

Python活用基礎研修: 応用編1

pandasとMatplotlibを使用したデータ解析

Python活用基礎研修について

研修の目標

- Pythonを通してプログラミングの基礎を学ぶ.
- 実際に手を動かして簡単なデータ解析およびAI（機械学習）を体験する.

入門編

- 入門編1: Pythonプログラミングの基本的なルールを学ぶ.
- 入門編2: NumPyを用いたプログラミングを学ぶ.

応用編: 各種モジュール（=便利なツール）の使い方を学ぶ

- 応用編1（データ解析）: pandas, Matplotlib
- 応用編2（AI, 機械学習）: scikit-learn

研修の進行速度および理解度把握のためのご協力をお願い

本日の研修では、研修の進行速度が適切かを把握するために、Zoomの**リアクション機能**を用いたレスポンスをお願いすることがございます。研修中にこちらから進行速度が早すぎないか質問した際に、**リアクション機能**の絵文字で**進行速度**や**理解度**に対する感想をお願いいたします。

リアクションの例

- 👍, 👉 : ちょうど良い または 理解できている
- 😐, 🤔 : すこし早い または 少し難しい
- 😞, 🥲 : もうすこしゆっくりが良い または 前の説明範囲で躓いている

前回の1.6 練習問題の答え

要素が全て0の3x3の配列が代入された変数 `c_3d` について、以下の配列になるように変数 `c_3d` を操作せよ。

```
[[0 0 0]  
 [0 1 0]  
 [0 0 0]]
```

解答例1

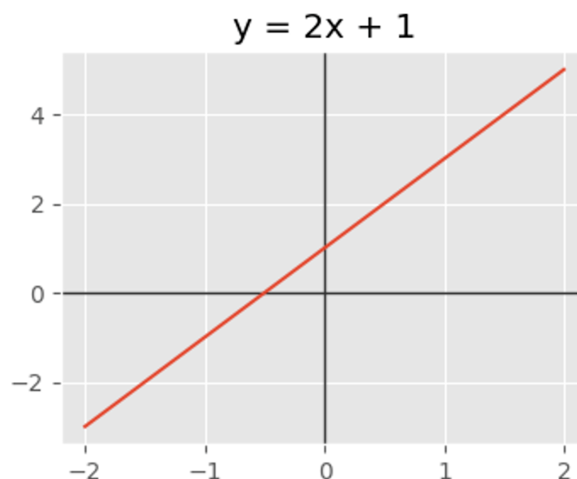
```
c_3d[1][1] = 1
```

解答例2

```
c_3d[1,1] = 1
```


前回の1.9. 練習問題の答え

一次方程式 $y = 2x + 1$ について, $x_1d = [-2, -1, 0, 1, 2]$ の配列をNumPyで作成し, x_1d の各値に対応する y の値を求めよ.



解答例

```
x_1d = np.array([-2, -1, 0, 1, 2])  
y_1d = 2 * x_1d + 1 # スカラー値との演算は, 配列の全ての要素毎に処理される.
```

前回の2.6 練習問題: その1の答え

以下の成績表について,

1. 3x3の配列をNumPyで生成せよ.

	名前	国語	数学	英語
0	山田	80	90	75
1	田中	50	95	67
2	鈴木	73	56	95

解答例

```
table_3d = np.array([[80, 90, 75], [50, 95, 67], [73, 56, 95]])  
table_3d
```

前回の2.6 練習問題: その1の答え

成績表について,

2. `sum` 関数を用いて, 山田の3科目の合計点を求めよ.

解答例1

```
np.sum(table_3d[0]) # 山田の行を選択後に関数sumを用いる
```

解答例2

```
np.sum(table_3d, axis=1)[0] # 関数sumの引数axisを用いて山田の行を指定
```

前回の2.6 練習問題: その1の答え

成績表について,

3. `mean` 関数を用いて, 数学の平均点を求めよ.

解答例1

```
np.mean(table_3d[:, 1]) # 数学の列を選択後に関数sumを用いる
```

解答例2

```
np.mean(table_3d, axis=0)[1] # 関数meanの引数axisを用いて数学の列を指定
```

前回の2.6 練習問題: その2の答え

`array` 関数を用いずに以下の配列をそれぞれ生成せよ.

1. 値が3から始まり19で終わる, 奇数のみで構成された1次元配列

```
[3 5 7 9 11 13 15 17 19]
```

```
np.arange(3, 20, 2)
```

2. 値が0から始まり8で終わる, 3x3の配列

```
[[0 1 2]  
 [3 4 5]  
 [6 7 8]]
```

```
np.arange(9).reshape((3,3))
```

前回の2.6 練習問題: その3の答え

1. `array` 関数を用いずに以下の配列を変数 `f_2d` として生成せよ.

```
[[0. 0. 0. 0.]  
 [0. 1. 1. 0.]  
 [0. 1. 1. 0.]  
 [0. 0. 0. 0.]
```

解答例

```
f_2d = np.zeros((4,4)) # まず全ての要素の4x4の2次元配列を生成  
f_2d[1:3, 1:3] = 1     # その後, スライスを用いて縦横1から2まで範囲を参照し, 1を代入  
f_2d
```

前回の2.6 練習問題: その3の答え

2. 以下の配列 `g_2d` は配列 `f_2d` と同じ形状 4x4 の配列である.

配列 `g_2d` について, 配列 `f_2d` で値が1の範囲のみ合計値を求めよ.

```
g_2d = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]])  
# [[ 0  1  2  3]  
#   [ 4  5  6  7]  
#   [ 8  9 10 11]  
#   [12 13 14 15]]
```

解答例

```
np.sum(g_2d * f_2d) # g_2dとf_2dは同じ形状の配列なので要素毎の乗算が可能. その後, 合計を算出.
```

0. はじめに

0.1. pandasとMatplotlib

1. pandasの基本

2. Matplotlibの基本

3. データ解析とグラフ描画

0.1. pandasとMatplotlib

pandasは高速かつ効率的な機能によって、データ解析を支援するPythonのモジュール。特に、表形式データおよび時系列データを操作するための各種機能が提供されている。

Matplotlibはグラフ描画のためのPythonのモジュール。MATLABユーザにとって簡単に習得できるように設計されている。

利用する準備

```
import pandas as pd
import matplotlib.pyplot as plt
```

0. はじめに

1. **pandas**の基本

1.1. データ構造: DataFrameとSeries

1.2. データの読み込み

1.3. データの書き込み

1.4. データの概要の取得

1.5. データの参照

1.6. 練習問題

2. Matplotlibの基本

3. データ集計とグラフ描画

1.1. データ構造: DataFrameとSeries

pandasでは, 主に**DataFrame**と**Sereis**の2種類のデータ構造を扱う.

- **DataFrame**: 行と列で構成されるテーブル形式のデータ構造. 2次元データ.
- **Sereis**: DataFrameの1行または1列を保持するデータ構造. 1次元データ.

基本的にデータはDataFrameで管理され, その操作や抽出の結果がSeriesで取得される.

Series

Series

	名前	国語	数学	英語
0	山田	80	90	75
1	田中	50	95	67
2	鈴木	73	56	95

DataFrame

1.1. データ構造: DataFrameとSeries

DataFrameは、`DataFrame` 関数を用いて構築できる。

```
# data(必須)    : DataFrameに格納するデータを指定. 指定するデータは配列でもリストでも良い.  
# index(任意)   : 行名を指定. 指定しない場合, 0からの連番が付与.  
# columns(任意): 列名を指定. 指定しない場合, 0からの連番が付与.  
pd.DataFrame(data, index=None, columns=None)
```

Noneとは？

Noneは、Pythonにおいて値が存在しないことを表現するための特別な定数。

```
# データのみで構築する場合  
data_2d = np.array([[80, 90, 75], [50, 95, 67], [73, 56, 95]])  
data_df = pd.DataFrame(data_2d)                                     # DataFrameの元となる2次元配列  
# 行名や列名を指定して構築する場合                                # 上記, 1変数でDataFrameを構築  
data_2d = np.array([[80, 90, 75], [50, 95, 67], [73, 56, 95]])    # DataFrameの元となる2次元配列  
table_col = ['英語', '数学', '国語']                               # 列名のリスト  
table_idx = ['山田', '田中', '鈴木']                               # インデックス名のリスト  
table_df = pd.DataFrame(data_2d, columns=table_col, index=table_idx) # 上記, 3変数でDataFrameを構築
```

1.2. データの読み込み: excel形式のファイル

`read_excel` 関数を用いることで, excel形式のファイルをDataFrameとして読み込める.

```
# filepath_or_buffer(必須): ファイルのパス. ローカルパスはもちろんURLでも良い.  
# sheet_name(任意)       : 読み込むシートを指定. シート番号またはシート名で指定.  
# header(任意)           : 列見出しの列名となる行を指定. デフォルトでは0行を列名に使用.  
# index_col(任意)        : 行見出しとなる列を指定. デフォルトでは0からの連番.  
# usecols(任意)          : 読み込む列を指定. 列番号または列名で指定.  
pd.read_excel(filepath_or_buffer, sheet_name, header, index_col, usecols)
```

```
excel_read_file_path = './data/titanic.xlsx'  
titanic_df = pd.read_excel(excel_read_file_path)
```

Titanic Data: タイタニック号の死亡者と生存者を示す有名な定量的データセット

URL: <https://hbiostat.org/data/repo/titanic.html>

1.2. データの読み込み: csv形式のファイル

`read_csv` 関数を用いることで, csv形式のファイルをDataFrameとして読み込める.

```
# filepath_or_buffer(必須): ファイルのパス. ローカルパスはもちろんURLでも良い.  
# sep(任意)                : 区切り文字を指定. デフォルトは','. これを'\t'にすればtsv形式も扱える.  
# names(任意)              : 列見出しの列名を任意で設定. デフォルトでは空.  
# header(任意)             : 列見出しの列名となる行を指定. namesが空のとき, 0行を列名に使用.  
# index_col(任意)          : 行見出しとなる列を指定. デフォルトでは0からの連番.  
# usecols(任意)            : 読み込む列を指定. 列番号または列名で指定.  
pd.read_csv(filepath_or_buffer, sep, names, header, index_col, usecols)
```

```
csv_read_file_url_path = './data/titanic.csv' # URLを定義  
titanic_df = pd.read_csv(csv_read_file_url_path)
```

1.3. データの書き込み: excel形式のファイル

`to_excel` 関数を用いることで, DataFrameをexcel形式のファイルとして書き出せる.

```
# filepath_or_buffer(必須): 既存のファイルパスの場合は上書き. 存在しないファイルパスの場合は新規作成.  
# sheet_name(任意)       : 書き出す際のシート名を指定. デフォルトは'Sheet1'.  
# header(任意)           : 列見出しを書き出すかどうかを指定. Trueで書き出し, Falseで書き出さない.  
# index(任意)            : 行見出しを書き出すかどうかを指定. Trueで書き出し, Falseで書き出さない.  
df(=DataFrameの変数).to_excel(filepath_or_buffer, sheet_name, header, index)
```

真理値: TrueとFalse

Pythonでは, 真偽値を表す特別な定数として `True` と `False` を実装している.

例えば `if` 文における条件式の判定結果は, この真偽値を用いて表現されている.

```
a = 60
```

```
print(a > 100)
```

```
excel_write_file_path = './data/write_test_titanic.xlsx'  
titanic_df.to_excel(excel_write_file_path, sheet_name='')
```

1.3. データの書き込み: csv形式のファイル

`to_csv` 関数を用いることで, DataFrameをcsv形式のファイルとして書き出せる.

```
# filepath_or_buffer(必須): 既存のファイルパスの場合は上書き. 存在しないファイルパスの場合は新規作成.  
# sep(任意)                : 区切り文字を指定. デフォルトは','. これを'\t'にすればtsv形式も扱える.  
# header(任意)             : 列見出しを書き出すかどうかを指定. Trueで書き出し, Falseで書き出さない.  
# index(任意)              : 行見出しを書き出すかどうかを指定. Trueで書き出し, Falseで書き出さない.  
df(=DataFrameの変数).to_csv(filepath_or_buffer, sep, header, index)
```

```
csv_write_file_path = './data/write_test_titanic.csv'  
titanic_df.to_excel(csv_write_file_path)
```


1.4. データの概要の取得

データの概要を把握するためのプロパティや関数が実装されている。

DataFrameがもつ固有のデータ（プロパティ）を確認する方法

```
titanic_df.shape      # DataFrameの形状を取得.  
titanic_df.columns    # DataFrameの列見出しを取得.  
titanic_df.index      # DataFrameの行見出しを取得.  
titanic_df.dtypes     # DataFrameの各列のデータ型（int型, float型, object型...）を取得.  
  
titanic_df.info()     # 上記を同時に把握できるinfo関数.
```

DataFrameからいくつか選んで中身を確認する方法

```
titanic_df.head(n=5)   # DataFrameを最初からn行取得. デフォルトはn=5  
titanic_df.tail(n=5)   # DataFrameを最後からn行取得. デフォルトはn=5  
titanic_df.sample(n=10) # DataFrameからランダムにn行取得. デフォルトはn=1
```

1.5. データの参照: 単独の値を参照

`at` または `iat` を用いることで, DataFrameから単独の値を参照できる.

- `at['行名', '列名']`: 行名と列名を指定し値を取得.
- `iat[行番号, 列番号]`: 行番号と列番号を指定し値を取得.

```
titanic_df.at[890, 'Name']          # 行名に890, 列名にNameを指定  
titanic_df.iat[890, 3]              # 行番号に890, 列番号に3を指定  
titanic_df.at[890, 'Pclass'] = 1    # 参照と代入を組み合わせた値の置換も可能
```

1.5. データの参照: 単独の値または複数の行を参照

`loc` または `iloc` を用いることで, DataFrameから単独または複数の値を参照できる.
この方法では, **スライスを用いることで参照範囲を指定**できる.

- `loc`: **行名** と **列名**, または, その **リスト** か **スライス** を指定し値または行を取得.
- `iloc`: **行番号** と **列番号**, または, その **リスト** か **スライス** を指定し値または行を取得.

単独の値を参照する場合

```
titanic_df.loc[890, 'Name'] # インデックスに890, 列名にNameを指定  
titanic_df.iloc[890, 3]    # インデックスに890, 列番号に3を指定
```

複数の値を参照する場合

```
titanic_df.loc[890, 'Name':'Age'] # インデックスに890, 'Name'から'Age'の列名を抽出  
titanic_df.iloc[886:891, 3:6]    # インデックスに886から891まで, 3から6までの列番号を抽出
```

1.5. データの参照: 単独の行または複数の行を参照

[] を用いることで, DataFrameから単独または複数の値を参照できる.
ただし, この選択方法は下記のルールに限られる.

- 行のみ参照: 行名 または 行番号 の スライス を指定. 必ず スライス を使うことに注意.
- 列のみ参照: 列名 または 列名のリスト を指定. スライス を使えないことに注意.

行のみ参照

```
titanic_df[886:891] # 886行目から890行目の範囲の複数行を抽出
```

列のみ参照

```
titanic_df['Name'] # 単一列をSeriesとして抽出  
titanic_df[['Name']] # 単一列をDataFrameとして抽出  
titanic_df[['PassengerId', 'Name']] # 複数列をDataFrameとして抽出
```

1.6. 練習問題

DataFrame `titanic_df` について,

1. 全ての列を含む, 最初から10行だけ抽出せよ.
2. `Survived`, `Pclass`, `Sex` の列のみを含む, 最初から10行だけ抽出せよ.
3. `PassengerId` が `300` である1行のみを抽出せよ.

0. はじめに

1. pandasの基本

2. **Matplotlibの基本**

2.1. データの描画

2.2. グラフの装飾

2.3. グラフの保存

2.4. pandasのplot関数

2.5. 練習問題

3. データ集計とグラフ描画

2.1. データの描画: 単一

グラフは, 主に `Figure` および `Axes` というデータ構造で構成される.

- `Figure`: グラフの土台となる画像領域を表現する型.
- `Axes`: `Figure` 上の1つ分のグラフ領域を表現する型.

ここではグラフの描画する関数の中で汎用性の高い `subplots` 関数を確認する.

```
# nrows(任意) : Axesを行方向へ並べる数. デフォルトは1.  
# ncols(任意) : Axesを列方向へ並べる数. デフォルトは1.  
# figsize(任意): Figureの縦横のサイズをインチ指定. デフォルトは(6.4, 4.8).  
fig, axes = subplots(nrows, ncols, figsize)
```

```
x_np = np.arange(-5, 6, 1)  
y_np = 2 * x_np + 1  
fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(6.4, 4.8))  
axes.plot(x_np, y_np)  
plt.show()
```

描画データのx軸となる配列を準備
描画データのy軸となる配列を準備
描画のための画像領域とグラフ領域を生成
グラフ領域にplot関数を用いて描画
作成したグラフを画面に表示するよう命令

2.1. データの描画: 複数

`nrows` または `ncols` に2以上を指定した場合, `axes` は多次元リストの構造で生成される. そのため配列の参照と同じように, インデックスでグラフ領域を指定する必要がある.

```
# nrows(任意) : Axesを縦方向へ並べる数. デフォルトは1.  
# ncols(任意) : Axesを横方向へ並べる数. デフォルトは1.  
# figsize(任意): Figureの縦横のサイズをインチ指定. デフォルトは(6.4, 4.8).  
fig, axes = subplots(nrows, ncols, figsize)  
axes[1][2] # もしnrows=2, ncols=3とした場合に, 右下 (2行3列目) のグラフ領域を選択する方法
```

```
x_np = np.arange(-5, 6, 1)  
y_np = 2 * x_np + 1  
z_2d_np = np.random.randn(20, 20)  
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12.8, 4.8))  
axes[0].plot(x_np, y_np)  
axes[1].scatter(z_2d_np[0, :], z_2d_np[:, 1])  
plt.show()
```

```
# 描画データのx軸となる配列を準備  
# 描画データのy軸となる配列を準備  
# 描画データのxy軸となる2次元配列を準備  
# 描画のための画像領域とグラフ領域を生成  
# 1列目のグラフ領域にplot関数を用いて描画  
# 2列目のグラフ領域にscatter関数を用いて描画  
# 作成したグラフを画面に表示するよう命令
```


2.2. グラフの装飾

グラフの装飾として様々な関数が実装されており, `Figure` および `Axes` から呼び出せる.

- 画像領域のタイトルの追加: `fig.suptitle(タイトル名)`
- グラフ領域のタイトルの追加: `axes.set_title(タイトル名)`
- 軸ラベルの追加: `axes.set_xlabel(x軸ラベル名)`, `axes.set_ylabel(y軸ラベル名)`
- 凡例の表示: `axes.legend()` or `axes.legend([凡例1 , 凡例2 , ... , 凡例N])`

```
x_np = np.arange(-5, 6, 1)
y_np = 2 * x_np + 1
z1_2d_np = np.random.randn(20, 20)
z2_2d_np = np.random.randn(20, 20)
fig, axes = plt.subplots(ncols=2, figsize=(12.8, 4.8))
fig.suptitle('Main figure')
axes[0].plot(x_np, y_np, label='linear equation')
axes[0].set_title('y = 2x+1')
axes[0].legend()
axes[1].scatter(z1_2d_np[0, :], z1_2d_np[:, 1])
axes[1].scatter(z2_2d_np[0, :], z2_2d_np[:, 1])
axes[1].set_title('np.random.randn')
axes[1].legend(['z1', 'z2'])
```

```
# 描画データのx軸となる配列を準備
# 描画データのy軸となる配列を準備
# 描画データのxy軸となる2次元配列を準備
# 描画データのxy軸となる2次元配列を準備
# 描画のための画像領域とグラフ領域を生成
# 画像領域のタイトルを設定
# 1列目のグラフ領域にplot関数を用いて描画
# 左 (1行1列目) のグラフ領域毎のタイトルを設定
# plot関数の凡例(=label)を既に与えているため, 関数に引数を与えずに呼び出し
# 2列目のグラフ領域にscatter関数を用いて描画
# 2列目のグラフ領域にscatter関数を用いて描画
# 右 (1行2列目) のグラフ領域毎のタイトルを設定
# 凡例名をlistで与えて呼び出し
```

2.3. グラフの保存

生成したグラフをファイルとして保存する場合, `savefig` 関数を用いる.

```
# fname(必須): 保存先のファイル名またはパスを指定. 拡張子で保存形式を指定可能.  
# format(任意): 画像の拡張子を指定. デフォルトはpng. fnameの拡張子を優先.  
# dpi(任意): 解像度を数値で指定. デフォルトは100.  
plt.savefig(fname, format, dpi)
```

注意点として, `show` より後に `savefig` を呼び出すと**真っ白の画像が生成される**.

```
x_np = np.arange(-5, 6, 1)  
y_np = 2 * x_np + 1  
z_2d_np = np.random.randn(20, 20)  
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12.8, 4.8))  
axes[0].plot(x_np, y_np)  
axes[1].scatter(z_2d_np[0, :], z_2d_np[:, 1])  
save_fig_path = './data/sample_fig.png'  
plt.savefig(save_fig_path)
```

```
# 描画データのx軸となる配列を準備  
# 描画データのy軸となる配列を準備  
# 描画データのxy軸となる2次元配列を準備  
# 描画のための画像領域とグラフ領域を生成  
# 1列目のグラフ領域にplot関数を用いて描画  
# 2列目のグラフ領域にscatter関数を用いて描画
```

2.4. pandasのplot関数

pandasの `DataFrame` には、様々なグラフを描画する関数が実装されている。
Matplotlibの描画関数と比べると、細かな装飾はできない代わりに簡単に使用できる。
そのため、状況に応じて使い分けることが多い。

- Matplotlib: 自由度が高く、細部まで調整ができる。
- pandas: `Figure` や `Axes` を定義せず、簡単に使用ができる。

```
# データの準備
x_np = np.arange(-5, 6, 1)
y_np = 2 * x_np + 1
xy_df = pd.DataFrame({'x_data': x_np, 'y_data': y_np})

# Matplotlibの場合
fig, axes = plt.subplots()
axes.plot(xy_df['x_data'], xy_df['y_data'])
axes.set_xlabel('x_data')
axes.legend(['y_data'])

# pandasの場合
xy_df.plot('x_data')
```

描画データのx軸となる配列を準備
描画データのy軸となる配列を準備
DataFrameの生成

描画のための画像領域とグラフ領域を生成
グラフ領域にplot関数を用いて描画

2.5. 練習問題

2020年の豊中市の気温と降水量のデータ^[1]について,

1. DataFrameとして読み込め.
2. 読み込んだDataFrameについて, 降雨量の折れ線グラフを描画せよ.

1. データの読み込み

```
drill_2020_file_path = './data/toyonaka_temp_2020.csv'
```

```
drill_2020_df = ???
```

2. 降雨量の折れ線グラフを描画

```
???
```

ヒント: 折れ線グラフの関数

Matplotlib: `Axes.plot()` または pandas: `DataFrame.plot()` を用いると描画できる.

Matplotlib: `Axes.plot(data)` # 引数dataに描画したい列を入れる.

pandas: `DataFrame.plot(y='列名')` # 引数yを使って描画する列を選択できる.

[1]: 出典: 気象庁「過去の気象データ検索」(<https://www.data.jma.go.jp/stats/etrn/index.php>)

- 0. はじめに
- 1. pandasの基本
- 2. Matplotlibの基本
- 3. **データ集計とグラフ描画**
 - 3.1. 指定した条件で行を抽出
 - 3.2. 指定した条件で並び替え
 - 3.3. 要素の値の置換
 - 3.4. ユニークな値の確認
 - 3.5. 欠損値: NaN
 - 3.6. 統計量の算出
 - 3.7. 相関値の算出
 - 3.8. 練習問題

3.1. 指定した条件で行を抽出: 比較演算子

比較演算子を用いることで, 特定の条件に従う行のみを抽出できる.

```
titanic_df[titanic_df['Age'] < 10]          # 列Ageの値が10未満の行を抽出
titanic_df[titanic_df['Sex'] == 'male']     # 文字列の要素も抽出条件にできる

# 性別が男性の乗客について年齢のヒストグラムを確認: pandas
titanic_df[titanic_df['Sex'] == 'male'].plot.hist(y='Age')
titanic_df[titanic_df['Sex'] == 'male'].hist(figsize=(9,7)) # 全ての列についても描画可能

# 性別が男性の乗客について年齢のヒストグラムを確認: Matplotlib
fig, axes = plt.subplots()                  # 描画のための画像領域とグラフ領域を生成
extracted_df = titanic_df[titanic_df['Sex'] == 'male'] # 性別が男性の行の抽出結果を代入
axes.hist(extracted_df['Age'], label='Age')    # グラフ領域にhist関数を用いて描画
axes.set_ylabel('Frequency')
axes.legend()
```

3.1. 指定した条件で行を抽出: query関数

`query` 関数を用いることで, より柔軟な条件を指定できる.

具体的には, 各列に対する抽出条件を文字列で記述. `[]` の抽出をより直感的にした記法.

```
titanic_df[titanic_df['Age'] < 10] # []を用いて記述する場合
titanic_df.query('Age < 10')      # query関数を用いて記述する場合
```

```
age_check = 65
titanic_df.query('Age > @age_check') # 条件文字列の中で, 変数を使用するには変数名の前に``@``をつける
titanic_df.query('30 <= Age < 50')  # 2つの比較演算子で値の範囲を指定可能
titanic_df.query('Age < Fare / 2')  # 列同士で比較したり, 算術演算子で計算して比較したりすることもできる

titanic_df[titanic_df['Sex'] == 'male'] # 条件文に文字列が含まれる[]の処理を, query関数の処理に置換する場合,
titanic_df.query('Sex == "male"')      # 左のように書くことで 文字列の中の文字列 を表現できる
```

文字列の中の文字列

Pythonにおいて文字列の中に文字列を定義する場合, `'` と `"` を組み合わせる.

パターン1: "文字列の中の'文字列'" or パターン2: '文字列の中の"文字列"'

3.2. 指定した条件でソート

`sort_values` 関数を用いて、各列の要素に基づく単一および複数の条件でソートできる。

```
# by(必須)          : ソートの基準となる列名を指定。  
#                  : リストを用いて列名を複数できる。その場合、リストの後ろの列名からソートされる。  
# ascending(任意): ソートの昇順/降順をTrue/Falseで指定。デフォルトはTrueで昇順。  
#                  : byで複数列を基準とした場合、それぞれの昇順/降順をリストで渡すことができる。  
df.sort_values(by, ascending)
```

```
titanic_df.sort_values('Age') # Ageを昇順したソート結果を取得  
titanic_df.sort_values(['Fare', 'Age'], ascending=[False, True]) # Ageを昇順後に、Fareの降順したソート結果を取得
```

ソート後の並び順に従ってindexを振り直す場合、`reset_index` 関数を用いる。

引数 `drop` を `True` にするとソート前のindexを破棄する。

```
titanic_df.sort_values('Age').reset_index() # 新たなindexを付与  
titanic_df.sort_values('Age').reset_index(drop=True) # 新たなindexを付与し、ソート前のindexを破棄
```


3.3. 要素の値の置換

`replace` 関数を用いることで、要素の値を指定して古い値から新しい値へ置換できる。
また、Pythonの `dict` という辞書型のデータ構造を用いると列毎に置換内容を指定可能。
なお、ここでは解説しないが正規表現も扱える。

```
titanic_df.replace('C', 'Cherbourg') # 文字列  
titanic_df.replace(['C', 'Q'], ['Cherbourg', 'Queenstown']) # 複数の値  
titanic_df.replace(1, '1等') # 数値を指定し、文字列へ置き換えもできる  
titanic_df.replace({'Pclass': {1: '1等'}})
```

辞書型のデータ構造: `dict`

Pythonは、索引（`key`）と値（`value`）を紐づけたデータ構造 `dict` を扱える。

リストで例えると `key` がインデックス、`value` が変数の値といった関係である。

```
sample_dict = {'key1': value1, 'key2': value2, ..., 'key3': value3}
```

```
sample_dict['key1'] # 索引 'key1'に紐づけられた値 value1 を参照
```

3.4. ユニークな値の確認

`unique` 関数によって列(=Series)に含まれるユニークな値の一覧を取得できる。
また `value_counts` 関数を用いることで、各ユニークな値の出現回数を取得できる。

```
titanic_df['Embarked'].unique() # ユニークな値以外に、nanという欠損値があることも確認できる
result_series = titanic_df['Embarked'].value_counts() # nan以外のユニークな値について出現回数を計算

# 出港地毎の乗客数を棒グラフと円グラフで確認: pandas
result_series.plot.bar()
result_series.plot.pie()

# 出港地毎の乗客数を棒グラフと円グラフで確認: Matplotlib
fig, axes = plt.subplots(ncols=2, figsize=(12.8, 4.8))
axes[0].bar(result_series.index, result_series) # x軸にvalue_countsのユニーク値, y軸に頻度を指定
axes[1].pie(result_series, labels=result_series.index) # グラフの元となる値として頻度, 各ラベルとしてユニーク値を指定
axes[1].set_ylabel('Embarked')
```

欠損値: NaN, nan

pandasでは、欠損値を一律に `NaN` という特別な定数で表現する。

`NaN` は, not a numberの略称である。

3.5. 欠損値: NaN

欠損値 NaN について様々な処理をする関数が実装されている.

isnull 関数を用いることで各要素が NaN かどうかを判定できる.

また sum 関数を組み合わせることで, NaN の個数を算出できる.

```
titanic_df.isnull()          # どの要素がNaNかTrue/Falseで取得  
titanic_df.isnull().sum()    # sum関数を組み合わせることで, NaNの出現回数を算出
```

dropna 関数を用いることで NaN を含む行や列を削除できる.

引数 how に応じて削除の方針を変更できる.

また, 引数 axis に応じて how で削除する方向を行か列か変更できる.

```
titanic_df.dropna(how='all', axis=0) # all, 0: 全ての値がNaNである行を削除  
titanic_df.dropna(how='any', axis=0) # any, 0: NaNを一つでも含む行を削除  
titanic_df.dropna(how='any', axis=1) # any, 1: NaNを一つでも含む列を削除
```

3.5. 欠損値: NaN

`fillna` 関数を用いると `NaN` を指定した値に置換できる.

`replace` 関数と同様に, 列毎に置換する値を指定する場合は `dict` を用いる.

```
titanic_df.fillna(-1) # 全ての列のNaNを-1へ置換
age_mean_val = titanic_df['Age'].mean() # 列Ageの平均値を算出
titanic_df.fillna({'Age': age_mean_val}) # 列AgeについてのみNaNを平均値へ置換
titanic_df.fillna({'Age': age_mean_val, 'Cabin': -1}) # 列毎に異なる置換
```

`interpolate` 関数を用いる前後の値など特定の法則に従って `NaN` の値を補間する.

引数 `method` によって補間方法を変更できる. デフォルトは `linear` (=線形補間).

```
titanic_df.interpolate() # 線形補間
titanic_df.interpolate(method='ffill') # NaNではない一つ前の値で補間
titanic_df.interpolate(method='bfill') # NaNではない一つ後の値で補間
titanic_df.interpolate(method='spline', order=3) # 3次のspline曲線で補間
```

3.6. 統計量の算出

PythonやNumPyと同様に, pandasには数値計算の関数が実装されている.

```
titanic_df.count()          # 数値計算できる全ての列について実行される
titanic_df['Age'].mean()     # 単一の列を選択しても使用可能
titanic_df.std()
titanic_df.min()
titanic_df.max()

# 上記の関数で算出される各列の要約統計量を同時に把握できる関数
titanic_df.describe()

# Fareについて値のばらつきを箱ひげ図で確認: pandas
titanic_df['Fare'].plot.box()

# Fareについて値のばらつきを箱ひげ図で確認: Matplotlib
fig, axes = plt.subplots()
axes.boxplot(titanic_df['Fare'])
```

3.7. 相関値の算出

`corr` 関数を用いて列間の相関値を算出できる. 欠損値 `NaN` は除外されて算出される.

```
# method(任意): 相関係数の種類を指定できる.  
#               'pearson': ピアソンの積率相関係数 (デフォルト)  
#               'kendall': ケンドールの順位相関係数  
#               'spearman': スピアマンの順位相関係数  
df.corr(method)
```

```
titanic_df.corr() # 計算可能な列間について相関係数が算出される  
  
# Survivedと相関の高い連続値 Fareについて散布図で確認: pandas  
titanic_df.plot.scatter(x='Fare', y='Survived', alpha=0.1)  
  
# Survivedと相関の高い連続値 Fareについて散布図で確認: Matplotlib  
fig, axes = plt.subplots() # 描画のための画像領域とグラフ領域を生成  
axes.scatter(titanic_df['Fare'], titanic_df['Survived'], alpha=0.1) # グラフ領域にscatter関数を用いて描画  
axes.set_xlabel('Fare')  
axes.set_ylabel('Survived')
```

3.8. 練習問題

2020年の豊中市の気温と降水量のデータ(`toyonaka_temp_2020.csv`)

および2021年の豊中市の気温と降水量のデータ(`toyonaka_temp_2021.xlsx`)について,

1. それぞれをDataFrameとして読み込め.
2. 降雨量の折れ線グラフをそれぞれのDataFrameで描画せよ.
3. 気温の棒グラフをそれぞれのDataFrameで描画せよ.
4. 気温と降雨量の相関をそれぞれのDataFrameで求めよ.
5. 各DataFrameの気温と降雨量の散布図を1枚の画像で描画せよ.

1. データの読み込み

```
drill_2020_file_path = './data/toyonaka_temp_2020.csv'
```

```
drill_2020_df = ???
```

```
drill_2021_file_path = './data/toyonaka_temp_2021.xlsx'
```

```
drill_2021_df = ???
```