for Applied Mechanics
13      inagaki@riam.kyushu-u.ac.jp
14
15      Revision History
16      ----------------
17      [11-April-2017] Creation
18
19      Copyright
20      ---------
21      2017 Shigeru Inagaki (inagaki@riam.kyushu-u.ac.jp)
22      Released under the MIT, BSD, and GPL Licenses.
23
24  ″″″
25  import numpy as np
26  import scipy.integrate as desol
27  import matplotlib.pyplot as plt
28
29  def predator_prey(f, t, a, b, c, d):
30      ″″″ return left hand sides of ordinary differential equations
31
32          model equation:
33              dx/dt = ax - bxy
34              dy/dt = cxy - dy
35
36      f[0]     - x: Population of prey
37      f[1]     - y: Population of predator
38      t        - Time
39      a,b,c,d - Control parameters
40      ″″″
41      return [a*f[0]-b*f[0]*f[1], c*f[0]*f[1]-d*f[1]]
42
43
44  #model
45  #eq1 = r″$fracdxdt = ax - bxy$″
46  #eq2 = r″$fracdydt = cxy - dy$″
47  eq1 = r″$dx/dt = ax - bxy$″
48  eq2 = r″$dy/dt = cxy - dy$″
49
50  #input parameters
51  a = 1.0
52  b = 1.0
53  c = 1.0
54  d = 1.0
55  header = r″$a=0:.1f, b=1:.1f, c=2:.1f, d=3:.1f$″.format(a, b, c, d)
56
57  #initial condition
58  f0 = [1.0, 0.1]
59
60  #independent variable
61  nt = 1000
62  tmax = 30.0
63  dt = tmax / nt
64  t = dt * np.arange(nt)
65
66  f = desol.odeint(predator_prey, f0, t, args=(a,b,c,d))
67
68  #plot
69  prey = f[:,0]
70  predator = f[:,1]
71
72  fig = plt.figure()
73  ax = fig.add_axes([0.15, 0.1, 0.8, 0.8])
```

```
74 ax.plot(t, prey, color='r', label=r"$x$: prey")
75 ax.plot(t, predator, color='b', label=r"$y$: predator")
76 handles, labels = ax.get_legend_handles_labels()
77 ax.legend(handles, labels, loc='best')
78 ax.text(21, 2.7, eq1, fontsize=16)
79 ax.text(21, 2.5, eq2, fontsize=16)
80 ax.set_xlabel("Time")
81 ax.set_ylabel("Population")
82 ax.set_title(header)
83 plt.show()
```



プログラムの内容は以下のようになっています．

1．解析する関数（この場合 predator_prey）を定義する
　・第１引数 f が微分方程式中の未知関数
　・第２引数 t が関数のパラメータ（時間に対応）
　・第３‐６引数 a，b，c，d が定数
　・戻り値がパラメータ t における dx/dt，dy/dt を与える
2．微分方程式の定数 a，b，c，d を与える
3．微分方程式の初期値 f0 を与える
4．未知関数の解析範囲（時間）を与えるパラメータ列 t を用意する
5．関数`scipy.integrate.odeint`に1-4を引数にして呼び出す
6．戻り値がパラメータtに対応する未知関数fの各値となる

帯状流とプラズマ乱流の相互作用を当てはめて考えてみると，乱流を餌として発生・成長する帯状流は捕食者の役割を，またプラズマ圧力勾配により発生する線形不安定性を源として成長する乱流は被食者の役割を果たします．

このようにPython を用いることで，簡単にモデルの計算と可視化をすることができます．コーディングの時間を短縮し，試行錯誤に多くの時間を割けるのがPython の利点でもありますので，みなさんもまずは簡単なプログラムを作成し，動作を確認してみてください．

3.1.4　まとめ

本章では，NumPy/SciPyの特徴と基本的な使用法，簡単なシュミレーションの例を紹介しました．NumPy が他の多くのライブラリの基礎となっているため，NumPy の基本を理解することが Python を用いた科学技術計算にとって重要です．本章で紹介したNumPyにおけるndarrayやユニバーサル関数，ブロードキャスティングの概念は Python の機能を大幅に拡張しており，これらの概念に慣れることがプログラミングの効率を大きく向上させます．さらに，SciPyを用いることでNumPyの機能の上に構築された様々な科学技術計算アルゴリズムを利用できます．SciPy は非常に巨大なパッケージですので，効率良く計算を進めるため，処理を実装する前に SciPy で既に実装されていないかどうかぜひ確認してみてください．

科学技術計算において，特に解析対象や解析方法がその都度変化するプラズマ実験では，実験条件に対応して柔軟にコードを組まなければなりません．例えば，数分間の実験周期中に直前のプラズマ放電で得られたデータを解析し，それを基に次の放電の条件を決めるといった場合，非常に短時間にコードを組んで解析を進めることが要求されます．こういった状況では，実行速度よりも開発速度が重要になることが多く，Python はその用途に適しています．

本講座で Python の使い方を一通り覚えたら，まずは自身の研究でも試してみてください．すぐに Python の柔軟性や開発のスピード感を味わってもらえると思います．

参考文献

［1］http://www.numpy.org
［2］https://www.scipy.org
［3］Wes McKinney：Python によるデータ分析入門（オライリー・ジャパン，2013）.
［4］中久喜健司：科学技術計算のための Python 入門（技術評論社，2016）.

[5] Z.Yoshida *et al.,* Phys. Plasmas, **17**, 112507 (2010).
[6] https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.io.html
[7] 小林すみれ　他：プラズマ・核融合学会誌 **92**, 211 (2016).
[8] http://people.sc.fsu.edu/~jburkardt/f77_src/odepack/odepack.html

鈫持尚輝（けんもち なおき）

1987年静岡県浜松市生まれ．東京大学大学院新領域創成科学研究科　助教．2011年京都大学工学部物理工学科卒業．2016年同大学院エネルギー科学研究科博士後期課程修了．博士（エネルギー科学）．学生時代は京都大学 Heliotron J で，現在は東京大学 RT-1装置にて熱・粒子輸送を研究．両装置でトムソン散乱計測装置開発に従事．趣味は空手（京都大学空手道部コーチ）．最近，生命保険に加入したが，空手が危険スポーツとみなされ保険料が大幅に上がってしまった．

J. Plasma Fusion Res. Vol.94, No.3 (2018) 150‑161

講座 今日からはじめる Python

3．Python による科学技術計算

3. Scientific Computing in Python

3.2 matplotlib の使い方

3.2 How to Use matplotlib

吉 沼 幹 郎
YOSHINUMA Mikirou
核融合科学研究所

（原稿受付：2018年 1 月23日）

Python を用いて，データをグラフに描画するときに利用される matplotlib というモジュールの使い方を説明します．このモジュールを利用することで，Python を用いた行われた計算の途中や結果の数値を確認することが簡単にできるようになります．また，論文掲載用の図として利用できる品質のさまざまなグラフを描画することができます．ここでは，簡単な 2 次元データの散布図から，凡例表示やエラーバーの表示方法，3 次元データのヒートマップの表示について説明します．

Keywords:
Python, matplotlib, data visualization, graph, control plot

3.2.1 はじめに

Python を用いて，実験データの解析を行うとき，あるデータのセットに対して，Python で記述されたなにかの処理を適用して，結果を得るということを繰り返します．そのよなとき，得られた結果をグラフとして描画したいと思うことでしょう．数値をファイルに書き出し，グラフ描画ソフトウェアで読み込み，描画するという手順を踏んでもよいのですが，処理の途中経過などは，いちいちファイルに書き出すのを面倒に感じます．ここでは，そのような状況でも便利に利用できる，Python で取り扱っているデータから直接グラフを描画できる matplotlib モジュールの使い方を紹介します．matplotlib は，Python でグラフを描画するときに利用される標準的なモジュールとなっており，二次元の散布図からヒートマップなど様々なグラフを描画することができます．matplotlib のホームページのギャラリーでは，どのようなグラフが描画できるのか一覧できます．

matplotlib は，グラフ描画に使われる要素（オブジェクト），例えば軸や線やラベルといったものが詰まったもので，それらを適切に設定，操作することで，グラフ全体を構成します．matplotlib で，それらのオブジェクトを操作するには，オブジェクトごとに持っている操作手続き（メソッド）を直接呼び出す方法（オブジェクトインターフェース）と，オブジェクトの操作手続きを代行する関数を呼び出す方法（関数インターフェース）があります．前者の方法は，記述が増える代わりに，matplotlib のすべての機能を使うことができます．Python で構築するアプリケーションソフトウェアに matplotlib を組み込んで使う場合に便利です．後者の方法は，記述が少なくなるため，Jupyter などのインタラクティブな環境で使うときに便利です．

Python のモジュールを利用するためには，利用したいモジュールをインポートする必要がありました．matplotlib というモジュールは，複数のモジュールファイルで構成されており，どれをどのようにインポートしたらよいか分かりにくいかもしれません．グフラフ描画に利用されるものが matplotlib.pyplot というモジュールです．しばしば，スクリプトの冒頭で‘import matplotlib.pyplot as plt’と記述されインポートされています．これは，matplotlib.pyplot モジュールを plt という名前でインポートすることを意味します．このようにすると，matplotlib.pyplot に含まれる関数，例えば，plot()や show()などが，’plt.plot(...)’や’plt.show()’のように，モジュール名’plt.’をつけて記述することで呼び出すことができます．matplotlib には，MATLAB と互換性のある関数名を使いたい人のために matplotlib.mlab モジュールも用意されています．

ここではインタラクティブな環境を利用することを考え，pylab というモジュールをインポートして関数イン

National Institute for Fusion Science, Toki, GIFU 509-5292 Japan

author’s e-mail: yoshinuma@nifs.ac.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research

ターフェースを使うスタイルで説明していきます．そこで，スクリプトの冒頭には，‘from pylab import *’と記述しています．このように pylab モジュールを import すると，matplotlib.pyplot,matplotlib.mlab,numpy モジュールに含まれる関数を，モジュール名無しに，関数名だけで使えるようになり，記述量が少なくなります．ただし，このようにモジュール名を省略すると，異なるモジュールにおいて同名の関数がある場合に上書きされてしまう問題が起こりますので注意が必要です．同名の関数や変数があるかどうかは，インタラクティブな環境でその名前を評価してみるとわかりますので，疑わしいときは確認するとよいでしょう．ここでは，気軽にスクリプトを入力して試していただくために，記述量が少なくなるスタイルをとりましたが，関数などの出所を明確にするためにも，モジュール名を省略しないスタイルをお勧めします．

matplotlib によるグラフ描画の説明には，プロットするデータが必要です．プロットするデータはすべてスクリプト内で手短に生成していますので，試すために，データファイルを用意する必要はありません．ぜひ，リストにあるスクリプトを入力，改変して，動きを確認していただきたいと思います．

3.2.2　はじめの一歩

まず，インタラクティブな環境を起動して **List 1** にあるものを実行してみましょう．説明のため，余計な線や文字が入っていますが，**図 1** に描画結果を示します．"a"という名前の配列に－1から1まで0.1刻みの数値を設定（2行目）し，その内容をグラフにプロット（3行目）するものです．plot()関数に与えた a という配列の値が y 軸の値としてプロットされています．配列一つを plot()関数に与えると，x 軸の値は配列のインデックス番号が使われます．このように配列にどのような値がはいっているかを簡単にグラフにして確認することができます．数値列を生成する arange()関数は numpy モジュールによって提供されていますが，pylab モジュールを 1 行目の記述で import したことで，モジュール名を省略して利用できるようになっています．

List 1：pylab のインポートと配列内容のプロット

```
1 from pylab import *
2 a = arange(-1, 1.1, 0.1)
3 plot(a)
4 show()
```

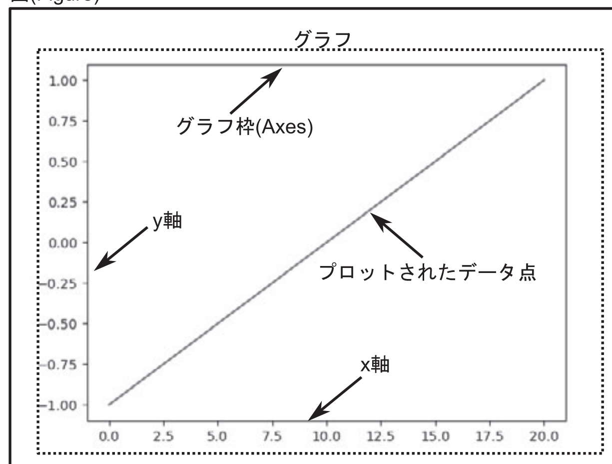


図 1　List 1 による描画結果と図（Figure）の構成要素．

図 1 を見ながら，グラフを描画するときのいくつかの重要な構成要素について説明します．グラフが描画されている領域全体が Figure と呼ばれる要素です．本文では「図(Figure)」と書くことにします．matplotlib ではこの図(Figure）にグラフを描画し，それを表示させます．図(Figure）の中には，x 軸，y 軸で囲まれた四角の領域があります．本文では，「グラフ枠(Axes)」と書くことにします．グラフ枠（Axes）は x 軸，y 軸によって構成され，その中にデータ点がプロットされます．グラフ枠（Axes）は，図 (Figure) の中に一つ以上配置することができます．関数を呼ぶスタイルで matplotlib を利用する場合，plot()関数によって，グラフ枠（Axes）内にデータ点をプロットし，show()関数によって，グラフ枠が配置された図(Figure）を描画します．図（Figure）やグラフ枠（Axes）は自動的に作成され，操作の対象となる図（Figure）やグラフ枠（Axes）も自動的に決められます．関数を呼び出した結果は，現在対象になっている図（Figure）やグラフ枠(Axes）に現れます．ですから，現在対象になっている図(Figure）やグラフ枠（Axes）というものを意識して操作するとよいでしょう．

3.2.3 (x, y)データをプロットする

ここでは単純なxy散布図のグラフ描画を通して，matplotlibの各種操作を説明していきます．

3.2.3.1 データ点をプロットしてみる

単純なガウス分布関数をプロットしてみましょう．**List 2**を実行してみてください（**図2**）．

List 2：ガウス分布関数

```
1 from pylab import *
2 xdata = arange(-2, 2.2, 0.2)
3 ydata = exp(-xdata**2)
4 plot(xdata, ydata, 'o')
5 show()
```

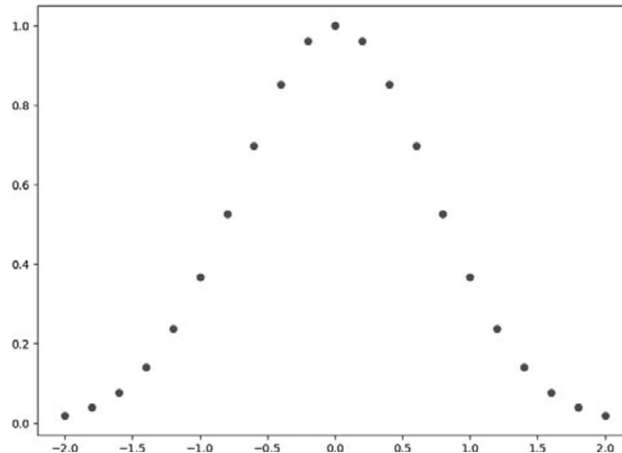


図2 List 2による描画結果．

2行目，3行目でプロットするデータの配列を作り，xdata，ydataとしています．それらをplot()関数の1番目の引数，2番目の引数としてそれぞれ与えます．plot()関数に渡す3番目の引数で，プロットに用いるマーカーを指定します．アルファベット小文字の'o'を渡すことで，塗りつぶされた丸記号を指定しています．注意していただきたいのは，plot()関数の引数は，前の章で配列を一つ与えたときとは異なっています．このように，同じ関数でも，引数の与え方に応じて適切に振る舞いが変化するものもあります．

3.2.3.2 マーカーや線種の指定とオーバープロット

plot()関数に渡す三番目の引数でマーカーを変更することができました．この指定に使われる文字は，インタラクティブな環境で'help(plot)'を実行すると表示される説明で読むことができます．丸●，三角▲，四角■，ダイヤ◆の記号は，それぞれ'o'，'^'，'s'，'D'で指定します．実線，破線，点線，一点鎖線は，それぞれ'-'，'--'，':'，'-.'で指定します．マーカーで点を打ち，さらに線で結びたいときは，これらを組み合わせます．例えば，丸でプロットして，点線で結びたいときは，'o:'という文字の列を三番目の引数に渡します．マーカーのサイズは，markersize（あるいはms）というキーワード引数を使って指定できます．キーワード引数とは，Pythonの関数で用いられる引数の種類の一つで，順不同に渡せるばかりでなく，渡さなかった場合にデフォルトの値が使われるため，指定しなくてもよい引数です．線の太さは，linewidth（lw）というキーワード引数を使って指定できます．いくつかマーカーとマーカーのサイズ，線種を変えてプロットしてみます．plot()関数の呼び出しを重ねるとデータをオーバープロットできます（**List 3**，**図3**）．

List 3：マークを変えてオーバープロット

```
1 from pylab import *
2 xdata = arange(-2, 2.2, 0.2)
3 ydata = exp(-xdata**2)
4 plot(xdata, ydata, 'o--', markersize=4, linewidth=2)
5 plot(xdata, ydata*0.7, 's', ms=6)
6 plot(xdata, ydata*0.5, '^', ms=8)
7 plot(xdata, ydata*0.3, 'o-', ms=8, lw=4)
8 show()
```

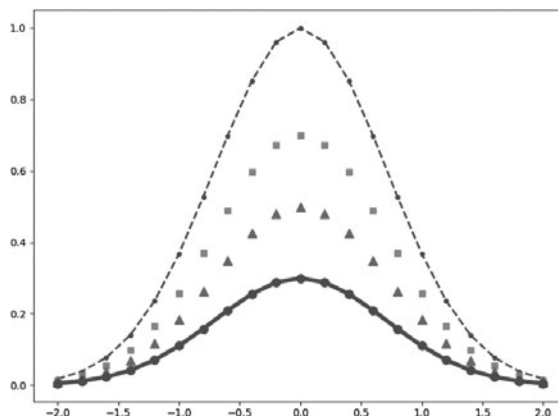


図３　List 3 による描画結果.

3.2.3.3　色の指定

plot()関数を繰り返し呼び出してオーバープロットすると色が自動的に変わっていきます．自分で色を指定したいときは，plot()関数を呼び出すときに color（あるいは c）というキーワード引数に色を示す値を渡します．色を示す値には，色の名前，色を表す１文字，RGB(A)の16進数文字列，RGB(A)の値をもつ tuple などいろいろあります．ここで(A)と書きましたが，これは透明度を指定するアルファチャンネルの値も指定できるからです．色を表す１文字には，‘b’，‘g’，‘r’，‘c’，‘m’，‘y’，‘k’，‘w’の８文字があり，それぞれ青（blue），緑（green），赤（red），水色（cyan），赤紫（magenta），黄（yellow），黒（key），白（white）に対応します．色の名前は多くありますが，実際の色をイメージすることは難しいので，先の８文字以外の色を使いたい場合は，RGB の16進数文字列で指定することをお勧めします．RGB の16進数文字列とは，00（10進数で０）から FF（10進数で255）の大きさで赤（Red），緑（Green），青（Blue）の各色の強さを指定する方法です．先頭に‘#’をつけて，例えば赤なら‘#FF0000’となります．白は，RGB 各色が最大値ですので，‘#FFFFFF’となります．一方，黒は各色が最小値の‘＃000000’となります．**List 4** にいろいろな方法で色を指定した例を示します．最後の例のように色を１文字で指定する場合，スタイル指定の文字列に含めてしまうこともできます．手早く色を変えてプロットしたいときは，こちらの指定が楽でしょう．マーカー内部の色，境界の色はそれぞれ markerfacecolor（mfc），markeredgecolor（mec）というキーワード引数に色を指定する値を渡すことで個別に設定できます．これを用いて，白抜きや中抜きのマーカーを利用することもできます（**List 4**，**図４**）．

List 4：さまざまな色指定の方法

```
1  from pylab import *
2  xdata = arange(-2, 2.2, 0.2)
3  ydata = exp(-xdata**2)
4  plot(xdata, ydata, 's--', color='#FF0000') # 16進数文字列で指定
5  plot(xdata, ydata*0.8, 's--', c=(0, 0, 1)) # tupleで指定
6  plot(xdata, ydata*0.6, 's--', c='y') # 色の文字で指定
7  plot(xdata, ydata*0.5, 's--', c='m') # 色の文字で指定
8  plot(xdata, ydata*0.4, 's--', c='magenta') # 色の名前で指定
9  plot(xdata, ydata*0.3, 's--', c='bisque') # 色の名前で指定
10 plot(xdata, ydata*0.2, 's--g')
11 show()
```

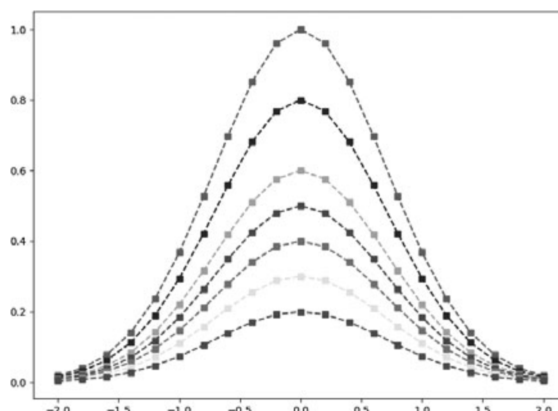


図４　List 4 による描画結果.

3.2.3.4 軸の設定

リスト **List 5** と**図 5** に，軸の目盛りや範囲を設定する例を示します．matplotlib では，図（Figure）をディスプレイに表示する前，すなわち show()関数を呼ぶ前に設定を済ませておく必要があります．

4 行目の tick_params()関数によって，軸に目盛りや目盛りのラベル描画をさせるかの設定ができます．right キーワード引数，top キーワード引数に True を設定することで，x 軸，y 軸の対向側にも目盛りを振るようにしています．directionキーワード引数に'in'を設定することで，目盛りをグラフ内側に向けて描画するようにしています．5 行目の grid()関数によって，目盛りにあわせて格子状の線を描画させます．6 行目，8 行目の xticks()関数，yticks()関数によって，目盛りを振る場所を指定できます．第一引数には，目盛りを振る位置を数値の配列にして渡します．第二引数に，文字列のリストを渡すと，目盛りのラベルにその文字列が使われます．例では，目盛り位置は，arange()関数を用いて生成しています（6 行目，7 行目）．y 軸のラベルは，目盛り位置（yticks_positions）の数値を'%4.3f'（小数点以下 3 桁）でフォーマットした文字列のリストを第二引数に渡しています．9，10行目の xlim()，ylim()関数によって，それぞれ x 軸の範囲，y 軸の範囲を設定します．第一引数に下限値，第二引数に上限値を指定します．軸の方向は下限値，上限値の大小関係によって適切に変化します．各関数のパラメータの詳細については，plot()関数のときと同様に，インタラクティブ環境で'help（関数名）'を実行することで読むことができます．

List 5：軸の範囲とラベルの設定

```
1  from pylab import *
2  xdata = arange(-2, 2.2, 0.2)
3  ydata = exp(-xdata**2)
4  tick_params(right=True, top=True, direction='in')
5  grid(True)
6  xticks(arange(-3.0, 3.1, 0.5))
7  yticks_positions = arange(0.0, 3.1, 0.25)
8  yticks(yticks_positions, ["%4.3f" % v for v in yticks_positions])
9  xlim(-2.5, 2.5)
10 ylim(0, 1.5)
11 plot(xdata, ydata, 's-', ms=8, c='r')
12 show()
```

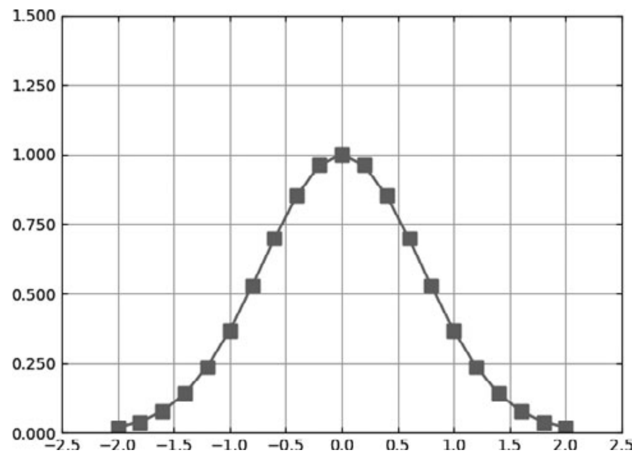


図 5 List 5 による描画結果．

3.2.3.5 タイトルと凡例の表示

List 6 と**図 6** に，グラフタイトル，軸のラベルや凡例の表示を行う例を示します．4，5 行目の xlabel()，ylabel()関数によって，それぞれ x 軸のラベル，y 軸のラベルを設定します．ラベルとして書き出す文字列を第一引数に指定します．6，7 行目でデータをプロットしています．プロットされたデータと凡例に表示する文字を一致させるために，plot()関数を呼び出すときに，キーワード引数 label に凡例として表示される文字列を設定しています．8 行目の legend()関数で凡例を表示させる指示を行いますが，凡例として使用される文字列が指定された後，すなわち plot()を行った後に行います．9 行目で title()関数によってタイトルの文字列を設定しています．文字列には，TEX形式の数式表記も利用できます．**List 6** では，タイトルの文字列として，r'Function...'のように文字'r'をつけています．この例では使用していませんが，TEX表記ではしばしば'\'文字（機種によっては'¥'文字）を利用します．Python では文字列の表記中の'\'文字は，特別な文字（例えば改行やタブ文字）を表記するために使われます．また，'\'文字を表記するためにも使われるため，文字列データの中に'\'文字を入れたいときは'\\'のように二回書く必要があります．文字列の前に'r'文字をつけると書いたものをそのまま Python の文字列データとすることができます．ですからこのような記述をしておくと，TEX表記の文字列を書きやすくなります．文字の大きさは，xlabel()，ylabel()，title()，legend()関数の fontsize キーワード引数に数値を設定する

ことで，変更することができます．legend()関数には，表示設定のためのキーワード引数が多く存在します．例では numpoints＝1 を指定して，凡例におけるマーカーの数を１つに設定しています．

List 6：凡例を表示

```
from pylab import *
xdata = arange(-2, 2.2, 0.2)
ydata = exp(-xdata**2)
xlabel('x-axis', fontsize=14)
ylabel('y-axis', fontsize=14)
plot(xdata, ydata, 'o-', ms=8, c='r', label='A=1.0')
plot(xdata, ydata*0.5, 's-', ms=8, c='b', label='A=0.5')
legend(fontsize=16, numpoints=1)
title(r'Function␣of␣$y=Ae^{-x^2}$', fontsize=20)
show()
```

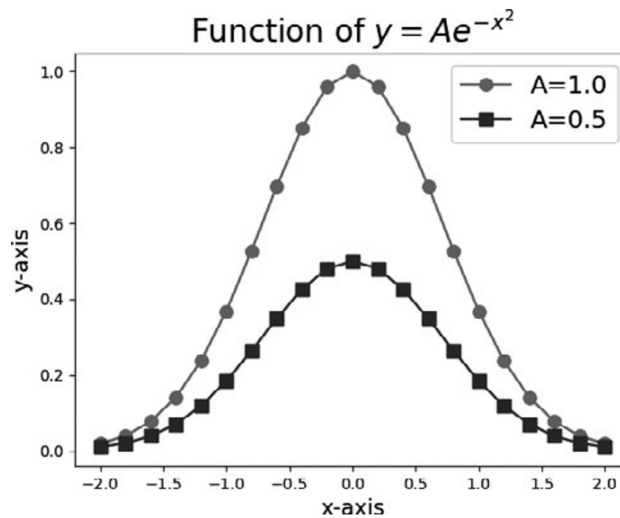


図 6　List 6 による描画結果

3.2.3.6　y 2 軸の利用

異なる値を同じ x 軸上にプロットしたい場合，x 軸を共通にして，y 軸をもうひとつ増やしたいことがあります．もう一つの縦軸を y2 軸とここでは呼ぶことにします．グラフ枠の左側の枠線が y 軸に用いられ，y2 軸は y 軸の反対側（右側）の枠線に描かれます．

List 7 および**図 7** に y2 軸を利用する例を示します．5 行目から 9 行目までは，これまで行ってきた x-y 軸へのプロットです．11行目で twinx()関数を呼び出しています．twinx()関数を用いると，x 軸を共有した‘新しい y 軸’（＝y2 軸）をもったグラフ枠（Axes）ができ，操作対象が新しくできたグラフ枠(Axes)に変化します．y2 軸は twinx()を行った後で作られますので，その範囲設定，ラベル設定，y2 軸上へのプロット，レジェンドの描画は，その後で行います．

List 7：y 2 軸を使う

```
from pylab import *
xdata = arange(-2, 2.2, 0.2)
ydata = exp(-xdata**2)

ylim(0, 1.5)
xlabel('x-axis', fontsize=14)
ylabel('y-axis', fontsize=14)
plot(xdata, ydata, 'o--', ms=8, c='r', label='on␣x-y')
legend(fontsize=16, loc='upper␣left')

twinx()
ylim(0, 3)
ylabel('y2-axis', fontsize=14)
plot(xdata, ydata, 's--', ms=8, c='b', label='on␣x-y2')
legend(fontsize=16, loc='upper␣right')
show()
```

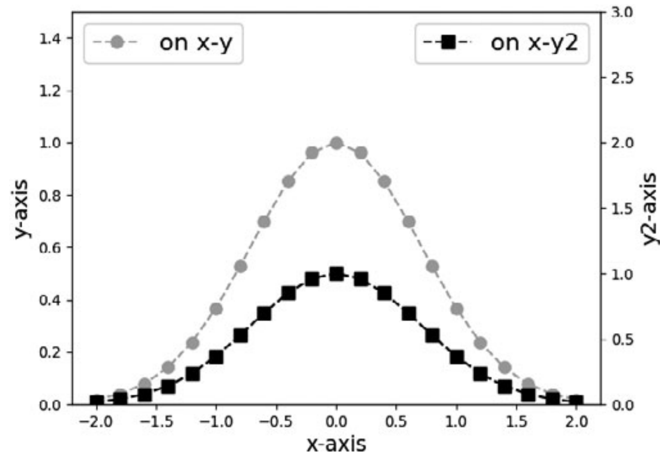


図7　List 7による描画結果.

3.2.3.7　一つの図に複数のグラフを描く

しばしば，複数のグラフを用いて異なる物理量を同時に載せることもあります．グラフを一つ一つ作成しておいて，あとで合わせることもできますが，ここではmatplotlibで一つの図に複数のグラフを描く方法を説明します．

複数のグラフを作る場合，それぞれにグラフ枠(Axes)を作成する必要があります．これまで行ったtwinx()関数も新たにグラフ枠（Axes）を作成する効果がありました．ここではsubplot()関数を用いて新たにグラフ枠（Axes）を作成します．3行2列，合計6枚のグラフ枠（Axes）を並べたい場合，subplot()関数の第1引数に行数の3を，第2引数に列数の2を与え，第3引数に，どの位置のグラフ枠（Axes）を作成するかをインデックス番号で指定します．インデックス番号は**図8**のように，まず行方向に変化します．4番の位置にプロットしたいなら，subplot(3, 2, 4)とすると，その位置に描画されるグラフ枠（Axes）が作成され，それが操作対象になります．また，行数，列数，インデックスが10より小さい場合，それらを一つの引数にまとめてsubplot（324）のように指定することもできます．

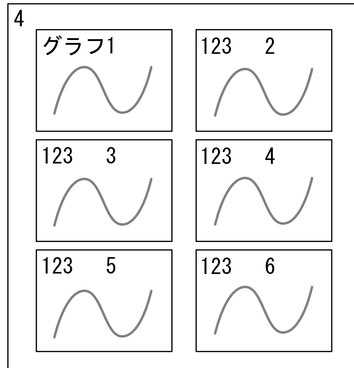


図8　subplot()関数によるレイアウトとインデックス番号.

List 8および**図9**に1行2列のレイアウトでプロットする例を示します．5行目で左側のグラフ枠（Axes）を作成しています．6行目で枠の上と右側にも目盛りを振るように，tick_params()関数で設定しています．11行目で右側のグラフ枠（Axes）を作成しています．こちらの枠は，左側の目盛りの数値を表示させないように，12行目のtick_params()関数でlabelleft=Falseを設定しています．15行目のtight_layout()関数でグラフ枠の間隔を調整しています．このtight_layout()関数を使うと，各グラフの間隔を大まかに変えることができます．h_pad，w_padというキーワード引数にフォントサイズを基準にした値を設定します．たとえば，横方向の間隔を2文字分ほど広げたいならw_pad＝2.0とします．位置の調整には，subplots_adjust()関数も使うことができます．subplots_adjust（wspace＝0.0，hspace＝0.0）とするとグラフ同士が密着します．ここでキーワード引数wspace,hspaceには，それぞれグラフのx軸の幅，y軸の幅を基準とした値を渡します．

List 8：subplot()関数を用いた１行２列のレイアウト

```
from pylab import *
xdata = arange(-2, 2.2, 0.2)
ydata = exp(-xdata**2)

subplot(1, 2, 1)
tick_params(top=True, right=True, direction='in')
plot(xdata, ydata, 'o-', ms=8, c='r')
xlabel('x-axis', fontsize=14)
ylabel('y-axis', fontsize=14)

subplot(1, 2, 2)
tick_params(top=True, right=True, labelleft=False, direction='in')
plot(xdata, ydata, 's-', ms=8, c='b')
xlabel('x-axis', fontsize=14)
tight_layout(w_pad=0.0)
show()
```

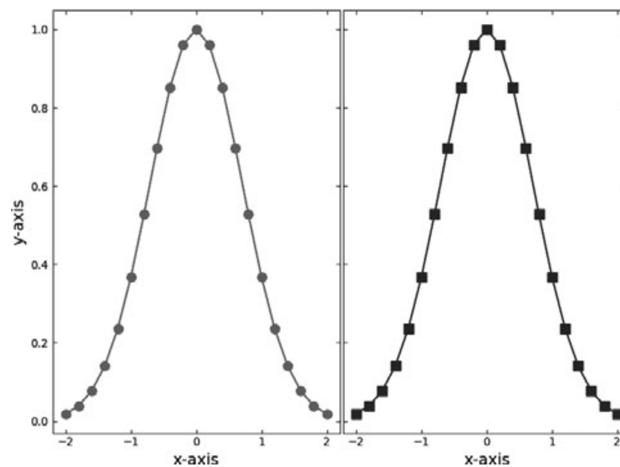


図９　List 8 の描画結果.

3.2.3.8　図(Figure)の大きさの変更と保存 Change size of figure and save to file

これまで自動的に作成されるデフォルト設定の図(Figure)を利用してきましたが，figure()関数を用いると，大きさを指定して図(Figure)を作成することができます．この関数を使うと，操作対象が新たに生成された図(Figure)に切り替わります．**List 9** には，８インチ×６インチの画像を 100 dpi で png（Portable Network Graphics）形式で保存し，また表示する例を示します．４行目の figure()関数の figsize キーワード引数にサイズを表す(8, 6)のタプルを設定しています．dpi キーワード引数に，100を設定しています．６行目の savefig()関数によって，画像ファイルとして保存しています．第一引数に出力ファイル名を指定します．拡張子によって自動的に画像形式が変換されます．拡張子は，'.png'，'.svg'，'.eps'というところがよく使われるものでしょう．保存と表示を行いたい場合は，savefig()関数を使ってから show()関数を使ってください．例では，カレントディレクトリ（現在の作業ディレクトリ）に'output_gauss.png'という名前で画像ファイルが作成されますので，同名のファイルがないことを確認してから実行してください．

List 9：図の保存

```
from pylab import *
xdata = arange(-2, 2.2, 0.2)
ydata = exp(-xdata**2)
figure(figsize=(8, 6), dpi=100)
plot(xdata, ydata, 'o-', ms=8, c='r')
savefig('output_gauss.png', dpi=100)
show()
```

3.2.4　いろいろなプロット

ここまで，matplotlib で単純な散布図を描画して，軸の範囲を設定したり，ラベルを付けたりしてきました. 次に，誤差棒のついたプロットやヒートマップをプロットする方法を紹介します.

3.2.4.1　誤差棒のついたグラフ Plot with error bars

誤差棒のついたグラフを描画するには errorbar()関数を使います．第１引数，第２引数にそれぞれ x，y のデータを渡すのは，plot()関数と同じです．errorbar()関数には誤差の値を渡すキーワード変数 xerr と yerr があります．x 軸方向の誤差は xerr に，y 軸方向の誤差は yerr に渡します．また，マーカーの指定には，キーワード変数 fmt を用います．

誤差棒の描き方を指定するためにキーワード変数 capsize, capthick,elinewidth があります．工字型の誤差棒を表示したい場合は，capsize を適切に指定してください．capsize で誤差棒両端の線の長さを，capthick で誤差棒両端の線の太さを，elinewidth で誤差棒の太さを指定できます．**List 10** に例を示します．

List 10：誤差棒のついたグラフ

```
from pylab import *
xdata = arange(-2, 2.2, 0.2)
ydata = exp(-xdata**2)
yedata = 0.1*ones(xdata.size)
errorbar(xdata, ydata, yerr=yedata, fmt='o', ms=8, c='r', capsize=5)
show()
```

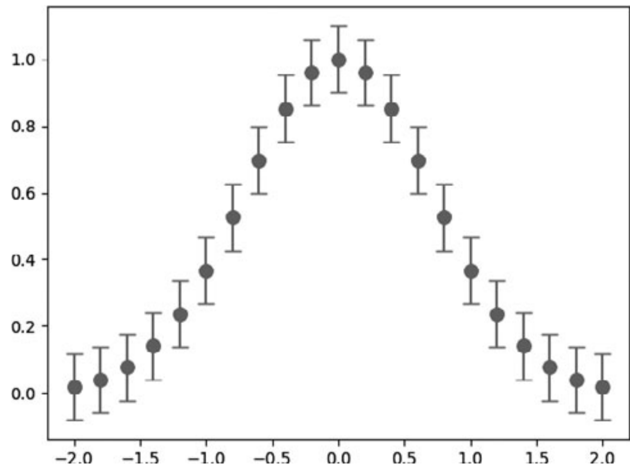


図10 List 10 の描画結果．

3.2.4.2 等高線のプロット Contour plot

ここから 3 次元のデータ（二次元の配列に大きさが入ったもの）のプロットをしていきます．3 次元プロットのためのデータを二次元配列で与えるときに混乱しやすいのは，x 軸，y 軸方向と 2 次元配列の行方向，列方向の関係です．向きが分かりやすいように，x 軸方向に非対称，y 軸方向に対称な $f(x, y) = \sin(x) + \cos(y)$ という関数の値を z としてプロットしてみます．

等高線は，contour()関数を用いて描きます．等高線の間を塗りつぶす contourf()関数もありますが，おおよその使い方は contour()関数と同じです．contour()関数には 2 次元の配列を一つ渡すと，x 軸，y 軸はそれぞれは列方向，行方向のインデックス番号が使われます．2 次元配列のデータを確認するときには便利かと思います．よく行うのは，x, y，z の 3 次元データを与えて，等高線を描かせることでしょう．

式（1），（2）で示される x, y のリストに対して，z の二次元配列をどのような並びで作成するかを式（3）に示します．配列 x，配列 y の要素数がそれぞれ z の列数，z の行数に一致する必要があります．すなわち行方向が y 軸の方向，列方向が x 軸の方向になります．contour()関数の x と y には，式（4），（5）に示す xm，ym のような 2 次元配列を渡しても問題ありません．

List 11にプロットした例を示します．contour に与えるデータを作成する箇所が重要な場所です．x，y のリストから xm，ym のようなリストを作成する関数が meshgrid()です．levels キーワード引数には，線を引くレベルを指定できます．

等高線にラベルをつけるには，プロットした等高線オブジェクトを clabel()関数に渡す必要があります．contour()関数は，等高線をプロットした後，等高線オブジェクトを返してきますので，それを cntr 変数に保持して（6 行目），clabel()関数に渡しています（7 行目）．ラベルの書式の設定はclabel()関数のキーワード引数fmtに，Pythonの書式指定文字列（C言語と似ています）を渡すことで行います．例では，小数点以下 2 桁の表示を指定しています．

contour()関数には，他にも設定があります．colors キーワード引数に色の指示値を与えると，その色が順番に使われますが，一つだけ渡すと単色で線を引くことができます．詳しくは，matplotlib ホームページのギャラリーを見て調べるとよいでしょう．

$$x = [x_0, x_1, \cdots, x_n] \tag{1}$$

$$y = [y_0, y_1, \cdots, y_m] \tag{2}$$

$$\begin{aligned} z = [&[f(x_0, y_0), f(x_1, y_0), \\ &[f(x_0, y_1), f(x_1, y_1), \\ &\cdots \\ &[f(x_0, y_m), f(x_1, y_m), \cdots, f(x_n, y_m)]] \end{aligned} \tag{3}$$

$$\begin{aligned} xm = [&[x_0, x_1, \cdots, x_n], \\ &[x_0, x_1, \cdots, x_n], \\ &\cdots \\ &[x_0, x_1, \cdots, x_n]] \end{aligned} \tag{4}$$

$$ym = [[y_0, y_0, \cdots, y_0], \quad [y_1, y_1, \cdots, y_1], \quad \cdots \quad [y_m, y_m, \cdots, y_m]] \tag{5}$$

List 11：等高線のグラフ

```
from pylab import *
x=arange(-3, 3.1, 0.1)
y=arange(-3, 3.1, 0.1)
xm, ym = meshgrid(x, y)
zm = sin(xm) + cos(ym)
cntr = contour(x, y, zm, levels=[-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5])
clabel(cntr, fmt='%.2f')
show()
```

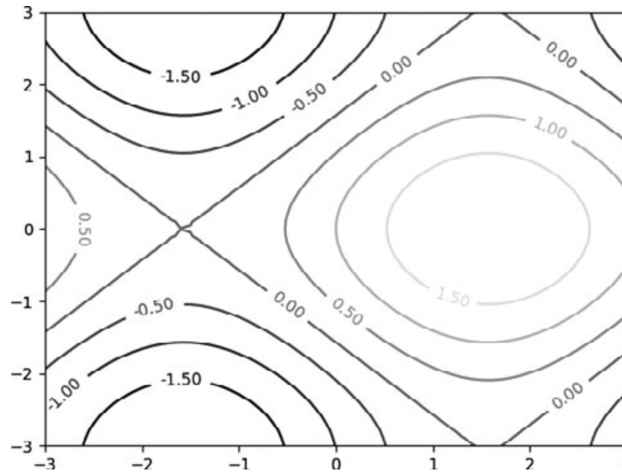


図11　List 11 の描画結果.

3. 2. 4. 3　ヒートマップのプロット　Heat map

(x, y, z)の値の分布を色の変化で示したい場合にヒートマップが用いられます．ヒートマップのプロットには，pcolormesh()関数を使います．例を **List 12**および描画結果を**図12**に示します．ほぼ，前節のcontour()関数の代わりに，pcolormesh()関数を使うだけです．キーワード引数 cmap に，カラーマップ（値と色を対応させるもの）を指定します．カラーマップの名前は，よく使われるもので‘jet’，‘hot’,‘rainbow’などがあります．色と値の対応を示すために，同時にカラーバーを描画することが一般的です．カラーバーは，colorbar()関数に，対象となるヒートマップを渡すことで描画できます．対象となるヒートマップは，pcolormesh()関数で描画したときに得られます．７行目で，pcolormesh()で得られたヒートマップを hmap に保存しておき，８行目で colorbar()関数に渡しています．colorbar()関数からは，描画したカラーバーが得られます．９行目では，８行目で得られたカラーバー cbar からカラーバー自身のグラフ枠（Axes）を取り出して，y 軸のラベルを設定する機能を呼び出してカラーバーにラベルを設定しています．

このプロットでは気が付かないかもしれませんが，描画範囲を狭めてメッシュが拡大されると，色の塗り方に問題が現れます．わかりやすいように，関数を $f(x, y) = e^{-(x^2+y^2)}$ を z として pcolormesh()関数でヒートマップを描画しました（**図13**左，**List 13**）．同時に等高線図も描画し，$(x, y) = (0, 0)$ 付近を拡大しています．この図のように，色と等高線がずれて見えます．これは，メッシュ内の色が，そのメッシュの左下の値に応じた色で塗られているためです．メッシュの値と塗りの中心をあわせたい場合には，塗りに使われるメッシュを計算に使われたメッシュサイズの半分だけシフトさせる必要があります．**List 13** の６行目，７行目でずらしたメッシュ位置を作成し，20行目の pcolormesh()関数に渡しています．注意していただきたいのは，この pcolormesh()に渡している shifted_x と shifted_y は，pcolormesh()でヒートマップの表示をずらして適切にするために渡しているだけで，ここで渡している zm の値は，あくまでも xm，ym の位置で計算したものであることです．（ずらした後の位置（shifted_x，shifted_y）で計算したものではありません．）

６行目，７行目のずらした位置を得るところについて少し説明します．’r_’は，numpy モジュールが提供するオブジェクトで，r_[a, b, c]とするとオブジェクトの機能（メソッド）によって，a，b，c，を行方向へ連結した配列を返してくれます．式を見ると，x(y)の２番目の要素から最後まで，間隔の半分を差し引いてずらしています．その先頭と最後に，x(y)の先頭の要素 x[0](y[0])と最後の要素 x[-1](y[-1])を連結させ付け加えています．

List 12：ヒートマップによる表示

```
from pylab import *
x=arange(-3, 3.1, 0.1)
y=arange(-3, 3.1, 0.1)
xm, ym = meshgrid(x, y)
zm = sin(xm) + cos(ym)
xlim(-3.0, 3.0)
hmap = pcolormesh(x, y, zm, vmin=-3, vmax=3, cmap='jet')
cbar = colorbar(hmap)
cbar.ax.set_ylabel('colorbar label')
show()
```

List 13：メッシュをずらす

```
from pylab import *
x=arange(-0.4, 0.41, 0.2)
y=arange(-0.4, 0.41, 0.2)
xm, ym = meshgrid(x, y)
zm = exp(-1*(xm**2 + ym**2))
shifted_x = r_[x[0], x[1:] - diff(x)*0.5, x[-1]]
shifted_y = r_[y[0], y[1:] - diff(y)*0.5, y[-1]]

subplot(121) # for using original mesh (left figure)
xlim(-0.5, 0.5)
ylim(-0.5, 0.5)
title('use x, y')
pcolormesh(x, y, zm, vmin=0, vmax=1, cmap='jet')
contour(x, y, zm, levels=[0.9, 0.95], colors='k')

subplot(122) # for using shifted mesh (right figure)
xlim(-0.5, 0.5)
ylim(-0.5, 0.5)
title('use shifted x, y')
pcolormesh(shifted_x, shifted_y, zm, vmin=0, vmax=1, cmap='jet')
contour(x, y, zm, levels=[0.9, 0.95], colors='k')

tight_layout(w_pad=3)
show()
```

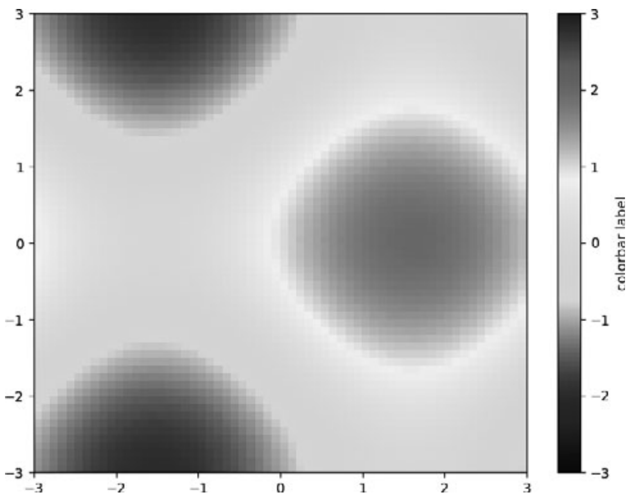


図12　List 12 の描画結果.

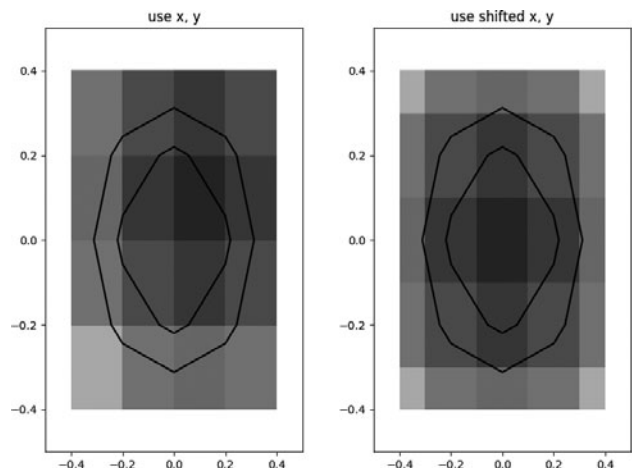


図13　List 13 の描画結果.

3.2.4.4　色付き散布図のプロット Scatter plot with color depend on values

zの値に応じて，色を変化させた散布図を描きたいこともあります．その場合は，scatter()関数を使います．例を**List 14**に示します．プロットする位置の座標x，yをそれぞれ第1引数，第2引数に渡します．キーワード引数cにzを渡します．キーワード引数markerにマーカーの種類を渡します（**図14**）．マーカーに渡す値については，plot()関数を参考にしてください．例では，'o'を渡して，丸を用いています．マーカーの大きさは，キーワード引数sにpointの二乗の単位で指定します．すなわち，10pointのマーカーなら100をsに渡します．範囲やカラーマップについては，pcolormesh()関数と同じです．

List 14：色付きの散布図による表示

```
1 from pylab import *
2 x = -3.0 + 6.0*random(100)
3 y = -3.0 + 6.0*random(100)
4 z = sin(x) + cos(y)
5 sct = scatter(x, y, c=z, s=100, marker='o', vmin=-3, vmax=3, cmap='jet')
6 cbar = colorbar(sct)
7 cbar.ax.set_ylabel('colorbar␣label')
8 show()
```

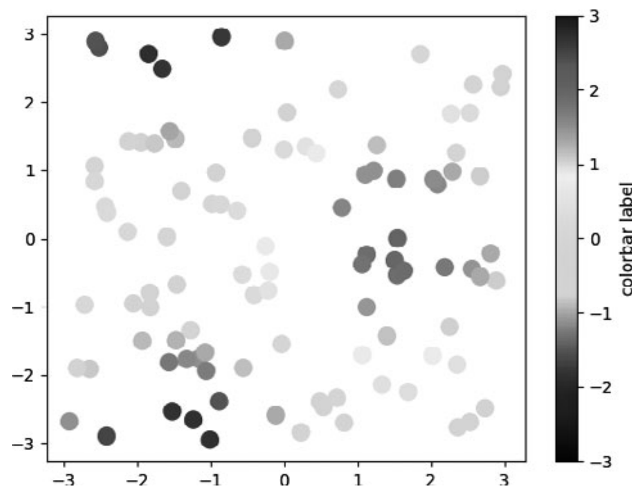


図14　List 14 の描画結果.

3.2.5　最後に

matplotlib の関数インターフェースを用いて，よく利用されるグラフを描画できるように紹介してきました．matplotlib は，オブジェクトの集合として構成されているため，関数インターフェースを使っていても，オブジェクトを関数に渡したり，オブジェクトの機能を呼び出したりと，どうしてもオブジェクトを介した記述が出てきてしまいました．オブジェクトについては，Python とそのモジュールを利用していくと自然とわかってくるかと思います．また，matplotlib の公式ページには，多くのプロット例があり，それを描画したスクリプトを見ることができます．公式のページ以外にも，インターネットでアクセスできる情報も豊富にありますので，検索してみると，解決策が見つかることでしょう．

さて，matplotlib を使ってみた感想はいかがでしょうか．望みのグラフを得るためには，いちいち設定しないといけないと，面倒に感じたでしょうか．そのような場合にこそ，いろいろな設定の手間なく，最低限のデータさえ与えれば望むようなプロットができる自分用の plot 関数を Python で書くときです．ぜひ，Python と matplotlib を活用してください．

J. Plasma Fusion Res. Vol.94, No.3 (2018) 162‐170

講座 今日からはじめる Python

3．Python による科学技術計算

3. Scientific Computing in Python

3.3 Python を用いた多次元データ解析・機械学習

3.3 Multi-Dimensional Data Mining and Machine-Learning with Python

藤 井 恵 介
FUJII Keisuke
京都大学 工学研究科

（原稿受付：2018年１月23日）

データ駆動科学が第４の科学的手法と言われ近年注目を集めています．なかでも Python は，データ科学の分野で最も人気を集めている言語です．Tensorflow や chainer，pytorch など，最新の深層学習ライブラリは全て Python にて実装されています．ただしデータ科学に重要なのは，深層学習など先端的なアルゴリズムだけではありません．手持ちのデータを素早く可視化して内容を理解したり，既存のアルゴリズムに渡せるように複数データを合成したり，不良値や欠損値を埋めたりのような前処理の部分が意外と重要で手間がかかることが知られています．

ここでは，このようなデータ科学にフォーカスされたライブラリを紹介します．まず多次元データ処理ライブラリとして，xarray の使い方を概観します．これらのライブラリはアルゴリズムというようりは，多様なデータを素早く扱うことができるインターフェースを提供するものです．次に，機械学習アルゴリズムが実装されているライブラリとして scikit-learn を紹介します．有名なアルゴリズムが数多く集められているライブラリで，様々なアルゴリズムを気軽に試すことが特徴です．最後に，高速な GPU 計算を簡単に実現することを目的として開発されている cupy というライブラリを簡単に紹介します．

なお，今回の記事で紹介する計測データには https://github.com/PlasmaLib/python_tutorial/tree/master/data からアクセスできます．ぜひダウンロードして，各自の PC で操作を体験してみてください．

Keywords:
Python, data science, libraries, machine learning, GPU computation

3.3.1 pandas と xarray

Python の有名なデータ解析ライブラリに pandas*1があります．pandas は，汎用的なデータ解析に関するインターフェースを提供するものです．その注目すべき特徴の１つが，座標付きのデータ（labeled data）をうまく扱うことができる点です．

例えば時系列データを扱うとき，時間データと計測値の２つを保持しておく必要が有ります．ある時間領域を切り出す時は同じ操作を両データに適用する必要が有りますし，またある時刻での計測値を知りたいときは，時間データから対応する要素番号を探しだし，計測データに対してその要素番号を用いてアクセスすることが必要です．これらの操作は前節で紹介した `np.ndarray` でももちろん可能ですが，毎回各自がコーディングしていると時間がかかるだけでなく，ミスの元になります．

このようなデータ解析に必要な汎用的な操作をまとめてインターフェースとして提供しているのが pandas です（日付関連の操作や，欠損値に対する操作，複数のデータをまとめる操作など，他にも多様な便利機能がまとめられています）．pandas はアンケート結果や株価の推移など，一般のデータ解析に広く用いられています．

一方でプラズマ計測データなど多くの物理データは，複数の座標軸を持ちます．例えば，電子温度分布の時間変化データは，計測位置と時間という２つの軸を持つことになります．ある時間における分布が知りたかったり，ある位置の温度の時間変化が知りたかったりします．しかし pandas は１次元のデータ（表に表すことのできるデータ）しか扱うことができません．xarray*2*3は，pandas を多次元

＊1 https://pandas.pydata.org/
＊2 https://xarray.pydata.org/
＊3 Hoyer, S. & Hamman, J., (2017). xarray: N-D labeled Arrays and Datasets in Python. Journal of Open Research Software. 5(1), p.10

Kyoto University, KYOTO 606-8501, Japan

author's e-mail: fujii@me.kyoto-u.ac.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research

データに拡張したもので，地球科学研究分野から生まれたものです．まだ新しいライブラリですが，他種類の多次元データを扱うことの多いプラズマ研究でも有用だと思われますので，ここで紹介します．

3. 3. 1. 1　xarray のインストール

Anaconda distribution を用いて Python 開発環境を整えた人は

```
conda install xarray
```

としてください．Native の Python を用いている人は

```
pip install xarray
```

とするとよいでしょう．

3. 3. 1. 2　xarray の使い方

ここでは，実際のプラズマ実験データを使って xarray の使い方を説明します．核融合科学研究所の大型ヘリカル装置（LHD）で計測されたトムソン散乱による電子温度・密度の結果を読み込みましょう．ファイルの読み込みプログラムやいくつかの計測データの例を https://github.com/PlasmaLib/python_tutorial/tree/master/dataに用意しました．これを *data* フォルダに保存して以下のように読み込みましょう．

```
In [1]: import xarray as xr  # xarray は xr と省略するのが一般的なようです

In [2]: import sys

In [3]: sys.path.append('data')

In [4]: import eg  # 読み込みプログラムも data/eg.py として保存しておいてください

In [5]: thomson = eg.load('data/thomson@115500.dat')  # thomson データの読み込み

In [6]: print(thomson)
<xarray.Dataset>
Dimensions:      (R: 140, Time: 246)
Coordinates:
  * Time         (Time) float64 0.0 33.0 66.0 100.0 133.0 166.0 200.0 233.0 ...
  * R            (R) float64 2.409e+03 2.436e+03 2.463e+03 2.49e+03 ...
    ShotNo       int64 115500
Data variables:
    Te           (Time, R) float64 90.0 20.0 22.0 83.0 46.0 54.0 27.0 0.0 ...
    dTe          (Time, R) float64 1e+05 1e+05 1e+05 1e+05 1e+05 1e+05 1e+05 ...
    n_e          (Time, R) float64 0.0 0.0 0.0 0.0 0.0 0.0 1.0 30.0 0.0 3.0 ...
    dn_e         (Time, R) float64 2.0 2.0 2.0 2.0 2.0 2.0 2.0 243.0 1.0 ...
    laser        (Time, R) float64 913.0 913.0 913.0 913.0 913.0 913.0 913.0 ...
    lasernumber  (Time, R) float64 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 ...
Attributes:
    NAME:     THOMSON
    Date:     11/19/201215:24
```

まず，`print` 文を用いた時の出力が綺麗に整形されていることがわかります．`Dimensions` の行に`(R: 140, Time: 246)`とあるのは，含まれる変数は，２つの次元`Time`と`R`に依存するということ，それらの大きさがそれぞれ140，246であることを示しています．`Coordinates` セクションには，それら軸の座標の値が表示されています．LHDのトムソン散乱では，電子温度や密度やその推定誤差などが得られますが，それらが`Data variables`セクションに含まれています．`Data variables` セクションの例えば `Te` の列には（`Time, R`）とありますが，これはこの変数が `Time` と `R` に依存するということを示しています．

xarray オブジェクトの `Coordinates` や `Data variables` には，辞書型のように`['Te']`というようにすることでアクセスできます．

```
In [7]: thomson['Te']
Out[7]:
<xarray.DataArray 'Te' (Time: 246, R: 140)>
array([[   90.,    20.,    22., ...,     0.,    82.,  1796.],
       [    6.,    17.,    10., ...,    19.,    26.,    13.],
       [    7.,     0.,    40., ...,    14.,     6.,    57.],
       ...,
       [    6.,   133.,     0., ...,    43.,    40.,    23.],
       [   10.,     9.,    48., ...,    70.,    53.,    58.],
       [    7.,    39.,    14., ...,    26.,     0.,    15.]])
Coordinates:
  * Time     (Time) float64 0.0 33.0 66.0 100.0 133.0 166.0 200.0 233.0 ...
  * R        (R) float64 2.409e+03 2.436e+03 2.463e+03 2.49e+03 2.517e+03 ...
    ShotNo   int64 115500
Attributes:
```

```
    Unit:     eV
```

このように１種類のデータを選んでも，同時に座標情報が付随していることがわかります.

ラベルを用いたインデクシング

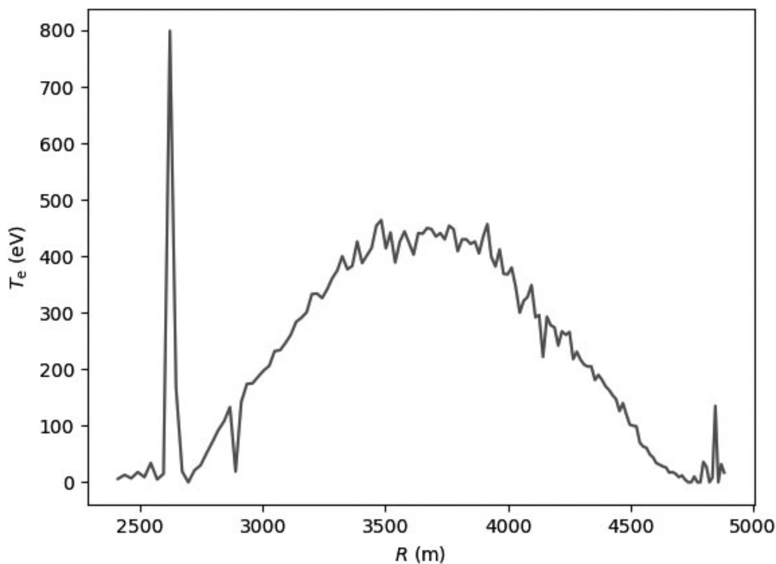
Coordinates セクションに Time，R が表示されているように，このデータには座標情報も付属します．.sel メソッドを用いることで，座標軸を元に要素を選択することができます.

```
In [8]: thomson.sel(Time=6800.0, method='nearest')
Out[8]:
<xarray.Dataset>
Dimensions:      (R: 140)
Coordinates:
    Time         float64 6.8e+03
  * R            (R) float64 2.409e+03 2.436e+03 2.463e+03 2.49e+03 ...
    ShotNo       int64 115500
Data variables:
    Te           (R) float64 6.0 13.0 7.0 18.0 9.0 34.0 5.0 15.0 799.0 167.0 ...
    dTe          (R) float64 9.0 5.0 1e+05 29.0 0.0 12.0 1.0 2.0 134.0 0.0 ...
    n_e          (R) float64 -4.0 -13.0 1.0 -9.0 -3.257e+04 19.0 49.0 91.0 ...
    dn_e         (R) float64 2.0 1.0 2.0 3.0 121.0 3.0 5.0 6.0 1.0 59.0 8.0 ...
    laser        (R) float64 912.0 912.0 912.0 912.0 912.0 912.0 912.0 912.0 ...
    lasernumber  (R) float64 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 ...
Attributes:
    NAME:     THOMSON
    Date:     11/19/201215:24
```

ここでは，Time 軸が 6800.0 に最も近い計測値を取得しています．Dimensions の行から Time が消えて（R：140）だけになったことからもわかるように，全ての計測値を一度にインデクシングしていることがわかります.

ある時刻の結果だけグラフに描きたい，ということもよくありますが，その場合も，.sel メソッドを用いることで１行で実現できます.

```
In [9]: plt.plot(thomson['R'], thomson['Te'].sel(Time=6800.0, method='nearest'))
Out[9]: [<matplotlib.lines.Line2D at 0x7f67c71b3470>]

In [10]: plt.xlabel('$R$ (m)')
Out[10]: <matplotlib.text.Text at 0x7f67cdedd6d8>

In [11]: plt.ylabel('$T_mathrme$ (eV)')
Out[11]: <matplotlib.text.Text at 0x7f67cdee6780>
```



座標名を利用した操作

xarray オブジェクトは，軸に名前をつけて保持しているので，データが格納されている配列の軸の順序（１つ目の軸が Time，２つ目の軸が R に対応している，など）を覚えておく必要がありません.

例えば，プラズマ中心での電子温度の時間変化が知りたいとしましょう．ただし，ある一点の計測値のみを用いるとノイズが大きくなるので，ある程度の範囲内でデータを平均するのがよいでしょう．今回は，プラズマ中心付近の 3500～3800 mm での温度を平均することにします．ま

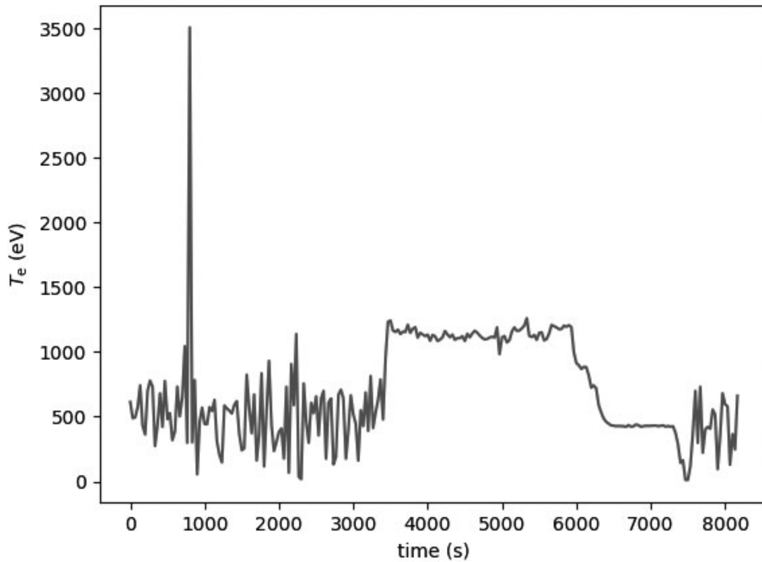
ず，データに付属する軸情報から，この範囲内に収まるデータのみを切り出します．上記と同様に sel メソッドを用いることができます．

```
# R = 3500.0 - 3800.0 間のデータを選択
In [12]: thomson_center = thomson.sel(R=slice(3500.0, 3800.0))
```

```
In [13]: thomson_center
Out[13]:
<xarray.Dataset>
Dimensions:      (R: 17, Time: 246)
Coordinates:
  * Time         (Time) float64 0.0 33.0 66.0 100.0 133.0 166.0 200.0 233.0 ...
  * R            (R) float64 3.502e+03 3.521e+03 3.54e+03 3.559e+03 ...
    ShotNo       int64 115500
Data variables:
    Te           (Time, R) float64 2.099e+03 8.383e+03 1.446e+04 5.0 ...
    dTe          (Time, R) float64 1e+05 1e+05 2.882e+04 1e+05 1e+05 1e+05 ...
    n_e          (Time, R) float64 0.0 0.0 3.0 11.0 0.0 0.0 0.0 3.0 0.0 0.0 ...
    dn_e         (Time, R) float64 0.0 1.0 8.0 51.0 0.0 0.0 1.0 26.0 1.0 1.0 ...
    laser        (Time, R) float64 913.0 913.0 913.0 913.0 913.0 913.0 913.0 ...
    lasernumber  (Time, R) float64 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 ...
Attributes:
    NAME:     THOMSON
    Date:     11/19/201215:24
```

このデータを大半径方向に平均することにしましょう．平均操作には mean メソッドを用います．引数として，2 つの軸（R, Time）のうち，どちらの方向に平均をとるかを指定します．

```
In [14]: thomson_center.mean(dim='R')  # 'R' 方向の平均
Out[14]:
<xarray.Dataset>
Dimensions:      (Time: 246)
Coordinates:
  * Time         (Time) float64 0.0 33.0 66.0 100.0 133.0 166.0 200.0 233.0 ...
Data variables:
    Te           (Time) float64 3.493e+03 4.575e+03 8.478e+03 834.7 ...
    dTe          (Time) float64 8.549e+04 1.951e+04 9.317e+03 7.085e+03 ...
    n_e          (Time) float64 1.0 1.474e+03 861.5 1.352e+03 668.2 373.5 ...
    dn_e         (Time) float64 5.647 12.18 12.12 16.76 10.59 9.647 9.118 ...
    laser        (Time) float64 913.0 914.0 910.0 912.0 909.0 912.0 912.0 ...
    lasernumber  (Time) float64 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 ...

In [15]: plt.plot(thomson_center['Time'], thomson_center['Te'].median(dim='R'))
Out[15]: [<matplotlib.lines.Line2D at 0x7f67c7d77f98>]

In [16]: plt.xlabel('time (s)')
Out[16]: <matplotlib.text.Text at 0x7f67c71f3fd0>

In [17]: plt.ylabel('$T_mathrme$ (eV)')
Out[17]: <matplotlib.text.Text at 0x7f67c723e6a0>
```



座標を用いた異種データの結合

異なる時間間隔で計測されたデータ間を結合したい時もあると思います．例えば，トムソン散乱と干渉計による電子密度の計測結果を比べることを考えます．ただし，トムソン散乱と干渉計では計測タイミングが異なるので直接は比較できません．この場合に限らず，複数種のデータを扱う際はこのような操作が必須です．pandas や xarray には簡便な方法が用意されています．

ここではまず，干渉計のデータを読み込みましょう．同様に，data ディレクトリ内にデータを用意しておいてください．

```
In [18]: fir = eg.load('data/firc@115500.dat')  # 干渉計データの読み込み

In [19]: fir
Out[19]:
<xarray.Dataset>
Dimensions:   (Time: 1311)
Coordinates:
  * Time      (Time) float64 0.1 0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18 ...
    ShotNo    int64 115500
Data variables:
    nL(3669)  (Time) float64 0.0001822 0.0001822 0.0001822 0.0001822 ...
    nL(3759)  (Time) float64 -0.003244 -0.003244 -0.003244 0.005867 ...
Attributes:
    NAME:     firc
    Date:     11/19/201222:20
```

干渉計の計測時刻は（0.1, 0.11, ...）となっており，トムソン散乱の計測時刻（0.3, 0.33, ...）と異なることがわかります．ここでは，トムソン散乱の計測時刻に最も近い計測時刻での干渉計のデータを集めてくることにしましょう．そのためには reindex メソッドを利用できます．引数に，比較したい軸（今回は Time）とその比較対象，およびその方法を指定します．今回は最近傍のもの（nearest）を取得します．

```
# thomson の時刻が [ms]なので [s] に修正する.
In [20]: thomson['Time'] = thomson['Time'] * 0.001

# thomson の時刻に最も近い fir 計測結果を取得する.
In [21]: fir_selected = fir.reindex(Time=thomson['Time'], method='nearest')

In [22]: fir_selected
Out[22]:
<xarray.Dataset>
Dimensions:   (Time: 246)
Coordinates:
  * Time      (Time) float64 0.0 0.033 0.066 0.1 0.133 0.166 0.2 0.233 0.266 ...
    ShotNo    int64 115500
Data variables:
    nL(3669)  (Time) float64 0.0001822 0.0001822 0.0001822 0.0001822 ...
    nL(3759)  (Time) float64 -0.003244 -0.003244 -0.003244 -0.003244 ...
Attributes:
```

```
NAME:     firc
Date:     11/19/201222:20
```

これで両者の演算が可能になりました．

その他の特徴

xarray は他にも様々な便利機能を備えています．

- 軸・データの関係を記録する netCDF*4ファイルへの入出力
- dask*5を用いたメモリに格納できない規模のデータの取り扱い，並列計算

ここではこれらを説明する紙面の余裕がありませんが，どれも広い分野のデータ解析に有用な機能となっています．xarray の document ページ http://xarray.pydata.org をご参考ください

こういったライブラリが提供するツールは，コーディングさえすれば NumPy だけでも実現できます．そのため，時間をかけてライブラリの使用法を習得しようというインセンティブが湧かないかもしれません．しかし毎回自身でコーディングすることは，試行錯誤のスピードを低下させるだけでなく，ケアレスミスも誘発します．

最初に使用法を覚える段階はまどろっこしく自身で作った方が早いように感じますが，慣れてしまうとこのようなライブラリを用いる方が圧倒的に操作が楽に確実になります．有用なツールの習得に時間をかけるのは，ちょっとした投資と言えるかもしれません．

3.3.2 機械学習ライブラリ scikit-learn

Scikit-learn は Python で最も有名な機械学習ライブラリでしょう．主な特徴は以下のようなものです．

- 一般的な機械学習アルゴリズムが網羅されている
- ドキュメントがよく整備されており，素人でも簡単に利用できる
- 簡単に理解できる平易な実装となっている
- 計算負荷の高いところは C++で実装されているため比較的高速

機械学習というと，IT 関係，人工知能関係のごく限られた内容を想定されるかもしれませんが，そんなことはありません．機械学習とは統計と言い換えてもほとんど問題ないでしょう．実験データや計算データなど，データ解析に関わる問題であれば関わる可能性も高いと思います．

主に

- 回帰問題
- 教師有り分類問題
- 教師無し分類問題
- 次元圧縮・特徴抽出
- モデル選択
- データ前処理

に関する様々なアルゴリズムが実装されています．

3.3.2.1 scikit-learn のインストール

Anaconda distribution を用いて Python 開発環境を整えた人は

```
conda install scikit-learn
```

としてください．Native の Python を用いている人は

```
pip install scikit-learn
```

とするとよいでしょう．

3.3.2.2 scikit-learn の使用例

scikit-learn では様々な手法が利用できますが，ここでは線形回帰を紹介します．線形回帰とは，計測データ y を基底関数 $(\phi(x))$ の線形和で近似するものです．

$$y_i \approx \sum_j w_j \phi_j(x_i)$$

ここで，(w_j) は線形結合の係数です．$\phi(x)$ にどういった式を用いるかで，多項式近似や，スプライン近似などを表すことができる汎用的なモデルです．scikit-learn では `linear_model` モジュールとして提供されています．ここでは，前節で紹介したトムソン散乱による電子温度分布を複数のガウス関数の和で近似することにします．

最も簡単な近似法は，最小二乗法です．最小二乗法では，

$$\sum_i \left(y_i - \sum_j w_j \phi_j(x_i) \right)^2$$

を最小化する w_j を求めます．

最小二乗法は，ノイズがガウス分布に従っていると仮定したものです．しかし，一般にノイズが綺麗なガウス分布に従っているとは限りません．例えば前節に示したトムソン散乱による電子温度分布にも，外れ値と言えるようなデータ点も見受けられます．

このような外れ値を含むようなデータの近似アルゴリズムも様々に存在します．scikit-learn でも複数種類実装されていますが，ここでは Huber 回帰という手法を用いたものを紹介します．

Huber 回帰では，通常の最小二乗法と異なり，以下のコスト関数を最小化します．

$$\sum_i \mathcal{H}\left(\frac{y_i - \sum_j w_j \phi_j(x_i)}{\sigma} \right)$$

ここで，σ は外れ値を除いたノイズの分散で，これもデータから推定します．$\mathcal{H}(z)$ は Huber のロス関数で，以下のように定義されるものです．

$$\mathcal{H}(z) = \begin{cases} z^2 & |z| \leq 1 \\ 2|z| - 1 & |z| > 1 \end{cases}$$

簡単には，近似値の近くにある点については通常の二乗誤差を考え，σ より遠く離れている点については一乗誤差

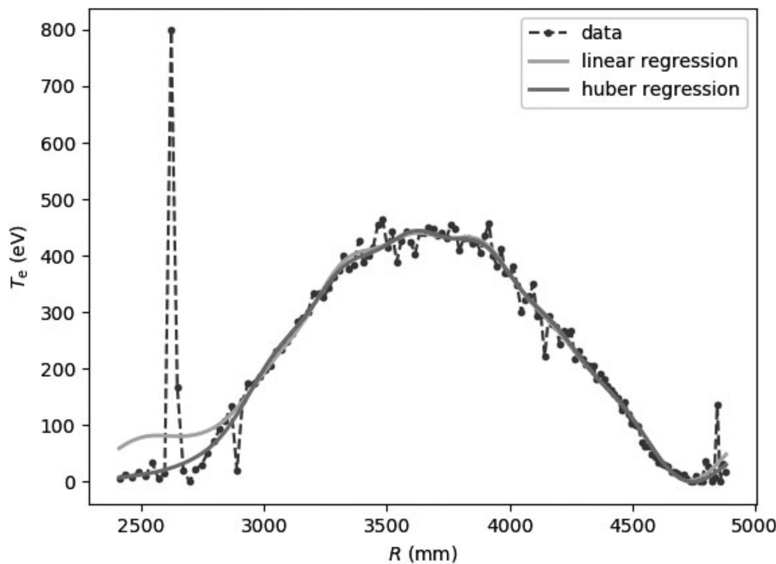
*4 https://www.unidata.ucar.edu/software/netcdf/

*5 https://dask.pydata.org/en/latest/

に緩和してロス関数に加えるものです．遠く離れている点についてのコストが小さくなるため，外れ値に引っ張られることが少なくなります．

ここでは，scikit-learn を用いて Huber 回帰を行ってみます．

```
In [1]: from sklearn import linear_model  # linear_model モジュールを用います

# data ここでは6800 msに得られたTeの分布を解析します
In [2]: Te = thomson.sel(Time=6800, method='nearest')['Te'].values

In [3]: R = thomson['R'].values

# basis R:2500--5000 を10分割した点を中心とするガウス関数の和で近似しましょう
In [4]: centers = np.linspace(2500, 5000, 10)

In [5]: phi = np.exp(-((R.reshape(-1, 1) - centers) / 200)**2)

# 最小二乗法
In [6]: lin = linear_model.LinearRegression(fit_intercept=False)

# フィッティング
In [7]: lin.fit(phi, Te)
Out[7]: LinearRegression(copy_X=True, fit_intercept=False, n_jobs=1, normalize=False)

# 求めたフィッティング係数を用いた予測
In [8]: Te_lin_fit = lin.predict(phi)

# ロバスト最小二乗法
In [9]: rob = linear_model.HuberRegressor(fit_intercept=False)

# フィッティング
In [10]: rob.fit(phi, Te)
Out[10]:
HuberRegressor(alpha=0.0001, epsilon=1.35, fit_intercept=False, max_iter=100,
        tol=1e-05, warm_start=False)

# 求めたフィッティング係数を用いた予測
In [11]: Te_rob_fit = rob.predict(phi)

In [12]: plt.plot(R, Te, '--o', ms=3, label='data')
Out[12]: [<matplotlib.lines.Line2D at 0x7f67c3607940>]

In [13]: plt.plot(R, Te_lin_fit, label='linear regression', lw=2)
Out[13]: [<matplotlib.lines.Line2D at 0x7f67c7db60b8>]

In [14]: plt.plot(R, Te_rob_fit, label='huber regression', lw=2)
Out[14]: [<matplotlib.lines.Line2D at 0x7f67c358d9e8>]

In [15]: plt.legend(loc='best')  # 凡例を表示する
Out[15]: <matplotlib.legend.Legend at 0x7f67c7db6198>

In [16]: plt.xlabel('$R$ (mm)')
Out[16]: <matplotlib.text.Text at 0x7f67c4a4a400>

In [17]: plt.ylabel('$T_mathrme$ (eV)')
Out[17]: <matplotlib.text.Text at 0x7f67c35ca518>
```



通常の最小二乗法では，異常値に引きずられて $R=2500$ mm 付近で多くの計測点から離れているのに対し，Huber 回帰ではこれら異常値に頑健なフィッティングができていることがわかります．

また実装面では，`LinearRegression` と `HuberRegressor` は引数・戻り値などの使い方が統一されています．そのため，`LinearRegression` では外れ値に影響されすぎていると感じれば，すぐに `HuberRegressor` などのロバストな回帰手法を試すことができるようになっています．他にも様々なアルゴリズムがよく似たインターフェースで提供されており，簡単に試行錯誤を重ねることができる，ということがこのようなライブラリを用いることのメリットだと思います．

この節では，回帰問題を通して scikit-learn の使い方を簡単に紹介しました．上記の回帰問題からもわかるように，全てのデータに無条件で合うモデルは存在しません．よいモデルはデータに依存するため，色々なモデルを適用してみて結果を見るというような多数回の試行錯誤が必要です．scikit-learn では，そのような試行錯誤を簡単にできるよう工夫されて作られています．たくさんの例がドキュメントページにまとめられているので，ぜひそちらもご覧ください．

3.3.3 Python での高速計算 〜NumPy互換GPU計算ライブラリCupy〜

Cupy は，簡単に GPU 計算を実装するためのライブラリです．これまで紹介した数値計算ライブラリ NumPy と同じ API を提供しているため，NumPy でプログラムを作っておいて，きちんと動くことを確認してから Cupy に変更する，といった使い方が可能です．なんと言ってもデバッグが簡単で，非常に気軽に GPU 計算を試すことができます．ここでは Cupy の使い方を簡単にご紹介します．

3.3.3.1 Cupy の特徴

Cupy は日本のベンチャー企業 Preferred Networks が主導して開発しているオープンソースライブラリです．同企業が開発している深層学習ライブラリ Chainer で用いられています．

GPU 計算には，例えば NVIDIA が提供するライブラリの CUDA[1] を呼び出して実行する必要があります．しかしそのインターフェースは非常に低レベルで，なかなか素人が気軽に使えるものではありません．さらに，もしそのインターフェースに習熟したとしても，低レベルなインターフェースからバグのないアルゴリズムを組み立てることは非常に骨の折れるものとなります．

Cupy を用いることで，すごく気軽に GPU 計算を行うことが可能になります．Python から GPU 計算を行うライブラリは複数ありますが，Cupy の特徴はなんといっても，NumPy と（ほとんど）同じ API を提供する点です．そのため，使い方の習得が容易で，デバッグも非常に簡単です．NumPy で開発したプログラムを GPU 用に移植することも簡単でしょう．

3.3.3.2 Cupy のインストール

Cupy をインストールするには，NVIDIA 製 GPU とそのドライバ，さらに CUDA ライブラリをインストールしておく必要があります．これらのインストールについては https://docs-cupy.chainer.org/en/stable/install.html を参照してください．

上記作業の後に，

```
pip install cupy
```

とすることで Cupy をインストールすることができます．

3.3.3.3 Cupy の使い方

上で述べたように，Cupy は NumPy と同じ API で設計されているので，使い方もほとんど NumPy のものと同じです．以下は，ランダムに生成した行列の行列積を求めるプログラム例です．

```
import numpy as np
A_cpu = np.random.randn(10000, 20000)
B_cpu = np.random.randn(20000, 30000)
# numpy を使って CPU で行列積を計算する
AB_cpu = np.dot(A_cpu, B_cpu)
```

この計算を，Cupy を用いて GPU 上で行うには，以下のようにします．

```
import cupy as cp  # cupyはcpと略すのが一般的なようです

# np.ndarrayからGPU上のメモリにデータを移動する
A_gpu = cp.ndarray(A_cpu)
B_gpu = cp.ndarray(B_cpu)

# cupyを使ってGPUで行列積を計算する
AB_gpu = cp.dot(A_gpu, B_gpu)

# メインメモリ上にデータを移動する
AB_cpu2 = AB_gpu.get()  # AB_cpu2 は np.ndarray 型
```

A_gpuはcp.ndarray型のオブジェクトであり，実体はGPU メモリ上に生成されています．cp.dot 関数で，GPU メモリ上に確保された配列の行列積を GPU を用いて計算します．この関数の戻り値もcp.ndarray型のオブジェクトであり，実体は同様に GPU メモリ上にあります．get メソッドを実行することで，この配列をCPU メモリ上に移すことができます．この戻り値はnp.ndarray型となります．

その他にも，要素積や FFT，特異値分解など，NumPy の多くの操作が実装されています．詳しくは，公式ページ[2]をご覧ください．

ここでは，非常にお手軽にGPU計算を行えるライブラリとして Cupy を紹介しました．もちろん，GPU を使って高速化するには GPU 計算についての知識が必要です．例えば，GPU は行列計算のような並列処理は得意で CPU より数10倍速く計算できることもしばしばですが，条件分岐が重なるなど並列化が難しい処理の速度は非常に遅くなります．また，メインメモリ・GPU メモリ間のデータの移動がボトルネックとなることが多いため，そういった操作は最小限に留める必要があります．

とは言え，このような GPU 計算の長所・短所も，色々なプログラムを実装しながら学んでいけるというのがこういったライブラリを使うメリットだと思います．

3.3.4 まとめ

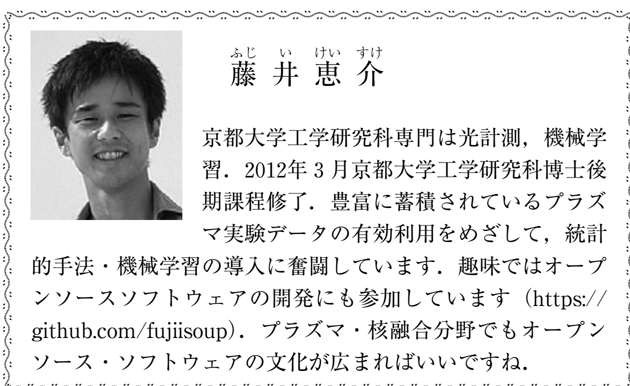
この章では，データ科学に有用なライブラリを紹介しました．xarray で行うことのできる操作は非常に基本的で日常的に行うものです．そのためこのような操作は自身でもできると感じる方も多いでしょう．また，scikit-learn を用いた機械学習や　Cupy で行う GPU 計算は，そのコアの部分が隠蔽されたインターフェースになっているのでもどかしく，自身で作らないと安心できないと感じるかもしれません．

しかし，毎回低レベルな部分からプログラムを開発するには非常に労力がかかります．デバッグ作業も骨の折れる作業でしょう．これらのライブラリの特徴は，すぐに使えることにあります．ファインチューニングを施した高速なアルゴリズムを開発するよりも，得られたデータに対して様々なアルゴリズムを気軽に試して，よりたくさんの試行錯誤を行うことが，データ解析を進める上でも効果的だと思います．

なお，これらのオープンソースライブラリは，複数人でテストを重ねながら開発が進められことが一般的です．そのため，個人でプログラムを開発するよりもバグが少ないことが期待されます．ソースコードも公開されていますので，内容を確認したいという方はぜひソースコードを読んでみてください．

参 考 文 献

［1］https://developer.nvidia.com/cuda-downloads
［2］https://cupy.chainer.org/



藤井恵介（ふじい けいすけ）

京都大学工学研究科専門は光計測，機械学習．2012年 3 月京都大学工学研究科博士後期課程修了．豊富に蓄積されているプラズマ実験データの有効利用をめざして，統計的手法・機械学習の導入に奮闘しています．趣味ではオープンソースソフトウェアの開発にも参加しています（https://github.com/fujiisoup）．プラズマ・核融合分野でもオープンソース・ソフトウェアの文化が広まればいいですね．

J. Plasma Fusion Res. Vol.94, No.4 (2018) 212‐215

講座 今日からはじめる Python

4. Python の活用

4. Application of Python

4.1 LHD 実験における Python の活用

4.1 Application of Python in LHD Experiment

江 本 雅 彦
EMOTO Masahiko
核融合科学研究所

（原稿受付：2018年 2 月26日）

myView2 は LHD の実験データを閲覧するため，主として共同研究者向けに開発されたアプリケーションであり，共同研究者が利用しやすいよう，無料配布可能で，様々な OS で実行可能にするため，Python を用いて開発されました．この章では，myView2 の概要を紹介するとともに，多様な環境で動かすアプリケーションを作成する際の注意点を解説します．

Keywords:
myView2, LHD

共同研究施設である核融合科学研究所で行われる大型ヘリカル実験には，多くの国内および海外の大学・研究所の研究者や学生が参加しています．彼らの多くは，短期滞在中に実験を行ってデータの解析等をするため，実験で収集されたデータは，携帯したノートパソコンで直ぐにでも取得・閲覧できる必要があります．この様な目的の為に共同研究者に配布可能なデータの可視化ソフトの条件としては，以下のようなものが挙げられます．

1．Windows を始めとする，多くの OS 上で動作すること
　研究者が携帯するノートパソコンの OS は，Windows だけでなく，MacOS もあり，また，多くの研究者が利用するという点では，Linux での利用もあり，これらの OS 上で動作する必要があります．

2．実行環境が無料で配布できること
　データの可視化を行うためのソフトウェアとしては，IDL や MATLAB 等が広く使用されていますが，これらは商用のソフトウェアです．この様なソフトウェアを使用することは，利用者に負担がかかるため，オープンソース，または，無料で配布できる必要があります．

3．カスタマイズが容易であること
　可視化するデータは，既存のものだけでなく，共同研究者が新たに設置した機器によって取得されたデータも扱える必要があります．また，関心ある内容に応じて表示形式を自由に変えられる必要もあります．

以上のような条件を満たすため，オープンソースのプログラミング言語である Python を用いた，可視化ツール my-View2 を開発しました（**図 1** ）．

myView2 は，ほぼ全て（*）を Python2.7 で記述しており，Windows，MacOS，Linux 上で動作可能です．このソフトは，大きく分けると，本体部分と，実験データの読み込みを行うデータローダの 2 つから構成されています．データローダはデータ取得に必要なインタフェイスを実装した Python のモジュールであり，百以上の LHD 実験データに特化したデータモジュールが提供されています（**図 2，3** ）．ただし，myView2 自体は汎用の可視化ツールとして使うことを目的に作られたものであり，それぞれのデータフォーマットに対応したデータローダを用意することで，本体側のソフトウェアを変更することなく，大型ヘリカル装置（LHD）実験以外の様々な実験データを可視化することができます．

グラフは，画面上の任意の矩形領域に複数のデータを表示することができ，通常の X-Y グラフの他にヒートマップ表示を行うことができます（**図 1** 参照）．

使用するデータローダや，グラフの配置等は，設定ファイルとして保存することができ，
計測毎に必要な情報を把握しやすいようにあらかじめレイ

（*）一部，LHD の解析データを取得するためのコマンドが，バイナリで配布されています．

National Institute for Fusion Science, Toki, GIFU 409-5292, Japan

author's e-mail:emo@nifs.ac.jp

Ⓒ2018 The Japan Society of Plasma
Science and Nuclear Fusion Research

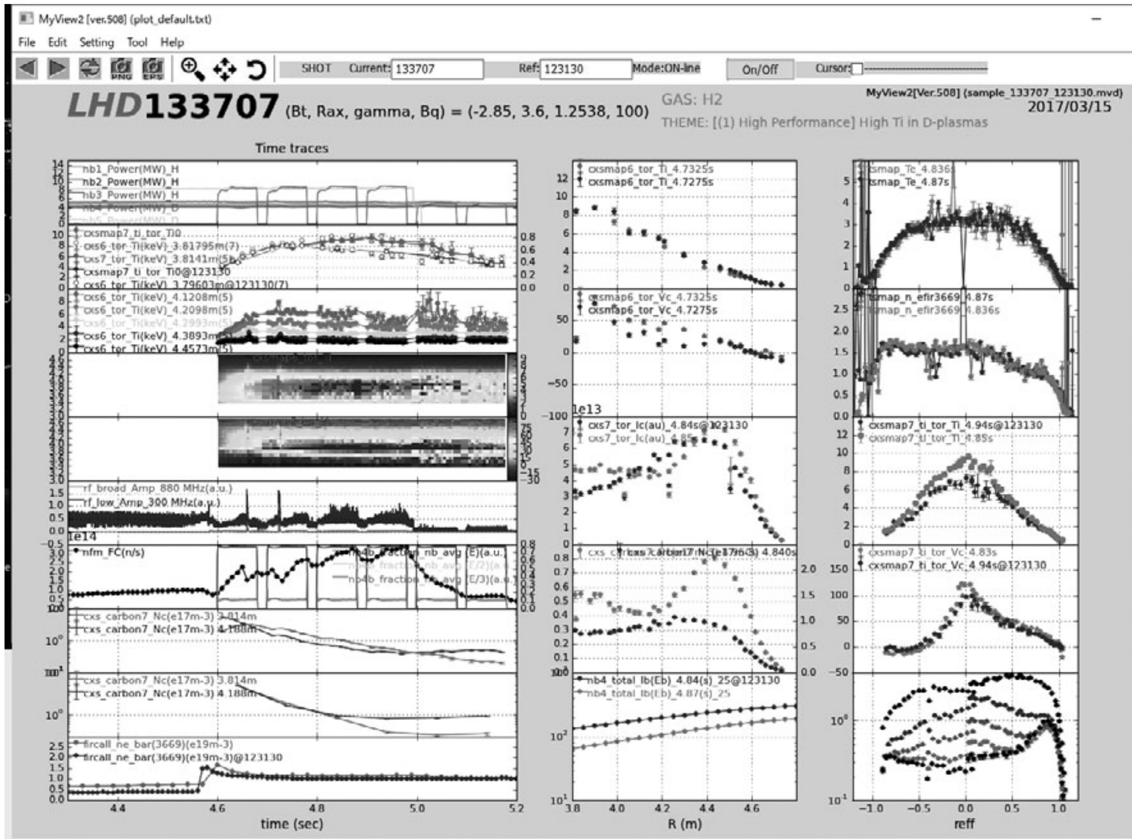


図 1　myView2 実行画面（Windows 10 で実行）.

```
class InterFace(mask.Mask):
      # コンストラクタ
      def __init__(self, dataid, **kwargs):
      # データの取得
      def load(self, **kwargs):
      # x 軸の配列データを取得
      def varsx(self, **kwargs):
      # y 軸の配列データを取得
      def varsy(self, **kwargs):
      # データの次取得
      def dataset(self, **kwargs):
```

図 2　データーローダクラス.

アウトしたものを提供することが可能です．このような設定ファイルはネットワーク共有フォルダ上で提供されており，これらのファイルを使うことにより，所外の共同研究者が必要なデータを可視化したい場合でもすぐに参照できます．表示するデータは，データサーバから直接取得する他，ハードディスク上にキャッシュされたものを表示することができます．データの取得先をキャッシュに切り替えることで，データサーバに接続されていない状態でもデータを可視化でき，共同研究者が自身の研究所に戻ってからもデータの解析を継続することが可能です．また，データベースサーバに接続可能な場合は，キャッシュされたデータと比較を行い，データが更新されていれば，データベー

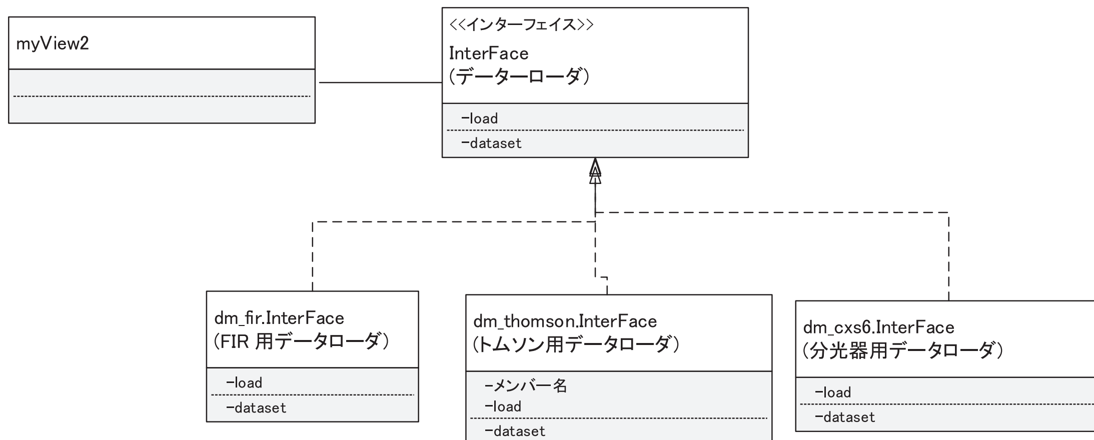


図 3　クラス図　各データローダーはデータローダーインタフェイスを実装.

スサーバから再取得を行います.

また，LHD 実験に特化した機能として，実験シーケンスに応じて自動的にデータの更新を行うことができます．この機能を利用して，実験中は制御室の中央に用意した10数台の PC で直近のデータを表示させ，実験責任者が実験の状況を把握するのに貢献しています.

myView2 の開発言語である Python は，核融合プラズマを含む広い科学技術分野で利用されており，３章で紹介したように，科学技術計算を行うためライブラリである SciPy や機械学習のためのライブラリである scikit-learn 等，便利なライブラリが多数公開されています．myView 2では，matplotlib および，wxPython 等を利用しています．matplotlib は２次元のみならず，３次元のプロットが行えます．wxPython は Python で記述された，プラットフォームに依存しない GUI（グラフィカル・ユーザインターフェース）を提供するライブラリであり，wxPython を利用することで，様々な OS 上で動作する GUI アプリケーションを作ることができます.

これらのライブラリは，実行速度を得るために，Python だけでなく，C 言語等で書かれた外部ライブラリを必要としています．一般的には，これらのライブラリはソースコードで配布され，各ユーザがコンパイルしてインストールするという作業が必要です．標準的なコンパイラが用意されている Linux や MacOS 等では，pip という Python のパッケージ管理コマンドを使うことで，必要なファイルを取得し，自動的にコンパイルおよびインストールが行われます．一方，Windows ではバイナリによる配布が基本ですので，Windows でこれらのライブラリを個別にインストールする際には注意が必要です.

Python 自体はオープンソースで公開されているため，Windows 用の Python は無料・有料を含め複数のディストリビュータが存在しています．これらの処理系に Python の拡張ライブラリを追加する際には，コンパイラの種類や依存しているバージョン等に矛盾が無いようにしなければなりません．OS 標準（もしくはそれに準ずる）Python と，C コンパイラが提供される MacOS や Linux では，標準的な環境でコンパイルすることで，一貫性のあるパッケージを用意することができますが，Windows ではそれが困難になります.

また，OS が提供するライブラリのバージョンにも注意が必要です．myView2 は GUI に wxPython 2 以降を使用しています．wxPython は個々の OS が提供する GUI ライブラリの違いを吸収するように実装され，どの OS でも同じインタフェイスを提供しています．Linux の場合，wxPython は下位の GUI ライブラリとして GTK を利用しますが，CentOS6 で提供している GTK のバージョンは 1.2 であり，myView2 が必要としている wxPython の新しいバージョンをインストールすることができません．CentOS 等，多くの Linux ディストリビューションでは，ライブラリやアプリケーションのバージョンは複雑に依存しているので，単純にライブラリを置き換えることは容易ではありません.

このようなパッケージ管理にまつわる煩わしさを避けるため，よく使われるパッケージが予め用意され，簡単に必要なパッケージのインストールができる，Python の開発環境を使用することが勧められています．この種の開発環境としては，Anaconda　や Enthought Canopy があり，myView2 は双方の環境での動作チェックを行っています．ただし Anaconda は Python の一貫した環境を提供するものであり，OS の標準ライブラリを置き換えることは行いません．このため，CentOS6 では先に述べたような wxPython の問題が発生し，myView2 を動作させることができません．一方，Canopy はこれらのライブラリも提供するので，動作可能です．このことから，myView2 の実行環境として Enthought Canopy が推奨されています.

myViw2 自体は，すべて Python で記述されていますが，OS によって振る舞いが違う部分があるため，一部 OS 毎に処理を分けている箇所があります．一つは，LHD の実験データを取得する部分です．この部分は Python で全てを記述するのが困難であるため，あらかじめ各 OS 用に用意したデータ取得用のバイナリコマンドを呼び出しています．この際，OS 毎にバイナリの保存場所や呼び出しの際の環境変数設定が異なるため，OS 毎に依存した処理を行っています.

また，myView2 では，上記の様に LHD 実験に対応するため，実験シーケンスに応じて，自動的にグラフを更新する機能がありますが，これは，マルチスレッドで実現しています．ところが，Linux でグラフィックの書き換えを複数のスレッドで同時に行うと，クラッシュしてしまいます．そこで，画面の描画を一つのスレッドによって行わせるために，callAfter というメソッドを呼び出しています．一方，このような問題は MacOS や Windows では生じず，逆に MacOS では画面の書き換えが失敗してしまうため，**図４**のように，OS 毎に関数の定義を変えることによって，この問題に対処しています.

最近，韓国 NFRI（National Fusion Research Institute）の KSTAR システムで myView2 を利用できるように作業を行いましたが，汎用的に作成したはずの myView2 のソースコード中に核融合科学研究所内のサーバ名等が陽に書か

```
def CallNow(callableObj, *args, **kw):
     callableObj(*args, **kw)

if platform.system() == 'Linux':
     CallAfter = wx.CallAfter
else:
     CallAfter = CallNow
…
…

def UpdateGraph(self):
     #更新されたデータを描画する
…
….
     CallAfter(self.canvaspanel.update_real)
```

図４　OS によって描画処理を変える.

れている部分が多く見つかりました．LHD および KSTAR 用に別々にソースコードを持つことは，管理が複雑になるため，これらの依存部分はソースコードから分離し，設定ファイルから読み出すようにしました．設定ファイルは，空白を含むファイル名や構造を持ったデータを容易に扱え，かつ，視認性が良いことから，JSON 形式のファイルを採用しました（**図5**）．

自分の研究のために，Python を使った解析プログラムやツールを開発することは多いでしょうが，開発したプログラムを多くの人に使用してもらうためには，自分以外の多様な環境に対応するよう心掛けてほしいと思います．また，Python 自体は様々な環境で動作するプログラミング言語ですが，便利だからといって，インターネット上に公開されている様々な Python のパッケージを使用するプログラムを作ってしまうと，特定の環境下だけでしか動かなくなったり，依存するパッケージのサポート終了によって，保守ができなくなったりすることがあるので，注意が必要です．

```
{
    ”cachefilename”: ”cache.txt”,
    ”cachefoldername”: ”cache”,
    ”commandfoldername”: ”bin”,
    ”config_version”: ”1.1.1”,
    ”datamodulefilename”: ”datamodules.txt”,
    ”datamodulefoldername”: ”DataModules”,
    ”docpage”: ”http://kaiseki-web.lhd.nifs.ac.jp/software/myView2/myView2-Eng.pdf”,
    ”enabledmodulefilename”: ”enabled.txt”,
…
…
}
```

図5　設定ファイルの例．

講座 今日からはじめる Python

4. Python の活用

4. Application of Python

4.2　JT-60SA 実験における Python の利用

4.2 Application of Python in JT-60SA Experiment

浦 野　 創

URANO Hajime

量子科学技術研究開発機構那珂核融合研究所

（原稿受付：2018年 2 月26日）

JT-60SA では Python を利用した汎用データ解析ソフトウェア（eDAS）の開発を行っています．eDAS は JT-60SA 実験に関連する複数の異なる種類の大容量データを扱う他，将来的な機器や装置の改良に備え，プラグイン形式での柔軟な拡張性を有する設計の下で開発されています．研究所内外のユーザが利用するような汎用ソフトウェアの場合は，イベント管理を widget で構成された GUI として実装し，特にデータアクセスやグラフ描画をインタラクティブな操作で行えるように開発することが重要となります．

Keywords:

experiment analysis software, database, plugin, wrapper, graph, GUI, Python, JT-60SA

4.2.1　eDAS の概要

大型実験装置でのデータ解析ソフトウェアは，単に大容量データを扱うというだけでなく，複数の異なる種類のデータを扱いながらデータ解析を行う必要があるため，その目的に準じたユーザインターフェースと可視化機能，そして解析されたデータの出力機能が要求されます．量子科学技術研究開発機構が幅広いアプローチ（BA）協定の下で欧州と共同で設計・製作を進めている大型トカマク装置 JT-60SA[1]では，その汎用データ解析ソフトウェア（eDAS）を Python で開発しています．

解析対象となるデータには，実験室レベルと大型装置において冒頭で述べたような違いはある一方で，データベースやメモリ容量など，要求される解析に足るハードウェア側の受け皿が整備されていれば，Python を用いたデータ解析ツールとしてはソフトウェアの機能に大きな違いは生じません．しかし，その一方で解析ソフトウェアとして複数の異なる種類の大容量データを扱うということは，データ解析の目的は多様にわたることは容易に想像できます．従って，大型装置における汎用データ解析ソフトウェアは，十分な機能の拡張性を有する設計の下で開発される必要があります．

eDAS は，JT-60SA における汎用データ解析ソフトウェアの総称であり，放電波形描画ツール（eGIS），プラズマ平衡解析ツール（eSURF），プラズマ空間分布解析ツール（eSLICE）の 3 つのツールから成っています．**図 1** に JT-60SA の解析サーバ内における汎用データ解析ソフトウェア周りのデータフローを示します．JT-60SA ではデータの種類に応じた複数のデータベース（DB）を配置しており，eDAS の各ツールは解析の目的に応じて DB との入出力を行います．DB アクセスについては，C 言語や Fortran で記述された関数群をライブラリ化しており，Python で wrapper 処理したオブジェクトをユーザに提供しています．

上記の各 DB とのデータの入出力を行うためのオブジェクトの他，一部のグラフ描画用のオブジェクトなど，

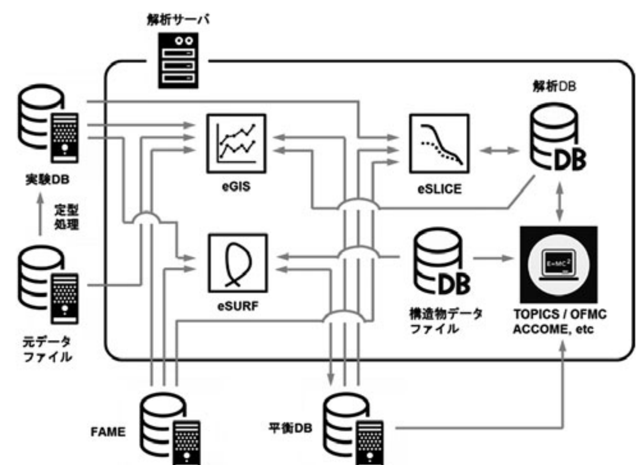


図 1　JT-60SA の解析サーバ内における汎用データ解析ソフトウェア周りのデータフロー．

National Institutes for Quantum and Radiological Science and Technology, Naka, IBARAKI 311-0196, Japan

authors's e-mail: urano.hajime@qst.go.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research

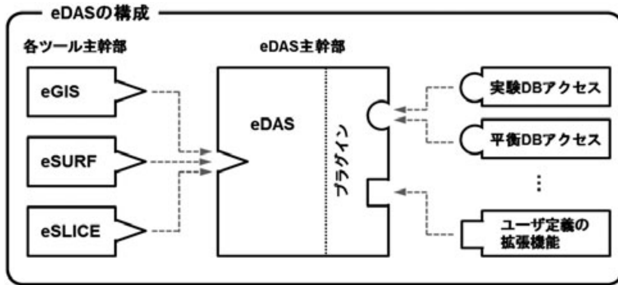


図2　eDASのソフトウェア構成．プラグインによる機能の拡張性を踏まえた設計となっている．

eDAS内の各ツールにはいくつかの共通のクラスが存在します．そのため，eDASは**図2**に示すようにソフトウェア全体に共通なオブジェクトで構成される主幹部と各ツールの主幹部が組み合わさって機能する設計になっています．これはソフトウェアの構成としては，Microsoft Officeの中のWordやExcel等の位置づけに似ています．DBアクセス用のオブジェクトはプラグイン形式で開発されているため，DBの仕様変更があった場合には，このオブジェクトをアップデートするだけで全てのツールが利用できます．プラグイン形式の最大の利点はソフトウェアの拡張性であり，将来的な機能拡張に加え，ユーザ定義のオブジェクトを容易に取り込むことができます．

eDASでは，Python標準のライブラリに加えて，いくつかの外部パッケージを利用しています．eGIS，eSURF，eSLICEの各ツールにおけるグラフ作成およびグラフのカスタマイズにmatplotlibを，eDAS内の配列演算にnumpy[2]を，MHD揺動のスペクトル解析などの科学計算処理のためにscipy[3]を用いています．また，**図1**に示す各DBとのデータの入出力に関しては，Python標準のライブラリであるctypes[4]を利用して，DBアクセスライブラリ本体にwrapper処理することでeDASに接続しています．

そして，汎用ソフトウェアではGUIを用いたインタラクティブな操作が求められるため，eDASでは全体のGUI構築とイベント管理のためにPySide[5]を導入しています．PySideはPythonでGUI開発を行うためのライブラリであり，特徴としてクロスプラットフォームに対応したアプリケーションの開発が行えることが挙げられます．ここでは非常にシンプルなGUIプログラムを紹介します．

```
from PySide import QtGui

class simpleBtnWidget(QtGui.QWidget):
        # PySide のQtGui.QWidget を継承したクラス

        def __init__(self, *args, **kwargs):

            super().__init__(*args, **kwargs)

            # レイアウトの設定
            self.setLayout(QtGui.QVBoxLayout())

            # ボタンwidget の作成
            self.this_btn = QtGui.QPushButton("
            Push", parent=self)

            # Widget の配置
            self.layout().addWidget(self.this_
            btn)
            # ボタンをクリック時のイベントをボタンへ接続する
            self.this_btn.clicked.connect(self.
            clicked_btn_action)

        # ボタンをクリックした際の動作を設定
        def clicked_btn_action(self):
            print("Hello World !")

if __name__ == "__main__":
        import sys

        # アプリケーションの作成
        app = QtGui.QApplication(sys.argv)

        # simpleBtnWidget クラスのインスタンスを作成
        btn_widget = simpleBtnWidget()

        # widget の表示
        btn_widget.show()

        # 実行
        sys.exit(app.exec_())
```

これはボタンのwidget（GUIを構成する部品要素）をウインドウ上に配置し，ボタンを押すと,"Hello World!"を出力するプログラムです．PySideのイベントの管理は，GUIで何らかのアクションが発生した場合，Signal（マウスがクリックされた等の信号）にconnect（接続）されたSlot（動作）が実行されます．上のサンプルコードでは，ボタンwidgetがクリックされた場合（clicked）に，connect先の動作（clicked_btn_actionメソッド）が実行されます．eDASは，あらゆるイベント管理をこのようなPySideベースのwidgetで構成されたGUIとして実装しており，特にDBアクセスやグラフ描画をインタラクティブな操作で行えるように開発しています．次節では，eDASの各ツールの紹介を述べていきますので，読者の皆さんには，Python

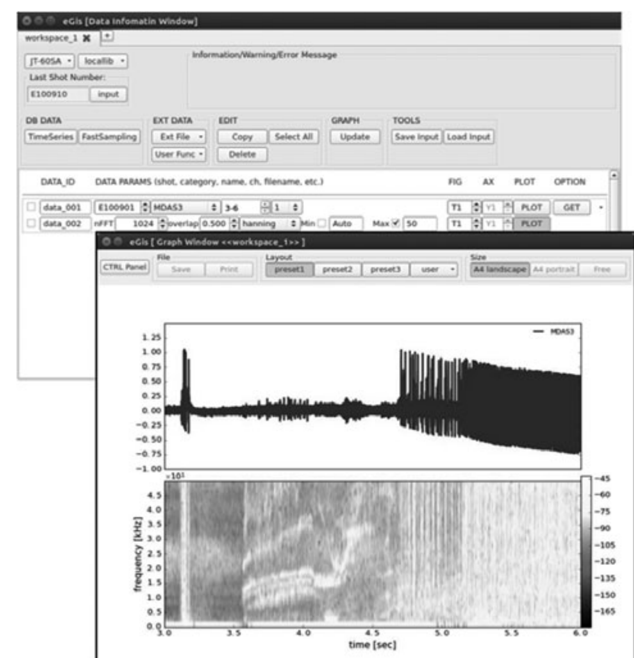


図3　放電波形描画ツール（eGIS）のユーザインターフェース．磁気プローブ信号からMHD揺動解析を行っている例．

でこのようなことができるという感触を掴んでいただければ幸いです．

4.2.2 eDAS の機能

図2に示した通り，eDASの各ツールはその主幹部を共有していますが，エンドユーザから見た場合には，eGIS，eSURF，eSLICEの3つのツールが独立に存在しているように見えます．紙面の都合上，機能の詳細を説明することはできませんが，ここでは各ツールの概要を順に述べたいと思います．

図3に示すのは，放電波形描画ツール（eGIS）のユーザインターフェースです．eGISは，解析するショット番号，データ名等をGUI上で指定し，DBからデータを取得してグラフ描画を行う基本的なツールです．単に取得したデータを表示するだけでなく，例えば複数の実験データからグリンワルド密度の時系列データを作成・プロットするなど，簡単な演算を行うこともできる設計になっています．**図3**では，MHD揺動解析のために磁気プローブ信号をスペクトル分析している様子を表しています．また，**図1**に示すように，eDASはデータの種類に応じた様々なDBにアクセス可能なため，実験DB上の計測データと輸送コード等で計算した解析DB上のシミュレーション結果をグラフで比較することもできる仕様になっています．

図4に示すのは，プラズマ平衡解析ツール（eSURF）のユーザインターフェースです．eSURFは，解析するショット番号，時刻等をGUI上で指定することで，プラズマの平衡データを作成し，その磁束分布を描画する他，既に平衡DBに格納された平衡データを呼び出すツールです．例えばレファレンスとなる磁束分布と実験結果を比較する他，平衡パラメータの数値テーブルを表示したり，各種の計測視野やNBIの軌道，ECHの共鳴面等を平衡データ上に描画する機能を持っています．また，JT-60SAでは，放電終了後に各放電の平衡計算を自動的に行い，平衡データの時系列データを逐次DBに格納する予定であり（**図1**のFAME），eSURFではこの時系列データを読み込み，アニメーションとして描画することもできる仕様になっています．

図5に示すのは，プラズマ空間分布解析ツール（eSLICE）のユーザインターフェースです．eSLICEは，解析するショット番号，時刻等をGUI上で指定することで，実験DBから呼び出した多チャンネル計測器のデータを，平衡DBから読み込んだプラズマの磁気面上にマッピング処理，またはアーベル変換によって，空間分布として描画するツールです．マッピングデータをフィッティングする際にも係数をシークバーで変動させる等，インタラクティブに最適な条件を調整できるようなGUIを導入しています．さらに，**図1**に示すように，eSLICEにはマッピング及びフィッティングされた空間分布データが解析DBに格納された後，各種の輸送コード用の入力ファイルを作成する機能を有しています．

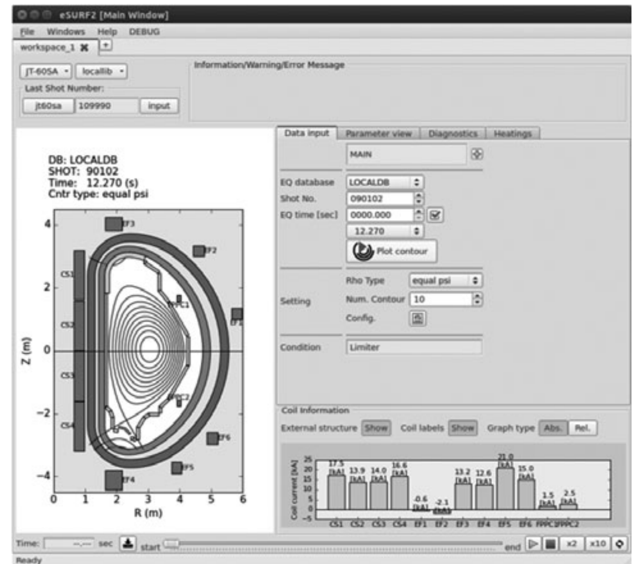


図4 プラズマ平衡解析ツール（eSURF）のユーザインターフェース．平衡DBとの入出力や計測視野及びNBIの軌道，ECHの共鳴面等を表示する．

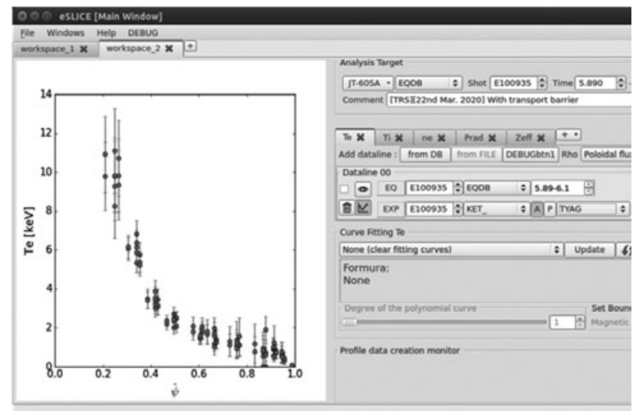


図5 プラズマ空間分布解析ツール（eSLICE）のユーザインターフェース．計測データを磁気面上にマッピング処理し，空間分布として描画する．

4.2.3 まとめ

本章ではJT-60SAにおいてPythonを利用した汎用データ解析ソフトウェア（eDAS）について述べました．eDASは汎用としての多様な機能を有する一方で，イベント管理された小さなコンポーネントの集合体に過ぎません．1つだけ注意しなければならないのは，データ解析の目的が多岐にわたるため，機能の拡張性を想定した設計になっている必要があるという点です．研究活動の進展に伴って，解析手法が変化する他，各種のDBや機器類，そして実験装置自体も改良されていくため，ソフトウェアはそれらの変化に対して柔軟に対応できる仕様が求められます．eDASはこのような機能拡張に関するオブジェクトをプラグイン形式で採用する設計になっています．

図1に示したeDASは基本的にJT-60SAの解析サーバ上での利用を想定した図になっていますが，Pythonは様々なOSとの互換性を持つため，リモートでの実験参加者にeDASをパッケージとして配布し，専用のDBアクセスモジュールを通してデータの入出力を行いながら，リモート

PC 側で eDAS を実行することも将来的にできるように設計しています．そのため，eDASの開発ではLGPLライセンスで使用することができるフリーのライブラリのみを利用しています．

また，研究所内外のユーザが利用する汎用ソフトウェアの場合は，イベント管理をwidgetで構成されたGUIとして実装し，特にデータアクセスやグラフ描画をインタラクティブな操作で行えるように開発することが重要です．特に**図1**に示すように複数のDBを利用するような環境であっても，シンプルなデータアクセスの構造をGUI上に仮想的に持ち込むことでユーザ側からは複雑なデータ構造を意識せずにデータ解析を行えるような工夫が重要となってきます．

浦野　創（うらの はじめ）

量子科学技術研究開発機構那珂核融合研究所上席研究員．工学博士．2002年北海道大学大学院工学研究科博士課程修了．独国Max Planck研究所，日本原子力研究開発機構を経て現職．専門は周辺プラズマ物理，閉じ込め物理等．2011年国際原子力機関よりNuclear Fusion Award受賞．2017年よりITPA周辺ペデスタル物理TG議長．最近は子どもたちにマジックを見せて笑顔にさせるのが楽しい．

最後に本章を執筆するにあたり，高度情報科学技術研究機構の若狭有光氏，並びにデジタルサーブ株式会社の阪本三姫氏の両者には，JT-60SA のための eDAS 開発及び DB 開発における技術支援スタッフとして協力していただきました．感謝いたします．

参 考 文 献

［1］H. Shirai *et al.*, Nucl. Fusion **57**, 102002 (2017).
［2］http://www.numpy.org
［3］https://www.scipy.org
［4］https://docs.python.org/2/library/ctypes.html
［5］https://wiki.qt.io/PySide/ja

講座 今日からはじめる Python

5．おわりに

5. Summary

吉沼幹朗
YOSHINUMA Mikirou
核融合科学研究所

（原稿受付：2018年２月26日）

ここでは，Python をさらにどの程度まで学べばよいかの指針を示します．また，本講座を通して理解していただきたいことを述べます．

5.1 学習の指針

この講座は，Python を始めてみようかと考えている人に向けて書かれました．さらに Python を学ぶ人のために，どこまで学んだらよいかを少し書きたいと思います．

本講座では，詳しい関数の定義（可変長引数，キーワード引数をもつ関数の定義）やオブジェクト指向プログラミングで用いられるクラス定義については説明しておりません．また環境に依存してくるため C 言語や Fortran で記述された自作の関数を，Python から呼び出す方法などについても説明しませんでした．簡単なデータ処理は，対話型環境に手続き的に（処理を並べるように）記述するだけで済みます．しかし，複雑な処理を見通しよく記述するなら，関数の定義について学ぶとよいでしょう．関数の定義までわかれば，我々が行うほとんどのことを Python で行えると思います．

関数の定義ができるようになったら，モジュールや名前空間について学んでください．すなわち，import 文の動作についてです．便利なモジュールを扱うときに，ストレスを感じにくくなるでしょう．自分で作成した関数をモジュールとして別ファイルにしてコレクションすることもできるようになります．Python では，クラスを定義して，オブジェクト指向のプログラミングもできますが，既存のモジュールで提供されるオブジェクトを利用できる程度に理解されていれば，実用上問題ないでしょう．多くの人が理解できるスクリプトという意味では，ここまでの知識で処理を記述することをお勧めします．

より汎用に利用できるモジュールを作成したり，既存のモジュールを調べたりしたいときは，オブジェクト指向やクラスの定義について詳細を学んでください．例外処理（エラーが起きた場合の対処），匿名関数（ラムダ式），関数アノテーションなどより高度な機能が使われることもあります．

これらについて多くのことは Web 上で検索することで解決すると思います．ひと通り学ぶなら，Python の公式ページのドキュメント（https://docs.python.org/ja/3/）を読むことをお勧めします．本で学ぶなら，『初めての Python 第３版』[1]には Python についての詳細が書かれています．Python についてだけでなく機械学習などの応用まで含めた形での入門書としては，『みんなの Python 第４版』[2]がよいでしょう．また，Numpy や pandas を用いたデータ分析について『Python によるデータ分析入門』[3]があります．Python やよく利用されるモジュールは日々更新されていますので，なるべく新しく出版された書籍を読むことをお勧めします．

5.2 おわりに

プログラミングといっても，大きく二つの立場があると思います．一つは，製品を生み出すためのプログラミング，もう一つは，思考の道具としてのプログラミングです．前者は，プログラミングの専門家が仕様に沿ってバグのないプログラムの作成をめざすものです．後者は，データの理解のために，試行錯誤しながら手続きを施し，結果を観察していく，道具としてのプログラミングです．プログラミングの専門家ではない実験家が行いたいのは後者のプログラミングでしょう．

実験データを観察しながら，解析手法を考え，思いついたアイディアを実行するためのスクリプトを書き，それを適用した結果をまた観察する．これを多くのデータに対して，繰り返し適用してみたりする．手動でデータを処理するには，少しめんどくさい量でも，スクリプトにしてしまうと，気軽に間違わずに何回でも処理できます．自分でスクリプトを書けば，どうしてそのような出力が得られたのかも考えられるでしょう．このような作業を行うことに，Python は適しています．処理の本質的でない部分に対するプログラミング（型の指定やメモリの確保など）にあまり気を取られずに記述できます．子ども向けの Python の解説書[4]も出ていることから，プログラミングの専門家ではない人でも容易に扱えるものと感じています．その作

National Institute for Fusion Science,Toki, GIFU 409-5292, Japan

author's email:yoshinuma@nifs.ac.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research

成された処理を誰でも実行でき，それについて議論することができます．

この講座を読んで，誰でもすぐに試すことのできるPythonというプログラミング言語環境があって，それは容易に学ぶことができるものであり，すでにプラズマ・核融合分野でも使っている人が多くいることを理解していただけたでしょうか．Pythonを用いて，皆さんの手元にある実験データを読み込み，いろいろな解析のアイディアを試していただくことを願っています．

最後になりますが，Pythonやそのモジュールを作成されている，各オープンソースコミュニティに感謝いたします．この講座を連載する機会を作り，とりまとめていただきました，九州大学稲垣滋教授に感謝申し上げます．

吉沼幹朗（よしぬま みきろう）

核融合科学研究所，ヘリカル研究部，助教．2000年，東北大学大学院工学研究科電気・通信工学専攻博士課程修了．LHDにおいて中性粒子ビームを用いたプラズマ計測（荷電交換分光，モーショナルシュタルク効果分光）を行っています．特にプラズマ中の流れや電場構造の形成に関心をもって研究を行っています．LHDの実験シーケンスに同期させた解析プログラムの実行や解析データの可視化にPythonを利用しています．

参考文献

［1］M. Lutz（著），夏目 大（翻訳）：初めてのPythonへ第3版（オライリージャパン，2009）．

［2］柴田 淳：みんなのPythonへ第4版（SBクリエイティブ，2016）．

［3］W. McKinney（著），小林儀匡（翻訳），鈴木宏尚（翻訳），瀬戸山雅人（翻訳），滝口開資（翻訳），野上大介（翻訳）：Python によるデータ分析入門 - NumPy,pandasを使ったデータ処理（オライリージャパン，2003）．

［4］J.R. Briggs（著），磯蘭水（翻訳），藤永奈保子（翻訳），鈴木悠（翻訳）：たのしいプログラミングPythonではじめよう！（オーム社，2014）．