計算数学・計算数学特論

講義用テキスト

(2022 年度版)

鳴門教育大学

計算数学・データ科学入門

はじめに

本講義の目的は, プログラミング言語 ”Python” を用いて初等数学の諸問題を解く方法を学び, アルゴリズムの概念や数学に関連したプログラミングの活動を体験することである. 積分の計算や確率や統計の問題を解くためのプログラムを実際に作成できるようになること到達目標とする.

Python は機械学習や人工知能 (Aritificial Intelligence: AI) に関連した便利なライブラリが準備されていることから、近年幅広く利用されるようになったプログラミング言語である。そこで、高等学校で必修化される「情報 I」でも主に用いられる言語に採用されている。しかし、便利であるということは、様々な数学的処理のブラックボックス化 (中身が見えない) が行われているということでもあり、プログラミングの初学者には不向きな面もあると考えている。そこで、本講義では、Python の便利なライブラリ群 (Numpy, Scipy, Pandas) を使わないプログラムを先に作成し、その後ライブラリの関数を用いた ”Python らしい処理” を紹介する、という方針を取った。したがって、このテキストの例題はある意味 ”Python らしくない” ものが多いのだが、四則演算などできる限り基本的な計算だけでプログラムを組み立てていくことで, 変数や代入, 繰り返し処理, アルゴリズムなど, プログラミングに特有の考え方の理解が深まると考えている[*1].

また, このテキストで力を入れた部分は統計処理に関する部分である. ここでは, ヒストグラムや相関係数など, 高校までに習う学習内容をプログラミングを用いて計算していく. ライブラリを用いればすぐさま求まるようなものばかりだが, やはりまずは四則演算・平方根などの基本的な数学操作のみでプログラムを作成する. その過程で, これらの統計的手法の意味の理解が深まることを期待している。また, 実際に統計量を計算することを通して, 統計量の定義や性質の理解を深めることは, 統計学・データ分析の学習には不可欠であると考えている. 最後に, Python のライブラリを用いて統計量の計算やグラフの作成が極めて用意に実現できることを確かめる [*2].

[*1] 外部ライブラリ以外にも, リスト内包記号やスライスなど, Python 特有の便利な文法も使わないようにした. 興味がある人はこれらについて調べ, このテキストのプログラムがもっと簡潔に記述できないか考えてみるとかなりの力がつくと思う. また, 文字列の扱いもデータ分析では必須の技術だが, このテキストではその利用は最小限に留めた. これはできるだけ数学の内容に重点を置くためである.

[*2] もちろん, 統計量の計算もグラフ作成も紙と鉛筆でできるが, コンピュータを用いることで, より多くのデータを, より速く処理したり試行錯誤したりすることができるので, 時間の節約になる. この種の ”手軽さ” がデータ分析の非常に重要なポイントなのだが, 往々にして軽視されることが多い.

目次

第 I 部
Python プログラミング入門

プログラムを作成するにはプログラミング言語を用いる. 普通の言語に日本語, 英語, ドイツ語など, たくさんの種類があるのと同じように, 計算機の言語にもいくつかの種類がある. 科学技術計算でよく用いられる C 言語や FORTRAN, 身近なものでは Excel にも VBA と呼ばれるプログラミング言語の機能を持っている. 最近, データを扱う際によく用いられている言語に Python がある. この講義では, Python を使い数学の様々な問題を学習していくことにする.

Python を学ぶ利点として, 以下のような点が挙げられる:

- 高等学校の「情報 I」ではメインに用いられるプログラミング言語に採用されている.
- プログラムを簡潔に記述できる (これを記述性が高いという).
- 大規模な統計処理も行えるほど処理速度が速い.
- 最近はデータサイエンティストという職種が現れてきたこともあり, 就職にも有利にはたらく可能性がある.

英語などの言語と同じく, プログラミング言語を使う場合にも, まずその **文法 (ルール)** を覚えなければいけない (といっても, 英語の文法ほど覚えることは多くない). 第 1 部では, まず Python の基本的な文法を学習することから始める.

Python にかかわらず, プログラミングを勉強する上で、大事なことは次の 2 点である:

- 例題で与えられたプログラムと, その実行結果を良く比較する. 特に, 「なぜ, そのような結果が得られたのか」を論理的に理解する.
- 例題を作成したら, さらにそれを修正して, 結果をどう変わるか, を調べてみる.

このふたつは, 非常に大事なことなので, 授業中時間が余った場合には必ず試して欲しい.

1 Phython プログラミングの概要

数学とコンピュータとの関わりは近年次第に深まりつつある. 実際, 理論的な解析だけでは解けないような難しい数学の問題であっても, コンピュータを用いれば (近似的ではあるが) 解の様子を理解できる場合が沢山ある[*1]. また, 大量のデータを入手しやすくなったことから, データサイエンスと呼ばれる分野の重要性が広く認識されるようになってきた. データサイエンスの主な目的は, コンピュータを用いたデータ分析や人工知能 (Aritificial Intelligence; AI) の開発である.

この章では, Python プログラミングのプロセスを概観する. Python は沢山あるプログラミング言語のひとつだが, 極めて高機能であることから, データサイエンスや AI の分野で最も良く用いられている[*2].

1.1 Google Colaboratory

Python のプログラム作成と実行には, Google Colaboratory というブラウザ (Chrome や Firefox など. Chrome が推奨) 上で利用できる環境を用いる. まず, Chrome (など) を開き次のウェブページにアクセスしよう (ブックマークしておくこと):

`https://colab.research.google.com/notebooks/welcome.ipynb`

次に右上にあるログインボタンを押し, ログインしよう. もし google アカウントが無ければアカウントを作成して欲しい. ログインができたら,

「ファイル」 (上部のタブ) → 「ノートブックを新規作成」

と順次選択すると, 新しいタブが開かれる. ここに Python のプログラムを作成していく. その前に, 画面左上部に「Untitled.ipynb」のようなタイトルがあると思うので, これをクリックし編集する. ここでは第 1 章なので「ch1」としよう. この授業では, 章ごとに新しいファイルを作成していくので, 以上の手順は繰り返し行う. また, 既に存在するファイルを開く場合は,

「ファイル」 (上部のタブ) → 「ノートブックを開く」

として開きたいファイルを選択しよう.

1.2 コードセルの作成

今度は実際にプログラムを記述していくセル (コードセルという) を作成する. そのために画面の左上にある

+コード

[*1] 身近な例として天気予報や, 車や飛行機の設計などがある.

[*2] プログラミングの基本的な考え方は, どの言語でもそれほど変わらない. どれかひとつの言語で基本的な考え方をマスターすれば, 他の言語も (その基本部分は) 容易に習得できる. ちなみに, 筆者は研究では主として C 言語と Python を用いている.

というボタンを押すと, 空白の枠 (コードセル) ができる. ここにプログラムを書き込んでいく.

1.3　プログラムの作成

では, 次のような python プログラムを作成してみよう. プログラムの内容については次回以降解説するので, 今は気にしなくても良い. 先ほど作成したコードセルに以下のような内容 (背景が灰色の部分) を書き込んでみよう:

例題 1.1　　**ex1.1**

```
i = 21
x = 3.14

print('整数値', i)
print('実数値', x)
```

また, 左上にある

+テキスト

というボタンを押すと, 空白の枠 (テキストセル) ができる. ここにプログラムの名前を書き込んでおこう. ここでは "ex1.1" と記入する (図 1 を参照). この作業は必ずしも必要ではないが, 多くのプログラムを作成することになるので, 整理のために行う. また, 試験やレポート課題の採点の際にも必要なので, 必ず記入するように.

図 1　Google Colaboratory のスナップショット. プログラムのタイトル "ex1.1"を入れてある.

1.4 プログラムの実行

プログラムが完成したら, プログラムを実行してみよう. 以下の 2 つの方法がある.

(1) 実行したいコードセルに, カーソルがある状態で

$$\boxed{\text{Control}} + \boxed{\hookleftarrow}$$

を同時に押す.

(2) 実行したいコードセルの左端にある ▷ という形のボタンを押す.

以上のいずれかの方法で, プログラムを実行すると, コードセルの下に次のように表示されるはずである.

実行結果

```
整数値 21
実数値 3.14
```

1.5 プログラムの保存

作成したプログラムを保存するには,

ファイルタブ → 保存

とするか,

$$\boxed{\text{Ctrl}} + \boxed{\text{S}}$$

とすれば (インターネット上に) 保存される. ブラウザがフリーズした場合などに備えて, 保存は小まめに行ってください.

1.6 演習問題

以下の 3 つのプログラムを作成しよう. 今回は, プログラミングの流れを覚えることが目的なので, プログラムの詳細については理解しなくても良い.

課題 1.1 (基本) 授業の例題 (ex1.1) の場合を参考に, 次のプログラムを作成せよ (例題と同様に, まずボタン +コード を押し, 新しいコードセルを開き, その中に記述する).

課題 1.1 **pr1.1**

```
a = 5
b = 3

print('加算', a + b)
print('減算', a - b)
print('乗算', a * b)
print('除算', a / b)
print('商',   a // b)
print('余り', a % b)
print('べき乗', a ** b)
```

`a ** b` でべき乗 a^b の値が計算できる. このとき 2 つの * の間にスペースを入れないこと.

課題 1.2 (基本) さらに, 次のプログラムも作成し, 実行せよ.

課題 1.2 **pr1.2**

```
for i in range (1, 10):
    print (i, i*i)
```

課題 1.3 (基本) プログラミングの最も簡単な使い方は電卓として使うことである. 次のプログラムを作成・実行し, その結果を吟味せよ.

課題 1.3 **pr1.3**

```
1/81, 1/9801
```

2　変数と演算　(Python の文法 1)

前回の例題 1.1 (ex1.1) を詳しく調べてみることから始める．ブラウザのブックマークから Google Colaboratory を開き (1.1 節を参照), 次のように前回作成したファイルを開こう．

「ファイル (上部のタブ)」 → 「ファイルを開く」 → 「ch1.ipynb」

2.1　変数と型

それでは, Python の文法を勉強することにしよう．まず, 図 2 の解説を良く読んで欲しい．

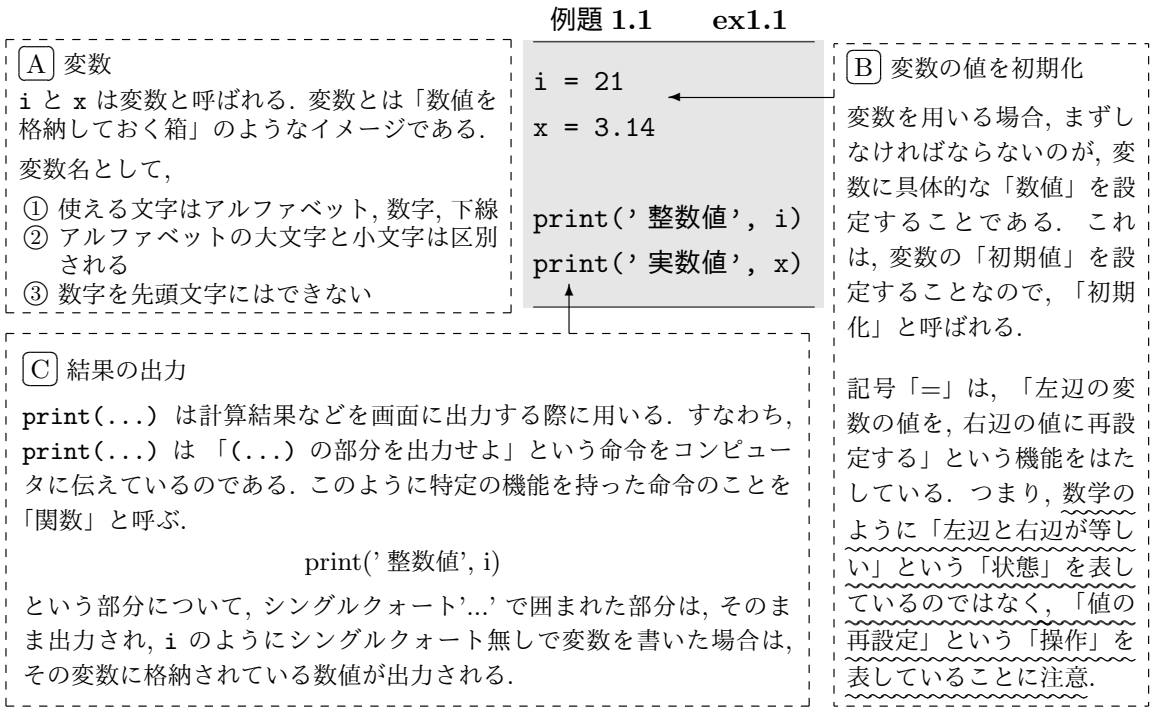

図 2　例題 1.1 (ex1.1) の解説

ex1.1 を実行すると次の結果が得られる[*1]:

実行結果

```
整数値 21
実数値 3.14
```

整数値の変数 i の値「21」と、実数値の変数 x の値「2.5」が表示される．

i の値には小数点以下の部分が無いのに対し, x の値には小数点以下の数が表示されていることに注意しよう．これは, Python が変数の種類 (整数値か実数値なのか) を自動的に判断していることを意味している．この変数の種類のことを「変数の型」と呼ぶ [*2].

[*1] この例から分かるように, Python の計算は「変数」を用いて行う (もちろん「1+2=3」のように, 直接数を扱うこともできる)．このことは, 別に Phython に限った話ではなく, どのようなプログラミング言語でもやはり「変数」を用いて様々な計算を行う．そして, プログラミングにおけるこうした「変数」の扱いは, 中学数学で習う「変数」の扱いととても良く似ている．この他にも, 「関数」や「代入」など, 数学用語が頻出する．つまり, (数値シミュレーションにせよ, ソフトウェアの開発にせよ) プログラミングという作業を行うためには, 数学の知識をきちんと身につけていることが重要なのである．

[*2] C 言語などのプログラミング言語では, 変数の型を明示的にコンピュータに伝える必要がある．Python でこの作業

2.2　処理の流れ

今度は, 新しいファイルを作成しよう. Google Colaboratory の上部にある「ファイル」から

「ファイル」 (上部のタブ) → 「ノートブックを新規作成」

を選択してファイルを作成する. 画面上部の「Untitled.ipynb」の部分をクリックして「ch2.ipynb」に変更しよう (右図参照).

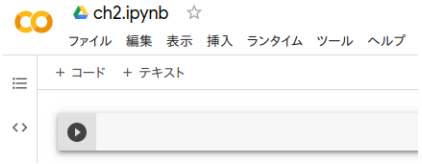

図 3　新しいファイル「ch2.ipynb」の作成

ここでは, プログラムの処理の流れを見てみよう. 前の例を少し編集した次のプログラム (ex2.1) を考える. このプログラムから, 実行後どのような結果が得られるのか予想してみよ.

例題 2.1　　ex2.1

```
i = 1
x = 2.5

print(i, x)

i = 10
x = 5

print(i, x)
```

処理の流れ

A　i と x の値を設定.

B　i と x の値を出力.
出力結果: 1 2.5

C　i と x の値を再設定.

D　i と x の値を再度出力.
出力結果: 10 5

まとめ

1. プログラムの処理は, 上から順に行われる.

2. 変数には, 常に「ある数値」が設定されている. この「数値」は, 処理が進むと変更されていく (この点は, 数学の「変数」とは若干異質なので, 戸惑う人が多い).

　左の例では, 変数 i の値は, 最初 1 だったが, 途中で 10 に変更された.

図 4　例題 2.1 (ex2.1) の解説

が不要になっているのは, Python における変数には「値」だけでなく「型」の情報も保有していることによる (その実体は C 言語の構造体と呼ばれるもの).

では, 実際に実行してみよう:

実行結果

```
1 2.5
10 5
```

1 列目の

”1 2.5”

は 1 つ目の `print(i,x)` の出力であり, 2 列目

”10 5”

は 2 つ目の `print(i,x)` の出力である. このように複数の print 関数を用いると, 改行が行われる.

また, プログラムは上から順番に処理されていることに注意しよう. もう一度, 図 4 の解説を読んで, このような出力が得られた理由を理解しよう.

2.3　四則演算

次に, 四則演算 $(+, -, \times, \div)$ である. 等号「=」の使い方ももう一度確認しよう.

例題 2.2　　**ex2.2**

Ⓐ 変数 x の値を 1.1 に設定.

```
x = 1.1
```

Ⓑ 変数 y の値を「変数 x の値」に設定. そして出力.
出力: 1.1 1.1

```
y = x
print(x, y)
```

Ⓒ 変数 y の値を「x+2 の値」に設定. そして出力.
出力: 1.1 3.1

```
y = x + 2
print(x, y)
```

Ⓓ 変数 y の値を「x-2 の値」に設定. そして出力.
出力: 1.1 -0.89999...

```
y = x - 2
print(x, y)
```

Ⓔ 変数 y の値を「$(x-1)\times 2$ の値」に設定. そして出力.
出力: 1.1 0.20000...

```
y = (x - 1) * 2
print(x, y)
```

Ⓕ 変数 y の値を「$((x+2)\times 3)\div(3+x)$ の値」に設定. そして出力.
出力: 1.1 2.26829...

```
y = ((x + 2) * 3)/(3 + x)
print(x, y)
```

まとめ

1. 四則演算については, 通常の数学記号と同じ意味と考えて問題ない. ただし, 乗算 $\times$ には `*`, 割算 $\div$ には `/` を用いる. また, 乗算の記号「*」は省略できない (間違い易いので注意).

2. 丸括弧 ( ) の使い方も数学と同じ. ただし, 数式中では丸括弧しか使えない (Ⓕ を見よ).

3. 一方, 「=」はこの例でも, 「左辺の変数の値を, 右辺の値に再設定」という機能をはたしている.
したがって, 左辺の変数の値は変化するが, 右辺に現れる変数の値は変わらないことに注意.
この例では, y (左辺) の値は変化していくが, x (右辺) の値は全く変わらない.

図 5　例題 2.2 (ex2.2) の解説

実行結果

```
1.1 1.1
1.1 3.1
1.1 -0.8999999999999999
1.1 0.20000000000000018
1.1 2.2682926829268295
```

左側の列の数字は x の値を示し, 右側の列の数字は y の値を示している. 特に, x の値が変化していないことに注意. 値が変化するのは左辺の変数 (代入された変数) y だけなのである.

練習　上のプログラム ex2.2 では, -0.9 が -0.8999··· などとなってしまっている. これを修正するために, print(x, y) を次のように修正してみよう:

```
print(f'{x:5.2f} {y:5.2f}')
```

このとき, {...} の中の変数 x と y は, これらの変数に代入されている数値に置き替えられる. また, :5.2 の意味は

出力領域を 5 マス準備し, 小数点以下は 2 マス

を意味し, 5.2f の f は実数値であることを示している. 整数値を出力する際には 5d などとする (次ページの練習を参照).

2.4　代入文

既に何度もでてきた記号「=」による処理を,「代入」と呼ぶ. また,

```
y = x
```

のような文は「代入文」と呼ばれ,「変数 x に, 変数 y の値を代入する」と説明される. ここでは, 代入文の極めて重要な使用例を紹介しよう. この方法は今後頻繁に用いられる.

例題 2.3　　ex2.3

Ⓐ 変数 i の値を 1 に設定. そして出力.
出力: 1

```
i = 1
print(i)
```

Ⓑ 変数 i の値を「$i+1$ の値」に設定. そして出力.
出力: 2

```
i = i + 1
print(i)
```

Ⓒ 変数 i の値を「$i \times 5$ の値」に設定. そして出力.
出力: 10

```
i = i * 5
print(i)
```

Ⓓ 変数 i の値を「$i-4$ の値」に設定. そして出力.
出力: 6

```
i = i - 4
print(i)
```

Ⓔ 変数 i の値を「$i/2$ の値」に設定. そして出力.
出力: 3

```
i = i / 2
print(i)
```

まとめ

例えば,

```
i = i + 1;
```

を数学の等式と考えると全く無意味であることが分かるだろう. この文は,

「変数 i の新しい値」(左辺) を「変数 i の古い値 + 1」(右辺) に設定

という処理をしている. (プログラムとその実行結果を, 論理的に納得できるまで, 照し合せるように).

図 6　例題 2.3 (ex2.3) の解説

ex2.3 を実行すると以下のようになる:

実行結果

```
1
2
10
6
3.0
```

練習　最初の 4 つの `print(i)` を次のように修正してみよう:

```
print(f'{i:3d}')
```

ここで, `:3d` の意味は

出力領域を 3 マス準備し整数値を出力

ということである.

2.5　コメント (注釈)

プログラムが複雑になってくると, 途中に説明を書いておくと分かりやすくなる. Python では # (ハッシュ) から行末までは「コメント」として扱われ, プログラムとしては処理されない. プログラムの例を見てみよう. これは, 5 人の学生の成績の平均値 (mean) を求めるプログラムである.

例題 2.4　　**ex2.4**

```
# 5 人の点数
v=75
w=95
x=90
y=60
z=85

# 平均点の計算
mean = (v+w+x+y+z) / 5

# 平均点の出力
print('平均点=', mean)
```

A # (ハッシュ) から行末まではプログラムとして処理されない (無視される). そこで, # の後にプログラムの説明を記述することができる.

図 7　例題 2.4 (ex2.4) の解説

ex2.4 を実行すると以下のようになる:

実行結果

```
平均点= 81.0
```

練習　上のプログラム ex2.4 について, 5 人の点数の設定部分を次のように変更し実行してみよ:

```
# 5 人の点数
v,w,x,y,z = 75,95,90,60,85
```

これは, 数学の式で書けば,

$$(v, w, x, y, z) = (75, 95, 90, 60, 85)$$

というベクトルの式のように見ることができ, 数学的にも自然な書式と言える. そこで, このテキストでも頻繁に用いることにする.

2.6　演習問題

課題 2.1 (基本)　例題 ex2.4 を改良して, さらに 5 人の成績の分散[*3] を計算し, その値を次のように出力するプログラムを作成せよ.

実行結果

```
平均点= 81.0
分散= 154.0
```

平均点は, 5 人の成績を v, w, x, y, z とすると

$$\overline{X} = \frac{v+w+x+y+z}{5}$$

と表せる. 一方, 分散は, $\overline{X^2} - (\overline{X})^2$ であるので[*4],

$$\frac{v^2+w^2+x^2+y^2+z^2}{5} - \left(\frac{v+w+x+y+z}{5}\right)^2$$

を計算すれば良い. プログラム名は, pr2.1 とすること. x の 2 乗 (x^2) は, 例えば

```
x ** 2
```

として計算できる [*5].

なお, python には平均値や分散を計算する関数がある. 次のプログラムを実行して結果が同じになることを確認せよ.

課題 2.1.1　**pr.2.1.1**

```
import numpy as np

v,w,x,y,z = 75,95,90,60,85

print(np.mean([v,w,x,y,z]))
print(np.var([v,w,x,y,z]))
```

ここで, `np.mean()` は「ライブラリ np に含まれる関数 mean()」という意味がある.

[*3] データの散らばり具合を特徴付ける量. 分散が大きい程, 散らばりが大きい.

[*4] $\overline{X^2}$ は 2 乗した値の平均である.

[*5] プログラムが完成したら, 階乗の記号 ** を用いずに作成できないか考えてみよう.

課題 2.2 (基本)　プログラムの途中で, 変数 x の値と変数 y の値を入れ替えたいとしよう. そこで, 次のようなプログラムを書いたが, うまく入れ替わらない (実際に実行してみよ).

```
x = 2
y = 5

print('x, y = ', x, y)

x = y
y = x

print('x, y = ', x, y)
```

そこで, 上のプログラムを, うまく値が入れ替わるように修正せよ (変数の数を増やしても良い). プログラム名を pr2.2 とする. ただし,

といった式を用いないように. すなわち, 変数 x, y に設定されている数値自体が変更されるようにすること.

類題 (難)[*6]　変数を増やさずに, 値の入れ替えをやってみよ[*7].

[*6] 授業では類題の解説は一部を除いて省略する. 解答例はウェブページで公開する.

[*7] この類題の方法は, あくまで変則的なやり方であり, 普通は課題 2_2 の方法を用いる. 類題の方法は, 変数が少ないという点は良いのだが, 非常に読みにくいプログラムであるため. なお, Python に特有の記法として,

```
x, y = y, x
```

とすることもできる.

3　条件分岐　(Python の文法 2)

プログラミングで複雑な計算を行うには, 「条件による処理の分岐」と「繰返し処理」のふたつが特に重要である. この章では, 「条件による処理の分岐」を解説する. これは,

”条件 A_1”を満たすならば, ”処理 X_1”を実行
”条件 A_2”を満たすならば, ”処理 X_2”を実行
”条件 A_3”を満たすならば, ”処理 X_3”を実行
⋮　　⋮

のように, 条件によって違う処理を実行する場合に用いる.

3.1　if 文

では, ひとつ目の例をみてみよう. これは, 絶対値を出力するプログラムである.

例題 3.1　ex3.1

```
a = -10

print('元の値=', a)

if a < 0:

    a = -a

print('絶対値=', a)
```

A a の初期値を設定。

B 条件分岐.

C a の符号を変える:
「a の新しい値」を
「$-1\times(a$ の古い値$)$」に設定.

まとめ

1. if 文の書式は,

```
if 条件:
␣␣␣␣処理内容
次の処理内容
```

ここで, ␣ は半角スペースを表している. ”処理内容”と”次の処理内容”を区別しているのは, 行頭の空白 (インデントと呼ばれる) の有無であることに注意.

2. もし, ”条件”が満たされていれば, ”処理内容”が実行される.

3. もし, ”条件”が満たされていなければ, ”処理内容”を実行せず, ”次の処理”に進む.

4. 例題の”条件”は $a < 0$ である. 例えば $a = -10$ であれば, この条件は満たされているので, `a = -a` が実行され, a の値は 10 になる.

図 8　例題 3.1 (ex3.1) の解説: 絶対値を求める

ex3.1 を実行すると以下のようになる:

実行結果

```
元の値= -10
絶対値= 10
```

練習　上のプログラム ex3.1 について, 変数 a の初期値を様々な値に変えて実行し, 確かに絶対値が出力されることを確認せよ.

3.2　else 節付き if 文

前節の if 文は, ”条件”を満たさない場合には, 何も処理を行わなかった. しかし, ”条件”を満たさない場合にも, なんらかの処理をしたいことがある. このような場合には, else 節を用いると良い. 以下の例は, ふたつの変数の値を比較して, 大きい方の値 (最大値) を出力するプログラムである.

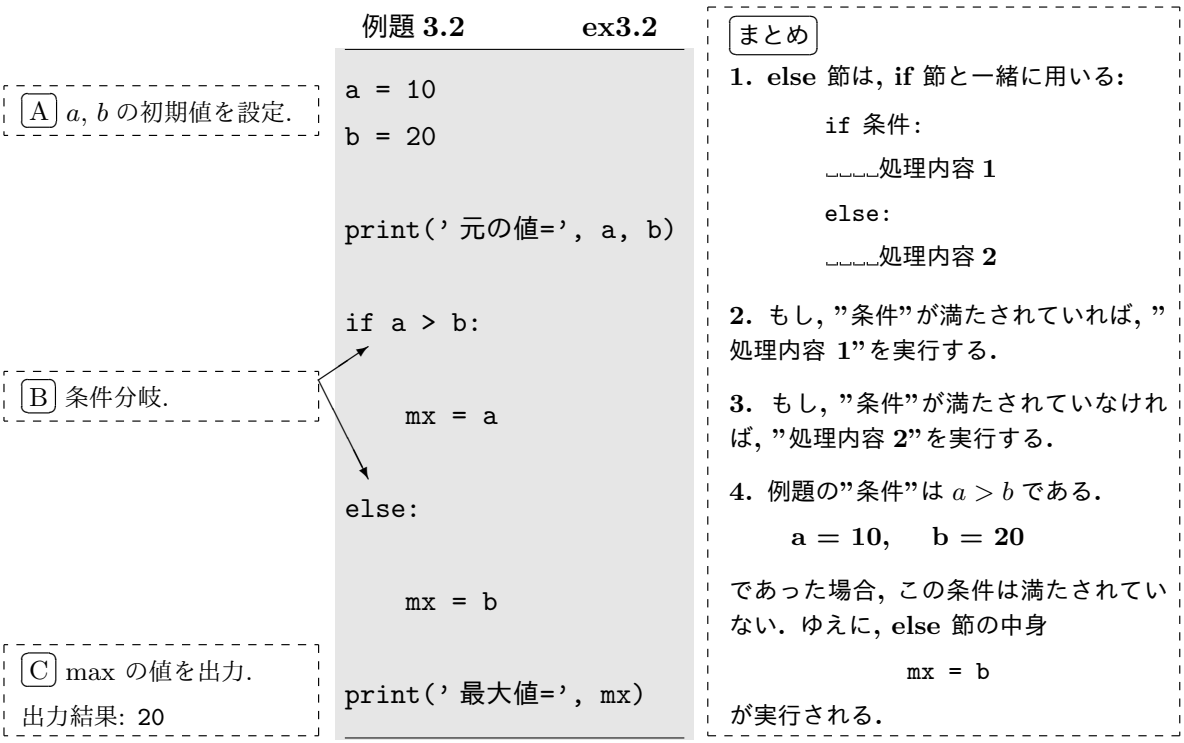

図 9　例題 3.2(ex3.2) の解説: 2 つの数値のうち大きい方の値を出力する

この例の場合にも, if 節と else 節を合わせた部分を if 文と呼ぶ. ex3.2 を実行すると以下のようになる:

実行結果

```
元の値= 10 20
最大値= 20
```

練習　上のプログラム ex3.2 について,

(1) 変数 a, b の初期値を様々な値に変えて実行し, 確かに最大値が出力されることを確認せよ.
(2) 最小値を出力するにはどのように修正すれば良いか, 考えよ.
(3) 上のプログラムは, else 節を用いずに, ひとつの if 節だけで書くことができる. どのようにすれば良いだろうか? (プログラム名を ex3.2.1 とする)
(4) python には最大値を求める関数 `max(...)` がある. 例えば,

```
max(a,b)
```

で `a`, `b` のうち大きい方の値が得られる. これを用いたプログラム ex3.2.2 を作成せよ.

3.3　elif 節付き if 文

条件が 2 つ以上ある場合には, elif 節を用いる. 以下の例は, 変数の符号を判定するプログラムである (変数が正ならば 1 を出力, 変数が負ならば -1 を出力, 変数が 0 ならば 0 を出力する).

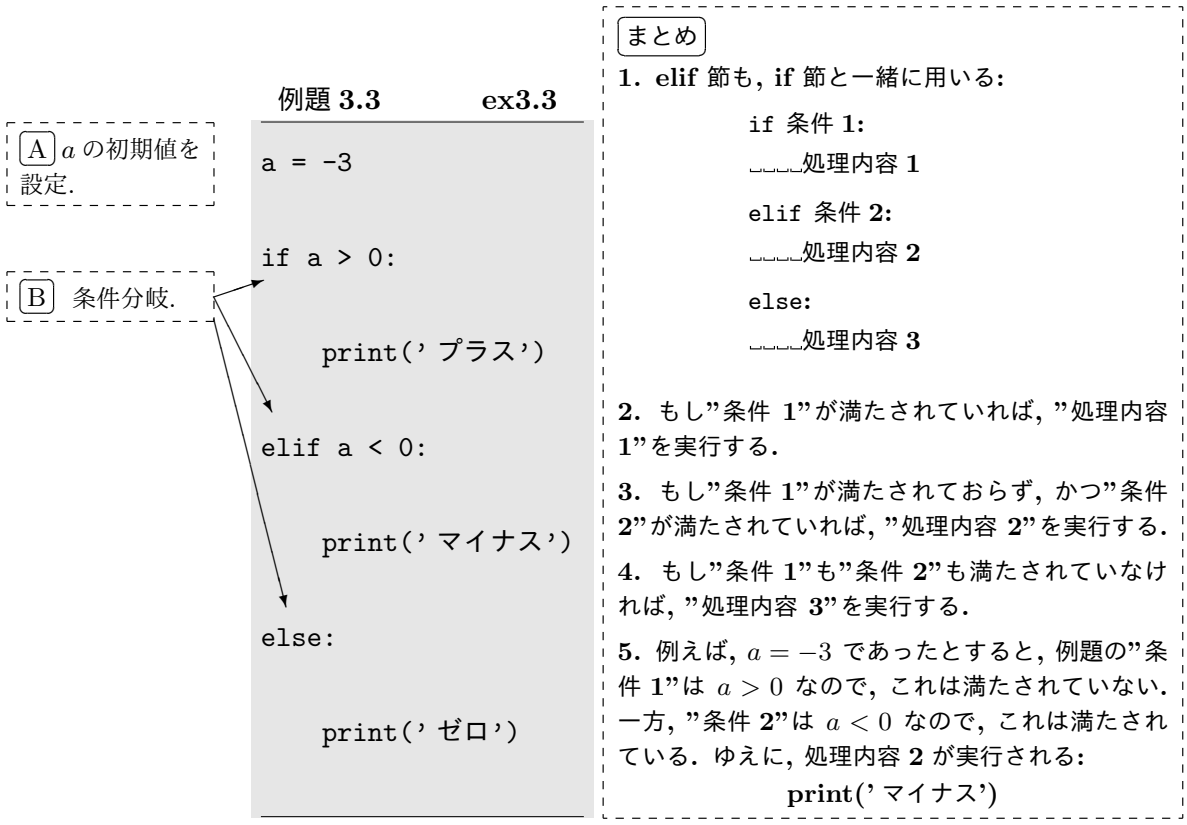

図 10　例題 3.3 (ex3.3) の解説: 整数の符号を判定する

上の例では, elif 文はひとつだが, 必要に応じていくらでも増やすことができる. また, 最後の else 文は (処理する内容がなければ) 省略できる. ex3.3 を実行すると以下のようになる:

実行結果

```
マイナス
```

練習　上のプログラム ex3.3 について,

(1) 変数 a の初期値を代えて実行し, 意図した結果が得られるか否か確認せよ.

3.4　値比較演算子

if 節や elif 節の条件には, 不等号 $>$ や $<$ の他に, 以下のようなものをよく用いる.

```
if i == j:
    "処理内容"
```

変数 i と j の値が等しい場合, ”処理内容”を実行する. イコールが 2 つであることに注意.

```
if  i != j:
    "処理内容"
```

変数 i と j の値が等しくない場合、”処理内容”を実行する。数学記号の $\neq$ に相当する。

```
if i <= j:
    "処理内容"
```

変数 i と j の値が $i \leq j$ を満たす場合、”処理内容”を実行する。

```
if i >= j:
    "処理内容"
```

変数 i と j の値が $i \geq j$ を満たす場合、”処理内容”を実行する。

3.5　論理演算子

ふたつの条件をともに満たす場合 (and) や, どちらかを満たす場合 (or), ある条件を満たさない場合 (not) に, 処理を実行したい, というケースもある. これらの場合には, 以下にあげる例のように

and　・　or　・　not

を用いる.

```
if i == j and a < b:
    "処理内容"
```

変数 i と j の値が等しく, かつ 変数 a が b より小さいとき, ”処理内容”を実行する.

```
if i == j or a < b:
    "処理内容"
```

変数 i と j の値が等しいか, あるいは 変数 a が b より小さいとき, ”処理内容”を実行する.

```
if not i = j:
    "処理内容"
```

変数 i と j の値が等しくないとき, ”処理内容”を実行する.

前節とこの節では if 文を例にして説明したが, 比較演算子と論理演算子は全て, elif 文の条件にも用いることができる.

3.6 演習問題

課題 3.1 (基本) 実数 a, b, c を与えたとき, 2 次方程式 $ax^2 + bx + c = 0$ の解の値

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

を出力したい. そのためには, 判別式 $D = b^2 - 4ac$ の正負を判定し, 実数解を持つかどうかをチェックする必要がある. また, 平方根 $\sqrt{\ldots}$ の値を計算するには, `numpy` というライブラリを読み込む必要がある. すなわち, プログラム内に

```
import numpy as np
```

という 1 行を追加する (下の pr3.1.py の 1 行目を参照). すると平方根の値を `np.sqrt(...)` (square root の略) とすることで計算できる.

そこで, 以下のようなプログラムを作成したが, まだバグが残っている. 正しい結果を出力するプログラムに修正せよ (類似のバグをひとつとカウントして, 合計 3 つのバグがある)[*8].

演習問題 3.1 **pr3.1**

```
import numpy as np

a = 1
b = 3
c = 2

D = b * b - 4ac

if D => 0:

    x1 = (-b + np.sqrt(D)) / 2 * a
    x2 = (-b - np.sqrt(D)) / 2 * a

   print(x1, x2)

else:

    print('解無し')
```

[*8] 授業後, 次のページに演習問題の解答を公開する: `http://mygch.g2.xrea.com/CMATH/python.html`

課題 3.2 (基本) 3 つの変数 a, b, c の値を与えたとき, これらの変数の最大値を出力するプログラムを作成せよ. プログラム名は, pr3.2 とする.

(ヒント: このプログラムは, ふたつの if 節だけで作成できる. つまり, else 節 や elif 節を用いずに作成できる. ただし, 正しく動作するならば, else 節 や elif 節を用いたプログラムでも良い).

類題 (やや難) 3 つの変数 a, b, c の値を与えたとき, これらの変数の中央値[*9]を出力するプログラムを作成せよ. プログラム名は, pr3.2.1 とする.

なお, Python にはもちろん中央値を計算する関数がある[*10]. 次のプログラムを実行して結果が同じになることを確認せよ.

演習問題 3.2.2　**pr3.2.2**

```
import numpy as np
print(np.median([2,5,4]))
```

課題 3.3 (やや難) ふたつの整数型の変数 a と b の値を与えたとき, 絶対値が大きい方の値を出力するプログラムを作成せよ (絶対値を出力するわけではないことに注意)[*11]. python には, 絶対値を与える関数 (abs) があるが, ここでは練習のため, これを用いずに作成すること. また, プログラム名は, pr3.3 とする.

(ヒント: 変数 a, b を二乗した値の大小によって出力を変える, などの方法が考えられる. ただし, 正しく動作するならば, その他の方法でも良い).

課題 3.4 (やや難) 与えられた数が, 奇数か偶数かを判定するプログラムを作成せよ. プログラム名は, pr3.4 とする.

[*9] あるデータ (数の集合) $\{a_n\}$ があったとき, これらの中央値とは, $\{a_n\}$ を小さい順に並べたとき, その数列の中央にくる値のことである (n が偶数の場合):

$$a_{i_1} < a_{i_2} < \cdots < \underbrace{a_{i_{n/2}}}_{\text{中央値}} < \cdots < a_{i_{n-1}} < a_{i_n}$$

中央値は, 統計学でよく使用されるもので, 最近では中学 1 年生で習う.

[*10] いくつかの方法があるが, ここでは numpy と呼ばれる科学技術計算で良く用いられるライブラリを利用している.

[*11] 簡単のため, $a = 5$, $b = -5$ のように絶対値が等しい場合には, どちらを出力しても良いこととする.

4 繰り返し処理 (Python の文法 3)

プログラミングをしていると, 同じ処理を繰り返し行うことが少なくない. 例えば, 以下のような場合がある:

(a) 和の計算: $\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n$ (級数の和)

(足し算を「繰り返す」)

(b) 乗積の計算: $\prod_{i=1}^{n} a_i = a_1 \times a_2 \times \cdots \times a_n$ (例えば, $a_i = i$ であれば階乗 $n!$ となる)

(掛け算を「繰り返す」)

(c) 積分値の計算: $\int_a^b f(x)dx$

(積分を「和」で近似し、その和の計算を (a) と同様に行う)

(d) 漸化式の計算: $a_{n+1} = f(a_n)$

(「a_n の値から a_{n+1} の値を求める」, という処理を繰り返す)

(e) 微分方程式などの時間発展: $\frac{dx(t)}{dt} = f(x)$

(微小時間の時間発展を「漸化式」で近似し, その漸化式を (d) と同様に計算する)

(f) 多数の要素からなる系の, 各要素に対し, 同じ処理を「繰り返す」ような場合.

(例えば, 偏微分方程式で記述される系など)

Python では, こうした繰返しの処理は「for 文」または「while 文」を用いる[*12]. これらは問題に応じて使い分ける必要があるが, この授業では簡単のため繰返し処理には, 「for 文」を主に用い, 必要に応じて「while 文」を用いる.

この章では, まず 「for 文」と 「while 文」の例を紹介した後, (a) 和の計算, (b) 漸化式の計算 (c) 積分の計算, を for 文の使い方を学習する (「while 文」はこの章の課題で, 複利計算の問題と, 第 8 章でユークリッドの互除法のプログラムを作成する際に使用する).

[*12] これらに加えて, `numpy` が提供するベクトル演算 (universal function) がある. これは極めて高速であり, かつプログラムが簡潔に記述できるため, 統計処理や科学技術計算では必須のツールとなってきている. しかし, 簡潔過ぎる記述は初学者向きとは言えないため, このテキストではほとんど用いない (ただし, 例外として, 関数のグラフを表示する際にのみ用いる).

4.1　for 文

まず, 簡単な例で, for 文の使い方を見てみよう.

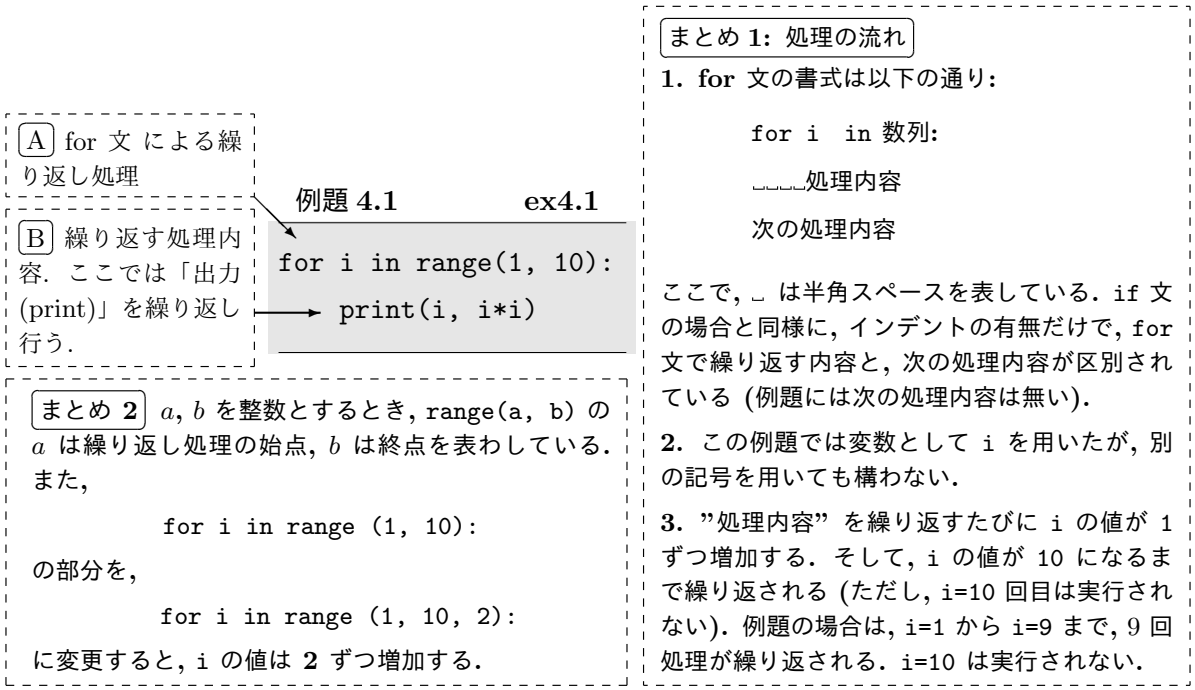

図 11　例題 4.1 (ex4.1) の解説: for 文

ex4.1 を実行すると以下のようになる (なぜ, このような結果になるのか, 理解して下さい):

実行結果

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
```

練習　上のプログラム ex4.1 について,

(1)「for i in range(1, 10)」 を 「for i in range(1, 10, 2)」 と書き換えるとどのように結果が変わるか確かめよ.

(2)「range(1, 10, 2)」 の 3 つ目の数値を変えて, 結果がどう変わるか確かめてみよ.

(3) print(i, i*i) を print(f'{i} {i*i:2d}') として実行してみよ (ex4.1.1 とする).

4.2　while 文

for 文の場合「何回繰り返すか」はあらかじめ分かっていたが, これが分からないようなケースもある (課題 4_6 や例題 6_2). 「何回繰り返すか」は分からないが, 代わりに「繰り返し処理を続ける条件」だけが分かっている, というようなケースである. そのような場合は, while 文を用いる. ここでは, 先の例題 4.1 と同様の処理を while 文を用いて行ってみよう.

例題 4.2　　ex4.2

```
i = 1

while i < 10:
    print (i, i*i)
    i = i+1
```

A while 文による繰り返し処理

B 繰り返す処理内容. ここでは「出力 (print)」を繰り返し行う.

まとめ 2 例題の while 文について,

```
i = i+1
```

の部分を,

```
i = i+2
```

に変更すると, i の値は **2** ずつ増加する.

まとめ 1: 処理の流れ

1. while 文の書式は:

```
while 条件:
    処理内容
```

2. 「処理内容」が実行されるたびに,「条件」がチェックされ,

- 「条件」が満たされていれば,「処理内容」を繰り返し,
- 「条件」が満たされていなければ, 繰り返し処理を終了する.

例題の場合は, i>=10 になったときに繰り返し処理が終了する.

図 12　例題 4.2 (ex4.2) の解説: for 文

ex4.2 を実行すると以下のようになる (前の例題と同じ):

実行結果

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
```

練習　上のプログラム ex4.2 について,

(1)「i = i+1」 を 「i = i+2」や「i = i+3」 と書き換えるとどのように結果が変わるか確かめよ. また, なぜそのような結果になったか, 考えよ.

(2) 条件はもっと複雑な式にすることもできる. 例えば, 条件「i<10」を「i*i % 12 != 0」とするとどうなるか (ex4.2.1 とする). また, なぜそのような結果になったか, 考えよ.

4.3　和の計算

平均値や分散の計算, 最小 2 乗法によるフィッティングなどの統計処理, あるいは積分値の計算など, **和を求める計算** は, 数値シミュレーションで頻繁に使用する. そこで, まず for 文を使った和の (簡単な) 計算例を見てみよう.

A sum の初期値を 0 に設定

B for 文による 繰り返し処理

C 繰り返す処理内容. 「変数 sum に i の値を加え」, さらに「出力 (print)」を行っている.

例題 4.3　　**ex4.3**

```
sum = 0
for i in range(1,11):

    sum = sum + i
    print (i, sum)
```

まとめ

1. ここでは,「和」の計算を,「足し算の繰り返し」として実現している.

2. 具体的には、

「**sum** の新しい値」

を

「**sum** の古い値 $+$ i の値」

に設定する処理 (`sum = sum + i`) を繰り返し行っている.

3. (以下の点は忘れがちなので要注意) **for** 文で和を計算する場合, 最初に和を計算する変数 (今の場合 `sum`) を, **0** に初期化 (`sum = 0`) しておく必要がある (**0** から少しずつ加算していくため).

図 13　例題 4.3(ex4.3) の解説: 和の計算例

ex4.3 を実行すると以下のようになる:

実行結果

```
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
```

ex4.3 を実行すると以下のようになる:

練習　上のプログラム ex4.3 について,

(1)「`for i in range(1,11)`」 を 「`for i in range(1,11, 2)` 」 と書き換えるとどのように結果が変わるか確かめよ. また, なぜそのような結果になったか, 考えよ.

(2) 他にも, いろいろと条件を変えて, 結果がどう変わるか確かめてみよ. 例えば, for 文を「`for i in range (5,16)`」 にするなど.

(3) `print(i, sum)` を `print(f'{i:2d} {sum:2d}')` として実行してみよ (ex4.3.1 とする).

4.4　漸化式の計算

次に微分方程式の時間発展の計算などで重要になる, 漸化式の計算を行う. ここでは,

$$a_{n+1} = ra_n, \qquad a_1 = 1$$

で定義される等比数列を第 25 項まで, 計算してみよう. a_1 から a_{25} まで求めるには, 25 個の変数を定義すれば良さそうである. 実際, そのようにして計算することもできるが, それでは項の数が増えるに従い, 変数の数も増大してしまう. 実は, a_n を 25 個 (実は, 何個であっても) 計算するために**必要な変数の数はたったひとつ**である. このことを, 以下のプログラムで確認してみよう.

例題 4.4　　**ex4.4**

```
a = 1
r = 0.9

for n in range(1, 26):

    print (n, a)
    a = r * a
```

A a の初期値を 1 に設定.

B 繰り返しの条件. 25 回処理を繰り返す.

C 繰り返す処理内容. 「出力 print(...)」と,「変数 a の値を再設定 (`a = r * a`)」をしている.

まとめ

このプログラムのポイントは,

`a = r * a`

という式である. これは,

「a の新しい値」(左辺)

を

「r の値」×「a の古い値」(右辺)

に再設定, という処理をしている.

この式は, 漸化式 $a_{n+1} = ra_n$ に対応している. このプログラムでは, このように a という変数をただひとつだけ宣言し, その値を更新していくことで, a_n の値を次々に求めていることに注意.

図 14　例題 4.4 (ex4.4) の解説: 漸化式のプログラム例

ex4.4 を実行すると以下のようになる:

実行結果

```
1 1
2 0.9
3 0.81
⋮
23 0.0984770902183612
24 0.08862938119652508
25 0.07976644307687257
```

練習　上のプログラム ex4.4 について,

(1) 変数 r の値を 1.1 にして計算してみよ. また, 得られた結果について (数学的に) 考察せよ.

(2) `print(n, a)` を `print(f'{n:2d} {a:7.5f}')` として実行してみよ (ex4.4.1 とする).

4.5　積分の計算

4.5.1　台形公式

積分は, 和の計算 (4.3 節) と漸化式の計算 (4.4 節) の両方を用いて, 近似的に計算する. まず, 積分

$$\int_a^b f(x)dx$$

を考えてみよう. 和の形で近似するために, 積分区間 $[a,b]$ を N 等分し,

$$x_n = a + \frac{b-a}{N}\ n = a + nh \quad (n = 0, \cdots, N) \quad (1)$$

とする. 特に, $x_0 = a$, $x_N = b$ が成り立つ (図 15). また, h は各分割の幅

$$h = \frac{b-a}{N} \quad (2)$$

である. このとき, 積分を以下のように分割しよう:

$$\int_a^b f(x)dx = \sum_{n=0}^{N-1} \int_{x_n}^{x_{n+1}} f(x)dx \quad (3)$$

次に, 上式の各積分を, 図 16 のように, 台形の面積で近似する. すなわち,

$$\int_{x_n}^{x_{n+1}} f(x)dx \fallingdotseq \frac{f(x_n)+f(x_{n+1})}{2} \times (x_{n+1}-x_n)$$
$$= \frac{f(x_n)+f(x_{n+1})}{2} \times h$$

すると, 元の積分 (3) は,

$$\int_a^b f(x)dx \fallingdotseq \sum_{n=0}^{N-1} \frac{f(x_n)+f(x_{n+1})}{2} \times h \quad (4)$$

のように, 和の計算により近似値が求まる. 上の計算式 (4) は近似式であるが, 分割の個数 N を増やすことで, 近似の精度を良くすることができる. また, 式 (4) を計算するには, x_n と x_{n+1} の値も必要であることに注意しよう. これらは, 漸化式

$$x_{n+1} = x_n + h \quad (5)$$

から求めることができる.

では, 次に具体的に積分の計算を行う. 以下の積分を計算してみよう:

$$\int_0^{\pi/2} \sin x dx$$

ここでは, 分割の数を $N = 100$ とする.

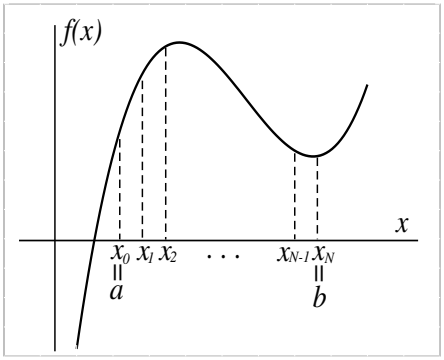

図 15　積分領域の分割

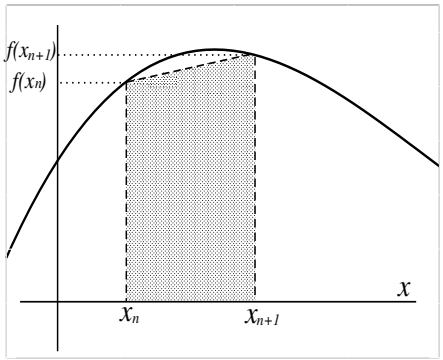

図 16　各積分領域を台形で近似

例題 4.5　**ex4.5**

```
import numpy as np

a = 0
b = np.pi / 2

N = 100
h = (b - a) / N

x = a
sum = 0
for i in range (0, N):

    sum = sum + (np.sin(x) + np.sin(x+h))/2 * h
    x = x + h

print('積分値=', sum)
```

A 三角関数 np.sin(...) と円周率 np.pi を用いるために numpy モジュールをインポートする．as np をつけることで, np.sin(...) と短縮できる．

B 積分範囲を与える変数 a と b の値を 0 と $\pi/2$ に設定．

C 分割の幅 h の値を, 定義式 (2) に従い設定．

D 変数 x の初期値を, 積分範囲の左端の値 a に設定．また, 和 (積分値) を表す変数 sum の初期値を 0 に設定．

E 式 (4) の右辺の和の計算．さらに, x_n の値も更新している．(これは漸化式 $x_{n+1} = x_n + h$ であることに注意)

まとめ

1. 積分の計算を和で近似して計算する．

2. 理論式 (1)〜(4) を忠実にプログラムに書き直していることに注目．このように数値シミュレーションでは, プログラミングする前に理論式を整理しておくことが必要である．

図 17　例題 4.5 (ex4.5) の解説: 数値積分のプログラム例

ex4.5 を実行すると以下のようになる:

実行結果

```
積分値= 0.9999794382396074
```

練習　上のプログラム ex4.5 について, 分割の個数 N を変えて実行してみよ．

4.6　演習問題

課題 4.1 (基本)

$$\left(1+\frac{1}{n}\right)^n$$

の値を $n = 10, 20, 30, \cdots, 100$ の場合に計算し出力するプログラム (pr4.1 とする) を for 文を用いて作成せよ.

* * *

Python では, べき乗 a^i の値は `a ** i` で計算できる.

類題 (基本)　上の課題で計算した量の極限値

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

で定義される数値は高校数学 (数 III) で習う. この数値はどのような記号で表わされ, またなんと呼ばれるか?

課題 4.2 (基本)　for 文を用いて, 下記のような結果を出力するプログラム pr4.2 を作成せよ.

実行結果

```
1  1
2  2
3  6
4  24
5  120
6  720
7  5040
8  40320
9  362880
10  3628800
```

ヒント: この問題は和の計算 (ex4.3) と同様の方法で計算できる.

類題 (難)　for 文を用いて, 下記のような結果を出力するプログラム pr4.2.2 を作成せよ. 下の出力の 2 列目はコンビネーション ${}_{10}C_i$ の値を $i = 0$ から $i = 10$ まで出力している.

実行結果

```
 0,     1
 1,    10
 2,    45
 3,   120
 4,   210
 5,   252
 6,   210
 7,   120
 8,    45
 9,    10
10,     1
```

for 文は入れ子状にすることができる. つまり,

```
for j in range (0, 9):
   for i in range  (0, 9):

      ...
```

のように, for 文の中に for 文を入れることができる. pr4.2.2 ではこのような, 2 重の for 文を使う必要がある.

課題 4.3 (やや難) 以下のような出力を出すプログラム pr4.3 を `for` 文を用いて作成せよ (九九の表):

実行結果

```
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 3  6  9 12 15 18 21 24 27
 4  8 12 16 20 24 28 32 36
 5 10 15 20 25 30 35 40 45
 6 12 18 24 30 36 42 48 54
 7 14 21 28 35 42 49 56 63
 8 16 24 32 40 48 56 64 72
 9 18 27 36 45 54 63 72 81
```

for 文は入れ子状にすることができる. つまり,

```
for j in range(1, 10):
   for i in range(1, 10):

      ...
```

のように, for 文の中に for 文を入れることができる. プログラム pr4.3 では, このような 2 重の for 文を使うと簡潔なプログラムになる. また, 出力は,

```
print (f'{i*j:2d}', end=' ')
```

のようにすると良い. ここで, `end=' '` は出力の後に, (改行の替わりに) スペースをひとつ入れることを指示している (通常は改行が入る).

課題 4.4 (基本) 次の漸化式を考えよう:

$$a_{n+1} = a_n - ra_n$$
$$b_{n+1} = b_n + r(a_n - b_n)$$

a_n と b_n を第 25 項目まで求めるプログラムを作成せよ (pr4.5). ただし, $r = 0.01$, $a_1 = 1000$ 及び $b_1 = 0$ とすること. (第 25 項目の値は, $a_{25} = 785.678...$, $b_{25} = 190.467...$ となる)

ちなみに, 第 1000 項まで計算してグラフにすると次のようになる:

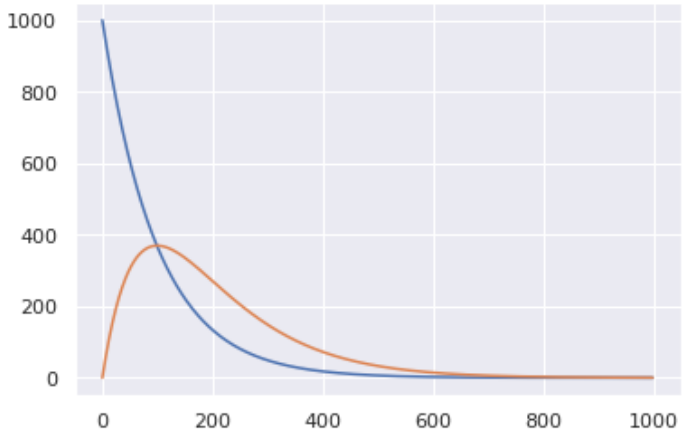

図 18　横軸は n, 縦軸は a_n (青い線) と b_n (赤い線) の値である. これは 2 段階で崩壊する放射性元素の簡単な数理モデルになっている (グラフを出力には次章で学習するリストを用いている. この図を出力するプログラムを pr4.4.1 とする).

類題 (やや難) 以下の漸化式で与えられるフィボナッチ数列を第 3 項 (a_3) から第 25 項 (a_{25}) まで出力するプログラム pr4.4.2 を作成せよ. さらに, 隣り合った項の比の値 a_{i+1}/a_i が黄金比 $(1+\sqrt{5})/2 = 1.618\cdots$ に近づくことを確かめよ.

$$a_{i+2} = a_{i+1} + a_i, \qquad a_2 = a_1 = 1$$

(ヒント: 上の漸化式で、$b_i = a_{i+1}$ とおいて, 2 変数の漸化式

$$\begin{cases} a_{i+1} & = (a_i \text{と } b_i \text{の関数}) \\ b_{i+1} & = (a_i \text{と } b_i \text{の関数}) \end{cases}$$

の形に変形できれば, 以前の問題 (pr4.4 など) と類似の課題となる. ただし, 他のやり方でも可).

課題 4.5 (基本) 台形公式を用いて積分

$$\int_0^1 \frac{4}{1+x^2} dx$$

の近似値を求めるプログラム (pr4.4) を作成せよ. 区間 $[0, 1]$ を 20 等分程度で OK.
また, 実際に積分を行うことで, 計算結果が正しいことを確かめよ.

課題 4.6 (やや難) 複利型預金において, 金利が 5% であるとする. ただし, 年複利すなわち, 1 年ごとに 5% ずつ増加するものとする. 元金と利子の合計が, 元金の 2 倍になる年数を求めるプログラム (pr4.6) を作成せよ. (ヒント: while 文を使う)

5　リスト　(Python の文法 4)

プログラムで使用する変数が多くなると, ひとつひとつ宣言していくのは大変である. そのような場合には, 「リスト」を用いると良い[*13]. リストを for 文などの繰り返し処理と組み合わせることで, プログラムが非常に単純化される.

5.1　リスト

成績の平均値を求めるプログラムは既に作成しているが (ex2.4), この例題では 5 人の学生の試験の点数を 5 つの変数に代入し足し合わせていた. しかし, 学生の人数が多くなるとこの方法は面倒である. そこで, ex2.4 をリストを使って書き直したのが以下のプログラムである:

例題 5.1　**ex5.1**

```
x = [80, 90, 100, 70, 60]  A リストの生成

mean = x[0] + x[1] + x[2] + x[3] + x[4]
mean = mean / 5

print (mean)
```

解説

1. リストの生成と初期化は次のように行う:

```
x = [80, 90, 100, 70, 60]
```

2. 上のように宣言した場合, `x[0]` には 80 が, `x[4]` には 60 が代入される. (`x[5]` を使うとエラーが出るので注意).

3. 配列は, 数学のベクトルのようなものと考えて良い.

図 19　例題 5.1 (ex5.1) の解説: リストの使用例 (平均の計算)

ex5.1 を実行すると以下のようになる:

実行結果

```
80.0
```

ここでは, 整数値だけのリストを作成したが, 整数値と実数値 (あるいは文字列なども) が混在したリストを作成することも可能である[*14] [*15].

[*13] Python にはリストに類似のものとして, タプルや辞書と呼ばれるものがある. これらの中で, 辞書はデータ分析では最も重要なものであるが, 本講義では利用しない.

[*14] すなわち, リストの要素 `x[0]`, `x[1]` などは全て, 通常の Python の変数であるため, 個々の変数の型が異っていても問題無いのである. これに対して, `numpy` で定義される「配列 (ndarray)」 は全ての要素が同じ型でなくてはならない. 一見すると, リストの方が便利そうだが, 「配列 (ndarray)」 では個々の要素の情報が少くなるため, 計算時間は大幅に短縮される. そのため, 科学技術計算やデータ分析では通常「配列 (ndarray)」が用いられる.

[*15] また, 一度作成したリストに要素を追加したり, 要素を削除したりすることもできるが, このテキストではこれらの操作は用いない.

5.2　リストと for 文の組合せ

前節の例 (ex5.1) は, 若干簡潔になったが, リストを使ったメリットはそれほど感じられない. 実は, リストは for 文と組み合わせて用いることで, より便利な道具となる. 以下のプログラム (ex5.2) は ex5.1 と同じ結果を出すが, 平均値の計算に for 文を用いている.

例題 5.2　　**ex5.2**

```
x = [80, 90, 100, 70, 60]

sum = 0
for i in range(0, 5):
    sum = sum + x[i]

mean = sum / 5

print (mean)
```

解説

1. リストの括弧 [ ] の中は整数なので, x[i] のように, 整数値を取る変数 i を括弧 [ ] の中に入れて計算を行うことが可能である.

2. i の値は **for** 文の繰り返し処理毎に値が **1** づつ増加する. したがって, **for** 文の繰り返し処理により,

```
sum = sum + x[0];
sum = sum + x[1];
sum = sum + x[2];
sum = sum + x[3];
sum = sum + x[4];
```

という順序で計算が行われる. i の値が **4** のときの処理が終わると, **for** 文のループを終了する.

図 20　例題 5.2 (ex5.2) の解説: リストと for 文の組合せ

ex5.2 を実行すると, ex5.1 と同じ結果が得られる.

練習　上のプログラム ex5.2 について,

(1) 上のプログラム ex5.2 で平均点を求めるだけであれば, リストを用いない方が簡潔かもしれない. しかし, 平均を求める以外にも様々な統計処理を行う場合, リストにデータを保存した方が便利であることが多い. そこで, 上のプログラムを分散も同時に出力できるように改良せよ. 分散については, pr2.1 を参照. このプログラム名を ex5.2.1 とする.

(2) 課題 3.2 の類題と同様に平均や分散を求める関数がある. 次のプログラムを実行して結果が同じになることを確認せよ.

```
import numpy as np

x = [80, 90, 100, 70, 60]
print (np.mean(x), np.var(x))
```

5.3　リストと for 文と if 文の組合せ

次に, if 文も同時に使う例を見てみよう. 次の例 ex5.3 は, 最高点と最低点を出力する.

例題 5.3　**ex5.3**

```
x = [80, 90, 100, 70, 60]

xmax = x[0]
xmin = x[0]

for i in range(1, 5):

    if x[i] > xmax:
        xmax = x[i]
    if x[i] < xmin:
        xmin = x[i]

print (xmax, xmin)
```

A この例のように, リストを if 文の条件の中で使うことが可能である.

図 21　例題 5.3 (ex5.3) の解説: 配列と For 文の組合せ

ex5.3 を実行すると以下のようになる:

実行結果

```
    100 60
```

練習　上の例 ex5.3 について,

(1) 不要な処理をしている部分がある. どこか考えよ.

(2) (最大値)−(最小値) を「範囲」と呼び, データの散らばり具合を表す統計量である (中学校数学の内容). 上の 2 点の修正を加えた上で, 範囲も出力するプログラムを作成せよ (ex5.3.1 とする).

(3) 最大値と最小値は組み込み関数 `max()` と `min()` を用いて計算することができる. すなわち,

```
print (max(x), min(x))
```

とするだけで最大値, 最小値が求まる. このことを確かめよ.

5.4　リストの応用例: ソート

プログラムを作成していると, 数字を順番に並べることが必要になることが多い. 次のプログラムは成績の良い順に並べるプログラムである.

例題 5.4　**ex5.4**

```
x = [80, 90, 100, 70, 60]

for i in range(0, 5):
    for j in range(i+1, 5):

        if x[i] < x[j]:

            tmp  = x[i]
            x[i] = x[j]
            x[j] = tmp

    print(i+1, x[i])
```

A 各 x[i] ($i = 0, \cdots, 4$) に対し, x[j] ($j = i+1, \cdots, 4$) との大小関係を調べる.

B x[i] と x[j] の値の入れ替え.

図 22　例題 5.4 (ex5.4) の解説: ソート

ex5.4 を実行すると以下のようになる:

実行結果

```
1 100
2 90
3 80
4 70
5 60
```

練習

(1) 上の例 ex5.4 について, 何故このプログラムでソートができるのか, 考えよ.

(2) x[i] と x[j] の値の入れ換えを 1 行で記述できないか考えよ (ヒント: 脚注 *7). ex5.4.1 とする.

(3) Python の組み込み関数 len を用いるとリストの長さ (要素数) を知ることができる. 例えば, ex5.4 の 2 行目に

```
N = len(x)
```

という式を入れると, N にはリストの長さ (ex5.4 の場合 5) が代入される. これを用いて, リストの長さが変わっても動作するプログラムに改良せよ. ex5.4.1 とする.

(4) Python の組み込み関数 sorted を利用してみよう. すなわち, 次の式を実行してみよ:

```
print(sorted(x, reverse=True))
```

さらに, 次のようにした場合はどのような結果になるか確認せよ.

```
print(sorted(x))
```

5.5　リストの応用例: 度数分布

これまで, データの平均値や分散を計算してきた (これらの量を統計量と呼ぶ). 統計量はデータの特徴を理解するために適しているが, データの全体像を知るには, 度数分布やヒストグラムを用いることが望ましい. ここでは, リストを用いて度数分布を求めよう. 次のプログラムのデータ (x) は, 徳島ヴォルティスの 2019 年の得点である (全 42 試合)[*16].

例題 5.5　　**ex5.5**

```
x = [3,1,0,1,1,1,2,1,1,1,
     0,3,0,1,0,0,1,2,2,1,
     1,1,3,1,5,0,0,1,6,2,
     1,1,1,2,1,2,3,2,7,0,
     2,3]
N = len(x)

h = [0,0,0,0,0,0,0,0]
for i in range(0, N):
    h[x[i]] = h[x[i]] + 1

for i in range(0, 8):
    print(f'{i}点: {h[i]:2d}回')
```

A 徳島ヴォルティスの 2019 年の得点のデータ.

B 各得点の度数を代入するためのリスト h. 最初に 0 にしておく.

C 第 i 試合の得点 x[i] に対応する度数 h[x[i]] を 1 増やす.

D 得点 i の度数 h[i] を出力.

図 23　例題 5.5 (ex5.5) の解説: 度数分布

ex5.5 を実行すると以下のようになる:

実行結果

```
0点:  8回
1点: 18回
2点:  8回
3点:  5回
4点:  0回
5点:  1回
6点:  1回
7点:  1回
```

[*16] このデータはホームページ http://mygch.g2.xrea.com/CMATH/python.html からコピー・ペーストできる.

練習　次の問に答えよ.

(1) ex5.5 に次の 4 行を追加してヒストグラムを表示させよ (ex5.5.1):

プログラムに追加する部分

```
cv = [0,1,2,3,4,5,6,7]
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.bar(cv, h, width = 0.5)
```

ここで, 関数 bar の変数とパラメータの意味は次の通りである:

① cv (class value: 階級値) は横座標の値のリスト (他の変数名でもよい)

② h は縦座標の値 (棒の高さ) のリスト (他の変数名でもよい)

③ width はヒストグラムの各棒の幅 (width はパラメータの名前を表しているので, 別の名前に変えることはできない. 0.5 はパラメータの値であり, こちらは調整する)

(2) データ (x) から, 度数分布 h もコンピュータで求め, ヒストグラムをかくことができると便利である. そのような関数 hist を使ってみよう.

例題 5.5.2　**ex5.5.2**

```
x = [3,1,0,1,1,1,2,1,1,1,
     0,3,0,1,0,0,1,2,2,1,
     1,1,3,1,5,0,0,1,6,2,
     1,1,1,2,1,2,3,2,7,0,
     2,3]

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.hist(x, bins=8, range=(-0.5,7.5), rwidth=0.5)
```

ここで, 関数 hist のパラメータには次の意味がある:

① bins は階級の個数 (ここでは, 0 点から 7 点までなので 8 つの階級があれば良い)

② rangle は階級の最小値と最大値

③ rwidth はヒストグラムの各棒の幅

これらパラメータの順序は入れ換え可能である.

(3) 関数 hist には戻り値がある (戻り値についての詳細は 6 章を参照). この戻り値は「度数分布」のリストと「階級の端点の座標」のリストである. 上の例の場合,

```
[8., 18., 8., 5., 0., 1., 1.]
[-0.5, 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5]
```

の 2 つのリストを得ることができる. 具体的には,

```
r = plt.hist(x,...e)
```

のようにすることで, 変数 r には上の 2 つのリストが代入される. このことを利用して, 円

グラフを作成してみよう.

例題 5.5.3　**ex5.5.3**

```
x = [3,1,0,1,1,1,2,1,1,1,
     0,3,0,1,0,0,1,2,2,1,
     1,1,3,1,5,0,0,1,6,2,
     1,1,1,2,1,2,3,2,7,0,
     2,3]

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
r = plt.hist(x, bins=8, range=(-0.5,7.5))
plt.close()
l = ["0","1","2","3","4","5","6","7"]
plt.pie(r[0], labels=l)
```

5.6　演習問題

課題 5.1 (基本) ふたつの n 次元ベクトル

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$$
$$\boldsymbol{y} = (y_1, y_2, \cdots, y_n)$$

が与えられたとき, これらの内積

$$\sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

の値を出力するプログラム (pr5.1) を作成せよ ($n = 6$ としよう). n 次元ベクトル x_i, y_i ($i = 1, \cdots, n$) を, リスト `x[i]`, `y[i]` (`i=1,2,...,n`) を用いて表すと良い. ふたつのベクトルの値は適当に設定して良い.

類題 (基本)　ふたつの 3 次元ベクトル

$$\boldsymbol{x} = (x_1, x_2, x_3)$$
$$\boldsymbol{y} = (y_1, y_2, y_3)$$

が与えられたとき, これらの外積

$$\boldsymbol{x} \times \boldsymbol{y} = (x_2 y_3 - x_3 y_2,\ x_3 y_1 - x_1 y_3,\ x_1 y_2 - x_2 y_1)$$

を出力するプログラム (pr5.1.1) を作成せよ (出力はベクトルであることに注意).

課題 5.2 (やや難) 6 人の学生の数学の得点 X と 英語の得点 Y が

学生	A	B	C	D	E	F
数学 (X)	80	70	60	90	70	50
英語 (Y)	90	80	70	80	60	80

であったとき, 数学と英語の点数の相関係数 R_{xy} を計算するプログラム (pr5.2) を作成せよ.

ヒント X の平均と分散を $\overline{X}$ と S_x^2, Y の平均と分散を $\overline{Y}$ と S_y^2 と表す. さらに, X と Y の共分散を S_{xy} と表す. 分散と共分散は以下の式で定義される:

$$S_x^2 = \overline{X^2} - (\overline{X})^2$$

$$S_y^2 = \overline{Y^2} - (\overline{Y})^2$$

$$S_{xy} = \overline{XY} - \overline{X} \times \overline{Y}$$

このとき, 相関係数は

$$R_{xy} = \frac{S_{xy}}{S_x S_y}$$

で与えられる. 平均と分散は既に計算しているので (pr2.1 を参照), 後は $\overline{XY}$ が計算できれば良い. これは, 各学生の「(数学の点) × (英語の点)」の平均値である:

$$\overline{XY} = \frac{\mathtt{x[0]} \times \mathtt{y[0]} + \cdots + \mathtt{x[5]} \times \mathtt{y[5]}}{6}$$

解説 相関係数は X と Y の相関の強さを表し, その値が

- 1 に近いときは相関が強い (数学の点が良いひとは, 英語の点も良い)
- 0 に近いときは相関が弱い (数学の点と英語の点の間に関連がない).
- -1 に近いときは, 反相関が強い (数学の点が良い人は, 英語の点が悪い).

と表現される. このように, 相関係数は 2 つのことがらの関連性を調べたい場合によく用いられる.

類題 1 (基本)　ライブラリを用いて相関係数を計算し, さらにデータをグラフにしてみよう (このような図を散布図と呼ぶ). 次のプログラムを実行してみよ (pr5.2.1 とする).

課題 5.2.1　**pr5.2.1**

```
x = [80, 70, 60, 90, 70, 50]
y = [90, 80, 70, 80, 60, 80]

# 相関係数の計算
import numpy as np
print(f'{np.corrcoef(x,y)}')

# 散布図の描画
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.plot(x, y, 'o')
```

ここで, `plt.plot(...)` がグラフを描画する関数である. 1 つ目と 2 つ目の変数は, それぞれ x 座標と y 座標のリストであり, `'o'` は円形のシンボルを用いてプロットすることを指示している.

さて, このプログラムを用いて,

(1) どのようなデータの場合に, 相関係数が 1 になるのか
(2) どのようなデータの場合に, 相関係数が -1 になるのか

を, データの数値や個数を変えることで調べよ.

類題 2 (基本)　箱ひげ図も作成してみよう. 次のプログラムを実行してみよ (pr5.2.2 とする).

課題 5.2.2　**pr5.2.2**

```
x = [80, 70, 60, 90, 70, 50]
y = [90, 80, 70, 80, 60, 80]

# 箱ひげ図の描画
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
l = ['Math', 'English']
plt.boxplot([x, y], labels=l, whis=(0, 100))
```

ここで, `plt.boxplot(...)` が箱ひげ図を描画する関数である. ここで, 関数 `boxplot` の変数とパラメータの意味は次の通りである:

① `[x, y]` 箱ひげ図の元になるデータのリスト (`x`, `y` もリストなので, リストのリストになっている)

② `label` は, 横座標のラベル (変数名) を表すリスト.

③ `whis` は whisker (ひげ) の略で, 箱ひげ図の「ひげ」を描く位置を指定している. `(0, 100)` は 0% と 100% の位置のデータ値, すなわち最小値と最大値を表わしている. 例えば, これを `(10, 90)` とすると 10% と 90% の位置のデータ値の所に「ひげ」が描かれる. 0%〜10% と 90%〜100% の範囲に入るデータは外れ値としてプロットされる[*17].

課題 5.3 (やや難) 次のような 10 個のデータに対して, 中央値を計算するプログラムを作成せよ.

プログラムの最初の部分

```
x = [9, 1, 10, 6, 3, 2, 1, 9, 7, 4]
N = len(x) # データの個数
```

`len(x)` という関数を用いると, リストの要素数が分かる. 上の例の場合は `10` という値が得られる.

中央値とはデータを小さい順に並べたとき, ちょうど中央に位置する数値のことである. 要素数 N が奇数の場合は

$$(N \text{ が奇数の場合}) \qquad a_0 \le a_1 \le \cdots \le \underbrace{a_{\frac{N-1}{2}}}_{\text{中央値}} \le \cdots \le a_{N-2} \le a_{N-1}$$

例えば, $N = 5$ の場合, a_0, a_1, a_2, a_3, a_4 であり, a_2 が中央値である.

要素数 N が偶数の場合には, ちょうど中央に位置する数値が無い:

$$(N \text{ が偶数の場合}) \qquad a_0 \le a_1 \le \cdots \le a_{\frac{N}{2}-1} \le a_{\frac{N}{2}} \le \cdots \le a_{N-2} \le a_{N-1}$$

($N = 4$ とすると, a_0, a_1, a_2, a_3 だが, a_1 も a_2 も中央ではない). そこで, この場合は中央付近の 2 数の平均

$$\frac{a_{\frac{N}{2}-1} + a_{\frac{N}{2}}}{2}$$

を中央値とする. プログラムはデータの個数を変えても動作するように作成すること (ヒント: `N/2` は実数となりリストの大括弧 `[···]` に入れるとエラーが発生する. `N//2` とすると整数値となる).

課題 5.4 (難) 課題 5.3 のデータについて, 第 1 および第 3 四分位数を求めるプログラムを作成せよ[*18]. プログラムはデータの個数を変えても動作するように作成すること (ヒント: 4 通りに場合分けする).

[*17] `whis = 2.0` のように数値をしていると,

(第 1 四分位数) − `whis` × (四分位数範囲)　〜　(第 3 四分位数) + `whis` × (四分位数範囲)

の位置にひげが描かれる. `whis` を指定しない場合は `whis = 1.5` として描かれる. ちなみにこれが箱ひげ図の元々の定義 (by Tukey) だそうである.

[*18] 四分位数の定義は複数あるが, ここでは高校数学の定義に従う. すなわち, データを中央値を境に前半部分と後半部分に分ける (データ数が偶数のときは, 中央値を除いて前半・後半に分ける). 前半部分の中央値を第 1 四分位数, 後半部分の中央値を第 3 四分位数と定義する.

6 関数 (Python の文法 5)

次の積分値を求めるプログラムを作成するとしよう:

$$\int_0^1 \frac{1+x}{(1+x^2)^2}dx$$

この積分計算を 4.5 節の方法で記述すると台形公式の部分が次のように長くなってしまう:

台形公式の和を計算する部分

```
sum = sum + ((1+x) / (1+x**2)**2 + (1+x+h) / (1+(x+h)**2)**2)/2 * h
```

これでは, どのような関数の積分なのか, 一見して分からない (可読性が悪い). このような場合に, 関数を用いると分かりやすく, また編集しやすいプログラムにすることができる[*19].

6.1 積分の計算 (再)

関数を用いて上の積分をするプログラムを作成すると次のようになる:

例題 **6.1** **ex6.1**

```
# 積分する関数を記入
def f(x):

    return ( (1+x) / (1 + x**2)**2 )

# 積分範囲 [a,b] と分割数 N を記入
a, b, N = 0, 1, 100

# 以下は変更不要
h = (b - a) / N
x = a
sum = 0
for i in range (0, N):

    sum = sum + (f(x) + f(x+h))/2 * h
    x = x + h

print('積分値=', sum)
```

Ⓐ 関数の定義の仕方は

```
def 関数 (変数):
␣␣␣␣処理内容
次の処理内容
```

ここで, ␣ は半角スペースを表している. "処理内容"と"次の処理内容"を区別しているのは, 行頭の空白 (インデントと呼ばれる) の有無であることに注意.

`return(...)` の `(...)` の部分の値が「関数の値」となる. この値を「戻り値」という.

Ⓑ `f(x)` は

$$\frac{1+x}{(1+x^2)^2}$$

の値に置き換わる. 同様に, `f(x+h)` は

$$\frac{1+x+h}{\{1+(x+h)^2\}^2}$$

の値に置き換わる.

図 24 例題 6.1 (ex6.1) の解説: Python の関数を用いた積分の計算.

[*19] この例の関数 `f(x)` は数学の関数と類似しているので, 比較的理解しやすい. これ以降の例題で少しずつ一般化していく.

ex6.1 を実行すると以下のようになる:

実行結果

```
積分値= 0.8926844981987139
```

6.2 平均の計算 (再)

ex6.1 では関数 `f(x)` の `x` は普通の変数だったが, リストを変数に取ることも可能である (数学的には多変数関数である). そのような例として, 平均値の計算をするプログラムを関数を用いて作り直してみよう.

例題 6.2 **ex6.2**

```
def f_mean(lst, N):
    """平均値の計算"""

    sum = 0
    for i in range(0, N):
        sum = sum + lst[i]
    return(sum / N)

x = [80, 90, 100, 70, 60]
print (f_mean(x, len(x)))
```

A 変数 `lst` はリストである. また, `N` は普通の変数である (`lst` の要素数を表す). このように関数は複数の変数を取ることができる.

B リスト `lst` に格納されている数値の平均値を計算. `for` 文の部分は 5.2 節のプログラムと同じであることに注意.

C 最後に平均値を `return()` で戻す (関数はここまで).

D `len(x)` はリスト `x` の要素数を与える (この例の場合 5).

図 25 例題 6.2 (ex6.2) の解説: 関数を用いた平均値の計算.

ex6.2 を実行すると以下のようになる:

実行結果

```
80.0
```

6.3 平均と分散の計算 (再)

これまでの例では, `return` 文で関数が戻すことができる数値は 1 つだけであった. しかし, 場合によっては複数の数値を戻したい場合がある. このように複数の戻り値を持つプログラムを作成してみよう (数学的には, 多変数のベクトル値関数ということができる).

例題 6.3 **ex6.3**

```
def f_describe(lst, N):
    """平均値と分散の計算"""
    m, v = 0, 0
    for i in range(0, N):
        m = m + lst[i]
        v = v + lst[i]*lst[i]

    m = m / N
    v = v / N  - m * m
    return(m, v)

x = [80, 90, 100, 70, 60]
print (f_describe(x, len(x)))
```

A 平均値と分散を戻す関数.

B 平均値と分散の計算. 5.2 節のプログラムを参照.

C 戻り値が 2 つ (平均値と分散) ある.

D 関数 `f_describe(..)` を呼び出すと, リスト `x` の平均値と分散が得られる.

図 26 例題 6.3 (ex6.3) の解説: 関数を用いた平均値と分散の計算.

ex6.3 を実行すると以下のようになる:

実行結果

```
(80.0, 200.0)
```

6.4 相関係数の計算 (再)

最後に, 課題 5.2 で計算した相関係数のプログラムをもう一度作成しよう. 相関係数の作成には, 平均値と分散を求める必要があるので, 前節で作成した関 数 `f_describe()` を利用する必要がある. もう 1 度, この関数定義を書くのは面倒なので (コピー・ペーストするだけだが, プログラムが長くなるのも問題なので), `f_describe()` を `cmath.py` というファイルに記述して, このファイルを `import` するようにしよう.

まずは, `cmath.py` をダウンロードしておく:

cmath.py のダウンロード

```
!wget http://mygch.g2.xrea.com/cmath.py
```

このとき次のような表示が得られれば, 正常にダウンロードできている.

実行結果

```
--2020-07-28 08:16:02--  http://mygch.g2.xrea.com/cmath.py
Resolving mygch.g2.xrea.com ... 150.95.8.244
Connecting to mygch.g2.xrea.com |150.95.8.244|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 740
Saving to: ‘cmath.py’

cmath.py            100%[===================>]     740  --.-KB/s    in 0s

2020-07-28 08:16:03 (93.1 MB/s) - ‘cmath.py’
saved [740/740]
```

これでプログラムのあるディレクトリ (フォルダのこと) に `cmath.py` が入っている状態になったので, プログラム内で `import` すれば, `f_describe()` を用いることができる. 実際のプログラムは次のようになる.

例題 6.4 **ex6.4**

```
import numpy as np
import cmath as cm

def f_cv_cc (lst1, lst2, N):
    """共分散と相関係数の計算"""

    m1, v1 = cm.f_describe(lst1, N)
    m2, v2 = cm.f_describe(lst2, N)

    v12 = 0
    for i in range(0, N):
        v12 = v12 + lst1[i] * lst2[i]

    v12 = v12 / N - m1 * m2
    r12 = v12 / np.sqrt (v1 * v2)
    return (v12, r12)

x = [80, 70, 60, 90, 70, 50]
y = [90, 80, 70, 80, 60, 80]
vxy, rxy = f_cv_cc(x, y, len(x))
print (f'相関係数:{rxy:5.3f}')
```

A 関数 f_describe() を使用するために, cmath.py をインポート.

B 相関係数を計算する関数.

C 共分散と相関係数を戻り値にする (関数 f_cv_cc() はここまで).

D 関数 f_cv_cc() を用いて相関係数と共分散を得る.

図 27 例題 6.4 (ex6.4) の解説: 関数を用いた相関係数の計算.

ex6.4 を実行すると以下のようになる:

実行結果

```
相関係数:0.274
```

練習 実は `cmath.py` には, 関数 `f_cv_cc()` も含まれているので, ex6.4 で `f_cv_cc()` の関数定義が無くてもプログラムは正常に動く (ただし, 関数定義を削除する以外に 1 点だけ修正する所がある). 実際に試してみよ (ex6.4.1 とする).

7 グラフィックス入門 (Python の文法 6)

この章と次章ではグラフィックス処理を通して関数のグラフや統計データを表示する方法を学ぶ[*20]. ここでは matplotlib というモジュールを使用する[*21].

以下の 2 つの場合に分けて説明する:

(1) 数値データの表示 (ヒストグラムや散布図など)

(2) 関数グラフの表示

(1) の 数値データをグラフにする方法は既にこれまでに紹介してきた. より高度な数値データの表示は **??** で紹介する. ここでは, (2) の関数の表示方法を調べてみよう.

7.1 リストの利用

まずは, $y = \sin x$ のグラフをかいてみよう.

例題 7.1 **ex7.1**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

n = 101
x = [0] * n
y = [0] * n

for i in range(0, n):

    x[i] = i * 2 * np.pi / 100
    y[i] = np.sin(x[i])

plt.plot(x, y, '-')
```

A `numpy` (配列の利用のため), `matplotlib` (グラフ描画のため), `seaborn`(綺麗なグラフ作成のため) の各モジュールを読み込んでいる.

B $y = \sin x$ 上の点 (x, y) を 100 個使ってグラフを表示させる. リスト `x` と `y` には, 次の `for` 文で値を代入する

C `x[i]` の値は 0 から 2π までの区間を 100 等分する座標 (なので 0.01 をかけている).

D `'-'` はを線を使って表示することを指示している.

図 28 例題 7.1 (ex7.1) の解説: リストを使用した関数の表示.

このプログラムを実行するだけならば `numpy` を読込むは必要無いが, 後のプログラムで使用するため入れておいた. このすぐ後の解説を参照.

[*20] 指数関数や三角関数 (加法定理を含む) や媒介変数表示, テイラー展開などを扱う. そもそもこれらの数学自体が良く分からないという人は, 高校の教科書 (指数関数・三角関数・媒介変数表示) や解析の教科書 (テイラー展開) などで復習して下さい.

[*21] matplotlib の使い方には大きく分けて 2 通りあり, これが混乱の原因になっている. 1 つは各種の命令を直接的に行う方法で, もう 1 つはオブジェクト志向型の方法である. 前者は直感的に分かり易いが, 複雑なグラフになるとやや扱いにくい. 後者は, 複数の図を並べるような複雑な図を作成する場合に便利であるが, 初心者向けではない. 本章では, 3 次元グラフの場合にのみ後者を利用する.

ex7.1 を実行すると次のようなグラフが得られる:

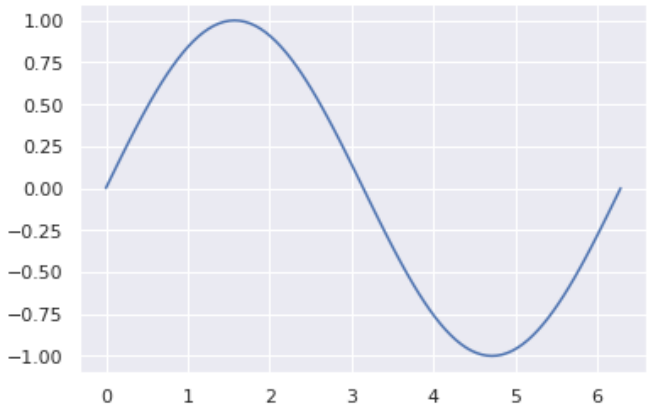

図 29　横軸は x, 縦軸は $y = \sin x$ を表す.

ex7.1 の中で,

ex7.import

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

の部分は, この章の全てのプログラムに記述しているが, 実は 1 度実行すればそれ以降は省略できる (ただし, ノートブックを開き直す度に, 最初に `import` する必要がある). 上のプログラム (ex7.import) を別途作成しておき, ノートブックを開いたときに最初に実行するようにすれば, この章のプログラムからこの 3 行を省略しても実行可能である.

7.2 numpy 配列 (ndarray) の利用

関数を 1 つ表示するだけなのに, リストを用いた前の例は少し大げさで面倒だ. ここでは, numpy の配列 (ndarray と呼ばれる. 以下, 単に配列と呼ぶことにする) を用いた方法を紹介する. 配列はリストと同じように, 数値の集合 (数列) を表している [*22]. 以下の例では, 配列 `x` には

$$[0,\ 0.01 * 2\pi,\ 0.02 * 2\pi,\ \cdots,\ 0.99 * 2\pi,\ 2\pi]$$

という 101 個の点からなる配列を生成している. これに対し, `np.sin(x)` は, 配列 `x` の各要素を sin で変換した値の配列を表している. すなわち,

$$[\sin(0),\ \sin(0.01 * 2\pi),\ \sin(0.02 * 2\pi),\ \cdots,\ \sin(0.99 * 2\pi),\ \sin(2\pi)]$$

という配列を意味している[*23].

ではこの配列を用いて, $\sin x$ に加えて, $\cos x$ と $\cos^2 x$, $\sin x \cos x$ のグラフも表示させてみよう.

例題 7.2 **ex7.2**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

x = np.linspace(0, 2*np.pi, 101)
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), ':')
plt.plot(x, np.cos(x)**2, '--')
plt.plot(x, np.sin(x)*np.cos(x), '-.')
```

A `np.linspace (a, b, n)` で x 座標の値の配列を生成している. 引数の意味は `a` は始点, `b` は終点, `n` は点の個数である. `n=101` の場合, 区間 `b-a` を 100 等分する. ここでは, `0` から 2π までの項数 101 の数列を与える.

B `plt.plot` を複数回使用すると, 複数の関数が重ねて描画できる. `'...'` の部分を変えることで線の種類を変えられる.

図 30 例題 7.2 (ex7.2) の解説: numpy を使用した関数の表示.

[*22] もちろん, 数値以外のデータ (文字列・真偽値など) を格納したリストや配列もあるが, このテキストでは扱わない.

[*23] sin 関数にベクトル $\boldsymbol{x}$ を代入して, 別のベクトル $\sin(\boldsymbol{x})$ を得るようなイメージだが, 通常の数式ではこのような式はありえないことに注意. numpy のベクトル演算は大変便利だが, このような普通の数学との不整合が沢山あるため, 数学の理解が不十分な場合, 余計な混乱を招く恐れがあると思う. なので, このテキストでは (やむを得ず) numpy の配列の利用は関数の描画の場合のみに制限した.

ex7.2 を実行すると次のようなグラフが得られる:

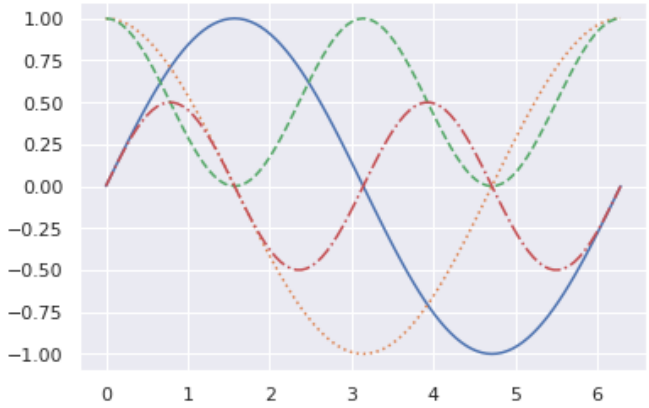

図 31　横軸は x, 縦軸は y を表す. どれも平行移動・拡大・縮小しているだけで, 同じ形をしている (当然ですが).

練習

1. $[-2\pi, 2\pi]$ の範囲で,

$$y = \sin x$$
$$y = x - \frac{x^3}{6}$$
$$y = x - \frac{x^3}{6} + \frac{x^5}{120}$$

のグラフを重ねて表示せよ (pr7.2.1 とする). ただし, y 座標の範囲を $[-2, 2]$ に制限しよう. そのために, 次の 1 行をプログラムに記述しておくこと:

```
plt.ylim(-2,2)
```

2. テーラー展開とは何か調べ, プログラム中の 2 つ目と 3 つ目の式がどのように得られたのか考えよ.

7.3　媒介変数表示

媒介変数表示とは,

$$\begin{cases} x &= \cos(t) \\ y &= \sin(t) \end{cases}$$

のように, 変数 x と y の関係を, t など別の変数を媒介として表す表示方法である. これは, t を消去すると $x^2+y^2=1$ という関数, すなわち円を表していることが分かる. これは簡単な例だが, 媒介変数表示が少し複雑になると, 関数形をイメージすることが困難になる.

まず, 円と楕円の媒介変数表示から, その曲線形を表示させてみよう.

例題 7.3　　**ex7.3**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

plt.axis('equal')
plt.xlabel('x')
plt.ylabel('y')

t = np.linspace(0, 2*np.pi, 201)
plt.plot(np.cos(t), np.sin(t), '-')
plt.plot(2*np.cos(t), np.sin(t), '--')
plt.plot(np.cos(t), 2*np.sin(t), ':')
```

A `plt.axis('equal')` で, x 軸, y 軸のスケールを一致させている. また, `plt.xlabel`, `plt.xlabel` で x 軸, y 軸, それぞれの変数名を与えている.

B t 座標の範囲を 0 から 2π まで 200 等分している. `t` の実体は配列である.

C 媒介変数表示の x 座標と y 座標, および線種を与えている.

図 32　例題 7.3 (ex7.3) の解説: 経験的確率.

ex7.3 を実行すると次のようなグラフが得られる:

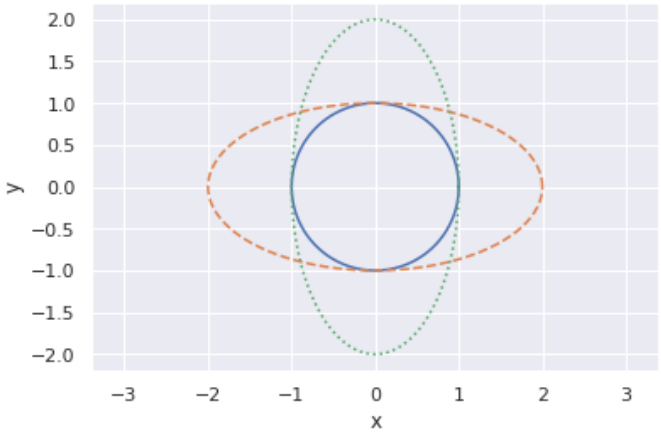

図 33　横軸は x, 縦軸は y を表す. 円と楕円

練習　以下の問に答えよ.

(1) 例題で表示した円は半径 1 の円だった. これを半径 2 の円にするには, どのように修正すれば良いか?

(2) 次の媒介変数表示が表す曲線を表示せよ (ex7.3.1):

$$\begin{cases} x = \cosh(t) \\ y = \sinh(t) \end{cases}$$

この曲線は双曲線と呼ばれ, 関数形は

$$x^2 - y^2 = 1$$

となる. cosh と sinh は, 双曲線関数と呼ばれ, 以下で定義される:

$$\begin{cases} \cosh(t) = \dfrac{e^t + e^{-t}}{2} \\ \sinh(t) = \dfrac{e^t - e^{-t}}{2} \end{cases}$$

cosh と sinh は `np.cosh(...)`, `np.sinh(...)` とすると計算できる.

(3) 次の媒介変数表示が表す曲線を表示せよ (ex7.3.2):

$$\begin{cases} x = 2\cos(t) - \cos(2t) \\ y = 2\sin(t) - \sin(2t) \end{cases}$$

これは, カーディオイド (心臓形) と呼ばれる図形である.

7.4　等高線図

次に $f(x,y)$ で与えられるような「曲面 (2 変数関数)」を, 3 次元プロットしてみよう. ここでは例として

$$f(x,y) = \cos(x)\sin(y) + \sin(xy)$$

という曲面を表示してみる (少し複雑な式なので, Python の関数を用いてプログラムを作成した).

例題 7.4　　**ex7.4**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

def f(x, y):
    a = np.cos(x)*np.sin(y)
    b = np.sin(x*y)
    return (a + b)

X = np.linspace(0, 6, 50)
Y = np.linspace(0, 5, 50)
x, y = np.meshgrid(X, Y)

plt.axis('equal')
plt.contourf(x, y, f(x,y), 15)
plt.colorbar()
```

A　2 変数関数 $f(x,y)$ の定義式が長いので, 「Python の関数」として定義した.

B　`X`, `Y` は x 軸, y 軸上の値 (それぞれ 50 個) を格納した配列である. 関数 `meshgrid` を用いて, 配列 `X`, `Y` から, xy 平面上の点 $50 \times 50 = 2500$ 個を生成している.

C　`contourf` で等高線を書いている (等高線の間に色を付けている). `15` は等高線の本数である. `colorbar` で, 図 35 の右端にあるカラーバーを表示させている.

図 34　例題 7.4 (ex7.4) の解説: 等高線図.

ex7.4 を実行すると次のようなグラフが得られる:

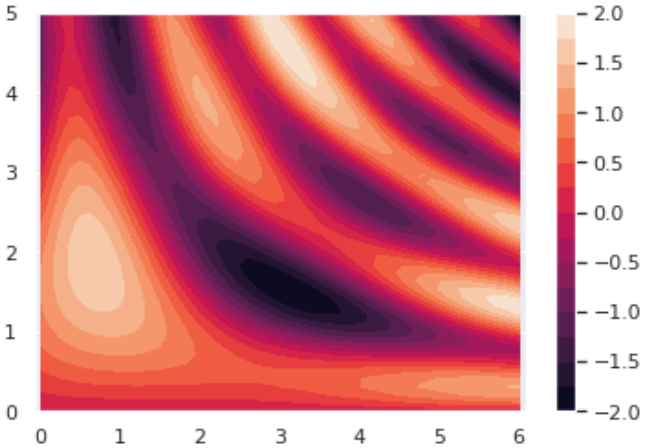

図 35　横軸は x, 縦軸は y を表す. 等高線が描かれ, それらの間が異る色で色付けされている.

練習

1. `contourf` を `contour` に変更するとどうなるか.
2. 次の曲面を (右辺の符号が $+$ の場合と $-$ の場合をそれぞれ) 表示してみよ.

$$f(x, y) = x^2 \pm y^2$$

これらの曲面は符号がプラスの場合「放物面」, マイナスの場合は「双曲放物面」と呼ばれる.

7.5　3 次元表示: 曲面プロット

前節と同様に

$$f(x,y) = \cos(x)\sin(y) + sin(xy)$$

という 2 変数関数について考える. ここでは, これを 3 次元空間内にその曲面を表示してみよう. 3 次元プロットの場合, オブジェクト志向の記法を用いる必要ある. オブジェクト志向の記法は複数の座標軸を含む複雑な図を作成する際に便利である (とはいえこのテキストの範囲内では, オブジェクトやインスタンスについての正確に理解は必要なく, 関数がどのような命令をしているのかを正確に理解するだけで十分である. 興味ある人は巻末の文献を参照されたい).

例題 7.5　　**ex7.5**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from mpl_toolkits import mplot3d

def f(x, y):
    a = np.cos(x)*np.sin(y)
    b = np.sin(x*y)
    return (a + b)

X = np.linspace(0, 6, 50)
Y = np.linspace(0, 5, 50)
x, y = np.meshgrid(X, Y)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.set_xlabel('x')
ax.view_init(60, 30)
ax.plot_surface(x,y,f(x,y),cmap='viridis')
```

A 3 次元プロットでは, オブジェクト志向の記法を用いて表示する. ここで, `fig` と `ax` はインスタンス (クラスオブジェクトの具体物) と呼ばれる. `fig` は (場合によっては複数の) 座標軸や図に関わる全ての情報を含み, `ax` は単一の座標軸 (座標平面や座標空間と言った方が分かりやすい) とそれに関わる全ての情報 (軸のラベルや目盛りなど) を含む.

`plt.figure()` と `plt.axes()` という 2 つの関数でこれらのインスタンスを生成している.

最後の 3 行の関数の行頭は `ax.` がついているが, これは座標軸 `ax` に対する命令であることを明示している.

B `view_init(仰角, 方位角)` は始点の位置を指定している. 仰角は z 軸とのなす角, 方位角は xy 平面への射影が x 軸とのなす角である.

C `plot_surface()` が曲面を表示する関数. `cmap` は色指定.

図 36　例題 7.5 (ex7.5) の解説: 3 次元プロット.

ex7.5 を実行すると次のようなグラフが得られる:

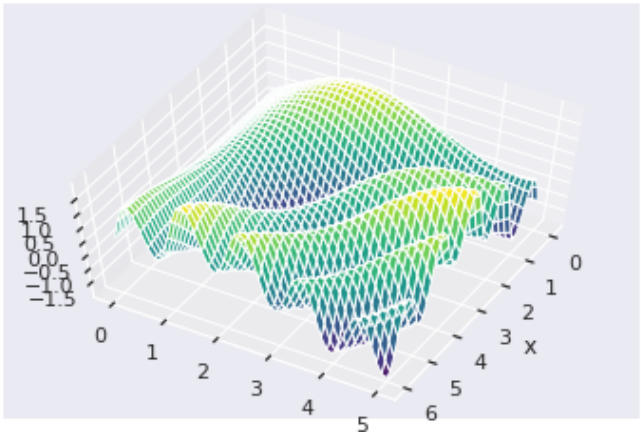

図 37　水平面は xy 平面,
縦軸は z 軸 を表す.

練習　前節と同様に, 次の曲面を (右辺の符号が $+$ の場合と $-$ の場合をそれぞれ) 表示してみよ.

$$f(x,y) = x^2 \pm y^2$$

7.6 3 次元表示: 媒介変数表示

3 次元の場合でも媒介変数表示を用いることができる. 媒介変数表示を用いると, より多様な曲面を扱えることを見てみる. ここでは, トーラスと呼ばれる曲面を描いてみよう. トーラスは以下の式で定義される:

$$x = (R + r\cos v)\cos u,$$
$$y = (R + r\cos v)\sin u,$$
$$z = r\sin v$$

ここで, u と v が媒介変数である (R と r の意味については, それぞれ $u = 0$ のときと, $v = \pi/2$ のときに, 上の式がどのような切断面・接平面を表しているかを考えてみよ).

トーラスを表示するプログラムを Python で作成してみよう.

例題 7.6 **ex7.6**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from mpl_toolkits import mplot3d

u = np.linspace(0, 2*np.pi, 50)
v = np.linspace(0, 2*np.pi, 50)
u, v = np.meshgrid(u, v)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.view_init(50, 30)
ax.set_zlim(-5,5)
ax.set_xlabel('x')
ax.set_ylabel('y')

r, R = 2, 9
x = (R + r * np.cos(v)) * np.cos(u)
y = (R + r * np.cos(v)) * np.sin(u)
z = r * np.sin(v)

ax.plot_surface(x, y, z, cmap='viridis')
```

A オブジェクト志向の記法では, 表示範囲や軸名の指定の仕方も異なる.

これまでは, `plt.xlim(...)`, `plt.xlabel(...)` などを用いていた. 「x 座標の範囲は (...) にせよ」, 「x 座標のラベルは (...) とせよ」と, コンピュータに指示を出すようなイメージである.

それに対し, オブジェクト志向では,「x 座標の範囲は (...)」,「x 座標のラベルは (...)」という風に, `ax` という座標軸 (すなわち, オブジェクト) に性質を付与していくイメージである.

(補足: 実は Python の変数やリストなどは全てオブジェクトなのであるが, このテキストでは "オブジェクトらしさ" をできるだけ見せずに解説してきた. プログラミングの初心者にはイメージしにくいと考えたからである. 興味のある人は巻末の文献などを参照されたい)

B `u`, `v` が 0 から 2π の範囲を動くとき, `x`, `y`, `z` の値をプロットしている.

図 38 例題 7.6 (ex7.6) の解説: 3 次元媒介変数表示.

ex7.6 を実行すると次のようなグラフが得られる:

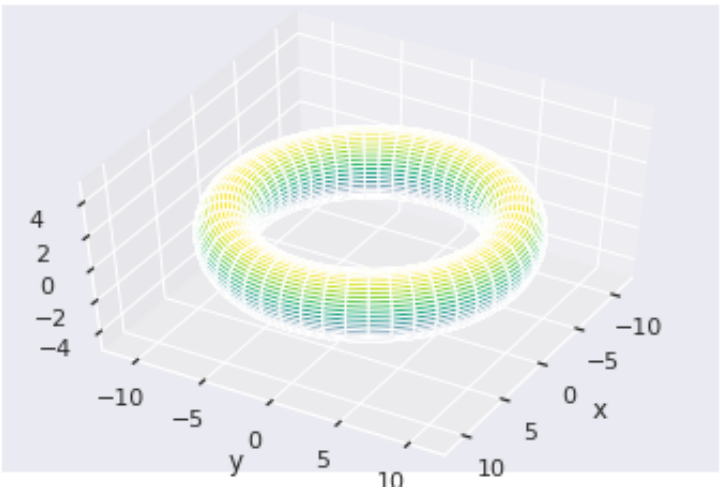

図 39　横軸は試行回数 N, 縦軸は 1 の目の相対度数 r/N を表す. 試行回数が増えるに従い, $1/6 = 0.1666..$ に近づいていくように見えなくもないが, 少し分かりにくい.

練習　次の問に答えよ.

(1) R や r の値を変えて表示させてみよ.

(2) ex7.6 の中の

```
u = np.linspace(0, 2*np.pi, 50)
```

を

```
u = np.linspace(0, 1.6*np.pi, 50)
```

としてみよ. なぜ, そのような結果が得られたのか説明せよ.

(3) 次の曲面を (右辺の符号が $+$ の場合と $-$ の場合をそれぞれ) 表示してみよ.

$$f(x, y) = x^2 \pm y^2$$

これらの曲面は符号がプラスの場合「放物面」, マイナスの場合は「双曲放物面」と呼ばれる.

7.7 演習問題

課題 7.1 (基本) $y = \sin(x) + \sin(3x)/3 + \sin(5x)/5$ をグラフにせよ (pr.7.1 とする)[*24].

課題 7.2 (やや難) 次の関数のグラフを表示せよ (pr7.2 とする).

$$y = \sum_{j=0}^{100} \frac{\sin((2j+1)x)}{2j+1} \qquad (-10 < x < 10)$$

(ヒント: pr7.1 を修正するのが良い. pr7.1 では 3 つの sin 関数の足し算だった部分を for 文として書き直せば OK)

類題 (難) 次の関数のグラフを表示せよ (pr7.2.1 とする).

$$y = x \prod_{j=1}^{100} \left(1 - \frac{x^2}{j^2\pi^2}\right) \qquad (-4\pi < x < 4\pi)$$

ここで, 記号 $\prod_{j=1}^{n} a_j$ はかけ算を表わしている: $\prod_{j=1}^{n} a_j = a_1 \times a_2 \times \cdots \times a_n$.

(ヒント: 関数 $y = y(x)$ は、積の数 100 を無限大にすると、$y = \sin(x)$ に収束する)

類題 (難) 次の関数のグラフを表示せよ (pr7.2.2 とする).

$$y = \sum_{j=0}^{12} \frac{x^j}{j!} \qquad (0 < x < 4)$$

課題 7.3 (基本) 次のようなグラフを出力するプログラムを作成せよ (pr.7.3).

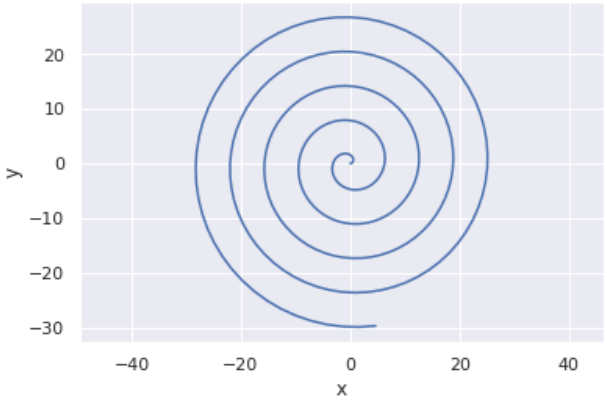

図 40 媒介変数表示を用いて渦を表示した. 逆回りの渦はどうすれば出力できるだろうか?

[*24] 項数をさらに増やした関数, すなわち, $f(x) = \sin(x) + \sin(3x)/3 + \sin(5x)/5 + \sin(7x)/7 + \cdots$ はどのような関数に近付くだろうか? このような, sin や cos を使った展開はフーリエ級数展開と呼ばれ, (連続な関数についてのみ使えるテーラー展開と違い) 不連続な関数でも展開できる. このため, 応用範囲が極めて広い. 次の課題も参照.

第 II 部
数学への応用

第 I 部では, Python プログラミングの文法の基本を学んだ. これからは主として数学の問題をテーマとして扱う.

8　整数と最大公約数　(応用編 1)

今週は、整数に関連する話題を扱う。特に、整数についての割り算 (商・余り) と最大公約数を求めるプログラムを作成する。

8.1　商と余りの計算

既に割り算については何度も出てきたが, 話を整数の範囲に留めるならば, 商や余りと呼ばれる量が必要になる. 例えば,

$$27 \div 7 = 3 \text{ 余り } 6$$

のような式の, 商 (上の式の ”3”) や余り (上の式の ”6”) を求めてみよう.

例題 8.1　　**ex8.1**

```
print('割り算', 27 / 7)
print('商', 27 // 7)
print('余り', 27 % 7)
```

Ⓐ `27 // 7` は「27 を 7 で割ったときの商の値」となり, この場合 `3` という **整数値** が得られる. **小数点以下が切り捨て** になることに注意.

Ⓑ `27 % 7` は「27 を 7 で割ったときの余りの値」となり, この場合 `6` という **整数値** が得られる.

図 41　例題 8.1 (ex8.1) の解説: 商と余りの計算

ex8.1 を実行すると以下のようになる:

実行結果

```
割り算 3.857142857142857
商 3
余り 6
```

練習　上のプログラム ex8.1 について, 桁を揃えて出力するにはどのように改良すれば良いか (ex8.1.1).

8.2　最大公約数とユークリッドの互除法

最大公約数 (great common divisor) を求める手法として, ユークリッドの互除法というものがある. 厳密な証明は, 代数学の授業に譲り, ここでは図形を用いて簡単に解説を行う. 例として, 22 と 8 の最大公約数を知りたいとする. このとき, 図 42 のような, 辺の長さが 22 と 8 で与えられる長方形を考える. この長方形を, できるだけ少ない数の正方形で被覆することを考える. まずは, 一辺の長さが 8 の正方形ふたつが入る (正方形 a と b). 次に, 一辺の長さが 6 の正方形がひとつ入る (正方形 c). 最後に, 大きさ 2 の正方形が 3 つ (正方形 d, e, f) 入り, 長方形を完全に被覆する. このとき, 一番小さい正方形の一辺の長さが, 最小公約数になる.

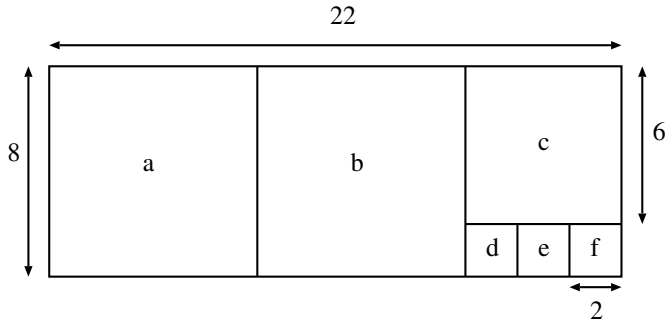

図 42　ユークリッドの互除法の概念図. (ここでは, 22 と 8 の最大公約数を求めている)

この過程を, もう少し詳しく見てみる. 辺の長さが 8, 22 の長方形を $(22, 8)$ のように表すことにする (長い方の辺の値を最初に書く). このとき, 上の図形の計算は,

(1) 長方形 $(22, 8)$ について, $22 \div 8 = 2$ 余り 6

(2) 長方形 $(8, 6)$ について, $8 \div 6 = 1$ 余り 2

(3) 長方形 $(6, 2)$ について, $6 \div 2 = 3$ 余り 0

のようなものであることが分かる. つまり, n_1 と n_2 $(n_2 > n_1)$ というふたつの最大公約数を計算する場合,

$$n_2 \div n_1 = q \text{ 余り } r$$

という割算をし, 長辺の長さ n_2 を余り r で置き換える, という処理を余りが 0 になるまで繰り返しているわけである[*25]。

このような処理をプログラムで書くと次のようになる. ここで, 繰り返し処理には for 文ではなく, while 文を用いていることに注意しよう. 「互除法の余りを求める計算」を何回行えば良いか, はあらかじめ分からない. したがって, 処理回数を最初に指定する必要がある For 文は使いにくいのである[*26].

[*25] もう少し正確に言うと、整数 n_1 と n_2 の最大公約数を $\gcd(n_1, n_2)$ と表すとすると、$n_1 < n_2$ のとき、

$$\gcd(n_1, n_2) = \gcd(n_1, r)$$

となることが一般的に示される. ちなみに, gcd は greatest common divisor (最大公約数) の頭文字.

[*26] 第 9 章で紹介する break 文を用いれば, for 文を使って書くことも可能だが, やや煩雑なプログラムになってしまう.

例題 8.2　　**ex8.2**

```
n1 = 378
n2 = 234

if n1 > n2:

    nt = n1
    n1 = n2
    n2 = nt

while n2 % n1 != 0:

    nt = n1
    n1 = n2 % n1
    n2 = nt

    print (n1, end=' ')

print ('\nGCD:', n1)
```

A　2 つの数 n1 と n2 の最大公約数を求める.

B　n1 > n2 であれば, 値を入れ替える. つまり, n1 が小さい値になるようにする.

C　n1 = 0 となると (すなわち, 余りが 0 になると), Do 文の繰り返し処理を終了する. n1 > 0 であれば, 処理を繰り返す.

D　n2 % n1 は, n2 を n1 で割った余りを意味する. 例えば, 378 % 234 = 144 である. ここでは, 余りの値 n2 % n1 を新しい n1 の値として再設定している.

図 43　例題 8.2 (ex8.2) の解説: 最大公約数の計算

ex8.2 を実行すると以下のようになる:

実行結果

```
144 90 54 36 18
GCD: 18
```

最後の数字 "18" が最大公約数である.

練習　上のプログラム ex8.2 について, 次の問に答えよ.

(1) n1 = 377 及び n2 = 233 として計算してみよ (n1 = 2584 及び n2 = 4181 なども試してみよ). 実行したときに得られた数列はなんと呼ばれる数列か?

(2) ex8.2 は (空行を除いて) 13 行のプログラムである. これを同じ処理をする 8 行のプログラムに書き直せ (ex8.2.1 とする; ヒント: 脚注 *7).

8.3　3 つの数の最大公約数 (関数の利用)

n_1、n_2、n_3 を 3 つの正整数とし, $\gcd(n_1, n_2, n_3)$ をそれらの最大公約数とする. さらに, $\gcd(n_1, n_2)$ を n_1 と n_2 の最大公約数とする. このとき

$$\gcd(n_1, n_2, n_3) = \gcd(\gcd(n_1, n_2), n_3)$$

が成り立つ. つまり, 上式式の右辺を用いて, まず

$$n_1 \text{ と } n_2 \text{ の最大公約数 } \gcd(n_1, n_2)$$

を求め, 次に

$$\gcd(n_1, n_2) \text{ と } n_3 \text{ との最大公約数 } \gcd(\gcd(n_1, n_2), n_3)$$

を求める, というふうに段階的に計算することで, 3 整数の最大公約数 $\gcd(n_1, n_2, n_3)$ が求められる.

この計算では, 2 回最大公約数を求める計算をしている. このように複数回同様の処理をする場合, for 文でまとめるのが基本だが, すでに 2 数の最大公約数でも複雑なプログラムになっているので, これを for 文でまとめるとさらに階層が 1 つ増えて読みにくいプログラムとなってしまうだろう (これを, 「可読性が悪い」という). このような場合には関数を用いるとプログラムが簡潔にまとまる. 実際に関数を用いたプログラムを見てみよう.

例題 8.3　　**ex8.3**

```
def gcd(m1, m2):

    if m1 > m2:
        m1, m2 = m2, m1

    i = 1
    while m2 % m1 != 0:

        m1, m2 = m2 % m1, m1
        i = i+1

    return(m1)

n1, n2, n3 = 378, 234, 336
print('GCD =', gcd(gcd(n1, n2), n3))
```

A　2 つの数 n1 と n2 の最大公約数を求める関数. この関数の中身は ex8.2.1 の内容と同じである.

B　最後に m1 の値 (最大公約数) を戻り値とする.

C　gcd(n1, n2) は n1 と n2 の最大公約数を求め, さらにこれと n3 との最大公約数を求めている.

図 44　例題 8.3 (ex8.3) の解説: 最大公約数の計算

ex8.3 を実行すると以下のようになる:

実行結果

```
GCD = 6
```

練習　上のプログラム ex8.3 について, 次の問に答えよ.

(1) `n1 = 377` 及び `n2 = 233` として計算してみよ (`n1 = 2584` 及び `n2 = 4181` なども試してみよ). 実行したときに得られた数列はなんと呼ばれる数列か?

(2) ex8.3 は (空行を除いて) 13 行のプログラムである. これを同じ処理をする 8 行のプログラムに書き直せ (ex8.3.1 とする; ヒント: 脚注 ∗7).

8.4　演習問題

課題 8.1 (基本) 西暦の年月日からその日が何曜日であるかを算出するプログラム (pr6.1 とする) を作成しよう. まず考えている年の最初の 2 桁を J (例えば 2012 年ならば $J = 20$), 残りの 2 桁を K (例えば 2012 年ならば $K = 12$) とする. さらに, 月を m, 日を d とする (例えば, 5 月 23 日ならば, $m = 5$, $d = 23$). ただし月が 1 月, 2 月の場合はそれぞれ前年の 13 月, 14 月とみなす (例えば, 2012 年 1 月 1 日なら 2011 年 13 月 1 日とみなす). このとき

$$h = \left(d + \left\lfloor \frac{(m+1)26}{10} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J \right)$$

とおく. ここで, $\left\lfloor \frac{n_1}{n_2} \right\rfloor$ という記号は n_1 を n_2 で割ったときの商を表わす. 最後に, 上の式で求めた h を 7 で割った余りの値 (0 〜 6) がその日の曜日を表わしている:

$0 \Longrightarrow$ 土曜日　　$1 \Longrightarrow$ 日曜日
$2 \Longrightarrow$ 月曜日　　$3 \Longrightarrow$ 火曜日
$4 \Longrightarrow$ 水曜日　　$5 \Longrightarrow$ 木曜日
$6 \Longrightarrow$ 金曜日

これは, ツェラーの公式と呼ばれている.

課題 8.2 (基本) 2 つの正の整数が与えられたとき、それらの最小公倍数 (least common multiple) を求めるプログラム pr6.2 を作成せよ。

9　素数　(応用編 2)

今週は、素数に関するプログラムを作成する.

9.1　素数の判定

今度は, ある正の整数が与えられたときに, それが素数 (prime number) か否かを判定するプログラムを作ってみよう. ある正の整数 n が与えられたとき, n が素数であるか否かは, n より小さい正の整数で割り切れるかどうかを調べればよい.

例題 9.1　　**ex9.1**

```
n = 97
check = 1

for i in range(2, n):
    if n % i == 0:
        check = 0
        break

if check == 1:
    print(f'{n}は素数である')
else:
    print(f'{n}は素数でない')
```

A　n に代入した数が素数か否かを判定する.

B　n より小さい正の整数 i で, n が割り切れるかどうかを判定する. そのために, for 文で i の値を変えながら繰り返し処理を行っている.

C　n が i で割り切れたとき (すなわち, n % i == 0 であるとき), check という変数の値を 0 に変える.
　さらに, break 文を用いて, for 文の繰り返し処理を強制終了させている (あるひとつの整数 i で割り切れれば, それ以上チェックする必要がないので).

D　n を割り切る正整数 i < n が無かった場合 (つまり, n が素数である場合), check = 1 (初期値から変化無し) となっている. したがって, この if 節の処理内容が実行される.
　一方, n を割り切る正整数 i < n が存在した場合 (すなわち, 素数ではない場合), check = 0 となっているはずなので, else 節の処理内容が実行される.

図 45　例題 9.1 (ex9.1) の解説: 素数の判定

ex9.1 を実行すると以下のようになる:

実行結果

```
97 は素数である
```

練習 以下の問に答えよ.

(1) `n` が素数でなかった場合に, `n` の素因数をひとつ出力するように例題のプログラム ex9.1 を修正せよ (ex9.1.1 とする).

(2) 上のプログラムでは, $n = 0$ や $n = 1$ を素数と判定してしまう. この問題点を解決するにはどのような方法が考えられるか? ex9.1 を元に修正せよ (ex9.1.2 とする; 素因数を出力しなくても良い)

(3) 4 桁の素数をひとつ見つけよ [ヒント: $p_1, p_2, p_3, \cdots, p_n$ を互いに異る素数であるとする (素数であれば, どんな数でも良いし, 順番もなんでも良い)]. このとき,

$$p_1 \times p_2 \times \cdots \times p_n + 1$$

は $p_i\,(i = 1, \cdots, n)$ 以外の素数であるか, あるいは $p_i\,(i = 1, \cdots, n)$ 以外の素数で素因数分解される [何故か考えよ]. 実際に手計算で 4 桁の素数を見つけ, それが実際に素数であることを, プログラム ex9.1(またはその修正版) を使ってチェックせよ.

9.2　素数の判定: プログラムの改良

例題 9.1 では調べたい自然数 n より, 小さい全ての自然数 $i < n$ について因数ではないことをチェックしていた. しかしこの計算は非常に無駄が多く, n が大きな値の場合には, とても計算量が多くなってしまう. プログラムはできるだけ計算量を少なくする方が望ましいが, 何か良い方法はあるだろうか? 次の例題で示すように, 実は $i \leq \sqrt{n}$ となる i についてだけ調べれば十分なのである.

例題 9.2　　**ex9.2**

```
import numpy as np

n = 97
m = int(np.sqrt(n))

check = 1

for i in range(2, m+1):

    if n % i == 0:
        check = 0
        break

if check == 1:
    print(f'{n}は素数である')
else:
    print(f'{n}は素数でない ({i}で割り切れる)')
```

A 平方根の値を与える関数 `np.sqrt(...)` を用いるために, numpy ライブラリを読込む.

B `np.sqrt(n)` は一般に非整数になる. そこで, `int(...)` を用いて, その整数部分に直している.

C 調べたい数が `n = 97` の場合, 2 から $9 \leqq \sqrt{97}$ までの自然数で割り切れないことをチェックする.

図 46　例題 9.2 (ex9.2) の解説: 素数の判定 (改良版)

ex9.2 を実行すると以下のようになる:

実行結果

```
97 は素数である
```

練習

(1) 自然数 n が素数か否かを判定するとき, $2 \leqq i \leqq \sqrt{n}$ となる全ての自然数 i が, n の約数でないことを調べれば十分である理由を考えよ.

(2) ex9.2 では `n = 1` や負の整数を素数と判定してしまう. これらを素数ではないと判定するにはどうのようにすれば良いか. p

9.3 演習問題

課題 9.1 (基本) 2 以上 100 以下の全ての素数を出力するプログラム pr9.1 を作成せよ. for 文を 2 重にする必要がある (ex9.2 を元にして作成しよう).

類題 (難) ある自然数 n が素数か否かを判定する場合, $\sqrt{n}$ 以下の素数 iで割り切れないことを示せば十分である (pr9.1 では, $\sqrt{n}$ 以下の自然数 iに対して調べていたはずである). この事実に基いて, pr9.1 を改良し, やはり 2 以上 100 以下の全ての素数を出力するプログラム pr9.1.1 を作成せよ。
ヒント: 次のように, リストを定義する:

```
prime = [0] * 50
prime[0] = 2
```

すなわち, 50 個の要素からなるリスト `prime` を作成し[*27], 最初の素数 "2" だけ, 予めリストの要素として与えておく.

課題 9.2 (難) 整数の桁を並べ替えて, 最大にしたものから最小にしたものの差を取る. この操作によって元の値に等しくなる数をカプレカ数と呼ぶ. 例えば, 495 の場合

$$954 - 459 = 495$$

となるからカプレカ数である. プログラム (pr7.2) を用いて, 4 桁のカプレカ数を全て求めよ.

課題 9.3 (難) 『2 よりも大きな偶数は 2 つの素数の和として表すことができる』ことをゴールドバッハ予想という[*28]. 与えられた偶数を 2 つの素数の和で表すプログラム pr9.3 を作成せよ. ただし, 2 つの素数による表し方は 1 通りではない (例えば, $14 = 3 + 11 = 7 + 7$). プログラムでは 1 つだけ表し方を見つければ良い.

[*27] 要素は 50 個にしたのは, 1 以外の奇数 49 個と唯一の偶数の素数 2 を合わせて 50 個なので. 実際にはこれより少ないけれど, 50 個を超えることはないのでこのようにした. `append` というメソッドを用いると, リストに要素を追加できるので, 最初に大きなリストを用意する必要が無くなる (普通はそうします). これは,

```
prime.append(n)
```

のようにする. すると, `n` に代入されている数値が, リスト `prime` に追加される.

[*28] 未解決問題. 2012 年の時点で 4×10^{18} まで成立することがコンピュータで確かめられている.

10　方程式の根　(応用編 3)

$x^2 - a = 0$ や $ax^3 + bx^2 + cx + d = 0$ のように, x のべき乗 x^n で表される方程式を代数方程式と呼ぶ. 一方, 方程式に $\sin x$ や $\tan x$, $\log x$ などの関数を含む場合には超越方程式と呼ばれる. この章では, これらの方程式を数値計算を用いて解いてみよう.

10.1　平方根の計算

電卓で「$\sqrt{2}$」 を計算すれば, 「1.41421356...」という数値が得られる. このような計算はいったいどのようにして実現されているのだろうか? ここでは, 平方根の値を求めるアルゴリズムとして, 「バビロニアの平方根 (紀元前 1800-1600 年頃)」を紹介する[*29]. これは,

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{6}$$

という漸化式を用いる方法である. ただし, x_n の初期値 x_0 の値は正でなければいけない (正であれば, どんな値でも良い). n を大きくしていくと, x_n の値は $\sqrt{a}$ に近付いていく. では, この方法を用いたプログラムを作成してみよう.

例題 10.1　　**ex10.1**

```
a = 2
x = 5

for i in range(1, 11):

    x = 0.5 * (x + a / x)
    print(x)
```

解説

1. ここでは, $\sqrt{a} = \sqrt{2}$ の値を求めている.

2. x の初期値は正であればどのような数値を設定しても構わない.

図 47　例題 10.1 (ex10.1) の解説: 平方根の計算

[*29] 平方根 $\sqrt{a}$ の値を求めることは、代数方程式

$$x^2 - a = 0$$

の正の解 (すなわち根) を求めることと同じであることに注意. 次節以降で, $f(x) = 0$ という形の方程式を解くための一般的な手法を解説する.

ex10.1 を実行すると以下のようになる:

実行結果

```
2.7
1.7203703703703703
1.44145536817765
1.4144709813677712
1.4142135857968836
1.4142135623730954
1.414213562373095
1.414213562373095
1.414213562373095
1.414213562373095
```

練習

1. 上のプログラム ex10.1 について, `a` や `x` の初期値をいろいろと変えて計算してみよ.
2. なぜ, 式 (6) で平方根が計算できるのか考えよ. ヒントは「面積が a の長方形」と「その長方形の 2 辺の長さの平均値」.
3. 立方根の計算も同様にできるだろうか? 2. の答えが分かったら, それを元に予想してみよ (答えは, 後述のニュートン法を用いると分かる. また, 課題 8.2 でプログラムを作成する).

10.2　ニュートン法

より一般的な代数方程式・超越方程式の場合でも計算ができるアルゴリズムを考えてみよう．ここでは, ニュートン法と呼ばれる方法を紹介する．(次節で扱う 2 分法に比べて) ニュートン法は根を得るまでの計算量が少なくなることが知られている[*30]．

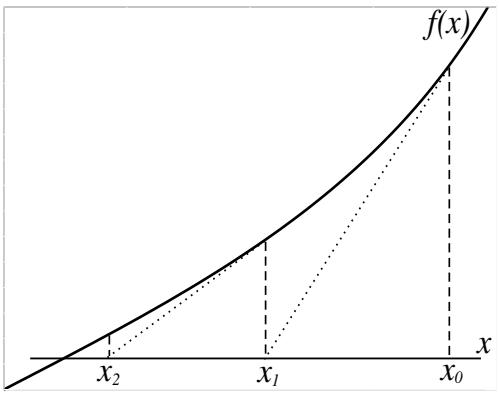

図 48　ニュートン法の解説図

ここでは, ニュートン法の考え方を図 48 を用いて説明をする．まず, 任意の点 $x = x_0$ を選ぶ (点 x_0 のことを初期点と呼ぶ)．この点 $(x_0, f(x_0))$ で関数 $f(x)$ に接する接線を考えよう．接線の方程式は,

$$y = f'(x_0)(x - x_0) + f(x_0)$$

となる．次に, この接線が x 軸と交わる点を x_1 とおくと, x_1 は上式で $y = 0$ として得られる．すなわち,

$$0 = f'(x_0)(x_1 - x_0) + f(x_0)$$

これを x_1 について解くと,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{7}$$

という式が得られる．同様に, 点 $(x_1, f(x_1))$ における接線を考えれば, x_2 が決まる．このような処理を続けていけば, x_n $(n = 0, 1, 2, \cdots)$ という数列は, $f(x) = 0$ の根に近付いていくことが, 図 48 からも分かるだろう．この数列は式 (7) を一般化した漸化式

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{8}$$

によって決められる．

具体例として, $f(x) = x^2 - a$ とすれば, これは a の平方根 $\sqrt{a}$ を求める問題となる．実際, 上の漸化式 (8) に $f(x) = x^2 - a$ を代入すれば,

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

となり, 式 (6) に一致する．例題 10.1 (ex10.1) は, 実はニュートン法の例であったのである．

[*30] その代わり, x の初期点が根から離れている場合などに不安定な振る舞いをすることがある．一方, 次節で紹介する 2 分法は, 最初に与える区間 $[a, b]$ 内に解があれば, 必ず収束するため, より安定である (ただし, ニュートン方より遅い)．

10.3　2 分法

ここでは例として, 超越方程式 $\tan x - 1 = 0$ の解 $\frac{\pi}{4}$ の値を求めてみよう. $f(x) = \tan x - 1$ とすると,

$$f(0) = -1 < 0$$

$$f(1) = \tan 1 - 1 = 1.557 - 1 > 0$$

なので, $f(x) = 0$ の解は 0 と 1 の間にあることが分かる. そこで, 変数 `x_min` と `x_max` を

`x_min`: 解の存在範囲の下限

`x_max`: 解の存在範囲の上限

と定義する. また, これらの初期値を `x_min`=0, `x_max`=1 とし, `x_min` と `x_max` の中点を `x_mid` とする:

$$x_{\mathrm{mid}} = \frac{x_{\max} + x_{\min}}{2}.$$

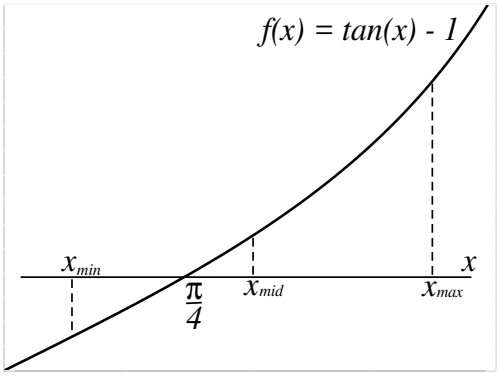

図 49　この場合 ($f(x_{\mathrm{mid}}) > 0$) は, `x_max` の値を `x_mid` に置き換える. `x_min` の値はそのまま.

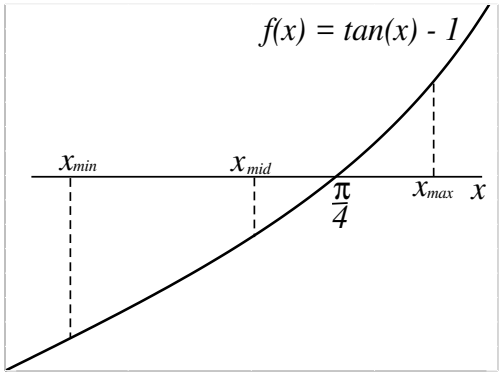

図 50　この場合 ($f(x_{\mathrm{mid}}) < 0$) は, `x_min` の値を `x_mid` に置き換える. `x_max` の値はそのまま.

次のような条件分岐を用いた処理を考えよう (当然 `if` 文等を用いる):

(1) もし左側の図 ($f(x_{\mathrm{mid}}) > 0$) のようになっていれば, $f(x) = 0$ の解は `x_min` と `x_mid` の間にあるので, `x_max` を `x_mid` で置き換え, 一方 `x_min` の方はそのままにする.

(2) 逆に右側の図 ($f(x_{\mathrm{mid}}) < 0$) のようになっていれば, $f(x) = 0$ の解は `x_mid` と `x_max` の間にあるので, `x_min` を `x_mid` で置き換え, `x_max` はそのままにする.

この処理を 1 回するたびに解の存在範囲 (つまり, `x_min` から `x_max` までが解の存在範囲) が半分になるので, この方法は二分法 (bisection method) と呼ばれる. この処理を繰り返し行うプログラムを以下に示す:

例題 10.2　　**ex10.2**

```
import numpy as np

xmin = 0
xmax = 1

for i in range(1, 20):

    xmid = (xmax + xmin) / 2

    if np.tan(xmid) - 1 > 0:

        xmax = xmid

    else:

        xmin = xmid

    print(f'{xmin:9.7f}, {xmax:9.7f}')
```

A 解の存在範囲の初期値を設定. この後の for 文で, この範囲を繰返し狭めていく.

B 中点の値 `xmid` を計算.

C $f(x_{\mathrm{mid}}) > 0$ ならば, x_{max} の値を x_{mid} に再設定する (図 49).

D $f(x_{\mathrm{mid}}) < 0$ ならば, x_{min} の値を x_{mid} に再設定する (図 50).

図 51　例題 10.2 (ex10.2) の解説: 二分法による π の計算

ex10.2 を実行すると以下のようになる:

実行結果

```
0.5000000, 1.0000000
0.7500000, 1.0000000
0.7500000, 0.8750000
0.7500000, 0.8125000
0.7812500, 0.8125000
0.7812500, 0.7968750
0.7812500, 0.7890625
0.7851562, 0.7890625
0.7851562, 0.7871094
0.7851562, 0.7861328
0.7851562, 0.7856445
0.7851562, 0.7854004
0.7852783, 0.7854004
0.7853394, 0.7854004
0.7853699, 0.7854004
0.7853851, 0.7854004
0.7853928, 0.7854004
0.7853966, 0.7854004
0.7853966, 0.7853985
```

10.4　演習問題

課題 10.1 (基本)　立方根 $a^{\frac{1}{3}}$ を求める漸化式をニュートン法に従って (紙と鉛筆で) 導き, さらにそのプログラム pr10.1 を作成せよ.

課題 10.2 (基本)　$x^2e^{-x}-1=0$ の解を 2 分法を用いて求めるプログラム pr10.2 を作成せよ. 最初に与えるの根の範囲 `xmid` と `xmax`, 及び if 文の条件に注意. Python では指数関を用いるために, `numpy` モジュールを読み込む必要がある:

```
import numpy as np
```

その上で,

```
import np.exp(...)
```

とすることで指数関数の値が得られる.

ヒント: まず, 増減表を作成して, グラフの概形と根のおおよその位置を把握しよう. もし, 関数形の見当がつかない場合は, $f(x)=x^2e^{-x}-1$ の値を出力するプログラムを作成し, 数値を出してみると良い. さらに, その数値をグラフに表示させてみよう.

例題 10.2.1　**pr10.2.1**

```
import matplotlib.pyplot as plt
import numpy as np

# -1 から 5 まで、100 個の点を表示する
x = np.linspace(-1, 5, 100)
plt.plot(x, x*x*np.exp(-x)-1)
```

11　行列と連立 1 次方程式　(応用編 4)

今週は, 線形代数に関する問題を扱う. 簡単のため, 3×3 行列のみを扱うことにする.

11.1　行列の足し算

まずは次式で与えられる, 行列の足し算を考える:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{pmatrix}$$

第 5 章で, ベクトルを扱うとき, リスト `x[i]` という変数を用いた. これを拡張した, 多重リスト `a[i][j]` という変数を用いると, 行列の計算を容易に行うことができる. 例題 11.2.1 (ex11.2.1) はふたつの 3×3 行列の足し算を行うプログラムである.

例題 11.1　　**ex11.1**

```
a = [[ 5, -2,  1],
     [ 2,  1,  3],
     [ 3,  2, -4]]

b = [[ 3,  4,  3],
     [-1, -2,  2],
     [ 1,  2, -3]]

for i in range(0, 3):
    for j in range(0, 3):
        c = a[i][j] + b[i][j]
        print (f'{c:2d}', end=' ')
    print('')
```

Ⓐ 3×3 行列 A と B の要素 `a[i][j]` と `b[i][j]` を定義している.
　この例のように定義した場合, 実際に変数として使えるのは, `a[0][0]` から `a[2][2]` までの 9 個である (これは, 第 5 章で紹介したリストと同じルールである). したがって, 行列要素 a_{ij} の添字 ij リストの要素番号 `[i][j]` は 1 つずれることに注意.

Ⓑ 行列の要素 `[i][j]` 毎に足し算をするため, 2 重の for 文を使う.

Ⓒ 行列の要素を足し合わせた値を出力.

図 52　例題 11.1(ex11.1) の解説: 行列の足し算

ex11.1 を実行すると以下のようになる:

実行結果

```
 8  2  4
 1 -1  5
 4  4 -7
```

練習

1. 上のプログラム ex11.1 について, 2 つ目の `print` 文が無い場合はどのような出力が得られるか予想せよ. また, 実際に実行して確認してみよ.
2. 行列の計算については, ライブラリ numpy が非常に便利である. 以下の 2 行を最後に追加して実行してみよ.

追加部分

```
import numpy as np
print (np.array(a) + np.array(b))
```

上のプログラムの意味を簡単に説明しておこう. リスト `a` を, 関数 `np.array()` に渡すことで, numpy の配列 (ndarray) に変換される. この配列はリストと同じようなものだが, 上の追加部分のように足し算をすると, 「要素毎に足し算」が行われる. numpy を用いると, このようなベクトル演算を通して, 極めて簡潔にプログラムが書けるようになる[*31].

[*31] 足し算の場合は数学的に自然な演算になっているが, 例えば掛け算

np.array(a) * np.array(b)

とした場合にも要素毎に掛け算がなされる. これは自然な行列の積とは異なることに注意されたい (自然な行列の積は `*` の代わりに `@` を用いると計算できる). このように numpy は大変便利なのだが, 数学的には不自然な記述となることが多い. そこで混乱を避けるため, このテキストでは, numpy のベクトル演算はできるだけ使用しないようにした (関数のグラフを表示する際に使用しているのみである). 数学の知識に不安が無く混乱する恐れが無い人は, ぜひ numpy のベクトル演算に挑戦してみて欲しい.

11.2　連立 1 次方程式 (第 1 ステップ)

連立 1 次方程式は中学校で習う初等的な問題であるにも関わらず, 応用上極めて重要であり, 様々な数値シミュレーション手法が開発されている. 実用的なプログラムは難易度が高いので, この授業では特別な場合に連立 1 次方程式を解く (比較的) 簡単なプログラムを作成する. 特に, 変数が 3 つの連立 1 次方程式を考える (以下では, x, y, z が変数, a_{ij} と b_i は定数である):

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1 \quad \cdots \ ① \\ a_{21}x + a_{22}y + a_{23}z &= b_2 \quad \cdots \ ② \\ a_{31}x + a_{32}y + a_{33}z &= b_3 \quad \cdots \ ③ \end{aligned}$$

これを解くためには, まず 3 つの式の最初の係数が 1 になるように, 両辺を a_{i1} で割り算をする:

$$\begin{aligned} ① \div a_{11} \quad &\Rightarrow \quad x + \frac{a_{12}}{a_{11}}y + \frac{a_{13}}{a_{11}}z = \frac{b_1}{a_{11}} \quad \cdots \ ①' \\ ② \div a_{21} \quad &\Rightarrow \quad x + \frac{a_{22}}{a_{21}}y + \frac{a_{23}}{a_{21}}z = \frac{b_2}{a_{21}} \quad \cdots \ ②' \\ ③ \div a_{31} \quad &\Rightarrow \quad x + \frac{a_{32}}{a_{31}}y + \frac{a_{33}}{a_{31}}z = \frac{b_3}{a_{31}} \quad \cdots \ ③' \end{aligned} \tag{9}$$

この時点で, $a_{i1} \neq 0$ $(i = 1, 2, 3)$ を仮定していることに注意[*32]. さらに,

$$\begin{aligned} ①' \quad &\Rightarrow \quad x + \frac{a_{12}}{a_{11}}y + \frac{a_{13}}{a_{11}}z = \frac{b_1}{a_{11}} \quad \cdots \ ①'' \\ ②' - ①' \quad &\Rightarrow \quad \left(\frac{a_{22}}{a_{21}} - \frac{a_{12}}{a_{11}}\right)y + \left(\frac{a_{23}}{a_{21}} - \frac{a_{13}}{a_{11}}\right)z = \frac{b_2}{a_{21}} - \frac{b_1}{a_{11}} \quad \cdots \ ②'' \\ ③' - ①' \quad &\Rightarrow \quad \left(\frac{a_{32}}{a_{31}} - \frac{a_{12}}{a_{11}}\right)y + \left(\frac{a_{33}}{a_{31}} - \frac{a_{13}}{a_{11}}\right)z = \frac{b_3}{a_{31}} - \frac{b_1}{a_{11}} \quad \cdots \ ③'' \end{aligned}$$

複雑になったので, 係数を別の記号で書き直してみると, 次のような形になる:

$$\begin{aligned} x + a'_{12}y + a'_{13}z &= b'_1 \quad \cdots \ ①'' \\ a'_{22}y + a'_{23}z &= b'_2 \quad \cdots \ ②'' \\ a'_{32}y + a'_{33}z &= b'_3 \quad \cdots \ ③'' \end{aligned} \tag{10}$$

まずはここまでのステップをプログラムに書いてみよう. そのために, 連立 1 次方程式は行列とベクトルを用いて以下のように書くことできることに注意する:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

[*32] 実用的なプログラムにするには, $a_{i1} = 0$ となるような場合でも処理できるようにプログラムを作成する必要がある.

そこで, 上記の計算を配列を用いて表すのが良さそうである. 実際に作成したプログラムを見てみよう.

例題 11.2　　**ex11.2**

```
a = [[ 5, -2,  1],
     [ 2,  1,  3],
     [ 3,  2, -4]]

b = [13, -7, 23]

for i in range(0, 3):
    ta = a[i][0]
    for j in range(0, 3):
        a[i][j] = a[i][j] / ta
    b[i] = b[i] / ta

for i in range(1, 3):
    for j in range(0, 3):
        a[i][j] = a[i][j] - a[0][j]
    b[i] = b[i] - b[0]

print("---a'ij---")
for i in range(0, 3):
    for j in range(0, 3):
        print (f'{a[i][j]:6.3f}', end=' ')
    print('')

print("---b'i---")
for i in range(0, 3):
    print (f'{b[i]:6.3f}')
```

A　3 × 3 行列 A の要素 a[i][j] を初期化

B　ベクトル b(i) を初期化

C　行列の各要素 a(i,j) を, その行の先頭要素 ta = a(i,1) で割る.
b(i) についても同様に, その行の先頭要素 ta = a(i,1) で割る.
この 2 重の for 文が終了した時点で, 本文の式 (9) のような形になっている.

D　2 行目以降 ($i \geq 2$) の各要素 a(i,j) から, その列の先頭要素 a(1,j) を引く.
b[i] についても同様に, 列の先頭要素 b(1) を引く.
この 2 重の For 文が終了した時点で, 本文の式 (10) のような形になっている.

図 53　例題 11.2 (ex11.2) の解説: 連立 1 次方程式 (第 1 ステップ)

ex11.2 を実行すると以下のようになる:

実行結果

```
---a'ij---
 1.000 -0.400  0.200
 0.000  0.900  1.300
 0.000  1.067 -1.533
---b'i---
 2.600
-6.100
 5.067
```

11.3　連立 1 次方程式 (第 2 ステップ)

式 (10) をもう一度書くと,

$$\begin{aligned} x &+ a'_{12}y &+ a'_{13}z &= b'_1 \quad \cdots \quad ①'' \\ & a'_{22}y &+ a'_{23}z &= b'_2 \quad \cdots \quad ②'' \\ & a'_{32}y &+ a'_{33}z &= b'_3 \quad \cdots \quad ③'' \end{aligned}$$

であった. ここで, 前節と同じ種類の計算を式 ②'' と式 ③'' に対して行うと (①'' はそのまま),

$$\begin{aligned} x &+ a'_{12}y &+ a'_{13}z &= b'_1 \quad \cdots \quad ①'' \\ & y &+ a''_{23}z &= b''_2 \quad \cdots \quad ②''' \\ & & a''_{33}z &= b''_3 \quad \cdots \quad ③''' \end{aligned} \tag{11}$$

の形になりそうである. これは, 第 1 ステップと同じ計算なので, 前節のプログラムを修正すればできそうだ. 具体例に書いてみると, 下図左が第 1 ステップ, 下図右が第 2 ステップである. 数字の 0 を 1 に, 1 を 2 に置き換えただけである.

第 1 ステップ

```
for i in range(0, 3):

    ta = a[i][0]
    for j in range(0, 3):

        a[i][j] = a[i][j] / ta

    b[i] = b[i] / ta

for i in range(1, 3):

    for j in range(0, 3)

        a[i][j] = a[i][j] - a[0][j]

    b[i] = b[i] - b[0]
```

第 2 ステップ

```
for i in range(1, 3):

    ta = a[i][1]
    for j in range (1, 3):

        a[i][j] = a[i][j] / ta

    b[i] = b[i] / ta

for i in range(2, 3):

    for j in range(1, 3):

        a[i][j] = a[i][j] - a[1][j]

    b[i] = b[i] - b[1]
```

このふたつのステップはほとんど同じなので, for 文でまとめられるだろう. 実際に, 作成したのが以下のプログラムである.

例題 11.3　　**ex11.3**

```
a = [[ 5, -2,  1],
     [ 2,  1,  3],
     [ 3,  2, -4]]

b = [13, -7, 23]

for k in range(0, 2):

    for i in range(k, 3):
        ta = a[i][k]
        for j in range(k, 3):
            a[i][j] = a[i][j] / ta
        b[i] = b[i] / ta

    for i in range(k+1, 3):
        for j in range(k, 3):
            a[i][j] = a[i][j] - a[k][j]
        b[i] = b[i] - b[k]

print("---a''ij---")
for i in range(0, 3):
    for j in range(0, 3):
        print (f'{a[i][j]:6.3f}', end=' ')
    print('')

print("---b''i---")
for i in range(0, 3):
    print (f'{b[i]:6.3f}')
```

A 第 1 ステップと第 2 ステップをまとめて行うために, for 文を使う. `k=0` が第 1 ステップの計算, `k=1` が第 2 ステップの計算に対応する.

B 第 1 ステップで, 数値の「1」だった所を「k」に置き換えた. 以下同様.

C 第 1 ステップで, 数値の「2」だった所を「k+1」に置き換えた.

図 54　例題 11.3(ex11.3) の解説: 連立 1 次方程式 (第 2 ステップ)

ex11.3 を実行すると以下のようになる:

実行結果

```
---a''ij---
 1.000 -0.400  0.200
 0.000  1.000  1.444
 0.000  0.000 -2.882
---b''i---
 2.600
-6.778
11.528
```

11.4　演習問題

課題 11.1 (基本) 2 つの 3×3 行列が与えられたとき, それらの積を求めるプログラムを作成せよ (プログラムとしては, 3 重の for 文を用いることが望ましいが, それ以外でも可).

さらに, numpy の配列を用いた結果と一致することを確認せよ.

numpy による行列の積

```
import numpy as np
print(np.array(a) @ np.array(b))
```

課題 11.2 (基本) テキストの式 (11) まで式変形ができれば, x, y, z の値を求める (つまり, 連立方程式を解く) のは容易だろう. まず始めに z を求め, その結果を元に y を求める. 最後に, y, z の値を用いて x を求めれば良い.

そこで, 例題 11.3 (ex11.3) を元に, 3 変数の連立 1 次方程式の解を求めるプログラム (pr11.2) を作成せよ (x, y, z を表す変数を新たに宣言する必要がある. また, 追加する部分は, for 文を用いることが望ましいが, それ以外の方法でも可).

12　確率と統計: 乱数の生成　(応用編 5)

さいころを次々に振って出た目を並べたような, でたらめな数の並びを**乱数列**と呼ぶ. では, 同じ計算であれば何度繰り返しても同じ答えを出してしまうコンピュータは, でたらめな数の列を発生してくれるだろうか? 厳密にいうと, 答えは No. でも, 乱数そっくりの **「疑似」乱数**ならば, コンピューターは作ってくれる[*33]. Python には, メルセンヌ・ツイスターと呼ばれる方法を用いて疑似乱数を発生させる関数 `randint` や `uniform` が用意されている. これはこれまでに用いた, `sin` や `cos` などの数学関数と同じようにして, 用いることができる.

こうして作成した疑似乱数は多様な用途に用いることができる. ここでは一例として, 疑似乱数を用いた数値積分の方法 (モンテカルロ積分) について紹介する.

12.1　整数値を取る乱数

まず, 整数値を取る乱数を作成してみよう. サイコロの目の値をイメージすると良いだろう.

例題 12.1　　ex12.1

```
import random as rd

for i in range (0, 10):

    x = rd.randint(1, 6)
    print (x)
```

A 乱数を生成する関数 `randint` を使用するために `random` というモジュールを読み込む.

B 10 回繰り返す.

C `rndint (1, 6)` は 1 から 6 までの整数値を取る. これを繰り返し実行すると, その度に異なる値がランダムに得られる.

図 55　例題 12.1 (ex12.1) の解説: 整数値を取る疑似乱数の生成

for 文で randint 関数を 10 回呼び出しているので, 1 から 6 までのランダムな数が 10 個出力される. これは, サイコロを 10 回振ったような状況をイメージすると良い [*34]. ex12.1 を実行すると以

[*33] どのようにして乱数を作るのか, を理解するにはかなり高度な数学の知識を必要とする.

[*34] ここで生成している乱数は, 離散的な確率変数とみなすことができる. その取り得る値は 1 から 6 までの整数で, それぞれの値を取る確率は一様に 1/6 である.

下のようになる (みなさんの実行結果はテキストに載せたものと異なると思います)

実行結果

```
2
5
5
3
4
3
4
1
6
2
```

練習

1. 上のプログラム ex12.1 について, 実行する度に結果が変わることを確認せよ.
2. (1 から 20 までの目が出る) 正 20 面体サイコロを 10 回振ったときの目の値をコンピュータ上で疑似的に作りたい. どうすれば良いか?
3. 上のプログラム ex12.1 について, 1 から 6 の目がおおよそ同じ頻度で出ていることを頻度分布を出力することで確かめよう (プログラム名を ex12.1.1 とする). 試行回数を 900 回として作成した下のプログラムを完成させよ. ここで, リスト `h` を用いてサイコロの目の度数をカウントする. 例えば, `h[1]` が 「1 の目が出た回数」になるようにしよう.

練習 12.1.1　　**ex12.1.1**

```
import random as rd

h = [0, 0, 0, 0, 0, 0] # 出た目の度数

for i in range (0, 900):

    x = rd.randint(1, 6)
    # ここに 1 行追加せよ

for i in range (0, 6):
    print (i+1, h[i])
```

さらに, 上のプログラムに次の 4 行を追加して度数分布多角形[*35]を作成せよ (プログラム名を ex12.1.2 とする):

[*35] 定義が分からに人は中学校の教科書を調べて欲しい.

追加部分

```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
x = [1, 2, 3, 4, 5, 6]
plt.plot(x, h)
```

4. 前問で作成したグラフの縦軸を相対度数[*36]にするにはどのようにすれば良いだろうか?

[*36] 上の脚注と同様.

12.2　実数値を取る乱数

次に, 実数値をとる乱数を生成させよう.

例題 12.2　　**ex12.2**

```
import random as rd

for i in range (0, 10):

      x = rd.uniform(0, 1)
      print (x)
```

A `uniform (0, 1)` とすると, 0 から 1 の間のランダムな数が得られる. この例では, `for` 文を用いて, 10 回 `uniform` を使用しているが, 使用するたびに違う値が出力される.

具体的には, 0 から 1 の間の実数値が一様な確率で得られる.

図 56　例題 12.2 (ex12.2) の解説: 実数値を取る疑似乱数の生成

ex12.2 を実行すると以下のようになる: [*37][*38]:

実行結果

```
0.4665046506861684
0.12740589128550606
0.8477451389419861
0.26588833869688255
0.9344595156683886
0.942956645501784
0.2625795622595175
0.40740168329913096
0.25444835612428185
0.6781901822763671
```

練習

1. 上のプログラム ex12.2 について, 実行する度に結果が変わることを確認せよ.
2. 0 から 2π の間の実数値をランダムに出力するようにしたい. どうすれば良いか?
3. X は 0 から 2 の間の実数値をランダムに取る確率変数であるとする. 新たな確率変数 Y を $Y = \sqrt{2X}$ と定義する. 確率変数 Y の値をランダムに 10 個出力するプログラム ex12.2.1 を作成せよ. また, 何度か実行してみて, Y の確率密度関数の関数形を予想せよ. 平方根

[*37] みなさんの乱数列の値はテキストに載せたものと異なるはずである.

[*38] ここで生成している乱数は, 連続的な確率変数とみなすことができる. その取り得る値は 0 から 1 の間の実数値で, 確率密度は一様密度

$$f(x) = \begin{cases} 1 & (0 \leq x < 1) \\ 0 & (\text{上記以外の } x) \end{cases}$$

に従う. この一様密度を利用すれば, 色々な確率密度に従う確率変数を作り出すことができる. 例えば, 標準正規分布と呼ばれる (極めて重要な) 確率密度関数に従う確率変数であれば, ボックス=ミュラー法 (Box-Muller's method) と呼ばれる方法がある (あるいは, 別の確率密度の例として, 次の例題の練習 (3) も参照).

[`np.sqrt(···)`] を利用するには,

```
import numpy as np
```

という 1 行が必要である.

ヒント: 例題 ex12.2 と同様の方法で, 0 から 2 の間の実数値をランダムに 10 個の発生させるとする. これを $x_i\,(1 \leq i \leq 10)$ と表そう. これは, 確率変数 X の観測値 (実現値) と見なすことができる. このとき確率変数 Y の観測値は $y_i = \sqrt{2x_i}\,(1 \leq i \leq 10)$ で与えられる.

12.3　ヒストグラムの作成

どのような値がどのような頻度で発生するかを知るためには, ヒストグラムが基本的な道具である. ここでは, 実数値乱数のヒストグラムを作成しよう. ヒストグラムの縦軸の長さは次式で与えられる[*39]:

$$\frac{(\text{度数})}{(\text{試行回数}) \times (\text{階級幅})}$$

例題 12.3　　**ex12.3**

```
import random as rd

h = [0] * 20

for i in range(0, 1000):

    x = rd.uniform(0, 2)
    n = int(x * 10)
    h[n] = h[n] + 1

for n in range(0, 20):
    cv = 0.1 * n + 0.05
    h[n] = h[n] / 1000 / 0.1
    print (f'{cv:4.2f} {h[n]:7.5f}')
```

A　20 個の 0 から成るリストの生成

B　0 から 2 の間の実数値を一様に取る乱数を発生させている.

C　`x * 10` は 0 から 19.999... までの実数値を取る.
`int(...)` は実数の整数部分を求める関数である. したがって, `n` は 0 から 19 までの整数値を取る.
この `n` は `x` がどの階級に入ったかを示す. 例えば, $n = 3$ であれば, `x` は 0.3〜0.4 の階級に入っている.

x	x * 10	n
0〜0.1	0.0〜1.0	0
0.1〜0.2	1.0〜2.0	1
0.2〜0.3	2.0〜3.0	2
0.3〜0.4	3.0〜4.0	3
⋮	⋮	⋮
1.8〜1.9	18.0〜19.0	18
1.9〜2.0	19.0〜20.0	19

D　階級値の計算

図 57　例題 12.3 (ex12.3) の解説: 実数値を取る疑似乱数の生成

ex12.3 の `h = [0] * 20` は次の文と同じ役割をしている (20 個の 0 から成るリストの生成):

```
h = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

ex12.3 を実行すると以下のようになる: [*40]:

[*39] ヒストグラムの定義として, (度数)/(試行回数) を縦軸に取る方が一般的だが, ここではヒストグラムを確率密度関数と対応させるために, (階級幅) で割っている. 詳細は高校数学の教科書か計算数学特論の授業を参照.

[*40] ここで生成している乱数は, 連続確率変数とみなすことができる. 確率密度は次の一様密度に従う:

$$f(x) = \begin{cases} \frac{1}{2} & (0 \le x < 2) \\ 0 & (\text{上記以外の } x) \end{cases}$$

実行結果

```
0.05 0.46000
0.15 0.58000
0.25 0.42000
0.35 0.61000
0.45 0.60000
   ⋮
（中略）
   ⋮
1.55 0.35000
1.65 0.54000
1.75 0.55000
1.85 0.59000
1.95 0.52000
```

練習

1. 上のプログラム ex12.3 について, 実行する度にヒストグラムの形状が変わることを確認せよ.
2. 得られたデータをグラフにしてみよう. グラフにするには, ヒストグラムの値だけでなく, 階級値 cv もリストにしておく必要がある. そのために,

   ```
   cv = [0] * 20
   ```

   と階級値のリストを作成し, cv[n] に n 番目の階級値が代入されるように例題を修正せよ (ex12.3.1 とする). 最後に,

   ```
   import matplotlib.pyplot as plt
   import seaborn as sns; sns.set()
   plt.bar(cv, h, width = 0.1)
   ```

   として, ヒストグラムを出力せよ.
3. プログラム ex12.3.1 を修正し, x のヒストグラムの代わりに y = np.sqrt (2 * x) のヒストグラムを出力してみよ (import numpy as np を忘れずに). 元の x と比較してヒストグラムにどのような違いがありますか? [*41] (プログラム名を ex12.3.2 とする)
4. この節では, データからヒストグラムの数値を表すリスト h を作成した. この作業は (もちろん) 自動的に行うことが可能である. 以下のプログラムを作成・実行し, ex12.3.1 と類似

[*41] y は, やはり連続的な確率変数である. その取り得る値は 0 から 2 の間の実数値で, 確率密度は

$$g(y) = \begin{cases} \frac{y}{2} & (0 \leq y < 2) \\ 0 & (\text{上記以外の } u) \end{cases}$$

に従う. つまり, 2 に近い値ほど出やすい.

の結果が得られることを確認せよ (違いについても吟味せよ).

練習 12.3.3 **ex12.3.3**

```
import random as rd

x = [0] * 1000

for i in range(0, 1000):

    x[i] = rd.uniform(0, 2)

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.hist(x, bins=20, density=True)
```

12.4　確率変数の和

X と Y という 2 つの確率変数の和 $X+Y$ について考えてみよう．X を数学の点, Y を英語の点だとすると, $X+Y$ は合計点を意味する．合計点のヒストグラムはどのような形になるだろうか?

例題 12.4　　**ex12.4**

```
import random as rd

h = [0] * 40
cv = [0] * 40

for i in range(0, 1000):

    x = rd.uniform(0, 2)
    y = rd.uniform(0, 2)

    n = int((x+y) * 10)
    h[n] = h[n] + 1

for n in range(0,40):
    cv[n] = 0.1 * n + 0.05
    h[n] = h[n] / 1000 / 0.1
    print (f'{cv[n]:4.2f} {h[n]:7.5f}')

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.bar(cv, h, width = 0.1)
```

A　0 から 2 の間の実数値を一様に取る乱数を 2 つ (`x` と `y`) 発生させている．

B　`(x +y) * 10` は 0 から 39.999... までの実数値を取る．
`int(...)` は実数の整数部分を求める関数である．したがって, `n` は 0 から 39 までの整数値を取る．
この `n` は `x` がどの階級に入ったかを示す．例えば, $n=3$ であれば, `x` は 0.3〜0.4 の階級に入っている．

x	x * 10	n
0〜0.1	0.0〜1.0	0
0.1〜0.2	1.0〜2.0	1
0.2〜0.3	2.0〜3.0	2
0.3〜0.4	3.0〜4.0	3
⋮	⋮	⋮
3.8〜3.9	38.0〜39.0	38
3.9〜4.0	39.0〜40.0	39

図 58　例題 12.4 (ex12.4) の解説: 実数値を取る疑似乱数の生成

ex12.4 を実行すると以下のようになる (図は省略する):

実行結果

```
0.05 0.01000
0.15 0.04000
0.25 0.03000
   ⋮
（中略）
   ⋮
3.75 0.09000
3.85 0.06000
3.95 0.00000
```

確率論によると, 2 つの確率変数 X と Y がともに, 一様密度

$$f(x) = \begin{cases} \frac{1}{2} & (0 \leq x < 2) \\ 0 & (\text{上記以外の } x) \end{cases}$$

に従うとき, それらの和 $Z = X + Y$ の確率密度 $g(z)$ は, 三角密度

$$g(z) = \begin{cases} \frac{z}{4} & (0 < z \leq 2) \\ 1 - \frac{z}{4} & (2 < z \leq 4) \\ 0 & (\text{上記以外の } x) \end{cases}$$

に従うことが示せる.

練習

1. 上のプログラム ex12.4 について, 実行する度に結果が変わることを確認せよ.
2. プログラム ex12.4 を修正し, 例題と同じ一様密度に従う 3 つの変数の和

   $$x + y + z$$

   のヒストグラムを出力せよ (プログラム名を ex12.4.1 とする; 出力が長くなるので数値は出力しなくても良い). 例題とどのような違いがあるか述べよ?

12.5　モンテカルロ積分

乱数を使うと定積分の計算が可能である．どのように用いるか想像できるだろうか? (しばらく考えてみよ)

＊ ＊ ＊

ここでは, 例として単位円 (半径 1 の円) の面積 S を計算してみる (厳密な答えは, もちろん円周率 π である). まず, 下の図のような 1 辺の長さが 1 の正方形を考えよう．この正方形の中にランダムにたくさんの点を打つ．そのためには, 0 と 1 の間の一様乱数を 2 個発生させて, そのうちの一方を点の x 座標, もう一方を点の y 座標とする．その点が 4 分円の中に入ったかどうかを判断し, 点の総数を N, そのうち 4 分円の中に入った点の個数を m とする．点の総数を多くすれば

$$\lim_{N\to\infty}\frac{m}{N}=\frac{\text{4 分円の面積}}{\text{正方形の面積}}=\frac{\pi}{4}$$

となるはずである．従って, 単位円の面積は (上の式を 4 倍して) $S=4\lim_{N\to\infty}\dfrac{m}{N}$ と計算できる[*42].

これにより, 関数 $f(x)=\sqrt{1-x^2}$ の下側の面積を計算したことになる．「面積」＝「積分値」 だから, 結局次の積分値を計算したことになる:

$$S=4\int_0^1 \mathrm{d}x\sqrt{1-x^2}$$

このため, こうした計算手法はモンテカルロ積分と呼ばれる．
このプログラムは次ページのようになる:

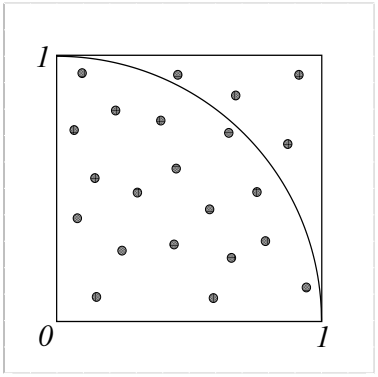

図 59　黒丸はランダムな点。

[*42] このような計算は, アナログでもできるでしょう．正 20 面体サイコロ を 2 回振り, その値からランダムな座標 (x,y) を作れば良い.

例題 12.5　　ex12.5

```
import random as rd

m = 0
N = 10000

for i in range(0, N):

    x = rd.uniform(0, 1)
    y = rd.uniform(0, 1)

    if  x * x + y * y < 1:

        m = m + 1

print(4.0 * m / N)
```

A 1 辺の長さが 1 の正方形の中に, ランダムな点 (x, y) を, 疑似乱数を用いて生成.

B ランダムな点 (x, y) が 4 分円の中に入ったかどうかを判断. 入っている場合には, m の値を 1 増加させる. ここで, m は 4 分円内部に入ったランダムな点の総数であることに注意.

図 60　例題 12.5 (ex12.5.c) の解説: モンテカルロ積分の例

ex12.5 を実行すると以下のようになる[*43]:

実行結果

```
3.1448
```

[*43] みなさんの値とは異なる.

12.6　演習問題

課題 12.1 (基本) 半径が 1 の球の体積をモンテカルロ積分で計算するプログラム pr12.1 を作成せよ. サンプル数 (ランダムな点の数) は 100000 程度で良いだろう. 半径 1 の 球は

$$x^2 + y^2 + z^2 < 1$$

を満たす領域で定義される[*44].

課題 12.2 (基本) 平面上を運動する虫を考えよう. この虫は 1 秒間にきっちり 1 cm の長さの線分に沿って動く. ただ運動 (線分) の方向はランダムであるとする. つまり, i 秒後の虫の位置の x 座標を x_i, y 座標を y_i とすると

$$x_{i+1} = x_i + \cos\theta_i$$
$$y_{i+1} = y_i + \sin\theta_i$$

であり, θ_i $(i = 1, 2, \cdots)$ は 0 と 2π の間の一様な乱数である. 時刻 0 のときに虫は原点にいたとする. このとき, 虫の軌跡 (x_i, y_i) を計算するプログラムを作成しよう (pr12.2 とする). 時刻 1000 秒まで計算し, 軌跡のグラフを作成せよ[*45].

ヒント: 漸化式の計算と同様にして計算できるが, グラフを作成するために x と y はリストにする必要がある. また, θ_i の値には, 乱数 `uniform` を用いる.

課題 12.3 (やや難) 0, 1, 2, 3 という 4 つの数字をランダムに入れ換えて順列を作ることを考えよう. このとき, 全ての順列が同じ確率で現れるようにしたい. 次の 2 つのプログラムを考えたが, 正しいのはどちらか.

```
import random as rd

a = [0, 1, 2, 3]
for i in range(0, 3):

    x = rd.randint(0, 3)
    a[i], a[x] = a[x], a[i]

print(a)
```

```
import random as rd

a = [0, 1, 2, 3]
for i in range(0, 3):

    x = rd.randint(i, 3)
    a[i], a[x] = a[x], a[i]

print(a)
```

[*44] ちなみに, 半径 1 の n 次元球は

$$x_1^2 + x_2^2 + \cdots + x_n^2 < 1$$

を満たす領域で定義される. さて, この体積はいくらでしょうか?

[*45] この例は, 高校物理や化学に出てくる「ブラウン運動」という現象と関連している.

13　確率と統計: 2 項分布と正規分布　(応用編 6)

気体中や液体中の分子の速度は正規分布に従うことが知られている (マクスウェル・ボルツマン分布と呼ばれる). 他にも試験の点数の分布や身長の分布なども正規分布に近似的に従うとことが指摘されている. あるいは工業製品の品質 (寸法や質量, 寿命など) も正規分布に従うことが多い. このように様々な現象に表れることから, 正規分布は統計学において最も重要な確率密度関数であると考えれる (実際, 推定や検定などの統計分析手法には, 正規分布に従うことを前提とするものが多い).

正規分布がこのように様々が現象に表われる理由は, n 個の確率変数の和

$$X_1 + X_2 + \cdots + X_n$$

の確率密度関数が, n が大きいとき正規分布に近づくためである (もちろん, いくつかの条件が必要だが, 比較的緩い条件で成立する). このような性質を「中心極限定理」と呼ぶが, その証明はかなり難しいので, ここでは 2 項分布のシミュレーションを通して, このことを体験することにしよう[*46].

13.1　正規分布の概形

まずは正規分布を表示させて, どのような形かを確認しよう. 一般の正規分布の確率密度関数は

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

という形をしている. ここで, μ は期待値, σ^2 は分散を表している. この確率密度関数を $N(\mu, \sigma^2)$ と表すことが多い. 一方, 標準正規分布は

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

で定義される (すなわち, $\mu = 0$, $\sigma^2 = 1$ の正規分布が標準正規分布である). 先の記号で表わせば, 標準正規分布は $N(0, 1)$ と表わされる. 統計の本や高校の教科書には「標準正規分布表」という表が掲載されていて, 標準正規分布に関わる確率を算出できるようになっている. まずは, 標準正規分布を表示してみよう.

[*46] 2 項分布が正規分布に近づくことは高校数学でも紹介されている. 調べてみよ.

例題 13.1　　**ex13.1**

```
import numpy as np

c = 1/ np.sqrt(2*np.pi)
x = np.linspace(-3, 3, 61)
y = c * np.exp(-x*x/2)

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.plot(x, y)
```

A `np.sqrt(...)` は平方根, `np.pi` は円周率.

B x の範囲は -3〜3. この区間に 61 個の点を取る.

C `np.exp(...)` は指数関数.

図 61　例題 13.1 (ex13.1) の解説: 標準正規分布のグラフを出力する.

ex13.1 を実行すると次のようなグラフが得られる:

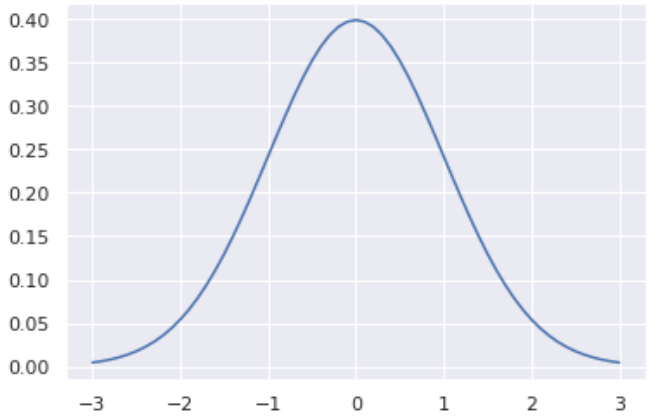

図 62　横軸は試行回数 N, 縦軸は 1 の目の相対度数 r/N を表す. 試行回数が増えるに従い, $1/6 = 0.1666..$ に近づいていくように見えなくもないが, 少し分かりにくい.

練習

1. 正規分布 $N(1, (0.5)^2)$ のグラフを重ねて表示せよ. ただし, x の範囲は, 例題と同じとすること. プログラム名を ex13.1.1 とする.

```
plt.plot(x, y1)
plt.plot(x, y2)
```

   のようにすれば, 2 つの関数を表示できる.

2. プログラムを ex13.1.1 を関数を用いたプログラムに修正せよ (ex13.1.2 とする).

13.2　2 項分布 (試行回数 1 回) に従う乱数

2 項分布 $f(x)$ は次式で定義される:

$$f(x) = {}_nC_x p^x (1-p)^{n-x} \qquad (x = 0, 1, \ldots, n)$$

2 項分布は,

> 結果が 2 通り (「成功 or 失敗」) の試行を, 独立に n 回繰り返したときに, 成功の回数が x 回である確率を表している. ただし, 成功する確率が p であるとする.

ここで, 2 通りの結果として「成功 or 失敗」を考えたが, 「表 or 裏」, 「支持 or 不支持」などであっても構わない. 具体例としては, 「コイン投げ」をイメージすれば良いだろう. 表が出る確率が p のコインを n 回投げたとき, x は表が出た回数である. まずは, コイン投げの回数 n が $n = 1$ の場合を考えよう.

例題 13.2　　**ex13.2**

```
import random as rd

h = [0, 0]

for i in range(0, 1000):

    x = rd.randint(0, 1)
    h[x] = h[x] + 1

print(h[0], h[1])
```

A 度数分布を作成するために, 1 枚のコイン投げを 1000 回行う.

B `rd.randint(0, 1)` で 0 か 1 を確率 1/2 で得る.

C `h(x)` は度数分布. `x = 0` だった場合には, `h(0)` の値を 1 だけ増やし, `x = 1` だった場合には, `h(1)` の値を 1 だけ増やす.

図 63　例題 13.2 (ex13.2) の解説: 2 項分布 (試行回数 1 回) に従う乱数の生成

ex13.2 を実行すると以下のようになる:

実行結果

```
484 516
```

練習

1. 上のプログラム ex13.2 について, 実行する度に結果が変わることを確認せよ.
2. コインの表が出る確率が 0.6 であるとする. このようなコインをシュミレーションするプログラムに修正せよ (ヒント, `rd.uniform` を用いると良いでしょう). プログラム名を ex13.2.1 とする.
3. (公平な) コイン投げを 2 回したとき, 表の回数の度数分布を計算するプログラムを作成せよ. プログラム名を ex13.2.2 とする. これは試行回数は 2 回の 2 項分布に対応する.

13.3　2 項分布 (試行回数 10 回) に従う乱数

次に, コイン投げの回数 n が大きい場合の 2 項分布についてのプログラムを作成しよう. 以下のプログラムでは, $n = 10$ としている. n が大きいので, `for` 文を用いる方が良いだろう.

例題 13.3　　**ex13.3**

```
import random as rd

h = [0] * 11

for i in range(0, 1000):

    sum = 0
    for j in range(0, 10):

        x = rd.randint(0, 1)
        sum = sum + x

    h[sum] = h[sum] + 1

for j in range(0, 11):
    print(f'{j:2} {h[j]:3}')
```

A 頻度分布を作成するために, 10 枚のコイン投げを 1000 回行う.

B コインを 10 枚振る.

C `sum` は 10 枚コインを振ったときの, 表の回数を表す.

図 64　例題 13.3 (ex13.3) の解説: 2 項分布 (試行回数 10 回) に従う乱数の生成

ex13.3 を実行すると以下のようになる:

実行結果

```
 0   1
 1   8
 2  46
 3 113
 4 197
 5 255
 6 199
 7 127
 8  49
 9   3
10   2
```

1 列目は「表の枚数」
2 列目は「度数」

練習

1. 上のプログラム ex13.3 について, 実行する度に結果が変わることを確認せよ.
2. 数値の出力の代わりに, 以下の 4 行を追加することで, グラフを作成せよ (プログラム名を ex13.3.1 とする)

```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
x = range(0, 11)
plt.plot(x, h, 'o')
```

   最後の `'o'` は「線を引く」のではなく「点をプロットする」ように指示している.
3. 正規分布 $N(5, 2.5)$ の値の 1000 倍

$$f(x) \times 1000$$

   のグラフを重ねて表示せよ (ヒント: 図 61 を参照. 1000 倍しているのは, 確率から度数に変換するためである). プログラム名を ex13.3.2 とする.
4. コインの表が出る確率を 0.4, 0.3, 0.2 と変化させたとき, グラフの変化を確認せよ. また, 正規分布の期待値と分散を対応する値に変更したとき[*47], 2 項分布と正規分布は良く一致するか[*48]. プログラム名を ex13.3.3 とする.

[*47] 2 項分布の期待値は np, 分散は $np(1-p)$ である. 高校数学の教科書を参照.

[*48] $p < 0.5$ のとき, 2 項分布を正規分布で良く近似できる目安は, $pn > 5$ である (あくまで目安であることに注意). ここで n は試行回数であり, 例題では $n = 10$ に設定している.

13.4　標本平均と標本分散

ある集団 (母集団と呼ぶ) から, いくつかの要素を抽出して調べることで, この母集団の期待値 μ や分散 σ^2 を推定することを考える[*49]. この抽出された要素の集合を「標本」と呼ぶ. ここでは, 母集団の要素が確率分布 $f(x)$ に従うとし (離散でも連続でも良い), この確率分布の期待値と分散が μ と σ^2 で与えられるとする. 標本から得られる量と区別するため, これらの量を「母平均 μ, 母分散 σ^2」と呼ぶ. この μ と σ^2 の値を推定する方法を考えよう.

標本の要素の算術平均 (全部を足し合わせて, 要素数で割って得られる量) を計算すれば, 母平均 μ に近い値を取ることが予想できるだろう. 実際, 期待値 μ の推定量として, 標本平均 $\overline{X}$ が最も基本的である:

$$\boxed{\text{標本平均}} \qquad \overline{X} = \frac{X_1 + X_2 + \cdots + X_n}{n}$$

標本平均は確率変数の和なので, n が大きいとき正規分布に従うことを注意しておこう[*50].

同様に, 母分散 σ^2 を推定する場合にも, 算術平均を用いる.

$$(\text{母分散}) = (\text{母平均からのズレの 2 乗の期待値})$$

であることから,

$$(\text{標本分散}) = (\text{標本平均からのズレの 2 乗の算術平均})$$

と定義することが自然であろう. そこで, 標本分散 S^2 を次のように定義する [*51]:

$$\boxed{\text{標本分散}} \qquad S^2 = \frac{(X_1 - \overline{X})^2 + \cdots + (X_n - \overline{X})^2}{n}$$

ただし, プログラムでは標本分散は次式を用いて計算する (上の定義と一致することを確かめよ):

$$S^2 = \frac{X_1^2 + \cdots + X_n^2}{n} - (\overline{X})^2$$

また, 標本分散の平方根 (の正の根) を標本標準偏差と呼び S と表わす.

ここでは, 2 項分布 ($n = 1$) の標本平均と標本分散をプログラムを用いて計算してみよう[*52]. そのために, ex13.2 を元に, 次のようなプログラムを作成しよう.

[*49] 期待値と分散の定義については高校の教科書を参照.

[*50] 標本平均も下の標本分散も, 5.2 節で学んだ平均と分散と同じものである (したがって, プログラムも当然類似している). 違う点は母集団というより大きな集合を想定しているかどうか, の違いだけである.

[*51] 分散の推定には不偏分散と呼ばれる量を用いることが多いが, 標本の数が大きい場合は標本分散とほとんど変わらない.

[*52] Python にはこれらを計算する関数 `np.mean` と `np.var` があるが, ここではこれらの関数を用いずに求めてみる

例題 13.4　　**ex13.4**

```
import random as rd

N = 1000
sum1 = 0
sum2 = 0

for i in range(0, N):

    x = rd.randint(0, 1)
    sum1 = sum1 + x
    sum2 = sum2 + x * x

mean_x = sum1 / N
var_x = sum2 / N - mean_x * mean_x
print(f'{mean_x:5.3f}  {var_x:5.3f}')
```

A 1 枚のコイン投げを 1000 回行う. x は表ならば 1, 裏ならば 0 と解釈する.

B sum1 は標本平均の定義式の分子, sum2 は標本分散の 2 つ目の式の第 1 項の分子を表す.

図 65　例題 13.4 (ex13.4) の解説: 標本平均と標本分散

ex13.4 を実行すると以下のようになる:

実行結果

```
0.488  0.250                    (←標本平均と標本分散)
```

練習

1. 上のプログラム ex13.4 について, 実行する度に結果が変わることを確認せよ. N の値を変えたときどのような変化があるか調べよ.
2. 2 項分布 $(n = 1)$ の期待値と分散がどのような式で与えられるか調べよ. この値を, 上で計算した標本平均と標本分散と比較せよ.
3. コインの表が出る確率が 0.6 であるとする. このようなコインをシュミレーションするプログラムに修正せよ. プログラム名を ex13.4.2 とする.
4. $n = 10$ の 2 項分布に従う確率変数について, 標本平均と標本分散を求めるプログラム ex13.4.3 を作成せよ.

13.5　正規乱数

これまで, 一様乱数や整数値を取る乱数を生成してきた. ここでは, 正規分布に従う乱数を発生させる (正規乱数と呼ぶ). 下の例題では, リストを使用する必要は無いが, 練習問題でグラフを作成するため, 正規乱数をリスト `y[i]` に保存している.

例題 13.5　　ex13.5

```
import random as rd

N = 10
y = [0] * N

for i in range(0, N):

    y[i] = rd.gauss(1, 0.5)
    print (y[i])
```

A 期待値 1, 標準偏差 0.5 の正規分布 $N\left(1, 0.5^2\right)$ に従う乱数を生成している. 標準偏差と分散の区別に注意.

図 66　例題 13.5 (ex13.5) の解説: 正規乱数の値を出力する.

ex13.5 を実行すると以下のようになる:

実行結果

```
1.0226214815833061
1.846356123868509
0.7293900642605431
⋮
(中略)
⋮
0.4699517275718422
1.1223437446364026
0.36358451199485753
```

練習

1. 例題を修正し, データの標本平均と標本分散を出力するプログラムを作成せよ. プログラム名を ex13.5.1 とする.
2. さらに, 正規乱数を図示してみよ (縦軸は `y[i]` とする. 横軸は試行回数 `i` とするが, 図示するには `i` を要素とするリストが必要なので, これを `x[i] = i` などとして値を設定せよ). また, `N` を変化させてみよ ($N = 100$ など). プログラム名を ex13.5.2 とする.
3. 得られたデータからヒストグラムを作成せよ. ヒストグラムの定義は 12.3 節 を参照すること. 階級幅は 0.1 とし, -3.0 から 3.0 の範囲で作成せよ. プログラム名を ex13.5.3 とする.
4. 前問のヒストグラムと, 例題 13.1 の練習問題 1 で作成した $N(1, (0.5)^2)$ のデータを線グラ

フにして 1 つのグラフとして表示せよ.

課題 13.1 (基本)　100 人の試験の点数データを以下に示す. このデータの標本平均 $\overline{X}$ と標本分散 S^2 を求めるプログラム pr13.1 を作成せよ.

76	57	69	70	52	77	39	67	56	91
32	64	73	80	48	66	57	65	47	57
62	78	72	42	90	67	61	68	52	74
64	82	59	73	44	50	62	50	48	69
72	58	61	52	39	65	53	79	64	89
62	54	60	71	66	64	50	74	79	44
68	50	59	75	74	55	58	66	48	79
68	63	67	73	60	72	56	77	65	69
60	42	87	49	72	42	75	53	58	42
51	54	80	34	61	82	81	49	79	52

課題 13.2 (発展)　上の試験のデータのヒストグラムを描画するプログラム pr13.2 を作成せよ. ただし, 階級幅は 4 とする [*53].

課題 13.3 (基本)　上のヒストグラムに, 正規分布 $N(\overline{X}, S^2)$ を重ねて描画するプログラム pr13.3 を作成せよ.

課題 13.4 (発展)　課題 3 と同じグラフを,

`np.mean`,　`np.var`,　`plt.hist`

用いて作成せよ. プログラム名を pr13.4 とする.

課題 13.5 (発展)　正規乱数を元に, $\chi 2$ 乗分布

$$g(y) = \frac{1}{\sqrt{2\pi y}} \exp\left(-\frac{y}{2}\right)$$

に従う乱数を生成するプログラム pr13.5 を作成せよ

[*53] 関数 `plt.hist` を用いると簡単だが (普通はそうするが), ここではこの関数ではないく. `plt.bar` を用いて作成すること.

14　確率と統計: 推定と検定　(応用編 7)

この章では, 応用上重要な区間推定と検定をやってみよう. データには 13.5 節で導入した正規乱数を用いる[*54].

14.1　正規乱数

例によってデータはコンピュータで生成するが, 雰囲気を出すためにデータを「みかん 1 個の重さ」としよう. みかん 10 個の重さを測ったときのデータを次のプログラムで生成する.

例題 14.1　　**ex14.1**

```
import random as rd

N = 10
x = [0] * N

for i in range(0, N):

    x[i] = rd.gauss(120, 30)
    print (x[i])
```

A みかんの重さは, 期待値 150g, 分散 30^2 の正規分布に従うとする.

ただし, 現実的な状況では, これらの値 (期待値・分散) は未知数であることに注意しよう. 統計学ではこれらの量を「母平均」・「母分散」と呼ぶ. また, これらの値を推定することが統計学の重要な課題である.

図 67　例題 14.1 (ex14.1) の解説: 母平均の信頼区間.

ex14.1 を実行すると以下のようになる:

実行結果

```
113.1703524607762
132.78918443042645
108.17288360560886
135.37857888327034
76.86494554448001
125.64941086088463
124.31067517005306
146.56355011314807
102.90153575146098
183.72774198277654
```

このデータは期待値 120, 標準偏差 30 のデータとして妥当なものだろうか?

[*54] 個々のデータが正規分布に従わない場合であっても, 標本平均は正規分布に近似的に従うことが多いので, 標本が大きい場合にはこの章の方法は一般的に利用できる.

練習

1. 上のプログラム ex14.1 について, 実行する度に結果が変わることを確認せよ.
2. 標本平均 $\overline{X}$ と標本標準偏差 S を求めるプログラムに改良せよ. これらの統計量の定義については 13.4 節を参照せよ. プログラム名を ex14.1.1 としよう.

   得られた標本平均と標本分散が, 母平均・母分散と異っていることに注意しよう. 少ないデータから, 母平均・母分散を精密に推定するのは難しい.

   さらに, Python の関数を用いた結果と一致することを確認せよ:

   ```
   print(np.mean(x), np.std(x))
   ```

3. 問 2 で作成したプログラムについて, 標本の大きさ N をいろいろな値に変えて実行してみよ. どのような違いがあるか?

14.2　母平均の区間推定: 母分散が既知の場合

標本の性質から, 母平均がどのような区間に入るかを推定してみよう. このようなデータ分析を区間推定という[*55].

母分散が既知の場合の, 母平均の区間推定は次式で与えられる[*56]:

$$\left[\overline{X} - \frac{\sigma}{\sqrt{n}} z_1, \overline{X} + \frac{\sigma}{\sqrt{n}} z_1\right]$$

この区間のことを信頼区間と呼ぶ. z_1 の値は危険率によって異り, 危険率が 5% の場合は, $z_1 = 1.960$ である. すなわち, 危険率が 5% の場合には, 100 回区間推定すると, 「母平均 μ が信頼区間に入らないこと」が 5 回程度生じる.

では, 上の信頼区間を計算するプログラムを作成しよう (ex14.1_1 を修正すると良い)

例題 14.2　　**ex14.2**

```
import random as rd
import numpy as np

N = 10
x = [0] * N
sum1 = 0
sum2 = 0

for i in range(0, N):

    x[i] = rd.gauss(120, 30)
    sum1 = sum1 + x[i]
    sum2 = sum2 + x[i] * x[i]

mean_x = sum1 / N
std_x  = np.sqrt(sum2 / N - mean_x * mean_x)

lcl1 = mean_x - 30 / np.sqrt (N) * 1.960
rcl1 = mean_x + 30 / np.sqrt (N) * 1.960

print('母分散既知: ', lcl1, rcl1)
```

A 分散 $\sigma^2 = 30^2$ を既知として, 信頼限界を求める.

図 68　例題 14.2 (ex14.2) の解説: 母平均の信頼区間.

[*55] この後「検定」と呼ばれる統計的手法を紹介するが、ビッグデータの時代となってどちらかというと「区間推定」の方が重要性を増している、と考える統計学者がいる。これは、「検定」の場合、データの数 (標本の大きさ) が多くなると、大抵は「有意」になってしまうという問題があるからである。その点「区間推定」にはこのような問題は無く、データの数を多くなれば、それだけ予測の精度が上昇するだけである。

[*56] 母平均が分からないのに母分散が分かっている、という状況は普通ありえない。しかし、(後述するように) この式を少し修正することで母分散が未知の場合も区間推定が行える。

ex14.2 を実行すると以下のようになる:

実行結果

```
母分散既知:  113.6976956674707 150.88608095105084
```

練習

1. 上のプログラム ex14.2 について, 実行する度に結果が変わることを確認せよ. 特に, 母平均 $\mu = 120$ が推定された区間に入らないことがあることを確認せよ. 100 回実行するとき, このようなことが何回起きるか?
2. 標本の大きさ N をいろいろな値に変えて実行してみよ. どのような違いがあるか?

14.3　母平均の区間推定: 母分散が未知の場合

母分散が未知の場合, 前節の式はそのままでは使えない. しかし, 前節の式の母分散を標本平均で置き換えた式で, (近似的に) 信頼区間が与えられることを示すことができる. すなわち, 母分散が未知の場合の「母平均の区間推定」は次式で与えられる [*57]:

$$\left[\overline{X} - \frac{S}{\sqrt{n}} z_1, \overline{X} + \frac{S}{\sqrt{n}} z_1\right]$$

危険率 5% の場合は, 前節と同様に $z_1 = 1.960$ である.

例題 14.3　**ex14.3**

```
import random as rd
import numpy as np

N = 10
x = [0] * N
sum1 = 0
sum2 = 0

for i in range(0, N):

    x[i] = rd.gauss(120, 30)
    sum1 = sum1 + x[i]
    sum2 = sum2 + x[i] * x[i]

mean_x = sum1 / N
std_x  = np.sqrt(sum2 / N - mean_x * mean_x)

lcl1 = mean_x - 30 / np.sqrt (N) * 1.960
rcl1 = mean_x + 30 / np.sqrt (N) * 1.960

print('母分散既知: ', lcl1, rcl1)

lcl2 = mean_x - std_x / np.sqrt (N) * 1.960
rcl2 = mean_x + std_x / np.sqrt (N) * 1.960

print('母分散未知: ', lcl2, rcl2)
```

A 母分散 ($\sigma^2 = 30^2$) が未知の場合は、母分散の代わりに標本分散 S^2 (プログラムでは平方根をとって標本標準偏差 `std_x` にしてある) を用いて信頼限界を求める.

図 69　例題 14.3 (ex14.3) の解説: 母平均の信頼区間.

[*57] 高校の教科書で紹介されている式と同じである. あくまで近似式であり, n が大きい程正確である.

ex14.3 を実行すると以下のようになる:

実行結果

```
母分散既知:  108.46011125033208 145.6484965339122
母分散未知:  114.99249924062357 139.11610854362073
```

練習

1. 上のプログラム ex14.3 について, 実行する度に結果が変わることを確認せよ. 特に, 母平均 $p = 0.5$ が推定された区間に入らないことがあることを確認せよ. 100 回実行するとき, このようなことが何回起きるか?
2. コインの表が出る確率が 0.6 であるとする. このようなコインをシュミレーションするプログラムに修正せよ.

14.4　検定

推定と同じくよく使われる統計的手法に「検定」がある. ここでも, 具体例を通して考えてみたい. ある機械が袋につめる砂糖の重さは、$\mu = 100$ g になるように調整されている。機械が正しく調整されているかどうかを確かめるために、30 袋を無作為に選んで重さを測った。この 30 個のデータから標本平均 $\overline{X}$ を求め、機械が正しく調整されているか否かを検定しよう。

機械は正しく調整されている、と仮定する。すなわち、重さの期待値が

$$\mu = 100\,(\mathrm{g})$$

であることを仮定する (すなわち正しく調整されている).

30 個は十分大きな標本と考えられるから, 上の仮説の基で, $\overline{X}$ は 正規分布 $N(100, \sigma^2/30)$ に従うことが予想される[*58]. したがって,

$$Z = \frac{\overline{X} - 100}{\frac{\sigma}{\sqrt{30}}}$$

は, 標準正規分布 $N(0,1)$ に従う[*59]. しかし, 母分散 σ^2 が未知なので, σ^2 を標本分散 S^2 に置き換えて,

$$Z = \frac{\overline{X} - 100}{\frac{S}{\sqrt{30}}}$$

を考えてみよう. これが, $|Z| > 1.96$ (棄却域と呼ぶ) に入っていれば, そのような事象 が生じる確率は 5% 未満であり、ほとんどあり得ないことが起きたことになる。そこで、「最初に立てた仮説が間違っていた」と考え、これを棄却するのである[*60]。$|Z| < 1.96$ であれば、仮説が間違っているとは結論できない (正しいとも何とも言えない)。

では, 実際に検定を行うプログラムを作成してみよう。

[*58] $X_1, X_2, \cdots, X_n$ が互いに独立であり, 全て期待値 μ, 分散 σ^2 に従うとき、

$$X_1 + X_2 + \cdots + X_n$$

の期待値は $n\mu$, 分散は $n\sigma^2$ である. したがって、

$$\frac{X_1 + X_2 + \cdots + X_n}{n}$$

の期待値は μ, 分散は σ^2/n である.

[*59] この変換を, 標準化変換と呼ぶ.

[*60] 背理法の進め方に類似していることに注意しよう。

例題 14.4　　**ex14.4**

```
import random as rd
import numpy as np

N = 30  # 試行回数
sum1 = 0
sum2 = 0
x = [0] * N

for i in range(0, N):

    x[i] = rd.gauss(103, 5)
    sum1 = sum1 + x[i]
    sum2 = sum2 + x[i] * x[i]

# 標本平均と標本分散
mean_x = sum1 / N
std_x  = np.sqrt(sum2 / N - mean_x * mean_x)

#検定
Z = (mean_x - 100) / (std_x / np.sqrt(N))

if Z*Z > 1.96*1.96:
    print('棄却（調整されていない）')
else:
    print('採択（なんとも言えない）')
```

A 機械は平均 103 g になるように調整されているとする (もちろん, これは未知とする).

B 仮説 $\mu = 100$ の元では, $\overline{X}$ は正規分布 $N\left(100, \frac{S}{30}\right)$ に従うから, 標準化変換

$$Z = \frac{\overline{X} - 100}{\frac{S}{\sqrt{30}}}$$

を行うと, Z は $N(0,1)$ に従う.

図 70　例題 14.4 (ex14.4) の解説: 検定の例

ex14.4 を実行すると以下のようになる:

実行結果

```
棄却（調整されていない）
```

練習

1. 上のプログラム ex14.4 について, 実行する度に結果が変わることを確認せよ. 特に, 母平均が $\mu = 100$ であっても, 誤って棄却されることがあることを確認せよ (これを第 1 種の誤りという). 100 回実行するとき, このようなことが何回起きるか? $N = 400$ にすると変化はあるか?
2. 今度は, 母平均が $\mu = 103$ であるとする. 今度は誤って採択されてしまう場合がある (これを第 2 種の誤りという). 100 回実行するとき, このようなことが何回起きるか? $N = 400$ にすると変化はあるか?

15　確率と統計: 大数の法則　(応用編 8)

ある試行を n 回行ったとき, ある事象 E が S_n 回生じたとする. このとき,

$$\frac{S_n}{n}$$

を事象 E の相対度数と呼ぶ. また, n を大きくするとき, 相対度数がある値に限りなく近づくとき, その値を経験的確率という. 数学的確率が定義できる場合には, 相対度数が数学的確率に近付くことが数学的に証明できる. これは大数の法則と呼ばれる. このことから, 数学的確率と経験的確率が一致することも分かる. ここでは, サイコロ振りの仮想実験を例に, このことを確かめてみよう.

15.1　経験的確率

1 の目が出る割合 (すなわち相対度数) が試行回数とともにどのように変化していくかを見てみよう.

例題 15.1　　**ex15.1**

```
import random as rd
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

n = 3000
time = [0] * 30
freq = [0] * 30

sum = 0
for i in range(0, n):

    x = rd.randint(1, 6)
    if (x == 1):
        sum = sum + 1

    if ((i+1) % 100 == 0):
        time[i // 100] = i+1
        freq[i // 100] = sum / (i+1)

plt.plot(time, freq, '-o')
```

A `N` は試行回数. `time` と `freq` というリストにデータを保存していく (100 回の試行毎に 1 回保存するので, `N` の 1/100 の長さに定義してある).

B `x` はサイコロの目の値.

C 1 の目が出た回数を数える.

D 試行回数が 100 の倍数のとき, 試行回数と相対度数を記録する.

E `'-o'` は丸プロット (o) を線 (-) でつなげることを意味する.

図 71　例題 15.1 (ex15.1) の解説: 経験的確率.

ex15.1 を実行すると次のようなグラフが得られる:

図 72　横軸は試行回数 N, 縦軸は 1 の目の相対度数 r/N を表す. 試行回数が増えるに従い, $1/6 = 0.1666..$ に近づいていくように見えなくもないが, 少し分かりにくい.

練習

1. 上のプログラム ex15.1 について, 実行する度に結果が変わることを確認せよ.
2. 試行回数 N を 10000 に変えて実行してみよ (他にも変えるべき箇所がある). 例題とどのような違いがあるか?

15.2　大数の法則

前節のプログラムでは, 1 の目の相対度数が $\frac{1}{6}$ に近付いているように見えなくはないが, それほどはっきりしなかった. ここでは, 3000 回の試行を 10 回行い, それらを図示してみよう.

例題 15.2　　**ex15.2**

```
import random as rd
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

n = 3000
time = [0] * 30
freq = [0] * 30

for j in range(0, 10):
    sum = 0
    for i in range(0, n):

        x = rd.randint(1, 6)
        if (x == 1):
            sum = sum + 1

        if ((i+1) % 100 == 0):
            time[i // 100] = i+1
            freq[i // 100] = sum / (i+1)

    plt.plot(time, freq, 'o-', alpha=0.5)
```

A 3000 回の試行を 10 回行う.

B `alpha=0.5` は透過度 50% を意味する. 重なると見にくいので, 透き通って見せるのである.

図 73　例題 15.2 (ex15.2) の解説: 大数の法則.

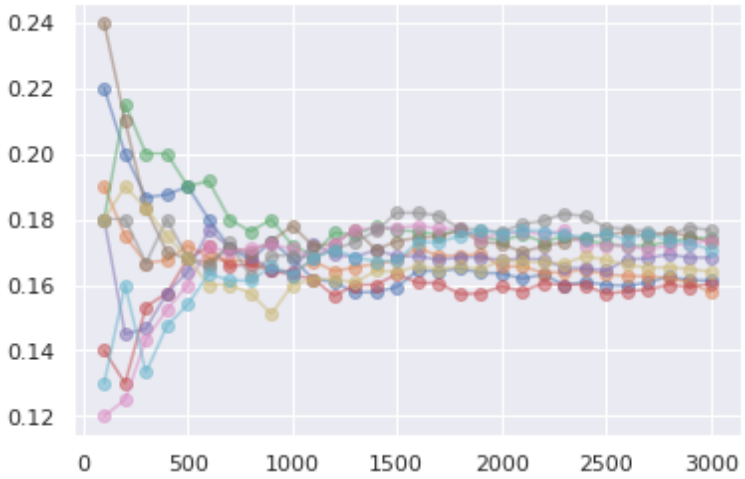

図 74　横軸は試行回数 N, 縦軸は 1 の目の相対度数 r/N を表す. $1/6 = 0.1666..$ の周りにばらついているが, 試行回数が増えるに従い, ばらつきが小さくなっていくように見える.

大数の法則を, 今考えているサイコロ振りの場合に当てはめると次のようになる. すなわち, 任意の実数 ϵ に対し, 次の式が成立する:

$$\lim_{n\to\infty} P\left(\left|\frac{S_n}{n} - \frac{1}{6}\right| \geq \epsilon\right) = 0$$

つまり, 相対度数 $\frac{S_n}{n}$ は $\frac{1}{6}$ の周りにばらついているが, 試行回数を大きくすれば, このばらつきはいくらでも小さくできることを意味する (ϵ 以上ばらつく確率は 0 に収束する). したがって, 上の図に見られるばらつきの減少は大数の法則の表われと見ることができるだろう.

練習

1. 上のプログラム ex15.2 について, 実行する度に結果が変わることを確認せよ.
2. 試行回数 N を 10000 に変えて実行してみよ (他にも変えるべき箇所がある). 例題とどのような違いがあるか?

15.3　標本平均の期待値と分散

15.3.1　一般的な性質

確率変数 X_i $(i = 1, 2, \cdots, n)$ が互いに独立で, かつそれぞれが同じ母集団分布 $f(x)$ に従うことすると, 次式が成り立つ:

$$\begin{cases} E[X_i] &= \mu \\ V[X_i] &= E[(X_i - \mu)^2] = \sigma^2 \end{cases}$$

このとき標本平均

$$\overline{X} = \frac{S_n}{n} = \frac{X_1 + X_2 + \cdots + X_n}{n}$$

について, 以下の式が成り立つ:

$$E[\overline{X}] = \mu \qquad \text{(標本平均の期待値)}$$

$$E[(\overline{X} - \mu)^2] = \frac{\sigma^2}{n} \qquad \text{(標本平均の分散)}$$

15.3.2　サイコロ振りの場合

再び, サイコロ振りの例に戻ると, X_i を

$$X_i = \begin{cases} 1 & (1 \text{ の目が出た場合}) \\ 0 & (1 \text{ 以外の目が出た場合}) \end{cases}$$

と定義すれば, $\overline{X}$ は 1 の目の相対度数を表している. また, μ と σ^2 は

$$\mu = 1 \times \frac{1}{6} + 0 \times \frac{5}{6} = \frac{1}{6}$$

$$\sigma^2 = \left(1 - \frac{1}{6}\right)^2 \times \frac{1}{6} \left(0 - \frac{1}{6}\right)^2 \times \frac{5}{6} = \frac{5}{36}$$

となる. したがって, 標本平均の期待値と分散は

$$E[\overline{X}] = \frac{1}{6}$$

$$E[(\overline{X} - \mu)^2] = \frac{5}{36n}$$

この 2 つの式から, $\overline{X}$ の期待値 $E[\overline{X}]$ は 母平均 μ に一致することと, 分散 $V[\overline{X}]$ ($\overline{X}$ の μ の周りのばらつき) が標本の大きさ n を増やすと小さくなることが分かる. このことから, 標本平均 $\overline{X}$ は母平均 μ の"近似値"[*61] のようなものであり, かつ標本を大きくすると, 母平均 μ からの誤差は小さくなっていくことが分かる.

前節のプログラムを改良して, バラツキの幅を n の関数として描いてみよう. すなわち, 「期待値 ± 標準偏差」の範囲

$$\mu - \frac{\sigma}{\sqrt{n}} \quad \sim \quad \mu + \frac{\sigma}{\sqrt{n}}$$

[*61] "近似値"といっても確率的な量だから, ものすごくズレている可能性も (小さいが) あることに注意.

を赤い点線で,「期待値 ±2× 標準偏差」の範囲

$$\mu - \frac{2\sigma}{\sqrt{n}} \quad \sim \quad \mu + \frac{2\sigma}{\sqrt{n}}$$

を青い点線で描いてみよう.

前節のプログラムでは, 1 の目の相対度数が $\frac{1}{6}$ に近付いているように見えなくはないが, それほどはっきりしなかった. ここでは, 3000 回の試行を 10 回行い, それらを図示してみよう.

例題 15.3　　**ex15.3**

```
import random as rd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

n = 3000
time = [0] * 30
freq = [0] * 30

for j in range(0, 10):
    sum = 0
    for i in range(0, n):

        x = rd.randint(1, 6)
        if (x == 1):
            sum = sum + 1

        if ((i+1) % 100 == 0):
            time[i // 100] = i+1
            freq[i // 100] = sum / (i+1)

    plt.plot(time, freq, 'o-', alpha=0.3)

plt.ylim(0.10,  0.23)
t = np.linspace(100, 3000, 30)
s = np.sqrt(5/t) / 6
plt.plot (t, 1/6 + s,   'r:', lw = 3)
plt.plot (t, 1/6 - s,   'r:', lw = 3)
plt.plot (t, 1/6 + 2*s, 'b:', lw = 3)
plt.plot (t, 1/6 - 2*s, 'b:', lw = 3)
```

A ここまでは, ex15.2 と同じ.

B 試行回数 `t` は 100 回から, 3000 回までの範囲に 30 個の点を取り図示する. 標準偏差 `s` は

$$\sqrt{\frac{5}{36t}} = \frac{1}{6}\sqrt{\frac{5}{t}}$$

で与えられる.

C `r` と `b` は赤と青を意味し, `lw` で線の太さを指定している.

図 75　例題 15.3 (ex15.3) の解説: 大数の法則.

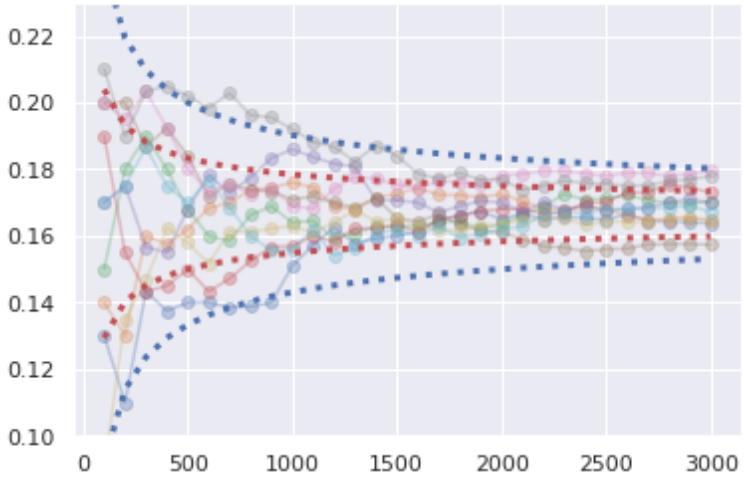

図 76　横軸は試行回数 N, 縦軸は 1 の目の相対度数 r/N を表す. $1/6 = 0.1666..$ の周りにばらついているが, おおよそ「期待値 $\pm 2\times$ 標準偏差」(青い点線) の間に収まっている.

16　確率と統計: 相関係数と回帰直線　(応用編 9)

16.1　母相関係数

相関係数は第 5 章でも簡単に扱ったが, ここでは推測統計の観点から考えてみよう. 確率変数 X と Y の母相関係数 r_{xy} とは

$$r_{xy} = \frac{s_{xy}}{\sigma_x \sigma_y}$$

で定義される. ここで, s_x, s_y は X, Y の母標準偏差であり, s_{xy} は母共分散である:

$$\begin{aligned} s_x &= E[(X-\mu_x)^2] \\ s_y &= E[(Y-\mu_y)^2] \\ s_{xy} &= E[(X-\mu_x)(Y-\mu_y)] \end{aligned}$$

ここで, μ_x と μ_y は母平均 (期待値) である.

この母相関係数に対する標本値として, 標本相関係数が次のように定義される:

$$R_{xy} = \frac{S_{xy}}{S_x S_y}$$

当然, この標本相関係数が母相関係数の推計値になっている.

ここでは, Z_{12} を 12 面体サイコロの目の値, Z_{06} を 6 面体サイコロ (普通のサイコロですね) の目の値とする. さらに, 確率変数 X, Y を

$$\begin{aligned} X &= Z_{12} + Z_{06} \\ Y &= Z_{12} - Z_{06} \end{aligned}$$

と定義する. すると, X, Y の期待値は

$$\begin{aligned} \mu_x &= E[Z_{12} + Z_{06}] = E[Z_{12}] + E[Z_{06}] = \frac{13}{2} + \frac{7}{2} = 10 \\ \mu_y &= E[Z_{12} - Z_{06}] = E[Z_{12}] - E[Z_{06}] = \frac{13}{2} - \frac{7}{2} = 3 \end{aligned}$$

同様に分散も Z_{12}, Z_{06} が独立であることから,

$$\begin{aligned} \sigma_x &= V[Z_{12} + Z_{06}] = V[Z_{12}] + V[Z_{06}] = \frac{143}{12} + \frac{35}{12} = \frac{89}{6} \\ \sigma_y &= V[Z_{12} - Z_{06}] = V[Z_{12}] + V[Z_{06}] = \frac{143}{12} + \frac{35}{12} = \frac{89}{6} \end{aligned}$$

一方, 母共分散 s_{xy} は

$$s_{xy} = E[XY] - \mu_x \mu_y = E[Z_{12}^2 - Z_{06}^2] - 30 = E[Z_{12}^2] - E[Z_{06}^2] - 30 = \frac{205}{4} - \frac{35}{12} - 30 = \frac{55}{3}$$

以上より, 母相関係数は

$$r_{xy} = \frac{\frac{55}{3}}{10 \times 3} = \frac{11}{18} \fallingdotseq 0.611...$$

16.2　標本相関係数のシミュレーション

次に相関係数を求めてみよう．相関係数の計算には, 個々の変量の期待値や分散が必要なので, 16.3 で作成した関数 `f_describe()` を利用したい．しかし, また関数を 1 から書くのは面倒である (コピペすれば良いが, それでもずいぶん長いプログラムになってしまう)．そこで, `f_describe()` を `cmath.py` というファイルに保存して, このファイルを `import` することを考えよう．

例題 16.1　　**ex16.1**

```
import random as rd
import cmath as cm

n = 100
z06 = [0] * n
z12 = [0] * n
x = [0] * n
y = [0] * n

for i in range(0, n):
    z06[i] = rd.randint(1,  6)  # 6 面サイコロ
    z12[i] = rd.randint(1, 12)  # 12 面サイコロ
    x[i] = z12[i] + z06[i]  # 目の和
    y[i] = z12[i] - z06[i]  # 目の差

mx, vx = cm.f_describe(x, n)
my, vy = cm.f_describe(y, n)
sxy, rxy = cm.f_cv_cc(x, y, n)
print (f'X: 標本平均 {mx:4.1f}, 標本分散 {vx:4.1f}')
print (f'Y: 標本平均 {my:4.1f}, 標本分散 {vy:4.1f}')
print (f'標本相関係数: {rxy:5.3f}')
```

A pr6.4 で作成した標本平均, 標本分散, 相関係数を求める関数を cmath.py というファイルにまとめた．それを最初に import している．

図 77　例題 16.1 (ex16.1) の解説: 標本相関係数.

ex16.1 を実行すると以下のようになる:

実行結果

```
X: 標本平均  9.8, 標本分散 17.4
Y: 標本平均  3.1, 標本分散 17.0
標本相関係数: 0.640
```

練習　以下の問に答えよ.

(1) numpy の関数 `np.corrcoef(x, y)` を使って, 結果が同じであることを確認せよ.

16.3　散布図

相関係数は 2 変量の関係性を調べるときに良く使用される統計量だが, 2 変量間の直線的な関係性の指標である (回帰直線の所で解説する). しかし, 2 つの変量間に 2 次関数的に関係性があるような場合もある. 相関係数ではそのような関係性を見落とす可能性が高い. そこで, 相関係数を用いる場合は, 散布図も同時に作成して, どのような関数関係がありそうかを確かめることが重要である. そこで, 次に散布図を作成してみよう.

例題 16.2　　**ex16.2**

```
import random as rd
import cmath as cm

n = 100
z06 = [0] * n
z12 = [0] * n
x = [0] * n
y = [0] * n

for i in range(0, n):
    z06[i] = rd.randint(1,  6)  # 6 面サイコロ
    z12[i] = rd.randint(1, 12)  # 12 面サイコロ
    x[i] = z12[i] + z06[i]  # 目の和
    y[i] = z12[i] - z06[i]  # 目の差

# 相関係数の計算
sxy, rxy = cm.f_cv_cc(x, y, n)
print (f'標本相関係数: {rxy:5.3f}')

# 散布図の描画
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.plot(x, y, 'o')
```

A 相関係数の計算に, numpy の関数を利用した.

B 'o' とすると円形のシンボルでプロットする. 's' や 'd' も試してみよ.

図 78　例題 16.2 (ex16.2) の解説: 標本相関係数.

ex16.2 を実行すると以下のようになる:

実行結果

```
標本相関係数：0.672
```

また, 次のようなグラフが表示される:

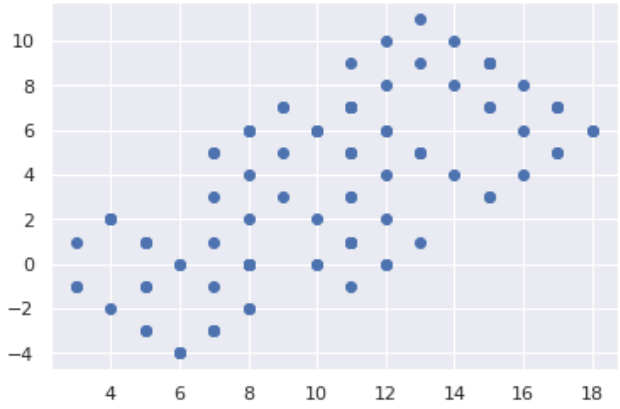

図 79　横軸はサイコロの目の和 x, 縦軸は目の差 y を表す. やや, 右肩上りになっている様子が見られる. 点の数が 100 個未満であることに注意. プロットが重なっているのである.

16.4　2 次元ヒストグラム

散布図は 2 変量の関係性を調べる際には大変便利だが, データが多い場合や, 先の例のようにデータが離散的な値しか取らない場合には, プロットした点やシンボルが重なってしまい, どのような値を取り易いのか分かりにくくなる. そのような場合には, 2 次元ヒストグラムが便利である. ここでは. `histgram2d` 関数を用いて 2 次元ヒストグラムを作成する.

例題 16.3　　**ex16.3**

```
import random as rd

n = 100
z06 = [0] * n
z12 = [0] * n
x = [0] * n
y = [0] * n

for i in range(0, n):
    z06[i] = rd.randint(1,  6)  # 6 面サイコロ
    z12[i] = rd.randint(1, 12)  # 12 面サイコロ
    x[i] = z12[i] + z06[i]  # 目の和
    y[i] = z12[i] - z06[i]  # 目の差

# 散布図の描画
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.hist2d(x, y, bins=17, cmap='Blues')
plt.colorbar()
```

A `bin=17` で 17×17 個の階級を用意している. `cmap` は色指定.

B `colorbar` は図の右端にある「どの色がどの数値に対応するか」を示すカラーバーを表示させる.

図 80　例題 16.3 (ex16.3) の解説: 2 次元ヒストグラム.

ex16.3 を実行すると次のような図が得られる:

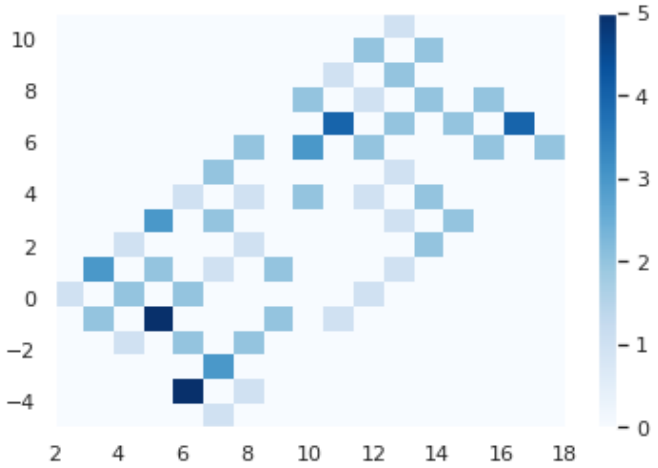

図 81　横軸は試行回数 N, 縦軸は 1 の目の相対度数 r/N を表す. 試行回数が増えるに従い, $1/6 = 0.1666..$ に近づいていくように見えなくもないが, 少し分かりにくい.

16.5　回帰直線

2 変量データの間の関係を 1 本の直線 (1 次関数) で表わすことを考える. このような直線を回帰直線と呼ぶ. この回帰直線は

$$y = Ax + B$$

ただし, A, B は次式で与えられることが示される:

$$A = \frac{S_{xy}}{S_x^2}$$

$$B = \overline{Y} - A\overline{X}$$

例題 16.4　　**ex16.4**

```
import random as rd
import numpy as np
import cmath as cm

n = 100
z06 = [0] * n
z12 = [0] * n
x = [0] * n
y = [0] * n

for i in range(0, n):
    z06[i] = rd.randint(1,  6)  # 6 面サイコロ
    z12[i] = rd.randint(1, 12)  # 12 面サイコロ
    x[i] = z12[i] + z06[i]  # 目の和
    y[i] = z12[i] - z06[i]  # 目の差

mx, vx = cm.f_describe(x, n)
my, vy = cm.f_describe(y, n)
sxy, rxy = cm.f_cv_cc(x, y, n)
A = sxy  / vx
B = my - A * mx

# 散布図と回帰直線の描画
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
plt.plot(x, y, 'o')
x2 = np.linspace (2, 18, 100)
plt.plot (x2, A*x2 + B, '-', lw=5, alpha=0.5)
```

A 回帰直線の傾き `A` と, 切片 `B` を求めている

B 回帰直線 $y = Ax + B$ の表示. 線を太くし (`lw = 5`), プロットと重ならないように, 透過するよう指定している (`alpha=0.5`).

図 82　例題 16.4 (ex16.4) の解説: 標本相関係数.

ex16.4 を実行すると次のような図が得られる:

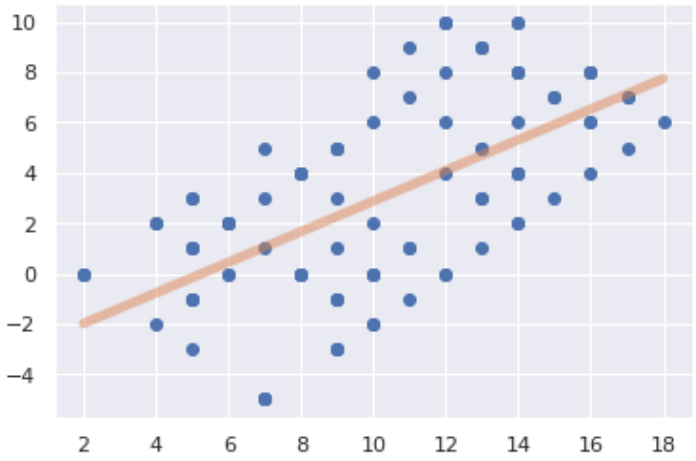

図 83　回帰直線を表示した所. 予想と一致しているだろうか? 違和感を感じませんか?

練習　以下の問に答えよ.

(1) numpy にも共分散を求める関数 `cov(x, y)` がある. 使い方を調べ, 同じ結果になることを確認せよ.