

## 機械学習用テキスト

### 1. 機械学習のための基礎

#### ○ 関数の定義

今まで、使ってきた、len() や sum() などは、[組み込み関数](#)といます。  
関数は、自分で作ることも可能です。関数を定義するには **def** を用います。

```
def 関数名(引数):  
    処理
```

例えば、greeting という関数を作成してみましょう。

```
 #'Hello'を表示する関数 greeting  
def greeting():  
    print('Hello')
```

・関数は、定義すると呼び出すことが可能です。

```
 #関数 greeting を呼び出し  
greeting()
```

※ 1行目はヘッダと呼ばれ、関数名はその関数を呼ぶのに使う名前、引数はその関数へ渡す変数の一覧です。引数がない場合もあります。

#### ○ 引数と戻り値

- ・『**引数**』とは関数を呼び出す際に与える値のことで、関数は与えられた引数に従って処理を実行します。
- ・『**戻り値**』は関数を実行した時に返される値のことです。

※ 数学で学習する関数「 $z = f(x, y)$ 」をイメージすると『**引数**』と『**戻り値**』が理解しやすいでしょう。これは、関数  $f$  に引数「 $x$  ,  $y$ 」を渡すと、戻り値「 $z$ 」が返ってくるということを意味しています。

-----サンプルコード-----

```
def calc_product(x, y): # 関数 calc_product の定義  
    p = x * y  
    return p
```

#この時点で、関数 calc\_product に p が返されている状態 } 難しい・・・  
つまり、この後、引数などに変化があれば、p の値も変化する。

☆ 実行してみよう！

-----関数実行サンプル-----

```
calc_product(3,5)
```

結果：15

☆ 引数ってどんな時に使うの？

上記のプログラムからも分かるように、渡した値によって処理結果が変わってほしい場合には、引数を渡す必要がある！

## 2. モジュール

### (ア) **import**

Python では特別な関数や値をまとめたもの（これを**モジュール**といいます）を使うために、`import` という文を使います。

具体的には次のように記述します。

```
import モジュール名
```

たとえば、数学関係の機能をまとめた `math` というモジュールがあります。これらの関数や値を使いたいときは、以下のようにして `math` モジュールをインポートします。そうすると、**math.関数名** という形で関数を用いることができます。

実行は、以下のような形で記述します。

```
モジュール名.モジュールの中の関数名
```

※「`.`」を使うことで `math` モジュールの中の `○○` 関数を使うということを示す。

【例】 -----

```
import math # import はセルの一番上に記述します
print(math.sqrt(2)) # sqrt は平方根を計算する関数
print(math.pi) # π の値
print(math.sin(math.pi/4)) # sin 関数
print(math.cos(0)) # cos 関数
print(math.log(32,2)) # 2 を底とする 32 の対数
```

### (イ) **from**

基本的に、以下のように記述します。

```
from モジュール名 import モジュールの中の関数名
```

【例】 -----

```
from math import sqrt
print(sqrt(2)) # sqrt は平方根を計算する関数
```

```
from math import pi
print(pi) # π の値
```

```
from math import sin
print(sin(math.pi/4)) # sin 関数
```

```
from math import cos
print(cos(0)) # cos 関数
```

```
from math import log
print(log(32,2)) # 2 を底とする 32 の対数
```

### (ウ) **as**

モジュール名が長すぎるなどの理由から別の名前としたい場合は、`as` を利用する方法もあります。

【例】

```
numpy as np
```

### 3. pandas

Pandas とは、データ分析に特化したライブラリである。

表形式のデータとは以下のようなものである。

名前	年齢	住所	血液型
佐藤	21	東京都	A
斎藤	30	岐阜県	不明
鈴木	18	埼玉県	O
田中	26	千葉県	AB

このような、表形式のデータのことを DataFrame という。

表形式のデータにすることで、条件検索などのフィルタリング処理などがしやすくなるという利点がある。

○ 実際に Dataframe を作成してみよう！

```
import pandas as pd

pd.DataFrame({
    '名前': ['佐藤', '斎藤', '鈴木']
    '年齢': [21, 30, 18]
    '住所': ['東京都', '岐阜県', '埼玉県']
    '血液型': ['A', 'AB', 'O']
})
```

? DataFrame の作成って、データの数が多くなったら大変では・・・？

↓

☆ 通常は、Excel ファイルや CSV ファイルといった元から表形式になっているデータを取り込むことで作成します。

☆ 実際に Excel ファイルを読み込んで実行してみよう！

① Excel ファイルの単純な読み込み

```
import pandas as pd

df = pd.read_excel('user_data.xlsx')
```

② Excel ファイルのシート(2月分のデータのみ)を指定して読み込み

```
import pandas as pd

df = pd.read_excel(
    'user_data.xlsx', sheet_name='2月'
)
```

☆ DataFrame は、抽出条件で操作が可能！(次は CSV ファイルで練習)

【例】で挙げている表の中から 20 歳以上の人を抽出する。

```
import pandas as pd

df = pd.read_csv('xxx.csv')
df[df['年齢'] >= 20]
```

☆ 複数条件も可能！

【例】で挙げている表の中から 20 歳以上かつ A 型以外の人を抽出する。

```
import pandas as pd

df = pd.read_csv('xxx.csv')
df[(df['年齢'] >= 20) & (df['血液型'] != 'A')]
```

「~以外」を指定するときは、!= を使用する

Python のリストで表形式のデータを作成することも可能。

しかし、pandas を用いた方が良い。なぜだろう？

```
# データの作成
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
cities = ['New York', 'San Francisco', 'Los Angeles']

# データの結合
data_list = list(zip(names, ages, cities))

# データの表示
for item in data_list:
    print(item)
```

Pandasを使用すると、データの構造を持つDataFrameが利用でき、データのフィルタリングや処理が容易になります。一方で、組み込み関数では手動でリストを操作する必要があり、データの管理が複雑になります。Pandasはデータ処理を効率的かつ柔軟に行うためのツールセットを提供しています。

`list(zip(names, ages, cities))`は、`zip`関数を使用して複数のリストをまとめ、それをリストに変換するPythonの表現です。

例えば、`names`の1番目の要素は'Alice'、`ages`の1番目の要素は25、`cities`の1番目の要素は'New York'です。これらが`zip`関数で組み合わせられ、`(Alice, 25, New York)`というタプルができます。これをリストにまとめると、`[(Alice, 25, New York), (Bob, 30, San Francisco), (Charlie, 35, Los Angeles)]`というリストが得られます。

**処理が複雑になったときには、リストの中にタプルを作成するような形になり、非効率である！**

**それよりも、Dataframe で一括に管理できる Pandas が効率的である！**

【pandas を用いて統計量を調べる際によく使われるもの】

```
import pandas as pd

df = pd.read_csv('xxx.csv')

df.mean() ← 平均
df.max() ← 最大値
df.min() ← 最小値
df.sum() ← 合計
```

☆ 小規模データ → `list(zip)`

大規模データ → `pandas`

#### 4. NumPy

NumPy とは、高度な数値計算ができるモジュールである。

- まずは、インストールをして処理をするための準備をしよう！

```
install numpy
```

- インストールしたら、NumPy をインポートします。

```
import numpy as np
```

#### ☆ ndarray を理解しよう！

ndarray とは、データの加工・計算をするために作られた配列のことである。Python にもリストがあるから、わざわざ新しく作る必要がないと感じる人もいるかもしれません。

しかし、リストとは決定的に異なることがあります。

- ① まずは、1次元配列を作成して比較してみましょう。

```
import numpy as np
x = np.array([10, 14, 19])
x_2 = x * 2
print(x_2)
---Python のリスト---
y = [10, 14, 19]
y_2 = y * 2
print(y_2)
```

実行結果からも分かるように、NumPy を用いることで、配列内に数値を全て2倍した数値が返されたのに対し、リストを用いた記述では、リスト内の数値が2回繰り返される形で表示されている。

このように、配列内の数値を変化させて返したいときなどは NumPy を用いると便利である。

また、1次元だけでなく、2次元配列や3次元配列でも利用が可能である

#### ○ 多次元配列の作成と配列の変形

##### 1. 多次元配列の作成

多次元配列の作成については以下のような記述が基本的である。

```
import numpy as np

x = np.array(多次元リスト)
```

#### ☆ 2次元配列の作成

```
import numpy as np

x_2 = np.array(
    [[10, 14, 15],
     [20, 24, 26]]
)
print(x_2)
```

#### ☆ 3次元配列の作成

```
import numpy as np

x_3 = np.array(
    [[[10, 11], [13, 14], [16, 17]],
     [[20, 21], [23, 24], [26, 27]]]
)
print(x_3)
```

○ 配列は初期値を設定する場合もある。

初期値を設定する理由としては、配列の形(行数や列数など)を決めておき、処理が行いやすいようにするためである。また、未定義値の混入を防ぐためという目的もある。

よく使用される初期値は、0もしくは1である。それぞれの作り方を練習してみよう！

【要素が全て0の配列】

基本的な形は、以下の通りです。

```
x = np.zeros(作りたい配列のサイズ)
```

※ ここでサイズは shape ともいい()内に数値を指定する。

以下のコードを実行し、結果を見てみよう！

```
x = np.zeros(3)
print(x)
```

```
x = np.zeros((2, 3))
print(x)
```

※ 全て1の配列を作りたい場合は「np.zeros」の箇所を「np.ones」と記述する。

## 2. 配列の変形

(ア) reshape

配列の形を変形するものとして「reshape」がある。

```
x = np.array(
[[10, 12, 14],
 [20, 22, 24]]
)
result = x.reshape((3, 2))
print(result)
```

実行結果からも分かるように、2行3列の配列が3行2列に変化している。

(イ) flatten

要素の値はそのまま、1次元配列に変形するものとして「flatten」がある

```
x = np.array(
[[[1, 2], [3, 4], [5, 6]],
 [[7, 8], [9, 10], [11, 12]]]
)
result = x.flatten()
print(result)
```

実行結果を見ると、1次元配列に形が変わっていることが分かる！

※ 配列の変形に関しては、「手書き数字の判定」で重要になる！覚えておこう！

【NumPy 関数のまとめ】

np. max(配列およびリスト) ← 最大値  
np. min(配列およびリスト) ← 最小値  
np. sum(配列およびリスト) ← 合計  
np. mean(配列およびリスト) ← 平均  
np. std(配列およびリスト) ← 標準偏差  
np. var(配列およびリスト) ← 分散  
np. median(配列およびリスト) ← 中央値

この他にも、多くの関数が存在します。自分で調べて学習してみてください。

## 5. matplotlib

matplotlib はグラフを描画するためのモジュールです。

簡単な折れ線グラフを描画してみよう！

【x軸が1～5、y軸が1～25の折れ線グラフを作成する。】

```
import matplotlib.pyplot as plt

# データの準備
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

# グラフの描画
plt.plot(x, y, label='y = x^2')

# ラベルとタイトルの設定
plt.xlabel('X 軸')
plt.ylabel('Y 軸')
plt.title('シンプルな折れ線グラフ')

# 凡例の表示
plt.legend()

# グラフの表示
plt.show()
```

## ☆ 紹介 ☆

数学で学習する、ヒストグラムや箱ひげ図も描画することは可能です。

サンプルコードも載せておきますので、時間があれば取り組んでみてください。

```
import matplotlib.pyplot as plt
import numpy as np

# サンプルデータ: テストの点数
test_scores = [55, 58, 65, 65, 66, 67, 68, 70, 72, 72, 75, 78, 80, 82, 85, 88, 90, 92, 93, 95, 97, 98]

# グラフを1行2列で準備
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# ヒストグラム
axes[0].hist(test_scores, bins=10, color='skyblue', edgecolor='black')
axes[0].set_title('テストの点数分布 (ヒストグラム)')
axes[0].set_xlabel('点数')
axes[0].set_ylabel('生徒数')

# 箱ひげ図
axes[1].boxplot(test_scores, vert=False, patch_artist=True, boxprops=dict(facecolor='lightgreen'))
axes[1].set_title('テストの点数 (箱ひげ図)')
axes[1].set_xlabel('点数')

# グラフの表示
plt.tight_layout()
plt.show()
```

## 機械学習① 「アヤメの分類」

### 1. データをダウンロードしよう

データの確認とドライブのマウント

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt
import seaborn as sns # matplotlib を使用しグラフを表示する際に、よ
りきれいにグラフを表示させるためのもの

iris_df = pd.read_csv("マウントしてパスのコピー")
iris_df.head()
```

### 2. 出力されたグラフからアヤメの品種間での特徴の違いを読み取ろう

```
f, ax=plt.subplots(2,2,figsize=(18,8))

sns.countplot(x="SepalLengthCm",hue='Species',data=iris_df,
ax=ax[0][0])
ax[0][0].set_title('SepalLengthCm vs Species')

sns.countplot(x="SepalWidthCm",hue='Species',data=iris_df,a
x=ax[1][0])
ax[1][0].set_title('SepalWidthCm vs Species')
```

```
sns.countplot(x="PetalLengthCm",hue='Species',data=iris_df,
ax=ax[0][1])
ax[0][1].set_title('PetalLengthCm vs Species')

sns.countplot(x='PetalWidthCm',hue='Species',data=iris_df,a
x=ax[1][1])
ax[1][1].set_title('PetalWidthCm vs Species')
plt.show()
```

### 3. 実際に pandas を用いて機械学習をしてみよう！

```
#特徴量とラベルの分割
X = iris_df.drop(["Id","Species"], axis=1)
y = iris_df['Species']

#データの標準化
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

#データの分割
X_train, X_test, y_train, y_test =
train_test_split(X_scaled, y, test_size=0.2,
random_state=42)

#学習・テスト
svm_model = SVC(kernel='linear', random_state=42)

svm_model.fit(X_train, y_train)

y_pred = svm_model.predict(X_test)

#評価
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

## 機械学習②「手書き数字の判定」

### 1. 手書き数字のデータセット

scikit-learn に標準で付属している、Optical Recognition of Handwritten Digits Data Set(手書き数字の光学認識データセット)を使用します。8×8 ピクセルの手書き数字のデータが 5620 個ほど用意されています。

#### (1) データの確認

どのようなデータかを確認します。Google を起動して、以下のプログラムを記入して実行してみます。データセット内の 15 個の手書き数字を出力するプログラムです。

```
# 画像表示のために、matplotlib.pyplot を plt として読み込む。  
(matplotlib.pyplot は、表やグラフを作成してくれるモジュールの一つ。)
```

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

```
# 手書きデータを読み込む
```

```
from sklearn import datasets    ←sklearn から datasets を読み込む。  
digits = datasets.load_digits() ←digits にデータを読み込む。
```

```
# 15 個連続で出力する
```

```
for i in range(15):
```

```
    plt.subplot(3, 5, i+1) # 3 行 5 列でサブプロットを作成[i+1 は画像の位置]
```

```
    plt.axis("off") # 軸目盛を off(on も試してみよう！)
```

```
    plt.title(str(digits.target[i])) # [i]で数字を指定。str で文字列に変換することで、  
    グラフのタイトルとして表示
```

```
    plt.imshow(digits.images[i], cmap="gray") ←cmap の後に色を指定。  
    今回はグレースケール！
```

```
plt.show()
```

実行すると、以下のような画像データが表示されます。(図 1)



図 1：手書き数字のデータは 8×8 ピクセルの画像データ

#### (2) 画像のフォーマットを確認する(手書き数字の準備も含む)

手書き数字は、8×8 ピクセルで、各ピクセルは 0 から 16 までの値で表されています。今回の画像データは、0 が透明(背景色で黒色)で、16 が線のある部分(白色)を表します。以下のプログラムを入力して確認してみましょう。実行結果は図 2 のようになります。

```
d0 = digits.images[0]  
plt.imshow(d0, cmap="gray")  
plt.show()  
print(d0)
```

<実行結果>



図 2：手書き数字のデータは 8×8 ピクセルの画像データ

## 2. 画像を機械学習する

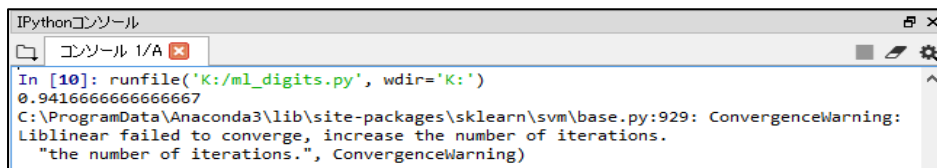
手書き数字の画像データをもとに機械学習を行います。数字の画像と正しい数字の組み合わせ（教師）を用意したうえで、手書き画像の数字を予想させます。

### (1) 画像を学習する

1つの数字の画像のデータは8×8ピクセルの二次元配列となっているため、一次元配列にして64個のピクセル値のデータとします。また、全体のデータのうち8割を学習用データ、2割をテスト用データに振り分け、学習とテストを行います。

```
from sklearn.model_selection import train_test_split
from sklearn import datasets, svm, metrics
from sklearn.metrics import accuracy_score
#データを読む
digits = datasets.load_digits()
x = digits.images
y = digits.target
x = x.reshape((-1, 64)) # 二次元配列を一次元配列に変換
# データを学習用とテスト用に分ける
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.2)
#データを学習する
clf = svm.LinearSVC()
clf.fit(x_train,y_train)
#予測して精度を確認する
y_pred = clf.predict(x_test)
print(accuracy_score(y_test, y_pred))
```

<実行結果>



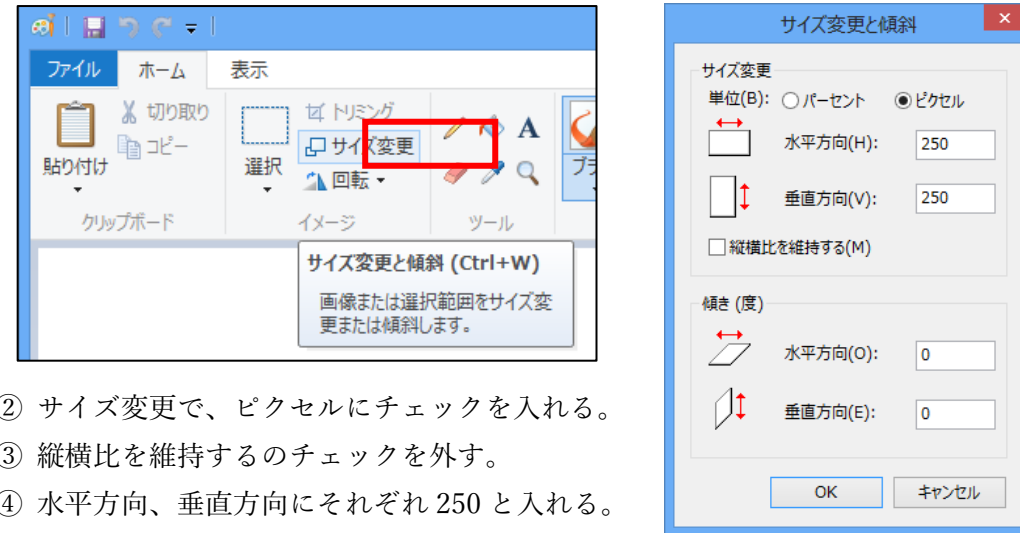
```
IPythonコンソール
コンソール 1/A
In [10]: runfile('K:/ml_digits.py', wdir='K:')
0.9416666666666667
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\svm\base.py:929: ConvergenceWarning:
Liblinear failed to converge, increase the number of iterations.
"the number of iterations.", ConvergenceWarning)
```

## 3. 手書き数字の画像を用意する

デスクトップの『すべてのアプリを表示する』のアイコンをダブルクリック。ペイントを開きます。

### (1) 正方形の画像を用意する

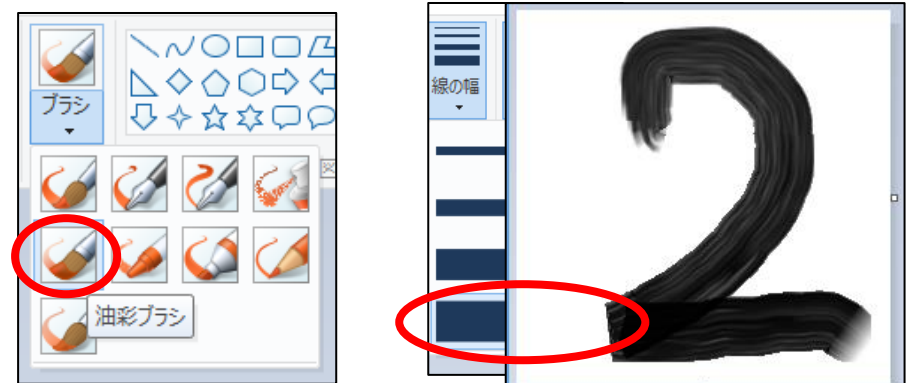
① 『ホーム』タブから『サイズ変更』をクリックします。



- ② サイズ変更で、ピクセルにチェックを入れる。
- ③ 縦横比を維持するのチェックを外す。
- ④ 水平方向、垂直方向にそれぞれ 250 と入れる。

### (2) 数字を描いて保存する。

① 油彩ブラシを選択 ②線の太さは 30pt か 40pt ファイル名：my2.png



#### 4. 学習済みデータの保存と呼び出し

学習済みのデータを保存するには、pickle モジュールを用います。

以下のコードで実行してみましょう！

(1) 学習済みデータを保存する。

```
# 学習済みデータを保存する
import pickle
with open("digits.pkl", "wb") as fp:
    pickle.dump(clf, fp)
```

(2) 学習済みデータを呼び出す(読み込み)。

```
# 学習済みデータを読み込む
import pickle
with open("digits.pkl", "rb") as fp:
    clf = pickle.load(fp)
```

#### 5. 画像を判定する

以下のプログラムを実行し、用意した手書き数字の画像を判定してみましょう。

ファイル名：predict-myimage.py

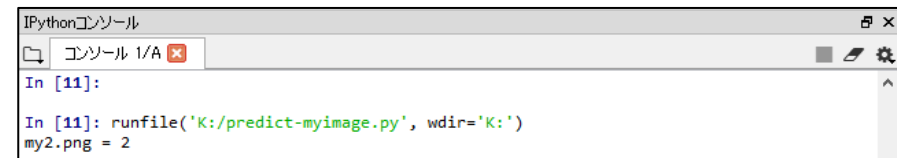
```
import cv2
import pickle
def predict_digit(filename):
    #学習済みデータを読み込む
    with open("digits.pkl", "rb") as fp:
        clf = pickle.load(fp)
    #自分で用意した手書きの画像ファイルを読み込む
    my_img = cv2.imread(filename)
    #画像データを学習済みデータに合わせる
    my_img = cv2.cvtColor(my_img, cv2.COLOR_BGR2GRAY)
    my_img = cv2.resize(my_img,(8,8))
```

```
my_img = 15 - my_img // 16 #白黒反転させる
# 二次元配列を一次元配列に変換
my_img = my_img.reshape((-1,64))
#データを予測する
res = clf.predict(my_img)
return res[0]
```

# 画像ファイルを指定して実行

```
n = predict_digit("my2.png")
print("my2.png = " + str(n))
n = predict_digit("my4.png")
print("my4.png = " + str(n))
```

<実行結果>



```
IPythonコンソール
コンソール 1/A
In [11]:
In [11]: runfile('K:/predict-myimage.py', wdir='K:')
my2.png = 2
```

<考えよう>

① 手書き数字が正しく判定されないことがあるがそれは何故だろうか？

② 精度のよいプログラムにするには、どのような工夫が必要だと考えますか。

## 1. 単回帰分析

アイスクリームの支出金額と気温のデータから回帰式を求め、気温による予測を行ってみましょう。

<前準備>ドライブを【 】します

```
from google.colab import drive
drive.mount('/content/drive')
```

### 1. 必要なライブラリを読み込みます

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

### 2. データを読み込みます。

データ内容は以下の通りです。

ファイル名: ice.csv

year: 年

month: 月

spending: 支出

temperature: 気温

```
df = pd.read_csv("ice.csv のパスをコピー") #データの読み込み
df.head()
```

## 3. 単回帰分析を行う

説明変数を temperature、目的変数を spending として、単回帰分析を行います。読み込んだデータの分布を散布図で確認します。

### ☆ 散布図の作成

```
# 説明変数 temperature
X = df[['temperature']] #[行,列], :は全ての列, 行は0~スタート (IDは除く)

# 目的変数 spending
Y = df[['spending']]

plt.plot(X, Y, 'r.')
plt.show()
```

### ☆ 直線の式を求める

```
# sklearn.linear_model.LinearRegression クラスを読み込み
from sklearn import linear_model
lr = linear_model.LinearRegression()

# 予測モデルを作成
lr.fit(X, Y)

# 回帰係数
print("回帰係数:", lr.coef_)

# 切片
print("切片:", lr.intercept_)

# 決定係数
print("決定係数:", lr.score(X, Y))
```

4. 作成した予測モデルから回帰直線をプロットしてみます。

#回帰直線をプロット

```
plt.plot(X, Y, 'o')  
plt.plot(X, lr.predict(X), linestyle="solid")  
plt.show()
```

<課題1>

回帰係数（傾き）と切片から、回帰式を作って、販売数を予測するプログラムを作成しましょう。

支出金額を予測する際の気温の変数名をx1とし、予測値をy1とする。

x1はキーボード入力しましょう。

【入力の書式】

- ・ 文字列の入力

```
srt = input('入力エリアに表示するメッセージ')
```

- ・ 数値の入力（型変換が必要）

```
su = int(input(('入力エリアに表示するメッセージ')))
```

