

計算物理学II

第5回

数値計算プログラミング入門（1）： 変数, 型, 四則演算

秋山 進一郎

2025年10月31日

授業日の確認

- ・ 全10回

- ・ 第1回：10月3日（金）

- ・ 第2回：10月10日（金）

- ・ 第3回：10月17日（金）

- ・ 第4回：10月24日（金）★

- ・ 第5回：10月31日（金）

- ・ 第6回：11月14日（金）★

- ・ 第7回：11月21日（金）

- ・ 第8回：12月5日（金）★

- ・ 第9回：12月12日（金）

- ・ 第10回：12月19日（金）★

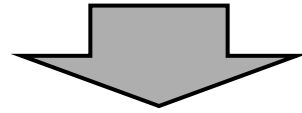
- ・ ★の付いた授業にてレポート課題を配布予定

今日（と次回）の授業の目標

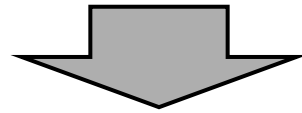
- **Pythonを使ってみる**
 - **変数と型, 四則演算**
 - **条件分岐, ループ**
 - **関数, スコープ**

毎回の授業の流れ

“compphy2”に移動する



“git fetch”と“git merge”で演習資料を入手



**“lecture*”に移動し、
“lecture_material_*.pdf”を開いて演習開始**

本日の演習内容

- Pythonを使ってみる
 - 変数と型, 四則演算
 - 条件分岐, ループ
 - 関数, スコープ

この授業では「Python3」を使います

- Pythonには2系（Python2）と3系（Python3）がある
 - Python2のサポートは2020年に切れている
 - 特別な理由がない限りはPython3を使う

Python3が使えることを確認

- Terminalから以下のコマンドを実行しよう
 - Python3のインストールされている場所を確認できます

```
akiyama@:~$ which python3  
/usr/bin/python3
```

- 今度は以下のコマンドを実行し, Python3のバージョンを確認しましょう

```
akiyama@:~$ python3 --version  
Python 3.10.12
```

使えるライブラリの確認

- ・ 今度は以下のコマンドを実行しよう

```
akiyama@:~$ pip freeze
```

- ・ 全学計算機に用意されているPythonのライブラリのリストを見ることができます
 - ・ 例えば, Numpy, Scipyがインストールされているか確認してみましょう
 - ・ pip freezeを実行して出てきたリストの中にnumpyやscipyがあればOK
 - ・ ライブラリがアルファベット順にリストされているのに注意して探してみましょう
 - ・ Numpy, Scipyとは代表的なライブラリで, 今後使う予定です

対話モードを使ってみる

- Terminalにpython3と打ってEnterを押すと対話モードが立ち上がります
 - 対話モードを立ち上げてみましょう
 - 以下の画面になればOK

```
akiyama@:~$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

足し算をさせてみましょう 1/2

- 整数(integer)同士の足し算をしてみましょう
 - 1+1と入力してEnterを押す

```
akiyama@:~$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
```

- 小数点を含む数字（浮動小数点数(float)）同士の足し算をしてみましょう
 - 1.0+1.0と入力してEnterを押す

足し算をさせてみましょう 2/2

- 整数と浮動小数点数の足し算をしてみましょう
 - 1+1.0と入力してEnterを押す

```
[>>> 1+1.0  
2.0
```

- 整数と浮動小数点数を足すと浮動小数点数になる

掛け算をさせてみましょう

- $2*2$ と入力してみる
- $2*2.0$ と入力してみる
- 整数と浮動小数点数を掛けると浮動小数点数になる

```
[>>> 2*2  
4  
[>>> 2*2.0  
4.0
```

割り算をさせてみましょう

- `/` を使うと割り算の結果が浮動小数点数で得られる
- `//` を使うと割り算の余を切り捨てた整数値が得られる
- 右の4つの計算を実行し、答えを確認しましょう

```
[>>> 4/2  
2.0  
[>>> 4//2  
2  
[>>> 5/2  
2.5  
[>>> 5//2  
2
```

浮動小数点数を扱う時の注意

- 浮動小数点数は、計算機の内部的にて、その数値に最も近い近似値を扱っている
 - 浮動小数点数には誤差がある
 - $0.1+0.1$ と入力してEnterを押す

```
[>>> 0.1+0.1  
0.2
```

- $0.2+0.1$ は？

```
[>>> 0.2+0.1  
0.300000000000000000000004
```

なぜ??

- ・ そもそも浮動小数点数とは、実数を有限桁の小数による近似値として扱う方式である
- ・ 計算機は二進数表現に基づいている
 - ・ 0.1を二進数で表現すると？（時間のある時に考えてみましょう）

真偽値 (bool)

- ・ 真偽値 (bool) とは真 (True) と偽 (False) の二値をとる型のこと
 - ・ 数値の比較をすると真偽値が返ってくる
 - ・ `1==1` と入力してみよう
 - ・ `==` は右辺と左辺が等しいことを意味する比較演算子
 - ・ 次に `1==2` と入力してみよう

```
[>>> 1==1  
True
```

```
[>>> 1==2  
False
```


浮動小数点数同士の等号比較には要注意！

- ・ 真偽値は後で学ぶ条件分岐やループの終了条件でよく使う
 - ・ 浮動小数点数には誤差があるため、等号比較は信頼できないことを肝に銘じておこう
- ・ $0.3 == 0.2 + 0.1$ と入力してみよう

```
[>>> 0.3==0.2+0.1  
False
```

- ・ 右辺の方が0.3よりもほんの少し大きいため、この命題は偽（False）である

複素数(complex)

- Pythonでは虚数単位としてjを用いる
 - `2+1j`と入力してみよう
- `real`, `imag`で複素数の実部, 虚部を取り出せる
 - `(2+1j).real`と入力してみよう
 - `(2+1j).imag`と入力してみよう
 - 複素数の実部, 虚部は整数値で入力しても浮動小数点数になることに注意しよう

```
[>>> 2+1j
(2+1j)
[>>> (2+1j).real
2.0
[>>> (2+1j).imag
1.0
```

文字列(string)

- ダブルクォーテーションやクォーテーションで囲まれた文字は文字列として扱われる
- 文字列同士は足し算可能（文字列と数値の足し算は不可）
- 以下の足し算を実行してみよう

```
[>>> "Hello "+"World!"  
'Hello World!'
```

- 数字もダブルクォーテーションやクォーテーションで囲めば文字列となる

```
[>>> "My favorite number is "+"7"  
'My favorite number is 7'
```

変数と型とは？

後日もう少し
丁寧に説明します

- ・ 変数とは、計算機上で再利用可能なようにラベル付けされた“値”のこと
- ・ 型とは、データの種類のこと
 - ・ 整数(int)型, 浮動小数点(float)型, 複素数(complex)型, 文字列(str)型, ...
- ・ Pythonでは、変数に対して事前に型を宣言する必要がない
 - ・ このようなプログラミング言語のことを「動的型付き言語」という
 - ・ C言語やFortranでは変数の型を事前に宣言する必要がある
 - ・ 型宣言が不要なので、プログラムの記述量が削減される
 - ・ ただし、処理速度はC言語やFortranに劣る

変数を使ってみよう 1/2

- aという変数を用意し, aに1という整数値を代入してみよう
 - a=1と書いて実行してみよう (実行しても特に何も出ません)
- printという命令を使うと変数の中身を見ることができる
 - print(a)と書いて実行してみよう
- typeという命令を使うと変数の型を見ることができる
 - type(a)と書いて実行してみよう
- aという変数を用意し, 1という整数値を代入したため, 変数aはint型になっています

```
[>>> a=1  
[>>> print(a)  
1  
[>>> type(a)  
<class 'int'>
```

変数を使ってみよう 2/2

- 今度は、aに“1”という文字列を代入してみよう
 - a=“1”と書いて実行してみよう（実行しても特に何も出ません）
- printという命令を使うと変数の中身を見ることができる
 - print(a)と書いて実行してみよう
- typeという命令を使うと変数の型を見ることができる
 - type(a)と書いて実行してみよう
 - 文字列を代入したため、変数aはstr型になっています

```
[>>> a="1"  
[>>> print(a)  
1  
[>>> type(a)  
<class 'str'>
```

int型, float型への変換

- intやfloatで囲むと整数や浮動小数点数に変換できる
- 今, aはstr型なので, 1+aと書いて実行するとErrorになる
 - Errorが出ることを確認し, メッセージを読んでみましょう
- int(a)とするとint型へ変換されて演算が可能になる
 - 1+int(a)と書いて実行してみよう

```
[>>> 1+a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
[>>> 1+int(a)
2
```

対話モードの終了

- Pythonの対話モードを終了するには, `exit()`を実行する
 - 対話モードを終了しましょう

```
>>> exit()  
akiyama@:~$
```


Pythonスクリプトファイル

- Pythonインタプリタには二種類ある
 - ① 対話モード（先ほどまで使っていたモード）
 - ② Pythonスクリプトファイルを使う方法
- ②の方法では、スクリプトファイルのパスを引数にして、python3コマンドを実行する
 - Pythonスクリプトファイルの拡張子は.py
- 以降では②の方法を使っていきます

Pythonスクリプトファイルを実行してみる

- まず, lecture5のsrcディレクトリをex_srcの名前でcp -r しましょう

```
akiyama@:~/compphys2/lecture5$ cp -r ./src ./ex_python
```

- cp -r したら, ex_pythonに入り, 一度lsで見ておきましょう

```
akiyama@:~/compphys2/lecture5/ex_python$ ls  
message.py  test_message.py
```

- vi message.pyで中身を見てみよう
 - このPythonコードの意味は後で説明
 - 以下のコマンドを実行しよう
 - Hello World!と出ればOK

```
def func(message):  
    print(message)  
  
func("Hello World!")
```

```
akiyama@:~/compphys2/lecture5/ex_python$ python3 message.py  
Hello World!
```

モジュール

- message.pyのように、Pythonコードで書かれたファイルはモジュールとも呼ばれます
 - Pythonでは、スクリプトとモジュールの間に明確な違いはない
 - この授業では、以下のようにスクリプトとモジュールという言葉を使い分けます
 - スクリプト = python3コマンドで直接実行されるもの
 - モジュール = Pythonのコードの中で読み込んで利用するもの
- 次のスライドでは、message.pyをモジュールとして使ってみます

message.pyをimportして使う

- 今度は, test_message.pyを見てみよう
 - vi test_message.pyで中身を見てみよう
 - test_message.pyでは, message.pyの中で定義されていたfuncという関数が呼び出されている
- test_message.pyを実行する前に, message.pyの最後の行をコメントアウトしておきましょう (単に以降の見栄えのため)
 - コメントアウトするには#をつける
 - test_message.pyを実行し, Hello!と出力されればOK

```
import message  
message.func("Hello!")
```

```
def func(message):  
    print(message)  
#func("Hello World!")
```

```
akiyama@~/compphys2/lecture5/ex_python$ python3 test_message.py  
Hello!
```

Pythonのライブラリ

- Pythonには非常に多くのライブラリが用意されている
 - ライブラリとはパッケージの集まりであり、パッケージとはモジュールの集まりである
 - ライブラリで実行できる計算はライブラリを使うべし
 - 例えば、基本的な行列演算（行列ベクトル積, 行列行列積, 固有値分解, and more ...）のPythonコードを自力で書く, ということは実践的にはほとんどしない
 - 基本的な数値計算ほど, ライブラリを使った方が圧倒的に計算が速いことがほとんど
 - ただし, 数値計算アルゴリズムを学ぶ目的では, 一度自分でコードを書いてみるのは良い勉強になる（ライブラリの中でこういった計算が行われているのかを知っておくことは実践的にも有益）

オフサイドルール 1/3

- Pythonではコードブロックをインデントで表現するため、コード上の空白には意味がある
- コードブロックとは、いくつかの文で表現されたコード上の塊のこと
- 具体例を見た方が早いので、以下のようなPythonコードを書いてみよう
- ex1_if.pyというファイルを作り、以下の6行のコードを入力

```
1  x = 1
2
3  if x > 0 :
4      print("Pro")
5  else:
6      print("Con")
```

printの前にインデント（空白）を入れること！

インデントはスペース4つが推奨
エディタが自動的に入れてくれるかもしれません

コードブロックは、
「キーワード（今の場合はifやelse） + （コロン）」
で始まる

オフサイドルール 2/3

- ファイルを保存したら、python3 ex1_if.py で実行してみよう

- Proと出力されればOK

- 今度はex1_if.pyのインデントをわざと消してみよう

- 右のように6行目のインデントをわざと消す

```
1  x = 1
2
3  if x > 0 :
4      print("Pro")
5  else:
6  print("Con")
```

- ファイルを保存したら、再びex1_if.py で実行してみよう

- 下のようなErrorがでる → インデントは確かにプログラム上で意味を持っている

```
akiyama@:~/compphys2/lecture5/ex_python$ python3 ex1_if.py
File "/home/akiyama/compphys2/lecture5/ex_python/ex1_if.py", line 6
    print("Con")
    ^
IndentationError: expected an indented block after 'else' statement on line 5
```

オフサイドルール 3/3

- ・ このように、インデントでコードブロックを表現するのがPythonの大きな特徴
 - ・ インデントでコードブロックを表現する方法を**オフサイドルール**と呼ぶ
- ・ インデントは同じ高さであれば空白何文字分でもよい
 - ・ ただし、**スペース4つ**で入れていくのが強く推奨される
 - ・ コードブロックの中にコードブロックを作る（ネスト）ような場合もある
 - ・ ネストするごとにインデントが深くなっていく

forによる繰り返し処理

- ex1_for.pyというファイルを作り，次の2行のコードを書こう

```
1  for i in range(10):  
2  |      print(i)
```

- ex1_for.pyを実行してみよう
- forによる繰り返し処理はループと呼ばれる
 - ループの中で値が変化しながら繰り返し実行される変数のことをループカウンタという
 - forがある文の最後に：があり，それ以降がコードブロック
 - コードブロックは何行でも良いが，同じブロックは同じ幅のインデントでないとダメ
- なお，Pythonのオフセット（基準値）は0であることに注意

ループカウンタに使う文字はなんでもよい

- ex2_for.pyというファイルを作り，次の3行のコードを書こう
 - ex2_for.pyを実行してみよう
- この例から以下のことが分かるだろう
 - ループカウンタに使う文字はなんでもよい
 - コードブロックは何行でもよい

```
1  for loop_count in range(10):  
2      a = loop_count*2.0  
3      print(a)
```

ループカウンタを使わないループ

- ex3_for.pyというファイルを作り，次の2行のコードを書こう

- ex3_for.pyを実行してみよう

```
1  for _ in range(10):  
2      print("Hello World!")
```

- Hello World!が10回表示されればOK
- コードブロックでループカウンタを使う処理が不要な場合，ループカウンタには文字を割り振る必要がなく，単に _ とすればよい

ifによる条件分岐

- ex2_if.pyというファイルを作り，右のコードを書こう
 - ex2_if.pyを実行してみよう
- $x < y$ と表示されればOK
- 1行目ではxに1, yに2を代入している
- 条件分岐の文（if文）はifから始まり，複数の条件分岐をさせる場合にはelif（else if）を使う
 - if, elifの後ろには真偽値を与える式を書いて：をつける
- elseの後ろにはコロンのみ
 - elseは「上（if, elif）で考慮した条件以外の場合」という意味になっている

```
1  x, y = 1, 2
2
3  if x == y:
4      print("x = y")
5  elif x > y:
6      print("x > y")
7  else:
8      print("x < y")
```

if文を書くときの注意

- どんな入力が来ても必ずどこかのコードブロックが実行されるようにif文を書きましょう
 - 左のコードだとa=0の場合の処理がない
 - 右のコードではa=0の場合にも対応している
 - if文の最後はelseにすると条件漏れを防ぐことができる

```
1  a = 0
2
3  if a > 0:
4      print("Positive")
5  elif a < 0:
6      print("Negative")
```

```
1  a = 0
2
3  if a > 0:
4      print("Positive")
5  else:
6      print("Not Positive")
```

while文

- 「ある条件が満たされている限り繰り返し処理を行う」には、whileを使う
- ex1_while.pyというファイルを作り、以下のコードを書こう
 - if同様、whileの後ろには真偽値を与える式を書く
 - 5行目はループが回るごとにxが1増えることを意味する
 - 数学的には変な書き方だが、(Pythonに限らず) プログラミングではお決まりの書き方
 - 5行目は6行目のように書くことも可能

```
1  x = 0
2
3  while x < 10:
4      print(x)
5      x = x + 1
6      #x += 1
```

ループ構文から抜ける方法

- forやwhileはループを作る構文である
- ある条件が満たされたらループを終了するような処理をしたいこともあるだろう
 - continueやbreakを使う

continue

- continueは「以下の処理をスキップし、次のループへ飛ぶ」という指示を与える
- ex1_continue.pyというファイルを作り、以下の4行からなるコードを書いて実行しよう
 - インデントの入れ方に注意
 - not xは「xが偽ならTrue, そうでなければFalse」となる真偽値
 - a % bは「a /b の剰余」を与える

```
1  for i in range(10):  
2      if not i%2 == 0:  
3          continue  
4      print(i)
```


クイズ

- ex1_continue.pyは「iが奇数の時はループをスキップする（print(i)しない）」
- 同じ出力結果を得るコードとして、「iが偶数の時だけprint(i)する」というコードを書いてみましょう
 - ファイル名はex2_continue.pyにしましょう
 - ループカウンタの走る範囲はex1_continue.pyの時と同じにしましょう
 - continue文を使わずに、for文とif文だけ

クイズの答え

- 「iが偶数の時だけprint(i)する」というコードの解答例（左下）

```
1  for i in range(10):  
2      if i%2 == 0:  
3          print(i)
```

```
1  for i in range(10):  
2      if not i%2 == 0:  
3          continue  
4      print(i)
```

- continue文を使った時との違いは， print(i)の書かれているコードブロック
 - continueを使わない場合， print(i)はif文のブロックにいる
 - continueを使った場合， print(i)はfor文のブロックにいる
- continue文を上手くと， やりたい処理（print(i)）をより浅いブロックに配置できる

break

- 「ある条件が満たされたらループを終了させる」には, breakを使う
- ex1_break.pyというファイルを作り, 以下のコードを書いてみよう
 - while True: は無限ループを作る
 - random.randint(0,1)は確率1/2で0か1を返す乱数
 - この例では, 無限ループを抜けるためにbreakを使っている
- 何度か実行して遊んでみましょう

```
1  import random
2
3  point = 5
4
5  print("Your initial point: ",point)
6
7  while True:
8      point += random.randint(0,1)*2 -1
9      print("Your point: ",point)
10     if point == 0:
11         print("No more point!")
12         break
13     if point == 10:
14         print("You got enough point!")
15         break
16
17  print("Game over")
```

関数

- Pythonではよく使う機能を**関数 (function)** として定義し、何回も使うことができる
- 関数へのインプットのことを**引数 (argument)** と呼ぶ
 - 引数は「ひきすう」と読む
 - 関数の実行結果を値で返す場合、返す値は**return**文で指定する
- 関数の定義の仕方は「def 関数名 (引数) :」であり、: 以下にコードブロックを作る
- 異なる.pyスクリプトファイルに書かれた関数もimportして使うことができる
 - すでに、冒頭で実行したmessage.pyとtest_message.pyではこのような関数の使い方をしていました

関数を作ってみよう

- 例として、足し算をする関数を作り、それを使ってみましょう
- ex_addition.pyというファイルを作り、以下のコードを書いてみよう
 - 実行結果を確認しましょう

```
1  def add(x,y):  
2      return x+y  
3  
4  a = add(1,1)  
5  print(a)  
6  
7  b = add(add(5+1j,2-3j),add(2+1j,1+4j))  
8  print(b)
```

関数を使う時の注意 1/2

- ・ スクリプトの中で、関数はどこでも定義できます
- ・ ただし、関数を使うことができるのは関数を定義した後です
- ・ **スクリプトは上の文から順番に実行される**ということを忘れないように

関数を使う時の注意 2/2

- 関数は何個でも定義できます
- 関数をたくさん定義するとスクリプトファイルの行数が増え、どこがプログラムのメインの部分なのか見づらくなります
- そういう場合には、下のコードの4行目のようなif文を書きます
 - 関数の定義はこのif文よりも上に書く
 - プログラムのメイン部分はこのif文以下のコードブロックに書く

```
1  def add(x,y):  
2      return x+y  
3  
4  if __name__ == '__main__':  
5        
6      a = add(1,1)  
7      print(a)  
8  
9      b = add(add(5+1j,2-3j),add(2+1j,1+4j))  
10     print(b)
```

ローカル変数

- ・関数内で宣言されている変数はローカル変数と呼ばれる
- ・例えば、以下のコードで変数xはローカル変数になっている

```
1  def func():  
2      x = 1  
3      print(x)
```


ローカルスコープ

- ex_scope.pyというファイル名で、以下のようなコードを書き、実行してみよう
- 実行するとNameErrorが出る
 - エラーメッセージを読んでみましょう
- ローカル変数が有効な範囲のことをローカルスコープと呼ぶ

```
1  def func():  
2      x = 1  
3      print(x)  
4  
5  func()  
6  print(x)
```

グローバル変数とグローバルスコープ

- ・関数の外（インデントがない地の文）で宣言されている変数を**グローバル変数**と呼ぶ
- ・グローバル変数が有効な範囲のことを**グローバルスコープ**と呼ぶ
- ・先ほどのex_scope.pyを右のように修正しよう
- ・修正を保存し実行しよう（1が出力されればOK）
- ・4行目の変数xがグローバル変数になっている

```
1  def func():  
2      |    print(x)  
3  
4  x = 1  
5  func()
```

スコープに関する注意 1/2

- ・ ローカルスコープからグローバルスコープは見える
- ・ 実際、修正後のex_scope.pyではローカルスコープ内のprint(x)でグローバル変数xが出力された

```
1  def func():  
2      |    print(x)  
3  
4  x = 1  
5  func()
```

スコープに関する注意 2/2

- ・ 反対に、グローバルスコープからローカルスコープは見えない
- ・ 実際、修正前のex_scope.pyではローカルスコープ内で変数xが定義されており、グローバルスコープでprint(x)と書いてもErrorになる

```
1  def func():  
2      x = 1  
3      print(x)  
4  
5  func()  
6  print(x)
```

クイズ

- 右のコードを実行した時, 7行目でprintされるxの値は?
- 右のコードを書いて調べてみましょう
 - 先ほどのex_scope.pyを編集してしまって構いません

```
1  def func():  
2      x = 2  
3      print(x)  
4  
5  x = 1  
6  func()  
7  print(x)
```

答えは1

- 関数funcの中でx=2と代入されたので、6行目でx=1がx=2で上書きされて、7行目のprint(x)の結果は2だ！ → スコープという概念に慣れていないとよく起こる間違い
- スコープという概念があり、コーディングの際には注意が必要だ、という認識を持っておきましょう

練習問題：Collatz問題 1/2

- Collatz（コラッツ）問題とは
 - 任意の正の整数 n に対し,
 - n が偶数なら n を2で割る
 - n が奇数なら n を3倍して1を足す
 - 「どんな n から始めても有限回の手続きで必ず1に到達する」というのがCollatzの予想
 - Collatz予想の証明はまだないが, $n=2^{68}$ までの初期値には反例が見つかっていないそうです

D. Barina, Convergence verification of the Collatz problem, J. Supercomput. 77, 2681-2688 (2021)

練習問題：Collatz問題 2/2

- 以下のような関数を実装しよう
 - 関数の名前はcollatzにしましょう
 - 正の整数nをインプット（引数）にする → `def collatz(n) :`
 - インプットした数字が1になるまで以下の二種類の処理を繰り返す
 - nが偶数ならnを2で割る
 - nが奇数ならnを3倍して1を足す
 - 各ステップでの数字をprintで出力させ, 1に至るまでの変遷を見れるようにする

ヒント

- 関数の引数を整数nとして
 - 「nが1になるまで~を繰り返す」 → `while n != 1:`
 - int型a, bに対し, 「`a != b`」はaとbが等しくないときTrue（真偽値）を返す

n=3の時の答え

- 例えば, n=3の時の数字の遍歴は右のようになります
- 自分の得た出力と一致しているでしょうか? 一致していればOK
- 他のnでもやってみましょう

```
3
10
5
16
8
4
2
1
```

書いたプログラムが実行できない場合

- ・プログラムを書いたが実行するとErrorになる
 - ・コードのどこかに**バグ**がある
 - ・Errorメッセージをよく読む
 - ・どこにバグがあるのかを特定し、修正する (**デバッグ**)

プログラム開発時の一般的な注意事項

- お勧めできない開発スタイル
 - いきなり長いコードを書いて、全部書き終わってから実行してみる
- コーディングにバグはつきもの
- 長いコードを書けば書くほどバグの混入する可能性が高まり、バグの特定も困難を極め、デバッグがしんどくなる
- こまめにコードを実行しながらコーディングすれば、バグの発見が早くなる
 - 例えば、新しく関数を定義するごとにその関数にバグがないかテストする
 - 前は正しく実行できたのに今はできない → 追加コードにバグがある可能性が高い
 - バージョン管理システム（Gitなど）を活用する

プログラムは実行可能だが正しく動作しているのか？

- プログラムが書けたら、本当に正しく動作しているのかをテストしないといけない
- 有効なテストの方法
 - 厳密解が分かっている問題を解かしてみる
→ 厳密解を再現できなければ、バグがある
 - 簡単なテストを考案し、それを確認する
(例) 「計算が正しく行われた場合、変数aと変数a'は一致しているはず」 → 確認
 - 他の人が書いたコードで得られた結果と比較する
→ 不一致ならば、どちらかあるいは両方にバグがある

役に立つサイトなど

- [Python.org](https://python.org)
- ライブラリをimportして使う場合, 公式ドキュメントを確認して正しく使うように
 - [Numpy](https://numpy.org)
 - [SciPy](https://scipy.org)
 - [SymPy](https://sympy.org)

練習問題：Leibnizの公式 1/2

- Leibniz（ライプニッツ）の公式

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots$$

- 交代級数であり， $\pi/4$ に収束する
- この公式を使って，円周率 π を近似的に計算する関数を作しましょう
 - 両辺を4倍したものを考える

練習問題：Leibnizの公式 2/2

- 以下のような関数を実装しよう
 - 関数名はleibnizにしましょう
 - 項数をインプット（引数）にする → `def leibniz(num_terms) :`
 - for文を使って分数の足し引きを実行する
 - 分子は全て4
 - 分母は1から始まる奇数
 - 足し算と引き算を交互に繰り返す
 - 足し引きして得られた値（ π の近似値）を最後にprintする

ヒント

- $1 + 1 + 1 + \dots$ の場合は下のように書くことができる

```
1  def arithmetic_sequence(num_terms):  
2      total = 0  
3      common_difference = 1  
4  
5      for _ in range(num_terms):  
6          total = total + common_difference  
7  
8      return total
```

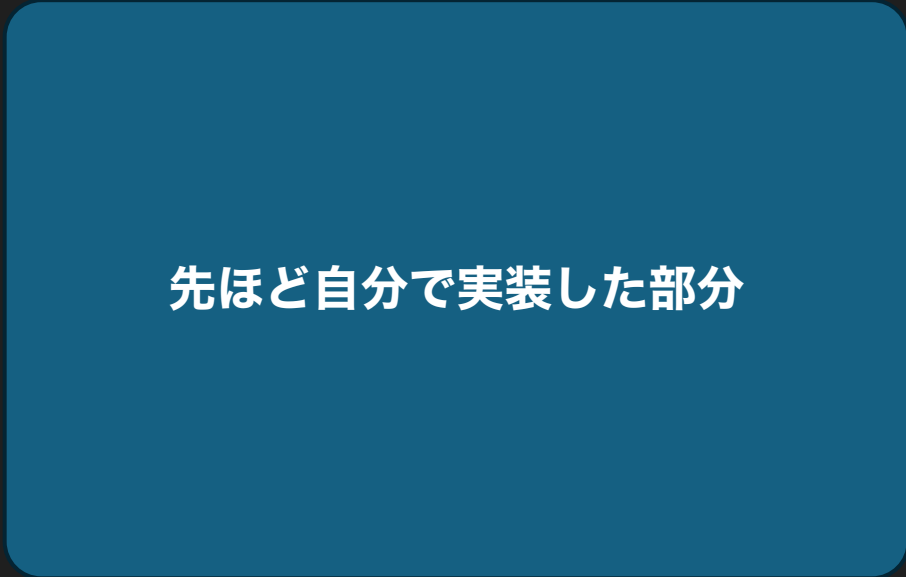
leibnitz(100), leibnitz(1000)の答え

• leibnitz(100)= 3.1315929035585537

• leibnitz(1000)= 3.140592653839794

どのくらい円周率を近似できているか調べてみよう

- Pythonでは標準モジュールのmathから円周率の値を参照できます
 - `math.pi`で π の値を使用できる
- 先ほど書いたコードを右のように修正してみましょう
- `num_terms`を大きくしていくとerrorは小さくなりますか？

```
1  import math
2
3  def leibnitz(num_terms):
4      
5
6
7
8
9
10
11
12
13
14
15  num_terms = 1000
16  error = leibnitz(num_terms) - math.pi
17  print(leibnitz(num_terms))
18  print(error)
```