2021年度 東京成徳大学 特別講座 資料
東京工科大学コンピュータサイエンス学部
福西広晃

第03回 Pythonによるテーブルデータの操作と統計計算

質問
Excelでできることを挙げてみてください

Pandasとは

- データ操作を高速かつ効率的に扱うデータ解析支援ライブラリ
  - ライブラリとは、よく使う機能・関数をまとめて、簡単に使えるようにしたもの

- Pandasでできること
  - データの読み込み、整形、加工、結合、保存
  - データの検索・表計算・集計
  - データの可視化(グラフ作成)

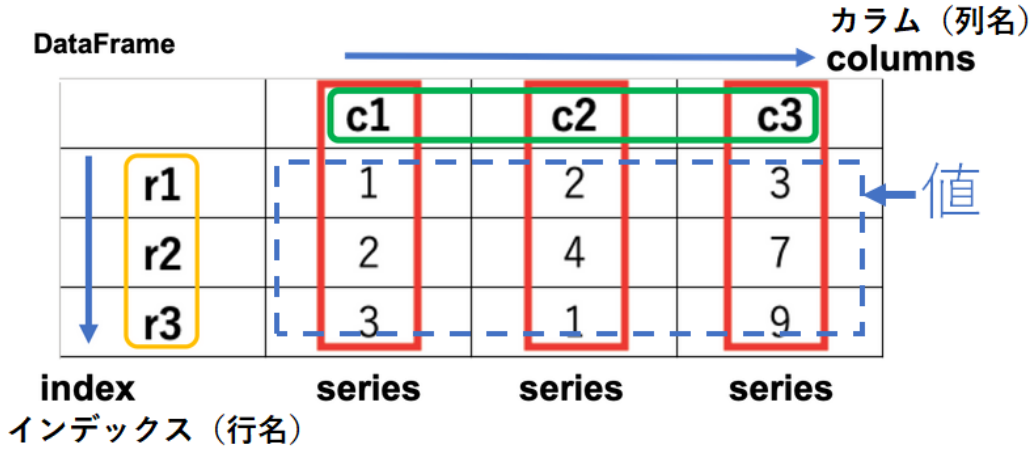
- これらの操作はエクセルでも可能であるが、Pandasを利用すればエクセルでは扱えないような **大量データの高速処理** 、 **ルーチンワークの自動化** が可能になり作業の効率化に繋がる。また、手作業ベースのエクセルの問題は、１度行った一連の操作の再現が困難であることである。Pandasによって操作がプログラム化されていれば何度でも同じ操作を行うことができる。

- pandasを使えることは、データサイエンスやAIの専門領域では必須であるが、日常業務でも利用できないわけではない。業務データ（例えば顧客データ）に対する処理を自動化し、業務効率化を行うことができれば企業内での評価も上がることでしょう

Pandasのデータ型

- Pandasでは変数とテーブルデータ（表形式のデータ）を繋げて扱う
- Pandasには `DataFrame`型 (データフレーム型)と `Series`型 (シリーズ型)のデータ構造がある
  - **データフレーム型** は、表形式データ（ラベル付き2次元データ）で **値** 、 **インデックス** （行）、 **カラム** （列）から成る（Excelシートのイメージ）

- シリーズ型 は、１次元データで 値 、 インデックス から成る（リスト型に類似）



準備

- Pandasを使ってExcel形式のファイルを出力するために openpyxl がインストールする必要があります。
- インストール方法
  - Anacondaの場合
    - `! conda install -c anaconda openpyxl -y`
  - Google Colaboratoryの場合
    - `! pip install openpyxl`

Google Colaboratoryの場合は２行目のコメントアウトを外して実行

In [ ]:
```
# ! conda install -c anaconda openpyxl -y
! pip install openpyxl
```

- Google Colaboratoryにgoogleドライブ上のファイルを読み込みたい場合（またはデータを出力したい場合）は以下を実行しておく必要がある（毎回の作業前）
  - Go to this URL in a browserの後に記載されているリンク先をクリックして、指示通りに許可やログインをすると、コードが表示される
  - コードをコピーして、Enter your authorization codeに張り付けてEnterキーを押す
  - Colab Notebooksフォルダにdatasetフォルダを作成し、AAA.csvを置いたとする。これをpandasで読み込む場合の例は、
    `df = pd.read_csv("drive/My Drive/Colab Notebooks/dataset/AAA.csv", encoding="shift_jis")`

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

Pandasのインポート

- インポートとは pandas を使用できるように呼び出すこと。以下のように記述。
  `import pandas as pd`
- as pd とは、プログラム内でpandasを pd という名前で使用するということ。慣習的に pd とする。

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

ファイルからのデータの読み込み

csvファイルの読み込み

- csvファイルはカンマ区切りで並べられたデータ(メモ帳で開いて確認してみよう)
- read_csv を用いて読み込み

  データフレーム名 = pd.read_csv(csvファイル, encoding=文字コード)
  - 第1引数は読み込むcsvファイル(引数とは、括弧内に記載する利用条件と考えてください)
  - encodingのデフォルトは utf_8 （国際標準）。データを出力した際に文字化けする場合は shift_jis （日本語ファイル、国勢調査などのオフィシャルなデータによくある）をencodingで指定すると解決する可能性が高い。

    文字コードとは、コンピュータで文字を扱うために各文字に割り当てられるバイト表現、あるいはその対応関係のこと。コンピュータはこの対応関係を頼りにデータを「エンコード」(普通の文字を機械語に直す)したり、「デコード」(機械語を普通の文字に戻す)したりしている。この対応関係の指定を間違えると、「文字化け」が起こる（ shift_jis の文字コードで書かれたファイルを utf_8 で読み込むと文字化けする）

- pandasで読み込んだデータフレーム（表形式データ）を表示する場合は print ではなく display を用いる

  display(データフレーム名)

以下は、datasetフォルダの「都道府県別人口推移.csv」を df_population という変数(データフレーム)に読み込む例。このデータは、 1920-2015年の各都道府県の人口推移データ である。encodingを shift_jis として読み込む必要あり。

In [ ]:
```
# ファイルの指定

# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/都道府県別人口推移.csv"

# Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/都道府県別人口推移.csv"

# データの読み込み
df_population = pd.read_csv(file, encoding="shift_jis")

# データの表示
display(df_population)
```

Excelファイルの読み込み

- read_excel を用いて読み込む
  - 第1引数は読み込むcsvファイル

- sheet_nameでExcelのシート名を指定（1シートしかない場合は記述しなくてもよい）
- encodingは不要

In [ ]:
```
# ファイルの指定

# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/都道府県別人口推移.xlsx"

# # Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/都道府県別人口推移.xlsx"

# データの読み込み
df_population = pd.read_excel(file, sheet_name="Sheet1")

# データの表示
display(df_population)
```

練習

平成のみのデータを含む「都道府県別人口推移_平成.xlsx」の「sheet1」を df_population_H という名前の変数（データフレーム）に読み込んでみよう。また、 df_population_H を表示してみよう。ファイルはdatasetフォルダに含まれている。

In [ ]:
```

```

解答例

In [ ]:
```
# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/都道府県別人口推移_平成.xlsx"

# # Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/都道府県別人口推移_平成.xlsx"

df_population_H = pd.read_excel(file, sheet_name="Sheet1")

display(df_population_H)
```

ファイルへのデータの出力

df_populationのデータフレームを出力するとする

In [ ]:
```
display(df_population)
```

データフレームをcsvファイルとして出力

データフレーム名.to_csv(csvファイル名, index=False, encoding="文字コード")

- ファイルの出力先とファイル名は第1引数で指定
- index=Falseのオプションを付けると、データフレームのインデックスは出力されない
- encodingは文字コードを指定（shift_jis, utf-8, utf_8_sigなど）

In [ ]:
```
# 出力ファイル名を指定（Googleドライブの場合）
```

```
outfile = "drive/My Drive/Colab Notebooks/出力結果のテスト.csv"

# 出力ファイル名を指定（Anacondaの場合，Google Colaboratoryの場合は削除してください）
outfile = "出力結果のテスト.csv"

# 出力
df_population.to_csv(outfile, index=False, encoding='shift_jis')
```

データフレームをExcelファイルとして出力

```
データフレーム名.to_excel("Excelファイル名", sheet_name = "シート名",
index=False, encoding="文字コード")
```

- ファイルの出力先とファイル名は第1引数で指定
- sheet_nameは、エクセルのシート名を設定（名前は自由）
- index=Falseのオプションを付けると、データフレームのインデックスは出力されない
- encodingは文字コードを指定（shift_jis, utf-8, utf_8_sigなど）

- **既存のExcelファイルにシートを追加して出力する場合**

```
with pd.ExcelWriter("Excelファイル名", engine="openpyxl", mode='a') as
writer:
    データフレーム名.to_excel(writer, sheet_name = "シート名",
index=False, encoding="文字コード")
```

df_populationのデータフレームを「出力結果のテスト.xlsx」の「都道府県別人口推移_出力1」のシートに出力する例
以下を実行後にフォルダ内の「出力結果のテスト.xlsx」を開いて確認してみよう

In [ ]:
```
# 出力ファイル名を指定（Googleドライブの場合）
outfile = "drive/My Drive/Colab Notebooks/出力結果のテスト.xlsx"

# # 出力ファイル名を指定（Anacondaの場合，Google Colaboratoryの場合は削除してください）
outfile = "出力結果のテスト.xlsx"

# 出力
df_population.to_excel(outfile, sheet_name="都道府県別人口推移_出力1", index=False, e
```

df_populationのデータフレームを「出力結果のテスト.xlsx」の「都道府県別人口推移_出力2」のシートとして**追加**する例
以下を実行後にフォルダ内の「出力結果のテスト.xlsx」を開いて確認してみよう

In [ ]:
```
# シート追加
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_population.to_excel(writer, sheet_name="都道府県別人口推移_出力2", index=False
```

練習

(1) 前の練習で用いた df_population_H をExcelファイルとして出力してみよう。出力ファイル名は「出力結果のテスト2.xlsx」、シート名は「都道府県別人口推移_平成_出力1」とする。インデックスは出力せず、文字コードはshift_jisとする。

In [ ]:
```

```

(2) 「出力結果のテスト2.xlsx」にシートを追加して df_population_H を再度出力してみよう。シート名は「都道府県別人口推移_平成_出力2」とする。インデックスは出力せず、文字コードはshift_jisとする。

In [ ]:

```
```

解答例

In [ ]:

```
#(1)
# 出力ファイル名を指定（Googleドライブの場合）
outfile = "drive/My Drive/Colab Notebooks/出力結果のテスト2.xlsx"

# # 出力ファイル名を指定（Anacondaの場合，Google Colaboratoryの場合は削除してください）
outfile = "出力結果のテスト2.xlsx"

df_population_H.to_excel(outfile, sheet_name="都道府県別人口推移_平成_出力1", index=F

#(2)
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_population.to_excel(writer, sheet_name="都道府県別人口推移_平成_出力2", index=
```

データ構造の確認

データの行数と列数を確認

- データ分析において、サンプルサイズを報告することは必須です

行数と列数を同時に確認

```
行数, 列数 = データフレーム名.shape
```

df_populationの列数と行数を確認

In [ ]:

```
rows, cols = df_population.shape

print("行数: ", rows)
print("列数: ", cols)
```

行数のみを確認

- 組み込み関数の len関数 を実行

In [ ]:

```
rows = len(df_population)
print("行数", rows)
```

練習

前の練習で使用した df_population_H の行数と列数を出力してみよう

In [ ]:

```
```

解答例

In [ ]:
```
rows, cols = df_population_H.shape

print("行数：", rows)
print("列数：", cols)
```

特定の列または行の抽出

先頭行または末尾行のデータを抽出

- 先頭x行を抽出
  データフレーム名.head(x)
- 先頭5行の場合のみ データフレーム名.head() でも可能
- データをざっくりと確認するために head() は非常によく使う

- 末尾x行を抽出
  データフレーム名.tail(x)

In [ ]:
```
# 先頭3行を抽出
df_rows = df_population.head(3)
display(df_rows)

# 末尾3行を抽出
df_rows = df_population.tail(5)
display(df_rows)
```

列を抽出

1列抽出

データフレーム名["列名"]

- 抽出された1列は Series型

In [ ]:
```
# 人口（総数）の列を抽出
df_col = df_population["人口（総数）"]

display(df_col)
```

複数列を抽出

データフレーム名[["列名1","列名2", ・・・]]

- 抽出された複数列は DataFrame型

In [ ]:
```
# 人口（総数）、人口（男）、人口（女）の3列を抽出
df_cols = df_population[["人口（総数）", "人口（男）", "人口（女）"]]
```

```
display(df_cols)
```

練習

(1) 前の練習で用いた df_population_H の先頭10行をdf_rowsに代入し、表示してみよう

In [ ]:

(2) df_population_H の都道府県コードと都道府県名の２列をdf_colsに代入し、表示してみよう

In [ ]:

解答例

In [ ]:
```
#(1)
df_rows = df_population_H.head(10)
print(df_rows)
```

In [ ]:
```
#(2)
df_cols = df_population_H[["都道府県コード", "都道府県名"]]
print(df_cols)
```

条件に一致したデータの抽出

条件式の真偽値で各行を判定

- 演算子一覧 |||| |:-:|:-:|:--| |比較演算子| < | 小さい | | | > | 大きい | | | <= | 以上 | | | >= | 以下 | | | == | 等しい | | | != | 等しくない | |論理演算子| & | 両者を満たす | | | | | どちらか片方を満たす | | | != | 満たさない | ||||
- Pandasの論理演算子は and 、 or 、 not の表記が使えないことに注意

西暦（年）が2000に一致する行であるかをTrue（真）とFalse（偽）で判定

In [ ]:
```
TrueFalse = (df_population["西暦（年）"] == 2000)

# 真偽値を出力(シリーズ型で出力)
display(TrueFalse)
```

各行の真偽値を用いてTrue（真）のみを抽出

データフレーム名[真偽値]

In [ ]:
```
# 別名のデータフレームに結果を保持
df_TrueFalse = df_population[TrueFalse]

# 表示は先頭5行に限定
display(df_TrueFalse.head(5))
```

真偽値の判定を[...]に直接記述（慣れたらこちらで書く人が多い）

In [ ]:
```
df_TrueFalse = df_population[df_population["西暦（年）"] == 2000]

display(df_TrueFalse.head(5)) # 表示は先頭5行に限定
```

抽出した結果をExcelファイルに出力

- データフレーム名: df_TrueFalse
- Excelファイル名: 出力結果.xlsx
- シート名: 人口推移_2000年
- index：出力なし
- encoding: shift_jis

In [ ]:
```
# 出力ファイル名を指定（Googleドライブの場合）
outfile = "drive/My Drive/Colab Notebooks/出力結果.xlsx"

# # 出力ファイル名を指定（Anacondaの場合, Google Colaboratoryの場合は削除してください）
outfile = "出力結果.xlsx"

# 出力
df_TrueFalse.to_excel(outfile, sheet_name="人口推移_2000年", index=False, encoding="s
```

論理演算子を用いた複数条件による抽出

西暦（年）が1980年以降かつ都道府県名が奈良県に一致する行を抽出

In [ ]:
```
TrueFalse2 = (df_population["西暦（年）"] >= 1980) & (df_population["都道府県名"] ==

df_TrueFalse2 = df_population[TrueFalse2] # 別名のデータフレームに結果を保持

display(df_TrueFalse2) # 表示
```

抽出した結果をExcelファイルに出力(シートを追加)

- データフレーム名: df_TrueFalse2
- Excelファイル名: 出力結果.xlsx （既に存在するファイル, 上記でoutputの変数に代入されているのでここでは代入する記述は不要）
- シート名: 人口推移_1980年以降_奈良県
- index：出力なし
- encoding: shift_jis

In [ ]:
```
# シート追加
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_TrueFalse2.to_excel(writer, sheet_name="人口推移_1980年以降_奈良県", index=Fal
```

練習

(1) 前の練習で用いた df_population_H から全年の東京都の人口を抽出して df_TrueFalse_01 に代入し、先頭5行を表示してみよう

In [ ]:
```

```

(2) df_population_H から西暦2010年以降の人口を抽出して df_TrueFalse_02 に代入し、先

頭5行を表示してみよう

In [ ]:

(3) df_population_H から西暦2010年以前の沖縄県の人口を抽出して df_TrueFalse_03 に代入し、先頭5行を表示してみよう

In [ ]:

(4) (3)の抽出結果をExcelファイルに出力してみよう(シートを追加)

- データフレーム名: df_TrueFalse_03
- Excelファイル名: 出力結果.xlsx （既に存在するファイル, 上記でoutputの変数に代入されているのでここでは代入する記述は不要）
- シート名: 人口推移_2010年以前_沖縄
- index：出力なし
- encoding: shift_jis

In [ ]:

解答例

In [ ]:

```
# (1)
TrueFalse       = df_population_H["都道府県名"] == "東京都"
df_TrueFalse_01 = df_population_H[TrueFalse] # 別名のデータフレームに結果を保持
display(df_TrueFalse_01.head(5))           # 表示は先頭5行に限定

# (2)
TrueFalse       = (df_population_H["西暦（年）"] >= 2010)
df_TrueFalse_02 = df_population_H[TrueFalse] # 別名のデータフレームに結果を保持
display(df_TrueFalse_02.head(5))             # 表示は先頭5行に限定

# (3)
TrueFalse       = (df_population_H["西暦（年）"] <= 2010) &  (df_population_H["都道府
df_TrueFalse_03 = df_population_H[TrueFalse] # 別名のデータフレームに結果を保持
display(df_TrueFalse_03.head(5))             # 表示は先頭5行に限定

# (4)
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_TrueFalse_03.to_excel(writer, sheet_name="人口推移_2010年以前_沖縄", index=Fal
```

データ分析

Boston house-prices:
「1970年代後半における（米国マサチューセッツ州）ボストンの住宅価格」のデータを使用

- CRIM： 町別の「犯罪率」
- ZN： 25,000平方フィートを超える区画に分類される住宅地の割合＝「広い家の割合」
- INDUS： 町別の「非小売業の割合」
- CHAS： チャールズ川のダミー変数（区画が川に接している場合は1、そうでない場合は0） ＝「川の隣か」

- NOX： 「NOx濃度（0.1ppm単位）」＝一酸化窒素濃度（parts per 10 million単位）。この項目を目的変数とする場合もある
- RM： 1戸当たりの「平均部屋数」
- AGE： 1940年より前に建てられた持ち家の割合＝「古い家の割合」
- DIS： 5つあるボストン雇用センターまでの加重距離＝「主要施設への距離」
- RAD： 「主要高速道路へのアクセス性」の指数
- TAX： 10,000ドル当たりの「固定資産税率」
- PTRATIO： 町別の「生徒と先生の比率」
- B： 「1000(Bk - 0.63)」の二乗値。Bk＝「町ごとの黒人の割合」を指す
- LSTAT： 「低所得者人口の割合」
- MEDV：「住宅価格」（1000ドル単位）の中央値。通常はこの数値が目的変数として使われる

In [ ]:
```
import pandas as pd

# ファイルの指定

# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/boston.xlsx"

# Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/boston.xlsx"

# データをpandasで読み込み
df_boston = pd.read_excel(file, sheet_name="boston")

# 先頭10行を表示
display(df_boston.head(10))
```

統計量の算出

基本統計量の一覧を出力

- 以下を実行すると、各列（数値列）の基本統計量が計算される
  データフレーム名.describe()
- 計算結果はデータフレームとして得られる
- 最初にデータの状況を確認する場合に便利 |||| |:-:|:-:|:--| |count |標本数 |行数 | |mean |平均値 |算術平均 | |std |標準偏差 |データのばらつき | |min |最小値 |最も小さい値 | |25% |第1四分位数 |データを小さい順に並び替えたとき、データ個数を小さい方から数えて4分の1番目にあたる値| |50% |第2四分位数(中央値) |データを小さい順に並び替えたとき、データ個数を小さい方から数えて4分の2番目にあたる値|
  |75% |第3四分位数 |データを小さい順に並び替えたとき、データ個数を小さい方から数えて4分の3番目にあたる値|
  |max |最大値 |最も大きい値 | ||||

In [ ]:
```
# 統計量を計算
df_describe = df_boston.describe()

# 表示
display(df_describe)
```

結果をExcelファイルに出力(シートを追加)

- データフレーム名: df_describe
- Excelファイル名: 出力結果.xlsx （既に存在するファイル, 上記でoutputの変数に代入されているのでここでは代入する記述は不要）
- シート名: boston_基本統計量
- index：出力あり
- encoding: shift_jis

In [ ]:
```
# シート追加
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_describe.to_excel(writer, sheet_name="boston_基本統計量", encoding="shift_jis"
```

個別の統計量の計算

要素の個数	データフレーム名.count()	
算術平均	データフレーム名.mean()	
標準偏差	データフレーム名.std()	n-1で割った不偏標準偏差
最小値	データフレーム名.min()	
最大値	データフレーム名.max()	
中央値	データフレーム名.median()	
最頻値	データフレーム名.mode()	最も出現回数の多い値
ユニークな値の個数	データフレーム名.nuique()	重複を除いた件数
特定の列のユニーク値の出現頻度	データフレーム名.value_counts()	Series型メソッド

ユニークな値の個数（重複を除いた件数）

In [ ]:
```
num_unique = df_boston.nunique()

#  ユニークな値の個数（重複を除いた件数）を出力
display(num_unique)
```

特定の列のユニークな値の出現頻度

In [ ]:
```
vcount = df_boston["RAD"].value_counts()

# 特定の列のユニークな値の出現頻度を出力
display(vcount)
```

練習

5教科（国語 英語 社会 数学 理科）のテスト結果のデータを用いる。以下に「5教科成績.xlsx」の「5教科成績」のシートを df_score のデータフレームに読み込む（このデータは架空データである）。

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

```
# ファイルの指定

# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/5教科成績.xlsx"

# # Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/5教科成績.xlsx"

# データをpandasで読み込み
df_score = pd.read_excel(file, sheet_name="5教科成績")

# 先頭10行を出力
display(df_score.head(10))
```

(1) 基本統計量の一覧を表示してみよう。計算結果は、 df_describe_02 のデータフレームに格納するものとする。
また、出力結果を見て、 学生数 、 最も平均点の高い教科 、 最もばらつきが大きい教科 (標準偏差)、 真ん中の学生の点数が最も低い教科 （中央値）、 最高点が最も高い教科 を確認してみよう。

In [ ]:

(2) クラスごとの学生数を表示してみよう（特定の列のユニークな値の出現頻度を参考にしてください）。結果をvcount_02に格納するものとする。

In [ ]:

(3) (1)の結果をExcelファイルに出力してみよう(シートを追加)

- データフレーム名: df_describe_02
- Excelファイル名: 出力結果.xlsx （既に存在するファイル, 上記でoutputの変数に代入されているのでここでは代入する記述は不要）
- シート名: 5教科成績_基本統計量
- index：出力あり
- encoding: shift_jis

In [ ]:

解答例

In [ ]:

```
# (1)
df_describe_02 = df_score.describe()
display(df_describe_02)

# (2)
vcount_02 = df_score["クラス"].value_counts()
display(vcount_02)

# (3)
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_describe_02.to_excel(writer, sheet_name="5教科成績_基本統計量", encoding="shif
```

ピボットテーブル

ここでは、アパレル店の1年間の販売データを用いる。以下に「販売データ.xlsx」の「実績管理表」のシートを df_sale のデータフレームに読み込む（このデータは架空データである）。

In [ ]:

```
import pandas as pd

# ファイルの指定

# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/販売データ.xlsx"

# Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/販売データ.xlsx"

# データをpandasで読み込み
df_sale = pd.read_excel(file, sheet_name="実績管理表")

# 表示
display(df_sale)
```

質問
このような日々の販売データが記録されていたとして、どのような分析が考えられるでしょうか

- ピボットテーブルは、ユニーク値ごとにグループ化して統計量の算出するため用いられる
    - 例1: 商品ごとや、社員ごとの売上金額の合計を集計（この社員の売上金額はいくらか、この商品の売上金額はいくらか）
        - ある項目のユニーク値ごとにグループ化して集計
    - 例2: 商品と社員の2項目を組み合わせた集計（この社員のこの商品に対する売上金額はいくらか）
        - **クロス集計** と呼ばれる
- ピボットテーブルは特に **クロス集計** を行うときに威力を発揮
- ある項目のユニーク値ごとにグループ化して集計する場合は、Pandasの groupby メソッドを使うなどの代替手段がある

- 基本の書き方
    - ある項目のユニーク値ごとにグループ化して集計（集計結果はデータフレーム）
      データフレーム名.pivot_table(index = "集計単位の列名"
      values = "集計する列名"
      aggfunc = "集計方法")
    - クロス集計（集計結果はデータフレーム）
      データフレーム名.pivot_table(index = "集計単位の列名1"
      columns = "集計単位の列名2"
      values = "集計する列名",
      aggfunc = "集計方法")
      集計方法には、 sum (合計)、 mean (平均)、 count (頻度)、または 独自の関数 が設定できる。今回は sum (合計)のみを扱う。

ある項目のユニーク値ごとにグループ化して集計

社員ごとの売上金額を集計

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = "社員ID",   # 集計単位の列名
                               values  = "売上金額", # 集計する列名
                               aggfunc = "sum")      # 集計方法
# 集計結果を出力
display(df_pivot)
```

商品名ごとの売上金額を集計

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = "商品名",   # 集計単位の列名
                               values  = "売上金額", # 集計する列名
                               aggfunc = "sum")      # 集計方法
# 集計結果を出力
display(df_pivot)
```

練習

上記のdf_saleのデータフレームを用いるものとする。ピボットテーブルを用いて性別ごとの売上金額の合計を出力してみよう。
集計結果は、 `df_pivot_02` のデータフレームに格納するものとする。

In [ ]:
```

```

解答例

In [ ]:
```
df_pivot_02 = df_sale.pivot_table(index   = "性別",   # 集計単位の列名
                                  values  = "売上金額", # 集計する列名
                                  aggfunc = "sum")      # 集計方法
# 集計結果を出力
display(df_pivot_02)
```

クロス集計(2項目の組み合わせて集計)

社員ごとの商品分類ごとの売上金額を出力

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = "社員ID",   # 集計単位の列名1
                               columns = '商品分類', # 集計単位の列名2
                               values  = "売上金額", # 集計する列名
                               aggfunc = "sum")      # 集計方法
# 集計結果を出力
display(df_pivot)
```

合計列を追加

- 引数に `margins = True` を追加すると合計列及び合計行が作成される。その際の合計列及び合計行の名前は `all` となる
- 合計列及び合計行の名前を `all` から変更したい場合には、引数に `margins_name = "合計"` を追加する

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = "社員ID",   # 集計単位の列名1
                               columns = "商品分類", # 集計単位の列名2
```

```
                                values  = "売上金額", # 集計する列名
                                aggfunc = "sum",      # 集計方法
                                margins = True,       # 合計列の追加
                                margins_name = "合計") # 合計列の列名の指定
# 集計結果を出力
display(df_pivot)
```

複数の集計方法で同時に集計

- 今回のデータでは、countは社員の出現回数

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = "社員ID",
                               columns = "商品分類",
                               values  = "売上金額",
                               aggfunc = ["sum", "count"],
                               margins = True,
                               margins_name = "合計")
# 集計結果を出力
display(df_pivot.astype(int)) # 要素を整数値に変換
```

集計結果の行名（インデックス）を複数項目で設定

- 各社員の売上月ごとの売上金額の合計を集計

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = ["社員ID", "売上月"],
                               columns = '商品分類',
                               values  = "売上金額",
                               aggfunc = "sum",
                               margins = True,
                               margins_name = "合計")
display(df_pivot)
```

出力結果の列名（カラム）を複数項目で設定

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = "社員ID",
                               columns = ["商品分類","商品名"],
                               values  = "売上金額",
                               aggfunc = "sum",
                               margins = True,
                               margins_name = "合計")
display(df_pivot)
```

nan（欠損値）を特定の値で補完

- 引数に `fill_value= 値` を追加
- 今回のデータではnanを0で補完

In [ ]:
```
df_pivot = df_sale.pivot_table(index   = "社員ID",
                               columns = ["商品分類","商品名"],
                               values  = "売上金額",
                               aggfunc = "sum",
                               margins = True,
                               margins_name = "合計",
                               fill_value= 0)
display(df_pivot)
```

結果をExcelファイルに出力(シートを追加)

- データフレーム名: df_pivot
- Excelファイル名: 出力結果.xlsx （既に存在するファイル, 上記でoutputの変数に代入されているのでここでは代入する記述は不要）
- シート名: 販売データ_ピボットテーブル
- index：出力あり
- encoding: shift_jis

In [ ]:
```
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_pivot.to_excel(writer, sheet_name="販売データ_ピボットテーブル", encoding="shi
```

練習

上記で使用した df_sale のデータフレームを用いるものとする

(1) 各社員の商品名ごとの販売数量（数量）をピボットテーブルを用いて集計し、集計結果を df_pivot_02 のデータフレームに代入してみよう。また、集計結果を出力し、商品ごとの販売数トップの社員を挙げてみよう。

In [ ]:
```

```

(2)各売上月の商品名ごとの売上金額をピボットテーブルを用いて集計し、集計結果をdf_pivot_03のデータフレームに代入してみよう。また、集計結果を出力し、各商品の季節ごとの販売状況を考察してみよう。

In [ ]:
```

```

(3) (2)の結果をExcelファイルに出力してみよう(シートを追加)

- データフレーム名: df_pivot_03
- Excelファイル名: 出力結果.xlsx （既に存在するファイル, 上記でoutputの変数に代入されているのでここでは代入する記述は不要）
- シート名: 販売データ_ピボット2
- index：出力あり
- encoding: shift_jis

解答例

In [ ]:
```
#(1)
df_pivot_02 = df_sale.pivot_table(index   = "社員ID",
                                  columns = "商品名",
                                  values  = "数量",
                                  aggfunc = "sum",
                                  margins = True,
                                  margins_name = "合計",
                                  fill_value= 0)
display(df_pivot_02)
```

商品ごとの販売数Top社員

- シャツ:　　a036
- ジャケット:　a003
- ジーンズ:　　a013
- ダウン:　　a023

- ニット:　　a003
- ハーフパンツ: a003
- ロングパンツ: a013

In [ ]:
```
#(2)
df_pivot_03 = df_sale.pivot_table(index    = "売上月",
                                  columns = "商品名",
                                  values   = "売上金額",
                                  aggfunc = "sum",
                                  margins = True,
                                  margins_name = "合計",
                                  fill_value= 0)
display(df_pivot_03)
```

In [ ]:
```
#(3)
with pd.ExcelWriter(outfile, engine="openpyxl", mode='a') as writer:
    df_pivot_03.to_excel(writer, sheet_name="販売データ_ピボットテーブル2", encoding=
```

以降、付録

データの前処理

- 取得したデータは、大抵汚れているもの（そのままでは分析に使えない）
- データを取得したら、分析できる状態かを確認し、問題があれば整形する必要がある。これ行程を 前処理 と呼ぶ。
- 特に、 **データの重複** や **データの欠損** の確認は集計値（合計、平均など）に影響を与えいるので注意が必要

重複と欠損を含むデータの読み込み

In [ ]:
```
# ファイルの指定

# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/都道府県別人口推移.xlsx"

# Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/都道府県別人口推移_欠損_重複.xlsx"

# データの読み込み
df_population = pd.read_excel(file, sheet_name="Sheet1")

# データの出力
display(df_population)
```

重複したデータへの対応

- 重複した行は削除

重複している行を確認

データフレーム名.duplicated()

- 重複した行の内、最初の行は False （重複でない）扱い、それ以降は True （重複）扱い

In [ ]:
```
TrueFalse = df_population.duplicated()

# 重複した行の判定結果を出力
display(TrueFalse)
```

重複している行を抽出

データフレーム名[真偽値]

- True（重複行）の行のみ抽出される

In [ ]:
```
df_duplicated = df_population[TrueFalse]

# 重複行の出力
display(df_duplicated)
```

重複を除去したデータを抽出

データフレーム名.drop_duplicates()

In [ ]:
```
df_drop_duplicates = df_population.drop_duplicates()

# 重複した行を除去したデータを抽出
display(df_drop_duplicates)
```

練習

(1)「都道府県別人口推移_平成_欠損_重複.xlsx」のsheet1のデータを df_population_02 のデータフレームに読み込み、先頭5行を出力してみよう

In [ ]:
```

```

(2) df_population_02 において重複した行を df_duplicated_02 のデータフレームに代入し、出力してみよう

In [ ]:
```

```

(3) df_population_02 の重複した行を除去して df_drop_duplicates_02 のデータフレームに代入し、先頭5行を出力してみよう

In [ ]:
```

```

解答例

In [208]:
```
#(1)
# ファイルの指定

# Google Colaboratoryの場合
```

```
file = "drive/My Drive/Colab Notebooks/dataset/都道府県別人口推移_平成_欠損_重複.xlsx"

# Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/都道府県別人口推移_平成_欠損_重複.xlsx"

# データの読み込み
df_population_02 = pd.read_excel(file, sheet_name="Sheet1")

# データの出力
display(df_population_02.head(5))
```

	都道府県コード	都道府県名	元号	和暦（年）	西暦（年）	人口（総数）	人口（男）	人口（女）
0	1	北海道	平成	2	1990	5643647.0	2722988.0	2920659.0
1	1	北海道	平成	2	1990	5643647.0	2722988.0	2920659.0
2	1	北海道	平成	2	1990	5643647.0	2722988.0	2920659.0
3	2	青森県	平成	2	1990	1482873.0	704758.0	778115.0
4	3	岩手県	平成	2	1990	NaN	680197.0	736731.0

In [209]:

```
#(2)
TrueFalse = df_population_02.duplicated()

df_duplicated_02 = df_population_02[TrueFalse]

# 重複行の出力
display(df_duplicated_02)
```

	都道府県コード	都道府県名	元号	和暦（年）	西暦（年）	人口（総数）	人口（男）	人口（女）
1	1	北海道	平成	2	1990	5643647.0	2722988.0	2920659.0
2	1	北海道	平成	2	1990	5643647.0	2722988.0	2920659.0
8	6	山形県	平成	2	1990	1258390.0	607041.0	651349.0
15	13	東京都	平成	2	1990	11855563.0	5969773.0	5885790.0
16	13	東京都	平成	2	1990	11855563.0	5969773.0	5885790.0
17	13	東京都	平成	2	1990	11855563.0	5969773.0	5885790.0

In [ ]:

```
#(3)
df_drop_duplicates_02 = df_population_02.drop_duplicates()
```

```
# 重複した行を除去したデータを抽出
display(df_drop_duplicates_02.head(5))
```

欠損値の対応（値が欠けている場合）

- 要素が NaN (Not a Number)となっていたら欠損値のこと
- 欠損値の代表的な対応方法は以下の２つ
  - 欠損値を含む行を削除する
  - 欠損値を別の値で補完する（ただし、別の値を入れることに根拠がある場合）

重複した行が除去されたdf_drop_duplicatesのデータフレームに対して欠損値の対処を行う

各列の欠損値の数をカウント

```
データフレーム名.isnull().sum()
```

In [ ]:
```
nans = df_drop_duplicates.isnull().sum()

# 各列の欠損値の数を出力
print(nans)
```

欠損値が1つでもある行の確認

```
データフレーム名.isnull().any(axis=1)
```

- 欠損値が１つでもある行は True 、全くなければ False
- axis=1は行方向に欠損値があるかを判定、axis=0なら列方向に欠損値があるかを判定

In [ ]:
```
TrueFalse = df_drop_duplicates.isnull().any(axis=1)

# 欠損値が1つでもある行の判定結果を出力
display(TrueFalse)
```

欠損値が1つでもある行の抽出

```
データフレーム名[真偽値]
```

In [ ]:
```
df_nan = df_drop_duplicates[TrueFalse]

# 欠損値が1つでもある行を出力
display(df_nan)
```

欠損値のある行を削除

```
データフレーム名 = データフレーム名.dropna()
```

In [ ]:
```
df_dropna = df_drop_duplicates.dropna()

# 欠損値を１つでも含む行を除去したデータを抽出
display(df_dropna)
```

欠損値を別の値で補完

- 根拠がある場合は、欠損値を別の値で補完することがある
  - 例1：気温のセンサーデータに欠損値がある場合、急な気温の変動がないだろうと想定して、前後の値の平均値で補完
  - 例2：売上データに欠損値がある場合、合計値に影響を与えないように0で補完

データフレーム名.fillna(値)

In [ ]:
```
# 欠損値を0で補完
df_fillna = df_drop_duplicates.fillna(0)

# 欠損値を別の値で補完したデータを出力
display(df_fillna)
```

練習

(1) 前の練習で重複した行を除去した df_drop_duplicates_02 のデータフレームに関して、各列の欠損値の数をカウントして出力してみよう

In [ ]:
```

```

(2) df_drop_duplicates_02 において 1 つでも欠損値ある行を df_nan_02 のデータフレームに代入し、出力してみよう

In [ ]:
```

```

(3) df_drop_duplicates_02 の欠損値を0で補完して df_fillna_02 のデータフレームに代入し、先頭5行を出力してみよう

In [ ]:
```

```

解答例

In [ ]:
```
#(1)
nans = df_drop_duplicates_02.isnull().sum()

# 各列の欠損値の数を出力
print(nans)
```

In [ ]:
```
#(2)
TrueFalse = df_drop_duplicates_02.isnull().any(axis=1)

df_nan_02 = df_drop_duplicates_02[TrueFalse]

# 欠損値が1つでもある行を出力
display(df_nan_02)
```

In [ ]:
```
#(3)
# 欠損値を0で補完
df_fillna_02 = df_drop_duplicates_02.fillna(0)
```

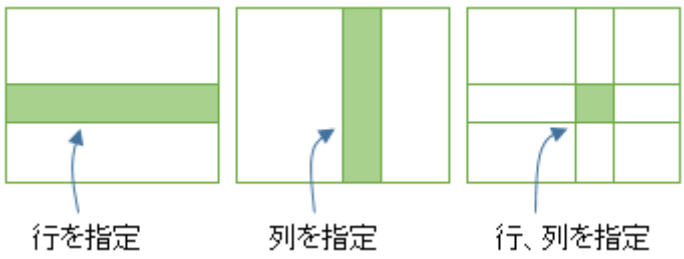
```
# 欠損値を別の値で補完したデータを出力
display(df_fillna_02.head(5))
```

特定の要素の選択、取得、変更

- loc または iloc のメソッドが使用されることが多い

- loc の基本の記述方法（行名と列名で位置を指定）
  - 単独の要素を選択
    - データフレーム名.loc['行名', '列名']
  - 複数の要素を選択
    - データフレーム名.loc[['行名1', '行名2', ...], ['列名1', '列名2', ...]]
  - スライス表記（：（コロン）を使って範囲で指定)
    - データフレーム名.loc['行名1' : '行名2', '列名1':'列名2']

- iloc の基本の記述方法（行番号と列名で位置を指定、行番号と列番号は0からスタート）
  - 単独の要素を選択
    - データフレーム名.iloc['行番号', '列番号']
  - 複数の要素を選択
    - データフレーム名.iloc[['行番号1', '行番号2', ...], ['列番号1', '列番号2', ...]]
  - スライス表記（：（コロン）を使って範囲で指定)
    - データフレーム名.iloc['行番号1' : '行番号2', '列番号1':'列番号2']
    - 行番号2-1または列番号2-1までが抽出されることに注意



データの読み込み

In [210]:
```
# ファイルの指定

# Google Colaboratoryの場合
file = "drive/My Drive/Colab Notebooks/dataset/都道府県別人口推移.xlsx"

# Anacondaの場合(Google Colaboratoryの場合は削除してください)
file = "dataset/都道府県別人口推移.xlsx"

# データの読み込み
df_population = pd.read_excel(file, sheet_name="Sheet1")

# データの出力
display(df_population)
```

	都道府県コード	都道府県名	元号	和暦（年）	西暦（年）	人口（総数）	人口（男）	人口（女）
0	1	北海道	大正	9	1920	2359183	1244322	1114861
1	2	青森県	大正	9	1920	756454	381293	375161
2	3	岩手県	大正	9	1920	845540	421069	424471
3	4	宮城県	大正	9	1920	961768	485309	476459
4	5	秋田県	大正	9	1920	898537	453682	444855
...	...	...	...	...	...	...	...	...
934	43	熊本県	平成	27	2015	1786170	841046	945124
935	44	大分県	平成	27	2015	1166338	551932	614406
936	45	宮崎県	平成	27	2015	1104069	519242	584827
937	46	鹿児島県	平成	27	2015	1648177	773061	875116
938	47	沖縄県	平成	27	2015	1433566	704619	728947

939 rows × 8 columns

単独の要素の抽出

In [ ]:

```
# 行名：10,
# 列名：人口（総数）
# の値を抽出

# locを使用
display(df_population.loc[10, "人口（総数）"])

# ilocを使用
display(df_population.iloc[10, 5])
```

単独の要素の変更

In [ ]:

```
# locを使用
df_population.loc[3, "人口（総数）"] = 0
display(df_population.head())

# ilocを使用
df_population.iloc[3, 7] = 0
display(df_population.head())
```

複数の要素を抽出

In [ ]:
```
# 行名：10, 11, 12
# 列名：人口（総数）, 人口（男）, 人口（女）
# の値を抽出

# locを使用
display(df_population.loc[[10,11,12], ["人口（総数）", "人口（男）", "人口（女）"]])

# ilocを使用
display(df_population.iloc[[10,11,12], [5, 6, 7]])
```

複数の要素をスライスを用いて抽出

In [ ]:
```
# locを使用
display(df_population.loc[0:2, "人口（総数）": "人口（女）"])

# ilocを使用(抽出されるのは行番号-1、列番号-1まで)
display(df_population.iloc[0:2, 5:7])
```

：（コロン）のみで範囲の指定がない場合は全てを意味する。以下では、locで行全部、ilocで列全部を表している

In [ ]:
```
# locを使用
display(df_population.loc[:, "人口（総数）": "人口（女）"])

# ilocを使用(抽出されるのは行番号-1、列番号-1まで)
display(df_population.iloc[0:2, :])
```

以下のように

- ：（コロン）の右側の指定がない場合は、左側の指定以降全てを表す
- ：（コロン）の左側の指定がない場合は、右側の指定以前全てを表す

In [ ]:
```
# 行名が10より前を抽出
display(df_population.loc[:10, "人口（総数）": "人口（女）"])

# 列名が「人口（総数）」より後を抽出
display(df_population.loc[0:2, "人口（総数）":])
```

複数の要素の変更

In [ ]:
```
df_population.loc[:, "人口（総数）": "人口（女）"] = 0
display(df_population)

df_population.iloc[0:2, :] = 0
display(df_population)
```

列名に対する列番号、行名に対する行番号を調べる方法

- get_loc メソッドを使用

- 列番号を調べる場合: データフレーム名.columns.get_loc(列名)
- 行番号を調べる場合: データフレーム名.index.get_loc(行名)

In [ ]:

```
# 列名が「人口（女）」の列番号を取得
display(df_population.columns.get_loc("人口（女）"))

# 行名が「10」の行番号を取得
display(df_population.index.get_loc(10))
```
