セミナーの進め方

- **ご質問**に関して
  - **講義中**のお願い
    - **質疑応答の時間**にまとめて取り上げますので、一度**チャット欄**に投稿していただけると助かります
    - **口頭で質問したい場合は、概要だけ**で構いません。質疑応答の時間にお答えします
  - **講義後**も可能
- **環境と資料**に関して
  - 環境は`Google Colab`
  - 本セミナーでのコードは、`Google Colab`で動作確認済み
    - 他の`Python`実行環境でも利用可能ですが、環境ごとに仕様が異なるため、動作が変わることがある
  - 講義の方針に関して
    - 厳密には正確でない部分がある場合もございますが、**分かりやすさを重視**
      - 本セミナーでは初心者の方にも理解いただけるよう、**例え**を用いて解説
      - 講義で扱った範囲内のトピックに基づいて説明し、**例外の共有を極力避ける**
  - 講義資料
    - コードのみ：**セミナー中**に送付
    - 解説付きコード：**本セミナー終了後**に送付
- **トラブルシューティング**
  - 画面は**Ctrl キー+マウス ホイールで、拡大または縮小**できます

    ※大きめに表示していますが、各自調整をお願いします。
  - `ノートブック の読み込みエラーを防ぐため、無圧縮形式の zip ファイルを使用`しています。

    ※そのためファイルサイズが大きくなっていますが、ご了承ください。

はじめに

目的とセミナーの流れ

`NumPy`、`Pandas`、`Matplotlib`といった**データサイエンスに欠かせないライブラリ**の基本操作を通じながら、**基礎体力を養う**こと。

手法だけでなく**原則を重視**することで、 受講者が**セミナー後に自ら応用できる力**を身につけることを目指す。

ゴルフのスイングと同じで、単に繰り返し練習するだけでは上達せず、正しいフォームという原則を理解した上で練習することが効果的

- 今後に利用できる`Python`が操作できる`Google Colaboratory`の環境構築
- 変数、オブジェクト、関数などの`Python`の中心仕組みを学ぶ
- あるあるエラーの防止策についても触れ、初心者がつまずきやすいポイントを克服

そのために、**コード実行→解説→原則→演習**の流れで進める。

――つまり、

下記のように、ありがちな"順序通りに登場して順に説明する"進め方は取りません。

- 変数・データ型（`int`, `float`, `str`, `bool`, `list`, `dict`など）
- 演算子（算術、比較、論理）

- 条件分岐（if, elif, else） など

実際のコードでは、複数の構文や概念が入り混じりながら、一連の動きとしてまとまって存在するためである。

そうした“まとまり”の中で、
それぞれの要素がどう関係し合って動いているのかを捉えていく。

▍ Python とデータ分析ライブラリとの親和性

Python はデータ分析**ライブラリが豊富**で**データサイエンス**分野に強い

↓ ↓

データ **科学(サイエンス)**

まさに、**データを活用して、科学的(サイエンス)に問題を解決**することを意味

|

v

新たなコーディングほぼ不要

↓

手作業ミス減少 & 💻 ⚠️ 独自実装の誤り低減

↓

利用が進む

|

v

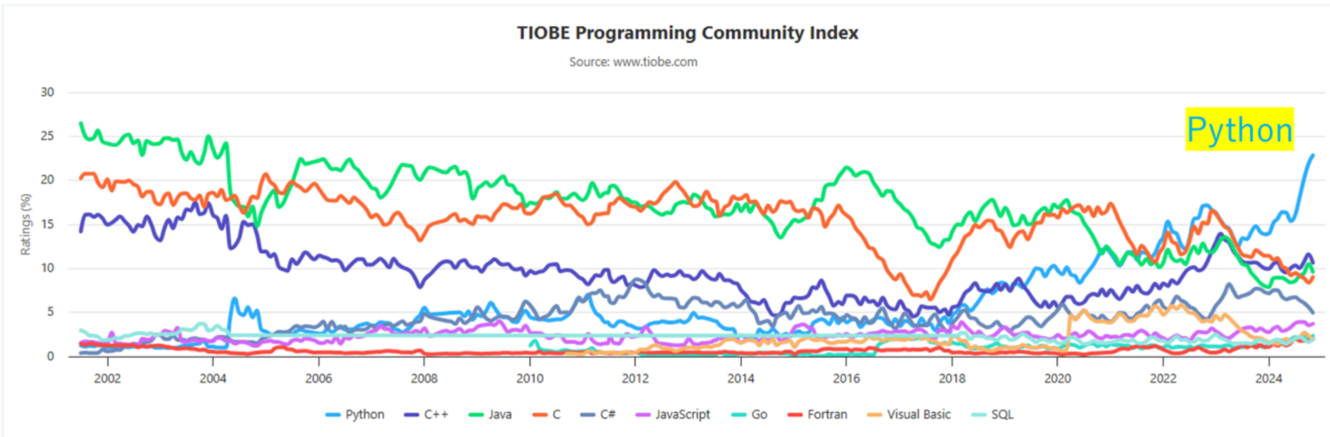
◎ 利用の増加

Python は、 TIOBE Index(1) の 2024年11月版 において、 22.85% のレーティングで **1位** にランクイン。

前年同期比で **8.69% 増加**。 Python は過去にも 1位 を獲得しており、現在も利用がさらに拡大している。

利用者が増えることで、 **ライブラリ やレファレンスが充実**し、さらに多くの利用者を引き寄せるという好循環が生まれている。

(1) 世界中の熟練エンジニアの数、コース、サードパーティベンダーの数に基づいてプログラミング言語の人気を測定する指標。

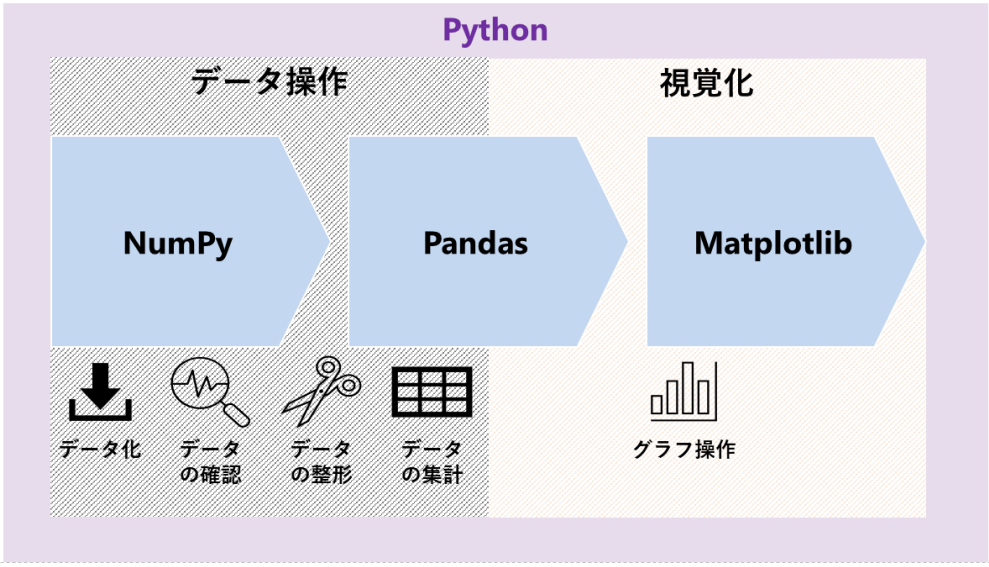


IOBE Index - TIOBE.「TIOBE Index」TIOBE、[https://www.tiobe.com/tiobe-index/、アクセス日：2024年12月6日](https://www.tiobe.com/tiobe-index/) より改変。

Python の中でも、

NumPy、 Pandas、 Matplotlib は、実際のデータ分析との親和性が高く、

いわばデータ分析の三種の神器であると言える。



環境とその準備

本講座では、 Python の実行環境として Google Colab を使用します。

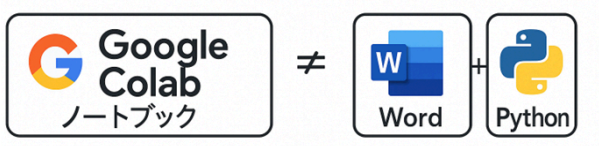
(1) Google Colab とは？

Google Colab はブラウザ上で Python コードを実行できる無料のツールで、特別な環境設定が不要で簡単に始められます。

Google Colab のノートブックを基盤している。
ノートブックとは、 Word のように文章を書けて、 Python のコードも **実行できるもの**。

つまり、**マニュアル書のような役割**も担える。

- コードと説明を一体化できるため、手順や意図を明確に記録できる
- 別の人への引継ぎが可能になる
- 従来のエディタのコメントアウト機能の限界を打破する
  など



- Google Colab 特有の機能は主に以下です。
  - 主要 ライブラリ がインストール済み
  - Gemini と連携済み
  - 簡易的な自動コーディング機能を搭載
  - GitHub と連携可能
  - クラッシュ時の自動復元
  - ノートブックをワンクリックで共有可能
  - 多発するバージョン不整合エラーが起きづらい(そのための仮想環境を作成せずに済む)
  - 無料　など

(2) Google Colab の準備方法

別資料のPDFに沿って、進めてください。

Numpy を学びましょう

NumPy を使って、数値データを「データ操作」する方法を理解する。

💬 コメント

最初の解説には、かなり長い。

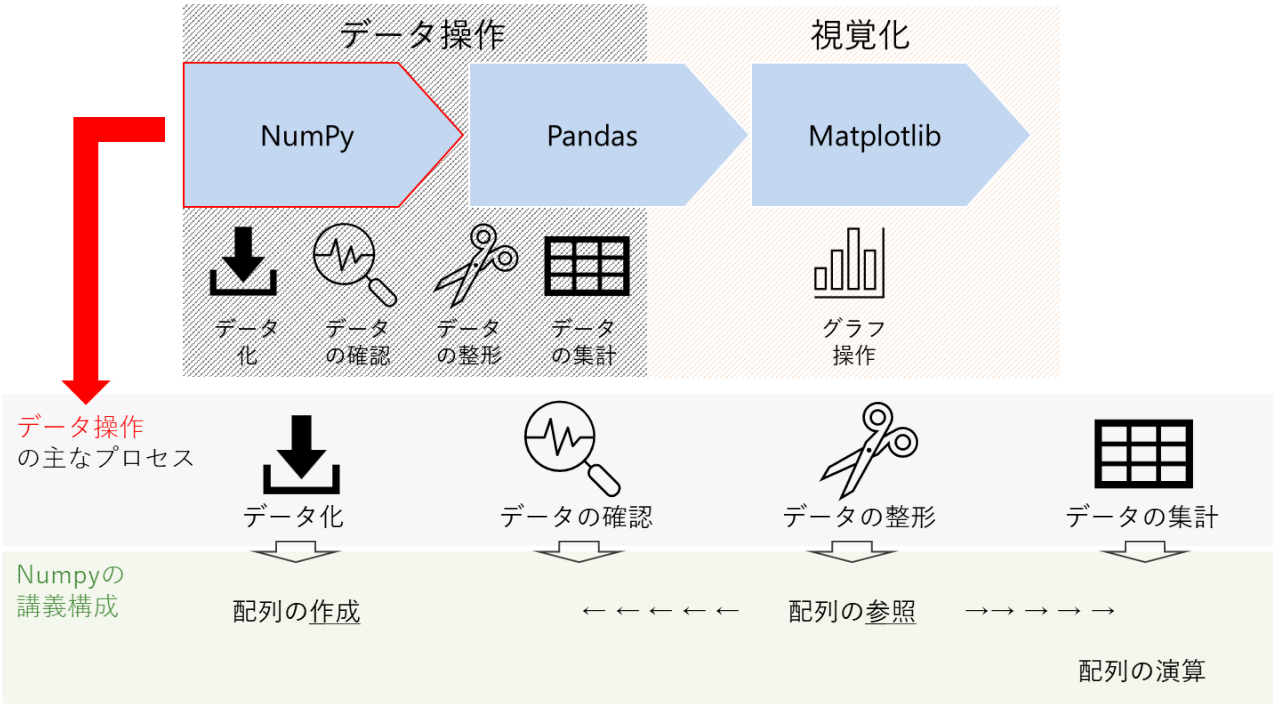
なんとなく説明して、次に行くことが容易ではあるが、

それは本講座の狙いでない。

裏を返せば、こちらにエッセンスが詰まっている。 本講座の基盤になる。

大胆に言えば、ここの全集中力を費やしてほしい。

▌Numpy によるデータ分析の位置付けと講義範囲



NumPy とは？

NumPy は、**Number（数や数学）** と **Python** を組み合わせた名前の通り、

Python で大量の数値データを効率よく扱うためのライブラリ（ツール）。

特徴

- 数学的な計算を素早く行うために設計されている
- 大きなデータでも簡単に計算できる
- データの高速処理が可能で、データサイエンスで広く活用されている

具体例

- **リストや表にある数字をまとめて計算**する
- **特定の範囲での集計**を効率よく実施する

NumPy は、数値データを扱うあらゆる場面で非常に役立つ便利なツールです

NumPy を使うための準備

使用手順

(1) NumPy のインストール > (2) NumPy のインポート

1. NumPy のインストール

NumPy を使用するには、まずPython環境に NumPy をインストールする必要がある
インストールには、以下のように、どちらかの pip install を使用する

ただし、 Google Colaboratory の環境下では、不要。 （Google Colaboratory は自宅ではなく、既に NumPy が用意されたオフィスだから）

```
#pip install numpy  # コマンドラインやターミナルから実行する際に、推奨されている標準的なコマンドです
#!pip install numpy  # Jupyter Notebook や Google Colabratory などのコードセルから実行する場合に、推奨されているコマンドです
```

■■ 原則：ライブラリとは ■■

過去の人々の知識が集まった本棚(≒ライブラリ)のイメージ。

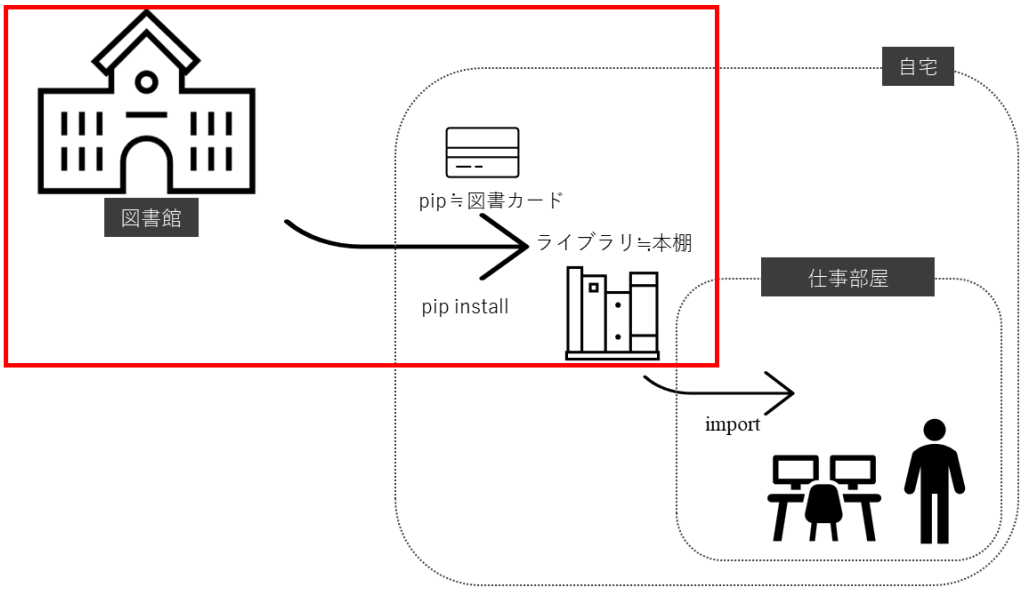
これらを利用すれば、 **誰でも同じ成果**を得ることができる。

ライブラリは、先人達の知恵を使い、

自分ですべての処理を細かく書かなくても、

あらかじめ用意された機能を使うことで、

複雑な処理を簡単に実行できるためのもの。



図書館で**図書カード**（pip）を使い、 **ライブラリ（本棚）を借りる**（pip install）ようなイメージ。

――つまり、

Numpy という本棚(≒ライブラリ)を使用するイメージ。

■■ 原則：ライブラリのインストール ■■

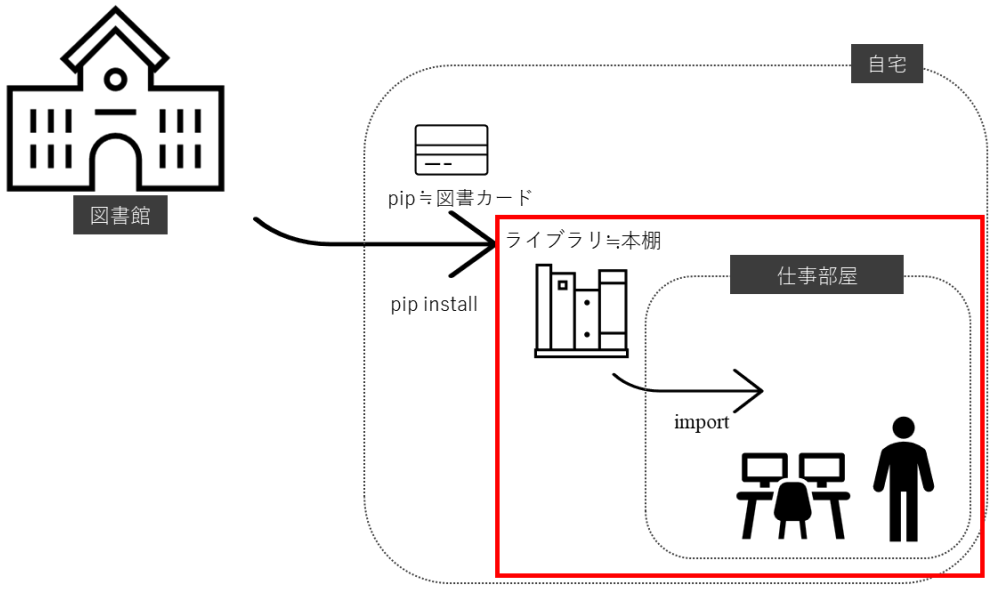
構文は以下

```
```python
 # コマンドラインやターミナルから実行する場合
 pip install ライブラリ
 # ノートブック内のコードセルから実行する場合
 !pip install ライブラリ
```
```

続いて、インストールしたライブラリをインポートにして、アクセスできる。

2. Numpy のインポート

```
1 import numpy as np
```



自宅の作業部屋に持ってくる（import）ようなイメージ。

――つまり、

Matplotlib という本棚(≒ライブラリ)から、

pyplot という本(モジュール)を取り出すイメージ。

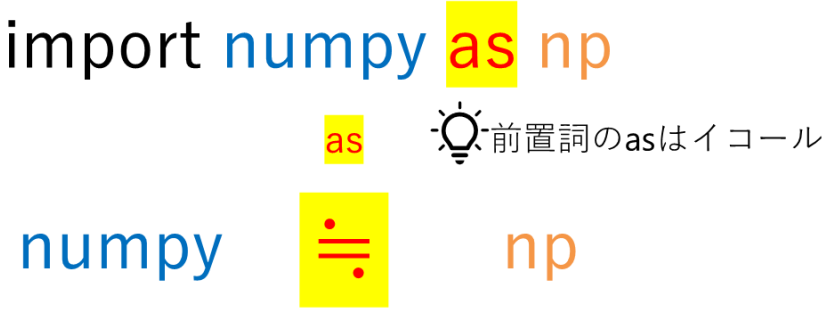
■■ 原則：ライブラリのインポート ■■

構文は以下

```
import ライブラリ
```

、または、

```
import ライブラリ.モジュール
```



as np により、 np という**短縮名**で使用できるように設定です(厳密にはエイリアスの設定)。

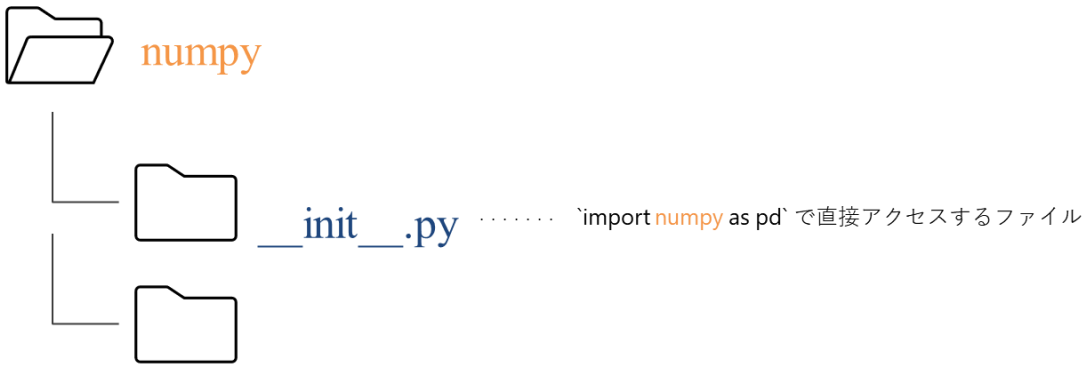
- numpy に別名（np）を付けるために、前置詞 as が使われる。
  - 前置詞 as は「イコール（=）」のような働き

    (as の後ろに名詞が1つしかないため、 as は前置詞。前置詞は後ろに名詞を持つ特性があるため)

📝 内部メカニズム

- `import numpy as np` のメカニズムに関して

  `import numpy as np` でアクセスするファイルは `__init__.py` です。`__init__.py` がエントリーポイントで、このファイルを通して `Numpy` の基本機能が利用できる。



- miniforge3の場合のアクセスイメージ

```
~/miniforge3            # 例: miniforge仮想環境におけるnumpyのディレクトリ構造（環境によって異なる）
└── envs
    └── myenv           # 仮想環境の名前（例: myenv）
        └── lib
            └── pythonX.X  # Pythonバージョンに依存（例: python3.9）
                └── site-packages
                    └── numpy
                        ├── __init__.py       # `import numpy as np` でアクセスするファイル
                        ├── core
                        │   ├── __init__.py
                        │   ├── numeric.py
                        │   └── ...
                        ├── linalg
                        │   ├── __init__.py
                        │   ├── lapack_lite.py
                        │   └── ...
                        └── random
                            ├── __init__.py
                            ├── _generator.py
                            └── ...
```

■■ 原則：コメントアウトとは ■■

`Python` では、コメントアウトを `#` を使って表現する。

「コメントアウト」とは、その名の通り、

コメント（注釈）＋アウト（除外する） ⇒ **「コメントを入れるが、コードの処理からは除外する」**役割を果たす。

という意味。

そのため、`#` 以降に書かれた内容はプログラムとして実行されない。

```
pip install numpy # コマンドラインやターミナルから実行する際に、推奨されている標準的なコマンドです
                  # 以降に書かれた内容はプログラムとして実行されない
```

この特性を利用して、次のような**活用法**がある。

- **コードへの注釈**

  コメントアウトを使うことで、コードの目的や補足情報を伝える。

  ```
  !pip install numpy  # Jupyter Notebook や Google Colab などのコードセルから実行する場合に推奨されているコマンドです
  ```

- **一時的にコードを無効化**

  実行したくないコードをコメントアウトすることで、無効化できる。

  ```
  # pip install numpy  # コマンドラインやターミナルから実行する際に推奨されている標準的なコマンドです
  ```

Numpy の操作

配列の基本操作

NumPy を使って、データを**配列**という形で扱います。
配列は、複数の値を一つにまとめて管理するものです。

Python の普通のリストに似ていますが、 NumPy の配列は計算が速く、
多くの数値データを効率的に処理することができます。

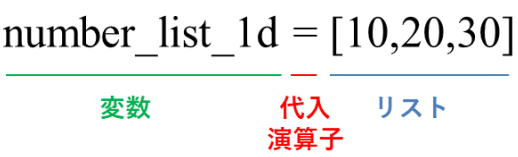
配列の作成

1次元配列の具体的な作成方法

```
1 # リストから1次元配列を作成
2 number_list_1d = [10,20,30]
3 array_1d = np.array(number_list_1d)
4 print(array_1d)
```

📝 解説

◎ number_list_1d = [10,20,30] の解説



■■ 原則：変数とは ■■

- 左辺：number_list_1d は**変数**
  - number_list_1d は [10,20,30] のリストオブジェクトへの参照を持つ変数
  - number_list_1d という名前を通じて、[10,20,30] というデータを操作できる
    - 変数：
      - 由来は、**値が変わる（変化する）可能性があるもの**
        - **値に名前**を付け、**その名前を通じて値を参照および操作**する仕組み
        - また、変数に格納された値は**変更可能**であり、プログラムの動的な挙動を実現
      - プログラム内で特定の値を一時的に保存するために、**メモリ上の住所を確保**し、**その住所に名前を付ける**仕組み
        - この住所には**値を格納したり、別の値に置き換え**ることができる

        😊 変数は便利：

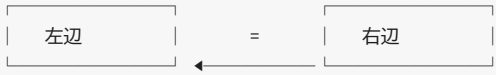
・一度保存した値を何度も使え、同じ値を繰り返し書く手間が省ける

・ 後で値を変えたくなったときも、一か所直すだけで済むので便利

など

■■ 原則：代入演算子とは ■■

- `=`**は代入演算子**
  - 右辺の値を左辺の変数に代入する動作を指す
  - 数学的な`=`により、**「左辺と右辺が等しい状態にする」**と考えると分かりやすい



- 右辺の`[10, 20, 30]`は、リストと呼ばれる。厳密には、`list`クラスで作られたリストオブジェクト

■■ 原則：オブジェクトとは ■■

◎ オブジェクトとは

オブジェクトは**操作対象**です。炊飯器のようなものです。

オブジェクト（炊飯器）を対象にして、以下の操作を行う

- **メソッド・プロパティ等**
  オブジェクトに`.`を付けてメソッドやプロパティを使用

  例: `number_list_1d.append(90)` など

  ```
  オブジェクト.メソッド
  オブジェクト.プロパティ
  ```

- **引数として使用する**
  引数に、`number_list_1d`が参照するリストオブジェクトを渡して使用

  例: `np.array(number_list_1d)` など

  ```
  関数(引数)
  オブジェクト.メソッド(引数)
  ```

💬 コメント

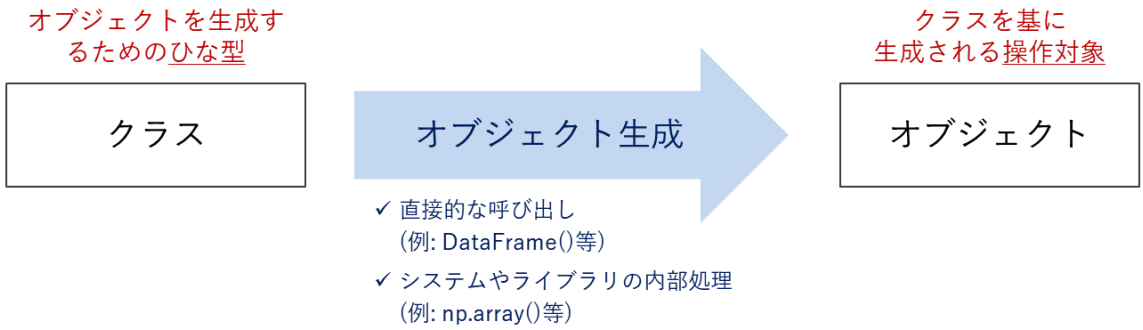
オブジェクトは、クラス（設計図）を基に生成される。

（設計図無しには、炊飯器が生成されないことと同様です）

オブジェクトは、**直接的な呼び出し（例: DataFrame()）**とシステムやライブラリの**内部処理（例: np.array()）により生成**される。

そのため、見かけ上、オブジェクト生成がされないように見えるケースがある。

しかし、実際はオブジェクト生成され、オブジェクトを操作しているのである。



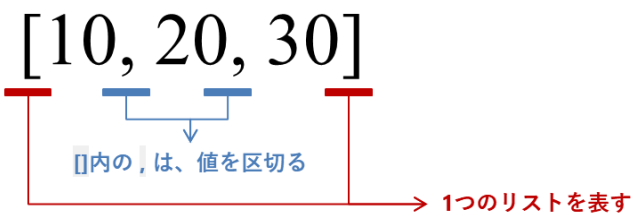
なお、`number_list_1d = [10,20,30]`は、以下のように書いても同じ結果になります：

```
number_list_1d = list([10,20,30])
#number_list_1d = [10,20,30]
number_list_1d
```

■■ 原則：リストとは ■■

リストとは、**複数の値を1つの変数名とインデックスで管理**する仕組み

- **角括弧 [ ] を使い、その中に値をカンマ , で区切って記述**
  - [ ] で Python では1つのリストを表す
  - [] 内の , は、CSV のように値を区切る役割

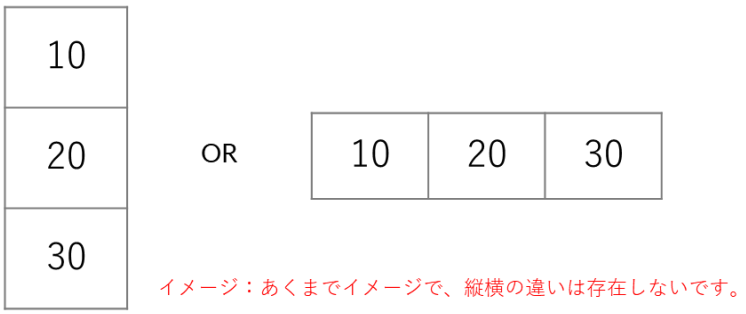


▌1次元リスト

単純に1つのリスト。単にデータを並べたリスト。

```
array_1d = [10, 20, 30]
array_1d
```

1次元リストの中で、各要素を , で区切る。



イメージ：あくまでイメージで、縦横の違いは存在しないです。

▌2次元リスト

2次元リストには複数のリストが含まれている。

「行」「列」のような形で、EXCEL データのような集まり。

- 行の区切り：1次元リストと異なり、各リストを , で区切る

- 列の区切り：各リストの値は、1次元リストと同様に、各要素を`,`で区切る

10	15	20
25	30	35
40	45	50

```
array_2d = [
    [10,15, 20],
    [25, 30, 35],
    [40, 45, 50]
]
array_2d
```

◎ `np.array(number_list_1d)`の解説

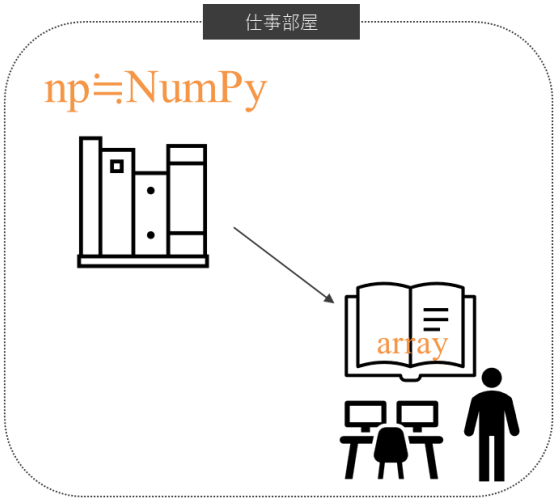
```
# リストから1次元配列を作成
number_list_1d = [10,20,30]
array_1d = np.array(number_list_1d)
print(array_1d)
```

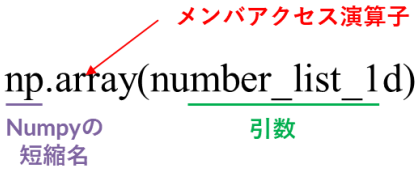
の

```
np.array(number_list_1d)
```

を解説する。

`np`（ライブラリ）と名付けした本棚から、特定の本の章やページの`array`(関数)を取り出すイメージ。





■■ 原則：関数とは ■■

関数とは、**適切な材料さえ用意すれば、料理ができる炊飯器**のような存在。

誰でも美味しい同じ味が再現でき、**誰でも同じ結果**を得られる。

料理という出力（戻り値）を得るために、**炊飯器という処理（関数）**を選び、

材料という**条件（引数）**を準備すればOK。

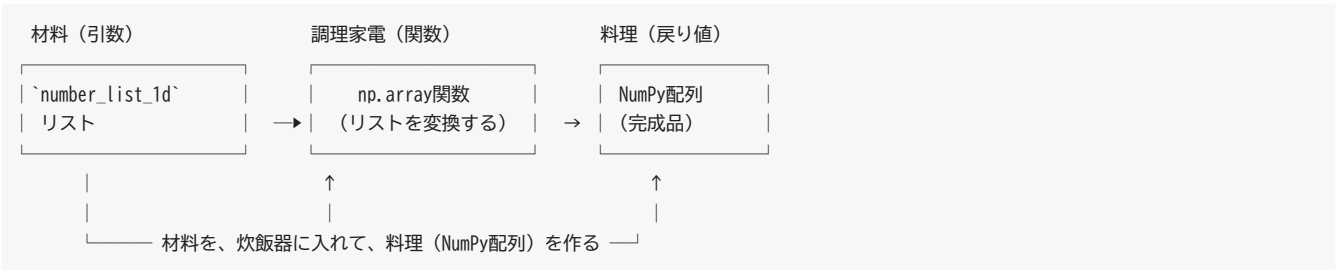
💬 コメント

オブジェクトも炊飯器の例えであり、関数との違いに混乱するかもしれない。

その点に関しては巻末の「おまけ」を参照。

つまり、`np.array(number_list_1d)` は、下図のような関係性である。

- **材料（引数）**：リスト `number_list_1d`（データの集まり）
- **炊飯器（関数）**：`np.array`
- **料理（戻り値）**：NumPy 配列



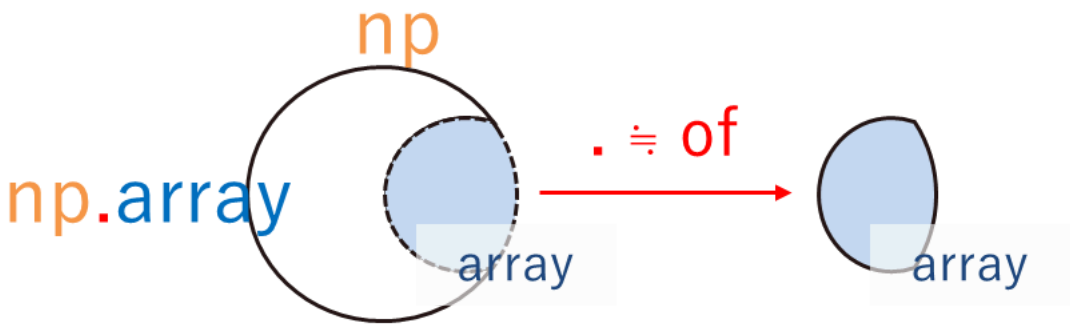
> **補足**: `np.array()` は `NumPy` の **`ndarray` クラス**のオブジェクトを生成している。

■■ 原則：`.` を侮るな ■■

`.` は、英語の `of` に相当し、

`np.array` の場合では「全体（`NumPy`）の中の一部（`array`）」を示す。

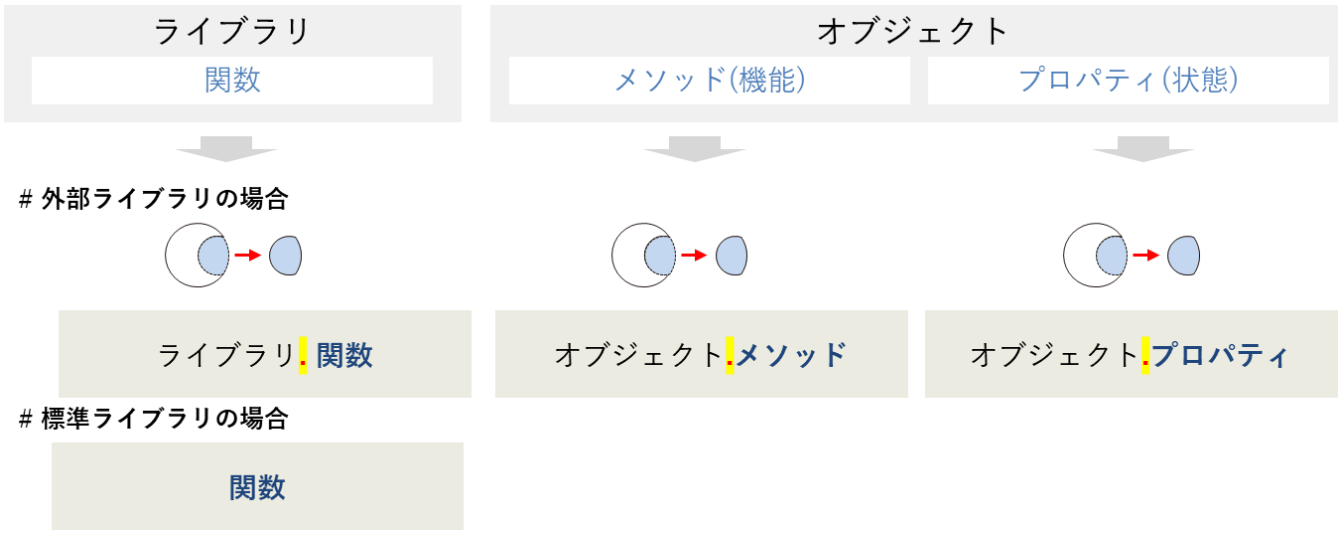
`NumPy` が持つ特定の機能である `array` にアクセスする際に使用する記号。



ofは「全体の中の特定部分を指す」

Python では、ライブラリ や オブジェクト に対して ドット (.)でアクセスし、 メソッド や プロパティ を呼び出して処理を行う。

※ オブジェクト.メソッド().メソッド()... のような記述は、見た目上 メソッド が連続しているように見えるが、実際には 各メソッド が オブジェクト を返しているため、それを受け取って次の メソッド を呼び出すという仕組みになっている。



▌ print(array_1d) の print() に関して

```
# リストから1次元配列を作成
number_list_1d = [10,20,30]
array_1d = np.array(number_list_1d)
print(array_1d)
```

の

```
print(array_1d)
```

を解説する。

組み込み関数	外部ライブラリの関数
print() 関数	外部 ライブラリ np.array() 関数

`print()` は**Pythonの組み込み関数**で、追加のライブラリをインポートすることなく、すぐに使うことができる。

一方、`numpy` のようなライブラリは**外部モジュール**として、`import` 文を使って呼び出す必要がある。

- `print()`:
  - **組み込み関数**
    - `Python` に最初から含まれている標準関数なので、特にインポートする必要はない
- `numpy（例: np.array）`:
  - **外部ライブラリの関数**
    - `import numpy as np` のように、最初にモジュールをインポートしてから使う

> ノートブック環境での `print()` の使いどころ

通常のPython環境では、`print()` を使わないと出力されない。

ノートブック環境では、`print()` なしでも表示される。

`numpy` では、`print()` なしだと、そのまま表示すると見づらい。

`pandas.DataFrame` は `print()` なしの方が表形式で整形されて見やすい。

適宜、使い分けてください。

◎ リストでは不十分？`NumPy` を使う理由

リストは大量のデータや計算に不向きです。

- `NumPy` は**高速な数値演算・統計処理**が可能
- リストでは数値演算に**ループ処理**が必要だが、`NumPy` 配列なら**「+」演算子**一つで複数要素の加算などが簡単に実行できる

▼ 2次元配列の具体的な作成方法

```
# リストのリストから2次元配列を作成
number_list_2d = [[10, 15, 20],[25, 30, 35],[40, 45, 50]]
array_2d = np.array(number_list_2d)
print(array_2d)
```

配列の参照

`NumPy` の配列では、**インデックス**または**スライサー**を使って、**特定の要素や範囲**を参照できる。

配列参照し、様々な操作を実施する

▼ 単一参照

インデックスを使って、**配列内の特定の要素を取り出す**。

▼ `Numpy` の1次元配列の場合

```
 # array_1d =  [10, 20, 30]
 print(array_1d[2])
```

◎ array1d[2] に関して

■■ 原則：インデックスによる参照 ■■

▌インデックスとは

Python のリストと配列は**主にインデックス番号**を基準に要素を参照する。

⚠️ 実際のデータは 1 から始まりますが、 Python ではインデックスが 0 から始まる

▎インデックスの定式化　※本セミナー独自

インデックス指定の定式化

実際の行や列番号 = インデックス + 1

⚠️ ただし、インデックスが1始まりに変更したり、アルファベット等の場合は、定式化が成り立たないので注意

▎インデックスの記述方法

Numpy の1次元配列を扱うとき、

単一の要素を参照する場合 は、 array_1d[インデックス] と記述する。

`Numpy` の1次元配列　～ 単一の要素を参照する場合　～

`array_1d[インデックス]`

array_1d[1] の場合は以下の要素を参照する。

実際の行や番号	1	2	3
	10	20	30
インデックス	0	1	2

Numpy の2次元配列の場合

```
# array_2d = [[10, 15, 20],[25, 30, 35],[40, 45, 50]]
print(array_2d[0, 1])
```

◎ array_2d[0, 1] に関して

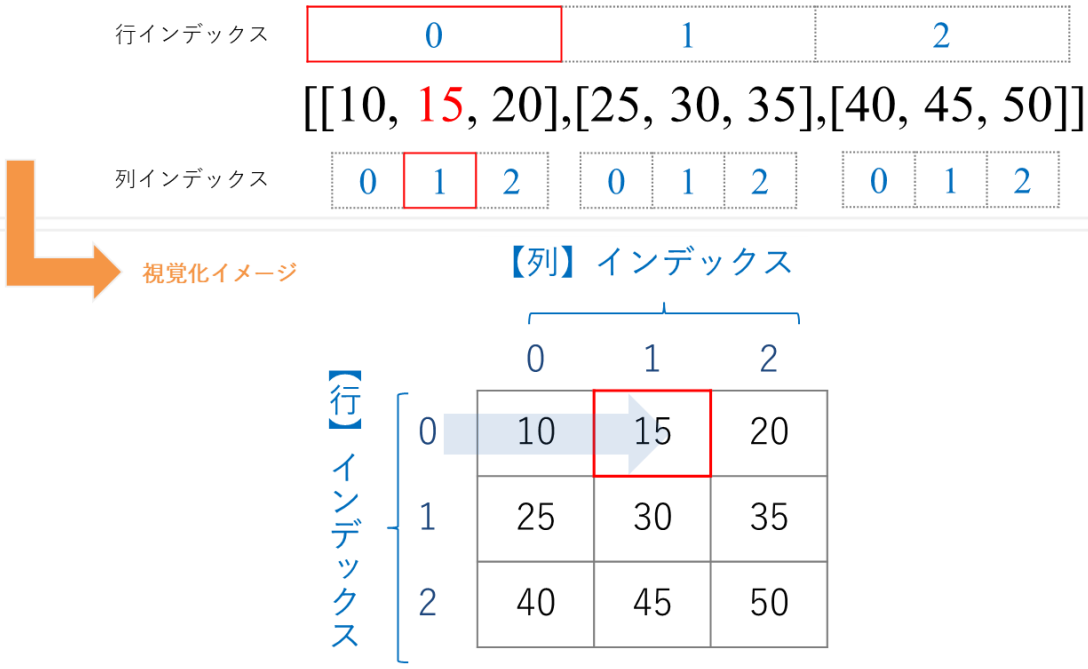
`Numpy` の2次元配列　～ 単一の要素を参照する場合　～

`array_2d[行インデックス, 列インデックス]`

- Numpy の2次元配列を扱うとき、単一の要素を参照する場合は、 array_2d[行インデックス, 列インデックス] と記述する。
  - 具体的に、 array_2d[0,1] は以下の要素を参照
    - 行：インデックス 0 → 実際の行番号は 0 + 1 = 1 行目

- 列：インデックス `1` → 実際の列番号は `1 + 1 = 2` 列目

したがって、`array_2d[0, 1]` は、**実際の1行2列目の要素**を参照



範囲参照

部分参照

```
1 # array_1d =  [10, 20, 30]
2 print(array_1d[0:2])
```

◎ `array_1d[0:2]` に関して

■■ 原則：スライスによる参照 ■■

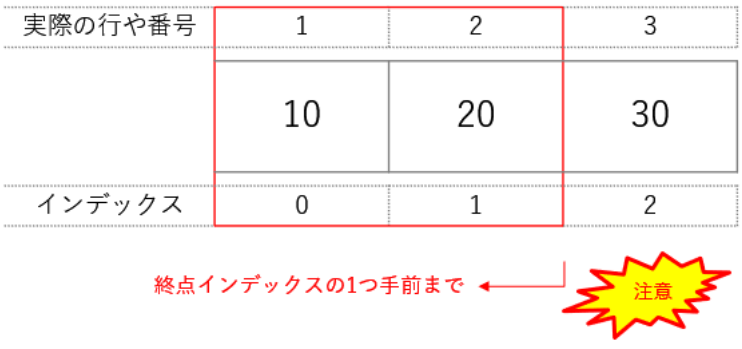
単一の要素ではなく、**範囲**を参照する方法。

スライスと呼びます。

「スライス（slice)」は「切り取る」という意味であり、 配列の一部を切り出して取得することに由来する。

範囲を指定する際、**終点のインデックス**は含まれません。

```
始点 <= x < 終点
```



`array_1d[0:2]` の場合は 開始=0, 終了=2 を指定し、

インデックス 0 の要素 `10` からインデックス 1 の要素 `20` までを参照する。

▌構文まとめ

★ 単一の要素を参照

1次元：

```
[インデックス]
```

2次元：

```
[行インデックス，列インデックス]
```

★ 範囲内の要素を参照（スライス）

- ステップ=指定した間隔で参照
- `ステップ`は省略すると、`1`

1次元：

```
[開始:終点:ステップ]
```

2次元：

```
[行の開始:行の終点:ステップ，列の開始:列の終点:ステップ]
```

全参照

列全体を参照したいケース。

```
# array_2d = [[10, 15, 20],[25, 30, 35],[40, 45, 50]]
print(array_2d[0, :])
```

```
 # 2次元配列から最初の行を取り出す
 print(array_2d[0, ])
```

◎ `array_2d[0, :]`、`array_2d[0, ]`に関して

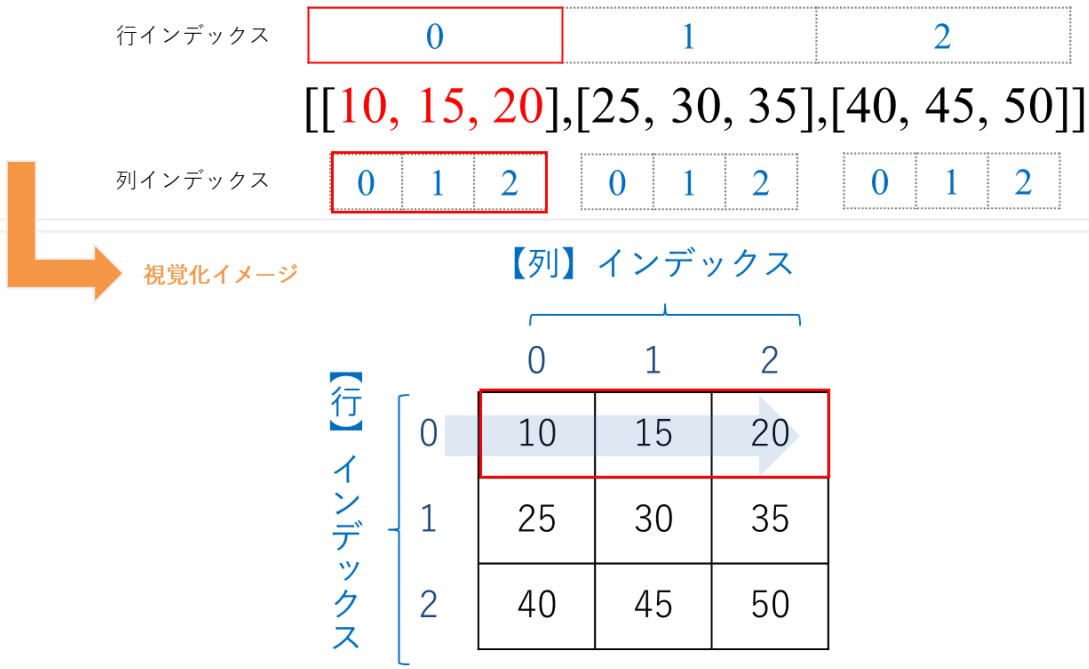
■■ 原則：スライスの省略形 ■■

```
`array_2d[行の開始:行の終点:ステップ，列の開始:列の終点:ステップ]`
```

`array_2d[0, :]`は、 `列の開始`、`列の終点``を省略し、`:`が残った形。

さらに、

`array_2d[0, ]`は、`array_2d[0, :]`をさらに省略した形。



配列の演算

四則演算

```
# 2つの配列を作成
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
# 配列同士の足し算
result_add = array1 + array2
print(result_add)
```

```
# 配列同士の掛け算
result_multiply = array1 * array2
print(result_multiply)  # 結果: [ 4 10 18]
```

`array1 + array2` 、`array1 * array2` が使える点は、

まさに前述の『リストでは不十分？NumPyを使う理由』で挙げたポイントそのもの。

統計関数

NumPyには、データの合計や平均など、便利な関数が組み込まれている。
これにより、大量のデータをすばやくまとめることができる。

```
# 配列の合計を求める
sum_array = np.sum(array1)
print(sum_array)  # 結果: 6
```

```
# 配列の平均を求める
mean_array = np.mean(array1)
print(mean_array)  # 結果: 2.0
```

❓ FAQ

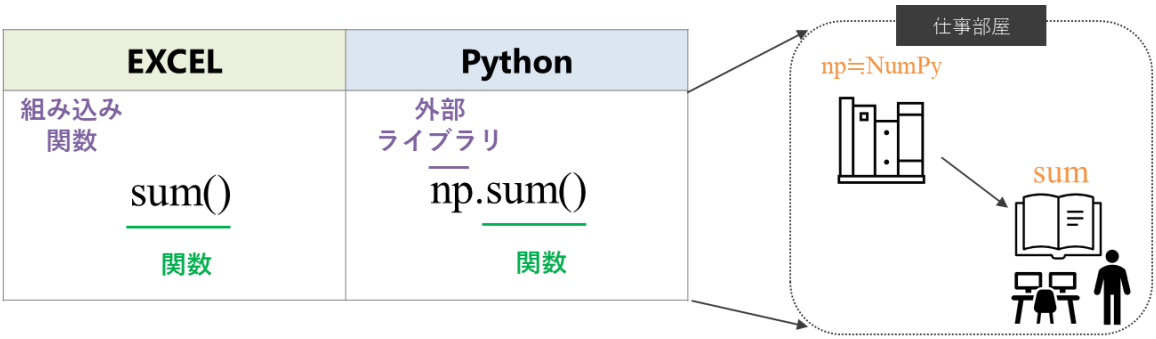
Q: Python では `np.sum`、Excelでは `sum` なのでしょうか？

A:

内部関数か外部ライブラリかの違いにより、記述が異なる。

`Excel では sum が組み込み関数`として提供されているため、そのまま使える

一方、 `np.sum` と記述するのは、 NumPy ライブラリの `sum` 関数を使用しているため。



ワーク

問題

1. **2次元配列の作成**
   `[[10, 15, 20], [25, 30, 35], [40, 45, 50]]` を使って、 `NumPy` の 2 次元配列を作成してください。

```
1 コーディングを開始するか、AI で生成します。
```

2. **2次元配列の要素の取り出し**
   先ほど作成した`Numpy`の 2次元配列 から、2行目の3列目の要素を取り出してください。

```
1 コーディングを開始するか、AI で生成します。
```

3. **範囲の取り出し（スライシング）**

先ほど作成した`Numpy`の 2次元配列 から、2行目までの全要素をスライスして取り出してください。

```
1 コーディングを開始するか、AI で生成します。
```

4. **配列同士の足し算**

配列 `[1, 2, 3]` と `[4, 5, 6]` を足し合わせた結果を求めてください。

```
1 コーディングを開始するか、AI で生成します。
```

5. **配列の合計**
   配列 `[10, 20, 30, 40, 50]` の全要素の合計を求めてください。

```
1 コーディングを開始するか、AI で生成します。
```

解答と説明

1. **2次元配列の作成**
   `[[10, 15, 20], [25, 30, 35], [40, 45, 50]]` を使って、 `NumPy` の 2 次元配列を作成してください。

```
1 array_2d = np.array([[10, 15, 20], [25, 30, 35], [40, 45, 50]])
2 print(array_2d)
```

2. **2次元配列の要素の取り出し**
   先ほど作成した`Numpy`の 2次元配列 から、2行目の3列目の要素を取り出してください。

```
1 array_2d = np.array([[10, 15, 20], [25, 30, 35], [40, 45, 50]])
2 print(array_2d[1, 2])
```

3. **範囲の取り出し（スライシング）**

先ほど作成した`Numpy`の 2次元配列 から、2行目までの全要素をスライスして取り出してください。

```
array_2d = np.array([[10, 15, 20], [25, 30, 35], [40, 45, 50]])
print(array_2d[:2, :])
```

4. **配列同士の足し算**

配列 `[1, 2, 3]` と `[4, 5, 6]` を足し合わせた結果を求めてください。

```
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
result = array1 + array2
print(result)
```

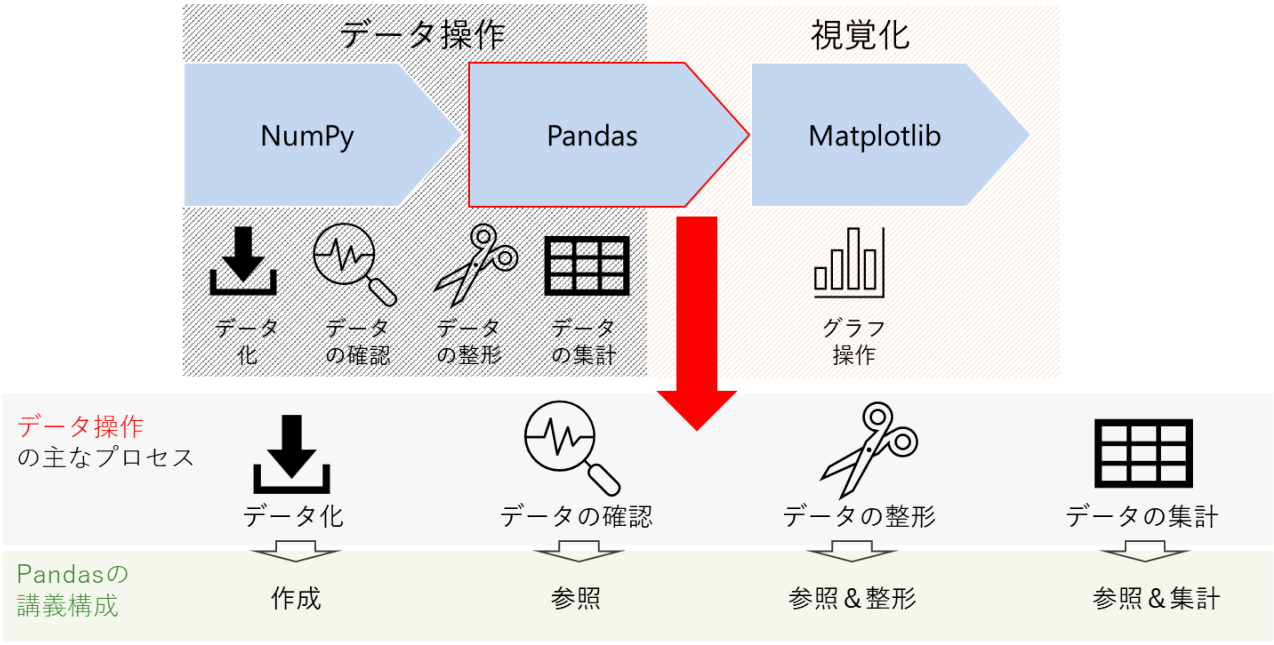
5. **配列の合計**
   配列 `[10, 20, 30, 40, 50]` の全要素の合計を求めてください。

```
array = np.array([10, 20, 30, 40, 50])
total_sum = np.sum(array)
total_sum
```

Pandasを学びましょう

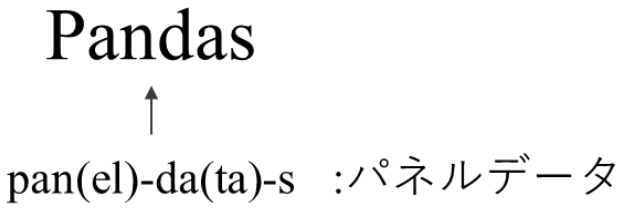
Pandas でも、数値データを「データ操作」する方法を理解する。

Pandas によるデータ分析の位置付けと講義範囲



Pandas とは？

Pandas のデータフレームを使うと、データを**表形式(テーブル形式)**で管理できる。



NumPy の配列とは異なり、Pandas のデータフレームを使うと、

データを**表形式（テーブル形式）**で管理できる。

Numpyの配列

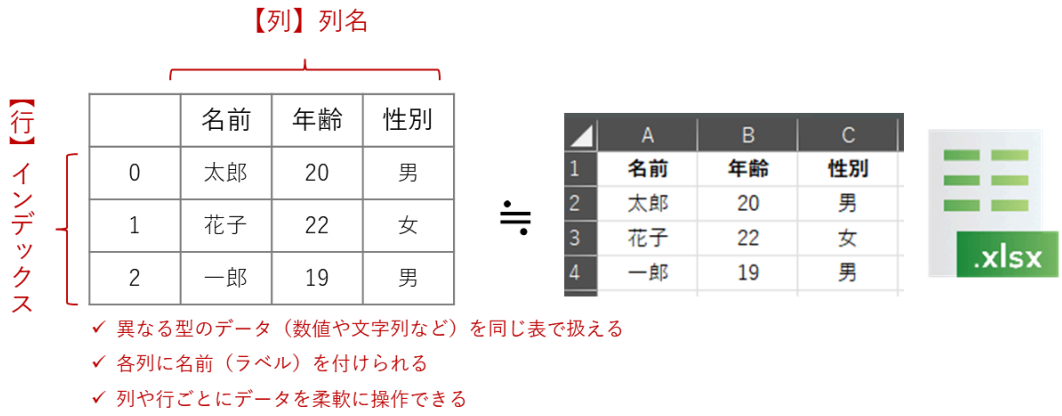
```
[['太郎' '20' '男']
 ['花子' '22' '女']
 ['一郎' '19' '男']]
```

Pandasのデータフレーム

	名前	年齢	性別
0	太郎	20	男
1	花子	22	女
2	一郎	19	男

表形式（テーブル形式）とは、行と列からなるデータ構造であり、 **Excel の表**のようなイメージ。

> **Excel に近い体験**を提供し、データ分析の初心者でも使いやすい



Pandas を使うための準備

(1) Pandas のインストール > (2) Pandas のインポート

1. Pandas のインストール

Google Colaboratory の環境下では、不要操作。

```
#pip install pandas   # コマンドラインやターミナルから実行する際に、推奨されている標準的なコマンドです
#!pip install pandas  # Jupyter Notebook や Google Colabratory などのコードセルから実行する場合に、推奨されているコマンドです
```

2. Pandas のインポート

```
import pandas as pd
```

データフレームの基本操作

データフレームは、Excel の表のように、

行と列で構成されるデータ構造のようなもの。

作成

データフレームを作成するには、次の2つの方法がある。

1. pd.DataFrame() を使う
2. CSV 等のデータ読み込み(自動的に、データフレーム生成)

手動でデータフレームを作成する

```
data = {'名前': ['太郎', '花子', '一郎'], '年齢': [20, 22, 19], '性別': ['男', '女', '男']}
df = pd.DataFrame(data)
df
```

📝 解説

---

1. 右辺について

   右辺は { '名前': ['太郎', '花子', '一郎'], '年齢': [20, 22, 19], '性別': ['男', '女', '男'] } 。

   これは **辞書型（dict）** と呼ばれる。

■■ 原則：辞書型 ■■`

辞書型：複数の情報を管理するデータ構造

'名前': ['太郎', '花子', '一郎']

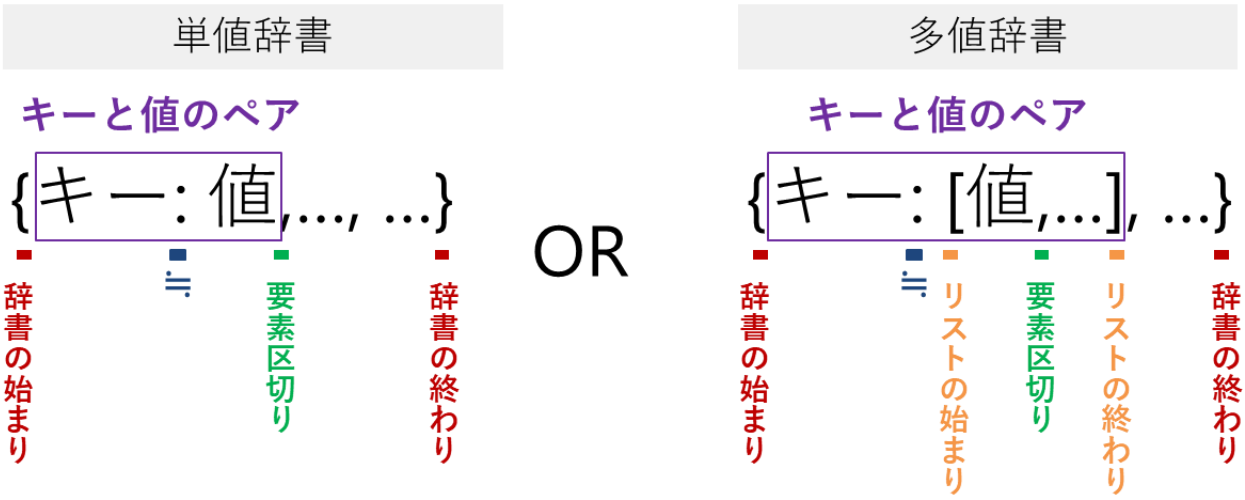
'年齢': [20, 22, 19]

'性別': ['男', '女', '男']

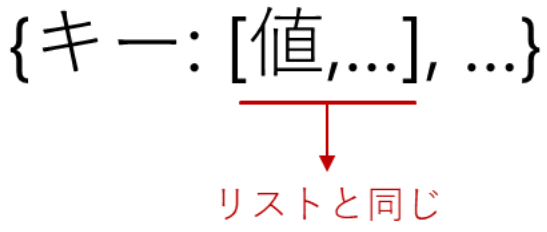
辞書（dict）型は以下の特徴は、**キーと値のペアでデータを管理**できること。

キーを指定して要素にアクセスし、様々な操作をする。

辞書の**構文**は次のようになる。



特に、多値辞書の場合は、リストが組み込まれていることに気づくだろうか？



リスト（単体）：

```
['いちご', 'ぶどう', 'メロン']
```

このリストにキーをつけ、{} でひとつの形にまとめると、辞書になる：

```
{
    'fruit': ['いちご', 'ぶどう', 'メロン']
}
```

このような形を見ておくと、値が 1 つだけの場合も理解しやすい。

値が 1 つだけならリストにする必要はなく、[] は省略できる：

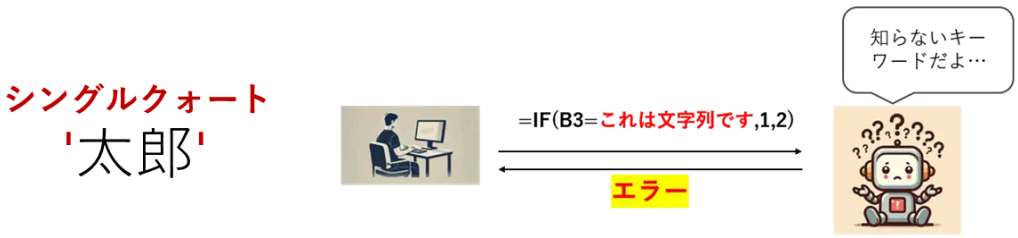
```
{
    'fruit': 'いちご'
}
```

2. シングルクォート

■■ 原則：シングルクォート ■■

シングルクォートで囲むと**「そのままのテキスト」**を表し、
囲まないと変数名やキーワードとして解釈され、変数に保存された値やキーワードの持つ特定の意味や動作を参照。

もし、**記述コードが変数名やキーワードに存在しない場合はエラー**する。



> ***なぜ、ダブルクォーテーション(")で囲む必要がある？***
>
> *PCに、ダブルクォーテーションで(")囲むことは「これは文字列です」と教えるための記号です。*
>
> *もしダブルクォーテーション(")で囲まなければ、Excelはこれらを文字列として理解できず、*
> *数値や数式として誤って処理する可能性があります。*
>
> *Excelの**予約語は、文字列を囲まなくてもエラーになりません**。*
> *これは次のセクションである「例外ルール」に直接関連しています。*
>
> *予約語とは、Excelがあらかじめ認識しているキーワードのことで、*
> *例えば「TRUE」「FALSE」や「SUM」などが該当します。*

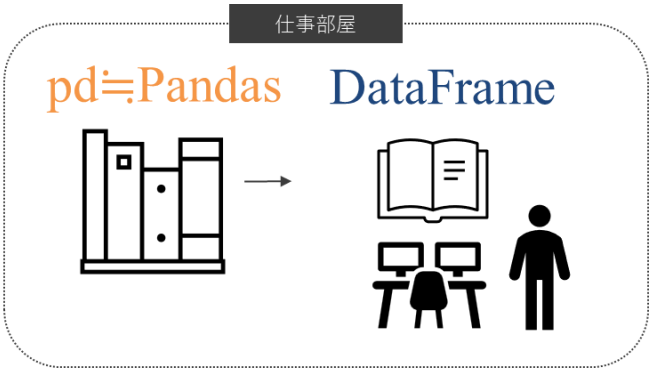
> EXCELの豆知識 -ダブルクォーテーション(")と例外ルールへの考察- | データ分析ドットコム.https://biz-data-analytics.com/excel/bits-of-excel-knowledge/9613/

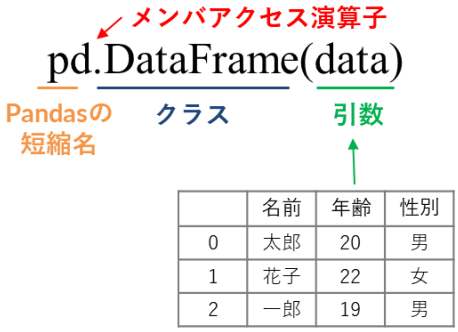
3. pd.DataFrame(data) に関して

Pands に属した DataFrameクラス を使って、

引数に指定した data (辞書型データ)に基づき、

新しい DataFrame オブジェクト(≒データフレーム)を生成する。





	名前	年齢	性別
0	太郎	20	男
1	花子	22	女
2	一郎	19	男

CSVファイルからデータフレームを作成する

```
sdata_url='https://www.salesanalytics.co.jp/wp-content/uploads/2024/11/data_jp_utf8.csv'
df = pd.read_csv(sdata_url)
df
```

📝 解説

---

1. `read_csv` に関して

pd.read_csv
自動的にDataFrameオブジェクトを生成

📌 Pandas の `read_csv` 関数によるデータフレーム作成

手動で `DataFrame` を作成する場合、
`df = pd.DataFrame(data)` のように
クラスを明示的に呼び出す必要があある。

しかし、`pd.read_csv()` を使えば、
`DataFrame` を明示せずに

> ✅ 実行するだけで、新しい`DataFrame`オブジェクトが自動生成

🔍 なぜ `DataFrame` を明示的に指定しなくてもよいのか？

📌 `pd.read_csv()` は、内部的に **`pandas.DataFrame` クラスのコンストラクタ** を使ってデータフレームを生成。
そのため、**明示的に `DataFrame` を指定しなくても、新しいデータ　フレームが作成** される。

🛠 コンストラクタとは？

> 🎯 *オブジェクトの生成時に自動実行される初期化処理

📌 日本語 CSV の読み込みトラブル解決ガイド

`CSV` を `Pandas` で読み込もうとすると、
以下のようなエラーが発生することがある。

🚨 **エラーの原因**

`read_csv` はデフォルトで **`UTF-8`** としてファイルを読み込むため。

📌 **エンコーディングの違い（概念図）**

❌ `UTF-8` で読み込もうとする → 解読できずエラー発生

```
┌──────────┐
| CSV      |  ← ❌ 読めない
└──────────┘
```

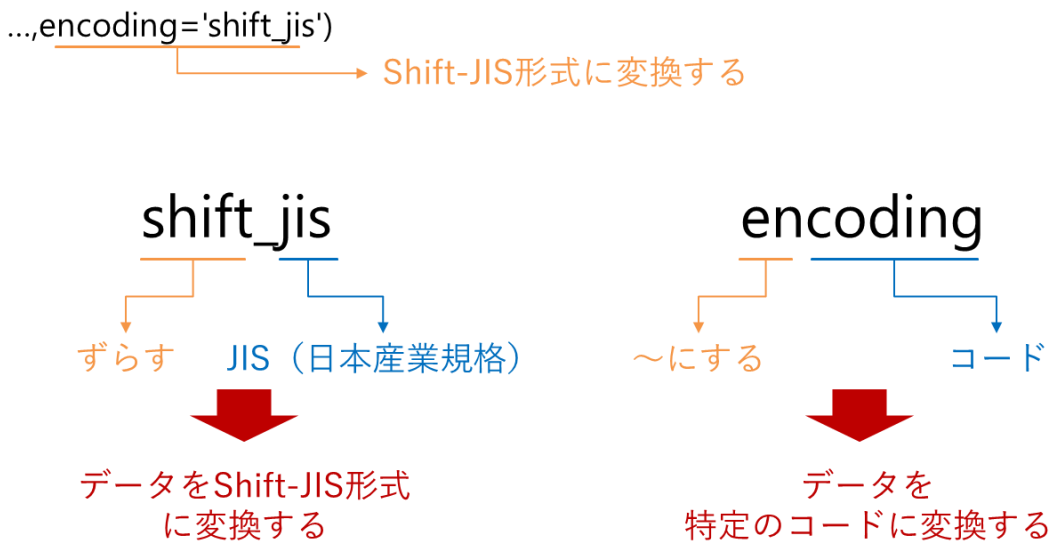
✅ `shift-jis` を指定すると → 正しくデータを取得できる

```
┌──────────┐
| CSV      |  ← ✅ 読める
└──────────┘
```

🛠 対策：エンコーディングを指定する

```python
# 日本語Windows環境で作成されたCSVファイルを読み込む
df_s = pd.read_csv('https://www.salesanalytics.co.jp/wp-content/uploads/2025/03/data_jp_shfit-jis.csv', encoding='shift_jis')

# 内容を確認
df_s
```



💾 メモ帳 でのファイル保存時のポイント

新しく 'CSV' ファイルを作成する場合、
「名前を付けて保存」の **エンコード** 設定で **`UTF-8`** を選択すると、 将来的な文字化けやエラーを防ぐことができる。

🔄 既存の CSV を UTF-8 に変換する手順

1. メモ帳で 'CSV' ファイルを開く
2. 「名前を付けて保存」を選択
3. 「エンコード」設定を UTF-8 に変更し、保存

■■ 原則：関数に複数の引数を渡す方法 ■■

関数に**複数の引数を渡す場合、それぞれの引数はカンマ（`,`）** で区切りる。

```
pd.read_csv( 'ファイル名.csv' , encoding='shift_jis' )
               |               |
               |               └── 文字コードを指定（encoding='shift_jis'）
```

└── 読み込むCSVファイル（'ファイル名.csv'）

参照

行列の参照

特定の行や列を参照して操作できる。

まずは、参照のお話。

`[]` により**参照**ことができる。

◎ 列を参照するケース

📜 記法

〇 1列の場合

`データフレーム['列名1']` と記述する

`データフレーム['列名']`

└── `'列名'`: 指定した列のデータを取り出す

```
1 # 列を選択
2 df['名前']
```

📝 解説

データフレーム `df` の中から `名前` 列を選択して出力する。

`df['名前']`

└── `名前``: 取得したいデータの列名を入力

〇 複数列を取り出すケース

📜 記法

`データフレーム[['列名1', '列名2']]`

↓

複数列の同時取り出し

データの取り出しの外側 `[]` ←データフレームから必要な部分（列または行）を取り出す

↑ ↑

`[['列名1', '列名2']]`

└──>内側 `[]` で、列名を指定

取得したいデータの列名をカンマで区切って指定

```
1 # 複数の列を選択
2 df[['名前', '年齢']]
```

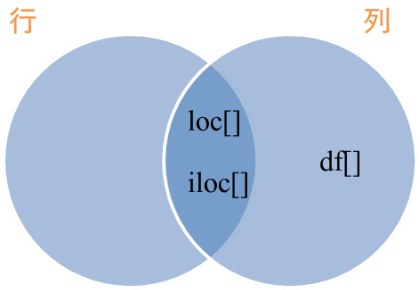
◎ 行を参照するケース

📝 解説

列ではなく、行を参照させたケースでは、`loc`、`iloc` を使用する必要がある。

- `loc`, `iloc` は **行と列の両方で指定**でき、柔軟。

💬 `df[...]` は本来列アクセス用だが、スライスのときだけ行アクセスにもなる。 それが例外的で、`.iloc` を使うべきとされている。



◎ `loc` と `iloc` を使う方法

loc

location(場所)

- ✓ ラベル（行・列の名前）でアクセス
- ✓ インデックスはラベル扱いなので注意

iloc

Integer(整数)
location(場所)

- ✓ 行番号・列番号でアクセス

⚠ 注意

`loc` は位置指定だけでなく名前やラベルでデータを選べるため、より便利で広く使われている。

○ 1行選択

`データフレーム.loc[行番号]` や `データフレーム.loc[行名]` で指定する。

`データフレーム.loc[行番号]`
　　└── 行番号: インデックス番号による指定行の取り出し

```
1 # 特定の行を選択
2 df.loc[0]  # 最初の行
```

📝 解説

データフレームのインデックス `0` の行（最初の行）を選択して出力する。

`df.loc[0]`
　└── インデックス `0` の行を取得

○ 複数行選択

データフレーム.loc[開始:終了] での範囲指定する。

データフレーム.loc[開始:終了]

└── 開始:終了 : 範囲指定による複数行の取り出し（開始から終了-1の行）

```
# 特定の行を選択
df.loc[1:2]
```

```
df[:]
```

📝 解説

データフレームのインデックス 1〜4 の行（2行目から5行目）をスライスして出力する。

`df.iloc[1:5]`

└── インデックス `1〜4` の行を取得

⚠️ ⚠️ Pandas の挙動に注意 ⚠️ ⚠️

データ構造によって、**スライスの挙動が異なる**。

- 始点や終点に指定できるものが異なる
- 終点が含まれるかどうかは、データ構造によって異なる

データ構造	スライス方法	終点の含まれ方	備考
Pythonリスト	始点:終点	含まれない	標準的なPythonスライス
Pandasデータフレーム（iloc）	iloc[始点:終点]	含まれない	Pythonスライスと同じ動作
Pandasデータフレーム（loc）	loc[始点:終点]	含まれる	**ラベルベースで、終点も含むのがPandas特有の仕様**
Pandasデータフレーム（df[始点:終点]）	df[0:3] など	含まれない	.iloc 相当の位置ベース処理として扱われる

――つまり、

Pandasデータフレーム（loc）のみ例外です。
df[0:3] のような一見曖昧な構文も、内部的には .iloc[0:3] として処理されるため、終点は含まれません。

終点の挙動に関しては、注意するしかないです。

┌──┐

例外： Pandas の loc のみ、終点が**含まれる**

└──┘

⚠ 注意

- 1列または1行のみを取り出すと、データは 1次元の `Series` 型 になる
- 複数列または複数行を取り出すと、データは 2次元の `DataFrame` 型 になる

😓 Series は 1次元のため、

後続の処理（例：機械学習モデルへの入力、結合、表形式前提の処理など）で

`DataFrame` を期待しているとエラーや不具合が発生することがある。

👉 安定した処理を行うためには、 たとえ1列や1行でも `[['列名']]` や `[[行番号]]` のように2次元で抽出することをおすすめする。

ケース	統一感	型
`[['列A', '列B']] → [['列A']]`	✅ 一貫性あり	DataFrame
`[['列A', '列B']] → ['列A']`	❌ 一貫性なし	Series（1次元）

データの先頭を参照

`head()` により、単純にデータフレームの先頭数行を参照する。

引数で行数を指定しない場合、**デフォルトで先頭5行**を表示する。

```
df.head()
```

	名前	年齢	性別
0	太郎	20	男
1	花子	22	女
2	一郎	19	男

head()

デフォルトで先頭5行

整形

欠損値の処理

- 欠損値のデータを削除

  `Pandasオブジェクト.dropna()` で欠損値を含む行を削除できる。


```
df_with_na = {
'名前': ['太郎', '花子', '一郎', '次郎', '三郎'],
'年齢': [20, None, 19, 23, 21],
'点数': [85, 92, None, 88, 90]
}
df_with_na = pd.DataFrame(df_with_na)
# 欠損値を含む行を削除
df_with_na = df_with_na.dropna()
df_with_na
```

- 欠損値を指定の値で埋める

  `Pandasオブジェクト.fillna(値)` で欠損値を、引数に指定した値で埋めることができる。


```
df_with_na = {
'名前': ['太郎', '花子', '一郎', '次郎', '三郎'],
'年齢': [20, None, 19, 23, 21],
'点数': [85, 92, None, 88, 90]
}
df_with_na = pd.DataFrame(df_with_na)
# 欠損値を特定の値で埋める
# 'value'に0を指定して、欠損値を0で埋める
df_with_na = df_with_na.fillna(value=0)
df_with_na
```

集計

データの要約統計量を把握する。

```
# 合計、平均、中央値、ユニークな顧客数、トランザクション数の計算
total_sales = df['年齢'].sum()
average_sales = df['年齢'].mean()
max_sales = df['年齢'].max()
min_customers = df['年齢'].min()

print(total_sales, average_sales, max_sales, min_customers)
```

📝 解説

---

df の 年齢 列を選択し、その列の値を合計する

df['年齢'].sum()

|

v

df の中から 年齢 列を選択

> SUM≒合計

他の主なメソッド：

sum ≒ 合計

▶ 指定された列の全行の値を加算して合計値を計算

mean ≒ 平均

▶ 指定された列の値の合計を行数で割った平均値を計算

max ≒ 最大値

▶ 指定された列の値を最大値を計算

min ≒ 最小値

▶ 指定された列の値を最小値を計算

shape ≒ サイズ

▶ 行数と列数、 shape[0] で行数を取得可能

など

ワーク

問題

1. **データフレームの作成**

data={'名前': ['太郎', '花子', '一郎', '次郎', '三郎'],'年齢':[20, 22, 19, 23, 21],'点数':[85, 92, 78, 88, 90]} を使って、Pandas のデータフレームを作成してください。

```
1 コーディングを開始するか、AI で生成します。
```

2. **データフレームの表示** 問題1で作成したデータフレームの最初の2行を表示してください。

```
1 コーディングを開始するか、AI で生成します。
```

3. **列の抽出** 問題1で作成したデータフレームから、 loc を使用して「名前」の列を選択して表示してください。

```
1 コーディングを開始するか、AI で生成します。
```

4. **行の抽出** 問題1で作成したデータフレームから、 loc を使用して3行目を取り出して表示してください。

```
1 コーディングを開始するか、AI で生成します。
```

5. **データの集計（平均）** 問題1で作成したデータフレームの「点数」列の**平均値**を計算してください。

```
1 コーディングを開始するか、AI で生成します。
```

6. **欠損データの処理（削除）**
   以下のデータを使ってデータフレームを作成し、 dropna() を使って**欠損値を含む行を削除**してください。

```
'名前': ['太郎', '花子', '一郎'] ,
'年齢': [20, None, 19] ,
'点数': [85, 92, None]
```

```
コーディングを開始するか、AI で生成します。
```

解答と説明

1. **データフレームの作成** data={'名前': ['太郎', '花子', '一郎', '次郎', '三郎'],'年齢':[20, 22, 19, 23, 21],'点数':[85, 92, 78, 88, 90]} を使って、Pandas のデータフレームを作成してください。

```
# データフレームを作成
import pandas as pd
data={'名前': ['太郎', '花子', '一郎', '次郎', '三郎'],
'年齢':[20, 22, 19, 23, 21],
'点数':[85, 92, 78, 88, 90]}
df=pd.DataFrame(data)
df
```

2. **データフレームの表示** 問題1で作成したデータフレームの最初の2行を表示してください。

```
# 最初の2行を表示
df.head(2)
```

3. **列の抽出** 問題1で作成したデータフレームから、loc を使用して「名前」の列を選択して表示してください。

```
# 複数列を抽出
df.loc[:, '名前']
```

4. **行の抽出** 問題1で作成したデータフレームから、loc を使用して3行目を取り出して表示してください。

```
# 2の行を抽出
df.loc[2]
```

5. **データの集計（平均）** 問題1で作成したデータフレームの「点数」列の**平均値**を計算してください。

```
# 点数の平均値を計算
mean_score = df['点数'].mean()
mean_score
```

6. **欠損データの処理（削除）**
   以下のデータを使ってデータフレームを作成し、dropna() を使って**欠損値を含む行を削除**してください。

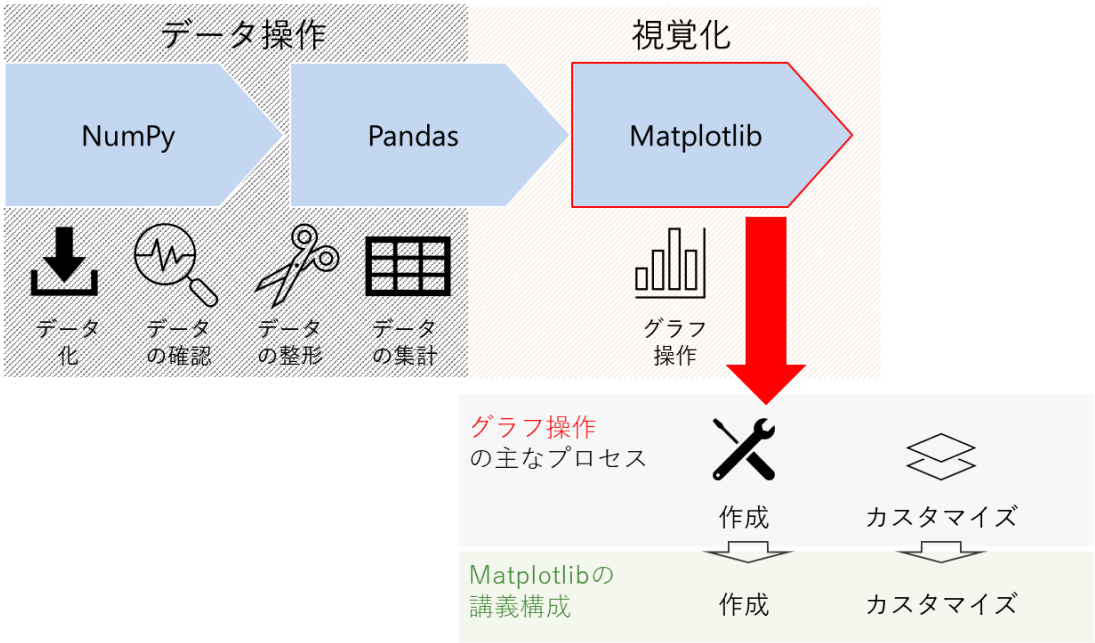
   ```
   '名前': ['太郎', '花子', '一郎'] ,
   '年齢': [20, None, 19] ,
   '点数': [85, 92, None]
   ```

```
# 欠損データを含むデータフレームを作成
data_with_na = {'名前': ['太郎', '花子', '一郎'],
'年齢': [20, None, 19],
'点数': [85, 92, None]}
df_na = pd.DataFrame(data_with_na)
# 欠損値を削除
cleaned_df = df_na.dropna()
cleaned_df
```

Matplotlibを学びましょう

Matplotlib を活用して、データを視覚的に表現する技術を習得する。

Matplotlibによるデータ分析の位置付けと講義範囲



Matplotlibとは？

その名から連想される通り、Matplotlibは、Pythonの**データ可視化ライブラリ**。

Matplotlibを使うための準備

(1) Matplotlibのインストール > (2) Matplotlibのインポート

1.Matplotlibのインストール

Google Colaboratoryの環境下では、不要操作。

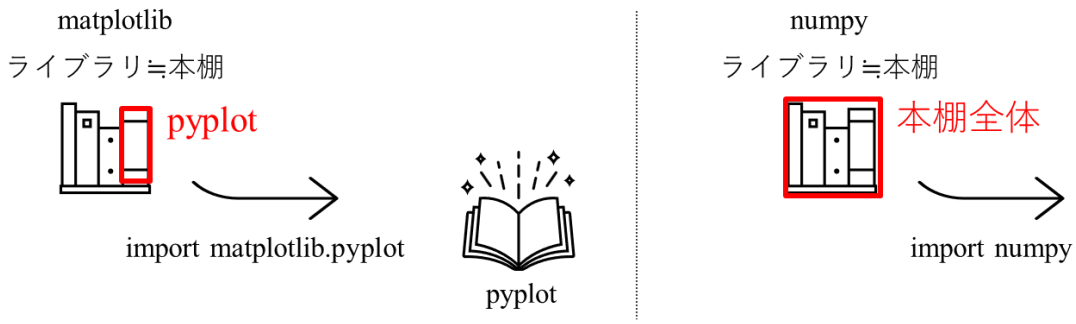
```
#pip install matplotlib   # コマンドラインやターミナルから実行する際に、推奨されている標準的なコマンドです
# !pip install numpy  # Jupyter Notebook や Google Colabratory などのコードセルから実行する場合に、推奨されているコマンドです
```

2.matplotlibのインポート

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

---

import matplotlib.pyplotと import numpyの違い




```
import numpy
```

|

▼

- **本棚そのもの（ライブラリ全体）を直接操作** する形式イメージ

  ※ `__init__.py` を読み込み、ライブラリ全体にアクセスできるため

```
import matplotlib.pyplot
```

|

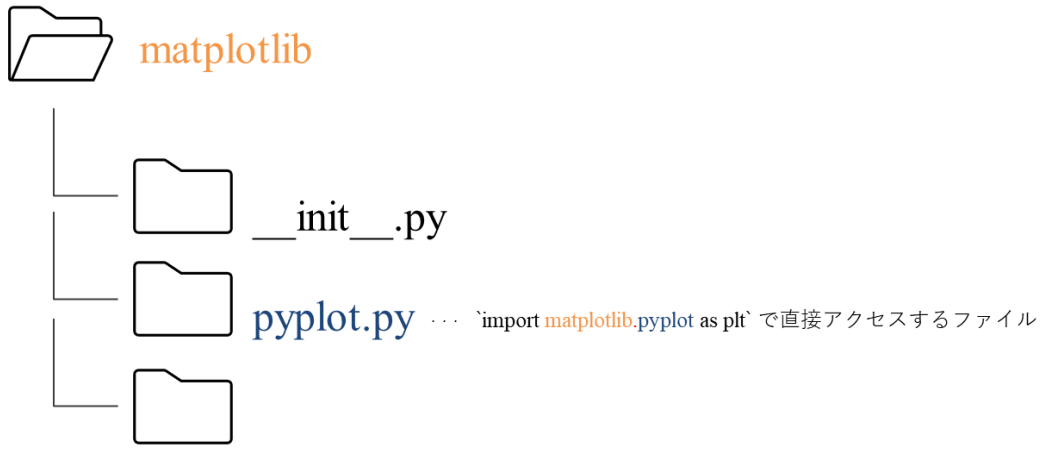
▼

- **`matplotlib` という本棚から `pyplot` という本**を取り出すイメージ
- **本棚の中の特定の本**（ライブラリの中の特定のモジュール）を指定 ※ `import` だけでは、全機能にアクセスできないため

  ```
  ライブラリ.モジュール
  ```

- `matplotlib.pyplot` のアクセスイメージ

  `matplotlib.pyplot` は `numpy` や `pandas` と違い、 `import` だけでは全機能にアクセスできず、 **`pyplot.py` を明示的に指定**する必要がある。



- `miniforge3` の場合

```
/miniforge3             # 例: miniforge仮想環境におけるmatplotlibのディレクトリ構造（環境によって異なる）
└── envs
    └── myenv           # 仮想環境の名前（例: myenv）
        └── lib
            └── pythonX.X  # Pythonバージョンに依存（例: python3.9）
                └── site-packages
                    └── matplotlib
                        ├── __init__.py
                        ├── pyplot.py        # `import matplotlib.pyplot as plt` のエントリーポイントとなるモジュール
```

```
├── cbook
│   ├── __init__.py
│   ├── deprecation.py
│   └── ...
├── axes
│   ├── __init__.py
│   ├── _axes.py
│   └── ...
├── figure.py          # グラフの全体を管理するクラス
└── style
    ├── __init__.py
    ├── core.py
```

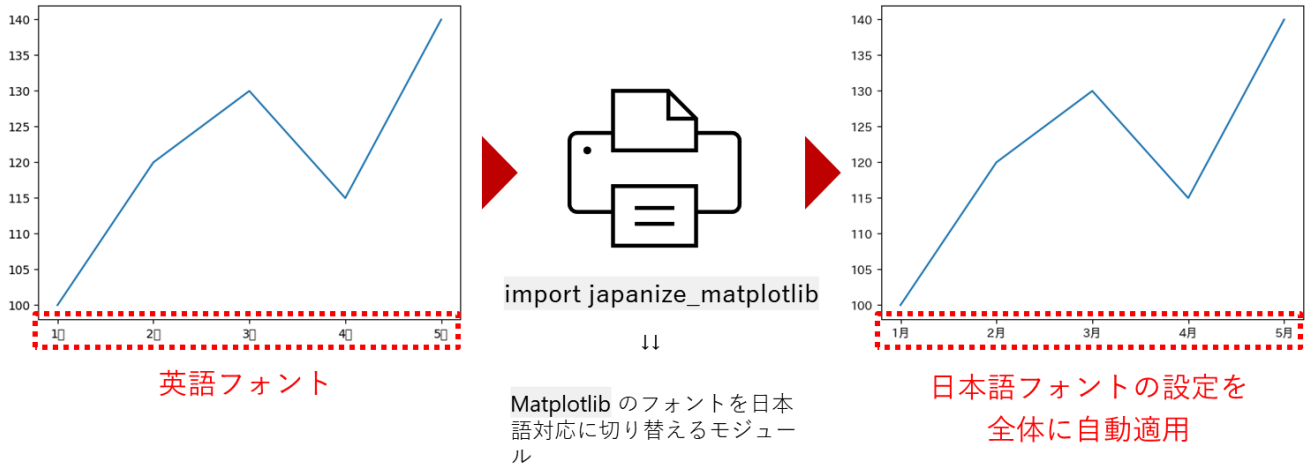
3. 日本語表示のための設定

   `Matplotlib` でグラフに日本語を表示するには、

   `japanize-matplotlib` をインストールし、インポートする必要がある。

```
# 必要なライブラリをインストール（インストール後はカーネル再起動を忘れないようにしてください。）
!pip install japanize-matplotlib
import japanize_matplotlib # 日本語フォントを自動設定
```

`import japanize_matplotlib` を実行すると、
日本語フォントの設定が**全体に**自動適用される。

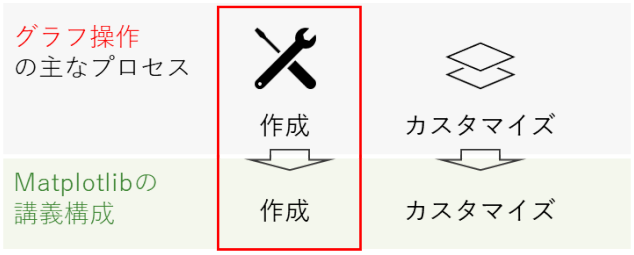


なお、`import japanize_matplotlib` では `as` を指定していません。
これは、`Numpy` や `Pandas` と異なり、**一度設定すれば全体に適用され、再利用の必要がない** ため。

グラフ操作

`Numpy` と `Pandas` などでデータ操作後に、`Matplotlib` でグラフ操作より、データを視覚的にする。

グラフ作成



折れ線グラフ

データの時間的な推移や連続するデータの変化を視覚化する。

```
# サンプルデータ
月 = ['1月', '2月', '3月', '4月', '5月']
売上 = [100, 120, 130, 115, 140]
# 折れ線グラフの作成
plt.plot(月,売上)
```

📝 解説

◎ `plt.plot()` 関数に関して

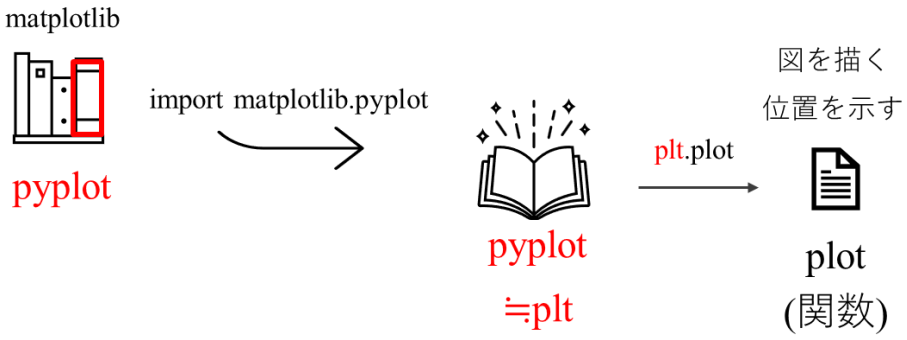
`plot()` は `plt` に属する関数。

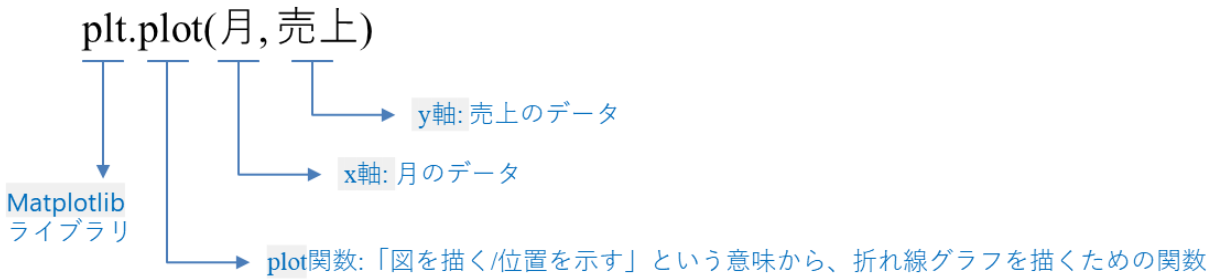
例えるなら、`pyplot` という本の1ページが `plot()` にあたり、

そこに記された「グラフを描く方法」を使い、

視覚化するイメージ。

編集するにはダブルクリックするか Enter キーを押してください





▌ `plt.plot()` の主な引数一覧表

例えば、`plt.plot(月, 売上, color='blue', linestyle='--', marker='o')` とすると、

青色の破線でデータ点に丸いマーカーを付けたグラフが表示される。

引数名	説明	指定例
x	横軸に用いるデータをリストまたは配列として指定	x=['1月', '2月', '3月']
y	縦軸に用いるデータをリストまたは配列として指定	y=[100, 200, 150]
color	線の色を指定	color='blue' color='red'
linestyle	線の種類を指定	linestyle='-' linestyle='--' linestyle=':'
linewidth	線の太さを数値で指定	linewidth=2
marker	各データ点の印を表す記号を指定	marker='o' marker='s'
label	凡例に表示される名称を文字列で設定	label='売上'
alpha	線の透明度を0から1の範囲で設定	alpha=0.5

■■ 原則：グラフ描画の仕組み ■■

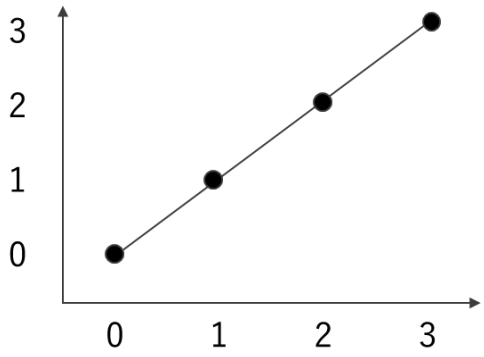
グラフは **すべて位置情報に基づいて描画** されます。

この仕組みは、折れ線グラフ・棒グラフ・横棒グラフなど、すべてに共通する。

例えば、先のコードを実行すると：

`['1月', '2月', '3月']` は、自動的に整数位置へ変換される。

```
1月    2月    3月
 ↓      ↓      ↓
 0      1      2
```



この整数位置（`0, 1, 2`）に基づいてデータが描画される仕組み。

なお、**数値リスト** を渡した場合、
そのままX軸の位置情報として解釈 される。

❓ X軸を省略すると？

X軸を指定せず、Y軸データのみを渡した場合はどうなるのか？

```
plt.plot([10, 20, 30])
```

`X軸` → 自動で `[0, 1, 2]` が割り当てられる
`Y軸` → 指定されたリスト `[10, 20, 30]`

いざ実行すると、

データ点を線で結ぶため、見かけ上は `[0, 1, 2]` には見えないが、

X軸には `[0, 1, 2]` が設定されているので問題ない。

棒グラフ

棒グラフは、カテゴリごとの値を比較するのに適する。

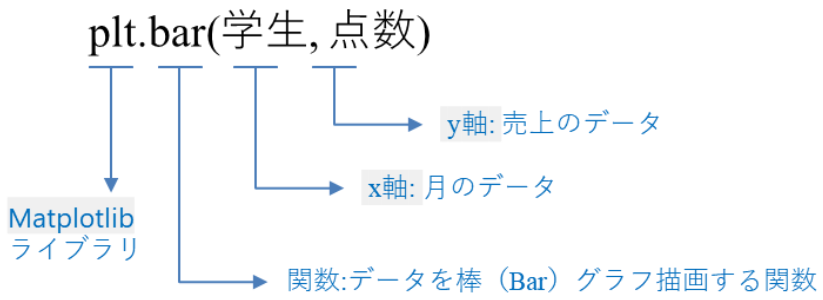
```
# サンプルデータ
学生 = ['太郎', '花子', '一郎', '次郎', '三郎']
点数 = [85, 92, 78, 88, 90]
# 棒グラフの作成
plt.bar(学生, 点数)
```

📝 解説

◎ `plt.bar()` 関数に関して

データを**棒（Bar）**の高さで表現するイメージ。

`plt.bar()` は、`Matplotlib` で棒グラフを描画するための関数。。



▌plt.bar() 関数の主な引数一覧表

例えば、plt.bar(学生, 点数, color='blue', width=0.5) とすると、

青色のバーが幅 0.5 で表示されるグラフが描画されます。

引数名	説明	設定例
x	横軸に表示するカテゴリデータを指定する	x=['1月', '2月', '3月']
height	縦軸に対応する値のリストや配列を指定する	height=[100, 200, 150]
color	各バーの色を指定する	color='blue' color='red'
edgecolor	バーの枠線の色を指定する	edgecolor='black' edgecolor='gray'
width	バーの幅を数値で指定する	width=0.5
align	バーの配置位置を文字列で指定する	align='center' align='edge'
label	凡例に表示するラベルを文字列で指定する	label='売上'
alpha	バーの透明度を0から1で指定する	alpha=0.7

円グラフ

データ全体に対する各部分の割合を表示する。

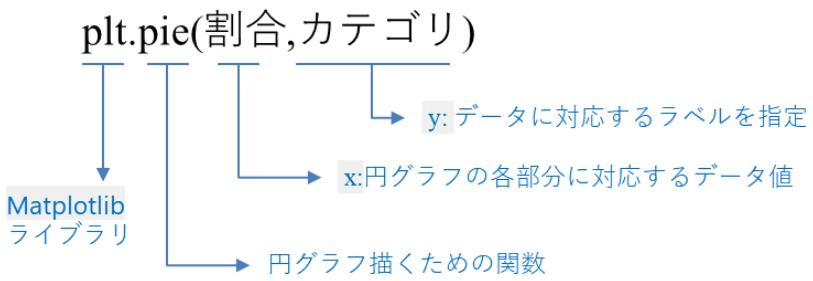
```
# サンプルデータ
カテゴリ = ['A', 'B', 'C', 'D']
割合 = [40, 30, 20, 10]
# 円グラフの作成
plt.pie(割合, labels=カテゴリ)
```

📝 解説

◎ plt.pie() 関数に関して

plt.pie() は、Matplotlib で円グラフを描くための関数。

見た目が「パイ（Pie）」に似ていることから付けられと言われている。



▌plt.pie() の主な引数一覧表

例えば、

```
plt.pie([40, 30, 20, 10], labels=['A', 'B', 'C', 'D'], autopct='%.1f%%', startangle=90, explode=[0, 0.1, 0, 0])
```

とすると、

A～Dのセグメントが割合付きで表示され、セグメントBが中心から少し離れた円グラフが描画される。

引数名	説明	設定例
x	各セグメントに対応するデータ値をリストまたは配列で指定する	x=[40, 30, 20, 10]
labels	各データに対応するラベルをリストで指定する	labels=['A', 'B', 'C', 'D']
colors	各セグメントの色をリストで指定する	colors=['red', 'blue', 'green']
autopct	割合を表示するためのフォーマット文字列を指定する	autopct='%.2f%%'
startangle	開始角度を数値で指定する	startangle=90
explode	特定のセグメントを中心から離すためのオフセットをリストで指定する	explode=[0, 0.1, 0, 0]
shadow	影を付けるかどうかを真偽値で指定する	shadow=True
radius	円の半径を数値で指定する	radius=1.5

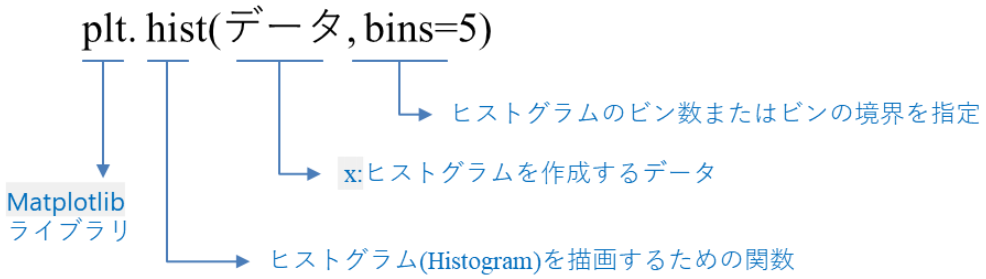
ヒストグラム

データの分布を視覚化する。

```
# サンプルデータ
データ = [10, 20, 20, 30, 40, 40, 40, 50, 60, 70]
# ヒストグラムの作成
plt.hist(データ, bins=5)
```

解説

◎ plt.hist() 関数



plt.hist() は、Matplotlib でヒストグラム(Histogram)を描画するための関数

```
`Histogram= histos（柱）　＋　gramma（記録）`
```

数値データの分布を示し、**値の範囲(bins)**ごとの頻度（度数）を表すグラフ。

---

bins (≒ 箱) とは

---

bins はデータを区切る **箱（範囲）** を意味し、

ヒストグラムで **データの分布を可視化する際の区切り方** を決定。

```
┌──────────┐  ┌──────────┐  ┌──────────┐
│          │  │          │  │          │
│   bin=1  │  │   bin=2  │  │   bin=3  │  ・・・
│          │  │          │  │          │
└──────────┘  └──────────┘  └──────────┘
```

データは bins で指定した数の箱に分割される。

多いと詳細が、少ないと全体像が見えやすくなる。

▌ plt.hist() の主な引数一覧表

例えば、plt.hist(データ, bins=10, color='blue', alpha=0.5) とすると、青色で透明度0.5のヒストグラムが表示される。

なお、**bins(ビンズ)は「箱」や「容器」**を意味し、データの範囲を区間ごとに分割する役割を持つ。

引数名	説明	設定例
x	ヒストグラムを作成するデータの配列を指定する	x=データ
bins	ビンの個数またはビン境界の配列を指定する	bins=10
range	データの範囲をタプルで指定する	range=(0, 100)
density	確率密度として表示するかどうかを真偽値で指定する	density=True
color	バーの塗りつぶし色を指定する	color='blue'
edgecolor	ビンの境界線の色を指定する	edgecolor='black'
alpha	バーの透明度を0から1で指定する	alpha=0.5
label	凡例に表示するラベルを文字列で指定する	label='データ分布'

散布図

散布図は、2つの変数の関係を示すのに適する。

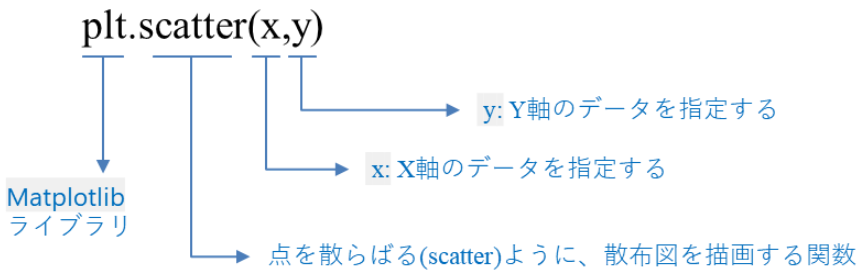
```
# サンプルデータ
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
# 散布図の作成
plt.scatter(x, y)
```

解説

◎ plt.scatter() 関数

plt.scatter() は、Matplotlib で**散布図**を描画するための関数。

データ点がグラフ上に**散らばる(scatter)**イメージ。



▌plt.scatter() の主な引数一覧表

例えば、plt.scatter(年齢, 収入, s=50, c='blue', alpha=0.5) とすると、青色の透明度0.5でサイズ50の散布図が表示される。

引数名	説明	設定例
x	横軸に用いるデータをリストまたは配列で指定する	x=年齢
y	縦軸に用いるデータをリストまたは配列で指定する	y=収入
s	各データ点のサイズを数値または配列で指定する	s=50
c	各データ点の色を指定する	c='blue'
cmap	色のマッピング方法を文字列で指定する	cmap='viridis'
marker	データ点のマーカーの形状を記号で指定する	marker='o'
alpha	データ点の透明度を0から1で指定する	alpha=0.5
label	凡例に表示するラベルを文字列で指定する	label='データ群'

複数の折れ線グラフの描画

複数の折れ線グラフを描画することができる。

```
# サンプルデータ
x =  ['1月', '2月', '3月', '4月', '5月']
y1 = [2, 4, 6, 8, 10]
y2 = [1, 3, 5, 7, 9]
# 複数のグラフを同じプロットに描画
```

```
6 plt.plot(x, y1)
```

■■ 原則：グラフ描画の仕組み ■■ ■■

`matplotlib` の関数は、`plt.show()` などを記述しない限り、

各コードは個別に動作しながらも状態を保持しながら

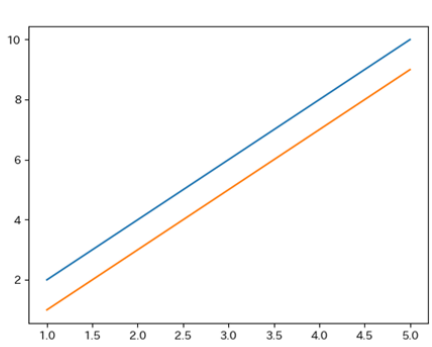
....最終的に1つの図としてまとめて表示される仕様。

――つまり、

同じ紙に新しい描画を追加していくイメージ。

タイトルや凡例も、該当コードを実行するたびに描画が追加され、

スケッチを重ねるように反映される。



plt.plot(x, y1)　・・・ 1回目の呼び出しで描画

plt.plot(x, y2)　・・・ 2回目の呼び出しで描画

したがって、下図は白い紙（≒ノートブックのコードセル）に、

`plt.plot()` を2回実行し、2つの折れ線グラフを描画。

◎ `plt.show()` とは

`show` は `plt` に属する関数。

グラフを描画(≒`show`)する機能。

```
1 plt.plot([1, 2, 3], [4, 5, 6])  # グラフを描画する
2 plt.show()        # 画面に表示する
```

しかし、

ノートブックでは、**`plt.show()` が記述せず**とも、**コードセル単位で描画**される。

(この違いを明瞭にするため、本セミナーでは敢えて `plt.show` を記述しないコードになっている)

```
1 plt.plot([1, 2, 3], [4, 5, 6])
```

積み上げグラフ

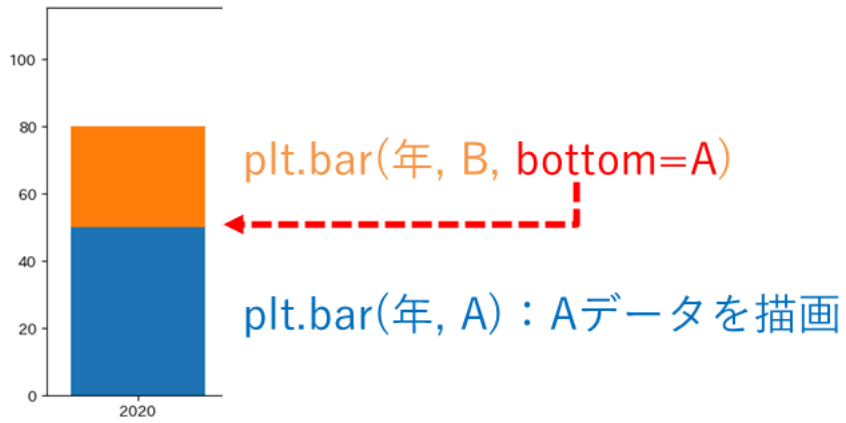
積み上げグラフは、各カテゴリの合計を表し、データの内訳を視覚化。

```
1   # サンプルデータ
2   年 = ['2020', '2021', '2022']
3   A = [50, 60, 70]
4   B = [30, 35, 40]
5
6   # 積み上げグラフの作成
7   plt.bar(年, A)
8   plt.bar(年, B, bottom=A)
```

📝 解説

先程のメカニズムに基づき、`plt.bar` を2回呼び出して、2つの棒グラフを描画しているイメージ。

ただ、`bottomm=A` という引数で、描画位置を調整。



`plt.bar()` を使用して縦向きの積み上げ棒グラフを作成。 それぞれのデータを積み上げる際に、 `bottom` 引数を使って開始位置を設定。

```
年 = ['2020', '2021', '2022']
A = [50, 60, 70]
B = [30, 35, 40]
plt.bar(年, A, label='A')  # Aのデータを表示
plt.bar(年, B, bottom=A, label='B')  # BのデータをAの上に積み上げる
```

- **`plt.bar`**：縦向きの棒グラフ（バーグラフ）の作成関数。
  - **第1引数（年）**：X軸上の位置指定
    - 例： `年 = ['2020', '2021', '2022']` の場合、X軸上では順に `0, 1, 2` の整数が自動割当。グラフ上での表示順に対応。
- **第2引数（B）**：各年に対する棒の高さ。
  - 例： `B = [30, 35, 40]` の場合、対応する位置における棒の高さは `B` の値に準拠。
- **`bottom=A`**：`B` の棒グラフの下端（基準位置）
  - `A` を基準とする積み上げ。
  - 例： `A = [50, 60, 70]` の場合
    - `2020` ：高さ `50` を下端とする棒
    - `2021` ：高さ `60` を下端とする棒
    - `2022` ：高さ `70` を下端とする棒
  - `bottom` の指定による積み上げ構造。
- **`label='B'`**：凡例に表示されるラベル名

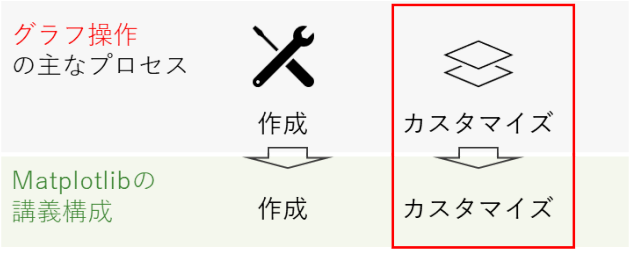
⚠️ 積上グラフの分析上の注意

積み上げグラフを使う際には、 積み上げるデータの意味が明確である必要がある。

積み上げるべきデータは、 **異なるカテゴリーが合算して全体を構成する場合**である。

グラフのカスタマイズ

`Matplotlib` を使ってグラフを作成した後、タイトルや軸ラベル、色などを変更してグラフを見やすくすることができる。



タイトルと軸ラベルの追加

グラフには、**タイトル**や**軸ラベル**を追加することで、

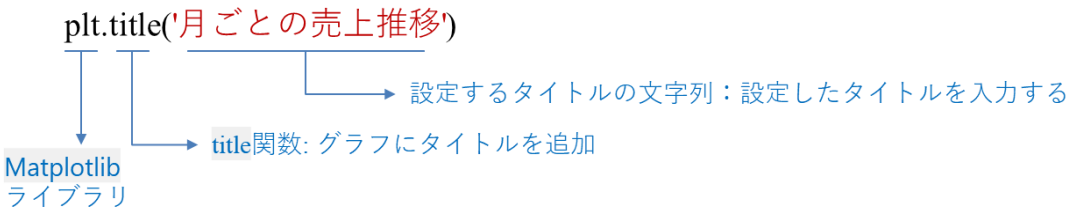
何を表しているのかをわかりやすくできる。

```
# サンプルデータ
月 = ['1月', '2月', '3月', '4月', '5月']
売上 = [100, 120, 130, 115, 140]
# 折れ線グラフの作成
plt.plot(月, 売上)
plt.title('月ごとの売上推移')
plt.xlabel('月')
plt.ylabel('売上（万円）')
```

📝 解説

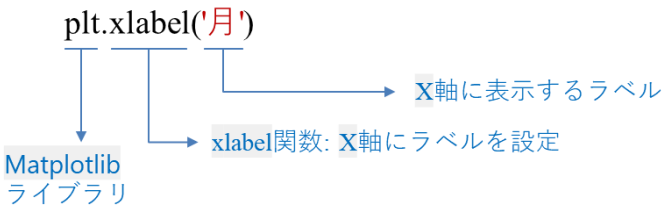
- `plt.title('月ごとの売上推移')` に関して

  `title`(見出し)の通り、グラフ全体の**「タイトル」**を設定。



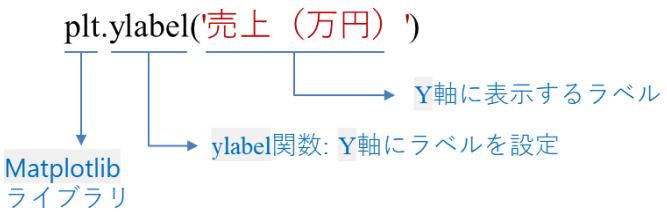
- `plt.xlabel('月')` に関して

  `xlabel` ≒ `x` + `label` の語源の通りに、**X軸(横軸)にラベル(`label`)**を設定。



- `plt.ylabel('売上（万円）')` に関して

  `ylabel` ≒ `y` + `label` の語源の通りに、**Y軸(縦軸)にラベル(`label`)**を設定。



色やスタイルの変更

`plt.plot` の引数設定を工夫するだけで、

グラフの**色**や**線のスタイル**を変更して、

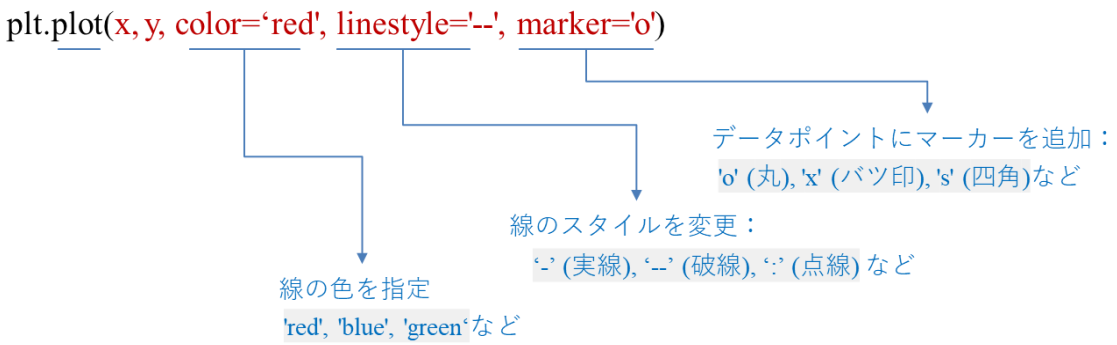
見た目をよりわかりやすくしたり、

複数のデータを区別しやすくすることができる。

```
# 折れ線グラフの色とスタイルを変更
plt.plot(月,売上,color='green',marker='o',linestyle='--')
```

📝 解説

- `plt.plot(月, 売上, color='green', linestyle='--', marker='o')` に関して
  `plot関数` の引数を指定し、グラフカスタマイズします。



💬 引数(材料)を変えれば、料理もマイナーチェンジ(白米、炊き込みご飯など)できる。

▌引数の一例

- **`color`** : 線の色を指定
  - 例: `'red'`, `'blue'`, `'green'`
- **`linestyle`** : 線のスタイルを変更
  - 例: `'-'` (実線), `'--'` (破線), `':'` (点線)
- **`marker`** : データポイントにマーカーを追加
  - 例: `'o'` (円), `'s'` (四角), `'D'` (ダイヤ)

❓ Python 関数は、どのように引数を認識しているのか？

```
引数（Arguments） ──┬── 1. 位置引数（positional arguments）
                    │
                    └── 2. キーワード引数（keyword arguments）
```

1. **位置引数（Positional Arguments）**
   - 名前を持たず、順序に基づいて関数に渡される引数
   - 順序が重要で、正しい順序で指定する必要あり
     例えば、以下の `年` と `C` を指す。

     ```
     plt.bar(年,C,...)
     ```

2. **キーワード引数（Keyword Arguments）**
   - `名前=値` の形式で指定する引数
   - 順序は関係なく、任意の順序で指定可能
     例えば、以下の `label='A'` を指す。

     ```
     plt.bar(年,C,label='A')
     ```

<details>
<summary>凡例の追加</summary>

複数のデータセットを1つのグラフに表示する場合、

凡例（データのラベル）を追加して、
</details>

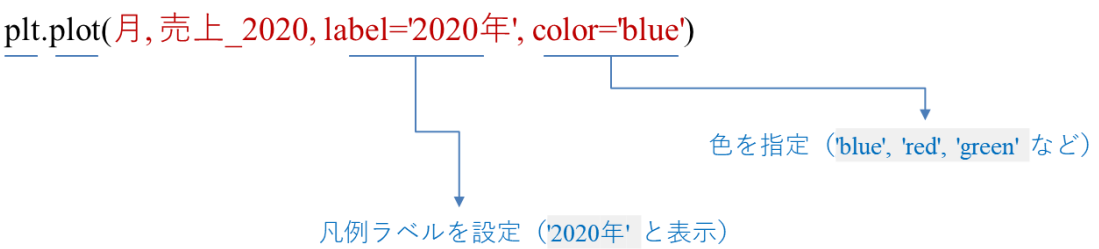
どの線がどのデータを表しているかを分かりやすくする。

```
# サンプルデータ
売上_2020 = [100, 120, 130, 115, 140]
売上_2021 = [110, 125, 135, 120, 145]
# 複数の折れ線グラフを作成
plt.plot(月, 売上_2020, label='2020年', color='blue')

# 凡例の追加
plt.legend()
```

📝 解説

- `plt.plot(月, 売上_2021, label='2021年', color='orange')` に関して

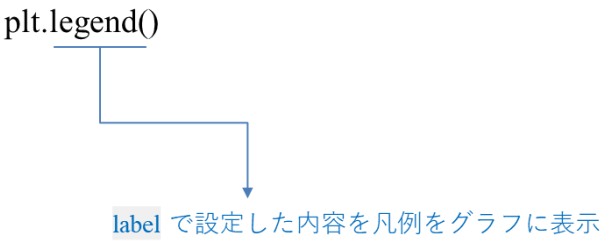


- `plt.legend()` に関して

  `label` で凡例の内容を決め、`plt.legend()` で表示する仕組み。

  ```
  label ──────→ plt.legend() でグラフ上に表示
  ```

  `legend` には**伝説**や**物語**といった意味が生まれ、さらに地図や図表の**凡例**を指すようになったと言われている。



Y軸の制約を設定

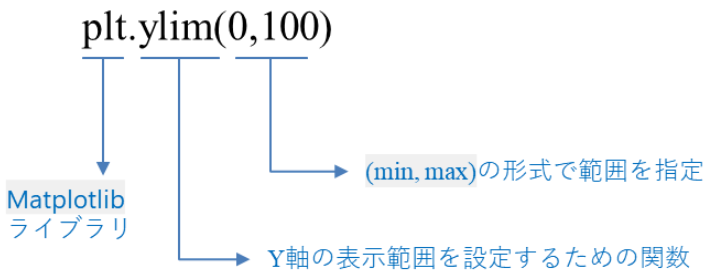
`ylim` で制約を設定し、同一スケールで適切に比較でき、視覚的な誤解を避けられる。

特に、**競合比較や時系列分析** などで有用。

```
plt.ylim(0, 100)
```

📝 解説

- `plt.ylim(0, 100)` に関して
  - **Y軸の表示範囲**を設定するための関数
  - `(min, max)` の形式で範囲を指定



`ylim` は、`y`（Y軸）と `limit`（制限、範囲）を組み合わせた名称。

```
ylim ──┬── y（Y軸）
       │
       └── limit（制限、範囲）
```

ワーク

問題

1. **複数の折れ線グラフの作成**

月ごとの売上データを使って、2つのデータセットを1つのグラフに表示してください。それぞれのデータに対して、「2020年」と「2021年」の凡例を付けてください。月のラベルは1月から5月までとします。

データ:

- `月 = ['1月', '2月', '3月', '4月', '5月']`
- `売上_2020 = [100, 120, 130, 115, 140]`
- `売上_2021 = [110, 125, 135, 120, 145]`

```
1  コーディングを開始するか、AI で生成します。
```

説明:

2. **円グラフの作成**
   カテゴリごとの割合 `カテゴリ = ['A', 'B', 'C', 'D']`、`割合 = [40, 30, 20, 10]` を使って、円グラフを作成してください。

```
1  コーディングを開始するか、AI で生成します。
```

3. **ヒストグラムの作成**
   `データ = [10, 20, 20, 30, 40, 40, 40, 50, 60, 70]` を使って、ヒストグラムを作成してください。

```
1  コーディングを開始するか、AI で生成します。
```

4. **散布図の作成**
   2つの変数 `x = [1, 2, 3, 4, 5]`、`y = [2, 4, 6, 8, 10]` を使って、散布図を作成してください。

```
1 コーディングを開始するか、AI で生成します。
```

5. **グラフのカスタマイズ**

次のデータをもとに、折れ線グラフを作成し、以下の要素を追加して見やすくカスタマイズしてください。

データ

- 月：`['1月', '2月', '3月', '4月', '5月']`
- 売上：`[100, 120, 130, 115, 140]`

カスタマイズの要件

- **線の色**：緑

- **線のスタイル**：破線
- **マーカー**：丸
- **グラフのタイトル**：適切なタイトルを追加
- **X軸ラベル**：月
- **Y軸ラベル**：売上

```
コーディングを開始するか、AI で生成します。
```

解答と説明

なお、問題1〜4の解答例ではグラフのカスタマイズが行われていますが、必須ではありません。

1. **複数の折れ線グラフの作成**

月ごとの売上データを使って、2つのデータセットを1つのグラフに表示してください。それぞれのデータに対して、「2020年」と「2021年」の凡例を付けてください。月のラベルは1月から5月までとします。

データ:

- `月 = ['1月', '2月', '3月', '4月', '5月']`
- `売上_2020 = [100, 120, 130, 115, 140]`
- `売上_2021 = [110, 125, 135, 120, 145]`

説明:

```
# サンプルデータ
月 = ['1月', '2月', '3月', '4月', '5月']
売上_2020 = [100, 120, 130, 115, 140]
売上_2021 = [110, 125, 135, 120, 145]
# 複数の折れ線グラフの作成
plt.plot(月, 売上_2020, label='2020年')
plt.plot(月, 売上_2021, label='2021年')
plt.legend()
```

2. **円グラフの作成**
   カテゴリごとの割合 `カテゴリ = ['A', 'B', 'C', 'D']`、`割合 = [40, 30, 20, 10]` を使って、円グラフを作成してください。

```
# サンプルデータ
カテゴリ = ['A', 'B', 'C', 'D']
割合 = [40, 30, 20, 10]
# 円グラフの作成
plt.pie(割合, labels=カテゴリ)
```

3. **ヒストグラムの作成**
   `データ = [10, 20, 20, 30, 40, 40, 40, 50, 60, 70]` を使って、ヒストグラムを作成してください。

```
# サンプルデータ
データ = [10, 20, 20, 30, 40, 40, 40, 50, 60, 70]
# ヒストグラムの作成
plt.hist(データ, bins=5)
```

4. **散布図の作成**
   2つの変数 `x = [1, 2, 3, 4, 5]`、`y = [2, 4, 6, 8, 10]` を使って、散布図を作成してください。

```
# サンプルデータ
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
# 散布図の作成
plt.scatter(x, y)
```

5. **グラフのカスタマイズ**

次のデータをもとに、折れ線グラフを作成し、以下の要素を追加して見やすくカスタマイズしてください。

データ

- 月：`['1月', '2月', '3月', '4月', '5月']`
- 売上：`[100, 120, 130, 115, 140]`

カスタマイズの要件

- **線の色**：緑

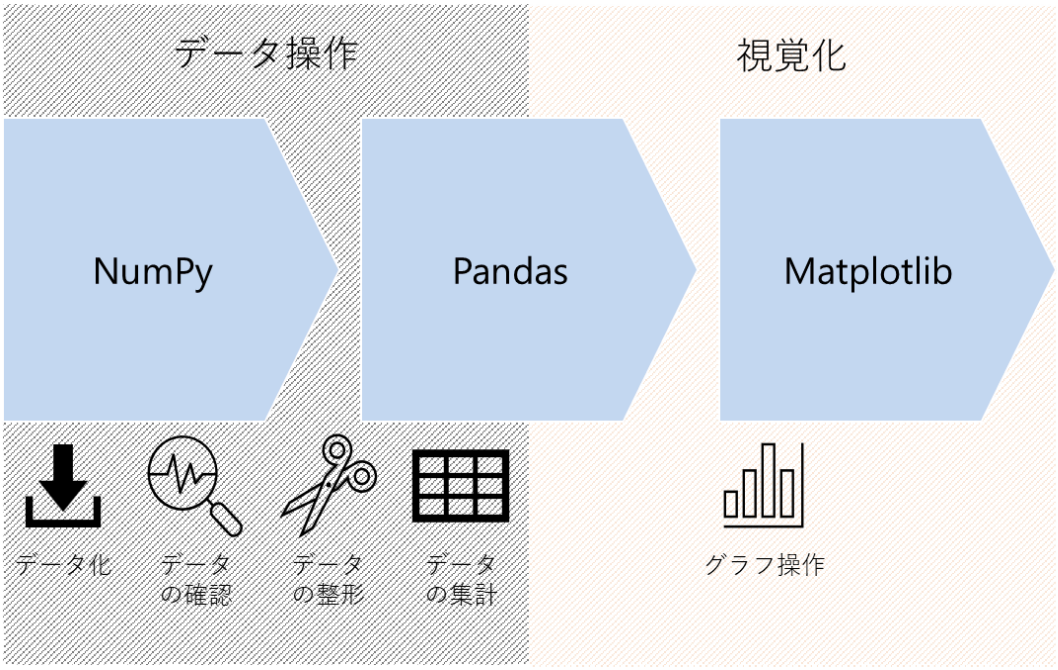
- **線のスタイル**：破線
- **マーカー**：丸
- **グラフのタイトル**：適切なタイトルを追加
- **X軸ラベル**：月
- **Y軸ラベル**：売上

```
# データ
月 = ['1月', '2月', '3月', '4月', '5月']
売上 = [100, 120, 130, 115, 140]

# 折れ線グラフの作成
plt.plot(月, 売上, color='green', linestyle='--', marker='o')
# タイトルと軸ラベルの追加
plt.title('月ごとの売上推移')
plt.xlabel('月')
plt.ylabel('売上（万円）')
```

おまけ

NumPy、Pandas、Matplotlibの再整理



NumPy

数値データを効率的に処理するためのライブラリ。

- 用途
  - リストや配列を用いた足し算、掛け算などの**高速な数値計算**が可能
- 特長
  - 軽量で数値データに特化
- 制約
  - ラベル付きデータや**柔軟なデータ操作**には適さない

Pandas

表形式のデータ（例：CSVやExcel）を扱うの強いライブラリ。

- 用途
  - **データの読み込み、整理、加工、集計、可視化**が可能
- 特長
  - 構造化データの操作に優れ、**Excelのような柔軟なデータ操作**を実現

- 制約
  - **処理速度が低下**する場合がある
  - 基本的な可視化機能しかない

▌Matplotlib

データを視覚的に表現するためのグラフ作成ライブラリ。

- 用途
  - **グラフを作成**が可能
- 特長
  - Pandas や NumPy で整理・計算したデータを用いて、結果を視覚的に伝える
- 制約
  - 一部、対応できない可視化がある

関数、メソッド、プロパティの違い

関数は独立し、メソッドとプロパティはオブジェクトに属する

np.array はメソッドのように見えますが、実際には NumPy の独立した関数であり、メソッドではない。

メソッドは引数を取らないが、プロパティは取らない。

チャレンジ穴埋め問題

【問題】

次の文章【(1)〜(13)】の空欄に入れるべき語句を、【選択肢】から選んでください。

1. Python でコメントアウトを行う際、コードの実行から除外するために行の先頭に記述する記号は【(1)】である。
2. Python のリストのスライスは、角括弧内に「開始:終了」と記述し、開始位置から【(2)】の要素を参照する記法であり、これは【(3)】と呼ばれる。
3. import numpy as np」と記述することで、NumPy ライブラリを短縮名【(4)】として利用できる。
4. 関数とは、入力となる【(5)】を受け取り、特定の処理を実行して【(6)】を返す仕組みである。
5. 【(7)】はオブジェクトに属します。たとえば、Pandas の DataFrame が持つ loc はその一例であり、オブジェクトの【(8)】は、その状態や属性（例：shape）を示す。
6. Python のリストは、角括弧【(9)】で囲み、各要素は【(10)】で区切って記述する。

7. Pandas では、DataFrame の特定の行や列へアクセスする際、ラベル指定には【(11)】を用い、位置指定の場合は【(12)】を使用する（ラベル指定と位置指定では動作が異なる）。
8. Matplotlib のグラフ描画は、コードセル内で関数を複数回呼び出すと、前回までの描画状態が保持され、同じ図に新たな描画が【(13)】される仕様となっている。

【選択肢】

A. # B. 終了位置の直前まで
C. スライス
D. np
E. 引数
F. 戻り値
G. メソッド
H. プロパティ
I. []
J. ,
K. loc
L. iloc
M. 追加

【回答】

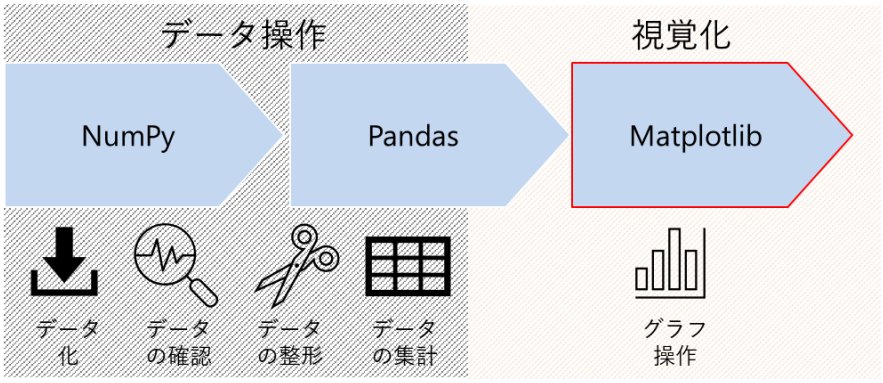
---

1. Python でコメントアウトを行う際、コードの実行から除外するために行の先頭に記述する記号は【'A. #'】である
2. Python のリストのスライスは、角括弧内に「開始:終了」と記述し、開始位置から【'B. 終了位置の直前まで】の要素を参照する記法であり、これは【'C. スライス'】と呼ばれる
3. 「import numpy as np」と記述することで、NumPy ライブラリを短縮名【'D. np'】として利用できる
4. 関数とは、入力となる【'E. 引数】を受け取り、特定の処理を実行して【'F. 戻り値】を返す仕組みである
5. 【'G. メソッド'】はオブジェクトに属します。たとえば、Pandas の DataFrame が持つ loc はその一例であり、オブジェクトの【'H. プロパティ'】は、その状態や属性（例：shape）を示す
6. Python のリストは、角括弧【'I. []'】で囲み、各要素は【'J. ,'】で区切って記述する
7. Pandas では、DataFrame の特定の行や列へアクセスする際、ラベル指定には【'K. loc'】を用い、位置指定の場合は【'L. iloc'】を使用する（ラベル指定と位置指定では動作が異なる）
8. Matplotlib のグラフ描画は、コードセル内で関数を複数回呼び出すと、前回までの描画状態が保持され、同じ図に新たな描画が【'M. 追加】される仕様となっている

フィナーレ

NumPy、Pandas、Matplotlib といったデータサイエンスに欠かせないライブラリの基本操作を通じながら、基礎体力を養うことを目標にする。

ハイライト



1. はじめに
    - Python がデータ分析に普及している理由

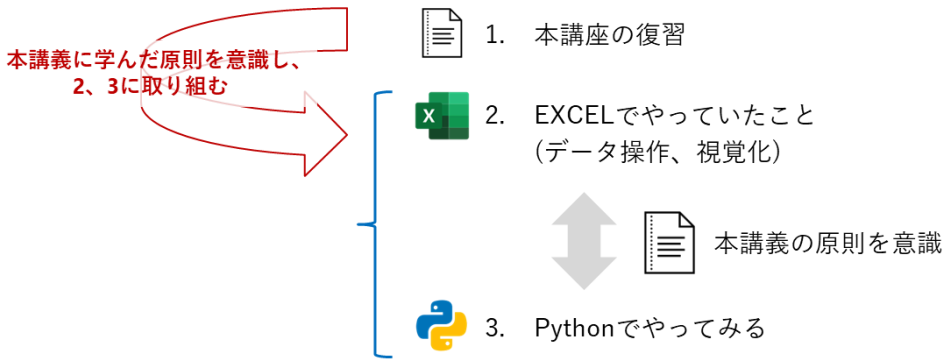
    - `NumPy`, `Pandas`, `Matplotlib` の個別の役割とデータ分析との関連性
    - `Google Colaboratory` の環境構築のデモを含む基礎知識
2. `NumPy` による数値計算の基礎
    - 講義：配列の作成、取り出し、演算を含む基本操作
    - ワーク: 二次元配列の操作、配列同士の足し算
3. `Pandas` によるデータ操作の基礎
    - 講義：作成と読み込み、取り出し、集計
    - ワーク: 集計や欠損データの処理
4. `Matplotlib` によるグラフ描画の基礎
    - 講義：グラフの作成、カスタマイズのテクニック
    - ワーク: 折れ線グラフ、円グラフ、散布図など

▼

これらを通じて、以下のことを学習しました：

- `Python` のデータサイエンスにおける重要性
- 今後に利用できる `Python` が操作できる `Google Colaboratory` の環境構築
- `NumPy`、`Pandas`、`Matplotlib` といったデータサイエンスに欠かせないライブラリの基本操作
- 講義を通じて、変数、オブジェクト、関数などの `Python` の基礎仕組みを学ぶ
- 講義とワークショップの両面による理解を深める学習体験
- あるあるエラーの防止策についても触れ、初心者がつまずきやすいポイントを克服

ネクストステップ



まず、**本講義の復習**である。

次は、`Excel` で**既に行ったデータ操作や視覚化をPythonで実現**することに取り組む。

その際、本講義で**学んだ原則に立ち戻り**ながら、やりたい操作に該当する`numpy`、`pandas`、`matplotlib`などのライブラリのメソッドやプロパティを調べてみる。

このプロセスを繰り返すことで、確実にスキルが身につく。

また、新しいライブラリや手法に興味を持つことも自然な流れである。

ただし、まずは**今回学んだ内容を確実に自分のもの**にすることが大切。

基礎がしっかり固まっていれば、 次のステップに進む際もスムーズに成長することができる。

この取り組みを通じて、学んだことを実践し、自分の力に変えていく。