

演習I・第10回

システム情報学部 中島涼輔

@K502

2025/6/20

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)	第12回:クラス(7/4)
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

- ・ 関数（教科書 5-1）
- ・ 関数の引数（教科書 5-2）



関数とは

- 複数の命令文を1つのかたまりにしたもの
- プログラムコードが読みやすくなる
- プログラムコードの修正がしやすくなる
 - 長いプログラムコードになると特に重要

プログラムコード



- 黄色の部分は同じ命令文のかたまり
- 黄色の部分を変更する場合は、3か所修正しないといけない

プログラムコード



関数

- プログラムコードが短くなって読みやすくなる
- 1か所だけ修正すれば良い

関数とは

- 複数の命令文を1つのかたまりにしたもの
- プログラムコードが読みやすくなる
- プログラムコードの修正がしやすくなる
 - 長いプログラムコードになると特に重要
- 組み込み関数**: はじめから準備してくれている関数
例: `print`関数、`id`関数、`len`関数
- ユーザー定義関数**: 自分で定義する関数

構文: 関数の定義

```
define
  def 関数名():
    インデント      処理内容
```

具体例(ユーザー定義関数 使用前)

print('エラー') の下に
print('指示にしたがってください') を追加したい



```
num1 = input('10以下の整数を入力してください：')
if int(num1) > 10:
    print('エラー')
num2 = input('5以上の整数を入力してください：')
if int(num2) < 5:
    print('エラー')
```

具体例(ユーザー定義関数 使用前)

print('エラー') の下に
print('指示にしたがってください') を追加したい



```
num1 = input('10以下の整数を入力してください：')
if int(num1) > 10:
    print('エラー')
    print('指示にしたがってください')
num2 = input('5以上の整数を入力してください：')
if int(num2) < 5:
    print('エラー')
    print('指示にしたがってください')
```

2か所の変更が必要(めんどくさい)

具体例(ユーザー定義関数 使用後)

print('エラー') の下に
print('指示にしたがってください') を追加したい

```
def input_error():
    print('エラー')
    print('指示にしたがってください')

num1 = input('10以下の整数を入力してください：')
if int(num1) > 10:
    input_error()
num2 = input('5以上の整数を入力してください')
if int(num2) < 5:
    input_error()
```

The diagram illustrates the flow of control from the function definition to its calls. A grey speech bubble labeled "共通部分を関数に" (Common part as a function) points to the first two lines of the `input_error` function. Two yellow arrows originate from the `input_error()` call in the main code and point to two yellow boxes containing the text "関数の呼び出し" (Function call), which are positioned next to the respective `input_error()` calls in the main code.

具体例(ユーザー定義関数 使用後)

print('エラー') の下に
print('指示にしたがってください') を追加したい

The screenshot shows a code editor with a dark theme. A dashed-line box highlights a user-defined function named `input_error()`. Inside this box, the first two lines of the function are shown: `def input_error():` and `print('エラー')`. A green square icon is placed over the second line. The word "関数名" (function name) is written above the second line. The word "インデント" (indentation) is written to the left of the first line. To the right of the second line, the word "関数" (function) is written. Below this box, the main script starts with `num1 = input('10以下の整数を入力してください:')`. It then contains an if-statement: `if int(num1) > 10:`, followed by a call to the `input_error()` function: `input_error()`. An orange arrow points from the text "関数の呼び出し" (function call) to this line. The script continues with another `input()` statement and an if-statement involving `num2`.

```
def input_error():
    print('エラー')
    print('指示にしたがってください')

num1 = input('10以下の整数を入力してください:')
if int(num1) > 10:
    input_error() ← 関数の呼び出し
num2 = input('5以上の整数を入力してください')
if int(num2) < 5:
    input_error() ← 関数の呼び出し
```

具体例(ユーザー定義関数 使用後)

さらに `print('エラー')` を `print('inputエラー')` に変更したいときは、関数を変更するだけ

The image shows a code editor interface with a dark theme. A dashed-line box highlights a user-defined function named `input_error()`. The code within this function prints two messages: 'inputエラー' and '指示にしたがってください'. A callout bubble points to the first `print` statement with the text 'ここを変更するだけ' (Change here). Below the function, the main script uses `input_error()` twice to handle user input validation. The first call is associated with a callout '関数の呼び出し' (Function call) pointing to the line `input_error()`. The second call is also associated with a '関数の呼び出し' callout pointing to the same line. The word 'インデント' (Indent) is written vertically on the left side of the editor.

```
def input_error():
    print('inputエラー')
    print('指示にしたがってください')

num1 = input('10以下の整数を入力してください：')
if int(num1) > 10:
    input_error() ❷ 関数の呼び出し
num2 = input('5以上の整数を入力してください')
if int(num2) < 5:
    input_error() ❸ 関数の呼び出し
```

練習

次のプログラムコードを print_value という名前
の関数にして呼び出してください



```
print('==== 関数はじめ ===')  
value = 100  
print(f'value: {value}')  
print(f'ID: {id(value)}')  
print('==== 関数おわり ===')
```

※ 作成したプログラムコードは後で使います

答え

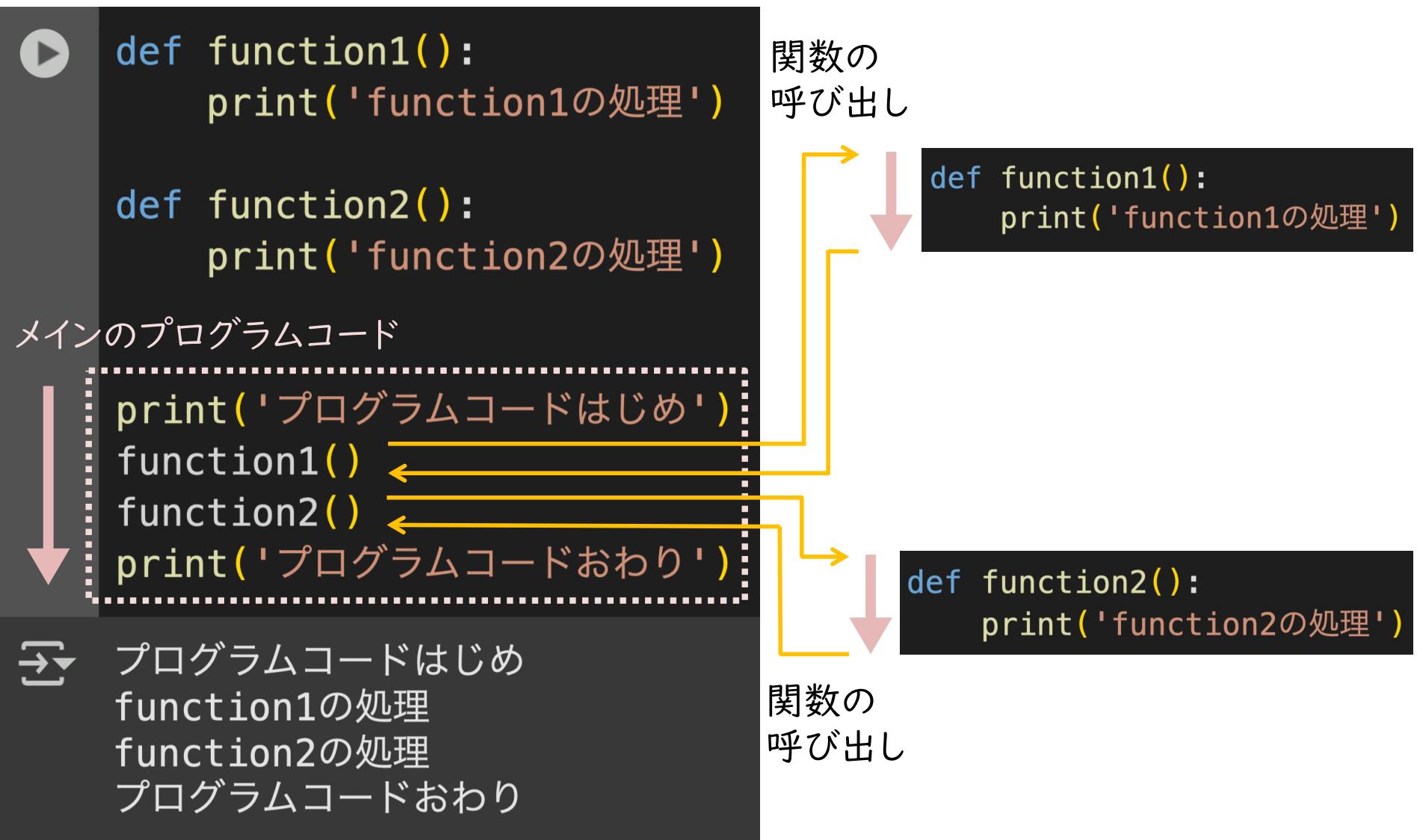
```
▶ def print_value():
    print('==== 関数はじめ ====')
    value = 100
    print(f'value: {value}')
    print(f'ID: {id(value)}')
    print('==== 関数終わり ====')

print_value()
```

```
→ ===== 関数はじめ =====
      value: 100
      ID: 10760904
      ===== 関数終わり =====
```

※ 作成したプログラムコードは後で使います

関数の呼び出しの流れ



関数の中で関数を呼び出すこともできる



```
def function_inner():
    print('function_innerの処理')

def function_outer():
    print('function_outerはじめ')
    function_inner()
    print('function_outerおわり')
```

メインのプログラムコード

```
print('プログラムコードはじめ')
function_outer()
print('プログラムコードおわり')
```

→ プログラムコードはじめ
function_outerはじめ
function_innerの処理
function_outerおわり
プログラムコードおわり

関数の呼び出し

```
def function_outer():
    print('function_outerはじめ')
    function_inner()
    print('function_outerおわり')
```

関数の呼び出し

```
def function_inner():
    print('function_innerの処理')
```

関数の定義位置に注意

- ・コンピュータはプログラムコードを上から下へ読んでいく
- ・関数の定義は、関数の呼び出しの前に行われている必要がある

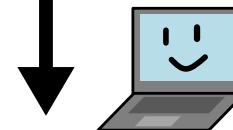
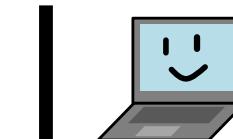


```
def function_before():
    print('関数の呼び出しの前')
```

```
function_before()
```



関数の呼び出しの前

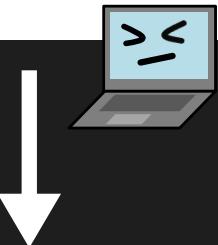


あとで
function_before って
関数を使うらしい
覚えておこう

さっき覚えた関数!

関数の定義位置に注意

- ・コンピュータはプログラムコードを上から下へ読んでいく
- ・関数の定義は、関数の呼び出しの前に行われている必要がある



なにこれ
知らない関数…

```
▶ function_after()  
  
def function_after():  
    print('関数の呼び出しの後')  
  
→ -----  
NameError Traceback (most recent call last)  
<ipython-input-6-1306401372> in <cell line: 0>()  
----> 1 function_after()  
      2  
      3 def function_after():  
      4     print('関数の呼び出しの後')  
  
NameError: name 'function_after' is not defined
```

変数のスコープ

- 変数には、その変数を参照できる範囲（スコープ）がある

```
▶ def print_value():
    print('== 関数はじめ ===')
    value = 100 # ローカル変数
    print(f'value: {value}') # ローカル変数
    print(f'ID: {id(value)}') # ローカル変数
    print('== 関数終わり ===')

print_value()
print(f'value: {value}') # ???
print(f'ID: {id(value)}') # ???
```

```
→ === 関数はじめ ===
      value: 100
      ID: 10760904
      === 関数終わり ===
```

```
NameError
<ipython-input-18-377409913> in <cell line: 0>()
      8
      9     print_value()
---> 10    print(f'value: {value}') # ???
      11    print(f'ID: {id(value)}') # ???
```

```
NameError: name 'value' is not defined
```

関数の中で定義された
変数はローカル変数で、
その関数の中でしか参
照することができない

先ほどの練習問題にこの2行を追加

注) エラーにならない場合は



del value

してください

18

練習



```
def print_value():
    print('== 関数はじめ ==')
    value = 100
    print(f'value: {value}')
    print(f'ID: {id(value)}')
    print('== 関数おわり ==')
```

← この「value = 100」があるときとないときで、結果はどのように変わりますか？

さらに、これを追加

```
value = 50
print_value()
print(f'value: {value}')
print(f'ID: {id(value)})')
```

そのような結果になる理由をまわりの人と議論してください

答え: value = 100 がある場合



```
def print_value():
    print('== 関数はじめ ==')
    value = 100 # ローカル変数
    print(f'value: {value}') # ローカル変数
    print(f'ID: {id(value)}') # ローカル変数
    print('== 関数おわり ==')
```

```
value = 50 # グローバル変数
print_value()
print(f'value: {value}') # グローバル変数
print(f'ID: {id(value)}') # グローバル変数
```



```
== 関数はじめ ==
value: 100
ID: 10760904 ← 違うID
== 関数おわり ==
value: 50
ID: 10759304
```

関数の中の
value(ローカル
変数)と関数の外
の value (グロ
ーバル変数)は名
前は同じだが、別
の変数(別のイン
スタンスを参照)

答え: value = 100 がない場合



```
def print_value():
    print('==== 関数はじめ ====')
    print(f'value: {value}') # グローバル変数
    print(f'ID: {id(value)}') # グローバル変数
    print('==== 関数終わり ====')
```

```
value = 50 # グローバル変数
print_value()
print(f'value: {value}') # グローバル変数
print(f'ID: {id(value)}') # グローバル変数
```



```
==== 関数はじめ ====
value: 50
ID: 10759304 ← 同じID
==== 関数終わり ====
value: 50
ID: 10759304
```

関数の中の
value と関数の
外の value は
同じ変数

参照した変数が
関数の中で定義
されていない場
合は、グローバ
ル変数が参照さ
れる

[発展] 関数の中でグローバル変数の値を変更する場合



```
def print_value():
    print('== 関数はじめ ==')
    global value
    value = 100 # ローカル変数ではなくグローバル変数になる
    print(f'value: {value}') # グローバル変数
    print(f'ID: {id(value)}') # グローバル変数
    print('== 関数おわり ==')

value = 50 # グローバル変数
print_value()
print(f'value: {value}') # グローバル変数
print(f'ID: {id(value)}') # グローバル変数
```



```
== 関数はじめ ==
value: 100
ID: 10760904 ← 同じID
== 関数おわり ==
value: 100
ID: 10760904
```

global 文：

関数の外で定義された変数の値を関数の中で変更することができる

ただし、プログラムコードの挙動が分かりづらくなるので、global 文の使用は個人的にはあまりお勧めしません。グローバル変数の値を関数を介して変更したい場合は、次に説明する引数と来週勉強する戻り値(返り値)を使った方が分かりやすいです

引数

- 関数を呼び出すときに、関数に値を渡すことができる
- 渡される値を**引数**という

```
def add_one(num):  
    print(num + 1)  
  
add_one(5)  実引数  
  
→ 6
```

関数の定義：

関数名の後ろの () に
渡される値を受け取る
ローカル変数を書く

```
def add_one(num):  
    print(num + 1)
```

関数の呼び出し：

関数名の後ろの () に渡す値を入れる

引数が複数ある場合も同様

- 引数が複数ある場合は引数列を渡す
- 引数を並べる順番に注意

The diagram illustrates the execution of a Python function with multiple arguments. On the left, a dark gray box contains the function definition:

```
def subtract(num1, num2):
    print(num1 - num2)
```

Below the definition, two calls to the function are shown:

```
subtract(5, 2)
subtract(2, 5)
```

On the right, a dark gray box shows the output of these calls:

```
3
-3
```

Two arrows point from the arguments in the second call to the corresponding parameters in the function definition: a blue arrow points from the value 2 to num1, and a green arrow points from the value 5 to num2.

At the bottom, another dark gray box contains the same function definition:

```
def subtract(num1, num2):
    print(num1 - num2)
```

練習

3つの要素をもつリスト a と b を、3次元デカルト座標 (x, y, z) におけるベクトルの成分表示だとみなして、内積と外積を計算する関数をつくってください（結果を関数の中で `print` してください）

モジュール(`import` 文)の使用は禁止

プログラムコードが正しく動作していそうか自分で手計算して確認すること！



```
def inner_product(vec1, vec2):
    print()

def cross_product(vec1, vec2):
    print()

a = [1, 1, 2]
b = [2, 2, 4]
inner_product(a, b)
cross_product(a, b)
```

内積 (inner product)

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^3 a_i b_i$$

外積 (cross product)

$$(\mathbf{a} \times \mathbf{b})_i = \sum_{j=1}^3 \sum_{k=1}^3 \varepsilon_{ijk} a_j b_k$$

ここで、 ε_{ijk} は Eddington のイプシロン

答えの例(他でもOK)

```

▶ def inner_product(vec1, vec2):
    result = 0
    for i in range(3):
        result += vec1[i] * vec2[i]
    print(result)

def cross_product(vec1, vec2):
    result = ['x', 'y', 'z']
    result[0] = vec1[1] * vec2[2] - vec1[2] * vec2[1]
    result[1] = vec1[2] * vec2[0] - vec1[0] * vec2[2]
    result[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0]
    print(result)      j          k      j          k
                        i
a = [1, 1, 2]
b = [2, 2, 4]
inner_product(a, b)
cross_product(a, b)

```

12
[0, 0, 0]

内積 (inner product)

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^3 a_i b_i$$

外積 (cross product)

$$(\mathbf{a} \times \mathbf{b})_i = \sum_{j=1}^3 \sum_{k=1}^3 \varepsilon_{ijk} a_j b_k$$

ここで、 ε_{ijk} は Eddington のイプシロン

キーワード引数

- 変数名を指定して値を渡すこともできる（キーワード引数）
- キーワード引数を使えば、引数を並べる順番を気にしなくて良い

```
def subtract(num1, num2):
    print(num1 - num2)

subtract(num1=5, num2=2)
subtract(num2=2, num1=5)
```

3
3

関数の呼び出し
(実引数)で「=」

デフォルト引数

- 関数の呼び出して値が渡されなかった場合に、**デフォルト値**を使うことができる（**デフォルト引数**）



```
def factorial(num, end=1):  
    result = 1  
    for i in range(end, num+1):  
        result *= i  
    print(result)
```

関数の定義（仮引数）で「=」
普通の引数よりも後ろに

```
factorial(5) # = 5!  
factorial(5, 1) # = 5!  
factorial(5, 3) # = 5 * 4 * 3
```

ローカル変数 end に渡す値を省略すると end に 1 が渡される。
factorial(5, 1) と同じ



```
120  
120  
60
```

先週勉強した sorted 関数は…

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>

`sorted(iterable, /, *, key=None, reverse=False)`

iterable の要素を並べ替えた新たなリストを返します。

デフォルト引数

2つのオプション引数があり、これらはキーワード引数として指定されなければなりません。

key には 1 引数関数を指定します。これは *iterable* の各要素から比較キーを展開するのに使われます (例えば、`key=str.lower` のように指定します)。デフォルト値は `None` です (要素を直接比較します)。

reverse は真偽値です。`True` がセットされた場合、リストの要素は個々の比較が反転したものとして並び替えられます。



```
a = [10, 30, 40, 0, 1]
print(sorted(a))
print(sorted(a, reverse=True))
```

キーワード引数



```
[0, 1, 10, 30, 40]
[40, 30, 10, 1, 0]
```

可変長引数

- 引数の数が関数の呼び出しのときに柔軟に決まるようになりたい場合は**可変長引数**を使う
 - print 関数も可変長引数をもつ関数

```
def sum(*nums):  
    total = 0  
    for num in nums:  
        total += num  
    print(total)  
  
sum(1, 2)  
sum(1, 2, 3)
```

3
6

変数名の最初に
「*」をつけた引数

```
▶ print('a', 2)  
print('a', 2, [0])  
→ a 2  
a 2 [0]
```

イメージはアンパック代入
関数の中の nums はタプルになる

```
nums = 1, 2
```

また、引数に「**」をつけると
可変長のキーワード引数になり、
辞書(dict)で関数に渡される

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: 6/22(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。)
 - 今回は、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、正弦関数（ $\sin(x)$ ）の $x=0$ まわりの Taylor 展開を $2n+1$ 次の項まで計算するユーザー定義関数と、余弦関数（ $\cos(x)$ ）の $x=0$ まわりの Taylor 展開を $2n$ 次の項まで計算するユーザー定義関数を含む未完成のプログラムコードです。

ここで、「正弦関数（ $\sin(x)$ ）の $x=0$ まわりの Taylor 展開の 3 次の項まで」とは x^3 の項まで、すなわち $x - \frac{x^3}{3!}$ のことを指します。「5 次の項まで」であれば、 $x - \frac{x^3}{3!} + \frac{x^5}{5!}$ です。

```
taylor_sin
"""正弦関数の x=0 まわりの Taylor 展開を 2n+1 次の項まで計算する関数"""

print(f'{:>3} : {}')

taylor_cos
"""余弦関数の x=0 まわりの Taylor 展開を 2n 次の項まで計算する関数"""

print(f'{:>3} : {}')

##### ここより上を書き換える #####
if __name__ == "__main__":
    pi = 3.141592653589793

    print('sin(pi/6)')
    print('---')
    for n in range(6):
        taylor_sin(x=pi/6, n=n)
        print('---')
        print('ans : 0.5')
        print('')

    print('cos(pi/3)')
    print('---')
    for n in range(9):
        taylor_cos(x=pi/3, n=n)
        print('---')
        print('ans : 0.5')
```

$$\sin(x) = \sum_{k=0}^n \frac{(-1)^k}{(2k+1)!} x^{2k+1} + O(x^{2n+3})$$

$$\cos(x) = \sum_{k=0}^n \frac{(-1)^k}{(2k)!} x^{2k} + O(x^{2n+2})$$

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。

課題

期待される出力結果は、以下の通りです。次数が大きくなるにつれ、正解に近づいていく様子を確認してください。

```
sin(pi/6)
-----
1 : 0.5235987755982988
3 : 0.49967417939436376
5 : 0.5000021325887924
7 : 0.4999999918690232
9 : 0.5000000000202799
11 : 0.4999999999999643
-----
ans : 0.5

cos(pi/3)
-----
0 : 1
2 : 0.45168864438392464
4 : 0.501796201500181
6 : 0.4999645653289127
8 : 0.500000433432915
10 : 0.499999963909432
12 : 0.5000000000217777
14 : 0.49999999999990047
16 : 0.5000000000000004
-----
ans : 0.5
```

注1)

「"""」で囲まれた文字列は `docstring` と呼ばれ、関数の説明を記述する複数行コメントです。`def` の次の行から始めることができます。
(参考：<https://peps.python.org/pep-0257/#what-is-a-docstring>)

注2)

`__name__` のように、ダブルアンダースコア「`__`」(Python ではダンダーと呼ぶ)で挟まれた変数は、Python で特別な意味をもった変数です。とりあえずは、「`if __name__ == "__main__":`」より下がメインのプログラム、それより上がユーザー定義関数を書く場所だと思ってもらって大丈夫です。
(参考：https://docs.python.org/ja/3.13/library/_main_.html)

課題の答えの例

```
def taylor_sin(x, n):
    """正弦関数の x=0 まわりの Taylor 展開を 2n+1 次の項まで計算する関数"""

    factor = x
    result = factor
    for k in range(1, n+1):
        factor *= -1 * (x**2) / ((2*k)*(2*k+1))
        result += factor
    print(f'{2*n+1:>3} : {result}')

def taylor_cos(x, n):
    """余弦関数の x=0 まわりの Taylor 展開を 2n 次の項まで計算する関数"""

    factor = 1
    result = factor
    for k in range(1, n+1):
        factor *= -1 * (x**2) / ((2*k-1)*(2*k))
        result += factor
    print(f'{2*n:>3} : {result}'')
```

問1

次のような関数が定義されています。

```
def function(a, *b, c = 5):
    print(a, b, c)
```

関数を呼び出したとき、エラーになるものはどれでしょうか？

◆◆◆ ヒント ◆◆◆

関数の呼び出し（キーワード引数、デフォルト引数、可変長引数）に関する問題です。

関数の呼び出しを行う際にキーワード引数を普通の引数（位置引数という）と混在させる場合は、キーワード引数を位置引数より後ろに並べなければなりません。また、引数の順番の入れ替えが可能なのは、キーワード引数同士の順番に限られます。

ちなみに、関数の定義では、位置引数、可変長引数、キーワード引数の順番に並べる必要があります。

```
function(1, 2, 3, c = 4)
```

```
function(c = 1, 2, 3, 4)
```

正解

```
function(c=2, a=1)
```

```
function(1, 2, 3, 4)
```

以下のプログラムコード

```
def add_one(value):
    value += 1

value = 1
add_one(value)
print(value)
```

を実行すると

1

と表示されます。これは、関数の中の `value` がローカル変数である一方、関数の外の `value` はグローバル変数であるためと学習しました。

それでは、以下のプログラムコード

```
def add_one(vector):
    vector[0] += 1
    vector[1] += 1
    vector[2] += 1

vector = [1, 2, 3]
add_one(vector)
print(vector)
```

を実行すると、どう表示されるでしょうか？

◆◆◆ ヒント ◆◆◆

ローカル変数、グローバル変数に関するひっかけ問題です。

プログラミング言語における関数の引数に関する挙動を表すものとして「値渡し」「参照渡し」といった用語がありますが、第7回の授業で勉強したように、Python における変数には値が格納されているのではなく、インスタンスの所在地番号が格納されているので、「値渡し」や「参照渡し」のどちらでもない「参照の値渡し」が行われます。そのため、引数がイミュータブルなオブジェクトなのかミュータブルなオブジェクトなので挙動が変わってしまうことがあります。この問題の場合でも、ミュータブルであるリストは、最初の int 型の例とは異なる挙動を示します。今は完璧に理解する必要はないですが、イミュータブルかミュータブルかで異なる挙動を示すことがあるということをうっすら覚えておくと良いでしょう。

[1, 2, 3]

[2, 3, 4]

正解

エラーになる

問3

以下のプログラムコード

```
def make_list(arg, new_list=[]):
    new_list.append(arg)
    print(new_list)

make_list(1)
make_list(2)
```

を実行すると、どのように表示されるでしょうか？

◆◆◆ ヒント ◆◆◆

デフォルト引数に関するひっかけ問題です。

この問題もリストがミュータブルであることが想定外の結果を生み出してしまう。make_list を初めて呼び出したとき、new_list のデフォルト値は空のリスト ですが、2回目呼び出したときは new_list のデフォルト値は [1] になってしまいます。

[1]
[2]

[1]
[1, 2]

正解

エラーになる