はじめに

セミナーの進め方

- ご質問に関して
  - **講義中**でも可能
    - **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
      - **すべてのルールやパターンを網羅していない**
  - 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。

目的

`Python`でデータ分析を行う上で、 `NumPy`、`Pandas`、`Matplotlib` は、いわばデータ分析の三種の神器です。

```
データ分析の三種の神器

┌──────────────┐   ┌──────────────┐   ┌──────────────┐
│ 🔢 NumPy     │   │ 🐼 Pandas    │   │ 📊 Matplotlib │
└──────────────┘   └──────────────┘   └──────────────┘
```

その中で、`Pandas`を使用してデータ分析の基本を学ぶことを目的とします。

データの読み込みや加工、基本的な集計処理など、`Pandas`を活用したデータ操作のスキルを習得できる内容です。

講義は`Python`の基礎を軽く押さえてから、**データ分析の流れ**に沿って、`Pandas`を学びます。

```
┌──────────────────────┐
│ (0) 🐍 `Python`の基礎  │
└──────────────────────┘
            |
            v
┌──────────────────────┐   ┌──────────────────────┐   ┌──────────────────────┐   ┌──────────────────────┐   ┌──────────────────────┐
│ (1) 📂 データの読み込み │ --->│ (2) ✏ データの操作  │ --->│ (3) 🔧 データ加工  │ --->│ (4) 📊 集計       │ --->│ (5) 📈 データの可視化 │
└──────────────────────┘   └──────────────────────┘   └──────────────────────┘   └──────────────────────┘   └──────────────────────┘
```

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。

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 Colaboratory とは？

Google Colaboratory (Colab) のノートブックを基盤にしています。
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です。
つまり、**マニュアル書のような役割**も担えます。

- 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 Colaboratory ノートブック	≒	Word	+	Python

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
- Git Hub 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

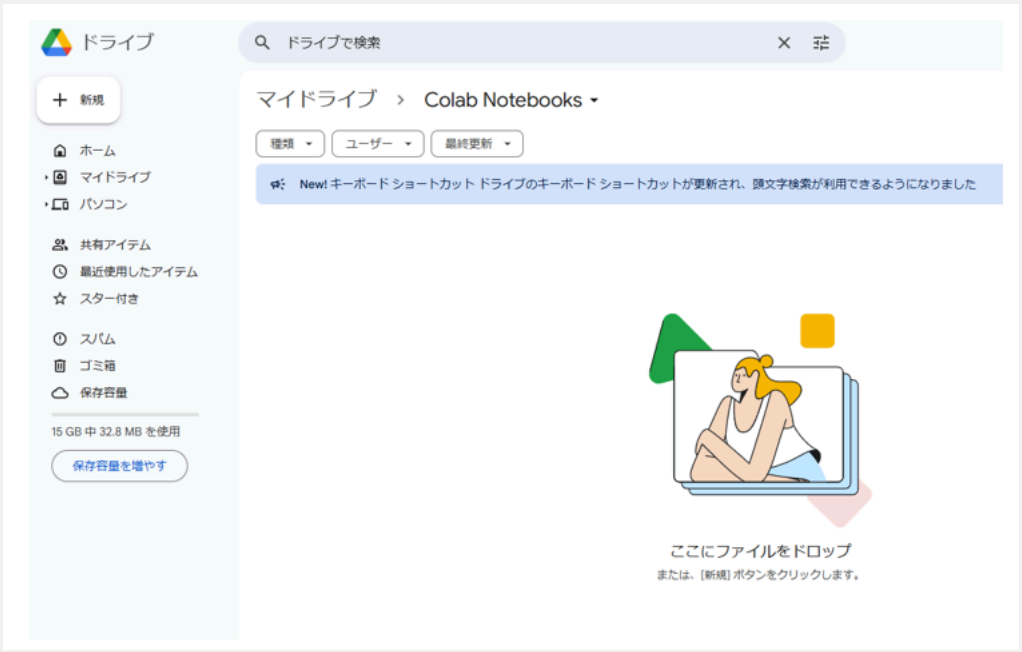
Google Colab の準備方法

1. GoogleColab (https://colab.research.google.com/?hl=ja) にアクセスし、 Google アカウントでログインします。
   - Google アカウントがない方は、別途作成が必要です。
   - Google アカウント作成ページ(https://www.google.com/intl/ja/account/about/)

2. ［ファイル］メニューから［ドライブの新しいノートブック］をクリックします。

3. Google ドライブ(https://drive.google.com/drive/home)に自動的に [Colab Notebook] が作成され、そのノートブックが Google ドライブに保存されます。

4. 作成された [Colab Notebook] に、講義資料として提供されたノートブックをアップロードします。
上記はパス指定せずに、実行できるフォルダになります。
別のフォルダにもアップロードできますが、 パス指定がややこしいので指定箇所にアップロードください。



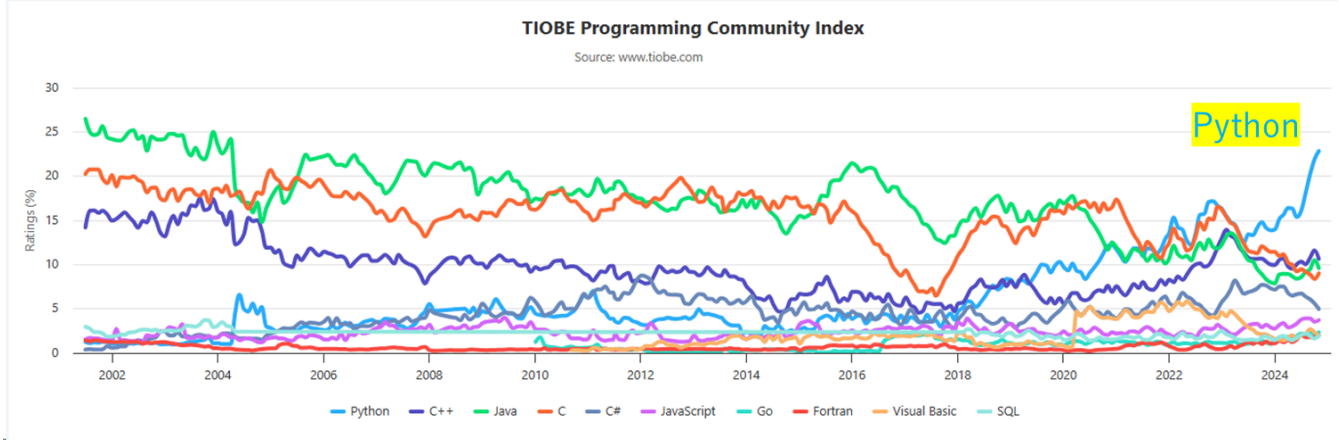
Pythonの基礎

```
┌──────────────────────────┐
★ (0) `Python`の基礎   ★
└──────────────────────────┘
          |
          v
┌──────────────────────┐   ┌──────────────────────┐   ┌──────────────────────┐   ┌──────────────────────┐   ┌──────────────────────┐
| (1) データの読み込み | --->| (2) データの操作   | --->| (3) データ加工 | --->| (4) 集計     | --->| (5) データの可視化 |
└──────────────────────┘   └──────────────────────┘   └──────────────────────┘   └──────────────────────┘   └──────────────────────┘
```

Ptyhonとは何か

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です。

```
                    ──────────        ──────────────
                              |        ┌──────┴──────┐
                              |        ↓             ↓
                              |        データ        科学(サイエンス)
                              |        ※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
📚 ライブラリが豊富
    ↓
📒✨ 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
```



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

TIOBEインデックスの2024年11月のデータによると、 `Python` は22.85%のレーティングを獲得し**1位**となっています。

（参考: TIOBE Index for November 2024）

前年同期比で8.69%増加しており、過去にも1位を獲得していることから、現在も利用がさらに拡大しています。

利用者が増えることで、**多くのフォーラムやチュートリアルが充実**し、学習や実務においての利便性が向上しています。

> TIOBEインデックス=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です。

変数

変数とは、値を格納するための器です。

```
x = 10  # 数値型の変数
name ='Alice'  # 文字列型の変数
```

📝 解説

本例は単純ですが、実際のプログラムではもっと複雑な値を扱うため、変数があると整理しやすくなります。

- 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

データ型とデータ構造

多様なデータ型とデータ構造があります。

◎ 基本データ型

- 数値型（`int`, `float`）
- 文字列型（`str`）
- 真偽値（`bool`）

  など

```
age = 25  # int
price = 19.99  # float
is_student = True  # bool
```

🤖

`Python` は**型を自動で判断する言語**です。

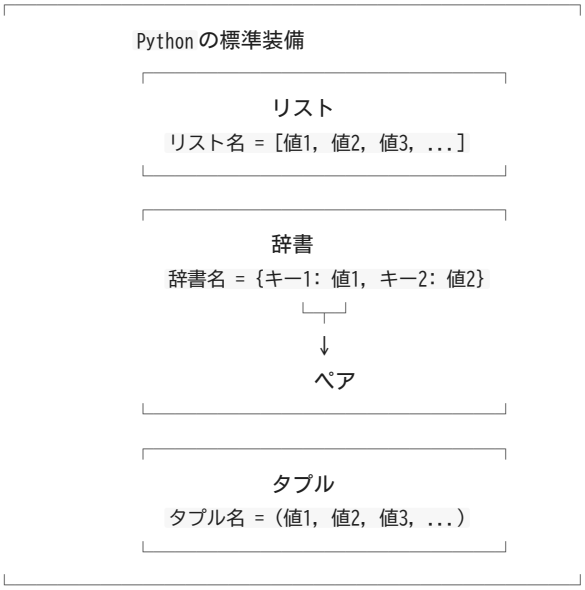
数値を入れれば数値型、文字列を入れれば文字列型になります。

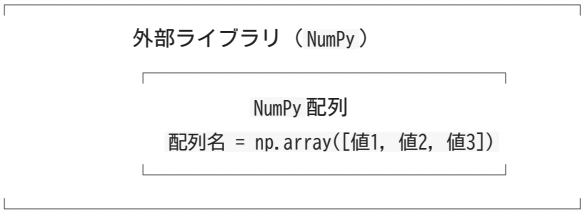
型を意識することで、**バグが減り、より良いプログラム**が書けます。

- なぜ型が大切？
  - エラーを早く見つけられる（文字列＋数値はできない）
  - プログラムの品質

  など
- ただし、以下の場合があるため注意
  - 特定の機能は決まった型のみ使えないケースがある。例えば、`upper()` は文字列だけ
  - ユーザーの意図とは異なる型を設定することがある

◎ データ構造





辞書（dict）型は**複数**の意味や関連情報を管理できる特徴があります。

一方、**リスト、タプル、NumPy 配列**は主に**1つ**の意味を扱う際に用います。

```
├──────────────────────┤
 "name": "Alice"
 "age": 25
 "job": "Engineer"
└──────────────────────┘
```

```
my_list = ["Alice", "Bob", "Charlie"]
my_list
```

- リスト（`list`）型
  - 最も基本的で柔軟な配列型
  - 異なるデータ型の要素の格納が可
  - 要素の追加、削除、変更が自由
  - `[]` での宣言

```
my_dict = {"name": "Alice", "age": 25, "job": "Engineer"}
my_dict
```

```
my_dict = {
    "name": ["Alice", "Bob"],
    "age": [25, 30],
    "job": ["Engineer", "Doctor"]
}
my_dict
```

- 辞書（`dict`）型:
  - キーと値のペアでのデータ管理
  - キーでの要素アクセス
  - `{}` での「キー:値」形式での宣言

```
my_tuple = ("Alice", "Bob", "Charlie")
my_tuple
```

- タプル（`tuple`）型
  - リストに類似、要素変更不可
  - 不変性が必要な場合に適性
  - `()` での宣言

```
import numpy as np
my_array = np.array(["Alice", "Bob", "Charlie"])
```

- NumPy 配列
  - 同一データ型のみの格納、要素数変更不可
  - NumPy ライブラリでの使用

プログラムの処理の流れ

> 文部科学省. (2020). *高等学校情報科『情報Ⅰ』教員研修用教材（本編）*. 文部科学省. https://www.mext.go.jp/content/20200722-mxt_jogai02-100013300_005.pdf を参考に作成

1. 順次処理（命令を上から順番に実行）

```
┌──────────┐
│   処理1   │
└──────────┘
     │
     │
     ▼
┌──────────┐
│   処理2   │
└──────────┘
```

📌 上から順に処理が実行される

2. 分岐処理（条件分岐≒条件によって異なる処理を実行）

```
┌──────────┐
│   条件   │ ────┐
└──────────┘     │
     │           │
     ▼           ▼
┌──────────┐ ┌──────────┐
│  処理1   │ │  処理2   │
└──────────┘ └──────────┘
     │      ◀────┘
     ▼
[ 次の処理へ ]
```

📌 条件判定の結果に応じて 処理1 または 処理2 を実行

```
if x > 5:
  print('xは5より大きい')
else:
  print('xは5以下')
```

3. 反復処理（ループ≒条件を満たす間、同じ処理を繰り返す）

```
┌──────────┐
│ ループ条件 │
└──────────┘
     │
     ▼
┌──────────┐
│  ループ  │
└──────────┘
     │
     ▼
┌──────────┐
│   処理   │
└──────────┘
     │ └──────────► （ループ継続）
     │
     ▼
[ 終了 ]

📌 条件を満たす間は処理を繰り返し、満たさなくなったら終了
```

```
for i in [1, 2, 3, 4, 5]:
  print(i)
```

📝 解説

リスト `[1, 2, 3, 4, 5]` の中の数字を順番に取り出して `print(i)` を実行する

`for` 〜 `in` ...は、「ある期間の間、ずっと」という意味から派生していそうです。

> 富田一彦．(2011)．*The Word Book とみ単*．大和書房．を参考に作成。

- `for` 〜 `in` 、`:` 、 インデントのプログラム上の意味
  - 前置詞 `for`

    向かう――が、到達ーーーしていない　⇒　その期間ずっと

    ```
    ──► ーーーーー ●
    ```

  - 前置詞 `in`

    囲まれているイメージ　⇒　範囲のある時間など

    ```
    ┌─────┐
    │  ●  │
    ```

```
└──────┘
```

- `:`に関して
  - `for`文の**宣言部分の終了**を示す
  - 次の行から始まる繰り返し処理のブロックの**開始**を示す

```
   前                 後

┌──────┐          ┌──────┐
│  A   │   ≒      │  B   │
└──────┘ (同格)    └──────┘
```

◎ コロンは**前後のものを紐づける同格としての役割**があり、「実行するのはこのブロックですよ」という意味を表現しています

- インデントのプログラム上の意味
  - **開始**
    - コロン（`:`）で終わる行（例：`if`文、`for`文、関数定義など）の直後から始まる
  - **範囲**
    - コロンの次の行から、同じインデントレベルが続く限りグループに属する
  - **終了**
    - インデントが戻る（左に寄る）か、ファイルの終わりで終了する

```
for ループ:  ←  (コロン `:` でグループ開始)

 ┌──────────────┐
 │ 繰り返し処理  │  ← インデント開始：同じインデントレベルに統一
 └──────────────┘

                  ← インデント終了：処理の終了
```

ライブラリの導入

データ分析には、`Python`標準ライブラリ以外に便利なツールが多数あります。

ライブラリは、`PC`に便利なソフトをインストールして使うようなものです。

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ます。

```
┌──────────────┐          ┌──────────────────────┐
│ 💻 PC        │ ◄──────  │ 📖ライブラリ(≒（便利なソフト）│
│（Python環境）│          │ をインストール        │
└──────────────┘          └──────────────────────┘

                                   ↖ 🤖
```

`NumPy`の使用準備

```
┌──────────────────────┐   >   ┌──────────────────────┐
│ (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ります。
   インストールには、以下のように、どちらかの`pip install`を使用します。

   ただし、`Google Colab`の環境下では、不要です。

```
#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  # 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. `Numpy`のインポート

   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、アクセスできるようにします。

```
import numpy as np
```

- 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**数値データを効率よく扱う**ためのライブラリです。

```
              数値計算のための Python ライブラリ
                          ↓
                       NumPy
                          ↑
┌──────────────────────────────────────────┐
│                                          │
↓                                          ↓
Number（数や数学）                          Python
```

Pandas の使用準備

```
┌──────────────────────┐   ┌──────────────────────┐
│（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
   Pandas を使用するには、まず Python 環境に Pandas をインストールする必要があります。
   インストールには、以下のように、どちらかの pip install を使用します。

   ただし、 Google Colab の環境下では、不要です。

```
#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 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

   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、アクセスできるようにします。

```
import pandas as pd
```

- Pandas とは

  Pandas の名前の由来は pan(el)-da(ta)-s という説があります。（もちろん、パネルデータ以外も扱えます）

```
┌──────────────────────────────────┐
 データ分析のための Python ライブラリ
      Pandas  ←  pan(el)-da(ta)-s
└──────────────────────────────────┘
```

  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）**で管理できます。

  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**のようなイメージです。

```
┌──────────────┐   ┌──────────────────┐
   Excel の表     ≒   Pandas のデータフレーム
└──────────────┘   └──────────────────┘
```

  - **異なる型のデータ**（数値や文字列など）を同じ表で扱える
    → 現実世界のデータ管理に近く、多様なデータを一つの表で整理できる
  - **各列に名前（ラベル）**を付けられる
  - **列や行ごとにデータを柔軟に操作**できる

```
🚌↗   ┌──────────────────────────────────────────────┐
      Pandas の操作は 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
      └──────────────────────────────────────────────┘
```

Matplotlib の使用準備

```
┌──────────────────────┐   ┌──────────────────────┐
│（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

   Matplotlibを使用するには、まずPython環境にMatplotlibをインストールする必要があります。
   インストールには、以下のように、どちらかのpip installを使用します。
   ただし、Google Colabの環境下では、不要です。

```
#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 Matplotlib  # Jupyter ノートブック 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 pyplot モジュールのインポート

   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、アクセスできるようにします。

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

- Matplotlibとは？

```
                データ可視化のためのPythonライブラリ
                              ↓
                          Matplotlib
                              ↑
  ┌──────────────────────────────────────────────────┐
  │                          │                       │
  ↓                          ↓                       ↓
  MATRIX（行列）          Plot（図を描く）          lib（ライブラリ）
```

  - **Mat**：**行列（MATRIX）**や数値データを扱う文脈に関連
  - **plot**：**「図を描く」「グラフを描く」**という意味の英単語
  - **lib**：プログラムで使う**「ライブラリ（library）」**を意味する略語

  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**です。

データの読み込み

```
┌──────────────────────┐
| (0) `Python`の基礎   |
└──────────────────────┘
          |
          v
┌──────────────────────┐   ┌──────────────────┐   ┌──────────────────┐   ┌──────────────────┐   ┌──────────────────┐
★ (1) データの読み込み ★ --->| (2) データの操作  | --->| (3) データ加工  | ---> | (4) 集計   | ---> | (5) データの可視化 |
└──────────────────────┘   └──────────────────┘   └──────────────────┘   └──────────────────┘   └──────────────────┘
```

データ分析では、まずデータを正しく取り扱うことが重要です。

CSVやExcelファイルの読み込みからデータの基本操作、欠損値の扱い方までを学びます。

- CSVやExcelファイルの読み込み Pythonのライブラリを使うと、データファイルを簡単に読み込むことができます。

```
# CSVファイルを読み込んでデータフレームに変換
transactions = pd.read_csv('https://www.salesanalytics.co.jp/wp-content/uploads/2025/01/transactions.csv')
transactions
```

```
pd（Pandas）ライブラリ
    ↑
    pd.read_csv( 'https://www.salesanalytics.co.jp/wp-content/uploads/2025/01/transactions.csv')
          |        └──> 読み込むファイルのURLを指定する
          |
          v
       read_csv をデータフレームとして読み込む関数
         →  「CSVを読む（read）」 で、CSV形式のファイルを データフレームに変換する
```

- Excelファイルの読み込み

```
# Excelファイルを読み込む
# transactions = pd.read_excel('https://www.salesanalytics.co.jp/wp-content/uploads/2025/01/transactions.xlsx')
```

データの操作

```
┌──────────────────────┐
│ (0) `Python`の基礎   │
└──────────────────────┘
          │
          v
┌──────────────────────┐   ┌──────────────────────┐   ┌──────────────────┐   ┌──────────────┐   ┌──────────────────────┐
│ (1) データの読み込み │ --->★ (2) データの操作 ★ --->│ (3) データ加工 │ ---> │ (4) 集計 │ ---> │ (5) データの可視化 │
└──────────────────────┘   └──────────────────────┘   └──────────────────┘   └──────────────┘   └──────────────────────┘
```

行と列から成る表形式のデータフレームと呼びます。

列

↓

name	age	job
Alice	25	Engineer
Bob	30	Designer
Carol	28	Manager
David	35	Doctor

└──────────> **行**

データの確認

データの全体像や統計情報を確認します。

◎ `head`

`オブジェクト.head` により、データの先頭行を表示し、 データの概要を把握します。

`head()` の名前の由来は「データの先頭（`head`）」から来ています。

```
head
 ▲
 │
 └──────head = 先頭
```

```
# 最初の5行を表示
transactions.head(10)
```

📝 解説

デフォルトは5行が表示されます。

行数を指定したい場合は、引数に数値を渡します。 例えば、 `df.head(1)` とすると先頭1行が表示されます。

◎ `info`

`オブジェクト.info()` により、データの「情報」を簡潔にまとめます。

`info` の名前の由来は「情報（information）」から来ています。

```
info

 ▲

 │
```

└──── info ≒ 情報

```
# データの概要を確認
transactions.info()
```

📝 解説

```
<class 'pandas.core.frame.DataFrame'>                ←  データ型
RangeIndex: 100 entries, 0 to 99                     ←  行数と行インデックスの範囲
Data columns (total 5 columns):                      ←  列数

 ──────────────────────────────────────────────────── 列のインデックス
 |      ──────────────────────────────────────────── 各列の名称
 |      | ────────────────────────────────────────── 各列の非欠損値の数
 |      |                |
 |      |                |           ──────────────── 各列のデータ型
 |      ↓                ↓           ↓
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   TransactionID  100 non-null    int64
 1   CustomerID     100 non-null    int64
 2   ProductID      100 non-null    int64
 3   Date           100 non-null    object
 4   Quantity       100 non-null    int64

dtypes: int64(4), object(1)                          ← データ型の内訳
memory usage: 4.0+ KB                                ← メモリ使用量
```

項目	説明
`<class 'pandas.core.frame.DataFrame'>`	← データ型（DataFrame であることを示す）
`RangeIndex`	← 行数と行インデックス範囲（例: 100行、インデックスは0〜99）
`Data columns`	← 列数（5列）
`#`	← 列のインデックス番号
`Column`	← 列名（例: `TransactionID`, `CustomerID` など）
`Non-Null Count`	← 各列の非欠損値の数
`Dtype`	← 各列のデータ型（例: `int64`, `object` など）
`dtypes`	← データ型の内訳（例: `int64` 型が4列、`object` 型が1列）
`memory usage`	← メモリ使用量（例: 4.0+ KB）
`None`	← メソッドの戻り値（返り値がない）

詳細は以下を参考にしてください。

> 「データの全貌を一目で把握！PandasのDataFrame.info()を使いこなそう ｜ データ分析ドットコム」https://biz-data-analytics.com/python/11026/（2025年1月17日）

⚠ データの型が違う場合のトラブルシューティング

上記の結果では、`Date` 列が `object` 型（文字列型）として読み込まれています。

`Date` 列を日付型にしないと、後々以下のような処理が面倒になる場合があります。

- データのフィルタリング（特定の日付範囲のデータを抽出するなど）
- 日付の計算（例: 期間の差を計算する）

  など

🛠 対処方法

`read_csv()` の `parse_dates` 引数を使用して、`Date` 列を日付型として読み込む必要があります。

`parse_dates` はその名前の通り、「日付を解析する（parse dates）」ためのものです。

`parse_dates`

▲

│
└──── 「日付を解析する（parse_dates）」

※読み込み後に、型変換する方法もあります。

🚨 ↘

┌──┐
他の列の型も厳密に修正する必要があります。

TransactionID、CustomerID、ProductID がゼロ始まりの場合、

データの読み込み時に数値型で変換されるため、先頭の 0 が自動削除されます。

そのため、文字列型などに指定する必要があります。

└──┘

◎ transactionsのDataFrameをメモリから削除

transactions の DataFrame を日付型にして読み込むため、 一度、 transactions の DataFrame をメモリから削除します。

```
# transactionsのDataFrameをメモリから削除
del transactions
```

📝 解説

---

del： delete の略で、指定した変数を削除します。

```
# 日付型を指定してデータを読み込む
transactions = pd.read_csv(
    'https://www.salesanalytics.co.jp/wp-content/uploads/2025/01/transactions.csv',
    parse_dates=['Date']
)
```

Date 列が日付型になっているかを確認します。

```
# データの型を確認
transactions.info()
```

info() でデータの概要を把握し、欠損値や列の型を確認・調整することが大切です。

◎ shape

オブジェクト.shape により、データの**行数と列数**を確認し、データのサイズを把握します。

shape の名前の由来は「データの形状（shape）」から来ています。

---

shape

▲

│

└──── shape ≒ 形状

```
# データの形状を確認（行数と列数）
transactions.shape
```

◎ describe

オブジェクト.describe() により、各列の**基本統計量**を確認し、**現状を把握してビジネス課題の解決に活用**。

describe の名前の由来は「記述（describe）」から来ています。

```
describe

▲

│

└──── describe ≒ 記述
```

```
1 # 各列の基本統計量を確認
2 transactions.describe()
```

```
                                列名
                                 ↓

           TransactionID   CustomerID   ProductID   Quantity
count  →  100.000000    100.000000   100.000000   100.000000
mean   →  1050.500000     28.730000   150.440000     4.730000
std    →    29.011492     13.538464    29.165609     2.585332
min    →  1001.000000      2.000000   101.000000     1.000000
25%    →  1025.750000     20.750000   122.750000     3.000000
50%    →  1050.500000     30.000000   154.500000     4.000000
75%    →  1075.250000     39.250000   169.000000     7.000000
max    →  1100.000000     50.000000   199.000000     9.000000
```

統計量	説明
'count'	各列のデータポイント（レコード）の総数を示す
'mean'	各列の平均値を示す
'std'	各列の標準偏差を示し、データのばらつきを表す
'min'	各列の最小値を示す
'25%'	各列の第1四分位点（全データを4等分した際の下位25%の境界）を示す
'50%'	各列の中央値（データの中心値）を示す
'75%'	各列の第3四分位点（全データを4等分した際の上位25%の境界）を示す
'max'	各列の最大値を示す

🔒 留意点

非数値データが含まれる場合、デフォルトでは数値列のみが対象となりますが、 `include='all'` を指定し、
`print(transactions.describe(include='all'))` すべての列が出力されます。

`オブジェクト.print` により、**データ全体**を表示します。
データを直感的に把握し、内容を確認する際に便利です。

`print` の名前の由来は**表示する（`print`）**から来ています。

```
print

▲

│

└──── print ≒ 表示
```

◎ `print`

```
1 # データフレーム全体を表示する
2 print(transactions)
```

```
                                列名
                                 ↓

           TransactionID   CustomerID   ProductID   Quantity
```

```
0        1001        2        101        1
1        1002        4        150        3
2        1003        6        154        2
3        1004        8        199        5
4        1005       10        120        7
```

表示内容	説明
列名	データフレームのカラム名を表示します

◎ 欠損値の確認

```
# 各列の欠損値の数を確認
transactions.isnull().sum()
```

📝 解説

`Pandasオブジェクト.isnull().sum()` で各列の欠損値の個数を確認できます。

```
transactions.isnull().sum()
                  |      └──── .sum() : True=1 の False=0 の数を合計
                  ↓.isnull() : 欠損値の有無の判定
                                 ├ 欠損値が存在する → True=1
                                 └ 欠損値が存在しない → False=0
```

❓ FAQ

Q: `count()` ではなく `sum()` を使用するのはなぜですか？

A:

`transactions.isnull()` は各セルが欠損値かどうかを `True/False` で示し、`True` は `1`、`False` は `0` として扱われます。

そのため、`sum()` を使うと欠損値の数を正しく計算できます。

一方、`count()` は非欠損値の数を数えるため、欠損値の数を求めるのには適していません。

行列の参照

特定の行や列を参照して操作できます。

まずは、参照のお話です。

`[]` により**参照**ことができます。

◎ 列を取り出すケース

📜 記法

○ 1列の場合

`データフレーム['列名1']` と記述します

```
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
```

```
# 列を選択
transactions['Date']
```

📝 解説

データフレーム `transactions` の中から `Date` 列を選択して出力します。

```
transactions['Date']
                 └── Date: 指定した列のデータを取得
```

○ 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

⚠ 注意

- 1列を取り出すと **1次元データ**（`Series` 型）
- 複数列を取り出すと **2次元データ**（データフレーム型）

  😔データ抽出時に1列は自動的に `Series` 型になるため、後の処理でエラーが発生することがあります。

```
# 複数の列を選択
transactions[['Date', 'CustomerID']]
```

📝 解説

データフレーム `transactions` の中から `Date` 列と `CustomerID` 列を選択して出力します。

```
transactions[['Date', 'CustomerID']]
                 └── Date, CustomerID: 取得したいデータの列名をカンマで区切りで入力
```

◎ 行を取り出すケース

📜 記法

- `.iloc` と `.loc`

```
┌──────────────────────────────────────────────┐
  .iloc : データを取得するためのインデックス指定
  │
  └──────▶ iloc: integer location（整数の場所）
                ↳ 行番号・列番号でアクセス
  .loc : データを取得するためのインデックス指定
  │
  └──────▶ 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
└──────────────────────────────────────────────┘
```

○ 1行選択

`データフレーム.iloc[行番号]` や `データフレーム.loc[行名]` で指定します。

```
データフレーム.iloc[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
```

```
# 特定の行を選択
transactions.iloc[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）を選択して出力します。

```
transactions.iloc[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.iloc[開始:終了]` での範囲指定します。

```
データフレーム.iloc[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
transactions.iloc[1:5]
```

📝 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します。

```
transactions.iloc[1:5]
                  └── インデックス 1~4 の行を取得
```

○ 条件を満たす行を取り出すケース

`データフレーム[条件式]` での条件指定し、行をと取り出します。

```
データフレーム[条件式]  （例: データフレーム[データフレーム['Quantity'] > 4]）
```

🔍 `[]`をさらに詳しく

`データフレーム[データフレーム['Quantity'] > 4]`の外側 `[]` で「行の取り出し」、**内側 `[]` で「列の指定」**をしています。

- `データフレーム['Quantity']`: `Quantity` 列を指定
- `データフレーム['Quantity'] > 4`: この列に対して `4` より大きい値を持つ行を条件式として指定

```
        条件を満たす行の取り出し
              ↑
データフレーム[データフレーム['Quantity'] > 4]
                                  └── Quantity` 列 に対する条件設定
```

```
# 条件を満たす行を抽出
filtered_data = transactions[transactions['Quantity'] > 4]
filtered_data
```

📝 解説

データフレーム `transactions` の中から `Quantity` 列の値が 4 より大きい行を抽出し、新しいデータフレーム `filtered_data` に格納して出力します。

- `transactions['Quantity'] > 4` の部分での **`Quantity` 列** に対する条件設定
- その後、**`transactions[...]`** での、**条件を満たす行**の取り出し

🔍 さらに詳しく言うと・・・

---

1. `transactions['Quantity'] > 4` が **`True/False`** を生成
2. **`True` の位置**（行番号）を **記憶**
3. **記憶した位置** の行を `transactions` から取り出す

└──────────>実際は、**`True` の位置の行を記憶して、取り出す**という動作

データ加工

```
┌──────────────────┐
│ (0) 🐍 `Python`の基礎  │
└──────────────────┘
          │
          v
┌──────────────────┐  ┌──────────────────┐  ┌──────────────────┐  ┌──────────────────┐  ┌──────────────────────┐
│ (1) 📂 データの読み込み │ --->│ (2) ✏ データの操作  │ --->★ (3) 🔧 データ加工 ★ ---> │ (4) 📊 集計   │ ---> │ (5) 📈 データの可視化 │
└──────────────────┘  └──────────────────┘  └──────────────────┘  └──────────────────┘  └──────────────────────┘
```

欠損値の処理

- 欠損値の除去

```
# 欠損値を含む行を削除
#transactions = transactions.dropna()
```

📝 解説

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

`transactions = transactions.dropna()`

└── `.dropna()` ： 欠損値を含む行を削除するメソッド

│ │

│ └──▶ `na`:欠損値（Not Available）

│

└──▶ `drop`:削除

- 欠損値の補完

```
# 欠損値を特定の値で埋める
# transactions = transactions.fillna(0)
```

📝 解説

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ます。

`transactions = transactions.fillna(0)`

└── `.fillna(値)` ： 欠損値を指定した値で埋めるメソッド

│ │

│ └──▶ `na`:欠損値（Not Available）

│

└──▶ `fill`:埋める

```
# データの読み込み
products = pd.read_csv('https://www.salesanalytics.co.jp/wp-content/uploads/2025/01/master_products.csv')
customers = pd.read_csv('https://www.salesanalytics.co.jp/wp-content/uploads/2025/01/master_customers.csv',parse_dates=['JoinDate'])
```

```
products.head()
```

```
customers.head()
```

```
customers.info()
```

別データから列の追加

```
================================================================================
┌──────────────────────┐  ┌──────────────────────┐
    左データフレーム          右データフレーム
      transactions              products
├──────────────────────┤  ├──────────────────────┤
  TransactionID          ->  ProductID
  CustomerID             |   ProductName
  ProductID ──────────────┘   Category
  Date                         Price
  Quantity
├──────────────────────┤  ├──────────────────────┤


            |
            ▼
┌──────────────────────────────────────────────────┐
  左データフレーム:  transactions
└──────────────────────────────────────────────────┘
  TransactionID
  CustomerID
  ProductID
  Date
  Quantity
  TotalAmount ← Quantity × Price の結果が格納
└──────────────────────────────────────────────────┘
```

```
transactions
```

```
# 売上額を計算して新しい列に追加
transactions['TotalAmount'] = transactions['Quantity'] * transactions['ProductID'].map(
    products.set_index('ProductID')['Price']
)
transactions.head()
```

📝 解説

```
                                        ┌──────> transactions の購入数量
transactions['TotalAmount'] = transactions['Quantity'] * transactions['ProductID'].map(products.set_index('ProductID')['Price'])
  |                                       —
  |                                       ×   2つのデータの ProductID に対応する価格を取得
  ▼                                                          |
                                                             |
新しい列 TotalAmount が生成：                                  |
新しい列名を指定して値を代入すると、                            |
自動的にその列を作成する仕様                                   |
                                                             |
                                                             |
                                                             |
                                                             ▼
```

📜 記法

```
`左データ['結合キー'].map(右データ.set_index('結合キー')['取得したい列'])`
```

結合キーをもとに `右データ` から `取得したい列` の値を取得します。

ただし、取得した値を代入しない限り `左データ` に列は追加されません。

`set_index('結合キー')['取得したい列']` は一般的な DataFrame 操作の組み合わせです：

- `右テーブル.set_index('結合キー')` は DataFrameオブジェクト を返す
- その後、 `['取得したい列']` を使って特定の列を選択しているだけで、通常の df[] の考え方と同じです

本例では、 `ProductID` をもとに `products` から `Price` を取得し、 `transactions` の `Quantity` と掛け算して `TotalAmount` を作成しています。

◎ マッピング操作の確認

以前出力した内容と照らし合わせることで、変換の正確性を確認します。

同じ個所、追加箇所がどうなっているかを確認します。

```
┌──────────────────────┐      比較      ┌──────────────────┐
  before                ◀──────▶         after
  データ                                 データ
└──────────────────────┘                └──────────────────┘
```

⚠️ 記録が必要

※新しく取得した列は、以下のコード実行後で違和感がないかで確認する

```
transactions.isnull().sum() # 欠損値の確認
```

通常、 `transactions.csv` や `master_products.csv` に欠損値は存在しないはずです。 欠損値が発生した場合、以下の可能性を確認します。

- `transactions.csv` または `master_products.csv` のデータ自体の問題
- コードの指定ミスや参照カラムの誤り

```
transactions.head(10)  # 最初の10行を表示
```

```
transactions.info()  # データの概要を確認

```

```
transactions.shape  # データの形状を確認（行数と列数）
```

```
transactions.describe()  # 基本統計量を確認
```

😔 `map()` メソッドの確認方法はやや不便

ただし、このアプローチには不便な点があります。

本来、 `map()` メソッドの利点は**元のデータを保持したまま、新しい配列を作成**することにあります。

そのため、次のような不便さが残ります。

- 元のデータと照らし合わせの不便さ
  - 元のデータに対する確認メソッドの出力を保持しておく必要がある
  - 計算に使用した列は、別途、 `map()` しないと追加されないため、**計算の検算**

    など

    ↓

    この課題は、後述する `merge()` メソッドを使用することで解決できます。

    `merge()` メソッド は元のデータを保持しながら、新しいデータを作成するためです。

    なお、 `map()` メソッドの利点はインデックスを生成しない点など多々あります。

別のデータを結合させ、新しいデータを作成

```
==============================================================================
┌──────────────────┐   ┌──────────────────┐
  左データフレーム        右データフレーム
    transactions           products
├──────────────────┤   ├──────────────────┤

  TransactionID       -●   ProductID
  CustomerID          |   ProductName
  ProductID >──────────┘   Category
```

```
Date                        Price
Quantity
├──────────────────────┤  ├──────────────────────┤

            │
            ▼

┌──────────────────────────────────────────────────────┐
新しいデータフレーム:  transactions_with_products
└──────────────────────────────────────────────────────┘

TransactionID      ← transactions の列
CustomerID         ← transactions の列
ProductID          ← transactions の列
Date               ← transactions の列
Quantity           ← transactions の列
ProductName        ← products の列
Category           ← products の列
Price              ← products の列
└──────────────────────────────────────────────────────┘
```

◎ transactionsをメモリから削除

map を使って、 transactions を上書きしたため、一度、 transactions をメモリから削除します。

```
# transactionsをメモリから削除
del transactions
```

📝 解説

---

del： delete の略で、指定した変数を削除します。

```
# 日付型を指定してデータを読み込む
transactions = pd.read_csv(
    'https://www.salesanalytics.co.jp/wp-content/uploads/2025/01/transactions.csv',
    parse_dates=['Date']
)
```

```
# transactionsとproductsのデータ結合
transactions_with_products  = pd.merge(transactions, products, on='ProductID', how='left')
transactions_with_products.head()
```

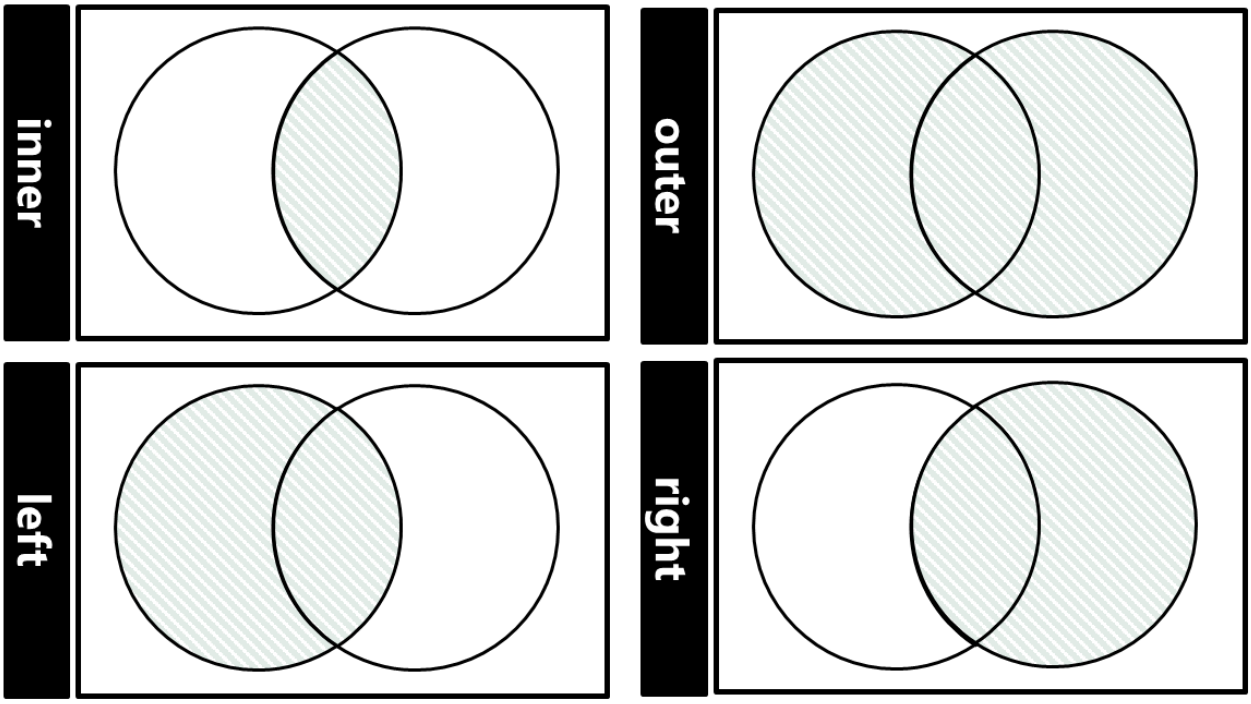
📝 解説

```
┌──────────────────────────────────────────────────────────────────────────────────────┐
transactions と products を ProductID を基準に結合（merge）し、
左側（left）のデータフレームのすべての行を保持（how='left'）します。
└──────────────────────────────────────────────────────────────────────────────────────┘

        ┌──> Pandas の merge 関数: 2つのデータを結合
pd.merge(transactions, products, on='ProductID', how='left')
              |         |             |              └──> how='left'  左側のデータのすべての行を保持
              ↓         |             |                                        |
      左側のデータ       ↓             |                                        |
                  右側のデータ          ↓                                        |
                                 結合キー：                                      |
                                 on（≒ 基準）：結合の基準となる列を指定          |
                                 ProductID 列を基準に結合                        ↓

                          ┌──────────────────────────────────────────────────┐
                          - 'inner'（≒ 内部）→ 一致するデータのみ
                          - 'outer'（≒ 外部）→ すべてのデータ
                          - 'right'（≒ 右）→ 右側のデータを保持　など
                          補足
```

- how を省略すると、'inner' になる



💡 コードと英語を結びつける記憶術

「どのように（how）」結合するかを決める際、

結合の**接触対象の列を**「on」で指定するイメージです。

```
┌──┐
│  │○ 前置詞の`on`は「接触」
└──┘
  └──────> 前置詞の「on」が「接触」を意味するように、２つのテーブル間の関連付けや条件に「基づく」結合を示します。

`how` ≒ 方法
  ──────▶ 「どのように」と訳され、何かを行う方法を示します。データ結合では、「どのように結合するか（方法）」を決めます。
```

◎ 結合操作の確認

以前出力した内容と照らし合わせることで、変換の正確性を確認します。

`merge()` メソッド は元のデータを保持しながら、新しいデータを作成します

そのため、 `map` メソッドより、照らし合わせが簡単です。

```
┌──────────┐        比較        ┌──────────┐
│ before   │◀──────▶│ after    │
│ データ   │                    │ データ   │
└──────────┘                    └──────────┘
```

```
# 結合前後のデータの確認
print("=== 欠損値の比較 ===")
print("¥n[結合前]")
print(transactions.isnull().sum())
print("¥n[結合後]")
print(transactions_with_products.isnull().sum())
```

⏳ 補足

通常、 `transactions.csv` や `master_products.csv` に欠損値は存在しないはずです。 欠損値が発生した場合、以下の可能性を確認します。

- `transactions.csv` または `master_products.csv` のデータ自体の問題
- コードの指定ミスや参照カラムの誤り

```
print("¥n=== 最初の10行の比較 ===")
print("¥n[結合前]")
print(transactions.head(10))
print("¥n[結合後]")
print(transactions_with_products.head(10))
```

```
print("¥n=== データの形状（行数，列数）の比較 ===")
print("¥n[結合前]", transactions.shape)
print("¥n[結合後]", transactions_with_products.shape)
```

```
print("¥n=== 基本統計量の比較 ===")
print("¥n[結合前]")
print(transactions.describe())
print("¥n[結合後]")
print(transactions_with_products.describe())
```

❓ FAQ

Q: データフレームの出力に `¥`（バックスラッシュ）が付いているのですが、大丈夫ですか？

A: はい、問題ありません。 `pandas` のデータフレームで列が長すぎると折り返され、その目印として `¥` が挿入される。

Q: `map()` を使用した場合、一致しない `ProductID` があるとデータの一部は消えますか？
A: `on=left` のような動きです。データ自体は消えず、`TotalAmount` には `NaN` が入ります。

Q: `map()` の結合キーの型が一致しない場合、どうなりますか？
A: マッチングが正しく行われず、すべての値が `NaN` になります。事前に型を統一することが重要です。

Q: `map()` で計算に使用した列は、自動的にデータフレームに追加されますか？
A: いいえ、`map()` で取得した値は、指定した計算にのみ使われます。元のデータフレームには自動で追加されないため、計算結果の検算をする場合は、別途 `map()` で対応する列を追加する必要があります。

🔍ちょっと豆知識

`map` ≒ マッピング ──▶ 既存データフレームに、別のデータから必要な値だけを引っ張ってくる操作

`merge` ≒ 結合 ──▶ インデックス以外の列（キー）や複数の列（キー）を基準にしても結合でき、新しいデータフレームを作成

◎ CSV に保存

```
# 加工後のデータを保存
transactions_with_products.to_csv('transactions_with_products.csv', index=False)
```

📝 解説

```
┌──────────────────────────────────────────────────────────────────────────┐
transactions_with_products データフレーム を、行番号なしで transactions_with_products.csv`に保存します
└──────────────────────────────────────────────────────────────────────────┘

transactions_with_products データフレーム
      ↖
    transactions_with_products.to_csv('transactions_with_products.csv', index=False)
                                |            └──>出力ファイル名を指定する    └──>行番号を保存のオプション。False= 保存しない
                                |
                                v
                        CSV に保存するメソッド
```

🔍ちょっと豆知識

```
`to` ≒ 「到達」

  ────→ ●
```

`to` は「到達」を意味し、データが **変換・出力** されることを示しています。

本例においては、 `transactions_with_products.to_csv` は データフレームを `CSV` に出力しています。

```
+--------------------------------------------+  to    +--------+
transactions_with_products                     --->      CSV
+--------------------------------------------+        +--------+
```

集計

```
┌──────────────────────┐
| (0) 🐍 `Python`の基礎 |
└──────────────────────┘
          |
          v
┌──────────────────────┐     ┌──────────────────┐     ┌──────────────────┐     ┌──────────┐     ┌──────────────────┐
| (1) 📁 データの読み込み | ---> | (2) ✏ データの操作 | ---> | (3) 🔧 データ加工 | ---> ★ (4) 📊 集計 ★ ---> | (5) 📈 データの可視化 |
└──────────────────────┘     └──────────────────┘     └──────────────────┘     └──────────┘     └──────────────────┘
```

単純集計

データ全体または特定のグループ全体に対して単一の集計操作を行う集計操作です。

- 単純集計（全体）

  データ全体を対象した集計のことです。

```
──────────┬──────────
 全体     |    値
──────────┴──────────
          └────────> 全体：データ全体を示す
```

```python
# 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 = transactions['Quantity'].sum()
average_sales = transactions['Quantity'].mean()
median_sales = transactions['Quantity'].median()
unique_customers = transactions['CustomerID'].nunique()
total_transactions = transactions.shape[0]

print(total_sales, average_sales, median_sales, unique_customers, total_transactions)
```

📝 解説

```
┌──────────────────────────────────────────────────────┐
transactions の Quantity 列を選択し、その列の値を合計します。
└──────────────────────────────────────────────────────┘

transactions['Quantity'].sum()
|                       |
v                       └────────> SUM ≒合計  ──────┐
transactions の中から Quantity 列を選択                |
                                                      v
                                  ┌──────────────────────────────────────────┐
                                  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
                                    median ≒ 中央値
                                    ▶ 指定された列の値を昇順に並べた際の中央に位置する値を計算
                                    nunique ≒ ユニーク（重複がない）
```

▶ 指定された列の中で重複を除いた値の個数を計算

`shape` ≒ サイズ

▶ 行数と列数、`shape[0]` で行数を取得可能

など

- 単純集計（グループ別）

  特定のグループの全体値を集計します。

グループ	値	
項目１	XXX	―> グループごとの全体を示す
項目２	XXX	
項目３	XXX	

```
# グループ化して集計
sales_by_customerid = transactions.groupby('CustomerID')['Quantity'].sum()
sales_by_customerid
```

📝 解説

`transactions` の `Quantity` 列を選択し、その列の値を `CustomerID` ごとに**合計（`sum`）**します。

```
transactions.groupby('CustomerID')['Quantity'].sum()
|                                          └──────> SUM ≒合計
v
```

グループ化されたデータから `Quantity` 列を選択します。

💡 `df['Quantity'].sum` とは、操作上は同じイメージ

`df['Quantity']` に、`groupby` メソッドを追加しているので、違うように見えるだけで、操作上は同じです。

==========

`df['Quantity'].sum` ：**1. 通常選択**

```
    |          └──────────────┐
    ↓                         ↓
```

`transactions.groupby('CustomerID')['Quantity'].sum` ：**2. グループ化の選択**

==========

◎ イメージ

1. **通常選択（`df['Quantity'].sum()`）**

   元のデータフレーム：

```
+----+--------+---------+---------+
| ID | Gender | Product | Quantity|
+----+--------+---------+---------+
| 0  |  Male  |   A     |   10    |
| 1  | Female |   B     |   15    |
| 2  |  Male  |   C     |    5    |
| 3  | Female |   A     |   20    |
+----+--------+---------+---------+
```

↓ `df['Quantity']` の参照イメージ

```
'Quantity'列の全ての値を参照 → [10, 15, 5, 20]
```

↓ `.sum()`の集計を実行

```
50  # (10 + 15 + 5 + 20)
```

2. **グループ化後の列選択（`df.groupby('Gender')['Quantity'].sum()`）**

   元のデータフレーム：

```
+----+--------+---------+---------+
| ID | Gender | Product | Quantity|
+----+--------+---------+---------+
| 0  |  Male  |   A     |   10    |
| 1  | Female |   B     |   15    |
| 2  |  Male  |   C     |    5    |
| 3  | Female |   A     |   20    |
+----+--------+---------+---------+
```

↓ `df.groupby('Gender')['Quantity']`参照のイメージ（実際にデータは分割されない）

```
'Male' → 'Gender'列が'Male'の'Quantity'値を参照 → [10, 5]
'Female' → 'Gender'列が'Female'の'Quantity'値を参照 → [15, 20]
```

↓ `.sum()`の集計を実行

```
 'Gender'
'Male'      15  # (10 + 5)
'Female'    35  # (15 + 20)
Name: 'Quantity', dtype: int64
```

クロス集計

- クロス集計の構成

  2つの分類項目（行と列）で区切られた表に、集計値を表示する分析手法です。

  クロス集計の構成は、行、列、集計値の主に3つです。

```
                ┌──────────────────┐
                │ 2.列             │
────────────────┼──────────────────┤
  1.行          │ 3.集計値         │
────────────────└──────────────────┘
```

  主な3つの要素：

  1. 行：
  - データを縦方向に分類する項目
    - 例：性別（男性/女性）、年齢層など
  2. 列：
  - データを横方向に分類する項目
    - 例：商品カテゴリー、地域など
  3. 集計値：
    - 合計（売上金額など）
    - 平均（平均購入額など）
    - 件数（購入回数など） など

  この3つの要素を組み合わせることで、データの関係性や傾向を分かりやすく把握することができます。

- クロス集計

  クロス集計は、データをカテゴリごとに分割して比較し、特定の関係性を明らかにする分析手法です。

  商品のカテゴリごとに販売数量を集計したクロス集計表です。

主な目的は、商品カテゴリごとの販売行動を具体的に把握することです。

│

▼

1点買い（数量が1）が多いカテゴリには個別割引を提供し、多点買い（数量が5以上）が多いカテゴリにはまとめ買い割引を適用するなど、購買パターンに応じたプロモーションを設計できます。

また、購買傾向を把握することで、在庫の調整や過不足の予防にもつながります。

Quantity	1	2	3	4	5	6	7	8	9
Books	1	0	4	3	1	0	3	0	4
Clothing	3	3	3	4	8	0	0	5	3
Electronics	2	1	3	3	1	0	3	1	1
Home	6	2	2	4	2	1	7	1	2
Toys	2	2	2	1	3	1	0	1	1

◎ pivot_tableによるクロス集計

- 顧客ごとの平均購入数量をピボットテーブル形式で表示します。
  - 分析目的: 顧客の購入パターンを視覚化し、購入習慣や重点的なマーケティング施策を特定するため。

```
# ピボットテーブル作成
pivot_example = transactions.pivot_table(values='Quantity', index='CustomerID', aggfunc='mean')
pivot_example.head()
```

📝 解説

transactionsのQuantity列を選択し、その列の値をCustomerIDごとに**平均(mean)**します。

データ（Table）を異なる視点から「回転」（Pivot）させ、データを様々な角度から見る(a)

Table　　Pivot

↑　　↑

│　　│

└──────┘

│

transactions.pivot_table(values='Quantity', index='CustomerID', aggfunc='mean')

│　　│　　└──▶ (4)**集計関数**:各グループのQuantity列の平均を計算

(3)**集計対象列**　　↓

Quantity列を集計　　(1)**行ラベル**

CustomerIDを基準にグループ化して集計

(2)columnsを指定するとクロス集計となります。
　本例はindexのみを指定しているため、クロス集計のパートには含まれません

┌──────────────┐

│ (2)columns　　│

──────┼──────────────┤

(1)CustomerID　　│ (3)values,(4)aggfunc　　│

└──────────────┘

> (a)Excelピボットテーブル入門: 基本操作と構成要素の完全ガイド | データ分析ドットコム". https://biz-data-analytics.com/excel/excel_pivottable/7881/, (2025/02/05)より改訂-

◎ crosstabによるクロス集計

```
### コード例:
# クロス集計表の作成
cross_tab_quantity = pd.crosstab(
    index=transactions_with_products['Category'], columns=transactions_with_products['Quantity']
```

```
)
cross_tab_quantity
```

📝 解説

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
transactions_with_products の Category 列を行ラベル、 Quantity 列を列ラベルとするクロス集計表を作成し、各組み合わせの カウント（出現回数） を計算します。
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────┘

        ┌──────>Pandas の crosstab 関数:クロス集計表を作成
        |          └── index と columns で指定された項目の組み合わせに基づいて集計
        |          └── デフォルトの集計値は、行と列の値の組み合わせごとに 該当する組み合わせのカウント
        |
pd.crosstab(index=transactions_with_products["Category"], columns=transactions_with_products["Quantity"])
              └──>(1) Category 列の値を基準にして行を構成する      └──>(2) Quantity 列の値を基準にして列を構成する


                  ┌──────────────────────────┐
                  | (2)'columns'              |
        ──────────┼──────────────────────────┤
        (1)'index'| 'count':デフォルトの集計値 |
        ──────────└──────────────────────────┘
```

★ カテゴリごとの総売上 次に、 map() の例と同様に、カテゴリ別の総売上を集計した例を示します。

カテゴリ別の総売上を集計する目的は、各カテゴリの売上規模を把握し、それに基づいて具体的なアクションを取るためです。

限られたリソースを効率的に配分します。

たとえば、

- 売上が高いカテゴリ（例: Clothing）には、広告投資の増加や品揃えの拡充などの施策を検討
- 売上が低いカテゴリ（例: Electronics）では、価格調整や新製品の導入を検討

Category	Sales
Books	21,369.16
Clothing	29,162.87
Electronics	13,201.19
Home	24,999.17
Toys	15,061.48

```
### コード例:
# 売上高を計算し、新しい列を追加
transactions_with_products['TotalAmount'] = (
    transactions_with_products['Quantity'] * transactions_with_products['Price']
)
transactions_with_products
```

```
# カテゴリごとの売上高を求める
cross_tab_sales = transactions_with_products.pivot_table(
    values='TotalAmount', index='Category', aggfunc='sum'
)
cross_tab_sales
```

📝 解説

```
┌──────────────────────────────────────────────────────────────────────────────┐
transactions_with_products を、 TotalAmount の値を Category ごとに合計(sum)します。
└──────────────────────────────────────────────────────────────────────────────┘

pivot_table(values="TotalAmount", index="Category", aggfunc="sum")
            └──>(1)                └──>(2)            └──>(3)

              ┌──────────────────┐
              | 'columns'         |
──────────────┼──────────────────┤
'index'(2)    | (1)'value'        |  ← (3)'aggfunc'で設定
──────────────└──────────────────┘
```

```
transactions_with_products
      ↖
    transactions_with_products.pivot_table(values='TotalAmount',index='Category', aggfunc='sum')
                                |        ↓         ↓            └──>集約関数（合計）
                                |    集計する値    行のグループ化の基準   |
                                |      TotalAmount            Category
                                |                                      |
                                |                                      |
                                |                                      |
                                |                                      v
                                |                      aggregation function（集約関数） で、集約方法を指定。 sum（合計）を設定
                                v
        pivot_table メソッド: 引数に指定した要件に従い、集計した新しい表を作成
```

データの可視化

データ可視化は、データの特性やパターンを視覚的に理解を補助します。

```
┌──────────────────────┐
| (0) `Python`の基礎   |
└──────────────────────┘
           |
           v
┌──────────────────────┐   ┌──────────────────┐   ┌──────────────────┐   ┌──────────────────┐   ┌──────────────────────┐
| (1) データの読み込み | ---> | (2) データの操作  | ---> | (3) データ加工 | ---> | (4) 集計  | ---> ★ (5) データの可視化 ★
└──────────────────────┘   └──────────────────┘   └──────────────────┘   └──────────────────┘   └──────────────────────┘
```

基本的なプロット

◎ 棒グラフ:カテゴリごとの総販売数量

```
# カテゴリごとの総販売数量を棒グラフで可視化
category_quantity = transactions_with_products.groupby('Category')['Quantity'].sum()
category_quantity
```

```
category_quantity.plot(kind='bar', color='skyblue')
# plt.title('Total Quantity Sold by Category')
# plt.xlabel('Category')
# plt.ylabel('Total Quantity')
# plt.xticks(rotation=45)
```

📝 解説

```
┌──────────────────────────────────────────────────────────────────────┐
category_quantity を 棒グラフ（bar）として可視化、カラー（color='skyblue'）を適用します。
└──────────────────────────────────────────────────────────────────────┘

2行目コード=transactions_with_products.groupby('Category')['Quantity'].sum()
  ↓                 ┌──────>Pandas の plot メソッド: グラフとして可視化
category_quantity.plot(kind='bar', color='skyblue')
                          |          └──> color='skyblue' グラフの色をスカイブルーに指定
                          ↓
                kind='bar' 棒グラフとして描画

                  ┌──────────────────────────────────────────────┐
                   - kind（≒ 種類）→ グラフの種類を指定
                     - 'bar'（≒ 棒）→ 棒グラフ
                     - 'line'（≒ 線）→ 折れ線グラフ
                     - 'pie'（≒ 円）→ 円グラフ　など
                     補足
                     - kind を省略すると、'line' になる
                  └──────────────────────────────────────────────┘
```

🔍ちょっと豆知識

◎ `pandas` の `plot` メソッド

`pandas` の `plot` メソッドは、`matplotlib` のラッパーとして機能します。 ラッパーとは「包む」という意味で、`plot` メソッドを通じて `matplotlib` の機能を簡単に呼び出せます。

しかも、**`matplotlib` をインポートせずに使えます。ただし、インストールは必要**です。

```
┌──────────────────┐   呼び出し   ┌──────────────────────┐
│  Pandas の plot  │ ──────►  │  Matplotlib plot     │ ──────► グラフが描画される（表示）
│   ラッパー       │          │（実際の描画処理）    │
└──────────────────┘          └──────────────────────┘
```

★ `plt.title`

```
┌──────────────────────────────────────────────────┐
  タイトル（title）を追加し、グラフの内容を分かりやすくします。
└──────────────────────────────────────────────────┘

Matplotlib ライブラリ
  ↓      ┌──────>title 関数: グラフにタイトルを追加
plt.title('Total Quantity Sold by Category')
               └──> 設定するタイトルの文字列：設定したタイトルを入力する
```

★ `xlabel`、`ylabel`、`xticks`

✅グラフの軸ラベル（`xlabel`, `ylabel`）を設定し、 `xticks` を使って X 軸のラベルの回転角度を調整します。

```
┌──────────────────────────────────────────────────────────────┐
  xlabel 関数: X 軸にラベルを設定
  ↓
  plt.xlabel('Category')
                  └──> 'Category' X 軸に表示するラベル
└──────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────┐
  ylabel 関数: Y 軸にラベルを設定
  ↓
  plt.ylabel('Total Quantity')
                  └──> 'Total Quantity' Y 軸に表示するラベル
└──────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────┐
  xticks 関数: X 軸の目盛りの回転角度を設定
↓
  plt.xticks(rotation=45)
                  └──> rotation=45 X 軸のラベルを 45 度回転

                        ┌──────────────────────────────────┐
                         xticks の引数
                         plt.xticks(rotation=...) を指定し、X 軸のラベルの見やすさを調整

                           rotation を省略すると、 0 度（回転なし）になります。

                             - rotation （≒ 回転）→ X 軸のラベルの向きを調整
                             - 0 （≒ そのまま）→ ラベルを回転させずに表示
                             - 45 （≒ 45度回転）→ ラベルを 右斜め 45 度 に回転
                             - 90 （≒ 縦向き）→ ラベルを 垂直（90 度回転）に配置 など
                        └──────────────────────────────────┘
└──────────────────────────────────────────────────────────────┘
```

日付ごとの累積売上

日付ごとの累積売上を折れ線グラフで可視化します。

売上の成長傾向や増加ペースを分析し、需要の変化や市場動向を把握します。

```
# 日付ごとの累積売上を折れ線グラフで可視化
#transactions_with_products['Date'] = pd.to_datetime(transactions_with_products['Date'])
daily_sales = transactions_with_products.groupby('Date')['Price'].sum().cumsum()
daily_sales
```

FAQ

Q: `daily_sales` の内容が `Price` と表示されますが、問題ありませんか？

A: 問題ないです。`daily_sales` は `pandas.Series` オブジェクトで、その名前が `"Price"` に設定されているためです。

📝 解説

```
═══════════════════════════════════════════════════════════════════════
transactions_with_products から 日別（Date）の売上（Price）を集計し、
その合計を 累積（cumsum()）して 売上の推移 を計算します。
═══════════════════════════════════════════════════════════════════════

transactions_with_products.groupby('Date')['Price'].sum().cumsum()
                                                  └──> .cumsum： 累積和を計算するメソッド
                                                          │
                                                          ├──▶ cum： cumulative（累積の）
                                                          │      ↳ 先頭から順に値を足していく
                                                          │
                                                          └──▶ sum： sum（合計）
                                                                 ↳ 数値を合計する演算

                                                              │
                                                              ↓
┌──────────────────────────────────────────────────────────────────────┐
                          cumsum 動作イメージ
└──────────────────────────────────────────────────────────────────────┘
```

元のデータフレーム（一部）：

```
+---------------+------------+------------+-------------+
| TransactionID | Date       | Price      | Category    |
+---------------+------------+------------+-------------+
| 1001          | 2025-01-01 |  295.88    | Books       |
| 1002          | 2025-01-02 | 1306.45    | Clothing    |
| 1003          | 2025-01-03 |  432.50    | Home        |
| 1004          | 2025-01-04 | 2444.96    | Electronics |
| 1005          | 2025-01-05 | 1135.28    | Electronics |
| ・・・         | ・・・      | ・・・      | ・・・       |
+---------------+------------+------------+-------------+
```

↓ `.groupby('Date')['Price'].sum()` を実行

日ごとの売上合計：

```
+------------+----------+
|    Date    | Price    |
+------------+----------+
| 2025-01-01 |  295.88  |
| 2025-01-02 | 1306.45  |
| 2025-01-03 |  432.50  |
| 2025-01-04 | 2444.96  |
| 2025-01-05 | 1135.28  |
| ・・・      | ・・・    |
+------------+----------+
```

↓ `.cumsum()` を実行

累積売上の計算：

```
+------------+--------------+
|    Date    | Cumulative   |
|            |   Sales      |
+------------+--------------+
| 2025-01-01 |     295.88   |
| 2025-01-02 |    1602.33   |
| 2025-01-03 |    2034.83   |
| 2025-01-04 |    4479.79   |
| 2025-01-05 |    5615.07   |
| ・・・      |    ・・・     |
+------------+--------------+
```

└──┘

```
# 例：日別の売上合計を累積する
元のデータフレーム：
+----------------+------------+------------+-------------+
| TransactionID  | Date       | Price      | Category    |
+----------------+------------+------------+-------------+
| 1001           | 2025-01-01 |  295.88    | Books       |
| 1002           | 2025-01-02 | 1306.45    | Clothing    |
| 1003           | 2025-01-03 |  432.50    | Home        |
| 1004           | 2025-01-04 | 2444.96    | Electronics |
| 1005           | 2025-01-05 | 1135.28    | Electronics |
| ・・・          | ・・・      | ・・・      | ・・・       |
+----------------+------------+------------+-------------+

↓ `.groupby('Date')['Price'].sum()` を実行

日ごとの売上合計：
+------------+----------+
|    Date    | Price    |
+------------+----------+
| 2025-01-01 |  295.88  |
| 2025-01-02 | 1306.45  |
| 2025-01-03 |  432.50  |
| 2025-01-04 | 2444.96  |
| 2025-01-05 | 1135.28  |
| ・・・      | ・・・    |
+------------+----------+

↓ `.cumsum()` を実行

累積売上の計算：
+------------+--------------+
|    Date    | Cumulative   |
|            |   Sales      |
+------------+--------------+
| 2025-01-01 |     295.88   |
| 2025-01-02 |    1602.33   |
| 2025-01-03 |    2034.83   |
| 2025-01-04 |    4479.79   |
| 2025-01-05 |    5615.07   |
+------------+--------------+
```

FAQ

Q: 時系列データで累積和を計算する前にソートは必要ですか？

A: 時系列データで累積和を計算する際は、まず `sort_values()` で日付順にソートしてから `cumsum()` を適用することをお勧めします。 `groupby('Date')` の時点で、 `Date` がインデックスになります。

```
# 日付でソートしてから計算
transactions_with_products.groupby('Date')['Price'].sum().sort_index(ascending=True).cumsum()
```

```
daily_sales.plot(kind='line', marker='o', color='green') #講義用にマーカーを設定
```

```
# makerなしのほうが見やすいため
daily_sales.plot(kind='line', color='green')
```

📝 解説

`daily_sales` を 折れ線グラフ（`line`）として可視化し、

データポイントにマーカー（`marker='o'`）を追加し、線の色（`color='green'`）を指定します。

```
daily_sales.plot(kind='line', marker='o', color='green')
                 │                │            └──> 線の色を緑に指定
                 │                │
                 │                └──> 各データポイントを円（●）で表示
                 │
                 ↓
```

```
┌──────────────────────────────────────────────┐
│                 折れ線グラフとして描画                │
└──────────────────────────────────────────────┘
```

- `kind='line'` により、折れ線グラフを描画
- 引数を省略した場合は、`'line'`（線）が適用

ヒストグラム

`Pandas` の場合

販売数量の分布をヒストグラムで可視化しました。

```
# 販売数量の分布をヒストグラムで可視化
transactions['Quantity'].plot(kind='hist', bins=10)
```

📝 解説

`transactions` の `'Quantity'` 列をヒストグラム `plot` として可視化し、ビン数 `bins=10` を指定します。

```
                 Pandas の plot メソッド
                        ↓        ┌──────>kind='hist'：ヒストグラムを指定
transactions['Quantity'].plot(kind='hist', bins=10)
                 │                              ↓
                 ▼                          データを10個の範囲に分割
          'Quantity' 列を可視化対象に設定

                                                  │
                                                  ▼
```

```
┌──────────────────────────────────────────────────────────────┐
│                          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** を決定します。

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

データは `bins` で指定した数の箱に分割されます。

多いと詳細が、少ないと全体像が見えやすくなります。

学びの隣り道： seaborn の場合

- `seaborn` でのヒストグラム作成します。

```
# 販売数量の分布をヒストグラムで可視化
import seaborn as sns
sns.histplot(transactions['Quantity'], bins=10,color='orange')
```

- `matplotlib` でのヒストグラム作成

```
  # matplolibでヒストグラム作成
  plt.hist(transactions['Quantity'], bins=10)
  plt.xlabel('Quantity')
  plt.ylabel('Frequency')
  plt.title('Distribution of Quantity')
  plt.show()
```

📝 解説

`transactions` の `'Quantity'` 列を ヒストグラム `histplot` として可視化しビン数 `bins=10` を指定し、カラー `color='orange'` を適用します。

```
Seaborn ライブラリ
  ↓      ┌──────>histplot 関数: ヒストグラムを作成
sns.histplot(transactions['Quantity'], bins=10, color='orange')
               │                          ↓     └──> 色をオレンジに指定
               ▼                          データを 10 個の範囲に分割
          'Quantity' 列を可視化を対象設定
```

クロス集計の可視化

データを視覚的に表現することで、パターンや傾向を簡単に理解できます。

ヒートマップを使用します。

`Pandas` や `matplotlib` には、ヒートマップを直接描画する専用関数は存在しません。

しかし、 `Seaborn` の `sns.heatmap()` 関数を使用すれば、データの数値分布を視覚的に表現できます。

特に、クロス集計表が読みにくい場合に有効です。

機能	Pandas の Matplotlib のラッパー	Seaborn
ヒートマップ専用関数	×	✓ `sns.heatmap()`

```
×  `Pandas`の`Matplotlib`のラッパー には、ヒートマップを直接描画する専用関数は存在しない
──►  ○ `Seaborn` には `sns.heatmap()` というヒートマップを直接描画する専用関数が存在
```

```
  cross_tab_quantity
```

```
### ヒートマップを使用したクロス集計の可視化:
import seaborn as sns
#import matplotlib.pyplot as plt

sns.heatmap(cross_tab_quantity, annot=True, fmt='d', cmap='Blues')
#plt.title('Category-wise Quantity Heatmap')
```

これらの手法を使用することで、データ間の関係を明確にし、効果的な分析を行うことができます。

📝 解説

`cross_tab_quantity` を **ヒートマップ（`heatmap`）**として可視化し、 セルに数値（`annot=True`）を表示し、色（`cmap='Blues'`）を適用します。

```
sns.heatmap(cross_tab_quantity, annot=True, fmt='d', cmap='Blues')
  |     |        |                  |      |        |
  |     |        |                  |      |        └──> cmap='Blues' 色のスタイル（ブルー系）
  |     |        |                  |      └──> fmt='d' 表示する数値のフォーマット（整数）
  |     |        |                 ──> annot=True セル内に数値を表示
  |     |       ──> cross_tab_quantity ヒートマップで可視化するデータ
  |     ────┐
  |         |
  |         ▼
          heatmap関数: ヒートマップを作成
  |       ┌──────────────────────────────────────────────────────────────┐
  |        hitmapの引数
  |        ──────────────────────────────────────────────────────────────
  |        - annot（annotation：**注釈**） →True` にすると セル内に数値を表示
  |        -fmt （format：形式） → 数値の 表示フォーマット を指定
  |         -'d' （decimal：整数） → fmt='d' で 整数値として表示
  |        - cmap （colormap：色の地図）** → ヒートマップの色のパターン
  |         - 'Blues'（青系）** → 値が大きいほど濃い青になる
  |       └──────────────────────────────────────────────────────────────┘
  |
  └──>  Seaborn ライブラリ：データの海を可視化する Python ライブラリ
                          Seaborn
                             |
    ┌──────────────────────────────────────────────────┐
    |                                                  |
    ↓                                                  ↓
    Sea（データの海）                        Born（生まれる / 生成）
```

おわりに

学んだことの振り返り

```
┌──────────────────────┐
| (0) 🐍 `Python`の基礎  |
└──────────────────────┘
          |
          v
┌──────────────────────┐  ┌──────────────────────┐  ┌──────────────────────┐  ┌──────────────────────┐  ┌──────────────────────┐
| (1) 📂 データの読み込み | ---> | (2) ✏ データの操作  | ---> | (3) 🔧 データ加工 | ---> | (4) 📊 集計   | ---> | (5) 📈 データの可視化 |
└──────────────────────┘  └──────────────────────┘  └──────────────────────┘  └──────────────────────┘  └──────────────────────┘
```

0. 🐍 なぜ、`Python`？変数、データの型と構造、ライブラリなど
1. 📂 `CSV`や`Excel`のデータ読み込みなど
2. ✏ データの確認、行列の参照など
3. 🔧列の追加、結合など
4. 📊単純集計とクロス集計ど

5. 📈棒グラフ、ヒストグラム、ヒートマップなど

   こられを通じて、`Python`のコードの構造と仕組みを学びました。

次のステップ

まず、**本講義の復習**です。**今回学んだ内容を確実に自分のもの**にすることが大切です。

次は、**本講義に立ち戻り**ながら、Excelで**既に行ったデータ操作や視覚化をPythonで実現**することに取り組みましょう。

ただし、**Python プログラミングから学ぶ**のが、長期的には最も効果的です。

基礎があれば、次のステップに進む際もスムーズに成長できます。

```
+-----------+                          ┌──────────» 📗Excel の操作を再現
|           |                          |             ・📁データの読み込み
|           |──┬                       |               ・✏データの操作
| 🐼Pandas  |  |                       |             ・🔧データ加工
|           |  |                       |               ・📊基本集計とクロス集計
+-----------+  |──────▶ 📖本講義の復習    ・📈データの可視化
|           |  |                       |
| 📖基礎     |  |                       |
|           |──┴                       |
|           |                          └──────────» Python プログラミングを学ぶ
+-----------+                                       ・🔢 データ型
                                                    ・📦 変数
                                                    ・🔀 条件分岐
                                                    ・🔁 ループ
                                                    ・🛠 関数
                                                    ・📦 オブジェクト など
```