

# Python基礎講座

---

テキスト

# 目次（1日目）

- **1章** Pythonの概要を理解しよう
- **2章** Pythonの基本的な書き方を理解しよう
- **3章** データの扱い方を理解しよう
- **4章** 変数を理解しよう
- **5章** 配列（リスト・タプル・セット）を理解しよう
- **6章** 連想配列（ディクショナリ）を理解しよう
- **7章** 条件分岐のif文を理解しよう

# 目次 (2日目)

- **8章** 繰り返し処理のfor文を理解しよう
- **9章** 繰り返し処理のwhile文を理解しよう
- **10章** 配列や辞書の繰り返し処理を理解しよう
- **11章** 関数を理解しよう
- **12章** 関数の引数・戻り値を理解しよう
- **13章** 変数のスコープを理解しよう
- **14章** クラスを理解しよう
- **15章** 日付・時刻の処理を理解しよう
- **16章** パッケージ・モジュールについて理解しよう

# 1章 Pythonの概要を理解しよう

---

# 1章 Pythonの概要を理解しよう

| Pythonとはどのような言語なのか、基礎を学びます。

## 本章の目標

- Pythonとはどのような言語なのかを理解すること
- Pythonはどのようなときに使うのかイメージをつかむこと

# 1章 Pythonとは

Pythonは30年以上の歴史を持ちながら、いまだに国内外で高い人気を誇っている言語です。

汎用性が高い言語で、主に以下のような用途に使われています。

- AI（人工知能）・機械学習の開発
- Webアプリの開発
- 組み込みシステム開発（電子機器の制御プログラム）
- スクレイピング（Webデータの抽出）
- IoTの開発

## 先端技術の開発で特に人気が高い言語です

- AIや機械学習はもちろん、電子機器をインターネットに接続する「IoT」の分野でもよく使われる
- さまざまな分野で人気が高いため、しっかり学べば将来のキャリアアビジョンが広がる

# 1章 Pythonの特徴

| Pythonのおもな特徴は、以下の3つです

| ① 習得しやすい

初学者でも比較的簡単に  
習得できる

| ② 需要が高い

求人件数が多く需要が高い

| ③ ライブラリが豊富

優れたライブラリが豊富  
にある

# 1章 特徴① 初学者でも比較的簡単に習得できる

Pythonは数あるプログラミング言語の中でも習得しやすいため、初学者におすすめです

プログラミング言語にはさまざまな文法（コードの書き方）があります：

- 変数
- 条件分岐
- 配列
- 関数

この文法が難しいと、初学者ではつまずいてしまいます。

# 1章 特徴① Pythonの文法はシンプル

## | Pythonの文法はシンプルで理解しやすいものばかりです

- 他の言語に比べて記述するコードが少なくて済む
- 初めてプログラミングを学ぶ方でも、スムーズに学習できる

# 1章 特徴① PythonとC言語の比較

## Python

```
for i in range(0, 5, 1):
    print(i)
```

## C言語

```
int main(void)
{
    int i = 0;
    for(i=0; i<5; i++){
        printf("%d\n", i);
    }
    return 0;
}
```

# 1章 特徴③ 優れたライブラリが豊富にある

| Pythonには優れた「ライブラリ」が豊富にあるため、効率的な開発が可能です

ライブラリとは？

- 簡単に使えるよう部品化された便利なプログラムのこと
- 料理に例えれば、手間をかけずにだしを取れる「即席だし」のようなもの

特にデータサイエンス分野では、優れたPythonライブラリが充実しています。

# 1章 特徴③ 代表的なライブラリ

## scikit-learn

(サイキット・ラーン)

- 機械学習全般のアルゴリズムを簡単に実装できる

## TensorFlow

(テンソルフロー)

- 深層学習（ディープラーニング）のアルゴリズムを簡単に実装できる

## Pythonの優れたライブラリ



**TensorFlow**

機械学習ライブラリ

**scikit-learn**

データ分析ライブラリ

# 1章 Pythonで実現できること

Pythonで実現できることは数多くありますが、特によく使われる分野は以下の2つです

## ① AIや機械学習の開発

モデルの構築に使われる

## ② Webアプリの開発

バックエンド開発で人気

# 1章 実現できること① AIや機械学習の開発

| PythonはAIや機械学習の「モデル」を構築するときに使われる言語です

モデルとは？

- 与えられた入力データから、何らかの結論を出力するプログラムのこと

顔認証モデルの例：

- 入力された顔の画像から特徴的な部分を探す
- これまでの顔パターンと照らし合わせる

# 1章 実現できること① 高度な処理とPython

## 画像処理や統計分析といった高度な処理をプログラムで行う必要があります

- こうした処理をモデル内で実現するときに、Pythonの優れたライブラリが役に立つ
- モデルを構築する前段階でもデータ分析が欠かせない
- すっきりしたコードが書けるPythonは、効率的にデータ分析するうえでも有用

# 1章 実現できること② Webアプリの開発

Pythonはデータサイエンス分野に限らず、Webアプリの開発でも高い人気があります

Webアプリを構成するプログラムは、フロントエンド・バックエンドの2種類です。

## フロントエンド

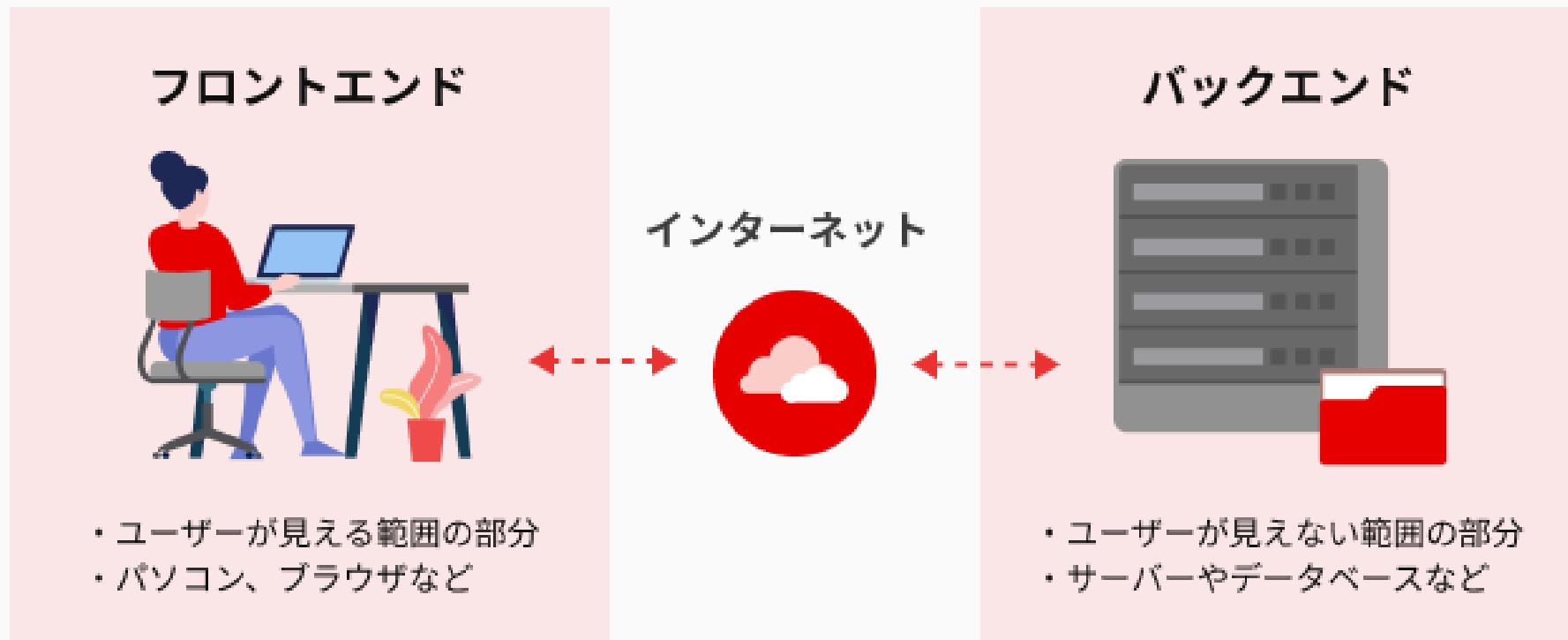
Webサイトやアプリのうち、目に見える部分

## バックエンド

Webサイトやアプリのうち、目に見えない部分  
(ユーザーの個人情報などを保存する  
サーバー側)

# 1章 実現できること② フロントエンド・バックエンド

## フロントエンド・バックエンドのイメージ



# 1章 実現できること② Pythonのフレームワーク

| Pythonには、Webアプリの開発に便利な「フレームワーク」が数多くあります

フレームワークとは？

- よく使われる一連の機能をひとまとめにした枠組みのこと

例：

- アカウント登録やユーザー認証といった機能は、Pythonのフレームワークを使えば少ない労力で実装できる
- 動画共有サービス「YouTube」には、Pythonのフレームワーク「Django」が使われている

## 本章では以下の内容を学習しました

### Pythonとは

- AIをはじめとする幅広い開発分野で使われるプログラミング言語

### Pythonの特徴

- ① 初学者でも比較的簡単に習得できる
- ② 優れたライブラリが豊富にある

# 1章まとめ（続き）

## Pythonで実現できること

- ① データ分析やAI・機械学習の開発
- ② Webアプリの開発

Pythonは汎用性・需要が高く、初学者でも習得しやすい言語です

## 2章 Pythonの基本的な書き方を理解しよう

---

# 2章 Pythonの基本的な書き方を理解しよう

## Pythonの基本的な書き方を学びます

### 本章の目標

- Pythonの書き方の基本ルールを覚えること

### いよいよPythonの本格的な学習をスタートします

- 前章の環境構築で、Python学習の大きな壁は乗り越えた
- 本章では、まずPythonの基本的な書き方を学ぶ

## 2章 書き方の特徴

Pythonの文法はシンプルなものなの、ほかのプログラミング言語と比べて独特です

- 同じく人気言語の「JavaScript」と比較して、書き方の特徴をつかむ
- 本教材で初めてプログラミング言語を学習する人はPythonの書き方のみを参照

## 2章 サンプルコード：JavaScriptの場合

| 年齢が未成年なら「ジュースで乾杯！」、成年なら「お酒で乾杯！」と出力

```
let age = 18; // 年齢

if( age < 20 )
{
    // 未成年の場合
    console.log("ジュースで乾杯！");
}
else
{
    // 成年の場合
    console.log("お酒で乾杯！");
}
```

## 2章 サンプルコード：Pythonの場合

### | 同じプログラムをPythonで書いた場合

```
age = 18 # 年齢

if age < 20:
    # 未成年の場合
    print("ジュースで乾杯！")
else:
    # 成年の場合
    print("お酒で乾杯！")
```

PythonのほうがJavaScriptよりも記述量が少なく、シンプル

## 2章 Pythonの基本ルール

| Pythonの特徴ともいえる基本的なルールは以下の3つです

### | ① セミコロン不要

文の最後にセミコロン  
(:) は不要

### | ② インデント

中かっこ{}で囲わず、イ  
ンデントする

### | ③ コメント

シャープ (#) またはク  
ォーテーションを使う

## 2章 ルール① 文の最後にセミコロン (;) は不要

多くのプログラミング言語だと、文の最後にセミコロン (;) を書く必要があります

JavaScriptの場合：

```
console.log("ジュースで乾杯!");
```

Pythonの場合：

```
print("ジュースで乾杯!")
```

## 2章 ルール① 改行が処理の区切り

### | Pythonでは、処理と処理の区切りに「改行」を用います

- セミコロンの代わりに、改行で処理の区切りを判断する
- 改行を見て「前の処理が終わったんだな」と判断できる
- ほかの言語と混同しないよう、しっかり覚えておく

## 2章 ルール② 中かっこ{}で囲わず、インデントする

```
age = 18 # 年齢

if age < 20:
    # 未成年の場合
    print("ジュースで乾杯！")
else:
    # 成年の場合
    print("お酒で乾杯！")
```

## 2章 ルール② インデントとは

| インデントとは、空白を入れて文の始まりを後ろにずらすことです

- 同じ数のインデントが入った行が、処理のまとまりと認識される
- Pythonのインデントは、半角スペース4つ分で表現されることが多い
- 文が長くなる場合は半角スペース2つ分でも可

## 2章 ルール② JavaScriptとの比較

JavaScriptなど多くのプログラミング言語だと、複数の処理をまとめるとときに中かっこで囲います

JavaScript

```
if( age < 20 )  
{  
    console.log("未成年")  
}
```

中かっこ{}で囲む

Python

```
if age < 20:  
    print("未成年")
```

インデントでまとめる

## 2章 ルール③ コメントはシャープ (#) またはクオーテーションを使う

| コメントとは、プログラム実行時には無視される記述のことです

- プログラムの「メモ書き」として使われる
- 処理を一時的に無効化したいときに、コメント化（コメントアウト）する場合もある

## 2章 ルール③ コメントの書き方

### | Pythonでコメントをつける場合のキーワード

| 種類        | 書き方  |
|-----------|--|
| 1行だけコメント化 | シャープ (#)   |
| 複数行をコメント化 | シングルクオーテーション3つ ("") ではさむ または ダブルクオーテーション3つ ("""") ではさむ |

## 2章 ルール③ 1行コメントの例

| シャープを入れると、その行の終わりまでがコメントとして無視されます

```
age = 18 # 年齢  
  
if age < 20:  
    # 未成年の場合  
    print("ジュースで乾杯!")
```

# 年齢 と # 未成年の場合 がコメント

## 2章 ルール③ 複数行コメントの例

| クオーテーション3つではさむと、複数行をコメント化できます

```
age = 18

if age < 20:
    ...
    # 未成年の場合
    print("ジュースで乾杯!")
    ...
else:
    print("お酒で乾杯!")
```

「'''」の書き出しは、コメントしたい部分のインデントと合わせる

### 本章では以下の内容を学習しました

#### 書き方の基本ルール

- ・ ① 文の最後にセミコロン (;) は不要
- ・ ② 処理のまとめは中かっこ{}で囲わず、インデントでまとめる
- ・ ③ コメントにはシャープ (#) またはクォーテーションを使う

応用レベルのプログラムを書くためには、基本が欠かせません

## 3章 データの扱い方を理解しよう

---

# 3章 データの扱い方を理解しよう

| Pythonで使われるデータの種類や簡単な計算などを学びます

## 本章の目標

- データ型の種類と特徴を理解する
- 数値の計算方法を学ぶ
- 文字列の結合方法を学ぶ

# 3章 データ型とは

| データ型とは、プログラムが扱うデータの種類を表すものです

- データには「数値」や「文字」など、さまざまな種類がある
- 種類ごとに扱い方が変わる
- 例えば数値は足し算ができるが、文字は結合になる

# 3章 よく使われるデータ型

| Pythonでよく使われるデータ型を4つ紹介します

| ① 文字列型 (str)

文字や文章を扱う

| ③ 浮動小数点型 (float)

小数を含む数値を扱う

| ② 整数型 (int)

整数の数値を扱う

| ④ 論理型 (bool)

TrueまたはFalseの値

# 3章 データ型① 文字列型 (str)

## | 文字や文章のデータを表す型です

- strとはstring（ストリング）の略で「ひも」という意味
- 文字が連なっている状態をイメージするとわかりやすい
- シングルクオーテーション（'）またはダブルクオーテーション（"）で囲んで表現する

# 3章 データ型② 整数型 (int)

## 整数の数値データを表す型です

- intとはinteger（インテジャー）の略で「整数」という意味
- 小数点を含まない数値のこと
- 正の数、負の数、ゼロを含む

# 3章 データ型③ 浮動小数点型 (float)

## | 小数を含む数値データを表す型です

- float（フロート）とは「浮動小数点数」という意味
- 小数点以下の値を持つ数値を扱うときに使う
- 科学的な計算や金銭の計算などで重要

# 3章 データ型④ 論理型 (bool)

| TrueまたはFalseの2つの値を持つ型です

- bool (布尔) とはboolean (ブーリアン) の略
- 条件分岐などで使われる
- Pythonでは先頭が大文字でTrueまたはFalseと書く

# 3章 数値を計算してみよう

## | Pythonでは算術演算子を使って計算ができます

- 足し算、引き算、掛け算、割り算などの計算ができる
- 計算に使う記号を算術演算子という
- Pythonの算術演算子は数学の記号と似ているが、一部異なる

# 3章 算術演算子の一覧

| よく使われる算術演算子を紹介します

| 演算子 | 意味     | 例        | 結果       |
|-----|--------|----------|----------|
| +   | 足し算    | $5 + 3$  | 8        |
| -   | 引き算    | $5 - 3$  | 2        |
| *   | 掛け算    | $5 * 3$  | 15       |
| /   | 割り算    | $5 / 3$  | 1.666... |
| %   | 剰余（余り） | $5 \% 3$ | 2        |

### 3章 計算の実行例

#### Visual Studio Codeで計算を実行してみましょう

```
# 足し算  
print(5 + 3) # 結果: 8  
  
# 引き算  
print(5 - 3) # 結果: 2  
  
# 掛け算  
print(5 * 3) # 結果: 15  
  
# 割り算  
print(5 / 3) # 結果: 1.666666666666667  
  
# 剰余（余り）  
print(5 % 3) # 結果: 2
```

# 3章 文字列を結合してみよう

## | 文字列同士を連結（結合）する方法を学びます

- 複数の文字列を1つにつなげることを結合という
- Pythonには文字列を結合する方法がいくつかある
- 状況に応じて使い分けると便利

# 3章 文字列結合の方法

| 文字列を結合する代表的な方法を紹介します

| 方法       | 説明                    |
|----------|-----------------------|
| +演算子     | 文字列同士を+でつなげる          |
| f文字列     | f"..."の中に{変数}を埋め込む    |
| format() | "{}"に.format()で値を埋め込む |

# 3章 +演算子で結合

| +演算子を使って文字列を連結できます

```
first_name = "山田"  
last_name = "太郎"  
  
# +演算子で結合  
full_name = first_name + last_name  
print(full_name) # 結果: 山田太郎
```

- シンプルで直感的な方法
- 文字列同士の結合に使う

# 3章 f文字列で結合

## | f文字列を使うと変数を埋め込んで結合できます

```
name = "山田"  
age = 25  
  
# f文字列で結合  
message = f"私の名前は{name}で、年齢は{age}歳です."  
print(message) # 結果: 私の名前は山田で、年齢は25歳です。
```

- Python 3.6以降で使える
- 変数を{}で囲んで埋め込む

# 3章 format()で結合

## | format()メソッドを使う方法もあります

```
name = "山田"  
age = 25  
  
# format()で結合  
message = "私の名前は{}で、年齢は{}歳です。".format(name, age)  
print(message) # 結果: 私の名前は山田で、年齢は25歳です。
```

- {}がプレースホルダーになる
- format()の引数が順番に埋め込まれる

# 3章まとめ

| 本章では以下の内容を学習しました

## データ型

- 文字列型 (str)、整数型 (int)、浮動小数点型 (float)、論理型 (bool) の4つ

## 数値の計算

- 算術演算子 (+、 -、 \*、 /、 %) を使って計算ができる

## 文字列の結合

- +演算子、f文字列、format()の3つの方法がある

## 4章 変数を理解しよう

---

## | Pythonの変数について学びます

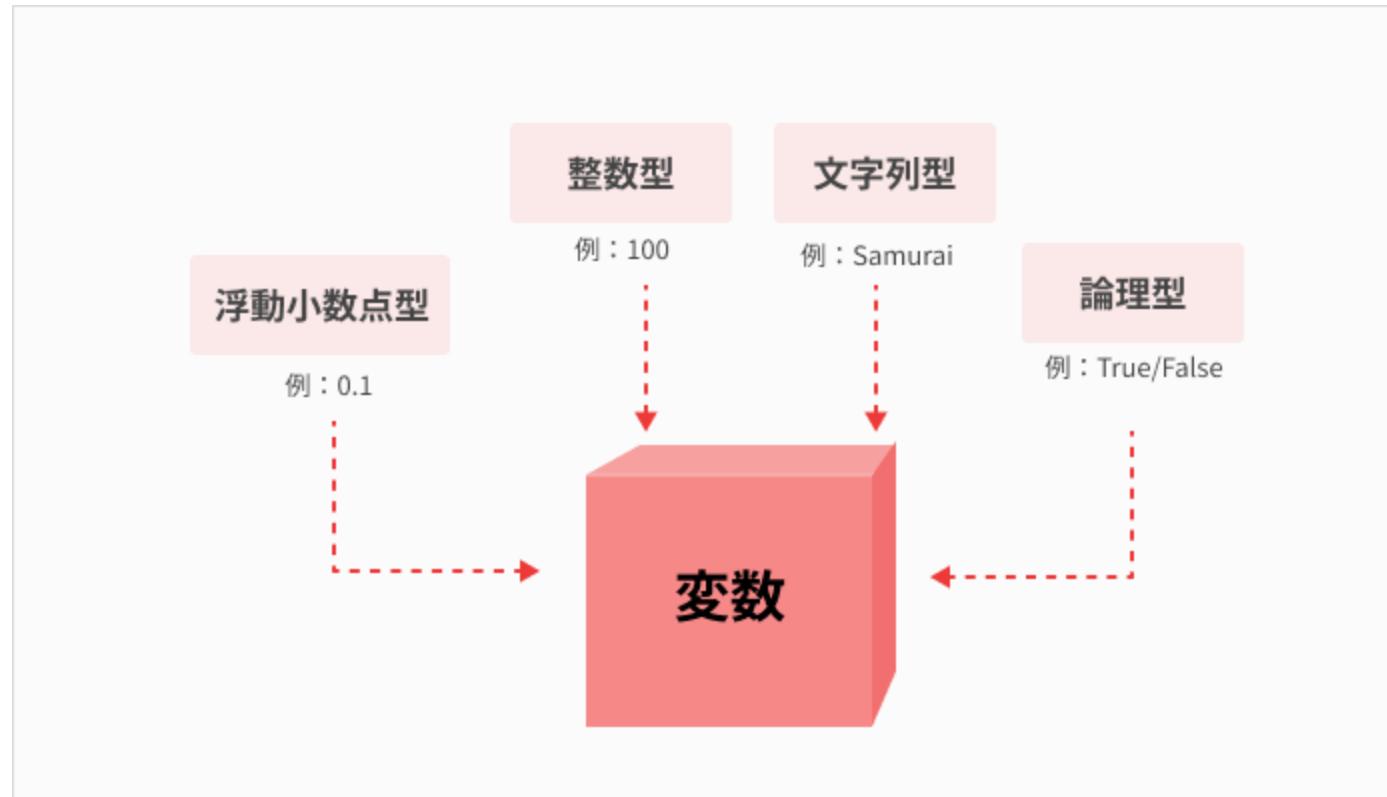
### 本章の目標

- 変数とは何か概要をつかむこと
- 変数の使い方を知ること
- 変数を実際に使ってみること
- 変数名のつけ方ルール（命名規則）を知ること

| 変数とは簡単にいえば、文字列や数値などのデータを入れる箱のようなものです

- 変数の中身はいつでも入れ替えられる
- Pythonをはじめとするプログラミング言語では、変数を当たり前のように使う

# 4章 変数とは



# 4章 なぜ変数が必要なのか

実際のプログラムやサービスでは、データの中身が変わることが当然のようにあります

- 前章で扱ったデータは「45」「晴れ」のように中身が決まったものだけ
- 実際のサービスでは、ユーザーごとにデータを変える必要がある

# 4章 変数が必要な例

画面に「おへりなさい、○○さん」と表示するWebアプリを考えましょう

The screenshot shows the homepage of the SAMURAI ENGINEER Plus application. On the left sidebar, there are navigation links: ホーム (Home), タイムライン (Timeline), カリキュラム (Curriculum), 教材 (Materials), Q&A (Q&A) with sub-links for お悩みQ&A and 技術Q&A, and 学習ログ (Learning Log). The main content area is titled "ホーム" (Home) and displays a welcome message "おかえりなさい、SAMURAIさん". Below this, there is a summary of learning statistics: SAMURAI (user icon), 総学習回数 (Total study sessions: 2), 完了教材数 (Completed materials: 2), 総レッスン数 (Total lessons: 0), and レッスンチケット (Lesson ticket: 1). A red box highlights the name "SAMURAIさん" in the welcome message. Below these stats, there are two cards: one for "次回レッスン未定" (Next lesson scheduled) with a link to "レッスンを行う >" and another for "学習中の教材" (Currently learning materials) showing "HTML/CSSの基礎を学ぼう" with a progress bar at 3/23 (13%).

## | 文字列をそのまま記述する場合、ユーザーごとに変えなければならず大変です

- ○○の部分を「侍太郎」「侍花子」などユーザーごとに変える必要がある
- このとき、変数を使ってデータを変えていく

## | Pythonで変数を使うには「変数名 = 変数値」と記述するだけでOKです

```
user_name = "侍太郎"
```

- このように記述することを「変数を定義する」という
- Pythonでは変数定義時にデータ型を指定する必要がない

# 4章 変数の使い方

## 変数の使い方



`user_name = ”侍太郎”`  
で値を代入するだけで変数が使える

## | データを代入する時点で、データ型が決まります

- 文字列を代入すれば文字列型
- 整数を代入すれば整数型として扱われる
- ほかの多くの言語では変数定義時にデータ型の指定が必要なため、Pythonのほうが楽

# 4章 変数名だけ書くとエラーになる

| 変数定義時にデータを指定しないとエラーになります

The screenshot shows a Jupyter Notebook cell with the following content:

```
user_name
user_name = "侍太郎"
```

Below the code, an error message is displayed:

```
NameError Traceback (most recent call last)
<ipython-input-1-7482852b6568> in <module>
----> 1 user_name
      2 user_name = "侍太郎"

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

A button labeled "SEARCH STACK OVERFLOW" is visible at the bottom of the cell.

# 4章 変数を使ってみよう

| 変数に値を代入し、その変数の中身を出力してみます

```
user_name = "侍太郎"  
print(user_name)
```

- Visual Studio Codeを開き、新しいコードセルを追加
- 上記のコードを入力して実行

## 4章 変数を使ってみよう：実行結果

The screenshot shows a Jupyter Notebook cell. On the left, there is a progress bar with a green checkmark icon and the text "0 秒". Next to it is a play button icon. The main area contains Python code:

```
user_name = "侍太郎"  
print(user_name)
```

The output of the code is displayed below the code cell:

侍太郎

## 4章 変数の中身を入れ替えてみよう

一度定義した変数の中身は、値を代入し直すだけで入れ替えられます

```
user_name = "侍太郎"  
print(user_name)
```

```
user_name = "侍花子"  
print(user_name)
```

## 4章 変数の中身を入れ替えてみよう：実行結果



0  
秒



```
user_name = "侍太郎"  
print(user_name)
```

```
user_name = "侍花子"  
print(user_name)
```



侍太郎  
侍花子

# 4章 変数を使って計算・連結してみよう

| 変数は数値や文字列と組み合わせて計算や連結が可能です

```
# 整数型と浮動小数点型の足し算
```

```
number1 = 5
```

```
number2 = 2.5
```

```
print(number1 + number2)
```

```
# 文字列型と文字列型の連結
```

```
last_name = "侍"
```

```
first_name = "太郎"
```

```
print(last_name + first_name)
```

## 4章 変数を使って計算・連結してみよう：実行結果

✓  
0  
秒



# 整数型と浮動小数点型の足し算

```
number1 = 5
number2 = 2.5
print(number1 + number2)
```

# 文字列型と文字列型の連結

```
last_name = "侍"
first_name = "太郎"
print(last_name + first_name)
```



7.5

侍太郎

# 4章 文字列の中で変数を表示する方法

## | f文字列を使うと、文字列の中に変数を埋め込めます

```
last_name    = "侍"
first_name   = "太郎"
sister_name = "花子"

# 3つの変数を文字列内に埋め込んで表示
print(f"私の名前は{last_name}{first_name}です。妹の名前は{sister_name}です。")
```

- 文字列の直前に「f」をつける
- 変数を表示したい部分に {変数名} と記述

## 4章 フォーマット済み文字列リテラル

| この記述方法を「フォーマット済み文字列リテラル」といいます

```
f"文字列{変数名1}文字列{変数名2}文字列{変数名3}....."
```

- 文字列に変数を埋め込んで表示することを変数展開と呼ぶ

## 4章 変数展開の実行結果

✓  
0  
秒



```
last_name = "侍"  
first_name = "太郎"  
sister_name = "花子"
```

# 3つの変数を文字列内に埋め込んで表示

```
print(f"私の名前は{last_name} {first_name}です。妹の名前は{sister_name}です。")
```

私の名前は侍太郎です。妹の名前は花子です。

# 4章 変数名のつけ方ルール（命名規則）

| 変数名のつけ方には以下の3つのルールがあります

## ① 使える文字

半角英字・半角数字・アンダースコア

## ② スネークケース

単語をアンダースコアで区切る

## ③ わかりやすい名前

変数の中身がわかる名前にする

## 4章 ルール① 変数名に使える文字

変数名に使える文字は以下の3種類です

| 使える文字          | 例             |
|----------------|---------------|
| 半角英字 (a~z、A~Z) | name, Age     |
| 半角数字 (0~9)     | number1, age2 |
| アンダースコア (_)    | user_name     |

- 全角文字や日本語は基本的に使用を避ける

## 4章 ルール① 使えない文字

| 以下の文字は変数名に使えず、エラーになります

| 使えない文字       | 例          |
|--------------|------------|
| 変数名の先頭に数字    | 1snake     |
| アンダースコア以外の記号 | snake\case |

## 4章 ルール② スネークケースで記述する

| Pythonでは変数名を「スネークケース」で付けることが推奨されています

### スネークケース

アルファベットで複合語  
(複数の単語から成り立つ語) を  
記述する際に、単語と単語の間を  
\_ (アンダースコア) で区切る記法

例：user\_name、user\_number



**snake\_case**

# 4章 スネークケースとは

| **単語と単語の間をアンダースコア（\_）で区切る記法です**

- 英字はすべて小文字にする
- 1単語のみの場合は、アンダースコアがなくても構わない
- 例： user\_name 、 phone\_number 、 age

## 4章 キャメルケースとの違い

| JavaScriptなどでは「キャメルケース」が推奨されています

### キャメルケース

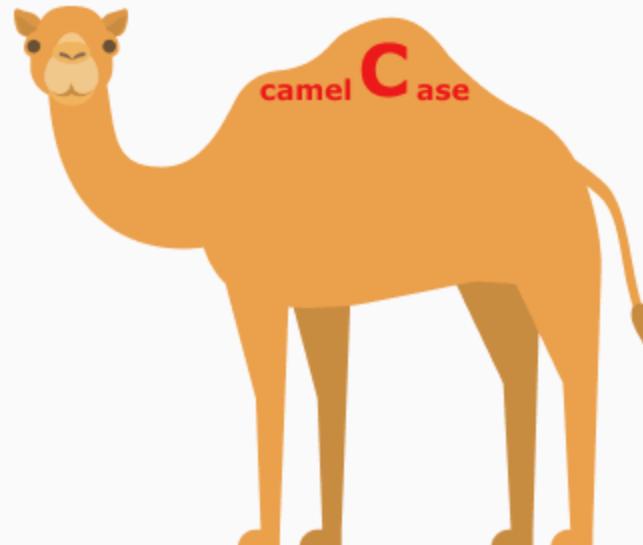
アルファベットで複合語を記述する際に各単語の先頭を大文字にする記法

### ローワーキャメルケース

例：userName、userNumber

### アッパーキャメルケース

例：UserName、UserNumber



# 4章 変数名の推奨・非推奨・NG

## 変数名の例を確認しましょう

# 推奨

```
snake_case = "◎" # 全部小文字・単語をアンダースコアで区切る  
snake      = "◎" # 1単語の場合はアンダースコアがなくてもOK
```

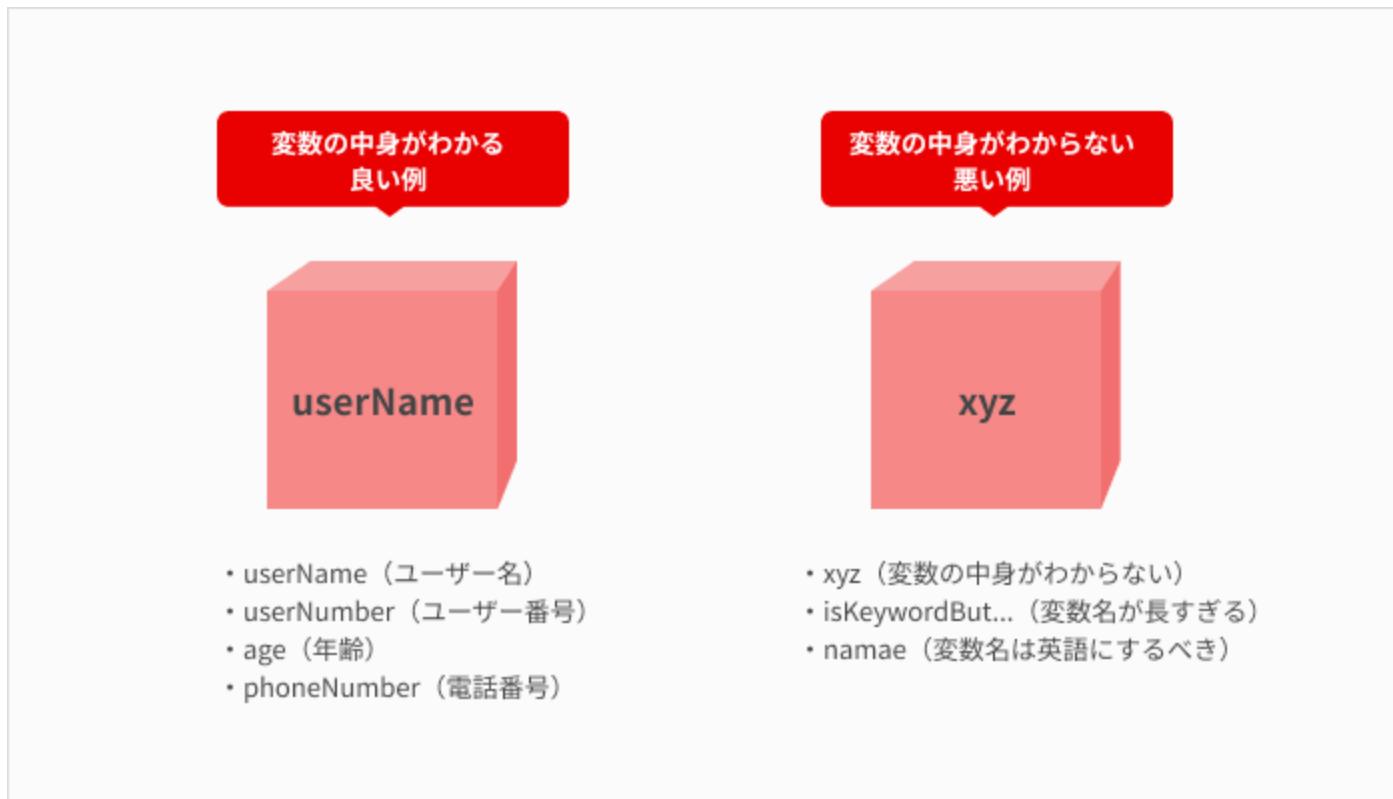
# 非推奨（エラーにはならない）

```
camelCase    = "△" # キャメルケースは非推奨  
SNAKE       = "△" # 全角は非推奨
```

# NG（そもそもエラーになる）

```
1snake = "x"      # 先頭が数字は不可
```

# 4章 ルール③ 変数の中身がわかる名前にする



## 4章 良い変数名の例

| 変数の中身がわかるような名前をつけましょう

| 変数名           | 意味      |
|---------------|---------|
| user_name     | ユーザー名   |
| user_number   | ユーザー番号  |
| age           | 年齢      |
| phone_number  | 電話番号    |
| alert_message | 警告メッセージ |

## 4章 悪い変数名の例

| 以下のような変数名は避けましょう

| 悪い例                             | 理由          |
|---------------------------------|-------------|
| xyz                             | 変数の中身がわからない |
| iskeywordbutnothingor_something | 変数名が長すぎる    |
| namae                           | 変数名は英語にするべき |

## 本章では以下の内容を学習しました

### 変数とは

- 文字列や数値などのデータを入れる箱のようなもの
- 変数名 = 変数值 で値を代入するだけで使える

### 変数の活用

- 値を再代入すれば中身を入れ替えられる
- 数値や文字列と組み合わせて計算や連結ができる
- f文字列で変数展開ができる

## 変数名のつけ方ルール（命名規則）

- ① 使える文字は半角英字・半角数字・アンダースコア（\_）
- ② 変数名はスネークケースで記述する
- ③ 変数の中身がわかるような名前にする

変数はプログラミングの基本中の基本です。しっかり覚えましょう

## 5章 配列（リスト・タプル・セット）を理解しよう

---

# 5章 配列（リスト・タプル・セット）を理解しよう

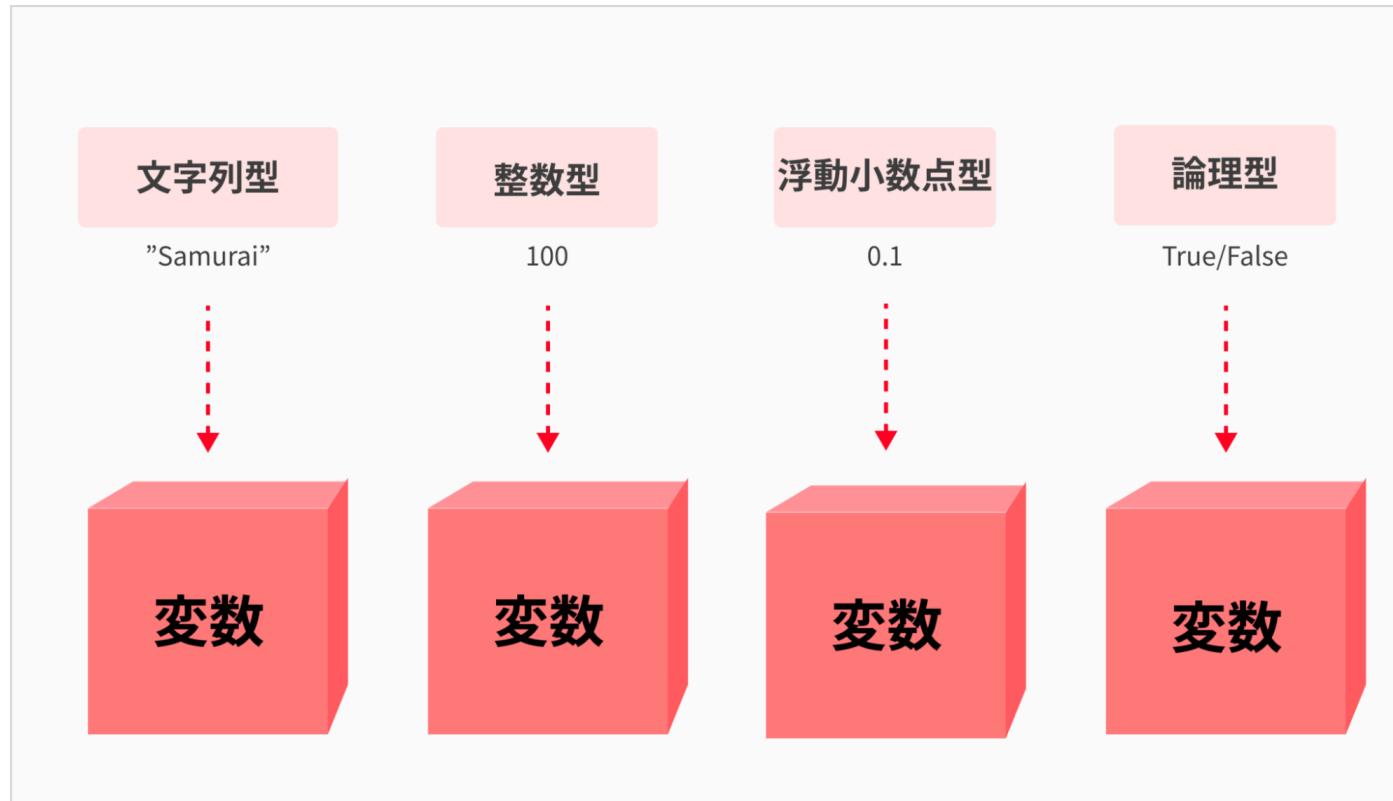
## 配列の概要を知り、実際に使ってみます

### 本章の目標

- 配列とは何か、概要をつかむこと
- 配列の作り方、使い方を知ること
- 実際に配列を使ってみること

# 5章 なぜ配列が必要なのか

| これまでには、変数に1つのデータしか入れていませんでした



# 5章 複数のデータをまとめて管理したいケース

Amazonのようなショッピングサイトで買い物をするシーンをイメージしてください

The screenshot shows the Amazon shopping cart interface. A red box highlights the product listing for a 'Fintie取り外し可能なBluetoothキーボード'. Another red box highlights the total price of ¥5,980. A third red box highlights the quantity selector set to '数量: 1'.

商品名  
価格  
数量

【Fire HD 10, Fire HD 10 Plus 2021年発売 第11世代用】Fintie取り外し可能なBluetoothキーボード

¥5,980

数量: 1

小計 (1 個の商品) (税込): ¥5,980  
この注文での獲得ポイント: 60pt

「商品名」 「価格」 「数量」 が保管されている

# 5章 変数と配列の違い

| 複数のデータは、1つのまとめたデータとして管理したほうが楽です

## 変数

変数1="商品A"  
変数2="1200円"  
変数3="1個"  
変数6="3個"  
変数7="商品C"  
変数4="商品B"  
変数5="800円"  
変数9="2個"  
変数8="2500円"

## 配列

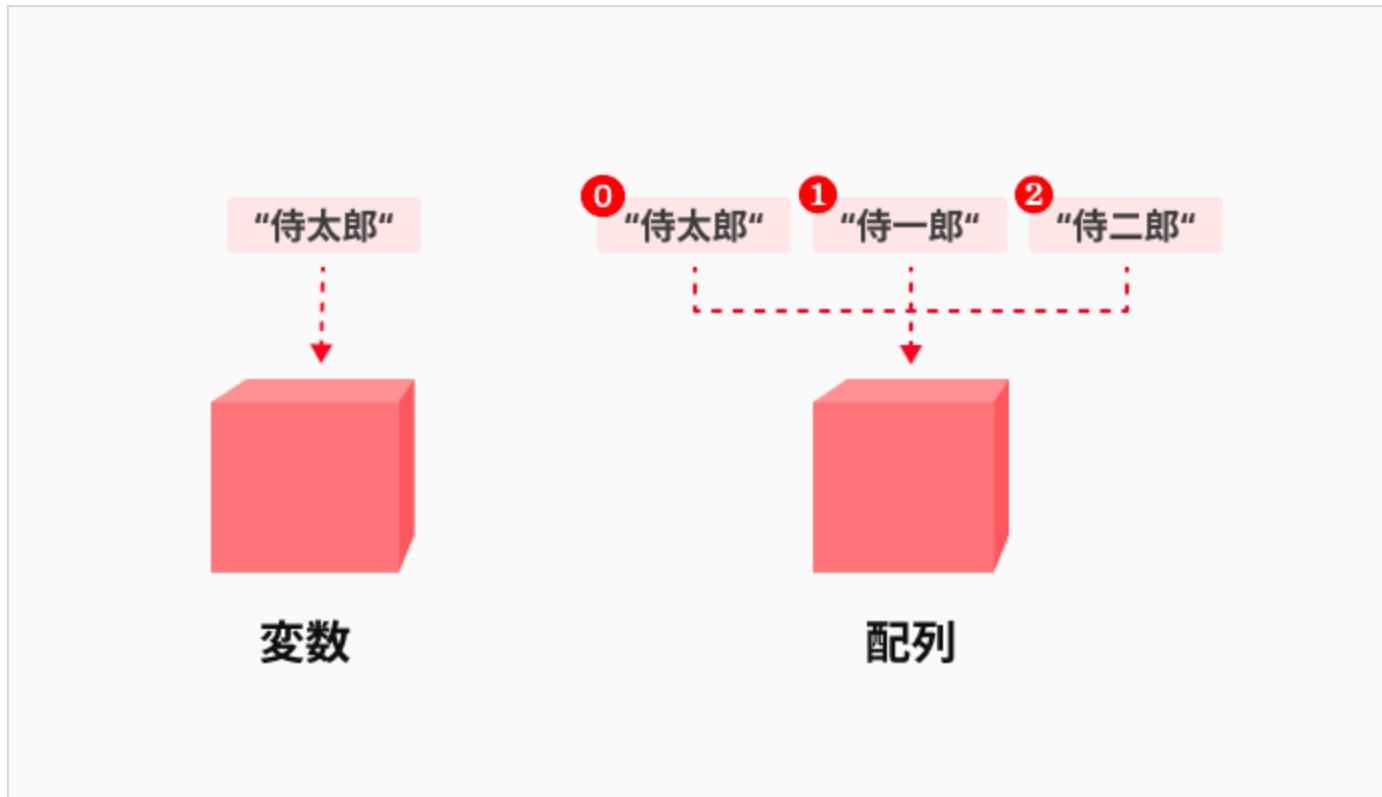
配列1 = ["商品A", "商品B", "商品C"]  
配列2 = ["1200円", "800円", "2500円"]  
配列3 = ["1個", "3個", "2個"]

# 5章 配列とは

| 配列とは簡単にいえば、複数のデータのまとめのことです

- 配列を使うことで、複数のデータを1つにまとめられる
- 配列に入れる1つひとつのデータを要素と呼ぶ

# 5章 配列のイメージ



# 5章 Pythonの配列の種類

| Pythonの配列には、以下の4種類が存在します

| 種類                   | 説明                |
|----------------------|-------------------|
| リスト (list)           | 要素の追加・変更・削除が可能    |
| タプル (tuple)          | 要素の追加・変更・削除が不可    |
| セット (set)            | 順番と重複の概念を持たない     |
| ディクショナリ (dictionary) | キーと値のペアで管理（7章で学習） |

# 5章 リストの作り方

| リスト (list) とは、要素の追加・変更・削除が行える配列のことです

```
リスト名 = [要素1, 要素2, 要素3, ...]
```

- 大かっこ [] ではさみ、その中に要素をカンマ (,) 区切りで入れる
- 配列名は複数形にするのが一般的

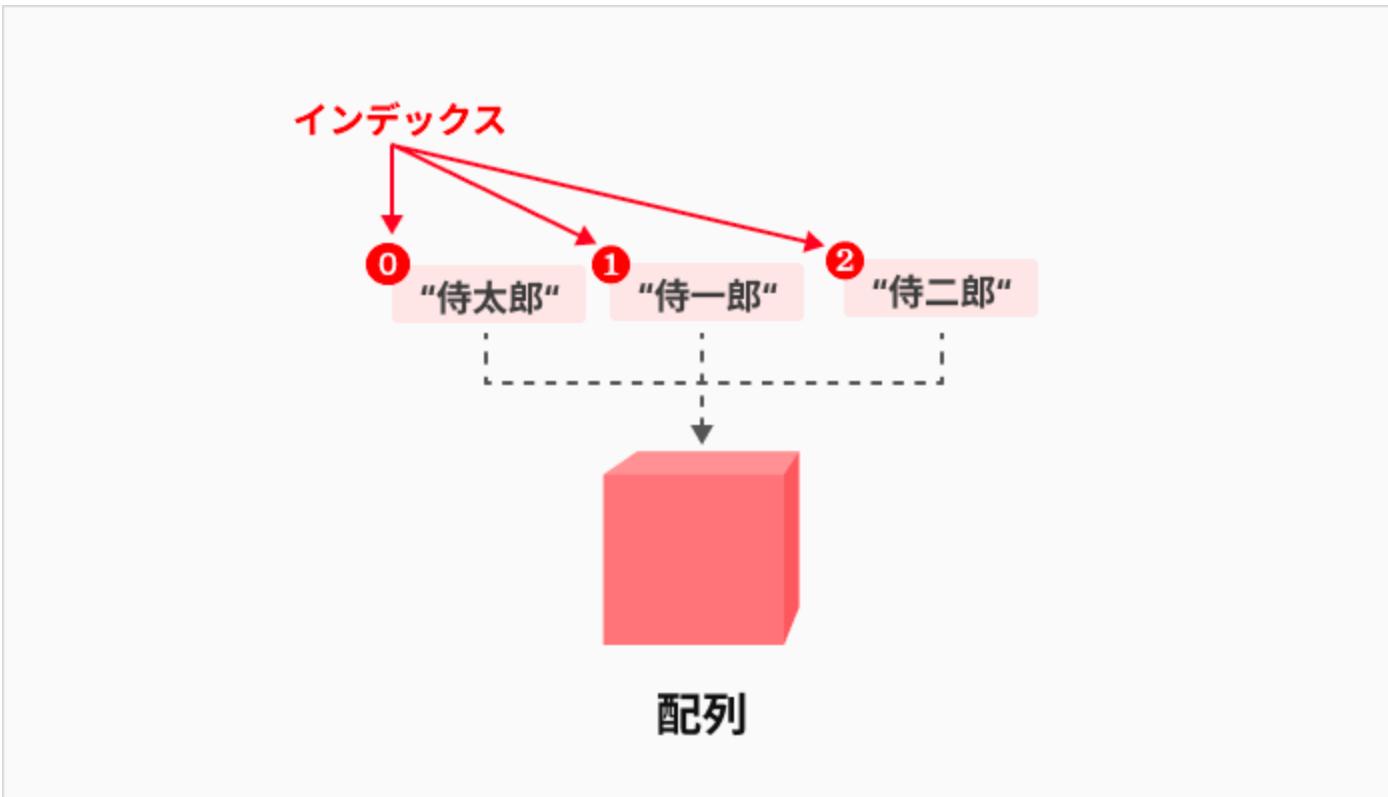
# 5章 リストの作成例

## | リストを作成してみましょう

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]  
user_ages  = [36, 33, 29, 25, 22]
```

- 文字列のリスト、数値のリストなど、様々なデータを格納できる

# 5章 インデックスとは



# 5章 リストの要素を取り出す方法

## | リスト名[インデックス] で要素を取り出せます

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]  
# 2番目の要素である「侍一郎」を取り出して表示する  
print(user_names[1])
```

- インデックスは「0」から始まる点に注意
- 2番目の要素を取り出すにはインデックス「1」を指定

# 5章 リストの要素を更新する方法

| リスト名[インデックス] = 新しい値 で要素を更新できます

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]  
# 2番目の要素を更新する  
user_names[1] = "侍花子"
```

- 単独の変数と同様に、新しい値を再代入すればOK

# 5章 リストの要素を追加・削除する方法

| メソッドを使って要素を追加・削除できます

| メソッド   | 説明               |
|--------|------------------|
| append | 要素を末尾に追加         |
| pop    | 指定したインデックスの要素を削除 |

リスト名.append(追加する要素の値)

リスト名.pop(削除する要素のインデックス)

# 5章 リストの追加・削除の例

## appendとpopを使ってみましょう

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

# 6番目の要素を追加する
user_names.append("侍五郎")

# 3番目の要素を削除する (0始まりのため2を指定)
user_names.pop(2)
```

## 5章 リストの追加・削除の結果

✓ 0 秒

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

# 6番目の要素を追加する
user_names.append("侍五郎")
print(user_names)

# 3番目の要素を削除する
user_names.pop(2) # 0始まりのため、3番目のインデックスは2
print(user_names)
```

[‘侍太郎’, ‘侍一郎’, ‘侍二郎’, ‘侍三郎’, ‘侍四郎’, ‘侍五郎’]  
[‘侍太郎’, ‘侍一郎’, ‘侍三郎’, ‘侍四郎’, ‘侍五郎’]

# 5章 リストを使ってみよう

Visual Studio Codeで以下のコードを実行してみましょう

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

print(user_names[1]) # 2番目の要素だけを表示

user_names[1] = "侍花子" # 2番目の要素を更新
print(user_names)

user_names.append("侍五郎") # 6番目の要素を追加
print(user_names)

user_names.pop(2) # 3番目の要素を削除
print(user_names)
```

# 5章 リストを使ってみよう：実行結果

✓ 0 秒

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

print(user_names[1]) # 2番目の要素だけを表示

user_names[1] = "侍花子" # 2番目の要素を更新
print(user_names)

user_names.append("侍五郎") # 6番目の要素を追加
print(user_names)

user_names.pop(2) # 3番目の要素を削除 (0始まりのため2を指定)
print(user_names)
```

侍一郎  
['侍太郎', '侍花子', '侍二郎', '侍三郎', '侍四郎']  
['侍太郎', '侍花子', '侍二郎', '侍三郎', '侍四郎', '侍五郎']  
['侍太郎', '侍花子', '侍三郎', '侍四郎', '侍五郎']

# 5章 タプルの作り方

| タプル (tuple) とは、要素の追加・変更・削除が行えない配列のことです

タプル名 = (要素1, 要素2, 要素3, ...)

- 小かっこ () で要素をはさむ
- 値の書き換えができないため、不变のデータを管理するのに適している

# 5章 タプルの用途

| タプルは不变のデータを管理するのに適しています

| 適している例

- ユーザーID
- 国名コード
- 固定の設定値

| 適していない例

- 更新日時
- 在庫数
- ユーザーの入力値

# 5章 タプルの要素を取り出す方法

## | タプル名[インデックス] で要素を取り出せます

```
country_names = ("日本", "アメリカ", "イギリス", "フランス")  
# 3番目の要素である「イギリス」を取り出す  
print(country_names[2])
```

- リストと同様に、インデックスは「0」から始まる

# 5章 タプルの値は変更できない

| タプルのデータを書き換えようとするときエラーになります

The screenshot shows a Jupyter Notebook cell with the following code:

```
country_names = ("日本", "アメリカ", "イギリス", "フランス")  
# 3番目の要素を書き換える  
country_names[2] = "ドイツ"
```

When run, it produces the following output:

```
TypeError Traceback (most recent call last)  
<ipython-input-15-2570c19ce039> in <module>  
      2  
      3 # 3番目の要素を書き換える  
----> 4 country_names[2] = "ドイツ"  
  
TypeError: 'tuple' object does not support item assignment
```

At the bottom of the cell, there is a button labeled "SEARCH STACK OVERFLOW".

# 5章 タプルを使ってみよう

Visual Studio Codeで以下のコードを実行してみましょう

```
country_names = ("日本", "アメリカ", "イギリス", "フランス")
# 3番目の要素を取り出す
print(country_names[2])
# すべての要素を取り出す
print(country_names)
```

## 5章 タプルを使ってみよう：実行結果

✓  
0  
秒

```
country_names = ("日本", "アメリカ", "イギリス", "フランス")  
  
# 3番目の要素を取り出す  
print(country_names[2])  
  
# すべての要素を取り出す  
print(country_names)
```

イギリス  
('日本', 'アメリカ', 'イギリス', 'フランス')

# 5章 セットの作り方

| セット (set) とは、「順番」と「重複」の概念を持たない配列のことです

セット名 = {要素1, 要素2, 要素3, ...}

- 中かっこ {} で要素をはさむ
- セットは「集合」とも呼ばれる

## 5章 セットの特徴① 順番の概念を持たない

| セットでは、各データにどのような順番が与えられるかわかりません

```
✓ 0 秒
▶ country_names = {"アメリカ", "イギリス", "日本", "フランス"}  
# セット全体を表示  
print(country_names)  
['日本', 'イギリス', 'フランス', 'アメリカ']
```

## 5章 セットの特徴② 重複の概念を持たない

| セットでは同じ値は1つしか存在しません

The screenshot shows a Jupyter Notebook cell with the following code:

```
# 1～10までに含まれる素数のセット (2と5が重複)
prime_numbers = {2, 5, 2, 7, 3, 5, 5}

# セット全体を表示
print(prime_numbers)
```

The output of the cell is:

```
{2, 3, 5, 7}
```

The line `prime_numbers = {2, 5, 2, 7, 3, 5, 5}` and its resulting output `{2, 3, 5, 7}` are highlighted with red boxes.

# 5章 セットの要素を取り出す方法

## | セットではインデックスでの取り出しができません



# 1~10までに含まれる素数のセット  
prime\_numbers = {2, 3, 5, 7}

# セット全体を表示  
print(prime\_numbers)

# セットの3番目の要素を表示 ⇒ できないためエラー  
print(prime\_numbers[2])

[2, 3, 5, 7]

-----  
TypeError

Traceback (most recent call last)

<ipython-input-3-ae90ad45b565> in <module>

6

7 # セットの3番目の要素を表示 ⇒ できないためエラー  
----> 8 print(prime\_numbers[2])

TypeError: 'set' object is not subscriptable

SEARCH STACK OVERFLOW

# 5章 セットの要素を追加・削除する方法

## | メソッドを使って要素を追加・削除できます

| メソッド   | 説明          |
|--------|-------------|
| add    | 要素を追加       |
| remove | 指定した値の要素を削除 |

セット名.add(追加する要素の値)

セット名.remove(削除する要素の値)

# 5章 セットの追加・削除の例

## | addとremoveを使ってみましょう

```
prime_numbers = {2, 3, 5, 7}  
  
prime_numbers.add(11)      # 11をセットに追加  
prime_numbers.remove(3)    # セットから3を削除
```

## 5章 セットの追加・削除の結果

✓



0  
秒

# 1～10までに含まれる素数のセット

```
prime_numbers = {2, 3, 5, 7}
```

# セット全体を表示

```
print(prime_numbers)
```

prime\_numbers.add(11) # 11をセットに追加

```
print(prime_numbers)
```

prime\_numbers.remove(3) # セットから3を削除

```
print(prime_numbers)
```



{2, 3, 5, 7}

{2, 3, 5, 7, 11}

{2, 5, 7, 11}

# 5章 セットを使ってみよう

| Visual Studio Codeで以下のコードを実行してみましょう

```
country_names = {"アメリカ", "イギリス", "日本", "フランス"}  
  
print(country_names) # セット全体を表示  
  
country_names.add("ドイツ") # 「ドイツ」を追加  
print(country_names)  
  
country_names.remove("イギリス") # 「イギリス」を削除  
print(country_names)
```

## 5章 セットを使ってみよう：実行結果

✓ 0 秒

```
country_names = {"アメリカ", "イギリス", "日本", "フランス"}  
  
# セット全体を表示  
print(country_names)  
  
country_names.add("ドイツ") # 「ドイツ」をセットに追加  
print(country_names)  
  
country_names.remove("イギリス") # セットから「イギリス」を削除  
print(country_names)
```

['日本', 'イギリス', 'フランス', 'アメリカ']  
['イギリス', 'フランス', '日本', 'ドイツ', 'アメリカ']  
['フランス', '日本', 'ドイツ', 'アメリカ']

# 5章 リスト・タプル・セットの比較

## 3種類の配列の違いをまとめます

| 特徴       | リスト | タプル | セット     |
|----------|-----|-----|---------|
| 記号       | [ ] | ( ) | { }     |
| 追加・変更・削除 | ○   | ×   | 追加・削除のみ |
| 順番       | あり  | あり  | なし      |
| 重複       | あり  | あり  | なし      |

# 5章 まとめ

## | 本章では以下の内容を学習しました

### 配列とは

- 複数のデータのまとめのこと
- 配列に入っているそれぞれの値を「要素」という
- 各要素には「0」から順番にインデックスが振られている

### 配列の種類

- リスト、タプル、セット、ディクショナリの4種類

# 5章まとめ（続き）

| 本章では以下の内容を学習しました

## リスト

- 要素の追加・変更・削除が可能、[ ] で作成

## タプル

- 要素の追加・変更・削除が不可、( ) で作成

## セット

- 順番・重複の概念を持たない、{ } で作成

## 6章 連想配列（ディクショナリ）を理解しよう

---

# 6章 連想配列（ディクショナリ）を理解しよう

## 連想配列の概要を学び、実際に使ってみます

### 本章の目標

- 連想配列（ディクショナリ）とは何かを理解する
- ディクショナリの基本操作（作成・取得・追加・変更・削除）を習得する

# 6章 連想配列（ディクショナリ）とは

## | インデックスの代わりに「キー」を使う配列

### 通常の配列（リスト）

- 番号（インデックス）で要素を管理
- 例：`list[0]`、`list[1]`

### 連想配列（ディクショナリ）

- 「キー」という名前で要素を管理
- 例：`dict["name"]`、`dict["age"]`

# 6章 ディクショナリの作成

## | {} と キー: 値 の形式で作成します

```
# ディクショナリの作成  
person = {"name": "田中", "age": 25, "city": "東京"}
```

### 構文

```
変数名 = {  
    "キー1": 値1,  
    "キー2": 値2,  
    "キー3": 値3  
}
```

### ポイント

- {} で囲む
- キーと値を : でつなぐ
- 各ペアは , で区切る

# 6章 値の取得

## | ディクショナリ[キー] で値を取得します

```
person = {"name": "田中", "age": 25, "city": "東京"}  
print(person["name"]) # 田中  
print(person["age"]) # 25
```

リストとの違い

| リスト     | ディクショナリ    |
|---------|------------|
| list[0] | dict["キー"] |
| 番号でアクセス | キーでアクセス    |

## | ディクショナリ[キー] = 値 で追加・変更できます

新しいキーの場合 → 追加

```
person = {"name": "田中"}  
person["age"] = 25  
# {"name": "田中", "age": 25}
```

既存のキーの場合 → 変更

```
person = {"name": "田中", "age": 25}  
person["age"] = 30  
# {"name": "田中", "age": 30}
```

## | pop() メソッドで要素を削除できます

```
person = {"name": "田中", "age": 25, "city": "東京"}  
person.pop("city")  
print(person) # {"name": "田中", "age": 25}
```

### 構文

ディクショナリ.pop("キー")

- 指定したキーの要素が削除される
- リストの `pop()` と似ているが、キーを指定する点が異なる

# 6章 ディクショナリの操作まとめ

## | 基本操作を整理しましょう

| 操作 | 書き方          | 例                  |
|----|--------------|--------------------|
| 作成 | {キー: 値}      | {"name": "田中"}     |
| 取得 | 辞書[キー]       | person["name"]     |
| 追加 | 辞書[新キー] = 値  | person["age"] = 25 |
| 変更 | 辞書[既存キー] = 値 | person["age"] = 30 |
| 削除 | 辞書.pop(キー)   | person.pop("age")  |

## | 本章では以下の内容を学習しました

### 連想配列（ディクショナリ）とは

- インデックス（番号）の代わりに「キー」で要素を管理する配列
- キーと値のペアでデータを格納する
- { } と キー: 値 の形式で作成する

# 6章まとめ（続き）

| 本章では以下の内容を学習しました

## 値の取得

- ディクショナリ["キー"] で値を取得

## 要素の追加・変更

- ディクショナリ["キー"] = 値 で追加または変更

## 要素の削除

- ディクショナリ.pop("キー") で削除

## 7章 条件分岐のif文を理解しよう

---

# 7章 条件分岐のif文を理解しよう

条件分岐とは何か、概要を学び、実際にコードを書いてみます

## 本章の目標

- 条件分岐とは何か、概要をつかむこと
- if文の書き方を知り、実際にコードを書いてみること
- 比較演算子の種類を知ること

## | プログラミングでは条件によって処理を分けたい場面がたくさんあります

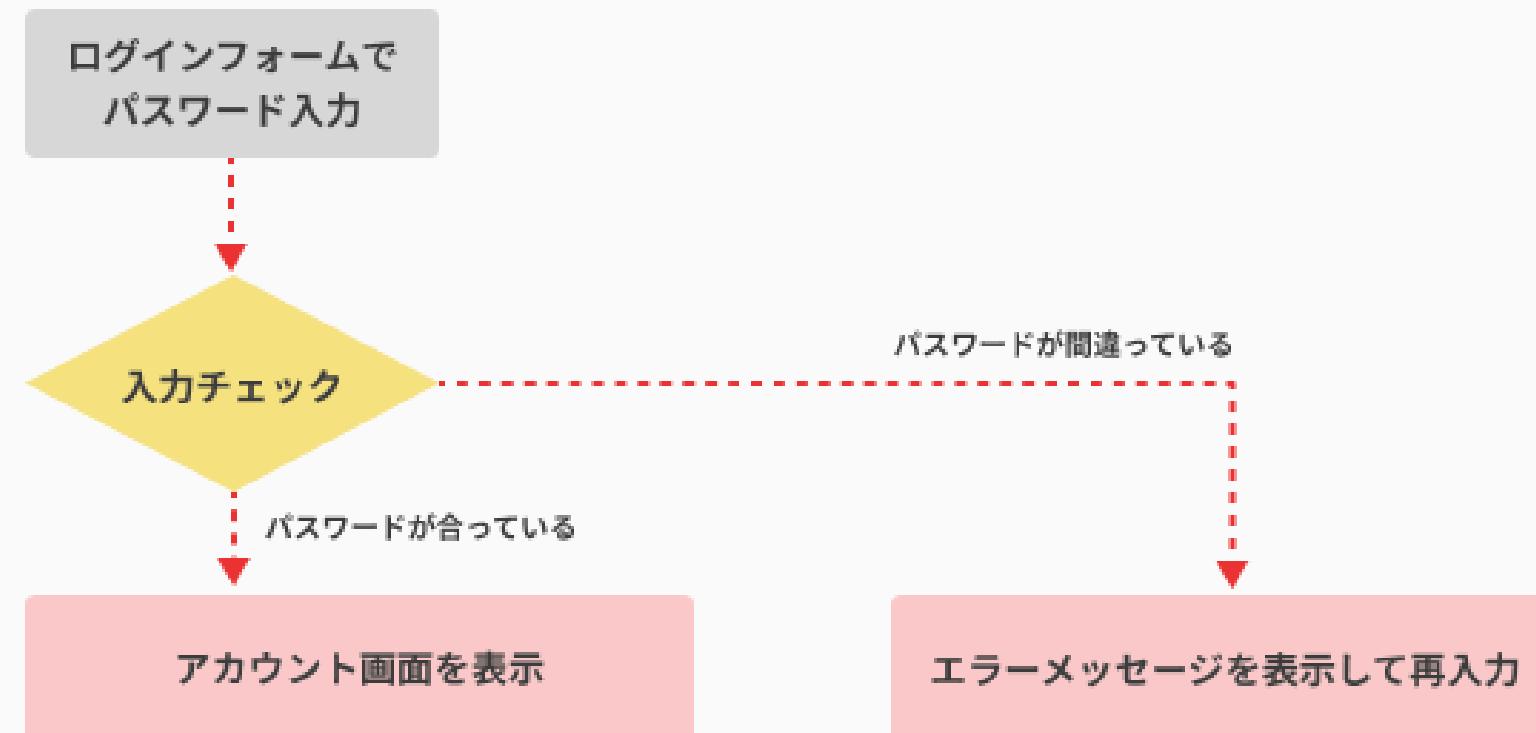
- ・「ある条件に当てはまるときだけこの処理を実行したい」という場面で使う
- ・条件分岐を使うことで様々な機能を実装できる

### 条件分岐の例

- ・もし金曜日なら商品を割引する
- ・未ログインであればログイン画面を表示する

## 条件

### 条件分岐の例

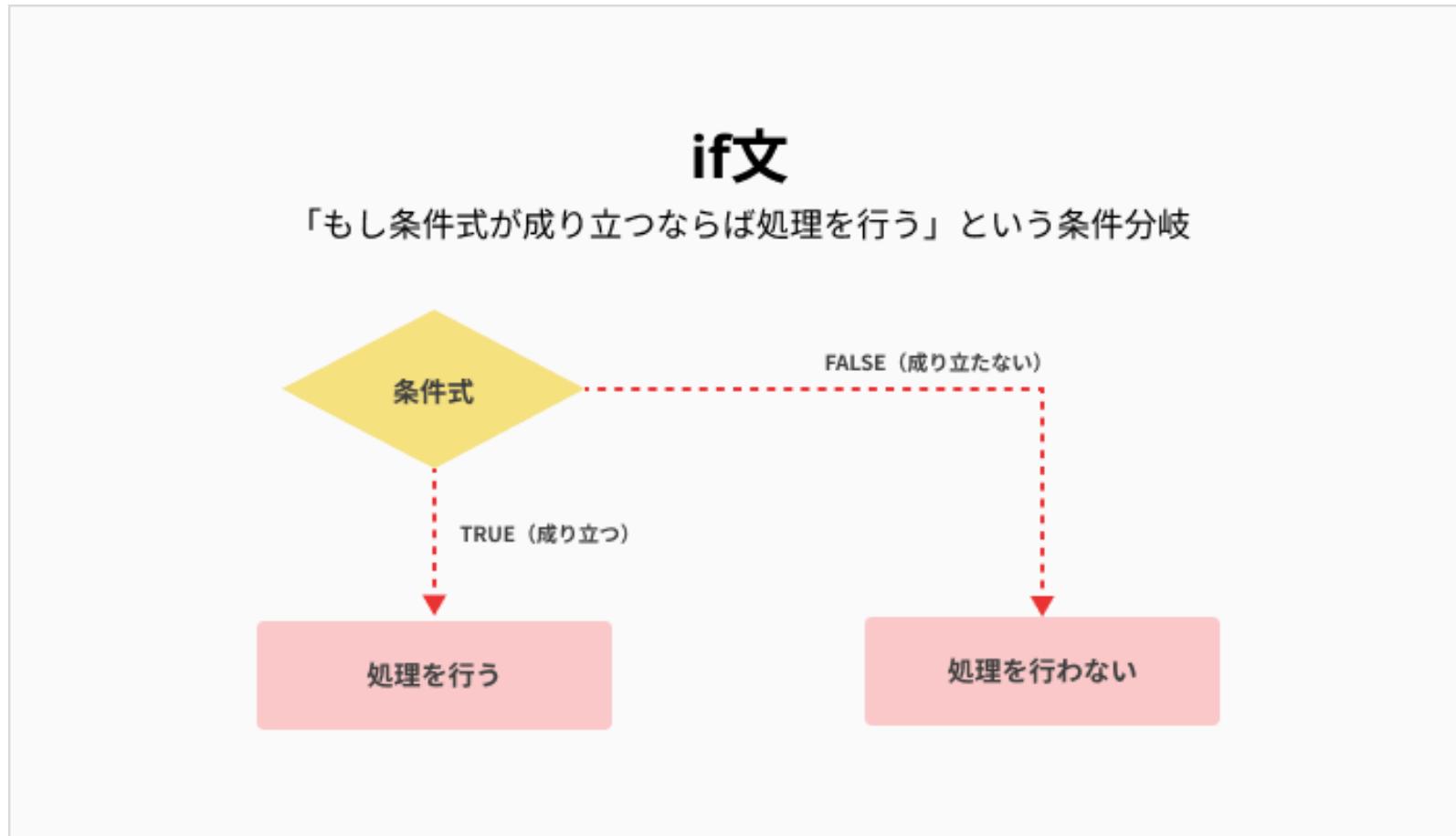


「必須事項が入力されていたらフォーム内容を送信し、未入力であればエラーメッセージを表示する」

- Pythonで条件分岐を行うときは、一般的に if文 を使う
- if文は最も基本的な条件分岐構文

# 7章 if文とは

「もし条件式が成り立つならば、処理を行う」という条件分岐構文です



## 条件が成り立つかどうかを判定する式のことです

- 条件が成り立てば True を返す
- 成り立たなければ False を返す
- if文ではTrueが返されたときに処理が実行される

# 7章 if文の書き方

| Pythonのif文は以下のように書きます

if 条件式:

　　→ 条件が成り立つときの処理

　　インデント

# 7章 if文の書き方のポイント

## | インデントが重要です

### 正しい書き方

```
if 条件式:  
    処理1 # 条件成立時のみ  
    処理2 # 条件成立時のみ
```

同じインデントの処理がまとめて実行される

### 間違った書き方

```
if 条件式:  
    処理1 # 条件成立時のみ  
    処理2 # 常に実行される
```

インデントがないと別の処理になる

# 7章 if文の使用例

## | 変数numの値によって処理を分岐させます

```
num = 50

# 変数numが10より大きいなら、文字列を出力する
if 10 < num:
    print("変数numは10より大きいです")

# 変数numが20より小さいなら、文字列を出力する
if num < 20:
    print("変数numは20より小さいです")
```

numは50なので、前者の条件のみ成り立つ

## 条件式から返ってきた値のことです

- 条件式は True (真) または False (偽) を返す
- この返ってきた値を 戻り値 または 返り値 という
- 四則演算の計算結果なども戻り値の1つ

# 7章 戻り値を出力してみよう

## | 算術演算子と比較演算子の戻り値を比較します

```
print(45 + 18) # 算術演算子を使った場合の戻り値を出力する  
print(45 > 18) # 比較演算子を使った場合の戻り値を出力する
```

## 7章 戻り値を出力してみよう

✓ 0 秒

▶ print(45 + 18) # 算術演算子を使った場合の戻り値を出力する  
print(45 > 18) # 比較演算子を使った場合の戻り値を出力する

⇨ 63  
True

# 7章 比較演算子とは

## 条件式に使う記号のことです

| 比較演算子              | 処理の内容                |
|--------------------|----------------------|
| <code>==</code>    | 2つの値が等しい場合はTrueを返す   |
| <code>!=</code>    | 2つの値が等しくない場合はTrueを返す |
| <code>&gt;</code>  | 左辺が右辺より大きい場合はTrueを返す |
| <code>&gt;=</code> | 左辺が右辺以上の場合はTrueを返す   |
| <code>&lt;</code>  | 左辺が右辺より小さい場合はTrueを返す |
| <code>&lt;=</code> | 左辺が右辺以下の場合はTrueを返す   |

## 7章 等価演算子の注意点

| データ型を合わせないと正しく比較できません

```
print("5" == 5) # False (文字列と整数は等しくない)
```

# 7章 データ型を合わせる方法

## | int関数でデータ型を変換してから比較します

```
num1 = 5
num2 = "5"

if num1 == int(num2):
    print("num1とnum2は等しいです")
```

# 7章 if文を書いてみよう

## | 3つのパターンで条件分岐を学びます

### | ① ifのみ

もし○○であれば、●●  
する

### | ② if + else

○○であれば●●し、そ  
れ以外は▲▲する

### | ③ if + elif + else

複数の条件で分岐する

## 「値が4であれば『大当たりです』と出力する」プログラム

```
import random

# 変数numに0~4までのランダムな整数を代入する
num = random.randint(0, 4)

print(num)

# 変数numの値が4であれば、「大当たりです」と出力する
if num == 4:
    print("大当たりです")
```

## 7章 パターン② if + else

### | elseで「それ以外のとき」の処理を追加します

**if** 条件式:

条件が成り立つときの処理

**else**:

条件が成り立たないときの処理

- elseを記述することで、条件式が成り立たないときにも処理を行える

## 7章 パターン② if + else

「4なら大当たり、それ以外ならはずれ」と出力するプログラム

```
import random

num = random.randint(0, 4)
print(num)

if num == 4:
    print("大当たりです")
else:
    print("はずれです")
```

# 7章 パターン③ if + elif + else

## | elifで条件式を複数作れます

**if** 条件式A:

条件Aが成り立つときの処理

**elif** 条件式B:

条件Bが成り立つときの処理

**else:**

どの条件も成り立たないときの処理

- elifはいくつでも追加できる
- ifとelseは1つの条件分岐で一度しか記述できない

## 7章 パターン③ if + elif + else

「4なら大当たり、3なら当たり、それ以外ならはずれ」

```
import random

num = random.randint(0, 4)
print(num)

if num == 4:
    print("大当たりです")
elif num == 3:
    print("当たりです")
else:
    print("はずれです")
```

# 7章 論理演算子とは

| 複数の条件式を組み合わせるときに使います

| 論理演算子 | 意味  | 説明                 |
|-------|-----|--------------------|
| and   | かつ  | すべての条件が成り立つときにTrue |
| or    | または | 1つでも条件が成り立てばTrue   |

## | andとorを使った条件分岐

```
# すべての条件が成り立つ場合にのみ処理を行う
if 10 < num and num < 30:
    print("変数numは10より大きく、30より小さいです")

# 1つでも条件が成り立てば処理を行う
if num == 10 or num == 30:
    print("変数numは10または30です")
```

# 7章 論理演算子を使ってみよう

## | andとorの動作を確認するコード

```
import random

num = random.randint(0, 4)
print(num)

# and条件：numが1より大きく、かつ3より小さい（2のみ成立）
if 1 < num and num < 3:
    print("変数numは1より大きく、3より小さいです")
else:
    print("and条件が成り立ちませんでした")
```

# 7章 論理演算子を使ってみよう（続き）

## or条件の例

```
# or条件 : numが1または3 (1か3のとき成立)
if num == 1 or num == 3:
    print("変数numは1または3です")
else:
    print("or条件が成り立ちませんでした")
```

# 7章 範囲判定の簡潔な書き方

## | 1つの変数がある範囲内かどうか判定する場合

### 通常の書き方

```
if 1 < num and num < 3:  
    処理
```

### 簡潔な書き方

```
if 1 < num < 3:  
    処理
```

## 本章では以下の内容を学習しました

### 条件分岐とは

- 条件によって処理を分けること
- Pythonではif文を使う

### if文の構文

- `if` 条件式: で条件分岐を開始
- `elif` で追加の条件を指定
- `else` でどの条件も成り立たないときの処理を指定

# 7章まとめ（続き）

| 本章では以下の内容を学習しました

## 比較演算子

| 演算子                                  | 意味         |
|--------------------------------------|------------|
| <code>==</code>                      | 等しい        |
| <code>!=</code>                      | 等しくない      |
| <code>&gt;</code> <code>&gt;=</code> | より大きい / 以上 |
| <code>&lt;</code> <code>&lt;=</code> | より小さい / 以下 |

## 本章では以下の内容を学習しました

### 論理演算子

- and（かつ）：すべての条件が成り立つときにTrue
- or（または）：1つでも条件が成り立てばTrue

条件分岐は様々な機能を実装するための基本です

## 8章 繰り返し処理のfor文を理解しよう

---

# 8章 繰り返し処理のfor文を理解しよう

| 繰り返し処理とは何か、概要を学び、実際にコードを書いてみます

## 本章の目標

- 繰り返し処理とは何か、概要をつかむこと
- for文の書き方を知り、実際にコードを書いてみること
- break文とcontinue文の使い方を知ること

# 8章 なぜ繰り返し処理が必要なのか

| 1~10までの整数を順番に出力したい場合を考えてみましょう

```
print("1")
print("2")
print("3")
print("4")
print("5")
# ... 続く
```

同じようなコードを何度も書くのは手間がかかり、コードが無駄に長くなる

# 8章 繰り返し処理を使うと

## | コードをスッキリさせられます

```
for i in range(1, 11):  
    print(i)
```

- たった2行で1～10までの整数を出力できる
- コードが短く、読みやすくなる

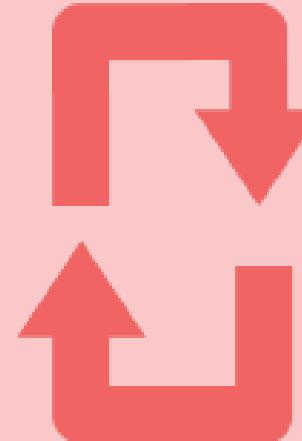
## 繰り返し処理とは

一般的な処理



1度だけ実行

繰り返し処理



何度も実行

# 8章 Pythonの繰り返し処理

| 代表的な繰り返し処理はfor文とwhile文の2つです

| 構文     | 使うケース                |
|--------|----------------------|
| for文   | 繰り返す回数があらかじめわかっている場合 |
| while文 | 繰り返す回数があらかじめわからない場合  |

本章ではまず for文 について学びます

# 8章 for文とwhile文の使い分け

## | それぞれの具体例

### for文の例

- 1～10までの数字を順番に表示する
- 配列の中の都道府県名を順番に取り出す

### while文の例

- サイコロで6の目が出るまで繰り返す
- 条件を満たすまで処理を続ける

## | for文は「決まった回数」同じ処理を繰り返し行いたいときに使います

**for** ループ変数 **in** 反復可能オブジェクト:  
    ブロック (繰り返し行いたい処理)

- 反復可能オブジェクトの後には : (コロン) を記載
- 繰り返し行われる処理はインデントを下げる

# 8章 反復可能オブジェクトとは

| 複数の情報を持ち、繰り返し処理ができるオブジェクトのことです

- 配列（リスト、タプル、セット）
- 辞書（ディクショナリ）
- 文字列
- range型

これらから情報を1つずつ取り出して処理を繰り返す

## | 反復可能オブジェクトから情報を取り出し、ループ変数に格納して処理します

1. 反復可能オブジェクトから情報を取り出す
2. 取り出した情報をループ変数に格納
3. ブロック内で処理する
4. 1～3を反復可能オブジェクトの中身の分だけ繰り返す

## 1～10までの整数を順番に出力するコード

```
for i in range(1, 11):
    print(i)
```

- `i` : ループ変数
- `range(1, 11)` : 1から10までを含むrange型を生成
- `print(i)` : ループ変数の値を出力

# 8章 for文の使用例

ループ変数 i に  
反復可能オブジェクト内の  
情報を繰り返しごとに格納

range関数を使い、  
1～10の範囲の整数を

作成する  
開始数 終了数

```
for i in range(1, 11):
    print(i)
```

↑ ループ変数 i の値を出力する

range関数で1～10を含む反復可能オブジェクトを生成。

繰り返しごとにループ変数に値が格納され、出力される。

反復可能オブジェクトの中身が全て処理されると繰り返し処理が終了。

# 8章 range関数とは

| 開始数から終了数までの連続した数値を要素として持つrange型を生成します

`range(終了数)`

`range(開始数, 終了数)`

`range(開始数, 終了数, ステップ)`

- 開始数を指定しない場合は 0 から開始
- 終了数は要素に含まれない 点に注意
- ステップは間隔 (省略すると1ずつ増える)

# 8章 range関数の例

## | 様々な書き方とその結果

```
range(3)  
# 要素：0, 1, 2
```

```
range(6, 9)  
# 要素：6, 7, 8
```

```
range(5, 11, 2)  
# 要素：5, 7, 9 (2つおきに増加)
```

# 8章 for文を書いてみよう

| Visual Studio Codeで実行してみましょう

```
for i in range(1, 11):
    print(i)
```

# 8章 for文を書いてみよう

## | 実行結果

✓  
0秒

▶  
`for i in range(1, 11):  
 print(i)`

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

# 8章 break文とは

| 繰り返し処理の途中で強制終了し、ループから抜け出す命令です

```
for i in range(1, 11):
    print(i)

    # ループ変数の値が5であれば、繰り返し処理を強制終了
    if i == 5:
        break
```

- 一般的に条件分岐の if 文 と組み合わせて使う

# 8章 break文を使ってみよう

## | 20が出たら繰り返し処理を終了するプログラム

```
import random

for i in range(1, 11):
    num = random.randint(1, 20)
    print(f"{i}回目の結果は{num}です。")

    if num == 20:
        print("20が出たので繰り返し処理を強制終了します。")
        break
```

# 8章 break文を使ってみよう

✓  
0  
秒

▶

```
#ランダムな整数を利用するため、記述が必要です。
import random

for i in range(1, 11):
    #変数numに1~20までのランダムな整数を代入する
    num = random.randint(1,20)

    print(f"{i}回目の結果は{num}です。")

#変数numの値が20であれば、break文で繰り返し処理を強制終了する
if num == 20:
    print("20が出たので繰り返し処理を強制終了します。")
    break
```

1回目の結果は10です。  
2回目の結果は18です。  
3回目の結果は15です。  
4回目の結果は11です。  
5回目の結果は2です。  
6回目の結果は20です。  
20が出たので繰り返し処理を強制終了します。

# 8章 break文を使ってみよう

✓  
0  
秒



#ランダムな整数を利用するため、記述が必要です。

```
import random
```

```
for i in range(1, 11):
```

#変数numに1~20までのランダムな整数を代入する

```
    num = random.randint(1,20)
```

```
    print(f"{i}回目の結果は{num}です。")
```

#変数numの値が20であれば、break文で繰り返し処理を強制終了する

```
    if num == 20:
```

```
        print("20が出たので繰り返し処理を強制終了します。")
```

```
        break
```

1回目の結果は13です。

2回目の結果は7です。

3回目の結果は16です。

4回目の結果は3です。

5回目の結果は1です。

6回目の結果は11です。

7回目の結果は7です。

8回目の結果は16です。

9回目の結果は3です。

10回目の結果は1です。

# 8章 continue文とは

| 繰り返し処理の途中で中断し、次のループに進む命令です

break文

ループを完全に抜け出す

continue文

今回のループをスキップして次へ進む

# 8章 continue文の使用例

## | 1~10のうち偶数のみを出力するコード

```
for i in range(1, 11):
    # 奇数（2で割った余りが1）であれば次のループへ
    if i % 2 == 1:
        continue

    print(i)
```

奇数のときはprint(i)が実行されずにスキップされる

## 8章 continue文の使用例

✓  
0  
秒



```
for i in range(1, 11):
    # ループ変数iの値が奇数(2で割った余りが1)であれば、値を出力せずにcontinue文で次のループに進む
    if i % 2 == 1:
        continue

    print(i)
```

```
2
4
6
8
10
```

# 8章 break文とcontinue文の違い

## 処理の流れを比較しましょう

| 文        | 動作          | 使用例               |
|----------|-------------|-------------------|
| break    | ループを完全に終了   | 条件を満たしたら処理を終える    |
| continue | 今回のループをスキップ | 特定の条件のときだけ処理をスキップ |

## | 本章では以下の内容を学習しました

### 繰り返し処理とは

- ・「決まった回数」または「条件を満たしている間」同じ処理を繰り返すこと
- ・Pythonではfor文とwhile文がある

### for文

- ・繰り返す回数があらかじめわかっている場合に使う
- ・range関数で連続した数値を生成できる

## | 本章では以下の内容を学習しました

### **break文**

- 繰り返し処理の途中で強制終了し、ループから抜け出す

### **continue文**

- 繰り返し処理の途中で中断し、次のループに進む

どちらも条件分岐のif文と組み合わせて使うことが多い

## 9章 繰り返し処理のwhile文を理解しよう

---

# 9章 繰り返し処理のwhile文を理解しよう

## while文の書き方を知り、実際にコードを書いてみます

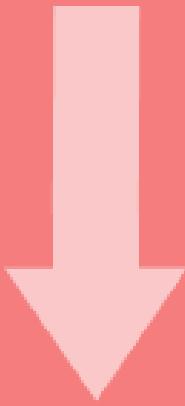
### 本章の目標

- while文の書き方を知り、実際にコードを書いてみること
- 無限ループの危険性を理解すること

## 繰り返し処理

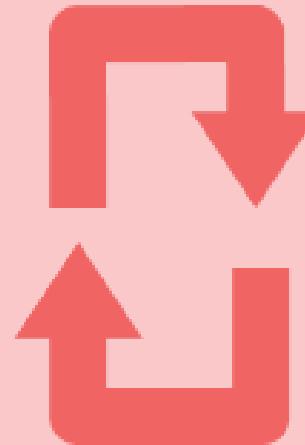
### 繰り返し処理とは

一般的な処理



1度だけ実行

繰り返し処理



何度も実行

# 9章 for文とwhile文の使い分け

## | それぞれの使うケースを確認しましょう

| 構文     | 使うケース                | 例            |
|--------|----------------------|--------------|
| for文   | 繰り返す回数があらかじめわかっている場合 | 1~10までの数字を表示 |
| while文 | 繰り返す回数があらかじめわからない場合  | 6の目が出るまで繰り返す |

本章では while文 について学びます

# 9章 while文とは

「条件を満たしている間」同じ処理を繰り返し行える構文です

**while** 条件式:

条件を満たしている間、繰り返す処理

- if文やfor文と同じように 条件式 を使う
- 条件式がTrueの間、処理を繰り返す

# 9章 比較演算子の復習

## 条件式に使う比較演算子を確認しましょう

| 比較演算子                                | 処理の内容                   |
|--------------------------------------|-------------------------|
| <code>==</code>                      | 2つの値が等しい場合はTrueを返す      |
| <code>!=</code>                      | 2つの値が等しくない場合はTrueを返す    |
| <code>&gt;</code> <code>&gt;=</code> | 左辺が右辺より大きい / 以上の場合はTrue |
| <code>&lt;</code> <code>&lt;=</code> | 左辺が右辺より小さい / 以下の場合はTrue |

# 9章 while文の使用例

| 変数numの値が0以外である間、処理を繰り返す

```
import random

# 変数numに0~4までのランダムな整数を代入
num = random.randint(0, 4)

# 変数numの値が0以外である間、繰り返す
while num != 0:
    num = random.randint(0, 4)
    print(num)
```

# 9章 while文の使用例

## 条件式の解説

```
while num != 0:
```

- `num != 0` が条件式
- 変数numの値が0以外のときにTrueを返す
- Trueの間、ブロック内の処理が繰り返される

# 9章 while文を書いてみよう

| Visual Studio Codeで実行してみましょう

```
import random

num = random.randint(0, 4)
print(f"最初の値は{num}です。")

while num != 0:
    num = random.randint(0, 4)
    print(f"現在の値は{num}です。")
```

# 9章 while文を書いてみよう



#ランダムな整数を利用するため、記述が必要です。

```
import random
```

# 変数numに0~4までのランダムな整数を代入する

```
num = random.randint(0, 4)
```

# 変数numの最初の値を出力する(確認用)

```
print(f"最初の値は{num}です。")
```

# 変数numの値が0以外である間、変数numの値を出力し続ける

```
while num != 0:
```

# 変数numに0~4までのランダムな整数を代入する

```
num = random.randint(0, 4)
```

# 次の条件式で比較される、変数numの現在の値を出力する

```
print(f"現在の値は{num}です。")
```

最初の値は2です。

現在の値は1です。

現在の値は2です。

現在の値は1です。

現在の値は3です。

現在の値は4です。

現在の値は1です。

現在の値は4です。

現在の値は0です。

# 9章 無限ループの危険性

## | while文で注意すべき最も重要なポイントです

```
num = 5  
  
# 条件式が常にTrueを返すので、無限ループになる  
while num == 5:  
    print(num)
```

- 変数numの値は常に5なので、条件式は常にTrue
- 処理が永遠に終わらない 無限ループ が発生

# 9章 無限ループを防ぐには

「条件式がFalseを返す可能性はあるか」を必ず確認

無限ループになる

```
num = 5
while num == 5:
    print(num)
```

numが変化しない

正常に終了する

```
num = 5
while num == 5:
    num = random.randint(0, 10)
    print(num)
```

numが変化する可能性あり

# 9章 while文でのbreak文

| for文と同様にbreak文が使えます

```
import random

num = random.randint(0, 4)
i = 1

while num != 0:
    num = random.randint(0, 4)

    if i == 5:
        print("5回目なので強制終了します。")
        break

    print(f"現在の値は{num}です。")
    i = i + 1
```

# 9章 while文でのbreak文

```
if i == 5:  
    print("5回目なので繰り返し処理を強制終了します。")  
    break  
  
# 次の条件式で比較される、変数numの現在の値を出力する  
print(f"現在の値は{num}です。")  
  
# カウンタ変数の値を1増やす  
i = i + 1
```

最初の値は4です。  
現在の値は2です。  
現在の値は3です。  
現在の値は2です。  
現在の値は3です。  
5回目なので繰り返し処理を強制終了します。

# 9章 while文でのbreak文

```
# カウンタ変数iの値が5であれば、break文で繰り返し処理を矯正終了する
if i == 5:
    print("5回目なので繰り返し処理を強制終了します。")
    break

# 次の条件式で比較される、変数numの現在の値を出力する
print(f"現在の値は{num}です。")

# カウンタ変数の値を1増やす
i = i + 1
```

最初の値は1です。  
現在の値は2です。  
現在の値は0です。

# 9章 while文でのcontinue文

| for文と同様にcontinue文も使えます

```
import random

sum = 0

while sum < 20:
    num = random.randint(1, 10)
    print(f"{num}が出ました。")

    if num % 2 == 0:
        print("偶数なので加算しません。")
        continue

    sum = sum + num
    print(f"現在の合計は{sum}です。")
```

# 9章 while文でのcontinue文

```
print("偶数なので加算しません。")  
continue
```

```
# 変数sumに変数numの値を加算する  
sum = sum + num  
print(f"現在の合計は{sum}です。")
```

- 9が出ました。  
現在の合計は9です。  
2が出ました。  
偶数なので加算しません。  
7が出ました。  
現在の合計は16です。  
7が出ました。  
現在の合計は23です。

## | 本章では以下の内容を学習しました

### while文

- ・「条件を満たしている間」同じ処理を繰り返す
- ・繰り返す回数があらかじめわからない場合に使う

### 無限ループ

- ・条件式が常にTrueだと処理が永遠に終わらない
- ・「条件式がFalseを返す可能性はあるか」を必ず確認

本章では以下の内容を学習しました

## break文とcontinue文

- while文でもfor文と同様に使える
- break文: ループを強制終了
- continue文: 今回のループをスキップして次へ

while文は条件に基づく繰り返しに便利ですが、無限ループに注意しましょう

# 10章 配列や辞書の繰り返し処理を理解しよう

---

# 10章 配列や辞書の繰り返し処理を理解しよう

| 配列や辞書（連想配列）を使った繰り返し処理の書き方を学びます

## 本章の目標

- 配列や辞書（連想配列）を使った繰り返し処理の書き方を知り、実際にコードを書いてみること

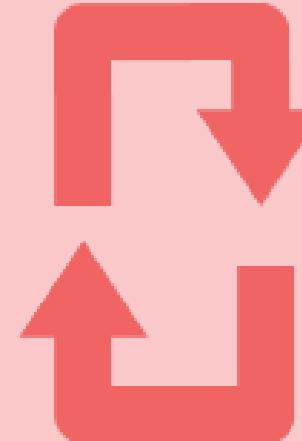
## 繰り返し処理とは

一般的な処理



1度だけ実行

繰り返し処理



何度も実行

# 10章 for文とwhile文の使い分け

## | それぞれの使うケースを確認しましょう

| 構文     | 使うケース  | 例                          |
|--------|--|----------------------------|
| for文   | 繰り返す回数があらかじめわかっている場合。または配列や辞書に対して繰り返し処理を行いたい場合 | 1～10までの数字を表示、配列の要素を順番に取り出す |
| while文 | 繰り返す回数があらかじめわからない場合                            | サイコロで6の目が出るまで繰り返す          |

本章では配列や辞書を利用した繰り返し処理について学びます

# 10章 配列に対する繰り返し処理

配列や辞書も反復可能オブジェクトなので、for文と組み合わせて使えます

- 配列や辞書の各要素を順番に取り出す
- 繰り返し処理の回数 = 配列や辞書の要素数となる

# 10章 配列を使ったfor文の書き方

| 配列の要素を1つずつ順番に取り出して処理できます

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

for user_name in user_names:
    print(user_name)
```

- user\_name : 取り出した要素を代入するループ変数名
- user\_names : 要素を取り出す配列（反復可能オブジェクト）名

# 10章 配列を使ったfor文の動作イメージ

| ループ変数には配列から取り出した要素が順番に代入されます

| 周回  | ループ変数の値           |
|-----|-------------------|
| 1周目 | user_name ← '侍太郎' |
| 2周目 | user_name ← '侍一郎' |
| 3周目 | user_name ← '侍二郎' |
| 4周目 | user_name ← '侍三郎' |
| 5周目 | user_name ← '侍四郎' |

繰り返し処理の中でこの変数を使えば、配列の各要素に対してさまざまな処理ができます

# 10章 配列を使ってfor文を書いてみよう

| 配列の要素を順番に出力するコードを書いてみましょう

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

# 配列user_namesの要素を1つずつ順番に出力する
for user_name in user_names:
    print(user_name)
```

# 10章 配列を使ってfor文を書いてみよう

✓  
0秒

```
▶ user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]  
  
# 配列user_namesの要素を1つずつ順番に出力する  
for user_name in user_names:  
    print(user_name)
```

侍太郎  
侍一郎  
侍二郎  
侍三郎  
侍四郎

# 10章 辞書に対する繰り返し処理

| 辞書（連想配列）に対してもfor文で繰り返し処理ができます

```
personal_data = {"name": "侍太郎", "age": 36, "gender": "男性"}
```

この辞書のキーと値を順番に出力するには？

# 10章 辞書を使ったfor文の書き方

## | items()関数を使ってキーと値をセットで取り出します

```
personal_data = {"name": "侍太郎", "age": 36, "gender": "男性"}  
  
for key, value in personal_data.items():  
    print(f"{key}は{value}です。")
```

- personal\_data : 要素を取り出す辞書名
- .items() : 辞書からキーと値をセットで取り出す関数
- key : 取り出したキーを代入するループ変数名
- value : 取り出した値を代入するループ変数名

# 10章 辞書を使ったfor文の動作イメージ

| ループ変数にはキーと値が順番に代入されます

| 周回  | keyの値    | valueの値 |
|-----|----------|---------|
| 1周目 | "name"   | "侍太郎"   |
| 2周目 | "age"    | 36      |
| 3周目 | "gender" | "男性"    |

繰り返し処理の中でこれらの変数を使えば、辞書の各キー、各値に対してさまざまな処理ができます

# 10章 値だけを取り出す場合

| values()関数を使うとキーを省略して値だけを取り出せます

```
personal_data = {"name": "侍太郎", "age": 36, "gender": "男性"}  
  
for value in personal_data.values():  
    print(value)
```

- .values() : 辞書から値だけを取り出す関数

# 10章 辞書を使ってfor文を書いてみよう

| 辞書のキーと値を順番に出力するコードを書いてみましょう

```
personal_data = {"name": "侍太郎", "age": 36, "gender": "男性"}  
  
# 連想配列personal_dataのキーと値を1つずつ順番に出力する  
for key, value in personal_data.items():  
    print(f"{key}は{value}です。")  
  
# 連想配列personal_dataの値を1つずつ順番に出力する  
for value in personal_data.values():  
    print(value)
```

# 10章 辞書を使ってfor文を書いてみよう

✓ 0秒

```
personal_data = {"name": "侍太郎", "age": 36, "gender": "男性"}  
  
# 連想配列personal_dataのキーと値を1つずつ順番に出力する  
for key, value in personal_data.items():  
    print(f"{key}は{value}です。")  
  
# 連想配列personal_dataの値を1つずつ順番に出力する  
for value in personal_data.values():  
    print(value)
```

nameは侍太郎です。  
ageは36です。  
genderは男性です。  
侍太郎  
36  
男性

# 10章 補足：配列のインデックスを取り出す

## | enumerate関数を使うと値だけでなくインデックスも取り出せます

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

# 配列user_namesのインデックスと値を1つずつ順番に出力する
for index, value in enumerate(user_names):
    print(f"{index} : {value}")
```

- enumerate関数：配列からインデックスと値をセットで取り出す関数
- インデックス=配列の各要素に対して「0」から順番に振られる番号

# 10章 補足：配列のインデックスを取り出す

✓ 0 秒



```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

# 配列user_namesのインデックスと値を1つずつ順番に出力する
for index, value in enumerate(user_names):
    print(f"{index} : {value}")
```

0 : 侍太郎  
1 : 侍一郎  
2 : 侍二郎  
3 : 侍三郎  
4 : 侍四郎

# 10章 配列・辞書でのbreak文

## 配列や辞書を利用したfor文でもbreak文が使えます

- 繰り返し処理の途中で強制終了し、ループから抜け出す
- 一般的に条件分岐のif文と組み合わせて使う

# 10章 break文の使用例

## 配列から特定の値を見つけたら処理を終了する

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

# 検索したいユーザー名を代入する変数
target = "侍二郎"

for user_name in user_names:
    print(user_name)

    # 変数user_nameと変数targetの値が一致すれば、break文で強制終了
    if (user_name == target):
        print(f"{target}さんが見つかったので、繰り返し処理を強制終了します。")
        break
```

# 10章 break文の使用例

✓  
0  
秒

```
user_names = ["侍太郎", "侍一郎", "侍二郎", "侍三郎", "侍四郎"]

# 検索したいユーザー名を代入する変数
target = "侍二郎"

for user_name in user_names:
    print(user_name)

# 変数user_nameと変数targetの値が一致すれば、break文で繰り返し処理を強制終了する
if (user_name == target):
    print(f"{target}さんが見つかったので、繰り返し処理を強制終了します。")
    break
```

```
侍太郎
侍一郎
侍二郎
侍二郎さんが見つかったので、繰り返し処理を強制終了します。
```

## 配列や辞書を利用したfor文でもcontinue文が使えます

- 繰り返し処理の途中で中断し、次のループに進む
- 一般的に条件分岐のif文と組み合わせて使う

# 10章 continue文の使用例

## 特定の条件を満たさない要素をスキップする

```
score = {  
    "国語": 80,  
    "数学": 55,  
    "理科": 70,  
    "社会": 85,  
    "英語": 60  
}  
  
print("合格した科目は以下のとおりです。")  
  
for key, value in score.items():  
    # 点数が70より小さければ、continue文で次のループに進む  
    if (value < 70):  
        continue  
    print(f"{key} : {value}点")
```

# 10章 continue文の使用例

✓ 0 秒

```
score = {  
    "国語": 80,  
    "数学": 55,  
    "理科": 70,  
    "社会": 85,  
    "英語": 60  
}  
  
print("合格した科目は以下のとおりです。")  
  
for key, value in score.items():  
    # 変数valueの値(点数)が70より小さければ、キー(科目名)と値(点数)を出力せずにcontinue文で次のループに進む  
    if (value < 70):  
        continue  
  
    print(f"{key} : {value}点")
```

合格した科目は以下のとおりです。  
国語 : 80点  
理科 : 70点  
社会 : 85点

## | 本章では以下の内容を学習しました

### 配列・辞書の繰り返し処理

- for文を使えば、配列や辞書に対して繰り返し処理を行うことができる
- 「繰り返し処理の回数＝配列や辞書の要素数」となる

### 辞書の便利な関数

- items()：キーと値をセットで取り出す
- values()：値だけを取り出す

| 本章では以下の内容を学習しました

## enumerate関数

- 配列のインデックスと値をセットで取り出せる

## break文とcontinue文

- 配列や辞書を利用したfor文でも使える
- break文：繰り返し処理の途中で強制終了し、ループから抜け出す
- continue文：繰り返し処理の途中で中断し、次のループに進む
- 一般的に条件分岐のif文と組み合わせて使う

# 11章 関数を理解しよう

---

## 関数とは何か概要を学び、実際に使ってみます

### 本章の目標

- 関数とは何か概要をつかむこと
- 関数の作り方・呼び出し方を知ること
- 関数を実際に使ってみること

# 11章 なぜ関数が必要なのか

| 同じ処理を何度も書くのは大変で、コードが無駄に長くなります

```
print("おはようございます！")
print("昨日はよく眠れましたか？")
print("今日も一日頑張りましょう!")
```

```
print("おはようございます！")
print("昨日はよく眠れましたか？")
print("今日も一日頑張りましょう!")
```

処理の一部を変更したいときに、すべてのコードを書き換えなければなりません

# 11章 関数を使えば解決できる

## 一度関数を作るだけで何度も再利用できます

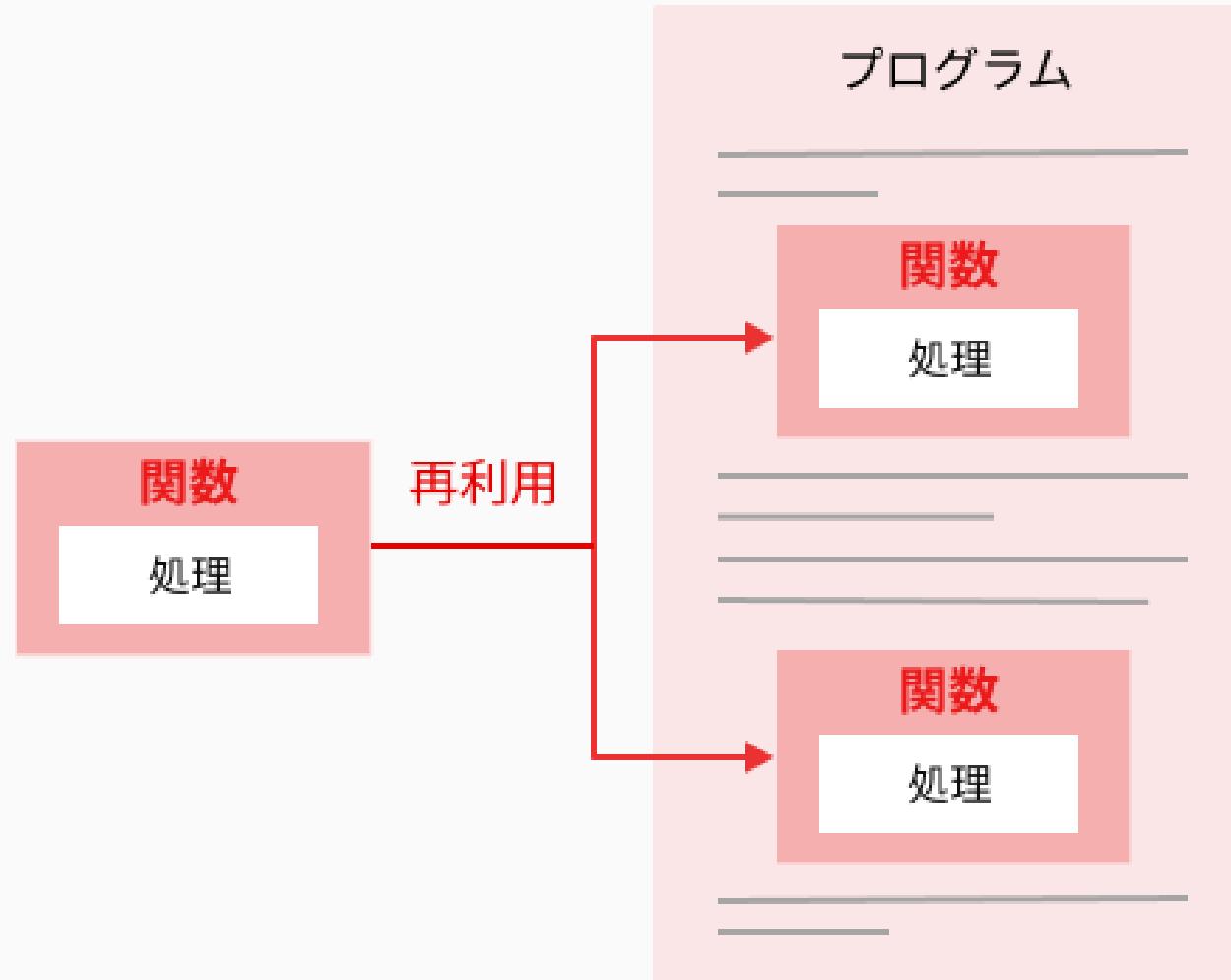
```
# 朝のあいさつを出力する関数を作成する
def say_good_morning():
    print("おはようございます！")
    print("昨日はよく眠れましたか？")
    print("今日も一日頑張りましょう！")

# 関数を呼び出す（1回目）
say_good_morning()

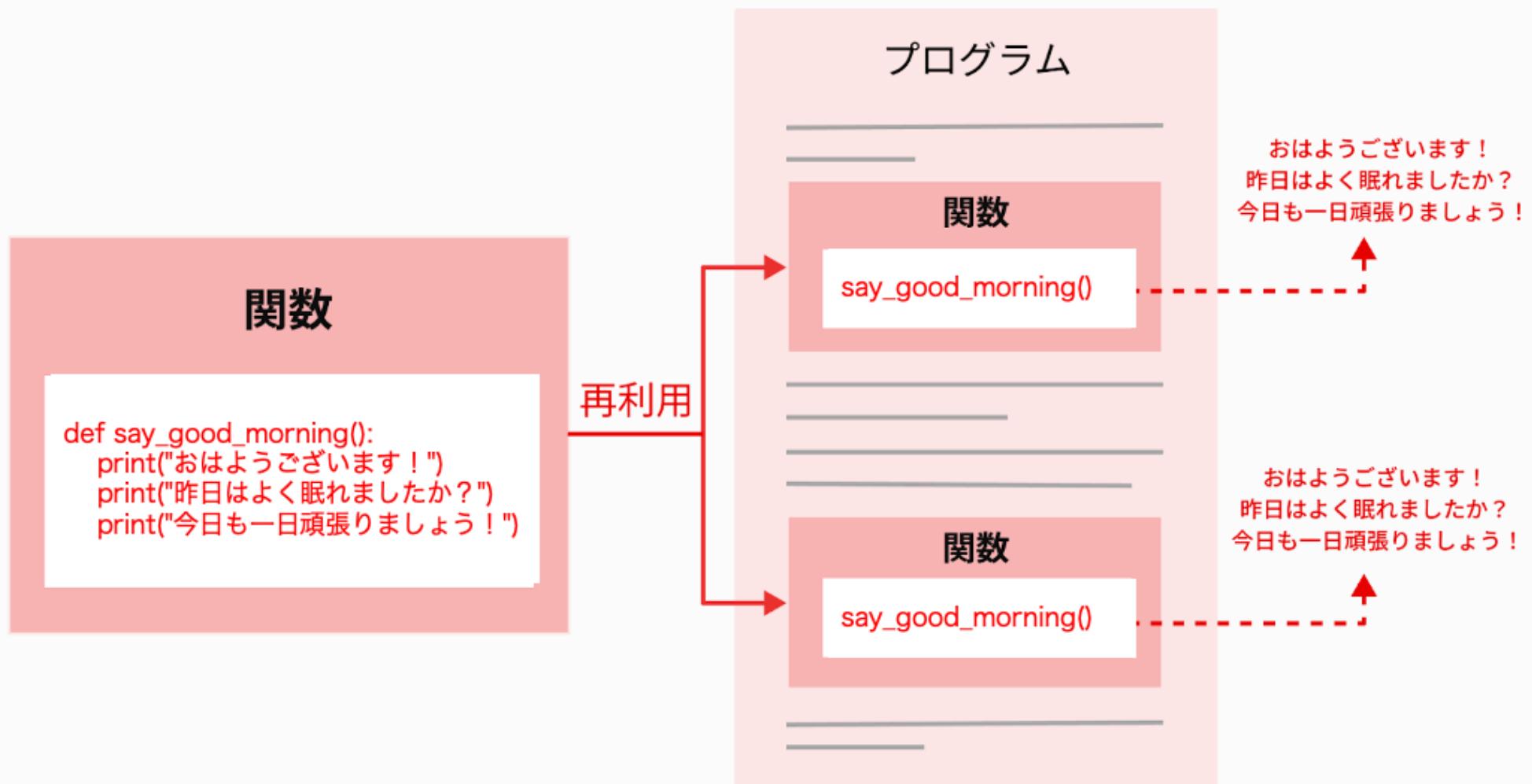
# 関数を呼び出す（2回目）
say_good_morning()
```

出力内容を変えたいときも、一度コードを書き換えれば済みます

# 11章 関数とは



# 11章 関数とは



## 関数を使うことで得られるメリットをまとめます

- 複雑なコードを1つにまとめられる
- 同じ処理を行うときに何度も再利用できる
- プログラミングの生産性を高め、素早い開発ができるようになる

## | defキーワードを使って関数を定義します

```
def 関数名():  
    一連の処理
```

- def : 「define」「definition」（定義する）の略
- 関数を作成することを「関数を定義する」と呼ぶ

## 朝のあいさつの例

```
def say_good_morning():
    print("おはようございます！")
    print("昨日はよく眠れましたか？")
    print("今日も一日頑張りましょう！")
```

- 関数名は say\_good\_morning
- インデントされた部分が関数の処理内容

# 11章 関数名のつけ方

「動詞 + 目的語」の形になると処理内容がわかりやすい

| 処理の内容    | 関数名       | 例                        |
|----------|-----------|--------------------------|
| ○○を追加する  | add_○○    | add_product (商品を追加する)    |
| ○○を削除する  | remove_○○ | remove_product (商品を削除する) |
| ○○が存在するか | has_○○    | has_product (商品が存在するか)   |
| ○○の状態か   | is_○○     | is_purchased (購入されたか)    |

## | 関数を呼び出す（実行する）には、関数名を記述するだけ

```
# 関数を定義
def say_good_morning():
    print("おはようございます！")

# 関数を呼び出す
say_good_morning()
```

- 関数名の後に () をつけて呼び出す

# 11章 関数を書いてみよう

## 2つの関数を定義して呼び出してみましょう

# 朝のあいさつを出力する関数を作成する

```
def say_good_morning():
    print("おはようございます！")
    print("昨日はよく眠れましたか？")
    print("今日も一日頑張りましょう！")
```

# 夜のあいさつを出力する関数を作成する

```
def say_good_evening():
    print("こんばんは！")
    print("今日も一日お疲れさまでした。")
```

# 朝のあいさつを出力する関数を呼び出す

```
say_good_morning()
```

# 夜のあいさつを出力する関数を呼び出す

```
say_good_evening()
```

# 11章 関数を書いてみよう

✓



0 秒

```
# 朝のあいさつを出力する関数を作成する
def say_good_morning():
    print("おはようございます!")
    print("昨日はよく眠れましたか?")
    print("今日も一日頑張りましょう!")
```

```
# 夜のあいさつを出力する関数を作成する
def say_good_evening():
    print("こんばんは!")
    print("今日も一日お疲れさまでした。")
```

```
# 朝のあいさつを出力する関数を呼び出す
say_good_morning()
```

```
# 夜のあいさつを出力する関数を呼び出す
say_good_evening()
```

おはようございます!  
昨日はよく眠れましたか?  
今日も一日頑張りましょう!  
こんばんは!  
今日も一日お疲れさまでした。

## | 本章では以下の内容を学習しました

### 関数とは

- 一連の処理をひとまとめにして、何度でも再利用できるようにする仕組み
- 複雑なコードを1つにまとめられる
- 同じ処理を行うときに何度も再利用できる

## 本章では以下の内容を学習しました

### 関数の作り方

```
def 関数名():  
    一連の処理
```

### 関数の呼び出し方

- 関数名を記述するだけでよい（例： `say_good_morning()` ）
- 関数は呼び出して初めて実行される

## 12章 関数の引数・戻り値を理解しよう

---

# 12章 関数の引数・戻り値を理解しよう

## 関数で使う引数・戻り値と型ヒントについて学びます

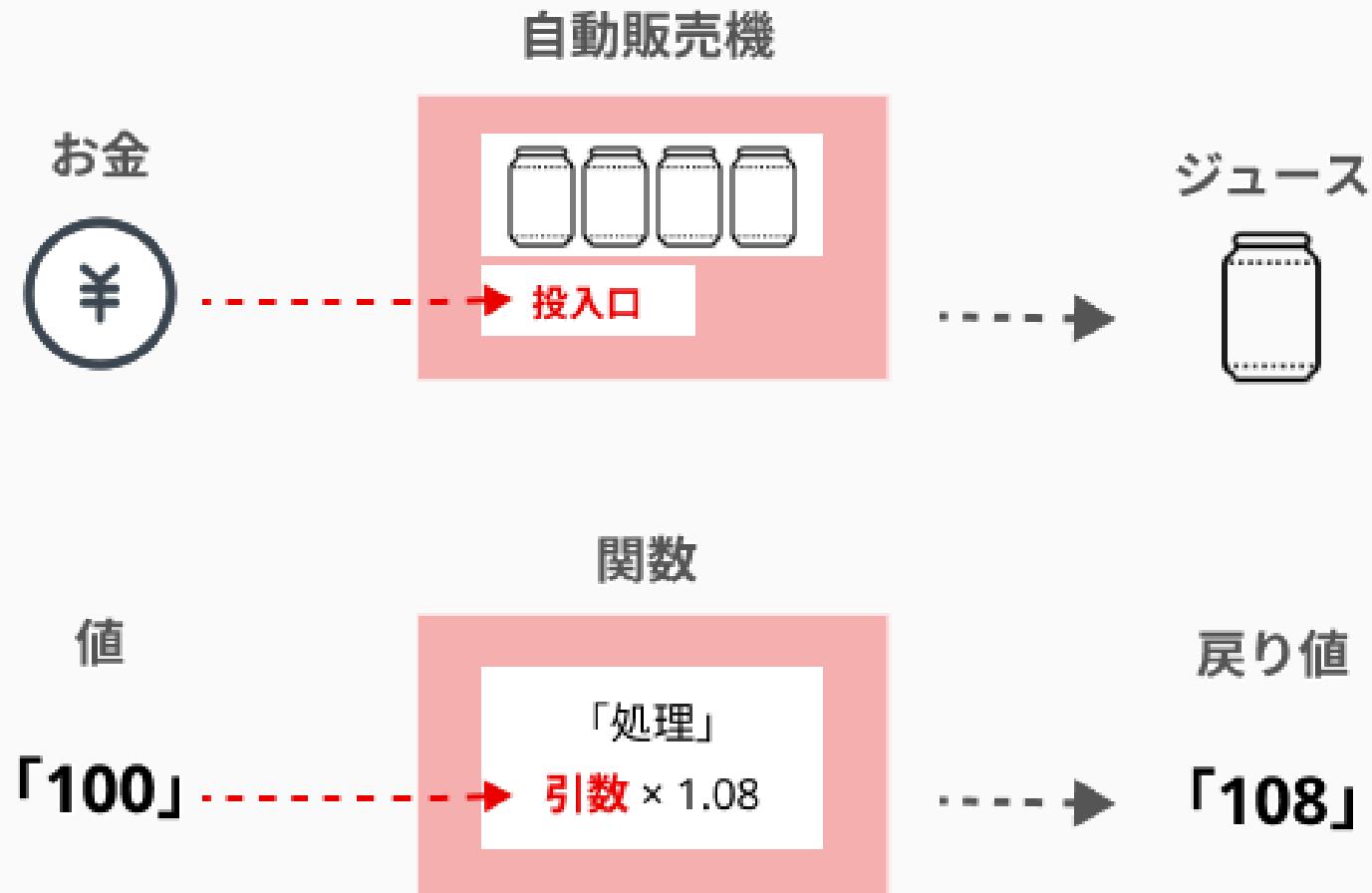
### 本章の目標

- 引数について学び、関数をより柔軟に使いこなせるようになること
- 戻り値について学び、関数から処理結果を受け取る方法を知ること
- 引数や戻り値のデータ型を指定する「型ヒント」について知ること

## 12章 引数とは

| 引数（ひきすう）とは、関数に与えるデータのことです

# 12章 引数とは



## | 決まっていない値を使って処理を行いたいときに引数が必要です

- 朝のあいさつを出力する関数 → 決まった処理なので引数は不要
- 購入金額に送料を加算する関数 → 購入金額は決まっていない値なので引数が必要

## 購入金額に送料を加算する関数



## | 関数の丸括弧()内に引数名を記述します

```
def 関数名(引数名):  
    引数を使った一連の処理
```

- 関数の定義時にはまだ引数の具体的な値がわからない
- 変数を使って処理を作りおくイメージ

## 購入金額に送料を加算する関数の例

```
def calculate_total(price):
    # 与えられた引数priceに送料を加算し、変数totalに代入する
    total = price + 500

    # 変数totalの値を出力する
    print(f'{total}円')
```

## | 関数を呼び出すときに引数を記述します

```
def calculate_total(price):
    total = price + 500
    print(f'{total}円')
```

```
# 関数を呼び出し、引数として購入金額を渡す
calculate_total(1200)
```

- 関数の()内に引数を記述することを「引数を渡す」と表現する

## 12章 引数を渡して関数を呼び出す

✓  
0 秒



```
# 与えられた引数$priceに送料を加算し、その値を出力する関数を定義する
def calculate_total(price):
    # 与えられた引数priceに送料を加算し、変数totalに代入する
    total = price + 500

    # 変数totalの値を出力する
    print(f"{total}円")

# 関数を呼び出し、引数として購入金額を渡す
calculate_total(1200)
```

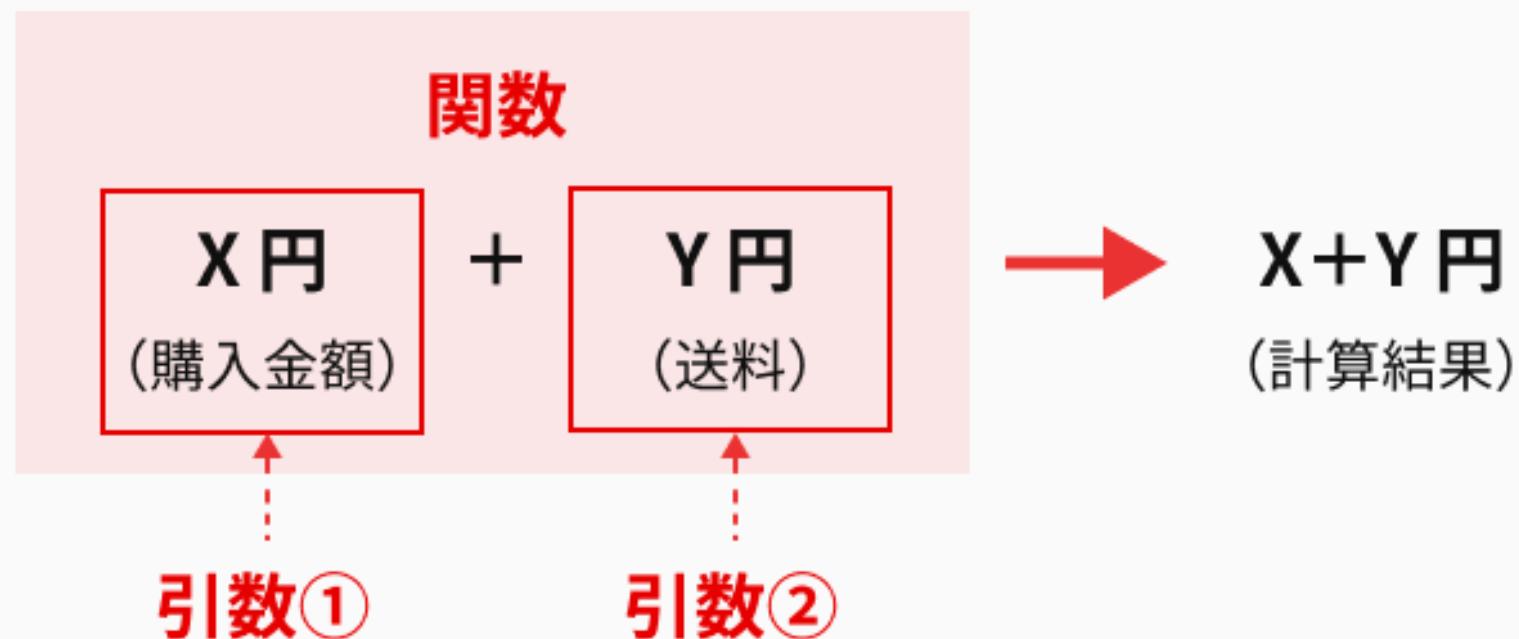
1700円

## 12章 引数を渡して関数を呼び出す

(例) 購入金額が1200円の場合



## 引数が複数の場合



# 12章 複数の引数を受け取る関数

## カンマ区切りで引数を記述します

```
def add_two_arguments(price, shipping_fee):
    # 与えられた引数priceと引数shipping_feeを加算
    total = price + shipping_fee
    print(f'{total}円')

# 関数を呼び出し、引数として購入金額と送料を渡す
add_two_arguments(1200, 500)
```

## 12章 複数の引数を受け取る関数

✓  
0 秒

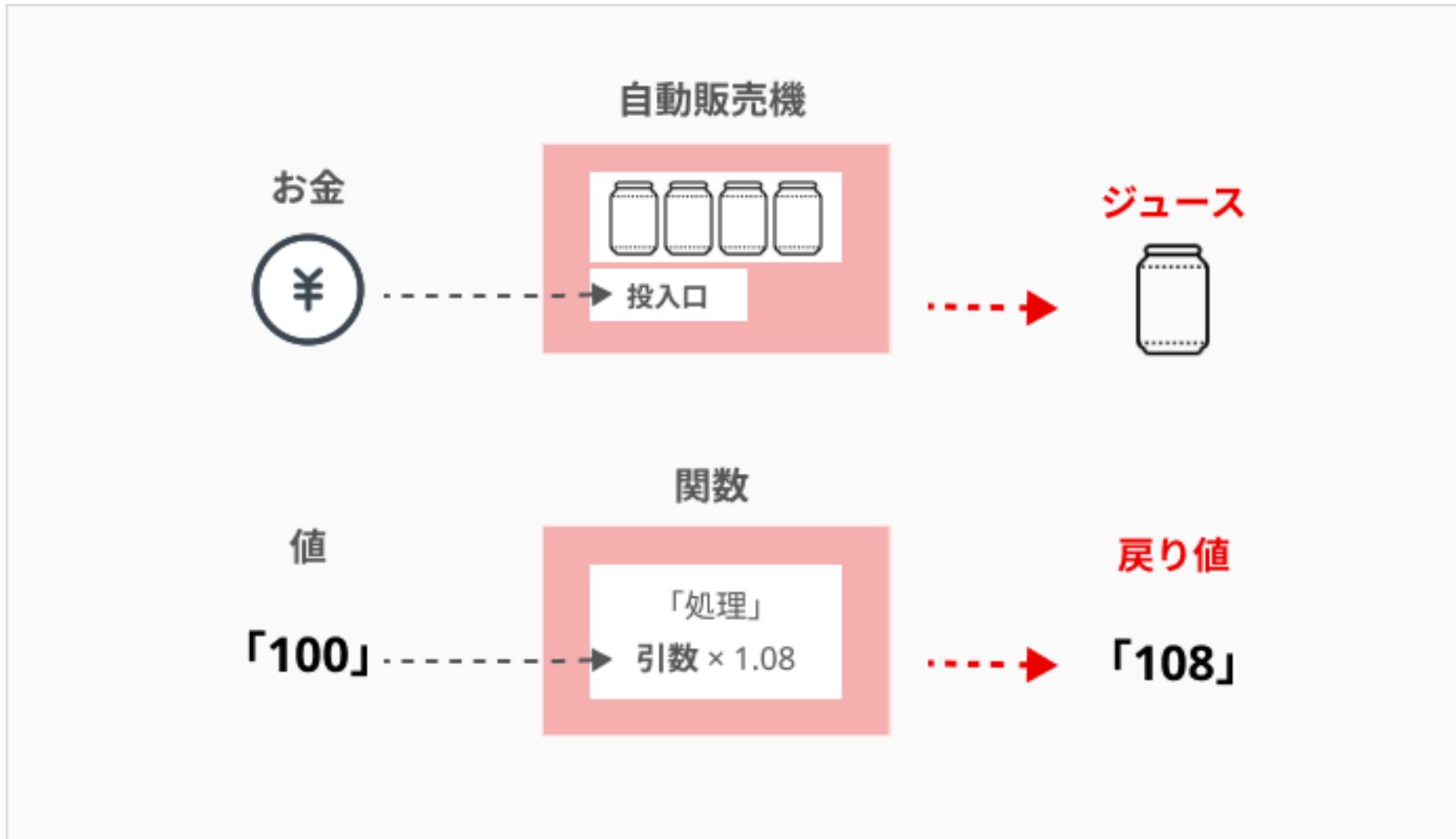
```
def add_two_arguments(price, shipping_fee):
    # 与えられた引数priceと引数shipping_feeを加算し、変数totalに代入する
    total = price + shipping_fee

    # 変数$totalの値を出力する
    print(f"{total}円")

    # 関数を呼び出し、引数として購入金額と送料を渡す
add_two_arguments(1200, 500);
```

1700円

# 12章 関数の戻り値



## | 関数から返ってくる値のことを戻り値（または返り値）といいます

- 戻り値を返さなければ関数だけで処理が終わる
- 戻り値を返すことで他のコードに活用できる
- `return`を記述すれば戻り値を返せる

# 12章 戻り値を使った関数の例

## | 商品が購入されたかどうかチェックする関数

```
# 購入済みかどうかを判定する変数
purchased = True

# 戻り値を返す関数を定義する
def is_purchased():
    if (purchased):
        return True
    else:
        return False

# 戻り値を返す関数を条件式に使う
if (is_purchased()):
    print("商品は購入済みです。")
```

## 12章 戻り値を使った関数の例



1  
秒

```
# 購入済みかどうかを判定する変数
```

```
purchased = True
```

```
# 戻り値を返す関数を定義する
```

```
def is_purchased():
    if (purchased):
        return True
    else:
        return False
```

```
# 戻り値を返す関数を条件式に使う(Trueであれば処理が実行される)
```

```
if (is_purchased()):
    print("商品は購入済みです。")
```

⇨ 商品は購入済みです。

## 本章では以下の内容を学習しました

### 引数

- 引数（ひきすう）とは、関数に与えるデータのこと
- 関数には複数の引数を渡すこともできる

### 戻り値

- 関数内でreturnを記述すれば、戻り値を返せる
- 戻り値を返すことで他のコードに活用できる

# 13章 変数のスコープを理解しよう

---

# 13章 変数のスコープを理解しよう

| 変数の有効範囲を知り、関数の内外で変数を正しく使う方法を学びます

## 本章の目標

- スコープ（変数の有効範囲）について知り、関数の内外で変数を正しく使えるようになること

# 13章 なぜスコープを学ぶのか

## 変数の有効範囲を知っておかないとエラーにつながります

- 中身が入っていない
- 想定とは異なる中身が入っている

関数の中で変数を使うときは、使える範囲（有効範囲）に注意が必要です

## | 変数を使える範囲（有効範囲）のことです

- 変数の値を取得したり、変更したりできる範囲
- 代表的なスコープは以下の2つ
  - ローカルスコープ：関数の中
  - グローバルスコープ：関数の外

# 13章 Pythonのスコープ (LEGB)

## | Pythonのスコープは4つに分かれます

| スコープ                 | 説明    |
|----------------------|-------|
| Local (ローカル)         | 関数の中  |
| Enclosing (エンクロージング) | 外側の関数 |
| Global (グローバル)       | 関数の外  |
| Built-in (ビルトイン)     | 組み込み  |

本章では代表的なローカルスコープとグローバルスコープを学びます

# 13章 ローカルスコープ

```
user_name = "侍花子"
```

グローバルスコープ

```
def show_user_name():
    user_name = "侍太郎"
    print(user_name)
```

ローカルスコープ

```
show_user_name()
print(user_name)
```

グローバルスコープ

## | ローカルスコープで定義された変数をローカル変数といいます

- ローカル変数はその関数の中でしか使えない
- 「関数の中で定義した変数はその関数の中でしか使えない」

# 13章 ローカル変数の正しい使い方

## | ローカルスコープの範囲内でローカル変数を使う

```
def show_user_name():
    # ローカル変数を定義する
    user_name = "侍太郎"

    # ローカルスコープの範囲内でローカル変数を使う
    print(user_name)

show_user_name()
```

# 13章 ローカル変数の正しい使い方

✓  
0  
秒



```
def show_user_name():
    # ローカル変数を定義する
    user_name = "侍太郎"

    # ローカルスコープの範囲内でローカル変数を使う
    print(user_name)

show_user_name()
```

侍太郎

# 13章 ローカル変数の間違った使い方

## | ローカルスコープの範囲外でローカル変数を使う

```
def show_user_name():
    # ローカル変数を定義する
    user_name = "侍太郎"
    print(user_name)

show_user_name()

# ローカルスコープの範囲外でローカル変数を使う（エラー）
print(user_name)
```

# 13章 ローカル変数の間違った使い方



```
def show_user_name():
    # ローカル変数を定義する
    user_name = "侍太郎"

    # ローカルスコープの範囲内でローカル変数を使う
    print(user_name)

show_user_name()

# ローカルスコープの範囲外でローカル変数を使う(エラーが発生する)
print(user_name)
```

侍太郎

```
NameError          Traceback (most recent call last)
<ipython-input-2-1b255860fdae> in <module>
      9
     10 # ローカルスコープの範囲外でローカル変数を使う(エラーが発生する)
--> 11 print(user_name)
```

NameError: name 'user\_name' is not defined

SEARCH STACK OVERFLOW

# 13章 ローカル変数の間違った使い方

## 「未定義の変数user\_name」 というエラーが表示されます

- 変数user\_nameは関数の中で定義したローカル変数
- ローカルスコープの範囲外（関数の外）では未定義として扱われる

# 13章 グローバルスコープ

```
user_name = "侍花子"
```

グローバルスコープ

```
def show_user_name():
    user_name = "侍太郎"
    print(user_name)
```

ローカルスコープ

```
show_user_name()
print(user_name)
```

グローバルスコープ

## | グローバルスコープで定義された変数をグローバル変数といいます

- グローバル変数はプログラム全体でアクセス可能
- 関数の中でも扱える

# 13章 グローバル変数の使い方

## グローバルスコープの範囲内でグローバル変数を使う

```
# グローバル変数を定義する
user_name = "侍花子"

def show_user_name():
    # ローカル変数を定義する
    user_name = "侍太郎"
    print(user_name)

show_user_name()

# グローバルスコープの範囲内でグローバル変数を使う
print(user_name)
```

# 13章 グローバル変数の使い方

✓  
0 秒

```
▶ # グローバル変数を定義する  
user_name = "侍花子"  
  
def show_user_name():  
    # ローカル変数を定義する  
    user_name = "侍太郎"  
  
    # ローカルスコープの範囲内でローカル変数を使う  
    print(user_name)  
  
show_user_name()  
  
# グローバルスコープの範囲内でグローバル変数を使う  
print(user_name)
```

侍太郎  
侍花子

# 13章 グローバル変数とローカル変数は別物

## | 同じ変数名でも、関数の中で値が更新されていません

- グローバル変数として定義した `user_name`
- ローカル変数として定義した `user_name`

これらは別物として扱われます

# 13章 関数の中でグローバル変数を使う

## | ローカル変数がない場合、グローバル変数が使われます

```
# グローバル変数を定義する
user_name = "侍花子"

def show_user_name():
    # ローカル変数を定義しない場合
    # user_name = "侍太郎"
    print(user_name)  # グローバル変数が使われる

show_user_name()
print(user_name)
```

# 13章 関数の中でグローバル変数を使う



```
# グローバル変数を定義する
user_name = "侍花子"

def show_user_name():
    # ローカル変数を定義する
    # user_name = "侍太郎"

    # グローバルスコープの範囲外でグローバル変数を使う(エラーは発生しない)
    print(user_name)

show_user_name()

# グローバルスコープの範囲内でグローバル変数を使う
print(user_name)
```

侍花子  
侍花子

## | 本章では以下の内容を学習しました

### スコープとは

- 変数を使える範囲（有効範囲）のこと
- ローカルスコープ（関数の中）とグローバルスコープ（関数の外）の2つに分けられる

| 本章では以下の内容を学習しました

## ローカル変数とグローバル変数

- ローカルスコープで定義された変数をローカル変数という
- グローバルスコープで定義された変数をグローバル変数という
- 同じ変数名でもローカル変数とグローバル変数は別物として扱われる
- 関数外でローカル変数を使った場合、エラーが発生する

## 14章 クラスを理解しよう

---

# 14章 クラスを理解しよう

クラスを理解し、似たようなモノ（オブジェクト）を大量生産する方法を学びます

## 本章の目標

- クラスを理解し、似たようなモノ（オブジェクト）を大量生産できるようになること
- クラスをより柔軟に使いこなせるようになること

# 14章 なぜクラスが必要なのか

## | 大量のデータをゼロから個別に作るのは大変です

- Amazonのようなショッピングサイトで大量の商品データを作る場合
- ゼロから個別の商品データを作るのは極めて大変
- クラスを使えば、類似するデータを大量に作れる

## オブジェクト



### 変数

- ・商品名
- ・価格
- ・カテゴリー
- ・在庫数 など

### 関数

在庫数を更新する関数 など

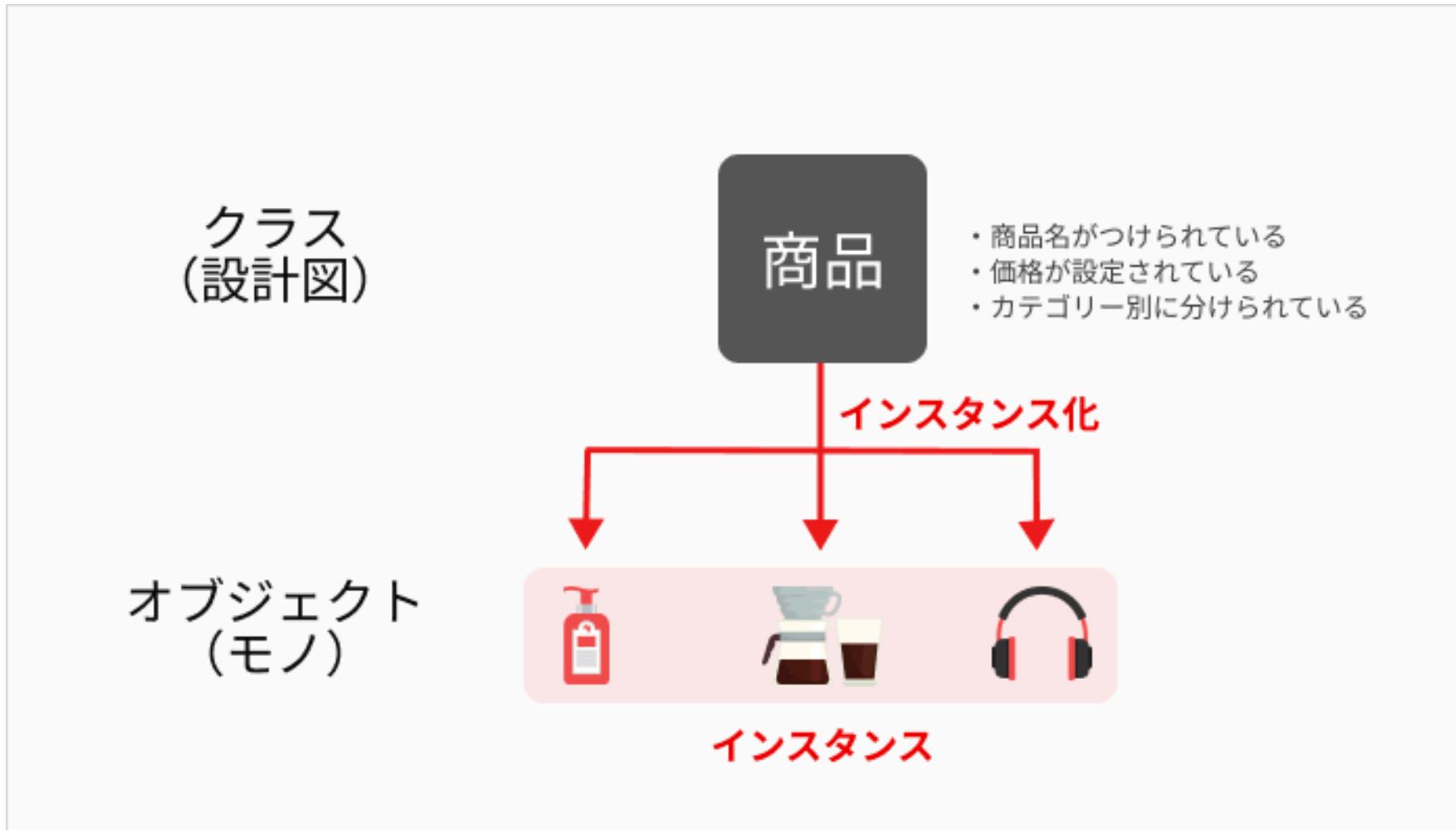
# 14章 クラスとは



## | クラス（設計図）から同じ特徴を持ったオブジェクトを大量生産できます

- ・ インスタンス：クラスをもとに作られたオブジェクトのこと
- ・ インスタンス化：クラスをもとにインスタンスを作ること
- ・ インスタンス＝「実体」という意味

# 14章 クラスとインスタンス



## | classキーワードを使ってクラスを定義します

```
class クラス名:  
    クラスの特徴
```

- クラス名は慣習的に先頭を大文字にする

```
class Product:  
    クラスの特徴
```

# 14章 インスタンス化する方法

「クラス名()」と記述してインスタンス化します

```
# クラスを定義する  
class Product:  
    クラスの特徴
```

```
# インスタンス化する  
shampoo = Product()
```

## オブジェクト



属性

### 変数

- ・商品名
- ・価格
- ・カテゴリー
- ・在庫数 など

### 関数

在庫数を更新する関数 など

## | クラス内のコンストラクタで属性を定義します

```
class Product:  
    def __init__(self):  
        # 属性を定義する  
        self.name = ""
```

- `def __init__(self):` はコンストラクタ（後述）
- `self.name` が属性

# 14章 属性へのアクセス方法

| インスタンスと属性名を「.（ドット）」でつなぎます

```
class Product:  
    def __init__(self):  
        self.name = ""  
  
shampoo = Product()  
  
# 属性にアクセスし、値を代入する  
shampoo.name = "シャンプー"  
  
# 属性にアクセスし、値を出力する  
print(shampoo.name)
```

# 14章 属性を使ってみよう

✓  
0  
秒

```
class Product:  
    def __init__(self):  
        # 属性を定義する  
        self.name = ""  
  
shampoo = Product()  
  
# 属性にアクセスし、値を代入する  
shampoo.name = "シャンプー"  
  
# 属性にアクセスし、値を出力する  
print(shampoo.name)
```

シャンプー

| オブ

## オブジェクト



### 変数

- ・商品名
- ・価格
- ・カテゴリー
- ・在庫数など

### メソッド

### 関数

在庫数を更新する関数など

# 14章 メソッドの書き方

## | クラス内でdefを使ってメソッドを定義します

```
class Product:  
    def __init__(self):  
        self.name = ""  
  
    # メソッドを定義する  
    def set_name(self, name):  
        self.name = name
```

# 14章 メソッドへのアクセス方法

| インスタンスとメソッド名を「. (ドット)」でつなぎます

```
class Product:  
    def __init__(self):  
        self.name = ""  
  
    def set_name(self, name):  
        self.name = name  
  
shampoo = Product()  
  
# メソッドにアクセスして実行する  
shampoo.set_name("シャンプー")
```

# 14章 メソッドを使ってみよう

## | 属性に値を代入・出力するメソッドを作成します

```
class Product:  
    def __init__(self):  
        self.name = ""  
  
    def set_name(self, name):  
        self.name = name  
  
    def show_name(self):  
        print(self.name)  
  
coffee = Product()  
coffee.set_name("コーヒー")  
coffee.show_name()
```

# 14章 メソッドを使ってみよう

```
✓ 0 秒
class Product:
    def __init__(self):
        self.name = ""

    # メソッドを定義する
    def set_name(self, name):
        self.name = name

    def show_name(self):
        print(self.name)

# インスタンス化する
coffee = Product()

# メソッドにアクセスして実行する
coffee.set_name("コーヒー")
coffee.show_name()
```

⇨ コーヒー

## | インスタンス化する際に処理を行うメソッドのことです

- コンストラクタによって実行される最初の処理を初期化という
- 「商品を作ると同時に出品する」といった最初の処理を設定できる
- constructor=「建設者」という意味

# 14章 コンストラクタの書き方

| **def \_\_init\_\_(self):** から始まります

```
class Product:  
    def __init__(self):  
        # 属性を定義する  
        self.name = ""
```

- `self` は初期化されるインスタンスそのものを指す
- 引数を追加して属性に値をセットできる

# 14章 コンストラクタで属性に値をセット

引数を使って属性に値を代入できます

```
def __init__(self, name, age, gender):
    # self.~~~は全てこのインスタンスの属性になる
    self.name = name
    self.age = age
    self.gender = gender
```

# 14章 コンストラクタを使ってみよう

## Userクラスを作成し、属性に値を代入します

```
class User:  
    def __init__(self, name, age, gender):  
        self.name = name  
        self.age = age  
        self.gender = gender  
  
# インスタンス化する  
user = User("侍太郎", 36, "男性")  
  
# 属性にアクセスし、値を出力する  
print(user.name)  
print(user.age)  
print(user.gender)
```

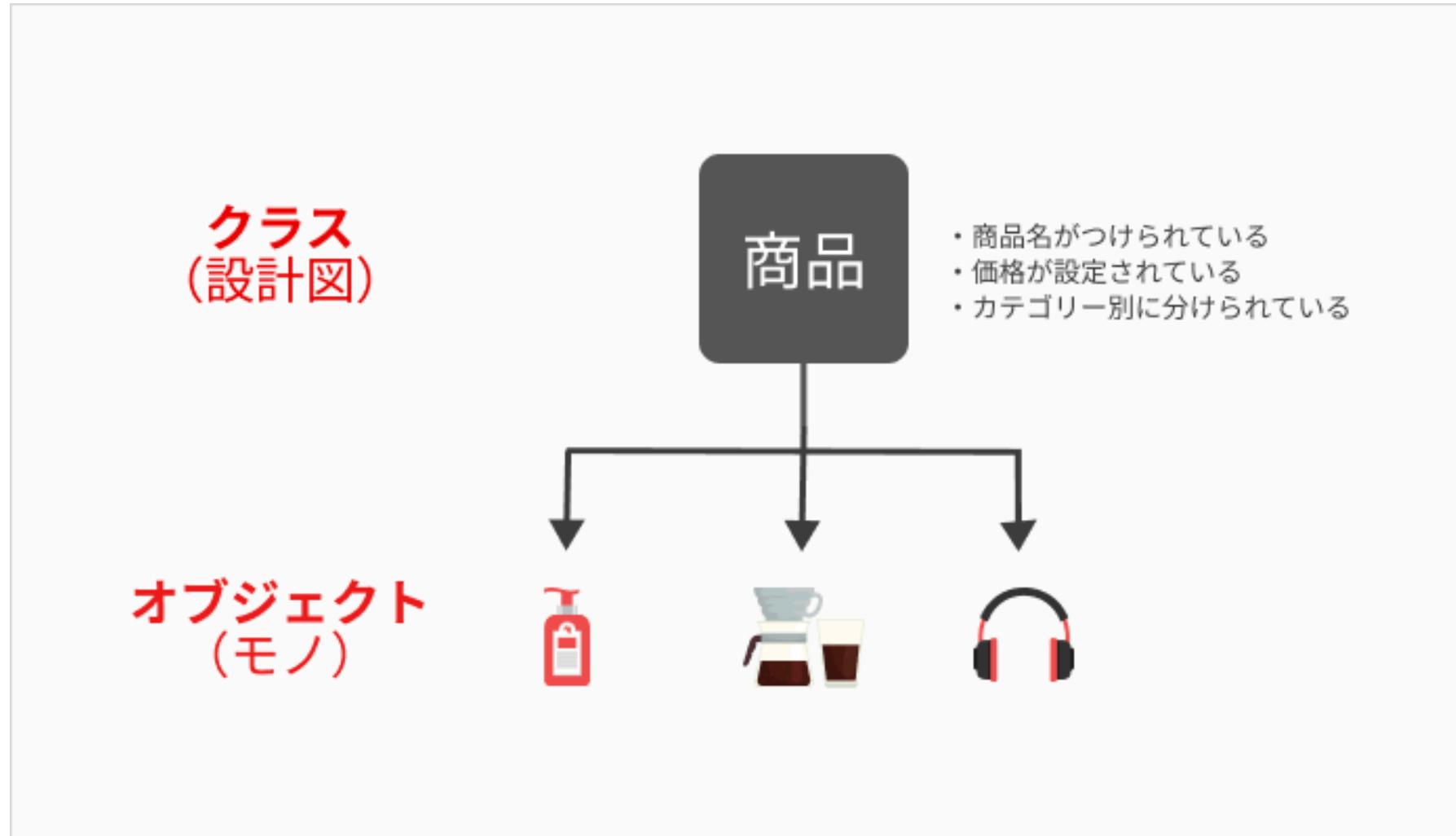
# 14章 コンストラクタを使ってみよう

✓ 0 秒

```
class User:  
    # コンストラクタを定義する  
    def __init__(self, name, age, gender):  
        self.name = name  
        self.age = age  
        self.gender = gender  
  
    # インスタンス化する  
user = User("侍太郎", 36, "男性")  
  
    # 属性にアクセスし、値を出力する  
print(user.name)  
print(user.age)  
print(user.gender)
```

□ 侍太郎  
36  
男性

# 14章 クラスのまとめ図解



## | 本章では以下の内容を学習しました

### オブジェクトとクラス

- オブジェクト：独自の変数や関数をひとまとめにしたもの
- クラス：モノ（オブジェクト）の設計図
- インスタンス：クラスをもとに作られたオブジェクト
- インスタンス化：クラスをもとにオブジェクトを作ること

## | 本章では以下の内容を学習しました

### 属性・メソッド・コンストラクタ

- 属性：オブジェクトが持つデータ（変数）
- メソッド：オブジェクトが持つ関数
- コンストラクタ：インスタンス化するときに初期化を行うメソッド

```
class クラス名:  
    def __init__(self):  
        初期化の内容  
  
    def メソッド名():  
        一連の処理
```

## 15章 日付・時刻の処理を理解しよう

---

# 15章 日付・時刻の処理を理解しよう

## | Pythonで日付や時刻を取得する方法を学びます

### 本章の目標

- Pythonで日付や時刻を取得する方法を知ること
- datetimeモジュールとtimedelta関数を使って日時の差を比較したり、計算したりする方法を知ること

# 15章 なぜ日時の処理を学ぶのか

| プログラミングでは日付や時刻を取得・計算する機会が多いです

- 学習記録アプリ
- スケジュール管理アプリ
- 日時の操作がメインになることも多い

# 15章 日時を取得する2つの方法

| Pythonで日時を取得する主な方法は2つあります

| モジュール         | 用途               |
|---------------|------------------|
| timeモジュール     | シンプルな時刻取得・経過時間計算 |
| datetimeモジュール | 日時の取得・比較・計算      |

モジュール=日時を扱うための便利な部品（次章で詳しく解説）

## | 現在の時刻を取得できます

```
# timeモジュールをインポート
import time

# 現在時刻をUNIXタイムスタンプで取得
print(time.time())

# 現在時刻を日時のフォーマットで取得
print(time.strftime("%Y年%m月%d日%H時%M分%S秒", time.localtime()))
```

# 15章 日時のフォーマット

## | さまざまな表示形式で日時を表示できます

- 「2015年3月19日」
- 「2015-03-19」
- 「2015/03/19 12:30:00」

strftime関数を使ってフォーマットを指定します

# 15章 フォーマット文字一覧

## 日時のフォーマットを指定する特殊な文字

| 文字 | 意味        | 例         |
|----|-----------|-----------|
| %Y | 年 (4桁)    | 2015、2022 |
| %m | 月 (01~12) | 01、12     |
| %d | 日 (01~31) | 01、31     |
| %H | 時 (00~23) | 00、23     |
| %M | 分 (00~59) | 00、59     |
| %S | 秒 (00~59) | 00、59     |

## Visual Studio Codeで日本時間を使う場合

```
!rm /etc/localtime  
!ln -s /usr/share/zoneinfo/Asia/Tokyo /etc/localtime  
!date
```

- タイムゾーン=同じ標準時を使う地域のこと
- 日本はAsia/Tokyo

# 15章 タイムゾーンの設定



The screenshot shows a terminal window with a light gray background. On the left side, there is a vertical toolbar with a green checkmark icon, a play button icon, and a '0 秒' (0 seconds) timer. The main area of the terminal contains three commands in blue text:

```
!rm /etc/localtime  
!ln -s /usr/share/zoneinfo/Asia/Tokyo /etc/localtime  
!date
```

Below these commands, the system's current date and time are displayed in black text: "Sat Sep 24 23:02:24 JST 2022".

# 15章 timeモジュールで日時を取得

## | フォーマットを指定して現在の日時を取得

```
# timeモジュールをインポート
import time

# 現在の日時を指定したフォーマットで出力する
print(time.strftime("%Y年%m月%d日%H時%M分%S秒", time.localtime()))
```

# 15章 timeモジュールで日時を取得

✓  
0  
秒



```
# timeモジュールをインポート
import time

# 現在の日時を指定したフォーマットで出力する
print(time.strftime("%Y年%m月%d日%H時%M分%S秒", time.localtime()))
```

2022年09月24日23時46分54秒

## | 1970年1月1日0時0分0秒からの経過秒数

- 例：「1426690800」「1652662122」
- インターネットの世界で共通の整数型の値として扱える形式

```
import time  
  
# 現在時刻をUNIXタイムスタンプで取得  
print(time.time())
```

## 15章 UNIXタイムスタンプ

✓  
0 秒



```
# timeモジュールをインポート  
import time  
  
# 現在時刻をUNIXタイムスタンプで取得  
print(time.time())
```

1664031600.9133644

# 15章 datetimeモジュール

| 日時の取得・比較・計算ができる便利なクラスが用意されています

| クラス名               | 用途       |
|--------------------|----------|
| datetime.datetime  | 日付と時刻    |
| datetime.date      | 日付       |
| datetime.time      | 時刻       |
| datetime.timedelta | 時間差・経過時間 |

# 15章 datetimeクラスの主な関数

## | 日時を取得・変換するための関数

| 関数名                   | 用途               |
|-----------------------|------------------|
| now()                 | 現在時刻を取得          |
| strptime(文字列, フォーマット) | 文字列をdatetimeに変換  |
| strftime(フォーマット)      | datetimeから文字列に変換 |

# 15章 datetimeモジュールの使い方

## 日時を取得・変換する

```
import datetime

# 現在の日時を取得する
now = datetime.datetime.now()

# 指定した日付を取得する
date = datetime.datetime.strptime("2015-03-19", "%Y-%m-%d")

# 指定した日時を取得する
date_time = datetime.datetime.strptime("2015-03-19 12:15:30", "%Y-%m-%d %H:%M:%S")
```

# 15章 datetimeモジュールを使ってみよう

## 日時を特定のフォーマットで出力する

```
import datetime

# 指定した日時を取得する
date_time = datetime.datetime.strptime("2015-03-19 12:15:30", "%Y-%m-%d %H:%M:%S")

# 取得した日付date_timeの日時を特定のフォーマットで出力する
print(date_time.strftime("%Y年%m月%d日%H時%M分%S秒"))
```

# 15章 datetimeモジュールを使ってみよう

✓  
0  
秒

```
# datetimeモジュールをインポート
import datetime

# 指定した日時を取得する
date_time = datetime.datetime.strptime("2015-03-19 12:15:30", "%Y-%m-%d %H:%M:%S")

# 取得した日付date_timeの日時を特定のフォーマットで出力する
print(date_time.strftime("%Y年%m月%d日%H時%M分%S秒"))
```

2015年03月19日12時15分30秒

# 15章 年月日時分秒を個別に取り出す

| 取得した日付の各要素にアクセスできます

```
import datetime

date_time = datetime.datetime.strptime("2015-03-19 12:15:30", "%Y-%m-%d %H:%M:%S")

print(date_time.year)      # 年
print(date_time.month)     # 月
print(date_time.day)       # 日
print(date_time.hour)      # 時
print(date_time.minute)    # 分
print(date_time.second)    # 秒
```

# 15章 年月日時分秒を個別に取り出す

```
✓ 0 秒
▶ # datetimeモジュールをインポート
import datetime

# 指定した日時を取得する
date_time = datetime.datetime.strptime("2015-03-19 12:15:30", "%Y-%m-%d %H:%M:%S")

print(date_time)

# 年,月,日,時,分,秒だけをそれぞれ取り出すことも可能です
print(date_time.year)
print(date_time.month)
print(date_time.day)
print(date_time.hour)
print(date_time.minute)
print(date_time.second)
```

```
2015-03-19 12:15:30
2015
3
19
12
15
30
```

# 15章 timedeltaクラス

| 2つの日時の差を比較できます

| 属性            | 内容         |
|---------------|------------|
| days          | 日数         |
| seconds       | 秒数（日数分含まず） |
| microseconds  | マイクロ秒      |
| total_seconds | トータルの秒数    |

# 15章 2つの日時の差を比較する

datetimeオブジェクト同士を引き算するとtimedeltaオブジェクトが得られます

```
import datetime

now = datetime.datetime.now()
date_time = datetime.datetime.strptime("2015-03-19 12:15:30", "%Y-%m-%d %H:%M:%S")

# 2つのdatetimeの差を取得
interval = now - date_time

print(f"総日数は{interval.days}日です。")
```

# 15章 2つの日時の差を比較する

✓  
0  
秒

```
# datetimeモジュールをインポート
import datetime

# 現在時刻を取得する
now = datetime.datetime.now()

# 指定した日時を取得する
date_time = datetime.datetime.strptime("2015-03-19 12:15:30", "%Y-%m-%d %H:%M:%S")

# 2つのdatetimeの差を取得し、変数intervalに代入する(interval=間隔)
interval = now - date_time

# 取得した日時の差を特定のフォーマットで出力する
print(f"総日数は{interval.days}日です。")
```

総日数は2746日です。

# 15章 日時を加算・減算する

| **timedeltaオブジェクトを使って日付を計算できます**

```
import datetime

now = datetime.datetime.now()

# 1年間の日数差を作成
td_1y = datetime.timedelta(days=365)

# 現在の日時に1年を加算
add = now + td_1y

# 3日間の日数差を作成
td_3d = datetime.timedelta(days=3)

# 現在の日時から3日を減算
sub = now - td_3d
```

# 15章 日時を加算・減算する

✓  
0  
秒

```
# datetimeモジュールをインポート
import datetime

# 現在時刻を取得する
now = datetime.datetime.now()

# 1年間の日数差を変数td_1yに代入する
td_1y = datetime.timedelta(days=365)

# 現在の日時に1年を加算し、変数addに代入する
add = now + td_1y

# 3日間の日数差を変数td_3dに代入する
td_3d = datetime.timedelta(days=3)

# 現在の日時から3日を減算し、変数subに代入する (sub=「減算」を意味するsubtractionの略)
sub = now - td_3d

# 加算・減算した日時を特定のフォーマットで出力する
print(add.strftime("現在から1年後は%Y年%m月%d日%H時%M分%S秒です。"))
print(sub.strftime("現在から3日前は%Y年%m月%d日%H時%M分%S秒です。")))
```

現在から1年後は2024年04月13日05時40分27秒です。  
現在から3日前は2023年04月11日05時40分27秒です。

# 15章 timeモジュールとdatetimeモジュールの使い分け

## | 用途によって使い分けます

| 用途           | 使うモジュール       |
|--------------|---------------|
| 現在の日時を取得     | どちらでもOK       |
| プログラムの実行時間計算 | timeモジュール     |
| 日時の比較や計算     | datetimeモジュール |

## | 本章では以下の内容を学習しました

### timeモジュールとdatetimeモジュール

- timeモジュール：現在時刻、UNIXタイムスタンプを取得
- datetimeモジュール：日時の取得・比較・計算が可能
- フォーマット文字を使ってさまざまな形式で日時を表示できる

| 本章では以下の内容を学習しました

## timedeltaクラス

- 2つの日時の差を比較できる
- 日付の加算・減算が可能

| 属性            | 内容      |
|---------------|---------|
| days          | 日数      |
| seconds       | 秒数      |
| total_seconds | トータルの秒数 |

# 16章 パッケージ・モジュールについて理解しよう

---

# 16章 パッケージ・モジュールについて理解しよう

## | Pythonにおけるモジュール、パッケージについて学びます

### 本章の目標

- Pythonにおけるモジュール、パッケージについて理解する
- モジュールやパッケージを利用するためのimport文について理解する

## | 1つのスクリプトファイルを「モジュール」と呼びます

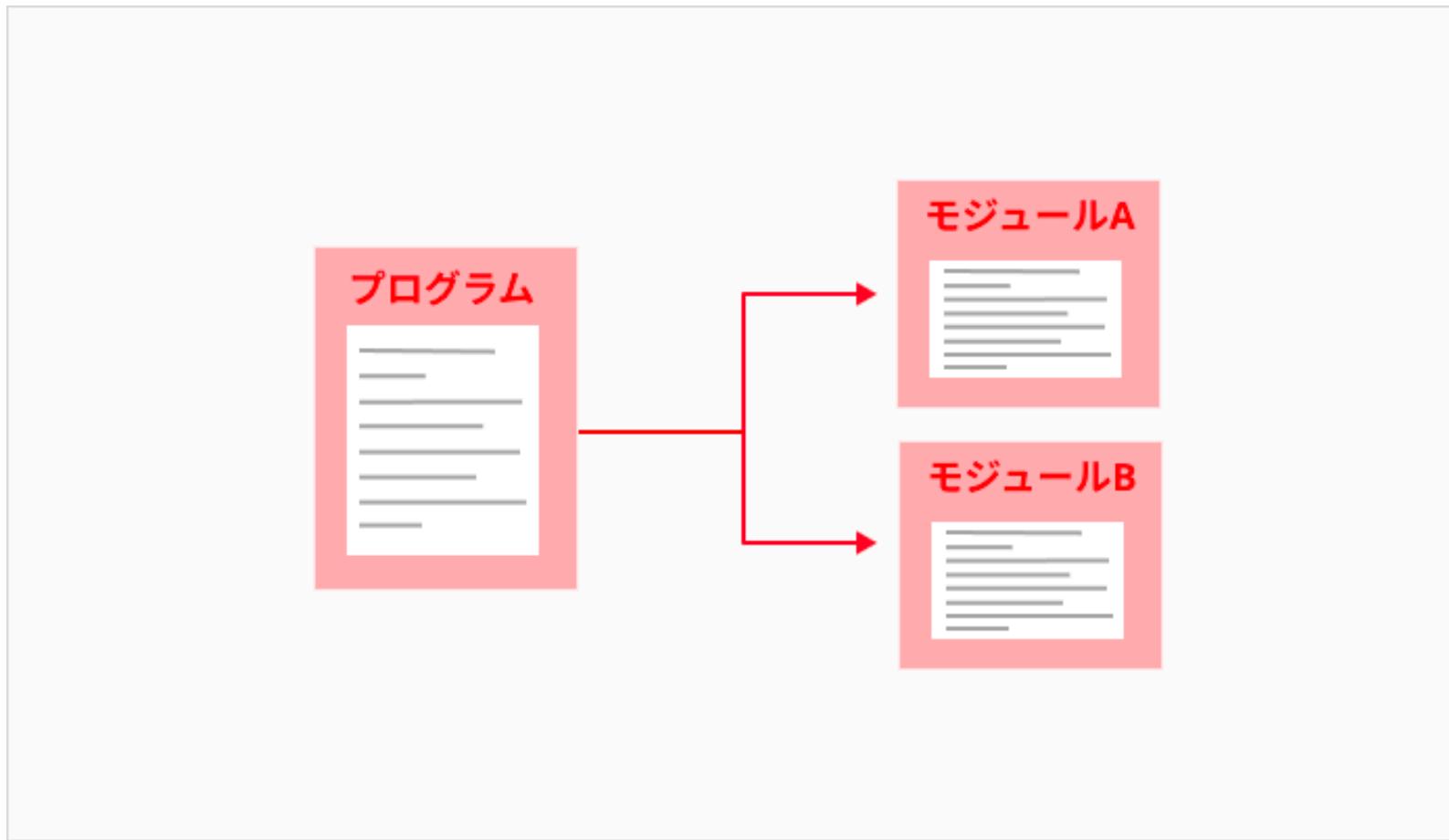
- 開発を進めると、多くの機能を実装するためファイルが肥大化していく
- 1つのファイルに大量のコードがあると、処理の影響が見えづらくなる
- 思わぬミスやエラーの原因になりかねない

# 16章 モジュールで機能を分割する

## 機能ごとにファイルを分けて整理できます

- 1つの大きなスクリプトファイルを機能ごとに複数のファイルに分解
- この1つひとつのスクリプトファイルをモジュールという
- モジュールは基本的にファイル名に `.py` をつけて管理する

# 16章 モジュールのイメージ



## モジュールは組み合わせて使えます

- モジュールは他のモジュールから呼び出せる
- パズルのように組み合わせて使える
- 再利用性が高く、効率的な開発が可能

## 配布用にまとめられたソフトウェアのことです

- 第三者が利用できるように、配布用としてひとまとめに梱包されたソフトウェア
- 第三者が作ったパッケージを利用することで、効率的に開発できる
- 反対に、第三者が使えるパッケージを開発して貢献することも可能

## | 標準ライブラリと外部ライブラリがあります

### | 標準ライブラリ

- Pythonに初めから実装されている
- インストール不要
- importで利用可能

### | 外部ライブラリ

- 有志が作成したライブラリ
- pipでインストールが必要
- 標準でカバーできない部分を補う

# 16章 代表的な標準ライブラリ

| Pythonに初めから実装されているパッケージです

| パッケージ名   | 利用用途                     |
|----------|--------------------------|
| sys      | プログラムの実行・停止、ファイルの作成・移動など |
| math     | 三角関数、対数、円周率などの計算機能を提供    |
| datetime | 日付の変換や日付同士の計算（第16章で解説）   |

# 16章 代表的な外部ライブラリ①

## 数値計算・統計処理に関するライブラリを紹介します

| パッケージ名     | 利用用途               |
|------------|--------------------|
| Numpy      | 配列や行列といった科学技術計算を行う |
| matplotlib | 計算したデータをグラフとして表示   |

# 16章 代表的な外部ライブラリ②

## 数値計算・統計処理に関するライブラリを紹介します

| パッケージ名  | 利用用途                  |
|---------|-----------------------|
| pandas  | 統計用のデータ処理、統計量の表示、グラフ化 |
| sklearn | 機械学習（分類、回帰、クラスタリングなど） |

# 16章 パッケージ管理システム：pip

| 外部ライブラリをインストールするためのツールです

- 外部ライブラリ・パッケージを利用するにはpipを使うのが一般的
- pipはPythonに標準で付属しているパッケージ管理システム

# 16章 pipの使い方

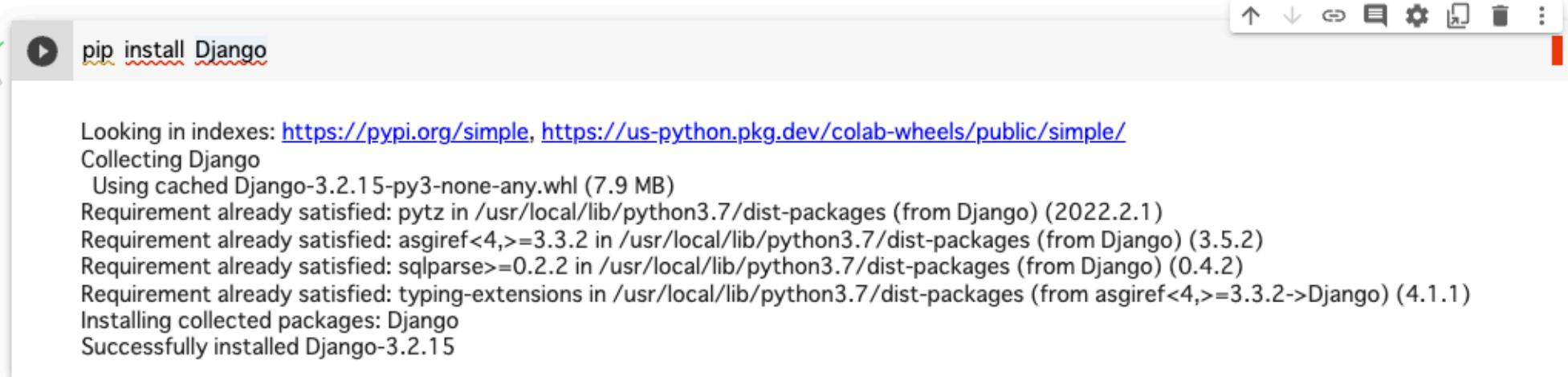
| パッケージをインストールするコマンドです

```
pip install パッケージ名
```

- ターミナルまたはVisual Studio Codeで実行
- 例：Djangoをインストールする場合

```
pip install Django
```

# 16章 pip installの実行例



A screenshot of a terminal window titled "pip install Django". The window has a progress bar at the top indicating the command is running. The output of the command is displayed below:

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting Django
  Using cached Django-3.2.15-py3-none-any.whl (7.9 MB)
Requirement already satisfied: pytz in /usr/local/lib/python3.7/dist-packages (from Django) (2022.2.1)
Requirement already satisfied: asgiref<4,>=3.3.2 in /usr/local/lib/python3.7/dist-packages (from Django) (3.5.2)
Requirement already satisfied: sqlparse>=0.2.2 in /usr/local/lib/python3.7/dist-packages (from Django) (0.4.2)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from asgiref<4,>=3.3.2->Django) (4.1.1)
Installing collected packages: Django
Successfully installed Django-3.2.15
```

# 16章 import文とは

## モジュールやパッケージを利用するための文です

- 自作のモジュールや標準ライブラリを利用するときに使用
- インストールしたパッケージ内のモジュールを利用するときにも使用
- Pythonのスクリプトファイルの先頭に記述する

# 16章 import文の書き方

## importでモジュールを読み込みます

```
import django
print(django.get_version())
# 3.0.6
```

- `import モジュール名` でモジュールを読み込む
- `モジュール名.関数名()` で関数を呼び出す

## | 本章では以下の内容を学習しました

### モジュールとパッケージ

- モジュール：1つひとつのスクリプトファイル (.pyファイル)
- パッケージ：配布用にまとめられたソフトウェア
- モジュールは組み合わせて使うことで効率的に開発できる

## | 本章では以下の内容を学習しました

### ライブラリとpip

- 標準ライブラリ：Pythonに組み込まれている（datetime, math等）
- 外部ライブラリ：有志が作成（Numpy, pandas, sklearn等）
- pip：外部ライブラリをインストールするためのツール
- import文：モジュールやパッケージを利用するため記述