



技能向上訓練 A I 基礎コース

Python基礎

Google Colaboratory

- <https://colab.research.google.com/notebooks/intro.ipynb>
- クラウド上で実行される、Jupyter Notebook
- Googleアカウントでログインが必要



ノートブック を新規作成

- ファイル→ノートブックを新規作成





Untitled0.ipynb ☆

ファイル 編集 表示 挿入 ランタイム ツール ヘルプ すべての変更を保存しました



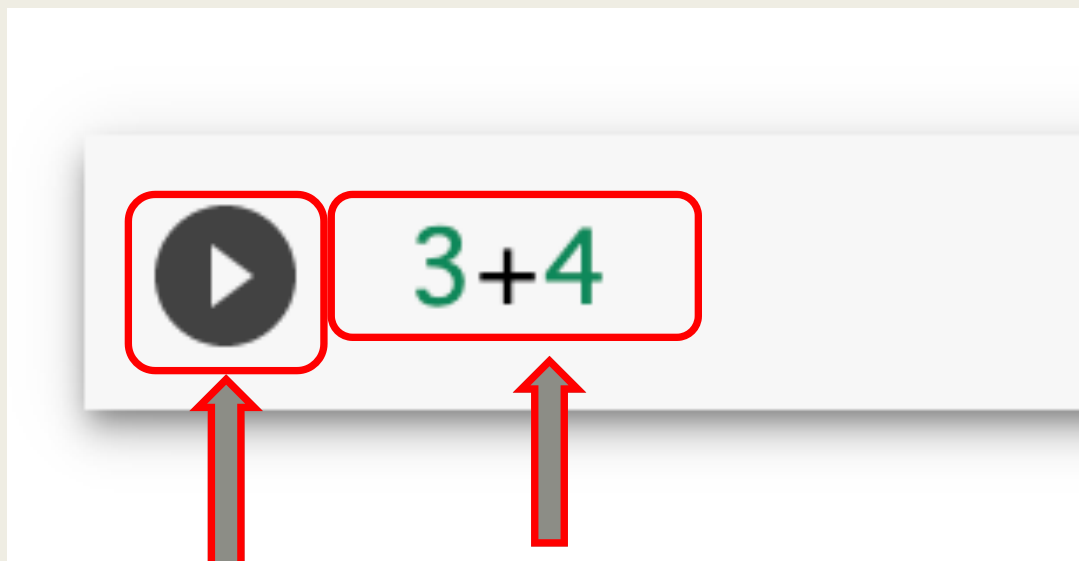
+ コード + テキスト



|



セル
ここに、Pythonコードを入力



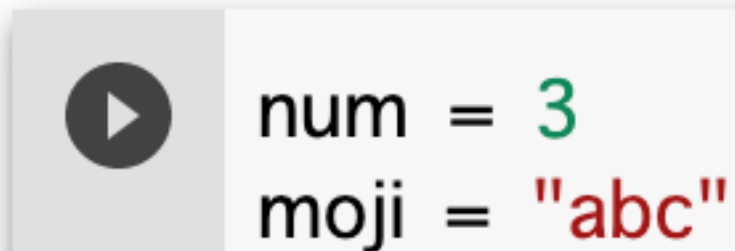
①セルに、
「3+4」と入力

②実行をクリック
又は、「Shift + RETURN」



実行結果が表示される

変数にデータを代入

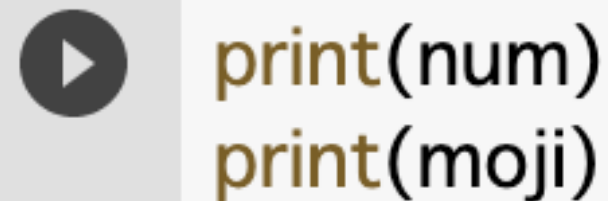


```
num = 3  
moji = "abc"
```

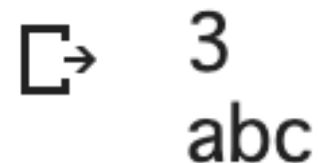
Numに 3 を代入

Mojiに文字列abcを代入

代入分のみなので、実行しても画面には何も表示されない。



```
print(num)  
print(moji)
```



```
3  
abc
```

print()で変数の内容を表示

変数データ型



```
a = 123
b = "moji"
c = 3.14
print(type(a))
print(type(b))
print(type(c))
```



```
<class 'int'>
<class 'str'>
<class 'float'>
```

データの種類をデータ型と言う。
type()でデータ型を確認できる。

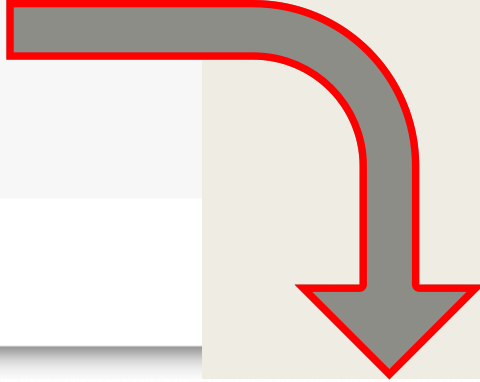
int:整数
str:文字列
float:実数

リスト型

- 変数を横に並べたもの。
- 一つの名前と、添字で要素を管理できる。

▶ `lst = [10, 20, 30, 40, 50]`
`lst[2]`

☐→ 30



	0	1	2	3	4
lst	10	20	30	40	50

スライシング

- リスト[先頭 : 末尾 + 1]にて、リストの範囲指定ができる。
- 先頭、末尾 + 1 の値は、省略しても良い。

```
[19] lst[1:4]
```

```
↳ [20, 30, 40]
```

```
[20] lst[:3]
```

```
↳ [10, 20, 30]
```

```
[21] lst[3:]
```

```
↳ [40, 50]
```

lst [1 : 4] → lst[1] ~ lst[3]

lst [: 3] → (先頭) lst[0] ~ lst[2]

lst [3:] → lst[3] ~ lst[4](最終)

Numpy


- PythonにはNumpyという便利なライブラリがあります。Numpyを利用すると、Python標準のリスト型に比べて、多次元配列のデータを効率よく扱うことができます。また、Numpyは標準偏差や分散といった統計量を出力してくれる関数が用意されており、科学技術計算の基盤となっています。
- Numpyを使用するには、以下のようにコードをうち、インポートをしてください。



```
import numpy as np
```

numpy.ndarray

- Numpyには、ndarrayと呼ばれるデータ型が用意されています。ndarrayは一見すると、Python標準のリスト型(list)と似ていますが、ベクトルや行列のように、内部の要素を一括して計算できる便利な機能があります。

```
 # arrayを定義する  
ary = np.array([1, 2, 3])  
print(ary)
```

```
 [1 2 3]
```

Ndarrayの計算



```
# aryのすべての要素に5をかける  
print(ary * 5)
```



```
[ 5 10 15]
```

画面に表示しただけは値は更新されないので注意。

```
[27] print(ary)
```

```
    ary = ary * 5
```

```
    print(ary)
```

代入処理

```
[ ] [1 2 3]
```

代入前

```
[ ] [ 5 10 15]
```

代入後

numpyでの基礎統計量

- 通常、標準偏差などを計算したい場合は自分で実装する必要がありますが、Numpyは基本的な統計量を計算する関数を用意しています。

▶ # range関数で、0~9までの連続した整数値を生成する
ary = np.array(range(10))
ary

☞ array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

統計計算

- 以下のコードを入力すると表示される値を確認してください。

```
[ ] # aryの平均値  
ary.mean()
```

4.5

```
[ ] # aryの分散  
ary.var()
```

8.25

```
[ ] # aryの標準偏差  
ary.std()
```

2.8722813232690143

```
[ ] # aryの最小値  
ary.min()
```

0

```
[ ] # aryの最大値  
ary.max()
```

9

行列の計算

- Narrayにて、行列を生成し、行列の計算を行うことができる。

```
[34] a = np.array([[1,2],[3,4]])  
      b = np.array([[5,6],[7,8]])  
  
      print(a@b) #行列積を求める
```

```
↳ [[19 22]  
    [43 50]]
```


Pandas

- Pandasは、Pythonでのデータ分析ライブラリとして最も活用されているライブラリのひとつです。Pandas はNumpyを基盤に、シリーズ(Series)とデータフレーム(DataFrame)というデータ型を提供します。
- Pandasを利用するには、まず次のようにインポートを行います。



```
import pandas as pd
```

pandas.Series

- ここまで、Python標準のリスト型(list)と、Numpyライブラリのndarrayを扱いましたが、Pandasライブラリにも同様の機能を有するシリーズ(Series)というデータ型があります。

```
▶ srs = pd.Series([10, 20, 30, 40, 50])  
srs
```

```
↳ 0    10  
   1    20  
   2    30  
   3    40  
   4    50  
   dtype: int64
```

リストやndarrayとの共通点

```
[38] # list/ndarrayとの共通点1 添字を指定して任意の要素を取り出せる  
      srs.iloc[3]
```

```
↳ 40
```

```
[39] # list/ndarrayとの共通点2 スライシングが使える  
      srs.iloc[ 1 : 3 ]
```

```
↳ 1  20  
   2  30  
   dtype: int64
```

リストやndarrayとの相違点

[40] # list/arrayと相違点1 indexを指定できる

```
srs = pd.Series([10, 20, 30, 40, 50], index=["one", "two", "three", "four", "five"])  
srs
```

```
↳ one    10  
   two    20  
   three  30  
   four   40  
   five   50  
   dtype: int64
```

▶ # list/ndarrayと相違点2 np.arrayよりも便利な機能(メソッド)が充実している
要約統計量を出力する
srs.describe()

```
↳ count    5.000000  
   mean     30.000000  
   std      15.811388
```

pandas.DataFrame

- Pandasライブラリで最も使われるデータ型にデータフレーム(DataFrame)があります。Seriesは1次元のデータしか扱えませんが、DataFrameは行列のような多次元のデータも扱えます。

▶ **# DataFrameの生成**
まずはサンプルデータを含んだ、DataFrameを生成してみます。
`df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`
`df`



	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

indexとcolumnの指定

- DataFrameはインデックス(index)やカラム(column)を指定することにより、Excelのシートのようにデータを扱うことができます。

```
▶ df = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], index=["a", "b", "c"], columns=["A", "B", "C"])  
df
```

↳

	A	B	C
a	1	2	3
b	4	5	6
c	7	8	9

indexとcolumn名の変更

- pandas.DataFrame.renameを使うことで、行や列の名前を変更することが出来ます。

```
[44] df.rename(columns={"A":"a"},index={"a":"A"})
```

↳

	a	B	C
A	1	2	3
b	4	5	6
c	7	8	9

注意

- DataFrameを更新したつもりでも、更新をしなければ変更はされません。
- 先ほどの行列名を変更したdfを再度表示すると、もとに戻っていることがわかります。

```
[44] df.rename(columns={"A":"a"},index={"a":"A"})
```

```
↳
```

	a	B	C
A	1	2	3
b	4	5	6
c	7	8	9

```
▶ df
```

```
↳
```

	A	B	C
a	1	2	3
b	4	5	6
c	7	8	9

解決方法①

オプションで「inplace=True」を追加

```
[46] df.rename(columns={"A":"a"},index={"a":"A"},inplace=True)
```

▶ df

↳

	a	B	C
A	1	2	3
b	4	5	6
c	7	8	9

解決方法②

代入して、上書きをする

```
[48] df = df.rename(columns={"C": "c"},index={"c": "C"} )
```



df



	a	B	c
A	1	2	3
b	4	5	6
C	7	8	9

データの消去①

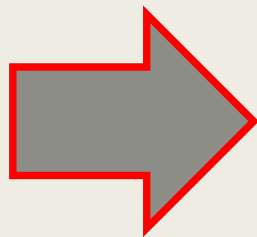
- `pandas.Series.drop`は、列と行の削除を行うことができます。デフォルトでは、行の削除(`axis=0`)を指定しています。



```
df.drop('C')  
df
```



	a	B	c
A	1	2	3
b	4	5	6
C	7	8	9



```
df.drop('C',inplace = True)  
df
```



	a	B	c
A	1	2	3
b	4	5	6

データの消去②

- axis=1を指定することで、列を削除することが出来る。



```
df.drop('c', axis=1, inplace=True)
```



	a	B
A	1	2
b	4	5

データの取得

- `pd.DataFrame.loc`と`pd.DataFrame.iloc`を使うことで、`DataFrame`の指定した行や指定した列を取得することができます。
- データの取得を行う前に、再度データフレームを作り直します。

```
▶ df = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], index=["a", "b", "c"], columns=["A", "B", "C"])  
df
```



	A	B	C
--	---	---	---

a	1	2	3
---	---	---	---

b	4	5	6
---	---	---	---

c	7	8	9
---	---	---	---

```
[55] # b行のデータを取得  
df.loc["b"]
```

```
↳ A    4  
   B    5  
   C    6  
   Name: b, dtype: int64
```

```
[56] # C列のデータを取得  
df.loc[:, "C"]
```

```
↳ a    3  
   b    6  
   c    9  
   Name: C, dtype: int64
```



0,1行目のデータを取得

```
df.iloc[ [0, 1] ]
```



	A	B	C
a	1	2	3
b	4	5	6



1列目のデータを取得

```
df.iloc[ :, [1] ]
```



	B
a	2
b	5
c	8

データの表示

- Dataframeにはheadとtailというメソッドが用意されています。headは先頭から指定した数の行データを表示します。tailは、末尾から指定した数の行データを表示します。

▶ df.head(2)

↳

	A	B	C
a	1	2	3
b	4	5	6

▶ df.tail(2)

↳

	A	B	C
b	4	5	6
c	7	8	9

条件によるデータの抽出

- Dataframeにはqueryというメソッドが用意されています。文字列で条件をすることで、条件を満たした行だけを取り出すことができます。SQLに詳しい方は、SELECT～WHERE句をイメージすると分かりやすいかもしれません。



B列が5の行を抽出

```
df.query('B == 5')
```



	A	B	C
--	---	---	---

b	4	5	6
----------	---	---	---

条件によるデータの抽出

- 以下のコードを入力し、実行結果を確認してください。

```
[ ] # B列が5以上の行を抽出  
df.query('B >= 5')
```

```
[ ] # B列が5以上で、かつC列が7以上の行を抽出  
df.query('B >= 5 & C >=7')
```

```
[ ] # A列が1、またはC列が9の行を抽出  
df.query('A == 1 | C ==9')
```

```
↳      A  B  C  
b    4  5  6  
c    7  8  9
```

```
↳      A  B  C  
c    7  8  9
```

```
↳      A  B  C  
a    1  2  3  
c    7  8  9
```

Pythonの基本文法

- 選択(if文)
- 繰り返し
- 関数

選択 (if 文)

- 選択(if文)
- if 文の使い方は以下の通りです。例えばJava言語では{ }で囲んでいた部分が、Pythonでは字下げによって行われていることに注意してください。
- if 条件:
実行したい文

※ifの中の行は、必ずインデント(字下げ)が必要です

選択 (if 文)

```
▶ lang = "python"  
if lang == "python":  
    # インテントされた部分がifの中身として認識される  
    print("I love Python!")  
    print("I love Programming!")  
  
print("End")
```

```
↳ I love Python!  
   I love Programming!  
   End
```

- 実行後、赤枠の中を別の文字列に変更してみて、再度実行してみましょう。

選択 (if 文)

- ところで、lang が "python" 以外のときにもなにか実行したい場合があるとしみましょう。そんなときに使うのが else文 です。else 文は、if文の条件を満たさなかった場合に実行したい文を書くことができます。 else文の使い方は、

if 条件:

 実行したい文

else:

 実行したい文

選択 (if 文)

```
▶ lang = "python"  
if lang == "python":  
    print("I love python!")  
else:  
    print("I love other language!")
```



```
I love python!
```

- 実行後、赤枠の中を別の文字列に変更してみて、再度実行してみましょう。

繰り返し(for文)

- 同じ処理を繰り返したい場合は、for文を使用します。他の言語に比べると、繰り返し条件の記述に独特の部分があるので、以下の実行例を参考にしてください。
- for 変数 in リスト型:
 繰り返したいしたい文
 ※ifと同様に、for中の行は、必ずインデント(字下げ)が必要です

繰り返し(for文)



```
# xを0から4まで増分させながら繰り返す  
for x in [0, 1, 2, 3, 4]:  
    print(x)
```



```
0  
1  
2  
3  
4
```

- 実行後、赤枠の中を別の数値リストに変更してみて、再度実行してみましょう。

繰り返し(for文)

```
[72] # xを0から4まで増分させながら繰り返す
      for x in range(5):
          print(x)
```

```
☞ 0
   1
   2
   3
   4
```

- range(5)の5を他の値に変更して再度確認してみましょう。

繰り返し(for文)

```
[74] # xを2から4まで増分させながら繰り返す
      for x in range(2, 5):
          print(x)
```

```
↳ 2
   3
   4
```

- range(2,5)を他の値に変更して再度確認してみましょう。

繰り返し(for文)

```
[75] # xを0から9まで2ずつ増分させながら繰り返す
      for x in range(0, 9, 2):
          print(x)
```

```
☐→ 0
      2
      4
      6
      8
```

- range(0,9,2)を他の値に変更して再度確認してみましょう。

関数の定義(def)

- Pythonに限らず、多くのプログラミング言語に関数(サブルーチン、メソッドとも)というものがあります。関数は数学で扱うように「何かの値を入れると、何らかの値を出力する」という役目がありますが、プログラミングの場合には、それと同時に「いくつかのまとまった処理をひとまとめにする」という役割もあります。Pythonでの関数の定義の方法は以下の通りです。

一時間数 $y = 3x + 5$ の関数を作ってみる。

def 関数名(引数):

 実行したい文

 ...

 return 出力したい値

```
[76] def func(x):  
      y = 3 * x + 5  
      return y
```

```
[77] func(4)
```

```
↳ 17
```

関数の定義(def)

- 引数はいくつあっても良い。

```
[78] # x * y を 計算する関数を定義する(関数名:calc)
      def calc( x , y ):
          result = x * y
          return result
```

```
[79] calc(4,5)
```

```
↳ 20
```

関数の定義(def)

- 関数の結果を変数に代入することも可能

```
[80] # 戻り値を変数(kakezan1と2)に代入する
      kakezan1 = calc(5,5)
      kakezan2 = calc(10,10)
      print( kakezan1 + kakezan2 )
```

```
↳ 125
```

Matplotlibの基礎

- データを分析する際に、値を見るだけでは全体像を捉えにくいので、まずはグラフで描画して俯瞰することが多いです。ここでは、Pythonでよく利用されるグラフ描画のライブラリであるMatplotlibの簡単な使い方をご紹介します。
- まずは、matplotlibを使用するために、インポートします。



```
# Matplotlibライブラリを、pltという別名でできるように宣言する  
import matplotlib.pyplot as plt
```


Matplotlibの基礎

- グラフの表示の前に、pandasのデータフレームで、グラフの元となるデータを作成します。

```
[81] # データフレームを作成する  
# サンプルデータ:気温(°C)とアイスクリームの売上(個)の関係  
df = pd.DataFrame([ [10,20],[15,30],[15,40],[25,70],[30,100] ], columns=["気温", "売上"])  
df
```

☞ 気温 売上

0	10	20
---	----	----

1	15	30
---	----	----

2	15	40
---	----	----

3	25	70
---	----	----

4	30	100
---	----	-----

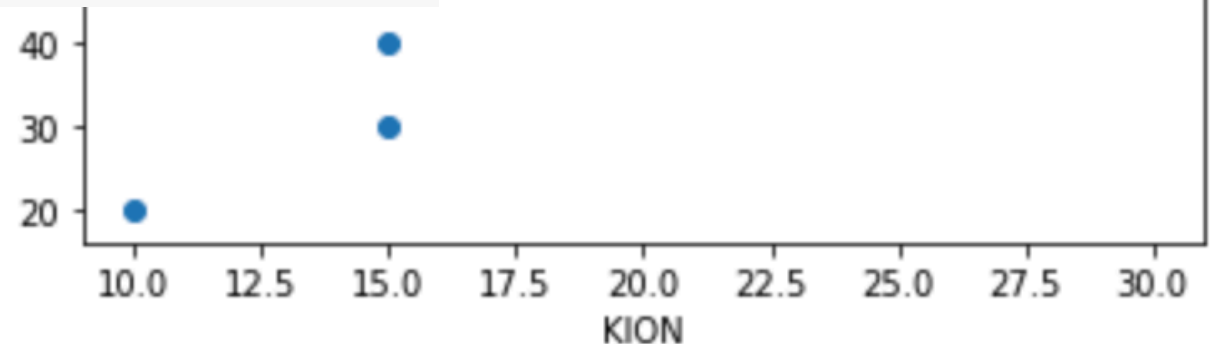
Matplotlibの基礎

■ 散布図



散布図1

```
plt.figure()  
plt.scatter(x=df['気温'], y=df['売上'])  
plt.xlabel('KION')  
plt.ylabel('URIAGE')  
plt.show()
```



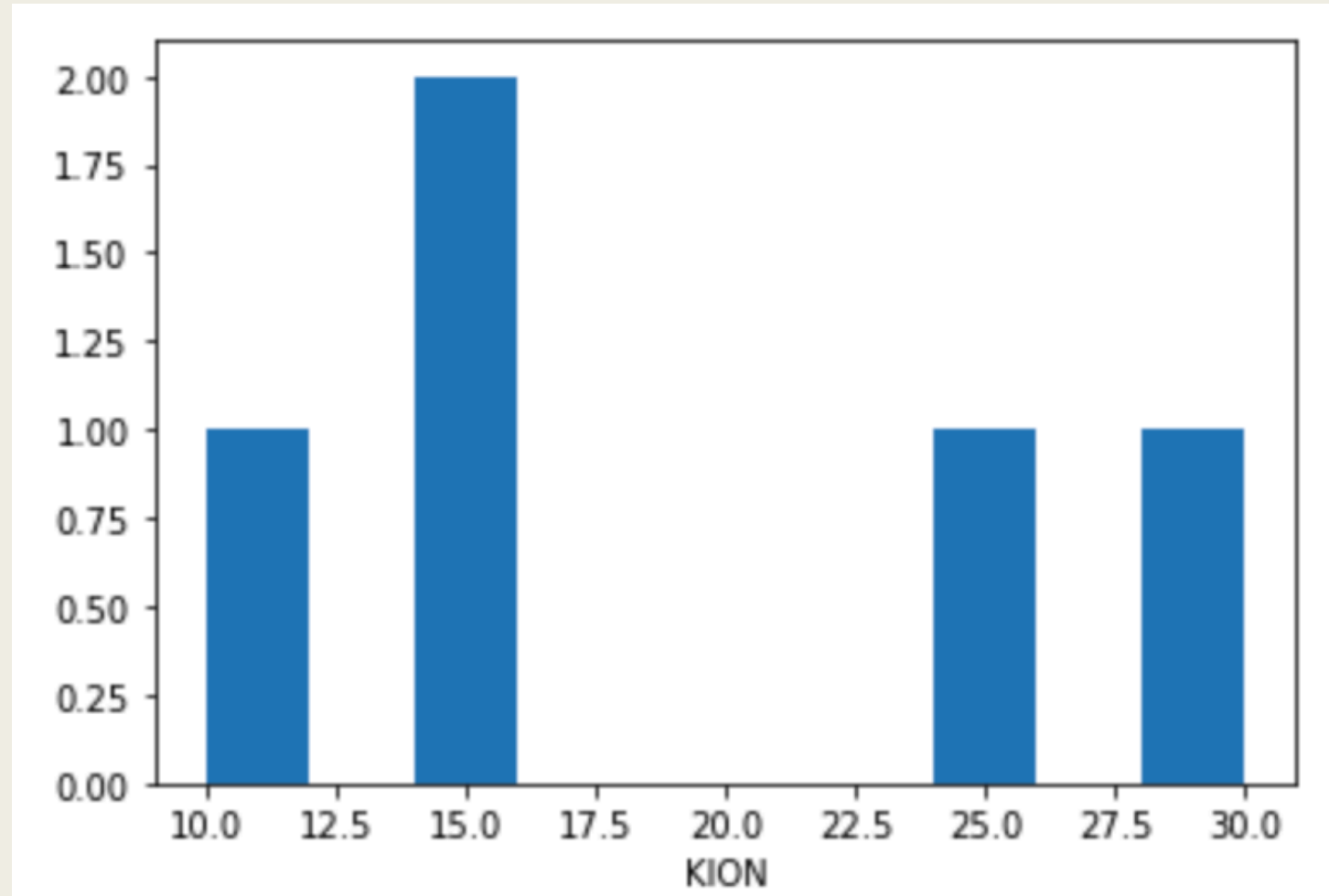
Matplotlibの基礎

■ ヒストグラム



ヒストグラム

```
plt.figure()  
plt.hist(df['気温'])  
plt.xlabel('KION')  
plt.show()
```



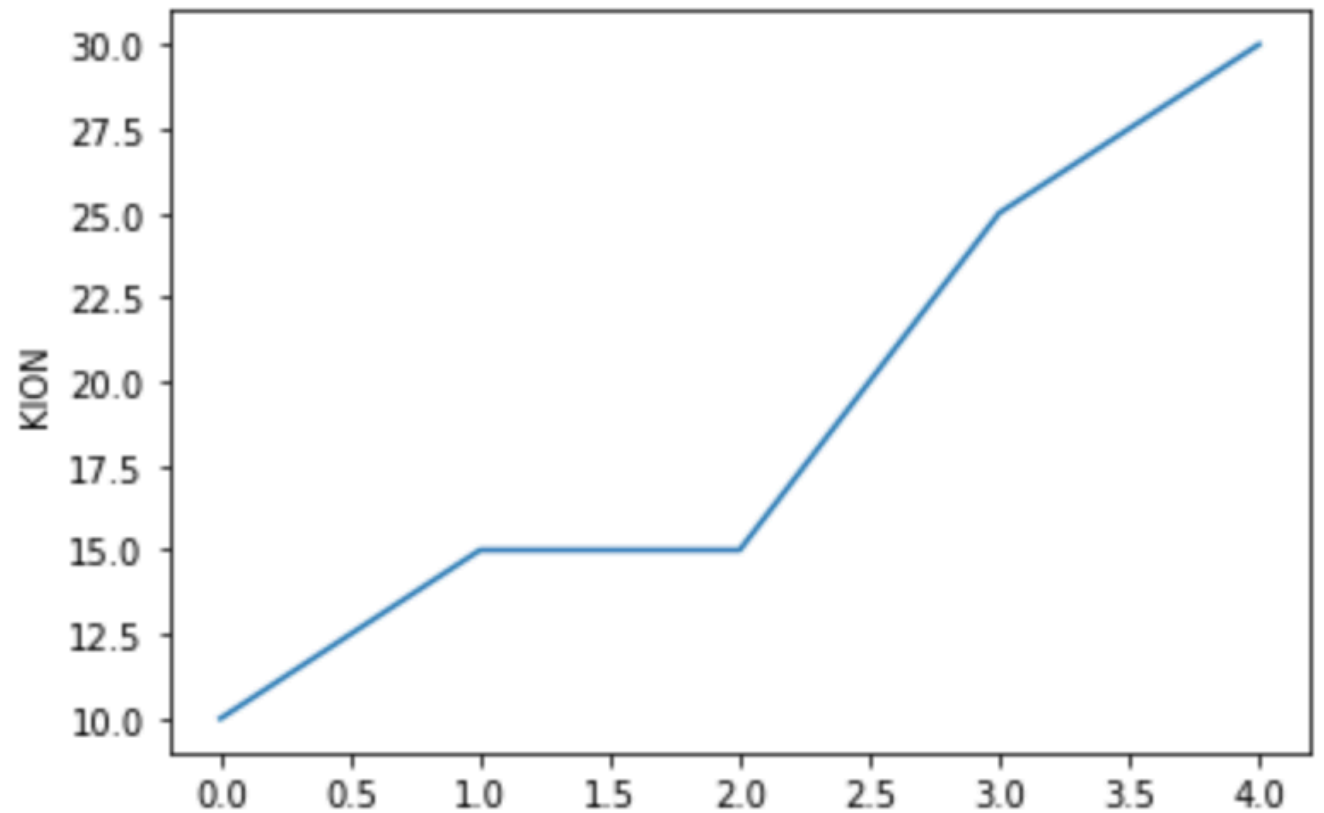
Matplotlibの基礎

■ 折れ線



折れ線

```
plt.figure()  
plt.plot(df['気温'])  
plt.ylabel('KION')  
plt.show()
```



Matplotlibの基礎

■ 円グラフ



円グラフ

```
plt.figure()
```

```
plt.pie(df['売上'], labels=df['気温'])
```

```
plt.show()
```

