数学科のための Python プログラミング

佐々木格　(信州大学理学部)

2026 年 7 月 31 日

目次

概要

本書は，数学科の学生を対象とした Python の入門テキストです。Python は洗練されたデザインを持ち，直感的で読みやすいコードが書けることが大きな特徴です。数学の勉強で登場する様々な概念を，手元の PC で気軽に計算・実験できるようになることを目指しています。また，パソコンでの数値計算を通じて，数学そのものの理解を深めることも目的としています。

前半では，CUI の基本操作から，変数・制御構文・関数・データ構造といったプログラミングの基礎文法を解説します。後半では，数式処理パッケージ SymPy による代数計算やテイラー展開，Matplotlib による関数の可視化，NumPy による線形代数演算，微分方程式の数値シミュレーション（ルンゲ＝クッタ法），精度保証付き数値計算ライブラリ Arb の活用，そしてモンテカルロ法まで，数理計算の様々なトピックを紹介します。

なお，Python や本資料で扱う各種ライブラリはすべて自由ソフトウェアとして公開されており，自身の PC や Jupyter Notebook などのブラウザ環境で手軽に利用することができます。

第 I 部
Python の基礎

1　必要なコンピュータの操作

この資料は Windows，macOS，Linux での操作を想定して書かれています。これらパソコンで広く使われる OS にはそれぞれバリエーションがあり，操作法が若干異なります。以下の解説はお使いの環境によって細部が異なりますので，適宜読み替えて理解してください。

主な用語や記号の読み替え例：

- 端末：Windows PowerShell，ターミナル（macOS/Linux）
- テキストエディタ：Visual Studio Code (VS Code)，メモ帳など[*1]
- ディレクトリ：フォルダ
- 記号：\ (バックスラッシュ) と ￥ (円マーク) は同じものとして扱います。

以下，Python3 と Visual Studio Code が適切にインストールされているものとして解説を行います。

1.1　GUI と CUI

人間とコンピューターとの間で情報をやりとりする手段を「ユーザーインターフェース」といいます。操作の方法には主に次の 2 種類があります。

- **グラフィカル・ユーザーインターフェース (GUI)**：
  ディスプレイに表示されたアイコンやボタンをマウスでクリックして操作する。直感的だが，開発者の気分で変更されるため，手順の共有や自動化が難しい。
- **キャラクター・ユーザーインターフェース (CUI)**：
  キーボードによる文字入力（コマンド）によって操作する。コマンドを覚える必要はあるが，手順を正確に記録・共有でき，自動化や並列処理にも向いている。

現在のパソコンでは，Windows や macOS のデスクトップ（GUI）上で「ターミナル」や「PowerShell」といったアプリケーションを起動することで，CUI による操作を行うことができます。

CUI の操作はコマンド名が変更になることがほとんどないため，操作手順をメモとして保存しておけば，数年後でも同じように動作させることが可能です。これは「再現性」が重視される科学の世界において，非常に大きなメリットとなります。

1.2　CUI の起動方法

Windows では標準的に **Windows PowerShell** を使用します。macOS や Linux では **ターミナル（端末）** を使用します。起動方法は以下の通りですが，OS のバージョン等によって細部が変更されることがあります。

■Windows：

(1) **スタートメニューから：** Windows キーを押し，「`powershell`」と入力して `[ENTER]` を押す。
(2) **フォルダから直接：** 作業したいフォルダ内の何もない場所で**右クリック**（または **Shift ＋ 右クリック**）し，「ターミナルで開く」あるいは「PowerShell ウィンドウをここで開く」を選択する。

[*1] MS Word などは不可。文字装飾のための余計な情報が書き込まれ，プログラムが正しく動作しないためです。

■macOS (ターミナル)

- Command + スペース を押し，検索欄に「terminal」と入力して [ENTER] を押す。または，Launchpad 内の「その他」フォルダなどにある「ターミナル」をクリックする。

■Linux (端末 / ターミナル)

- スタートメニューから「端末」や「ターミナル」を探して起動する。または，ファイルマネージャーの右クリックメニューから「端末で開く」を選択する。

基本的に macOS と Linux のターミナルの操作法は共通です。Windows の PowerShell も，基本的なコマンドについてはこれらと共通の操作ができるようになっています。

1.3 CUI における操作

Windows の PowerShell や macOS/Linux のターミナルを起動すると，次のような表示が現れます。

■ Windows (PowerShell) の例：

```
PS C:\Users\sasaki>
```

■ macOS / Linux の例：

```
MacBook-Air:~sasaki$
or
sasaki@HP:~$
```

最後の sasaki の部分はユーザー名やパソコン名になっているはずです。

コンピュータのデータは「ファイル」という単位で保存されており，それらは「ディレクトリ（フォルダ）」という階層構造で管理されています。端末を開いた直後の位置は，通常「ホームディレクトリ」と呼ばれます。作業を行う際は，目的のファイルがあるディレクトリへ移動し，コマンドを入力して処理を実行します。

基本的なコマンド一覧

PowerShell では，多くのコマンドが macOS / Linux と共通化（エイリアス設定）されています。

機能	Windows (PowerShell)	macOS / Linux
中身を表示	ls (または dir)	ls
現在の場所を表示	pwd	pwd
ディレクトリを移動	cd *dir* 名	cd *dir* 名
一つ上の階層へ移動	cd ..	cd ..
ディレクトリ作成	mkdir *dir* 名	mkdir *dir* 名
ホームへ戻る	cd	cd
ジョークコマンド	—	sl

1.3.1 講義用ディレクトリの作成

講義で作成するファイルを入れるためのディレクトリ dataproc1 を作成します。

まずは，前の項の表を参考にして **ホームディレクトリ** へ移動してください。次に，デスクトップへ移動するために以下のコマンドを入力します。

`cd Desktop`　または　`cd デスクトップ`

移動できたら，そこでディレクトリ作成コマンドを入力します。

```
mkdir dataproc1
```

入力後，画面上のデスクトップに `dataproc1` というフォルダが出現したか確認してください。

■注意：OneDrive の影響について　Windows では，Microsoft の仕様変更により，デスクトップの場所が標準的な場所から移動していることがよくあります。もし「`cd Desktop`」で移動した先で作成したはずのディレクトリが見当たらない場合は，以下を試してみてください。

- `cd OneDrive\Desktop`　（OneDrive 同期が有効な場合）
- `cd OneDrive\ デスクトップ`　（上記かつ日本語名の場合）

`ls`（または `dir`）と入力して，自分が知っているファイル名が表示されるか確認しながら進むのがコツです。

2　Python プログラムの実行手順

この資料では Python プログラムを実行する方法として次の 2 つを紹介します。

(1) Python のプログラムが書かれたファイルを作成して端末から実行する
(2) インタラクティブシェルを使う

2.1　準備：拡張子の確認

デスクトップに作成したフォルダ（`dataproc1`）に，好みのテキストエディタで `test.txt` というファイルを作成してみましょう。

ファイル名の末尾，ドット（`.`）以降の文字は **拡張子** と呼ばれ，ファイルの種類を識別するために使われます。拡張子には次のようなものがあります：

`.txt`　：プレーンテキストファイル（中身は文字のみ）
`.pdf`　：PDF ドキュメント（文書閲覧用）
`.py`　：Python プログラム（Python 実行用）
`.tex`　：LaTeX ソースファイル（数式などの文書作成用）
`.jpg`　：JPEG 画像ファイル（写真データなど）

■重要：拡張子の表示設定　Windows の初期設定では，これらの拡張子が非表示になっていることが多く，トラブルの原因になります。エクスプローラーの「表示」メニュー（Windows 11 なら「表示」→「表示」→「ファイル名拡張子」）から，必ず **拡張子を表示する設定** に変更してください。

メモ帳などでファイルを保存する場合，`hello.py` というファイル名で保存したつもりが，実は `hello.py.txt` になっており，コマンド `python hello.py` では「ファイルが見つからない」とエラーが出て実行できないことがよくあります。

また，Python はファイルの中身さえ正しければ `.txt` という名前のままでも（正しいファイル名を指定す

れば）実行自体は可能です。しかし，拡張子が正しくないと **VS Code でプログラムに色がつきませんし**，どのファイルがプログラムなのか人間にもコンピュータにも判別しにくくなります。Python プログラムは必ず `.py` で保存してください[*2]。

2.2 最初のプログラム (Hello World)

プログラムは **テキストエディタ** を使って書きます。テキストエディタには，Windows 標準の「メモ帳」や macOS の「テキストエディット」など様々な種類がありますが，プログラムを効率よく書くための「コードエディタ」と呼ばれる高機能なものもあります。ここでは，その代表格である **Visual Studio Code (VS Code)** を使った手順を解説します。

■1. プログラムファイルの作成

1. VS Code を起動します。
2. 上部メニューの「ファイル」→「新しいテキストファイル」を選択します。
3. 「ファイル」→「名前を付けて保存」を選択し，保存場所をデスクトップの `dataproc1` フォルダに，ファイル名を `hello.py` として保存します。
4. エディタ上に以下の 1 行を入力し，必ず **保存（Ctrl + S）** してください。

```
1 print('Hello World!')
```

■2. プログラムの実行 作成したプログラムを **端末（PowerShell やターミナル）** から実行します。

プログラムを実行するために，`dataproc1` フォルダの中身が見えているウィンドウに戻りましょう。フォルダ内の空きスペースで **[Shift] + [右クリック]**（Windows 11 の場合は右クリックのみでも可）し，「ターミナルで開く」または「PowerShell ウィンドウをここで開く」を選択します。

以下は，Windows の PowerShell でプログラムを実行したときの例です[*3]。

```
1 PS C:\Users\sasaki\Desktop\dataproc1> python hello.py
2 Hello World!
```

ここで入力した `python hello.py` というコマンドは，「python というプログラム実行機を使って，`hello.py` というファイルに書かれた命令を読み上げなさい」という指示をコンピュータに与えています。

■もし実行できなかったら 「ファイルが開けません」といったエラーが出る場合は，今いるディレクトリに `hello.py` が存在しない可能性が高いです。

- **`ls` コマンドで確認：** 端末に `ls` と入力し，表示される一覧に `hello.py` があるか確認してください。
- **不一致の解消：** 一覧にない場合は，「保存した場所」と「今開いている端末の場所」がズレています。もう一度，正しいフォルダから端末を起動し直してください。

2.3 日本語の扱いと注意点

Python3 では，特別な設定なしでプログラム内に日本語（漢字やひらがな）を使うことができます。次のプログラムを作成して，実行してみましょう。

[*2] 正確なファイル名を付けることはコンピュータ操作の基本です。今後，提出された課題のファイル名に不備がある場合は大きく減点対象とします。

[*3] macOS や Linux の場合は，コマンドを `python3` に読み替えてください。

- ファイル名：**hellojp.py**

```
print('こんにちは，Pythonの世界へ！')
```

```
> python hellojp.py
こんにちは，Pythonの世界へ！
```

ここで，’ ’は半角でなければりません。

2.3.1 文字コード（UTF-8）について

コンピュータ内部で文字を表現するためのルールのことを「文字コード」といいます。Python は **UTF-8** という世界標準の文字コードをデフォルトで使用します。

VS Code などの現代的なエディタを使っていれば自動的に UTF-8 で保存されますが，古いエディタや特殊な設定で「Shift-JIS」などの異なる文字コードで保存してしまうと，実行時にエラーが出たり，文字化けが起こったりします。

2.3.2 コメントアウト

プログラムの中にメモを書き残したいときは，記号「#」を使います。# 以降の文字はプログラムの実行時には無視されます。

```
print('こんにちは')  # ここは実行に影響しないコメントです
# この行全体が無視されます
```

2.3.3 【最重要】全角スペースの罠

プログラミングにおいて，最も注意しなければならないのが **全角スペース** です。

プログラム内に「全角スペース」を
混ぜてはいけません！

見た目では半角スペース「 」と全角スペース「　」の区別がつきにくいですが，コンピュータにとっては全く別物です。プログラムの途中に全角スペースが混じると，原因不明のエラーに悩まされることになります。

VS Code などのエディタを使用すると，全角スペースが四角い枠で囲われたり，色がついて表示されたりするため，この間違いをすぐに発見できます。これが「メモ帳」ではなくプログラミング用のエディタを推奨する大きな理由の一つです。

2.4 Python インタラクティブシェル

Python インタラクティブシェル（対話モード）は，入力したプログラムをその場で 1 行ずつ実行するシステムです。電卓のように手軽に計算を試すことができます。

起動するには，端末から `python` と入力します[*4]。

```
PS ...\dataproc1> python
Python 3.10.x ...
Type "help", "copyright" or "license" for more information.
```

[*4] macOS や Linux の場合は `python3` です。

```
>>>
```

カーソルが最後の行で点滅して入力を受け付けています。ここで計算をしてみましょう：

```
>>> 1+3
4
>>> 5-3
2
>>> 4*3
12
>>> 9/4     # スラッシュは割り算
2.25
>>> 10/3
3.3333333333333335    # 値は厳密ではない場合がある！
>>> 18//5   # スラッシュ2つで「商」
3
>>> 12%5    # 割り算の「余り」
2
>>> 2**10   # アスタリスク2つで「冪（べき）乗」：2の10乗
1024
```

■演算上の注意点

- **割り算の記号：** 割り算には **スラッシュ** (/) を使います。
- **冪乗の書き方：** 多くの言語では ^ を使いますが，Python では ** を使います。2^{10} は 2**10 と書きます。
- **精度の限界：** 10/3 の結果が最後の方でズレているように，コンピュータの数値計算（浮動小数点数）には限界があることを知っておいてください。

■SageMath との違い（予告）　後半で解説する Sage では，数学者が使いやすいように「^」で冪乗を計算できたり，「5/3」が小数ではなく $\frac{5}{3}$ という厳密な分数（有理数）として扱われたりするようにカスタマイズされています。

2.4.1 全角スペースの実験

全角スペースがどのような挙動を示すか，実際に試してみましょう。

```
>>> print('あ　い')      # 「あ」と「い」の間は半角スペース
あ い
>>> print('う　え')      # 「う」と「え」の間は全角スペース
う　え
```

上のように，引用符（'）で囲まれた「文字列」の中であれば，全角スペースもただの文字として扱われます。しかし，プログラムの構造に関わる部分（数式やコマンドの間）に全角スペースが入ると，即座にエラーとなります。

```
>>> 4+6         # 4+6を計算
10
>>> 4 +6        # 4と+の間に半角スペース（これはOK）
10
>>> 4　+6       # 4と+の間に全角スペースを混入させた
```

```
  File "<stdin>", line 1
    4  +6
     ^
SyntaxError: invalid character in identifier
```

エラーメッセージに SyntaxError: invalid character（識別子に無効な文字がある）と表示されたら，まずは **全角スペースの混入** を疑ってください。

■インタラクティブシェルの終了　インタラクティブシェルを終了して元の画面に戻るには，exit() と入力するか，ショートカットキー

- **Windows:** Ctrl + Z を押した後に Enter
- **macOS / Linux:** Ctrl + D

を使用します。

```
>>> exit()                          # exit() と打って Enter
PS C:\Users\...\dataproc1>          # 元のプロンプトに戻る
```

3　変数，予約語，文字列，数値，データの型

プログラムで扱うデータには，数値だけでなく文字など様々な種類があります。ここでは，データを扱うための基本的なルールを学びます。

3.1　変数

プログラムの中で，数値や文字などのデータを一時的に保存しておくための「箱」のようなものを **変数**（variable）と呼びます。まずは，インタラクティブシェルを用いて変数の挙動を確認しましょう。

```
>>> a = 6              # 変数 a に 6 を代入
>>> a                  # a の内容を表示
6
>>> a = 8              # 変数 a の値を 8 に変更
>>> a
8
>>> b = 3              # 変数 b に 3 を代入
>>> b
3
>>> a + b
11
```

定義されていない（値を代入していない）変数名を呼び出すと，次のようなエラーが発生します。

```
>>> c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

このように，コンピュータが知らない名前を使おうとすると NameError が出ます。

変数名は 1 文字である必要はありません。

```
>>> ame = 4
```

```
>>> mikan = 5
>>> ame + mikan
9
```

■変数名の命名ルール 変数名はある程度自由に決めることができますが，コンピュータが正しく認識するために次のルールを守る必要があります。

- **使える文字：** アルファベット（a--z，A--Z），数字（0--9），アンダーバー（_）。
- **始まりの文字：** 必ずアルファベットかアンダーバーから始めなければなりません。**数字から始めることはできません**（例：7-Eleven や 8loom は不可）。
- **大文字と小文字の区別：** ame と AME は，全く別の変数として扱われます。
- **予約語の禁止：** 次に説明する「予約語」を変数名として使うことはできません。

3.2 予約語

次の単語は Python の文法上，特別な意味を持つ「予約語（Keyword）」です。これらは変数名として使うことができません。

Python の予約語（代表的なもの）

```
False   None    True    and     as      assert  break
class   continue def    del     elif    else    except
finally for     from    global  if      import  in
is      lambda  nonlocal not    or      pass    raise
return  try     while   with    yield
```

VS Code などのエディタでは，これらの単語を入力すると自動的に色が変わるため，一目で判別できるようになっています。

3.3 セミコロンによる文の区切り

Python では通常，1 行に 1 つの文（命令）を書きますが，セミコロン ; を使うことで，複数の文を 1 行にまとめて記述することもできます：

```
>>> a = 5; b = 3; c = a + b; c
8
```

これは便利ですが，多用するとプログラムが読みづらくなるため，使いどころには注意が必要です。

3.4 代入演算子

Python において記号 = は，右辺のデータを左辺の変数へ格納する「代入（assignment）」を意味し，**代入演算子**と呼ばれます。数学的な「等号（等しい）」とは意味が異なることに注意してください。

例えば，数学的に「偽」である命題 $3 = 4$ も，Python では単に「意味をなさない文」としてエラーになります。

```
>>> 3 = 4
  File "<stdin>", line 1
SyntaxError: cannot assign to literal
```

また，変数 a の値を 5 増やしたい場合は，次のように記述します。

```
1 >>> a = 3
2 >>> a = a + 5    # aの値を5増やす
3 >>> a
4 8
```

2 行目の a = a + 5 という表記に違和感があるかもしれませんが，これは「現在の a の値（3）に 5 を足した結果（8）を，改めて a に格納する」という処理を意味しています。

また，以下の挙動を確認しましょう。

```
1 >>> a = 3
2 >>> b = a     # bに「aの値」を代入する
3 >>> b
4 3
5 >>> a = 4     # 後からaの値を変える
6 >>> b
7 3             # bの値は3のまま変わらない
```

変数を「箱」に例えると分かりやすくなります。a = 3 によって箱 a に数値 3 が入ります。次に b = a とすると，箱 b には，その時点での箱 a の中身である 3 がコピーされて入ります。したがって，後から箱 a の中身を 4 に書き換えても，箱 b の中身には影響しません。

■複合代入演算子 変数の値を増減させる操作は頻出するため、これを簡潔に書く**複合代入演算子**（+=, -= など）が用意されています。

```
1 >>> a = 4
2 >>> a += 1  # a = a + 1 と同じ
3 >>> a
4 5
5 >>> a -= 2  # a = a - 2 と同じ
6 >>> a
7 3
```

3.5 文字列

ここまでに変数，文字列，数値を扱いました。数値は 65, −3, 9.23 等と表されたもの，文字列は 'Hello' のようにクォーテーションマーク（引用符）で囲まれたもの，変数はそれらのデータを名前を付けて管理するためのものです。それぞれについてもう少し詳しく解説します。

3.5.1 文字列の定義

文字列はシングルクォーテーション（' '）やダブルクォーテーション（" "）で囲んで定義します。

```
1 >>> x = 'hello world'
2 >>> x
3 'hello world'
```

ここで x は変数で，'hello world' が文字列です。

2 つの文字列は + 演算子でつなげることができます。

```
1 >>> aa = 'Alice'     # 変数aaに文字列を代入
2 >>> bb = ' and Bob'
```

```
>>> cc = aa + bb
>>> cc
'Alice and Bob'      # 文字列aaとbbがつながっている
>>> print(cc)
Alice and Bob
```

最後のように文字列を print() すると，外側のクォーテーションがとれた中身が表示されます。

3.5.2 文字列の中での引用符と改行

文字列を定義する際，文字列の中に引用符（クォーテーション）を含めたい場合は，外側と内側で種類を使い分けます。

```
>>> a = "This is a 'pen'."
>>> print(a)
This is a 'pen'.
```

改行を含む文字列を作りたい場合は，改行したい場所に \n（改行コード）を挿入します。

```
>>> a = 'aaa\nbbb\nccc'
>>> a
'aaa\nbbb\nccc'
>>> print(a)       # print()を使うと \n の部分で実際に改行される
aaa
bbb
ccc
```

\n を使わずに，ソースコード上の見た目通りに改行したい場合は，トリプルクォーテーション（""" """ または ''' '''）で囲みます。

```
>>> a = '''aaa
... bbb
... ccc'''
>>> print(a)
aaa
bbb
ccc
```

3.5.3 エスケープシーケンス

バックスラッシュ（\）を使うことで，改行文字や引用符などの特殊な記号を表現できます。これをエスケープシーケンスと呼びます。

記号	意味	役割
\\	\	バックスラッシュそのもの
\'	'	シングルクォーテーション
\"	"	ダブルクォーテーション
\n	行送り	改行（New Line）
\ 改行	⇝	行末の改行を無視する（コードを見やすくするため）

例えば，外側と同じ引用符を文字列内で使いたい場合は，次のように書きます。

```
>>> aa = 'クォーテーションマークは「\'」とか「\"」ですよ'
```

```
>>> print(aa)
クォーテーションマークは「’」とか「"」ですよ
```

↑ LaTeX の色付けの設定がちょっとおかしいけど気にしないで。

3.6 データの型

Python では，データが「どのような種類のものか」を厳密に区別しており，これを **データの型**（data type）と呼びます。

文字列は `str` 型（string）と呼ばれます。

```
>>> type('hello')        # type()関数で型を調べることができます
<class 'str'>            # str型であることを示している
```

数値の型でよく使うものには，**整数型**（`int`）と **浮動小数点数型**（`float`）の 2 種類があります。

```
>>> type(123)            # 123の型は？
<class 'int'>            # 整数型（integer）
>>> type(3.14)           # 3.14の型は？
<class 'float'>          # 浮動小数点数型（floating point number）
>>> a = 3.14             # 変数aに3.14を代入
>>> type(a)
<class 'float'>          # 変数aが指すデータの型が表示される
```

■型の違いによる特性 整数型（`int`）の演算は厳密に行われますが，浮動小数点数型（`float`）の計算では内部的な処理の都合上，ごくわずかな誤差が生じることがあります。では，これらが混ざった計算はどうなるでしょうか？

```
>>> aa = 10              # int型
>>> bb = 3.14            # float型
>>> cc = aa + bb
>>> cc
13.14
>>> type(cc)
<class 'float'>          # 結果はfloat型になる
```

計算の中に一つでも `float` が混ざると，結果も `float` になります。扱う数値がすべて整数の場合に限り（割り算を除いて）厳密な計算が可能ですが，小数を含む場合は誤差の可能性を意識しておく必要があります。一回一回の誤差はごくわずかですが，何万回，何億回と計算を繰り返すようなプログラムでは，誤差が累積して無視できない大きなミス（計算違い）を引き起こすことがあります。

3.7 数値と文字列の変換（型変換）

データには「型」があるため，異なる型同士を計算させようとするとエラーになることがあります。例えば，文字列と数値を足すとどうなるでしょうか？

```
>>> 'Alice' + 1999
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

このように TypeError（型エラー）が発生します。Python にとって，「文字としてのデータ」と「計算するための数値」をそのまま足し合わせることはできません。文字列と数値を連結させたい場合は，数値をあらかじめ文字列に変換しておく必要があります。これを **型変換** と呼びます。

型変換のための関数

- str()：データを文字列に変換する（例：str(123) → '123'）
- int()：数字の文字列を整数に変換する（例：int('123') → 123）
- float()：数字の文字列を小数に変換する（例：float('123.4') → 123.4）

数値 1999 を文字列に変換してから結合すれば，エラーは起きません。

```
>>> aaa = str(1999)      # 数値1999を文字列に変換してaaaに代入
>>> aaa
'1999'
>>> type(aaa)
<class 'str'>            # aaaの型は文字列（str）になっている
>>> 'Alice' + aaa        # 文字列同士なら「+」で結合できる
'Alice1999'
```

逆に，「文字列として読み込んだ数字」を計算に使いたい場合は，数値型に変換します。

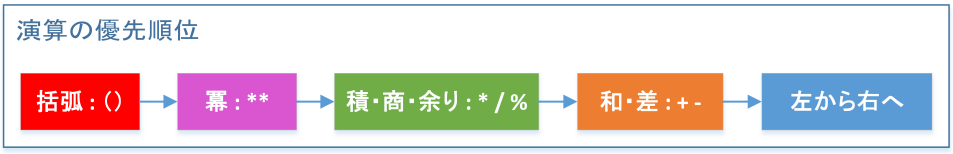
```
>>> aa = '123'           # これは文字列
>>> int(aa) + 7          # 整数に変換すれば計算が可能
130
>>> float(aa)            # 小数として扱うなら float()
123.0
```

3.8　演算子と計算の順序

Python では，数値計算は次の **演算子**（operator）で行います。

記号	意味	例（実行結果）
+	和	10 + 20 → 30
-	差	20 - 10 → 10
*	積	4 * 5 → 20
/	割り算	10 / 5 → 2.0 （常に float 型を返す）
//	商	10 // 5 → 2 （整数部分のみを返す）
%	余り	8 % 5 → 3
**	冪（べき）	2 ** 3 → $2^3 = 8$

括弧で囲んだ部分は，例外なく最優先で計算されます。演算の優先順序は次のようになっています：



例えば，5*6/2**2 は，次のような順序で計算されます。

```
5 * 6 / 2 ** 2 = 5 * 6 / (2 ** 2)   # まず「冪」を計算
               = 5 * 6 / 4
```

```
                = (5 * 6) / 4        # 次に「左側にある乗算」を計算
                = 30 / 4
                = 7.5
```

通常の演算は「左から右」へ計算されますが，**冪計算（**）だけは「右から左」に計算される** という特殊な性質があります。

```
>>> 3 ** 3 ** 3
7625597484987
>>> 3 ** (3 ** 3)      # 3**27 と同じ（上の結果と一致）
7625597484987
>>> (3 ** 3) ** 3      # 27**3 と同じ
19683
```

このように，優先順位のルールはありますが，複雑な計算式では括弧（ ）を使い，誰が見ても計算順序が明確になるように書くほうがよいでしょう。

3.9 指数表記

Python（および多くのプログラミング言語）では，非常に大きな数や非常に小さな小数を表示するときに，e を用いた**指数表記**が使われます。

- **表示例**: `1.2676506002282294e+30`
- **数学的な意味**: $1.2676506002282294 \times 10^{30}$

ここで，e+30 は「10 を 30 乗して掛ける」という意味です。逆に非常に小さい数（0.0001 など）の場合は，`1.0e-4`（$= 1.0 \times 10^{-4}$）のようにマイナスで表示されます。

以下の実行例で，整数（`int`）と浮動小数点数（`float`）の表示の違いを確認しましょう。

```
>>> 2 ** 100        # 整数（int）として表示
1267650600228229401496703205376

>>> 2 ** 100.0      # 浮動小数点数（float）として表示
1.2676506002282294e+30
```

整数はどれほど巨大になっても全ての桁を正確に保持しますが，浮動小数点数は「有効数字」の範囲（約 15〜17 桁）までしか保持せず，それ以降は指数表記によって「おおよその大きさ」として扱われます。

【練習問題】精度の壁

以下の 2 つの計算結果を比較せよ。数学的には $(a+b)-a=b$ が成立するはずだが・・・

```
>>> (2**53 + 1.0) - 2**53
# いくつになったかな？
>>> (2**53 - 2**53) + 1.0
# 上と同じかな？
```

このように，浮動小数点数（`float`）では計算の順序を変えるだけで結果が変わることがあります。巨大な数と小さな数を同時に扱う計算では，特に注意が必要です。

コンピュータは人に比べたら格段に高速で巨大な数の計算をこなしますが，実際にプログラムを作る際には計算はある程度の数値の範囲で行い，さらに誤差を見積もる必要があります。

4　プリント (print)

インタラクティブシェルでは，変数の中身を知りたければ変数名を入力して Enter を押すだけで済みました。しかし，プログラムを「ファイル」に書き込んで実行する場合には，変数名を書いただけでは画面に何も表示されません。

例えば，次のプログラムをファイルから実行しても，画面には何も出力されません。

- ファイル名：**print01.py**

```
1 a = 3
2 a             # 変数名を書くだけでは表示されない
```

ファイルから実行する際に，変数の値を画面に表示するためには print() 関数を使う必要があります。

- ファイル名：**print02.py**

```
1 a = 3
2 print(a)      # 3 が表示される
```

実行結果の例

```
3
```

- ファイル名：**print03.py**

```
1 a = 6
2 b = 4
3 a + b         # 計算は行われるが、結果は表示されない
4 print(a)      # aの内容が表示される
5 print(a + b) # 計算結果が表示される
```

実行結果の例

```
6
10
```

実行結果を見るとわかるように，3 行目の計算は出力されず，print() を使った行だけが画面に表示されます。また，print() を複数回呼び出すと，標準ではその都度「改行」されて表示されます。

■改行させない表示　print() の後に改行をさせたくない場合は，引数に end=" " を追加します。これにより，改行の代わりに指定した文字（この場合は半角スペース 1 つ）が最後に挿入されます。

- ファイル名：**print04.py**

```
1 print(6, end=" ")  # 改行せず，最後にスペースを入れる
2 print(4)           # そのまま横に続きが表示される
```

実行結果の例

```
6 4
```

また，次のようにカンマ（,）で区切ることで，文字列や変数を一度に並べて表示することができます。

- ファイル名：**print05.py**

```
1 name = 'Alice'
2 age = 19
3 print('名前は', name, 'で，年齢は', age, '歳です。')
```

実行結果の例

```
名前は Alice で，年齢は 19 歳です。
```

カンマで区切った場所には自動的に半角スペースが挿入されます。

一方で，次のように文字列を + でつなげてから表示する方法もあります。この場合，数値型は str() で文字列に変換しなければならず，また自動的なスペース挿入も行われません。

- ファイル名：**print06.py**

```
1 name = 'Alice'
2 age = 19
3 print('名前は' + name + 'で，年齢は' + str(age) + '歳です。')
```

実行結果の例

```
名前は Alice で，年齢は 19 歳です。
```

4.1 練習問題

4.1.1 問題：ほとんど整数の計算 1

$e^{\pi} - \pi$ は整数に非常に近い値をとることが知られています。実際，$e^{\pi} - \pi = 19.9990999792\ldots$ です[*5]。このような，無理数から作られる数で整数に極めて近い値を「ほとんど整数（Almost integer）」と呼びます。

さて，次の値もまた「ほとんど整数」の一種です。

$$e + \pi + e\pi + e^{\pi} + \pi^{e} \tag{1}$$

この値を，以下の近似値を用いて計算し，表示するプログラムを作成しなさい。

- e = 2.71828182846
- pi = 3.14159265359

■作成上の注意

1. ファイル名は problem01.py とすること。
2. 実行した際，画面に計算結果の数値が表示されるようにすること（print を忘れないこと）。
3. 端末（コマンドプロンプトや PowerShell）から python problem01.py と入力して実行を確認すること。

ちなみに，この量がなぜほとんど整数になるのかについては，（おそらく）まだ合理的な説明が得られていない未解決の謎です。

4.2 誤差の問題

以下の計算を順に実行し，実行結果を確認せよ。

```
1 >>> 2**53
2 >>> (2**52 + 1) - 2**52
3 >>> (2**53 + 1) - 2**53
4 >>> (2**52 + 1.0) - 2**52
```

[*5] この事実は 1986 年ごろに見つかっていたが，2023 年 9 月にこれがヤコビのテータ関数に関するある級数から説明されることが（youtube のコメントで）わかった。by Wikipedia

```
>>> (2**53 + 1.0) - 2**53
>>> (2**53 - 2**53) + 1.0
```

この実行結果から，以下の重要な事実を読み取ることができます。

- **整数（int）の計算**: 2**52 や 2**53 に 1 を足しても正確に計算されます。Python の整数には精度の限界がないためです。
- **浮動小数点数（float）の「53 ビット」の壁**: コンピュータの float（倍精度浮動小数点数）は，内部的に約 53 ビットの有効数字を保持しています。
  - 2^{52} までは、1.0 という最小単位を保持する余裕があります。
  - 2^{53} に達すると，有効数字の桁が足りなくなり，1.0 を足そうとしても無視されてしまいます（これを**情報落ち**と呼びます）。

コンピュータは便利な道具ですが「理論上の数学」と「有限のコンピュータ」の差を常に意識する必要があります。

5　様々なデータの形式

5.1　リスト (list)

リストとはいくつかのデータ (要素) の集まりです。まずはリストを作る事から始めましょう。

```
>>> a = ['Alice', 'Bob', 2,3,8]    # リストを定義して変数aに入れる
>>> a                              # aの内容を確認
['Alice', 'Bob', 2,3,8]
```

当然ですが，リストの型は list です。

```
>>> type(a)                        # aの型を確認
<class 'list'>                     # aの型はリスト
```

続いて，リストの成分を呼び出すには次のようにします：

```
>>> a[0]                           # aの第1成分
'Alice'
>>> a[-1]                          # aの最後の成分
8
>>> a[3]                           # aの4番めの成分
3
>>> a[-2]                          # aの最後から2番めの成分
3
```

上のようにリストの成分の番号は 0 から始まります。存在しない成分を呼び出すとエラーになります：

```
>>> a[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

スライスという機能を使って，リストの一部を切り出すことができます：

リストからの要素の取り出し（スライス）

- a[:n] は先頭から n 個の要素（インデックス 0 から $n-1$）
- a[n:] はインデックス n 以降のすべての要素
- a[-n:] は末尾から n 個の要素
- a[n:m] はインデックス n から $m-1$ までの要素

具体的なリストを作ってスライスしてみましょう：

```
>>> a = ['a','b','c','d','e','f','g','h']
>>> a[:3]
['a', 'b', 'c']
>>> a[-4:]
['e', 'f', 'g', 'h']
```

次にリストを足すと（+）、結合された新しいリストが返されます：

```
>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c']
>>> c = a + b
>>> c
[1, 2, 3, 'a', 'b', 'c']
```

リスト自身への要素の追加は append() という「メソッド」を使って行います。また、要素の値を直接書き換えることも可能です。

```
>>> a = [1, 2, 3]
>>> a.append('Alice')  # 末尾に追加
>>> a
[1, 2, 3, 'Alice']
>>> a[0] = 'Bob'          # 0番目を書き換え
>>> a
['Bob', 2, 3, 'Alice']
```

要素を削除するには pop() や remove() を使います。

```
>>> aa = ['a', 'b', 'c', 'd', 'e', 'b', 23, 8, 13]
>>> aa.pop()          # 末尾の要素を取り除いて返す
13
>>> aa.pop(3)         # 3番目（'d'）を取り除く
'd'
>>> aa.remove('b')  # 最初に見つかった 'b' を削除
>>> aa
['a', 'c', 'e', 'b', 23, 8]
```

連続する整数からなるリストは，range() 関数を使うと便利です。

```
>>> list(range(5))          # 0から5の手前まで（5までじゃないよ！！）
[0, 1, 2, 3, 4]
>>> list(range(3, 6))       # 3から6の手前まで
[3, 4, 5]
>>> list(range(-2, 4))      # -2から4の手前まで
[-2, -1, 0, 1, 2, 3]
```

5.2 タプル (tuple)

タプル（tuple）は，数学における「組」や「順序対」に相当します。リストと似ていますが，最大の違いは一度作成すると**内容を変更できない**（Immutable：イミュータブルという）という点です。タプルは要素を丸括弧（ ）で囲んで作成します。

```
>>> a = (3, 7, 'abc')      # タプルの定義
>>> type(a)
<class 'tuple'>
>>> a.pop()                # 変更を試みるとエラーになる
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'pop'
```

数値，文字列，タプルなどは Immutable なデータです。プログラムの中で意図せず値を書き換えられたくない重要なデータを扱うために，このような型が用意されています。

5.3 辞書 (dictionary)

キー（key）と値（value）の対応関係を保持するデータ形式を辞書（dict）と呼びます。要素は波括弧 { } で囲み，**キー: 値** の形で記述します。

```
>>> a = {'birth':1534, 'type':'A', 'name':'Nobunaga', 'death':1582}
```

これは織田信長のプロフィールです。

keys() や values() メソッドを使い，登録されているキーや値のリスト（のようなもの）を取得します。

```
>>> list(a.keys())         # キーのリスト
['birth', 'type', 'name', 'death']
>>> del a['type']          # 指定したキーと値を削除
>>> a['hobby'] = 'tea'     # 新しい要素の追加
>>> print(a)
{'birth': 1534, 'name': 'Nobunaga', 'death': 1582, 'hobby': 'tea'}
```

辞書のキーには，先ほどの Immutable なデータ（数値，文字列，タプルなど）しか使えません。

また，キーと値のペア（タプル）を並べたリストから辞書を作ることもできます。

```
>>> pairs = [(1, 'One'), (2, 'Two'), (3, 'Three')]
>>> d = dict(pairs)
>>> print(d)
{1: 'One', 2: 'Two', 3: 'Three'}
```

5.4 集合 (set)

要素を集めたものにリストやタプルがありましたが，順番を気にしない要素の集まりが set です。set の要素になれるのは，immutable なデータ（数値，文字列，タプルなど）に限られます。

集合は次のように波括弧　を使って定義します：

```
>>> a = {1, 4, 3, 2, 2, 2}
>>> a
{1, 2, 3, 4}     # 重複が除かれ，順序のない状態で保持される
```

```
>>> a.add(5)       # aに5を付け加える
>>> print(a)
{1, 2, 3, 4, 5}
```

実行結果を見ると，重複が除かれているのがわかります。要素の並び順は入力時と変わることがありますが，これは set が数学の集合と同様に「順序を持たない」性質を持つためです。

関数 set() を使い，リストを集合に変換することもできます。

```
>>> a = [1, 2, 3, 1, 2]
>>> b = set(a); b
{1, 2, 3}
```

数学における集合の演算 $\cup, \cap, \setminus$ などに対応する演算子も用意されています。

- ファイル名：**set01.py**

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
print(A | B)    # A∪B（和集合）
print(A & B)    # A∩B（共通部分）
print(A - B)    # A\B（差集合）
```

実行結果の例

```
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
```

集合そのものに要素を加えたり，共通部分を取って更新したりする複合代入演算子

```
|=,    &=,    -=
```

が用意されています。例えば，

```
>>> a = {1,2,3}; b = {3,4}
>>> a |= b    # 集合aに集合bの要素を加える。
>>> a
{1,2,3,4}
```

となります。

6　キーボードからのデータの取得

キーボードから入力した文字や数値に応じて，プログラムの結果を変化させることを考えます。キー入力を取得するには input() という関数を使います。次のファイルを作って実行してみましょう。

- ファイル名：**input1.py**

```
namae = input('あなたの名前を入力してください。：')
print('こんにちは', namae, 'さん')
```

作成したファイルを実行すると，次のような表示が出ます：

```
> python input1.py
あなたの名前を入力してください。：
```

そこで，信州花子と入力して Enter キーを押せば

```
こんにちは 信州花子 さん
```

と出力されます。上のプログラムは次の手順を実行しました：

1.「あなたの名前を入力してください。：」と画面に表示
2. 変数 namae を作成。キーボード入力を待つ。
3. 入力された文字列を変数 namae に代入
4.「こんにちは・・・さん」と画面に表示

input() で入力された文字列は変数 namae に格納されるわけです。

実は，入力するデータが「数値」である場合には注意が必要です。Python の input() 関数は，入力されたデータを数字であっても常に「文字列（str 型）」として扱うからです。そのため，計算に利用するには int() 関数や float() 関数を使って，文字列を適切な数値データに変換する必要があります。

次は，入力した数値の 2 乗を出力するプログラムです。

- ファイル名：**input2.py**

```
1 aa = input('数値（整数）を入力してください：')     # aaは入力された「文字列」
2 num = int(aa)                                  # ここで文字列を「整数(int型)」に変換
3 print('入力された数値の2乗は', num**2, 'です。')
```

これを実行すると次のようになります：

```
1 > python3 input2.py
2 数値(整数)を入力してください：9
3 入力された数値の2乗は 81 です。
```

このプログラムで 1.5 などの整数以外を入力すると，変換に失敗してエラーとなります。小数を扱いたい場合は，float() 関数を使って変換してください。

これらを組み合わせて，BMI（体格指数）を計算するプログラムを作ってみましょう。

- ファイル名：**bmi1.py**

```
1 namae = input('あなたの名前を入力してください：')
2 shintyo = input('あなたの身長(cm)は何センチですか？：')
3 shintyo = float(shintyo)  # 数値に変換
4 weight  = input('あなたの体重(kg)は何キログラムですか？：')
5 weight = float(weight)    # 数値に変換
6
7 # cmをmに換算してBMIを計算 (100^2 = 10000)
8 bmi = 10000 * weight / (shintyo**2)
9 print(namae, 'さんのBMIは', bmi, 'です。')
```

プログラムを実行して自分の BMI を計算してみよう。

7　論理型と比較演算子

数学における「正しい主張（命題）」を**真**（しん），そうでないものを**偽**（ぎ）といいます。Python では真を True，偽を False で表します。これらは bool 型（論理型）と呼ばれる特別なデータであり，予約語として扱われます。

文法に注意しながら，インタラクティブシェルでの例を見てください。

```
1 >>> a = 3        # aに3を代入（手続き）
2 >>> a == 3       # aの値は3と等しいか？（比較・検証）
3 True
```

```
>>> a == 2        # aの値は2と等しいか？
False
>>> a > 2
True
>>> 3 != 2        # 3と2は「等しくない」か？
True
```

特に注意が必要なのは，等号 == です。Python において = は「代入」を意味する記号として既に使われているため，数学的な「等しい」という比較には == と 2 つ重ねて書くルールになっています。

二つの数値を比較して True か False を返すこれらの記号は，**比較演算子**と呼ばれます。

数値に対して使える比較演算子

==	!=	<	>	<=	>=
=	≠	<	>	≤	≥

不等号と等号の順番は，「小なりイコール」という言葉の順（< の後に =）と覚えると良いでしょう。また，!= は「not equal」の略語です。

True と False に対しては，論理演算 and（かつ），or（または），not（否定）を行うことができます。これらは数学における論理記号 $\wedge, \vee, \neg$ に対応します。

```
>>> a = True      # aをTrueとする
>>> b = False     # bをFalseとする
>>> a and b       # a かつ b （両方が真のときのみTrue）
False
>>> a or b        # a または b （少なくとも一方が真ならTrue）
True
>>> not a         # a の否定（真偽を反転させる）
False
```

Python の特徴的な機能として，不等号を数学の慣習通りに連続して使うことができます。

```
>>> a = 4
>>> 3 < a < 5
True
```

上の 3 行目は，内部的には 3 < a and a < 5 と書いた場合と同じ意味になります。他の多くのプログラミング言語ではこのような書き方はできませんが，Python では数学と同じ直感的な記述が可能です。

リストや集合などのデータの集まり（コレクション）に対して使える比較演算子もあります。

コレクションに対して使える比較演算子

== != in

記号 ==, != は数値のときと同様，中身が一致するかどうかを判定します。一方，in は数学の記号 $\in$ に相当し，次のように「要素が含まれているか」を調べるために用います。

```
>>> a = [3, 6, 5, 1]      # a をリスト [3, 6, 5, 1] とする
>>> 5 in a                # 5 は a の要素か？（5 \in a）
True
>>> 7 in a                # 7 は a の要素か？（7 \in a）
False
```

また，この比較演算子は，データの型を調べる type() 関数と組み合わせて使うこともできます。

```
>>> a = [3, 5, 6, 1]
>>> type(a) == list        # a の型はリストか？
True
>>> type(a) == tuple       # a の型はタプルか？
False
```

まあすぐに使うことは無いので忘れても構いません。

数値の比較において，浮動小数点数が絡む場合は「精度の限界」による誤差に注意が必要です。

```
>>> type(3)              # 3の型は
<class 'int'>            # 整数（int型）
>>> type(3.0)            # 3.0の型は
<class 'float'>          # 浮動小数点数（float型）
>>> 3 == 3.0             # 異なるデータ型同士だが
True                     # 値が数学的に等しければ True と判定される
>>> 3 == 3.00001         # 明らかに異なる場合は
False                    # もちろん False
>>> 3 == 3.00000000000000001
True                     # 差が非常に小さいと，同一視される
```

以前説明した通り，コンピュータは数値を有限桁の二進数で管理しています。そのため，あまりに小さい差はメモリ上に記録しきれず，内部的には「同じ数値」として扱われてしまいます。

浮動小数点数を含む計算に常に誤差が伴うのと同様に，比較演算においてもこの誤差の影響が現れます。したがって，厳密な等号判定が必要な場面（特に数値計算）では，単純に `==` で比較するのではなく，「差の絶対値が十分に小さいか」を確認するなどの工夫が必要になることがあります。

8　条件分岐

8.1　if 文

条件（`True` か `False`）に応じて，処理を変化させるときに `if` 文という決められた文法を用います。

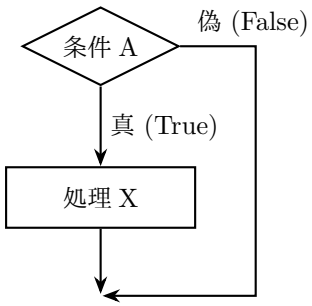


図 1　if 文の基本構造

上のような手順を視覚的に表したグラフをフローチャートと呼びます。この手順を Python では次のように記述します。

Python での if 文の構造

```
if 条件A:
    処理X（の1行目）
    処理X（の2行目）
    ...
```

- **条件 A** が `True` であれば処理 X を実行し，`False` であれば何もしません。
- `if` の行の行末には，必ず**コロン「:」**を打ちます。
- 処理 X の行は，必ず先頭に**インデント（字下げ）**を入れます。

ここで，処理 X の前のインデント（字下げ）は単なる見た目のためではなく，文法上極めて重要な役割を果たします。処理 X が数行にわたる場合，どこまでが `if` 文の支配下にある処理か（これを**ブロック**と呼びます）は，インデントが揃っているかどうかで厳密に判断されます。

具体的に，「処理 X を行い，次に条件 A を確認してそれが真なら処理 Y を行う。そうでなければ何もしない。いずれの場合でも最後に処理 Z へ進む」という手順を考えてみましょう。これをフローチャートで表すと次のようになります。

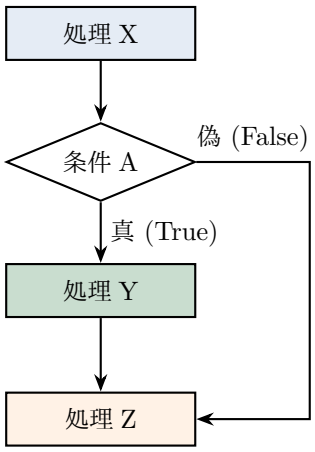


図 2 条件分岐を含む処理の流れ

これを Python では次のように記述します。インデントによって「処理 Y」だけが `if` 文の支配下（ブロック）に入っていることに注目してください。

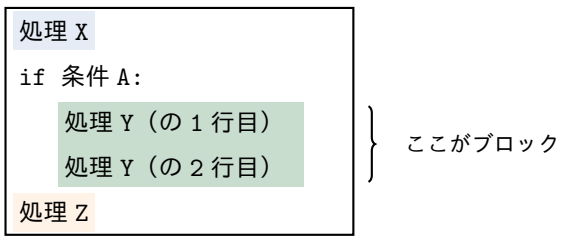


図 3 プログラムの構造とブロック

もちろん，処理 Y が 3 行以上になる場合は，同じようにインデントを揃えて記述すればよいです。

さて，では具体的に `if` 文を使った例題をやってみましょう。次の簡単なアルゴリズムを Python プログラムで書きます。

(1) 整数 a を入力させる。
(2) a が偶数なら，『a is even』と表示してから a を半分にする。(a が偶数でなければ何もしない。)
(3) 最終的な a の値を表示する。

- ファイル名：**if1.py**

```
1 a = input('Input integer a: ')
2 a = int(a)  # aを整数にする
3
4 if a%2 == 0:       # もしaが偶数なら
5    print('a is even')   # a is evenと表示
6    a = a//2             # そして，aを半分にする。インデントに注意。
7
8 print(a)            # aの値を表示
```

このプログラムの構造を先ほどの図との対応は次のようになっています。1–2 行目の準備（処理 X）に続き、4 行目の `if` 文によって 5–6 行目のブロック（処理 Y）を実行するかどうかが決まり、最後に 8 行目（処理 Z）へと合流します。

実際にこのプログラムを実行し、10 や 7 を入力して、条件によって処理が「スキップ」される様子を確認してみましょう。

8.2 if–else 文

次に，少し複雑な条件分岐を考えます。条件 A が真のときには処理 X，偽のときには別の処理 Y を実行したいとしましょう。

もちろん，これはこれまでに学習した `if` 文だけで書くこともできます。

```
1 if 条件A:       # 条件Aが真なら
2     処理X       # 処理Xを行い，偽なら行わない。
3
4 if not 条件A:   # 条件Aが偽なら
5     処理Y       # 処理Yを行い，真なら行わない。
```

ただ，条件 A を 2 回チェックするのは計算コストの無駄ですし，何より「条件 A が偽だったときのための処理だ」ということが 4 行目まで読み進めないと把握できない，というのは良いコードとは言えません。

このようなときのために，よりスマートに記述できる **if–else 文** という構文が用意されています。

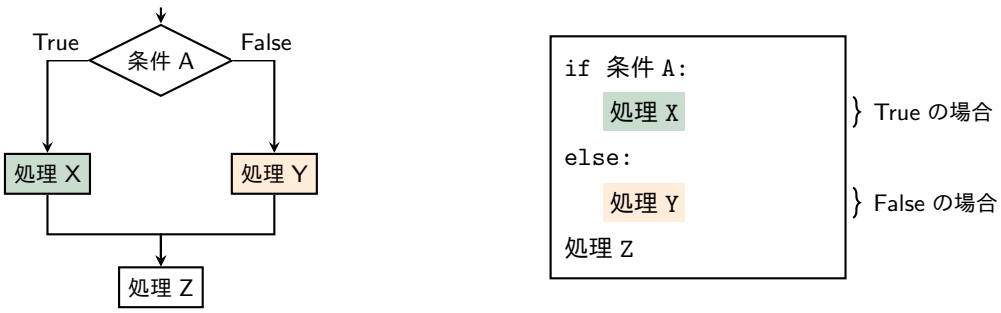


図 4 `if--else` 文の論理構造とコード構造の対応

Python での if–else 文の構造

```
if 条件A:
    処理X
else:
    処理Y
```

- 条件 **A** が True であれば処理 X を実行し，False であれば処理 Y を実行します。
- if 行と else 行の行末には，必ず**コロン「:」**を打ちます。
- else は，対応する if と同じインデント（字下げ）の位置に書きます。
- 処理 X と処理 Y の範囲は，インデントによって判断されます。

では，if-else 文を使った例題をやってみましょう。次のアルゴリズムを考えます。

(1) 整数 a を入力させる。
(2) もし a が偶数ならば，a を半分にする。
(3) そうでなければ，a を 3*a+1 にする
(4) a の値を表示する。

これをプログラムで書いてみましょう。

- ファイル名：**if2.py**

```
a = input('a?: ')
a = int(a)          # 入力したものを整数に変換する。

if a%2 == 0:        # もしaが偶数なら
    a = a//2        # aを半分にする
else:               # そうでなければ
    a = 3*a + 1     # aを3a+1にする

print(a)            # aの値をプリント
```

プログラムを実行し，いくつかの数値（たとえば 10 や 7）を入力して動作を試してみましょう。偶数のルートと奇数のルート，どちらか一方が選ばれて計算が進む様子が確認できるはずです。

8.3 if–elif–else 文

さらに多くの条件分岐を記述するためには **if–elif–else 文** を使います。なお、elif は else if の略です。

Python での if–elif–else 文の構造

```
if 条件A:
    処理X
elif 条件B:
    処理Y
else:
    処理Z

処理W
```

- **条件 A** が真の場合に処理 X を行います。
- 条件 A が偽で、かつ **条件 B** が真のときに処理 Y を行います。
- 条件 A も条件 B も成り立たないときに処理 Z を行います。
- いずれの場合でも処理 W に進みます。
- さらに条件 C，条件 D，……と分岐を増やしたい場合は，必要な数だけ `elif` を重ねることができます。

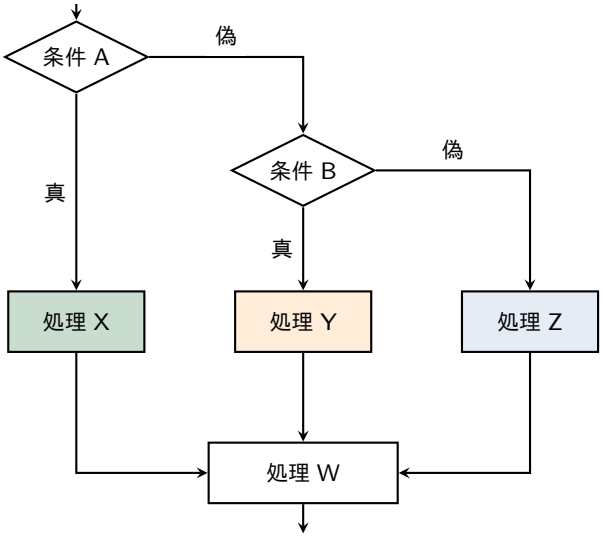


図 5　`if--elif--else` 文の論理構造

8.4　複雑な条件分岐の例（閏年の判定）

`if--elif--else` 文を活用して，入力された西暦が「閏年（うるうどし）」かどうかを判定するプログラムを作成しましょう。グレゴリウス暦における閏年の定義は以下の通りです：

- 西暦年が 4 で割り切れる年は閏年とする。
- ただし，4 で割り切れる年であっても，100 で割り切れる年は閏年としない。
- ただし，100 で割り切れる年であっても，400 で割り切れる年は閏年とする。

この定義は「ただし」による例外規定が重なっており，そのままではプログラムの条件式に落とし込みにくい形をしています。そこで，論理的に同値な次の条件に整理し直します。

(1) 西暦年が 400 で割り切れるならば，閏年である。
(2) 上記以外のとき（400 で割れないとき），100 で割り切れるならば，閏年ではない。
(3) 上記以外のとき（100 で割れないとき），4 で割り切れるならば，閏年である。
(4) 上記のどれにも当てはまらないとき，閏年ではない。

このように，より「強い」条件（範囲の狭い例外条件）から順に判定することで，`if--elif--else` 文で簡潔に記述できます。

- ファイル名：**uruuQ.py**

```
year = input('西暦を入力: ')
year = int(year)

if year % 400 == 0:       # 400で割り切れる
    print(year, 'は閏年です。')
elif year % 100 == 0:     # 400で割れず，100で割り切れる
    print(year, 'は閏年ではありません。')
elif year % 4 == 0:       # 100で割れず，4で割り切れる
    print(year, 'は閏年です。')
else:                     # 上の全ての条件に当てはまらない
    print(year, 'は閏年ではありません。')
```

論理的には「4 で割り切れるかどうか」から判定を始めてもプログラムは書けますが，その場合は条件の中にさらに `if` 文を入れる（入れ子構造にする）必要があり，構造が複雑になってしまいます。

- 定義に忠実な（少し複雑な）書き方

```
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print(year, 'は閏年です。')
        else:
            print(year, 'は閏年ではありません。')
    else:
        print(year, 'は閏年です。')
else:
    print(year, 'は閏年ではありません。')
```

この書き方でも正しく判定できますが，パッと見てどの `if` とどの `else` が対応しているのかが分かりにくくなります。プログラムを書く前に頭を使って少しの工夫をすることで，読みやすく素直なコードを書くことができるようになるのです。

8.5 練習問題

8.5.1 問題：BMI 計算プログラム

名前，身長，体重を入力させ，そのデータから BMI（体格指数）を計算・表示し，その値に応じて肥満度の判定結果を出すプログラムを作成しましょう。

プログラムは次の手順（アルゴリズム）に従うものとします。

(1) 名前を入力させ，変数 `namae` に代入する。
(2) 身長と体重を入力させ，それぞれ `shintyo`, `taijuu` に代入する。
(3) BMI を表示する。

(4) もし BMI $<$ 18.5 なら，「やせ気味です」と表示する。
(5) そうでないとき，BMI $<$ 25.0 なら，「ふつうです」と表示する。
(6) そうでないとき，BMI $<$ 30.0 なら，「太り気味です」と表示する。
(7) 上のどれにも当てはまらないとき，「太りすぎです」と表示する。

次のプログラムの ***** の部分を推測して，上の手順が正しく実行されるように完成させなさい。

- ファイル名：**bmi2.py**

```
namae = input('あなたの名前を入力してください：')
shintyo = input('あなたの身長は何センチですか？：')
shintyo = *****(shintyo)  # 数値として扱えるように変換
taijuu  = input('あなたの体重は何キログラムですか？：')
taijuu = *****(taijuu)

# BMIの計算式（身長がセンチメートルの場合）
bmi = 10000.0 * taijuu / (shintyo**2)
bmi = round(bmi, 1)     # BMIの値を小数点第2位で四捨五入する

print(namae, 'さんのBMIは', bmi, 'で', end='')

if **********:
    print('やせ気味です。')
elif **********:
    print('ふつうです。')
**** **********:
    *****('太り気味です。')
****:
    *****('太りすぎです。')
```

まだ完全初心者を対象としているので，穴埋め形式でかなりのヒントを出していますが，本来は上のようなプログラムは自分でノーヒントで作れなければなりません。

8.5.2　問題：実数解の個数判定

2 次方程式 $ax^2+bx+c=0$ の係数 a, b, c を入力させ，判別式 $D=b^2-4ac$ の値に応じて，実数解の個数を表示するプログラムを作成しなさい。ただし，係数 a, b, c は実数であるものとする。

(1) 係数 a, b, c を入力させる。
(2) 判別式 D を計算する。
(3) $D>0$ ならば「異なる 2 つの実数解」と表示する。
(4) $D=0$ ならば「重解」と表示する。
(5) $D<0$ ならば「実数解なし」と表示する。

ファイル名は hanbetsu.py にしてください。

※ $a=0$ の場合は 2 次方程式ではなくなるので，本当は「例外処理」が必要になりますが，今回は $a \neq 0$ と仮定して作成して構いません。余裕があれば $a=0$ の場合も考慮したプログラムを考えてみましょう。

9　For 文による繰り返し

9.1　For 文の文法

いくつかの処理の繰り返しを記述する方法を解説します。たとえば，hello をプリントするという処理を 5 回繰り返すには手動で

```
print('hello')
print('hello')
print('hello')
print('hello')
print('hello')
```

と書けばいいのですが，for 文を使い，次のように書くこともできます：

```
for j in range(5):
    print('hello')
```

ここで，range(5) は 0 から 4 までの連続した 5 つの整数を順に用意する命令です。以前やったように，list(range(5)) はリスト [0,1,2,3,4] を返すことを思い出しましょう。実際に，Python のインタラクティブシェルでこのプログラムを実行してみます：

```
>>> for j in range(5):
...     print('hello')
...
hello
hello
hello
hello
hello
```

上の for 文は j が 0 から 4 まで，その下の処理を繰り返すので，次のように j の値を使うこともできます：

```
for j in range(5):
    print(j)
```

実行結果の例

```
0
1
2
3
4
```

ここで j はダミー変数で，他の文字に変えても結果は同じです。例えば，次は上のものと同じです：

```
for x in range(5):
    print(x)
```

さて，for 文の構造は次のようになっています：

Python での for 文の構造

```
1 for j in range(10):
2     処理A（の1行目）
3     処理A（の2行目）
4     ...
5
6 処理B
```

- 変数 j の値を 0 から 9 まで変化させながら，処理 A を 10 回繰り返し実行します。
- for の行末には，必ず**半角コロン「:」**を打ちます。
- 処理 A の行は，必ず先頭に**インデント（字下げ）**をします。
- 繰り返しの処理が終わったら処理 B（インデントされていない）へ進みます。

ここで，if 文のときと同様にインデントされた部分をひとかたまり（ブロック）として繰り返しを行います。インデントから外れた行（上では処理 B）書かれているプログラムはが for 文の支配下のブロックから外れたことになり，その行は繰り返されず，10 回のループがすべて終了した後に実行されます。

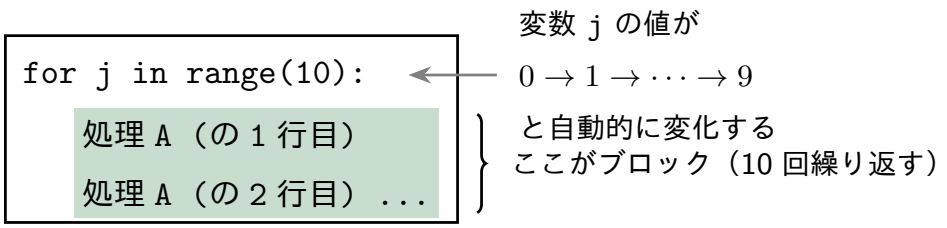


図 6　for 文の構造

この「変数 j の値が 1 つずつ変化しながら，同じ処理が並んで実行されていく」という一連の流れを視覚的に表すと，次のようになります。下の各処理 A の中でそのときの j の値を使うこともできます。

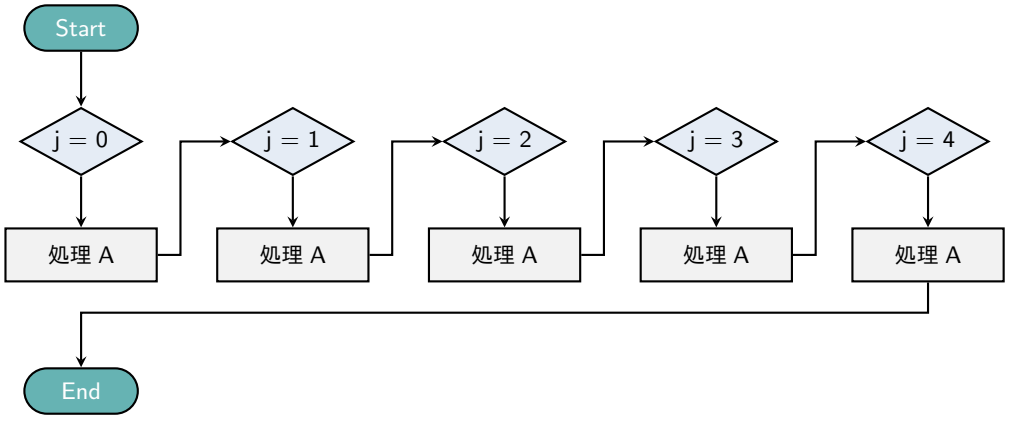


図 7　for 文で行われる処理の流れ（5 回の繰り返し）

9.2　for 文の簡単な例

for 文を使ったプログラムをいくつか書いて，実行結果を見てみましょう。

- ファイル名：**for1.py**

```
for j in range(5):              # 以下のブロックを5回繰り返す
    print('wanwan', end=' ')    # これと(end=' 'により改行されず半角スペースが空く)
    print('nyannyan')           # これをセットで繰り返す
```

実行結果の例

```
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
```

次のプログラムの最後の行は，for 文のインデントから外れているため，1 回しか実行されません。上のプログラムとの「見た目の違い」と「実行結果の違い」によく注目してみましょう。

- ファイル名：**for2.py**

```
for j in range(5):              # 以下のブロックを5回繰り返す
    print('wanwan', end=' ')    # これと
    print('nyannyan')           # これをセットで繰り返すが...

print('gaogao')                   # ここはインデントがないので繰り返さない
```

実行結果の例

```
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
gaogao
```

今度は，繰り返される処理の中で，自動的に変化していく変数 j の値そのものを利用してみます。0 から 9 までの数字と，その 2 乗（j**2）を並べて表示するプログラムです。

- ファイル名：**for3.py**

```
for j in range(10):                 # jを0から9まで変化させながら次を繰り返す
    print(j, j**2)                  # jの値と，jの二乗をプリントする

print('owari')                      # ここはインデントがないので繰り返さない
```

実行結果の例

```
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
owari
```

数字を順番に変化させながら，文字列と組み合わせて表示することも簡単にできます。人間がやると疲れてしまう退屈な数え上げも，コンピュータなら一瞬です。

- ファイル名：**for4.py**

```
for j in range(1, 101):         # jを1から100まで変化させて繰り返す（101の手前！）
    print('ひつじが', j, '匹')
```

実行結果の例

```
ひつじが 1 匹
ひつじが 2 匹
ひつじが 3 匹
.....
.....
.....
ひつじが 99 匹
ひつじが 100 匹
```

これまでは range() を使って数字の連番を作ってきましたが，あらかじめリストを用意しておき，その要素を先頭から順に取り出して処理することもできます。

- ファイル名：**for5.py**

```
aa = ['Alice', 'falls', 'down', 'a', 'rabbit', 'hole.']  # リストaaを定義

for i in aa:                # aaの要素を順番に1つずつ i に取り出して繰り返す
    print(i, end=' ')       # iをプリントする（後ろに半角スペースを空ける）
```

実行結果の例

```
Alice falls down a rabbit hole.
```

9.3 for 文でありがちなエラー

Python の文法の特徴にインデントで構文を判断するというものがありました。if 文のときと同様にインデントを間違えるとエラーとなります。

- ファイル名：**for6.py**

```
for j in range(5):
    print('wanwan')
  print('nyannyan')    # ここのインデントが浅い
```

実行結果の例

```
  File "for6.py", line 3
    print('nyannyan')
    ^
IndentationError: unindent does not match any outer indentation level
```

次のようにインデントを下げすぎてしまった場合も同じくエラーとなります：

- ファイル名：**for7.py**

```
for j in range(5):
    print('wanwan')
     print('nyannyan')    # ここがへん！
```

実行結果の例

```
  File "for7.py", line 3
    print('nyannyan')
    ^
IndentationError: unindent does not match any outer indentation level
```

9.4 for 文の応用 1 ：for 文の中に for 文を入れる

for 文のブロック（支配下）の中に，さらに別の for 文を丸ごと入れて，繰り返しを「二重」に記述することもできます。

まずは，外側と内側でどのように変数が動いていくのか，実行結果を観察してみましょう。

- ファイル名：**for8.py**

```
for i in range(5):            # iを0から4まで変化させて以下を繰り返す
    for j in ['a','b','c']:   # その各iに対してjをa, b, cと変化させて繰り返す
        print(i, j)           # iとjの組み合わせをプリント
```

実行結果の例

```
0 a
0 b
0 c
1 a
...
...
4 c
```

ここでインデント（字下げ）の深さに注目してください。3 行目の print は，「外側の for 文」と「内側の for 文」の両方の支配下にあるため，インデントが 2 段階分（スペース 8 個分）深くなっています。

この二重の繰り返しの仕組みを使えば「九九の掛け算の表」を出力させることができます：

- ファイル名：**for9.py**

```
1 for i in range(1, 10):            # iを1から9まで変化させる（縦の行）
2     for j in range(1, 10):        # 各iに対して，jを1から9まで変化させる（横の列）
3         print(i*j, end=' ')       # かけ算の結果を横に並べてプリント
4     print('')                     # 内側のループが1行分終わったら，ここで改行する
```

実行結果の例

```
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
```

5 行目の print('') は，内側の j のループからは外れていますが，外側の i のループの支配下には残っています。そのため，「横に 9 個数字を並べ終わるたびに，ちょうどいいタイミングで 1 回だけ改行が入る」という動きを実現しています。

9.5　for 文の応用 2：if 文と組合わせる

for 文のブロックの中に if 文を入れ，繰り返しの中で条件に応じて処理を枝分かれさせると，より複雑なプログラムを書くことができます。

次のプログラムは，変数 j の値が「偶数」なら is even，「奇数」なら is odd と画面に表示させる例です。

- ファイル名：**for10.py**

```
1 for j in range(1, 10):
2     if j%2 == 0:
3         print(j, 'is even')
4     else:
5         print(j, 'is odd')
```

実行結果の例

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
```

9.6　for 文の応用 3：break で処理を止める

for 文は指定された回数だけきっちり繰り返すのが得意ですが，処理の途中で「特定の条件を満たしたら，もうそれ以上繰り返さずに途中で切り上げたい」というケースもあります。

そんなときは break（ブレイク）という命令を使います。for 文の繰り返し中に break が実行されると，for 文の処理はその瞬間に直ちに強制終了し，次のプログラムへと進みます。

例えば，次のような処理を考えてみましょう。

- 2^{2^j} という計算を，j を $0, 1, 2, \ldots, 10$ と変化させながら次々とプリントする。
- ただし，この計算結果は凄まじい早さで巨大化していくため，値が **1 兆（1000000000000）** を超えた場合には，パソコンに無理をさせないためにその時点で for 文の処理を停止（脱出）したい。

- ファイル名：**for11.py**

```
for j in range(11):
    a = 2**2**j
    if a < 1000000000000:           # 1兆未満ならそのまま表示する
        print(a)
    else:                           # 1兆を超えてしまったら...
        print('Too big!!')
        break                       # これが実行されたら、ループは即座におしまい！
```

実行結果の例

```
2
4
16
256
65536
4294967296
Too big!!
```

9.7　for 文の応用 4：データの差分（速度と加速度）

科学計測やスポーツのデータ解析では，「時間ごとに変化する位置のデータ」から「速度」や「加速度」を計算する処理（差分）がよく使われます。

ここでは，直線状を動く物体の 0.1 秒ごとの位置データが，次のようなリストとして手元にあるとしましょう。

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
```

このデータを使って速度を計算するプログラムには，データの受け取り方に応じて **2 つのアプローチ**があります。それぞれの特徴と書き方の違いを学んでみましょう。

9.7.1　アプローチ A：すべてのデータが最初から見えている場合

データがすべて最初から 1 つのリストに揃っているなら，リストの「インデックス（添え字）」を使って，j 番目の位置と j+1 番目（1 つ未来）の位置を直接指定して引き算（差分）するのが一番簡単です。

- ファイル名：**for12.py**

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
dt = 0.1                            # 時間間隔 dt = 0.1[秒]
for j in range(len(xlis) - 1): # 最後のデータの「次」はないので(個数)-1 回繰り返す
    v = (xlis[j+1] - xlis[j]) / dt # 「1つ未来の位置」から「今の位置」を引いて時間で割る
    print(v)
```

実行結果の例

```
2.5
1.9999999999999996
1.2
0.40000000000000036
-0.5000000000000004
-1.399999999999999
-2.1000000000000005
-2.6999999999999993
-2.9000000000000004
```

※コンピュータの計算誤差により小数の末尾が少し怪しくなっていますが，2.5, 2.0, 1.2... と正しく平均速度が計算できています。

9.7.2　アプローチ B：データがリアルタイムに 1 つずつ届く場合

しかし，実際のデータ解析の現場では，最初からすべての未来のデータが手元にあるとは限りません。例えば，**GPS から現在の位置情報をリアルタイムに受信しながら速度を計算する場合**などです。このとき，手元にあるのは「今届いた位置」と「過去の位置」だけであり，「未来の位置」をあらかじめ覗き見ることはできません。また，データが数 GB もあるような巨大なファイルを扱う場合も，一度にすべてのデータをメモリ（リスト）に読み込むことはできません。

このような場合は，次のようにリストからデータを **1 つずつ順番に受け取りながら，その場で処理していく工夫** が必要になります。

```
# 届いたデータをその都度 x として受け取る（未来のデータは覗き見できない制限）
for x in xlis:
    # さあ，どうやって1つ前のデータと引き算しよう？
```

この制限の中で速度を計算するには，**「今届いたデータ（x）を処理し終わったら，それを『1 つ過去のデータ』として別の変数にキープしておく」** という工夫が必要です。

今回は，1 つ過去の位置を保存しておくための変数として y を用意します。最初のスタート（時刻 −0.1 秒）の位置を仮に `y = 0` とセットしておき，データが 1 つ届くたびに次のような手順でバトンタッチを行っていきます：

1. 0 秒の位置 `x = 0.39` が届く。
2. 過去の位置 y（**最初は 0**） との間で速度（`x - y) / dt` を計算して表示。
3. **次のターンのために，今の位置を過去の変数へ保存（`y = x` とする）。**
4. 0.1 秒の位置 `x = 0.64` が届く。
5. 1 つ過去の位置になった y（0.39） との間で速度を計算して表示。
6. **次のターンのために，今の位置を過去の変数へ保存（`y = x` とする）。**
7. （以下，繰り返し……）

この「過去の値を覚えておく箱（y）」を使ったプログラムは，次のようになります。

- ファイル名：**for13.py**

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
dt = 0.1
y = 0                       # 1つ過去の位置を保存する変数（最初は仮に0）

for x in xlis:              # 位置情報が1つずつ届き、x に代入される
    print((x - y) / dt)     # 「今の位置」から「1つ過去の位置」を引いて速度を計算
    y = x                   # 使い終わった今の位置を、次のターンのために過去の箱に保存！
```

実行結果の例

```
3.9                  ← ※最初のこれだけは、過去を 0 と仮定したため無意味な値になります
2.5
1.9999999999999996
1.2
0.40000000000000036
-0.5000000000000004
-1.399999999999999
-2.1000000000000005
-2.6999999999999993
-2.9000000000000004
```

実行結果を比べると，ダミーの過去から始めた最初の 1 行目を除いて，for12.py と全く同じ速度データが算出できていることがわかります。この「過去の値をキープする変数」の手法は，センサーデータの処理などで非常に重要になるプログラミングの定石です。

9.7.3　（参考）平均の加速度を求める

最後に「速度の差」をさらに時間間隔で割ったものが「平均の加速度」です。すべてのデータがリストに揃っている場合（アプローチ A の形式）で，位置データから一気に平均の加速度を計算して表示するプログラムは，次のようになります。

- ファイル名：**for14.py**

```
1 xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
2 dt = 0.1
3 for j in range(len(xlis) - 2): # 2つ未来のデータ(j+2)まで使うので(個数)-2回繰り返す
4     # 「後半の速度（未来の差分）」から「前半の速度（今の差分）」を引き、さらに dt で割る
5     a = ((xlis[j+2] - xlis[j+1]) - (xlis[j+1] - xlis[j])) / (dt * dt)
6     print(a)
```

実行結果の例

```
-5.0000000000000036    ← 0 秒から 0.2 秒の間の平均の加速度
-7.999999999999995     ← 0.1 秒から 0.3 秒の間の平均の加速度
-7.999999999999995
-9.000000000000007
-8.999999999999984
-7.000000000000016
-5.999999999999988
-2.000000000000007
```

9.8　練習問題

9.8.1　問題：$3n+1$ 問題（コラッツ予想）

数学の世界には「$3n+1$ 問題」と呼ばれる，とても不思議な未解決問題があります。任意の自然数 n に対して，次のような操作を繰り返します。

- もし n が偶数なら，n を半分にする
- もし n が奇数なら，n を $3n+1$ にする

この単純な操作を繰り返すだけで，スタートがどんな自然数であっても，最終的には必ず「1」にたどり着くだろうという予想です。現在も完全には証明されておらず，「角谷の問題」や「Collatz（コラッツ）予想」とも呼ばれています。

例えば，最初の数が 9 の場合，次のように数が変化していきます。

$$9 \to 28 \to 14 \to 7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

キーボードから入力された数 a に対して，この手順で変化していく数を次々に計算して表示するプログラムを作ってみましょう。今回は，次のアルゴリズム（処理手順）に従って作成します。

1. 自然数 a の値を入力させ，まずは最初の値をそのままプリントする。
2. 最大の繰り返し回数 maxiter を 1000 とする。
3. 以下の 4 から 7 の処理を，for 文を使って maxiter 回繰り返す。
4. もし a の値が 1 なら，break で直ちに for 文を終了する。
5. もし a が偶数なら，a を半分にする。
6. もし a が奇数なら（偶数でなければ），a を $3a+1$ にする。
7. if 文の条件分岐が終わったら，変化した後の a をプリントする。

以下の Python プログラムの **** の部分を自分で考えて，上のアルゴリズムが正しく実現するようにプログラムを完成させなさい。

- ファイル名：**collatz1.py**

```
a = int(input('a?: '))
print(a)

maxiter = 1000
for i in range(maxiter):
    if *********:
        break
    elif *********:
        a = a // 2
    else:
        *********
    print(a)
```

プログラムが完成したら実行し，a として 13 を入力してみましょう。プログラムが正しければ，次のように「1」に吸い込まれていく様子が出力されるはずです。

実行結果の例

```
a?: 13
13
40
20
10
5
16
8
4
2
1
```

（注意）今回のプログラムでは，念のため最大の繰り返し回数を 1000 回に設定しました。しかし，コラッツ予想のように「目標（1 になるまで）を達成するのに何回繰り返せばいいのか，やってみるまで回数が分からない処理」を書くには，実は for 文よりも適した while 文というもう 1 つの繰り返し構文があります（もしかしたら，1000 回では足りない数があるかもしれません！）。この while 文については，後の章で詳しく学習します。

9.8.2　問題：順次与えられた位置情報から加速度を算出する問題

上で説明した for13.py（※アプローチ B）と同様に，未来の位置データがあらかじめ見えておらず，位置情報が for x in xlis: のように「今この瞬間のデータ」として 1 つずつしか得られない制限の中で，物体の「平均の加速度」を順次計算して表示するプログラムを作ってみましょう。

加速度（2 階差分）を求めるには，「今届いた位置」だけでなく，「1 つ過去の位置」と「2 つ過去の位置」の，合計 3 つのデータがその都度必要になります。そこで，今回は過去を記憶しておくための変数を x0，x1 と 2 つ用意し，次のような手順（アルゴリズム）でバトンタッチを行っていくものとします。

1. 時間間隔を dt = 0.1 とする。
2. 2 つ過去の位置を保存する変数を x0 = 0（仮の初期値）とする。
3. 1 つ過去の位置を保存する変数を x1 = 0（仮の初期値）とする。
4. for 文により，最新の位置情報 x を 1 つ受け取る。
5.「2 つ過去」から「1 つ過去」の間の平均速度を v0 = (x1 - x0) / dt とする。
6.「1 つ過去」から「最新」の間の平均速度を v1 = (x - x1) / dt とする。
7. 2 つの速度の差から，この区間の平均の加速度を a = (v1 - v0) / dt とする。
8. 計算された加速度 a の値をプリントする。if 文の条件分岐などは使わず，そのまま素直に 5 行の数式と 1 行の print を記述してください。
9. **次のターンのために**，変数 x0 と x1 の中身を更新する（バトンタッチの「順番」に猛烈に注意すること！）。
10. 手順 4 から 9 を次のデータが届くまで繰り返す。

以下の for15.py のブロックの中身（インデントが入る部分）を自分で考えて，上のアルゴリズムが実現するようにプログラムを完成させなさい。

- ファイル名：**for15.py**

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
dt = 0.1   # 時間間隔
x0 = 0     # 2つ過去の位置を覚える箱
x1 = 0     # 1つ過去の位置を覚える箱

for x in xlis:    # 最新の位置情報 x を順次読み込む
    ****************************
    ****************************
    ****************************
    ****************************
    ****************************
    ****************************
```

（注意）この問題が全く解けない人は，1 つ過去の値を記憶させていた for13.py の仕組みがまだ本当の意味で理解できていません。もう一度 for13.py の解説に戻り，変数の値がループの中でどのように上書きされて動いていたのか，紙に書いて確認してください。また，このプログラムの特性上，最初に出力される 2 つのデータ（39.0 と -14.0）は、過去の値を仮に 0 としたことから出てきた無意味なダミーデータ（無視してよい値）となります。

正解の場合，出力は次のようになるはずです（for14.py の結果と 3 行目以降が完全に一致することを確認してください）。

実行結果の例

```
39.0
-13.999999999999998
-5.000000000000004
-7.999999999999996
-7.999999999999996
-9.000000000000007
-8.999999999999986
-7.000000000000015
-5.999999999999988
-2.0000000000000107
```

9.9　台形公式による積分計算

台形公式（trapezoidal rule）とは積分 $\int_a^b f(x)dx$ を台形の面積の和

$$D_n = \sum_{k=1}^{n}(x_k - x_{k-1})\frac{f(x_{k-1}) + f(x_k)}{2}, \qquad (a = x_0 < x_1 < \cdots < x_n = b)$$

で近似しようというものである。区間を n 等分する場合は，区間の幅を $h = (b-a)/n$ おくと $x_k = a + kh$ であり，$y_k = f(x_k)$ とするとき，

$$D_n = \frac{h}{2}(y_0 + 2y_1 + 2y_2 + \cdots + 2y_{n-1} + y_n)$$

である。さて，この公式を用いて，積分 $\int_0^1 4(1+x^2)^{-1}dx$ に関して D_8 と D_{100} を計算し表示するプログラムを書きなさい。ファイル名は `trapint.py` としよう。

9.9.1　問題：シンプソンの公式による積分計算

台形公式が区間を直線（1 次式）で近似したのに対し，区間を放物線（2 次式）で近似してより正確に積分の値を求めようとする方法を**シンプソンの公式（Simpson's rule）**と呼びます。

区間 $[a,b]$ を **偶数個** である n 等分する場合，区間の幅を $h = (b-a)/n$ ，各点の高さを $y_k = f(a+kh)$ とおくと，公式は次のように展開されます。

$$S_n = \frac{h}{3}\Big(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n\Big)$$

両端（y_0, y_n）を除いた間の項の係数が，「4, 2, 4, 2, …, 4」と交互に変化しています。

さて，この公式を用いて，前問と同じ積分 $\displaystyle\int_0^1 \frac{4}{1+x^2}dx$ に関して S_8, S_{100} を計算し表示するプログラムを書きなさい。ファイル名は `simpint.py` としましょう。

9.9.2　考察：分割数 n による精度と誤差の逆転現象

前問の台形公式（`trapint.py`）とシンプソンの公式（`simpint.py`）について，分割数 `n` の値を `100` と `8` に切り替えた場合，計算結果がどのように変化するか実験して下の表の空欄を埋めてみましょう。

なお，今回の積分（$\int_0^1 \frac{4}{1+x^2}dx$）の手計算による理論上の真の値（正確な円周率）は次の通りです。

真の値 (π)：3.1415926535897932...

表 1　公式と分割数による計算結果の比較

計算方法	分割数 $n = 100$ の結果	分割数 $n = 8$ の結果
台形公式	3.1415759869231290	
シンプソン公式		

【実験からわかる結果】

- **台形公式の場合**：$n = 8$ に減らすと，直線で近似する隙間（数学的な誤差）が大きくなり，結果は大きく悪化します。つまり「細かく分けるほど正確になる」という数学の直感がそのまま当てはまります。
- **シンプソン公式の場合**：驚くべきことに，数字を細かく分けた $n = 100$ のときよりも，たった 8 等分に減らした $n = 8$ のときのほうが，**真の円周率と完全に一致（小数第 16 位まで一致）します。**

（解説：マシンの限界と丸め誤差）なぜこのような逆転現象が起きるのでしょうか？現代のコンピュータにおいて，標準的に小数を扱う型である `float` 型（倍精度浮動小数点数・64 ビット規格）は，内部の物理的な仕組み上，十進法に直すとおよそ 15〜17 桁までしか正確に数字を記憶できません。

シンプソンの公式は放物線を使ってグラフの隙間を埋めるため，非常に高精度であり，そのため，たった 8 等分（$n = 8$）するだけで，数学的にはすでにコンピュータの `float` の精度限界（16 桁目）まで到達してしまいました。その上，ループでの足し算を 8 回しかしないため，計算の過程で発生するゴミがほとんど溜まりません。

しかし，これを「もっと正確にしよう」と欲張って $n = 100$ や $n = 10000$（1 万回ループ）へと処理を増やしていくと，数学的には十分な精度があるはずなのに，コンピュータが内部で `float` の計算のたびに発生する，ほんのわずかな「計算のゴミ（丸め誤差）」だけが蓄積していくため，増やせば増やすほど逆に数値が狂っていくという現象が起きます。プログラミングの数値計算においては，**「とにかく細かくして処理を増やせば正確になる」とは限らない** のです。

（さらに一歩先へ：精度と速度のトレードオフ）では，コンピュータは 16 桁以上の正確な計算はできないのか，というと，そんなことはありません。Python の `decimal` モジュールなどを使うことで，100 桁以上の「超高精度（多倍長精度）」な計算も可能です。ただし，これにはトレードオフがあります。

- **標準の `float`（64 ビット）**：CPU の専用回路で直接計算するため，圧倒的に高速。
- **超高精度（`decimal`）**：桁数は無限だが，ソフトウェアで無理やり処理するため，速度が数十〜数百倍遅くなる。

速度を最優先して `float` を使いつつアルゴリズム（シンプソン公式）の工夫で賢く解くのか，それとも速度を犠牲にして桁数を増やすのかは目的に応じて決めることになります。

10　ファイルの書き込み・読み込み

`for` 文を使って大量のデータを表示させる場合，端末の画面（コンソール）が埋め尽くされ，過去のデータが流れて見えなくなってしまいます。そこで，Python の実行結果を「ファイル」としてパソコンに保存したり，保存したデータを再び Python に読み込ませて解析したりする方法を説明します。

10.1　ファイルの書き込み（端末の機能：リダイレクトを使う）

プログラム側には一切手を加えず，手軽に実行結果をファイルに保存するには，端末（ターミナル）の**リダイレクト（>, >>）**という機能を使うのが最もスマートです。

Python プログラムでファイルに書き出したい内容を通常通り `print` するように作っておき，端末から次のように実行します。

```
python プログラム名.py > out.txt
```

このように末尾に `> out.txt` をつけることで，本来画面に表示されるはずだった `print` の中身が，そのまま `out.txt` というファイルの中に保存されます（ファイル名 `out.txt` の部分は自由に変えてよいです）。このとき，指定したファイルがフォルダ内になければ自動的に新規作成されます。

例えば，以前作成した `for3.py` の実行結果を `for3result.txt` に書き出すには，端末で次のように入力します。

```
python for3.py > for3result.txt
```

既存のファイルの中身を消さずに，末尾にデータを「追記」したい場合は，『`>>`』を使います。試しに次の

コマンドを実行してみましょう。

```
python3 for4.py >> for3result.txt
```

これで，先ほど作った for3result.txt のデータの後ろに，続けて for4.py の実行結果が追加されます。ファイルを開いて確認してみましょう。

10.2　ファイルの書き込み（Python の機能を使う）

もう一つの方法は，Python プログラムの内部から直接ファイルをコントロールする方法です。現代の Python では，with open 文 という構文を使うのが世界標準となっているようです。

- ファイル名：**write1.py**

```
abc = 'Taro\n'              # 末尾に改行記号 \n を付けた文字列
xyz = 'Hanako'              # 改行記号のない文字列

with open('test.txt', 'a') as f:    # 追記モード(a: append)でファイルを開く
    f.write(abc)                    # 文字列をファイルに書き込む
    f.write(xyz)                    # 文字列をファイルに書き込む
```

端末からこのプログラムを実行してみましょう。

```
python3 write1.py
```

新たにファイル test.txt が作られ，中身を覗くと Taro の後ろで改行されて Hanako と書き込まれているはずです[*6]。

with open(...) as f: の下にある，4 マス下がったブロック（インデント内）にいる間だけ，ファイル f が開いた状態になります。そして，このブロックを抜けた瞬間に、Python が自動的に安全にファイルを閉じます。

もう一度プログラムを実行すると，Hanako の後ろには改行がないため、2 回目の Taro は HanakoTaro と真横にくっついて追記されます。実際に動かして確認してみましょう。なお，追記（'a'）ではなく，過去のデータを消去して「上書き」したい場合は，モードを 'w'（write）にします。

```
with open('test.txt', 'w') as f:    # 上書きモード(w)で開く
```

10.3　ファイルの読み込み

保存されたテキストデータを Python 内で再利用するには，ファイルを読み込む命令である read や readlines を使います。これらも書き込みと同様に with open のインデントの中で処理します。

- read() ： ファイルの全内容を，丸ごと 1 つの巨大な「文字列」として読み込む。
- readlines() ： ファイルを 1 行ずつバラバラに分解し，文字列の「リスト」として読み込む。

実験用のデータファイルを作るため，まずは端末で以下を実行してください。

```
python3 for10.py > guuki.txt    # 実行結果を guuki.txt に保存
```

[*7]

[*6] また，f.write() は print() と違って自動で改行してくれません。そのため，行を分けたい場合は abc のように文字列の末尾に改行記号 \n を明示的に足しておく必要があります。

[*7] Windows の PowerShell 環境では，リダイレクトを使用するとファイルの文字コードが自動的に UTF-16 という形式で保存されてしまうことがあります。このまま Python で読み込むとエラー（文字化け）になるため，うまく動かない場合は、一度そのファイルをメモ帳等で開き，右下の保存エンコードを UTF-8（世界標準の文字コード）に指定して上書き保存し直してください。

まずは，丸ごと読み込む read() の例です。

- ファイル名：**read1.py**

```
with open('guuki.txt', 'r') as f:   # 読み込みモード(r: read)で開く
    aaa = f.read()                  # ファイルの全内容を文字列としてaaaに代入

print(aaa)                          # 読み込んだ中身を表示（withの外でOK）
```

実行結果の例

```
1 is odd
2 is even
3 is odd
...（中略）...
9 is odd
```

次に，各行をリストとして取り出す readlines() です。データ解析ではこちらが頻出します。

- ファイル名：**read2.py**

```
with open('guuki.txt', 'r') as f:   # 読み込みモードで開く
    aaa = f.readlines()             # 各行を要素とする「リスト」としてaaaに代入

print(aaa)                          # リストの中身を表示
```

実行結果の例

```
['1 is odd\n', '2 is even\n', '3 is odd\n', '4 is even\n', '5 is odd\n', '6 is even\n', '7 is odd\n', '8 is
```

出力されたリストを見ると，各行の末尾に \n がくっついていることが分かります。これはファイルに記録されていた「改行記号」そのものです。

11　while 文による繰り返し

上の 9.8.1 節の問題は $3n+1$ 問題に関するもので，与えられた自然数 n に対して，もし n が偶数なら n を半分にし，n が奇数なら n を $3n+1$ にする，という操作を n の値を表示しながら，最終的に $n=1$ になるまで繰り返し行うものでした。そこでは，for 文を使い，繰り返しの回数は最大 1000 回までと決めていましたが，この数列は何回のステップで $n=1$ となるか分からないし，そもそもどんな自然数から始めても最後に $n=1$ になることは証明されていません（反例も見つかっていません）。このように，何回繰り返すかわからない場合は for 文ではなく，次に説明する while 文によって記述するのが適当です。

11.1　while 文の文法

ある処理の繰り返し行うときに，繰り返す回数がわかっていれば for 文を使いますが，**繰り返す回数が分からない場合や永遠に繰り返しを行う場合には while 文によってプログラムを記述します。**

while 文

```
while 条件A:
    [  処理X  ]
    [  処理X  ]
```

- 条件 A が True であるあいだ，ずっと処理 X を繰り返す。
- 条件 A が False なら処理 X は行われず while 文は終了する。もし A が True だったら処理 X が実行され，X の終了後にまた条件 A をチェックします。同様に，A が True ならば処理 X を行い False なら while 文は終わり。この繰り返しは，条件 A が True である限りずっと続きます。
- オプションとして break, continue が用意されています：
  - 処理 X の実行中に break が現れると強制的に while 文から抜ける。
  - 処理 X の実行中に continue が現れたら，処理 X を中断し条件 A を確認するプロセスに戻る。

図 8　while 文の文法

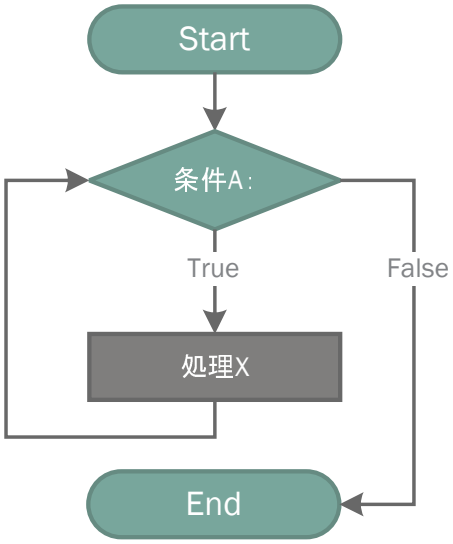


図 9　while 文の処理の流れ

次の while 文の簡単な例を見てみましょう。

- ファイル名：**while1.py**

```
1 a = 1
2 while a<10:    # aが10未満なら，3-4行目を行う。
3     print(a*a, end=' ')     # aの二乗をプリントする
4     a = a+1         # aの値を一つ増やす。2行目に戻る。
5
6 print('owari')    # while文を抜けたらこの処理を行う
```

実行結果の例

```
1 4 9 16 25 36 49 64 81 owari
```

ここで，上の説明での条件 A に相当するものは『a<10』で，処理 X は 3,4 行目です。

上のプログラムのように，while 文の中にカウンター（一つずつ増える変数）と停止条件を書くことで，for 文と同じ処理を行うことができます。上のプログラム while1.py と同じ事を for 文で書くこともできます。

```
1 for a in range(1,10):
2     print(a*a, end=' ')
```

```
3
4 print('owari')
```

11.2　while 文の例

11.2.1　$3n+1$ 問題の数列の生成

while 文を用いて前の節の問題（n が偶数なら半分にし，奇数なら $3n+1$ に変えることを繰り返してできる数列を作る問題）を書き直してみましょう：

- ファイル名：**while2.py**

```
1 a = int(input('Input an integer: '))   # 数を入力させて，その数をaに入れる
2 print(a, end=' ')       # aの値をプリント
3
4 while a != 1:           # a≠1である限り以下の処理を繰り返し行う
5     if a%2 == 0:        # aが偶数なら
6         a = a//2
7     else:               #aが偶数でないなら
8         a = 3*a+1
9     print(a, end=' ')
```

実行結果の例

```
Input an integer: 19
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

前回の課題と同じ結果が得られました。もっと大きな数ではどうなるでしょうか？例えば a=6171 から始めると 261 回の繰り返しの末に最終的に $a=1$ となります。

実行結果の例

```
Input an integer: 6171
6171 18514 9257 27772 13886 6943 20830 10415 31246 15623 46870 23435 70306 35153 105460
..........
... 略 ...
..........
1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

前回のプログラム collatz1.py では繰り返しの回数に上限がありましたが，while2.py では無制限の繰り返しの計算を行うことができます（もちろんコンピューターの容量の制限の中でではありますが）。

11.2.2　繰り返しが終わらないプログラム

次に while 文を使って永遠に終わらないプログラムを作ってみましょう：

- ファイル名：**while3.py**

```
1 j=1
2 while True:      # 条件は永遠に真なので，以下をずっと繰り返す！
3     print('ひつじが'+str(j)+'匹')
4     j = j+1
```

実行結果の例

```
ひつじが 1 匹
ひつじが 2 匹
ひつじが 3 匹
  ...
  ...
```

この while 文では条件がいつでも真（True）なので繰り返し処理が永遠に続きます。プログラムを停止するには CTRL + C もしくは CTRL + Z を入力します。

11.2.3 while 文の中で break, else, continue を使った例

while 文の繰り返しは break を書くことによって止めることができます：

- ファイル名：**while4.py**

```
j=1
while True:
    print('ひつじが'+str(j)+'匹')
    j=j+1
    if j > 123456:  # もし羊の数が123456を超えたらwhile文を終える
        break

print("Too much sheeps! I'm already asleep...zzz")
```

実行結果の例

```
ひつじが 1 匹
ひつじが 2 匹
ひつじが 3 匹
...
...
ひつじが 123456 匹
Too much sheeps! I'm already asleep...zzz.
```

while 文でも else を使うことができます。while 文の条件が False になったときだけに行う処理を書きます。while 直後の条件が成立しなければ else 以下を処理します：

- ファイル名：**while_else.py**

```
a=0
while a<5:
   a = a+1
   print(a)
else:
   print('owari')

print('END')
```

実行結果の例

```
1
2
3
4
5
owari
END
```

上のプログラムは意味のないプログラムですが，後で紹介するように break を while-else 文の中で使うと効果的なプログラムを作ることができます。

次に while 文の中で continue を使った文を作ります。その前に，ある文字列の中に特定の文字 (列) が入っているかどうかを確認する方法を紹介します。次は文字列 Shinshu の中に ns をいう文字が入っているかを調べています。

```
>>> 'ns' in 'Shinshu'
True
>>> 'sn' in 'Shinshu'
False
```

さて，羊を数えたいのですが，数字の 4 は不吉に感じるので，匹数に 4 を含むものは飛ばしながら最大 15 匹まで数えることにしましょう。

- ファイル名：**while_continue.py**

```
a=0
while a<15:
    a=a+1
    if '4' in str(a):    # もしaの中に数字4が含まれていたら
        continue         # 次のprintを行わずにwhileの条件確認に戻る
    else:                # そうでなければ
        print('羊が'+str(a)+'匹') # 羊を数える

print('zzz...')
```

実行結果の例

```
羊が 1 匹
羊が 2 匹
羊が 3 匹
羊が 5 匹
羊が 6 匹
羊が 7 匹
羊が 8 匹
羊が 9 匹
羊が 10 匹
羊が 11 匹
羊が 12 匹
羊が 13 匹
羊が 15 匹
zzz...
```

実は上のような処理は for 文を用いたほうが簡潔に書けます：

```
for i in range(1,16):
    if not '4' in str(i):    # もしiの中に数字4が含まれていないのなら
        print('羊が'+str(i)+'匹')

print('zzz...')
```

11.2.4 フィボナッチ数列の生成

フィボナッチ数列 $1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$，つまり漸化式

$$a_1 = 1, \quad a_2 = 1, \quad a_{n+1} = a_n + a_{n-1}, \quad n = 2, 3, 4, \cdots \tag{2}$$

で定義される数列を表示するプログラムを作ってみましょう。ただし，数列の値が 100000000 を超えたら停止するものとします。この場合でも，いつ a_n が 100000000 を超えるか分からないので while 文を使います。プログラムは次のアルゴリズムに従って書くことにしましょう：

1. maxvalue = 100000000 と置く。
2. a=1 と置く。b=1 と置く。
3. while 文で a<maxvalue が成り立っている間は，次の処理 4-5 を繰り返す。
4. a の値をプリントする。
5. a, b の値をそれぞれ b, a+b に置き換える。
6. a<maxvalue が偽になって while 文を抜けたら，次の数列の値をプリントする。

- ファイル名：**while5.py**

```
maxvalue = 100000000    # maxvalueを右辺の数字にセットする
a = 1      # a = 1 とおく
b = 1      # b = 1 とおく
while a < maxvalue:
```

```
5       print(a)              # ここが
6       a, b = b, a+b         # 繰り返されるブロック
7
8   print('The next value is', a)      # the next value isとaの値をプリントする
```

実行結果の例

```
1
1
2
3
5
...
63245986
The next value is 102334155
```

11.2.5　素数判定プログラム

次に与えられた自然数 (≥ 2) が素数かどうか判定するプログラムを作ってみましょう。与えられた数 n を 2 から順に $\sqrt{n}$ 以下のすべての自然数で割ってみて，どれかの数で割り切れたら n は合成数，そうでなければ n は素数です。そこで次の手順で判定する事にします：

1. 数 n の値を入力させる (ただし $n \geq 2$ とする)。
2. $i = 2$ とする。
3. $i^2 \leq n$ である間は，次の手順 4〜5 を繰り返す。
4. もし，n で i で割り切れたら「n は素数ではありません」とプリントして 3 の繰り返しを終了する。
5. そうでなければ i を $i + 1$ にする。
6. もし 3 の条件全てが偽であったら，「n は素数です」とプリントする。

- ファイル名：**while_primeQ.py**

```
1   n = input('Input an integer(>1): ') # キーボードから入力した数を変数nとする
2   n = int(n)     # nを整数に変換
3
4   i = 2                  # i=2 とおく
5   while i*i <= n:        # i*i≦ nなら以下を繰り返す
6       if n % i == 0:     # もしnがiで割り切れたら
7           print(n, 'is not a prime.')  # nは素数ではないとプリント
8           break                        # while文抜けて12行目以下へ進む
9       else:              # そうでなければ
10          i += 1         # iの値を一つ増やす
11  else:                          # while文の条件が偽となったら
12      print(n, 'is a prime.')    # nは素数であるとプリントする。
13
14  print('おわり')
```

実行結果の例

```
Input an integer: 1237
1237 は素数です。
おわり
```

実は上のプログラムでは $n = 1$ が素数と判定されてしまいます。

11.3　プログラミング言語とはなにか

プログラミング言語とはコンピューターでの処理を記述するための仕組みです。C，Python，Java など様々なコンピューター言語がありますが，書きやすさや速度の問題を別にすれば，これらはすべて同等の能

力を持ちます。例えば，C 言語でできる処理は Python でもできるし，その逆も成り立ちます。これはコンピューター言語がチューリング完全という性質を持つためです。大雑把に言えば，データを任意に読み書きでき，与えられた条件に応じて処理を変化させる事ができ，任意の回数繰り返しができるような言語はチューリング完全です。Python については，リストの操作と if 文，及び while 文だけで（メモリの有限性を除けば）チューリング完全になったといえます。これだけ知っておけば，原理的には，皆さんの工夫次第で，人工知能を作ったり，微分積分をさせたりといった処理が可能なはずなのです。

コンピューターを使うときに最初に必要となる心構えは，自分がすでに万能チューリングマシンを持っているという認識ではないかと思います。

12　練習問題

12.1　問題：素数判定プログラム

上のプログラム（`while_primeQ.py`）を修正し，$n = 1$ が素数でないと判定されるようにしなさい。ヒント：

- 1,2 行目は同じ
- もし $n = 1$ なら，n `is not a prime.` とプリントする。
- そうでなければ，上のプログラムの 4〜12 行目を実行する（適切にインデントを！）。
- `print('おわり')` はなくてもよい。

12.2　問題：ユークリッドの互除法

ユークリッドの互除法とは自然数 a, b の最大公約数を求める次のアルゴリズムの事です。

(i) $b = 0$ なら a が最大公約数である。
(ii) $b \neq 0$ なら a と b の最大公約数は b と r の最大公約数に等しい。ただし r は a を b で割った余り。

次の手順を行う Python プログラムを書きなさい：

1. 整数 a, b を入力させる。
2. $b \neq 0$ である間は，次の処理 3,4,5 を繰り返し行う（while 文を使う）。$b = 0$ ならステップ 6 へ。
3. a を b で割った余りを r と置く。
4. a に b を代入
5. b に r を代入
6. $b = 0$ になったらそのときの a が最大公約数なので a をプリントする。

次がプログラムの例です：

- ファイル名：**gcd.py**

```
1 a = int(input('Input an integer: '))
2 b = int(input('Input an integer: '))
3
4 while b != 0:
5     *********
6     *********
7     *********
8
9 print('The greatest common divisor is', a)
```

上の*****の部分を考え，プログラムを完成させなさい。

13　関数

特定の処理を繰り返すときには for や while で繰り返せばよいのですが，その処理をいろんな場所で自由に呼び出して使いたいときには**関数**を使うのが便利です。関数はひとかたまりの処理に名前を付けて，必要なところで使えるようにしたものです。

Python ではじめから用意されている関数があり，そのようなものを組み込み関数といいます。例えば，すでに何回も使っている

```
print(), int(), float(), str()
```

などは組み込み関数です。他にはリストの総和を求める関数 sum() や，最大値，最小値を返す max(), min() があります：

```
>>> sum([1,2,3,4,5])      # リストの合計を返す関数
15
>>> max([1,2,3,4,5])      # リストの最大値を返す関数
5
>>> min([1, 2, 3, 4, 5])    # リストの最小値を返す関数
1
```

図 10　プログラムを再利用しよう

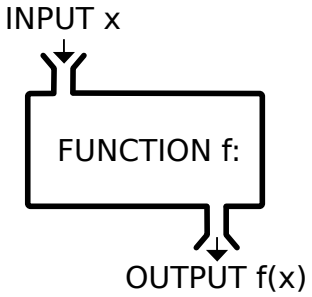


図 11　関数のイメージ

13.1　関数の定義のしかた

関数の定義

```
def 関数名(引数):                    # 引数は関数に渡されるデータの変数名
    [処理Xを記述したブロック]          # 関数が実行するプログラム
    return [戻り値]                  # 計算結果を出力（呼び出し元に返す）
```

- 関数は def 文で定義します。
- def の後に関数名と引数を書いたら，右端には必ずコロン「:」を付けます。
- 関数内の処理を記述するブロックは，定義が終わるまでインデントを保ちます。
- 引数（ひきすう）とは，外部から関数に渡されたデータを受け取るための変数名です。
- 引数や戻り値（return）は，処理内容によっては省略することも可能です。
- 関数名の大文字と小文字は明確に区別されます。
- return 文を実行した時点で関数の処理はその場で終了します。

13.2 関数の例題

それでは次の例題を見てみましょう。

- ファイル名：**func1.py**

```
def greeting():              # ここが
    print("I'm fine.")       # 関数の定義

print('How are you?')        # How are you?と表示する
greeting()                   # 関数を呼び出す
```

実行結果の例

```
How are you?
I'm fine.
```

上のプログラムでは greeting() は呼び出されたら print("I'm fine") を実行する関数です。関数は定義された段階 (上の例では 1,2 行目) では実行されず，プログラム中で呼びだされたときにだけ動きます。上の例では，関数は引数も戻り値も持ちません。

次に**引数と戻り値（return）**があるプログラムの例を見てみましょう。

- ファイル名：**func2.py**

```
def sanjou(a):       # 関数名はsanjou，引数はa
   return a*a*a      # aの３乗を【返す(return)】

print(sanjou(3))     # sanjou(3)で返される値をプリント
print(sanjou(10))    # sanjou(10)で返される値をプリント
```

実行結果の例

```
27
1000
```

sanjou() は引数の三乗を返す関数ですが，上のように返された値をプリントしたり代入したりすることができます。次の例では引数 a が奇数なら odd，偶数なら even を返す関数を定義しています：

- ファイル名：**func3.py**

```
def guuki(a):
  if a%2 == 0:        # もしaが偶数なら
     return 'even'    # evenを返す
  else:               # そうでなければ
     return 'odd'     # oddを返す

print(guuki(1232), guuki(99))
```

実行結果の例

```
even odd
```

次にもう少し実用的な関数を作ってみます。西暦 n 年に対してその年が閏年なら True を，そうでなければ False を返す関数 uruuQ を作ります。さらに西暦 1000 年から 2017 年までの閏年をすべて表示します。

- ファイル名：**func4.py**

```
def uruuQ(n):
    if n%400 == 0:
        return True
    elif n%100 == 0:
```

```
5           return False
6       elif n%4 == 0:
7           return True
8       else:
9           return False
10
11  for i in range(1000, 2026):
12      if uruuQ(i):                    # もしi年が閏年なら
13          print(i, end=' ')           # iをプリントする
```

実行結果の例

```
1004 1008 1012 1016 1020 1024 1028 ..... 1992 1996 2000 2004 2008 2012 2016 2020 2024
```

次に羊を好きなだけ数える関数を定義してみます：

- ファイル名：**func5.py**

```
1  def CountSheep(a):
2      for i in range(1, a + 1):           # iを1からaまで繰り返す
3          print(f'羊が{i}匹')             # f文字列を使ってすっきり表示
4
5  CountSheep(45)       # 関数を呼び出す
```

実行結果の例

```
羊が 1 匹
羊が 2 匹
・・・
・・・
羊が 45 匹
```

13.3　関数とローカル変数

関数の定義の中で新しく定義された変数は，その定義の中だけで**一時的に利用**できます。このような変数のことを**ローカル変数**といいます。これは関数の処理をそこだけで完結させるために補助的に用いるものです。ローカル変数はそれが定義された関数の外では使えません。これによって，別の場所で変数に同じ文字を独立して再利用することができます。ローカル変数でない変数を**グローバル変数**といいます。

次のような数学の問題を考えてみましょう。$N = 100$ とするとき，$S = \sum_{k=1}^{N} k^2$ を求めなさい。ここで，

$$S = 1 + 2^2 + 3^2 + \cdots + 100^2 = \sum_{k=1}^{N} k^2 = \sum_{j=1}^{N} j^2 = \sum_{\ell=1}^{N} \ell^2 \tag{3}$$

なので，$\sum$ の和を取るための変数 k, j, ℓ に特に意味は無く，他の文字[*8]を使っても構いません。そして，(3) の和で使われている k, j, ℓ は $\sum$ の外では意味を持ちません。一方，S と N には決まった数が代入されています。S と N に対応する変数がグローバル変数で，k のようにある処理の外では意味が無い変数がローカル変数に対応します。

次のプログラムは台形の面積を返す関数を定義していて，c と s はローカル変数になります。

- ファイル名：**daikei.py**

```
1  def daikei(a, b, h):    # a:上底, b:下底, h:高さ
2      c = a + b           # cはローカル変数
3      s = 1.0 * c * h / 2 # sはローカル変数
```

[*8] ただし，他で使われていないもの

```
4     return s
5
6  S = daikei(3, 5, 8)     # Sはグローバル変数
7  print(S)
8
9  print(c)    # これはエラー（cはここでは定義されていない）
```

実行結果の例

```
32.0
Traceback (most recent call last):
  File "daikei.py", line 8, in <module>
    print(c)
NameError: name 'c' is not defined
```

13.4 関数の説明の書き方

関数の定義を書くときに，その関数がどういう処理を行うのかをコメント（説明文）として書いておきましょう。コメントを書くことには，次のような意味があります。

1. 関数の処理を書く前にコメントを書くことで，行いたい処理が明確になる。
2. 後になってプログラムを見返したときに，何をやっていたのかすぐに思い出すことができる。
3. 他人がコードを読むときの助けになる。

Python では関数の定義（def）の直後に，その関数の説明を書く慣習があります。そこでは関数が行う処理をトリプルクォーテーション「""" と """」で囲んで文字列として記述します。

たとえば，先ほどの羊を数える関数 CountSheep(a) を定義するときには，次のように説明を書くとよいでしょう：

```
1  def CountSheep(a):
2     """ 羊を a 匹数える関数 """
3     for i in range(1, a + 1):          # iを1からaまで繰り返す
4         print('羊が' + str(i) + '匹')  # 羊がi匹とプリントする
5
6  CountSheep(45)
```

上では 2 行目が関数の説明文（ドキュメンテーション文字列）ですが，これは単なる文字列なのでプログラム実行時には何も影響しません。

13.5 def 文の応用：素数を判定する関数

以前，与えられた数字に対してそれが素数かどうかを判定するプログラム (while_primeQ.py) を作りましたが，それをもとに素数判定の関数を定義してみましょう。次で定義する関数は，引数 n が素数なら True を返し，そうでなければ False を返します。

- ファイル名：**primeq.py**

```
1  def primeQ(n):
2      """ nが素数ならTrue, 合成数ならFalseを返す関数を定義 """
3      if n < 2:
4          return False
5      else:
6          i = 2
7          while i**2 <= n:
8              if n%i == 0:    # nがiで割り切れるなら
```

```
            return False
        i = i + 1
    return True

# 1から100までの素数を列挙する。
for k in range(1, 101):
    if primeQ(k):              # もしkが素数なら
        print(k, end=', ')     # kをプリントする
```

実行結果の例

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
```

上のプログラムは小さい数に対してはすぐに答えを出しますが，例えば

$$111\,111\,111\,111\,111\,111\,111\,113$$

のような大きい数が素数かどうか判定するには長い時間がかかってしまいます（実際この数は素数です）。大きな数の素数判定を行うにはそれなりの工夫が必要になります。多項式時間で素数を判定する AKS 素数判定法や確率的だが高速で素数判定を行う Miller-Rabin 素数判定法といったものが知られています。一般に高速に動作するプログラムを作成するには，高度な数学的知識が必要になります。

13.6 練習問題

13.6.1 最大公約数を計算する関数

12.2 節の練習問題を参考にして，与えられた 2 つの自然数 a, b の最大公約数を返す関数を定義しなさい。そして，その関数を使って 23954187074819 と 8326543 の最大公約数を求めなさい。次のファイルの ********** を推測すればよい。

- ファイル名：**def_gcd.py**

```
def gcd(a, b):
    """ aとbの最大公約数を求める関数
        ユークリッドの互除法により最大公約数を計算する
    """
    **********:        # while文開始
        ********** # (a, b)を(b, a%b)に同時に置き換える
    return *

print(gcd(23954187074819, 8326543)) # 課題で指定された巨大数の計算
```

実行結果の例

```
1067
1
```

13.6.2 円周率を近似する分数

既約分数 a/b で円周率を近似するものを沢山見つけたい。初期条件を $a = 3, b = 1$ とし，$a/b < \pi$ なら a を増やし，$a/b > \pi$ なら b を一つ増やす，という操作を繰り返すことで，そのような既約分数を探してゆく。a/b が既約かどうかは，$\gcd(a, b) = 1$ かどうかで判断すればよい。

次のアルゴリズムを実行する Python プログラムを作成せよ。ファイル名は `approxPi.py` とすること。

(1) `pi = 3.141592653589793` とする。

(2) `a` と `b` の最大公約数を返す関数 `gcd(a, b)` を定義する。

(3) 変数 a，b，d にそれぞれ 3，1，0.14 を入れる（d は誤差評価に用いる）。
(4) a < 32000 かつ b < 10000 である間，以下の (5)–(8) を繰り返す。
(5) apr = a / b とおき，これを円周率の近似値とする。
(6) err = abs(pi - apr) とおく（ここで abs() は絶対値を返す組み込み関数）。
(7) もし a と b が互いに素かつ err < d ならば a，b，apr，err をプリントし，err の値を変数 d に入れる。
(8) もし pi > apr なら a の値を一つ増やし，そうでなければ b の値を一つ増やす。

13.6.3 2つの自然数が既約である割合

自然数 m, n が 1 以外の公約数を持たないときに，それらは既約であるという。もちろん，$\gcd(m, n) = 1$ と，m と n は既約であることは同値である。

$N = 10000$ とする。$1 \leq n, m \leq N$ を満たすすべての自然数 n, m について，(m, n) が既約であるものの個数 A，およびその割合 $A/10000^2$ を表示するプログラムを作成せよ。

補足：勝手に与えた自然数 m, n が既約である確率 P は

$$P = \left(\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots\right)^{-1} = \frac{6}{\pi^2} = 0.6079271018\cdots$$

であることが知られている。

13.6.4 フィボナッチ数列を返す関数

フィボナッチ数列は

$$a_1 = 1, \quad a_2 = 1, \quad a_n = a_{n-1} + a_{n-2} \quad (n = 3, 4, 5, \cdots)$$

で定義される数列である。具体的には $1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$ となる。a_n を計算するには次のようにすればよい。

(1) $n = 1$ または $n = 2$ なら 1 を返す。
(2) $a = 1, b = 1$ とする。
(3) 次の手順 (4) を $n - 2$ 回繰り返す。
(4) 変数 a, b の値を同時に $a + b, a$ に置き換える。
(5) 変数 a の値を返す。

上の手順で，フィボナッチ数列の第 n 項を計算する関数 fib(n) を定義し，fib(1)，fib(2)，$\cdots$，fib(10) を表示せよ。ファイル名は fib0.py とすること。

13.7 関数の再帰的な定義

階乗 $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$ を返す関数を定義してみましょう。このような関数を定義するには for 文を使って，掛け算を繰り返せばよいです。

- ファイル名：**kaijou1.py**

```
1 def kaijou(n):
2     """ nの階乗n!を返す関数をfor文を使って定義する """
3     a = 1
4     for i in range(1, n + 1):
5         a = a * i
```

```
6     return a
7
8 for i in range(1, 10):      # i=1,...,9に対して
9     print(kaijou(i), end=' ')      # kaijou(i)の値を表示する
```

実行結果の例

```
1 2 6 24 120 720 5040 40320 362880
```

さて，関数 $f(n)$ を漸化式で

$$f(0) = 1, \quad f(n) = n \cdot f(n-1)$$

で定義すると明らかに自然数 n に対して $f(n) = n!$ となります。このような関数の定義の仕方を**再帰的な定義** (recursive definition) といいます。Python では関数を再帰的に定義することができます。次のプログラムは階乗 $n!$ を再帰的に定義したものです：

- ファイル名：**kaijou2.py**

```
1 def kaijou(n):
2     """ n!を再帰的に定義する """
3     if n == 0:
4         return 1
5     else:
6         return n * kaijou(n-1)      # ここで自分自身kaijou(n-1)を呼び出す！
7
8 for i in range(10):              # i=0,1,2,…,9に対して
9     print(kaijou(i), end=' ')    # kaijou(i)をプリントする
```

実行結果の例

```
1 1 2 6 24 120 720 5040 40320 362880
```

最大公約数を求めるユークリッドの互除法も同じ手続きの繰り返しである事を利用して，gcd(a, b) を次のように再帰的に定義することも出来ます：

- ファイル名：**gcd_rec.py**

```
1 def gcd(a, b):
2     if b == 0:                # もしb=0なら
3         return a              # aを返す
4     else:                     # そうでなければ
5         return gcd(b, a%b)     # gcd(b, a%b)を返す
6
7 print(gcd(1428, 2618))
```

実行結果の例

```
238
```

上のプログラムでは，答えを得るまでに次のような動作が行われています：

$$\begin{aligned}
\mathtt{gcd}(1428, 2618) &= \mathtt{gcd}(2618, 1428) && (\because 1428 \div 2618 \text{ の余りは } 1428)\\
&= \mathtt{gcd}(1428, 1190) && (\because 2618 \div 1428 \text{ の余りは } 1190)\\
&= \mathtt{gcd}(1190, 238) && (\because 1428 \div 1190 \text{ の余りは } 238)\\
&= \mathtt{gcd}(238, 0) && (\because 1190 \div 238 \text{ の余りは } 0)\\
&= 238 && (\because \texttt{b == 0} \text{ となったので a の値を返す})
\end{aligned}$$

上の例のように，関数がそれ自身による繰り返しで定義可能な場合は，簡潔に関数を定義することが出来ます。

13.8　再帰的な定義の落とし穴と計算量

上で見たように関数が再帰的に定義できるというのはとても便利ですが，実はステップごとに計算量が増えていくような関数の定義には不向きです。以下でこのことを見てみましょう。

フィボナッチ数列

$$a_1 = 1, \quad a_2 = 1, \qquad a_{n+2} = a_{n+1} + a_n \qquad (n = 1, 2, \cdots)$$

を例に取ってみます。

この漸化式をそのまま使って，第 n 項を計算する関数 `fib(n)` を再帰的に定義してみましょう。

- ファイル名：**fib1.py**

```
1  def fib(n):
2      """ フィボナッチ数列の第n項を再帰的に計算する """
3      if n == 1 or n == 2:
4          return 1
5      else:
6          return fib(n-1) + fib(n-2)  # 自分自身を2回呼び出す！
7
8  m = 1
9  while True:
10     print(fib(m))
11     m += 1
```

実行結果の例

```
1
1
2
...
39088169
63245986
```

上のプログラムを実行すると，m が増えるにしたがって値が出るのが目に見えて遅くなるのがわかると思います。

なぜ，これほどまでに計算時間が必要になってしまうのでしょうか。それは，上のように **1 回の関数の中で自分自身を 2 回以上呼び出すような再帰的な定義**では，裏で「同じ計算」を何度も何度も重複して行う，非常に遠回りで非効率な処理が行われています。

例えば，たったの `fib(6)` の計算ですら，次のように次々と自分自身を呼び出して数式が膨れ上がっていきます。

$$\begin{aligned}
\texttt{fib}(6) &= \texttt{fib}(5) + \texttt{fib}(4) \\
&= (\texttt{fib}(4) + \texttt{fib}(3)) + (\texttt{fib}(3) + \texttt{fib}(2)) \\
&= ((\texttt{fib}(3) + \texttt{fib}(2)) + (\texttt{fib}(2) + \texttt{fib}(1))) + ((\texttt{fib}(2) + \texttt{fib}(1)) + 1) \\
&= (((\texttt{fib}(2) + \texttt{fib}(1)) + 1) + (1 + 1)) + ((1 + 1) + 1) \\
&= (((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1) \\
&= 8
\end{aligned}$$

上の計算の展開を見るとわかるように，`fib(6)` を計算するために，`fib` を 2 回呼び出していて，この呼び出しは最終的に `fib(1)`，`fib(2)` になるまで倍々に増え続けます。例えば `fib(46)` をこの方法で計算しようとすると，実に 29 億回以上 も関数が呼び出されることになります。

このように，1 回の処理の中で自分自身を複数回呼び出すような関数を愚直にそのまま計算させると，極めて非効率（計算量爆発）になる場合があるので注意が必要です。

13.9 再帰の工夫

しかし，少し工夫をして一度計算した関数の値を憶えておくことにより，計算の高速化を行うことができます。このような方法を**メモ化** (memoization) といいます。`fib(n)` に対するメモ化の手順には辞書[*9]を使います。

辞書 `memo` には `n` と対応する `fib(n)` の値を

`memo = {1:1, 2:1, 3:2, 4:3, 5:5, 6:8, 7:13, 8:21, ...}`

のように記録しておくことにしましょう。

(1) 初期状態の辞書 `memo = {1:1, 2:1}` を作る。
(2) 関数 `fib(n)` の定義を開始：もしキー `n` が `memo` の中にないなら，辞書 `memo` に `n : fib(n-1) + fib(n-2)` を記録する。そして `memo[n]` に対応する値を返す。

上の手順で `fib1.py` をメモ化してみましょう。

- ファイル名：**fib2.py**

```
1 memo = {1:1, 2:1}
2
3 def fib(n):
4     if n not in memo: # キー nが辞書にないなら
5         memo[n] = fib(n-1) + fib(n-2)  # 辞書に項目を追加
6     return memo[n]
7
8 print(fib(42))
```

実行結果の例

```
267914296
```

上のプログラムでは，`fib(42)` を呼び出すと，計算の過程で一度求まった値がその都度 `memo` に記録されていきます。

上のプログラムでどのように関数が呼び出されて `memo` が作られてるかを見るために次を実行してみましょう。

- ファイル名：**fib3.py**

```
1  memo = {1:1, 2:1}
2
3  def fib(n):
4     print('call fib(', n, ')')      # 関数の何番目が呼び出されたかを表示
5     if n not in memo:
6        memo[n] = fib(n-1) + fib(n-2)
7        print(n,'番目が辞書に追加, memo=', memo) # 辞書に追加されたら表示
8     print(memo[n])       # 関数が値を返す直前にそれをプリント
9     return memo[n]
10
11 print(fib(6))
```

[*9] 辞書は `s = {3:'a', 5:'b', 6:'c'}` のようなデータで，`3:'a'` の `3` をキー，`'a'` をその値と呼ぶことを思い出しましょう。また，`s[7]='d'` とすることで，項目 `7:'d'` を辞書に追加することができます：`s = {3:'a', 5:'b', 6:'c', 7:'d'}`。

実行結果の例

```
call fib( 6 )
call fib( 5 )
call fib( 4 )
call fib( 3 )
call fib( 2 )
1
call fib( 1 )
1
3 番目が辞書に追加, memo= 1: 1, 2: 1, 3: 2
2
call fib( 2 )
1
4 番目が辞書に追加, memo= 1: 1, 2: 1, 3: 2, 4: 3
3
call fib( 3 )
2
5 番目が辞書に追加, memo= 1: 1, 2: 1, 3: 2, 4: 3, 5: 5
5
call fib( 4 )
3
6 番目が辞書に追加, memo= 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8
8
8
```

これをみるとわかるとおり，`fib(6) = fib(5) + fib(4)` のはずですが，実際にはまず `fib(5)` だけが呼び出されていて，`fib(4)` の計算のほうは一旦保留されています。同様にして `fib(4)`，`fib(3)`，`fib(2)` が呼び出されていき，初期条件である `fib(2) = 1` を得ます。

次に `fib(1) = 1` となったので，ここで `fib(3) = 2` と確定し，辞書に `3:2` が追加されます。つぎは `fib(4)` の右側の項である `fib(2)` が呼び出されますが，これはすでに辞書にあるので即座に 1 を返し，`fib(4) = 3` が確定して辞書に追加されます。

このあとにメモ化の効果が現れます。`fib(5)` を確定させるために右側の項である `fib(3)` が呼び出されますが，ログを見ると「`call fib( 3 )`」のすぐ次の行で，辞書を更新することなく一瞬で「2」を返していることがわかります。

最後に `fib(6)` の右側の項である `fib(4)` が呼び出されたときも同様です。わざわざ下の枝（`fib(3)` や `fib(2)`）に潜り直すことなく，すでに辞書に登録されている 3 を一瞬で引き出しています。このように，一度通ったルートの計算を完全にパスすることで，計算量が爆発することなく一撃で答えの 8 に到達できるというわけです。

注意： 関数の最大再帰回数（呼び出しの深さ）は，Python3 ではデフォルトで 1000 回に設定されています。これは，再帰が無限に続いてコンピューターのメモリがパンクするのを防ぐための安全装置です。

そのため，上のプログラムのままで `fib(1001)` を計算させようとするとエラー（`RecursionError`）が発生してしまいます。もし，さらに大きな項まで計算させたい場合は，関数の定義よりも前に次のように記述しておくことで，再帰回数の上限を 2000 回（あるいはそれ以上）に変更することができます。

```
1 import sys
2 sys.setrecursionlimit(2000)
```

13.10 練習問題

13.10.1 円周率に収束する和の練習

和

$$f(n) = \sum_{k=1}^{n} 4\frac{(-1)^{k+1}}{2k-1}$$

は，$f(1) = 4$，$f(n) = f(n-1) + \frac{4(-1)^{n+1}}{2n-1}$ を満たすことを利用して，これを再帰的に定義しなさい。

- ファイル名：**sum1.py**

```
1 def f(n):
```

```
    if n == 1:
        return 4
    else:
        return f(n-1) + 4*(-1)**(n+1)/(2*n-1)

for j in range(1, 1000, 100):
    print(f(j))
```

実行結果の例

```
4
3.1514934010709914
3.1465677471829556
3.1449149035588526
3.144086415298761
3.143588659585789
3.143256545948974
3.1430191863875865
3.142841092554028
3.1427025311614294
```

13.10.2　メモ化の練習 1

上の問題の関数をメモ化して定義し，同様の計算をしなさい。ファイル名は `sum2.py` とすること。

13.10.3　メモ化の練習 2

数列 $\{a_n\}$ を

$$a_1 = 1, \quad a_2 = \frac{3}{4}, \quad a_{n+2} = 1 - \frac{1}{2}a_{n+1}a_n \qquad (n = 1, 2, \cdots)$$

によって定義する。数列 a_n を返す関数 `a(n)` をメモ化を行って定義し，`a(1)`, `a(2)`, $\cdots$, `a(100)` をプリントせよ。

※ メモ化せずに再帰的に a_n を定義した場合，どのようになるか試してみよ。

※ $\lim_{n\to\infty} a_n$ が存在することを示せ。（微分積分の問題）

13.10.4　組み合わせの再帰的な定義

m 個の異なるものの中から，n 個取り出す場合の数（コンビネーション）は，高校で習ったように

$$ {}_m\mathrm{C}_n = \frac{m!}{(m-n)!n!} \tag{4}$$

です。一方で，これは次の漸化式によって表すこともできます：

$$ {}_m\mathrm{C}_n = \begin{cases} 0 & (m < n \text{ または } n < 0) \\ 1 & (m = 0 \text{ または } m = n) \\ {}_{m-1}\mathrm{C}_n + {}_{m-1}\mathrm{C}_{n-1} & (\text{それ以外}) \end{cases} \tag{5}$$

ここで $n < 0$ のときや $m < n$ のときは ${}_m\mathrm{C}_n$ は意味がないので 0 としました。

式 (4) から計算する組み合わせの数を `C(m, n)`，式 (5) を用いて再帰的に定義するものを `D(m, n)` とし，`C(25, 12)` と `D(25, 12)` を計算するのに要する時間を比較せよ。

14　Python の便利な機能

14.1　リストの高度な操作（zip と enumerate）

数学の計算を効率よく行うために，ループ処理をスマートにする 2 つの関数を紹介します。

14.1.1　2 つのリストを同時に束ねる：zip 関数

ベクトル $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ と $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ の内積 $\sum x_i y_i$ を計算する際，zip 関数を使うと添字の管理が不要になります。

- ファイル名：**naiseki.py**

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
print( list(zip(x,y)) )
total = 0
for a, b in zip(x, y):
    total = total + a * b

print("内積の値は：", total)
```

実行結果の例

```
[(1, 5), (2, 6), (3, 7), (4, 8)]
内積の値は： 70
```

14.1.2　添字も同時に手に入れる：enumerate

リストの要素だけでなく，現在の添字（何番目の要素か）を同時に取得したいときは enumerate が便利です。

- ファイル名：**show_seq.py**

```
a = [3, 1, 4, 8, 16]
print( list(enumerate(a)) )  # 添字とセットにする

for i, val in enumerate(a):
    print("a_", i, " の値は ", val)
```

実行結果の例

```
[(0, 3), (1, 1), (2, 4), (3, 8), (4, 16)]
a_ 0  の値は  3
a_ 1  の値は  1
a_ 2  の値は  4
a_ 3  の値は  8
a_ 4  の値は  16
```

14.2　リスト内包表記

数学の集合の内包的記法 $\{x^2 \mid x \in A\}$ に似た感覚で，既存のリストから新しいリストを 1 行で鮮やかに生成する文法を**リスト内包表記**といいます。空のリストに append で要素を追加していく従来の書き方に比べ，コードが簡潔になります。- ファイル名：**naihou1.py**

```
# 0から9までの平方数のリストを普通に作る
aa = []
for i in range(10):
    aa.append(i*i)
print("aa=", aa)

# 0から9までの平方数のリスト（内包表記）
bb = [i * i for i in range(10)]
print("bb =", bb)
```

実行結果の例

```
aa = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
bb = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- ファイル名：**naihou2.py**

```
# 奇数の平方数だけをのリストを作る
cc = []
for i in range(20):
    if i % 2 == 1:
        cc.append(i*i)
print("cc=", cc)

# リスト内包表記で上と同じことをやる
dd = [i*i for i in range(20) if i%2 == 1]
print("dd =", dd)
```

実行結果の例

```
cc = [1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
dd = [1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

二重ループや内積の計算もリスト内包表記を用いると便利です。

- ファイル名：**naihou3.py**

```
# 二重ループ：直積集合のようなペアの生成
lis = ['a', 'b', 'c']
ee = [(i, j) for i in range(1, 5) for j in lis]
print("ee =", ee)

# zip と組み合わせて内積をたったの1行で計算する
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
naiseki = sum([a * b for a, b in zip(x, y)])
print("内積 =", naiseki)
```

実行結果の例

```
ee = [(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'),
 (3, 'a'), (3, 'b'), (3, 'c'), (4, 'a'), (4, 'b'), (4, 'c')]
内積 = 70
```

コラム：名前のない関数（lambda 式）と map

名前をわざわざ付けずに，引数と返り値の対応（写像 $x \mapsto x^2$）だけを指定して関数を定義する特別な文法を **lambda 式**（無名関数）といいます。また，その写像をリストの全要素に適用して像を得る関数が map です。

```
# 引数 a に対して a*a を返す名前のない写像 f を定義
f = lambda a : a * a

A = [1, 2, 3, 4]
# 写像 f をリスト A のすべての要素に一斉に適用する
S = list(map(f, A))
print(S) # [1, 4, 9, 16]

# lambda式はリストの要素として格納することも可能
actions = [lambda x, y : x + y,  lambda x, y : x - y,  lambda x, y : x * y]
for func in actions:
    print(func(3, 5))
```

実行結果の例

```
8
-2
15
```

Python ではリスト内包表記を使う方が直感的で処理も高速なため，lambda 式や map の出番はあまりありませんが，関数型プログラミングの基礎として他言語でも広く使われる非常に重要な概念です。

15　ライブラリ・モジュールの使い方

Python には「標準ライブラリ」と呼ばれる，便利な道具が集められた物はじめから用意されています。これらは、特定の用途ごとに「モジュール」という単位で様々な関数や機能がまとめられています。

モジュールを使うことで，**世界中の賢いプログラマが書いてくれた高度なプログラムを，自分でゼロから書かずにそのまま拝借できます。**

標準ライブラリにどのようなものがあるかは，Python の公式ホームページを見ると詳しく記載されています。例えば，数学に関するモジュール math を呼び出すためには，プログラムの先頭で次のように記述します：

```
1 import math
```

以下では，代表的なモジュール（math, random）を簡単に使って遊んでみますが，ここで 1 つ**注意点**があります。

重要：ファイル名の罠

自分で作成するプログラムのファイル名に，math.py や random.py といった標準ライブラリにある名前を**絶対に付けない**ようにしてください。[a]

[a] Python は import random と書かれたとき，まず「今自分が作業しているフォルダ」の中に random.py というファイルがないかを探します。もし同名のファイルが存在すると，本物の標準ライブラリではなく，あなたの書いたファイル（中身はまだ空か作成途中）を読み込んでしまうため，コードが正しくてもエラーになります。

15.1　math モジュール

数学でよく使う定数や関数が定義されているのが math モジュールです。（このモジュールの本格的な使い方は，後の章で改めて詳しく解説します。）

まずは，プログラムの先頭で import（輸入・取り込み）を宣言して，実際に使ってみましょう。

- ファイル名：**math1.py**

```
1 import math              # mathモジュールを丸ごとインポート
2 print(math.pi)           # mathの中にある円周率 pi を表示
3 print(math.e)            # mathの中にあるネイピア数 e を表示
4 print(math.sin(1))       # mathの中にある sin関数 を使う
```

実行結果の例

```
3.141592653589793
2.718281828459045
0.8414709848078965
```

毎回 math. と書くのが面倒な場合は，次のように「使いたい部品だけ」を名指しでインポートすることもできます。

- ファイル名：**math2.py**

```
1 from math import pi, sin  # mathモジュールから pi と sin だけを直接インポート
2 print(pi)                 # math. を付けずに直接使える！
```

```
3 print(sin(pi / 2))
```

実行結果の例

```
3.141592653589793
1.0
```

15.2　random モジュールで遊ぶ

コンピューターは本来「決められた通りにしか動かない機械」ですが，random モジュールを使うと，サイコロを振ったような「でたらめな数（乱数）」を作り出すことができます[*10]。さっそく，乱数を使った簡単なプログラムを試してみましょう。

- ファイル名：**randamu1.py**

```
1  import random
2
3  # randint(a, b) は a から b までの整数をランダムに返す
4  print(random.randint(1, 100))  # 1から100までの整数をランダムに返す
5
6  # random() は 0.0 以上 1.0 未満の実数をランダムに返す
7  print(random.random())
8
9  # choice(リスト) はリストの中から要素をランダムに1つ選ぶ
10 x = ["red", "green", "blue"]
11 print(random.choice(x))
12
13 # gauss(mu, sigma) は平均 mu, 標準偏差 sigma のガウス分布に従う乱数を返す
14 print(random.gauss(0, 1))
```

（※実行するたびに異なる結果が出力されます。）

乱数の「種（シード）」について：コンピューターが作る乱数は計算によって作られた「擬似乱数」であるため，計算の出発点となる「種（シード）」を指定すると，全く同じ乱数の列を再現することができます。

```
1 random.seed(0)  # 乱数の種を固定する（何度実行しても同じ結果になる）
```

数値計算のシミュレーションで「なぜその結果になったのか」を検証する際には，この再現性が非常に重要になります。

15.3　練習問題

15.3.1　数字当てゲーム

1 から 1000 までの整数がランダムに 1 つ選ばれます。あなたはそれを 10 回以内の予想で当てるゲームを作成しなさい。

【ゲームの進行手順（アルゴリズム）】

1. random モジュールをインポートする。
2. 1 から 1000 までの数を一つランダムに生成し，それを正解の変数 x とする。
3. 『1 から 1000 までの数字が生成されました。10 回以内で当ててみよう！』と画面に表示する。
4. for 文を使って，変数 j を 0 から 9 まで動かしながら以下の処理を繰り返す。

[*10] Python の乱数は「メルセンヌ・ツイスター」という日本人が開発した非常に優秀なアルゴリズムで作られています。

5. input 関数を用いて『残り○回です。予想を入力してください：』と表示し，キーボードから数字を受け取って整数型（int）に変換し，変数 a に入れる。
6. もし $a = x$ なら『正解！』と表示し，break で for 文から抜け出す。
   もし $a < x$ なら『もっと大きいよ』と表示する。
   もし $a > x$ なら『もっと小さいよ』と表示する。
7. for 文が終わった後，もし $x \neq a$ （最後まで当たらなかった）なら，『正解は x でした。残念 Game Over』と表示する。

ファイル名は find_number.py としてください。

コラム：このゲームの数学的な意味（二分探索）

1000 個の候補から 10 回で当てるのは難しそうに思えますが，実は「常に範囲の真ん中の数字」を答えていけば，必ず 10 回以内に正解を当てることができます（$1000 < 2^{10} = 1024$ のため）。このように，調べる範囲を半分ずつに絞っていく手法を情報科学では**二分探索（Binary Search）**と呼びます。

PCR 検査において，複数の検体を混合して同時に検査し，陽性反応が出たグループだけをさらに細かく分けて検査していく「プール検査」という手法も，この二分探索と似た数学的発想（検査回数の劇的な削減）に基づいています。

15.3.2 練習問題：ドイツ戦車の製造数を推定せよ

random モジュールを応用して，歴史上の有名な統計学のエピソードをクイズ形式で楽しんでみましょう。

第二次世界大戦中，連合軍はドイツ軍から捕獲した戦車につけられているシリアル番号（$1, 2, \ldots, N$）から，敵の総製造数 N の推定を行いました。今回はこの歴史的な背景をもとに，プログラムがランダムに集めてくれたヒントを頼りに，あなたが敵の総製造数 N を見破るクイズプログラムを作成してみましょう。

【プログラムの動作手順】

1. random モジュールをインポートする。
2. 正解となる総製造数 N を 500 から 10000 までの範囲でランダムに 1 つ生成する（この N はプレイヤーには秘密）。
3. 捕獲台数 k を 3 から 15 までの範囲でランダムに 1 つ生成する。
4. random.sample を使い，1 から N までの整数から重複なく k 個の数字（戦車番号）を抽出してリスト lis にする。
5. 画面に「敵の戦車を k 両捕獲しました！ 番号は lis です」と表示する。
6. input 関数を使い，『では，敵の総製造数 N を当ててください：』とプレイヤーに推測値を入力させる（整数型 int に変換すること）。
7. 入力された答えが，真の正解 N の **5% 以内** （$0.95N \leq$ 答え $\leq 1.05N$）であれば『お見事！大正解です！』と表示し，外れていれば『残念，ハズレです』と表示する。
8. 最後に『真の正解は N 両でした。』と表示し，さらに後述のコラムの数式が弾き出した「統計学的な推定値」も並べて画面に表示する。

ファイル名は german_tank.py としてください。

コラム：シリアル番号から生産数を割り出す（ドイツ戦車問題）

実際にこのクイズに挑戦してみると，少ない捕獲台数から真の総数を 5% 以内の誤差で当てるのは，人間の直感ではなかなか難しいものです。しかし第二次世界大戦中，連合軍はこれと全く同じ問題に直面しました。

当時，スパイの報告や捕虜の尋問といった従来の諜報活動がもたらす情報は矛盾が多く，分析官の「事前の思い込み」に左右されがちでした。その結果，敵の生産能力を実態よりも大幅に過大評価してしまう傾向にあ

りました。

そこで 1943 年初頭，連合軍の経済戦争部門（Economic Warfare Division）は，捕獲した敵の装備品に刻印されている「シリアル番号」に着目しました。ドイツの製造ラインでは，部品ごとに 1 から順に連続した整数（$1, 2, \ldots, N$）がナンバリングされていたため，回収したいくつかのサンプルの「最大値 m」と「データ数 k」から，背後にある総製造数 N を次のシンプルな公式で推定することが可能となったのです：

$$\text{統計学的な推定値} = m\left(1 + \frac{1}{k}\right) - 1$$

戦後，押収されたドイツ軍の公式な製造記録との「答え合わせ」が行われました。当時の分析官だった R. Ruggles らが 1947 年に発表した論文によると，従来の諜報組織による予測が実態から大きく外れていたのに対し，このシリアル番号分析から導き出した統計的な推定値は驚くほど正確でした。例えば，1940 年 6 月から 1942 年 9 月の生産数のデータでは，諜報部が「1,345 両」と予測して恐れていたのに対し，統計学者たちの推定値は「246 両」，実際の公式記録は「245 両」であり，ほぼピンポイントで的中させていたことが証明されています。

参考：

R. Ruggles and H. Brodie, “An Empirical Approach to Economic Intelligence in World War II,” *Journal of the American Statistical Association*, Vol. 42, No. 237 (1947).

URL: `https://www.cia.gov/readingroom/docs/CIA-RDP79R01001A001300010013-3.pdf`

16　Set 型の操作と応用

16.1　基本的な集合の操作

以前少し紹介しましたが Python には集合 (set) というデータの型があります。これはデータの集まりであることはリストと同じですが，順番がない事と重複が除かれる事がリストとは異なる所です。数学で扱う集合と同じですが，もちろん扱えるのは有限集合だけです。集合は {1,3,4} のように表します。集合同士の演算『union ∪，intersection ∩』をそれぞれ『|, &』で行うことができます。例えば，Python のインタラクティブシェルでの集合の操作は次のようにします：

```
>>> a = {1,3,4}      # 集合 aを定義
>>> b = {3,7}        # 集合 bを定義
>>> c = a | b        # cを集合aとbの和集合とする
>>> c
{1, 3, 4, 7}
>>> d = a & b        # dを集合aとbの共通部分とする
>>> d
{3}
```

集合の演算を行うには，`union()` や `intersection()` というメソッドを使うこともできます。また`|=`を使うことで，集合に他の集合の要素を加えることができます：

```
>>> a.union(b)          # a.union(b)は
{1, 3, 4, 7}            # aとbの和集合を返す
>>> a.intersection(b) # a.intersection(b)は
{3}                     # aとbの共通部分を返す
>>> a
{1, 3, 4}               # 上の操作でaの値が変わるわけではない
>>> b
```

```
{3, 7}                  # bもかわらない
>>> a |= b              # aにbの要素をすべて追加する
>>> a                   # aには
{1, 3, 4, 7}            # bの要素が追加される
```

集合に要素を追加するには add，要素を削除するには remove を使います：

```
>>> a.add(8)            # 集合aに要素8を加える
>>> a                   # aの値を聞く
{1, 3, 4, 7, 8}         # aに要素8が追加された
>>> a.remove(1)         # aから要素1を除く
>>> a
{3, 4, 7, 8}            # aから1が削除された
```

Python3 では辞書も set と同じように {1:4, 2:9, 3:11} のように中括弧で書く事を思い出しましょう。{} は空の辞書 (dict) なので気をつけなければなりません。空集合は set() で表わします。

```
>>> a = {}
>>> type(a)
<class 'dict'>      # {}は辞書です。
>>> a = {1}
>>> a.remove(1)
>>> a
set()               # 空集合はset()と記述する
```

16.2 Set 型の応用——四則演算で 10 を作る遊び

鉄道の切符に書かれている 4 桁の数字（または自動車のナンバーの 4 つの数字）から四則演算で 10 を作るゲームがあります。例えば，切符の数字が 2339 なら，

$$(2+9)-(3/3)=10$$

のようにして数字 10 を作ることができます。4 つの数字はどのような順番で計算してもかまいません。しかし 1111 のような数字からはどうやっても 10 を作ることは出来ません。どのような数字の組なら四則演算で 10 を作ることが出来るでしょうか？ 10 を作ることができるような数字の組を列挙するプログラムを Python で書いてみたいと思います。これを行うプログラムはいろいろな方法で書けると思いますが，ここでは set を使ってみます。（ちなみに切符の端に書かれている 4 桁の数字は 0 をとらないらしいですが，ここでは 0 を含む場合も考えます。）

16.2.1 2 項演算

まずは 2 つの数字の組 a,b の四則演算で作られる数 $a+b, a-b, b-a, a*b, a/b, b/a$ を要素に持つ集合を返す関数を作ります。これを nikou(a,b) としましょう。この場合，得られる数の順番や重複は意味がないので，set を使うのが便利なのです。割り算のときに 0 で割る可能性を排除するために場合分けが必要です：

- ファイル名：**nikou.py**

```
def nikou(a,b):
    if a !=0 and b!=0:      # aもbも0でないときは
        return {a+b, a-b, b-a, a*b, a/b, b/a} # 四則演算からできる集合
    elif b == 0:            # b=0のときは
        return {a, -a, 0}
    else:                   # a=0のときは
```

```
7         return {b, -b, 0}
8
9 print(nikou(3,4))     #3,4から四則演算で作ることができる数の集合を表示
```

実行結果の例

```
{0.75, 1, 7, 12, 1.3333333333333333, -1}
```

16.2.2　3 つの数の四則演算

つぎに，3 つの数 a,b,c の四則演算で作られる数の集合を返す関数 sankou(a,b,c) を作りたいと思います。まず 3 つの数はどの順番で計算するかによって 3 通りの順番があることに注意します。ある二項演算を $\circledast, \ominus$（つまり $\circledast, \ominus$ は＋－×÷のどれか）として

- $(a \circledast b) \ominus c$　：　a, b を先に計算してから次に c との演算を行う。
- $(a \circledast c) \ominus b$　：　a, c を先に計算してから次に b との演算を行う。
- $(b \circledast c) \ominus a$　：　b, c を先に計算してから次に a との演算を行う。

のように 3 つの計算の順番があります。

上のことに注意して 3 つの数 a,b,c に対する 2 項演算から作られる数の集合を返す関数 sankou(a,b,c) を定義してみましょう。

- ファイル名：**sankou1.py**

```
1  def nikou(a,b):
2      if a !=0 and b!=0:
3          return {a+b, a-b, b-a, a*b, a/b, b/a}
4      elif b == 0:
5          return {a,-a,0}
6      else:
7          return {b, -b, 0}
8
9  def sankou(a,b,c):          # a,b,cの四則演算で作られる集合を返す
10     ResultSet = set()       # ResultSetを空の集合とする
11     for i in nikou(a,b):        # a,bから作られる数iに対して
12         ResultSet |= nikou(i,c) # i,cの二項演算から作られる集合をResultSetに追加
13     for i in nikou(b,c):
14         ResultSet |= nikou(i,a)
15     for i in nikou(a,c):
16         ResultSet |= nikou(i,b)
17     return ResultSet        # ResultSetを返す
18
19 print(sankou(4,4,9))           # 4,4,9を 1 回ずつ使った四則演算で作られる集合
20 print('')                      # 改行
21 print(10 in sankou(4,4,9))    # sankou(4,4,9)には10は入っているか？
```

実行結果の例

```
{0.8888888888888888, 1, 1.125, 0, 0.5625, 1.7777777777777777, 0.1111111111111111, 7, 8.0,
 9, 10.0, 3.5555555555555554, 4.444444444444445, 1.75, 6.25, 1.25, 144, 17, 3.25, 20, 25,
 32, 40, -0.8, 52, 72, 0.8, 0.3076923076923077, -32, -20, -9, -8.0, -7, -1.75,
 -3.5555555555555554, -1, -1.25}

True
```

さて，これで 3 つの数字の組から 10 を作れるかどうかを判定する事が可能になりました。そこで三つの数字 $0 \leq a \leq b \leq c \leq 9$ で四則演算によって 10 を作ることができるものをすべて列挙してみましょう：

- ファイル名：**sankou2.py**

```
def nikou(a,b):
    if a !=0 and b!=0:
        return {a+b,a-b,b-a,a*b, a/b, b/a}
    elif b == 0:
        return {a,-a,0}
    else:
        return {b, -b, 0}

def sankou(a,b,c):       # a,b,cの四則演算で作られる集合を返す
    ResultSet = set()    # ResultSetを空の集合とする
    for i in nikou(a,b):         # a,bから作られる数iに対して
        ResultSet |= nikou(i,c) # i,cの二項演算でできる集合をResultSetに追加
    for i in nikou(b,c):
        ResultSet |= nikou(i,a)
    for i in nikou(a,c):
        ResultSet |= nikou(i,b)
    return ResultSet     # ResultSetを返す

counter = 0      # counterという変数を作り0にセットする

for i in range(10):
    for j in range(i,10):
        for k in range(j,10):
            if 10 in sankou(i,j,k):    # もし10がsankou(i,j,k)の要素なら
                print(i,j,k)           # i,j,kを表示して
                counter = counter + 1 # numberを1増やす

print('10を作れる数字の三つ組みの数は', counter, '個')
```

実行結果の例

```
0 1 9
0 2 5
...
...
9 9 9
10 を作れる数字の三つ組みの数は 75 個
```

16.2.3　4 つの数字の四則演算

つぎに 4 つの数 a,b,c,d から作ることのできる数の集合を返す関数を作ります。どれか 2 つの数を先に計算することで，3 つ組の場合に帰着します。例えば，a,b を先に計算してその計算結果と c,d との四則演算で作ることのできる数の集合を作るには，a,b のすべての計算結果 i に対して sankou(i,c,d) を作ればよいです。a,b,c,d のうちどの 2 つを先に計算するかについては，${}_4C_2 = 6$ 通りの場合があります。つまり，最初に計算する数は全部で (a,b),(a,c),(a,d),(b,c),(b,d),(c,d) の 6 通りです。このような手順で作られる数字の集合を返す関数を yonkou(a,b,c,d) とします。この関数は次のように定義します。

```
def yonkou(a,b,c,d):
    ResultSet = set()
    for i in nikou(a,b):              # a,bの計算結果iに対して
        ResultSet |= sankou(i,c,d)   # i,b,cから作られる数をResultSetに追加
    for i in nikou(a,c):
        ResultSet |= sankou(i,b,d)
    for i in nikou(a,d):
        ResultSet |= sankou(i,b,c)
```

```
    for i in nikou(b,c):
        ResultSet |= sankou(i,a,d)
    for i in nikou(b,d):
        ResultSet |= sankou(i,a,c)
    for i in nikou(c,d):
        ResultSet |= sankou(i,a,b)
    return ResultSet
```

結局，0 から 9 までをとる 4 つの数字 a, b, c, d で $a \leq b \leq c \leq d$ かつこれらの四則演算で 10 を作ることができる組をすべて列挙するプログラムを作成するには次のようにします：

- ファイル名：**make10.py**

```
def nikou(a,b):
    if a !=0 and b!=0:
        return {a+b, a-b, b-a, a*b, a/b, b/a}
    elif b == 0:
        return {a,-a,0}
    else:
        return {b, -b, 0}

def sankou(a,b,c):          # a,b,cの四則演算で作られる集合を返す
    ResultSet = set()       # ResultSetを空のリストとする
    for i in nikou(a,b):          # a,bから作られる数iに対して
        ResultSet |= nikou(i,c)   # i,cの演算から作られる集合をResultSetに追加
    for i in nikou(b,c):
        ResultSet |= nikou(i,a)
    for i in nikou(a,c):
        ResultSet |= nikou(i,b)
    return ResultSet          # ResultSetを返す

def yonkou(a,b,c,d):
    ResultSet = set()
    for i in nikou(a,b):              # a,bの計算結果iに対して
        ResultSet |= sankou(i,c,d)    # i,b,cから作られる数をResultSetに追加
    for i in nikou(a,c):
        ResultSet |= sankou(i,b,d)
    for i in nikou(a,d):
        ResultSet |= sankou(i,b,c)
    for i in nikou(b,c):
        ResultSet |= sankou(i,a,d)
    for i in nikou(b,d):
        ResultSet |= sankou(i,a,c)
    for i in nikou(c,d):
        ResultSet |= sankou(i,a,b)
    return ResultSet

counter = 0

for i in range(10):
    for j in range(i,10):
        for k in range(j,10):
            for l in range(k,10):
                if 10 in yonkou(i,j,k,l):
```

```
                    print(i,j,k,l)
                    counter = counter+1

print('10を作れる数字の4つ組みの数は', counter, '個')
```

実行結果の例

```
.......
.......
.......
8 8 9 9
9 9 9 9
10 を作ることができる 4 つの数字の組の数は 552 個
```

上のプログラムは結果的にはうまく動いていますが，本来なら桁落ちに注意が必要です。二項演算を浮動小数点で行っているために，演算の結果が正確には 10 のはずなのに，誤差により 10 ではなくなっている可能性があるからです。ただ，10 に非常に近い数は 10 であると認識されます：

```
>>> 9.99999999999999999 == 10
True
>>> 9.9999999999999999 == 10
False
```

16.3　練習問題

5 つの数字 $a, b, c, d, e (0 \leq a \leq b \leq c \leq d \leq e \leq 9)$ を 1 回ずつ使い四則演算（加算，減算，乗算，除算）で 100 を作ることを考える。100 を作れるような全ての数の組及びその個数を表示するプログラムを作成せよ。ファイル名は `make100.py` とすること。

17　GUI アプリの作り方

Python には標準で Tkinter というグラフィカルなソフトを作るためのツールが用意されています。ここでは，Tkinter を使って簡単な GUI プログラムを作る方法を紹介します。

17.1　Tkinter の基本

Tkinter は Python のモジュールです。次のようにインポートして使います。

- ファイル名：**gui1.py**

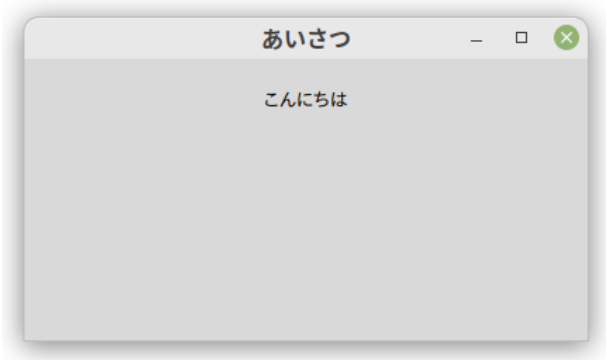
```
from tkinter import *

win = Tk() # ウィンドウを一つ定義
win.geometry("400x200")  # 窓のサイズ(横x縦)

moji = Label(win, text='こんにちは')
moji.pack()  # 窓に配置する

## ウィンドウを開始
win.mainloop()
```

Tkinter にはウィジェット（Widget，部品）が色々用意されていて，上の Label はウィジェットの一つです。

次のようにして，ウィンドウのタイトルを付けたり，文字の枠サイズを指定したりできます。

- ファイル名：**gui2.py**

```
1  from tkinter import *
2
3  win = Tk() # ウィンドウを一つ定義
4  win.geometry("400x200")  # 窓のサイズ（横x縦）
5  win.title("あいさつ")
6
7  label = Label(win, text='こんにちは')
8  label.pack(padx=20, pady=20)  # 枠サイズを指定
9
10 ## ウィンドウを開始
11 win.mainloop()
```



17.2　ボタンの設置

ボタンはウィジェットとして用意されていますが，まずボタンを押したときの動作を関数で書いておく必要があります。そして Button ウィジェットを定義するときに，その関数を指定します。

- ファイル名：**gui3.py**

```
1  from tkinter import *
2
3  win = Tk()
4  win.geometry("400x200")  # 横x縦
5
```

```
# ボタンの動作を表す関数を定義
def hello():
    print('なますて〜')

btn1 = Button(win, text="click me", command=hello) # ボタンを定義
btn1.pack() # ボタンを詰める

win.mainloop() # ウィンドウを開始
```

上を実行すると，「click me」と書かれたボタンのあるウィンドウが表示され，ボタンを押すとターミナル (CUI) の方に「なますて〜」とプリントされます。

次に，ターミナルではなくウィンドウに挨拶文が表示されるようするには次のようにします。

- ファイル名：**gui4.py**

```
from tkinter import *

win = Tk() ### メインのウィンドウの定義
win.geometry("400x200")  # ウィンドウサイズ
win.title('あいさつ')

aaa = Label(win, text = "No Data") ### 表示する文字の定義


def india(): # ボタン1の動作
    aaa["text"] = 'なますて〜'
def japan(): # ボタン2の動作
    aaa["text"] = 'こんにちは〜'

btn1 = Button(win, text="click me(india)", command=india) # ボタン1を作る
btn1.pack()

btn2 = Button(win, text="click me(japan)", command=japan) # ボタン2を作る
btn2.pack()

aaa.pack() ### aaaを配置

win.mainloop() ### ウィンドウを開始
```

ボタンを押すと，ウィンドウ内に「なますて〜」や「こんにちは〜」と表示されます。

次のようにしてテキストを文字列を削除するボタンを作ったりすることができます。

- ファイル名：**gui5.py**

```
from tkinter import *

win = Tk() # メインのウィンドウの定義
win.geometry("400x200")  # ウィンドウサイズ
win.title('あいさつ')

aaa = Label(win, text = "No Data") # 表示する文字の定義(まだpackしない)

def india(): # インドのあいさつ
    aaa["text"] = 'なますて〜'
def japan(): # 日本のあいさつ
    aaa["text"] = 'こんにちは〜'
def clear_aaa(): # aaaの文字を削除する関数
    aaa["text"] = ""

btn1 = Button(win, text="click me(india)", command=india)
btn1.pack()
btn2 = Button(win, text="click me(japan)", command=japan)
btn2.pack()
btn3 = Button(win, text="クリア", command=clear_aaa)
btn3.pack()

aaa.pack() # aaaを配置

win.mainloop() # ウィンドウを開始
```

17.3　文字列の取得

画面に文字を入力する箇所を作ったり，そこから文字列を取得したりするには Entry，StringVar ウィジェットを使います。

- ファイル名：**gui6.py**

```
from tkinter import *

win = Tk() ### メインのウィンドウの定義
win.geometry("400x200")  #ウィンドウサイズ
win.title('入力文字の取得')

aaa = Label(win, text='なまえ') ### テキストウィジェットを定義
aaa.pack() # 配置する

bbb = Label(win, text='') ### ラベルウィジェットを定義

### エントリー（入力する場所）
data0 = StringVar()
ccc = Entry(win, textvariable= data0)
ccc.pack()

# 関数を定義
def getname():
    hh = ccc.get()     # 入力された文字列を取得し変数hhに入れる
```

```
20      bbb["text"] = "こんにちは"+hh+"さん！" # bbbの文字を変更
21
22  btn1 = Button(win, text="btn1", command=getname) # ボタンを定義
23  btn1.pack()
24
25  bbb.pack()  # bbbを配置
26
27  win.mainloop() ### ウィンドウを開始
```

17.4　簡単な計算機の作成

入力されたものを計算するだけの簡単な GUI 計算機を作ります。文字列を式にして実行するには eval 関数を用います。これは

```
1  >>> moji = '2+3'
2  >>> eval(moji)
3  5
```

のように動作します。次は入力されたものを計算して表示するプログラムです。

- ファイル名：**gui7.py**

```
1  from tkinter import *
2
3  win = Tk() ### メインのウィンドウの定義
4  win.geometry("400x200")  # ウィンドウサイズ
5  win.title('簡単な計算機')
6
7  aaa = Label(win, text='計算式') # ラベルを定義
8  aaa.pack()
9
10 ### エントリー（入力する場所）
11 date0 = StringVar()      # 文字列を入れる変数
12 bbb = Entry(win, textvariable= date0) # エントリーの定義
13 bbb.pack()  # つめる
14
15 ccc = Label(win, text='') # 計算結果を表示する部分
16
17 # 関数を定義
18 def jikko():
19     hh = bbb.get() # 入力情報を取得
```

```
20      hh = str(eval(hh)) # 計算
21      ccc["text"] = "計算結果:"+hh # 結果の出力
22
23  def clearentry():
24      bbb.delete(0, "end") # エントリーのクリア
25
26  btn1 = Button(win, text="RUN", command=jikko) # RUNボタン
27  btn1.pack()
28  btn2 = Button(win, text="Clear", command=clearentry) # Clearボタン
29  btn2.pack()
30
31  ccc.pack() # 計算結果を表示するウィジェットをつめる
32
33  win.mainloop() ### ウィンドウを開始
```

math ライブラリをインポートしておけば，sin, log などの計算もできるようになります。

- ファイル名：**gui8.py**

```
1   from tkinter import *
2   from math import *
3
4   win = Tk() # メインのウィンドウの定義
5   win.geometry("400x200")  # ウィンドウサイズ
6   win.title('簡単な計算機')
7
8   aaa = Label(win, text='計算式') # ラベルを定義
9   aaa.pack()
10
11  # エントリー（入力する場所）
12  data0 = StringVar()
13  bbb = Entry(win, textvariable= data0)
14  bbb.pack()
15
16  ccc = Label(win, text='') # 計算結果を表示する部分
17
18  # 関数を定義
19  def jikko():
20      hh = bbb.get() # 入力情報を取得
21      hh = str(eval(hh)) # 計算
22      ccc["text"] = "計算結果:"+hh # 結果の出力
23
24  def clearentry():
25      bbb.delete(0, "end") # エントリーのクリア
```

```
26
27 btn1 = Button(win, text="RUN", comman=jikko) # RUNボタン
28 btn1.pack()
29 btn2 = Button(win, text="Clear", command=clearentry) #Clearボタン
30 btn2.pack()
31
32 ccc.pack() # 計算結果を表示するウィジェットをつめる
33
34 win.mainloop() ### ウィンドウを開始
```

第 II 部
応用編

18　Jupyter ノートブック

Jupyter（ジュピター）ノートブックは，Python などのコードをその場で実行し，結果を記録できるドキュメント形式です。また，Markdown（マークダウン）形式で数式や解説を書くこともできるため，「計算・実行可能なドキュメント形式」と言えます。

18.1　VS Code へのインストールと準備

VS Code に Jupyter の拡張機能を入れて，Jupyter ノートブックを利用します。

VS Code の拡張機能（左側の四角いアイコン）を開き，検索窓に `jupyter` と入力します。検索結果から，提供元が **Microsoft** と書かれている公式の **「Jupyter」** 拡張機能をインストールしてください。

インストールが完了したら，メニューの「ファイル」→「新しいファイル」をクリックし，`test01.ipynb` という名前で保存してください（Jupyter ノートブックの拡張子は `.ipynb` です）。

18.1.1　セキュリティに関する重要な設定：制限モードの解除

通常，VS Code は安全なテキストエディタですが，Jupyter ノートブックを使用すると，VS Code 上で実際にプログラムを動かすことになります。もし外部からダウンロードした出所不明のドキュメントを不用意に開いて実行してしまうと，**悪意あるプログラムが勝手に実行され，PC 内の重要なデータが破壊・漏洩したりする危険性** があります。このようなトラブルを防ぐため，VS Code で新しいフォルダやファイルを開いた直後は，機能を制限する **「制限モード」** になっていることがあります。

画面の左下に [制限モード] と書かれており，画面上部に

> 「制限モードは、安全なコード参照のためのものです。すべての機能を有効にするには、このウィンドウを信頼します。」

というメッセージが表示された場合は，[管理] をクリックし，作成者を信頼する手続きを行ってください。

18.1.2　編集画面の確認とカーネルの解説

`.ipynb` ファイルを開くと，画面には図 12 のような画面が表示されます。

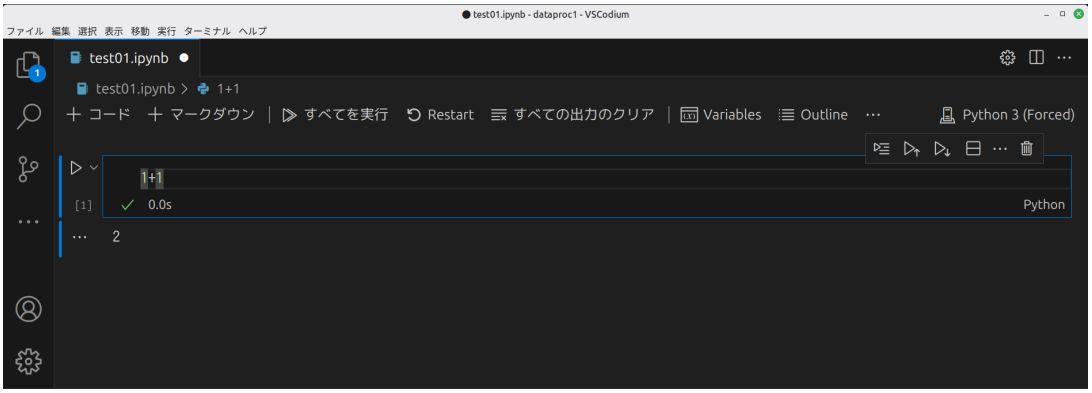


図 12　Jupyter Notebook の初期画面

まず，画面の右上に注目してください。そこには，このノートブックを動かすための**「カーネル」**を選択するエリア（Python 3.14.xx などと四角い枠で囲まれた部分）があります。最初は自動的に Python 3.13.xx（環境によっては Python 3 など）と表示されていると思います。

「カーネル」とは，私たちが書いたプログラムを裏で代わりに計算し，実行結果を返す**「計算係」**のことです。この環境を切り替えることで，Jupyter ノートブックでは Python だけでなく，Julia や R（統計分析ソフト）など，様々なプログラミング言語を動かすことができるようになります。今回は Python を使いますので，画面右上の枠内が Python 3.xx.xx（インストールしたバージョン）になっていれば，正しく設定されています。

画面中央にある ＋ コード の部分をクリックすると，長方形の枠（セル）が現れ，文字入力のためのカーソルが点滅します。ここで Python などのプログラムを記述し，1 行ずつ実行してその場で結果を確認していくことができます。

18.2 Jupyter Notebook の実行

試しに，最初のセルに 1+1 と入力して Shift + Enter を押してみましょう。セルの左側にある「再生（▷）ボタン」をクリックすることでも実行できます。

【初回実行時の注意】

初めて実行する際，画面の右下に「Jupyter の実行には ipykernel パッケージが必要です。インストールしますか？」というポップアップが表示されます。迷わず**「インストール」**をクリックし，約 30 秒〜1 分ほど待ってください。

セルの下に 2 と表示されれば，環境構築は成功です。

18.3 トラブルシューティング（実行できない・動かない場合）

もし上記の手順で結果 2 が出力されず，エラーが出たり無限にインストールの画面が繰り返されたりする場合は，PC 環境によって様々な原因が考えられます。以下の手順を順番に試してください。

- **ステップ 1：画面の「許可」はすべて OK にする**

お使いの PC のセキュリティ設定によっては，プログラムを実行する際やパッケージをインストールする際に，VS Code 自身や Windows，ウイルス対策ソフトから「通信の許可」や「実行の許可」を求められる画面が出ることがあります。これらはすべて許可してください。

- **ステップ 2：PC を再起動してもう一度試す**

インストール処理が内部で中途半端に止まっている場合があります。一度作業内容を保存し，**パソコン自体を完全に再起動** してから，もう一度 VS Code を開いて実行（Shift + Enter）を試してください。

- **ステップ 3：【環境の相性による対策】Python の入れ替え**

本授業では「Microsoft Store 版」からの Python のインストールを推奨していますが，お使いのパソコンのユーザー名（アカウント名）の長さや，フォルダの設定（OneDrive の同期など）との組み合わせによっては，稀にプログラムの通信や追加インストールが正常に受け付けられない問題が発生するようです[*11]。

ステップ 2 まで試しても同じ画面が繰り返される場合は，この相性問題に該当している可能性があります。以下の手順で公式サイト版の Python に入れ替えてみてください。

1. Windows の「設定」→「アプリと機能（またはインストールされているアプリ）」を開き，現在入って

[*11] MS Store がアホみたいに長いフォルダ名を付けるせいで，Windows のファイル名の長さの上限（260 文字制限）をかなり使い切ってしまい，そこに新たにサブフォルダを追加するとエラーになってしまう現象が確認されています。

いる **「Python」をすべてアンインストール（削除）** します。

2. Python の公式サイト（https://www.python.org/）から最新のインストーラーをダウンロードし，実行します。
3. インストーラーの最初の画面最下部にある **「Add python.exe to PATH」のチェックボックスに必ずチェックを入れてから**，[Install Now] をクリックします。
4. インストールの終盤（または完了直後）に，自動的に PowerShell という黒い画面（ターミナル）が立ち上がり，入力を求められます。画面の指示に従って，すべて **「y」**（または **「Y」**）を入力してエンターキーを押してください。これにより，パスの制限解除や最適化が適用され，正しく設定が完了します。
5. インストール完了後，VS Code を起動し，最初のセルを実行してみてください。

18.4 Jupyter ノートブックの基本

Jupyter ノートブックの重要な性質として，**「セルの最後に書かれた評価結果のみが出力される」**というルールがあります。新しいセルを作成し，次のように入力して実行してみましょう。

```
a, b, c = 1, 2, 3
a + b   # こちらの計算結果は画面に表示されない
b + c   # 最後に書かれたこちらのみが表示される
```

途中計算もすべて表示したい場合は，以下のように `print()` 関数を使用します。

```
a, b, c = 1, 2, 3
print(a + b)
b + c
```

Jupyter ノートブックを実行した作られた変数は，カーネルを初期化するまで内部で保持されています。上に続いて次のセルで

```
c
```

と入力すれば 3 が表示され，変数 `c` の値が保持されていたことがわかります。

Jupyter ノートブックは，好きなセルからバラバラに実行することができますが，変数は**「実行した順番」**に記憶されていきます。そのため，「上のセルを動かさずに下のセルだけを動かす」といった変則的な操作をすると，変数の中身がごちゃごちゃになり，正しいコードを書いているはずなのにエラーが発生することがあります。その場合は，ノートブックの上部メニューにある **[Restart]** ボタンをクリックして，カーネルを一度リセットしてください。

また，[▷▷すべてを実行] をクリックすると，一番上のセル空順番にすべてのコードの書かれたセルが実行されます。

なお，不要になったセルは，セルの右上に表示される**「ゴミ箱のアイコン（セルの削除）」**をクリックすることで消去できます。間違えて消してしまった場合は，`Ctrl + Z`（Mac は `Cmd + Z`）で元に戻せます。

18.5 Jupyter ノートブックでの Markdown 形式の活用

Jupyter ノートブックはプログラムを動かすだけでなく，**Markdown（マークダウン）**に従い，数式（LaTeX 形式）や見出しを交えた見栄えの良い解説ノートを同時に作成することができます。

Markdown とは，文字の前に特定の記号（#など）をつけることで，文章の構造（見出し，箇条書き，強調など）を明確にするための文章の書き方のことです。このルールに従って記述すると，Jupyter ノートブックがその構造を読み取り，自動的に Web ページのような見栄えに表示してくれます。

画面上部などにある ＋ マークダウン をクリックすると，Markdown モードのセルが新しく作成されます。

または，既にあるセルの右下を確認すると [Python] や [Markdown] といった文字が書かれており，これが現在のセルの「モード」を決めています。ここをクリックして [Markdown] モードに直接切り替えることもできます。

Markdown モードのセルに，以下のように入力してみましょう。

```
# 数値計算の基本

今日はJupyterの練習をします。
インラインで数式も綺麗に書けます： $f(x) = x^2 + 2x + 1$

独立した数式（別行立て）は，ドルマーク2個で囲むことで表示できます。
$$ \int_{-\infty}^\infty e^{-x^2}dx = \sqrt{ \pi } $$
```

入力が終わったら，コードの時と同じように **Shift + Enter（または再生ボタン）** を押してください。背側で **MathJax** という数式描画プログラムが働き，LaTeX 形式で書いた数式が美しいグラフィックとして画面に表示されます。

【※重要】表示された文字を修正したい場合

一度実行して綺麗に表示された Markdown セルを書き換えたい（編集に戻りたい）ときは，そのセルの上を **ダブルクリック** するか，セルを選択した状態で **Enter** キーを押してください。元の入力画面に戻ります。

【注意】

Markdown モードになっているセルでは，Python のコード（プログラム）は動きません。もし Python コードとして実行したいのであれば，右下の [Markdown] という文字をクリックして，Python モードに切り替えてください。

19　SymPy（数式処理パッケージ）

19.1　導入と使い方

Python には SymPy（symbolic python）という数式処理のライブラリが用意されています。数値を分数のまま扱ったり，x, y といった記号のまま文字式を厳密に計算したりすることができます。

SymPy を使うにはコードの最初に

```
from sympy import *
```

と書いておきます。Jupyter ノートブックであれば，最初のセルで一度実行しておけば，カーネルを初期化するまで有効になります。

次を実行してみましょう。

```
from sympy import *
sqrt(12)    # √12の計算（自動で簡素化される）
```

出力結果： $2\sqrt{3}$

次のようなこともできます：

```
var('x')              # xを文字式で使う宣言
solve( x**2 - 2, x)   # x^2-2=0をxについて解く
```

出力結果： `[-sqrt(2),sqrt(2)]`

Jupyter Notebook で SymPy を使用する際は，次のように数式の表示（MathJax）を有効化するのもよいでしょう：

```
init_printing()  # Jupyter上での数式表示を有効化
var('x')
solve( x**2 - 2, x)
```

出力結果： $[-\sqrt{2}, \sqrt{2}]$

19.2　厳密計算と近似値（小数）のコントロール

19.2.1　数式の厳密な保持

SymPy の最大の特徴は，無理数や数学定数，分数を近似値（小数）に落とすことなく，記号のまま厳密に扱える点にあります。代表的な数や定数の記法は以下の通りです。

- 有理数（分数）： `Rational(分子, 分母)` または `S(分子)/分母`
- 円周率 π： `pi`
- 自然対数の底 e： `E`（※大文字の E）
- 虚数単位 i： `I`（※大文字の I）
- 平方根 $\sqrt{\ }$： `sqrt()`

Python 標準の割り算「`/`」を使って `5/4` と書くと，SymPy が処理する前に Python が自動的に `1.25` という有限桁の小数（近似値データ）に変換してしまい，数学的な厳密性が失われます。分数を分数のまま厳密に保持したいときは，`Rational` などを利用します。

```
# 有理数の足し算（自動的に通分・約分されます）
Rational(3, 4) + Rational(5, 6)
```

出力結果： $\frac{19}{12}$

```
# 三角関数やオイラーの公式の厳密計算
print(sin(pi/3))
print(E**(pi * I))
```

出力結果： $\frac{\sqrt{3}}{2}$
-1

19.2.2　数値評価と計算誤差：.evalf(), .n()

これらの厳密な数式から具体的な小数の値（近似値）を知りたい場合は，後ろに `.evalf()` または `.n()` を付けます。この 2 つはまったく同じ命令（メソッド）です[*12]。引数に数字を渡すことで，任意の計算精度（表示桁数）を自由にコントロールできます。

```
>>> from sympy import *
>>> pi.evalf()       # デフォルトは15桁
3.14159265358979
```

[*12] 変数 `n` を他で使う場合は `evalf` を使うとよいでしょう。

```
>>> pi.n()            # 上とまったく同じ
3.14159265358979
>>> pi.evalf(30)      # 30桁の精度で表示
3.14159265358979323846264338328
>>> pi.n(30)          # 上とまったく同じ
3.14159265358979323846264338328
>>> sqrt(3).evalf(20)    # 20桁の精度で表示
1.7320508075688772935
```

一度小数（近似値）に変換してから計算を行うと，コンピュータ特有の丸め誤差が発生します。

```
# 円周率を一度小数にしてから、オイラーの公式を計算してみる
a = pi.n()
expr = E**(a * I)
expr.n()
```

出力結果： $-1.0 + 1.0 \cdot 10^{-16} i$

数学的には $e^{i\pi} = -1$ であり結果は実数になるはずですが，プログラム上では極小の虚数部分（$1.0 \times 10^{-16} i$）が残ってしまいました。このわずかな虚数データの混入は，のちの処理（グラフ描画など）で「実軸上にプロットできない」といった原因不明のエラーを引き起こす原因になります。こういったエラーを防ぐためには，途中まで厳密に計算し，最後に.evalf() をつけて数値にする。（遅い）もしくは，初めから float で計算しておき，最後に実部だけを取り出すことです：

```
>>> a = pi.n()
>>> expr = E**(a * I)
>>> re(expr)
-1.00000000000000
```

19.3 SymPy の内部データ構造と表示形式の性質

SymPy を用いた計算において注意しなければならないのは、画面上の表示形式と内部で保持されているデータ構造が異なっている点です。

SymPy はインタラクティブな利用において、数式を標準的な数式表記に近い形で人間が読みやすい形式に出力する仕様（**プリティプリント機能**）を持っています。

たとえば、有理数 Rational(3, 2) は、通常の出力では 3/2 と表示されますし、Jupyter Notebook 上では $\frac{3}{2}$ という分数の形で画面に表示されます。

オブジェクトの数学的な構造（クラスや属性）を正確に確認するには、srepr() という関数を使用します。

Python インタラクティブシェルでの次の実行例を通じて、SymPy のデータ管理の仕組みを観察してみましょう。

```
>>> from sympy import *
>>> a = Rational(3, 2)
>>> a
3/2                          # 見た目は 3/2 だが
>>> Rational(3, 2) == 3/2 # 実際には
False                        # 内部データ構造が異なる
```

```
>>> srepr(Rational(3, 2))
'Rational(3, 2)'
```

表示形式と内部の構造に注意しながら、いくつかの計算を確認してみましょう。

```
>>> b = 1/3; srepr(b)
'0.3333333333333333'          # こちらは通常のPythonのfloat型（2進数の近似値）
>>> c = S(1)/S(3); srepr(c)
'Rational(1, 3)'              # SymPyの有理数（厳密解）
>>> srepr(1)
'1'                           # Python標準の int型 の 1
>>> srepr(S(1))
'Integer(1)'                  # SymPyの整数型（Integer）の 1
>>> srepr(sin(pi))
'Integer(0)'                  # 数理的評価により厳密な整数 0 に変換された状態
>>> e = pi.n(30)
>>> srepr(e)
"Float('3.14159265358979323846264338327933', precision=103)" # 高精度なFloat型
```

上の実行結果から、通常の数値 '1' と Integer(1) という 2 つの表現が存在することが分かります。後者は SymPy 独自の整数を表しています。SymPy は、Python 標準とは異なる独自の数の体系を持っています。通常の Python の数値データを SymPy の世界のオブジェクトへと変換するのが、S()（sympify の略）という関数です。

では、厳密な値と近似値（Float）が混ざった計算はどのように行われるか見てみましょう。

```
>>> a = 1 + S(1)     # Python標準の 1 と SymPyの Integer(1) の和
>>> a
2                    # 画面上の出力は 2
>>> srepr(a)
'Integer(2)'         # 内部ではSymPyの整数型に自動的に統一されている
>>> b = S(1) + 1.5
>>> b
2.50000000000000
>>> srepr(b)
"Float('2.5', precision=53)"
>>> srepr(sqrt(2))
'Pow(Integer(2), Rational(1, 2))' # 内部では「2の1/2乗」として数式構造を保持
>>> sqrt(2) + 3/2
sqrt(2) + 1.5
>>> c = sqrt(2) + 2/3
>>> c
0.666666666666667 + sqrt(2)
>>> srepr(c)
"Add(Float('0.6666666666666663', precision=53), Pow(Integer(2), Ratio
```

```
20 nal(1, 2)))"
```

この実行結果から推測できるように、Add(a, b) は $a+b$ の和を意味し、Pow(a, b) は a^b を表現しています。

上のことからわかるように，SymPy の内部では，数（オブジェクト）はその種類に応じて適切な型（クラス）に割り振られ，独自のルールで計算が行われます。数には次のような分類があります：

数の種類	SymPy の型（クラス）	特徴と内部での保持方法
整数	Integer	完全に誤差のない正確な整数
分数	Rational	分子と分母をペアで保持する厳密な有理数
小数	Float	任意の精度を持つ浮動小数点数（近似値）
無理数・数学定数	Pow, pi など	sqrt(2) や pi（π）を数値化せず「記号」のまま保持

■異なる精度の小数演算による丸め誤差 一連の計算の途中で，不用意に異なる精度（桁数）の小数を混在させて演算を行うと，数値計算上の微小な誤差が不自然な形で表面化します。

```
1 b = Rational(1, 3).n(20)  # 10進数換算で20桁の精度を持つ小数
2 c = Rational(1, 3).n(10)  # 10進数換算で10桁の精度を持つ小数
3 print(b)
4 print(c)
5 print(b + c)
6 print(srepr(b + c))
```

実行結果の例

```
0.33333333333333333333
0.3333333333
0.66666666665454007 06
Float('0.6666666666654540070 6487', precision=70)
```

暗算でもできるように，b と c の和は 0.66666666663333333333 のはずです。出力結果が示す通り，b + c の評価結果は高い方の精度（20 桁）に合わせて出力されているように見えますが，実際には 11 桁目以降から 5454007... という意図しない数値の並びが出現し，値の正確性が失われています。

精度の低い c（10 桁）を高精度な b（20 桁・内部的には精度 70 ビットの計算空間）に合わせて拡張する際，不足しているビット列が機械的に 2 進数のゼロで埋められます。この「2 進数の世界でのゼロ埋め」という不連続な処理が，10 進数に再変換された段階でこのように顕著な丸め誤差（歪み）となって表面化することになるようです。

19.4 Sympy による代数的計算

SymPy や Mathematica, Maple といった計算機代数システム（CAS）の大きな特徴の一つが，文字式を扱うことができ，式の変形や微分や積分などの代数的計算を行うことができる点です。CAS を使いこなすために，数学で文字式や関数と呼んでいたものと，以前に解説した Python における関数（プログラム）を明確に区別することから始めます。

19.4.1 Python の関数と SymPy の数式

SymPy では Python の関数とは異なる「文字式（数式オブジェクト）」を扱いますが，まずは Python における関数の復習をします。Python における関数は，いくつかの処理をひとかたまりにして名前を付けたものでした。たとえば，次のように関数を定義しました：

```
1 def fib(n):                # 関数 fib の定義，引数 n
```

```
    a, b = 0, 1             # ここから
    for i in range(n):      #
        a, b = b, a+b       # ここまでが一連の処理
    return a                # 上の処理が終わったらaの値を返す

for i in range(1, 10):          # i=1,…,9 に対して
    print(fib(i), end=' ')      # fib(i)をプリント

print(type(fib))            # fibのデータの型を表示
```

実行結果の例

```
1 1 2 3 5 8 13 21 34
<class 'function'>
```

上のプログラムでは，関数 `fib()` が呼び出されると，そのときの引数 `n` に対して 2 行 ~4 行の処理を行い，それが終わると `a` の値を返す（`return`）ということをしています。

はじめて Python の関数を習ったときに違和感を覚えた人も多いと思います。それは，高校以前の数学で『関数』と呼ばれているものは**文字式**として定義されていて，方程式を解いたり微分したりといった操作ができるものだったからだと思います。たとえば

$$4x+3, \quad x^2, \quad \sin(x), \quad \frac{1}{x^2+1} \tag{6}$$

はどれも文字式でもあり，関数でもあります。ここでは，x は関数の変数（variable）や不定元（indeterminate）と呼ばれます。これらの関数は，x に具体的な数値を代入すれば，計算によりその関数の値が定まりますが，Python の関数のように計算手順を記述したものではありません。不定元 x のとりうる値は，実数かもしれないし，状況によっては複素数や行列かもしれません。

SymPy では，この『文字式としてのオブジェクト』を直接取り扱うことができます。ただし，Python の標準機能では `x` や `y` といった文字は定義されていないため，数式処理を行う前に，必ず `symbols()` という命令を使って「これからこの文字を数学の記号として使います」と宣言する必要があります。

```
from sympy import *

# x と y を数学用の記号（Symbol）として定義する
x, y = symbols('x y')

# 定義した文字を使って数式を作る
f = x**2
g = sin(y)

print(type(f))
```

実行結果の例

```
<class 'sympy.core.power.Pow'>
```

19.4.2 数式への値の代入（subs）

SymPy で定義した数式（関数）に対して，特定の変数に数値を代入して値を計算したい場合は，`.subs()`（substitution の略）という命令を使用します。

```
# f = x^2 の x に 5 を代入する
print(f.subs(x, 5))

```

```
# g = sin(y) の y に 1.2 を代入し，さらに .n() で小数にする
print(g.subs(y, 1.2).n())
```

実行結果の例

```
25
0.932039085967226
```

2 変数以上の文字式を扱う場合も，全く同じルールで計算できます。特定の変数だけに数値を代入したり，複数の変数に同時に数値を代入したりする場合は，.subs() の中に **{ 変数:　値 }** という形式（辞書型）で指定します。

```
# 2変数関数 h = (x + y)^2 を定義
h = (x + y)****2

# 1) y = 3 だけを代入する（x は文字のまま残る）
print(h.subs(y, 3))

# 2) x = 3, y = 6 を同時に代入して計算する
print(h.subs({x: 3, y: 6}))
```

実行結果の例

```
(x + 3)**2
81
```

19.4.3　数式への値の代入（subs と Lambda）

SymPy で定義した数式（関数）に対して，特定の変数に数値を代入して値を計算したい場合は，上のように .subs() 使用するのが確実ですが，普通の数学のように $f(2)$ や $g(3)$ とカッコをつけて直接数値を代入したい場面もあります。その場合は，SymPy の Lambda() という機能を使って，数式を「関数オブジェクト」に変換するとよいでしょう。

次のプログラムを実行してみましょう。

```
from sympy import *
x = symbols('x')

# まずは通常の数式オブジェクトとして定義
f = x**2

# 1) 従来の確実な方法（subs を使用）
print(f.subs(x, 2))

# 2) 数学と同じようにカッコで代入する方法（Lambda を使用）
g = Lambda(x, f)      # 数式 f を x の関数 g として再定義
print(g(3))           # カッコでそのまま代入して計算できる！
```

実行結果の例

```
4
9
```

19.5　SymPy に用意されている関数

SymPy では，はじめから様々な初等関数や特殊関数が定義されています。

SymPy で使える代表的な初等関数には，三角関数とその逆関数（sin, cos, tan, csc, sec, cot, asin, acos, atan），対数関数・指数関数（log, exp ※自然対数は log(x)，底が１０の常用対数は log10(x) を使

用），双曲関数とその逆関数（sinh, cosh, tanh, asinh）などがあります。

また，代表的な特殊関数であるガンマ関数 gamma(z) やゼータ関数 zeta(z)，さらには各種の直交多項式（ルジャンドル多項式など）も標準で用意されており，これら以外にも数多くの数学的な関数をそのまま扱うことができます。

試しに，いくつか特殊な値を入力して SymPy に計算させてみましょう。

```
from sympy import *

# 1）三角関数の自動簡約（特別な角は自動的に厳密解になります）
display(sin(pi / 3))

# 2）対数関数の計算（底を指定しない log は自然対数になります）
display(log(E))

# 3）ガンマ関数の厳密計算
display(gamma(Rational(1, 2)))
```

実行結果の例

```
sqrt(3)/2
1
sqrt(pi)
```

19.6　文字式の展開・因数分解・単純化

SymPy を使うと，複雑な文字式の展開や因数分解，そして式の単純化（整理）を簡単に行うことができます。

19.6.1　1. 式の展開（expand）

多項式を展開するときは expand() という命令を使用します。たとえば $(x+1)^3$ を展開してみましょう。

```
from sympy import *
x = symbols('x')

f = (x + 1)**3
expand(f)
```

実行結果の例

```
x**3 + 3*x**2 + 3*x + 1
```

19.6.2　2. 式の因数分解（factor）

多項式を因数分解するときは factor() を使用します。x^3+3x^2-2x-6 の因数分解をしてみましょう。

```
f = x**3 + 3*x**2 - 2*x - 6
factor(f)
```

実行結果の例

```
(x + 3)*(x**2 - 2)
```

※ factor() では基本的に有理数（整数係数）の範囲でしか因数分解されないため，これ以上（実数や複素数の範囲）分解して多項式の根を具体的に求める方法については，後述の solve() を使用します。

19.6.3　3. 式の単純化・整理（simplify）

複雑な数式をできる限り綺麗な形に集約したいときは `simplify()` を使用します。同類項のまとめはもちろん，三角関数の重要な公式なども自動的に適用してくれます。

```
# 例1：多項式の整理
f1 = x**2 + 2*x + 6 - 3*x
print(simplify(f1))

# 例2：三角関数の公式（sin^2(x) + cos^2(x) = 1）の適用
f2 = sin(x)**2 + cos(x)**2
print(simplify(f2))
```

実行結果の例

```
x**2 - x + 6
1
```

19.6.4　$x^{105}-1$ の因数分解と計算機による発見

コンピュータを用いて代数計算を行うことで，手計算だけでは気づくことが難しい，数学的な意外な事実に直面することがあります。その一例として，多項式 x^n-1 を整数係数の範囲で因数分解する（円分多項式への分解）問題を考えてみましょう。

まずは $x^{20}-1$ を因数分解させてみます。

```
print(factor(x**20 - 1))
```

実行結果の例

```
(x - 1)*(x + 1)*(x**2 + 1)*(x**4 - x**3 + x**2 - x + 1)*(x**4 + x**3 + x**2 + x + 1)*(x**8 - x**6 + x**4 - x
```

数学の綺麗に出力された形に整理すると，以下のようになります。

$$(x-1)(x+1)(x^2+1)(x^4-x^3+x^2-x+1)(x^4+x^3+x^2+x+1)(x^8-x^6+x^4-x^2+1)$$

このように，x^n-1 を因数分解したとき，現れる多項式の係数はいつも ±1（または 0）だけになるように思えます。はたして本当にそうでしょうか？

この予想に対する反例を，SymPy を使って $x^{105}-1$ を因数分解することで確かめてみましょう。

```
display(factor(x**105 - 1))
```

この実行結果を整理すると，驚くべきことに以下のようになります。

$$\begin{aligned}
&\left(x^{48}+x^{47}+x^{46}-x^{43}-x^{42}-\mathbf{2}x^{41}-x^{40}-x^{39}+x^{36}+x^{35}+x^{34}+x^{33}+x^{32}+x^{31}-x^{28}-x^{26}\right.\\
&\quad\left.-x^{24}-x^{22}-x^{20}+x^{17}+x^{16}+x^{15}+x^{14}+x^{13}+x^{12}-x^9-x^8-\mathbf{2}x^7-x^6-x^5+x^2+x+1\right)\\
&\times\left(x^{24}-x^{23}+x^{19}-x^{18}+x^{17}-x^{16}+x^{14}-x^{13}+x^{12}-x^{11}+x^{10}-x^8+x^7-x^6+x^5-x+1\right)\\
&\times\left(x^{12}-x^{11}+x^9-x^8+x^6-x^4+x^3-x+1\right)\left(x^8-x^7+x^5-x^4+x^3-x+1\right)\\
&\times\left(x^6+x^5+x^4+x^3+x^2+x+1\right)\left(x^4+x^3+x^2+x+1\right)\left(x^2+x+1\right)\left(x-1\right)
\end{aligned}$$

赤色で示したように，次数が 105 まで大きくなると，因数分解した結果の係数の中に ±1 **以外の数である**「**2**」が初めて出現します。これは，手計算では容易に到達できない，計算機代数（CAS）を使うからこそ直感的に体感できる数学の面白い一例です。

19.6.5　有理式の部分分数展開

有理式を

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

のように分解することを部分分数展開（あるいは部分分数分解）といいます。積分計算などの前処理として非常によく使われる変形です。

SymPy で部分分数展開を行うには，apart() という命令を使用します。

```
1 from sympy import *
2 x = symbols('x')
3
4 f = x**2 / (x + 1)**2
5 print(apart(f))
```

実行結果の例

```
1 - 2/(x + 1) + 1/(x + 1)**2
```

部分分数展開された式を元の通分された状態に戻したいときは，前節で登場した factor() を使用して因数分解します。

```
1 g = 1 - 2 / (x + 1) + 1 / (x + 1)**2
2 print(factor(g))
```

実行結果の例

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

19.6.6　連分数への展開

[*13]連分数は実数を整数の分数の列で表す方法で，非常に美しい性質を持ちます。実数 x に対して x を超えない最大の整数を $a_0 = \lfloor x \rfloor$ とし，$x_1 = 1/(x - a_0)$ とおきます。このとき

$$x = a_0 + \frac{1}{x_1}$$

です。同様に $a_1 = \lfloor x_1 \rfloor$，$x_2 = 1/(x_1 - a_1)$ とすると

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{x_2}}$$

となります。これを繰り返すことで，次の表現が得られます。

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cdots}}}}$$

これを x の**連分数表示**（連分数展開）といい，次のように簡略化して記述します。

$$x = [a_0; a_1, a_2, a_3, a_4, \cdots]$$

[*13] この節では必ず from sympy import *を読み込んでください。

x が有理数であれば，この連分数表示は有限の長さで終了することが知られています。たとえば，345/29 は次のように順次展開できます。

$$\begin{aligned}\frac{345}{29} =& 11 + \frac{26}{29} = 11 + \frac{1}{\frac{29}{26}} = 11 + \frac{1}{1 + \frac{3}{26}} = 11 + \frac{1}{1 + \frac{1}{\frac{26}{3}}} \\ =& 11 + \frac{1}{1 + \frac{1}{8 + \frac{2}{3}}} = 11 + \frac{1}{1 + \frac{1}{8 + \frac{1}{1 + \frac{1}{2}}}}\end{aligned}$$

したがって，$345/29 = [11; 1, 8, 1, 2]$ という有限のリストになります。

連分数表示の強力な特徴は，無理数や超越数であっても，非常に美しい規則的なパターンで表現できる点にあります。SymPy で数の連分数展開を行うには，`continued_fraction_iterator()` を使用します。この関数は，次の値を順番に生成する「イテレータ」と呼ばれるオブジェクト[*14]を返します。

次は有理数 345/29 の連分数展開を行ったものです。

```
1 aa = continued_fraction_iterator(Rational(345, 29))
2 list(aa)
```

実行結果の例

```
[11, 1, 8, 1, 2]
```

無理数の場合は展開が無限に続くため、`list()` で一括処理しようとするとプログラムが停止しなくなります。

```
1 sq3 = continued_fraction_iterator(sqrt(3))
2 [next(sq3) for i in range(15)]
```

実行結果の例

```
[1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

ここで，`next()` はイテレーターから要素を順番に 1 つずつ取り出す関数です。リスト内包表記を用いてこの操作を 15 回繰り返すことで，必要な項数だけを安全に切り出しています。出力結果から $\sqrt{3} = [1; 1, 2, 1, 2, \cdots]$ と「1, 2」の周期 2 の循環という美しい規則性を観察できます。

さらに，超越数であっても驚くべき規則的な展開を示す場合があります。自然対数の底（ネイピア数）e の連分数表示から，最初の 25 項を抽出してみましょう。

```
1 ee = continued_fraction_iterator(E)
2 [next(ee) for i in range(25)]
```

実行結果の例

```
[2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 14, 1, 1, 16, 1]
```

出力された数列の並びを観察すると，**「1, 1, 偶数」** というパターンが規則正しく並んでいる様子がわかります。

19.7　極限の計算（limit）

SymPy では，無限大 ∞ を `oo`（小文字のオー `o` を 2 つ並べたもの）で表します。関数の極限 $\lim_{x \to a} f(x)$ を求めるには，`limit()` という命令を使用します。

[*14] `range` や `zip` などの仲間

左右の極限を明示的に指定して計算させることも可能です。

```
from sympy import *
x = symbols('x')

# 1) ネイピア数の定義： lim_{x -> oo} (1 + 1/x)^x
f1 = (1 + 1 / x)**x
print(limit(f1, x, oo))

# 2) 1/x の右極限と左極限
print(limit(1 / x, x, 0, dir='+'))  # 右極限（x -> 0 + 0）
print(limit(1 / x, x, 0, dir='-'))  # 左極限（x -> 0 - 0）
```

実行結果の例

```
E
oo
-oo
```

なお，Jupyter Notebook 上で `init_printing()` を有効にしている場合，これらの結果は ∞ や $-\infty$ などの美しい数式記号で直接レンダリングされます。

19.7.1　文字への「仮定（条件）」の付与

次に，$\lim_{x\to 0} x^a$ という極限を考える場合，この結果がどうなるかは文字 a が「正の数か，負の数か」といった前提条件によって変化します。

SymPy では，`symbols()` で文字を定義する際に，あらかじめ `positive=True` などのオプションを付与することで，「この文字は正の実数として取り扱う」という仮定（条件）をコンピュータに指示することができます。

```
# a を「正の実数（positive）」として仮定して定義する
a = symbols('a', positive=True)

# a>0 の前提のもとで，x^a の右極限を計算する
print(limit(x**a, x, 0, dir='+'))
```

実行結果の例

```
0
```

このように，あらかじめ文字の性質（`positive=True`，`negative=True`，あるいは偶数を表す `even=True` など）を指定しておくことで，文字を含んだ複雑な極限や代数計算の不確定要素を解消し，正しく答えを導き出すことができます。

19.8　代数的な和の計算（summation）

Python には標準でリストの合計を出す `sum()` 関数が用意されていますが，SymPy で数学の和の記号（$\sum$）として有限和や無限級数を厳密に計算させたい場合は，`summation()` という命令を使用します。

$\sum_{k=a}^{b} f(k)$ の計算構文

```
summation(数式，(ダミー変数，開始値，終了値))
```

まずは，具体的な有限和（$1+2+\cdots+50$）を計算させてみましょう。

```
k, m = symbols('k m')

# 1から50までの和を求める
print(summation(k, (k, 1, 50)))
```

実行結果の例

```
1275
```

summation() の強力な点は，単に数値の和を出すだけでなく，文字 m を含んだ一般的な公式の導出も命令一つで行える点です。

$$\sum_{k=1}^{m} k^2 = \frac{m(m+1)(2m+1)}{6}$$

の公式を，SymPy に導出させて因数分解させてみましょう。

```
# 1から m までの k^2 の和を計算し，ans に代入
ans = summation(k**2, (k, 1, m))
print(ans)

# 導出された式を因数分解して綺麗に整理する
print(factor(ans))
```

実行結果の例

```
m**3/3 + m**2/2 + m/6
m*(m + 1)*(2*m + 1)/6
```

さらに，終了値に oo を指定すれば，数学の歴史に名を残す数々の有名な無限級数の厳密解（収束値）を一瞬で求めることも可能です。

```
# 1) ネイピア数の級数表示
print(summation(1 / factorial(k), (k, 0, oo)))

# 2) バーゼル問題（平方数の逆数和）
print(summation(1 / k**2, (k, 1, oo)))

# 3) 対数が現れる無限級数
print(summation((2**(-k)) / (k * (k + 1)), (k, 1, oo)))
```

実行結果の例

```
E
pi**2/6
1 - log(2)
```

19.9　微分・積分の厳密な計算

SymPy は，「微分」「積分」も記号のまま（厳密に）計算することができます。

19.9.1　微分（diff）

関数を微分するには，diff() を使用します。

```
var('x')
f = sin(x) * cos(x) # 関数の定義
print(diff(f, x))  # 1階微分

print(diff(sin(x), x, 2))  # 2階微分（変数の後ろに階数を指定）
```

実行結果の例

```
cos(x)
-sin(x)
```

19.9.2　積分（integrate）

関数を積分するには，integrate() を使用します。これ 1 つで「不定積分」と「定積分」の両方を計算し分けることができます。

```
1 print(integrate(log(x), x))              # 不定積分，積分定数 C は省略されます
2 print(integrate(sin(x), (x, 0, pi)))     # 定積分
3 print(integrate(exp(-x**2), (x,0,oo))) # 広義積分：区間[0,∞)で exp(-x^2) を積分
```

実行結果の例

```
x*log(x) - x
2
sqrt(pi)/2
```

積分は常に計算できるとは限りません。

```
1 integrate(sin(x)/(1+x**2),(x,0,1))
```

実行結果の例

```
Integral(sin(x)/(x**2 + 1), (x, 0, 1))
```

※ integral(...) は積分をしなさいという命令ですが，頑張って計算しようとしたけどできなかったので，積分の数式オブジェクト Integral(...) が返されました。これに対して n() を行えば内部で数値積分に切り替わって小数の近似解が出ます。

```
1 ans = integrate(sin(x)/(1+x**2),(x,0,1))
2 print(ans.n())
```

実行結果の例

```
0.321793544741077
```

19.10　テイラー展開（series）

関数を特定の点のまわりで多項式近似するテイラー展開は series() という機能を使い計算することができます。

```
1 # exp(x) を x=0 のまわりで 6次（O(x^6)）の項まで展開する
2 print(series(exp(x), x, 0, 6))
```

実行結果の例

```
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
```

※ 最後の O(x**6) は，6 次以上の高次の項（ランダウの記号）を意味します。もしグラフのプロットなどでこの剰余項（O）が邪魔な場合は，末尾に .removeO() を付けることで，純粋な多項式オブジェクトに変換することができます。

```
1 f_approx = series(exp(x), x, 0, 6).removeO()    # 剰余項を除去
2 print(f_approx)
```

実行結果の例

```
x**5/120 + x**4/24 + x**3/6 + x**2/2 + x + 1
```

19.11 練習問題

Jupyter Notebook で新しくノートブックを開き，ファイル名を **「sympy01.ipynb」** とした上で，以下の 3 つの問を解くプログラムを作成してください。

1. **【代数計算と代入】**
   次の多項式 f について，以下の問いに答えなさい。

$$f = x^3 - 6x^2 + 11x - 6$$

   (a) f を SymPy を用いて因数分解しなさい。
   (b) `Lambda` を用いて f を関数 g として定義し直し， カッコ書きの代入を用いて $g(4)$ の値を求めなさい。
2. **【無限級数の厳密和】**
   次の無限級数の和（収束値）を，SymPy を用いて厳密に求めなさい。

$$\sum_{k=1}^{\infty} \frac{1}{k(k+1)(k+2)}$$

課題提出時には，すべてのセルの実行結果（ログ）を残した状態の `.ipynb` ファイルを提出用データとします。

20 グラフの描画とデータの可視化

データの可視化は，コンピューターを用いた数値計算や数式処理において最も重要な活用例の一つです。複雑な数式や大量のデータも，グラフとして視覚的に捉えることで一目で理解できるようになります。

本章では，数式処理ライブラリである「SymPy」と，Python 標準の可視化ライブラリである「Matplotlib」の 2 つを紹介します。前者は x や y といった記号を含んだ数式のグラフ作成に用い，後者は数値データを可視化するツールです。

まずは前半として，特別な準備なしに数式をそのままグラフにできる `SymPy` を使った関数のプロットから見ていきましょう。

20.1 関数のグラフを描画（SymPy で plot）

1 変数関数 $f(x)$ のグラフをプロットするには，SymPy の `plot()` という関数を使用します。

plot —— 1 変数関数のグラフプロット

```
var('x')
plot(f(x), (x, a, b))
```

- 変数 `x` を使う宣言。
- `f(x)`：描画したい SymPy の数式（または関数のオブジェクト）。
- `(x, a, b)`：横軸の変数名 `x` と，描画する区間 $[a, b]$ を指定するタプル。

試しに，$\sin(x)$ を区間 $[-6, 6]$ でプロットしてみましょう。

```
from sympy import *
var('x')
plot(sin(x), (x, -5, 5))  # sin(x) を -5 から 5 の範囲で描画
```

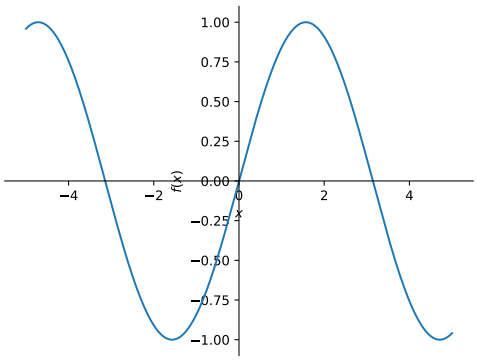


図 13　出力結果：sin(x) のグラフ

残念なことに，このプログラムではグラフのラベル x, f(x) がおかしな位置にあります。次のようにオプションを指定することでこれを解決します：

plot のオプション

plot ではオプションを指定することで，グラフの色を変更したり，ラベルを正しい位置に書くことができます。

```
plot(f(x), (x, a, b), color='色', title='タイトル',
     xlabel='横軸名', ylabel='縦軸名')
```

- color：'red', 'blue' などのほか，16 進数カラーコードも指定可能。
- xlabel, ylabel：明示的に軸の名前を指定することで，中央の不自然なラベルが消え，グラフの枠外（下部と左側）に正しく配置し直されます。

ラベルが不要な場合は

```
plot(sin(x**2), (x, 0, 5),  xlabel='', ylabel='', )
```

とします。ラベルを書きたい場合は例えば次のようにします。

```
plot(sin(x**2), (x, 0, 5), color='red',
   xlabel='x', ylabel='y', title=r'$\sin(x^2)$')
```

つぎに，原点で発散している関数 $1/x$ のグラフを描いてみましょう。

```
# ラベルを非表示にして 1/x を -2 から 2 の範囲で描画
plot(1/x, (x, -2, 2), xlabel='', ylabel='')
```

実際に実行してみると，グラフの縦軸の数値が極端に大きくなり，グラフが潰れてしまいます。これは $1/x$ の値が原点（$x = 0$）の前後で正負の無限大に激しく発散しているためです。

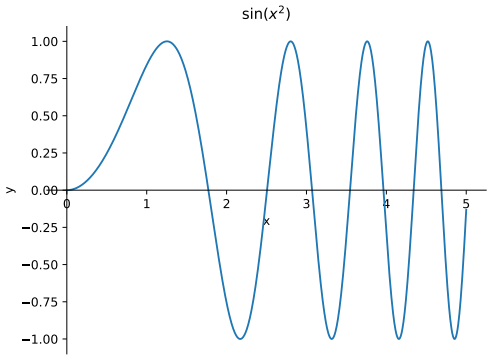


図 14　出力結果：オプション付きプロット

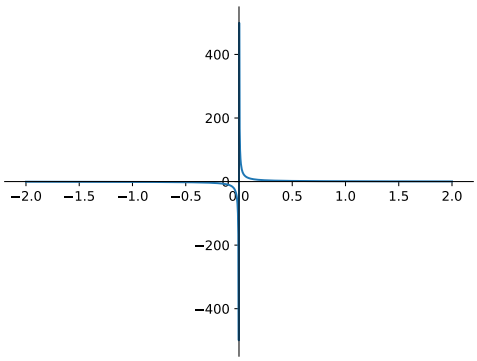


図 15　出力結果：$1/x$ のグラフ（値域未指定のため潰れた状態）

このような関数を正しく描画するには，グラフの縦軸の表示範囲（**値域**）を指定する必要があります。

関数 $f(x)$ のグラフを値域を指定して描画

```
plot(f(x), (x, a, b), ylim=(最小値, 最大値))
```

- `ylim=(最小値, 最大値)` が値域を制限する部分です。

関数 $1/x$ の値域を $[-3, 4]$ に制限して，再度プロットしてみましょう。

```
# 値域を制限してプロット
plot(1/x, (x, -2, 2), ylim=(-3, 4), xlabel='', ylabel='')
```

値域を制限したことで，原点付近の発散に邪魔されることなく，きれいな反比例の曲線が確認できるようになりました。

20.1.1　複数の関数の重ね合わせ

複数の関数のグラフを一度にプロットする方法には，大きく分けて 2 種類あります。

まずは，もっとも単純に `plot()` 関数の中に重ねたい関数を並べて一度に描画する方法です。

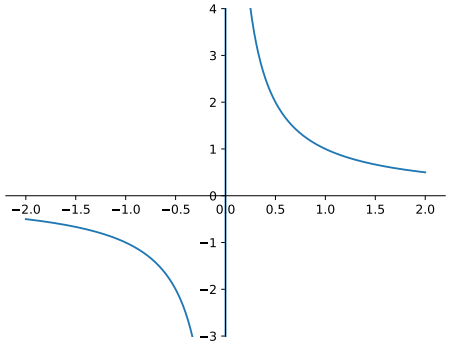


図 16　出力結果：$1/x$ を値域 $[-3, 4]$ に制限したグラフ

```
1 # sin(x) と cos(x) を同時にプロット
2 plot(sin(x), cos(x), (x, -5, 5), xlabel='', ylabel='')
```

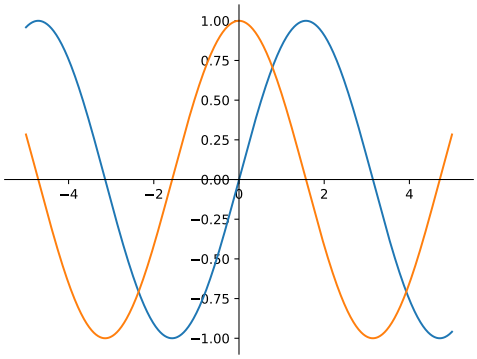


図 17　出力結果：$\sin(x)$ と $\cos(x)$ の重ね合わせ

もう一つの方法は，別々に作成したグラフ（プロットオブジェクト）を変数に代入し，後からそれらを合体させる方法です。プログラムの途中で動的にグラフを追加したい場合に非常に便利です。

合体させる場合は，1 つ目のグラフに対して `.append()` という機能を使い，2 つ目のグラフの要素（最初を表す `[0]` 番目）を追加します。

```
1 # 別々に作ったグラフを後から合体させる
2 # 画面に自動表示されないように show=False にしておく
3 p1 = plot(sin(x), (x, -5, 5), color='blue', xlabel='', ylabel='', show=False)
4 p2 = plot(cos(x), (x, -5, 5), color='red', show=False)
5
6 # p1 に p2 のグラフ要素を追加して表示
7 p1.append(p2[0])
8 p1.show()
```

※ `p1.append(p2[0])` とすることで，`p2` の中身が `p1` に上書き（追加）され，最終的に 1 つの画面に重なって表示されます。

20.1.2　陰関数と不等式の領域を描画

$x^2+y^2=1$ のような方程式で表される曲線（陰関数）や，$x^2+y^2>1$ などの不等式が表す平面上の領域を描画するには，SymPy の `plot_implicit()` を使用します。

SymPy では，等号（＝）は `Eq(左辺，右辺)` という形式で記述します。つまり，方程式 $x^2+y^2=4$ は `Eq(x**2+y**2，4)` と表します。

```
from sympy import *
var('x y')

# 円の方程式 x^2 + y^2 = 4 の描画
plot_implicit(Eq(x**2 + y**2, 4), (x, -3, 3), (y, -3, 3),
              aspect_ratio=(1, 1), xlabel='', ylabel='')
```

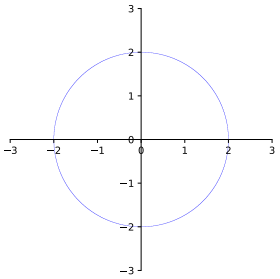


図 18　出力結果：円 $x^2+y^2=4$ のグラフ

※ デフォルトの表示は横長になるため，アスペクト比（縦横比）を指定しないと円が歪んで楕円が表示されます。`aspect_ratio=(1, 1)` を指定してアスペクト比（縦横比）を固定してください。

方程式の代わりに不等式をそのまま渡すことで，その条件を満たす平面上の領域を自動的に塗りつぶして描画することも可能です。

```
# 不等式が表す領域の描画
plot_implicit(x**2 + y**2 < 1, (x, -2, 2), (y, -2, 2),
              aspect_ratio=(1, 1), xlabel='', ylabel='')
```

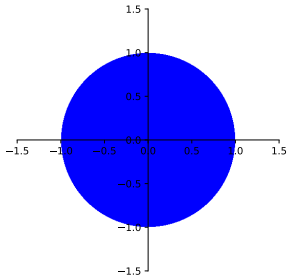


図 19　出力結果：不等式 $x^2+y^2<1$ の領域

20.1.3　パラメーター表示の曲線を描画

$x = f(t)$, $y = g(t)$ のように，パラメーター t を用いて表される曲線をプロットするには，plot_parametric() を使用します。

```
from sympy import *
var('t')
# パラメーター表示による円の描画
# plot_parametric( (xの式 , yの式), (変数 , 開始 , 終了) ) の形式で指定します
plot_parametric((cos(t), sin(2*t)), (t, 0, 2*pi), xlabel='', ylabel='')
```

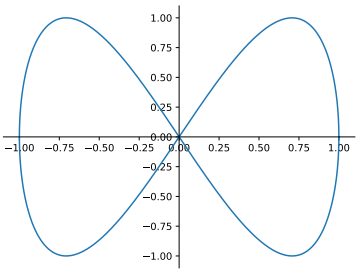


図 20　出力結果：パラメーター表示による円

20.1.4　3 次元グラフィックスの描画

2 変数関数 $z = f(x, y)$ のグラフ（3 次元空間上の曲面）を描画するには，plot3d() を使用します。

plot3d は from sympy import * では自動的に読み込まれないため，from sympy.plotting import plot3d と明示的に指定してインポートする必要があります。

plot3d —— 2 変数関数の 3 次元プロット

```
from sympy.plotting import plot3d
var('x y')
plot3d(f(x, y), (x, a, b), (y, c, d))
```

- 描画したい 2 変数関数に続けて，x の範囲 $[a, b]$ と y の範囲 $[c, d]$ をそれぞれ指定する。

例として，2 変数関数 $z = x^2 - y^2$ のグラフを，x, y ともに $[-2, 2]$ の範囲で描画してみましょう。この関数の形状は馬の鞍の形に似ていることから，鞍点（あんてん）と呼ばれています。

```
from sympy import *
from sympy.plotting import plot3d # 3次元プロット用に明示的にインポート
var('x y')

# 2変数関数の3次元プロット
plot3d(x**2 - y**2, (x, -2, 2), (y, -2, 2), xlabel='x', ylabel='y', zlabel='z')
```

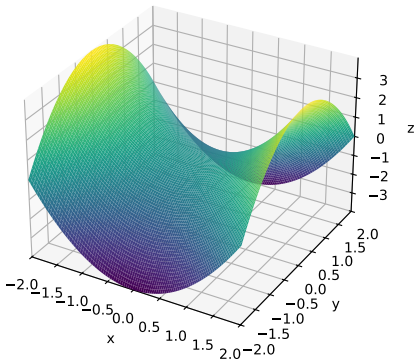


図 21　出力結果：$z = x^2 - y^2$ の 3 次元グラフ

20.2　Matplotlib によるデータの可視化

ここでは Python の標準的なグラフ描画ライブラリである `Matplotlib`（マットプロットリブ）を用いたデータの可視化を説明します。

数学の数値シミュレーション（例えば，微分方程式を数値的に解いた軌跡や計算した値の収束など）では，得られる結果は数式ではなく数値のデータ（リスト）になります。`Matplotlib` は，このような「数値データの集まり」を可視化するツールです。

`Matplotlib` を使用する際は，慣例として次のように `plt` という短い名前をつけてインポートします。

```
1 import matplotlib.pyplot as plt
```

20.2.1　リスト（データの集まり）をプロットする

まずは最も単純な例として，手動で作った数値のリストをプロットしてみましょう。データを 1 次元の線としてプロットするには，`plt.plot()` を使用します。

```
1  import matplotlib.pyplot as plt
2
3  # プロットしたい数値のリスト（データ）
4  y_data = [1, 10, 7, 20, 25, 50]
5
6  # データをプロット
7  plt.plot(y_data)
8
9  # グラフを画面に表示する命令
10 plt.show()
```

※ `plt.plot(y_data)` のようにリストを 1 つだけ渡した場合，指定したリストの値が自動的に「縦軸（y 軸）」の値として扱われます。横軸（x 軸）には，データの順番である $0, 1, 2, 3, 4$ が自動的に割り振られます。

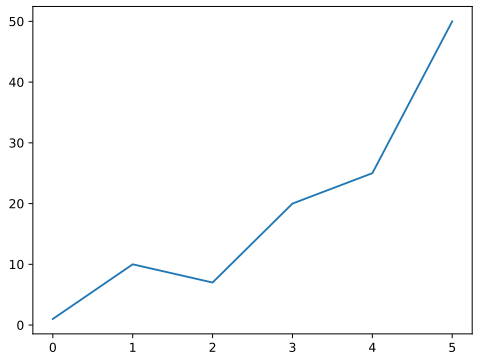


図 22　出力結果：リストデータのプロット

20.2.2　x 軸と y 軸の両方を指定してプロットする

先ほどは y の値だけを渡したため，横軸が自動的に $0, 1, 2, \ldots$ となってしまいました。数学の関数のように，任意の x 座標に対応する y 座標のグラフを描きたい場合は，`plt.plot()` に（**x 軸のリスト，y 軸のリスト**）の順番で 2 つのデータを渡します。

例として，いくつかの点の座標 (x, y) を直接指定して，それらを結ぶ折れ線グラフを描いてみましょう。

```
1 # x軸のデータと，それに対応するy軸のデータ
2 x_data = [0, 1, 2, 4, 5]
3 y_data = [1, 10, 7, 25, 50]
4
5 # x軸、y軸の順番でデータを渡してプロット
6 plt.plot(x_data, y_data)
7 plt.show()
```

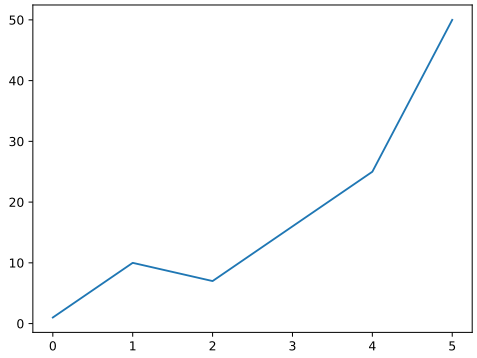


図 23　出力結果：x 軸と y 軸を指定したプロット

※ 実行結果を見ると，今度は横軸が自動連番ではなく，指定した `x_data` （$0, 1, 2, 4, 5$）の正しい位置にプロットされていることが分かります。このように，Matplotlib の基本は「**同じ要素数を持つ 2 つのリストをペアにして渡す**」ことです。

20.2.3 散布図

実験の測定データや統計データをプロットする場合，点と点の間を直線で結ぶのではなく，データが存在する座標に点（マーカー）を打ちたいことがあります。このようなグラフを**散布図（さんぷず）**と呼び，Matplotlib では `plt.scatter()` を使用します。

```
# 実験データなどのシミュレーション（5つの点の座標）
x_data = [0, 1, 2, 4, 5]
y_data = [0, 1, 4, 16, 25]

# 散布図（点）としてプロット
plt.scatter(x_data, y_data)
plt.show()
```

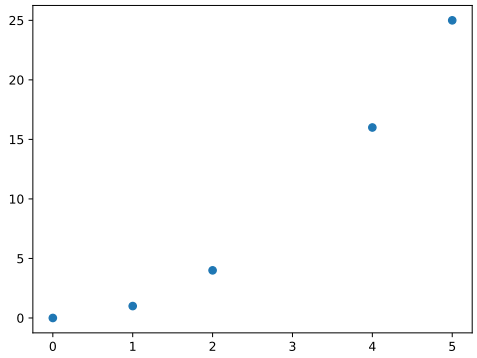


図 24 出力結果：散布図によるデータのプロット

20.2.4 数学の関数の値をデータにして描画する

数学の関数の値をデータにしてから，それをプロットしていましょう。

コンピュータで曲線を描くには，区間 $[a, b]$ を細かく等分した x 座標のリストと，それに対応する y 座標のリストをプログラムで自動生成します。ここでは，区間の等分定義公式：

$$x_j = a + \frac{b-a}{n} \cdot j \quad (j = 0, 1, \ldots, n)$$

をそのまま「リスト内包表記」を使って愚直に使います。

```
import matplotlib.pyplot as plt
from math import *

# パラメータの設定：区間 [a, b] を n等分する
a = -2*pi
b = 2*pi
n = 200

# 区間 [a, b] を n等分した (n+1) 個の x のリストを数式通りに生成
x_data = [a + (b-a)*j/n for j in range(n+1)]

# 各 x に対して sin(x)+0.2sin(7x) を計算した y のリストを生成
```

```
y_data = [sin(x)+0.2*sin(7*x) for x in x_data]

# プロットして表示
plt.plot(x_data, y_data)
plt.show()
```

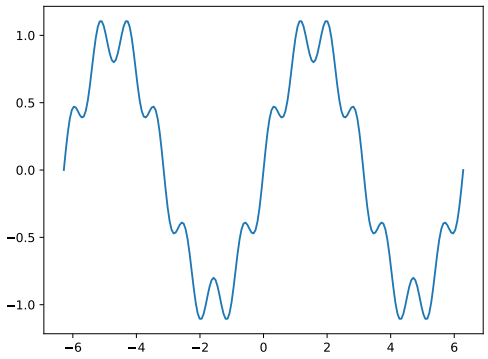


図 25 出力結果：数式から生成した $\sin(x) + 0.2\sin(7x)$ の折れ線グラフ

※ n=100 ぐらいにするとなめらかなはずの曲線がギザギザになります。これはもちろんデータの数が足りない為です。

20.2.5 グラフの装飾（タイトル・軸ラベル・グリッド線の追加）

描画したグラフが何のデータを表しているのかを明示するために，タイトルや軸のラベル，そして数値を読み取りやすくするための補助線（グリッド線）を追加してみましょう。

これらを設定するには，plt.plot() で線を指示したあと，plt.show() で画面に表示する「前」に，それぞれ以下の関数を呼び出します。

```
# （前項のデータx_dataとy_dataは生成されているとして）
plt.plot(x_data, y_data)

# グラフのタイトルを設定
plt.title('Waveform Graph')

# x軸とy軸のラベルを設定
plt.xlabel('x')
plt.ylabel('y')

# グリッド線（補助線）を表示
plt.grid(True)

plt.show()
```

20.2.6 複数のグラフを重ねて描画する（凡例の追加）

2 つの異なる関数を同じグラフ上に同時に描き，それらを比較したいことがあります。Matplotlib では，plt.show() を呼び出す前に plt.plot() を複数回呼び出すだけで，自動的に 1 枚のグラフの中に線が重ねて描画されます。

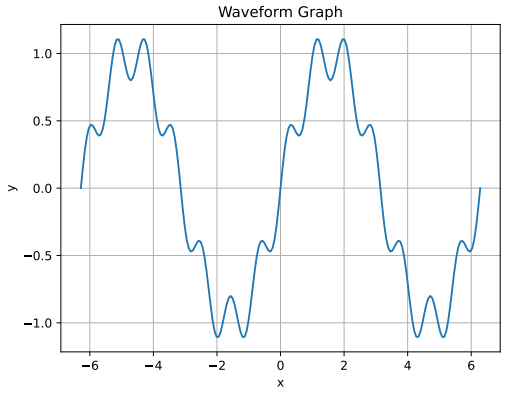


図 26　出力結果：タイトルと軸ラベルを追加したグラフ

その際，どちらの線がどの関数を表しているかを区別するために，**凡例（はんれい：レジェンド）**を追加するのが一般的です。

```
# （前項のデータ x_data の生成に続けて）
y_data1 = [sin(x) + 0.2 * sin(7 * x) for x in x_data]
y_data2 = [sin(x) for x in x_data]

# 1本目のプロット（labelでこの線の名前を指定しておく）
plt.plot(x_data, y_data1, label='Combined Wave')

# 2本目のプロット（続けて呼び出すと重ね書きされる）
plt.plot(x_data, y_data2, label='sin(x)')

# 凡例（ラベルの目次）をグラフ内に表示する命令
plt.legend()

# その他、軸ラベルなどの装飾
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)

plt.show()
```

※ Matplotlib の親切な機能として，2 本の線はユーザーが指定しなくても**自動的に異なる色（デフォルトでは青とオレンジ）**で塗り分けられます。

20.3　Matplotlib の小技（Tips）

20.3.1　アスペクト比の固定

数学のグラフ、特に円や正方形、あるいは直交する直線などを描くときに注意すべき点があります。Matplotlib は初期設定では「画面のウィンドウサイズ（長方形）」に合わせてグラフを自動的に引き伸ばしてしまうため、単位円を描いたつもりが横長の「楕円」に見えてしまうという幾何学的な歪みが生じます。

これを防ぎ、縦横のスケールを 1 : 1 にするには、プロットする際に以下の命令を 1 行追加します。

```
# 縦横の比率（アスペクト比）を1:1にする
plt.axis('equal')
```

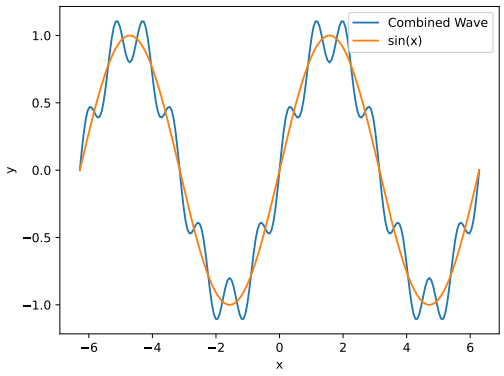


図 27　出力結果：2 本の波形を重ね合わせ，凡例を表示したグラフ

20.3.2　(x, y) のペアのリストをプロットする

数学の幾何学や線形代数では，座標データや 2 次元ベクトルを `[(1,2), (4,1), (3,4), (4,2)]` のように「x と y の組（タプル）が並んだリスト」として管理したくなります。しかし，Matplotlib は前述の通り「x 座標だけのリスト」と「y 座標だけのリスト」を別々の引数として受け取る仕様になっています。そのため，この組のリストをそのまま渡しても正しくプロットされません。

このようなデータを扱う場合は，以下のように「リスト内包表記」を用いて，各要素から x 成分と y 成分をそれぞれ抽出した別々のリストに分解してから `plt` に渡します。

```
# (x, y) のペアが並んだデータ
points = [(0, 0), (1, 1), (-1, 2), (-3, -1), (2, -4)]

# 各ペア（p）の0番目（x）と1番目（y）をそれぞれ取り出してリストにする
x_data = [p[0] for p in points]  # [0,1,-1,3,2]というリストになる
y_data = [p[1] for p in points]

# 分解したリストを渡してプロットする
plt.scatter(x_data, y_data)
plt.show()
```

※上では 5，6 行目でリスト内包表記で x 座標のリスト `x_data` と y 座標のリスト `y_data` を作っています[*15]。

20.3.3　グラフを保存する

表示されたグラフを右クリックやウィンドウの保存ボタンから保存すると png 画像として保存することができますが解像度が低く，レポート（LaTeX）に貼り付けたときに線や文字がぼやけてしまうことがあります。

大学のレポートや論文に掲載する図表は，どれだけ拡大しても線がぼやけない「ベクター形式（`.pdf` や `.eps`）」で保存するがよいです。

■1. Matplotlib の場合　`plt.show()` で画面に表示する「前」に，`plt.savefig()` を呼び出します。

```
# （グラフの描画・装飾処理のあとに）
plt.grid(True)  # 必要であれば

```

[*15] 実はこれは一行で `x_data, y_data = zip(*points)` と記述することができます。これは行列の転置をとるのような作業です。

```
# 画面に表示する前に、PDF形式で直接保存する
plt.savefig('my_graph.pdf')

plt.show()  # 表示したければ
```

※ 拡張子を .pdf に指定するだけで，Matplotlib が自動的に pdf ファイルとして保存してくれます。画像形式（.png）が必要な場合は，plt.savefig('my_graph.png', dpi=300) のように解像度（dpi）を明示的に指定して高画質化するとよいでしょう。

■2. SymPy の場合　SymPy の plot() 関数で描いたグラフを保存したい場合は，以下のように描画オブジェクトを一度変数に受け取ってから save() メソッドを呼び出します。

```
from sympy import *
x = symbols('x')
# グラフオブジェクトを作成
p = plot(sin(x), (x, -3, 3))

# PDFで保存する
p.save('sympy_graph.pdf')
```

※ Matplotlib とは異なり，SymPy の .save() に直接 dpi=300 のような画質の引数を渡すことはできません。そのため，SymPy で印刷用の高解像度な画像が欲しい場合は，最初から .pdf 形式で保存するのがよいでしょう。

21　SymPy による線形代数と数値計算

ここでは SymPy を使って行列を扱う方法を解説します。SymPy の強みは，手計算と同様に平方根や文字式などを含んだまま厳密に行列を操作できる点です。一般に 5 次以上の行列では一般には厳密に固有値を計算することはできませんが，スムーズに数値近似へと切り替えることもできます。

21.1　行列の定義と基本演算

SymPy で行列を扱うには，Matrix というオブジェクトを使用します。行列の行（横の並び）をリストとして表現し，それらをさらに全体のリストで囲むことで定義します。

```
from sympy import *
var('x')

# 2x2 行列の定義
A = Matrix([[1, 2],
            [3, 4]])

# 文字や分数を含む行列の定義
B = Matrix([[x, S(1)/2],
            [1,  x**2]])

print(A + A)  # 行列の和
print(B * A)  # 行列の積（積は * で表す）
print(A**3)   # 行列のべき乗（Aの3乗）
```

実行結果の例

```
Matrix([[2, 4], [6, 8]])
Matrix([[x + 3/2, 2*x + 2], [3*x**2 + 1, 4*x**2 + 2]])
Matrix([[37, 54], [81, 118]])
```

21.2　ベクトルの表現

SymPy ではベクトルも Matrix を用いて 1 列だけの行列（列ベクトル）または 1 行だけの行列（行ベクトル」として表現します。

数学の線形代数では通常，ベクトルといえば「列ベクトル」を指すことが多いため，単なるリストを Matrix に渡すとそれは列ベクトルになります。たとえば，列ベクトル $\binom{1}{2}$ は Matrix([1,2]) で表します。また，これは入れ子になったリストを直接使って Matrix([[1],[2]]) と表すこともできます。

```
v = Matrix([1, 2]) # 列ベクトル x=[1, 2]^T の定義

A = Matrix([[1, 2],
            [3, 4]])
A * v  # 行列とベクトルの積（Ax の計算）
```

実行結果の例

```
Matrix([[5],[11]])  # 列ベクトル (5, 11)^T
```

上の結果はタイプセットされている場合は $\begin{bmatrix}5\\11\end{bmatrix}$ となります。

行ベクトル $u = (1, 2)$ は次のように表します。

```
u = Matrix([[1,2]])
A = Matrix([[1, 2],
            [3, 4]])
u*A   # 右から行列をかける
```

実行結果の例

```
[7  10]   # 行ベクトル
```

21.3　行列式・逆行列・固有値の厳密計算

手計算では骨の折れる行列式や逆行列，固有値の計算も，メソッド（命令）を呼び出すだけでできます。上に続いて下を実行してみましょう。

```
# 行列式（determinant）の計算
print(A.det())  # 出力: -2

# 逆行列（Inverse）の計算
print(A.inv())  # 分数を含んだ厳密な逆行列が返る

# 3) 固有値（Eigenvalue）の計算
M = Matrix([[0, 1],
            [2, 3]])
print(M.eigenvals())
# 出力: {3/2 - sqrt(17)/2: 1, 3/2 + sqrt(17)/2: 1}
```

eigenvals() は固有値: 多重度の辞書形式を返します。固有値だけを取り出したければ次のようにします。

```
# 先ほどの行列 M の固有値（辞書のキー）を取り出す
egvals = list(M.eigenvals().keys())
```

```
print(egvals) # 固有値のリスト[3/2 - sqrt(17)/2, 3/2 + sqrt(17)/2]

print(egvals[0])           # 厳密解: 3/2 - sqrt(17)/2
print(egvals[1])           # 厳密解: 3/2 + sqrt(17)/2
print(egvals[0].evalf())   # 数値近似: 2.56155281280883
```

21.4　行列の対角化

M.diagonalize() を呼び出すことで，行列の対角化（変換行列 P と対角行列 $D(= P^{-1}MP)$ を返す）ができます。

```
M = Matrix([[0, 1],
            [2, 3]])
aa = M.diagonalize()
print(aa)
P = aa[0]; display(P)
D = aa[1]; display(D)
P.inv()*M*P
```

上で print した部分は

実行結果の例

```
(Matrix([
[-sqrt(17)/4 - 3/4, -3/4 + sqrt(17)/4],
[                1,                1]]), Matrix([
[3/2 - sqrt(17)/2,                0],
[                0, 3/2 + sqrt(17)/2]]))
```

となります。出力される行列は次のようになります。

$$P = \begin{bmatrix} -\frac{\sqrt{17}}{4} - \frac{3}{4} & -\frac{3}{4} + \frac{\sqrt{17}}{4} \\ 1 & 1 \end{bmatrix}, \qquad D = \begin{bmatrix} \frac{3}{2} - \frac{\sqrt{17}}{2} & 0 \\ 0 & \frac{3}{2} + \frac{\sqrt{17}}{2} \end{bmatrix}$$

を意味します。7 行目の出力は

$$\begin{bmatrix} \frac{-2\sqrt{17}-6}{3\sqrt{17}+17} + \frac{8\left(-\frac{\sqrt{17}}{4} - \frac{3}{4}\right)}{3\sqrt{17}+17} + \frac{12}{3\sqrt{17}+17} & \frac{-2\sqrt{17}-6}{3\sqrt{17}+17} + \frac{8\left(-\frac{3}{4} + \frac{\sqrt{17}}{4}\right)}{3\sqrt{17}+17} + \frac{12}{3\sqrt{17}+17} \\ \left(-\frac{\sqrt{17}}{4} - \frac{3}{4}\right)\left(\frac{3\sqrt{17}}{17} + 1\right) + \frac{3}{2} + \frac{13\sqrt{17}}{34} & \left(-\frac{3}{4} + \frac{\sqrt{17}}{4}\right)\left(\frac{3\sqrt{17}}{17} + 1\right) + \frac{3}{2} + \frac{13\sqrt{17}}{34} \end{bmatrix}$$

ですが，これは整理すれば D になってます。

```
print( simplify( P.inv()*M*P) )
```

実行結果の例

```
Matrix([[3/2 - sqrt(17)/2, 0], [0, 3/2 + sqrt(17)/2]])
```

21.5　固有値の数値計算

よく知られているように，5 次以上の一般の多項式には代数的な解の公式が存在しません（アーベル・ルフィニの定理）。実際に適当な 5 次行列を作って，厳密な固有値を計算させてみましょう。

```
# 5x5 行列の定義
M5 = Matrix([[1, 2, 0, 0, 3],
             [4, 5, 6, 0, 0],
             [0, 7, 8, 9, 0],
             [0, 0, 1, 2, 3],
```

```
6                [4, 0, 0, 5, 6]])
7
8  # 厳密解を試みる
9  print(M5.eigenvals())
```

実行結果の例

```
CRootOf(x**5 - 22*x**4 + 91*x**3 + 372*x**2 - 1014*x - 2664, 0): 1,
 CRootOf(x**5 - 22*x**4 + 91*x**3 + 372*x**2 - 1014*x - 2664, 1): 1,
 ... (以下、4 番目の根まで続く)
```

出力された `CRootOf(f(x), n)` とは，「多項式 $f(x) = 0$ の n 番目の複素数根」を意味する SymPy の表現です。解の公式がないけど，この 5 次方程式の解そのものを新しい記号として厳密に保持しているわけです。

21.5.1　SymPy による固有値の数値計算

しかし，このままでは具体的な数値が全く分かりません。SymPy の行列計算では，成分に一つでも `float` が混ざっていると，計算は（誤差を伴う）数値計算となります。

```
1  # 5x5 行列の定義
2  M5 = Matrix([[1, 2, 0, 0, 3],
3               [4, 5, 6, 0, 0],
4               [0, 7, 8, 9, 0],
5               [0, 0, 1, 2, 3],
6               [4, 0, 0, 5, 6]])
7
8  M5 = M5.n()    # 成分をfloatにする
9  print(M5.eigenvals())
```

実行結果の例

```
-2.35326689692323 - 0.365372656601148*I: 1, -2.35326689692323 + 0.365372656601148*I: 1, 3.77312346053621: 1,
```

SymPy はそもそも厳密な計算（計算機代数）を目的として作られていますので，小さいサイズの行列の計算なら SymPy で十分ですが，大きな行列の数値計算には不向きです。

少し規模を大きくして **40 行 40 列（40×40）のランダムな行列** の固有値を計算させ，その処理時間を計測してみましょう。

Jupyter ノートブックのセルの先頭に `%%time` と書くと，そのセルの実行時間をミリ秒単位で計測できます。

```
1  %%time
2  import random
3  from sympy import *
4
5  # 1) ランダムな小数成分（0以上1未満）を持つ 40x40 行列を SymPy で作成
6  data = [[random.random() for _ in range(40)] for _ in range(40)]
7  M_large = Matrix(data)
8
9  # 2) 固有値を計算（40個の複素数を含む辞書が返る）
10 egvals_dict = M_large.eigenvals()
```

実行結果の例

```
CPU times: user 20.8 s, sys: 4.72 ms, total: 20.8 s
Wall time: 21 s
```

たった 40×40 の行列の固有値の計算だけで，SymPy は 20 秒以上もかかりました。せっかくなので，得られた固有値（複素数のリスト）を平面にプロットしてみましょう。

```
1  import matplotlib.pyplot as plt
```

```
2
3  # 固有値の辞書からキーを取り出し，Pythonの複素数型（complex）に変換する
4  egvals_list = [complex(z) for z in egvals_dict.keys()] # 固有値のリスト
5
6  # 縦横比（アスペクト比）を 1:1 に固定する
7  plt.gca().set_aspect('equal')
8
9  # 実部（real）をx軸、虚部（imag）をy軸にして散布図を描く
10 x_coords = [z.real for z in egvals_list]
11 y_coords = [z.imag for z in egvals_list]
12 plt.scatter(x_coords, y_coords)
13
14 plt.show()
```

結果は次のようになります。

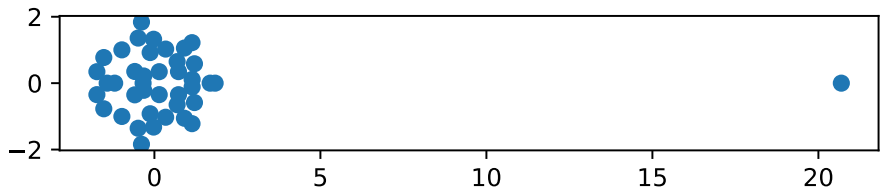


図 28　複素平面上での固有値の分布

実軸上の離れた場所に実固有値がポツンと 1 点だけ存在し，それ以外の残りの固有値は原点付近の円の中に集中的に分布しているのがわかります。一般に，すべての成分が正である行列には，絶対値が最大の正の実固有値がただ一つ存在し，その固有ベクトルの成分もすべて正となることが知られています。この特別な固有値は **Perron–Frobenius（ペロン・フロベニウス）固有値** と呼ばれます。

21.5.2　NumPy による計算との比較

さて，上では 40×40 といった行列の固有値の計算でもそれなりに時間を要しました。SymPy は本来，任意精度（多倍長）演算を背景とした「厳密な代数変形や数式処理」を本領とするライブラリであり，大量の浮動小数点演算を機械の限界まで高速に一括処理するタスクには最適化されていません。

ここでは，数値計算専用のライブラリ **NumPy**（ナムパイ）を使って，その演算性能を比較してみましょう。データは先ほどと全く同一のものを引き継ぎます。

```
1  %%time
2  import numpy as np
3
4  # 同一のデータを NumPy の多次元配列（ndarray）に変換
5  M_large_np = np.array(data)  # NumPyの行列の形式
6
7  # NumPy に最適化されたアルゴリズムで固有値を計算
8  res_numpy = np.linalg.eigvals(M_large_np)
```

実行結果の例

```
CPU times: user 445 μs, sys: 8 μs, total: 453 μs
Wall time: 438 μs
```

わずか約 0.0004 秒（0.4 ミリ秒で固有値が計算されました。演算結果の出力に要した時間は一目瞭然です。SymPy による処理には約 20 秒を要したのに対し，NumPy による演算はわずか 0.4 ミリ秒，約 **5 万倍の速**

度で完了しました。

NumPy は，その内部コアが C 言語や Fortran といったコンパイル言語で記述されており，現代の CPU のアーキテクチャ（SIMD 演算や BLAS/LAPACK といった高速線形代数ライブラリ）を最大限に活かすよう設計されています。

手元で厳密に数式モデルを構築する際には SymPy を用いて，大規模な数値実験やシミュレーションを行う際は NumPy を使うのが良いでしょう。

21.6 練習問題

21.6.1 問題 1：行列のべき乗によるフィボナッチ数列の計算

フィボナッチ数列 F_n（$F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$）は，行列のべき乗を用いて次のように表現できることが知られています。

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

SymPy を用いて $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ を定義し，行列のべき乗演算（`A**n`）を利用して，**フィボナッチ数列の第 256 項**（F_{256}）を厳密に計算するプログラムを作成しなさい。また，得られた値（非常に大きな整数になります）を答えなさい。

21.6.2 問題 2：ガウス乱数行列の固有値分布（SymPy 版）

平均 0，標準偏差 1 のガウス分布（正規分布）に従う乱数を用いて 100×100 の行列を作成し[*16]，その固有値の分布を複素平面上にプロットしてみましょう。

具体的には，以下の手順に従うプログラムを作成し，実行結果のグラフを提出してください。

1. `random.gauss(0, 1)` を用いて，各成分がガウス乱数である 100×100 の SymPy の行列 `M_gauss` を作成する。
2. `M_gauss` の固有値を数値計算（`.eigenvals()`）によって求める。
3. 得られた固有値を Python の複素数型（`complex`）に変換し，Matplotlib を用いて複素平面上に散布図としてプロットする（アスペクト比は 1:1 に固定すること）。

21.6.3 問題 3：ガウス乱数行列の固有値分布（NumPy 版）

【問題 2】と同じガウス乱数行列の固有値計算およびプロットを，NumPy を用いて行いなさい。

21.6.4 問題 4：行列の対角化と一般項の導出

【問題 1】で扱った行列 $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ について，SymPy の `.diagonalize()` メソッドを用いて対角化を行い，$A = PDP^{-1}$ となる可逆行列 P と対角行列 D を求めなさい（結果には黄金比 $\phi = \frac{1+\sqrt{5}}{2}$ が含まれます）。

さらに，得られた P と D から $A^n = PD^nP^{-1}$ を計算させることで，フィボナッチ数列の一般項を SymPy 上で導出し，その数式を表示しなさい。

[*16] Python の標準ライブラリ `random` を用いてガウス乱数（正規分布）を生成するには，`random.gauss(mu, sigma)` を使用します。ここで `mu` は平均（μ），`sigma` は標準偏差（σ）を指定するため，今回は `random.gauss(0, 1)` と記述することで標準正規分布に従う乱数が得られます。

22　NumPy による数値計算

前節では，数式処理ライブラリである SymPy を用いて行列の固有値の厳密計算やシンボリックな操作を扱いました。しかし，SymPy は数式としての正確さを維持するために膨大なメモリと計算時間を消費します。実際，SymPy（および Python の標準リスト）を用いて少し大きな行列（例えば 200×200 ぐらい）の固有値を計算しようとすると，非常に時間がかかってしまいます。前節の最後でみたように NumPy を用いれば，固有値の数値計算は 200×200 行列の固有値であれば 0.027 秒，3000×3000 ぐらいの行列の固有値であっても約 12 秒で計算することができます。

そこで本節では，Python における高速数値計算の事実上の標準ライブラリである **NumPy** を解説します。NumPy は，C 言語などで書かれ最適化された数値計算エンジンを裏で動かしており，大規模な行列演算やシミュレーションを高速で行うことができます。

22.1　Python のリストと NumPy 配列の「メモリ構造」の違い

NumPy が高速に計算できる理由を知るために，Python のリストと NumPy の配列の違いを見てみましょう。その鍵は，メモリ上でのデータの持ち方（データ構造）と，設計思想の違いにあります。

Python のリスト：　Python のリストは何でも入る汎用名簿であり，数値だけでなく，文字列や他のリストなど，異なる型のデータを 1 つのリストの中に混在させて自由に扱うことができます。また，扱う人間が「このデータは整数か？実数か？」といった細かい型情報を一切気にする必要がなく，コンピュータ（Python インタプリタ）が裏側ですべて自動的にハンドリングしてくれます。この柔軟性を実現するため，リストはメモリ上にバラバラに置かれたデータの「住所（ポインタ）」だけを並べた構造をしています。しかし，この仕組みは数値計算においては大きな代償を伴います。成分にアクセスして計算するたびに「バラバラの住所（メモリ番地）を一つずつ巡って，どのような型のデータ（数値であっても `int` か `float` か）かを確認する手間（型チェック）が毎回発生するため，動作が極めて遅くなり，余分な管理データによってメモリ消費も多くなります。

NumPy 配列（整列した専用教室）：　一方，NumPy の多次元配列（`ndarray`）は，数値計算に特化しています。そのため「何でも入る」という自由度はなく，**配列の中身はすべて同じデータ型**（例えば 64bit 浮動小数点数 `float64` のみ）に統一されていなければなりません。その代わり，C 言語の配列と同様に，メモリ上の「連続した 1 つの領域」にデータが隙間なくギチギチに一列に並びます。
これは，いわば「同じ制服（データ型）を着た人たちが，教室の座席（連続メモリ）に順番通り綺麗に整列して座っている状態」です。誰がどこに座っているか（住所）を都度探す必要もなく，全員の型も最初から分かっているため，余分なメモリ消費を極限まで抑え，余計なことは考えずに圧倒的なスピードでデータの読み書きや計算をさせることができます。

また，Python の `for` ループを使って配列の成分を 1 つずつ処理する代わりに，この「綺麗に整列した配列」に対して全成分を一括して高速処理する仕組みを**ベクトル化（Ufunc：Universal Function）**と呼びます[*17]。

[*17] 数学において「ベクトル」とは向きと大きさを持った量（あるいはベクトル空間の元）を指しますが，計算機科学・プログラミングの文脈では，単に「データを 1 次元に並べた配列（シーケンス）」のことをベクトルと呼びます。

22.2 NumPy の基本操作

NumPy を利用する際は，通常 `import numpy as np` という別名[*18]でインポートするのが慣習になっています。

ここでは，数値解析や微分方程式のシミュレーションにおいて，特によく使われる基本機能を紹介します。Python の標準リストによる実装と比較しながら，その合理的な設計思想を見ていきましょう。

```python
import numpy as np

# 1) リストから NumPy 配列を作成する
a = np.array([1.0, 2.0, 3.0])

# 2) 最初から NumPy で 3600x3600 のランダム行列（一様乱数）を直接作成する
M_large = np.random.rand(3600, 3600)

# 3) すべての成分が 0 の配列を作成
y_data = np.zeros(100)

# 4) 等差数列（グリッド）の生成
# 範囲 [0, 2] を 5等分する（端点を含む 5つの点を並べる）
t_coords = np.linspace(0, 2, 5)
print("linspace:", t_coords)
```

実行結果の例

```
linspace: [0.  0.5 1.  1.5 2. ]
```

それぞれのコードについて，Python のリストと比較した際の特徴を解説します。

1. **`np.array()` による変換：**
   既存の Python リストを NumPy 配列（`ndarray`）に一瞬で変換します。前述の通り，この変換が行われた瞬間に，メモリ上にバラバラに散らばっていたデータが，連続したメモリ領域へと綺麗に整列し直されます。
2. **NumPy による直接生成（`np.random.rand`）：**
   Python のリストでこれを行うには，2 重のリスト内包表記を用いて `[[random.random() for i in range(3600)] for j in range(3600)]` のように記述する必要があります。この場合，途中で膨大な一時オブジェクトが生成されてメモリを消費します。
   一方，`np.random.rand(3600, 3600)` は，リストを一切経由せず，最初から「連続したメモリ領域」を確保して乱数を書き込みます。無駄なデータ確認（型チェック）や一時オブジェクトの生成が一切ないため無駄がなく高速です。
3. **ゼロ配列の確保（`np.zeros`）：**
   微分方程式のシミュレーション（時間発展の計算など）では，各ステップにおける計算結果（位置や速度など）を記録していくためのゼロの配列が必要な場合があります。Python のリストでこれを行う場合，`y_data = [0.0] * 100` のように書きますが，NumPy の `np.zeros(100)` を使えば，最初から浮動小数点数（`float64`）のゼロがギチギチに 100 個詰まった専用メモリをが効率的な形で高速に確保されます。
4. **等差数列の生成（`np.linspace`）：**

[*18] エイリアスとかいうらしい

時間発展のシミュレーションにおいて，時間軸を等間隔の「刻み幅 Δt」で分割する際など，非常に多くの場面で使う関数です。

22.3　高速な配列演算と行列積

NumPy 配列同士の四則演算や数学関数の適用は，for 文を使わずに「配列全体に対して一括」で行うことができます。また，行列同士の積には演算子 @（または np.dot）を使用します。

```
import numpy as np

x = np.array([1, 2, 3])
y = np.array([10, 20, 30])

# 要素ごとの一括演算（ベクトル化演算）
print("x + y  =", x + y)
print("x * 2  =", x * 2)
print("sin(x) =", np.sin(x)) # np.sin（これはNumPyのsin） を使うことで全要素に一括適用される

# ベクトルの積
print(x*y)     # 要素ごとの積
print(x@y)     # 内積

# 行列の積
A = np.array([[1, 2],
              [3, 4]])
print("A * A（要素ごとの積）:\n", A * A)
print("A @ A（行列としての積）:\n", A @ A)
```

実行結果の例

```
x + y  = [11 22 33]
x * 2  = [2 4 6]
sin(x) = [0.84147098 0.90929743 0.14112001]
A * A（要素ごとの積）:
 [[ 1  4]
 [ 9 16]]
A @ A（行列としての積）:
 [[ 7 10]
 [15 22]]
```

22.3.1　【注意】SymPy の行列演算との最大の違い

前節で学んだ SymPy による行列計算と，本節の NumPy による計算では，演算子の意味が大きく異なるため，注意が必要です。

- **SymPy の * は「行列の積」:**
  SymPy では，行列同士に対して A * B と書くと，数学的な行列の積が計算されました。
- **NumPy の * は「要素ごとの積（アダマール積）」:**
  NumPy では，行列同士に対して A * B と書くと，同じ位置にある成分同士を掛け合わせる演算になります。上記の出力結果の A * A を見ると，各成分が $1^2 = 1$，$2^2 = 4$，$3^2 = 9$，$4^2 = 16$ となっており，これは数学的な行列の積 A^2 とは異なります。
- **NumPy の @ が「数学的な行列の積」:**
  NumPy において線形代数としての「行列の掛け算」を行いたい場合は，必ず @ 演算子を使用します。A @ A とすることで，私たちがよく知る線形代数の定義通りに行列の積（行と列の内積の組み合わせ）

が計算され，正しい答えが得られます。

- **NumPy での複素内積**：複素数を成分に持つ NumPy の配列 a,b に対して np.dot(a,b) で a と b の複素内積を計算します。ここで，a の成分の複素共役がとられることに注意しましょう。

```
a = np.array([1 + 1j, 2 - 3j])    # 複素ベクトル
b = np.array([1 - 2j, 3 + 4j])    # 複素ベクトル

# np.vdot を使うと自動的に a の複素共役を取って内積を計算する
res_vdot = np.vdot(a, b) # (1-1j)*(1-2j) + (2+3j)*(3+4j) = -7+14j
print("np.vdot(a, b) =", res_vdot)
```

このように，NumPy 配列は「要素ごとの一斉処理（ベクトル化）」が基本の設計思想になっているため，四則演算子（+, -, *, /）はすべて「要素ごとの処理」として割り当てられており，ベクトルの内積や行列の積のような線形代数の演算には専用の @ が用意されているのです。

22.3.2　NumPy と Python のリストとの速度の比較

最後に，Python の標準リストと NumPy の配列で，データの作成から実際の計算までを含めた総合的な実行速度にどれほどの差が生じるかを実験してみましょう。

ここでは，**100 万個のデータ**を作成して一斉に 2 倍にするという極めて単純な処理を，time モジュールを使って測定して比較します。

Jupyter Notebook や Python スクリプトで以下のコードを実行してみましょう。

```
import time
import random
import numpy as np

N = 1000000

# 1) Pure Python（リスト生成 + 2倍にする処理）
t0 = time.time()

points_list = [random.random() for _ in range(N)] # リスト生成
result_list = [x * 2 for x in points_list]        # 2倍にする

t1 = time.time()
time_list = t1 - t0
print(f"Pure Python: {time_list:.4f} 秒")


# 2) Pure NumPy（配列生成 + 2倍にする処理）
t2 = time.time()

points_np = np.random.rand(N)  # 配列生成
result_np = points_np * 2      # 2倍にする

t3 = time.time()
time_np = t3 - t2
print(f"Pure NumPy : {time_np:.4f} 秒")

# 速度比の計算
print(f"NumPy の方が約 {time_list / time_np:.1f} 倍高速！")
```

実行結果の例

```
Pure Python: 0.1427 秒
Pure NumPy : 0.0114 秒
NumPy の方が約 12.5 倍高速！
```

23　微分方程式の数値計算

23.1　常微分方程式とベクトル場（勾配場）

次の微分方程式を考えましょう。

$$\frac{dy}{dx} = y - x. \tag{7}$$

これの一般解は任意定数 C を用いて $y(x) = x + 1 + Ce^x$ と表されます。次に，微分方程式 (7) を初期条件

$$y(0) = \frac{2}{3} \tag{8}$$

の元で解いてみましょう。一般解に $x = 0$ を代入し，$y(0) = 1 + C = 2/3$ から定数を求めると $C = -1/3$ となるため，この初期値問題の厳密解（真の解）は

$$y(x) = x + 1 - \frac{1}{3}e^x \tag{9}$$

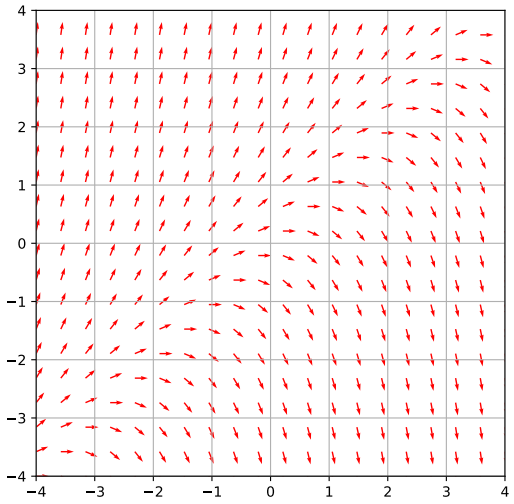
となります。

コンピュータによる描画を用いて，解の様子を視覚的に観察してみましょう。微分方程式 (7) の解曲線 $y = y(x)$ は，平面上の点 (x, y) で傾き $y - x$ を持ちます。この性質を利用して，各点 (x, y) に傾きの方向を向いた単位ベクトルを描いたものを**勾配場（slope field）**と呼びます。

Python の `matplotlib` に用意されている `quiver` 関数を使い，この勾配場を描画してみましょう。

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

# グリッド（格子点）の作成
x_vals = np.linspace(-4, 4, 20)
y_vals = np.linspace(-4, 4, 20)
X, Y = np.meshgrid(x_vals, y_vals)

# 各点におけるベクトルの成分 (dx, dy)
# dy/dx = Y - X より，方向ベクトルは (1, Y - X)
U = np.ones_like(X)
V = Y - X

# 単位ベクトルにするための規格化（長さの統一）
N = np.sqrt(U**2 + V**2)
U = U / N
V = V / N

# 描画
plt.figure(figsize=(6, 6))
plt.quiver(X, Y, U, V, color="red", angles="xy")
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.grid(True)
```

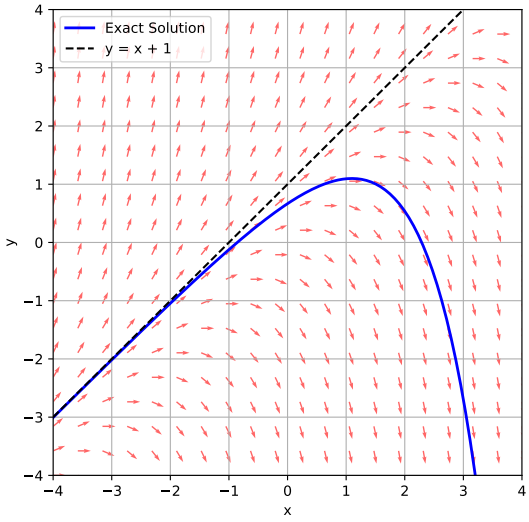
```
plt.show()
```



微分方程式 (7) の解は，初期条件 $(0, y(0))$ から出発して，上の絵の勾配場（矢印）をその向きに辿ることで得られます。

この勾配場の上に，漸近線 $y = x + 1$ （黒破線）と，初期条件を通る厳密解 (9)（青線）を重ねてプロットしてみます。

```
# 厳密解のプロット用データ
x_sol = np.linspace(-4, 4, 200)
y_exact = x_sol + 1 - (1/3) * np.exp(x_sol)
y_asym = x_sol + 1

plt.figure(figsize=(6, 6))
# 勾配場（赤矢印）
plt.quiver(X, Y, U, V, color="red", alpha=0.6, angles="xy")
# 厳密解（青線）
plt.plot(x_sol, y_exact, color="blue", label="Exact Solution", linewidth=2)
# 漸近線（黒破線）
plt.plot(x_sol, y_asym, color="black", linestyle="--", label="y = x + 1")

plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
```



勾配場を見れば，解の振る舞いは一目瞭然です。直線 $y = x + 1$ を境に，その上側と下側で $x \to \infty$ のときの行き先が大きく分かれる（上側は ∞ へ発散，下側は $-\infty$ へ発散する）ことが視覚的に理解できます。

23.2　Euler 法による数値シミュレーション

微分方程式の多くは手計算で解く（代数的に解を求める）ことはできませんが，コンピュータを用いて数値的に近似解を計算することが可能です。ここでは，最も基本的な数値計算法である **Euler（オイラー）法**を紹介します。

$f(x, y)$ を与えられた関数として，微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{10}$$

及び初期条件 $y(x_0) = y_0$ を考えます。(x_0, y_0) は与えられた数値です。上の微分方程式は無限小 dx を用いて

$$y(x + dx) = y(x) + f(x, y)dx \tag{11}$$

と表すことができます。Euler 法は無限小 dx を小さい数 Δx に置き換えることで解を近似しようとするものです。つまり，初期値 (x_0, y_0) とするとき，$x_1 := x_0 + \Delta x$ での y の値は

$$y_1 = y_0 + f(x_0, y_0)\Delta x \tag{12}$$

で近似される。次に $x_2 := x_1 + \Delta x$ での y の値は

$$y_2 = y_1 + f(x_1, y_1)\Delta x \tag{13}$$

で近似されるといった具合で近似解を構成します。

Euler 法では微分方程式 $\frac{dy}{dx} = f(x, y)$ において，微小な変化量 Δx に対する次の一歩は，次式で近似されます。

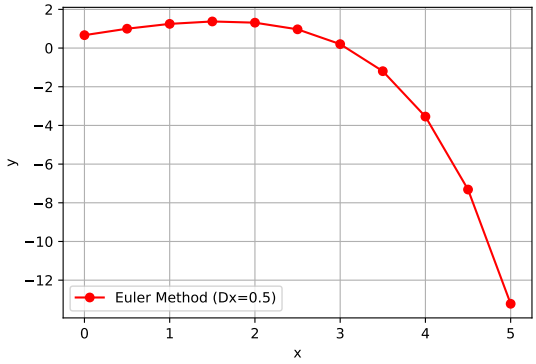
$$y_n = y_{n-1} + f(x_{n-1}, y_{n-1})\Delta x \tag{14}$$

NumPy の配列（`zeros` など）を使い，上の手順で近似解を計算しましょう。

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

# パラメータ設定
Dx = 0.5                    # 刻み幅 Delta x
x0, y0 = 0.0, 2/3           # 初期条件
steps = 10                  # ステップ数

# 計算結果を格納する NumPy 配列をあらかじめ確保
x_data = np.zeros(steps + 1)    # とりあえずは
y_data = np.zeros(steps + 1)    # 全部0にしておく

# 初期値を代入
x_data[0] = x0
y_data[0] = y0

# Euler法のメインループ
for j in range(steps):
    f_val = y_data[j] - x_data[j]                # 傾き f(x, y) = y - x
    x_data[j+1] = x_data[j] + Dx                 # x の更新
    y_data[j+1] = y_data[j] + f_val * Dx         # y の更新

# 計算結果の出力
for step in range(steps + 1):
    print(f"Step {step:2d}: (x, y) = ({x_data[step]:.2f}, {y_data[step]:.5f})")

# 描画
plt.figure(figsize=(6, 4))
plt.plot(x_data, y_data, marker="o", color="red", label="Euler Method (Dx=0.5)")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
```

実行結果の例

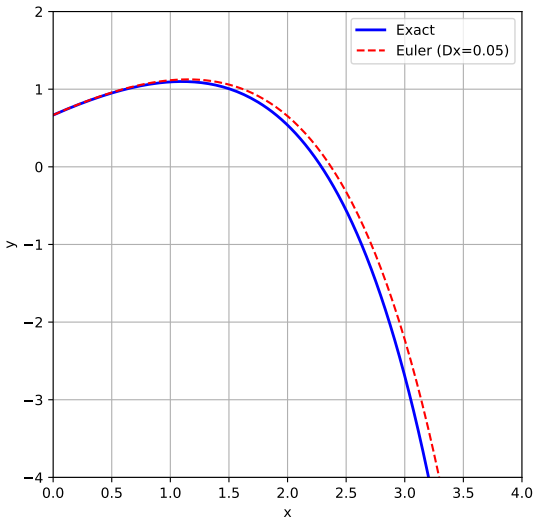
```
Step  0: (x, y) = (0.00, 0.66667)
Step  1: (x, y) = (0.50, 1.00000)
Step  2: (x, y) = (1.00, 1.25000)
Step  3: (x, y) = (1.50, 1.37500)
Step  4: (x, y) = (2.00, 1.31250)
Step  5: (x, y) = (2.50, 0.96875)
Step  6: (x, y) = (3.00, 0.20312)
Step  7: (x, y) = (3.50, -1.19531)
Step  8: (x, y) = (4.00, -3.54297)
Step  9: (x, y) = (4.50, -7.31445)
Step 10: (x, y) = (5.00, -13.22168)
```



23.2.1 刻み幅 Δx と精度の関係

オイラー法の近似精度は，刻み幅 Δx を小さくするほど向上します。刻み幅を `Dx = 0.05` と小さくし，ステップ数を 80 に増やして，厳密解（青線）と重ねて比較プロットしてみましょう。

```
# パラメータ設定
Dx = 0.05                   # 刻み幅を小さくする
x0, y0 = 0.0, 2/3
steps = 80                  # 4.0 まで計算するためステップ数を増やす

x_approx = np.zeros(steps + 1)
y_approx = np.zeros(steps + 1)

x_approx[0] = x0
y_approx[0] = y0

for j in range(steps):
    f_val = y_approx[j] - x_approx[j]
    x_approx[j+1] = x_approx[j] + Dx
    y_approx[j+1] = y_approx[j] + f_val * Dx

# 描画と真の解との比較
x_sol = np.linspace(0, 4, 200)
y_exact = x_sol + 1 - (1/3) * np.exp(x_sol)

plt.figure(figsize=(6, 6))
plt.plot(x_sol, y_exact, color="blue", label="Exact", linewidth=2)
plt.plot(x_approx, y_approx, color="red", label="Euler (Dx=0.05)", linestyle="--")
plt.xlim(0, 4)
plt.ylim(-4, 2)
plt.legend()
plt.grid(True)
plt.show()
```



刻み幅を細かくしたことで，赤い点線（オイラー法）が青い実線（厳密解）にかなり近づいたことが確認できます。学生の皆さんは，手元のコードの `Dx` をさらに小さく（例えば `0.001` に，ステップ数を `4000` に）変更して，どれほど精度が変わるか試してみましょう。

23.3　ルンゲ＝クッタ（Runge-Kutta）法

Euler 法はシンプルで実装が容易ですが，1 ステップ前の点における微分係数（傾き）のみを用いて次ステップの値を決定するため，解曲線が湾曲している領域では誤差が累積しやすく，ステップの刻み幅に比例した誤差が生じます。

この精度上の課題に対し，計算の刻み幅 Δx を極端に小さくして計算コストを増加させるのではなく，アルゴリズムの工夫によって「高次の近似精度」を達成するアプローチが，数値解析において標準的に用いられている **4 次のルンゲ＝クッタ（Runge-Kutta, RK4）法**です。

23.3.1　ルンゲ＝クッタ法のアルゴリズム

微分方程式 $\frac{dy}{dx} = f(x, y)$ において，現在値 (x_n, y_n) から次の一歩 (x_{n+1}, y_{n+1}) へ更新する際，RK4 では区間 $[x_n, x_n + \Delta x]$ 内の代表的な 4 点において微分係数を評価し，それらの重み付き平均を用いて近似解を構成します。

具体的には，以下の手順で 4 つの傾き（微分係数の評価値） k_1, k_2, k_3, k_4 を順次計算します。

$$k_1 = f(x_n, y_n) \qquad \text{（始点における傾き：Euler 法と同じ）} \tag{15}$$

$$k_2 = f\left(x_n + \frac{\Delta x}{2}, y_n + k_1 \frac{\Delta x}{2}\right) \qquad \text{（中間点における傾きの第 1 近似）} \tag{16}$$

$$k_3 = f\left(x_n + \frac{\Delta x}{2}, y_n + k_2 \frac{\Delta x}{2}\right) \qquad \text{（中間点における傾きの第 2 近似）} \tag{17}$$

$$k_4 = f\left(x_n + \Delta x, y_n + k_3 \Delta x\right) \qquad \text{（終点における傾きの近似）} \tag{18}$$

これら 4 つの傾きに対して，シンプソンの公式（Simpson's rule）に類似した以下の重み付き平均 k を求め

ることで，局所的に 4 次精度の近似更新を行います。

$$k = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{19}$$

$$y_{n+1} = y_n + k\Delta x \tag{20}$$

このアルゴリズムは，テイラー展開の 4 次の項まで厳密解と一致するように設計されているため，1 ステップごとの**局所打切り誤差**がステップ幅 Δx の 5 乗（$\mathcal{O}(\Delta x^5)$）に比例します。時間全体を通した累積誤差（大域打切り誤差）は Δx の 4 乗（$\mathcal{O}(\Delta x^4)$）に比例するため，4 次のルンゲ＝クッタ法は**「4 次精度」**の解法に分類されます。

1 次精度である Euler 法との精度の差は圧倒的です。例えば，刻み幅 Δx を $1/10$ にした場合：

- **Euler 法（1 次精度）**：誤差は $1/10$ にしか減りません。
- **ルンゲ＝クッタ法（4 次精度）**：誤差は $(1/10)^4 = 1/10000$ にまで激減します。

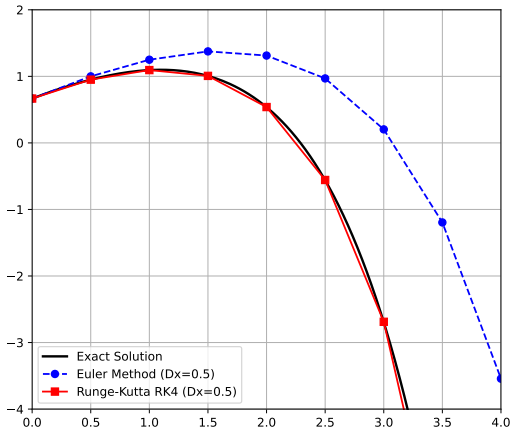
このように，ルンゲ＝クッタ法は非常に高い精度で微分方程式の解を得ることができるため，科学技術計算やシミュレーションのスタンダードとして広く用いられています。

23.3.2　NumPy によるルンゲ＝クッタ法の実装

それでは，Euler 法のときと同じ初期値問題（微分方程式：$y' = y - x$, 初期値：$y(0) = 2/3$）に対して，RK4 を実装してみましょう。Euler 法（青点線）とルンゲ＝クッタ法（赤実線）の近似精度を視覚的に比較するために，比較的大きな刻み幅 `Dx = 0.5` を設定してシミュレーションを行います。

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

# パラメータ設定
Dx = 0.5                    # 刻み幅
x0, y0 = 0.0, 2/3           # 初期条件
steps = 8                   # x=4.0まで計算するためのステップ数

# 1) NumPy配列の事前確保（ゼロクリア）
x_rk = np.zeros(steps + 1)
y_rk = np.zeros(steps + 1)
x_rk[0], y_rk[0] = x0, y0

# 傾きを定義する関数 f(x, y)
def f(x, y):
    return y - x

# 2) ルンゲ＝クッタ法のメインループ
for j in range(steps):
    x_n = x_rk[j]
    y_n = y_rk[j]

    # 4点における微分係数の評価
    k1 = f(x_n, y_n)
    k2 = f(x_n + Dx/2, y_n + k1*Dx/2)
    k3 = f(x_n + Dx/2, y_n + k2*Dx/2)
    k4 = f(x_n + Dx, y_n + k3*Dx)

    # 重み付き平均を用いた更新
```

```
    x_rk[j+1] = x_n + Dx
    y_rk[j+1] = y_n + (k1 + 2*k2 + 2*k3 + k4) / 6 * Dx

# 比較用のEuler法（同じDx=0.5で計算）
x_eu = np.zeros(steps + 1)
y_eu = np.zeros(steps + 1)
x_eu[0], y_eu[0] = x0, y0
for j in range(steps):
    x_eu[j+1] = x_eu[j] + Dx
    y_eu[j+1] = y_eu[j] + f(x_eu[j], y_eu[j]) * Dx

# 比較用の厳密解
x_exact = np.linspace(0, 4, 200)
y_exact = x_exact + 1 - (1/3) * np.exp(x_exact)

# プロット描画
plt.figure(figsize=(7, 6))
plt.plot(x_exact, y_exact, color="black", label="Exact Solution", linewidth=2)
plt.plot(x_eu, y_eu, marker="o", color="blue", linestyle="--",
         label="Euler Method (Dx=0.5)")
plt.plot(x_rk, y_rk, marker="s", color="red", label="Runge-Kutta RK4 (Dx=0.5)")
plt.xlim(0, 4)
plt.ylim(-4, 2)
plt.legend()
plt.grid(True)
plt.show()
```



23.4　練習問題

23.4.1　空気抵抗を受ける雨粒の落下シミュレーション

物理や工学で非常によく登場する，空気抵抗を受けながら落下する雨粒の速度変化をルンゲ＝クッタ法（RK4）を用いてシミュレーションしてみましょう。

質量 m の雨粒が重力加速度 g によって自由落下する際，速度 v に比例する空気抵抗（粘性抵抗係数 k）を

受けるとすると，速度の微分方程式は以下のように表されます。

$$\frac{dv}{dt} = g - \frac{k}{m}v \tag{21}$$

ここでは，簡単のため $g = 9.8$ [m/s^2], $\frac{k}{m} = 1.0$ [/s] とし，初期速度 $v(0) = 0.0$ [m/s]（静止状態から落下開始）とします。この微分方程式の厳密解（理論解）は $v(t) = 9.8(1 - e^{-t})$ となり，時間が経つと重力と空気抵抗が釣り合って，一定速度 9.8 m/s （終端速度）に収束することが数学的に分かっています。

課題

4 次のルンゲ＝クッタ法（RK4）を用いて，この速度変化を時間 $t = 0.0$ から $t = 5.0$ まで，時間刻み幅 $\Delta t = 1.0$ （計 5 ステップ）で計算し，厳密解および Euler 法との精度を比較する以下のプログラムを完成させなさい。特に，**ルンゲ＝クッタ法のメインループ部分（23 行目〜33 行目付近）の処理**を，テキストのアルゴリズムを参考に正しく実装しなさい。

ファイル名：rain.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # パラメータ設定
5  g = 9.8
6  gamma = 1.0                # k/m の値
7  Dt = 1.0                   # 意図的に大きくした時間刻み幅
8  steps = 5                  # t=5.0 までのステップ数
9
10 # 傾き関数 f(t, v)
11 def f(t, v):
12     return g - gamma * v
13
14 # 1) 配列の事前確保
15 t_rk = np.zeros(steps + 1)
16 v_rk = np.zeros(steps + 1)
17 t_rk[0], v_rk[0] = 0.0, 0.0
18
19 # 2) ルンゲ＝クッタ法（RK4）のループ処理
20 for j in range(steps):
21     t_n = t_rk[j]
22     v_n = v_rk[j]
23
24     # --- この部分を自分で考える ---
25     """ ルンゲ＝クッタ法（RK4）のアルゴリズムのとおり
26     4つの傾き k1, k2, k3, k4 の計算，および
27     時間 t_rk[j+1] と速度 v_rk[j+1] の更新を行う"""
28     # -------------------------
29
30 # 比較用のEuler法（同じDt=1.0で計算）
31 t_eu = np.zeros(steps + 1)
32 v_eu = np.zeros(steps + 1)
33 t_eu[0], v_eu[0] = 0.0, 0.0
34 for j in range(steps):
35     t_eu[j+1] = t_eu[j] + Dt
36     v_eu[j+1] = v_eu[j] + f(t_eu[j], v_eu[j]) * Dt
```

```
37
38 # 比較用の厳密解（滑らかな曲線を描くため細かく分割）
39 t_exact = np.linspace(0, 5, 200)
40 v_exact = g / gamma * (1 - np.exp(-gamma * t_exact))
41
42 # 描画処理
43 plt.figure(figsize=(7, 5))
44 plt.plot(t_exact, v_exact, color="black", label="Exact Solution (Analytical)")
45 plt.plot(t_eu, v_eu, marker="o", color="blue", linestyle="--",
46          label="Euler Method (Dt=1.0)")
47 plt.plot(t_rk, v_rk, marker="s", color="red", label="Runge-Kutta RK4 (Dt=1.0)")
48 plt.xlabel("Time t [s]")
49 plt.ylabel("Velocity v [m/s]")
50 plt.legend()
51 plt.grid(True)
52 plt.show()
```

24　数値積分と「丸め誤差」

これまでに学んだ微分方程式の解法（オイラー法やルンゲ＝クッタ法）では，刻み幅 h を小さくすればするほど，計算精度が向上することを見てきました。数学の理論上，刻み幅 $h \to 0$ の極限をとれば，近似解は厳密解にいくらでも近づくはずです。

しかし，実際のコンピュータ上で計算を行う場合，この「刻み幅を極限まで小さくすれば精度が上がり続ける」という考え方には大きな注意が必要です。なぜなら，計算機内部での数値表現には桁数の限界があり，計算の各ステップで不可避な小さな誤差が生じるからです。標準的な 64 ビット浮動小数点数（`float64`）の場合，1 回の計算あたり $10^{-16} \sim 10^{-15}$ 程度の誤差が発生します。そのため，精度を上げようとして刻み幅 h を小さくしステップ数を増やしすぎると，この微小な誤差が何千・何万回と累積し，無視できない大きな誤差へと膨れ上がります。

本節では，定積分の数値計算（数値積分）を題材にして，理論上の数学と現実の計算機との間に生じるギャップ，そしてアルゴリズムに由来する「打ち切り誤差」と，計算機に由来する「丸め誤差」のトレードオフについて実験・考察します。

24.1　数値積分の基本手法：定積分の近似

まず，関数 $f(x)$ の定積分

$$I = \int_a^b f(x)\,dx$$

をコンピュータで近似計算する代表的な 3 つの手法を整理しておきましょう。区間 $[a, b]$ を n 等分し，刻み幅を $h = \frac{b-a}{n}$，分割点を $x_k = a + kh \quad (k = 0, 1, \ldots, n)$ とします。

1. **矩形法（長方形近似）：**　各小区間の左端の値 $f(x_k)$ を高さとする長方形の面積の和で近似します。

$$I_{\text{rect}} = h \sum_{k=0}^{n-1} f(x_k)$$

理論上の誤差（打ち切り誤差）は

$$E_{\text{rect}} := |I - I_{\text{rect}}| \le \frac{b-a}{2} \max_{x \in [a,b]} |f'(x)| h$$

であり，これは $\mathcal{O}(h)$（1 次精度）です。

2. **台形法（台形近似）：**　各小区間を直線で結び，台形の面積の和として近似します。

$$I_{\text{trap}} = h\left[\frac{f(x_0)+f(x_n)}{2} + \sum_{k=1}^{n-1} f(x_k)\right]$$

理論上の打ち切り誤差は

$$E_{\text{trap}} := |I - I_{\text{trap}}| \le \frac{b-a}{12} \max_{x\in[a,b]} |f''(x)|h^2$$

であり，これは $\mathcal{O}(h^2)$（2 次精度）です。

3. **シンプソン法（2 次関数近似）：**　隣り合う 2 つの小区間をペアにして（n は偶数），3 点を通る 2 次関数で近似します。

$$I_{\text{simp}} = \frac{h}{3}\left[f(x_0) + 4\sum_{k=1(\text{奇数})}^{n-1} f(x_k) + 2\sum_{k=2(\text{偶数})}^{n-2} f(x_k) + f(x_n)\right]$$

テイラー展開の奇数次の項が巧みに打ち消し合うため，理論上の打ち切り誤差は

$$E_{\text{simp}} := |I - I_{\text{simp}}| \le \frac{b-a}{180} \max_{x\in[a,b]} |f^{(4)}(x)|h^4$$

となり，$\mathcal{O}(h^4)$（3 次ではなく 4 次精度！）という非常に高い精度が得られます。

24.2　打ち切り誤差 vs 丸め誤差のトレードオフ

数値計算に含まれる誤差は，大きく分けて以下の 2 つに分類されます。

1. **打ち切り誤差（Truncation Error）：**
   極限などの無限の操作を有限回の計算で打ち切る（近似する）ことによって生じる数学的な誤差です。刻み幅 h を小さくする（分割数 n を増やす）ほど，打ち切り誤差は小さくなります。
2. **丸め誤差（Rounding Error）：**
   コンピュータが実数を 64 ビット（仮数部 52 ビット：10 進数で約 15〜17 桁）といった有限の精度で表現・計算することによって生じる誤差です。刻み幅 h を小さくすると計算のステップ数（分割数 n）が膨大になり，$10^{-16} \sim 10^{-15}$ 程度の微小な丸め誤差が累積し，無視できない大きさになることがあります。

全体の誤差（全誤差）は，おおまかに打ち切り誤差 ＋ 丸め誤差で評価されます。そのため，h を小さくしていくと，はじめは打ち切り誤差が減少して計算精度が向上しますが，ある限界点（最適刻み幅 h_{opt}）を超えると，今度は累積した丸め誤差の影響が支配的となり，全誤差が再び増加に転じます。

24.3　Python による数値積分の実験

以下のテスト関数を用いて，実際にシンプソン法での分割数 n と誤差の関係をプロットしてみましょう。

$$\int_0^1 e^x\,dx = e - 1 \approx 1.718281828459045 \tag{22}$$

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

```
3
4  # 1. テスト関数と厳密解の定義
5  def f(x):
6      return np.exp(x)
7
8  true_val = np.e - 1.0  # 定積分 e^x の厳密解 (e - 1)
9
10 # 2. シンプソン法の実装
11 def simpson(f, a, b, n):
12     h = (b - a) / n
13     x = np.linspace(a, b, n + 1)  # x座標の配列を一度につくる
14     y = f(x)                      # 対応する y座標を一気に計算
15
16     # 係数 [1, 4, 2, 4, ..., 2, 4, 1] の作成 (スライス [start:stop:step] を利用)
17     weights = np.ones(n + 1)  # 全部 1 の配列をつくる
18     weights[1:-1:2] = 4   # 1+1番目から最後(-1番目)の手前までを2刻みで4にする
19     weights[2:-2:2] = 2   # 2+1番目から最後から2番目の手前までを2刻みで2にする
20
21     return (h / 3.0) * np.sum(weights * y)
22
23 # 3. 分割数 n と刻み幅 h の設定 (n = 2^1, 2^3, ..., 2^27)
24 n_powers = np.arange(1, 28, 2)     # 配列 [1, 3, 5, ..., 27]
25 n_vals = 2**n_powers               # 配列 [2, 8, 32, ..., 134217728]
26 h_vals = 1.0 / n_vals              # 各 n に対応する刻み幅 h の配列
27 errors = []
28
29 # 4. 各 n について誤差を計算
30 for n in n_vals:
31     calc_val = simpson(f, 0.0, 1.0, n)  # シンプソン法で計算
32     err = np.abs(calc_val - true_val)   # 絶対誤差を計算
33     errors.append(err)                  # リストに追加
34
35 # 5. 両対数プロット (Log-Log Plot) の描画
36 plt.figure(figsize=(8, 5))
37 plt.loglog(h_vals, errors, 'o-', label="Simpson's Error")
38
39 # 理論上の勾配 O(h^4) のガイド線
40 plt.loglog(h_vals[0:8], (h_vals[0:8]**4) * 0.1, '--',
41            label=r"Theoretical $O(h^4)$", color='gray')
42
43 plt.xlabel(r"Step size $h$")
44 plt.ylabel("Absolute Error")
45 plt.title("Truncation Error vs. Rounding Error")
46 plt.grid(True, which="both", ls="--")
47 plt.gca().invert_xaxis()  # 右から左へ h が小さくなるように軸を反転 (nが増える向き)
48 plt.legend()
49 plt.tight_layout()
50 plt.show()
```

このコードを実行して得られるグラフ（両対数プロット）では，右側（h が大きい領域）から左側（h が小さい領域）へ進むにつれて，最初誤差は理論通りの傾き h^4 で急激に減少していきます。しかし，刻み幅が $h \approx 10^{-4}$ 付近に達したところで点は消えていますがこれは `err=0.0` となったためです。さらに h を小さく

してステップ数を増やすと，$h \approx 10^{-6}$ 辺りから累積した丸め誤差によって全体の誤差が増大していく様子が確認できます。

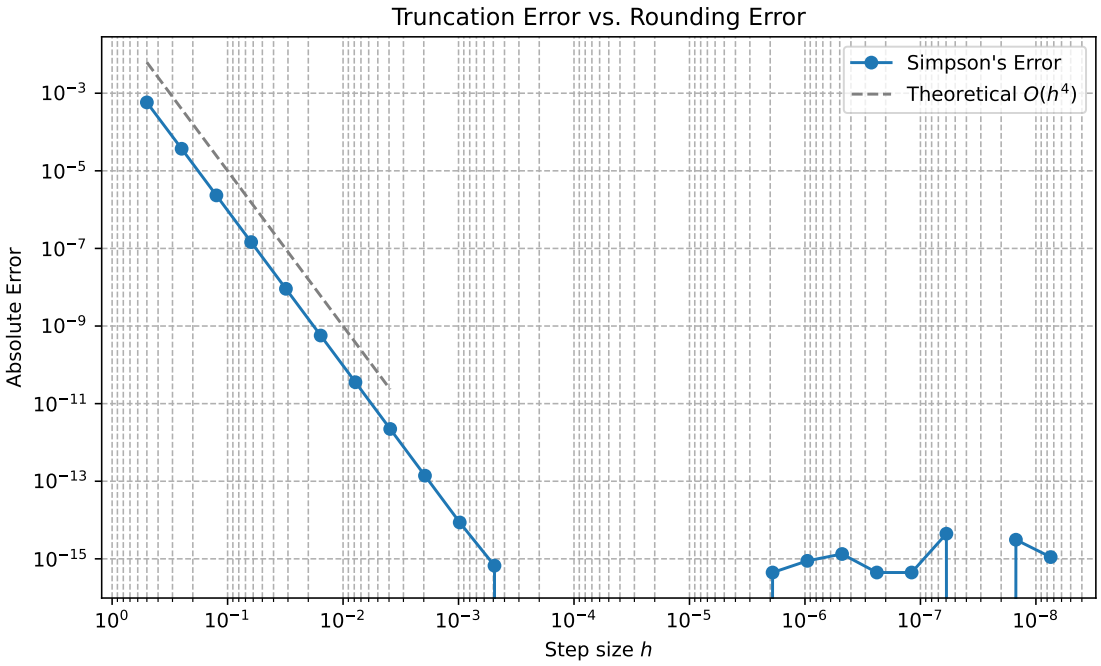


図 29　数値積分における刻み幅と誤差

24.4　練習問題

24.4.1　練習問題 1：円周率を導く数値積分の誤差

定積分

$$I = \int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

について，前節のコード（シンプソン法）と同様の実験を行い，分割数 n（刻み幅 h）の変化に対する絶対誤差の振る舞いを両対数プロットで描画しなさい。

24.4.2　練習問題 2：無限区間での数値積分（区間打ち切り vs 変数変換）

Fresnel 積分などに現れる無限区間の積分

$$I = \int_0^\infty \sin(x^2)\,dx = \sqrt{\frac{\pi}{8}} \approx 0.626657068657750$$

をコンピュータで計算する際，主に以下の 2 つのアプローチが存在します。

1. **手法 A（区間の切り捨て）：**
   十分大きな上端 L で積分を打ち切り，有限区間 $[0, L]$ の積分 $I \approx \int_0^L \sin(x^2)\,dx$ として計算する。
2. **手法 B（置換積分による有限区間化）：**

$x = \tan t$ と置換することで，無限区間 $[0, \infty)$ を有限区間 $\left[0, \frac{\pi}{2}\right)$ に変換して計算する。

$$I = \int_0^{\frac{\pi}{2}} \sin\left(\tan^2 t\right) \cdot \frac{1}{\cos^2 t}\, dt$$

【課題】

1. 手法 A において，$L = 10, 20, 50$ と大きくしたときに厳密解との誤差がどう変化するか確認しなさい。
2. 手法 B を用いて，区間 $\left[0, \frac{\pi}{2} - \varepsilon\right]$（例：$\varepsilon = 10^{-8}$）でシンプソン法を適用し，手法 A との精度の違いや，端点 $t \to \frac{\pi}{2}$ 付近で被積分関数が激しく振動・増大する挙動について考察しなさい。

25　精度保証付き数値計算

25.1　誤差付きで計算を行う考え方

倍精度浮動小数点数（`float64`）の計算では，各ステップで 10^{-16} 程度の丸め誤差が生じるため，これ以上の精度で計算できないばかりか，数値積分のように刻み幅を細かくすればするほど丸め誤差が累積・増大していきます。また，計算機から得られた 1 つの数値結果が「どこまでの桁数で数学的に正しいのか」を証明することも容易ではありません。

そこで自然に思いつくのが，単一の数値 x で計算するのではなく，誤差の範囲を抱えたまま計算を進めるというアイデアです。

例えば，円周率の値を無限の桁数で正確に表すことはできませんが，$3.14 < \pi < 3.15$ と表現しておけば，これは数学的に完全な真実です。同様に $1.41 < \sqrt{2} < 1.42$ であるため，これらの和や積は

$$3.14 + 1.41 < \pi + \sqrt{2} < 3.15 + 1.42$$
$$3.14 \times 1.41 < \pi \times \sqrt{2} < 3.15 \times 1.42$$

となるはずであり，次の不等式（保証区間）を得ます。

$$4.55 < \pi + \sqrt{2} < 4.57, \qquad 4.4274 < \pi \times \sqrt{2} < 4.4730$$

和に比べて積の方が誤差の幅（不確実さ）が大きくなることや，小数第 3 位以降の細かな積の計算は実質的に無駄になっていることがわかります。

数学的には，区間 $[a, b]$ と $[c, d]$ に含まれる数の和は区間 $[a + c, b + d]$ に含まれます。また，a, b, c, d がすべて正であれば，積は区間 $[ac, bd]$ に含まれます。実際の計算機で端点 ac や bd を計算する際にも丸め誤差が伴いますが，ac の方は下方に丸め（切り下げ），bd の方は上方に丸める（切り上げる）ことで，真の値を絶対に含む区間を構成できます

数を $[a, b]$ のように区間の両端で表現して計算する「区間演算」と，中心 m と 誤差半径 r を用いて

$$m \pm r \quad \left(\text{すなわち集合として } B(m, r) = \{x \in \mathbb{R} \mid |x - m| \leq r\}\right)$$

のように「中心値 ± 半径」の形で表現して計算する「ボール演算」があります。この表現にしておくと，「現在の推定値 m」と「保証された誤差の上限 r」がひと目で分かり，Arb ライブラリなどでも非常に扱いやすくなります。

25.2　対向丸めと計算機での実装

まずは，手動で区間演算を行ってみましょう。区間の上限や下限を足し算・掛け算する際，その計算自体にも浮動小数点数の丸め誤差が乗ってしまうため，それを厳密に制御する必要があります。そこで，プロセッサ（CPU）があらかじめ備えている「丸めモードの変更機能（対向丸め：Directed Rounding）」を利用します。

- **下限の計算：** 常に $-\infty$ 方向（切り捨て）に丸める（下向き丸め）
- **上限の計算：** 常に $+\infty$ 方向（切り上げ）に丸める（上向き丸め）

このように計算機へ指示することで，計算途中で生じる丸め誤差によって，真の値が保証区間の外へはみ出してしまう可能性を完全に排除します。

Python の標準ライブラリである `decimal` モジュールを使うと，この「丸めモードを制御して真値を挟み込む」挙動を手軽に実験できます[*19]。

```python
from decimal import Decimal, getcontext, ROUND_FLOOR, ROUND_CEILING

# 違いが分かりやすいよう，有効桁数をあえて 4 桁に制限
getcontext().prec = 4

# 下限の計算（常に下向きに丸める）
getcontext().rounding = ROUND_FLOOR
low = Decimal(1) / Decimal(3)    # 0.3333

# 上限の計算（常に上向きに丸める）
getcontext().rounding = ROUND_CEILING
high = Decimal(1) / Decimal(3)   # 0.3334

print(f"真の 1/3 は確実にこの区間にある: [{low}, {high}]")
```

実行結果の例

```
真の 1/3 は確実にこの区間にある: [0.3333, 0.3334]
```

このような 10 進数での厳密な数値表現や丸め制御は，精度保証付き数値計算の基礎となるだけでなく，金融システム（お金の計算）など 10 進数での正確な桁数が厳格に求められる現場でも幅広く活用されているそうです。

25.3　ライブラリ Arb の特徴と導入

中心値 + 誤差半径（エラーバー）による高度なボール演算（Ball Arithmetic）を高速かつ厳密に実装した C 言語の代表的なライブラリが Arb です[*20]。

Arb には，主に以下のような強力な特徴があります。

- **任意精度（任意桁数）のサポート：** 必要に応じて 100 桁，1000 桁といった任意の精度でボール演算を行えます。
- **自動的な誤差の厳密追跡：** 四則演算だけでなく，$\sin, \cos, \exp, \log$ や定積分・特殊関数に至るまで，すべての計算プロセスで生じる丸め誤差や打ち切り誤差を自動的に「半径（エラーバー）」へ反映させます。
- **超高速な C 言語実装：** 数論・代数計算のライブラリ FLINT をベースにしており，他の区間演算ライブラリに比べて圧倒的に高速動作します（AI の受け売りですが）。

Python では `python-flint`[*21] というパッケージを介して Arb の強力なボール演算機能を利用できるた

[*19] `Decimal` は 10 進数で正確に計算を行うための Python のデータ型です。通常の `float`（2 進数）のまま計算すると `decimal` モジュールの丸めモード設定が適用されないため，`Decimal(1)` や `Decimal(3)` のように `Decimal` オブジェクトに変換して演算を行います。なお，小数の場合は `float` を経由するとその時点で丸め誤差が入るため，`Decimal("2.15")` のように文字列として渡すのが鉄則です。

[*20] Arbitrary-precision ball arithmetic の略。現在は FLINT プロジェクトの一部として統合・開発されています。

[*21] FLINT: Fast Library for Number Theory

め，あらかじめインストールしておく必要があります。

- **Mac / Windows（Anaconda, 標準 Python など）：**
  ターミナルまたはコマンドプロンプトで以下を実行してください。

```
pip install python-flint
```

- **Linux（Ubuntu/Debian）や OS 管理下の環境でエラーが出る場合：**

```
pip install python-flint --break-system-packages
```

  を試すか，venv 等の仮想環境を作成して実行してください。

25.4　Arb による誤差ボールの表現と演算

Arb ライブラリでのボール演算の基本的な扱い方を試してみましょう。ここでは，誤差の広がり方（ボール半径の変化）が分かりやすいよう，内部の仮数部精度をあえて 16 ビット（10 進数で有効数字 4〜5 桁程度）の低精度に設定して計算を行ってみます。

```
from flint import arb, ctx

# あえて低精度（16 ビット）に設定
ctx.prec = 16    # ctx は計算環境（context）の設定モジュール

# 16 ビット精度での円周率 pi の包含ボール作成
p = arb.pi()
print("pi =", p)

# 文字列からの数値の定義（16 ビット精度へ丸められる）
x = arb("0.7")
print("x  =", x)

# 中心値と誤差半径を指定したボールの定義
a = arb("1.234", "0.001")  # 1.234 +/- 0.001
b = arb("5.678", "0.001")  # 5.678 +/- 0.001
print("a*b =", a*b)
```

実行結果の例

```
pi  = [3.142 +/- 4.29e-4]
x   = [0.7000 +/- 6.11e-6]
a*b = [7.0 +/- 0.0138]
```

出力結果の 1 行目にある `[3.142 +/- 4.29e-4]` は，真の円周率 π が

$$\pi \in [3.142 - 4.29 \times 10^{-4},\ 3.142 + 4.29 \times 10^{-4}]$$

という閉区間に確実に含まれている（包み込まれている）ことを意味しています。

また，2 行目の `x = arb("0.7")` では，10 進数の 0.7 を 2 進数（16 ビット精度）で正確に表現できないため，読み込みの段階で生じる表現誤差（丸め誤差）が自動的に半径 `+/- 6.11e-6` として保持されています。

さらに 3 行目の積 `a*b` では，元の半径（各 `0.001`）と掛け算自体の丸め誤差が合成され，結果の半径が `0.0138` へと広がる様子が観察できます。

Arb の優れた点は，`float64` の精度限界（約 16 桁）に縛られず，必要に応じて任意の精度（100 ビット，200 ビットなど）に桁数を伸ばしながら，末尾に発生する極小の丸め誤差をエラーバー（誤差半径）として保持し続けられる点にあります。

```
from flint import arb, ctx

# 内部の計算精度を 200 ビット（約 60 桁）に設定
ctx.prec = 200

# 200 ビット精度での pi の包含ボール作成
p = arb.pi()
print("pi (200bit) =", p)

# 200 ビット精度での sqrt(123) の包含ボール作成
x = arb("123")
print("sqrt(123) (200bit) =", x.sqrt())
```

実行結果の例

```
pi (200bit) = [3.14159265358979323846264338327950288419716939937
  51058209749 +/- 4.58e-59]
sqrt(123) (200bit) = [11.090536506409417162051600102609932918 46
  3376742454020022877 +/- 3.21e-58]
```

このように，精度を 200 ビットまで上げると，有効数字が約 60 桁まで一気に拡大します。その巨大な桁数の末尾に `+/- 4.58e-59` や `+/- 3.21e-58` といったエラーバーがあることからも信頼できる精度で値が得られていることが確認できます。

25.5　Arb による高度な計算とアルゴリズムの工夫

実際に複雑な関数や多項式を計算しようとするとき，計算ステップ数に応じて誤差範囲（ボール半径）が増大したり，計算の順序によって結果の精度が大きく変わったりすることがあります。

例えば，多項式の計算においては，簡単な工夫で計算量と丸め誤差の増加を抑えることができます。2 次関数 ax^2+bx+c をそのまま計算する場合，演算回数は (乗算, 加算) $=(3,2)$ ですが，

$$ax^2+bx+c \;\to\; (ax+b)x+c$$

のように変形して計算すると (乗算, 加算) $=(2,2)$ となり，乗算が 1 回少なくて済みます。同様に 3 次式 ax^3+bx^2+cx+d を直接計算すると (乗算, 加算) $=(6,3)$ ですが，

$$((ax+b)x+c)x+d$$

のように変形すると (乗算, 加算) $=(3,3)$ となります。一般に，n 次多項式 $a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$ を入れ子状に変形して計算することで，演算回数をわずか (乗算, 加算) $=(n,n)$ に抑えることができます。このような多項式の効率的な計算手法をホーナー法（Horner's method）と呼びます。

Arb には，こうした多項式評価の工夫をはじめとする高度な精度保証付きアルゴリズムが初めから内蔵されています。そのため，ユーザーが複雑なアルゴリズムを自作しなくても，組み込み関数を呼び出すだけで最適化された精度保証計算が行われます。

以下は，多項式評価や組み込み関数（sin, exp など）の計算例です。

```
from flint import arb, arb_poly, ctx

# 低精度（16 ビット）で誤差の蓄積の違いを観察
ctx.prec = 16
x = arb("6.53")
print("x            :", x)

```

```
# 1. 多項式 P(x) = x^3 - 3x^2 + 3x - 1 の計算比較
# (A) 単純に展開形のまま直接計算
p_direct = x**3 - 3*x**2 + 3*x - 1

# (B) Arb の多項式クラス (arb_poly) を用いた評価
# 係数リスト: [定数項, 1次項, 2次項, 3次項]
poly = arb_poly([-1, 3, -3, 1])  # -1 + 3x - 3x^2 + x^3

# 【修正箇所】poly(x) と関数呼び出しすることで内部で最適に評価されます
p_flint = poly(x)

print("直接計算     :", p_direct)
print("Arb(ホーナー):", p_flint)

# 2. 組み込み関数 (sin, exp) の精度保証計算
s = x.sin()
e = x.exp()

print("sin(x)       :", s)
print("exp(x)       :", e)
```

実行結果の例

```
x            : [6.530 +/- 5.88e-5]
直接計算     : [169.1 +/- 0.0292]
Arb(ホーナー): [169.1 +/- 0.0192]
sin(x)       : [0.2443 +/- 7.52e-5]
exp(x)       : [685.4 +/- 0.0421]
```

出力結果を比べると，まったく同じ多項式であっても，単純に展開して計算した **直接計算**（誤差半径 ± 0.0292）に比べ，Arb の多項式機能 `poly(x)` を用いた計算（誤差半径 ± 0.0192）の方が，誤差半径が約 34% 小さく抑えられていることがわかります。

また，$\sin(x)$ や $\exp(x)$ などの組み込み関数においても，テイラー展開等の打ち切り誤差と各ステップの丸め誤差がすべて厳密に評価され，真値を包含するボールとして正しく出力されていることが確認できます。`exp(x)` では誤差は `0.0421` とそれなりにありますが，相対誤差は $0.0421/685.4 \sim 6.14 \times 10^{-5}$ なので，`sin(x)` の誤差とほぼ変わりません。

25.6　Arb を用いた数値積分の精度保証

シンプソン法を用いて定積分

$$I = \int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

から円周率 π を計算する際，標準的な倍精度浮動小数点数（`float64`）では 10^{-15} 程度の精度限界壁にぶつかります。そこで，Arb ライブラリを用いて「丸め誤差」と「打ち切り誤差」の両方を追跡しながら，数学的に厳密な精度保証付きで計算してみましょう。

前節で確認したように，刻み幅 $h = 1/n$ に対するシンプソン合成公式は

$$I_{\text{simp}} = \frac{h}{3}\left[f(x_0) + 4\sum_{k=1(\text{奇数})}^{n-1} f(x_k) + 2\sum_{k=2(\text{偶数})}^{n-2} f(x_k) + f(x_n) \right]$$

であり，理論上の打ち切り誤差の上限は

$$E_{\text{simp}} \le \frac{b-a}{180}\max_{x\in[a,b]} |f^{(4)}(x)|\, h^4$$

で与えられます。被積分関数の 4 階導関数 $f^{(4)}(x) = \frac{24(5x^4-10x^2+1)}{(1+x^2)^5}$ の絶対値は区間 $[0,1]$ 内の $x=0$ で最大値 24 をとるため，打ち切り誤差の上限は

$$E_{\text{simp}} \le \frac{24}{180} h^4$$

と評価できます。

それでは，Arb を用いて I_{simp} を計算し，打ち切り誤差ボール $B(0, E_{\text{simp}})$ を合体（加算）させることで，真の π を確実に含む保証区間を構成してみましょう。

まずは計算精度を 128 ビット，分割数を $n = 1000$（したがって $h = 1/1000$）として実行します。

```
from flint import arb, ctx

ctx.prec = 128

n = 1000  # 分割数（偶数）
h = arb(1) / arb(n)

def f(x):
    return arb(4) / (arb(1) + x*x)

# 1. 端点 f(x_0) + f(x_n)
s = f(arb(0)) + f(h * n)

# 2. 奇数項 (k = 1, 3, ..., n-1)
s_odd = sum(f(h * i) for i in range(1, n, 2))

# 3. 偶数項 (k = 2, 4, ..., n-2)
s_even = sum(f(h * i) for i in range(2, n, 2))

# シンプソン合成公式（この時点での半径は「計算機の丸め誤差」のみ）
I_simp = (h / arb(3)) * (s + arb(4)*s_odd + arb(2)*s_even)

# 理論誤差の上限 E_max
E_max = (arb(24) / arb(180)) * (h**4)

# 【重要】丸め誤差に打ち切り誤差を合体させた、完全な保証付き積分値
I_guaranteed = I_simp + arb(0, E_max)

print("シンプソン計算値      :", I_simp)
print("Arb の真の pi         :", arb.pi())
print("理論誤差の上限        :", E_max)
print("精度保証付き積分の値 :", I_guaranteed)

# 真の pi がこの保証区間の中に【100%入っているか】の判定
print("精度保証付きの積分は真の pi を含むか？ :", arb.pi() in I_guaranteed)
```

実行結果の例

```
シンプソン計算値      : [3.14159265358979323842296084359728517 +/- 4.43e-36]
Arb の真の pi         : [3.14159265358979323846264338327795028842 +/- 1.06e-38]
理論誤差の上限        : [1.33333333333333333333333333333333333333e-13 +/- 5.61e-51]
精度保証付き積分の値 : [3.141592653590 +/- 3.41e-13]

精度保証付きの積分は真の pi を含むか？ : True
```

シンプソン法の計算値自身（I_simp）を見ると，前節で学んだ誤差の相殺効果により，実際には小数点以下

20 桁近くまで正しく一致していることが分かります。しかし，数学的証明としての精度保証を与えるために理論上限 $E_{\max} \approx 1.33 \times 10^{-13}$ をボール半径として足し合わせた結果，得られる保証区間は `[3.141592653590 +/- 3.41e-13]` となり，保証できる桁数は小数第 11 桁程度に止まります。これは理論誤差評価が「あらゆる関数における最悪値」を見積もっているためであり，厳密な証明を得るための代償と言えます。

初めから「数学的保証付きで小数点以下 20 桁まで正しい」と言い切るためには，理論誤差の項を 10^{-20} 以下まで小さくする必要があります。そのためには分割数を $n = 80000$ 程度に設定して計算します。実行結果は次のようになります：

実行結果の例

```
シンプソン計算値      : [3.14159265358979323846262643383279351 +/- 6.46e-34]
Arb の真の pi         : [3.14159265358979323846262643383279502842 +/- 1.06e-38]
理論誤差の上限        : [3.2552083333333333333333333333333333333e-21 +/- 9.60e-59]
精度保証付き積分の値 : [3.14159265358979323846 +/- 5.90e-21]

精度保証付きの積分は真の pi を含むか？ : True
```

25.7　練習問題

積分

$$\int_{-\infty}^{\infty} e^{-x^2} \sin(x^2) dx$$

を 10^{-10} の精度で厳密に計算するプログラムを作りなさい。（ただし，有限区間の積分にしてその積分にはシンプソンの公式を使うこと）

26　乱数とモンテカルロ法

26.1　乱数と擬似乱数

乱数とは，サイコロの目のように何らかの確率分布に従いつつも，予測可能なパターンの存在しない数列のことです。乱数は，物理現象や社会現象のような乱雑さ[*22]を含むシステムを記述・シミュレーションするためだけでなく，決定論的な量（定積分や図形の面積など）の近似値を計算するためにもしばしば用いられます。

コンピュータのように決定論的に動作する機械では，原理的に「真の乱数」を作り出すことはできません。しかし，計算によって出力された数値列であっても，実用上は乱数と見なして差し支えないような数列を生成することは可能です。このような，アルゴリズムによって計算生成される乱数を**擬似乱数（pseudorandom number）**と呼びます。

Python の標準ライブラリ（`random` モジュール）で採用されている擬似乱数生成器は，**メルセンヌ・ツイスター（Mersenne Twister）**と呼ばれるアルゴリズムです。このアルゴリズムは $2^{19937}-1$ という非常に長い周期を持ち，実用上十分な一様性を備えながら高速に生成できるのが特徴で，1997 年に日本人研究者（松本眞氏・西村拓士氏）によって開発されました。

どんな数列であれば「真の乱数」と呼んでよいのかという問いは，数理科学的に重要な問題です。ある問題にとっては十分にランダムに見える擬似乱数であっても，別の複雑な問題を解く際には規則性が浮き彫りになり，不適切な結果をもたらすこともあります。そのため，高品質な乱数をいかに高速かつ安全に生成するかという問題は，現在でも計算機科学や応用数学における重要な課題の一つです。

26.2　乱数モジュールの基本的な使い方

Python の標準ライブラリ `random` を使うと，様々な確率分布に従う擬似乱数を簡単に生成できます[*23]。

```
import random

x = random.randint(1, 100)  # 1から100までのランダムな整数
print(x)
```

乱数種（シード）と再現性

擬似乱数は内部の計算式によって生成されるため，発進点となる**乱数種（random seed）**を固定すると，常に全く同じ乱数列が再現されます。

- ファイル名：**random_seed.py**

```
import random

# 乱数種を固定する（同じseedからは常に同じ乱数が生成される）
random.seed(0)
print("seed(0):", [random.random() for _ in range(3)])

# もう一度 seed(0) を設定すると全く同じ結果が得られる
random.seed(0)
```

[*22] 乱雑の英訳は random（ランダム）ですが，日本語の「乱（ラン）」と頭の 2 文字が偶然一致しているのが少し不思議。

[*23] いかなる場合でもプログラムのファイル名に `random.py` を用いないでください。この名前のファイルが同じディレクトリ内に存在すると，`import random` を実行した際に標準モジュールではなくその自作ファイルが読み込まれてしまい，エラーの原因となります。

```
print("seed(0):", [random.random() for _ in range(3)])

# seedを指定しない（または省略）と，実行時刻等から自動で設定される
random.seed()
print("時刻依存:", [random.random() for _ in range(3)])
```

実行結果の例

```
seed(0): [0.8444218515250481, 0.7579544029403024, 0.420571580830845]
seed(0): [0.8444218515250481, 0.7579544029403024, 0.420571580830845]
時刻依存: [0.0104079725999231, 0.6052014863172019, 0.0631615144758921]
```

一見すると毎回異なる乱数が出る方が良さそうに思えますが，数値計算において再現性は極めて重要です。実験結果に予想外のエラーや奇妙な挙動が出た際，同じ乱数列を再現できなければ，原因がアルゴリズムのバグなのか単なる確率的な偶然なのかを突き止めることが困難になります[*24]。

一方，セキュリティ（パスワード生成など）では予測不能性が求められるため，実行時刻やマウスの動きなどから種を作ります[*25]。

代表的な乱数関数

random モジュールでよく使われる関数をまとめます。

よく使う乱数関数

- `random.random()`：区間 $[0, 1)$ の一様実数乱数を返す。
- `random.randint(a, b)`：$a \leq N \leq b$ となる整数 N をランダムに返す。
- `random.uniform(a, b)`：区間 $[a, b]$ の一様実数乱数を返す。
- `random.choice(sequence)`：リストやタプル等の要素から無作為に 1 つ選ぶ。
- `random.gauss(mu, sigma)`：平均 μ，標準偏差 σ の正規分布に従う実数乱数を返す。

- 使用例：**random_functions.py**

```
import random

colors = ["red", "green", "blue", "black", "white"]

print("choice :", [random.choice(colors) for _ in range(4)])
print("randint:", [random.randint(1, 6) for _ in range(5)])   # サイコロの目
print("gauss  :", [round(random.gauss(0, 1), 3) for _ in range(4)])
```

実行結果の例

```
choice : ['green', 'white', 'blue', 'red']
randint: [3, 6, 1, 4, 2]
gauss  : [-1.982, 0.288, -0.119, 1.804]
```

【補足：ガウス分布（正規分布）】

平均 μ，標準偏差 σ のガウス分布の確率密度関数 $P(x)$ は以下で定義されます：

$$P(x) := \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

乱数 X がこの分布に従うとは，値が区間 $[a, b]$ に入る確率が $\int_a^b P(x)dx$ となることを意味します。

[*24] 幽霊の目撃情報のように，再現できない現象の検証は極めて困難です。

[*25] 物理的に「真の乱数」を作りたい場合は，放射性核種の崩壊など量子力学的な確率現象を利用したハードウェア乱数生成器が用いられます。

26.3　モンテカルロ法 1 ：円周率の近似

乱数を用いて面積や定積分の値を近似計算する手法を**モンテカルロ法（Monte Carlo method）**と呼びます。ここでは，乱数を使って円周率 π を近似計算してみましょう。

円周率を求めるアイデアは次の通りです：2 次元の正方形領域 $[-1,1]\times[-1,1]$（面積 4）の中に一様に n 個の点をばらまき，そのうち半径 1 の単位円内部（$x^2+y^2<1$）に入った点の数を数えます。円の中に入った点の数を N_{in} とすると，ばらまいた点の総数 n との比率は，領域の面積比（正方形の面積：円の面積 $=4:\pi$）に近似的に等しくなるはずです：

$$\frac{\text{円の中にある点の数 } N_{\text{in}}}{\text{ばらまいた点の数 } n} \approx \frac{\text{円の面積}}{\text{正方形の面積}} = \frac{\pi\times 1^2}{4}$$

したがって，円周率 π は以下のように近似できます：

$$\pi \approx 4\times\frac{N_{\text{in}}}{n}$$

点をばらまく数 n を極限まで大きくすれば，大数の法則によりこの値は真の円周率 π に収束します。

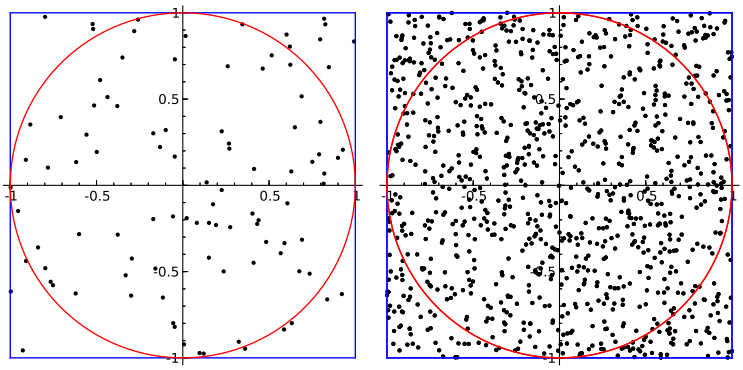


図 30　$[-1,1]^2$ の領域に一様に点をばらまいて円周率を近似する

この手順を Python でプログラムにすると，以下のようになります。

- ファイル名：**pi_monte_carlo.py**

```
import random

random.seed(0)

def count_inside_circle(n):
    """n個の点を[-1,1]x[-1,1]にばらまき，円の中に入った点の個数を返す関数"""
    p = 0  # 円の中に入った点のカウンタ
    for _ in range(n):
        x = random.uniform(-1, 1)  # -1から1までの一様実数乱数
        y = random.uniform(-1, 1)
        if x**2 + y**2 < 1.0:      # 単位円の内部にあるか判定（※ x^2 ではなく x**2）
            p += 1
    return p

n = 100
pi_approx = 4.0 * count_inside_circle(n) / n
print(f"n = {n} のときの円周率の近似値: {pi_approx}")
```

実行結果の例

```
n = 100 のときの円周率の近似値: 3.28
```

サンプル数 $n = 100$ では 3.28 となり，真の値（3.14159265...）に対してかなり大雑把な近似しか得られていないことが分かります。

次に，試行回数 n を増やしていったときに精度がどのように向上するかを見てみましょう。

```python
import random
random.seed(0)

total_count = 0      # 累積の全サンプル数
inside_count = 0     # 累積の円内サンプル数
oneloop = 1000000    # 1ループあたりに追加する点の数

while True:
    for _ in range(batch_size):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if x**2 + y**2 < 1.0:
            inside_count += 1
    total_count += batch_size
    pi_approx = 4.0 * inside_count / total_count
    error = abs(pi_approx - 3.141592653589793)

    # 出力
    print(f"{total_count} | {pi_approx} | {error}", end="")
```

実行結果の例

```
1000000 | 3.141132 | 0.00046065358979330284
2000000 | 3.140568 | 0.0010246535897930897
...
99000000 | 3.1414724444444446 | 0.00012020914534849325
100000000 | 3.1414602 | 0.00013245358979308008
101000000 | 3.1414743366336633 | 0.00011831695612984916
...
```

この実行結果を見てわかるように $n = 10000000$ 個もの点をばらまいても，まだ小数第 4 位（0.0001 の桁）あたりでフラフラ揺れていることが分かります。

確率論（中心極限定理）に基づくと，モンテカルロ法による標準誤差は，サンプル数 n に対して $\mathcal{O}(1/\sqrt{n})$ の割合でしか減少しません。つまり，精度を 1 桁（1/10）上げるためには，サンプル数を 100 倍に増やさなければならないということを意味しています。

1 次元や 2 次元の積分・面積計算において，モンテカルロ法は以前学んだ台形公式やシンプソンの公式に比べて圧倒的に効率が悪く，高精度な計算には向きません。

しかし，モンテカルロ法は高次元（10 次元や 100 次元など）の多重積分を行う場合に真価を発揮します。決定論的な数値積分（シンプソン法など）は次元が増えると計算量が爆発するいわゆる「次元の呪い」に直面しますが，モンテカルロ法は次元が増えても誤差の減少スピードが $\mathcal{O}(1/\sqrt{n})$ のまま変わらないという強力な性質を持っています。

26.4　モンテカルロ法 2：4 次元球の体積の計算

2 次元の円周率に続き，モンテカルロ法を用いて直感的なイメージの難しい 4 次元単位球の体積を計算してみましょう。

4 次元単位球 B_4 とは，4 次元実空間 $\mathbb{R}^4$ における以下の集合です：

$$B_4 = \{(x_1, x_2, x_3, x_4) \in \mathbb{R}^4 \mid x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 1\}$$

領域 $[-1, 1]^4$（一辺の長さ 2 の 4 次元超正方体，体積 $2^4 = 16$）の中に一様に n 個の点をばらまき，そのうち B_4 の内部に入る点の数をカウントします。

点の数の比率は，体積の比率（$16 : |B_4|$）に近似的に等しくなるため，

$$\frac{B_4 \text{ の中にある点の数}}{\text{ばらまいた点の数}} \approx \frac{B_4 \text{ の体積}}{16}$$

より，B_4 の体積は以下のように計算できます：

$$B_4 \text{ の体積} \approx 16 \times \frac{B_4 \text{ の中にある点の数}}{\text{ばらまいた点の数}}$$

これをプログラムにすると，次のようになります。

- ファイル名：**ball4d.py**

```python
import random

random.seed(0)

def count_inside_ball4d(n):
    """n個の点を[-1,1]^4にばらまき，4次元単位球の内部に入る点の個数を返す関数"""
    p = 0  # カウント用の変数
    for _ in range(n):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        z = random.uniform(-1, 1)
        w = random.uniform(-1, 1)

        # 4次元の距離の2乗が1未満か判定（※ x**2 や x*x を使用する）
        if x**2 + y**2 + z**2 + w**2 < 1.0:
            p += 1
    return p

n = 20000  # サンプル点の個数
vol_approx = 16.0 * count_inside_ball4d(n) / n
print(f"4次元単位球の体積の近似値 (n={n}): {vol_approx}")
```

実行結果の例

```
4 次元単位球の体積の近似値 (n=20000): 4.9088
```

解析的に求まる 4 次元単位球の厳密な体積 $|B_4|$ は以下の通りです：

$$|B_4| = \frac{\pi^2}{2} \approx 4.934802$$

得られた結果を真の値と比較してみると，やはり大まかに近い値が得られていることが確かめられます。

一般に d 次元単位超球の体積は $V_d = \frac{\pi^{d/2}}{\Gamma(d/2+1)}$ で表され，次元 d が大きくなると超正方体 $[-1, 1]^d$（体積 2^d）に対して超球の体積の割合は急速にゼロへと近づきます（高次元の「角」の部分に体積のほとんどが集中するためです）。このような高次元空間の複雑な領域に対しても，モンテカルロ法なら点の判定条件を変えるだけで容易に体積を近似できるのが大きな強みです。

26.5　モンテカルロ法 3：関数の積分（サンプル平均法）

前節までの「円周率の近似」や「4 次元球の体積」では，領域の中に乱数を打ち込み，「領域の内側か外側か（Hit か Miss か）」という 0 か 1 かの情報だけをカウントして面積・体積を求めていました。このようなアプローチをヒット・オア・ミス法（Hit-or-Miss method）と呼びます。これは，直感的でわかりやすい反面，関数の具体的な値（高さ）の情報を捨ててしまっているため，効率（誤差の小ささ）の面では不利になります。

そこで本節では，関数値そのものの平均をとることでより効率的に定積分 $\int_a^b f(x)\,dx$ を計算するサンプル平均法（Sample Mean method）と呼ばれるモンテカルロ法を学びます。

$a < b$ とします。まず，区間 $[a,b]$ に一様乱数 $X_j\,(j = 1, 2, \ldots, N)$ を発生させます。積分平均値の定理より，関数 $f(x)$ の区間 $[a,b]$ における平均値は $\frac{1}{b-a}\int_a^b f(x)\,dx$ です。大数の法則によれば，サンプル数 N を大きくしたとき，この平均値はサンプルの関数値の算術平均 $\frac{1}{N}\sum_{j=1}^{N} f(X_j)$ に確率収束します：

$$\frac{1}{b-a}\int_a^b f(x)\,dx = \lim_{N\to\infty}\frac{1}{N}\sum_{j=1}^{N} f(X_j)$$

したがって，目的の定積分 $\int_a^b f(x)\,dx$ は以下のように近似計算できます：

$$\int_a^b f(x)\,dx \approx \frac{b-a}{N}\sum_{j=1}^{N} f(X_j)$$

中心極限定理（Central Limit Theorem）により，この近似における標準誤差（Standard Error）はサンプル数 N に対して $\mathcal{O}(1/\sqrt{N})$ の割合で減少します[*26]。

26.5.1　例題：$\int_0^1 \sqrt{1-x^2}dx$ による円周率の計算と手法の比較

サンプル平均法の具体例として，以下の定積分を計算してみましょう：

$$I = \int_0^1 \sqrt{1-x^2}\,dx = \frac{\pi}{4} \quad (\approx 0.785398)$$

区間 $[0,1]$ の一様乱数 $X_j\,(j = 1, \ldots, N)$ による，サンプルの平均値は以下のようになります：

$$I \approx \frac{1}{N}\sum_{j=1}^{N}\sqrt{1-X_j^2}$$

- ファイル名：**pi_sample_mean.py**

```
import random
import math

random.seed(0)

def integrate_sample_mean(n):
    """ サンプル平均法による integral_0^1 sqrt(1-x^2) dx の計算"""
    total = 0.0
    for _ in range(n):
```

[*26] 確率論における「重複対数の法則」によれば，確率 1 で成り立つ最悪誤差の上限は $\mathcal{O}\left(\sqrt{\frac{\log\log N}{N}}\right)$ となりますが，$\log\log N$ は計算機で扱う範囲の N（例: $N = 10^{12}$ でも $\log\log N \approx 3.32$）では実質的に定数と見なせるため，実用上は $\mathcal{O}(1/\sqrt{N})$ として扱います。

```
        x = random.uniform(0, 1)
        total += math.sqrt(1.0 - x**2)
    return total / n

n = 100000
I_approx = integrate_sample_mean(n)
pi_approx = 4.0 * I_approx

print(f"サンプル平均法 (n={n}): {pi_approx:.6f}")
print(f"誤差: {abs(pi_approx - math.pi):.6f}")
```

実行結果の例

```
サンプル平均法 (n=100000): 3.142079
誤差: 0.000486
```

ヒット・オア・ミス (HoM) 法とサンプル平均 (SM) 法による計算精度を比較してみましょう。まず，HoM 法は $x^2 + y^2 < 1$ を判定しているだけなのに対し，SM 法は平方根の計算が必要となりますが，平方根の計算は乱数生成と比べても重たくないので，その違いを考慮する必要はありません。

Hom 法での分散を計算してみましょう。$[-1, 1]^2$ に一様に打たれた点が円の中に入る確率 p は $p = \pi/4 \approx 0.785398$ です。確率 p で 1，$1 - p$ で 0 をとる確率変数（ベルヌーイ試行）の分散は次のようになります：

$$\sigma^2_{\mathrm{HoM}} = p(1 - p) \approx 0.168547.$$

一方，SM 法の分散は

$$\sigma^2_{\mathrm{SM}} = \mathbb{E}[f(X)^2] - (\mathbb{E}[f(X)])^2$$

から計算されます。

$$\mathbb{E}[f(X)^2] = \int_0^1 \left(\sqrt{1 - x^2}\right)^2 dx = \frac{2}{3} \approx 0.666667$$
$$\mathbb{E}[f(x)] = \int_0^1 \sqrt{1 - x^2} dx = \frac{\pi}{4} \sim E[f(X)] \approx 0.785398$$

なので

$$\sigma^2_{\mathrm{SM}} = 0.666667 - (0.785398)^2 \approx 0.049817$$

となります。したがって，同じサンプル数なら約

$$\sigma_{\mathrm{HoM}}/\sigma_{\mathrm{SM}} \approx 1.84$$

倍だけ SM 法の方が高精度になります。つまり SM 法の誤差は HoM 法の誤差の約 1/1.84 倍であると期待できます。モンテカルロ法での収束の早さはサンプル数の平方根の逆数に比例するので，SM 法で得られた精度と同じだけの精度で HoM 法で計算しようと思えば $\sigma^2_{\mathrm{HoM}}/\sigma^2_{\mathrm{SM}} \approx 3.4$ 倍の試行回数が必要ということになります。

多重積分への拡張

モンテカルロ積分の最大の強みは，高次元の多重積分へ直ちに拡張できる点です。例として，n 次元単位超立方体 $[0, 1]^n$ 上の多重積分

$$I = \int_0^1 \cdots \int_0^1 f(x_1, \ldots, x_n)\, dx_1 \ldots dx_n$$

を考えます。

領域 $[0,1]^n$ 内に一様に発生させた N 個の n 次元ベクトルを $\boldsymbol{X}_j = (X_{j,1}, X_{j,2}, \ldots, X_{j,n})\,(j = 1, \ldots, N)$ とするとき，この多重積分は単に以下のように近似できます：

$$I \approx \frac{1}{N} \sum_{j=1}^{N} f(X_{j,1}, X_{j,2}, \ldots, X_{j,n})$$

重要なのは，次元 n が 10 次元になろうが 100 次元になろうが，誤差の減少スピードが常に $\mathcal{O}(1/\sqrt{N})$ のまま変化しないという点です。台形公式やシンプソン法などの決定論的手法では，各軸を M 分割すると全体で M^n 回の関数評価が必要となり，次元 n の増加とともに計算量が爆発しますが（次元の呪い），モンテカルロ法はこの問題を回避できる唯一の強力な手法となります。

26.6 練習問題

26.6.1 球の体積の計算

半径 1 の球の $\frac{1}{8}$ の体積（理論値：$V = \frac{1}{8} \times \frac{4}{3}\pi = \frac{\pi}{6} \approx 0.523599$）を求める以下の定積分を考えます：

$$I = \int_0^1 dx \int_0^{\sqrt{1-x^2}} dy\, \sqrt{1 - x^2 - y^2} \tag{23}$$

この積分を，以下の 2 つの方法で計算する Python プログラムを作成し，結果を比較しなさい。

1. **ヒット・オア・ミス法（HM 法）：**
   領域 $[0,1]^3$（体積 1）の中に一様乱数 (X, Y, Z) を発生させ，$X^2 + Y^2 + Z^2 \leq 1$ となる割合から体積を求める。
2. **サンプル平均法（SM 法）：**
   領域 $[0,1]^2$（面積 1）の中に一様乱数 (X, Y) を発生させ，高さを $f(X,Y) = \sqrt{\max(0, 1 - X^2 - Y^2)}$（ただし $X^2 + Y^2 > 1$ のときは 0）として，関数値の平均から体積を求める。

■条件：

- サンプル数はそれぞれ $N = 100000$ とする。
- 乱数シードは `random.seed(0)` で固定すること。
- ファイル名は `MonteCarloSphere.py` とし，実行結果として両手法の近似値と真値（$\pi/6$）との絶対誤差をプリントすること。

26.6.2 具体的な厳密値が値が求まらない関数の積分

$f(x, y, z) = \cos(x - y^2) e^{-x^2 - y^2} / (x^2 + z^2 + 1)$ とする。積分

$$\int_0^1 dx \int_0^2 dy \int_0^3 dz f(x, y, z) \tag{24}$$

をモンテカルロ法により計算するPython のプログラムを作成せよ。ただし，

- サンプル点の個数は 10000 個。
- 端末から実行したときに実行結果は積分の数値のみをプリントするようにする。
- ファイル名は `MonteCalroIntegral.py` とすること。

参考文献

[1] Python 公式ドキュメント https://docs.python.org/ja/3/

[2] Sage チュートリアル https://doc.sagemath.org/html/ja/tutorial/index.html

[3] Paul Zimmermann, Alexandre Casamayou, Nathann Cohen, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, Nicolas M. Thiéry, Erik Bray, John Cremona, Marcelo Forets, Alexandru Ghitza, Hugh Thomas, *Mathematical Computation with SageMath*, https://www.sagemath.org/sagebook/english.html