

## ▼ Python / Colab による高校数学 + $\alpha$

宇都宮大学 吉田勝俊

### 学習目標

《Step.1》プログラミング言語 Python を実行できるクラウド環境 Colab について速習します。

- [Python](#) とは、人気沸騰中のプログラミング言語で、汎用性の高さがウリです。これまで複数のプログラミング言語を組み合わせないと書けなかった処理が、これ1つで書けてしまいます。AI分野の標準言語となりつつあります。
- Colab ([Google Colaboratory](#)) とは、Google が無償提供する Python 実行環境のことです。クラウド型なので、ウェブブラウザから利用できます。

**ウェブブラウザは Chrome を推奨します！** ※その他のブラウザでは不具合が起るかも知れません。

- Python は、単体で PC にインストールしても使えますが（かつてはそれが主流だった）、ここでは場所を選ばない方法として、Python をウェブブラウザから Colab 経由で実行する方法を学びます。

《Step.2》高校数学 +  $\alpha$  を、Python でプログラミングしてみます。

《Step.3》Python の文法（制御文とユーザ関数）を紹介します。

---

## ▼ 《Step.1》Colab 入門

※参考まで、大学の授業動画を張っておきますが、内容は若干異なります。

[Python / Colab 超入門デモ - YouTube](#)

### ▼ 1. 基本的事項

- 「Colab ノートブック」 今みているこの文書。
- 「セル」 Colab ノートブック内の入力可能な枠。
- 「テキストセル」 文章を記述するためのセル。今読んでいるここがテキストセルである。
- 「コードセル」 Pythonのコードが実行できるセル。次行はコードセルである。

**実習** 上のコードセルに `1+2` を入力し、セル左端の **実行ボタン**（再生ボタンみたいな）をクリックせよ。

## 2. コードセルの実行方法

- 該当セルをクリックして選択すると、セルの左側に **実行ボタン**（再生ボタンみたいな）が現れる。それをクリックする。
- または、選択中にキーボードから **《Ctrl-Enter》** する（**Ctrlキーを押しながらEnterキーを押す**）。
  - ブラウザによっては **《Shift-Enter》** も受け付けるみたい
- この操作により、入力内容が Python（クラウド上）に送信され、Python からの返信がブラウザに表示される。
  - クラウド上の Python と、LINE するイメージ

## 3. コードセルの追加方法

- 既存のコードセルを増やす：
  - 既存のコードセルの上下境界に、マウスカーソルを合わせる。「+ コード」をクリックする。
- コードセルをゼロから作る：
  - 左上メニューの「挿入」→「コード」で追加できる。

## ▼ 4. コメントアウト


- コードセル内に **#** 文字を置くと、そこから行末までを python は無視します。コードセル内のメモ書きに使います。

**実習** 上のコードセルに `1+2#+3+4` を入力し、実行せよ。

**実習** **#** を削除した `1+2+3+4` を実行せよ。

## ▼ 警告！

### 1. 「隠れセル」問題

- **〔 ? 個のセルが非表示〕** という表示があったら、その表示をクリックしてください。
- 通信量削減のため Colab が隠してしまったセルが表示されます。

- ・ 隠れセルを全て、一括で表示するには、Colab メニュー「表示」→「セクションを展開」します。

## 2. 「入力した内容が消えてしまう」問題

実習目的でファイルを保存しない場合は、無視して大丈夫です。

- ・ このファイルは、閉じると、入力した内容が消えてしまう、一時ファイルです。
- ・ 自分が入力した内容を保存したいときは、Colab メニュー「ファイル」→「ドライブにコピーを保存」で各自保存してください。
- ・ コピーしたファイルは、各自のGoogleドライブの「Colab Notebooks」というフォルダに保存されます。



編集するにはダブルクリックするか Enter キーを押してください

### ▼ 《Step.2》 高校数学 + α

以下のコードセルを、順番に実行してみましょう。（実行ボタンをクリックするか、《Ctrl-Enter》）

#### ▼ ○四則演算・べき乗

```
1+2*3/4 # 1+2×3÷4
```

2\*\*3 # 2の3乗

## ▼ ○ベクトル・行列

ベクトル演算するには、それ用のライブラリ（拡張機能）をインポートする（読み込む）必要があります。

```
import numpy as np # ベクトル演算ライブラリ numpy を、短縮名 np でインポートする。
```

## ▼ 〔ベクトルの作成〕

```
v = np.array([1, -2, 3]) # ベクトル v の作成  
print(v) #成分の確認
```

## ▼ 〔ベクトルの成分〕 ※ Python の添字は 0 から始まる（C言語などと同じ）

```
print( v[0] ) # Python の第0成分 = 数学の第1成分  
print( v[1] ) # Python の第1成分 = 数学の第2成分  
print( v[2] ) # Python の第2成分 = 数学の第3成分
```

## ▼ 【ベクトルの内積】

```
w = np.array([1, -2, 5]) # ベクトル w の作成  
np.dot(v, w)             # 先ほどの v と w の内積
```

## ▼ ○関数とグラフ

## ▼ 〔x 軸の作成〕

- 等差数列として作成します。（等差でなくても構いません）
- ベクトルと同じデータ構造（numpy.array）でプログラミングします。

```
x0 = -2 # 初項. 軸の左端  
x1 = 2  # 末項. 軸の右端  
n  = 21 # 項数
```

```
x = np.linspace(x0, x1, n) #x = -2 ~ 2 を20等分 → 項数は21
```

```
print(x)
```

## ▼ 《関数の例》

```
y = x*(x-1)*(x+1) # 3次関数
```

```
print(y)
```

## ▼ 《グラフの作成》

グラフを作成するには、それ用のライブラリ（拡張機能）をインポートする（読み込む）必要があります。

```
import matplotlib.pyplot as plt # グラフ作成ライブラリを、短縮名 plt でインポートする。
```

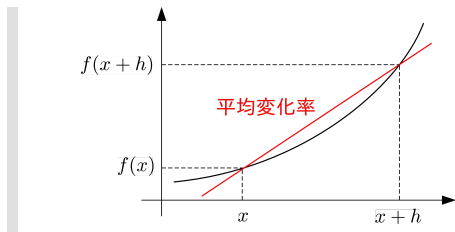
次のようにするとグラフが作成されます。

```
plt.plot(x, y) # グラフを描く  
plt.xlabel('x') # 横軸ラベル  
plt.ylabel('y') # 縦軸ラベル  
plt.grid() # グリッド線を付ける
```

## ▼ ○関数の微分

- 関数  $y = f(x)$  のグラフの傾きを、**導関数** と呼び、次のような数式で表します。

$$\frac{dy}{dx} \quad \text{または} \quad \frac{df(x)}{dx} \quad \text{または} \quad f'(x) \quad \stackrel{\text{定義}}{=} \quad \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$



- $f(x)$  の導関数を求めることを「 $f(x)$  を  $x$  で **微分** する」といいます。
- コンピュータは「 $h \rightarrow 0$ 」を扱えないので、次式で近似します。

$$f'(x) \stackrel{\text{近似}}{=} \frac{f(x+h) - f(x)}{h} \quad (h \text{ はなるべく小さい数})$$

[【参考情報】 高等学校数学II/微分・積分の考え - Wikibooks](#)

〔数学の定義通りの低精度計算〕※自家製

```
n = len(x) # x軸の項数  
my_x = [] # 空のリスト  
my_dydx = [] # 空のリスト
```

```
for i in range(n-1): #引き算するため、項数が1つ減ります
```

```

my_x.append(x[i])          # x軸の各区間の左端の値をリストに追加

h = x[i+1] - x[i]          # x軸の各区間の幅（公差）
f_dash = (y[i+1] - y[i])/h # f'(x) の近似式

my_dydx.append(f_dash)     # 計算値をリストに追加

print(my_dydx)

```

グラフを確認すると、左右対称なはずが、左にズレてる ∵ 各区間の左端を my\_x にしたので

```

plt.plot(my_x, my_dydx) # グラフを描く
plt.xlabel('x')         # 横軸ラベル
plt.ylabel('dy/dx')     # 縦軸ラベル
plt.grid()              # グリッド線を付ける

```

〔Numpyによる高精度計算〕 ※2次精度の差分公式という方法で計算しています

```

dydx = np.gradient(y, x, edge_order=2) #高精度バージョン

plt.plot(x, dydx)      # グラフを描く
plt.xlabel('x')        # 横軸ラベル
plt.ylabel('dy/dx')    # 縦軸ラベル
plt.grid()             # グリッド線を付ける

```

〔理論式  $f'(x) = 3x^2 - 1$  との比較〕 ※理論式は数Ⅱや数Ⅲの教科書を見てください

```

dydx_math = 3*x*x - 1

plt.plot(my_x, my_dydx, 'xk', label='Mine') # 自家製の微分
plt.plot(x, dydx, 'or', label='Numpy')     # Numpy による微分
plt.plot(x, dydx_math, label='Mathematics') # 正解：手計算による微分
plt.xlabel('x')                             # 横軸ラベル
plt.ylabel('dy/dx')                         # 縦軸ラベル
plt.grid()                                  # グリッド線を付ける
plt.legend()                                # 凡例を付ける

```

- 自家製（x）は、正解（線）から結構ズレました。さらに右端が1個足りません。
- Numpy（●）は、正解（線）と、だいたい合いました。

## ▼ ○行列

### ▼ 〔行列の作成〕

```
A = np.array([
    [-1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]) # 行列 A の作成
print(A) # 成分の確認
```

#### ▼ 〔行列の成分〕

```
print( A[2, 1] ) # Pythonの2行1列成分 = 数学の3行2列成分
```

#### ▼ 〔行列の行ベクトル〕 ":" で全成分を表す. この操作をPythonでは「スライス」という.

```
print( A[0, :] ) # Pythonの第0 行ベクトル = 数学の第1 行ベクトル
print( A[1, :] ) # Pythonの第1 行ベクトル = 数学の第2 行ベクトル
print( A[2, :] ) # Pythonの第2 行ベクトル = 数学の第3 行ベクトル
```

#### ▼ 〔行列の列ベクトル〕

```
print( A[:, 0] ) # Pythonの第0 列ベクトル = 数学の第1 列ベクトル
print( A[:, 1] ) # Pythonの第1 列ベクトル = 数学の第2 列ベクトル
print( A[:, 2] ) # Pythonの第2 列ベクトル = 数学の第3 列ベクトル
```

#### ▼ 〔行列の転置〕

```
Atrans = A.T # (行列).T で転置を表す
print(Atrans) # 成分の確認
```

#### ▼ 〔行列とベクトルの積〕 ※ベクトルの内積と同じ文法で書けます

```
np.dot(A, v) # 行列 A とベクトル v の積
```

#### ▼ 〔行列と行列の積〕

```
np.dot(A, Atrans) # 行列 A と行列 Atrans の積
```

#### ▼ 〔逆行列〕

逆行列を求めるには、それ用のライブラリ（拡張機能）をインポートする（読み込む）必要があります。

```
import numpy.linalg as la # numpy に含まれる線形代数ライブラリを、短縮名 la でインポートする。
```

逆行列を求め、元の行列に掛けると、単位行列になります。

```
Ainv = la.inv(A)          # A の逆行列
print('Ainv = ', Ainv)

AinvA = np.dot(Ainv, A) # 行列と逆行列の積
print('AinvA = ', AinvA)
```

### ※ 計算機誤差について

- AinvA は数学的には単位行列になるはずだが、そうになってない。
- 8.88178420e-16 等は  $8.88178420 \times 10^{-16}$  のコンピュータ表記である。
  - AinvA の非対角要素は、0 ではないものの、0 に極めて近い。
- このような真値からのズレは、電算機の丸め誤差から生じている。このような誤差を、**計算機誤差** という。
  - しばしば、次のように言い表す。「計算機誤差を考慮すると、AinvA は単位行列とみなせる」

---

## ▼ 《Step.3》 Python の制御文とユーザ関数

### ▼ ○繰り返し処理（ループ）

次のように for を使って記述します。

```
for 変数名 in range(反復回数): ←コロンを忘れずに！
    インデントブロック ※
    インデントブロック ※複数行に渡ってよい
    インデントブロック ※
```

- 「半角スペース」 日本語入力OFFのときに入力される英語の空白文字。他方、かな漢字のスペースを全角スペースという。
- 「インデント」 半角スペースによる字下げ。全角スペースでインデントすると文法エラーになる。
- 「インデントブロック」 インデントの文字数を揃えた複数行の処理（1行でもよい）。  
**途中の空行は無視してくれる。**
- Colab のコードセルは、インデント入力を補完する機能を備えている。



```
A = np.array([
    [-1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]) # 行列 A の作成

for i in range(3): # i は 0, 1, 2 と動く
    for j in range(3): # その中で, j は 0, 1, 2 と動く
        print((i, j), A[i, j])
```

## ▼ ○条件分岐

「もし〔条件〕ならば〔処理〕を行う」という文法を **条件分岐** といいます。

次のように if を使って記述します。

```
if 条件: ←コロンを忘れずに！
    インデントブロック（条件を満たしたときの処理）※複数行に渡ってよい
```

「もし〔条件〕ならば〔処理1〕を行い、さもなければ〔処理2〕を行う」という書き方もできます。

```
if 条件: ←コロンを忘れずに！
    インデントブロック（条件を満たしたときの処理1）
else: ←コロンを忘れずに！
    インデントブロック（条件を満たさないときの処理2）
```

条件を増やすこともできます。

```
if 条件1: ←コロンを忘れずに！
    インデントブロック（条件1を満たしたときの処理）
elif 条件2: ←コロンを忘れずに！
    インデントブロック（条件1を満たさず、条件2を満たしたときの処理）
else: ←コロンを忘れずに！
    インデントブロック（それ以外の処理）
```

※ elif は、いくつあっても（無くても）大丈夫です。

```
A = np.array([
    [-1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]) # 行列 A の作成

for i in range(3):
    for j in range(3):
        if i == j:
```

```

        print( '対角要素', A[i, j] )
    elif i<j:
        print( '上三角', A[i, j] )
    else:
        print( 'それ以外（下三角）', A[i, j] )

```

## ▼ ユーザ関数の定義

自分で新しい関数を作ることができます。def を使って次のように記述します。

```

def 関数名(引数1, 引数2, ...): ←コロンを忘れずに！
    '''
    関数の説明文をここに書く（人間用で実行はされない）
    '''
    インデントブロック ※複数行に渡ってよい
    return 戻り値 ←必要なければ省略可能

```

## ▼ 〔1 変数関数の例〕

```

def sanji(x):
    '''
    先ほどの3次関数
    '''
    y = x*(x-1)*(x+1)
    # 空行は無視される
    return y

```

sanji(x)

y = x\*(x-1)\*(x+1)

y

同じ答えが出せてます。

## ▼ 〔戻り値を複数返す関数〕

```

def sanji_and_diff(x):
    '''
    先ほどの3次関数の値と、その導関数を返す
    '''
    y = sanji(x)
    dydx = 3*x**2 - 1

```

```
return [y, dydx] #リスト[○, ○, …] で返せばよい
```

- ▼ 複数の戻り値は、カンマで並べた変数に一括代入できます.

```
fx, dfxdx = sanji_and_diff(x)
```

```
print( fx )  
print( x*(x-1)*(x+1) )  
print( '-----' )  
print( dfxdx )  
print( 3*(x**2) - 1)
```