

# 演習I・第12回

システム情報学部 中島涼輔  
@K502  
2025/7/4

パスワード管理アプリを使うのはおすすめです

第2回授業で配布された紙のパスワードが必要です

← 重要

if (第2回授業で問題なく接続でき、そのパソコンで今日（第13回）の授業を受けている):

elif

590

作業

elif

2cc

ださ

来週（第13回）の授業の後半では、第2回授業で接続テストを行った教育用計算機システムを使います。

不安がある人、接続テストでうまくいかなかった人は、（一応、授業中に少し時間を確保しますが）あらかじめ Beef+ のマニュアルに目を通したり、再インストールを行なっておいてもらえると助かります。

elif (第2回授業を欠席した) or (第2回授業とは違うパソコンで今日（第13回）の授業を受けている):

マニュアルの「2.1 SSL-VPN接続(初回)」と「2.2 計算機サーバーログイン手順(初回)」の作業を行なって、JupyterHub に接続してください。

もし接続ができない場合は、webブラウザ上でUNIXコマンドの練習ができるエミュレータ（別の環境で模倣するもの）を使って、今日（7/11）の授業を受けてください。 (→ <https://webvm.io> )

# スケジュール

---

第1回:ガイダンス(4/11)	第8回:文字列(5/30)
第2回:システム設定(4/18)	第9回:リスト(6/13)
第3回:Python概要(4/25)	第10回:関数(6/20)
第4回:変数・型(5/2)	第11回:高階関数(6/27)
第5回:条件分岐(5/9)	<b>第12回:クラス(7/4)</b>
第6回:繰り返し(5/16)	第13回:継承・UNIX(7/11)
第7回:オブジェクト(5/23)	第14回:ライブラリ(7/18)
	第15回:まとめ(7/25)

- シラバスから若干変更

# 今日の内容

---

- 新しいクラスを作る(教科書 6-1)
- メソッドの定義(教科書 6-2)



# クラスとは

- クラスとは、インスタンス（オブジェクト）が所属している種類のこと（「型」と同じ）

```
▶ def func():
    pass

    print(type(10))
    print(type(0.5))
    print(type('神大うりぼー'))
    print(type(True))
    print(type([1, 2]))
    print(type((3, 4, 5)))
    print(type(print))
    print(type(func))
    print(type(range(100)))

→ <class 'int'>
   <class 'float'>
   <class 'str'>
   <class 'bool'>
   <class 'list'>
   <class 'tuple'>
   <class 'builtin_function_or_method'>
   <class 'function'>
   <class 'range'>
```

これまで Python が用意してくれていたクラスと、その機能（メソッド）を利用していった

```
▶ # str クラスのメソッド
  print('hello'.upper())

  # list クラスのメソッド
  list1 = [1, 2]
  list1.append(3)
  print(list1)

→ HELLO
   [1, 2, 3]
```

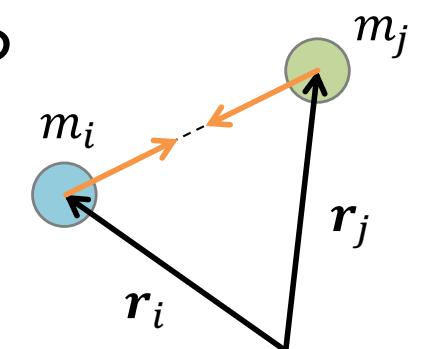
今回は、自分で新しいクラスを定義する

# なぜ新しいクラスをつくりたいのか？

例えば、 $n$  個の質点が互いに万有引力を及ぼしあって運動する様子をシミュレーションしたいとする

微分方程式をプログラムで解く方法は後期?とかに習うと思います

$$m_i \frac{d^2 \mathbf{r}_i}{dt^2} = -G \sum_{j(\neq i)}^n \frac{m_i m_j}{|\mathbf{r}_i - \mathbf{r}_j|^2} \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|}$$



```
G = 6.6743 * (10**(-11))

def gravitation_x(mass_i, mass_j, pos_i, pos_j):
    r = ((pos_i[0] - pos_j[0])**2 + (pos_i[1] - pos_j[1])**2 + (pos_i[2] - pos_j[2])**2)**(1/2)
    return -G * mass_i * mass_j * (pos_i[0] - pos_j[0]) / (r**3)

# 質点1, 質点2, 質点3, ... とリストに並べる
# 質量
mass = [10, 20, 30, 40, ...]
# 位置 (x,y,z)
position = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], ...]

# 質点3の位置のy成分
print(position[2][1])
# 質点1が質点4から及ぼされる万有引力のx成分
print(gravitation_x(mass[0], mass[3], position[0], position[3]))
```

一つの質点に関する情報や関数が別々のオブジェクトに格納

リストのインデックスが少し分かりづらい

# なぜ新しいクラスをつくりたいのか？

もし「質点（PointMass）クラス」を新しくつくれば…

```
G = 6.6743 * (10**(-11))

class PointMass:
    def __init__(self, mass, pos):
        self.mass = mass
        self.x = pos[0]
        self.y = pos[1]
        self.z = pos[2]

    def gravitation_x_from(self, other):
        r = ((self.x - other.x)**2 + (self.y - other.y)**2 + (self.z - other.z)**2) ** (1/2)
        return - G * self.mass * other.mass * (self.x - other.x) / (r**3)

pm = [
    PointMass(10, [1, 2, 3]),
    PointMass(20, [4, 5, 6]),
    PointMass(30, [7, 8, 9]),
    PointMass(40, [10, 11, 12]), ...
]

# 質点3の位置のy成分
print(pm[2].y)
# 質点1が質点4から及ぼされる万有引力のx成分
print(pm[0].gravitation_x_from(pm[3]))
```

クラスを定義  
(後で詳しく説明)

一つの質点に関する情報や関数を一つのオブジェクト(pm[i])にまとめられる  
(オブジェクト指向)

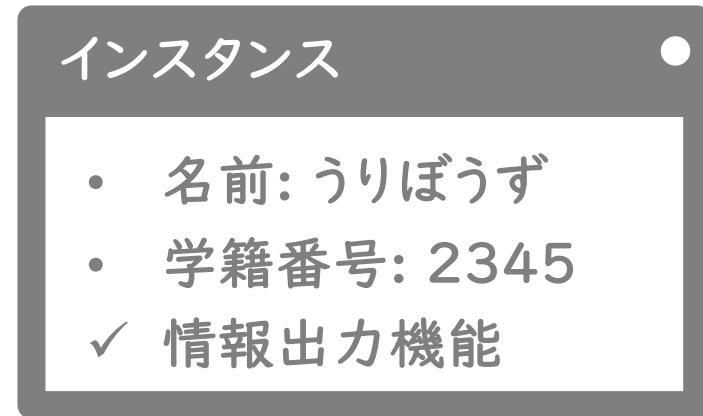
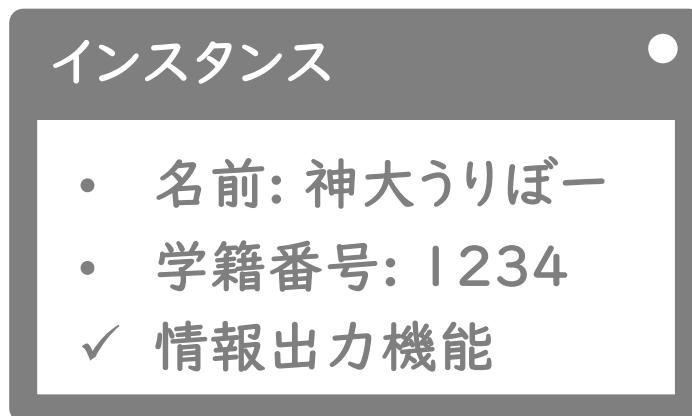
プログラムコードの可読性が上がる  
関数(メソッド)の引数が少なくなる

# クラスのつくり方(1) 構想を練る

- 具体例として、学生の情報を管理するシステムをつくる場合を考える  
テンプレートのようなもの



- StudentData クラスのインスタンスの例



# クラスのつくり方(2) クラスを定義

構文    `class クラス名:`  
インデント     初期化メソッドなどの定義

```
▶ class StudentData:  
    pass # 何も処理をしないという文（とりあえず）  
  
# StudentData クラスに所属するインスタンスを1つ生成してst1に代入  
st1 = StudentData()  
print(type(st1))  
  
⇨ <class '__main__.StudentData'>
```

- 今の状況



- クラス名はキャメルケースにするのが慣例  
↔ `student_data`

# クラスのつくり方(3) 初期化メソッド

初期化メソッド(コンストラクタ):

インスタンス生成のときに実行されるメソッド(関数みたいなもの)

構文    `def __init__(self):`  
      インデント    処理内容

Python では、ダブルアンダースコア「\_\_」(ダンダー)で挟まれたものは特別な意味をもつ

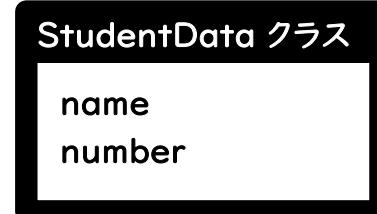
▶ `class StudentData:`  
    `| def __init__(self):` ←  
    `|     self.name = '未設定'`  
    `|     self.number = 9999 |`  
    `-----|`  
`初期化  
メソッド`    `st1 = StudentData() # インスタンス生成`

`print(st1.name)`    インスタンス変数を参照  
`print(st1.number)`

→ 未設定  
9999

初期化メソッド呼び出し  
`self` はそのインスタンス自身を指す  
(`self = st1`)

- 今の状況



# クラスのつくり方(4) インスタンス変数を変更

- インスタンス変数は、普通の変数のように変更できる

```
▶ class StudentData:  
    def __init__(self):  
        self.name = '未設定'  
        self.number = 9999  
  
    st1 = StudentData()  
  
    print(st1.name)  
    print(st1.number)  
  
    st1.name = '神大うりぼー'  
    st1.number = 1234  
  
    print(st1.name)  
    print(st1.number)  
  
→ 未設定  
    9999  
    神大うりぼー  
    1234
```

- 今の状況



- 練習

➤ うりぼううずのインスタンスをつくってください



# 答え



```
class StudentData:  
    def __init__(self):  
        self.name = '未設定'  
        self.number = 9999
```

```
st1 = StudentData()  
st2 = StudentData()
```

```
st1.name = '神大うりぼー'  
st1.number = 1234
```

```
st2.name = 'うりぼうず'  
st2.number = 2345
```

# クラスのつくり方(5) 引数ありの初期化メソッド

- 初期化メソッドは、関数のように引数を追加できる
  - デフォルト引数、キーワード引数などの使用も可



```
class StudentData:  
    def __init__(self, name='未設定', number=9999):  
        self.name = name  
        self.number = number  
  
st1 = StudentData(name='神大うりぼー', number=1234)  
st2 = StudentData()  
  
print(st1.name)  
print(st1.number)  
print(st2.name)  
print(st2.number)
```

→ 神大うりぼー  
1234  
未設定  
9999

self は書かないので注意

- 練習

- うりぼうずのインスタンスを初期化メソッドでつくってください

インスタンス  
st2  
name: うりぼうず  
number: 2345

# 答えの例(キーワード引数を使ってもOK)

```
class StudentData:  
    def __init__(self, name='未設定', number=9999):  
        self.name = name  
        self.number = number  
  
st1 = StudentData(name='神大うりぼー', number=1234)  
st2 = StudentData('うりぼうず', 2345)
```

self は書かないので注意

# 今までたくさん使ってきた int 関数などは…

## Python の組み込み関数の公式ドキュメント

<https://docs.python.org/ja/3.13/library/functions.html>

クラス

```
class int(number=0, /)
class int(string, /, base=10)
```

数または文字列から生成された整数オブジェクトを返します。引数が指定されない場合は 0 を返します。

```
class float(number=0.0, /)
class float(string, /)
```

数または文字列から生成された浮動小数点数を返します。

a = int(3.4)

同じ

st1 = StudentData('神大うりぼー', 1234)

浮動小数点数 (float 型) を整数 (int 型) に変換する 関数

だと説明してきたが、厳密には

int クラスの初期化メソッドの引数に「3.4」を入れて  
「3」という int クラスのインスタンスを生成している（らしい）

# クラスのつくり方(6) メソッド

メソッド(インスタンスマソッド)：

インスタンスが実行できる関数みたいなもの

第一引数は、そのインスタンス自身を表す **self**

```
▶ class StudentData:  
    def __init__(self, name='未設定', number=9999):  
        self.name = name  
        self.number = number  
  
    -----  
    | def print_data(self):           | メソッド  
    |     print(f'name: {self.name}') |  
    |     print(f'number: {self.number}') |  
    |-----|  
  
st1 = StudentData(name='神大うりぼー', number=1234)  
  
st1.print_data()  
  
→ name: 神大うりぼー1  
      number: 1234
```

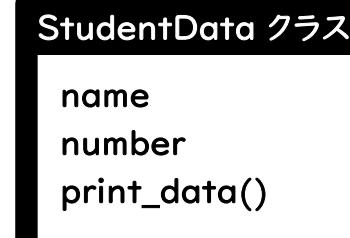
構文

def メソッド名(**self**, その他の引数) :

  処理内容

インデント

- 今の状況



# 練習

```
class StudentData:  
    def __init__(self, name='未設定', number=9999):  
        self.name = name  
        self.number = number  
  
    def print_data(self):  
        print(f'name: {self.name}')  
        print(f'number: {self.number}')  
  
st1 = StudentData(name='神大うりぼー', number=1234)  
  
st1.print_data()
```

→ name: 神大うりぼー<sup>一</sup>  
number: 1234

左のように、引数で与えたマークで情報を装飾してくれる新しいメソッドを追加してください（装飾の仕方は自由）

# 答えの例(他でもOK)

```
▶ class StudentData:  
    def __init__(self, name='未設定', number=9999):  
        self.name = name  
        self.number = number  
  
    def print_data(self):  
        print(f'name: {self.name}')  
        print(f'number: {self.number}')  
  
    def deco_print_data(self, mark):  
        print(mark*26)  
        print(mark + ' '*24 + mark)  
        print(mark + ' '*2 + f'{self.number:>7} {self.name:>8}' + ' '*2 + mark)  
        print(mark + ' '*24 + mark)  
        print(mark*26)  
  
st1 = StudentData(name='神大うりぼー', number=1234)  
st1.deco_print_data('♠')  
→ ♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠  
♠ ♠  
♠ 0001234 神大うりぼー ♠  
♠ ♠  
♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠♠
```

右寄せ, 7桁, 0埋め

# 練習

2×2 の正方形行列のクラス (Matrix2by2 クラス) をつくってください。  
モジュール (import 文) の使用は禁止

```
mat1 = Matrix2by2([[1, 2], [3, 4]])
```

このインスタンスを生成

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

1. 行列の4つの成分を4つのインスタンス変数にしてください
2. 行列が以下のように出力される print\_matrix メソッドをつくれてください

```
/1 2\  
\3 4/
```

← 括弧はなくてもOK

3. 行列のトレースを計算して、その結果を return 文で返す trace メソッドをつくれてください
4. (mat1 の) メソッドの引数に別の Matrix2by2 クラスのインスタンス (mat2) を入れると、行列の和 (mat1+mat2) を計算でき、その結果を return 文で返す plus メソッドをつくれてください

# 答えの例(他でもOK)

```

▶ class Matrix2by2:
    def __init__(self, matrix):
        self.a = matrix[0][0]
        self.b = matrix[0][1]
        self.c = matrix[1][0]
        self.d = matrix[1][1]

    def print_matrix(self):
        len_col1 = len(str(self.a))
        len_col2 = len(str(self.b))
        len_c = len(str(self.c))
        len_d = len(str(self.d))
        if len_col1 < len_c:
            len_col1 = len_c
        if len_col2 < len_d:
            len_col2 = len_d
        print(f'{self.a:>{len_col1}} {self.b:>{len_col2}}\\')
        print(f'\\{self.c:>{len_col1}} {self.d:>{len_col2}}\\')

    def trace(self):
        return self.a + self.d

    def plus(self, other):
        add_a = self.a + other.a
        add_b = self.b + other.b
        add_c = self.c + other.c
        add_d = self.d + other.d
        return Matrix2by2([[add_a, add_b], [add_c, add_d]])

```

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

```

mat1 = Matrix2by2([[1, 2], [3, 4]])
mat2 = Matrix2by2([[5, 6], [7, 8]])
mat1.print_matrix()
print(mat1.trace())
mat3 = mat1.plus(mat2)
mat3.print_matrix()

```

→ /1 2\  
  \3 4/  
  5  
  / 6 8\  
  \10 12/

```

print(f'{self.a} {self.b}')
print(f'{self.c} {self.d}')

```

でもOK

# クラス変数

```
▶ class StudentData:  
    num_student = 0  
  
    def __init__(self, name='未設定'):  
        self.name = name  
        self.number = StudentData.num_student + 1000  
        StudentData.num_student += 1  
  
    def print_data(self):  
        print(f'name: {self.name}')  
        print(f'number: {self.number}')  
  
    print(f'総学生数: {StudentData.num_student}')  
st1 = StudentData('神大うりぼー')  
print(f'総学生数: {StudentData.num_student}')  
st2 = StudentData('うりぼうず')  
print(f'総学生数: {StudentData.num_student}')  
  
st1.print_data()  
st2.print_data()  
  
→ 総学生数: 0  
総学生数: 1  
総学生数: 2  
name: 神大うりぼー1  
number: 1000  
name: うりぼうず  
number: 1001
```

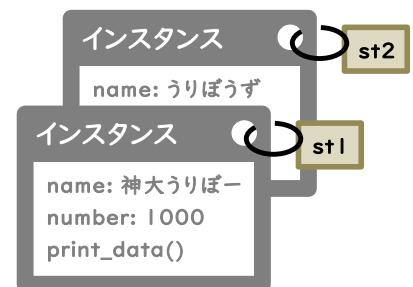
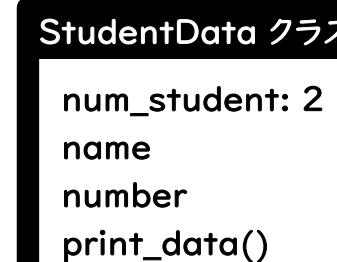
クラス変数

クラス変数を参照

- 個々のインスタンスではなく、クラス全体の情報（**クラス変数**）を格納できる
  - インスタンスを生成していないなくても参照可

インスタンスを生成した回数をカウントして、その数を学籍番号に使用

- 今の状況



# クラスメソッド



```
class StudentData:  
    num_student = 0  
  
    def __init__(self, name='未設定'):  
        self.name = name  
        self.number = StudentData.num_student + 1000  
        StudentData.num_student += 1  
  
    def print_data(self):  
        print(f'name: {self.name}')  
        print(f'number: {self.number}')  
  
    @classmethod  
    def print_num_student(cls):  
        print(f'総学生数: {cls.num_student}')
```

```
StudentData.print_num_student()  
st1 = StudentData('神大うりぼー')  
StudentData.print_num_student()  
st2 = StudentData('うりぼうず')  
StudentData.print_num_student()
```



総学生数: 0  
総学生数: 1  
総学生数: 2

クラスメソッド

クラスメソッド呼び出し

- クラス変数と同様に **クラスメソッド**もある
  - インスタンスを生成していないくとも実行可
  - **@classmethod** というデコレーターをつける
  - 第一引数は、そのクラスを表す **cls**

# 出席テストと課題

---

- BEEF+に載せています
- 出席テスト
  - Google Colaboratory で試しながら解答しても良いです
  - 全問正解するまで受験してください
- 課題
  - 提出期限: 7/6(日)の 23:59 まで
  - 実行に必要な全てのプログラムコードを提出してください  
(インデント、タイプミスに注意。提出画面で「</>」というボタンを押すと表示される枠の中にコピペをすると、インデントが崩れない気がします。)
  - 出力結果はペーストしないでください
  - モジュール(import 文)は使わないでください

# 課題

以下は、 $3 \times 3$  の正方行列のクラス（Matrix3by3 クラス）を定義しようとしている未完成のプログラムコードです。

```
Matrix3by3
    __init__
        self.entries =
            print_matrix
            a = self.entries
            for i in range(3):
                print(f' {a[i][0]} {a[i][1]} {a[i][2]}')

    determinant

    transpose

    multiply

#####
# ここまで上書き換える #####
if __name__ == "__main__":
    mat_a = Matrix3by3([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    mat_b = Matrix3by3([[9, 8, 7], [6, 5, 4], [3, 2, 1]])
    print('A =')
    mat_a.print_matrix()
    print('B =')
    mat_b.print_matrix()
    print('*10')
    print(f'det(A) = {mat_a.determinant()}')
    print('A^T =')
    mat_a.transpose().print_matrix()
    print('AB =')
    mat_a.multiply(mat_b).print_matrix()
```

# 課題

プログラムコード中の

##### ここより上を書き換える #####

より上の部分を書き換えて、プログラムコードを完成させてください。

ただし、Matrix3by3 クラスの定義において、以下のメソッドを追加してください。

- =====
- 1. 行列式を計算して、その結果を return 文で返す determinant メソッドをつくってください
- 2. 行列の転置を計算して、その結果を return 文で返す transpose メソッドをつくってください
- 3. (行列Aの) メソッドの引数に別の Matrix3by3 クラスのインスタンス (行列B) を入れると、行列の積 (AB) を計算でき、その結果を return 文で返す multiply メソッドをつくってください
- =====

期待される出力結果は、以下の通りです。

```
A =  
1 2 3  
4 5 6  
7 8 9  
B =  
9 8 7  
6 5 4  
3 2 1  
-----  
det(A) = 0  
A^T =  
1 4 7  
2 5 8  
3 6 9  
AB =  
30 24 18  
84 69 54  
138 114 90
```

$$\det(A) = \sum_{\sigma \in S_3} \operatorname{sgn}(\sigma) a_{1\sigma(1)} a_{2\sigma(2)} a_{3\sigma(3)}$$

ここで、 $S_3$  は3次対称群

$$(A^T)_{ij} = A_{ji}$$

$$C_{ij} = \sum_{k=1}^3 A_{ik} B_{kj}$$



# 課題の答えの例

```
class Matrix3by3:
    def __init__(self, matrix):
        self.entries = matrix

    def print_matrix(self):
        a = self.entries
        for i in range(3):
            print(f' {a[i][0]} {a[i][1]} {a[i][2]}')

    def determinant(self):
        a = self.entries
        det = 0
        for i in range(3):
            det += a[0][i] * a[1][(i+1)%3] * a[2][(i+2)%3]
            det -= a[0][i] * a[1][(i+2)%3] * a[2][(i+1)%3]
        return det

    def transpose(self):
        a = self.entries
        a_t = [[None]*3]*3
        for i in range(3):
            for j in range(3):
                a_t[i][j] = a[j][i]
        return Matrix3by3(a_t)

    def multiply(self, other):
        a = self.entries
        b = other.entries
        c = [[0]*3]*3
        for i in range(3):
            for j in range(3):
                for k in range(3):
                    c[i][j] += a[i][k] * b[k][j]
        return Matrix3by3(c)
```

---

問1

---

次のようなクラスが定義されています。

```
class FoodMenu:  
    def __init__(self, name, price):  
        self.name = name  
        self.price = price
```

この FoodMenu クラスのインスタンスを2つ、以下のように生成しました。

```
food1 = FoodMenu('うどん', 500)  
food2 = FoodMenu('サラダ', 100)
```

その後、以下のようなプログラムコードを実行するとエラーが起きます。

```
food1.allergy = '小麦'  
print(food1.allergy)  
print(food2.allergy)
```

このエラーが起きる理由について、正しく説明しているものはどれでしょうか？

---

問題文にはエラーが起きると書いてあるが、実際にはエラーは起きない。

---

インスタンス変数を新たに追加することは可能であるが、`allergy` というインスタンス変数をもっているのは `food1` のみなので

```
print(food2.allergy)
```

正解

の行でエラーになる。

---

FoodMenu クラスには、`allergy` というインスタンス変数は存在しないので、

```
food1.allergy = '小麦'
```

の行でエラーになる。

## 問2

次のような RGB カラーモデル<sup>(※1)</sup> を表すクラスが定義されています。

```
class Color:  
    def __init__(self, r, g, b):  
        self.__r = r  
        self.__g = g  
        self.__b = b  
  
    def get_rgb(self):  
        return self.__r, self.__g, self.__b
```

(※1) 赤 (R) 、緑 (G) 、青 (B) の三原色の混合で色を表す方法。

このクラスの定義に続いて、以下のプログラムコード

```
red = Color(r=255, g=0, b=0)  
red.__r = 0  
print(red.get_rgb())
```

を実行すると、どのように表示されるでしょうか？

(0, 0, 0)

[255, 0, 0]

255, 0, 0

[0, 0, 0]

(255, 0, 0)

正解

0, 0, 0

エラーになる

### ◆◆◆ ヒント ◆◆◆

アクセス制御に関する問題です。

インスタンス変数の名前の最初にダブルアンダースコア「\_\_」をつけると、そのインスタンス変数をクラス定義の外で参照することができなくなります（教科書 p.185）。このようにすることを「隠蔽」と呼び、クラス定義の外でインスタンスの情報を誤って書き換えてしまうのを防ぐことができます。ただし、実際には完全に隠蔽されているわけではなく、「\_\_r」という変数が、クラス定義の外では「\_Color\_\_r」という名前に置き換わっているだけなので、「red.\_\_r = 0」ではなく「red.\_Color\_\_r = 0」とすると、値が変わってしまいます。

### 問3

次のようなクラスと関数が定義されています。

```
class Duck:  
    def cry(self):  
        print('quack!')  
  
class Dog:  
    def cry(self):  
        print("bow-wow!")  
  
def animal_sounds(obj):  
    obj.cry()
```

bow-wow!  
quack!

これに続いて、以下のプログラムコード

```
animals = [Duck(), Dog()]  
for animal in animals:  
    animal_sounds(animal)
```

bow-wow!  
bow-wow!

を実行すると、どのように表示されるでしょうか？

quack!  
bow-wow!

正解

quack!  
quack!

エラーになる

◆◆◆ ヒント（というか、問題背景） ◆◆◆

Python は、「`a = 1`」とした後に「`a = 'Hello'`」とできるように、一つの変数に対して代入するリテラルのクラス（型）を固定する必要のない「動的型付け言語」です。そのため、`animal_sounds` 関数の仮引数 `obj` には、`Duck` クラスのインスタンスを代入することができれば、`Dog` クラスのインスタンスを代入することもできます。`animal_sounds` 関数内で `obj.cry()` が呼び出されるとき、`obj` に代入されているものが `Duck` クラスのインスタンスなのか `Dog` クラスのインスタンスなのに合わせて、`Duck` クラスの `cry` メソッドまたは `Dog` クラスの `cry` メソッドのどちらかが自動的に選択されます。このような動作は「ダック・タイピング」や「ポリモーフィズム」と呼ばれ、オブジェクト指向プログラミングの重要な要素の1つです。実は、このような動作はこれまでにもたくさん経験しています。「`1 + 1`」や「`'Hello' + 'Python'`」の演算子「`+`」は前後のリテラルが `int` 型なのか `str` 型なのかで挙動が異なっていました。