

自主学習資料：Python プログラミングの基礎

この資料は、気象データ分析チャレンジ！を受講するうえで必要なPythonの知識と、講義で使用するPythonの実行環境(Jupyter Notebook / Jupyter Lab)の最低限の使用法を学習するためのものです。これらについてすでに知識をお持ちの方は学習する必要はありません。

目次

- **1 Jupyter Notebook / Jupyter Lab について**
 - 1.1 Jupyter の起動
 - 1.2 ファイルのオープン
 - 1.3 Notebookの編集・プログラムの実行
 - 1.4 Jupyter の終了
- **2 Python の基礎**
 - 2.1 プログラムの主な構成要素
 - 2.1.1 コメント
 - 2.1.2 文字列
 - 2.1.3 数値
 - 2.1.4 算術演算子
 - 2.1.5 オブジェクト
 - 2.1.6 変数と代入文
 - 2.1.7 比較演算子
 - 2.2 Pythonの特徴的なイテラブルオブジェクト
 - 2.2.1 リスト
 - 2.2.2 タプル
 - 2.2.3 辞書
 - 2.2.4 文字列
 - 2.3 Python で使われる基本的な構文
 - 2.3.1 for文による繰り返し
 - 2.3.2 While文による繰り返し
 - 2.3.3 リスト内包表記 #r2.3.3
 - 2.3.4 条件分岐
 - 2.4 関数
 - 2.5 ライブラリとインポート
 - 2.6 多次元データ計算モジュールNumPy
 - 2.6.1 数値配列オブジェクト ndarray
 - 2.6.2 3次元の ndarray
 - 2.6.3 ndarray の要素の合計等
 - 2.6.4 無効値 nan
 - 2.7 表型データ処理モジュール Pandas
 - 2.7.1 オブジェクト DataFrame
 - 2.7.2 レコードの抽出

- 2.7.3 行や列の再編成
- 2.7.4 DataFrameの結合
- 2.7.5 集計
- 2.7.6 CSVファイルの書き出し/読み込み
- 2.8 時間処理モジュール datetime
 - 2.8.1 日付時刻オブジェクト datetime
 - 2.8.2 時間間隔オブジェクト timedelta
- 2.9 データ可視化モジュール Matplotlib, Seaborn
- **あとがき**
- **著作権について**

Copyright (c) 気象データ×IT勉強会 2022 All rights reserved.

1 Jupyter Notebook / Jupyter Lab について

Jupyter Notebook は、ブラウザ上でPythonプログラムを作成したり実行したりできる上、テキストも書き込める便利なソフトウェアです。そして、**Jupyter Lab** はその進化形として位置づけられ、複数画面表示や画面間でのセルのコピーなど機能が強化されています。両者の基本的な操作は同じで、作成したファイルも両者で共通に使用できるので、すでにどちらかがあれば新規インストールは不要です。新規で使い始める方は、ネットに評価記事がありますので、それを参考に、お好みの製品をインストールしてください。以降、両者を**Jupyter** と総称します。

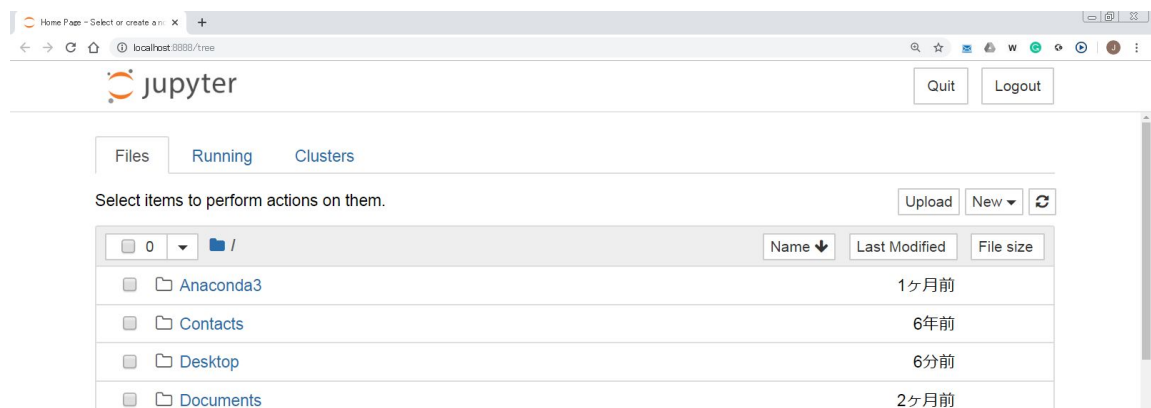


図1: Jupyter Notebookのホーム画面

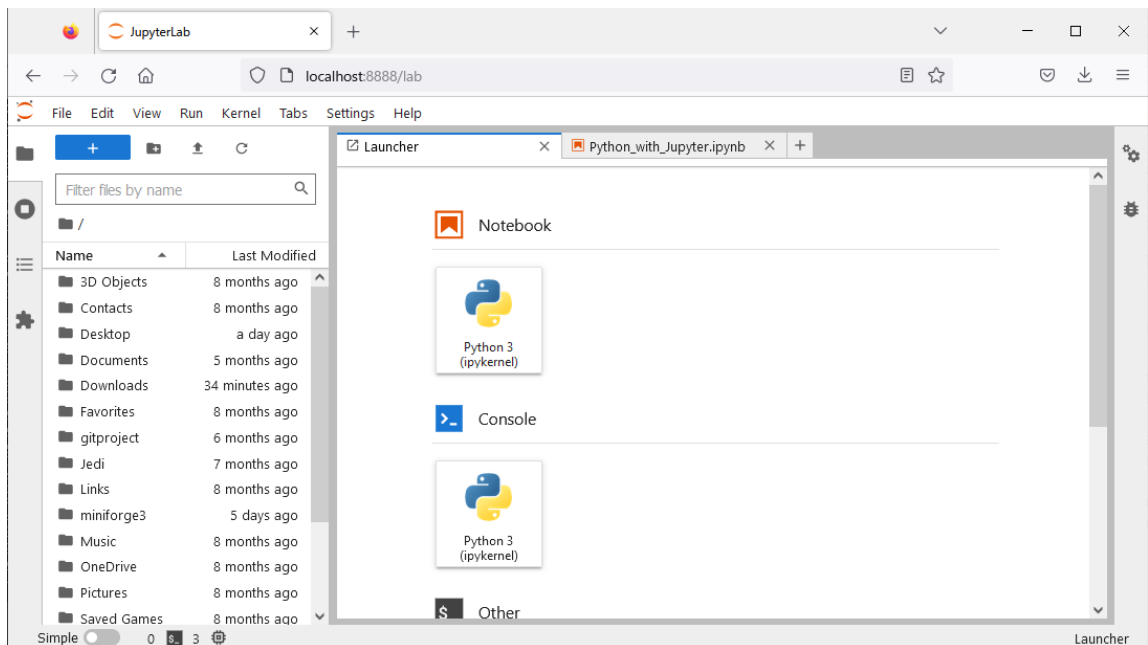


図2: Jupyter Labのホーム画面

1.1 Jupyter の起動

macOSの場合はターミナル、Windowsの場合はAnaconda Prompt等（Miniconda Prompt, Miniforge Prompt, ...）を開き、ここに、「**jupyter notebook**」との打ち込んでエンターキーを押します。

すると、いつもお使いの Webブラウザが起動し、その中で、図 1 のようにJupyter Notebookが開きます。Jupyter Labをお使いの場合は、「**jupyter lab**」と打ち込んでエンターキーを押します。すると図 2 のようにJupyter Labが開始されます。

jupyterは、起動させたプロンプトのカレントフォルダの場所でオープンし、それから下の階層にあるファイルにのみアクセスできます。

- **参考**：外付けHDD等に置かれているNotebookを開くには（今、仮にこのHDDのドライブレターを「H:」とします）、プロンプトを起動したら、まず、「**H:**」と打ち込みカレントフォルダをHDDに移動したうえで「**jupyter ○○○**」を打ち込みJupyterを起動してください。

1.2 ファイルのオープン

今回皆さんに配布するテキスト式のフォルダは、PCのデスクトップにおいてください。フォルダ中、拡張子が「**.ipynb**」となっているファイルが、Jupyter のファイル（以降、Notebook）です。

Jupyterの起動直後は、画面左側のフォルダリストにDesktop や Documents、Downloads などが表示されるはずです。この中から「Desktop」を見つけてダブルクリックし、さらに、テキスト式のフォルダと順に進み「Python_with_Jupyter.ipynb」を開いてください。

新規にNotebookを作成する場合は、それぞれ下の図のように操作します。

- **Jupyter Notebook** の場合 :

画面右上の「New」をクリックし、さらに「Python3」をクリックします。すると、下図のように、セルが一つだけの新規 Notebookが作成されます。

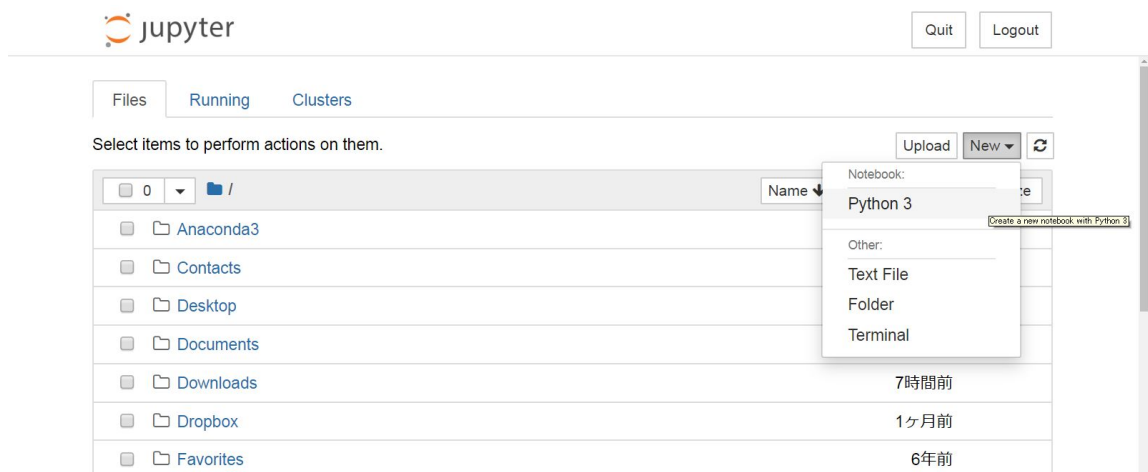


図2: 画面右上の「New」 > 「Python3」とクリック

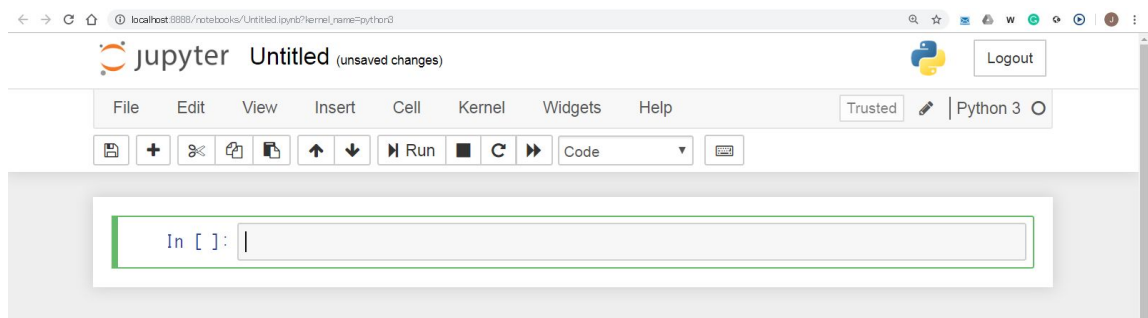


図3: 新規Notebook

- **Jupyter Lab** の場合 :

画面上部のJupyterのメニューバーから、「File」>「New」>「Notebook」と選択し、「python3 (ipykernel)」と表示されていることを確認し「Select」をクリックします。

1.3 Notebookの編集・プログラムの実行

Notebookは、セル(Cell)と呼ばれる単位で構成されています。Cellには、マークダウン(Markdown)とコード(Code)の二種類があり、前者は文章を書き込むために使用し、後者はプログラムを書くのに使用します。セルをダブルクリックすると枠線が表示され中身が編集可能となります。**Ctrl+Enter**を押すと、Markdownの場合は編集した内容が確定します。Codeの場合は、その部分のプログラムが実行されます。

プログラム実行後、各変数の値はそのまま保持されているので、その下のセルを実行すれば、プログラムを継続することができます。この仕様は、設定→データ処理→グラフ描画のような、一本道のプログラムにおいては大変便利です。例えば、グラフの体裁を変えたいときは、グラフ描画のセルだけを何度も手直し/実行すればよいからです。

とはいえ、手前までのセルで作られたデータ自体に変更が加えられるような処理の場合

は、関連する手前のセルも再実行する必要があります。このようなときは、メニューバーからRun> Run All Bove Slected Cell を選択すると、一番初めから現在のCellまでを一気に実行してくれます(Jupyter Lab の場合)。

ただし、このようにしていると、知らず知らずのうちに古い変数が残っていてそれが別なところで動作に影響を与えたりすることが起きるので、プログラミングが一区切りしたら、カーネルを再起動して動作確認をすることをお勧めします。カーネルの再起動はツールバーの丸矢印のボタンで行います。

新たなCellを追加するときには、ツールバーの左上の「+」ボタンをクリックします。

1.4 Jupyter の終了

Jupyter Notebook / Jupyter Lab は普通のアプリケーションソフトとは少し違う仕組みで動いています。お使いのPCの中にホームページサーバーが一時的に構築され、そこにブラウザがアクセスしているのです。このため、単にブラウザを閉じただけでは、ホームページサーバーが内部的に動き続けることになります。 正しく終了するには、データを保存した後、（ブラウザではなく）JupyterのメニューからFile> Shut downを選択して終了してください。すると、ブラウザが終了し、起動時に使用したコマンドプロンプトだけが残るので、これに「exit」と入力して終了します。

[目次に戻る](#)

2 Python の基礎

Jupyter の最低限の操作方法がわかったところで、Pythonの基本を学習します。Jupyter Lab を使用している方は、メニューバーの「View」から、「Show Line Numbers」をクリックしてオンにし、Codeセルに行番号を表示させてください。

2.1 プログラムの主な構成要素

2.1.1 コメント

コメントは、コードを読んでいる人に対するメッセージで、pythonがプログラムとして認識しないものです。一般に、プログラムは人と計算機との間の文書とされていますが、実際はそれと同じぐらに、人と人との文書でもあります。その意味で、コメントはプログラミング言語の構成要素で最も重要なものと言えます。Pythonではハッシュ（#）に続く部分はコメントとみなされ、コードとして解釈されません。行の途中にハッシュをおいた場合はハッシュから後ろがコメントとなります。

```
In [1]: # ハッシュ以降に書かれているものは全て無視されます
print('123') # 文の後に置くことも可能です
```

2.1.2 文字列

処理結果を文字で表現したり、与えられた文字に基づいて異なる作業をさせたりするために、Python は文字列を取り扱います。文字列はシングルクォテーション（'）または、ダブルクォテーション（"）で囲んで示します。

```
In [2]: print('WXBC気象データ分析チャレンジ！')
        print("WXBC気象データ分析チャレンジ！")
```

```
WXBC気象データ分析チャレンジ！
WXBC気象データ分析チャレンジ！
```

これらを使い分けると、クォテーションマーク自体を表示することができます。

```
In [3]: print('WXBC"気象データ"分析チャレンジ！')
        print("It's cool!")
```

```
WXBC"気象データ"分析チャレンジ！
It's cool!
```

2.1.3 数値

CやFORTRANなどのプログラミング言語では、数値を扱うときに整数なのか文字なのかを「型」として設定し、これから利用するという宣言をする必要があります。しかし、pythonには変数の型宣言が基本的に必要ないので、変数を使いたいときに値を代入するだけで使えます。

しかし、内部的には、整数型と浮動小数点型、その他沢山の型があります。通常は気にせず使えますが、方に注意を払わなければならない時もあります。値がどのような型なのかを調べるには関数 **type** を用います。

```
In [4]: print( type(1), type(1.), type("1") )

<class 'int'> <class 'float'> <class 'str'>
```

2.1.4 算術演算子

Python では、いわゆる四則演算やべき乗、剰余など、数値に対する演算を表に示す演算子で行うことができます。整数型と浮動小数点型を混ぜて演算してもエラーにはならずそれなりの結果を出しますが、結果の数値型は演算子の種類と演算する二つの数の型に依存します。

演算	記号	整数型同士	浮動小数点型同士	混合使用
加算	+	整数型	浮動小数点型	浮動小数点型
減算	-	整数型	浮動小数点型	浮動小数点型
乗算	*	整数型	浮動小数点型	浮動小数点型
除算	/	浮動小数点型	浮動小数点型	浮動小数点型
切捨て除算	//	整数型	浮動小数点型	浮動小数点型

演算	記号	整数型同士	浮動小数点型同士	混合使用
剰余	%	整数型	浮動小数点型	浮動小数点型
べき乗	**	整数型	浮動小数点型	浮動小数点型

上記は数値に対する演算ですが、演算子「+」と「*」については、文字列に対しても定義されています。

```
In [5]: print("文字列を" + "連結します。")
```

文字列を連結します。

```
In [6]: print("-" * 20 + " きりとり " + "-" * 20)
```

----- きりとり -----

2.1.5 オブジェクト

Pythonでは、文字と数字のほかに、オブジェクトと呼ばれるものを多用します。オブジェクトとは、「複数の情報をひとまとめにしたもの」と捉えることができます。例えば、気温は数値で表現できますが、実用上は、単位は何か、それはいつの値か、どここの値か、などの情報があって始めて意味を持つことが多いです。このような例は数多くあるので、プログラミングにおいても、このようなものをひとまとめの固まりとして取り扱うようになりました。これがオブジェクトです。オブジェクトを利用すると、複雑なプログラムを分かりやすく表現でき、開発時間やミスを削減することもできます。

Pythonで使われている定番のオブジェクトに、日付や時刻を表現するdatetimeオブジェクトがあります。これを例に、オブジェクトへの理解をもう少し深めましょう。時刻とは、ある起点からの経過を決められた時間間隔を単位として数値化したものです。つまり時刻は二つの数値を一塊として定義すればOKです。しかし、これだけでは少しも便利ではありません。オブジェクトは、それならではの操作も含めて作って定義します。datetimeオブジェクトで言えば、二つの時刻の間隔の計算や、特定の時刻を「〇〇年〇〇月〇〇日」のような文字列にする機能や、その逆の機能、タイムゾーンやサマータイムに応じた表現を求める機能などです。

オブジェクトとは、より正確には、**それら进行处理することで作り出す便利な機能のコレクションが用意されている情報の集合**と言えます。そして、この"便利な機能"のことをメソッドと呼びます。

オブジェクトのメソッドを使うには、オブジェクトの変数名とメソッド名をピリオド「.」で繋がめます。

オブジェクト.メソッド名(引数, ...)

2.1.6 変数と代入文

数学の x や a のように、Python でも特定の役割をする数や文字列、オブジェクト等に名前を付けることができます。これを変数と呼びます。変数名にはアルファベットの大文字小文字(区別されます)、数字、アンダースコア(_)が使えます。ただし、最初の1文字だけは数字が使えません。

数学で「 $x = 3$ とする。」のように、変数に特定の値を与えることをプログラミングでも代入とよび、Python では等号 (=) を一つ使います。

```
In [7]: x = 3
        print(x)
```

3

Pythonでは、以下のようにして一つの等号で複数の変数に値を代入することもできます。この例では lat、lon、site_name にそれぞれ36.0270、140.1104、"Tsukuba"が代入されます。

```
lat, lon, site_name = 36.0270, 140.1104, "Tsukuba"
```

「 $x = x + 3$ 」という数式はプログラミングが初めての人にとっては違和感がある書き方かもしれませんが、Python では、「今 x に格納されている数に3を加えたものを新たに x とする」という文として正しく取り扱われます。

```
In [8]: x = x + 3
        print(x)
```

6

Python では、「 $a = a + 10$ 」と同じ操作を次の文でも実行できます。このような記法を累積代入文と呼びます。

```
In [9]: a = 100
        a += 10
        print(a)
```

110

2.1.7 比較演算子

Python には、**True** と **False** という特別な値（あらかじめ組み込まれた定数）があります。これらはそれぞれ真（正しい/成り立つ）と偽（誤り/成り立たない）を意味し、条件によって処理を変更するときなどに使用します。比較演算子は、二つの変数を比較して両者の関係に従って、**True** または **False** を返します。Python で用いられる比較演算子を表に示します。

比較演算子	比較条件
$x < y$	x は y より小さい
$x \leq y$	x は y より小さい、もしくは x と y は等しい
$x > y$	x は y より大きい
$x \geq y$	x は y より大きい、もしくは x と y は等しい
$x == y$	x と y は等しい（等価である）
$x != y$	x と y は等しくない（等価でない）
$x \text{ is } y$	x と y は同じオブジェクトである
$x \text{ is not } y$	x と y は同じオブジェクトではない

比較演算子 比較条件

<code>x in y</code>	x はy に含まれる (配列のように複数の要素からなるオブジェクトyに対して)
---------------------	--

<code>x not in y</code>	x はy に含まれない
-------------------------	-------------

以下のCellを適宜編集し、比較演算子を使った式（比較式）を実行し、挙動を確認してください。

```
In [10]: 2 > 2
```

```
Out[10]: False
```

なお、Python では比較演算子を以下のように連結して使うこともできます。

`x < y < z`

連結した場合は、それぞれの比較の論理積が最終的な比較結果になります。つまり、以下のようにしたのと同じです。

`x < y and y < z`

[目次に戻る](#)

2.2 Pythonの特徴的なイテラブルオブジェクト

イテラブルオブジェクトとは、複数の数値や文字列などを並べたものです。Pythonには、**リスト**や**タプル**、**辞書**などの特徴的なイテラブルオブジェクトがあり、プログラムの中で頻繁に利用されます。

2.2.1 リスト

リストは、複数の要素をまとめて扱う際に最もよく使われるイテラブルオブジェクトで、**[要素, 要素, ...]** のように角括弧の中に要素を半角カンマで区切って格納したものです。他のプログラム言語で言うところの配列のようなものですが、リストはきわめて融通無碍で、様々なものを要素にとることができます。また、異なる種類のものを同居させることもできます。以下はいずれも正しいリストです。

<code>[1, 2, 3]</code>	(整数のリスト)
<code>[]</code>	(空のリスト)
<code>["Mesh", "Data"]</code>	(文字列のリスト)
<code>[[1.0, 2.0, 3.0], [2.1, 3.1, 4.1]]</code>	(リストのリスト)
<code>[10, 20, "Mesh", [100, 200, 300]]</code>	(様々な型の要素のリスト)

数値と同様、リストは変数に代入することができます。

演算子+をリストに適用するとリストを連結します。また、リストに算術演算子*と整数を用いるとリストが整数個連結されたものが得られます。

```
In [11]: lst1 = [1, 2, 3]
lst2 = [4, 5]
lst3 = lst1 + lst2
print(lst3)
print(lst3*3)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

リストに連なる個々の要素の場所のことをインデックスと言います。リストから特定の要素を取り出すには、インデックスを角括弧で括ってリストの後ろに付けます。この際、最初の要素を0として数えます。

```
In [12]: lst = ["零", "一", "二", "三", "四", "五", "六", "七"]
lst[0]
```

```
Out[12]: '零'
```

リストから複数の要素をまとめて取り出すこともできます。人間の数え方で3番目から5番目まで、すなわち、0始まりでは2番目から4番目である「二」～「四」を纏めて取り出すには `[2:5]` と指定します。

```
In [13]: lst[2:5]
```

```
Out[13]: ['二', '三', '四']
```

範囲の最後の数として指定した（0始まりの）数より1つ少ない範囲が切り出されることに注意してください。また、特定の要素を取り出した時は要素が裸で（一つ前のcellの例では文字として）抜き出され、範囲で取り出すとリストで抜き出されることにも注意しましょう。

- 上のセルを見て「あれ」と思われた方がいるかもしれません。これまで、変数の値などを表示させるのに関数 **print** を使っていましたが、上の例ではそれが無いにもかかわらず、内容が表示されました。Pythonは、プログラムが変数や式で終わっているとき、その内容、あるいは式の意味するところを表示します。これを利用すると、特定の変数が今どのような値を持っているかを簡便に確認することができ大変便利です。なお、表示するのは最後の一つだけです。

インデックスを用いてリストの要素を指定することをインデクシングと呼びます。インデクシングの記法は上記のほかにも様々あり、下記はいずれも有効な記法です。それぞれについて要素がどのように指定されるか、各自確認してみてください。

```
lst[:] lst[3:] lst[:4] lst[-2:] lst[:-3]
```

```
In [14]: lst[:]
```

```
Out[14]: ['零', '一', '二', '三', '四', '五', '六', '七']
```

リストの要素にリストが含まれているとき、含まれているリストの特定の要素を取り出すには、角括弧を連ねて指定します。下に示すリストが代入されている変数 **mixed** から、

「200」を取り出す場合考えましょう。

```
mixed = [10, 20, "Mesh", [100, 200, 300]]
```

mixed の要素であるリスト[100, 200, 300]の、mixed における場所はPythonの数え方では3番目、200 は要素であるリスト[100, 200, 300]のPython の数え方で1 番目なので、これらを用いて `mixed[3][1]` と表記することにより要素を取り出すことができます。

このほかに、**スライス**と呼ばれるインデクシングが可能です。スライスは2つのコロンで整数をはさんで、`[n:m:s]` のように記述します。このようにすると、**n 番目から m-1 番目の要素を、s 個間隔で指定する**という指示になります。インデックス番号の n や m は、0 から始まることに注意してください。n や m を省略したときは「すべて」という意味になります。また s が負のときは先頭からではなく末尾から指定することを意味します。

このため、`[::-1]`は、「末尾から1つずつ」という指示になり、リストの並び順を逆にするときなどに利用できます。よくこんな記法を考えたものです。

```
In [15]: lst[::-1]
```

```
Out[15]: ['七', '六', '五', '四', '三', '二', '一', '零']
```

代入文を使うと、インデクシングしたリストの要素を書き換えることができます。下の例では、整数だった要素を文字列に書き換えています。

```
In [16]: lst = [1, 2, 3, 4]
lst[-1] = "チャレンジ!"
lst
```

```
Out[16]: [1, 2, 3, 'チャレンジ!']
```

リストはオブジェクトなので、メソッド（=そのオブジェクト向けに特に用意されている機能/関数）もあります。結構たくさんありますが、よく使うものを3つだけ下表に示します。

メソッド	機能	例
append	最後尾に要素を一つ追加	lst = [1, 2, 3] lst.append("後はたくさん") →[1, 2, 3, '後はたくさん']
index	指定した値の最初に一致する要素の場所を知らせる (該当がないときはエラーとなる)	lst = ["a","b","c","d"] lst.index("d") →3
sort	配列を昇順に並び替える	lst = [10, 1, 7, 2] lst.sort() →[1, 2, 7, 10]

2.2.2 タプル

上で示した通り、リストは極めて融通無碍ですが、それゆえに、時にヒューマンエラーも引き起こします。そこで、Python には、リストに似ているけど融通が利かないタプルというデータ形式が用意されています。タプルは一度定義すると要素の中身や数の変更ができません。例えば、あるメッシュデータがあったとして、その縦横の要素数は、プログラム

の途中で変更すべきべきものではないので、リストではなくタプルで定義する方が望ましいといえます。タプルは、角括弧ではなく普通の丸括弧を用いて要素を括ります。

```
tsukuba = (36.0270, 140.1104)
```

タプルはリストと同じインデクシングができ、内容を参照することができますが、要素の変更・増減・並べ替えなどはできません。

2.2.3 辞書

Pythonでは、`[]` で括るリスト、`()` で括るタプルに加え、`{}` でくくる辞書というデータ型があります。これは、インデクシングを番号でなく文字列でできるようにしたものです。

「札幌が気温20℃、相対湿度40%」「仙台が気温25℃、相対湿度45%」「東京が気温35℃、相対湿度50%」、・・・というようなデータを取り扱うことを考えてみましょう。このようなデータをリストで管理するとすれば、以下のような形がよさそうです。

```
In [17]: data = [ ['札幌', [20, 40]],
                  ['仙台', [25, 45]],
                  ['東京', [35, 50]],
                  ['名古屋', [40, 30]],
                  ['大阪', [35, 45]],
                  ['福岡', [30, 45]],
                  ['沖縄', [25, 60]] ]

city = 3
place = data[city][0]
humidity = data[city][1][1]
print(place, humidity)
```

名古屋 30

しかし、このような管理では、それぞれの気象台がリストの何番目にあるかを把握していなければ使えません。このようなときに辞書は便利です。辞書を使って次のようにしてみましょう。

```
In [18]: dic_data = {'札幌': [20, 40],
                     '仙台': [25, 45],
                     '東京': [35, 50],
                     '名古屋': [40, 30],
                     '大阪': [35, 45],
                     '福岡': [30, 45],
                     '沖縄': [25, 60]}

city = '名古屋'
print(city, dic_data[city][1])
```

名古屋 30

同じ結果が得られました。辞書にすると、要素をインデックスではなく値で参照することができるので、並び順を気にすることなくデータを取り扱うことができます。

要素を指定するために使用する値のことをキーと呼びます。辞書にどのようなキーが使われているかを知るときには、辞書のメソッド **keys** を用いて以下のようにします。

```
In [19]: dic_data.keys()
```

```
Out[19]: dict_keys(['札幌', '仙台', '東京', '名古屋', '大阪', '福岡', '沖縄'])
```

2.2.4 文字列

文字列は、文字を要素とする特別なイテラブルです。なので上で説明したインデクシングを使うことができます。

以下に、文字列に用意されたメソッドの中から、よく使用するメソッドを2つ示します。

メソッド	機能	例
replace	文字列の一部を置換する	<pre>s = "abcdefg" s.replace("cd","1234") →"ab1234defg"</pre>
split	指定した文字を区切り記号と解釈しリストに分割する	<pre>s = "1.5,2.3,1.7" s.split(",") →["1.5", "2.3", "1.7"]</pre>

[目次に戻る](#)

2.3 Python で使われる基本的な構文

2.3.1 for文による繰り返し

for 文は、繰り返し処理のほとんどのケースで使われています。forの構文は、以下のように入ります。

for 変数 **in** イテラブルオブジェクト :
処理

ここで、**イテラブルオブジェクト**とは、先に学習したリストなど、要素を順番に取り出せるものです。for 文は、このイテラブルオブジェクトから要素を順次取り出してそれを「変数」に代入し、「処理」を繰り返します。これにより、イテラブルオブジェクトに並べられた要素に基づいて「処理」が繰り返されます。処理をする文はインデントします。

下の例で理解を深めてみましょう。この例は、先に、2.2.3 辞書 の学習で使用した **dic_data** を使用しています。

```
In [20]: cities = dic_data.keys()
for city in cities :
    humidity = dic_data[city][1]
    print(city, humidity, "%")
print("-----")

札幌 40 %
仙台 45 %
東京 50 %
名古屋 30 %
大阪 45 %
福岡 45 %
沖縄 60 %
-----
```

順に見てゆきましょう。

- ・ 1行目：この辞書に収録されている地点をメソッド **keys** で取り出して変数 **cities** に代入します。
- ・ 2行目：for文で、地点名をひとつずつ変数 **city** に代入し3行目と4行目の実行を指示します。
- ・ 3行目：辞書から対応する都市の湿度を取り出して変数 **huidity** に代入します。
- ・ 4行目：都市名と湿度値を画面表示します。

辞書に収録されているすべての都市に対して処理を実行したら、for構文は終了し5行目に移ります。

この例では3行目と4行目が他の行より字下げしてあることに注意してください。この字下げによって、どこからどこまでが繰り返し対象かを示しています。慣習として、Pythonでは半角スペース4つでインデントを構成します。また、for文の行末にはコロンを付けること忘れないでください。

- Pythonではインデントでコードブロックを表現するので、見やすさのために勝手にインデントしてはなりません。しかし、その一方で、Pythonは括弧と対応する括弧の間では自由に改行とインデントができます。2.2.3辞書 のcellにある `dic_data` の定義で、見やすさのために改行やインデントが使われているのは、そこが `{ }` の中のだからです。

リストの要素でループを回せるのはPythonの大きな特徴の一つですが、複数の配列の対応する要素を組み合わせるような場合は、やはり、0, 1, 2, 3,...という整数でループを回すことになります。これをいちいち[0, 1, 2, 3,...]というリストで与えては大変です。このため、整数列を作る **range** という組み込み関数が用意されています。関数`range`は引数を1つまたは3つとります。1つの場合、関数は0から引数の1つ手前までの連続した整数を作ります。これを利用して、以下のようにfor構文を作ることができます。

```
In [21]: for i in range(4):  
         print(i)  
         print(i,"までで終了")
```

```
0  
1  
2  
3  
3 までで終了
```

引数を3つ与えた場合は、3つの引数を順に **i, j, k** として、**i**からスタートして **j-1**までの間を、**k**毎に選んだ数列を与えます。以下のcellを実行して確認してください。

```
In [22]: for i in range(1,10,3):  
         print(i)  
         print(i,"までで終了")
```

```
1  
4  
7  
7 までで終了
```

for文では、時として、下の例のように、関数 **enumerate** が使われることがあります。この関数を用いると、イテラブルに加え各要素の番号も得ることができます。関数 **enumerate**

はこれら2つの値を出力するので、**i** と **city** で受け取ります。

```
In [23]: cities = dic_data.keys()

for i, city in enumerate(cities) :
    humidity = dic_data[city][1]
    print(i+1, city, humidity,"%")
print("-----")
```

```
1 札幌 40 %
2 仙台 45 %
3 東京 50 %
4 名古屋 30 %
5 大阪 45 %
6 福岡 45 %
7 沖縄 60 %
-----
```

2.3.2 While文による繰り返し

繰り返し処理をするには、for文以外に **while 文** があります。while文は、**条件が成り立っている間は、ずっと繰り返し処理する**のが特徴です。

whileの構文は、以下のように書きます。

```
while 条件式 :
    処理
```

条件式は、実行の結果 True であるか False であるかが定まる式で、2.5 で示した比較演算子を使った式が良く使われます。for構文と同じく、処理対応する部分がどこまでかは、インデントで示します。

while文を使った繰り返し処理は、きちんとプログラミングしないと、何回繰り返しても条件式が **True** にならなくなってしまうことがあります。この状態は「**無限ループ**」と呼ばれます。無限ループに陥ってJupyterが応答しなくなってしまうたら、ツールバーの ■ ボタンをクリックしてカーネルを停止してください。

2.3.3 リスト内包表記

繰り返し計算が何かのリストを作ることを目的にしている場合、for 文をリストの括弧で包んでしまい一行で終わりにしてしまう方法が、Python には用意されています。このような記述の方法を、リスト内包表記と呼びます。

リスト内包は以下の形式で書きます。

```
[式 for 変数名 in イテラブルオブジェクト]
```

ここで、「式」はイテラブルオブジェクトから目的とする配列要素を計算する式です。

例えば、5 の倍数に 1 を加えた数を要素とする、[1, 6, 11, 16, 21] というリスト **x** を作る場合、リスト内包表記では次のように記述します。

```
x = [i*5+1 for i in range(5)]
```

リスト内包表記は、for 文よりも高速で、また、リストを作っているのだということが一見してわかるので、慣れると便利です。

2.3.4 条件分岐

計算結果に基づいて次の処理を切り替えることを条件分岐と言います。Pythonでは、条件分岐はif, elif, else という構文を使って以下のように書きます。

```
if 条件式1:
    処理1 # 条件式1 が真(True)の場合の処理
elif 条件式2:
    処理2 # 条件式1 が偽(False)で条件式2 が真(True)の場合の処理
else:
    処理3 # 条件式1 も条件式2 も偽(False)の場合の処理
```

条件式 1 や条件式 2 は、実行の結果 **True** であるか **False** であるかが定まる式で、2.5 で示した比較演算子を使った式が良く使われます。for構文と同じく、処理 1 ～ 3 に対応する部分がどこまでかは、インデントを付けるか付けないかで示します。

value の値を様々に変えて下記コードを実行してみましょう。

```
In [24]: value = 1

if value <= 3:
    print(value, "じゃすくないな。")
elif 3 < value <= 5:
    print(value, "だしまあまあか。")
else:
    print(value, "だなんてラッキー！")
```

1 じゃすくないな。

if や else などの行末にはコロンを忘れずにつけてください。必要がなければ、elif や else は使わなくても構いません。また、elif は複数回を使うことができます。If文の中にさらにif文などを作る場合があります。その時は、その部分はさらに一段深いインデントで記述します。

[目次に戻る](#)

2.4 関数

気温変化のグラフを複数の地点について作成する場合、グラフを描くプログラムをひとまとめにして作っておき、必要となったときにそれを呼び出して使えたら便利です。このように、プログラムのいくつかの処理をひとまとめにして繰り返し利用できるようなしたものを関数と呼びます。数学のいう関数は、数を与えて数を決めるものですが、プログラミングにおける関数はより広い概念であり、「機能」に近いものです。Python では、関数を以下の構文で定義します。**def** の行の最後にコロンを付けることに注意してください。

```
def 関数名(引数, ..., キーワード引数=値):
    処理
    return 戻り値
```

関数名は、関数につける名前です（既存の関数の名前と重複しないように注意しましょう）。**引数**は関数に受け渡し情報を入れる変数のうち、関数を使うときに指定が必須な変数です。**キーワード引数**は、関数を使うときに必ずしも指定する必要がない引数です。**処**

理は、目的の処理をさせるための文の集まりで、インデントをして範囲を示します。戻り値には、結果として出力したい内容が代入されている変数を置きます。関数の目的によっては、戻り値が不要な場合もあります。そのようなときは `return` 戻り値 はなくてもかまいません。また、戻り値は複数でも構いません。変数でなく、式でも構いません。

それでは、税抜き価格から消費税込みの価格を求める関数 **ctax** を例に理解を深めましょう。関数 **ctax** は以下のように書くことができます。

```
In [25]: def ctax(p):
         rate = 0.1
         pt = int(p * (1+rate))
         return pt
```

この関数を使うときは、以下のようにします。

```
In [26]: zeinuki = 100                #税抜き価格を変数に代入する

         zeikomi = ctax(zeinuki)      #関数 ctax を呼び出して使う

         print("税込み価格:", zeikomi) #税込み価格を表示する
```

税込み価格: 110

Python では、関数にキーワード引数と呼ばれる特別な形式の引数を使うことができます。キーワード引数は、定義するときに、「引数=値」として、値まで書いておきます。そして、使うときもやはり「引数=値」として使います。この際、値は同じでも別でもかまいません。これ自体を省略することもできます。これは、デフォルト値を用意しておくときに便利です。使うときに何も書かなければ定義において指定された値が用いられ、キーワード引数が書かれていたらそれが使用されるからです。これを利用して、食料品等の税率にも対応できるようにしてみましょう。それは、以下のように書くことができます。

```
In [27]: def ctax2(p, isfood=False):
         if isfood==True :
             rate = 0.08
         else:
             rate = 0.1
         pt = int(p * (1+rate))
         return pt
```

通常（食料品等以外）商品の価格に対して関数を使用するには以下のようにします。

```
zeikomi = ctax2(zeinuki)
```

そして、食料品にこの関数を利用する場合は以下のようにします。

```
zeikomi = ctax2(zeinuki, isfood=True)
```

下のcellを適宜修正して実行し、キーワード引数を持つ関数の動作を確認してください。

```
In [28]: zeinuki = 100                #税抜き価格を変数に代入する

         zeikomi = ctax2(zeinuki, isfood=True) #関数 ctax を呼び出して使う

         print("税込み価格:", zeikomi) #税込み価格を表示する
```

税込み価格: 108

なお、関数 **ctax2** の定義において、キーワード引数 **isfood** の値は、**True** か **False** のどち

らなので、その下（2行目）のif文は以下で済ますことも可能です。

```
if isfood :
```

また、関数によっては引数を必要としないものがありますが、そのような関数を使うときも関数の後ろに引数のない括弧「()」をつける必要があります。

[目次に戻る](#)

2.5 ライブラリとインポート

とても便利な関数ができたとき、それを別なプログラムでも使いたくなります。プログラムそれぞれにその関数定義をコピーしても使いまわしは可能ですが、あまり賢い方法とは言えません。Python では便利な関数を一つのファイルにまとめておき、別なファイルに書かれているプログラムからそれを呼び出すことができます。便利な関数やオブジェクト定義を集めたファイルをライブラリとよびます。Python をインストールしたときに一緒にインストールされるライブラリを標準ライブラリと呼びます。一方、Python のインストールとは別にインストールが必要な、第三者が作ったライブラリを外部ライブラリまたはサードパーティー製ライブラリと呼びます。環境構築の際に `conda install` というコマンドでインストールした `numpy` などは外部ライブラリです。

ライブラリという呼び名のほかに、同じ意味でモジュール、パッケージという呼び名も使われます。利用の観点からはどれも同じですが、ライブラリ<モジュール<パッケージの順で規模が大きくなる傾向にあります。

これらを呼び出し利用可能にする操作を **インポート** と呼びます。インポートは簡単で、**import 文**と呼ばれる文をプログラムの始めに書いておくだけです。以下は、あるプログラムの冒頭に書かれているインポート文です。実用的なプログラムでは、多数のimport文記述されています。

```
from sys import exit
from datetime import datetime as dt
from datetime import timedelta as td
from math import floor, ceil
import numpy as np
import tempfile
import codecs
import urllib
import ssl
import xarray as xr
```

ライブラリ格納されている関数は、インポートをした後、次のようにして使うのが基本です。

ライブラリ名.関数名(引数, . . .)

しかし、ライブラリの名称は必ずしも短くないので、この記法だと不便なことが多々あります。また、多種多様なモジュールには名前が同じなのに機能が違う関数がたくさんあり、関数名だけでは混乱が生じます。そこで、モジュール名や関数名に別名を付けたり、関数を限定してライブラリ名を付けずに使えるようにしたりする方法が用意されました。上の引用に見られるように、インポート文に様な用法があるのはそのためです。

なお、インポート文はたいていプログラムの冒頭に書かれますが、これは慣習であり、その機能を使う前に書かれていればかまいません。

[目次に戻る](#)

2.6 多次元データ計算モジュールNumPy

多数のデータをひとまとまりにしたものとして、Python にはリストがあることを学習しました。1次元や2次元の数値データをリストで取り扱うことも不可能ではありませんが、これから説明するライブラリ NumPy（ナンパイ）が提供するデータ型 **ndarray** を使えば、圧倒的に強力かつ高速に多次元データを処理することができます。気象学を含む科学技術計算には必須のライブラリといってよいでしょう。

NumPy は大変よく使われるライブラリなので、インポートの方法も半ば慣例化していて、以下のようにします。

```
import numpy as np
```

2.6.1 数値配列オブジェクト ndarray

ndarray は、メッシュ気象データなどで使われる数値の多次元配列の操作を行うために用意されたデータ型で、numpy をインポートすると使えるようになります。

そして、関数 np.array 関数を使うと数値のみのリストやタプルから ndarray を生成することができます。

```
In [29]: import numpy as np

lst = [1, 2, 3, 4, 5]
data = np.array( lst )
data
```

```
Out[29]: array([1, 2, 3, 4, 5])
```

ndarray には算術演算子を使用できます。ndarray 同士に使用すると、相当する要素同士が演算されたものが得られます。ndarray は、一見リストに似ていますが、算術演算の結果は全く違うので混同しないようにしてください。

以下を順に実行し結果を確認してください。

```
In [30]: data + data
```

```
Out[30]: array([ 2,  4,  6,  8, 10])
```

```
In [31]: data - data
```

```
Out[31]: array([0, 0, 0, 0, 0])
```

```
In [32]: data * data
```

```
Out[32]: array([ 1,  4,  9, 16, 25])
```

```
In [33]: data / data
```

```
Out[33]: array([1., 1., 1., 1., 1.])
```

array と数値との演算では、全ての要素に対して数値が演算された結果が得られます。

```
In [34]: data + 10
```

```
Out[34]: array([11, 12, 13, 14, 15])
```

```
In [35]: data - 10
```

```
Out[35]: array([-9, -8, -7, -6, -5])
```

```
In [36]: data * 10
```

```
Out[36]: array([10, 20, 30, 40, 50])
```

```
In [37]: data / 10
```

```
Out[37]: array([0.1, 0.2, 0.3, 0.4, 0.5])
```

ndarray に比較演算子を適用すると、**True** または **False** の ndarray 要素それぞれが評価され、その結果が ndarray として戻ります。

```
In [38]: data > 2
```

```
Out[38]: array([False, False,  True,  True,  True])
```

```
In [39]: data == data
```

```
Out[39]: array([ True,  True,  True,  True,  True])
```

ndarray とリストを演算すると、リストが ndarray に変換されて演算が行われ、結果が ndarray として戻ります。

```
In [40]: data + lst
```

```
Out[40]: array([ 2,  4,  6,  8, 10])
```

なお、ndarray は、スライス等、リストで使用できたインデクシングは全て利用できます。

2.6.2 3次元の ndarray

ndarray は、任意の次元の数値配列を取り扱えますが、我々にとって身近な、3次元の配列を例に理解を深めましょう。

3次元の ndarray を作成する方法はいくつかありますが、所定のサイズの3次元配列を手っ取り早く新規作成するには、値がすべて0.0の配列を作る関数 **numpy.zeros** を用いるのが便利です。

一番目の次元（Python の数え変え方では第0次元）の要素が3個、二番目の次元(同第1次元)の要素が4個、三番めの次元（同第2次元）の要素が5個の3次元のゼロ配列 **arr** を、この関数を用いて新規作成するには次のようにします。

```
arr = np.zeros((3, 4, 5))
```

このようにして作った配列の中身をJupyterで表示させると、角括弧で多重に括られた形で表示されます。

```
In [41]: import numpy as np

arr = np.zeros((3, 4, 5))
arr
```

```
Out[41]: array([[[0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.]],

                [[0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.]],

                [[0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0.]])
```

今度は、各要素に格納される値が1ずつ増える同じサイズの3次元配列を、別な方法で作ってみます。

下の例で、**numpy.arange** は、0から始まる数値列を作る関数で、関数 **reshape** はそれを指定した要素毎に区切って多次元化するメソッドです(そう、ndarrayはオブジェクトです)。

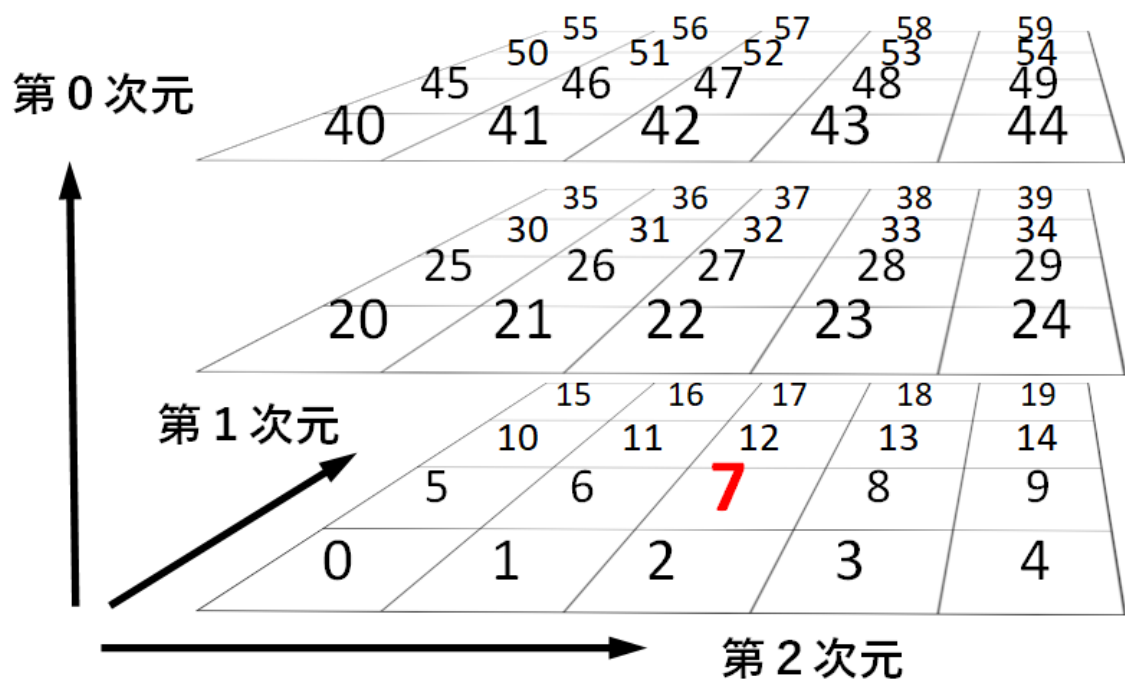
```
In [42]: arr = np.arange(60.0).reshape(3, 4, 5)
arr
```

```
Out[42]: array([[[ 0.,  1.,  2.,  3.,  4.],
                  [ 5.,  6.,  7.,  8.,  9.],
                  [10., 11., 12., 13., 14.],
                  [15., 16., 17., 18., 19.]],

                [[20., 21., 22., 23., 24.],
                  [25., 26., 27., 28., 29.],
                  [30., 31., 32., 33., 34.],
                  [35., 36., 37., 38., 39.]],

                [[40., 41., 42., 43., 44.],
                  [45., 46., 47., 48., 49.],
                  [50., 51., 52., 53., 54.],
                  [55., 56., 57., 58., 59.]])
```

さて、このような配列から、特定の要素を指定するには、インデックスをどのように指定したらよいでしょうか。3次元のndarrayは、(上の表示はさておき)図3のような3つの軸に沿った立体と考えると理解が容易です。

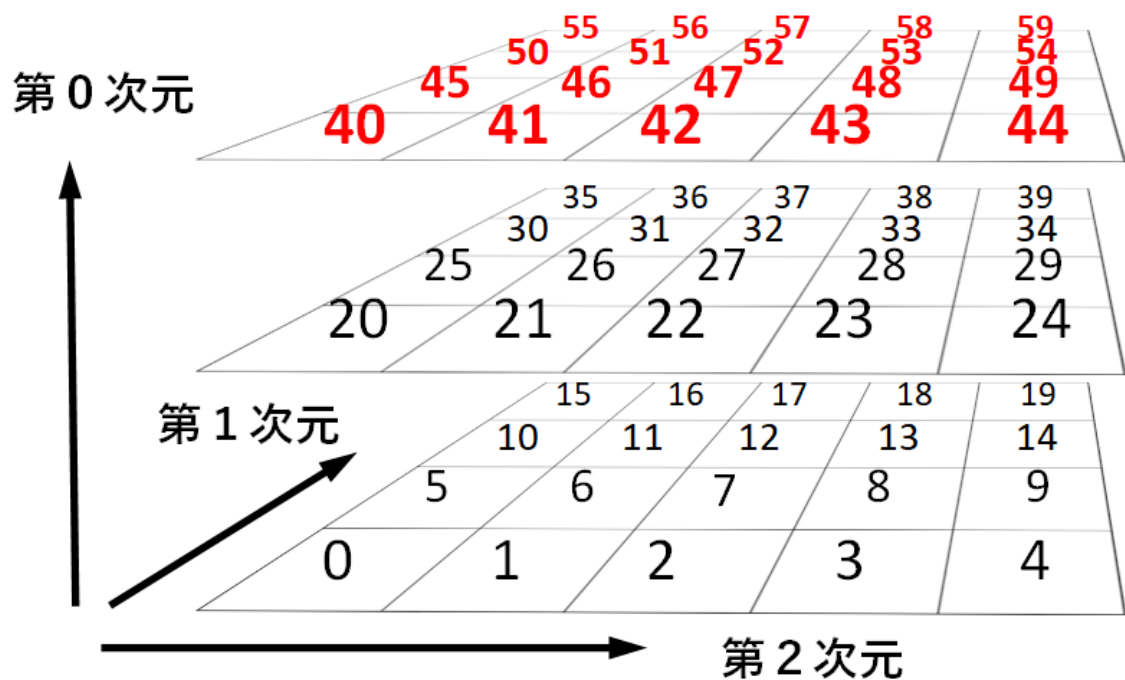


図中、7の値が入っている要素は、第0次元において0番目、第1次元において1番目、第2次元において2番目(いずれも0始まり)なので、これらを以下のように一つの角括弧の中に指定することにより抽出できます。

```
In [43]: arr[0,1,2]
```

```
Out[43]: 7.0
```

今度は、第0次元の0から数えて2番目の、すべての要素を指定してみます。

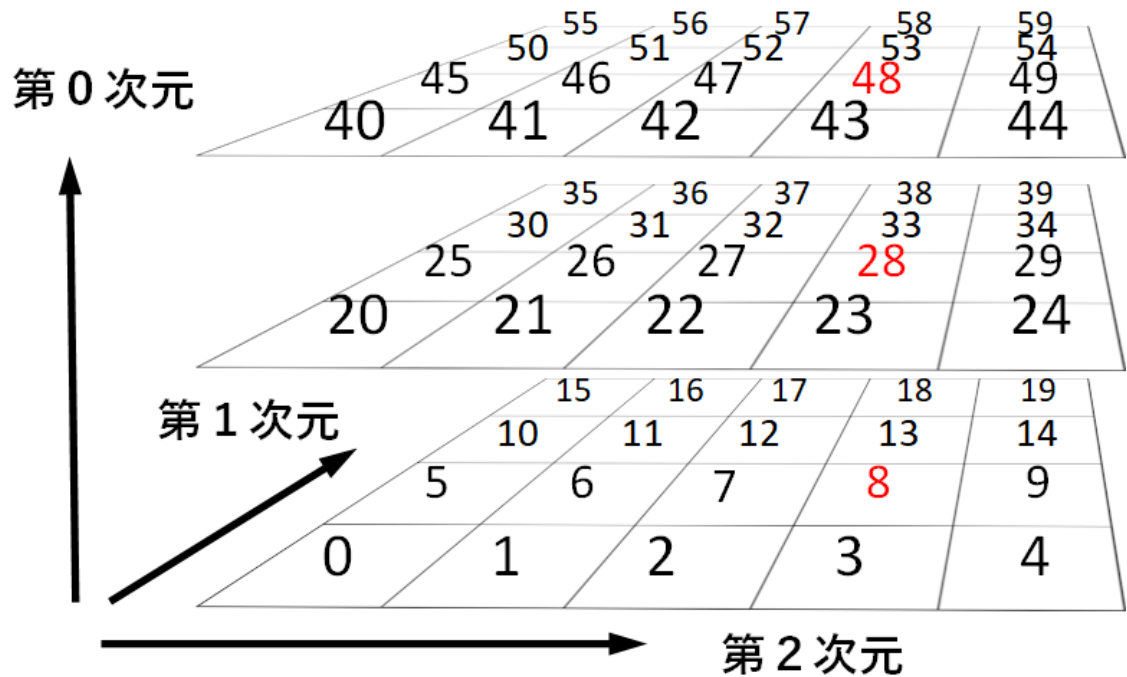


この場合は、「全て」を意味するコロンを用いて下のように指定します。


```
In [44]: arr[2,:,:]
```

```
Out[44]: array([[40., 41., 42., 43., 44.],
               [45., 46., 47., 48., 49.],
               [50., 51., 52., 53., 54.],
               [55., 56., 57., 58., 59.]])
```

それでは、第2次元が3番目、第1次元が1番目（いずれも0始まり）であるような要素をすべての第0次元から選ぶ方法は、どのようになるでしょうか。



この場合は、下のように指定します。

```
In [45]: arr[:,1,3]
```

```
Out[45]: array([ 8., 28., 48.])
```

2.6.3 ndarray の要素の合計等

ndarrayオブジェクトには沢山のメソッドが用意されていますが、メソッド **sum** を使うと、配列の各要素の合計を簡単に計算することができます。

```
In [46]: arr.sum()
```

```
Out[46]: 1770.0
```

メソッド **sum** はキーワード引数 **axis** を持っています。これを使うと、ある特定の次元軸に沿って合計をとることができます。下のように用いると、0次元軸に沿った合計を計算できます。なお、このメソッドを適用しても、arrayオブジェクト自身が書き換えられることはありません。

```
In [47]: arr.sum(axis=0)
```

```
Out[47]: array([[ 60.,  63.,  66.,  69.,  72.],
                [ 75.,  78.,  81.,  84.,  87.],
                [ 90.,  93.,  96.,  99., 102.],
                [105., 108., 111., 114., 117.]])
```

合計を計算する **sum** のほかに、平均を計算する **mean**、最大値を求める **max**、最小値を求める **min**、積み上げを行う **cumsum** があります。

2.6.4 無効値 nan

nan は、NumPy をインポートすると利用可能となる特別な浮動小数値で、**数として扱えない数(not a number)** を示します。数として扱えないので、nan と何かを演算すると結果は nan になります。また、nan と何かを比較しようとするとエラーとなります。

```
In [48]: arr = np.arange(60.0).reshape(3, 4, 5) # ndarrayオブジェクト arr を定義

print( arr.max(), arr.mean() )                #最大値と平均値を計算して表示

arr[0,1,3] = np.nan    # 8,28,48が入っていた場所にnanを代入

print( arr.max(), arr.mean() )                #nanが紛れているので集計できない

59.0 29.5
nan nan
```

この取り扱いには確かに筋が通っていますが、無効値は脇に置いて最大値や平均値を計算したいことはままあります。その時は、NumPyの関数 **nanmax** や **nanmean** を使います。

```
In [49]: arr = np.arange(60.0).reshape(3, 4, 5) # ndarrayオブジェクト arr を定義

print( arr.max(), arr.mean() )                #最大値と平均値を計算して表示

arr[0,1,3] = np.nan    #要素の1つにnanを代入

print( np.nanmax(arr) , np.nanmean(arr) )    #nanを除外して集計

59.0 29.5
59.0 29.864406779661017
```

2.6.5 ndarray オブジェクトの属性

一般に、オブジェクトは、利便性を高めるため「属性(attribute)」と呼ばれる情報を持つのが普通で、一般に、属性は以下のように参照します。

オブジェクト.属性

arrayオブジェクトもいくつかの属性を持っており、最もよく使われるのは属性 **shape** です。shapeは、そのオブジェクトの次元ごとの要素数のタプルです。以下のcellを実行して確認してください。

```
In [50]: arr.shape
```

```
Out[50]: (3, 4, 5)
```

オブジェクトの後ろにピリオド「.」を置きその後ろに名前をつなげて使うのはメソッド

と似ていますが、括弧「（）」を付けないことに注意してください。

[目次に戻る](#)

2.7 表型データ処理モジュール Pandas

Pandas（パンドス）は、列と行で整理が可能なデータの整理を効率的に行えるモジュールで、機械学習を行う前の**学習データの前処理**をするときに大変便利です。具体例を挙げると、顧客データの中からある条件（女性だけ）を満たす行を抽出したり、あるカテゴリー（男女別など）を設定してそれぞれの平均値（身長、体重など）を算出したり、複数の顧客データを結合したりするなどの操作ができます。

Pandasのも大変よく使われるモジュールなので、以下のような、慣習化したインポート文があります。

```
import pandas as pd
```

2.7.1 オブジェクト DataFrame

モジュールPandasをインポートすると、オブジェクト **DataFrame** を使うことができます。Dataframeは、行見出し（インデックス）と列見出し（カラム）で構成される表計算シートのようなオブジェクトです。

```
In [51]: import pandas as pd
```

それでは、いくつかの気象観測地点における気温と湿度の観測地をまとめた下記のデータをDataFrameに纏め、それを様々な操作してみましょう。

地点番号	地点名	気温	相対湿度	都道府県
18273	根室	1.0	79	北海道
62078	大阪	9.1	81	大阪
61111	舞鶴	4.8	98	京都
46106	横浜	8.5	65	神奈川
44132	東京	6.6	76	東京
14163	札幌	1.2	65	北海道
52146	高山	2.3	90	岐阜

DataFrameオブジェクトは様々な方法で作ることができますが、ここでは、Pandasが用意する関数 **DataFrame** を使用して辞書から作ることにします。下記を実行し、5要素の辞書を作ってください。

```
In [52]: #辞書の用意
dic1 = {'地点番号': ['18273', '62078', '61111', '46106', '44132', '14163', '52146'],
        '地点名': ['根室', '大阪', '舞鶴', '横浜', '東京', '札幌', '高山'],
        '気温': [1.0, 9.1, 4.8, 8.5, 6.6, 1.2, 2.3],
```

```
'相对湿度':[79,81,98,65,76,65,90],
'都道府県':['北海道','大阪','京都','神奈川','東京','北海道','岐阜']}]
dic1
```

```
Out[52]: {'地点番号': ['18273', '62078', '61111', '46106', '44132', '14163', '52146'],
'地点名': ['根室', '大阪', '舞鶴', '横浜', '東京', '札幌', '高山'],
'気温': [1.0, 9.1, 4.8, 8.5, 6.6, 1.2, 2.3],
'相对湿度': [79, 81, 98, 65, 76, 65, 90],
'都道府県': ['北海道', '大阪', '京都', '神奈川', '東京', '北海道', '岐阜']}
```

それでは次に、Pandasの関数 **DataFrame** でこの辞書から DataFrame オブジェクトを生成します。

下記を実行してください。

```
In [53]: #PandasのDataframeの生成
dfdata1 = pd.DataFrame(dic1)
dfdata1
```

```
Out[53]:
```

	地点番号	地点名	気温	相对湿度	都道府県
0	18273	根室	1.0	79	北海道
1	62078	大阪	9.1	81	大阪
2	61111	舞鶴	4.8	98	京都
3	46106	横浜	8.5	65	神奈川
4	44132	東京	6.6	76	東京
5	14163	札幌	1.2	65	北海道
6	52146	高山	2.3	90	岐阜

一番左列に表示されている 0, 1, 2, 3, 4, 5, 6 の値は、インデックスと呼ばれます。DataFrame関数が自動で付加しました。表計算ソフトでいう行番号にあたりますが、DataFrameのインデックスは文字も使うことができます。

```
In [54]: dfdata_abc = pd.DataFrame(dic1,index=['a','b','c','d','e','f','g'])
dfdata_abc
```

```
Out[54]:
```

	地点番号	地点名	気温	相对湿度	都道府県
a	18273	根室	1.0	79	北海道
b	62078	大阪	9.1	81	大阪
c	61111	舞鶴	4.8	98	京都
d	46106	横浜	8.5	65	神奈川
e	44132	東京	6.6	76	東京
f	14163	札幌	1.2	65	北海道
g	52146	高山	2.3	90	岐阜

行列の転置のように、**行と列を入れ替えたもの**は、属性 **T** で得ることができます。

```
In [55]: # 転置
```

```
dfdata_abc.T
```

```
Out[55]:
```

	a	b	c	d	e	f	g
地点番号	18273	62078	61111	46106	44132	14163	52146
地点名	根室	大阪	舞鶴	横浜	東京	札幌	高山
気温	1.0	9.1	4.8	8.5	6.6	1.2	2.3
相対湿度	79	81	98	65	76	65	90
都道府県	北海道	大阪	京都	神奈川	東京	北海道	岐阜

2.7.2 レコードの抽出

データ分析においては、**特定の条件を満たす行だけを取り出して**個別に分析することが良くあります。

DataFrameオブジェクト **df** から、指定した条件を満たすレコードだけを取り出す方法は、一般化して示すと次のようになります。

`df[条件を示す式]`

あまりに一般的過ぎてよくわかりませんね。具体例を示しましょう。いま、dfdata1から北海道の観測地点のレコードだけを取り出すとしましょう。この時、「条件を示す式」は以下ようになります。

```
dfdata1['都道府県']=='北海道'
```

複数の条件を組み合わせるには、「|」（または）、「&」（かつ）の論理演算子と条件の単位を示す括弧を使います。相対湿度の値が60～80%の範囲にある地点のレコードを取り出す場合の「条件を示す式」は以下のようになります。

```
(dfdata1['相対湿度']>=60) & (dfdata1['相対湿度']<= 80)
```

以下を実行し、レコードがどのように抽出されるかを確認して下さい。

```
In [56]:
```

```
# 条件（フィルター）  
dfdata1[dfdata1['都道府県']=='北海道']
```

```
Out[56]:
```

	地点番号	地点名	気温	相対湿度	都道府県
0	18273	根室	1.0	79	北海道
5	14163	札幌	1.2	65	北海道

```
In [57]:
```

```
# 複数条件（フィルター）  
dfdata1[(dfdata1['相対湿度'] >= 60) & (dfdata1['相対湿度'] <= 80)]
```

```
Out[57]:
```

	地点番号	地点名	気温	相対湿度	都道府県
0	18273	根室	1.0	79	北海道
3	46106	横浜	8.5	65	神奈川
4	44132	東京	6.6	76	東京
5	14163	札幌	1.2	65	北海道

2.7.3 行や列の再編成

DataFrameオブジェクト **df** の列を並べ替えるのは簡単です。以下のようにします。

`df[並べ直したい順序で書いた見出しのリスト]`

このとき、並べ替えだけでなく、取捨選択も同時にして構いません。

```
In [58]: dfdata1[ ['都道府県', '地点名', '気温', '相対湿度'] ]
```

```
Out[58]:
```

	都道府県	地点名	気温	相対湿度
0	北海道	根室	1.0	79
1	大阪	大阪	9.1	81
2	京都	舞鶴	4.8	98
3	神奈川	横浜	8.5	65
4	東京	東京	6.6	76
5	北海道	札幌	1.2	65
6	岐阜	高山	2.3	90

行を並べ替える場合は、メソッド **reindex** を使用する必要があります。同時の取捨選択も可能です。

`df.reindex(index=並べ直したい順序で書いたインデックスのリスト)`

表から、不要な列や行を削除するには、DataFrameオブジェクトのメソッド **drop** を使用します。

下のようになると、dfdata1 から湿度データを除去することができます。

```
In [59]: dfdata1.drop(['相対湿度'], axis = 1)
```

```
Out[59]:
```

	地点番号	地点名	気温	都道府県
0	18273	根室	1.0	北海道
1	62078	大阪	9.1	大阪
2	61111	舞鶴	4.8	京都
3	46106	横浜	8.5	神奈川
4	44132	東京	6.6	東京
5	14163	札幌	1.2	北海道
6	52146	高山	2.3	岐阜

ここで、キーワード引数 **axis** は列に対する操作の場合は **1** を、行に対する操作の場合には **0** を指定します。これはDataFrameオブジェクトの多くのメソッドで共通ですので、覚えておいてください。

と一いつつ、メソッド **drop** にはもう少しわかりやすいキーワード引数 **colmsn**、**index** が用意されていて、以下のようにすることもできます。

```
dfdata1.drop(columns=['相对湿度'])
```

ところで、表計算ソフトに慣れ親しんだ方の中には、「おいおい、表をそんなにいじくって大丈夫か」と心配された方もいるかもしれません。また、取捨選択で外したはずの「地点番号」の列が復活していることを不審を思った方もいるかもしれません。 DataFrame オブジェクトは、並べ替えや削除などを行っても、「操作をしたようになります」という **DataFrame**（これをviewといいます）を返すだけで、オブジェクト自体が変更されているわけではありません。なので、上の操作は累積してゆかないのです。不可逆的に変更するには、代入文を使い、viewで自分自身を上書きします。

```
dfdata1 = dfdata1.drop(columns=['相对湿度'])
```

それでは次に、DataFrameオブジェクトのソートを学びましょう。

まず、乱雑な順序のアルファベットをインデックスにしたDataFrameオブジェクト **dfdata_rand** を作成します

```
In [60]: dfdata_rand = pd.DataFrame(dic1,index=['b','e','g','c','a','d','f'])
dfdata_rand
```

```
Out[60]:
```

	地点番号	地点名	気温	相对湿度	都道府県
b	18273	根室	1.0	79	北海道
e	62078	大阪	9.1	81	大阪
g	61111	舞鶴	4.8	98	京都
c	46106	横浜	8.5	65	神奈川
a	44132	東京	6.6	76	東京
d	14163	札幌	1.2	65	北海道
f	52146	高山	2.3	90	岐阜

これを行**インデックス**で**ソート**します。それには、メソッド **sort_index** を使用し、以下のようになります。

```
In [61]: # indexによるソート
dfdata_rand.sort_index()
```

```
Out[61]:
```

	地点番号	地点名	気温	相对湿度	都道府県
a	44132	東京	6.6	76	東京
b	18273	根室	1.0	79	北海道
c	46106	横浜	8.5	65	神奈川
d	14163	札幌	1.2	65	北海道
e	62078	大阪	9.1	81	大阪
f	52146	高山	2.3	90	岐阜
g	61111	舞鶴	4.8	98	京都

特定の列の値でソートする場合には、次のように、メソッド **sort_values** を使用します。

```
In [62]: # 値によるソート、デフォルトは昇順
dfdata_rand.sort_values(by='地点番号')
```

```
Out[62]:
```

	地点番号	地点名	気温	相対湿度	都道府県
d	14163	札幌	1.2	65	北海道
b	18273	根室	1.0	79	北海道
a	44132	東京	6.6	76	東京
c	46106	横浜	8.5	65	神奈川
f	52146	高山	2.3	90	岐阜
g	61111	舞鶴	4.8	98	京都
e	62078	大阪	9.1	81	大阪

ソートの場合も、「ソートしたらこうなります」という **DataFrame** が返されているだけで、オブジェクトそのものは変更されていません。

2.7.4 DataFrameの結合

二つの表に共通する見出しの値を頼りに、二つの表を一つにまとめることを結合とよびます。Pandasの関数 **merge** を使うと **DataFrame** オブジェクトを結合することができます。

それでは、まず、結合する **DataFrame** **dfdata2** を用意しましょう。

```
In [63]: # 別のデータの準備
dict2 = {'地点番号': ['62078', '61111', '46106', '14163', '52146'],
         '緯度': [34.68, 35.45, 35.44, 43.06, 36.16],
         '降水': ['なし', 'あり', 'なし', 'あり', 'あり']}
dfdata2 = pd.DataFrame(dict2)
dfdata2
```

```
Out[63]:
```

	地点番号	緯度	降水
0	62078	34.68	なし
1	61111	35.45	あり
2	46106	35.44	なし
3	14163	43.06	あり
4	52146	36.16	あり

これを、**dfdata1** に関数 **merge** で結合します。この関数は、引数に2つ **DataFrame** だけを指定した場合、**同じ名前のカラムをキーとして内部結合**します。内部結合とは、値が両方に共通しているレコードだけを残して他を削除することです。この例の場合、キーは「地点番号」なので、両方に共通している **'62078'** , **'61111'** , **'46106'** , **'14163'** , **'52146'** の行だけが、結合された上で残されます。インデックスは振り直されます。

```
In [64]: # データのマージ
dfdata3=pd.merge(dfdata1,dfdata2)
dfdata3
```

```
Out[64]:
```

	地点番号	地点名	気温	相対湿度	都道府県	緯度	降水
0	62078	大阪	9.1	81	大阪	34.68	なし
1	61111	舞鶴	4.8	98	京都	35.45	あり
2	46106	横浜	8.5	65	神奈川	35.44	なし
3	14163	札幌	1.2	65	北海道	43.06	あり
4	52146	高山	2.3	90	岐阜	36.16	あり

2.7.5 集計

DataFrameオブジェクトを利用すると、様々なデータを集計することができます。この際、メソッド **groupby** を使うと、ある特定の列を軸とした集計ができます。

以下は「降水」の列を軸として、降水の有無しに分けて「気温」の平均を算出する例です。**スコア平均**を計算するにはメソッド **mean** を使います。他にも、最大値を計算する **max** や最小値を計算する **min**などのメソッドもあります。

```
In [65]: # データのグループ集計
dfdata3.groupby('降水')['気温'].mean()
```

```
Out[65]: 降水
あり      2.766667
なし      8.800000
Name: 気温, dtype: float64
```

DataFrame オブジェクトでは、相関係数のような統計値も計算できます。

以下は「緯度」の列と「気温」の列を軸として、緯度と気温の相関係数を算出する例です。**相関係数**は、0に近ければ**無相関**、1に近ければ正の相関、-1に近ければ負の相関であることを示す数です。

```
In [66]: # 緯度と気温の相関係数
dfdata3['緯度'].corr(dfdata3['気温'])
```

```
Out[66]: -0.7202142813485314
```

2.7.6 CSVファイルの書き出し/読み込み

DataFrameオブジェクトでさまざまに分析した結果をさらに他のアプリケーションで利用するとき、最も確実なのはおそらくCSVファイルを介したやり取りでしょう。

DataFrameオブジェクトはメソッド **to_csv** でDataFrameオブジェクトの内容を簡単にCSVファイルとして出力することができます。2.2.2 で作成したdfdata1をCSVファイルとして書き出してみましょう。

以下を実行してください。

```
In [67]: dfdata1.to_csv("result.csv", encoding='shift_jis')
```

フォルダ内にファイル **result.csv** が作られているはずです。表計算ソフトやテキストエディタ（「メモ帳」など）で中身を確認してみてください。

この逆に、CSVファイルの内容をDataFrameオブジェクトにするには、Pandasの関数 **read_csv** を使用します。今書きだしたCSVファイルを再び読み込んでDataFrameオブジェクト **csvdata** を作り出してみましょう。

以下を実行してください。

```
In [68]: csvdata_csv = pd.read_csv("result.csv", encoding='shift_jis', index_col=0)
csvdata_csv
```

```
Out[68]:
```

	地点番号	地点名	気温	相対湿度	都道府県
--	------	-----	----	------	------

0	18273	根室	1.0	79	北海道
1	62078	大阪	9.1	81	大阪
2	61111	舞鶴	4.8	98	京都
3	46106	横浜	8.5	65	神奈川
4	44132	東京	6.6	76	東京
5	14163	札幌	1.2	65	北海道
6	52146	高山	2.3	90	岐阜

CSVファイルは、ファイルの冒頭に空白行がおかれていたり、見出しがあったりなかったり、文字コードが特殊だったり、大変ルーズなデータフォーマットです。このため、メソッド **to_csv** や関数 **read_csv** には、それらに対応するための様々なオプションがキーワード引数として用意されています。オプションとその使い方については、「pandas csv オプション」などでインターネット検索すれば容易に知ることができます。

[目次に戻る](#)

2.8 時間処理モジュール datetime

気象データを分析する際には、日付や時刻、経過時間など取り扱う必要が生じることがよくあります。Pythonには、**datetime** と呼ばれるオブジェクトがあり、これを用いると時刻や時間を処理することができます。

2.8.1 日付時刻オブジェクト datetime

オブジェクト **datetime** を使用するには、ライブラリ **datetime** を以下のようにインポートします。

```
from datetime import datetime
```

そうすると、関数 **datetime** が使えるようになり、これを使ってオブジェクト **datetime** を生成したり処理したりできるようになります（誰が名付けたのでしょうか、モジュールの名前も、関数の名前も、作られるオブジェクトの名前も全部 **datetime** でややこしいですね）。

それでは、例として、2022年11月1日13時を表現するdatetimeオブジェクト **t** を作ってみましょう。

```
In [69]: from datetime import datetime

t = datetime(2022, 11, 1, 13, 0, 0, 0)

t
```

```
Out[69]: datetime.datetime(2022, 11, 1, 13, 0)
```

年や月、日、時、分の数値は、それぞれ、属性 **year**、**month**、**day**、**hour**、**minute** としてオブジェクトに保持されています。

下のcellの、ピリオドの後ろに属性名を書き込んで実行し、それらを確認してください。

```
In [70]: t.minute
```

```
Out[70]: 0
```

メソッド **weekday** を使用すると、月曜日～日曜日に相当する0～6の数値を得ることができます。

```
In [71]: youbi = ['月曜日', '火曜日', '水曜日', '木曜日', '金曜日', '土曜日', '日曜日']
youbi[t.weekday()] #数値を使ってリストから相当する曜日をを取り出す
```

```
Out[71]: '火曜日'
```

メソッド **now** を使うと、現在の時刻を得ることができます。

```
In [72]: tn = datetime.now()

tn
```

```
Out[72]: datetime.datetime(2022, 11, 8, 8, 35, 12, 224782)
```

datetimeオブジェクトは、**strftime** メソッドで様々な書式の文字列を作り出すことができます。それには以下のようにします。

```
In [73]: t.strftime("%Y年%m月%d日")
```

```
Out[73]: '2022年11月01日'
```

「年」や「月」など、数字以外の部分の文字列をまず作り、そのうえで年や月の数字の代理として「%Y」や「%m」を書き込みます。メソッド**strftime**では以下のメタ文字が利用できます。

```
%d : 0埋めした10進数で表記した月中の日にち
%m : 0埋めした10進数で表記した月
%y : 0埋めした10進数で表記した西暦の下2桁
```

%Y : 0埋めした10進数で表記した西暦4桁
 %H : 0埋めした10進数で表記した時 (24時間表記)
 %I : 0埋めした10進数で表記した時 (12時間表記)
 %M : 0埋めした10進数で表記した分
 %S : 0埋めした10進数で表記した秒
 %f : 0埋めした10進数で表記したマイクロ秒 (6桁)
 %A : ロケールの曜日名
 %a : ロケールの曜日名 (短縮形)
 %B : ロケールの月名
 %b : ロケールの月名 (短縮形)
 %j : 0埋めした10進数で表記した年中の日にち (正月が'001')
 %U : 0埋めした10進数で表記した年中の週番号 (週の始まりは日曜日)
 %W : 0埋めした10進数で表記した年中の週番号 (週の始まりは月曜日)

以下のcellを変更して実行し、日付や時刻の様々な文字列を作ってみてください。

```
In [74]: print(t.strftime('%A, %B %d, %Y'))
```

Tuesday, November 01, 2022

これとは逆に、「2020/11/30」のような文字列からdatetimeオブジェクトを作るには、関数 `datetime` の中の関数 **`strptime`** を使用して以下のようにします。

```
In [75]: strt = "2020/11/30"
datetime.strptime(strt, "%Y/%m/%d")
```

```
Out[75]: datetime.datetime(2020, 11, 30, 0, 0)
```

2.8.2 時間間隔オブジェクト `timedelta`

datetimeオブジェクトからdatetimeオブジェクトを引くと、その結果は、オブジェクト **`timedelta`** として戻されます。`timedelta`は、時間間隔を取り扱うためのオブジェクトです。

それでは、現在時刻と、2022年11月1日13時との時間差`ti`を求めてみましょう。

```
In [76]: ti = tn - t
ti
```

```
Out[76]: datetime.timedelta(days=6, seconds=70512, microseconds=224782)
```

これから推察できるように、datetimeオブジェクトにtimedeltaオブジェクトを加えることで、時間経過後の日時を定めることができます。これを実用するためには「1日後」とか「8時間後」などのような指定した時間の週間後timedeltaオブジェクトを作ることができなければなりません。これは、以下のようにして、関数 **`timedelta`** をインポートすることで可能となります。

```
from datetime import timedelta
```

関数 `timedelta`は以下のようにして使います。

```
In [77]: from datetime import timedelta
```

```
td = timedelta(days=1)
t2 = t + td
t2
```

Out[77]: datetime.datetime(2022, 11, 2, 13, 0)

```
In [78]: td = timedelta(hours=8)
t2 = t + td
t2
```

Out[78]: datetime.datetime(2022, 11, 1, 21, 0)

- Pythonには、このdatetimeオブジェクトのほかにも時間を取り扱うためのオブジェクトがいくつかあり、少々混沌としています。すでに学習した、モジュールNumPyには、日付の分布を表現するdatetime64[D]型なるオブジェクトがあり、モジュールPandasにはtimestampなるオブジェクトがあります。オブジェクトとメソッドは対になっているので、Pythonで時間を取り扱う時には、それが、どのオブジェクトで与えられているかを常に認識する必要があります。

[目次に戻る](#)

2.9 データ可視化モジュール Matplotlib, Seaborn

データ分析をする上で、対象となる**データを可視化**することはとても重要です。単に数字を眺めているだけでは、データに潜む傾向がなかなか見えなかったりしますが、データをビジュアル化することで、**データ間の関係性**なども見えてきます。ここでは、主に **Matplotlib** と **Seaborn** を使って、**データを可視化**する基本的な方法を身につけましょう。

Matplotlib（マットプロットリブ）は様々なグラフを描画することができるモジュールで、描画に関する一般の機能は、ほとんどの機能が「**matplotlib.pyplot.機能名**」で提供されています。グラフの大きさや軸の範囲、ラベルなどグラフを構成する様々な要素をこれで定めますが、それだけにこの長々とした名前は邪魔です。そこで、Matplotlibは以下のようにインポートすることが慣習となっています。

```
import matplotlib.pyplot as plt
```

このようにインポートすると、これらの機能は「**plt.機能名**」で利用できるようになります。

Seaborn はMatplotlibのグラフを、さらにきれいにするモジュールです。インポートするだけでグラフがきれいになり、また、いくつかの追加のスタイルが利用可能となります。

なお、Jupyter Notebook を使用している場合は、Notebook上へのグラフ表示が可能となるよう、以下の一文が必要です。Jupyter Labを利用している場合は特段記述する必要はありません。

```
%matplotlib inline
```

「%」で始まるこの文は、Jupyterで利用可能な文で、**マジックコマンド**と呼ばれます。

```
In [79]: # Matplotlib と Seabornの読み込み
```

```
# pyplotにはpltの別名で実行できるようにする
import matplotlib.pyplot as plt
import seaborn as sns

# Jupyter Notebook上でグラフを表示させるために必要なマジックコマンド
%matplotlib inline
```

Matplotlibでは、さまざまなグラフを描けますが、まずは、データ分析でよく使う散布図から始めましょう。散布図は、2つの組み合わせデータに対して、 x - y 座標上に点をプロットしたグラフです。

下のcellは、乱数で発生した100個の x と y の散布をプロットした図を作成するものです。10行目の **plt.subplots()** でグラフを描く台紙にあたるオブジェクト **fig** と、グラフの図柄にあたるオブジェクト（「軸」と呼ばれることが多い） **ax** を生成し、13行目で、**ax** 上にメソッド **plot** で点々をプロットします。 **ax.plot(x, y, 'o')** の最後の引数はグラフの形状を指定するもので、**o** は点で描くという意味です。

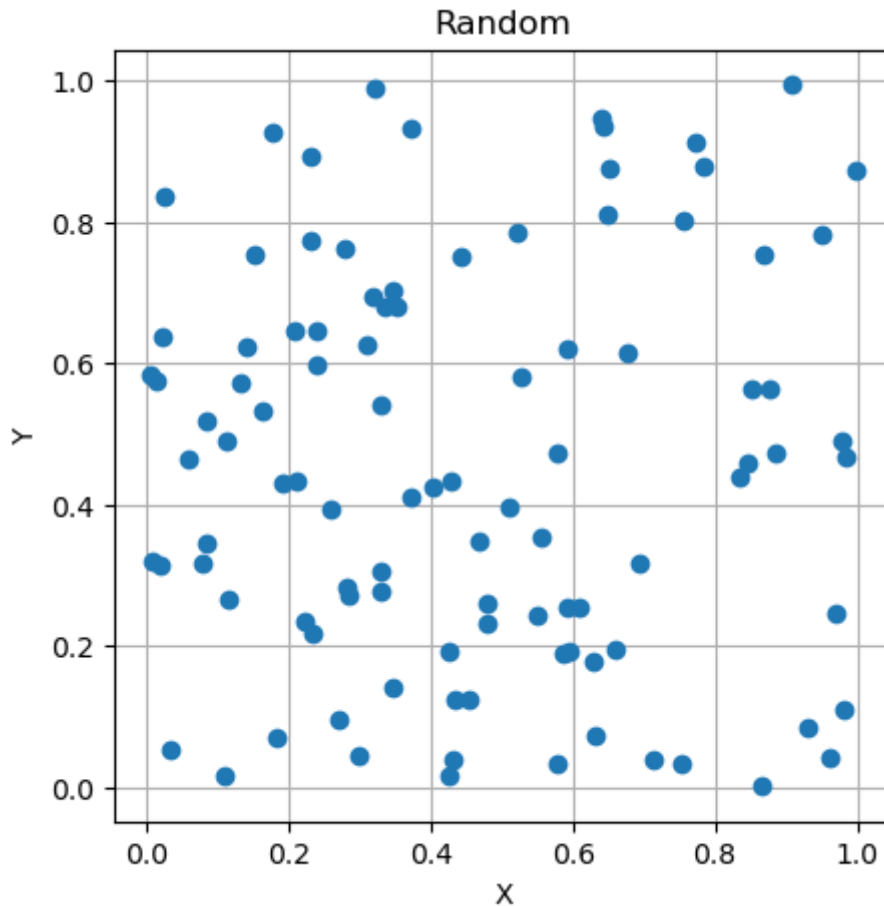
その他の動作については、コード中のコメントを参考にしてください。

```
In [80]: import numpy as np
# x軸のデータ（0から1までの一様分布の乱数生成）
x = np.random.rand(100)
# y軸のデータ（0から1までの一様分布の乱数生成）
y = np.random.rand(100)

# '台紙'のオブジェクトfigと'図柄'のオブジェクト（「軸」と呼ばれることが多い）axを生成
fig, ax = plt.subplots(figsize=(5,5))

# 画の作成
ax.plot(x, y, 'o')
# グラフタイトルの追加
ax.set_title('Random')
# x軸タイトルの追加
ax.set_xlabel('X')
# y軸タイトルの追加
ax.set_ylabel('Y')
# グリッド線（グラフの中にある縦線と横線）の表示
ax.grid()

plt.show() #表示
```



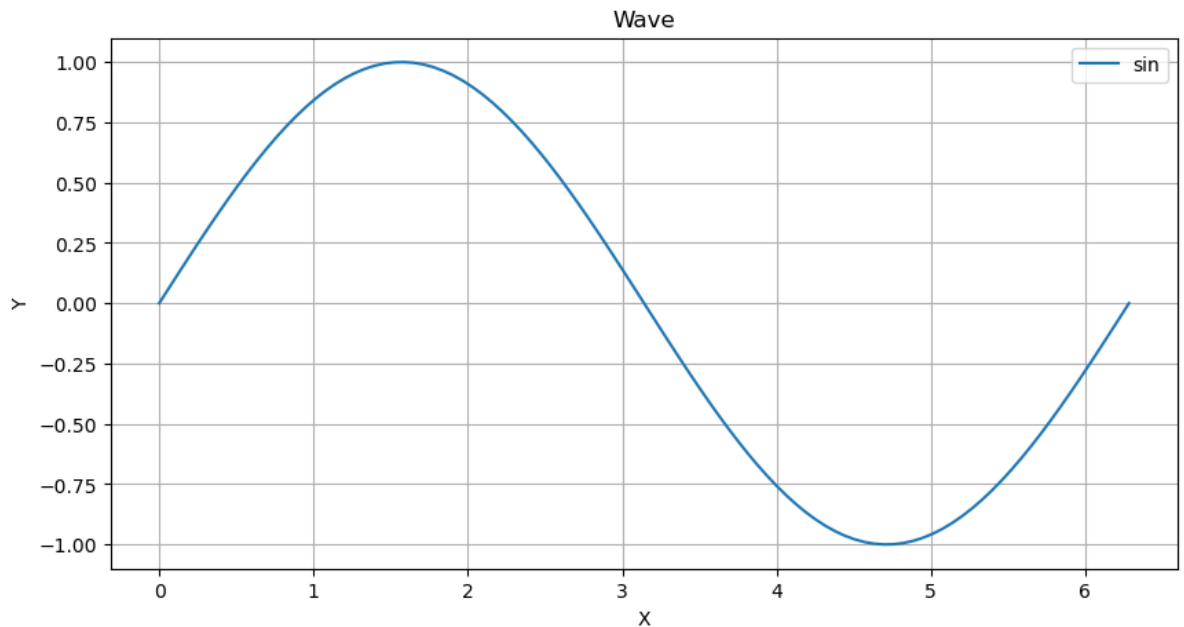
連続した値を与えれば、`plot` による描画は点ではなく曲線として示すことができます。たとえば次の例は、時系列など連続した点を曲線として描くものです。なお、NumPyの関数 `linspace(x1, x2, n)` は、`x1` と `x2` の間で等間隔に `n` 個並ぶ数字の配列を作る関数で、`NumPy.pi` は円周率です。

```
In [81]: # x軸のデータ (0から2πまでを100刻み (均等割) のデータを作成する)
x = np.linspace(0, 2*np.pi, 100)
# y軸のデータ (sinの計算)
y = np.sin(x)

# '台紙'のオブジェクトfigと'図柄'のオブジェクト (「軸」と呼ばれることが多い) axを生産
fig, ax = plt.subplots(figsize=(10,5))

# 画の作成
ax.plot(x, y, label="sin")
# グラフタイトルの追加
ax.set_title("Wave")
# X軸タイトルの追加
ax.set_xlabel('X')
# Y軸タイトルの追加
ax.set_ylabel('Y')
# グリッド線 (グラフの中にある縦線と横線) の表示
ax.grid()
# 凡例の追加
ax.legend()

plt.show() #表示
```

subplots に分割数を指定すると、一つの領域に複数の小さなグラフを埋め込むことができます。以下は、二つのグラフを2行1列に置く例です。

```
In [82]: import numpy as np

# プロットするデータ
x = np.linspace(-10,10,100)
y1 = np.sin(x)
y2 = np.sin(2*x)

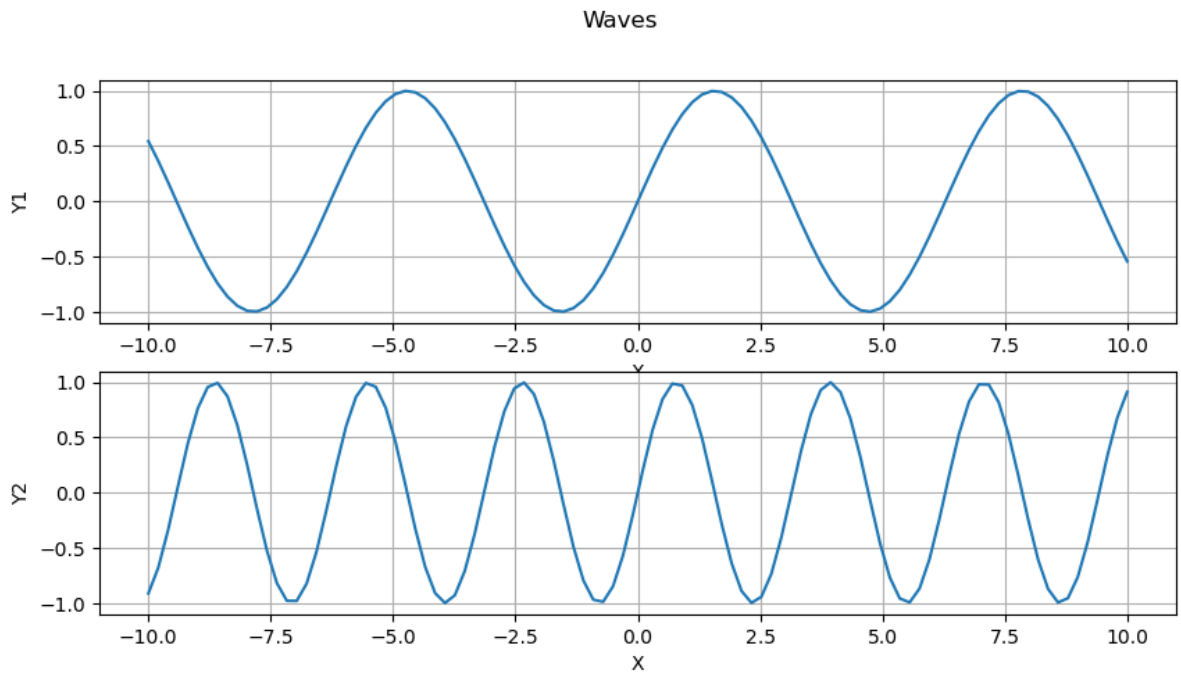
# '台紙'のオブジェクトfigと、'画'を二つ持つオブジェクトaxesを生成
fig, axes = plt.subplots(2,1,figsize=(10,5))

# 全体タイトルの追加
fig.suptitle("Waves")

# 画の作成
axes[0].plot(x, y1)
# X軸タイトルの追加
axes[0].set_xlabel('X')
# Y軸タイトルの追加
axes[0].set_ylabel('Y1')
# グリッド線（グラフの中にある縦線と横線）の表示
axes[0].grid()

# 画の作成
axes[1].plot(x, y2)
# X軸タイトルの追加
axes[1].set_xlabel('X')
# Y軸タイトルの追加
axes[1].set_ylabel('Y2')
# グリッド線（グラフの中にある縦線と横線）の表示
axes[1].grid()

plt.show() #表示
```



1枚の図に2つのグラフを重ねて表示することもできます。それには最初に軸オブジェクトを1つ作り、メソッド **twinx** を使って、これとx軸を共有するもう一つの軸オブジェクトを作り出します。

以下は、振幅が異なる二つの波を、一つのx軸に重ねて表示した例です。2種類のy軸を与えることで、2つのデータを比較しやすくなります。

```
In [83]: import numpy as np

# プロットするデータ
x = np.linspace(-10,10,100)
y1 = np.sin(x)
y2 = np.sin(2*x)*10 #振幅が10

# '台紙'のオブジェクトfigと、'画'のオブジェクトaxを生成
fig, ax = plt.subplots(figsize=(10,5))

# x軸を共有するもう一つのグラフ本体のオブジェクトx2を生成
ax2 = ax.twinx()

# 全体タイトルの追加
fig.suptitle("Waves")

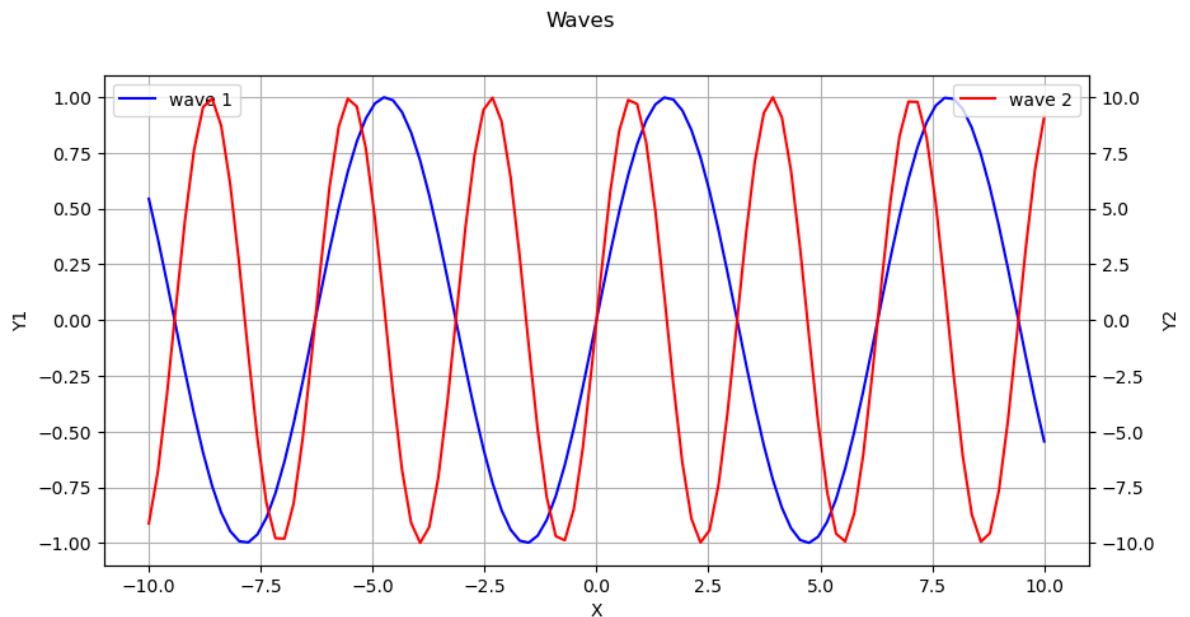
# x軸タイトルの追加
ax.set_xlabel('X')

# axにコア部分を作成
ax.plot(x, y1, color='blue', label='wave 1')
# y軸タイトルの追加
ax.set_ylabel('Y1')
# グリッド線（グラフの中にある縦線と横線）の表示
ax.grid()

# グラフのコア部分の作成
ax2.plot(x, y2, color='red', label='wave 2')
# y軸タイトルの追加
ax2.set_ylabel('Y2')
```

```
# 凡例を分けて追加
ax.legend(loc='upper left')
ax2.legend(loc='upper right')

plt.show() #表示
```



- Matplotlibのplot機能を用いた描画には、現在、二つの流儀があり、正式名ではありませんが、それぞれ、「MATLABスタイル」「オブジェクト指向スタイル」などと呼ばれています。このテキストではオブジェクト指向スタイルで記述しました。グラフについては、ここで説明しなかった様々な設定項目が山ほどあります。これらはネットで検索すれば使い方を調べられますが、その際、どちらの記法で説明されているかには注意してください。

この例のように、図柄のオブジェクト(ax)と図柄を置く領域のオブジェクト(fig)を明示的に作っていたらオブジェクト指向スタイルで、PyPlotの機能で (`plt.△△△(xx,...)` のような文を連ねて) グラフを組み立てていたらMATLABスタイルです。

[目次に戻る](#)

あとがき

本資料では、大変便利な **Jupyter Nptebok**、**Jupyter Lab** を利用して **Python** の基礎を学習しました。

気象データ分析での使用を念頭に説明事項を構成しているので、言語入門書としてはいびつな内容となっていますが、見よう見まねで気象データ分析にチャレンジされる皆さんが疑問に思うと想定されることがらについてはできるだけ丁寧に説明をしました。

本資料で紹介したライブラリやその関数には、説明し切れなかった多くの機能やオプションがありますが、それらは、ネットで検索することで容易に知ることができますので、この資料をスタート地点として、各自プログラミングスキルを磨いていってください。

著作権について

Copyright (c) 気象データ×IT勉強会 2022 All rights reserved.

<利用条件>

本書は、本書に記載した要件・技術・方式に関する内容が変更されないこと、および出典を明示いただくことを前提に、無償でその全部または一部を複製、翻案、翻訳、転記、引用、公衆送信等して利用できます。なお、全体または一部を複製、翻案、翻訳された場合は、本書にある著作権表示および利用条件を明示してください。

<免責事項>

本書の著作権者は、本書の記載内容に関して、その正確性、商品性、利用目的への適合性等に関して保証するものではなく、特許権、著作権、その他の権利を侵害していないことを保証するものでもありません。本書の利用により生じた損害について、本書の著作権者は、法律上のいかなる責任も負いません。

[目次に戻る](#)