# 第4回 データ構造化の代表的なデータ操作方法

第4回では、データ構造化プログラムにおける代表的なデータ操作の方法についてご紹介します。

今回学ぶ内容は、次のとおりです。

- pandasによるデータ操作
- 辞書・リストの操作

はじめに、本講義で使用するファイルを皆さんの環境にダウンロードするため、次のコードを実行してください。

In [ ]:

```
!wget https://github.com/tendo-sms/python_seminar_2022/raw/main/lecture4/files.zip .
!unzip files.zip
!mv files/* .
```

# pandasによるデータ操作

第3回では、pandasによるCSVファイルの入出力と、locによる値の編集について学びました。

今回は、さらにpandasの使用方法を掘り下げていきます。次のような内容をご紹介します。

- pandasで取り扱うデータ形式
- データの削除(drop、dropna、drop_duplicates)
- 欠損値の補完(fillna)
- データの並び替え
- データの抽出・置き換え
- 行ごと・列ごとの計算

pandasには、今回の限られた時間ではご紹介しきれないのほど、たくさんの機能があります。

ここでは、構造化プログラムで取り扱う実験データをイメージしたサンプルデータを例として、様々な加工を加えていくことで、よく使用される代表的な機能をご紹介します。

## pandasとは (おさらい)

pandasとは、第3回でご紹介したとおり、「配列データの整形・加工」を得意とするパッケージです。

データ構造化プログラムでは、機器が出力したCSVデータを整形・加工してメタデータとして登録するなどの目的で、pandasをよく利用します。

## pandasで取り扱うデータ形式

第3回では、padansのread_csv関数でCSVファイルからデータを読み込むと、データフレームという特別な配列に格納されるとご紹介しました。

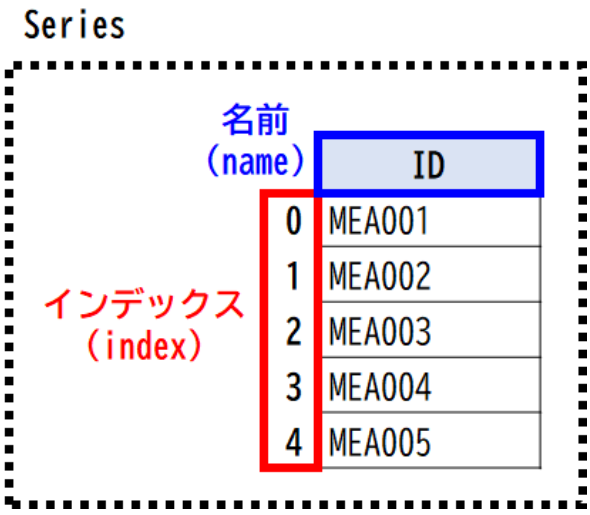
pandasで取り扱うデータ形式には、上述のデータフレーム(DataFrame)を含めて、「**Series**」「**DataFrame**」の2つの形式があります。

### Series

Excelファイル中の1行や1列のデータのような、「**1次元のデータ配列**」を表します。

Seriesは、配列の実体に加えて、付加情報として「**名前(name)**」と「インデックス(index)**」を持ちます。

Seriesのイメージは次のとおりです。



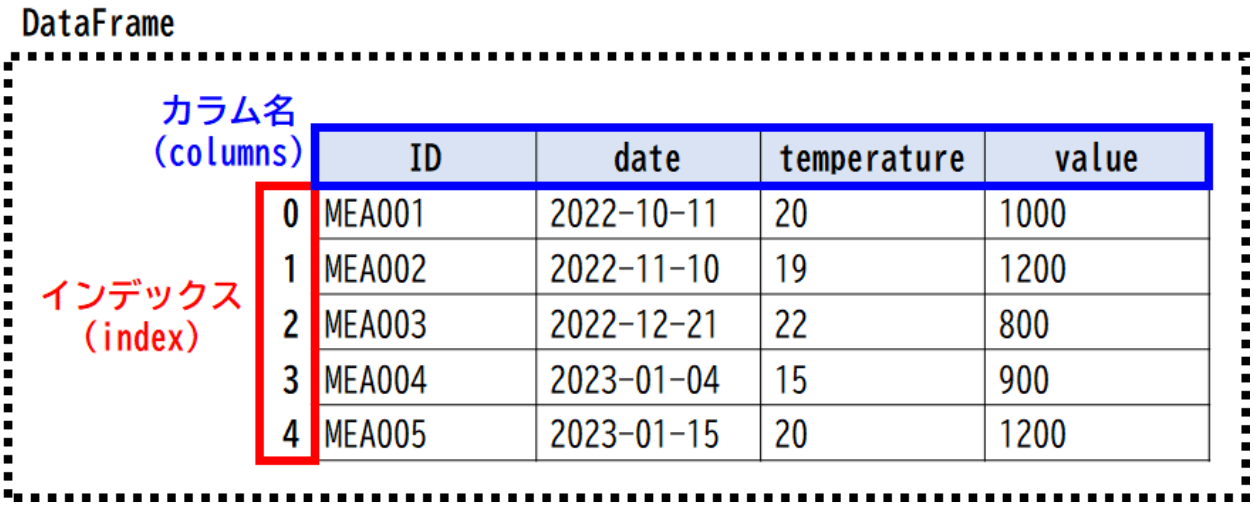

Seriesのデータを新規作成することはあまりありませんが、後述のデータフレームから1行ぶんのデータを抽出した結果、Series型となることはよくあります。

### DataFrame

Excelファイル中のテーブルデータのような、「**2次元のデータ配列**」を表します。

DataFrameは、配列の実体に加えて、付加情報として「**カラム名(columns)**」と「インデックス(index)**」を持ちます。ちょうど、Excelの列名(A、B、・・・)と行番号(1、2、・・・)のイメージです。

DataFrameのイメージは次のとおりです。




## 今回のサンプルデータ

今回は、次のような内容のCSVファイル(pandas_sample1.csv)を読み込んで、データの加工を行います。

| measureID | date | operator | temperature | measureValue | measureUnit |
|---|---|---|---|---|---|
| MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| MEA002 | 2022/11/2 | Yamada | 20 | 999 | sec |
| MEA005 | 2022/11/4 | Sato | R.T | 98 | min |
| | | | | | |
| MEA003 | 3-Nov-22 | Adam | 18 | | hour |
| MEA004 | 11-05-2022 | Yamada | | Failure | |
| MEA006 | 11-05-2022 | | 15 | 8 | hour |

# pandasによるCSVファイルの読み込み (おさらい)

今回も第3回と同様に、CSVからデータフレームにファイルを読み込んで、様々なデータ操作を行います。

次のソースコードを実行して、pandas_sample1.csvを読み込みましょう。今回のデータは日本語データを含んでいないため、encodingオプションは指定していません。

In [ ]:

```
import pandas as pd

df_init = pd.read_csv("pandas_sample1.csv")
print(df_init)
```

# (データの加工に入る前に) Google Colaboratoryの補足説明

このあと、上記で読み込んだデータフレームに対してデータの加工を行っていきますが、その前に、Google Colaboratoryの仕様について補足します。

Google Colaboratoryでは、ページ内の全てのソースコードはつながっています。

例えば、上記のソースコードで作成したデータフレームdf_initは、後続のソースコードでも参照できます。

In [ ]:

```
print(df_init)
```

前回までの講義では、混乱を防止するため、ソースコードごとに変数名を変えてすべてのデータを独立させることで、この仕様を意識しなくてよいサンプルコードとしていました。

今回のpandasのご紹介では、上記のソースコードで読み込んだデータフレームdf_initに対して、このあとステップバイステップで様々な加工を行っていきます。

このため、次の点に注意してください。

- サンプルコードごとに毎回CSVファイルからデータフレームへデータを読み込むことはしません。
- **ソースコードを動かすとき、それ以前のソースコードが一度も動いていないと正しく動作しないことがあります。**「pandasによるデータ操作」のソースコードは、必ず上から順番に実行するのが確実です。

# 空行の削除

インデックス4の行は全てのカラムに値がありませんので、不要な空行と考えられます。

| | measureID | date | operator | temperature | measureValue | measureUnit |
|---|---|---|---|---|---|---|
| **0** | MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| **1** | MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| **2** | MEA002 | 2022/11/2 | Yamada | 20 | 999 | sec |
| **3** | MEA005 | 2022/11/4 | Sato | R.T | 98 | min |
| **4** | | | | | | |
| **5** | MEA003 | 3-Nov-22 | Adam | 18 | | hour |
| **6** | MEA004 | 11-05-2022 | Yamada | | Failure | |
| **7** | MEA006 | 11-05-2022 | | 15 | 8 | hour |

このように欠測値を含む行は、「**dropnaメソッド**」で削除することができます。

メソッドの引数に「**how="all"**」と指定すると、すべてのカラムが値なしの場合(空行の場合)に、その行を削除します。

```
変数 = データフレーム.dropna(how="all")
```

最初に読み込んだデータフレームdf_initのdropnaメソッドを呼び出して、その動作を見てみましょう。

In [ ]:

```
print("変更前のデータフレーム")
print(df_init)

# 空行の削除
df_empty_line = df_init.dropna(how="all")

print("変更後のデータフレーム")
print(df_empty_line)
```

ここで注意が必要なのは、「df_init.dropna(how="all")」としたとき、**df_init自体の内容が変更されるわけではない**という点です。

**df_initの内容を元に変更が加えられた、新しいデータフレームが戻り値として得られます。**

上記のソースコードでは、戻り値として得られた新たなデータフレームを変数df_empty_lineに格納しています。

実際に、現在のdf_initの内容を見てみましょう。

In [ ]:

```
print(df_init)
```

元のdf_initは、インデックス4の空行が削除されずに残っていることが分かります。

このように、データフレームを加工するメソッドを呼び出すたびに新しいデータフレームが作成され、メモリも多く消費します。

そこで次のように、メソッドの引数に「inplace=True」と指定すると、元のデータフレーム自体が変更されます。

```
データフレーム.dropna(how="all", inplace=True)
```

dropnaも含めた、今回ご紹介する様々な加工メソッドは、ほとんどが「inplace=True」を指定できます。

加工前の値を取っておく必要がなければ、基本的には常に「inplace=True」を付けておくことでよいでしょう。

ただし今回の講義ではGoogle Colaboratoryを使っているため、「inplace=True」でデータフレームを直接加工すると、前に戻ってプログラムを再実行すると2重3重に加工がされてしまうなど、混乱が生じてしまいます。

そのため、今回の講義の例題では「inplace=True」を指定しないことにします。

## 重複行の削除

インデックス0と1の行は、measureIDも含めて全く同じ内容ですので、誤って重複登録されたものと考えられます。

| | measureID | date | operator | temperature | measureValue | measureUnit |
|---|---|---|---|---|---|---|
| **0** | **MEA001** | **2022/11/1** | **Suzuki** | **25** | **1000** | **sec** |
| **1** | **MEA001** | **2022/11/1** | **Suzuki** | **25** | **1000** | **sec** |
| **2** | MEA002 | 2022/11/2 | Yamada | 20 | 999 | sec |
| **3** | MEA005 | 2022/11/4 | Sato | R.T | 98 | min |
| **5** | MEA003 | 3-Nov-22 | Adam | 18 | | hour |
| **6** | MEA004 | 11-05-2022 | Yamada | | Failure | |
| **7** | MEA006 | 11-05-2022 | | 15 | 8 | hour |

このような重複行は、「**drop_duplicatesメソッド**」で削除することができます。

```
変数 = データフレーム.drop_duplicates()
```

In [ ]:

```
print("変更前のデータフレーム")
print(df_empty_line)

# 重複行の削除
df_dup = df_empty_line.drop_duplicates()

print("変更後のデータフレーム")
print(df_dup)
```

## 行を指定しての削除

インデックス6の行はmeasureValueがFailureなので、不要な行とみなして削除しましょう。

| | measureID | date | operator | temperature | measureValue | measureUnit |
|---|---|---|---|---|---|---|
| **0** | MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| **2** | MEA002 | 2022/11/2 | Yamada | 20 | 999 | sec |
| **3** | MEA005 | 2022/11/4 | Sato | R.T | 98 | min |
| **5** | MEA003 | 3-Nov-22 | Adam | 18 | | hour |
| **6** | **MEA004** | **11-05-2022** | **Yamada** | | **Failure** | |
| **7** | MEA006 | 11-05-2022 | | 15 | 8 | hour |

「**dropメソッド**」を使用すると、行を指定して削除することもできます。行の指定は、インデックスを用います。

```
変数 = データフレーム.drop(削除したい行のインデックス)
```

今回はインデックス6を指定します。

In [ ]:

```
print("変更前のデータフレーム")
print(df_dup)

# 行を指定しての削除
df_drop = df_dup.drop(6)

print("変更後のデータフレーム")
print(df_drop)
```

## 欠測値の補完

現在のデータフレームは、いくつかデータのない欠測値(プログラムの出力結果ではNaNと表示)があります。

| | measureID | date | operator | temperature | measureValue | measureUnit |
|---|---|---|---|---|---|---|
| **0** | MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| **2** | MEA002 | 2022/11/2 | Yamada | 20 | 999 | sec |
| **3** | MEA005 | 2022/11/4 | Sato | R.T | 98 | min |
| **5** | MEA003 | 3-Nov-22 | Adam | 18 | | hour |
| **7** | MEA006 | 11-05-2022 | | 15 | 8 | hour |

欠測値のままだと、データ構造化において計算結果が期待通りにならないなど、不都合が生じる場合があります。

そのような場合は、「**fillnaメソッド**」を使って欠測値を補完できます。

fillnaメソッドの引数に置き換えたい値のみを指定すると、データフレーム中のすべての欠測値が、指定した値に置き換えられます。

変数 = データフレーム.fillna(置き換えたい値)

ですがこの方法だと、すべてのカラムについて欠測値が置き換わります。現実的には、数値のカラムと文字列のカラムはそれぞれ別の値に置き換えたい、などのケースがほとんどでしょう。

そこでfillnaメソッドの引数に辞書を指定すると、列ごとに異なる値に置き換えることができます。

変数 = データフレーム.fillna({置き換えたい列1:置き換えたい値1, 置き換えたい列2:置き換えたい値2, ・・・})

今回は、operator列の欠測値を"NO NAME"に、measureValue列の欠測値を0に、それぞれ置き換えてみます。

In [ ]:

```
print("変更前のデータフレーム")
print(df_drop)

# 欠測値の補完
df_fill_dict = df_drop.fillna({"operator": "NO NAME", "measureValue": 0})

print("変更後のデータフレーム")
print(df_fill_dict)
```

## データの並べ替え

「**sort_valuesメソッド**」を使用すると、指定したカラムをキーとして表を並び替えることができます。

変数 = データフレーム.sort_values(by=キーとなるカラム名)

なお、デフォルトは昇順のソートです。降順に並べ替えるには、sort_valuesメソッドの引数に「**ascending=False**」を指定します。

変数 = データフレーム.sort_values(by=キーとなるカラム名, ascending=False)

ここでは、measureIDをキーとして昇順に並び替えてみます。

In [ ]:

```
print("変更前のデータフレーム")
print(df_fill_dict)

# データの並べ替え
df_sort = df_fill_dict.sort_values(by="measureID")

print("変更後のデータフレーム")
print(df_sort)
```

## インデックスの振り直し

ここまで様々な加工を行った結果、インデックスが歯抜けになり、順序も昇順ではなくなっています。

| | measureID | date | operator | temperature | measureValue | measureUnit |
|---|---|---|---|---|---|---|
| 0 | MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| 2 | MEA002 | 2022/11/2 | Yamada | 20 | 999 | sec |
| 5 | MEA003 | 3-Nov-22 | Adam | 18 | 0 | hour |
| 3 | MEA005 | 2022/11/4 | Sato | R.T | 98 | min |
| 7 | MEA006 | 11-05-2022 | NO NAME | 15 | 8 | hour |

そこで、「**reset_indexメソッド**」を使用すると、インデックスを振り直すことができます。

変数 = データフレーム.reset_index()

In [ ]:

```
print("変更前のデータフレーム")
print(df_sort)

# インデックスの振り直し
df_sort_r = df_sort.reset_index()

print("変更後のデータフレーム")
print(df_sort_r)
```

新しく連番のインデックスができましたが、もともとあったインデックスがカラム名"index"の新たなカラムとして残ってしまいました。

これを残さないようにするには、reset_indexメソッドの引数に「**drop=True**」を指定します。

変数 = データフレーム.reset_index(drop=True)

今度はdrop=Trueの指定ありで実行してみましょう。

In [ ]:

```
print("変更前のデータフレーム")
print(df_sort)

# インデックスの振り直し
df_sort_r = df_sort.reset_index(drop=True)

print("変更後のデータフレーム")
print(df_sort_r)
```

これで、インデックスの振り直しができました。

# データの抽出・置き換え (おさらい＋α)

ここからは、特定のデータを抽出して、値を置き換える方法をご紹介します。

具体的には、次の赤字で示す「R.T」の部分を抽出して、数値の「25」に置き換えます。

| | measureID | date | operator | temperature | measureValue | measureUnit |
|---|---|---|---|---|---|---|
| 0 | MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec |
| 1 | MEA002 | 2022/11/2 | Yamada | 20 | 999 | sec |
| 2 | MEA003 | 3-Nov-22 | Adam | 18 | 0 | hour |
| 3 | MEA005 | 2022/11/4 | Sato | R.T | 98 | min |
| 4 | MEA006 | 3-Nov-22 | NO NAME | 15 | 8 | hour |

第3回でもlocを用いて同様の加工を行いました。そのおさらいに加え、また別の方法もご紹介します。

## loc

「**loc**」は、データフレーム中の単一の要素、または複数の要素を抽出することができます。

locによる抽出は、次のとおりです。インデックスおよびカラム名を指定して抽出します。

データフレーム.loc[インデックス, カラム名]

次のとおり指定することで、複数要素の抽出も可能です。このとき、**「カラム名2」「インデックス2」も含めた要素を抽出します。**

データフレーム.loc[インデックス1:インデックス2, カラム名1:カラム名2]

要素を抽出する例を次に示します。

In [ ]:

```
print("単一の要素を抽出")
print(df_sort_r.loc[3, "temperature"])

print("複数要素を抽出")
print(df_sort_r.loc[1:3, "operator":"measureValue"])
```

### iloc

「**iloc**」は、locと同様の抽出ができますが、行番号および列番号を指定します。

```
データフレーム.iloc[行番号, 列番号]
```

複数要素の指定は、次のとおりです。ただしlocとは異なり、**「列番号2」「行番号2」を含まない、ひとつ手前までの要素を抽出します。**混乱しやすいので、注意してください。

```
データフレーム.iloc[行番号1:行番号2, 列番号1:列番号2]
```

要素を抽出する例を次に示します。

In [ ]:

```
print("単一の要素を抽出")
print(df_sort_r.iloc[3, 3])

print("複数要素を抽出")
print(df_sort_r.iloc[1:4, 2:5])
```

値の置き換えるときは、抽出した結果に代入式で置き換えたい値を設定します。

```
データフレーム.loc[インデックス, カラム名] = 置き換えたい値
データフレーム.iloc[行番号, 列番号] = 置き換えたい値
```

**locおよびilocで抽出した要素に値を代入すると、これまで行ってきた各種メソッドによる加工とは異なり、元のデータフレーム自体が変更されます。**

ここでは実際に、ilocを使って値を置き換えてみます。

前述のとおり、対象のデータフレーム自身を変更するとGoogle Colaboratory上では混乱の元となるため、先に「**copyメソッド**」を使ってデータフレームのコピーを作成し、そのコピーに対して値の変更を行います。

In [ ]:

```
# 先にデータフレームのコピーを作成
df_iloc = df_sort_r.copy()

print("変更前のデータフレーム")
print(df_iloc)

# コピーに対して値の変更を行う
df_iloc.iloc[3, 3] = 25

print("変更後のデータフレーム")
print(df_iloc)
```

### at、iat

locおよびilocに近い機能として、atおよびiatがあります。

atおよびiatは、単一要素の抽出・置き換えしかできません。抽出する箇所の指定方法は、locおよびilocで単一要素の抽出を行うときと同じです。

```
at[インデックス, カラム名]
iat[行番号, 列番号]
```

at、iatでできることはlocおよびilocでできますので、基本的にはlocおよびilocを覚えれば問題ありません。

ただし、単一要素の抽出においてはlocおよびilocよりも、atおよびiatの方が高速です。速度を重視するプログラムでは、atおよびiatの利用も検討してみてください。

## 列の追加

次の赤字部分で示すように、データフレームに新たな列を追加してみましょう。

| | measureID | date | operator | temperature | measureValue | measureUnit | secVal |
|---|---|---|---|---|---|---|---|
| **0** | MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec | **1000** |
| **1** | MEA003 | 2022/11/2 | Yamada | 20 | 999 | sec | **999** |
| **2** | MEA004 | 3-Nov-22 | Adam | 18 | 0 | hour | **0** |

| | measureID | date | operator | temperature | measureValue | measureUnit | secVal |
|---|---|---|---|---|---|---|---|
| 3 | MEA005 | 2022/11/4 | Sato | 25 | 98 | min | 5880 |
| 4 | MEA006 | 3-Nov-22 | NO NAME | 15 | 8 | hour | 28800 |

次のとおり記述することで、列を末尾に追加できます。

　　データフレーム名[追加したいカラム名] = 追加する値のリスト

実際に、列を追加してみます。

In [ ]:

```
# 先にデータフレームのコピーを作成
df_cadd = df_iloc.copy()

print("変更前のデータフレーム")
print(df_cadd)

# 列の追加
df_cadd["secVal"] =[1000, 999, 5880, 32400, 28800]

print("変更後のデータフレーム")
print(df_cadd)
```

## 行ごと・列ごとの計算

最後に、行ごとや列ごとの計算方法を示します。

ここでは、新たに次のようなデータを例に説明します。

| measureID | value1 | value2 | value3 | measureUnit |
|---|---|---|---|---|
| MEA001 | 1200 | 800 | 1200 | sec |
| MEA002 | 1100 | 1100 | 1000 | sec |
| MEA003 | 1200 | 900 | 1000 | sec |
| MEA004 | 800 | 1100 | 1200 | sec |
| MEA005 | 900 | 1000 | 1100 | sec |

様々なメソッドを用いて、行ごと・列ごとの計算ができます。

ここでは、メソッドの一部として次の機能をご紹介します。

| メソッド | 機能 |
|---|---|
| sum | 合計値 |
| mean | 平均値 |
| max, min | 最大値、最小値 |

以下のようにメソッドを呼び出します。

　　データフレーム.メソッド名(引数)

まずは、列方向の計算を実際に行ってみましょう。

In [ ]:

```
import pandas as pd

df_calc = pd.read_csv("pandas_sample2.csv")

print(df_calc)

print("列の合計値を計算")
print(df_calc.sum(numeric_only=True))

print("列の平均値を計算")
print(df_calc.mean(numeric_only=True))

print("列の最大値を計算")
print(df_calc.max(numeric_only=True))
```

ちなみに、計算結果は1次元となりますのでSeries型となります。

なお、各メソッドの引数に「numeric_only=True」を指定しています。これにより、数値のみを対象として計算を行うことができます。(measureIDやmeasureUnitは計算対象外となる)

行方向に計算する場合は、引数に「axis=1」を指定します。

In [ ]:

```
import pandas as pd

df_calc2 = pd.read_csv("pandas_sample2.csv")

print(df_calc2)
print("行の合計値を計算")
print(df_calc2.sum(axis=1, numeric_only=True))
print("行の平均値を計算")
print(df_calc2.mean(axis=1, numeric_only=True))
print("行の最大値を計算")
print(df_calc2.max(axis=1, numeric_only=True))
```

以上で、pandasの機能の解説を終わります。

冒頭でもご説明したとおり、pandasには、まだまだご紹介しきれないほどたくさんの機能があります。

何か配列データの操作を行いたいときは、自力でプログラムを作成する前に、panadsの便利な機能を使って一発で実現できないか、ぜひ調べてみてください。

## pandas vs Excel (+VBA)

配列データの操作というと、Excelを使えば十分では？と思われたかもしれません。

もちろん今回ご紹介した例のように、加工する場所や内容がピンポイントで分かっていて、数も少なければ、Excelでもできるでしょう。

しかし実際の構造化プログラムでは、大量のデータの中から加工が必要な箇所を見つけて、状況に応じて適切な内容で加工しなければなりません。そうなると必然的にプログラムで処理することになるので、pandasを利用するべきでしょう。

また、プログラムで加工するにしても、Excel VBAでよいのでは？とも思われたかもしれません。

確かにExcel VBAも配列データの操作が得意なプログラミング言語であり、pandasとVBAでは、慣れや好みの差しかないかもしれません。

ですが、データ構造化プログラムで行うのは配列データの操作だけではありません。ファイル入出力やメタデータの作成、グラフ描画など様々な処理があり、Pythonの豊富なパッケージ・モジュールを使って実現します(前回アピールしたimportの素晴らしさを思い出してください！)。

その流れの中で、配列データの操作だけVBAで行うのは非効率ですし、メンテナンスも大変ですよね。Pythonで一気通貫のプログラムとした方が、将来の再利用・保守まで含め、メリットが大きいのは間違いありません。

【小ネタ】

あくまで個人的な体験・感想ですが・・・

Excelをバージョンアップしたことで今までのVBAが動かなくなってしまった、というトラブルをよく見てきました。プログラムを修正できる人がおらず、サポートの切れた古いExcelが動作する環境をずっと持ち続ける、というケースもあります。

もちろんPythonでも同じようなケースはありえますが、サブスクリプション化が進むExcelと違って、それほど頻繁にバージョンアップの必要性に見舞われることはありません。また、1台のマシンに複数バージョンのPythonを混在させることも可能です。

VBAは歴史のあるプログラミング言語ですが、設計の古さを指摘する声もあります。近年ではExcelでVBAの代わりにJavaScriptが使えるようになる(Office スクリプト)など、Officeマクロが転換期を迎えつつある・・・のかもしれません。

## [ご参考] 紹介しきれなかったpandasの機能

ここでは、今回ご紹介しきれなかったpandasの機能について、キーワードのみですがご紹介します。

ここでご紹介した機能を使いたい場面にぶつかったら、ここで挙げたキーワードをヒントにpandasの公式リファレンス等を調べてみてください。

| やりたいこと | キーワード(メソッド名等) |
|---:|---:|
| CSVファイルの特定の列を、データフレームのインデックスにする | read_csv関数のindex_col引数 |
| ヘッダのないCSVファイルを読み込んでデータフレームにする | read_csv関数のheader=None引数 |
| Excelファイル(.xlsx)を読み込んでデータフレームにする | read_excel関数 |
| 複数行を一度に削除する | dropメソッドの引数にインデックスのリストを指定 |
| 欠測値を1行前の値で置き換える | fillnaメソッドの引数に「method="ffill"」または「method="pad"」を指定 |
| 行と列を入れ替える | データフレーム.T または transposeメソッド |

## 練習問題

「pandasによるデータ操作」の最後に、練習問題にチャレンジしてみましょう。

カレントディレクトリに、「practice1.csv」というファイルがあります。内容は次のとおりです。なお、文字エンコーディングはCP932です。

| measureID | date | operator | temperature | measureValue | measureUnit | memo |
|---|---|---|---|---|---|---|
| MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec | |
| MEA002 | 2022/11/2 | Yamada | 20 | Failure | sec | 実験失敗 |
| MEA005 | 2022/11/6 | Sasaki | 15 | 8 | hour | |
| MEA004 | 2022/11/5 | Adam | | Failure | sec | 実験失敗 |
| MEA003 | 2022/11/4 | Sato | R.T | 900 | sec | |

次の操作を行ってみてください。

- ファイルを読み込んでデータフレームを作成する。
- measureValueが"Failure"の行は削除する。
- measureUnitが"hour"の行について、次の変更を行う。
  - measureValueを秒単位の値に置き換える。
  - measureUnitを"sec"に置き換える。
- measureIDの昇順にソートする。

結果が次のとおりとなることを確認してください。

| measureID | date | operator | temperature | measureValue | measureUnit | memo |
|---|---|---|---|---|---|---|
| MEA001 | 2022/11/1 | Suzuki | 25 | 1000 | sec | |
| MEA003 | 2022/11/4 | Sato | R.T | 900 | sec | |
| MEA005 | 2022/11/6 | Sasaki | 15 | 28800 | sec | |

In [ ]:

```
import pandas as pd

# 文字コードがCP932なのでencodingの指定が必要です

df_practice = pd.read_csv("practice1.csv", encoding="cp932")
print(df_practice)

# インデックス1の行と3の行を削除
#   ※ 以下でもOKです
#      df_drop2 = df_practice.drop([1, 3])

df_drop1 = df_practice.drop(1)
df_drop2 = df_drop1.drop(3)

# 時間→秒への変換
#   ※ 以下でもOKです
#      df_drop2.loc[2, "measureValue":"measureUnit"] = [8 * 60 * 60, "sec"]

df_drop2.loc[2, "measureValue"] = 8 * 60 * 60
df_drop2.loc[2, "measureUnit"] = "sec"

# measureID列で昇順にソート

df_sorted = df_drop2.sort_values(by="measureID")

print(df_sorted)
```

# 辞書・リストの操作

第1回の講義で「複数の値からなるデータ型」として辞書とリストをご紹介しました。

この2つのデータ型は、構造化プログラムにおいて非常によく使われるデータ型です。

ここでは、辞書およびリストについてもう少し深堀りした使い方をご紹介します。

## 辞書について (おさらい)

辞書について、簡単に復習します。

辞書はキー(key)と値(value)のペアを複数集めたデータです。

- 辞書は、次の形式で表現します。,(カンマ)や:(コロン)の前後に空白を入れても構いません。

  {キー1:値1, キー2:値2, キー3:値3, ・・・}

- キーには文字列をはじめ、何種類かの型を使用できます。
- 値には、任意の型を使用できます。
- 辞書に含まれるキーと値のペアには、基本的に順序(何番目のペア、などの概念)はありません。(厳密には順序を意識したプログラムも作成できますが、本セミナーでは詳しく触れません)
- 値の重複を許可しますが、キーの重複は許可しません。

辞書の値を取得するには、次のような指定方法があります。

(1) 辞書[キー名]
(2) 辞書.get(キー名)
(3) 辞書.get(キー名、デフォルト値)

- (1)では、存在しないキー名を指定するとエラーとなります。
- (2)では、存在しないキー名を指定するとNone(値がないことを示す)が取得されます。
- (3)では、存在しないキー名を指定するとデフォルト値が取得されます。

次のプログラムを実行して、辞書の作成・値の取得の動作を再確認しましょう。

In [ ]:

```
# 辞書を作成する
measure_dict = {"date": "2023/1/26", "temperature": "R.T", "operator": "鈴木", "measureValue": 1000, "measureUnit": "sec"}

# 辞書の内容を画面に出力する
print(measure_dict)

# (1) 辞書名[キー名]
print("(1) キー名に対応する値を取得する")
print(measure_dict["date"])

# (2) 辞書名.get(キー名)
print("(2) get(キー名)では存在しないキーを指定するとNoneとなる")
print(measure_dict.get("temperature"))
print(measure_dict.get("tmp"))

# (3) 辞書名.get(キー名、デフォルト値)
print("(3) get(キー名, デフォルト値)では存在しないキーを指定するとデフォルト値となる")
print(measure_dict.get("measureValue", 0))
print(measure_dict.get("mValue", 0))
```

## 辞書の操作

今回は、より実践的な辞書の使い方として、次の操作をご紹介します。

1. 辞書の作成・要素の追加
2. 辞書の値を変更
3. 辞書の要素を削除
4. 辞書を空にする
5. 辞書の値の取り出し

### 辞書の作成・要素の追加

作成した辞書に要素を追加します。

構造化プログラムでメタデータとして辞書を作成する場合などは、最初に空の辞書を作成し、それに対してキーと値を次々と追加していくことが多いです。

そこでまずは、空の辞書を作成します。何も要素がないので、単に「{}」とすればOKです。

```
辞書を格納する変数 = {}
```

要素の追加方法は、次のとおりです。

```
辞書[追加したいキー名] = 追加したい値
```

空の辞書を作成し、そこにキーと値を次々と追加していく例を、次のソースコードで示します。

In [ ]:

```
# 空の辞書を作成
meta_dict_add = {}

print("要素追加前の辞書")
print(meta_dict_add)

# 要素を追加する
meta_dict_add["date"] = "2023/1/26"
meta_dict_add["temperature"] = "R.T"
meta_dict_add["operator"] = "鈴木"
meta_dict_add["measureValue"] = 1000
meta_dict_add["measureUnit"] = "sec"

print("要素追加後の辞書")
print(meta_dict_add)
```

### 辞書の値を変更

次に、辞書の特定の値を変更します。値の変更方法は、次のとおりです。

```
辞書[変更したい要素のキー名] = 変更後の値
```

辞書の値の追加と構文自体は同一ですが、指定したキー名が存在しなければ値の追加、存在していれば値の変更となります。

辞書の値を変更するソースコードの例を、次に示します。

In [ ]:

```
# 辞書を作成
dict_mod = {"date": "2023/1/26", "temperature": "R.T", "operator": "鈴木", "measureValue": 1000, "measureUnit": "sec"}

print("変更前の辞書")
print(dict_mod)

# 辞書の値を変更
dict_mod["temperature"] = 25

print("変更後の辞書")
print(dict_mod)
```

この例では、文字列だった値を整数値にしました。

このように、データ型の異なる値に変更しても問題ありません。

### 辞書の要素を削除

辞書の要素を削除します。特定の要素を削除する場合は、「**del文**」を使用します。

```
del 辞書[削除したいキー名]
```

なお、辞書に存在しないキー名を指定するとエラーとなります。

要素を削除する例を、次に示します。

In [ ]:

```
# 辞書を作成
dict_del = {"date": "2023/1/26", "temperature": "R.T", "operator": "鈴木", "measureValue": "1000", "measureUnit": "sec"}

print("変更前の辞書")
print(dict_del)

# 辞書の要素を削除
del dict_del["temperature"]

print("変更後の辞書")
print(dict_del)
```

### 辞書を空にする

「**clearメソッド**」を使用すると、辞書を空にすることができます。

```
辞書.clear()
```

例を次に示します。

In [ ]:

```
# 辞書を作成
meta_dict_clr = {"date": "2023/1/26", "temperature": "R.T", "operator": "鈴木", "measureValue": "1000", "measureUnit": "sec"}

print("変更前の辞書")
print(meta_dict_clr)

# 辞書を空にする
meta_dict_clr.clear()

print("変更後の辞書")
print(meta_dict_clr)
```

### 辞書の要素の取り出し

要素の削除と関連した機能として、「**popメソッド**」をご紹介します。

popメソッドを使用すると、指定したキーの要素が辞書から削除され、戻り値として削除した値が取得されます。

取り出した値を格納する変数 = 辞書.pop(値を取り出したいキー名, デフォルト戻り値)

存在しないキー名を指定した場合は、デフォルト戻り値が変数に格納されます。

例を次に示します。

In [ ]:

```
# 辞書を作成
meta_dict_del = {"date": "2023/1/26", "temperature": "R.T", "operator": "鈴木", "measureValue": "1000", "measureUnit": "sec"}

print("変更前の辞書")
print(meta_dict_del)

# 存在するキーを指定
ret_value = meta_dict_del.pop("date", "default_value")

print("存在するキー(date)を指定 - 戻り値と辞書の内容を確認")
print(ret_value)
print(meta_dict_del)

# 存在しないキーを指定
ret_value = meta_dict_del.pop("datetime", "NOT EXIST")

print("存在しないキーを指定 - 戻り値と辞書の内容を確認")
print(ret_value)
print(meta_dict_del)
```

## 辞書の要素をfor文で参照する

for文を用いて、辞書のキーや値を一つずつ取り出して処理することができます。

次のような機能を利用できます。

(1) キーの一覧を取得

```
for キーを格納する変数 in 辞書.keys():
    繰り返し処理
```

(2) 値の一覧を取得

```
for 値を格納する変数 in 辞書.values():
    繰り返し処理
```

(3) キーと値のペアの一覧をタプルで取得

```
for キーと値のペアのタプルを格納する変数 in 辞書.items():
    繰り返し処理
```

それぞれの例を、次に示します。

In [ ]:

```
# 辞書を作成
meta_dict_for = {"date": "2023/1/26", "temperature": "R.T", "operator": "鈴木", "measureValue": "1000", "measureUnit": "sec"}

print("(1) キーの一覧を取得")
for key in meta_dict_for.keys():
    print(key)

print("-----------------------------------")

print("(2) 値の一覧を取得")
for value in meta_dict_for.values():
    print(value)

print("-----------------------------------")

print("(3) キーと値のペアの一覧をタプルで取得")
for pair in meta_dict_for.items():
    print(pair)
```

なお「(3) キーと値のペアの一覧をタプルで取得」については、for文を次のとおり記述することで、キーと値を変数に分解して取得することもできます。

In [ ]:

```
# 辞書を作成
meta_dict_for2 = {"date": "2023/1/26", "temperature": "R.T", "operator": "鈴木", "measureValue": "1000", "measureUnit": "sec"}

print("(3) キーと値のペアの一覧をタプルで取得")
for key, value in meta_dict_for2.items():
    print(key)
    print(value)
```

# リストについて (おさらい)

リストについて、簡単に復習します。

リストは、複数の値を順序付けられた一つのデータとして扱うことができます。

- リストは、次の形式で表現します。,(カンマ)や:(コロン)の前後に空白を入れても構いません。

    [値1,値2,値3,・・・]

- 値の重複を許可します。
- 値には任意の型を使用できます。異なる型が混在していても構いません。

値を取得するには、次のように指定します。

    リスト[順序番号]

- 順序番号は、0から始まる点に注意が必要です。
- 順序番号にマイナスの整数を指定すると、「後ろから何番目」という意味になります。
- 指定した順序番号が配列のサイズを超えるとエラーになります。例えば長さ5の配列であれば、指定可能な順序番号は0から4(前からの参照)、および-1から-5(後ろからの参照)となります。

リストのうち一部の要素を切り出した別のリストを取得することができます。このような操作を「**スライス**」と呼び、次のような指定方法があります。

- リスト[X:Y]と指定すると、リストのX番目から(Y-1)番目までの要素を取得できます。
- リスト[X:]と指定すると、リストのX番目から最後までの要素を取得できます。
- リスト[:Y]と指定すると、リストの最初から(Y-1)番目までの要素を取得できます。
- リスト[X:Y:D]と指定すると、リストのX番目から(Y-1)番目までの要素をD個おきに取得できます。
- リスト[X::D]と指定すると、リストのX番目から最後までの要素をD個おきに取得できます。
- リスト[:Y:D]と指定すると、リストの最初から(Y-1)番目までの要素をD個おきに取得できます。

次のプログラムを実行して、リストの作成・値の取得の動作を再確認しましょう。

In [ ]:

```
# リストを作成
columns_list = ["date", "temperature", "operator", "measureValue", "measureUnit"]

print("リストの内容を画面に出力する")
print(columns_list)

print("2番目の要素を取得する")
print(columns_list[2])

print("後ろから2番目の要素を取得する")
print(columns_list[-2])

print("1番目から3番目の要素を取得する")
print(columns_list[1:4])
```

# リストの操作

今回は、より実践的なリストの使い方として、次の操作をご紹介します。

- リストの作成・要素の追加
- リストの連結
- リストへ値を挿入
- リストの値を変更
- リストの要素を削除
- リストを空にする
- リストの値の取り出し

## リストの作成・要素の追加

作成したリストに要素を追加します。

辞書の例と同様に、最初に空のリストを作成して、そこに値を次々と追加していきます。

空のリストを作成する方法は次のとおりです。何も要素がないので、単に「[]」とすればOKです。

```
リストを格納する変数 = []
```

要素を追加する方法には様々なものがありますが、最も基本的なやりかたとしては、次のとおり「**appendメソッド**」を使用します。

```
リスト.append(追加したい要素)
```

appendメソッドは、リストの末尾に単一の要素を追加します。

空のリストを作成し、そこに要素を次々と追加していく例を、次のソースコードで示します。

In [ ]:

```
# 空のリストを作成
list_apd = []

print("要素追加前のリスト")
print(list_apd)

# 要素を追加する
list_apd.append("date")
list_apd.append("temperature")
list_apd.append("operator")
list_apd.append("measureValue")
list_apd.append("measureUnit")

print("要素追加後のリスト")
print(list_apd)
```

## リストの連結

+演算子を使うことで、複数のリストを連結することができます。

```
リスト1 + リスト2 + リスト3 ・・・
```

例を次に示します。

In [ ]:

```
list_p1 = ["date", "temperature"]
list_p2 = ["operator"]
list_p3 = ["measureValue", "measureUnit"]

# リストを連結する
list_plus = list_p1 + list_p2 + list_p3

# リストの内容の確認
print(list_plus)
```

## リストへ値を挿入

リストの途中に単一の要素を挿入するときは、次のとおり「**insertメソッド**」を使用します。

```
リスト.insert(挿入する位置, 挿入する要素)
```

「挿入する位置」には、リストの何番目に追加するかを整数で指定します。0から開始する点に注意が必要です。

In [ ]:

```
list_ins = ["date", "temperature", "measureValue", "measureUnit"]

print("変更前のリスト")
print(list_ins)

# リストに値を挿入する
list_ins.insert(2, "operator")

print("変更後のリスト")
print(list_ins)
```

## リストへ複数の値を挿入

リストの途中に複数の要素を追加するときは、次のとおりスライスを使用します。

```
リスト[挿入する位置:挿入する位置] = 挿入するリスト
```

「挿入する位置:挿入する位置」としなければいけないのが、ちょっと分かりづらいですね・・・。注意するようにしてください。

例を次に示します。

In [ ]:

```
list_inssl = ["date", "temperature", "measureUnit"]

print("変更前のリスト")
print(list_inssl)

# リストに複数の要素を挿入する
list_inssl[2:2] = ["operator", "measureValue"]

print("変更後のリスト")
print(list_inssl)
```

## リストの値を変更

リストの値を変更します。値の変更方法は、次のとおりです。

```
リスト[変更する位置] = 変更後の値
```

例を次に示します。

In [ ]:

```
list_mod = ["date", "temperature", "operator", "measureValue", "measureUnit"]

print("変更前のリスト")
print(list_mod)

# リストの値を変更する
list_mod[3] = "value"

print("変更後のリスト")
print(list_mod)
```

## リストの複数の値をまとめて変更

リストの複数の要素を変更します。次のとおりスライスを使用します。

```
リスト[変更する開始位置:変更する終了位置+1] = 変更後の値のリスト
```

例を次に示します。

In [ ]:

```
list_modsl = ["date", "temperature", "operator", "measureValue", "measureUnit"]

print("変更前のリスト")
print(list_modsl)

# リストの複数の値を変更する
list_modsl[2:4] = ["opName", "value"]

print("変更後のリスト")
print(list_modsl)
```

## リストの要素を削除

リストの要素を削除します。辞書と同じように、delを使って削除ができます。

```
del リスト[削除する位置]
```

例を次に示します。

In [ ]:

```
list_del = ["date", "temperature", "operator", "measureValue", "measureUnit"]

print("変更前のリスト")
print(list_del)

# リストの要素を削除する
del list_del[3]

print("変更後のリスト")
print(list_del)
```

## リストの複数の要素を削除

delを使ってリストの複数の要素を削除するには、スライスを使用します。

```
del リスト[削除する開始位置:削除する終了位置+1]
```

例を次に示します。

In [ ]:

```
list_delsl = ["date", "temperature", "operator", "measureValue", "measureUnit"]

print("変更前のリスト")
print(list_delsl)

# リストの複数の要素を削除する
del list_delsl[2:4]

print("変更後のリスト")
print(list_delsl)
```

## リストの指定した値を削除

「**removeメソッド**」を使用すると、指定した値を削除することができます。

```
リスト.remove(削除したい値)
```

ただし、リスト中に該当する値が複数あった場合は、最初の要素のみ削除されます。

In [ ]:

```
list_remove = ["date", "temperature", "operator", "measureValue", "operator", "measureUnit"]

print("変更前のリスト")
print(list_remove)

# 指定した最初の要素を削除する
list_remove.remove("operator")

print("変更後のリスト")
print(list_remove)
```

この例では、リスト中に"operator"が2つあったため、最初の要素だけが削除されました。

## リストを空にする

「**clearメソッド**」を使用すると、リストを空にすることができます。

```
リスト.clear()
```

例を次に示します。

In [ ]:

```
list_clr = ["date", "temperature", "operator", "measureValue", "measureUnit"]

print("変更前のリスト")
print(list_clr)

# リストを空にする
list_clr.clear()

print("変更後のリスト")
print(list_clr)
```

## リストの値の取り出し

辞書と同様に、popメソッドを使ってリストの要素を取り出すことができます。

popメソッドを使用すると、指定した位置の値がリストから削除され、戻り値として削除した値が取得されます。

```
取り出した値を格納する変数 = リスト.pop(取り出す位置)
```

なお引数の「取り出す位置」を省略すると、末尾の要素が削除されます。

In [ ]:

```
list_pop = ["date", "temperature", "operator", "measureValue", "measureUnit"]

print("変更前のリスト")
print(list_pop)

# リストの要素を取り出す
pvalue = list_pop.pop(2)

print("位置指定あり - 取り出した値と変更後のリスト")
print(pvalue)
print(list_pop)

# リストの末尾の要素を取り出す
pvalue = list_pop.pop()

print("位置指定なし - 取り出した値と変更後のリスト")
print(pvalue)
print(list_pop)
```

## リストのコピー

次のソースコードの実行結果を予想して、実際に動かしてみてください。2つのprint関数の結果は、どうなるでしょうか。

In [ ]:

```
# 整数の例
value1 = 1
value2 = value1

value2 = 100

print(value1)
print(value2)
```

予想通りの結果になったのではないでしょうか。

それでは、次のソースコードの実行結果はどうでしょうか。

In [ ]:

```
# リストの例 (1)
list1 = [1, 2, 3]
list2 = list1

list2 = [4, 5, 6]

print(list1)
print(list2)
```

これも、予想通りではないかと思います。

最後に、次のソースコードはどうでしょうか。

In [ ]:

```
# リストの例 (2)
list3 = [1, 2, 3]
list4 = list3

list4[1] = 100

print(list3)
print(list4)
```

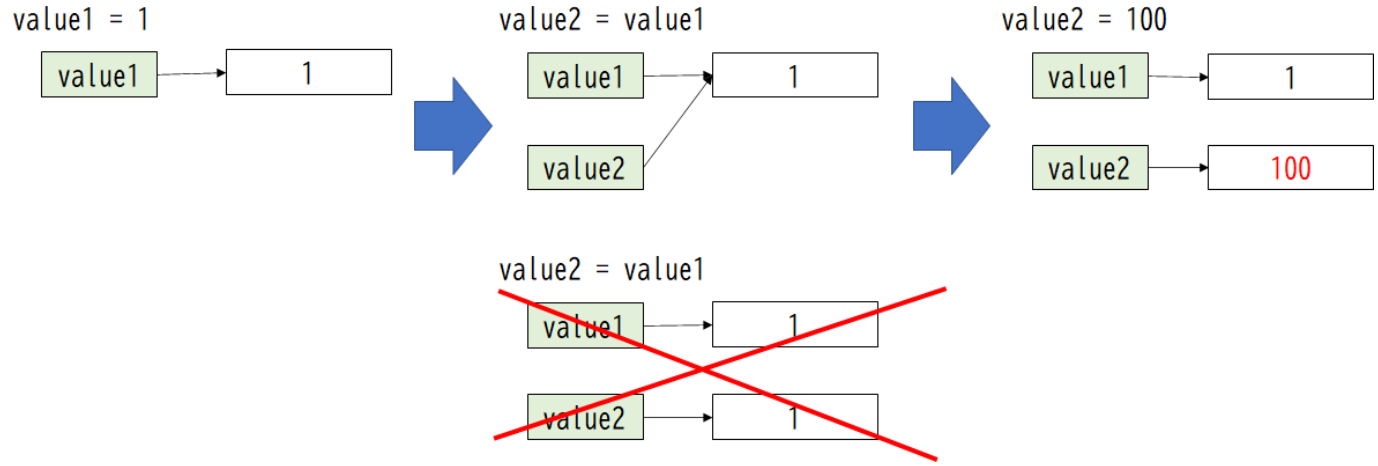
list4の2要素目が100となるのは予想どおりかと思いますが、list3の2番目の要素も100となってしまいました！？

### 代入式とcopyメソッド

ある程度プログラミングに慣れていないと、非常に分かりにくい話になってしまいますが・・・

今回の例は、それぞれ次のような動作イメージとなっています。

## ■ 整数の例

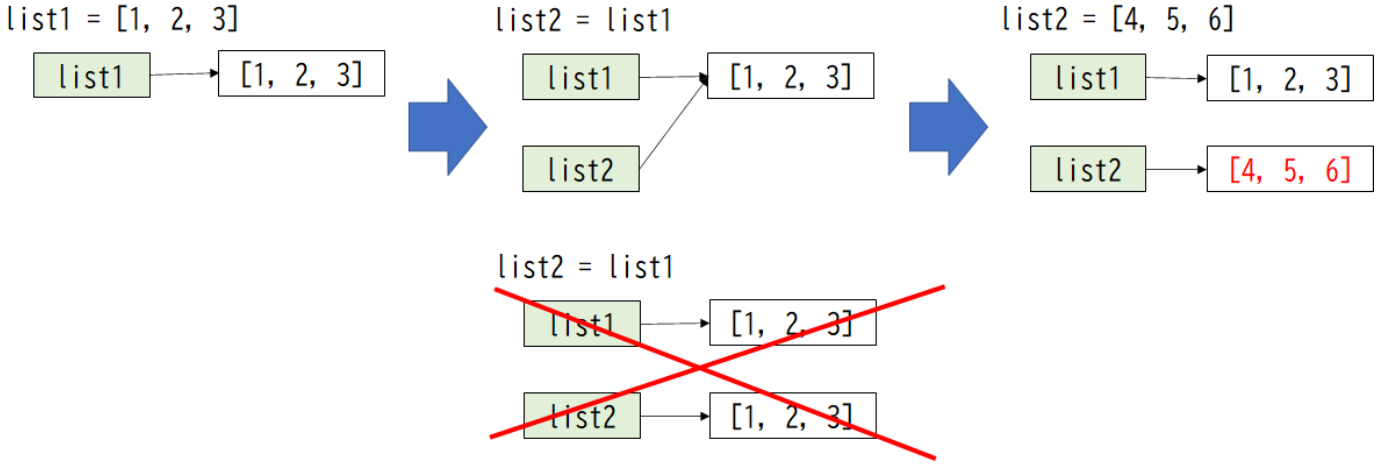



整数の例において、「value2 = value1」を実行した時点では、2つの変数は同じデータを指しています。

「同じ値」ではなく、「同じデータ」である点に注意してください。

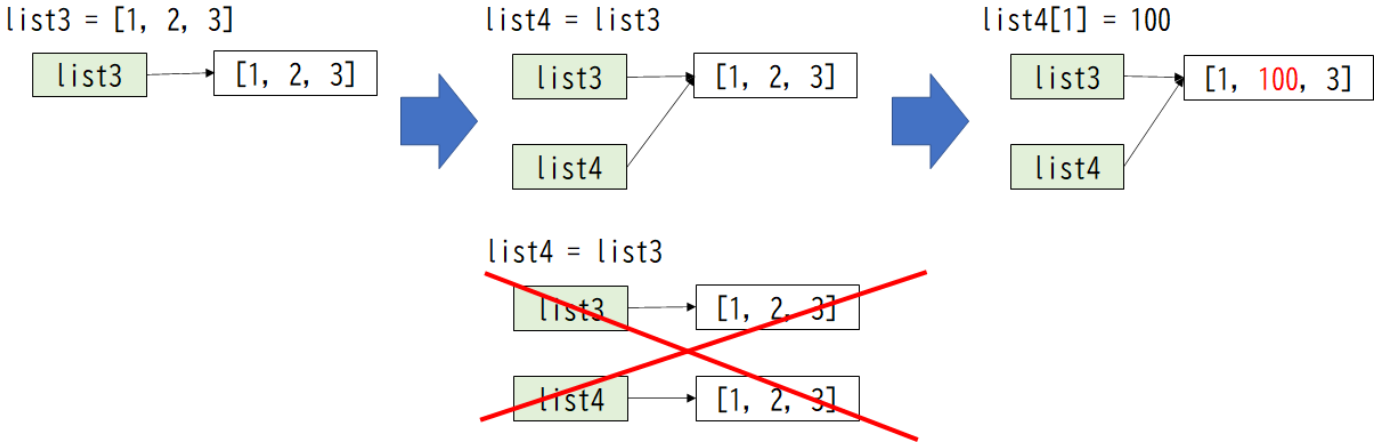
「value2 = 100」を実行した時点で、value1とvalue2は別のデータを指すことになります。

## ■リストの例（1）




リストの例(1)も、整数の例と同じイメージです。

## ■リストの例（2）




問題は、上記のリストの例(2)です。

「list4[1] = 100」を実行しても、list3とlist4は同じデータを指したままです。

そのため、list3もlist4も、リストの2要素目が100になってしまいました。

このような事態に陥らないようにするためには、「list4 = list3」の部分で代入式ではなく、copyメソッドを利用します。

リスト名.copy()

copyメソッドを使うと、戻り値としてリストの複製(値は同じだが別のデータ)を作成することができます。

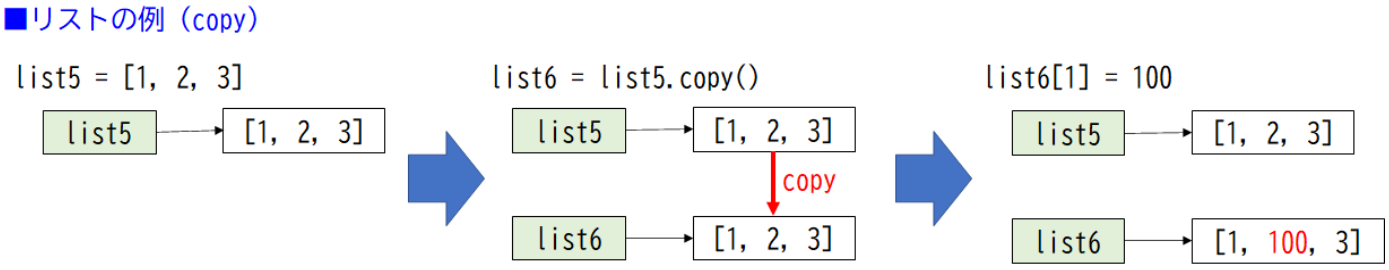
In [ ]:

```
# リストの例（copy）
list5 = [1, 2, 3]
list6 = list5.copy()

list6[1] = 100

print(list5)
print(list6)
```

copyメソッドを使うことにより、次のイメージで動作しました。




### [ご参考] deepcopy

リストのコピーにはcopyメソッドを使うのが基本ですが、さらに厄介なのは、入れ子のリスト(リストのリスト)です。

入れ子のリストをコピーしたい場合、copyメソッドを使っても期待どおりの動作になりません。

In [ ]:

```
rlist1 = [[1, 2, 3], [4, 5, 6]]

rlist2 = rlist1.copy()

rlist2[1][1] = 100

print(rlist1)
print(rlist2)
```

copyメソッドを使ったのに、rlist1の方も値が100になってしまいました！？

入れ子のリストを完全にコピーするときは、copyモジュールのdeepcopy関数を使う必要があります。

In [ ]:

```
# deepcopy関数はcopyモジュールで提供される
from copy import deepcopy

rlist3 = [[1, 2, 3], [4, 5, 6]]

rlist4 = deepcopy(rlist1)

rlist4[1][1] = 100

print(rlist3)
print(rlist4)
```

今回は詳しい説明をしませんが、入れ子のリストをコピーするときにはdeepcopyを使うべきでないか検討してみてください。

なおcopyメソッドとdeepcopy関数の使い分けについてですが、基本的には、**リストが入れ子でなければcopyメソッド、入れ子であればdeepcopy関数**と考えて構いません。

入れ子かどうかに関わらずdeepcopy関数、という考えでもプログラムの動作として問題はないのですが、必要のないimportは行わないほうが、プログラムとしてはすっきりします。

### 辞書のコピー

ここまではリストの例で説明をしてきましたが、辞書についても同じことが言えます。

入れ子の辞書のコピーにはdeepcopyを使う、という点も同様です。

## 練習問題

「辞書・リストの操作」の最後に、練習問題にチャレンジしてみましょう。

あらかじめ、辞書を作成して変数dict_parcticeに格納してあります。この辞書には、次のようなキーがあります。

| キー | 内容 |
|---|---|
| date | 測定日 |
| operator | 測定者 |
| temperature | 測定温度 |
| measureValue1 | 1回目の測定データ |
| measureValue2 | 2回目の測定データ |
| measureValue3 | 3回目の測定データ |
| measureValue4 | 4回目の測定データ |

for文を使って、measureValue1～measureValue4が示す4つの値のみを画面に出力してください。

In [ ]:

```
# 辞書データの作成
dict_parctice = {"date":"2023-01-06", "operator":"鈴木", "temperature":18, "measureValue1":1000, "measureValue2":600, "measureValue3":1200, "m

for key, value in dict_parctice.items():
    if key.startswith("measureValue"):
        print(value)
```

## データ構造化における辞書・リストの活用例

ここまでは、機能を理解しやすいようにシンプルな辞書やリストを使って説明してきました。

最後に、データ構造化において辞書やリストをどのように活用しているか、一例をご紹介します。

辞書・リストの活用例としては、メタデータが挙げられます。構造化プログラム内部で、メタデータを辞書として作成します。最終的に、この辞書データをmetadata.jsonというJSONファイルに出力しています。

辞書の値として、リストも使われています。

実際にARIM事業のデータ構造化プログラムで扱っているメタデータの例を示します。内容の細かい説明はしませんが、辞書の使われ方としてイメージを持ってもらえればと思います。

```
{
    "constant": {},
    "variable": [
        {
            "sample name": {
                "value": "test"
            },
            "chemical formula": {
                "value": "BaTiO3"
            },
            "typial particle size": {
                "value": 500.0,
                "unit": "nm"
            },
            "measurement start date and time": {
                "value": "2022-09-14T10:15:00"
            },

                   <<< 中略 >>>

            "electric field": {
                "value": 0.0,
                "unit": "V/m"
            },
            "coating material": {
                "value": "Au"
            },
            "support material": {
                "value": "Si"
            }
        }
    ]
}
```


以上で、第4回の講義を終わります。

第5回では、Matplotlibというパッケージを使って、構造化プログラムでもよく行われるグラフ作成などについて学びます。