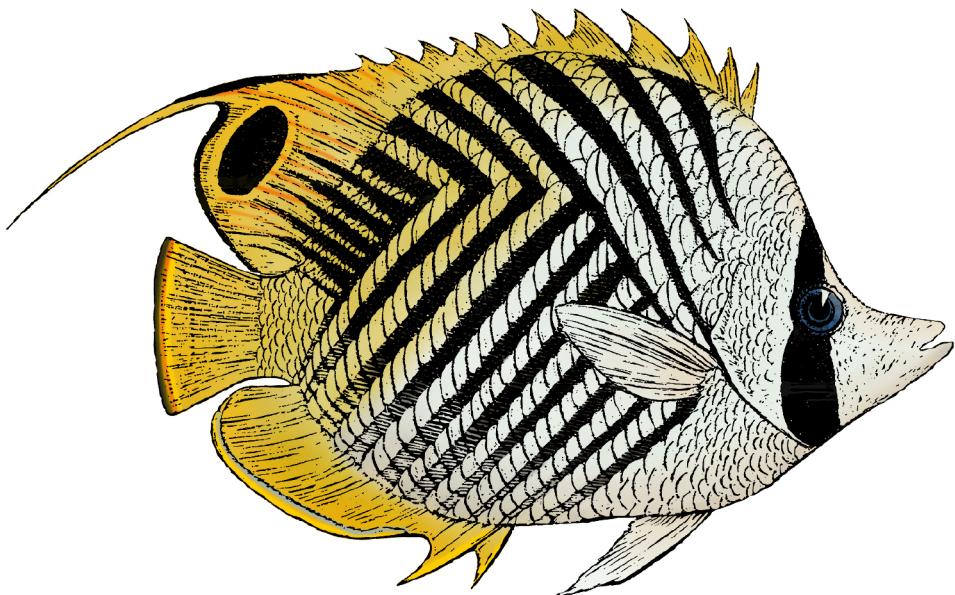


O'REILLY®  
オライリー・ジャパン

# データサイエンスのための ソフトウェア エンジニアリング入門

明快かつ速いコードを書く



Catherine Nelson 著  
漆戸 パティ 訳

本書で使用するシステム名、製品名は、いずれも各社の商標、または登録商標です。  
なお、本文中では™、®、©マークは省略している場合もあります。

---

# Software Engineering for Data Scientists

*From Notebooks to Scalable Systems*

*Catherine Nelson*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

©2025 O'Reilly Japan, Inc. Authorized Japanese translation of the English edition of Software Engineering for Data Scientists.

©2024 Catherine Nelson. All rights reserved. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

本書は、株式会社オライリー・ジャパンが O'Reilly Media, Inc. の許諾に基づき翻訳したものです。日本語版についての権利は、株式会社オライリー・ジャパンが保有します。

---

日本語版の内容について、株式会社オライリー・ジャパンは最大限の努力をもって正確を期していますが、本書の内容に基づく運用結果について責任を負いかねますので、ご了承ください。

# 賞賛の声

この本は、データサイエンティストが長年求めてきた「ミッシングリンク」であり、データサイエンスとソフトウェアエンジニアリングの間のギャップを巧みに埋めるものだ。ソフトウェアエンジニアリングにおける重要なスキルギャップを埋めるための明確で実践的なガイドを提供し、データサイエンティストのコーディングプラクティスを新たな高みへと引き上げる。まさに、私たちが待ち望んでいた本だ。

—— **Gabriela de Queiroz** Director of AI, Microsoft;  
Startup Advisor and Angel Investor

Catherine の著書は、個人的作業を「生産能力」にスケールアップする方法を解き明かしている。データサイエンティスト、開発者、または経営者のいずれであっても、スケーラブルなデータサービスへのアクセスを可能してくれる。スタートアップから大規模な企業まで、「ベストプラクティス」に従うことで、データプロジェクトの成功を確実なものにしてくれる。

—— **Carol Willing** Python 開発の主力メンバー  
Jupyter の開発により 2017 ACM Software System Award を受賞

この本は最高！ データサイエンティストの本棚に欠けていた最後の一冊だ。教育機関や業界のマネージャーたちは、ソフトウェアエンジニアのようなスキルを持ったデータサイエンティストをずっと探し求めてきたが、両者をつないでくれる本はこれまで存在しなかった。本書はソフトウェアエンジニアとデータサイエンティストをつなぐ「架け橋」となってくれるに違いない。

—— **Shawn Ling Ramirez** CEO, eloraHQ

『データサイエンスのためのソフトウェアエンジニアリング入門』は、データサイエンスのスキルを「アイデア」から「完璧に実装されたシステム」へと発展させたい人にとって必読の書だ。コーディングにおける最も重要なエンジニアリング的側面をナビゲートするための素晴らしいガイドである。この本を数年前に持っていたら、かなりの時間を節約できただろう。心からお勧めする。

—— **Laurence Moroney** AI Advocacy Lead, Google

データサイエンティストのバックグラウンドは多様だ。この多様性は多くの点でこの分野の強みとなってきたが、ソフトウェア開発のバックグラウンドを持つ同僚と密接に協力するためのソフトウェアエンジニアリング面のスキルが不足しているサイエンティストが少なくないことも事実だ。Catherine Nelson は本書で、強く求められていた両分野の架け橋を提供し、データサイエンティストが自身の仕事をレベルアップし、影響力を高めるための知識を伝授してくれている。

—— **Chris Albon** Director of Machine Learning,  
The Wikimedia Foundation

# まえがき

データサイエンスは「コード」によって行われる。機械学習システムの構築、データの分析、データ分布の可視化、統計分析……いずれについても、コーディング技術が必須である。ソフトウェア製品の開発に携わっているのなら、こうしたスキルは、うまく動作し、維持可能なコードを書くために不可欠だ。最終製品を開発するチームで働いていなくとも、他のデータサイエンティストが容易に利用できる、より堅牢で再現可能なコードを書くことは非常に有益である。また、一人で働いている場合でも、<sup>コードプラクティス</sup>良い習慣はコーディング速度を上げる手助けをしてくれる。<sup>エンジニアリング</sup>

正直なところ、筆者は以前から「ソフトウェア工学」の大切さを認識していたわけではない。最初に参加したチームで、筆者は唯一のデータサイエンティストだった。同僚はソフトウェアエンジニアとデザイナーで、他のデータサイエンティストから学ぶ機会がないため、スキル向上に苦労するだろうと心配していた。その懸念を同僚の開発者に伝えたところ、「より良いコードの書き方を学べば、データサイエンスにもプラスになるはずだ」と答えてくれた。この言葉は筆者の心に深く刻まれ、その後、ソフトウェアエンジニアリングのスキル向上が、データサイエンティストとしての成長にも欠かせないと気づいたのだ。これにより、チームのメンバーが利用しやすいコードを、そして自分が数か月後にプロジェクトに戻ったときにも簡単に改良できるコードを、書けるようになった。

本書の目的は、より良いデータサイエンスのコードを書くためのガイド役となることだ。テスト、エラー処理、ログなど、一般的なタスクにおける「最良の習慣」を説明する。プロジェクトが成長しても保守しやすく、堅牢なコードを書く方法を解説する。他人が利用しやすいコードの作り方を示し、本書を読み終える頃には、データサイエンスのコードを大規模なコードベースに統合できるようになる。

「生成 AI の時代には、ソフトウェアエンジニアリングのスキルは役に立たない」と

思っている人がいるかもしれない。「ChatGPT がコードを書いてくれるよ」と。しかし、AI アシスタントでコーディングを高速化できる時代でも、本書の内容は依然として有用なはずだ。本書でこれから示すように、コーディング時には常に数多くの選択肢が存在し、なぜ「このコードを選択するのか」を説明できなければならない。AI アシスタントが提示するコードを評価し、取り組んでいる問題に対して適切な選択をしているか検証する必要があるのだ。本書はそのための手伝いをする。

## 対象読者

本書はデータサイエンティストを対象としているが、データアナリスト、機械学習(ML) エンジニア、データエンジニアなど、関連分野で働く人々にも役に立つ内容となっている。筆者は、コードを書く人全員にとって役立つ確立されたソフトウェアエンジニアリングの原則を説明するが、例としては、データサイエンス分野のものを用いる。

比較的最近この世界に入ってきたデータサイエンティストを対象にしている。データサイエンスの学位を取得したばかりの人や、この業界で最初の職を得たといった人だ。本書は、入門的なデータサイエンスのコースには、多くの場合、含まれていない実践的なソフトウェアエンジニアリングのスキルをカバーしている。あるいは、正式なデータサイエンスのコースを受講しておらず、独学で学んだり、数学など他分野からデータサイエンスの世界に移ってきた人などだ。データサイエンスへの道のりがどうであれ、本書はそういった人のためのものだ。

経験豊富なデータサイエンティストも、本書から多くの学びを得られるだろう。特にソフトウェア開発者と頻繁に連携する仕事に従事している場合、本書は役立つはずだ。大規模なコードベースで効果的に働くためのスキルや、本番環境で効率的に動作する Python コードの書き方を学ぶことができる。

本書では、データサイエンスの基礎知識（データ探索、データ可視化、データ整形、基本的な機械学習、およびこれらに関連する数学的スキル）を理解しているものと仮定している。また、Python でのプログラミングの基礎（関数の書き方、制御構造の記述方法、NumPy、Matplotlib、pandas、scikit-learn などのモジュールの基本的な使い方）もすでに理解しているものと仮定している。こうした事柄を学んだことがない人には、次の書籍を推奨する。

- Jake VanderPlas 著『Python データサイエンスハンドブック 第 2 版』（オラ

イリー・ジャパン)

- Joel Grus 著『ゼロからはじめるデータサイエンス 第2版』(オライリー・ジャパン)
- Sam Lau + Joseph Gonzalez + Deborah Nolan 著『Learning Data Science』(O'Reilly Media)

本書は、データサイエンスと機械学習のスキルを学びたいソフトウェア開発者向けのものではない。そういう本を探しているのならば、Laurence Moroney 著『動かして学ぶ AI・機械学習の基礎』(オライリー・ジャパン) を手に取るとよいだろう。

### ソフトウェアエンジニアリング vs. データサイエンス

この段階で、データサイエンスとソフトウェアエンジニアリングの「考え方の違い」をはっきりさせておくことが有用だろう。データサイエンティストは、探索、発見、仮説検証といった科学的プロセスを重視するバックグラウンドを持っている。プロジェクトの最終結果は、開始時にわかっているものではない。一方、一般的なソフトウェアエンジニアリングは、何を構築するか計画し、最も適切な構築方法をデザインし、その後計画したものを構築するためのコードを書くプロセスだ。プロジェクトに期待される結果は、プロジェクトの開始時点ですでにわかっているのが基本だ。ソフトウェアエンジニアリングの実践においては、標準化と自動化を重視する。データサイエンティストは、エンジニアリング的な要素を活用することで、コードの品質を向上させることができる。このテーマについては、「1章 良いコードとは何か?」で詳しく説明する。

## なぜ Python か?

本書のすべてのコード例は Python で書かれており、多くの章では Python 固有のツールについて説明している。近年、Python はデータサイエンス分野で最も人気のあるプログラミング言語となった。次の引用は、Anaconda が 2021 年に 3,000 人を超えるデータサイエンティストを対象に行った調査 (<https://oreil.ly/kmmBp>) からのものだ。

回答者の 63% が Python を常にまたは頻繁に使用していると回答し、今年度の調査で最も人気のある言語となった。さらに、71% の教育者が Python を教えており、88% の学生がデータサイエンス/機械学習分野に進むための準備として Python を学んだと報告している。

Python はデータサイエンス向けのオープンソースライブラリが非常に充実しており、強力なサポートと活発なメンテナンスコミュニティを有している。トレンドをリードする大手企業は、主要な機械学習フレームワークとして Python を選択している。例えば、TensorFlow (Google) や PyTorch (Meta) などだ。このため、本番環境での機械学習コード開発において、優れたコーディングスキルが重要なデータサイエンティストの間で、Python は特に人気が高い。

筆者の経験では、Python コミュニティは親切で新規参入者にもやさしい雰囲氣があり、スキル向上に役立つ多くの優れたイベントが開催されている。そして、Python は筆者が大好きなプログラミング言語でもある。「本書の言語」として Python を選ぶのは自然な選択であった。

## 本書に書かれていない事柄

先ほど「対象読者」で述べたように、これはデータサイエンスやプログラミングの入門書ではない。また、次に挙げるトピックは本書には含まれていない。

### Python のインストール方法

本書では、Python（バージョン 3.9 以降）がすでにインストールされており、VS Code や PyCharm などのコードを記述できる IDE（統合開発環境）がインストールされていることを前提としている。Python のインストール方法については説明しないが、「10 章 コードの共有」で仮想環境の設定方法を説明する

### Python 以外のプログラミング言語

本書では、「なぜ Python か？」で説明した理由により、Python のみを扱う。R、Julia、SQL、MATLAB などの他の言語の例は含まれていない

### コマンドラインスクリプト

コマンドラインスクリプトシェル（shellスクリプト）は、ファイルやテキストを操

作するための強力な方法だ。Jeroen Janssens 著『コマンドラインではじめるデータサイエンス』(オライリー・ジャパン)など、他の参考文献で詳細に解説されているため、本書では取り上げない

### Python の上級プログラミング

本書の例題は比較的シンプルなコードが多い。より高度な Python のコーディングについては、Patrick Viafore 著『ロバスト Python』(オライリー・ジャパン)をお勧めする

## 本書のロードマップ

本書では、まず個々の関数の記述レベルでのベストプラクティスを概説し、続いてコーディングを改善する方法について詳しく説明する。後ろの章では、作成したコードを他のユーザーが使いやすくする方法、そして「デプロイ」に関する一般的なテクニックとソフトウェア開発におけるベストプラクティスについて説明する。

本書は 15 の章に分かれている。各章の概要は次のとおりである。

### 1 章 良いコードとは何か？

シンプルでモジュール化され、読みやすく、効率的で堅牢なコードの書き方の基本を紹介する

### 2 章 パフォーマンスの分析

コードのパフォーマンスを測定する方法と、データサイエンスのコードをより効率的に実行するための手法をいくつか説明する

### 3 章 データ構造

使用するデータ構造の選択に伴うトレードオフについて説明する。選択するデータ構造は、コードの効率に大きな違いをもたらす可能性がある

### 4 章 オブジェクト指向プログラミングと関数型プログラミング

プログラミングスタイルの基本について説明する。正しく使用すれば、構造化され効率的なコードを書くことができる

### 5 章 エラー、ログ、デバッグ

コードがうまく動作しない場合の対処法、「有用なエラー」の生成方法、そしてエラーの原因を特定するための戦略について説明する

## 6章 コードのフォーマット、linting、型チェック

自動化ツールを使って、コードを標準化する方法を説明する

## 7章 テスト

テストを通じて入力の変化に対してコードを堅牢にする方法を説明する。保守しやすいコードを書くための重要なステップだ

## 8章 デザインとリファクタリング

プロジェクトを、標準化された一貫性のあるテクニックを用いて構造化する方法と、ノートブックからスクリプトに移行する方法について説明する

## 9章 ドキュメンテーション

名前の付け方やコメントに関するベストプラクティスなど、コードを他の人が読みやすくするための手法を説明する

## 10章 コードの共有

Git を使ったバージョン管理の基本と、仮想環境でのプロジェクトの依存関係の管理方法について説明する。また、スクリプトを Python パッケージに変換する手順も示す

## 11章 API

API の概念とその利用方法を紹介し、FastAPI を用いた API 公開の例を示す

## 12章 自動化とデプロイ

コードのデプロイの基本、CI/CD や GitHub Actions を利用してコードのデプロイを自動化する方法、Docker コンテナでコードをクラウド環境にデプロイする方法を説明する

## 13章 セキュリティ

一般的なセキュリティリスク、こうしたリスクの軽減方法、そして機械学習特有のセキュリティ関連の脅威について説明する

## 14章 ソフトウェア開発の現場

アジャイル開発手法を含むソフトウェア開発チームの一般的なプラクティスやソフトウェアチームにおける一般的な役割について説明するとともに、より広範なコミュニティについても紹介する

## 15章 さらなる飛躍を求めて

最後に、コーディングが将来どのように変化していくかに関する考察と、「次に何ができるか」の提案を行う

## 章の依存関係

必ずしも、本書の各章を順番に読む必要はないが、「1章 良いコードとは何か？」から始めるのがよいだろう。1章では、優れたコードの書き方の基本を説明し、本書の残りの部分でより詳細に扱うトピックを紹介する。また、本書全体で使用するコード例もいくつか提示する。

「1章 良いコードとは何か？」に続く章の多くは、次に挙げる例外を除き、独立した内容になっている。

- 「3章 データ構造」を読む前に、「2章 パフォーマンスの分析」を読むことを推奨する
- 「6章 コードのフォーマット、linting、型チェック」「7章 テスト」「10章 コードの共有」「11章 API」は、「12章 自動化とデプロイ」を読む前に必ず読むこと

一部の章には、機械学習関連のトピックをより深く掘り下げた節が含まれている（節名に「機械学習」が含まれている）。機械学習に関心が（あまり）ない場合は、こうした節を飛ばしても、章の残りの部分を理解するために必要な情報を見落とすことはない。

## 表記上のルール

本書では、次に示す表記上のルールに従う。

### 太字（Bold）

新しい用語、強調やキーワードフレーズを表す。

### 等幅（Constant Width）

プログラムのコード、コマンド、配列、要素、文、オプション、スイッチ、変数、属性、キー、関数、型、クラス、名前空間、メソッド、モジュール、プロパティ、パラメーター、値、オブジェクト、イベント、イベントハンドラ、XML

タグ、HTML タグ、マクロ、ファイルの内容、コマンドからの出力を表す。その断片（変数、関数、キーワードなど）を本文中から参照する場合にも使われる。

#### 等幅太字（Constant Width Bold）

ユーザーが入力するコマンドやテキストを表す。コードを強調する場合にも使われる。

#### 等幅イタリック（Constant Width Italic）

ユーザーの環境などに応じて置き換えなければならない文字列を表す。



ヒントや示唆を表す。



興味深い事柄に関する補足を表す。



ライブラリのバグやしばしば発生する問題などのような、注意あるいは警告を表す。



翻訳者による補足説明を表す。



## サンプルコードの使用について

本書（原著）のコードは <https://github.com/catherinenelson1/SEforDS> から入手できる。

日本語版のコードは <https://github.com/urupat/se4ds/> から入手できる。

技術的な質問は [support@oreilly.com](mailto:support@oreilly.com) まで（英語で）ご連絡されたい。

本書の目的は、読者の仕事を助けることにある。一般に、本書に掲載しているコードは読者のプログラムやドキュメントに使用してかまわない。コードの大部分を転載

する場合を除き、我々に許可を求める必要はない。例えば、本書のコードの一部を使用するプログラムを作成するために、許可を求める必要はない。なお、オライリー・ジャパンから出版されている書籍のサンプルコードを CD-ROM として販売したり配布したりする場合には、そのための許可が必要だ。本書や本書のサンプルコードを引用して質問などに答える場合、許可を求める必要はない。ただし、本書のサンプルコードのかなりの部分を製品マニュアルに転載するような場合には、そのための許可が必要だ。

出典を明記する必要はないが、そうしていただければ感謝する。『データサイエンスのためのソフトウェアエンジニアリング入門』(Catherine Nelson 著、漆戸パティ訳、オライリー・ジャパン) のように、タイトル、著訳者、出版社、ISBN などを記載されたい。

サンプルコードの使用について、公正な使用の範囲を超えると思われる場合、または上記で許可している範囲を超えると感じる場合は、[permissions@oreilly.com](mailto:permissions@oreilly.com) まで（英語で）連絡されたい。

## 意見と質問

本書（日本語翻訳版）の内容については、最大限の努力をもって検証、確認しているが、誤りや不正確な点、誤解や混乱を招くような表現、単純な誤植などに気がつくこともあるかもしれない。そうした場合、今後の版で改善できるよう知らせていただければ幸いだ。将来の改訂に関する提案なども歓迎する。連絡先は次のとおりである。

株式会社オライリー・ジャパン  
電子メール [japan@oreilly.co.jp](mailto:japan@oreilly.co.jp)

本書の Web ページには次のアドレスでアクセスできる。

<https://www.oreilly.co.jp/books/9784814401321>  
<https://www.oreilly.com/library/view/software-engineering-for/9781098136192/> (英語)  
<https://github.com/catherinenelson1/SEforDS> (原書のサンプルコード)  
<https://github.com/urupat/se4ds/> (日本語版のサンプルコード、正誤表等のあるサポートサイト)

オライリーに関するそのほかの情報については、次のオライリーの Web サイトを参照されたい。

<https://www.oreilly.co.jp/>  
<https://www.oreilly.com/> (英語)

## オライリー学習プラットフォーム

オライリーはフォーチュン 100 のうち 60 社以上から信頼されている。オライリー学習プラットフォームには、6 万冊以上の書籍と 3 万時間以上の動画が用意されている。さらに、業界エキスパートによるライブイベント、インタラクティブなシナリオとサンドボックスを使った実践的な学習、公式認定試験対策資料など、多様なコンテンツを提供している。

<https://www.oreilly.co.jp/online-learning/>

また以下のページでは、オライリー学習プラットフォームに関するよくある質問とその回答を紹介している。

<https://www.oreilly.co.jp/online-learning/learning-platform-faq.html>

## 謝辞

本書の執筆にご協力いただいた皆様に心より感謝申し上げる。皆様からのコメント、フィードバック、議論、そしてサポートは、本当に貴重なものであった。

O'Reilly Media のチームと仕事ができたことは、本当に光栄な時間であった。感謝の念に耐えない。素晴らしい編集者であり、ずっとサポートしてくださった Virginia Wilson 氏とは、本当に楽しい時間を過ごした。Nicole Butterfield 氏には、貴重な全体的指示と書籍企画書作成プロセスへのご協力をいただいた。Jeff Bleiel 氏には、いくつかの章を徹底的にレビューしていただき、Chris Faucher 氏には、制作プロセスをスムーズに進めていただいた。

テクニカルレビューの William Jamir Silva、Ganesh Harke、Jo Stichbury、Antony Milne、Jess Males、Swetha Kommuri の各氏には、心から感謝したい。フィードバックは非常に建設的で細部にも配慮した有益な提案をいただき、最終的に本書をより良いものにすることができた。Rob Masson 氏は、最終稿への素晴らしいフィードバックをいただいたほか、執筆プロセス全体を通して綿密な議論をしてくださった。

Carol Willing、Ricardo Martín Brualla、Chris Trudeau、Michelle Liu、Maryam Ehsani、Shivani Patel、John Sweet、Andy Ross の各氏、そして Abigail Mesrenyame Dogbe のメンバーには、貴重な技術的議論と洞察に満ちた会話をしてくださったことに対して深く感謝したい。さらには、Python と PyLadies コミュニティ全体の一員であることからも大きな恩恵を受けた。時間を割いて活動してくださるボランティアの皆さんにも「ありがとう」を。

最後に、素晴らしい友人や家族へ。Rob、Mum、Richard、Lina、Salomé、Ricardo、Chris、Kiana、そして Katie。ありがとう！



# 目次

賞賛の声 .....	v
まえがき .....	vii

<b>1章 良いコードとは何か？ .....</b>	<b>1</b>
1.1 優れたコードが重要な理由 .....	1
1.2 変化する要件への対応 .....	3
1.3 単純さ .....	4
1.3.1 同じことを繰り返さない (DRY) .....	5
1.3.2 冗長なコードを避ける .....	7
1.4 モジュール化 .....	8
1.5 可読性 .....	9
1.5.1 標準と慣習 .....	10
1.5.2 命名法 .....	11
1.5.3 クリーンアップ .....	11
1.5.4 ドキュメンテーション .....	12
1.6 パフォーマンス .....	12
1.7 堅牢性 .....	13
1.7.1 エラーとロギング .....	13
1.7.2 テスト .....	13
1.8 まとめ .....	15

<b>2章 パフォーマンスの分析 .....</b>	<b>17</b>
2.1 パフォーマンス向上の方法 .....	18
2.2 実行時間の計測 .....	20
2.3 プロファイリング .....	23
2.3.1 cProfile .....	23
2.3.2 line_profiler .....	26
2.3.3 Memray によるメモリプロファイリング .....	27
2.4 時間計算量 .....	30
2.4.1 時間計算量の見積もり .....	30
2.4.2 オーダー記法 .....	32
2.5 まとめ .....	34
<b>3章 データ構造 .....</b>	<b>35</b>
3.1 Python のネイティブなデータ構造 .....	36
3.1.1 リスト .....	36
3.1.2 タプル .....	38
3.1.3 辞書 .....	39
3.1.4 セット（集合） .....	41
3.2 NumPy .....	42
3.2.1 NumPy の関数 .....	43
3.2.2 NumPy のパフォーマンスに関する考察 .....	44
3.2.3 Dask を使った配列演算 .....	47
3.3 機械学習における配列 .....	50
3.4 pandas .....	51
3.4.1 DataFrame の機能 .....	52
3.4.2 DataFrame のパフォーマンス .....	53
3.5 まとめ .....	55
<b>4章 オブジェクト指向プログラミングと関数型プログラミング .....</b>	<b>57</b>
4.1 オブジェクト指向プログラミング .....	58
4.1.1 クラス、メソッド、属性 .....	58
4.1.2 クラスの定義 .....	61
4.1.3 OOP の主要概念 .....	65

4.2	関数型プログラミング .....	69
4.2.1	ラムダ関数と <code>map()</code> .....	70
4.2.2	関数を DataFrame に適用 .....	71
4.3	どのパラダイムを使うべきか? .....	72
4.4	まとめ .....	72
<b>5章</b>	<b>エラー、ログ、デバッグ .....</b>	<b>75</b>
5.1	Python のエラー .....	75
5.1.1	Python のエラーメッセージ .....	75
5.1.2	エラー処理 .....	77
5.1.3	エラーの生成 .....	80
5.2	ロギング .....	82
5.2.1	何を記録するか .....	83
5.2.2	ロギングの設定 .....	83
5.2.3	ログの取り方 .....	85
5.3	デバッグ .....	87
5.3.1	デバッグの戦略 .....	87
5.3.2	デバッグ用ツール .....	88
5.4	まとめ .....	94
<b>6章</b>	<b>コードのフォーマット、linting、型チェック .....</b>	<b>95</b>
6.1	コードの形式とスタイルガイド .....	96
6.1.1	PEP 8 .....	97
6.1.2	import のフォーマット .....	98
6.1.3	Black を使ったフォーマッティング .....	100
6.2	linting .....	102
6.2.1	linting用ツール .....	103
6.2.2	IDE 内でのlinting .....	105
6.3	型チェック .....	106
6.3.1	型注釈 .....	107
6.3.2	mypy による型チェック .....	109
6.4	まとめ .....	110

---

<b>7章 テスト</b>	<b>111</b>
7.1 テストを書くべき理由	112
7.2 テストの時期	113
7.3 テストの書き方と実行方法	114
7.3.1 基本的なテスト	114
7.3.2 予期しない入力のテスト	116
7.3.3 pytest を使った自動テストの実行	118
7.4 その他のテスト用フレームワーク	120
7.5 テストの種類	120
7.5.1 単体テスト	121
7.5.2 統合テスト	122
7.6 データ検証	123
7.6.1 データ検証の例	123
7.6.2 Pandera を使ったデータ検証	124
7.6.3 Pydantic によるデータ検証	126
7.7 機械学習用のテスト	128
7.7.1 モデルトレーニングのテスト	129
7.7.2 モデル推論のテスト	130
7.8 まとめ	130
<b>8章 デザインとリファクタリング</b>	<b>133</b>
8.1 プロジェクトのデザインと構造	134
8.1.1 プロジェクトデザインの考慮事項	134
8.1.2 機械学習プロジェクトの例	136
8.2 コードのデザイン	138
8.2.1 モジュラーコード	139
8.2.2 コードデザインの枠組み	140
8.2.3 インタフェースと契約	141
8.2.4 結合度	141
8.3 ノートブックからスケーラブルなスクリプトへ	144
8.3.1 なぜノートブックの代わりにスクリプトを使うのか	144
8.3.2 ノートブックからのスクリプト作成	145
8.4 リファクタリング	148

8.4.1 リファクタリングの戦略 .....	149
8.4.2 リファクタリングの例 .....	149
8.5 まとめ .....	152
<b>9章 ドキュメンテーション .....</b>	<b>153</b>
9.1 コードベース内のドキュメンテーション .....	154
9.1.1 ネーミング .....	155
9.1.2 コメント .....	157
9.1.3 docstring .....	158
9.1.4 README、チュートリアル、その他の長いドキュメント .....	161
9.2 Jupyter ノートブックにおけるドキュメンテーション .....	162
9.3 機械学習の実験の文書化 .....	164
9.4 まとめ .....	165
<b>10章 コードの共有 .....</b>	<b>167</b>
10.1 Git を使ったバージョン管理 .....	167
10.1.1 Git の動作 .....	169
10.1.2 変更とコミットの追跡 .....	170
10.1.3 リモートとローカル .....	172
10.1.4 ブランチとプルリクエスト .....	174
10.2 依存ライブラリと仮想環境 .....	178
10.2.1 仮想環境 .....	179
10.2.2 pip による依存関係の管理 .....	181
10.2.3 Poetry による依存関係の管理 .....	182
10.3 パッケージング .....	185
10.3.1 パッケージングの基本 .....	186
10.3.2 pyproject.toml .....	188
10.3.3 パッケージのビルドとアップロード .....	189
10.4 まとめ .....	191
<b>11章 API .....</b>	<b>193</b>
11.1 API の呼び出し .....	194
11.1.1 HTTP メソッドとステータスコード .....	194

---

11.1.2 SDGs API からのデータの取得 .....	196
11.2 独自 API の作成 .....	199
11.2.1 独自 API のセットアップ .....	199
11.2.2 API への機能の追加 .....	203
11.2.3 GET エンドポイントの追加 .....	205
11.2.4 POST エンドポイントの追加 .....	205
11.2.5 独自 API の呼び出し .....	207
11.3 まとめ .....	209
<b>12 章 自動化とデプロイ .....</b>	<b>211</b>
12.1 コードのデプロイ .....	212
12.2 自動化の例 .....	214
12.2.1 コミット前フック .....	214
12.2.2 GitHub Actions .....	218
12.3 クラウドへのデプロイ .....	222
12.3.1 コンテナと Docker .....	223
12.3.2 Docker コンテナの構築 .....	224
12.3.3 Google Cloud への API のデプロイ .....	226
12.3.4 他のクラウドプロバイダーを使った API のデプロイ .....	228
12.4 まとめ .....	229
<b>13 章 セキュリティ .....</b>	<b>231</b>
13.1 セキュリティとは .....	231
13.2 セキュリティリスク .....	233
13.2.1 クレデンシャル、物理的セキュリティ、ソーシャルエンジニアリング .....	234
13.2.2 サードパーティのパッケージ .....	234
13.2.3 Python の <code>pickle</code> モジュール .....	235
13.2.4 バージョン管理システムのリスク .....	235
13.2.5 API のセキュリティリスク .....	236
13.3 セキュリティ対策 .....	237
13.3.1 セキュリティに関するレビューとポリシー .....	237
13.3.2 安全なコーディングのためのツール .....	237

13.3.3 シンプルなコードスキャン .....	238
13.4 機械学習のセキュリティ .....	241
13.4.1 機械学習システムへの攻撃 .....	241
13.4.2 機械学習システムのセキュリティ対策 .....	244
13.5 まとめ .....	244
<b>14章 ソフトウェア開発の現場 .....</b>	<b>247</b>
14.1 開発の原則やプラクティス .....	247
14.1.1 ソフトウェア開発のライフサイクル .....	247
14.1.2 ウォーターフォール型ソフトウェア開発 .....	249
14.1.3 アジャイルソフトウェア開発 .....	250
14.1.4 アジャイル・データサイエンス .....	251
14.2 ソフトウェア業界における役割 .....	252
14.2.1 ソフトウェアエンジニア .....	252
14.2.2 テストエンジニア .....	254
14.2.3 データエンジニア .....	254
14.2.4 データアナリスト .....	255
14.2.5 プロダクトマネージャー .....	256
14.2.6 UX リサーチャー .....	257
14.2.7 デザイナー .....	258
14.3 コミュニティ .....	258
14.3.1 オープンソース .....	259
14.3.2 イベントでの講演 .....	261
14.3.3 Python コミュニティ .....	262
14.4 まとめ .....	263
<b>15章さらなる飛躍を求めて .....</b>	<b>265</b>
15.1 コードの未来 .....	267
15.2 コードで描く未来 .....	270
15.3 おわりに .....	271
<b>索引 .....</b>	<b>273</b>

## コラム目次

ソフトウェアエンジニアリング vs. データサイエンス .....	ix
「作品」としてのコード .....	2
本書のデータ .....	14
早すぎる最適化 .....	18
タブにするかスペースにするかの大論争 .....	96
テスト駆動開発 .....	113
コードレビュー .....	177
バージョニング .....	178
その他の API フレームワーク .....	199
データの「プライバシー」 .....	232

# 1章 良いコードとは何か？

本書の目的は「より良いコード」を書く手助けをすることである。何がコードを「良い」ものにするのか？これについての考え方はいくつかある。最良のコードとは、最も速く実行されるコードかもしれない。あるいは、最も読みやすいコードかもしれない。別の定義としては、良いコードはメンテナンスが容易である、というものが考えられる。つまり、プロジェクトに何らかの変更が行われたとき、コードに戻って新しい要件を反映するように変更するのが簡単でなければならない。解決しようとしているビジネス上の問題の変化、新しい研究の方向性、コードベースの他の部分の更新などによって、コードの要件は頻繁に変わらう。

さらに、コードは複雑であるべきではなく、予期せぬ入力を受けても壊れるものであってはならない。コードに単純な新機能を追加するのは簡単であるべきだ。これが難しいなら、コードがうまく書かれていないことを示唆している。本章では、良いコードの持つ特性を紹介し、それぞれの例を示す。

## 1.1 優れたコードが重要な理由

優れたコードは、データサイエンス（DS）のコードがより大きなシステムと統合されるときに特に重要になる。例えば、機械学習（ML）モデルの本番稼働、より広く配布するためのパッケージの作成、他のデータサイエンティストのためのツールの構築などといったシミュレーションだ。最も意味があるのは、繰り返し実行されるような大規模なコードベースに関わる場合だ。プロジェクトが大きく複雑になればなるほど、優れたコードの価値は高まる。

ときには、明日のデモのために、一度きりのコードを「ハック」する必要もあるだろう。そのコードを本当に一度しか実行しないのであれば、美しいコードにするため

に時間を費やす必要はない。しかし筆者の経験では、1回きりのデモのために書いたコードでさえ、ほとんどの場合、また実行されるか、別の目的で再利用される。「緊急事態」が過ぎ去った後、自分のコードに戻り、将来の利用のために時間をかけて整理することを勧める。

## 「作品」としてのコード

多くのソフトウェアエンジニアは、コード自体を優れたものにする作業には価値があると考えている。効率的でエレガントなコードは本質的な価値を持つ。大工が美しい木製本棚に誇りを持つのと同じだ。長持ちするものを作ることで、仕事の満足感を得るのだ。

よい「作品」を作るためには、キーボードの前に座るたびに非常に多くの「小さな決断」を下すことになる。何に注目すればよいかがわかれれば、より良いコードが書けるようになる。ソフトウェアを書く技術を磨いて、自分が誇れるものを作るのは心地よいものだ。

良いコードは保守も簡単だ。メンテナンス「ビット腐敗 (bit-rot)」と呼ばれる現象がある。しばらく使われていないコードは「腐敗」し、更新が必要になる。これは、コードが依存しているものが（例えば、サードパーティーのライブラリや、ときには使っているOS さえ）変化するために起こる。しばらく使っていなかったコードを利用することになれば、おそらく更新作業が必要になるだろう。コードがきちんと構造化され、きちんと文書化されていれば、この作業はずっと簡単になる。



**技術的負債** (tech debt) とは、コードが「きちんと」書かれずに「やっつけ仕事で」書かれた結果、後回しにされた作業のことを指す。技術的負債には、ドキュメントの欠落、コードの不十分な構造化、変数名の付け方の不備、その他あらゆる「手抜き」がある。コードのメンテナンスやリファクタリングを困難にし、バグ修正に多くの時間を費やす要因となる。とはいっても、納期や予算の関係上、技術的負債が避けられないことが多い。コードを「磨く」時間がいつもあるわけではない。

## 1.2 変化する要件への対応

橋を作る際には、デザインを徹底的に練り上げ、建設計画を策定し、建設に取りかかるのが普通だろう。だがコードを書くときは、このようにはいかない。DS プロジェクトであれ、他分野のプロジェクトであれ、プロジェクトに取り組む中で変化に対応していかなければならず、あらかじめそのことを予期しておかなければならない。こうした変更の要因としては、例えば、研究過程での発見、ビジネス要件の変化、プロジェクトに取り入れたい技術革新といったものがある。優れたコードは、このような変化にも容易に適応できるものであるはずだ。

この**適応性**は、コードベースが大きくなればなるほど重要になる。1 つの小さなスクリプトであれば、変更を加えるのは簡単だ。しかし、プロジェクトが大きくなり、互いに依存し合う複数のスクリプトやノートブックに破壊的な変更を加えるようになると、より複雑になり、変更が難しくなる。最初から良い習慣を身につければ、大規模なプロジェクトでもコードは変更しやすいものになる。

DS はまだ比較的新しい分野だが、DS チームは、複数年にわたって同じコードベースで作業し、そのコードが多くの人によって作業され、そのうちの何人かは組織を辞めているような状況に遭遇し始めている。プロジェクトがある人から別の人へ引き継がれるこのような状況では、コードの品質がより重要になる。ドキュメントがきちんと作成され、それが読みやすいものになっていれば、他人がやった仕事を改良するのずっと容易になる。

**ソフトウェアエンジニアリング（ソフトウェア工学）**という学問分野は、何十年もの間、変化する要件や複雑性の増大に対処してきた。データサイエンティストである読者が活用できる、多くの有用な戦略を開発してきたのだ。「ソフトウェアエンジニアリング」について調べ始めると、Robert C. Martin が同名の本で推奨した「clean code」や「SOLID」などといった「原則」を表す用語に出会うことになるだろう (<https://oreil.ly/16t90>)。

本章では、こうした原則を、優れたコードの特徴を表す次の 5 つの言葉で表現する。

- 単純さ
- モジュール化
- 可読性
- パフォーマンス
- 堅牢性

本章の残りの部分で、それぞれについて詳しく説明する。

## 1.3 単純さ

複雑なものよりも単純なもののはうがいい。<sup>シンプル</sup>

—— Tim Peters : “The Zen of Python”

小さなプロジェクト、例えばデータを可視化するだけのノートブックや短いデータを操作するだけのスクリプトに取り組んでいるのであれば、一度にすべての詳細を頭に入れておくことができる。しかし、プロジェクトが大きくなり、より複雑になってくると、これは不可能になる。ML モデルのトレーニングステップは記憶できても、入力データのパイプラインやモデルのデプロイプロセスまでは難しくなる。

**複雑さ** (complexity) は、要件が変わったときにコードの修正を困難にするが、これを次のように定義できる。

複雑さとは、システムの理解や修正を困難にする、システムの構造に関連するあらゆるものを指す。

—— John Ousterhout 著『A Philosophy of Software Design』

これは正確な定義ではないが、経験を積めば、システムがより複雑になる時期の感覚がつかめるだろう。1つの特徴として、「変更によって、予想もしなかったものが予想もしなかったように壊れる」というものがある。例えば、自然言語処理 (NLP) テクニックを使って顧客が購入した商品を抽出するために、顧客レビューデータで ML モデルをトレーニングするとする。レビューを 512 文字に切り詰める前処理ステップが別にある。しかしモデルをデプロイするとき、推論コードに前処理ステップを追加するのを忘れてしまう。突然、入力データが 512 文字より大きくなってしまったため、モデルはエラーを出すようになる。このシステムはどう動作するかの予想が困難で、複雑さを増してきていることになる。

ここで議論しているコードの「複雑さ」は、一般に「偶発的な」複雑さを指す。これはプロジェクトの「本質的な」複雑さとは異なるものだ。ML プロジェクトが本質的に複雑になる代表的な要因としては、次のようなものが挙げられる。

- さまざまなタイプのモデルやさまざまな特徴の組み合わせを試して、どれが一番効果的かを確かめる必要がある

- 使用するデータに相互に依存するパラメーターが多いため分析が複雑になる

どちらも単純化することはできない。複雑さはプロジェクトの一部にすぎない。これに対して偶発的な複雑さは、「ある目標を達成するために、コード内のどの関数を変更する必要があるのかわからない」といったような状態を指す。

しかし、コードの複雑さを減らすのに役立つツールがある。すべてを少しずつシンプルにしていくことは、プロジェクトが大きくなったときに大きな利点となる。次節では、繰り返しを避けてコードの単純さ（簡潔さ）を保つ方法を説明する。また、「1.4 モジュール化」で説明するように、コードを再利用可能な断片に分割することで、コードが複雑になるのを避けられる。

### 1.3.1 同じことを繰り返さない (DRY)

良いコードを書く上で最も重要な原則の1つは、「情報を繰り返さない (Don't Repeat Yourself: DRY)」だ。すべての知識は、コードの中で1つの表現になるべきだ。例えば、情報が複数の場所で繰り返され、その情報が要求の変化によって更新される必要がある場合は、「1つの変更」が「多くの変更」につながってしまう。情報を更新する必要があるすべての場所を記憶することになってしまうのだ。これは煩雑で、コードの複雑さを増す。さらに、重複はバグを生む機会を増やし、またコードが長くなることで読み解くのに時間がかかるようになる。

よく似ているが同じではないコードを見るのは、精神的な負担が大きい。その2つのコードが同じことをやっているのか、それとも違うことをやっているのかを判断するのが困難になる。

単純な例を見てみよう。3つのCSVファイルを開き、pandasのDataFrameに読み込み、何らかの処理を行い、各DataFrameを返す。この例のデータは国連の「持続可能な開発目標 (SDGs)」(<https://oreil.ly/2MeuI>) のもので、本書では、以降このサイトのデータを使うことにする。このデータに関する詳細は、「1.7.2 テスト」のコラム「本書のデータ」にある。

まず、下に示すようなコードを考えてみよう (`example/ch01/ch01.ipynb`)。



訳者補記

本書で使うコードやデータ (CSVファイルなど) は、本書日本語版のGitHubリポジトリ (<https://github.com/urupat/se4ds/>) にある。

また、本書の例題の多くはノートブック (jupyter notebook) の形式になっている。例えば、`example/ch01/ch01.ipynb` を実行するには、次のコマンドの

のように example フォルダに移動し、jupyter notebook を実行する。

```
$ cd example
$ jupyter notebook
```

ブラウザにフォルダの一覧が表示されるので、ch01 → ch01.ipynb の順に選択すれば、1章の例題を見ることができる。

```
import pandas as pd

df = pd.read_csv("sdg_literacy_rate.csv")
df = df.drop(["Series Name", "Series Code", "Country Code"], axis=1)
df = df.set_index("Country Name").transpose()

df2 = pd.read_csv("sdg_electricity_data.csv")
df2 = df2.drop(["Series Name", "Series Code", "Country Code"], axis=1)
df2 = df2.set_index("Country Name").transpose()

df3 = pd.read_csv("sdg_urban_population.csv")
df3 = df3.drop(["Series Name", "Series Code", "Country Code"], axis=1)
df3 = df3.set_index("Country Name").transpose()
```

このようなコードでは必要に長く、繰り返しが多い。同じ結果を得るより良い方法は、繰り返しのコードを for ループの中に入れることだ。あるいは、次のように関数を作ってもよい。

```
def process_sdg_data(csv_file, columns_to_drop):
    df = pd.read_csv(csv_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("Country Name").transpose()
    return df
```

このような場合以外にも、より微妙なケースでコードの重複が生じことがある。次にいくつかの例を挙げる。

- データ処理のコードなど、非常に似たものを複数のプロジェクトで使ってしまっている場合もある。厳密に1種類のデータしか受け付けないようにするのではなく、少し異なるデータも受け付けるようにすれば、こうした重複を避けられる

- 同じようなプロジェクトに取り組んでいる複数の人が、同じようなコードを書くかもしれない。他人が使いやすいコードを書き、しっかりしたドキュメントを作成しておけば、この種の重複を減らすのに役立つ
- コメントやドキュメントも重複の一形態となりうる。同じ情報が、コードとそれを説明する文書で表現されているようなケースだ。コードがやっていることをそのまま記述するようなコメントは書かず、情報を追加するために使うようにしよう。これについては9章で詳しく説明する

**DRYの原則**は、良いコードを書く際に考慮すべき非常に重要なものだ。些細なことのように見えるかもしれないが、繰り返しを避けるためには、コードをモジュール化して読みやすくする必要がある。こうした概念については、本章の後半で説明する。

### 1.3.2 冗長なコードを避ける

コードの行数を少なくすることで、コードをシンプルにできることがある。これは、バグが発生する機会を減らし、他人が読んで理解しなければならないコードの量を減らすことを意味する。しかし、「短いコード」と「読みやすいコード」の間には、「トレードオフ」があることが多い。コードを読みやすくする方法については、「1.5 可読性」で説明する。

「簡潔でありながら読みやすいコード」を目指すべきだ。そのためには、組み込み関数を使わず同じような機能を持つ自作の関数を書いたり、余分な一時変数を使ったりするなど、コードを不必要に長くすることは避ける必要がある。また、前節で説明したように、繰り返しも避けるべきだ。

不要な一時変数の例を挙げよう。

```
i = float(i)
image_vector.append(i/255.0)
```

このコードは次のように単純化できる。

```
image_vector.append(float(i)/255)
```

もちろん、コードの行数を少なくすることには欠点もある。1行の中で多くのことが起こっている場合、何が起こっているのかを理解するのが難しくなる。つまり、コードの改良が難しくなり、バグが増える危険性がある。そういうたったの疑念が拭えないのなら、たとえ数行余計に使うことになっても、コードを読みやすく保つほうがよい。

## 1.4 モジュール化

**モジュール化**とは、長いコードを小さなコンポーネントに分割することだ。モジュール化されたコード（**モジュラーコード**）には、いくつかの重要な利点がある。

- コードが読みやすくなる
- 問題が発生した場所を特定しやすくなる
- 次のプロジェクトでコードを再利用しやすくなる
- テストしやすくなる（詳しくは7章参照）

これから大きなシステムを作ることになったとしよう。長いスクリプトを1つ書いて全部をこなすこともできる。始めのうちはそれでもよいかもしれない。しかし、大きなプロジェクトは細かく分割する必要がある。そのためには、できるだけプロジェクトの将来を先取りして考え、システム全体が何をするのか、それを分割するのに適した場所はどこかを予測する必要がある。<sup>プランニング</sup>この計画のプロセスについては、8章で詳しく説明する。

モジュラーコードの構築は継続的なプロセスであり、どんなに頑張っても最初から完全に正しいコードを書けるわけではない。プロジェクトが発展するにつれて、コードも変化していくはずだ。「8.4 リファクタリング」では、コードの改善に役立つテクニックを取り上げる。

大規模なDSプロジェクトをフローチャートのように考えることで、一連のステップに分けることができる（図1-1）。まずデータを収集、それを探索し、クリーニングし、そして可視化なのだ。

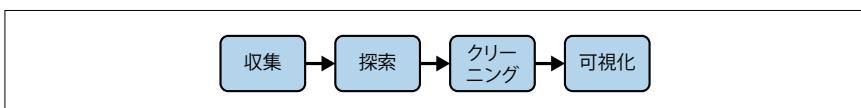


図1-1 大規模なデータサイエンス・プロジェクトを個別のステップに分解

最初は、一連のJupyterノートブックかもしれない。それぞれのシリーズが終了したら、データをファイルに保存し、それをまた次のノートブックにロードする。プロジェクトが成熟してくると、同じような分析を繰り返し実行することが多くなるかもしれない。そのとき、システムの骨格を決めることができる。例えば、データを抽

出する関数が 1 つあり、データをクリーニングする関数にそれを渡す。次の例では、`pass` 文を使って空の関数を作成している。これにより、この関数を呼び出しても、書き込まれる前にエラーが発生することはない。

例えば次のような手順を考えてみよう。

1. データをロード
2. 最大長にトリミングしてクリーンアップ
3. 何らかのプロットパラメーターでプロット

このシステムの「スケルトン（骨格）」は次のようなコードにすることができる。

```
def load_data(csv_file):
    pass

def clean_data(input_data, max_length):
    pass

def plot_data(clean_data, x_axis_limit, line_width):
    pass
```

このフレームワーク（枠組み）を作成することで、システムを個々のコンポーネントに分解し、それぞれのコンポーネントが入力として何を受け入れるべきかを知ることができる。Python ファイルのレベルでも同じことができる。オブジェクト指向プログラミングや関数型プログラミングのような「プログラミングパラダイム」を使えば、コードを関数やクラスに分解する方法を見つけ出すのに役立つ（これについては 4 章で詳しく説明する）。システムをどのように分割するにしても、各コンポーネントはできる限り独立し、自己完結していなければならない。

なお、モジュラーコードについては 8 章で詳しく説明する。

## 1.5 可読性

... コードは書かれるよりも、読まれることのほうがあるかに多い...

—— PEP 8

コードを書くとき、他人もそれを使えるようにすることが重要だ。別のプロジェクト、あるいは別の仕事に移るかもしれない。しばらくプロジェクトから離れ、1か月後、半年後、あるいは 6 年後に戻ってきたとき、書いた当時のことをまだ覚えている

だろうか？そのコードを書いたのには理由があり、何らかの重要なタスクのためにあるはずだ。コードを読みやすくすることで、「長持ちさせる」ことができるのだ。

コードを読みやすくする方法としては次のようなものが挙げられる。

- 選択したプログラミング言語の標準や慣例を守る
- 適切な名前を付ける
- 使われないコードを削除する
- コードのドキュメントを書く

こうした事柄を後回しにし、コードの機能に集中したくなるところだが、コードを書く時点で読みやすくなることに注意を払えば、複雑でなく保守しやすいコードを書けるようになる。本節でこうした事柄の概要を紹介し、「6章 コードのフォーマット、linting、型チェック」と「9章 ドキュメンテーション」でさらに詳しく説明する。

### 1.5.1 標準と慣習

コーディングのスタンダード（標準）と形式は、本書で取り上げるトピックの中では最もつまらない事柄のように思えるかもしれないが、実は重要なことだ。同じコードを表現する方法はたくさんある。「加算を行うための + の前後にスペースを書くかどうか」といった細かい事柄まで、検討の対象になる。「コーディング標準」は Python のコードを書くすべての人に一貫性を持たせるために開発されたもので、他人が書いたコードであっても見慣れたコードに感じられるようにすることを目的としている。これは、自分で書いていないコードを読んだり編集したりする労力を減らすのに役立つ。このトピックについては 6 章で詳しく説明する。

Python は多くのプログラミング言語と比べて、そもそも非常に読みやすい。コーディング標準に準拠することで、さらに読みやすくなる。Python の主なコーディング標準は PEP 8 (Python Enhancement Proposal 8。<https://oreil.ly/UwHsO>) で、2001 年に制定された。下の例は PEP 8 からの抜粋で、コードの細部にまで規約があることがわかる。Google の「Python スタイルガイド」(<https://oreil.ly/Q3eZL>)などのスタイルガイドは、PEP 8 を補完するガイダンスや情報を提供している。

次は PEP 8 による細かな指定の例で、さまざまな括弧とスペースに関して「正しい」書き方と「正しくない」書き方を示している。

```
# 正しい
spam(ham[1], {eggs: 2})
```

```
# 正しくない
spam( ham[ 1 ], { eggs: 2 } )
```

幸いなことに、コードがコーディング標準に準拠しているかどうかをチェックする自動化された方法がたくさんある。これにより、+の前後に空白が1つずつあるかどうかを調べてチェックするという退屈な作業から解放される。Flake8 や Pylint などの「リンター」は、コードがPEP 8に準拠していない場所をハイライトしてくれる。Blackなどの自動フォーマッターは、コーディング標準に準拠するようにコードを自動的に更新してくれる（6章でこうしたツールの使い方を説明する）。

## 1.5.2 命名法

DSのコードを書く際には、関数名、変数名、プロジェクト名、さらにはツール名など、多くの場面で名前を付ける必要がある。名前の選択は、作業のしやすさに影響する。説明的でない名前や不明確な名前を選ぶと、その「本当の意味」を頭の中に入れておく必要があり、認知的な負荷が高まる。例えば次のコードのように、pandasライブラリをpとしてインポートし、変数名としてxとfを使うこともできる。

```
import pandas as p

x = p.read_csv(f, index_col=0)
```

このコードはエラーもなく正しく実行される。しかし、次のコードのように、変数名をより情報量の多いものにし、標準的な規約に従えば、より読みやすいコードになる。

```
import pandas as pd

df = pd.read_csv(input_file, index_col=0)
```

命名法については9章で詳しく説明する。

## 1.5.3 クリーンアップ

コードを読みやすくするもう1つの方法は、関数を作成した後でコードを整理することだ。テストが終わり、動作していると確信が持てたら、コメントアウトされたコードを削除し、簡単なデバッグのために挿入したprint()関数への不要な呼び

出しを削除する。他人のコードにコメントアウトされた部分があると、混乱するものだ。

コードに整理整頓されていない部分があると、このプロジェクトではコードの質が悪くともかまわないというメッセージを送ることになる。これは、他のメンバーが良いコードを書こうという意欲を削ぐことになる。整頓されていないコードが、プロジェクトの他の部分でもコピーされ、使われてしまうかもしれない——これは「Broken Window Theory」(<https://oreil.ly/3sa7c>)と呼ばれている。プロジェクトに高い標準を設定することで、そのプロジェクトへの参加者全員が良いコードを書くようになる。

コードの品質を向上させる方法としては、コードのリファクタリング（全体の振る舞いを変えずにコードを変更すること）がある。コードの高速化、別のプロジェクトで使っているコードの再利用、コードの構造の改良といった目的でリファクタリングが行われる。このプロセスではテストが不可欠だ。新しいコードでも全体的な振る舞いが同じであることをチェックしてくれる。リファクタリングについては「8.4 リファクタリング」で説明する。

## 1.5.4 ドキュメンテーション

コードに関する説明（ドキュメンテーション）は、他人があなたのコードを読む助けになる。単純な「インラインコメント」、関数全体を説明する `docstrings`、GitHub リポジトリに表示される `README`、さらにはパッケージの使い方を利用者に教えるチュートリアルまで、すべてのドキュメントは、他人にコードの使い方を説明するのに役立つ。自分のコードを「将来の自分」に説明することになるかもしれない。他人に自分のコードを使ってもらいたいのなら、良いドキュメントを書いて、簡単に利用できるようにすることだ。

優れたドキュメントを書くだけでなく、それを保守し、更新し続ける必要もある。古いバージョンのコードについてのみ説明しているドキュメントは、まったくドキュメントがないよりも始末が悪い。混乱を引き起こし、その解消に余計な時間がかかることになる。9章で、あらゆる形式のドキュメントについて詳しく説明する。

## 1.6 パフォーマンス

「優れたコード」と呼ばれるようになるためにはパフォーマンスも重要だ。実行時間とメモリ使用量の両面においてである。コードの書き方を決めるとき、どのような

データ構造やアルゴリズムが効率的かを知っておくと役に立つ。著しく実行が遅くなるようなコードを改良する手段を知っておくことは有用だ。また、コードのどの部分の実行に時間がかかるのかを知ることも重要だ。

パフォーマンスが特に重要なのは、ユーザーが特定のアクションを取るたびに呼び出されるようなコードを書いている場合だ。ユーザーが増えたり、プロジェクトが成功すれば、自分が書いたコードが毎日何十万回、何百万回と呼び出されることになる。この場合、コードを少し改善するだけで、ユーザーはトータルで何時間も節約できることになる。大規模なアプリケーションの中で、自分の書いたコードが足を引っ張ることは避けたい。コードのパフォーマンスを測定する方法については2章で説明する。コードのパフォーマンスを最適化するために、最適なデータ構造を選択する方法については3章で紹介する。

## 1.7 堅牢性

良いコードは堅牢性にも優れていなければならない。つまり、失敗することなく、最初から最後までコードを実行できることだ。コードはまた、入力が予期せず変化した場合に、うまく対応できなければならない。予期せぬエラーを生成して、そのコードを含むシステム全体を機能不全に陥らせるのではなく、変化に対応できるようにデザインされていなければならない。エラー処理を適切に行い、何が起こったかをログに記録し、優れたテストを書くことで、コードをより堅牢にできる。

### 1.7.1 エラーとロギング

堅牢性のあるコードは、不正な入力があっても予期せぬ動作をしない。予期せぬ入力があったときにコードをクラッシュさせるのか、それともエラー処理をして何か対処するのかを選ぶべきだ。例えば、CSVファイルに期待したデータの半分の行がなかった場合、コードをエラーにするのか、それとも半分のデータだけを処理するのか。あるべき状態でないことを警告<sup>アラート</sup>で知らせるか、エラーを処理するか、黙って失敗させるか、明確な選択をすべきである。エラーについては5章で詳しく説明する。

エラーが処理されたとしても、エラーが発生したことを記録しておくことは重要である。これはロギングの1つの例だ。5章で、ロギングについて説明する。

### 1.7.2 テスト

テストは堅牢なコードを書くための鍵である。ソフトウェアエンジニアリングで

は、主にユーザー テストと自動 テストの 2 種類の テストを用いる。ユーザー テストでは、人がソフトウェアを（部分的に）使って、それが正常に動作するかを確認する。自動 テストは一般に、コードの一部に入力を送り、その出力が期待どおりであることを確認する。本書では、自動 テストのみを取り上げる。

テストが必要なのは、コードが自分の環境で完璧に動作したとしても、それが他人の環境で（あるいは将来の自分の環境で）動くとは限らないからだ。データは変化し、ライブラリは更新され、環境によって Python のバージョンは異なる。読者が書いたコードを誰かがその人の環境で使いたいと思ったら、その人がテストを実行して動作を確認すればよい。

テストにはいくつかの種類がある。単体 テスト（unit test）は单一の関数をテストし、エンドツーエンド テスト（end-to-end test）はプロジェクト全体をテストする。また、統合 テスト（integration test）は多くの関数を含むがプロジェクト全体よりは小さなコードの塊をテストする。テストに関する戦略とライブラリについては、7 章で詳しく説明する。テストが用意されていない大規模なコードベースがあるのならば、まずはテストを書くべきだ。そうすることで、問題が再発しないようにするのだ。

## 本書のデータ

本書では、国連の「持続可能な開発目標（SDGs）」のデータを使用する。SDGs は、2015 年に国連加盟国によって採択された「持続可能な開発のための 2030 アジェンダ」の一部である 17 の目標である。目標には、貧困の撲滅、飢餓の撲滅、教育の機会の提供、ジェンダーの平等などが含まれる。SDGs は補助的な目標に分けられ、200 以上の統計指標セットを用いて追跡される。指標は、こうした目標に向けた進捗状況を定量的に測定する。

例えば、目標 1 は「あらゆる場所のあらゆる形態の貧困を終わらせる」である。そして目標 1.1 は「2030 年までに、現在 1 日 1.25 ドル未満で生活する人々と定義されている極度の貧困をあらゆる場所で終わらせる」となっている。グローバル指標 1.1.1 は「国際的な貧困ラインを下回って生活している人口の割合（性別、年齢、雇用形態、地理的ロケーション（都市/地方）別）」である。

こうした指標のデータは、オンラインデータベース (<https://oreil.ly/X3rDE>) と API (<https://oreil.ly/rBEv->) で入手できる。本書のサンプルコードでは、こうした指標のデータを利用している。

## 1.8 まとめ

「良いコード」を書くことはさまざまな面で開発者を助けてくれる。

- 自分のコードを他人が使うのが簡単になる
- 最後にコードを触った半年後に自分の仕事に戻ったとき、何をしていたかが簡単に思い出せる
- コードをスケールアップし、より大きなシステムとのインターフェースをとる必要が生じても容易に行える
- 当初の計画にはなかった機能をコードに追加する必要が出た場合も、優れたコードならば作業が簡単になる

良いコードを書くための原則についてより詳しく知りたいのならば、次に挙げる本を推奨する。

- David Thomas + Andrew Hunt 著『達人プログラマー 第2版』(オーム社)
- John Ousterhout 著『A Philosophy of Software Design』(Yaknyam Press)

最後に、良いコードを書くために重要な事柄を挙げておこう。以下の各章で具体的に説明していく。

### 単純さ

コードにおいて、不必要的繰り返し、不必要的複雑さ、不要な行は避けるべきである

### モジュール化

コードは論理的に関数に分解され、入力と出力が明確に定義されていなければならぬ

### 可読性

コードは PEP 8 標準の形式に従い、十分に検討された名前を付け、きちんと文書化されていなければならない

### パフォーマンス

コードの実行に不必要に長い時間がかかったり、利用可能な範囲を超えるような「リソース」を要求したりしてはならない

### 堅牢性

コードに再現性があり、有用なエラーメッセージを出力し、予期せぬ入力にも失敗せずに処理できるものでなければならない

次章では、優れたコードの1つの側面であるパフォーマンスについて詳しく見ていく。

## 2章 パフォーマンスの分析

「1章 良いコードとは何か？」では、「良いコード」が持つべき性質について説明したが、本章では、その中から「パフォーマンス」について深く掘り下げていこう。より大きなシステムの一部を構成するコードを書くのであれば、自分のコードのパフォーマンスは非常に重要である。タイムリーに出力を返さなければならないし、利用可能なコンピューティングリソースを超えてはならない。

しかしその前に、もっと重要なことがある。それはコードがきちんと動作することだ。本章や次章のテクニックを適用する前に、コードが、「解決すべき問題をきちんと解決し、期待どおりの出力を返すこと」を確認しよう。最適化や高速化は、コードが正しく実行されるようになった後で行うべき事柄だ。

そして次に、パフォーマンス向上の必要性の確認も重要だ。システム全体から見て、自分のコードが期待されるパフォーマンスを知る必要がある。特定の時間内に結果を返す必要があるのか？ ユーザーはコードの結果に基づいて何らかのアクションを起こすのを待っているのか？ もしそうならば、速ければ速いほど「ユーザー体験」はより好ましいものになる。

例えば、作成中の機械学習（ML）モデルが、100ms（ミリ秒）以内に予測を返す必要があるとしよう。現状のコードで許容される時間内に結果を返しているのであれば、パフォーマンスの最適化に時間を費やす必要はないだろう。

しかし、どのような場合でも、本章で説明する原則を意識しておくに越したことはない。非効率なコードを書いているとき（例えば、急いで何かを「ハック」する必要があるときなど）に知っておくと、後で戻ってそういったコードを最適化できる。なお、効率的なデータ構造を選択する具体的な方法は、「3章 データ構造」で説明する。

データサイエンティストであれば、「何を測定するかが重要である」という考え方をお馴染みだろう。コードのパフォーマンスを改善するための最初のステップは、最

も遅いのはコードのどの部分か、そして、最もメモリを消費しているのはどの部分かを特定することだ。そうした部分を特定するためのデータを集めれば、「遅いコード」や「メモリを多く消費するコード」が、修正可能かどうかがわかる。コードに非効率な部分がまだ残っていて遅いのかもしれないし、すでに可能な限り効率的になっているかもしれない。このどちらなのかを判断するためのデータが必要なのだ。このために、「2.2 実行時間の計測」と「2.3 プロファイリング」でコードのパフォーマンスを測定する方法を詳しく説明する。

### 早すぎる最適化

早すぎる最適化は、プログラミングにおける諸悪の根源である（少なくともその主要因ではある）。

—— Donald Knuth 著『The Art of Computer Programming』

ドナルド・クヌース

Donald Knuthのこの言葉は、ソフトウェアエンジニアリングの世界で繰り返し聞かされる古典的なものだ。データサイエンスの世界についてみれば、この他にも「悪」があるかもしれないが、このアドバイスが有用であることは間違いない。筆者は Knuth のこの言葉を「コードに要求される速度やメモリ量を知る前に、また、どこにボトルネックがあるのかを知る前に、コードを最適化することは避けるべきである」という趣旨の警告だと解釈している。

## 2.1 パフォーマンス向上の方法

ここでは、優れたパフォーマンスとは、「コードが使用するコンピューティングリソースの量を最小限に抑えることだ」と定義する。コンピューターアーキテクチャの詳細については、それだけで膨大なトピックになるのでここでは触れない。しかし、あるタスクを完了するのにかかる時間や、必要なメモリ量に対して、コードがどのように影響するのかの測定は可能だ。つまり、「パフォーマンスの良いコード」とは、より高速なコード、使用するメモリ量がより少ないコード（「メモリフットプリント」が小さなコード）を意味する（ただし、多くの場合、より高速なコードがより多くのメモリを占有するという「トレードオフ」があることも事実ではある）。

コードのパフォーマンスを向上させる方法は、使用しているプログラミング言語に

よって異なる。Python の場合、主な選択肢は次のものだ。

### アルゴリズムの選択

書くコードの選択によって、そのパフォーマンスは大きく変わる。例えば、入れ子になったリスト処理は可能な限り避けるべきだ（リスト処理のループは1回だけにし、ループの入れ子は避ける）

### データ構造の選択

達成すべきタスクによっては、データ構造の選択が重要になる。例えば、リストを使った検索よりも、辞書を使った検索のほうがはるかに速い。これについては「3章 データ構造」で詳しく説明する

### 組み込み関数の利用

問題を解決する組み込みの関数が存在する場合、ほとんどの場合、自分で書くよりもその関数を使ったほうが効率的だ。組み込み関数の多くは C 言語で実装されており、一般的に Python で同じ関数を実装するよりも速い

### Python のコンパイル

Cython (<https://cython.org>)、Numba (<https://numba.pydata.org>)、PyPy (<https://www.pypy.org>) などのツールを使って Python を低レベル言語にコンパイルすれば、コードをより速く実行できる可能性がある。どれを使うかは場合による。Numba は Python のサブセットを含み、Cython は Python のスーパーセットに C 言語のオプションを追加したもので、PyPy は JIT (just-in-time) コンパイルを行う

### 非同期のコード

あるタスクの結果を待つ間に別のタスクを実行できればコードを高速化できる。例えば、遅いネットワークからのレスポンスを待っている間に別の作業ができるればそれだけ効率が上がる

### 並列・分散コンピューティング

**並列コンピューティング**では、コンピュータの複数のプロセッサーでコードを実行する。`multiprocessing` モジュールを使えば、一度に複数の CPU でコードを実行できる。**分散コンピューティング**では、同時に複数のコンピュータを使ってコードを実行する

本書では、新しいツールやテクニックを学ぶ時間を必要とせず、開発者が日常的に書いているコードを改善するためにできることに焦点を当てる。そのため、上のリストにあるすべての選択肢を取り上げるつもりはない。本章では、パフォーマンスの測定法とアルゴリズムの選択に焦点を当てる。コードのパフォーマンスの測定方法を知ることで、組み込み関数がどのような場合に改善をもたらすかがわかる。「3章 データ構造」では、ユースケースごとの最適なデータ構造の選び方も取り上げる。また「3.2.3 Dask を使った配列演算」で並列・分散コンピューティングについて簡単に触れる。

パフォーマンスについて深く掘り下げたいのであれば、次のステップとして、Michael Gorelick + Ian Osvald 著『ハイパフォーマンス Python 第2版』（オライリー・ジャパン）がよいだろう。この本には、Python 用のさまざまなコンパイラの詳細も含まれている。また、非同期コードに関する入門として、FastAPI のドキュメント (<https://oreil.ly/JtZED>) もお勧めだ。

## 2.2 実行時間の計測

コードのどの部分が遅いかを知る最も簡単な方法は、「関数」や「コードの特定部分」の実行にかかる時間を測定し、変更を加えてからもう一度測定することだ。そうしないと、スピードのアップやダウンの原因が何かはわからない。



本章では、いくつかの簡単な統計関数の例を使うが、説明のために使っているだけで、ここで使われているコードを実践で利用したり、こうした関数を自分で実装したりすることはお勧めしない。`statistics` パッケージや NumPy に入っている関数を使うべきだ。

次の例は、数値リストの最頻値 (mode) を計算する簡単なコードである。これを例にして、コードの実行時間を測定する方法を見てみよう。

```
def slow_way_to_calculate_mode(list_of_numbers): # 最頻値を計算する遅い方法
    result_dict = {}
    for i in list_of_numbers:
        if i not in result_dict:
            result_dict[i] = 1
        else:
```

```

result_dict[i] += 1

mode_vals = []
max_frequency = max(result_dict.values())
for key, value in result_dict.items():
    if value == max_frequency:
        mode_vals.append(key)

return mode_vals

```

次に、この関数に入力する 100,000 個のランダムな整数のリストを作成しよう。

```

import numpy as np # 実行前に「pip install numpy」が必要

random_integers = np.random.randint(1, 1_000_000, 1_000_000)

```

この関数の実行時間を測定する最も簡単な方法は、`time` モジュールを使うことだ。コンピュータの時計で、関数を実行する前と後の時刻を記録し、その差を出力する。

```

import time

start = time.time()
slow_way_to_calculate_mode(random_integers)
end = time.time()

print(end-start)

```

しかし、関数の実行時間は変動する可能性があるので、複数回測定するのがよい。これには `timeit` モジュールが使える。Jupyter ノートブックで作業しているのなら、セルマジックコマンド `%%timeit` を使うのが便利だ。

```

%%timeit
slow_way_to_calculate_mode(random_integers)

```

267 ms ± 23.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

ここでは、`timeit` はコードを 10 回ループし、10 回のループのセットを 7 回繰り返した後、実行の要約を出力している。`timeit` のデフォルトは、2 秒以内で実行できる実行回数とループ回数だが、`-r` フラグを使って実行回数を指定でき、`-n` フラグ

を使ってループ回数を指定できる。

スタンダードアローンのスクリプトを使う場合は、`timeit` を次のように使えばよい(`example/ch02/time_mode.py`)。

```
import numpy as np    # 「pip install numpy」が必要
import timeit

random_integers = np.random.randint(1, 100_000, 1000)

def slow_way_to_calculate_mode(list_of_numbers):
    result_dict = {}
    for i in list_of_numbers:
        if i not in result_dict:
            result_dict[i] = 1
        else:
            result_dict[i] += 1

    mode_vals = []
    max_frequency = max(result_dict.values())
    for key, value in result_dict.items():
        if value == max_frequency:
            mode_vals.append(key)

    return mode_vals

mode_timer = timeit.Timer(
    stmt="slow_way_to_calculate_mode(random_integers)",
    setup="from __main__ import slow_way_to_calculate_mode, random_integers",
)
time_taken = mode_timer.timeit(number=10)

print(f"Execution time: {time_taken} seconds")
```

しかし、このコードに本当の意味はあるのだろうか？ セルマジックコマンドの`%timeit` を使って出力された 267ms は速いのか遅いのか？ これは（どの程度のスピードが必要かの基準がない）単なる例なので、最頻値を計算する別の方法と比較し、それによってパフォーマンスが向上するかどうかを見てみよう。同じ計算を行う別のある方法を紹介しよう。

```
from collections import Counter
```

```
def mode_using_counter(list_of_numbers):
    c = Counter(list_of_numbers)
    return c.most_common(1)[0][0]
```

そしてまた、`timeit` を使って実行時間を計測する。

```
%%timeit
mode_using_counter(random_integers)
```

23.2 ms ± 737 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

ご覧のとおり、このバージョンのコードは実行にかかる時間が大幅に短縮されている。先ほどは 267ms だったが、今度は 23.2ms しかかかっていない。標準偏差（std. dev.）もかなり小さくなっている。実行時間の上限が抑えられることがわかる。

この例から 2 つの点を学ぶことができる。第 1 に、異なるコードでもまったく同じ機能を実行できるが、あるバージョンは別のバージョンより、はるかに高速になることがある。第 2 に、組み込み関数（この場合、`collections` モジュールの `Counter`）を使うほうが、自分で書くよりもずっと速いことが多い。`collections` および `itertools` の両モジュールには、一般的な関数を高速化するのに役立つ組み込み関数がたくさん用意されている。

## 2.3 プロファイリング

1 つのコードのまとめについてなら、`%%timeit` を使って簡単に時間が計測できる。ただ、より長い関数やスクリプト全体を対象にしたい場合、それぞれの行をノートブックのセルに分割し、別々に計測するのは面倒だ。そこでプロファイラの出番だ。プロファイラは、関数のどの部分で最も時間がかかっているかを教えてくれるほか、より詳しい情報も提供してくれるので、コードの「ボトルネック」を発見しやすくなる。なお、ここでついでに、メモリ使用量をプロファイリングする方法についても説明しておこう。

### 2.3.1 cProfile

`cProfile` は Python 組み込みのプロファイラで、これを使えば長いスクリプトの

ボトルネックの位置がだいたいわかる。この例では、乱数ジェネレーターを前節の mode 関数の中に入れて、プロファイリングをしてみよう。

```
import numpy as np
from collections import Counter

def mode_using_counter(n_integers): // Counterを使った最頻値の計算
    random_integers = np.random.randint(1, 100_000, n_integers)
    c = Counter(random_integers)
    return c.most_common(1)[0][0]
```

プロファイラを実行するには、次のようにコマンドを実行する。

```
%%prun  
mode_using_counter(10_000_000)
```

例えば、このように出力されるはずだ。

25 function calls in 2.585 seconds

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	2.467	2.467	2.467	2.467	{built-in method _collections._count_elements}
1	0.104	0.104	0.104	0.104	{method 'randint' of 'numpy.random.mtrand.RandomState' objects}
1	0.007	0.007	2.584	2.584	<string>:1(<module>)
1	0.006	0.006	0.006	0.006	{built-in method builtins.max}
1	0.000	0.000	2.467	2.467	__init__.py:649(update)
1	0.000	0.000	2.585	2.585	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'reduce' of 'numpy.ufunc' objects}
1	0.000	0.000	2.577	2.577	3744758285.py: (mode_using_counter)
1	0.000	0.000	2.467	2.467	__init__.py:581(__init__)
1	0.000	0.000	0.000	0.000	fromnumeric.py:69(_wrapreduction)
1	0.000	0.000	0.006	0.006	heapq.py:521(nlargest)
1	0.000	0.000	0.006	0.006	__init__.py:600(most_common)
1	0.000	0.000	0.000	0.000	<__array_function__ internals>:177 (prod)
1	0.000	0.000	0.000	0.000	{built-in method _abc._abc_instancecheck}

```

1    0.000    0.000    0.000    0.000 fromnumeric.py:2927(prod)
1    0.000    0.000    0.000    0.000 abc.py:117(__instancecheck__)
1    0.000    0.000    0.000    0.000 {built-in method
                                         builtins.isinstance}
1    0.000    0.000    0.000    0.000 {built-in method numpy.core.
                                         _multiarray_umath.
                                         implement_array_function}
1    0.000    0.000    0.000    0.000 {built-in method builtins.getattr}
1    0.000    0.000    0.000    0.000 fromnumeric.py:70(<dictcomp>)
2    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
1    0.000    0.000    0.000    0.000 {built-in method builtins.iter}
1    0.000    0.000    0.000    0.000 {method 'disable' of
                                         '_lsprof.Profiler' objects}
1    0.000    0.000    0.000    0.000 fromnumeric.py:2922
                                         (_prod_dispatcher)

```

この出力の tottime 列を見ると、このコードを実行する際にコンピュータが最も時間を費やした箇所がわかる。それは built-in method \_collections.\_count\_elements で、Counter 関数である。次に時間がかかったのは method 'randint' of 'numpy.random.mtrand.RandomState' objects で、これは乱数のリストを作成するステップである。他のステップはほとんど時間がかからなかった。`cProfile` を使うことの欠点は、関数呼び出しとコード内の行との対応を自分でとる必要があることだ。

`SnakeViz` パッケージを使えば、`cProfile` の結果をグラフィカルに表示できる。

`SnakeViz` は次のコマンドでインストールできる。

```
$ pip install snakeviz
```

Jupyter ノートブックで作業している場合は、`SnakeViz` の機能拡張を利用する。機能拡張は次のコマンドでロードできる。

```
%load_ext snakeviz
```

そして、次のように実行する。

```
%%snakeviz
mode_using_counter(10_000_000)
```

すると、`cProfile` の結果がグラフィカルに表示される（図2-1）。

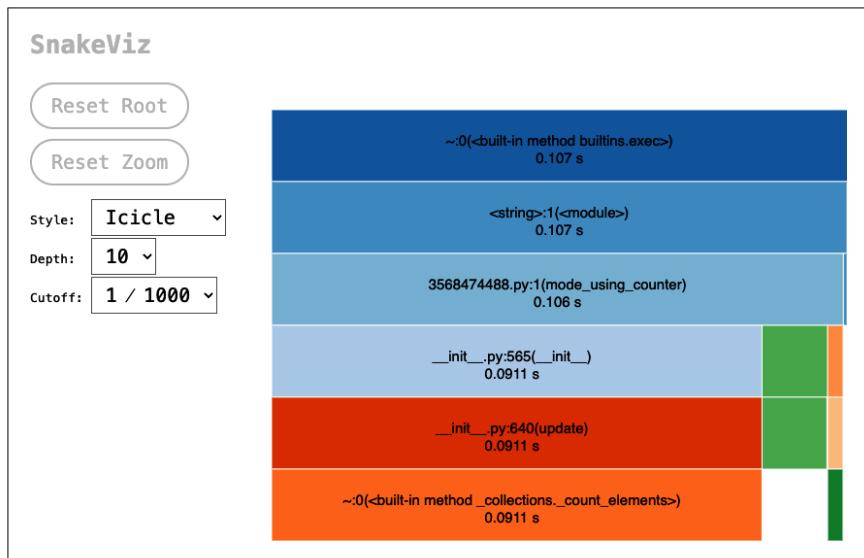


図2-1 SnakeViz を使って cProfile の出力を表示

このほうが、cProfile のデフォルトの出力よりもわかりやすいだろう。

### 2.3.2 line\_profiler

cProfile は、Python の内部動作の詳細を含め、コンピュータがどのように時間を使っているかの詳細な内訳を教えてくれる。しかし、その出力は読みやすいとは言えず、コードから呼び出される Python の組み込み関数に「深入り」しきていると言えるかもしれない。パッケージ `line_profiler` (<https://oreil.ly/KcGKg>) のほうが読みやすい出力を提供してくれる。

`line_profiler` はデフォルトではインストールされていないので、次のコマンドでインストールする必要がある。

```
$ pip install line_profiler
```

次に、`line_profiler` を Jupyter ノートブックに機能拡張としてロードする。

```
%load_ext line_profiler
```

どの関数を見るかは、`-f` フラグを使って明示的に指示する必要がある。

```
%lprun -f mode_using_counter mode_using_counter(10_000_000)
```

出力は例えば次のようになる。

```
Timer unit: 1e-09 s

Total time: 2.60904 s
File: .../3744758285.py
Function: mode_using_counter at line 1

Line # Hits          Time      Per Hit  % Time  Line Contents
=====
1                               def mode_using_counter
                                (n_integers):
2     1  124198000.0  124198000.0      4.8  some_list = np.random.randint
                                         (1, 100000, n_integers)
3     1  2479064000.0  2479064000.0     95.0  c = Counter(some_list)
4     1    5780000.0    5780000.0      0.2  return c.most_common(1)[0][0]
```

この出力は、`cProfile` の出力よりもはるかに読みやすい。この関数の時間の 4.8% が乱数リストの生成に費やされ、95% が組み込み関数 `Counter` を使ってカウントするのに費やされていることがわかる。このことから、この関数のパフォーマンスを向上させたいのであれば、`Counter` を含む行に注目すべきであることがわかる。最も時間がかかる行の特定は、長い関数やスクリプトでは特に重要である。

また、「デコレーター」を使ったスクリプトとともに `line_profiler` を使って、プロファイルしたい関数を特定することもできる。この方法の詳細については、`line_profiler` のドキュメント (<https://oreil.ly/xwGkQ>) を参照されたい。

### 2.3.3 Memrayによるメモリプロファイリング

コードの実行時間のプロファイリングだけでなく、メモリ使用量のプロファイリングも可能だ。プロジェクトによってはメモリ使用量の最適化が必要になる場合があるだろう。データのサイズが大きくなると、使用しているハードウェアの上限に達する可能性があるため、この点を考慮することが重要だ。CPU がコードの実行よりもメモリ管理に多くの時間を費やすようだと、全体の実行時間が長くなってしまう。

`Memray` (<https://oreil.ly/n-HqF>) は、Bloomberg によって開発されたメモリプロファイリング用のツールで、メモリ使用量に関するさまざまなレポートを出力して

くれる。

Memray は次のコマンドでインストールできる。

```
$ pip install memray
```

`mode_using_counter` 関数を含むスタンドアローンの Python スクリプト (`mode_using_counter.py`) で Memray を使う方法を見てみよう。

```
import numpy as np
from collections import Counter

def mode_using_counter(n_integers):
    random_integers = np.random.randint(1, 100_000, n_integers)
    c = Counter(random_integers)
    return c.most_common(1)[0][0]

if __name__ == '__main__':
    print(mode_using_counter(10_000_000))
```

スクリプトのメモリ使用量を収集するには、次のコマンドを使って Memray を実行する。

```
!memray run mode_using_counter.py
```

例えば、このような出力が表示されるはずだ。

```
Writing profile results into memray-mode_using_counter.py.27008.bin
79316
[memray] Successfully generated profile results.
```

```
You can now generate reports from the stored allocation records.
Some example commands to generate reports:
```

```
/Library/Frameworks/Python.framework/Versions/3.9/bin/python3.9 -m memray flamegraph memray-mode_using_counter.py.27008.bin
```

Memray は結果のバイナリファイル (`.bin`) を生成する。このバイナリファイルから、他のコマンドを実行してさまざまなレポートを作成できる。中でも `flamegraph` コマンドが最も便利である（次のコマンドの `.bin` ファイルの部分は、Memray が生成したファイル名に置き換える必要がある）。

```
!memray flamegraph memray-mode_using_counter.py.27008.bin
```

```
Calculating high watermark... ━━━━━━━━ 100% 0:00:00--::-
Processing allocation records... ━━━━━━ 100% 0:00:00--::-
Wrote memray-flamegraph-mode_using_counter.py.27008.html
```

これで（ノートブックと同じフォルダに）HTML ファイルが生成される。これをブラウザで開くと、レポート（memray flamegraph report）が画面に表示される（図2-2）。

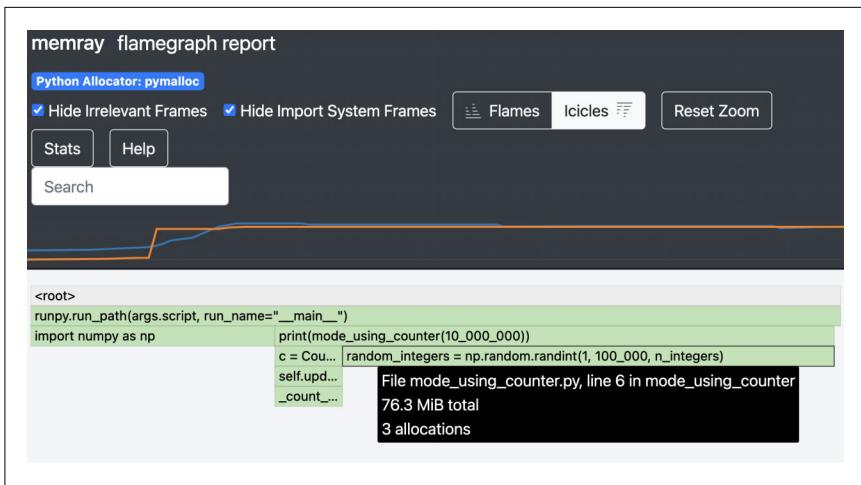


図2-2 memray flamegraph report

マウスオーバーすると、各コード行のメモリ使用量が表示される。この例では、次の行が最も使用量が多く、76.3MB だった。

```
random_integers = np.random.randint(1, 100_000, n_integers)
```

これは単純な例だが、もっと長いスクリプトの場合、このツールはメモリ使用量が多いコード行を突き止めるのに便利だ。これによって、スクリプトのどこに力を注ぐべきかがわかる。

また、memray 機能拡張を使えば、Jupyter ノートブックで Memray を使うこともできる。詳しくは Memray のドキュメント (<https://oreil.ly/i-E-J>) を参照され

たい。

どの方法を使うにせよ、「2.1 パフォーマンス向上の方法」に挙げたような方法を使ったコードの最適化の前にパフォーマンスを測定することが重要だ。事前に行うことで、どこにボトルネックがあるのかがわかり、正しい場所に力を注げることになる。

## 2.4 時間計算量

前節では、コードのボトルネックを見つける方法を紹介したが、将来データ量が増えることが予想される場合や、コードの一部の実行に非常に時間がかかり、測定ができなくなるといった場合もある。そこで**時間計算量** (time complexity) の概念が役に立つ。

時間計算量（あるいは単に「計算量」）は、入力のサイズが大きくなるにつれて、アルゴリズムの実行時間がどのように増加するかを説明してくれるものだ。これは全体的な傾向を意味し、特定のハードウェアには依存しない概念である。より具体的に考えるために、次の質問の答えを考えてみてほしい。

- 最初に  $n$  個の要素を持つ数字のリストに対して、あるコードを実行する
- 次にリストの長さを 2 倍にした場合、コードの実行時間はどうなるだろうか。  
時間は変わらないか、それとも 2 倍になるか、あるいはそのいずれでもないか

### 2.4.1 時間計算量の見積もり

具体例を見てみよう。この関数は、数値のリストと各数値に適用する重みのリストを入力として、加重平均を計算するものだ。

```
def weighted_mean(list_of_numbers, weights):
    running_total = 0
    for i in range(len(list_of_numbers)):
        running_total += (list_of_numbers[i] * weights[i])
    return (running_total/sum(weights))
```

`list_of_numbers` の長さが 2 倍になると、`for` ループの繰り返し回数も 2 倍になる。この関数のほかの行 (`running_total` の初期化や、リストの長さでの割り算) は実行時間が一定だ。この関数の実行時間は、処理のステップ数で見積もることができる。合計を初期化するのが 1 ステップで、合計への加算がそれぞれ別のステップに

なり（合計  $n$  ステップ）、結果を返すのもまた別の 1 ステップだ。したがって、 $n$  をリストの長さとすると、 $n + 2$  が全体のステップ数ということになる。 $n$  のサイズが大きくなるにつれて、ステップ数は直線的に増加するが、最後の  $+2$  は一定のままである。したがって、 $n$  が十分大きければ定数部分は無視できる。

次の関数は 2 つのリストを入力として受け取り、それらの共分散（2 つの確率変数間の相関の強さの尺度）を計算する（繰り返しになるが、これが最良の計算方法というわけではなく、あくまでも説明のための例である）。

```
def covariance(X, Y):
    cov_sum = 0
    for i in range(len(X)):
        for j in range(len(Y)):
            cov_sum += 0.5 * (X[i] - X[j]) * (Y[i] - Y[j])
    return cov_sum / (len(X) ** 2)
```

この関数には入れ子になった `for` ループがあるため、 $X$  と  $Y$  の両方のサイズが 2 倍になると、結果を計算するのにかかるステップ数は 4 倍になる。この関数では、 $X$  と  $Y$  の長さは同じになるので、コードの実行時間は、データサイズの二乗に比例して増加することになる。 $n$  をリストの長さとすると、全体のステップ数は  $n^2 + 2$  という式で表現できる。

実は同じ計算をする、より高速な方法がある。次の関数も共分散を計算するものだが、`for` ループを 1 回しか使っていない。

```
def covariance_fast(X, Y):
    avg_X = sum(X) / len(X)
    avg_Y = sum(Y) / len(Y)

    result = 0
    for i in range(len(X)):
        result += (X[i] - avg_X) * (Y[i] - avg_Y)

    return result / len(X)
```

ここでさらに注意したいのは、「2.2 実行時間の計測」や「2.3 プロファイリング」で紹介した時間計測やプロファイリングの手法が、ネストされた `for` ループが関数の中で最も時間がかかる部分だということを教えてくれる可能性があるという点だ。こうした分析をすることで、より効率的なアルゴリズムを見つけられるのだ。

## 2.4.2 オーダー記法

オーダー記法とは、データが大きくなるにつれて、コードがどの程度遅くなるかを表したものだ。

—— Ned Batchelder (PyCon 2018)

前節で紹介したような、「コードの実行時間がどのように増加するか」というパターンを、より形式的に記述するのに、**オーダー記法** (Big O <sup>オウ</sup> notation) が用いられる。これは、データセットが大きくなるにつれて、計算量（計算に必要なステップ数）が、どの程度の割合で増加するかを表現したものだ。使用するハードウェアには依存しない指標で、どのようなコンピュータで実行するかに関係なく、コードのパフォーマンスを予測するために用いられる。

オーダー記法は、前節の「重み付き平均」の例で登場したステップ数の式  $n + 2$  に似たようなものだが、定数部分は省かれる。実行時間は入力の大きさ  $n$  の関数として記述され、 $O(f(n))$  と書かれる。 $f(n)$  は、 $n$  の関数としてアルゴリズムが実行する演算数の上限である。例えば、加重平均の例は  $O(n)$  のアルゴリズムということになる。

加重平均の例が  $O(n)$  であって  $O(n + 2)$  ではないことを不思議に思うかもしれないが、 $n$  が十分大きくなると、2の影響は限りなく小さくなる。オーダー記法は、 $n$  が大きくなるにつれてどのように計算時間が増加するかの割合を示すもので、ステップが2つ増えてもそれは重要な意味を持たない。あくまでも、異なるアプローチ（アルゴリズム）を比較するのに役立つ近似値であり、コードの実行にかかる時間の正確な尺度ではない。あるコードの実行に  $n^2 + n$  ステップかかるのならば、その計算量は  $O(n^2)$  ということになる（ $+n$  の部分は、 $n$  が大きくなると  $n^2$  の部分に比べると無視できるほど小さくなる）。

よく登場するオーダー記法を挙げてみよう。

$O(1)$

**定数時間**とも呼ばれ、実行時間がデータセットのサイズに依存しないことを意味する。例としては、リスト内の特定の要素のルックアップが挙げられる。リストのサイズが小さくても、非常に大きくても、リストの最後の要素を返すにはほぼ同じ時間がかかる

$O(n)$

線形時間とも呼ばれ、重み付き平均の例で示したように、データセットの大きさに比例して実行時間が増加することを意味する

$O(n^2)$

二乗時間とも呼ばれ、データセットの大きさの二乗に比例して実行時間が長くなることを意味する。共分散の計算を2重のループを使って行った場合が当てはまる

計算量の増加具合をグラフに示したものが図2-3だ。この図には指數時間  $O(2^n)$ （「指數関数時間」とも呼ばれる）、対数時間  $O(\log n)$ 、線形対数時間  $O(n \log n)$  も示されている。データセットの大きさ  $n$  に対して、実行時間がそれぞれ  $2^n$ 、 $\log n$ 、 $n \log n$  倍に増加することを意味する。

再起的なアルゴリズムの多くは  $O(2^n)$  になる。また、多くの検索アルゴリズムは、 $O(n \log n)$  になる。

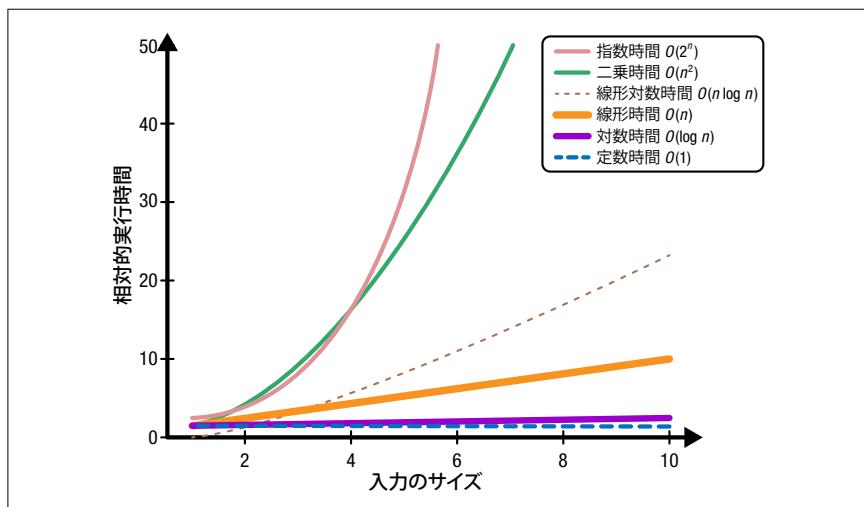


図2-3 入力のサイズと実行時間の関係

図からわかるように、 $O(n^2)$  や  $O(2^n)$  の場合、サイズが少しだけ大きくなるだけでかなり時間がかかるようになる。したがって、高性能なコードではこうしたアルゴリズム

ムは避けることになる。

また、「早すぎる最適化」を避けるために、ユースケースにとって現実的な  $n$  の範囲を検討しておく必要がある。例えば、 $n$  が 5 より大きくなることがないのであれば、コードの実行時間が線形に増加しそうが指数関数的に増加しそうがあまり問題にはならない。

次章では、データ構造の選択について学ぶ。その説明を読めば、オーダー記法の重要性をより実感できるだろう。

## 2.5 まとめ

コードのパフォーマンスを考える場合、まず最適化すべき時期を知る必要がある。データ量が少ない場合や、コードがすべての要件を満たしている場合は、最適化に時間をかける必要はない。

しかし、コードのパフォーマンスを向上させる必要があるのなら、最初にするべきなのは測定を行い、どこにボトルネックがあるかを特定することだ。最も簡単な方法は、関数やコードの実行時間を計ることだ。詳細が必要な場合や、長いスクリプトの「内訳」を知りたい場合は、プロファイラを使う。

また、データ量が増えるについて、コード実行時のステップ数がどのように増加していくかを考えることもパフォーマンスの向上に役立つ。これを説明するためにオーダー記法がよく使われる。

ある程度の規模の処理になるとコードの書き方は無数にあり、パフォーマンスもそれぞれ異なることをしっかりと理解しておこう。

# 3章 データ構造

2章では、コードの性能を測定する方法を説明した。本章では、データ構造の選択がコードのパフォーマンスにどのような影響を与えるかを示し、最適なデータ構造の選択方法について説明する。

データサイエンティストとしてコードを書くとき、データを記憶するためにさまざまなデータ構造を使う必要がある。どのデータ構造を使うか、多くの選択肢があるが、その中から解決しようとしている問題にできるだけ適したものを見つける必要がある。

実際、高性能プログラミングとは、データをどのように操作しようとしているのかを知り、その操作を高速にできるデータ構造を採用することと言っても過言ではありません。

—— 『ハイパフォーマンス Python 第2版』（オライリー・ジャパン）

取り組んでいる問題に対して正しいデータ構造を使うことが重要である。正しいデータ構造を選択できれば、パフォーマンスが向上し、使いやすくなる。また、コードがより予測しやすいものになり理解も容易になる。

本章では、データサイエンス (DS) のコードを書くときによく使うデータ構造について説明する。具体的には、Python のリスト、タプル、辞書、集合と、NumPy の配列、そして pandas の DataFrame だ。それぞれの長所と短所を説明し、時間とメモリ使用量の両面でより良いパフォーマンスを発揮するためには、どのような場面でどのような構造を選択すればよいのかを検討していく。また、最適なパフォーマンスを得るために各データ構造をどのように使うのがベストなのかも見ていく。

ここで取り上げるものの中にも、リンクリスト、ヒープ、キュー、二分探索木な

ど、多くのデータ構造を目にすることはあるだろう。こうしたデータ構造は、DSで用いることは必ずしも多くはなく、ソフトウェアエンジニアリング分野において（特にPython以外のプログラミング言語において）より重要になる。データ構造についてより詳しく学びたいのならば、Jay Wengrow著『A Common-Sense Guide to Data Structures and Algorithms in Python』(Pragmatic Bookshelf)をお勧めする。

## 3.1 Python のネイティブなデータ構造

本節では、Pythonの4つの基本的なデータ構造（リスト、タプル、辞書、セット）について説明する。すでに馴染みの人も多いだろうが、こうしたデータ構造がどのように実装されているのか、また一般的な演算に対してどの程度の時間計算量になるのかを見していく。説明のために「2.4 時間計算量」で説明したオーダー記法を使う。特定のデータ構造を使ったとき、どのような操作が素早くでき、どのような操作に時間がかかるてしまうのかを直感的に理解することには意味がある。特定のユースケースに最適なデータ構造を選択するのに役立ち、コードを最適化する必要がある場合、どこに注力すればよいかのヒントを与えてくれるはずだ。

### 3.1.1 リスト

リストはPythonで必須のデータ構造であり、頻繁に使われる。Pythonのリストは配列の一種であり、「順序」を持つデータ構造だ。リストの最初の要素、3番目の要素、といった具合に要素を特定する（「ルックアップ」する）ことができる。

Pythonのリストは動的配列であり、要素が追加されるとサイズが変わる。また、異なる型の要素を記憶できるので、例えば同じリストに文字列と整数の両方を記憶できる。

Pythonはリストのサイズに応じて連続したメモリ領域を確保し、ある要素の次の要素は、メモリ上でも隣に置かれる。このため、リストの要素にアクセスするのは簡単だ。Python インタプリタはリストの先頭要素のメモリ内の位置を知っており、例えばリストの5番目の要素にアクセスする場合、先頭から数えて5つ目の位置を見ればよい。

ここで、リストが大きくなるにつれて、リスト内の要素のルックアップにかかる時間がどう変化するかを測定してみよう。まず、10個の要素を持つリストを作成する(`example/ch03/ch03.ipynb`)。

```
small_list = list(range(10))
```

次に、`%%timeit`（「2.2 実行時間の計測」参照）を使って、リストの最後の要素を検索するのにかかる時間を測定する。

```
%%timeit
last_element = small_list[-1]
```

```
15.3 ns ± 0.034 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
```

次に、1万個の要素のリストを作成し、同じ実験を行う。

```
large_list = list(range(10_000))
```

そして、最後の要素を検索するのにかかる時間を測定する。

```
%%timeit
last_element = large_list[-1]
```

```
15.4 ns ± 0.0461 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
```

2番目のリストが1番目のリストの1,000倍のサイズであるにもかかわらず、かかる時間はどちらもほぼ同じである。リスト検索の計算量を「2.4.2 オーダー記法」で説明したオーダー記法で表すと  $O(1)$ （一定時間）ということになる。リストがどんなに大きくても、その中の要素にアクセスするのにかかる時間はほぼ同じである。

リストに要素を追加するたびに、メモリに余分なスペースが必要になる。Pythonは元のリストの長さ以上の余分なスペースを確保するが、このスペースが一杯になると、リスト全体を、連続したスペースが確保できる別のメモリ領域にコピーする必要が生じる。リストの末尾に追加するのは  $O(1)$  だが、確保されているスペースが一杯になったらリストをコピーするためのオーバーヘッドが生じる。

リストの途中に何かを挿入すると、その新しい要素以降のすべてを新しいメモリ位置に移動させる必要がある。つまり、`insert` 操作のオーダーは  $O(n)$  である（所要時間は  $n$  に比例する）。リストの途中から要素を削除する場合も  $O(n)$  である。リストの挿入と削除はどちらも  $O(n)$  であるが、これはリストが常に連続したメモリ上に

置かれるためである。

特定の長さを持つリストに要素を追加することがわかっている場合、`.append()` は最も効率的な方法ではないかもしれない。リスト内包表記 (<https://oreil.ly/ZqTu0>) を使うか、適切な長さの、すべてゼロからなるリストを作成し、必要な要素の値を更新するほうがよいかもしれない。リストの先頭と末尾に項目を追加したい場合は、データ構造として `collections` モジュールの `deque` (double-ended queue。**両端キュー**)。<https://oreil.ly/7a7CU>) が使える。

リスト内の要素を検索する必要がある場合、検索対象の要素とリスト内のすべての項目を比較する必要がある。リストが大きくなるにつれて処理時間がどのように変化するかは、前のコード例のリストを使って測定できる。

まず、10 個の要素を含むリストを検索するのにかかる時間を測定する。

```
%%timeit
4200 in small_list # 4200がsmall_listにあるか
```

66.2 ns ± 0.163 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)

次に、10,000 要素のリストについて同じことを測定する。

```
%%timeit
4200 in large_list
```

24.1 µs ± 82.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

およそ 1,000 倍の時間がかかる ( $1000\text{ns} = 1\mu\text{s}$ )。この方法で Python のリストを検索すると  $O(n)$  になる。リストの検索にはより効率的な方法があり、例えば二分探索 (binary search) の計算量は  $O(\log n)$  だ。しかし、何かが存在するかどうかを頻繁に検索する必要がある場合は、以下で説明する辞書やセットを使うほうがよいだろう。

### 3.1.2 タプル

Python のタプルも配列だが、サイズは固定だ。タプルはイミュータブル（不变）で、一度作成したら変更はできない。例えば、タプルの最後に要素を追加することはできない。

タプルは、記憶したい要素の数が限られており、要素が変化しない場合に便利な

データ構造だ。実は、タプルはメモリに保存されるのではなく、Python のランタイムに「キャッシュ」されるので、タプルの要素を検索するのはリストよりも速い。タプルの要素の検索は  $O(1)$  だ。

### 3.1.3 辞書

辞書も Python には欠かせないデータ構造で、「キーと値」の組から構成される。例えば人名と住所といったように、それ自体に順序を持たない、何らかの関連を持つデータのペアが存在する際に辞書が使える。

内部的には、Python の辞書は「ハッシュテーブル」を使って実現されている。ハッシュ関数を使い、キーをリストのインデックス（添字）に対応させる（図3-1）。この図の場合、「Heather Hancock」というキーはハッシュ関数によってインデックス 3 に変換され、これによってこの名前と結びつけられた住所を取り出すことができる。

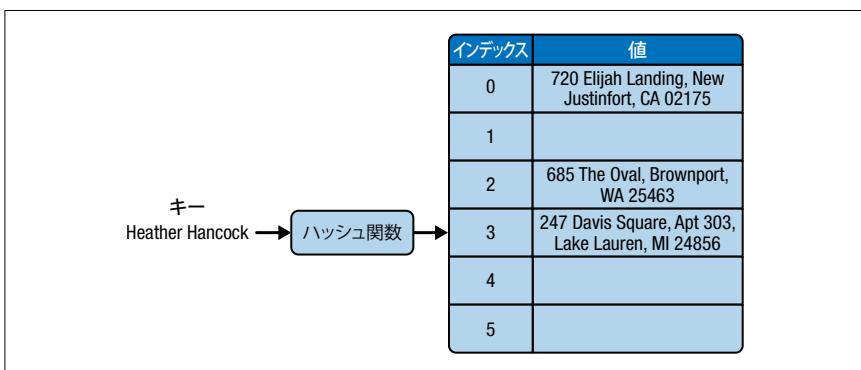


図3-1 ハッシュテーブルを使った辞書の検索

ハッシュ関数は、辞書のキー（文字列または整数）を、値のリストの該当するインデックスである整数にマッピングする。ハッシュ関数にはさまざまな種類がある。整数の「法 (modulus)」を取るのが簡単な例だが、Python 辞書のハッシュ関数はもっと複雑だ。

ハッシュ関数は、同じキーに適用されるたびに同じ整数を返さなければならない。辞書のキーも、正しい値を返せるように一意でなければならない。辞書のキーは、文字列、整数、float などのハッシュ可能な型である必要がある。Python のリストはこの目的には使えない。

ハッシュテーブルは便利な特徴を持っている。値の検索は  $O(1)$  なのだ（常に一定）。これはリストの要素のルックアップと同じだ。だが、辞書では要素の順序は問題にしない。「キーと値のペア」の挿入、更新、削除はすべて  $O(1)$  である。

辞書要素のルックアップがきちんと機能するか実験を行ってみよう。Faker ライブラリ (<https://oreil.ly/4JRuL>) は、このような場合に非常に便利だ。名前、住所、電子メールアドレスなどの個人データを含め、現実的と思われる偽のデータを作成してくれる。

次のコマンドで Faker のジェネレーターを初期化し、偽のデータを生成できる。

```
from faker import Faker  
  
fake = Faker()
```

そして、この辞書を使って実験を行う。

```
small_dict = {}  
  
for i in range(10):  
    small_dict[fake.name()] = fake.address()  
name = list(small_dict.keys())[0] # ←先頭要素のキーを得る（他のキーでも可）  
name  
  
'Carla Bryant'
```

10 個のキーと値のペアを持つこの小さな辞書で、ルックアップにかかる時間を計ることができる。

```
%%timeit  
small_dict['Carla Bryant'] # ←実際にあるキーを入力する必要がある
```

```
9.17 ns ± 0.0189 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
```

次に 10,000 のキーと値のペアの辞書を作ってみよう。

```
large_dict = {}  
for i in range(10_000):  
    large_dict[fake.name()] = fake.address()
```

```
list(large_dict.keys())[0]
name
```

```
'Margaret Cox'
```

同じように、キーから値を検索するのにかかる時間を測定する。

```
%%timeit
large_dict['Margaret Cox']
```

```
9.25 ns ± 0.00149 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
```

この辞書は前の辞書の 1,000 倍もあるのに、かかる時間はほぼ同じだ。辞書のルックアップは非常に効率的だ。欠点は、辞書がメモリを大量に消費するケースが多いことだ。

### 3.1.4 セット（集合）

セット（集合）もまた、固有の順序を持たないデータに有用なデータ構造である。セットは Python では、辞書に似たハッシュテーブルを使って実装される。しかし、キーと値のペアの代わりに、一意なキーだけを持つ。つまり、セット内の要素はすべて一意でなければならない。

セットには、項目の挿入、削除、更新がすべて  $O(1)$  であるという辞書と共通の性質がある。セット内の要素をルックアップするのも  $O(1)$  である。リストをセットに変換してから長さを調べるのは、リスト内のユニークな要素の数をカウントする効率的な方法だ。

セットの要素のルックアップは  $O(1)$  だが、リストの要素のルックアップは  $O(n)$  なので、リストをセットに変換してからルックアップするほうが効率的なのだろうか？ 実験をしてみよう。

先ほどの 10,000 要素の大きなリストを使って、リスト検索にかかる時間を測定できる。

```
%%timeit
4200 in large_list
```

```
14.6 µs ± 38.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

次に、リストをセットに変換し、ルックアップを実行するのにかかる時間を測定する。

```
%%timeit
large_set = set(large_list)
4200 in large_set
```

46.2 µs ± 106 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

リストをセットに変換してからルックアップを実行すると、リストをセットに変換する時間がかかるため、リストのルックアップの約3倍の時間がかかる。しかし、一度リストをセットに変換すれば、その後のルックアップは高速になる。

```
%%timeit
4592 in large_set
```

11.2 ns ± 0.0206 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)

セットのルックアップはリストのルックアップより1,000倍近く速い(1000ns = 1µs)。セットへの変換は、リストの中に特定の要素があるかを繰り返しチェックしたいときには最適な選択肢だが、チェック回数が多くないときには意味がない(繰り返しになるが、取り組んでいる特定の問題に対して、何がより速いかを測定して決める必要がある)。

なお、辞書とセットの内部構造については Luciano Ramalho のブログ記事(<https://oreil.ly/MFOuT>)に詳しい。



Pythonのネイティブなデータ構造としては、ここで取り上げなかったものもいくつかあるが、ユースケースによっては役に立つかもしれない。名前付きタブル(named tuple。<https://oreil.ly/TxRJc>)は、タブルに割り当てられた名前で要素を特定できるタブルである。dataclass(<https://oreil.ly/qtVog>)はデータの保存のために最適化されたクラスの一種である(クラスについては「4章 オブジェクト指向プログラミングと関数型プログラミング」で説明する)。

## 3.2 NumPy

本節では、Pythonで多次元配列を扱うためによく使われる数値演算ライブラリで

ある **NumPy** について見ていく。そのコアとなるデータ構造は `ndarray` と呼ばれる  $n$  次元配列である。DS で NumPy が人気がある理由を説明し、続いて NumPy を使って何ができるかを見ていく。さらには、パフォーマンスに関するいくつかの留意点も説明する。

NumPy の詳細が知りたい場合は、Wes McKinney 著『Python によるデータ分析入門 第3版』(オライリー・ジャパン) が、使い方に関する完全なリファレンスガイドを載せているのでお勧めだ。

### 3.2.1 NumPy の関数

NumPy は DS で頻繁に利用される  $n$  次元の配列を表現するためのものである。機械学習 (ML) では、多くの計算が行列 (2 次元配列) のデータに対して行われる。NumPy は、Python ネイティブのリストに比べて、こうしたデータを扱うのが簡単で、多くの ML ライブラリは NumPy をうまく扱ってくれる。

Python の入れ子になったリストを使って多次元のデータ構造を作ることは可能ではあるが、ややこしい。例えば、2 次元の配列があったとして、最初の列の値を調べるにはどうしたらよいのだろうか。

入れ子になったリストを使ってこれを行うには、すべての行を調べて最初の値を取り出すコードを書く必要がある。ここでは、リスト内包表記を使った方法を紹介する。

```
python_2d_list = [[1, 3, 5], [2, 4, 6], [7, 9, 11]]  
  
first_column = [python_2d_list[i][0] for i in range(len(python_2d_list))]
```

しかし、NumPy があれば、NumPy のスライス構文を使って単純に最初の列の値を調べることができる。

```
import numpy as np  
  
np_2d_array = np.array([[1, 3, 5], [2, 4, 6], [7, 9, 11]])  
  
first_column = np_2d_array[:, 0]
```



Python のリストの複数の要素をルックアップし、ルックアップされた要素に新しい変数名を割り当てるとき、その要素の新しいコピーが作成される。しかし、同じことを NumPy で行うと、これは元の配列の「ビュー」となる。つまり、上の例で `np_2d_array` の値を変更すると、`first_column` の対応する値も変更される。ビューの作成は、コピーの作成よりも高速でメモリ効率が良い。これも NumPy が「リストよりもパフォーマンスが良い」理由の 1 つである。

同じように、多次元データに対する他の多くの演算子は、入れ子になったリストよりも NumPy を使うほうがはるかに簡単である。例えば、行列の乗算、配列の連結や転置、変換などである。

しかし、NumPy では（次節で詳しく説明するように）配列は 1 つの型のデータしか記憶できない。そのため、複数の型（例えば、文字列と整数）を含む場合には使えない。

### 3.2.2 NumPy のパフォーマンスに関する考察

NumPy が同じ型のデータしか扱えないのは不便に思えるかもしれないが、パフォーマンス面で見ると、Python のリストに比べて大きな利点がある。

例えば次のコードを見てみよう。

```
mixed_type_list = ["one", 2, 3.14]
```

Python のリストには文字列、整数、浮動小数点数を記憶できる。同じデータを記憶するために NumPy を使って次のようなコードを書いてもエラーにはならない。

```
mixed_type_array = np.array(["one", 2, 3.14])
```

しかし、出力してみると、すべて文字列に変換されていることがわかる。

```
print(mixed_type_array)
```

```
['one' '2' '3.14']
```

NumPy がすべての要素を同じ型になるように変換したわけだ。

NumPy では要素の型も一緒に保存される。これは `.dtype` 属性を使って調べることができる。

```
integer_array = np.array([1, 2, 3])
print(integer_array.dtype)
```

int64

この型は標準の Python の `int` ではなく `int64` になる。NumPy は標準的な Python の型とは異なる型を使っており、これはパフォーマンスの向上にもつながる（これについては本節の後ろのほうで触れる）。

NumPy のすべての要素が同じ型であることから、特に**ベクトル化計算** (vectorized calculation) で、大きなパフォーマンス向上につながる。通常の Python の `for` ループでは、Python インタプリタがどの関数を適用すべきかを知るために、すべての要素の型をチェックする必要がある。しかし、NumPy はこのステップをスキップして、多くの数値演算で最適化された C コードを実行できる。NumPy では、配列内のすべての要素を繰り返し演算するのではなく、一度にすべての要素を対象にして演算するのだ。これはベクトル化 (vectorization) と呼ばれる。ベクトル化による高速化については Itamar Turner-Trauring の記事 (<https://oreil.ly/DumcS>) に詳しい。

Python のリストと比較して、NumPy を使うことでパフォーマンスが向上する例を紹介しよう。まず、NumPy を使って 100,000 個のランダムな整数の配列を生成し、それを通常のリストに変換する。

```
random_int_array = np.random.randint(1, 100_000, 100_000)
random_int_list = list(random_int_array)
```

そして、Python の `sum()` を使ってリストを合計し、「2.2 実行時間の計測」で紹介した `%timeit` を使って、それにかかる時間を計測してみよう。

```
%%timeit -r 7 -n 100
sum(random_int_list)
```

1.91 ms ± 166 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

次に、ベクトル化された `np.sum()` を使って同じ演算を行い、その所要時間を測定

する。

```
%%timeit -r 7 -n 100
np.sum(random_int_array)
```

```
30.5 µs ± 20.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

約 100 倍速くなった。これは非常に大きなパフォーマンスの向上だ。実行したい演算がベクトル化された NumPy メソッドで可能ならば、Python ネイティブメソッドや独自のコードを書くよりも、間違いなく NumPy を使うべきだ。NumPy のドキュメント (<https://oreil.ly/11vRd>) で、どのような演算が利用できるか調べられる。

NumPy を使うときは、後で配列への要素の追加が必要になるかも考慮する必要がある。通常の Python のリストとは異なり、NumPy が配列のスペースを確保するとき、余分なスペースは許さない。そのため、NumPy に要素を追加する場合、配列全体を毎回新しいメモリ位置に移動する必要がある。つまり、NumPy に要素を追加するのは  $O(n)$  ということになる。配列を適切なスペースで初期化することは間違いなく価値があり、これを行う簡単な方法は `np.zeros` を使うことである。

```
array_to_fill = np.zeros(1000)
```

そうすれば、配列に要素を追加するのではなく、すでにあるゼロを置き換えるべきことになる。

また、NumPy のさまざまな型を利用することで、NumPy のメモリスペースを大幅に節約できる。NumPy はメモリにロードされるため、大きな配列を扱う必要がある場合には、サイズをできるだけ小さくすることが肝要だ。

先ほどと同じように、ランダムな整数の配列を生成する。

```
random_int_array = np.random.randint(1, 100_000, 100_000)
```

. nbytes 属性を調べれば、これが占めるバイト数を調べることができる。

```
random_int_array.nbytes
```

```
800000
```

この配列は 800,000 バイトのスペースを取る。

.dtype を使って型を見てみよう。

```
random_int_array.dtype
```

```
dtype('int64')
```

この場合は int64 だ。NumPy のドキュメント (<https://oreil.ly/beqRz>) を参照すると、これは 64 ビットの整数で、範囲は -9,223,372,036,854,775,808 から 9,223,372,036,854,775,808 だ。これほど大きな整数を必要としない場合は、配列を 32 ビット整数型に変換してもよい。十分小さければ、16 ビット整数を使うこともできる。これには.astype メソッドを使う。

```
random_int_array_32 = random_int_array.astype(np.int32)
```

バイト数を調べれば、半分のスペースしか占めていないことがわかる。

```
random_int_array_32 nbytes
```

```
400000
```

次のコードのように配列を初期化する際に dtype を指定することもできる。

```
small_array = np.array([1, 3, 5], dtype=np.int16)
```

NumPy のドキュメント (<https://oreil.ly/uHITq>) には、利用可能なすべての型のリストがある。

### 3.2.3 Dask を使った配列演算

NumPy を使ってみたが、「さらなる最適化が必要」となった場合、**Dask** ライブライ (https://www.dask.org) が（素晴らしい）選択肢となる。Dask を使えば、配列の演算子を並列処理でき、より高速な計算や、コンピュータのメモリに収まらないデータの処理が可能になる。Dask は NumPy とよく似たインターフェースを持つが、少し複雑さが増すので、パフォーマンス向上が必要な場合のみ使う価値がある。

Daskを使えば、ローカルなPCでも分散システム（クラスター）上でも一度に複数のコアを使って計算を実行できる。

Daskは、配列を「チャンク」に分割し、1つまたは複数のチャンクに対して一度に計算を実行し、その結果を結合する。例えば、非常に大きな配列の最大値を見たい場合、その配列をいくつかのチャンクに分割し、各チャンクの最大値を見つけ、次に各チャンクのすべての結果を突き合わせて最大値を見つける。すべての演算がこのように並列化できるわけではないが、取り組んでいる問題に適用できるなら、さらなる速度向上が期待できる。

Daskはまた、システムのメモリよりも大きなデータに対して計算を実行することもできる。すべてのチャンク（配列全体）を一度にロードせずに、各チャンクを順次計算する。

次のコマンドでDaskをインストールできる。

```
$ python -m pip install "dask[complete]"
```

同じ演算をNumPyとDaskで実行し実行時間を比較してみよう。

大きな配列の最大値を見つける例を実行する。`np.random.randint()`を使って、ランダムな整数で埋め尽くされたNumPyの配列を作成する。次のコードでは、30億個の整数の配列を作成している。

```
large_np_array = np.random.randint(1, 100000, 30_0000_0000)
```

標準的なNumPyでこの計算にかかる時間を測定する。

```
%%timeit -r 1 -n 7
np.max(large_np_array)
```

```
6.43 s ± 0 ns per loop (mean ± std. dev. of 1 run, 7 loops each)
```

Daskの配列はNumPyとは異なるデータ構造である。NumPyの多くのメソッドがDaskでも再現されているので、次のコードでランダムな整数のDask配列を作成できる。

```
import dask.array as da

large_dask_array = da.random.randint(1, 100000, 30_0000_0000)
```

NumPy の配列から Dask の配列を作成することもできる。

```
large_dask_array = da.from_array(large_np_array)
```

Dask では、NumPy に比べて余分なステップがある。まず演算子を初期化する必要がある（この場合は`.max()`）。その後、`.compute()` メソッドを使って計算する。NumPy と比較するために、このステップの時間を測定してみよう。

```
%%timeit -r 1 -n 7
array_max = large_dask_array.max()
array_max.compute()
```

```
4.39 s ± 0 ns per loop (mean ± std. dev. of 1 run, 7 loops each)
```

Dask の配列で最大値を見つけるほうが NumPy よりも速い。

各チャックの計算は、別のコアや別のコンピュータで行うこともできる。Dask Distributed (<https://oreil.ly/yTAkI>) を使うとタスクをスケジューリングしてくれる。これを使うには、Client オブジェクトを生成する必要がある。

```
from dask.distributed import Client

client = Client(n_workers=4)
client
```

図3-2 は、ノート PC のすべてのコアを使うようにしたローカルのクライアントである。

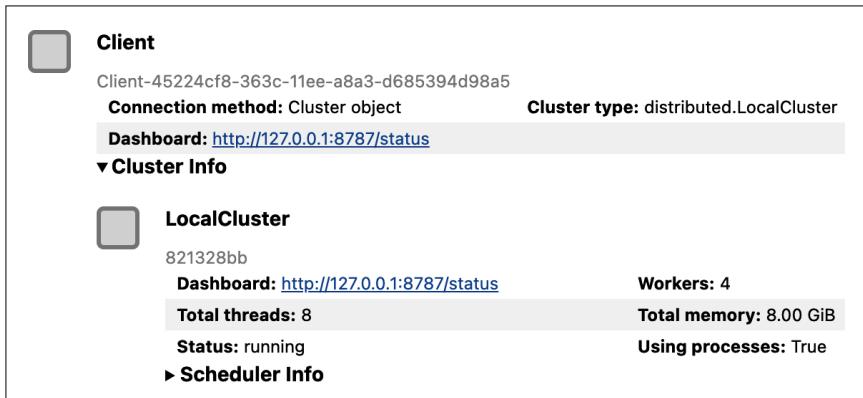


図3-2 Dask のローカルクラスターダッシュボード

このクライアントの準備ができたら、上で見たように Dask を使えば、指定したワーカー数に計算が分散される。詳細は、Dask のドキュメント (<https://oreil.ly/ES1I9>) に（素晴らしい）例がいくつがあるので、そちらを参照されたい。

### 3.3 機械学習における配列

ML アルゴリズムにおいて、配列の乗算は非常に大きな部分を占めている。ML におけるデータ（カテゴリー、画像、テキストなど）は行列（2 次元配列）やテンソル（高次元配列）に記憶されることが多い。ML モデルが大きくなるにつれて、モデルの学習を高速化するために、配列演算をより効率的にする努力がなされてきた。

最もよく使われている学習フレームワークである TensorFlow (<https://oreil.ly/zG6e>) と PyTorch (<https://oreil.ly/Bz2H0>) は、GPU 上で学習することで得られるスピードアップを利用した ML 用に最適化されたデータ構造を提供している。NumPy の配列はこの 2 つのフレームワークのどちらにも簡単に変換できる。

例を見てみよう。まず、先ほどと同じように NumPy の配列を作成する。

```
np_tensor = np.random.rand(4,4)
```

そして、TensorFlow の `tensor` 形式に変換する。

```
import tensorflow as tf  
  
tf_tensor = tf.convert_to_tensor(np_tensor)
```

または PyTorch の `tensor` 形式に変換する。

```
import torch  
  
pytorch_tensor = torch.from_numpy(np_tensor)
```

機械学習のデータ構造は GPU に最適化されている（GPU は多くのテンソル演算を並列実行できる）。多くの ML アルゴリズム（ニューラルネットワークのバックプロパゲーションを含む）は非常に並列化しやすい。

ここでスパース行列についても触れておきたい。これは、ほとんどがゼロの行列を格納するためのメモリ効率の良いデータ構造である。スパース行列が有用な例としては、テキストデータにおける単語の出現頻度のカウントが挙げられる。テキストのどのブロックでも、使っている語彙が大きければ、特徴量（feature）のほとんどはゼロになる。scikit-learn では、`CountVectorizer` がスパース行列 SciPy を利用する。NumPy の配列と相互に変換可能だが、スパース行列のほうがメモリ効率が良い。

ML モデルの学習の最適化はそれ自体が大きなトピックなので、ここでは詳しく説明しないが、PyTorch にも TensorFlow にも、プロファイラが用意されているので、学習用のコードのどこにボトルネックがあるのか検討するのに役立つであろう。

## 3.4 pandas

`pandas` は Python で DS を行うために非常によく使われているライブラリである。遅くてメモリを大量に消費するという評判があるが、本節でパフォーマンス向上させるさまざまな方法を紹介しよう。

`pandas` はもともと NumPy の上に構築された。NumPy の配列に適用される原則の多くは `pandas` の DataFrame にも適用されるが、`pandas` 固有の機能もある。2023 年 4 月にリリースされた `pandas` 2.0 では、バックエンドとして PyArrow というデータ構造 (<https://oreil.ly/vVzYz>) を利用するオプションが追加された。

NumPy 同様、`pandas` の DataFrame についても関連する関数をすべて紹介する

ことはしない。NumPy の節で紹介したが、Wes McKinney 著『Python によるデータ分析入門 第3版』（オライリー・ジャパン）がお勧めだ。

### 3.4.1 DataFrame の機能

pandas には 2 つの主要なデータ構造がある。DataFrame と Series である。DataFrame は 1 つ以上の Series で構成される。Series は、1 次元の NumPy に似ているが、インデックス（辞書のキーに相当するもの）も持っている。これにより、Series の中の項目を、その位置だけでなく、インデックスからも探すことができる。

次のように Series を作成できる。

```
import pandas as pd

usa_data = pd.Series([13.33, 14.02, 14.02, 14.25],
                     index=["2000", "2001", "2002", "2003"])
```

これは次のような構造で、年がインデックスとなる。

```
usa_data
2000    13.33
2001    14.02
2002    14.02
2003    14.25
dtype: float64
```

NumPy と同様、Series は連続したメモリブロックとして作成される。したがって、同じパフォーマンス上の考慮事項が適用されることになる。例えば、Series の末尾に新しい項目を追加するには、全体を新しいメモリ位置に移動させなければならぬので、時間がかかる。パフォーマンスの考慮点については、次節で詳しく説明する。

pandas の DataFrame は、pandas の Series を 2 次元に並べたもので、列インデックスもある。pandas の Series から次のように構築できる。

```
india_data = pd.Series([9.02, 9.01, 8.84, 8.84],
                       index=["2000", "2001", "2002", "2003"])

df = pd.DataFrame({'USA': usa_data, 'India': india_data})
```

これで次のような DataFrame が得られる。

	USA	India
2000	13.33	9.02
2001	14.02	9.01
2002	14.02	8.84
2003	14.25	8.84

NumPy とは異なり、DataFrame 内の各列は異なる型にできる。pandas は列の型として `object` も提供しており、異なる型のデータを Series 内で混在させることができる。pandas は欠損データを扱う関数も NumPy より多い。

pandas のデータ構造は、行と列の情報を持つ 2 次元の表データに特に便利だ。スプレッドシート形式のデータにも最適だ。データベースのテーブルと同じように使うこともでき、データに対して結合したりクエリしたりすることもできるが、プロジェクトが小さすぎて完全なデータベースをセットアップするほどではない場合に特に有用である。

### 3.4.2 DataFrame のパフォーマンス

NumPy のように、pandas ライブラリには、配列のすべての要素に一度に適用するベクトル化された演算がたくさん用意されている。これらの多くは、内部的に NumPy を利用している。ベクトル化された演算が利用できるような用途ならば、ほとんどの場合、これが最高のパフォーマンスを与えてくれるだろう。

例えば算術演算子がベクトル化される。

```
%%timeit
df["India_fraction"] = df["India"] / 100
```

```
21.3 µs ± 87.5 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

NumPy で利用可能なベクトル化演算子に加えて、pandas には `lower`、`strip`、`split` などのベクトル化文字列演算子 (<https://oreil.ly/CwzmR>) がある。通常の Python の `.lower()` メソッドよりも、`df['column_name'].str.lower()` を使ったほうが高速だ。

やろうとすることが組み込み関数では実行できない場合、定義した関数と `apply` を一緒に使うことができる (`apply` については次章で詳しく説明する)。下を見ればわかるように、これは計算にオーバーヘッドを追加する。

```
%%timeit
df["India_fraction"] = df["India"].apply(lambda x: x / 100)
```

21.5 µs ± 277 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

DataFrame のすべての行に対して何らかの演算を行いたい場合、すべての行を繰り返し処理するのが直感的に思えるかもしれない。そして、pandas はこのために `iterrows` メソッドを提供している。しかし、次に示すように、これはベクトル化された演算や `apply` を使用するよりもはるかに遅いので、避けるべきである。

```
%%timeit
df["India_fraction"] = [row['India'] / 100 for index, row in df.iterrows()]
```

40.1 µs ± 328 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

デフォルトでは、pandas の DataFrame はメモリにロードされる。そのため、DataFrame がコンピュータのメモリより大きい場合は問題がある。また、データ処理に必要なメモリ量の見積もりは難しいので、DataFrame がコンピュータのメモリより小さくても、問題が発生する可能性がある。pandas のメモリ管理は急速に改善されているが、DataFrame がコンピュータのメモリより大きい場合にできる基本的なことがいくつかある。まず、単純に作業したい列だけをロードする方法がある。例えば、`read_csv` の引数 `usecols` を使って、CSV ファイルから読み取りたい列を指定できる。次に、`chunksize` の引数を使用して、一度に行のサブセットのみを処理できるイテレータを作成する方法もある。最近、オプションの pandas バックエンドとして PyArrow (<https://oreil.ly/kCzMf>) が導入されたことで、よりメモリ効率の高いデータ型もサポートされるようになった。

より大量のデータを扱う場合のもう 1 つの選択肢は、Dask ライブラリを使うことだ。NumPy で見たように、データが大きすぎてメモリに収まらない場合、Dask はそれを分割してくれる。また、データ処理が遅く、複数のコアやコンピュータで並列処理したい場合にも最適だ。Dask は独自のデータ構造の DataFrame (<https://oreil.ly/yv913>) を持っており、データから直接作成することも、既存の pandas の DataFrame から作成することもできる。

また、DataFrame のパフォーマンスを向上させるために、**Polars** ライブラリ (<https://www.pola.rs>) への切り替えを検討してもよいかもしれない。これは

pandas ほどは「成熟」していないプロジェクトだが、より高速でメモリ効率に優れている。基礎となるコードは Rust で書かれており、Apache Arrow を使っているが、pandas に非常によく似たインターフェースを持っている。

## 3.5 まとめ

本章では、さまざまなユースケースでデータ構造を効果的に使う方法を見てきた。それぞれのデータ構造から得られるであろう期待される性能について考えるだけでなく、書いているコードで性能を測定し、さまざまなオプションを試して、取り組んでいる課題に対して何が最も効果的であるかを確認することが重要である。

多くの場合、Python の組み込みデータ構造だけで問題を解決できる。リストはインデックスを使ってルックアップする必要がある場合には効率的だが、リスト内の項目の検索は比較的遅い。辞書はキーを使った頻繁な検索に適しているし、セットは探している項目の存在をチェックする効率的な方法だ。

NumPy は多次元データに最適である。NumPy の配列は同じ型のデータしか保持しないため、ベクトル化された演算に対して、コンパイルされた高速な C コードを直接実行できる。これは非常に大きなスピードアップをもたらすので、実行したい計算にベクトル化されたものがあるのならば使うべきである。

また、NumPy の型を活用してメモリ使用量を減らせるほか、より大きな配列や並列化された計算には Dask を使うこともできる。ML プロジェクトでは、GPU に最適化されたデータ構造を使うとよい場合や、スペース行列が役に立つ場合もある。

pandas の DataFrame は表データや基本的なデータ分析に最適である。ただし、NumPy の配列と同じようなパフォーマンス上の考慮点がある。DataFrame の行を繰り返し処理するのは非効率だ。標準の pandas よりも良いパフォーマンスが必要な場合は、Dask ライブラリや Polars ライブラリを検討するとよいだろう。

本章で登場した概念についてさらに詳しく知りたい場合は、次の資料がお勧めだ。

- Alex Martelli + Anna Martelli Ravenscroft + Steve Holden + Paul McGuire 著『Python クイックリファレンス 第 4 版』(オライリー・ジャパン) には、Python のデータ構造の概要が書かれている
- NumPy と pandas の DataFrame の詳細については、Wes McKinney 著『Python によるデータ分析入門 第 3 版』(オライリー・ジャパン) を参照されたい

- Python のパフォーマンス最適化の詳細については、Micha Gorelick + Ian Osvald 著『ハイパフォーマンス Python 第 2 版』(オライリー・ジャパン) が詳しい
- Holden Karau + Mika Kimmins 著『Scaling Python with Dask』(O'Reilly Media) には、DS のための Dask の使い方が詳しく書かれている
- Web サイト「Python Speed」(<https://oreil.ly/ptdyp>) の DS 関連記事も参考になる

# 4章

## オブジェクト指向プログラミングと 関数型プログラミング

本章では、データサイエンス（DS）のキャリアで出会う可能性の高いプログラミングの「スタイル」を 2 つ紹介する。その 2 つとは、**オブジェクト指向プログラミング**（OOP：Object-Oriented Programming）と**関数型プログラミング**（FP：Functional Programming）だ。両方を知っておくことは非常に役に立つ。どちらかのスタイルのみを用いてコードを書くことがないとしても、どちらかのスタイルを多用するパッケージには遭遇するだろう。例えば、pandas や Matplotlib のような Python の標準的な DS パッケージがこれに該当する。OOP と FP を理解することで、出会ったコードをより効果的に使えるようになるはずだ。

OOP と FP は、基本的なコンピュータサイエンスの原則に基づいた**プログラミングパラダイム**である。プログラミング言語によっては、どちらか一方しかサポートしていないものや、どちらかを強く支持しているものもある。例えば、Java はオブジェクト指向言語であり、Python は両方をサポートしている。Python では全体的なスタイルとして OOP のほうが人気があるが、FP を使うこともできる。

こうした「スタイル」は、コードを分解する際の枠組み（フレームワーク）も与えてくれる。コードを書くときに、やりたいことをすべて 1 つの長いスクリプトとして書くこともできる。それでも問題なく実行できるが、メンテナンスやデバッグが大変だ。1 章で説明したように、コードを小さな塊に分解することが重要であり、OOP と FP のどちらもそのための方法を提案してくれる。

筆者自身は FP や OOP の原則に厳密に固執することはしない。OOP の原則に従って独自のクラスを定義することもあるし、FP の原則に準拠した関数を書くこともある。最近の Python プログラムのほとんどは、両方のパラダイムを組み合わせた中間位置にある。本章では、両方の基本を理解するために、両方のスタイルの概要を説明する。

## 4.1 オブジェクト指向プログラミング

Python ではオブジェクト指向プログラミングが一般的だ。しかし、この文脈における「オブジェクト」とは何だろうか。オブジェクトとは、名詞で説明できる「もの」と考えることができる。DS のコードでよく使われるオブジェクトは、pandas の DataFrame、NumPy の配列、Matplotlib の figure、scikit-learn の estimator などだ。

オブジェクトはデータを保持し、それに関連するアクションを持ち、他のオブジェクトと相互に作用する。例えば、pandas の DataFrame オブジェクトは、カラム名のリストを持っている。DataFrame オブジェクトに関連するアクションの 1 つは、列名の変更である。DataFrame は、新しい列として Series を追加することで、pandas の Series オブジェクトと相互作用することができる。

また、オブジェクトをカスタマイズされたデータ構造と考えることもできる。欲しいデータを保持できるように設計し、後でそれを使って何かできるようにするのだ。pandas の DataFrame を例にとると、pandas の設計者はデータを表形式で保持できる構造を考えた。そして、行や列でデータにアクセスし、そうした形式を使ってデータの演算ができる。

次節では、OOP の主な用語を紹介し、いくつかの例を見ていく。

### 4.1.1 クラス、メソッド、属性

**クラス** (class)、**メソッド** (method)、**属性** (attribute) は、OOP で頻繁に登場する重要な語だ。それぞれの概要を説明しよう。

- 「クラス」はオブジェクトを定義するものであり、そのオブジェクトを作るための青写真と考えることができる。この青写真から個々のオブジェクト（もの）が作成（生成）される。クラスから作成された「もの」をクラスのインスタンスと呼ぶ
- 「メソッド」は、そのクラスのオブジェクトに対してできる動作等を表す。オブジェクトの振る舞いを定義し、その属性を変更することもある
- 「属性」とは、そのクラスの性質を表す変数であり、オブジェクトごとに異なるデータを属性として記憶できる

こうした定義だけでは抽象的すぎて実感がわからないだろうから、具体例を挙げよ

う。今読み進めている本書『データサイエンスのためのソフトウェアエンジニアリング入門』は、`Book` というクラスのオブジェクトである。このオブジェクトの属性の 1 つはページ数で、もう 1 つは著者の名前である。このオブジェクトに対して呼び出せるメソッドは、`read`（読む）である。`Book` クラスのインスタンス（具体的な本）はたくさんあるが、それらはすべて一定のページ数を持ち、すべて読むことが可能だ。

Python では、クラスは通常、「キャメルケース」を使って命名される（单語の先頭を大文字で書いて境界を区別する）。例えば、`my_class` ではなく、`MyClass` のように書く。この形式の名前を見たらクラスであることが一目瞭然である。あるオブジェクトの属性やクラスを使うには、そのオブジェクトを表す変数のあとに「.」を書いて、その後ろに属性名あるいはメソッド名を書く。メソッドは引数を取ることができ、引数は (...) で囲んで指定する。引数がない場合でも () は省略することができないので、メソッド名に () を付けてメソッドであることを明示する場合もある。

例えば、pandas の DataFrame を考えてみよう。新しい DataFrame を作成するには、例えば次のようにする（多くの読者にはお馴染みだろう）。

```
import pandas as pd

my_dict = {"column_1": [1, 2], "column_2": ["a", "b"]}

df = pd.DataFrame(data=my_dict)
```

オブジェクト指向の観点から見ると、`df = pd.DataFrame(data=my_dict)` という行を実行した時点で、DataFrame 型の新しいオブジェクトが初期化され、その DataFrame の属性を設定するためのデータが渡されたことになる。

次のようにすれば、DataFrame の属性を調べることができる。

```
df.columns
df.shape
```

`.columns` と `.shape` が、オブジェクト `df` の属性である。

DataFrame オブジェクトに対して多くのメソッドを呼び出すことができる。メソッド `to_csv()` を呼び出してみよう。

```
df.to_csv("csv_file", index=False)
```

`csv_file` というファイルが作られて、`df` の内容が CSV 形式で書き出されているはずだ。

次は scikit-learn を使う例だ。新しいオブジェクトを作成して 1 つのメソッドを呼び出してみよう。2 つの配列 (`X_train` が特徴量、`y_train` がラベル) を使って機械学習 (ML) モデルをトレーニングする場合、次のようなコードを書くことになる。

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()
clf.fit(X_train, y_train)
```

この例では、新しい `LogisticRegression` 分類器 (classifier) オブジェクトを初期化し。`.fit()` メソッドを呼び出している。

別の例を挙げよう。これは 2 章の図 2-3 を作成するコードである。ここでは 2 つのオブジェクト、Matplotlib の `figure` オブジェクトと Matplotlib の `axes` オブジェクトが作成されている。その後、コメントで説明するように、これらのオブジェクトに対してさまざまな演算を行うためのメソッドが呼び出されている。

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

n = np.linspace(1, 10, 1000)
line_names = [
    "Constant",
    "Linear",
    "Quadratic",
    "Exponential",
    "Logarithmic",
    "n log n",
]
big_o = [np.ones(n.shape), n, n**2, 2**n, np.log(n), n * (np.log(n))]

fig, ax = plt.subplots() ❶
fig.set_facecolor("white") ❷

ax.set_xlim(0, 50) ❸
for i in range(len(big_o)):
```

```

    ax.plot(n, big_o[i], label=line_names[i])
ax.set_ylabel("Relative Runtime")
ax.set_xlabel("Input Size")
ax.legend()

fig.savefig(save_path, bbox_inches="tight") ❸

```

- ❶ オブジェクト `figure` と `ax` を初期化する
- ❷ `fig` オブジェクトの `set_facecolor` メソッドを、引数 "white" を指定して呼び出す
- ❸ ここからの数行のメソッドはすべて、`ax` オブジェクトを操作する
- ❹ 図を保存する (`fig` オブジェクトについて呼び出されるメソッド)

`figure` と `axes` には、この 2 つのオブジェクトを更新するために呼び出せるメソッドがたくさんある。



Matplotlib には 2 種類のインターフェースがあるため、混乱することがある。そのうちの 1 つはオブジェクト指向で、もう 1 つは MATLAB でのプロットを模倣するように設計されている。Matplotlib が最初にリリースされたのは 2003 年で、開発者は MATLAB を使い慣れた人たちに馴染みのあるものにしたいと考えた。最近では、前のコード例で示したようなオブジェクト指向のインターフェースを使うほうが一般的だ。しかし、どちらのタイプのコードもあるため、両方のインターフェースが存在しているわけだ。詳しくは次の記事が参考になる——「Why You Hate Matplotlib」(<https://oreil.ly/rYXUJ>)。

OOP にまつわる用語に馴染みがなくても、一般的な DS 関連のパッケージの多くでは、すでに頻繁に使われているはずだ。次のステップは、自分のコードでオブジェクト指向のアプローチを使えるように、自分のクラスを定義することだ。

## 4.1.2 クラスの定義

オブジェクト指向のスタイルで自分のコードを書きたいなら、クラスを定義する必要がある。簡単な例をいくつか紹介しよう。1 つ目は、あるテキストを決められた回数繰り返すもの、もう 1 つは、本書を通して使っている国連の持続可能な開発目標のデータ（1 章のコラム「本書のデータ」参照）を使ったものだ。

Python では、`class` 文を使って新しいクラスを定義する。

```
class RepeatText:
```

オブジェクトの新しいインスタンスが初期化されるたびに、いくつかの属性を記憶することが多い。これを行うために、`__init__`という特殊なメソッドを使う。

```
def __init__(self, n_repeats):
    self.n_repeats = n_repeats
```

`__init__`メソッドの最初の引数は、作成されるオブジェクトの新しいインスタンスを指す。慣習上、`self`という名前にしてしまうことが多い。この例では、`__init__`メソッドはもう1つ`n_repeats`という引数を取る。`self.n_repeats = n_repeats`という行は、`RepeatText`オブジェクトの新しいインスタンスが`n_repeats`属性を持つことを意味する。これは、新しいオブジェクトが初期化されるたびに行われる。

次のコードで新しい`RepeatText`オブジェクトを作成する。

```
repeat_twice = RepeatText(2)
```

そして、次のようにして`n_repeats`属性にアクセスできる。

```
print(repeat_twice.n_repeats)
```

2

別のメソッドを定義してみよう。`__init__`メソッドの定義と似ているが、通常の関数のように好きな名前を付けることができる。後述するように、オブジェクトの各インスタンスにこの振る舞いをさせたい場合は、`self`の引数が必要だ。

```
def multiply_text(self, some_text):
    print((some_text + " ") * self.n_repeats)
```

このメソッドは、対象とするクラスのインスタンスの`n_repeats`属性にアクセスする。つまり、このメソッドを利用する前に、`RepeatText`オブジェクトのインスタンスを作成する必要がある。



Python には、`self` を引数に取らない特殊なメソッドが 2 つある。クラスメソッドと静的 (static) メソッドだ。クラスメソッドは（クラスの 1 つのインスタンスだけでなく）クラス全体に適用されるもので、静的メソッドはクラスのインスタンスを作成せずに呼び出すことができるものだ。詳しくは Bill Lubanovic 著『入門 Python 3 第 2 版』(オライリー・ジャパン) などを参考にされたい。

新しく作成したメソッドは次のように呼び出すことができる。

```
repeat_twice.multiply_text("hello")
'hello hello'
```

このクラスの完全な定義を次に示す。

```
class RepeatText():

    def __init__(self, n_repeats):
        self.n_repeats = n_repeats

    def multiply_text(self, some_text):
        print((some_text + " ") * self.n_repeats)
```

別の例を見てみよう。1 章で紹介した国連の持続可能な開発目標のデータを使う。Goal5Data オブジェクトを作成し、ここに「Achieve gender equality and empower all women and girls」(<https://oreil.ly/tw77->) に関連するデータを記憶する。このオブジェクトは、この目標に関連するターゲットの 1 つ、「Target 5.5: Ensure women's full and effective participation and equal opportunities for leadership at all levels of decision-making in political, economic and public life.」のデータを保持する。

それぞれの国のデータを記憶するオブジェクトを作成して、同じように簡単に操作できるようにしたい。新しいクラスを作成してデータを保持するコードは次のようになる。

```
class Goal5Data():
    def __init__(self, name, population, women_in_parliament):
```

```
self.name = name
self.population = population
self.women_in_parliament = women_in_parliament❶
```

- ❶ この属性は、その国の統治機関における女性の議席占有率のリストを年別に記憶する

次のコードは、このデータの要約を表示するメソッドだ。

```
def print_summary(self):
    null_women_in_parliament = len(self.women_in_parliament) - \
        np.count_nonzero(self.women_in_parliament)
    print(f"There are {len(self.women_in_parliament)} data points for \
          Indicator 5.5.1, 'Proportion of seats held by women in national \
          parliaments'.")
    print(f"{null_women_in_parliament} are nulls.")
```

前の例と同じように、このクラスの新しいインスタンスを作成できる。

```
usa = Goal5Data(name="USA",
                 population=336262544,
                 women_in_parliament=[13.33, 14.02, 14.02, ...])
```

`print_summary` メソッドを呼び出すと、次のような結果が得られる。

```
usa.print_summary()
```

```
There are 24 data points for Indicator 5.1.1, 'Proportion of seats held by women in national parliaments'.
0 are nulls.
```

これをメソッドとして書くことで、コードがモジュール化され、組織化され、再利用しやすくなる。また、何をやっているのかが明確なので、このコードを使いたい人の助けになる。

次節では、このクラスを使ってもう 1 つの重要ポイントである「継承」を紹介しよう。

## 4.1.3 OOP の主要概念

OOP では、カプセル化、抽象化、継承、ポリモーフィズムといった用語をよく目にする。本節では、こうした用語の定義と、継承がどのように役立つかの例をいくつか紹介しよう。

### 4.1.3.1 継承

**継承** (inheritance) とは、あるクラスをベースにして別のクラスを作成して、そのクラスを「拡張」することを意味する。すでに書いたクラスと密接に関連する新しいクラスが必要な場合、ちょっとした変更を加えるためだけに、全体を重複させる必要がない。

自分自身のクラスを定義するときには継承を使う必要はないかもしれないが、外部ライブラリのクラスでは継承を使う必要があるかもしれない。本書の後半にある「7.6.3 Pydantic によるデータ検証」と「11.2.2 API への機能の追加」で、データ検証のための継承の例をいくつか紹介する。本節では、継承に遭遇したときに、「それが継承であることを認識する」方法を説明しよう。

継承を使っている場合、クラスの定義は次のようになる。

```
class NewClass(OriginalClass):  
    ...
```

NewClass クラスは、OriginalClass のすべての属性とメソッドを使用できるが、変更したいものは「オーバーライド」できる。OriginalClass のことを**親クラス**、それを継承する NewClass のほうを**子クラス**と呼ぶ。

次のコードは、前節の Goal5Data クラスを継承し、時系列データを扱えるようにした新しいクラス Goal5TimeSeries の例である。

```
from scipy.stats import linregress  
  
class Goal5TimeSeries(Goal5Data):  
    def __init__(self, name, population, women_in_parliament, timestamps):  
        super().__init__(name, population, women_in_parliament)  
        self.timestamps = timestamps
```

\_\_init\_\_ メソッドは、先ほどと少し異なる。super() を使うと、親クラス

の`__init__`メソッドが呼び出され、`name`、`population`、`women_in_parliament`の各属性が初期化される。

次のように、新しい`Goal5TimeSeries`オブジェクトを作成できる。

```
india = Goal5TimeSeries(name="India", population=1417242151,
                        women_in_parliament=[9.02, 9.01, 8.84, ...],
                        timestamps=[2000, 2001, 2002, ...])
```

また、`Goal5Data`クラスからメソッドにアクセスすることもできる。

```
india.print_summary()
```

There are 24 data points for Indicator 5.1.1, 'Proportion of seats held by women in national parliaments'. 0 are nulls.

また、子クラスに関連する新しいメソッドを追加することもできる。例えば、次の`fit_trendline()`メソッドは、回帰線をデータにフィットさせてその傾向を見る。

```
from scipy.stats import linregress

class Goal5TimeSeries(Goal5Data):
    def __init__(self, name, population, women_in_parliament, timestamps):
        super().__init__(name, population, women_in_parliament)
        self.timestamps = timestamps

    def fit_trendline(self):
        result = linregress(self.timestamps, self.women_in_parliament) ❶
        slope = round(result.slope, 3)
        r_squared = round(result.rvalue**2, 3) ❷
        return slope, r_squared
```

- ❶ `scipy` の `linregress` 関数を使って、線形回帰によってデータに直線を当てはめる
- ❷ 決定係数（R 二乗）を計算し、直線の適合度を判断する

新しいメソッドを呼び出すと、トレンドラインの傾きと、データへの直線のフィットの正規化平均二乗誤差が返される。

```
india.fit_trendline()  
  
(np.float64(0.292), np.float64(0.869))
```

自分のクラスについて継承を使うと、前に作成したクラスの機能を拡張できる。これはコードの重複を減らし、コードをモジュール化するのに役立つ。また、外部ライブラリのクラスを継承するのも便利だ。この場合も、すでにある機能については改めてコードを書かずに、新しい機能を付加させることができる。

#### 4.1.3.2 カプセル化

**カプセル化** (encapsulation) とは、クラスの詳細を外部から隠すことを意味する。外から見ることができるのはクラスのインターフェースだけで、内部で起こっているとの詳細は見えない。インターフェースは、メソッドと属性で構成される。Python ではあまり一般的ではないが、他のプログラミング言語では、クラスは外部から変更できないような隠された（プライベートな）メソッドや属性を使って設計されることが多い。

pandas はその好例だ。pandas は、実装の詳細を隠したままデータを操作できるメソッドや属性を提供することで、カプセル化を利用している。DataFrame オブジェクトはデータをカプセル化し、アクセス、フィルタリング、変換のためのさまざまなメソッドを提供する。「3 章 データ構造」で説明したように、pandas の DataFrame は NumPy を使っているが、そのことを知らなくても使うことができる。pandas の DataFrame インタフェースを利用してタスクを実行できるが、より深く掘り下げる必要がある場合は、NumPy のメソッドも使うことができる。



インターフェースは非常に重要である。なぜなら、他のコード（クラス）は、多くの場合、インターフェースに記述された属性やメソッドの存在を前提にして書かれることになるからだ。そのため、インターフェースを変更すると、コードが動作しなくなってしまう危険性がある。クラスの内部構造を変更するのはかまわない。例えば、より効率的にするためにメソッド内の計算方法を変更するといったことだ。だが、インターフェースは最初から使いやすいものにしておき、なるべく変えないようにすべきだ。インターフェースについては「8 章 デザインとリファクタリング」でさらに詳しく説明する。

### 4.1.3.3 抽象化

**抽象化** (abstraction) はカプセル化と密接に結びついた概念で、「適切な詳細レベルでクラスを扱う」ためのものだ。ある計算の詳細をメソッド内部にとどめて外部からは非公開にすることもできるし（選択肢 A）、インターフェースを介してアクセス可能にする（選択肢 B）こともできるが、どちらが好ましいかをしっかり検討すべきなのだ（選択肢 A のほうが抽象化のレベルは高くなり、選択肢 B のほうが抽象化のレベルは低くなる）。なお、抽象化も、どちらかと言うと他のプログラミング言語においてよく登場する概念だ。

### 4.1.3.4 ポリモーフィズム

ポリモーフィズムとは、異なるクラスに同じインターフェースを持たせることで、コードを単純化し、繰り返しを減らす効果を持つ。つまり、2つのクラスが同じ名前のメソッドを持つことができ、そのメソッドは似たような結果を生むが、内部の仕組みは異なる。この2つのクラスは親クラスと子クラスの場合もあれば、無関係の場合もある。

scikit-learn にはポリモーフィズムの素晴らしい例が含まれている。どの分類器も、異なるクラスとして定義されているにもかかわらず、あるデータに対して分類器をトレーニングするための同じメソッド (`fit()`) を持っている。次の例は、あるデータに対して2つの異なる分類器をトレーニングする例だ。

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train)

rf_clf = RandomForestClassifier()
rf_clf.fit(X_train, y_train)
```

`LogisticRegression` と `RandomForestClassifier` は異なるクラスだが、どちらも引数として学習データと学習ラベルを受け取る `.fit()` メソッドを持っている。メソッドの名前を共有することで、コードをあまり変更することなく分類器を簡単に変更できる。

以上、オブジェクト指向プログラミングの主な概念について簡単に説明した。このトピックは膨大なので、より深く学びたい方には Bill Lubanovic 著『入門 Python

3 第2版』(オライリー・ジャパン)をお勧めする。

## 4.2 関数型プログラミング

Pythonは関数型プログラミング(FP: Functional Programming)パラダイムをサポートしているが、純粋なFPスタイルでPythonのコードを書くことは一般的ではない。多くのソフトウェアエンジニアは、FPにはScalaなど他の言語のほうが適しているという意見を持っている。しかし、Pythonで利用できる便利なFP機能は知っておく価値がある。

関数型プログラミングは、その名前が示すように、関数を中心にすえたプログラミング手法である。FPにおける関数は、関数の外部に存在するデータを変更したり、グローバル変数を変更したりしてはならない。専門用語を使って言うなら、関数は「イミュータブル(不变)」であり、「純粋(pure)」であり、「副作用のないもの」でなければならない。関数の戻り値のみが状態の変化を引き起こす。例えば、リストに項目を追加する関数がある場合、その関数は既存のリストを変更するのではなく、リストの新しいコピーを返すことになる。厳密なFPでは、プログラミングは関数の評価のみで構成される。関数はネストされたり(ある関数が別の関数の中で定義されたり)、他の関数の引数として渡されたりする。

FPの利点としては次のようなものが挙げられる。

- 関数は与えられた入力に対して常に同じ出力を返すので、テストが簡単。関数の外側は何も変更されない
- データが変更されないため、並列化が容易である
- モジュール化されたコードを書くことが強制される
- より簡潔で効率的なものになる

Pythonでよく使われる関数型の概念には、ラムダ(lambda)関数や組み込み関数の`map()`や`filter()`がある。さらに、ジェネレーターもこのスタイルで書かれることが多く、リスト内包表記もFPの一種と考えることができる。FPについて知っておく価値のある他のライブラリには`itertools`(<https://oreil.ly/ldikd>)や`more-itertools`(<https://oreil.ly/qOMsA>)がある。次節では、ラムダ関数と`map()`について詳しく見ていく。

## 4.2.1 ラムダ関数と map()

ラムダ関数とは、名前を持たない小さな関数のことで、一度きりの作業を素早く行うために使えるものだ。通常の関数のように名前を付けて定義するわけではないので、無名関数（あるいは匿名関数）と呼ばれる。

ラムダ関数は次のような構文を持つ。

```
lambda <引数>:<式>
```

ラムダ関数は引数をいくつでも取ることができるが、式は1つしか持てない。ラムダ関数は、`map()` や `filter()` などの組み込み関数と一緒に使われることが多い。引数として関数を受け取り、イテレーション可能な要素（リストなど）のすべてにその関数を適用する。

簡単な例を挙げよう。「4.1.2 クラスの定義」の Goal 5 のデータに対して、次の関数を使えば、政府の役職に占める女性の割合のリストを、0以上1以下の割合のリストに変換することができる。

```
usa_govt_percentages = [13.33, 14.02, 14.02, 14.25, ...]
usa_govt_proportions = list(map(lambda x: x / 100, usa_govt_percentages))
```

この1行の中でいろいろなことが起こっている。この場合のラムダ関数は `lambda x: x/100` だ。この関数では、`x` は関数外では使われない一時変数である。`map()` によってリストのすべての要素にラムダ関数を適用している。そして最後に、`list()` が `map()` の結果をまとめて新しいリストを作成している。

結果を出してみよう。

```
print(usa_govt_proportions)
```

```
[0.1333, 0.1402, 0.1402, 0.1425, ...]
```

この関数を適用しても、元のデータは変更されないように注意してほしい。計算結果のデータから新しいリストが作成されるのだ。

## 4.2.2 関数を DataFrame に適用

上で見た `map()` の場合と同じように、DataFrame にいろいろな関数を適用することができる。これは、既存の列をもとに新しい列を作成したい場合に特に便利である。この場合も、別の関数を入力とする関数を使うことができる。pandas では、`apply()` を使う。

次のコードは、DataFrame の列にラムダ関数を適用するもだ。

```
df["USA_processed"] = df["United States of America"].apply(lambda x:
    "Mostly male"
    if x < 50
    else "Mostly female")
```

この例では、`United States of America` という列が、本章を通して使ってきた「政府の役職に就いている女性に関するデータ」である。ラムダ関数は、政府要職に占める女性の割合を引数として取り、その数値が 50% 未満の場合は "Mostly male" (男性が多数)、50% 以上の場合は "Mostly female" (女性が多数) を返す。

また、`df.apply()` とともに、別の場所で定義された名前が付いた関数を使うこともできる。

上のラムダ関数に `binary_labels` という名前を付ける。

```
def binary_labels(women_in_govt):
    if women_in_govt < 50:
        return "Mostly male"
    else:
        return "Mostly female"
```

`apply` 関数の引数に関数名を付けることで、列のすべての行に対してこの関数を呼び出すことができる。

```
df["USA_processed"] = df["United States of America"].apply(binary_labels)
```

このほうがラムダ関数を使うよりも好ましい手法と言えるだろう。なぜならば、将来、関数を再利用できるし、関数のテストやデバッグを別々に行える。また、ラムダ関数よりも複雑な関数を定義して使うこともできる。



pandas の `apply` 関数は、DataFrame のすべての行を繰り返し処理するため、組み込みのベクトル化関数よりも遅い。そのため、「ベストプラクティス」は、まだ実装されていないものにのみ `apply` を使うことである。リストの最大値を取得するような単純な数値演算や、ある文字列を別の文字列に置き換えるような単純な文字列演算は、すでに高速なベクトル化関数として提供されているので、可能な限り組み込み関数を使うべきだ。

## 4.3 どのパラダイムを使うべきか？

正直なところ、小さなスクリプトを書いたり、自分一人で短いプロジェクトに取り組んだりするだけなら、OOP にするか FP にするかこだわる必要はない。きちんと機能するモジュール化されたスクリプトを書けばよいのだ。

しかし、大規模なプロジェクトについては、扱っている問題の種類を考慮して、どちらのパラダイムが適しているかを考えるのがよいだろう。一群の「もの」があって、それに関して何かをする必要があるのならば、OOP のほうが扱いやすい可能性は高いだろう。振る舞いは似ているが属性やデータが異なるインスタンスの集まりと捉えることができる。ここで重要な点は、クラスのインスタンスを数多く作る必要があるかどうかだ。インスタンスが 1 つしかないのであれば、新しいクラスを書く価値はない。

固定されたデータに対して新しいことをしたいようならば、FP が良い選択かもしれない。また、大量のデータを持っていて、それに対する操作を並列化したい場合にも、FP を検討する価値がある。

いずれにしろ、これは「正解」「不正解」が決まる性質のものではない。一人で作業している場合は個人的な好みに従えばよいし、チーム内で主に使われているものを使えばコードの「標準化」が前進することになる。コードを読んだときに、何らかのパラダイムが使われていることを意識できれば、コードを読むのが楽になるだろうし、同じパラダイムを使ってコードの追加やメンテナンスをすることができる。

## 4.4 まとめ

OOP と FP は、多くのプロジェクトで使われているプログラミングパラダイムである。OOP はカスタマイズされたデータ構造であるオブジェクトを中心とした考え方であり、FP は（ベースのデータを変更しない）関数を中心とした考え方だ。

OOP では、クラスは新しいオブジェクトを定義し、そのオブジェクトは属性やメソッドを持つことができる。独自のクラスを定義することで、関連するメソッドやデータをまとめておくことができる。これは、似たようなオブジェクトのインスタンスが数多く存在する場合に最適なアプローチだ。継承を使ってコードの繰り返しを避け、ポリモーフィズムを使ってインターフェースの標準化が可能になる。

FP では、理想的にはすべてが関数の中にある。変化しないデータを持っていて、そのデータに対して多くのことを行いたい場合や、データに対して行っていることを並列化したい場合に特に便利な手法だ。ラムダ関数は、Python で最もよく使われる FP の応用例だ。

どちらのパラダイムを選択するかは、取り組んでいる問題によって異なるが、両方を意識しておくと役に立つことは間違いないだろう。



# 5章 エラー、ログ、デバッグ

本章では、コードをより堅牢にするテクニックを紹介する。堅牢性は、1章で説明した良いコードを書くための原則の1つだ。まず、エラー処理について説明し、何か問題が発生してもコードに予測可能な動作をさせる。続いて、ログを用いてコードに関する情報を表示・保存する方法を紹介する。これは、(他人が)コードの動作を理解する際の助けとなるほか、予期しないエラーが発生したときにその原因究明にも役立つ。最後に、デバッグについて議論しよう。デバッグとは問題の原因を突き止める方法だ。効率的なデバッグのための戦略やツールについて説明する。

## 5.1 Pythonのエラー

エラーとは、プログラムがやるべきタスクを完了する前に、コードが予期せずに停止してしまうことだ。これが起こると、そのコードに依存している別のシステムも止まってしまうかもしれない。これが起こってほしいこともあるが、コードを実行し続けるために別のこと（「エラー処理」）を行いたい場合もある。コードは、起こることが予想される事態に対して予測可能な動作をすべきであり、これによって堅牢性を確保できる。

本節では、Pythonのエラーメッセージの読み方や処理方法、自分独自のエラーの生成方法について説明する。

### 5.1.1 Pythonのエラーメッセージ

Pythonのエラーメッセージは不可解に見えるかもしれないが、有用な情報が詰まっている。エラーには構文エラーと例外エラーの2種類がある。構文エラーは、例えば括弧を閉じなかったり、関数の定義で `def` を忘れてしまったりといった、

「Python 言語において正しいコード」を書いたときに発生する。構文エラーがあると Python インタプリタがうまく解析できず、停止してエラーを返す。

他のすべてのエラーは「例外エラー」になる。例えば、関数への入力が見つからないとか、辞書には存在しないキーの値を検索しようとしたといった場合である。

次の例では、Python のエラーメッセージの解釈方法を示すために、わざとエラーを発生させる。これは 4 章で見たトレンドラインのコードを別の関数にしたものだが、関数の定義に `data` 引数が欠けている。

```
from scipy.stats import linregress

def fit_trendline(year_timestamps):
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared
```

この関数の定義に沿った形で、実行してみよう。

```
timestamps=[2000, 2001, 2002]
fit_trendline(timestamps)
```

しかし、これは次のようなエラーを引き起こす。

```
-----
NameError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 fit_trendline(timestamps)

Cell In[1], line 4, in fit_trendline(year_timestamps)
      3     def fit_trendline(year_timestamps):
----> 4         result = linregress(year_timestamps, data) ❶
      5         slope = round(result.slope, 3)
      6         r_squared = round(result.rvalue**2, 3)

NameError: name 'data' is not defined ❷
```

❶ エラーの原因となったコードの行

❷ エラーの種類と説明（場合によっては、修正方法の提案もある）

Python のエラーはトレースバックとして出力される。トレースバックには、エ

ラーが発生したときに呼び出された関数がすべて表示される。Python の内部や使用しているライブラリの詳細が含まれることもあるので、庄倒されてしまうかもしれないが、エラーの原因を突き止めるのに役立つ情報がたくさん含まれている。

Python のエラーメッセージに対処するお勧めの方法は、メッセージの最後から読み始めることがある。最後の行は、エラーの種類（上の例では `NameError`）と、（開発者を助けることができると処理系が考えた場合）エラーを修正する方法に関するいくつかの情報を教えてくれる。その上のほうには、エラーの原因となった行を指す矢印（`----->`）の付いたコードが表示される。これは問題を探す手がかりとはなるが、この例では問題は実際には前の行にあり、関数の定義で `data` 引数が欠けている。

また、インポートしたモジュールでエラーが発生することもある。例えば、インポートした関数に誤った型の入力を与えた場合だ。エラーが発生したコードの行が強調表示されるだけでなく、エラーメッセージにはインポートされた関数の行と、その関数が呼び出した他の関数の行も含まれるため、このようなエラーはより混乱しやすくなるが、エラーメッセージが非常に長い場合でも、同じ戦略を適用すればよい。メッセージの最後から読み始めて、上方向に向かって問題を解決しようと試みるのだ。



Python のエラーメッセージは 3.10 (<https://oreil.ly/Qpddl>) から 3.11 (<https://oreil.ly/J7nxs>) へのバージョンアップで大幅に改善された。新バージョンでは、より読みやすく、有益なメッセージが表示されるようになり、実際に誤りのある行を指すことが多くなった。このため（可能ならば）Python を最新バージョンにアップグレードするとよい。

## 5.1.2 エラー処理

エラーが発生してもコードを停止させたくない場合も多い。他の関数が、自分が書いた関数に依存している場合には特にだ。このような場合は、予測可能な動作をするように、エラーをうまく処理してやる必要がある。

このために使えるのが、`try` と `except` というキーワードだ。次のように、2つのコードブロックを使ってエラーを処理する。

```
try:
    <何らかの処理をするコード> # ←tryブロック
except KeyError:
    <このタイプのエラーが起きてしまったときにだけ実行したいコード> # ←exceptブロック
```

`try` ブロックには、実行したいがエラーに遭遇するかもしれないコードを記述する。`except` ブロックには、ある種のエラー（この場合は `KeyError`）が発生した場合にどうするかを記述する。例えば、辞書のキーの値を調べるコードがあるとする。そのキーが辞書に存在しない場合、エラーでコードを停止する代わりにデフォルト値を返すことにするといった具合だ。

特定のタイプのエラー（例外）発生時に実行したいコードを記述した複数の `except` ブロックを持つこともできるし、1つの `except` ブロックで複数のタイプのエラーを処理することもできる。後者の場合、次のような構文になる。

```
except (KeyError, ValueError):
```

また、`except` ブロックの後に、`else` ブロックを置くこともできる。ここには、エラーが発生しなかった場合に `try` ブロックの後に実行されるコードを書く。

Python のエラー処理のもう 1 つの特徴として、エラーが発生したかどうかにかかわらず実行される `finally` ブロックがあり、これは `try`、`except`、そしてオプションの `else` ブロックの後になければならない。`finally` ブロックは、開いているファイルを閉じるなど、リソースを解放するために使われることが多い。

図 5-1 にこの流れを示す。エラー処理の構文についての詳細は、Python のドキュメント (<https://oreil.ly/GkThO>) を参照されたい。

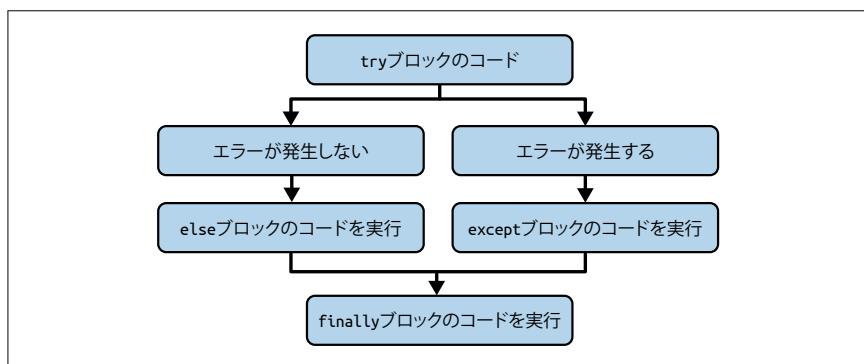


図 5-1 `try`、`except`、`else`、`finally` ブロックの実行順序

「5.1.1 Python のエラーメッセージ」で登場した `trendline` 関数を使って、今回は正しい引数でエラー処理を説明しよう。

```
from scipy.stats import linregress

def fit_trendline(year_timestamps, data):
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared
```

例外を起こさせる（「スロー」させる）ために、不正な入力で実行してみる。

```
women_in_parliament=[9.02, 9.01, 8.84, 8.84, 8.84]
timestamps= ["2000", "2001", "2002", "2003", "2004", "2005"]
fit_trendline(timestamps, women_in_parliament)
```

両方のリストが浮動小数点数か整数で構成されていなければならぬため、`ValueError` が起る。このエラーは、`try`、`except`、`else` のブロックを使って、エラーを出さずにコードが実行されるように処理できる。

```
def fit_trendline(year_timestamps, data):
    try:
        result = linregress(year_timestamps, data)
    except TypeError:
        print("*ERROR*: Both lists must contain only float or integers")
    else:
        slope = round(result.slope, 3)
        r_squared = round(result.rvalue**2, 3)
        return slope, r_squared
```

エラーが発生した場合、コードはメッセージ「\*ERROR\*: Both lists must contain only floats or integers」を表示する。エラーが発生しなかった場合、`else` ブロックが実行される。

エラーメッセージが有益で読みやすいことは重要だ。良いエラーメッセージは、何が間違っていて、コードのどこでエラーが発生したのか、明確なガイダンスを提供することで、作業時間を大幅に節約してくれる。エラーメッセージには、何が問題だったのか、そしてそれを解決するために何をすべきかを記述するのが理想的だ。

また、`except` ブロックでデフォルト値を返すこともできる。これは、関数が成功した場合に返す値と同じ形式にしておくとよいだろう。こうすることで、関数のイン

タフェースの一貫性を保つことができる。この関数が特定の型の答えを返すと仮定しているコードがある場合でも、（エラーが起こったかどうかにかかわらず）停止せずにコードが実行されることになるのだ。

次のコードは、`except` ブロックがデフォルト値を返す例である。

```
def fit_trendline(year_timestamps, data):
    try:
        result = linregress(year_timestamps, data)
    except TypeError:
        print("*ERROR*: Both lists must contain only float or integers")
        return 0.0, 0.0
    else:
        slope = round(result.slope, 3)
        r_squared = round(result.rvalue**2, 3)
        return slope, r_squared
```

この関数は、エラーが発生したか否かにかかわらず、2つの浮動小数点数のタプルを返すようになった。

```
fit_trendline(timestamps, women_in_parliament)
```

```
*ERROR*: Both lists must contain only float or integers
(0.0, 0.0)
```

筆者が本書の構想を練っていたとき、データサイエンスの仕事に就く前に書いたコードを見てみたところ、次のようなコードを書いていた。このコードはエラーをどのように処理してはいけないかを示している。

```
except:
    return
```

このような処理はだめだ。遭遇したエラーをすべて黙殺してしまい、関数が動作していないことがわからなくなってしまう。間違った入力を受け取ったときでも、関数が予測どおりに動作するほうがずっとよい。

### 5.1.3 エラーの生成

状況によっては、予期せぬことが起こったときにエラーを発生させたい場合もあるだろう。そうした関数に依存しているコードも停止させたい場合だ。このような場

合、Python の `raise` キーワードを使えば、通常はエラーが発生しないような場合でもエラーを発生させることができ、そのエラーの内容もカスタマイズできる。

次の例では、`fit_trendline` 関数の入力が空リストであるかどうかをチェックし、空リストであればエラーを発生させる。

```
from scipy.stats import linregress

def fit_trendline(year_timestamps, data):
    if not year_timestamps or data:
        raise ValueError("Timestamps and data cannot be empty lists")
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared
```

次の形式の構文を使って、カスタムエラーメッセージを指定できる。

```
raise {error}('メッセージ')
```

では、空のリストを引数に指定して実行してみよう。

```
fit_trendline([], [18.36, 18.36, 17.91])

-----
ValueError                                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 fit_trendline([], [18.36, 18.36, 17.91])

Cell In[1], line 5, in fit_trendline(year_timestamps, data)
      3 def fit_trendline(year_timestamps, data):
      4     if not year_timestamps or data:
----> 5         raise ValueError("Timestamps and data cannot be empty lists")
      6     result = linregress(year_timestamps, data)
      7     slope = round(result.slope, 3)

ValueError: Timestamps and data cannot be empty lists
```

エラーメッセージは、それを読み取る人にとって有益であるべきであり、そうすることでコードが読みやすくもなる。「An error occurred (エラーが発生しました)」は有用なエラーメッセージではない。

2章を執筆中に遭遇した（システム側が出してくれた）有用なエラーメッセージの

例を挙げよう。

```
"UsageError: Cell magic `%%lprun` not found (But line magic `%lprun` exists, did you mean that instead?)."  
セルマジック`%%lprun`は見つからない(ただし、ラインマジック`%lprun`が存在する。この意味か?)
```

このメッセージが何が間違っているのか、エラーを修正するにはどうすればよいのかを教えてくれている。

Pythonで必要なタイプのエラーが定義されていない場合は、カスタムエラーの定義もできる。エラーの名前をカスタマイズすれば、ユーザーにエラーの発生元を知らせることができる。新しいエラーの定義は次のような形になる。

```
class NewException(Exception):
```

これは「4.1.3.1 繙承」で見た継承の例で、`NewException` クラスは `Exception` クラスを継承することになる。カスタムエラーの定義については、Marcin Kozak の記事「Should we use custom exceptions in Python?」(<https://oreil.ly/P0bj9>) の記事が参考になる。

カスタムエラーを利用すれば、コードが読みやすく、そして保守しやすくなる。各種のエラーがきちんと定義されていれば、エラーを適切に処理でき、異なる箇所で発生しているエラーの混在を防止できる。このようにエラー処理を細かくすることで、意図しない結果を防ぐのに役立つだけでなく、デバッグやトラブルシューティングの効率も上げられる。

## 5.2 ロギング

ロギングとは「ログ」を取ること、すなわちコードの実行中に、コードが何をしたかを（ファイルなどに）記録していくことだ。ログによって、コードに何が起こっているかがわかり、ほかの開発者がコードを理解する助けとなる。より読みやすく、堅牢なコードを書く助けともなるはずだ。

ロギングによって、単にメッセージを出力させるのとは違い、検索可能な別ファイルに情報を保存できる。これは、長時間実行されるプロセスや、本番システムにデプロイされたコードに特に有効だ。このような状況では、何が起こっているのか、何が間違っているのかを見つけるためにシステムを止めたくはないはずだ。

ここではロギングの基本を紹介するが、詳細は Python ドキュメントの「Logging

HOWTO」(<https://oreil.ly/8s5Ey>) を参照されたい。

### 5.2.1 何を記録するか

Python の `logging` モジュールを使えば、どんなメッセージでも記録できる。どのようなログを取ると役に立つだろうか。ユースケースとプロジェクトに依存するが、例えば次のようなものが挙げられる。

- 長時間実行中のタスクが開始または終了したことを示すメッセージ
- 本番システムでどのような問題が起こったのかを知るためのエラーメッセージ
- 特定の関数をどの関数が呼び出したか
- 特定の関数の入力と出力
- データが保存されたファイルのパス

ログは、本番システムや長時間実行するタスクのデバッグに大いに役立つ。これについて、「5.3 デバッグ」で詳しく説明する。

ログは、コードで「やるべきことをやった」という証拠にもなる。これがプロジェクトにとって有用であれば、タスクが起こったという事実とタスクの結果を示すログメッセージを含めるようにする。例えば、機械学習モデルの学習が終わったときに、その精度をログに記録する。

戦略としては、ログを少なくするのではなく、多くするほうがお勧めだ。メッセージが標準化されていたり、特定のキーワードを含んでいれば、ログファイルを検索して簡単に見つけられる。ログメッセージを具体的に書くことも重要だ。「`started saving file` (ファイルの保存を開始)」といったメッセージは役に立たない。ファイルの内容は何か、ファイル名は何なのか、具体的に記録するのだ。この種の処理には `f` 文字列（フォーマット済み文字列リテラル）が便利だ。

```
f"Saving {variable_name} data to {file_name}"
```

次節では、Python の `logging` モジュールを使ってメッセージをログに記録する方法を説明する。

### 5.2.2 ロギングの設定

Python 標準ライブラリの `logging` モジュールは巨大だが、通常使うのはその中のごく一部の関数だけだ。このライブラリは、ログに記録されたメッセージを別々の

ファイルに保存してくれるので、保存方法については心配する必要はない。しかし、いくつかの設定が必要だ。

まずはログを記録する**深刻度** (severity。「重要度」とも呼ばれる) の設定だ。Python の `logging` モジュールには、`DEBUG` (深刻度が最も低い) から `CRITICAL` (最も高い) まで、5つの異なる深刻度レベルがある。これにより、使用するケースに応じて、ログをフィルタリングして、発見したいメッセージを簡単に見つけたり、あるレベル以上のメッセージだけをログに記録したりできる。例えば、コードを開発しているときには `DEBUG` レベルのメッセージをログに記録し、コードが本番稼働しているときには、エラーや、より深刻なメッセージだけをログに記録するといった具合だ。

表5-1 にロギングの5つのレベルを示す (Python のドキュメントから引用した)。

表5-1 Python のログレベル (出典: <https://oreil.ly/6iVoR>)

レベル	説明
<code>DEBUG</code>	デバッグ。詳細な情報であり、通常は問題を診断するときにのみ役立つ情報。
<code>INFO</code>	物事が期待どおりに動いていることを確認する
<code>WARNING</code>	警告。予期せぬことが起こった、または近い将来に何らかの問題が起こる可能性があることを示す (例えば「ディスク領域が少ない」など)。ソフトウェアは期待どおりに動いている
<code>ERROR</code>	エラー。より深刻な問題により、ソフトウェアが何らかの機能を実行できなくなっている
<code>CRITICAL</code>	深刻なエラー。プログラム自体が実行を継続できない可能性があることを示す

Python の `logging` モジュールのデフォルトのロギングレベルは `WARNING` である。つまり、ロギングレベルを変更せずに `INFO` レベルのメッセージを書いた場合、そのメッセージは保存されない。

何らかのログを記録したいときには、まずレベルをセットする。

```
import logging  
  
logging.basicConfig(level=logging.DEBUG)
```

この場合、レベルが `DEBUG` にセットされているので、`DEBUG` から `CRITICAL` までのすべてのレベルがログに記録される。この行は、Python スクリプトの最初に、ログを開始する前に記述する必要がある。

もう 1 つのデフォルト設定でよく指定されるものに、ログの保存場所がある。デフォルトではコンソールにログを出力するが、ファイルに保存しておくとコードの実行が終わった後に参照できて便利だ。

ログを別のファイルに保存したい場合は、次の設定を追加する。

```
logging.basicConfig(filename='chapter_5_logs.log', level=logging.DEBUG)
```

ファイルに書き込むときのデフォルトの設定は、コードが実行されるたびにログをファイルに追加する。ファイルを上書きしたい場合は、`filemode='w'` を指定する。

```
logging.basicConfig(filename='chapter_5_logs.log', filemode='w',
                    level=logging.DEBUG)
```



メッセージは単にファイルに保存するだけでなく、「ログビューア」を使うことで Python の `logging` モジュールからの出力を受け取り、別の場所に保存や表示ができる。Amazon Web Services (AWS) などのクラウドプロバイダーを使った本番システムには、このためのビルトインソリューションがあり、`logging` モジュールの `handlers` (<https://oreil.ly/apGQb>) を使うことが多い。スタンダードアローンのツールである Kibana (<https://oreil.ly/Cgkn7>) や Prometheus (<https://prometheus.io>) も、コードによって生成されたログから必要な情報を抽出してくれる。

### 5.2.3 ログの取り方

スクリプトの最初でロギングの設定をしたら、次のステップは何をログに記録するかを決めることがある。`logging.info` 関数を使い、`fit_trendline` 関数に INFO レベルのログ行をいくつか追加してみよう。`logging.info` は引数としてログに記録したいメッセージを受け取る。

```
from scipy.stats import linregress

def fit_trendline(year_timestamps, data):
    logging.info("Running fit_trendline function")
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
```

```
r_squared = round(result.rvalue**2, 3)
logging.info(f"Completed analysis. Slope of the trendline is {slope}.")
return slope, r_squared
```

この関数は以前と同じように実行できる。

```
fit_trendline(timestamps, data)
```

ログファイルに次のように書き込まれる。

```
INFO:root:Running fit_trendline function
INFO:root:Completed analysis. Slope of the trendline is -0.053.
```

ログにタイムスタンプを追加しておくと、特定の行をいつ実行したかがわかり、後で検索しやすくなる。これは、最初の設定で `format` 引数を使って行う。

```
logging.basicConfig(filename="chapter_5_logs.log",
                    level=logging.DEBUG,
                    format='%(asctime)s %(message)s') ❶
```

- ❶ `format` 引数には多くのオプションがあり、Python のドキュメント (<https://oreil.ly/wUJdq>) に説明がある

これにより、次のようなメッセージが表示される。

```
2025-05-13 21:18:25,766 Running fit_trendline function
2025-05-13 21:18:25,766 Completed analysis. Slope of the trendline is -0.053.
```

エラーメッセージをログに記録するのは良いアイデアだ。次に示すように `logging.exception` を使って記録できる。これで、トレースバック全体がログに記録されるので、これを見ながらデバッグできる。

```
def fit_trendline(year_timestamps, data):
    logging.info("Running fit_trendline function")
    try:
        result = linregress(year_timestamps, data)
    except TypeError as e:
        logging.error("Both lists must contain floats or integers.") ❶
```

```

    logging.exception(e) ❷
else:
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    logging.info(f"Completed analysis. Slope of the trendline is {slope}.")
    return slope, r_squared

```

- ❶ エラーメッセージを ERROR レベルでログに記録する
- ❷ トレースバック全体をログに記録する



Jupyter ノートブックで `logging` モジュールを使えば、結果をログファイルに保存できるので、ノートブックを再実行する必要がない。これは、結果をコピーして Markdown セルに貼り付けるよりも良い方法かもしれないが、追跡するファイルがもう 1 つ増えることになる。なお、「9.3 機械学習の実験の文書化」で説明するような、トレース機能を使うこともできる。

## 5.3 デバッグ

デバッグとは、コードのバグを発見し、それを取り除くことを意味する。バグとは、コードが予期しないエラーを出したり（「スロー」したり）、予期しない結果を返したりすることだ。デバッグはコードを書く上で必須のスキルであり、大規模なコードベースを扱うときには、どのような状況にも対応できるように、さまざまなテクニックを身につけておきたい。本節では、バグを発見するための戦略をいくつか説明し、次にデバッグのためのツールをいくつか説明する。

### 5.3.1 デバッグの戦略

バグっぽいコードについて情報を得るために筆者のお気に入りの方法は、そういうバグっぽいコードを実行して実験してみることだ（`print` 文を追加する！ 小さな変更を加える！）

—— Julia Evans

経験を積めば、難しいバグを追跡するための自分なりのお気に入りの戦略を思いつくだろう。しかし、Julia Evans の引用で推奨されているように、データサイエンティストが最も得意とすること、つまり実験を行うことから始めるのがよいだろう。コードにちょっとした変更を加え、それが結果にどう影響するかを見るのだ。

「7章 テスト」で詳しく説明するが、テストはデバッグにも役に立つ。テストは、自分が行った変更が、期待した結果を生み出す（テストがパスする）か、そうでない（テストが失敗する）のかを知らせてくれる。

もう1つのお勧めは、関数の入力であれ、特定のデータベースクエリの結果であれ、エラーの原因となったもののコピーを保存しておくことである。このためにはロギングが非常に役立つ。何が起こったかを記録しておくことができるのだ。

また、エラーメッセージの原因となっているミスを犯したのは、自分が初めてということは少ないだろう。インターネット上には、問題を解決するための情報が豊富にある。Googleでエラーメッセージを検索してみるのは、一番最初にやるべきことだ。

自分のコードについて他人に話すことも、バグを発見するための素晴らしい戦略だ。同僚に説明したり、カモのぬいぐるみに大声で話したりすることで、自分のコードを読んでいるときとは少し違った方法で、自分が何をしてきたかを考えさせられ、解決策が見つかるかもしれない。他人であっても自分の経験を共有し、あなたを助けてくれるはずだ。

誰のコードにもバグはある。すべての職場の文化がこれを奨励しているわけではないにせよ、「自分がわからないことがある」とか「間違ってしまった」と言っても許されるはずだ。筆者自身もたくさんのバグがあるコードを書いてきた。その中には、見つけるのが難しい微妙なバグもあれば、単純なバグもあった。本書を書いている最中にも、初期のオンライン版のコードにミスがあったことをメールで知らせてくれた人がいたのだ。

もう1つの優れたデバッグ戦略は、コーディングから離れることだ。お茶を入れたり、散歩に出かけたり、ほかのことをするのだ。まったく違うことをしているときや、新鮮な目でコードに戻ったときに、解決策が思い浮かぶことは少なくない。休憩は、バグの原因を思いつくのにも役立つ。

ときにはコードを詳細に掘り下げる必要がある。このようなとき、次節で紹介するツールが大いに役立つだろう。

### 5.3.2 デバッグ用ツール

デバッグのためのツールは、コードで何が起こっているかを正確に追跡するのに役立つ。変数の値の変化に関する情報を得られれば、そのコードが本当に想定どおりに動いているかどうかのチェックにとても役に立つ。これを行う最も簡単な方法は、`print` 文をコードに追加することだが、デバッグ用のツールを使えば、より多くの情報を得られる。コードを一時停止して特定の変数の値を表示し、その値を追跡しなが

らコードを一步ずつ進めていくことができるのだ。

2章のサンプルコードを使って説明しよう。この例では、`weighted_mean` 関数にバグがあり、正しい結果を返さない。

```
def weighted_mean(num_list, weights):
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights[0]) ❶
    return (running_total/len(num_list))
```

❶ `weights[0]` ではなく、`weights[i]` とするべきである

例えば、次のように `print` 文を挿入することで、バグを発見するのに役立つかもしれない。

```
def weighted_mean(num_list, weights):
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights[0])
        print(f"The running total at step {i} is {running_total}")
    return (running_total/sum(weights))
```

この関数を実行すると、次のような出力が得られる。

```
The running total at step 0 is 1
The running total at step 1 is 7
The running total at step 2 is 15
2.5
```

これは `running_total` (合計 [の途中経過]) が正しく計算されていないことを示している。ステップ1では19、ステップ2では35であるべきだ。

しかし、`print` 文を挿入してコードを見にくくするのは「ベストプラクティス」とは言えない。コードの結果の把握を困難にする。本番システムで実行されている場合なら、出力が何かわかりにくくなってしまうかもしれない。

もう1つの選択肢は、「5.2 ロギング」で説明したように、途中でログに記録することだ。次はその例である。

```

import logging

logging.basicConfig(filename="ch05_logs.log",
                    level=logging.DEBUG,
                    filemode='w',
                    format='%(asctime)s %(message)s')

def weighted_mean(num_list, weights):
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights[i])
        logging.debug(f"The running total at step {i} is {running_total}")
    return (running_total/len(num_list))

```

これにより、ログファイルに次のような出力がなされる。

```

2025-05-13 23:37:19,534 The running total at step 0 is 1
2025-05-13 23:37:19,537 The running total at step 1 is 7
2025-05-13 23:37:19,537 The running total at step 2 is 15

```

この出力は別のファイルに保存されるので、本番システムを運用中でも参照できる。  
`print` やロギングを用いる方法は、このような小規模なプログラミングでは問題ないかもしれないが、関数内に数十の変数がある場合はどうだろうか。ロギングしたい変数のいくつかが大きな辞書だったらどうだろうか。「手に負えなくなってしまう」のはおわかりだろう。このような状況を助けてくれるさまざまなツールが**デバッガ**だ。

デバッガーでは**ブレークポイント**を使う。ブレークポイントとは、コードの実行をブレークし、変数（または保存されたファイルやデータベース）の状態を確認するために選択する箇所である。コードを1ステップ実行して変数がどのように変化するかを確認し、すべてが期待どおりかどうか見守ることができる。

多くの統合開発環境にはデバッガーが含まれている。ここでは VS Code (Visual Studio Code) のデバッガーを使って説明するが、他の IDE（統合開発環境）でもよく似ている。

次のスクリプトをデバッグする方法を紹介しよう。

---

ch05\_debugger.py

---

```

1: def weighted_mean(num_list, weights):
2:     running_total = 0

```

```

3:     for i in range(len(num_list)):
4:         running_total += (num_list[i] * weights[0]) #❶
5:     return (running_total/len(num_list))
6:
7: weighted_mean([1, 6, 8], [1, 3, 2])

```

### ❶ この行にブレークポイントを追加した

デバッガーを使えば、関数のどの時点でも、各変数がどのような状態になっているか1ステップずつ確認していくことができる。ループの各ステップでの変化がハイライトされる。

図5-2はforループの最初の繰り返しを示している。ch05\_debugger.pyをVS Codeで開いて、forループ本体の1行目にブレークポイントを設定し、実行を開始したところだ。黄色のバックグラウンドでiの値は0で、running\_totalの値も0であることを示している。

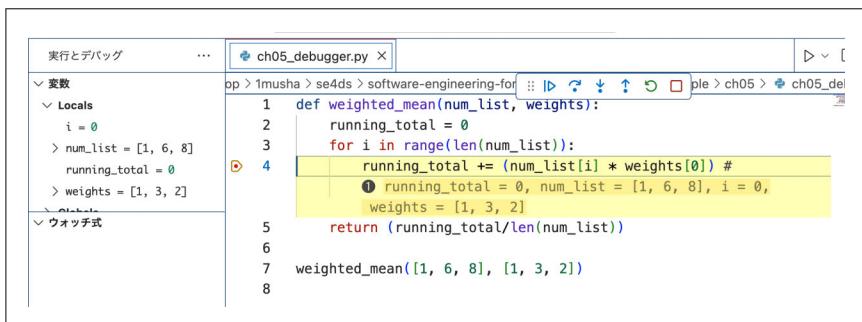


図5-2 デバッガーで変数の値を表示。最初の繰り返し

2回目の繰り返しになるとiの値は1で、running\_totalの値も1になる。ここまででは想定どおりだ。

3回目の繰り返しに入ったところを図5-3に示す。この図を見てわかるように、この時点ではrunning\_totalが正しくない。本来は $(1 * 1) + (3 * 6) = 19$ であるべきだが、7になっている。

```
software-engineering-for-data-scientists > src > ch05_debugger.py > weighted_mean
1 def weighted_mean(num_list, weights):
2     running_total = 0
3     for i in range(len(num_list)):
4         running_total += (num_list[i] * weights[0]) # => Breakpoint
5         ① running_total = 7, num_list = [1, 6, 8], i = 2,
6         weights = [1, 3, 2]
7     return (running_total/len(num_list))
8
weighted_mean([1, 6, 8], [1, 3, 2])
```

図5-3 デバッガーで変数を表示。3回目の繰り返し

バグは関数の4行目にある。次のように修正しなければならない。

```
running_total += (num_list[i] * weights[i])
```

デバッガーの使い方は、使用しているIDEによってやや異なるが、どれも似たような機能を持っている。詳細はVS Codeのドキュメント (<https://oreil.ly/j9K2a>) やPyCharmのドキュメント (<https://oreil.ly/xQkOW>)などを参照されたい。

図5-4に示すように、IDE(VS Code)でJupyterノートブックを実行している場合にも、Pythonファイルを開いているときと同じようにデバッグ機能を使える。スタンドアローンのスクリプトと同じように、ブレークポイントを追加できる。

```
ch05_debugger.ipynb > def weighted_mean(num_list, weights):
① running_total = 7
② running_total += (num_list[i] * weights[0]) # => Breakpoint
③ running_total = 7, num_list = [1, 6, 8], i = 2,
④ weights = [1, 3, 2]
return (running_total/len(num_list))
```

図5-4 IDEでノートブックをデバッグ

また、IDE を使っていなくてもデバッガーを使うことはできる。pdb (<https://oreil.ly/78KQu>) は Python 標準ライブラリに含まれているコマンドラインデバッガーだ。

これを使うには、`breakpoint()` メソッドをコードに追加する。

```
def weighted_mean(num_list, weights):
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights[0])
        breakpoint()
    return (running_total/len(num_list))

weighted_mean([1, 6, 8], [1, 3, 2])
```

あとは通常どおりスクリプトを実行するだけだ。

```
$ python3 ch05_debugger_pdb.py
```

コードはブレークポイントに達するまで実行され、その後 pdb デバッガーを開き、次のようなメッセージが表示される。

```
> /Users/.../example/ch05_debugger_pdb.py(4)weighted_mean()
-> for i in range(len(num_list)):
> /Users/.../ch05_debug.py(3)weighted_mean()
-> for i in range(len(num_list)):
(Pdb)
```

「p 〈変数名〉」で任意の変数の値を表示でき、n で次の行に進める。

```
(Pdb) p running_total
1
(Pdb) p weights
[1, 3, 2]
(Pdb) n
> /Users/.../example/ch05/ch05_debugger_pdb.py(5)weighted_mean()
-> running_total += (num_list[i] * weights[0])
(Pdb) n
> /Users/.../example/ch05/ch05_debugger_pdb.py(6)weighted_mean()
-> breakpoint()
(Pdb) n
> /Users/.../example/ch05/ch05_debugger_pdb.py(4)weighted_mean()
-> for i in range(len(num_list)):
(Pdb) p running_total
7
```

デバッグが終わったら、`q` でデバッグを停止し、スクリプトを終了する。また、`pdb` を使い終わったら、コード中の `breakpoint()` の呼び出しをすべて削除する必要がある。そうしないと、スクリプトを実行するたびに `pdb` が起動してしまう。「12章 自動化とデプロイ」で自動的に削除する方法を紹介する。

表5-2 に `pdb` コマンドのショートカットの一覧を示す。

表5-2 `pdb` のコマンド。Nina Zakharenko の記事 (<https://oreil.ly/z1sx2>) をベースに作成

コマンドのショートカット	機能
<code>p expr</code>	式 ( <code>expr</code> ) の値を出力 ( <code>print</code> )
<code>l</code>	現在の行の周辺のコードを出力 ( <code>list</code> )
<code>n</code>	次の行に進む ( <code>next</code> )
<code>c</code>	次のブレークポイントに進む ( <code>continue</code> )
<code>r</code>	現在の関数が戻るまで続ける ( <code>return</code> )
<code>q</code>	デバッグを終了 ( <code>quit</code> )

`pdb` は IDE のデバッガーと同じ機能を提供するが、使い勝手は良くはない。しかし、いろいろなツールを知っておけば、自分の状況に合ったものを使える。IDE でコードを開発している場合は付属しているデバッガーを使うのが簡単だが、コマンドラインで実行中なら `pdb` のほうが手軽だ。

## 5.4 まとめ

本章では、コードをより堅牢なものにするための方法を説明した。また、コードの問題を発見し解決するためのスキルについても紹介した。

エラー処理を適切に行うことで、予期せぬ入力に対するコードの堅牢性を保つことができる。必要な場合は、独自のカスタムエラーを発生させるべきである。

ロギングはコードが行ったことの記録を保存するもので、本番コードの場合や、実行に時間がかかるスクリプトには特に便利だ。Python 組み込みの `logging` モジュールには、必要な機能がすべて含まれている。

コード中のバグを特定して修正するための戦略やツールが数多くあるが、どれも役に立つ。デバッグ用のツールは、単純な `print` 文、ロギング、デバッガーなど多岐にわたる。7章では、テストによってデバッグを行う方法を説明する。

# 6章

## コードのフォーマット、 linting、型チェック

本章では、コード記述の面倒な部分を助けてくれるツールについて説明する。コードの整形（あるいは「フォーマッティング」）やlinting、型チェックを行ってくれるツールは、コードを解析して間違いや改善点を指摘したり直したりしてくれるものだ。整形ツールはコードの見た目（形式）を、lintingと型チェックのツールはコードが正しく機能することを確認してくれる。

なぜコードの形式が注目されるのか不思議に思うかもしれない。なぜコードの見た目が重要なのか。なぜ開発者は貴重な時間を費やして、+の前後のスペースの数の基準を設定するのか。それは、一貫性のある標準化された形式を持つコードのほうが読みやすいからだ。そして1章で説明したように、コードが読みやすければ、再利用される可能性が高くなる。整形ツールのおかげで、手作業でコードの見た目を改善する手間が省ける。

lintingと型チェックは、コードを堅牢にするのに役立つ。Pythonのコードを実行するとき、構文エラーはスクリプトのどこにあってもすぐにコードをクラッシュさせるが、ある種の間違い（例えば、変数名のスペルミス）では、その行になるまでエラーは起こらない。スクリプトの実行に時間がかかるとイライラさせられるが、「リンター」や関連のツールを使えば、コードを実行する前に、こうしたミスを発見できる。

本章での筆者の重要なメッセージは「手作業でコードをチェックするよりもツールを使うべきだ」ということだ。コードの形式チェックは退屈な作業だ。そんなことに時間を費やしたくはないだろう。ツールをセットアップしてIDEをうまく活用し、「標準」が自動的に適用されるようにする方法を紹介しよう。



本章で説明するツールの多くは、Jupyter ノートブックでは使えない。セルに分割されたコードには適しておらず、長いスクリプトで使うように設計されている。そこで本章では、スタンドアローンのスクリプトでツールを実行する方法を示す。

## 6.1 コードの形式とスタイルガイド

「スタイルガイド」に従ってコードを形式化することは、良いコードを書くための重要な要素である。スタイルガイドは、例えば Google のスタイルガイド (<https://oreil.ly/8yGQU>) のように企業によって提案されることもあるが、自分の組織に独自のスタイルガイドがないのならば、次節で説明する PEP 8 を使うのがよいだろう。

コードの形式<sup>フォーマット</sup>は見た目を変えるが、コードの動作は変えない。ここで言う「形式」には、例えばコード内の改行の位置、演算子の前後の空白、関数間の空行数などが含まれる。

一貫したスタイルを適用することで、コードがより読みやすくなる。一貫性のあるスタイルに則っていれば、新規のコードであっても、より速く読める。この一貫性によって、例えば括弧の欠落や不一致といった構文エラーを不用意に引き起こす危険も小さくなる。標準化されたコードであれば、何がどこにあるべきかがわかりやすくなるのだ。

一貫したスタイルのもう 1 つの大きな長所は、コードレビューの際にチームと形式について議論する必要がなくなることだ。コードの機能やプロジェクトの要件を満たしているかどうかのレビューに集中できるのだ。

### タブにするかスペースにするかの大論争

コードのインデント（字下げ）にタブとスペースのどちらを使うか？ これは開発者が議論することであり、インターネット上ではどちらが優れているかに関する長い議論がある。スペースはどのコンピュータでも一貫して表示されるが、タブはそうではない。しかし、タブのほうが入力が速く、保存される文字数が少ないのでファイルサイズが小さくなる。

2016 年、当時 Google で開発者支援を担当していた Felipe Hoffa は、10 億ファイル（14 テラバイト！）のコードを分析し、タブとスペースのどちらが人気があるかを分析した (<https://oreil.ly/d2K4X>)。その結果、スペースが圧倒

的に人気があった。

このような議論は、ソフトウェアエンジニアの思考に関する洞察を与えてくれるとも言えよう。開発者は、自分が書いているコードの細かな点にこだわるのだ。とはいっても、この議論はここに来て、ある程度解決されつつある。多くのIDEはタブを押すと、自動的にスペースを使ってコードをインデントしてくれる。つまり、タブを使うことによる「スピード」と、スペースを使うことによる「異なる環境での一貫性」の両方が実現されるようになったのだ。しかもこのプロセスは自動化されている。

### 6.1.1 PEP 8

Python Enhancement Proposal 8 (PEP 8。<https://oreil.ly/RqeYi>) は、Python の書式の標準を定める文書である。これは、Python が普及し始めた 2001 年に、Python 標準ライブラリのスタイルガイドとして Guido van Rossum、Barry Warsaw、Nick Coghlan によって書かれたものだ。Python でコードを書くすべての人の一貫性を高めるために、Python 開発者コミュニティによってデフォルトのスタイルガイドとして採用されている。

PEP 8 には次のように書かれている。

スタイルガイドは、「一貫性」に関するものである。このスタイルガイドとの一貫性は重要である。プロジェクト内での一貫性はより重要である。そして、1つのモジュールや関数内での一貫性が最も重要である。

PEP 8 には、コードの中で何をすべきで、何をすべきでないかというガイドラインがたくさんある。その 1 つに「if 文の『:』の後に改行を入れるべきである」というものがある。

```
# 正しい:
if foo == 'blah':
    do_blaah_thing()

# 違う:
if foo == 'blah': do_blaah_thing()
```

「違う」のほうで書いてもコードは実行されるが、「正しい」ほうがはるかに読みや

すい。

PEP 8 には、空白に関する記述が多い。なぜなら、空白はコードを読みやすく保つのに本当に役立つからだ。例えば、「=」の前後にスペースを置くのがベストプラクティスだとしているほか、関数定義前後の空行の数については、それがクラスの中の場合（1 行）、あるいは単独の場合（2 行）に分けて指定している。また、コメントの書き方や変数名の選び方についても触れている（本書の 9 章でも議論する）。そして、環境間の一貫性を保つために、インデントにはタブではなくスペースを使うことを勧めている。

PEP 8 にはもっと詳しく書かれているが、全部を読む必要はない。本章にあるツールの 1 つを使って、自分のコードがスタイルガイドに準拠していることを確認すればよいのだ。Flake8 や Pylint は IDE でコードをハイライトしてくれるので、どこを変更する必要があるかわかる。また、「6.1.3 Black を使ったフォーマッティング」でコードフォーマッターの Black について説明する。Black は PEP 8 に準拠しているが、独自のスタイル設定も備えている。

## 6.1.2 import のフォーマット

外部モジュールの import はバグを頻繁に引き起こす。コードを更新すると際に、インポートしたモジュールを更新するのを忘れるのはよくある間違いだ。このため、使用中のモジュールの明確なリストを持っておくとよい。

PEP 8 は、import をどのようにグループ分けするかの基準を定めている。

import は次の順序でグループ化されるべきである。

1. 標準ライブラリの import
2. 関連するサードパーティの import
3. ローカルなアプリケーションあるいはライブラリ固有の import

これを手動で行う必要はない。isort (<https://oreil.ly/7oNhq>) などのツールを使えば、モジュールの import を正しい順番に並べ替えしてくれる。

isort は次のコマンドでインストールする。

```
$ pip install isort
```

isort を実行する前の import は次のようにになっていたとしよう。

---

```
imports_sample.py (isort 前)
```

```
import time
from sklearn.metrics import mean_absolute_error

import sys, os
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd

from sklearn.neural_network import MLPRegressor
import matplotlib.pyplot as plt

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import FunctionTransformer, OneHotEncoder
```

isort は次のように実行する。

```
$ isort imports_sample.py
```

結果は次のようになる。

---

```
imports_sample.py (isort 後)
```

```
import os
import sys
import time

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import (FunctionTransformer, OneHotEncoder,
                                   StandardScaler)
```

このほうがはるかに読み取りやすく、PEP 8 に準拠している。IDE のプラグインとして isort を使うこともできる。なお、reorder-python-imports (<https://oreil.ly/ggPBW>) も isort とほぼ同じ機能を持つもので、重複する import を1つにしたり、1行で複数のライブラリから import しているものを1行に1つだけに変えるなどの機能を持つ。

### 6.1.3 Blackを使ったフォーマッティング

Blackはコード形式を自動化するツールだ。コードスタイルを強制するので、人間がレビューする必要はない。Blackの背景にある考え方、「醜いコードを素早く書いたとしても、ファイルを保存すれば魔法のように一貫性が保たれる」というものだ。Blackは、ツールの作者によって指定された一貫性のあるコーディングスタイルを適用する。このツールの名前は、ヘンリー・フォードの有名な言葉「Any color the customer wants, as long as it's black.（黒でありさえすれば、どんな色の車でも選べる）」に由来する。

BlackはPEP 8のサブセットを使用しているが、若干の違いがある。例えば、すべての行の長さを79文字に制限する代わりに、Blackは長い行を90文字前後になるように分割する適切な場所を見つけようとする。多くのオプションは変更できないが、Blackのドキュメント([https://oreil.ly/V1l\\_7](https://oreil.ly/V1l_7))に詳しく書かれているように、この行の長さの指定は変更できる。

Blackは次のコマンドでインストールできる。

```
$ pip install black
```

Jupyterノートブック上でもBlackを使いたい場合は、次の方法でインストールする。

```
$ pip install "black[jupyter]"
```

Blackが実際にどのように動作するか、次の例を使って確認しよう（この例は「6.2 リンター」のデモでも登場する）。2章の図2-3と類似のグラフを生成するコードだが、構文エラーが1つ、importが1つ抜けており、形式がPEP 8に準拠していない箇所がいくつかある。例6-1はBlackを実行する前のコードである。

例6-1 plot\_big\_o.py

```
# plot_big_o.py
import matplotlib.pyplot as plt

def plot_big_o(save_path)

    n = np.linspace(1, 10, 1000)
    line_names = ['Constant', 'Linear', 'Quadratic', 'Exponential', 'Logarithmic']
    big_o = [np.ones(n.shape), n, n**2, 2**n, np.log(n)]
```

```

fig, ax = plt.subplots ()
fig.set_facecolor("white")

ax.set_ylim(0,50)
for i in range(len(big_o)):
    ax.plot(n, big_o[ i], label= line_names[i ])
ax.set_ylabel('Relative Runtime')
ax.set_xlabel('Input Size')
ax.legend()

fig.savefig(save_path, bbox_inches='tight')

```

次のコマンドで Black を実行し、フォーマットする。

```
$ black plot_big_o.py
```

構文エラーのあるファイルは再フォーマットできないため、Black はこの時点で次のようなエラーメッセージを表示する。

```

error: cannot format plot_big_o.py: Cannot parse: 4:25: def plot_big_o(save_path)

Oh no! 💥 ❤️ 💥
1 file failed to reformat.

```

この場合、関数定義の最後の「:」(コロン) が欠けている。そこで、4 行目の「`def plot_big_o(save_path)`」を「`def plot_big_o(save_path):`」とする。

このエラーを修正して Black を再実行すると、スクリプトは次のようになる。

`plot_big_o.py` (Black を再実行後)

---

```

# plot_big_o.py
import matplotlib.pyplot as plt

def plot_big_o(save_path): ❶
    n = np.linspace(1, 10, 1000)
    line_names = ["Constant", "Linear", "Quadratic", "Exponential", "Logarithmic"] ❷
    big_o = [np.ones(n.shape), n, n**2, 2**n, np.log(n)]

    fig, ax = plt.subplots() ❸
    fig.set_facecolor("white")

    ax.set_ylim(0, 50)
    for i in range(len(big_o)):

```

```

    ax.plot(n, big_o[i], label=line_names[i])
    ax.set_ylabel("Relative Runtime")
    ax.set_xlabel("Input Size")
    ax.legend()

fig.savefig(save_path, bbox_inches="tight")

```

- ❶ 関数定義に続く空白行を削除
- ❷ 文字列を「'」(一重引用符) ではなく「"」(二重引用符) で囲む
- ❸ 余分なスペースを削除

例えば、次の行は、その下のように変更され、PEP 8 スタイルガイドに準拠している。

```

ax.plot(n, big_o[ i ], label=line_names[i ])
↓
ax.plot(n, big_o[i], label=line_names[i])

```

また、次のコマンドで Black の変更をプレビューすることもできる。

```
$ black plot_big_o.py --diff
```

Black に変更してほしくないコード行がある場合は、その行の末尾に「`# fmt: skip`」というコメントを付ければよい。また、「`# fmt: off`」というコメントをブロックの先頭に、「`# fmt: on`」というコメントをブロックの末尾に付けることで、`off` と `on` で囲まれたコードブロックをスキップすることもできる。

Black は上のコードをクリーンにしてくれたが、`import` の欠落については何もしなかった。そのためには、次節で説明するリンターを使う必要がある。Black については、12 章で自動実行について説明するときに再度触れる。

## 6.2 リンター

コードを実行する前にエラーチェックするツールを **リンター** (linter) と呼び、リンターを使ったチェックのことを **linting** (linting) と呼ぶ（「リント」で「linting」を意味する場合もある）。

linting という名前は、衣類乾燥機の糸くず (lint) に由来しており、この機能を果たす最初のツールは、コードからゴミを取り除くことから lint と呼ばれた。オ

リジナルの lint というプログラムは 1978 年に C 言語用に開発されたが、今ではリンターはどのようなプログラミング言語でも使われる存在になった。

Python のリンターはコードを分析し、実行時に失敗する原因となるいくつかの事柄を警告する。例えば、変数名のスペルミスやモジュールの import のし忘れで、コードを実行したときにエラーになってしまうようなものだ。

Python 用の一般的なlintツールとしては、Flake8 (<https://oreil.ly/JSTW>)、Pylint (<https://oreil.ly/zVVN2>)、Ruff (<https://oreil.ly/5v5fu>) がある。こうしたツールにはコードの形式のチェック機能があり、(PEP 8 かそのサブセットである) スタイルガイドへの準拠をチェックしてくれる。Pylint と Flake8 は、フォーマットの問題を発見してもコードを変更せず、代わりにハイライトする。つまり、自分で変更を加える前に、手作業で提案をレビューできる。また、Black の実行と組み合わせて最初に形式を変更することで、リンターが潜在的なエラーだけをピックアップすることもできる。Ruff は Black と同様にコードをlintingし、フォーマットも更新する。

## 6.2.1 リンティング用ツール

先ほどの**例6-1**を例に lint ツールを使う方法を見てみよう。

まずは、Pylint からだ。Pylint は次のコマンドでインストールできる。

```
$ pip install pylint
```

コマンドラインから**例6-1**の Pylint を実行すると次の結果が得られる。

```
$ pylint plot_big_o.py
*****
Module plot_big_o
plot_big_o.py:4:26: E0001: Parsing failed: 'expected ':' (plot_big_o, line 4)' (syntax-error)
```

構文エラーがあるため、Pylint はスクリプト全体の解析を完了しない。エラーの内容は正確には書かれていないが、4:26、つまり「4 行目の 26 文字目」にエラーがあることを意味している（前に見たように関数定義の末尾の:が足りない）。

このエラーを修正して Pylint を再実行すると、次のような出力が得られる。

```
$ pylint plot_big_o.py
*****
Module plot_big_o
plot_big_o.py:1:0: C0114: Missing module docstring (missing-module-docstring)
plot_big_o.py:2:0: E0401: Unable to import 'matplotlib.pyplot' (import-error)
plot_big_o.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)
```

```
plot_big_o.py:6:8: E0602: Undefined variable 'np' (undefined-variable)
plot_big_o.py:8:13: E0602: Undefined variable 'np' (undefined-variable)
plot_big_o.py:8:46: E0602: Undefined variable 'np' (undefined-variable)
plot_big_o.py:14:4: C0200: Consider using enumerate instead of iterating with range and len (consider-using-enumerate)
```

```
-----  
Your code has been rated at 0.00/10 (previous run: 0.00/10, +0.00)
```

今回、Pylint はスクリプトの残りの部分をスキャンすることができた。その結果、多くのエラー（E0602 のような E で始まるメッセージ）や、コードが規約に従っていない箇所（C0114 のような C で始まるメッセージ）が見つかる。このメッセージを見てコードを更新し、エラーを修正すればよい。どのように正しいバージョンにすればよいかが明確でない場合は、Pylint のドキュメント (<https://oreil.ly/MurWR>) が助けになる。

次は Flake8 を使ってみよう。Flake8 は、次のコマンドでインストールできる。

```
$ pip install flake8
```

**例 6-1** で Flake8 を実行すると、次のような出力が得られる。

```
$ flake8 plot_big_o.py
plot_big_o.py:4:25: E999 SyntaxError: invalid syntax
```

これは Pylint と同じ振る舞いだ。スクリプトの解析は完了せず、停止して構文エラーを知らせる。しかし、構文エラーを修正して Flake8 を再実行すると、異なる出力が得られる。

```
$ flake8 plot_big_o.py
plot_big_o.py:7:9: F821 undefined name 'np'
plot_big_o.py:8:80: E501 line too long (82 > 79 characters)
plot_big_o.py:9:14: F821 undefined name 'np'
plot_big_o.py:9:47: F821 undefined name 'np'
```

Flake8 は、Pylint とは異なるスタイルガイドを使用しているため、指摘するものも違っている。

次は Ruff を使ってみよう。Ruff は比較的新しいlintで、非常に高速に動作するよう設計されている。次のコマンドでインストールできる。

```
$ pip install ruff
```

Ruff はリントチェックとフォーマットチェックを分けている。次のコマンドではコードのリントチェックができる。

```
$ ruff check plot_big_o.py
```

Flake8 と同じように、Ruff は構文エラーで止まってしまうが、これを修正して再実行すると、次のようなエラーが列出される（周辺の行もリストされる）。

```
F821 Undefined name `np`  
  --> plot_big_o.py:7:9  
... 略...  
F821 Undefined name `np`  
  --> plot_big_o.py:9:14  
... 略...  
F821 Undefined name `np`  
  --> plot_big_o.py:9:47  
... 略...
```

次のような形式で `ruff` を実行すれば、コードのフォーマットを更新してくれる。

```
$ ruff format plot_big_o.py
```

これで「正しい」フォーマットにしてくれるが、何を変更したかは教えてくれない。各リンターは微妙に異なるスタイルガイドを採用しているため、コードの一貫性を保つためには、1つのリンターを選び、それに従うことが重要だ。チーム内でどのリンターを使うか合意形成しておこう。

## 6.2.2 IDE 内でのリンティング

IDE によっては、コードのリンティングやフォーマッティングに別のツールを実行するのではなく、コードを書きながらチェックするものもある。図6-1は、VS Code で `pylance` 拡張機能を使った例である。

The screenshot shows a VS Code window with the following details:

- File Path:** Users > musha > Desktop > 1musha > se4ds > software-engineering-for-data-scientists > src > example > ch06 > plot\_big\_o.py
- Code Content:**

```

1 # plot_big_o.py
2 import matplotlib.pyplot as plt
3
4 def plot_big_o(save_path):
5     n = np.linspace(1, 10, 1000)
6     line_names = ['Constant', 'Linear', 'Quadratic', 'Exponential', 'Logarithmic']
7     big_o = [np.ones(n.shape), n, n**2, 2**n, np.log(n)]
8
9     fig, ax = plt.subplots()
10    fig.set_facecolor("white")
11
12    ax.set_xlim(0,50)
13    for i in range(len(big_o)):
14        ax.plot(n, big_o[i], label= line_names[i])
15    ax.set_ylabel('Relative Runtime')
16    ax.set_xlabel('Input Size')
17    ax.legend()
18
19    fig.savefig(save_path, bbox_inches='tight')

```
- Output Panel:**
  - 問題 4 出力 デバッグ コンソール ターミナル ポート JUPYTER フィルター (例: テキスト、\*\*/\*.ts, !\*\*/node\_modules/\*\*)
  - エラーリスト:
    - ④ "n" が必要です Pylance [Ln 4, Col 26]
    - △ "np" が定義されていません Pylance[reportUndefinedVariable] [Ln 6, Col 9]
    - △ "n" が定義されていません Pylance[reportUndefinedVariable] [Ln 8, Col 14]
    - △ "np" が定義されていません Pylance[reportUndefinedVariable] [Ln 8, Col 47]

図6-1 VS コードでコーディングしながらlintingを行う

このツールは、実行時に発生するであろうエラーに下線を引き、下に説明を書いてくれる。Pylint や Flake8 などをlinterとして使うような設定も可能だ。さらには、開発中のユースケースには無関係な警告を無視するような設定もできる。また、この拡張機能を使ってVSCodeでJupyterノートブックのlintingもできる。

どのツールを選ぶにせよ、lintingを行うことで、多くのエラーを未然に発見し、コードの一貫性を保つことができるため、開発時間を大幅に節約できる。

## 6.3 型チェック

型チェックは、エラーが起きる前にバグを発見するもう1つの方法だ。「型 (type)」という用語は、Pythonで使われるオブジェクトのカテゴリー（種類）を表すものだ。整数 (int)、文字列 (string)、小数 (float。プログラミングでは「浮動小数点数」と呼ばれる) などである。関数が期待する入力の型と、実際に受け取る入力の型が不一致だとエラーになる。

例えば、次のコードでは、関数 `math.sqrt()` が (int や float などの) 「数值型」を期待しているが、引数 `my_int` として文字列 (string) を渡している

(example/ch06/ch06.ipynb)。

```
import math

my_float = '100'
print(math.sqrt(my_float))
```

これは次のようなエラーを引き起こす。

```
TypeError: must be real number, not str
```

さらに、Python は動的型付け言語であり、変数の型が変更可能である。これは Java のような言語とは異なる。Java などでは変数の型は一度指定すると固定され、変更はできない。

例えば次のコードはエラーなしで実行され、変数の型を変更する。

```
my_variable = 10
type(my_variable)

<class 'int'>

my_variable = 'hello'
type(my_variable)

<class 'str'>
```

`my_variable` は最初は整数 (`int`) だが、2 回目の代入のあとは文字列 `str` になる。バグの原因が型の間違いであるケースは多い。関数が期待している型とは異なる型を受け取ったり、正しくない型の結果を出力したりすることがある。これは非常によくあることなので、バグをチェックするためのテスト（次章で取り上げる）を大量に書かなくても済むように、バグを発見するためのツールが開発されている。

### 6.3.1 型注釈

型注釈 (type annotation) (型アノテーション) は型ヒント (type hint) とも呼ばれ、Python 3.5 で導入された。これはコードを読む際に、関数がどのような入力の型を期待したり返したりするかを伝えるものだ。型注釈は関数の期待される振る舞いを伝えるので、コードをより読みやすくするのに役立つ。また、より大きなコード

ベースにおける一貫性の確保と標準化にも役立つ。

型注釈は Python では比較的新しいもので、まだ多少議論の余地がある。読みやすさに役立つと考える人もいれば、コードを読みにくくすると思う人もいる。また、注釈の追加やチェックに余分な作業が発生するのも事実ではある。Python 言語の開発者は（PEP 484 で）、Python では「型注釈はオプションであり続ける」と明言している。自分のチームや組織が型注釈について態度を明確にしているのならば、その規定に従うべきだ。

型注釈は、「変数: 型」の形式で書く。例えば、関数の引数や戻り値の型を定義するには、次のようにする。

```
def 関数名(引数: 型) -> 戻り値の型:
```

**例 6-2** は、2 章で使った関数に型注釈を追加したものだ。この関数の定義でわかるように、この関数は `list` を入力とし `float` を返す。

例 6-2 mode\_using\_counter.py

---

```
from collections import Counter

def mode_using_counter(list_of_numbers: list) -> float:
    c = Counter(list_of_numbers)
    return c.most_common(1)[0][0]
```

また、次のようにすれば、入力リストに浮動小数点数だけを含めるように指定することもできる。

```
from collections import Counter
from typing import List

def mode_using_counter(list_of_numbers: List[float]) -> float:
    c = Counter(list_of_numbers)
    return c.most_common(1)[0][0]
```

Python 標準ライブラリ以外の型を使った型注釈の指定もできる。次は、NumPy の配列を使って型注釈を書いた例である。

```
import numpy as np

def array_operation(input_array: np.ndarray) -> np.ndarray:
    ...
```



Python の標準ライブラリ以外のライブラリのデータ構造の型チェックを行う場合、そのライブラリが型を提供していなければならぬ。型注釈はオプションの機能であり、ライブラリが型を提供していない場合、型チェッカーは型 Any が指定されているものとして扱う。

型注釈は、コードの機能には何の違いももたらさない。例えば、**例6-3** では1つの型注釈が間違っている。入力は list であるべきなのに float 型と注釈が付いている。それでもコードは正しく実行される。

#### 例6-3 mode\_using\_counter\_incorrect.py

---

```
from collections import Counter

def mode_using_counter(list_of_numbers: float) -> float:
    c = Counter(list_of_numbers)
    return c.most_common(1)[0][0]
```

型注釈は、型チェック用ツールとともに使う場合にのみ有用なので、基本的にはコードを実行したりデプロイしたりする前に型チェッカーを実行する。型チェッカーはコードを解析し、型が不一致かどうかをチェックする。テストを書いてこれを行うこともできるが、型チェッカーを使ったほうが簡単で速い。

本書の執筆時点でも最も人気のある型チェックツールは mypy (<https://mypy-lang.org>) で、次節でその使い方を説明する。他の型チェックツールとしては、人気が高まっている Pyright (<https://oreil.ly/2BIR2>) や、Pyre (<https://pyre-check.org>) がある。IDE も組み込みの機能として、あるいは Pyright などの拡張機能のインストールによって型チェックをサポートするものもある（型が不一致のときに IDE が教えてくれる）。

### 6.3.2 mypy による型チェック

次のコマンドで mypy をインストールできる。

```
$ pip install mypy
```

そして、次のようなコマンドで任意のスクリプトに対して実行できる。

```
$ mypy my_script.py
```

型注釈が正しい**例6-2**でmypyを実行すると、次の出力が得られる。

```
Success: no issues found in 1 source file
```

しかし、間違った型注釈を付けた**例6-3**でmypyを実行すると、次のような出力が表示される。

```
mode_using_counter_incorrect.py:4: error: No overload variant of "Counter" matches argument type "float" [call-overload]
mode_using_counter_incorrect.py:4: note: Possible overload variants:
mode_using_counter_incorrect.py:4: note:   def[_T] Counter(self, None = ..., /) -> Counter[_T]
mode_using_counter_incorrect.py:4: note:   def[_T] Counter(self, None = ..., /, **kwargs: int) -> Counter[str]
mode_using_counter_incorrect.py:4: note:   def[_T] Counter(self, SupportsKeysAndGetItem[_T, int], /) -> Counter[_T]
mode_using_counter_incorrect.py:4: note:   def[_T] Counter(self, Iterable[_T], /) -> Counter[_T]
Found 1 error in 1 file (checked 1 source file)
```

## 6.4 まとめ

本章では、コードのフォーマッティング、linting、型チェックがコードの品質向上させ、生産性の向上に役立つことを説明した。スタイルガイドに従って形式を整えることで、コードが読みやすくなる。lintingと型チェックは、コードが本番環境にデプロイされる前に潜在的なエラーを特定し、コードの堅牢性を高めてくれる。

ここで重要なことは、自分のチームや組織の標準に従うべきだということだ（まだ標準がない場合は導入するべきだ）。標準化された形式は、コードが何をしているのかを理解しやすいため、バグを防ぐのに役立つ。また、コードの細かな「形式」ではなく、「機能」に集中できる。

コードの形式、linting、型チェックについて覚えておくべき最も重要なことは、こうした作業を代行してくれるツールを使うことだ。手作業で行うのは、貴重な時間の使い方ではない。セットアップに時間がかかるかもしれないが、わずかな時間を投資すれば長期的には必ず報われる。

12章では、こうしたツールを自動化する方法を紹介する。また、次章では、コードの堅牢性を確保するためのもう1つの重要な側面である「テスト」について説明する。

# 7章 テスト

テストを書くことは、良いコードを書くために重要なスキルだ。簡単に言ってしまえば、**テスト**とは関数を呼び出し、その関数がやるべきことをやっているかどうかをチェックするコードのことだ。テストは、自分のコードが正しく動作しているという証拠と自信を与えてくれる。5章で見たロギングと同様、テストも最初はセットアップに手間がかかるが、プロジェクトが大きくなるにつれて、その手間が報われるようになる。

自分の書いたコードが本当に小さな1回限りの実験的なものであれば、テストはそれほど重要ではないだろう。しかし、より大きなシステムの一部だとしたらどうだろう。自分の書いたコードを他人が変更しているとしたらどうだろう。他人のコードが自作のコードを使って、結果を返しているとしたらどうだろう。テストは、自分のコードが動作していることを（ある程度）保証してくれるし、自分あるいは他人が行った変更によって自分のコードが動作しなくなったことも知ることができる。

テストは、自分一人でコードを書いている場合にも重要だ。コードを変更したときに、記憶に頼るのではなく、確実に動作し続けるためのセーフティネットとなる。また、テストはコードがどのように動くべきかの例を示してくれるので、コードを読みやすくし、保守しやすくなる。

未経験の人にとってはテストを書くのは難しく思えるかもしれない（うまくいかないときの動作を知っているベテランのほうがテストは楽にかける）。ただ、「7.3.1 基本的なテスト」で詳しく説明するように、手始めに自分のコードが思いどおりに動いているかをテストすることから始めればよい。そして、コードがうまく動作しない場面に出くわしたら、テストを追加していくばよい。

ソフトウェア開発におけるテストには、大きく分けて2種類ある——「自動化されたテスト（自動テスト）」と、ソフトウェア製品全体の「ユーザーテスト」だ。ユー

ザーテストは別のチームが担当する可能性が高いが、本番コードを書いているのであれば、自動テストを書くべきだ。本書では、自動テストのみを取り上げる。

本章では、テストを書く動機について深く掘り下げる後、非常に単純なテストと、やや複雑なテストの書き方を説明する。自動化する方法を紹介するので、テストを手動で実行する必要はない。また、データの検証や機械学習（ML）のテストについても詳しく見ていく。



本書で言う「テスト」には A/B テストは含まれない。A/B テストでは、例えばあるソフトウェア機能の 2 つの異なるバージョンなど、どのバリエーションが好みいかを確認するための実験を行う。ここで扱うテストは、コードが期待どおりに動くかどうかをチェックするものだ。

## 7.1 テストを書くべき理由

期待どおりの結果が出るまでコードを書き直すことはよくあることだ。それで「仕事は終わり」と言ってよいのかかもしれない。しかし、入力可能なものをすべて試していなかつたり、将来入力データが変わったりしたらどうだろう。ある時点ではうまくいっていたのに、予期せぬことが起きてコードが動かなくなってしまうという経験はよくある。十分な堅牢性がなかったのだ。コードがより大きなシステムの一部である場合、大きな問題だ。

コードに破壊的な影響を与えるかもしれない要因は数多い。ライブラリや言語の新しいバージョンが出るかもしれないし、OS をアップグレードするかもしれない。コードが将来動かなくなってしまう理由の（長い！）リストについては、例えば <https://oreil.ly/oG4oq> をチェックしてほしい。

さらに、コードを開発したのと異なる環境では、コードが正しく動作しない可能性がある。例えば、オンラインダッシュボード、他人のコンピュータ、Docker コンテナ、本番環境などだ。テストがあれば、コードを別の環境に移してテストを実行し、そこでの動作確認がすぐに行える。

テストは、自分のコードが意図したとおりに動作することを、ある程度保証してくれる。これは、自分のコードを他人が信頼するのに役立ち、コードが堅牢で良質であることを示すシグナルともなる。また、医療分野のようなリスクの高い状況で仕事をしている場合や、多くのユーザーが自分のコードに依存している場合など、自分のコードについてある程度の保証をする役目を担ってくれることになる。

テストはまた、コードが「破壊」されることを恐れずに、変更を加える自信を与えてくれる。テストを行うことで、変更を加えてもコードが関数として正しく機能することを確認できる。これはデバッグ（5章）やリファクタリング（8章）に非常に役立つ。

### テスト駆動開発

**テスト駆動開発** (TDD : Test-Driven Development) とは、開発者がプロジェクトの目的を達成する「オブジェクトや関数等のコードを書く前にテストを作成しておく」というソフトウェア開発プロセスである。その目的は、テストを使ってコードの要件を記述することであり、期待される振る舞いと関数の入出力をテストに指定する。そして、開発者はテストに合格するようにコードを書く。これにより、開発者はコードをモジュール化し、複雑な問題をテストしやすい断片に分解することができる。

このアプローチはデータサイエンス (DS) のプロジェクトではうまくいかない場合が多い。なぜなら、一般的な DS プロジェクトの開始時点では、どのような関数が必要なのか正確にはわからないからだ。探索的な作業から始めることが多く、コードがどのように構造化されるべきかがわかるのはプロジェクトの終盤になってからだ。TDD についてもっと知りたいのであれば、Michał Jaworski + Tarek Ziadé 著『エキスパート Python プログラミング 改訂 3 版』(KADOKAWA) が TDD の観点から Python 開発をカバーしている。

## 7.2 テストの時期

DS のプロジェクトでは、いつテストを書き始めるべきかを正確に知るのは難しいかもしれない。一般的に、プロジェクトの探索的な段階では、テストに時間をかける価値はない。なぜなら、どのコードが将来役に立つかわからないからだ。しかし、コードを再利用して修正する必要があるときは、テストを追加する良いタイミングになる。また、単発の分析では、コードにテストを追加する重要性は低くなる。結果を素早く得ることと、その結果が正しいことを確認するために時間をかけることはトレードオフがある。

プロジェクトの早い段階でテストを書くことで、名前の不統一、インポートの欠

落、スペルミス、構文エラーなどの単純なミスを発見できる。こうしたミスは、コードエディターでは発見できないこともある。

コードが本番環境にデプロイされる前や、誰かが使う前にテストを書くことは、コードが堅牢で再現可能であるという確信を得るために最も重要である。テストを追加するもう1つのタイミングは、コードに何か問題が生じて予期せぬエラーが発生したときだ。テストを追加すれば、エラーの原因を発見するのに役立つ。テストは、コードのリファクタリングが必要な場合にも非常に役立つ。

自分のコードだけをテストすればよいことを覚えておいてほしい。インポートしたライブラリや依存する他人のコードをテストする必要はない。これらはあなたの責任ではないし、すでに独自のテストがあるはずだ。

## 7.3 テストの書き方と実行方法

簡単なテストの構成と書き方について見ていく。期待する処理だけでなく、予期しない入力をテストする方法や、テストを自動的に実行する方法についても議論する。また、単体テストと統合テストという2つの一般的なテストの例も紹介する。

### 7.3.1 基本的なテスト

最も単純なテストは、関数が正しく実行され、正しく動作していることを確認するものだ。テストに何を含めるべきか悩んでいるのであれば、関数を開発しているときにJupyterノートブックに繰り返し表示したい事柄を出発点にするとよいだろう。

pytestのドキュメント (<https://oreil.ly/slpmH>) に記載されているように、テストは4つの段階に分けて構成する。

1. アレンジ——例えば、データをロードするなど、関数を実行するのに必要なものをすべてセットする
2. 実行——テストを実行する
3. アサート——関数の実行結果が期待どおりであることをチェックする
4. クリーンアップ——テストが痕跡を残さないようにする。例えば、ファイルを開いた場合は必ず閉じる

ここでは2章の`weighted_mean`関数を使い、期待どおりに動いているかチェックするテストを書いてみよう。これがその関数だ（scripyとpandsがインストール

されていない場合は、`pip` を使ってインストールが必要)。

```
def weighted_mean(num_list, weights):
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights[i])
    return (running_total/sum(weights))
```

次のコードはこの関数に関して、簡単な入力を確認するテストだ (`example/ch07/test1-basic/test_weighted_mean.py`)。

```
from ch07_functions import weighted_mean ❶

def test_weighted_mean(): ❷
    list_a = [1, 2, 4]
    list_b = [1, 2, 4] ❸

    result = weighted_mean(list_a, list_b) ❹

    assert result == 3 ❽
```

- ❶ テスト関数は関数とは別の Python ファイルにあり、テスト対象の関数は `ch07_functions.py` ファイルからインポートされる。テストを含むファイルは `test_〇〇〇.py` または `〇〇〇_test.py` の形式でなければならない
- ❷ テスト関数名は `test_`で始まるのが慣例である
- ❸ アレンジステップでは、テストする関数の入力データを作成する
- ❹ 実行ステップでは、選んだ入力データを使って関数を呼び出す
- ❽ アサートステップでは、関数が生成した結果が期待どおりかをチェックする

このテストでは、関数への典型的な入力から始めた。それから関数を実行し、`assert` 関数を使って結果をチェックしている。

次のステップは、テストを実行し、パスするかどうかをチェックすることだ。`pytest` のようなテストフレームワークを使ってテストを行うことができるが、これについては「7.3.3 `pytest` を使った自動テストの実行」で説明する。テストにパスすれば、少なくともこれらの入力データ値に対してコードが正しく動作していることがわかる。テストが失敗した場合は、次の 2 つを行う必要がある。

1. テストが正確であり、アサートステップでチェックしている値が、その入力セッ

トに対して本当に期待される値であることを確認する

2. テストが失敗する原因となっているコードの間違いを探し、パスするようにその間違いを修正する

別の例を見よう。他の章でも見た `fit_trendline` 関数の例だ。

```
from scipy.stats import linregress

def fit_trendline(year_timestamps, data):
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared
```

そして、これがそのテストだ (`example/ch07/test1-basic/test_fit_trendline.py`)。

```
from ch07_functions import fit_trendline

def test_fit_trendline():

    data = [1, 2, 3]
    timestamps = [2020, 2021, 2022]

    slope, r_squared = fit_trendline(timestamps, data)

    assert slope == 1
    assert r_squared == 1
```

今回は単純なデータを使うことにした。これは関数が本当に期待どおりに動作しているかを確認するためのものである。次節で、テストすべき他の入力について説明する。

テストを別のフォルダに置き、テストを含む Python ファイルの名前を `test_` で始めるのが一般的なやり方だ。こうすることで、テストを他のコードから切り離すことができ、「7.3.3 pytest を使った自動テストの実行」で説明するように、自動テストの際に簡単に見つけられる。

### 7.3.2 予期しない入力のテスト

「想定している入力」に対してテストするだけでなく、「想定外の入力」に対応できるかどうかをテストすることで、コードの堅牢性を高めることも有効だ。これは

「エッジケース」とも呼ばれ、極端あるいは異常な入力値で特別処理が必要なものということをいう。例えば、空の文字列や正しくない型のデータなどがこの例だ。

`weighted_mean` のテストの新バージョンを書いてみよう。予期せぬ入力に対してコードが堅牢であることをチェックするためのコードを加えている(`example/ch07/test2-unexpected/test_weighted_mean.py`)。

```
def test_weighted_mean():

    result = weighted_mean([1, 2, 4], [1, 2, 4])

    assert result == 3

    empty_list_result = weighted_mean([], [])

    assert not empty_list_result
```

このテストでは、`weighted_mean` が入力として空のリストを受け取った場合、`None` を返すことを期待している。つまり、このテストに合格するためには、`weighted_mean` 関数を更新して、この入力を処理する必要がある(`example/ch07/test2-unexpected/ch07_functions2.py`)。

```
def weighted_mean(num_list, weights):
    if not (num_list or weights):
        return None
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights[i])
    return (running_total/sum(weights))
```

次に、入力として不正な型のデータを受け取った場合に、関数が正しく動作することをチェックするテストを追加したい。そこで `weighted_mean` が入力リストの 1 つに文字列を受け取った場合、`None` を返すかテストする(`example/ch07/test2-unexpected/test_weighted_mean.py`)。

```
def test_weighted_mean():

    result = weighted_mean([1, 2, 4], [1, 2, 4])
    assert result == 3

    empty_list_result = weighted_mean([], [])
    assert not empty_list_result
```

```
wrong_types_result = weighted_mean(['one', 2, 4], [1, 2, 4])
assert not wrong_types_result
```

テストの追加と関数の更新がどのように連動しているかわかるだろう。多くの場合、「テストを更新してからコードを更新」という作業を繰り返すことになる。

想定外の入力を使ったテストは、コードがうまく動いているときには難しい。しかし、コードが壊れ、エラー処理のためにコードを更新したら、次に同じことが起こったときにコードが壊れないことを確認するためのテストを追加するべきだ。また、どのような入力が重大な結果をもたらすかを考え、それに対するテストを行うのもよい。

役に立つかかもしれないライブラリをいくつか紹介しておこう。Faker ライブラリ (<https://oreil.ly/A7Wb->) は、タイムスタンプやメールアドレスなど、さまざまな形式のフェイクデータを作成する。これは、テストコードに表示するべきではない個人データや機密データを扱う場合に重要だ。もう 1 つは Hypothesis ライブラリ (<https://oreil.ly/uTNW0>) で、コードが機能しないエッジケースの発見に役立つ。Hypothesis はプロパティベースのテストを実行し、さまざまなシナリオを記述したテストを書くことができる。

### 7.3.3 pytestを使った自動テストの実行

pytest (<https://oreil.ly/z1MRC>) は、テストの検索、収集、実行を行うフレームワークだ。pytest のようなフレームワークを使えば、関数がたくさんあってもテスト関数を 1 つずつ実行する必要はないし、すべてのテストを実行するために定型的なコードを書く必要もない。

pytest は Python 標準ライブラリの一部ではないので、次のコマンドを使ってインストールする。

```
$ pip install pytest
```

「`pytest <テストファイル>`」の形式のコマンドで、1 つのテストファイルを実行できる。

```
$ cd example/ch07/test1-basic
$ pytest test_weighted.py
```

フォルダ内のすべてのテストを実行したい場合は、`pytest` コマンドを単独で使う。`pytest` はこのコマンドを実行したフォルダ内のすべてのファイルを再帰的に（サブフォルダも含めて）検索し、`test_*.py` または`*_test.py` というパターンマッチングにマッチするファイルのテストを実行する（そうしたファイルの中で、`test_`で始まるテスト関数を実行する）。`pytest` がテストを見つけ出す方法の詳細は `pytest` のドキュメント (<https://oreil.ly/fGAj0>) を参照されたい。

すべてのテストにパスすれば、次のような出力が表示される。

```
$ cd ../test3-pytest-success
$ pytest
=====
platform darwin -- Python 3.11.5, pytest-8.3.5, pluggy-1.6.0
rootdir: /Users/.../example/ch07/test3-pytest-success
plugins: anyio-4.3.0, Faker-37.1.0, typeguard-4.4.2
collected 2 items

test_fit_trendline.py . [ 50%]
test_weighted_mean.py . [100%]

===== 2 passed in 0.42s =====
```

しかし、テストが失敗すると、次のような出力になる。

```
$ cd ../test4-pytest-fail
$ pytest
=====
platform darwin -- Python 3.11.5, pytest-8.3.5, pluggy-1.6.0
rootdir: /Users/.../example/ch07/test4-pytest-fail
plugins: anyio-4.3.0, Faker-37.1.0, typeguard-4.4.2
collected 2 items

test_fit_trendline.py . [ 50%]
test_weighted_mean.py F [100%]

===== FAILURES =====
----- test_weighted_mean -----
def test_weighted_mean():

    list_a = [1, 2, 4]
    list_b = [1, 2, 5]

    result = weighted_mean(list_a, list_b)

>         assert result == 3
```

```
E     assert 3.125 == 3

test_weighted_mean.py:10: AssertionError
=====
FAILED test_weighted_mean.py::test_weighted_mean - assert 3.125 == 3
=====
1 failed, 1 passed in 0.46s =====
```

pytest はテストが失敗したことを示すメッセージを出力し、失敗した行をハイライトする。また、アサーションエラーも出力し、コードが生成した正しくない値を示す。

pytest には、テストのスキップやログの処理など、より高度な機能がたくさんある。詳細は pytest のドキュメント (<https://oreil.ly/eSN4m>) を参照されたい。

## 7.4 その他のテスト用フレームワーク

pytest は、機能が充実していて使いやすいので、人気のあるテストフレームワークだ。しかし、他にもいくつかの選択肢がある。unittest (<https://oreil.ly/uRZyg>) は Python 標準ライブラリに含まれているので、依存するものを少なくしたい場合には有用だ。ただし、pytest よりもセットアップが少し複雑だ。

tox (<https://tox.wiki/en>) も知っておく価値のあるツールだ。仮想環境のセットアップ、アプリケーションのインストール、依存関係や Python バージョンの管理などを含むテストランナーだ。このツールは、コードが他の環境でも正しく動くことを確認するのに非常に役に立つ。

テストのカバレッジ (coverage) という概念についても触れておきたい。これは、すべてのテストを実行したときに、ファイル内のコード行が実行される割合のことである。テストカバレッジは pytest-cov プラグイン (<https://oreil.ly/BYfJR>) を使って測定できる。これはすべての関数がテストされたかどうかをチェックするのに役立つが、どの行がテストによって「実行」されたかをチェックするだけで、どの行が「テスト」されたかをチェックするわけではない。

テストは、コードを、(10 章で説明する) バージョン管理システム (VCS) に「コミット」する前に実行すべきだ。このあたりの事柄については 12 章でさらに詳しく説明する（テストを自動的に実行する方法についても説明する）。

## 7.5 テストの種類

さまざまなテストを説明する用語は多岐にわたる。Patrick Viafore は、著書『ロ

『バスト Python』（オライリー・ジャパン）の中で、ソフトウェアエンジニアリングでよく使われるテストの種類を 6 つに分類している。

#### **単体テスト** (unit test)

関数やクラスといったコードの単位が、開発者の期待どおりに動くかどうかをチェックする

#### **統合テスト** (integration test)

より大きなシステムが正しく接続されていることを確認する

#### **受け入れテスト** (acceptance test)

システムがユーザーの期待どおりに動くかどうかをチェックする

#### **負荷テスト** (load test)

データ量やユーザー数が増えても、システムが正常に機能することを確認する

#### **セキュリティテスト** (security test)

システムが攻撃に対して耐性があるかどうかをチェックする

#### **ユーザビリティテスト** (usability test)

システムが直感的に使えるかどうかをチェックする

本書では単体テストと統合テストに焦点を絞る。DS のコードを書く上で、この 2 つが最も役に立つと思うからだ。

### 7.5.1 単体テスト

本章でこれまで紹介してきたテストは、すべて単体テストだ。単体テストはコードの小さなコンポーネント（例えば、関数やクラス）を取り出してテストするもので、コードの開発中やデバッグ中に書くのが一般的だ。

単体テストは素早く実行すべきなので、小さなデータセットを使うのがベストだ。データベースへの接続など、実行に時間がかかる演算は避ける。代わりに仮のデータを使うのがよい。また、単体テストは「決定論的」であるべきだ。つまり、同じ入力に対して常に同じ結果が出るようにしなければならない。ランダムな要素を含むようなものは使わないことだ。

## 7.5.2 統合テスト

統合テストは、より大きなシステムが正しく機能することを確認する。スクリプト全体やプロジェクトの大きな塊をテストし、単体の関数を複数組み合わせてテストする。

(小規模な) 統合テストを実行してみよう。1章で見た `process_sdg_data` と 5 章で見た `fit_trendline` を使う。

これら 2 つの関数を使うと、Excel ファイルを受け取り、トレンドラインの傾きと  $R^2$  を出力できる (`example/ch07/integration.py`)。

```
from scipy.stats import linregress
import pandas as pd

def process_sdg_data(excel_file, columns_to_drop):
    df = pd.read_excel(excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("GeoAreaName").transpose()
    return df

def fit_trendline(year_timestamps, data):
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared
```

次が 2 つの関数を結びつけるコードも含めた統合テストのコードだ (`example/ch07/test_integration.py`)。

```
from integration import process_sdg_data, fit_trendline

def test_processing_trendline():
    df = process_sdg_data(
        "../data/SG_GEN_PARL.xlsx",
        [
            "Goal",
            "Target",
            "Indicator",
            "SeriesCode",
            "SeriesDescription",
            "GeoAreaCode",
            "Reporting Type",
            "Sex",
            "Units",
        ],
        ["Year"]
    )
    slope, r_squared = fit_trendline(df["Year"], df["Value"])
    assert round(slope, 3) == -0.001
    assert round(r_squared, 3) == 0.999
```

```

],
)
timestamps = [int(i) for i in df.index.tolist()]
uk_parl = df["United Kingdom of Great Britain and Northern Ireland"].tolist()

slope, r_squared = fit_trendline(timestamps, uk_parl)

assert slope == 0.836
assert r_squared == 0.868

```

このテストは、2つの関数を合わせた出力が、期待されるものと合致しているかをチェックする。これは特定のデータセットだけを使ったテストであるが、いろいろなデータセットに対して行う必要がある。統合テストに使うデータセットは代表的なものだけでなく、「7.3.2 予期しない入力のテスト」で説明したようなさまざまな種類のものを用意すべきである。

## 7.6 データ検証

データ検証 (data validation) は、DS プロジェクトにおける重要なテストの一種である。データが期待どおりのものであることを確認するプロセスである。本章で紹介した他のテストとは少し違う。なぜなら、コードではなく、扱っている「データ」をテストするものだからだ。

データ検証は、データエンジニアリングチームが担当する場合もあれば、MLパイプラインの一段階として行う場合もある。Catherine Nelson + Hannis Hapke 著『入門 機械学習パイプライン』(オライリー・ジャパン) には、ML パイプラインの一部としてのデータ検証についての詳細が記載されている。本節では、データを検証するためのいくつかの方法と、そのために使えるツールを紹介する。

### 7.6.1 データ検証の例

データの検証は重要である。なぜなら、DS プロジェクトの期間中にデータが頻繁に変化する可能性があるからだ。つまり、具体的な「値」ではなく、データの「特性」をテストすることが望まれるのだ。

一般的なデータ検証の手順は次のとおりである。

- データセットに必要な列がすべて存在することをテスト

- 各列のデータが正しい型であることをテスト
- カラムがすべてヌル（null）値でないこと、あるいはヌル値が一定の割合以上ないことをテスト
- データの統計的特性が期待どおりであることをチェック（データの平均、標準偏差、範囲など）
- 前項と関連するが、データ分布が前日、前週、といった関連する期間のデータと類似していることをチェック
- 異常なデータをチェック

次節では、Pandera モジュールを使ったデータの検証方法を説明する。Great Expectations (<https://oreil.ly/6dp8t>) もデータ検証のための優れたツールで、より高度な機能を備えているが、Pandera よりもセットアップに時間がかかる。さらにその後の節で、型注釈（型ヒント）を使うデータ検証のもう 1 つの選択肢である Pydantic を取り上げる。

## 7.6.2 Pandera を使ったデータ検証

Pandera モジュールは軽量なデータ検証のツールで、スキーマを使って pandas の DataFrame などのデータ構造（Dask DataFrame など類似のものも含む）を検証する。ここでは概要を説明するので、詳しい機能は Pandera のドキュメント (<https://oreil.ly/xvOrN>) を参照されたい。

Pandera は次のコマンドでインストールできる。

```
$ pip install pandera
```

図 7-1 は、4 章と同じデータを使って検証する DataFrame の一部を示している。

	Year	India	United States of America
0	2000	9.02	13.33
1	2001	9.01	14.02
2	2002	8.84	14.02
3	2003	8.84	14.25
4	2004	8.84	14.25

図 7-1 検証対象の DataFrame

データを検証するために、DataFrame がどのようなカラムを含むべきかを、各カラムのデータ型とともに定義するスキーマを定義する。

```
import pandera.pandas as pa

schema = pa.DataFrameSchema({
    "Year": pa.Column(int), ❶
    "India": pa.Column(float), ❷
    "United States of America": pa.Column(float) ❸
})
```

- ❶ int (整数) を持つ「Year」という名前のカラムがある
- ❷ float (浮動小数点数) を持つ「India」という名前のカラムがある
- ❸ float を持つ「United States of America」という名前のカラムがある

次に、次のコマンドでデータを検証する。

```
schema(df)
```

これはスキーマに従ってデータを検証し、すべてのカラムが存在することをチェックし、各カラムのデータ型を検証する。スキーマで指定されたカラムがすべて存在しない場合はエラーを出力 (throw) するが、余分なカラムが存在しても文句は言わない。

列の値の「プロパティ」を検証することもできる。引数 `checks=`を使って、すべての値が期待する範囲内にあることをチェックできる。これはさまざまな Pandera の Check オブジェクトを受け付ける。

```
schema = pa.DataFrameSchema({
    "Year": pa.Column(int, checks=pa.Check.in_range(2000, 2023)),
    "India": pa.Column(float, checks=pa.Check.in_range(0, 100)),
    "United States of America": pa.Column(float, checks=pa.Check.in_range(0, 100))
})
```

データ検証がパスすれば、必要なカラムがすべて存在し、各カラムのデータが正しい型であり、データが期待した範囲内にあることが確認できる。簡単な例ではあるが、これをもとに独自のデータセットの検証ができるだろう。

### 7.6.3 Pydanticによるデータ検証

Pydantic ([https://oreil.ly/ey\\_Op](https://oreil.ly/ey_Op)) を使えば、実行時にデータを検証できる。これは非常に便利なライブラリで、Python コミュニティで人気が高まっている。Pydantic は「6.3 型チェック」で紹介した型注釈を使ってデータを検証するが、mypy のような静的解析ツールではない。その代わり、検証はコードを実行するときに行われるので、別のツールを実行する必要はない。11 章で説明するように、API 用にコードをデプロイする際に特に役立つ。

Pydantic は次のコマンドでインストールできる。

```
$ pip install pydantic
```

Pydantic は **データスキーマ** という概念を使ってデータを検証する。まず、データの形式を定義するスキーマを定義し、そのスキーマを使って新しいデータが正しい形式であるかどうかをチェックする。次のコードは 1 章で紹介した国連の持続可能な開発目標データのスキーマ定義の例である。データの型を定義し、それが必須（存在しないとエラーになる）か任意（optional）かを定義している。

```
from pydantic import BaseModel
from typing import Optional
from datetime import datetime

class CountryData(BaseModel):
    country_name: str ①
    population: int ②
    literacy_rate_2020: Optional[float] ③
    timestamp: Optional[datetime] = None ④
```

- ① 国名は必須で、文字列か、エラーを出さずに文字列にキャストできるものでなければならない
- ② 母集団は必須で、整数でなければならない
- ③ 識字率は任意で、浮動小数点数か、浮動小数点数にキャストできるものでなければならない
- ④ タイムスタンプはオプションで、`datetime` オブジェクトか、`datetime` オブジェクトにキャストできるものでなければならない。データが渡されない場合、デフォルト値は `None` となる

次に、正しいデータとして検証されるデータの例を示す。

```
sample_data_correct = {  
    'country_name': 'India',  
    'population': 1417242151,  
    'literacy_rate_2020': 79.43,  
    'timestamp': datetime.now()  
}
```

このデータを使って、新しい `CountryData` オブジェクトを作成できる。データが正しい形式であればエラーは発生しない。

```
india = CountryData(**sample_data_correct)
```

これらのデータのどれでも調べることができる。

```
india.timestamp
```

```
datetime.datetime(2023, 3, 7, 16, 3, 41, 423508)
```

次のように条件に合わないデータを渡すとどうなるか見てみよう。

```
sample_data_incorrect = {  
    'country_name': 'United States',  
    'population': None,  
    'literacy_rate_2020': None,  
    'timestamp': None  
}  
  
united_states = CountryData(**sample_data_incorrect)
```

この場合、`population` フィールドが欠けている。Pydantic は次のエラーを出す（メッセージの最後の部分を示す）。

```
ValidationError: 1 validation error for CountryData  
population  
  Input should be a valid integer [type=int_type, input_value=None, input_type=NoneType]  
  For further information visit https://errors.pydantic.dev/2.6/v/int_type
```

Pydantic は、大規模なプロジェクトの入力データが期待どおりのものであるかどうかをチェックするのに非常に便利である。API での使い方の例は「11.2.2 API への機能の追加」で紹介しよう。

## 7.7 機械学習用のテスト

機械学習（ML）のコードは、他のコードとは少し違った方法でテストする必要がある。というのも、与えられたデータセットからどのようなモデルが得られるかは正確にはわからないからだ。ほとんどの ML アルゴリズムは、何らかのランダム性を含んでいる。しかし、これは ML コードをテストできないという意味ではない。複雑なトピックだが、本節で概要を説明し、いくつかの有用な戦略を紹介しよう。

**図 7-2** は、一般的な ML プロジェクトの構造を示している。すべてのステップのテストの可能性を順番に見ていき、特にいくつかのステップについて詳しく見ることにしよう。

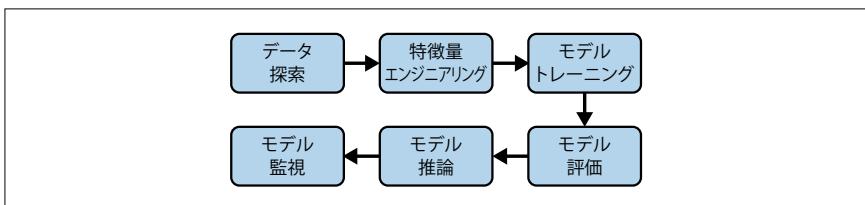


図 7-2 機械学習プロジェクトのステップ

### データ探索

ほかの機会に再利用するコードを書かない限り、プロジェクトの探索的な段階でテストを書く必要はない

### 特微量エンジニアリング

特微量エンジニアリングのコードの単体テストを書き、このステップにデータ検証を含めることができる

### モデルトレーニング

「7.7.1 モデルトレーニングのテスト」で詳しく説明する

## モデル評価

モデル評価では、あるデータセットに対する学習済みモデルの性能を評価する。これ自体は一種のテストであるが、モデルのトレーニング実行ごとに結果は微妙に異なる。例えば、精度が常に一定のパーセンテージ以上でなければならぬと強制することで、これをテストとして形式化できる

## モデル推論

「7.7.2 モデル推論のテスト」で詳しく説明する

## モデル監視<sup>モニタリング</sup>

本番でのモデル監視は、外部のツールやライブラリによって処理されることが多いが、自分でコードを書けば単体テストを行える

### 7.7.1 モデルトレーニングのテスト

テストモデルのトレーニングが難しいのは、入力データがあったとしても、期待される出力がわからないからだ。ニューラルネットワークの重みがどうあるべきか正確にはわからないし、保存したモデルファイルに入るすべてのパラメーターもわからない。特定のデータセットが特定のモデル構造を与えるかどうかをテストすることはできない。しかし、テスト可能な項目は間違いなくあり、トレーニングに時間（例えば数分以上）がかかる大きなモデルを使うのか、小さなモデルを使うのかによって、戦略は変わってくる。

モデルのトレーニングに長い時間がかかる場合、モデルの出力が正しいかどうかのテストや、最終的なモデルの精度のテストをするのは現実的ではない。その代わりに、トレーニングが期待どおりに実行されているか、モデルがクラッシュせずにトレーニングされているかをチェックする必要がある。そのためには、トレーニングコードに、トレーニングステップの前後でモデルの重みを比較し、重みが変化していくことを確認するテストを書けばよい。

ニューラルネットワークに関しては、最初の数回の学習ステップで損失が減少するかどうかのチェックを行うという選択肢もある。また、モデルが正しい形状の入力テンソルを受け入れ、正しい形状の出力を生成するかをチェックするテストを書くこともできる。エラーの原因となることが多いものだ。

小規模なモデルの場合は、学習プロセスの最後にモデルの存在をチェックする統合テストを書くのも有用だ。スピードアップのためにトレーニングデータの一部を使う

こともできるし、フェイクデータを使うこともできる。

## 7.7.2 モデル推論のテスト

モデルをトレーニングした後は、そのモデルが正しく予測を返すかどうかをチェックする必要がある。本番環境で推論を実行するモデルをテストするには、主に2つの方法がある。まず第1に、推論コードが正しく動作していることをチェックする単体テストと統合テストを書く。フェイクの入力データが与えられたときに予測を返すかどうか、空の入力を正しく処理するか、エラーをうまく処理しているかといった事柄のテストだ。モデルが非常に大きい場合は、より小さなサンプルモデルでいくつかテストを実行し、フルバージョンのモデルでいくつかテストを実行するとよいだろう。

第2に、特定のデータ例をモデルに送り、モデルの予測が期待どおりであることを確認する。これはモデルの評価とは異なる。評価データセット全体ではなく、小数のデータサンプルだけを使うものだ。回帰モデルの場合、与えられたデータサンプルに対するモデルの出力が正しい範囲内にあることをチェックできる。分類モデルの場合は、常に特定のクラスであるべき入力を与えてみる。このような予測を正しく行った場合にのみ、モデルをデプロイすることになる。

MLのテストについて詳しく知りたい場合は、Jeremy Jordanの記事 (<https://oreil.ly/FUtlq>) がお勧めだ。

## 7.8 まとめ

テストを書くことは、良いコードを書くための重要な要素であり、特に、より堅牢なコードを書く助けとなる。テストにより次のような利点が得られる。

- コードが期待どおりに動いていることが確認できる
- コード変更時に動作が確認できる
- 他の環境における動作の確認ができる
- 自分のコードへの信頼度が向上する
- デバッグやリファクタリングの助けとなる

最も単純なテストは、関数が簡単な入力で期待どおりの結果を返すかどうかをチェックするものだ。テストの際には、アレンジ、実行、アサート、クリーンアップの各ステップを構成する。次に、エッジケースを追加する。テストを手動で実行する

のではなく、pytest のようなテストフレームワークを使おう。

プロジェクト内のデータが期待どおりであることを確認するために、プロジェクトの総体的な特性を使ってデータを検証できる。Pandera は pandas の DataFrame を検証するための使いやすい選択肢である。

機械学習では、与えられたデータセットからどのようなモデルが生成されるか正確にはわからないため、一般的なソフトウェア開発とは異なるアプローチが必要になる。しかし、可能であれば単体テストも含め、テストを書くべきだ。



# 8章

## デザインとリファクタリング

本章では、コードの1行1行に関する細部について考えることから、より大きな全体像、つまりプロジェクトのデザイン方法、コードのアレンジ方法、そしてデザインが変わったときのコードのリファクタリングといった事柄について検討する。プロジェクトのハイレベルな構造をどのように組織化、標準化するか、そしてどのようにコードを再利用可能なモジュールに分割するのがよいのかを説明していく。

プロジェクト全体のレベルであれ、個々の関数のレベルであれ、優れたデザインからは多くの恩恵を得られる。プロジェクトのデザインが標準化されていれば、あるプロジェクトから別のプロジェクトに切り替える際の精神的な負担が軽減される。似たようなデザインのプロジェクトを経験していれば、プロジェクトに取り組むのが容易になる。コードがうまくデザインされていれば、その一部を他のプロジェクトで再利用しやすくなり、新しい機能を追加しやすくなる。

データサイエンティストとしての筆者の経験では、すべてのコードが1つの巨大な Jupyter ノートブックにあるプロジェクトを数多く見てきた。筆者自身もこのようなプロジェクトを作成したことがある。Jupyter ノートブックは、プロジェクトを始め、アイデアを下書きし、いろいろ試してみるには素晴らしい環境だ。しかし、プロジェクトの規模が大きくなり複雑になってくると、ノートブックでは限界がある。「8.3 ノートブックからスケーラブルなスクリプトへ」で、ノートブックを Python スクリプトに変えるフレームワークを紹介する。

データサイエンス (DS) では、プロジェクトの構造をいつデザインするべきかを正確に知ることは難しい場合が少なくない。プロジェクトを始めた時点では、プロジェクトの結果がどうなるのか見通せないので。しかし、プロジェクトの方向性が見えてきたときに、時間をかけて意図的にデザインすることには価値がある。そして、これを行うのに最適なのは、ノートブックからスクリプトに切り替えるときだろう。

プロジェクトのデザインを最初からきちんとできるなどとは思わないほうがよい。繰り返し行うことになるだろう。また、プロジェクト進行中にコードの詳細が変わるのは想定内の事態だ。コードの振る舞いはそのままに、コードの構造を調整するプロセスである「リファクタリング」は、ソフトウェアエンジニアリングの通常のプロセスであり、データサイエンティストにとっても非常に有用だ。リファクタリングについては、「8.4 リファクタリング」で詳しく説明する。

## 8.1 プロジェクトのデザインと構造

プロジェクトのデザインや構造について、確固としたルールはない。取り組んでいる対象、ビジネス、プロジェクトのタイムスケール、その他多くの要因に左右される。機械学習（ML）のプロジェクトは、何らかのビジネス上の決定を下すための分析プロジェクトとはまったく異なる構造を必要とするだろう。

とはいっても、本節では、プロジェクトをうまくデザインするためのアイデアをいくつか紹介したい。自分のデザインを（ある程度）標準化しておくことには利点も多い。新たにプロジェクトに従事する人が、何を期待されているのか想像がつくようになり、認知的負荷が軽減される。また、プロジェクトの標準的なデザインを共有していれば、チーム内の誰かが行った仕事を引き継ぐのも簡単になる。

本節では、プロジェクトを全体的なレベルでどのように構造化するか、ファイルやフォルダをどのように配置するかについて説明する。ML プロジェクトの例を挙げ、指定された構造を持つ新しいプロジェクトを自動的に作成するためのいくつかのツールについて説明する。

### 8.1.1 プロジェクトデザインの考慮事項

プロジェクト全体の計画をどのようにして構造化されたコードに落とし込めばよいのか。場合によっては、構造を決める前に、プロジェクトを開始し、プロジェクト全体の方向性を決めるための初期のアイデアを試す必要があるかもしれない。そのような場合は、できるだけ早い段階でプロジェクトのプランニングに時間を費やす価値がある。こういったステップは、Jupyter ノートブックに書かれた大量のコードを、きちんと構造化されたプロジェクトに昇華させるのに役立つことになろう。

プロジェクトのデザインに役立つであろう「問い合わせ」を列挙してみよう。

- プロジェクトの全体的な目標や目的は何か？ 解決すべき全体的な問題は何

か？

あまり重要でない細部に关心を向けるのではなく、これから書くすべてのコードがこのゴールに到達する助けとなるよう、常に全体像を念頭に置くべきである

- プロジェクトの全体的な入力と出力は何か？

例えば、ML プロジェクトでは、生データを入力とし、モデル予測を出力する

- プロジェクトのステークホルダー（利害関係者）は誰か？ ステークホルダーに確認する前に、どれだけの仕事をする必要があるのか？

例えば、プロジェクトマネージャー、他のチーム、あるいは製品の顧客まで対象範囲に含める必要があるのか

- プロジェクトのスケジュールは？ マイルストーンは何で、締め切りはいつか？ これを知ることで、このプロジェクトが、すぐに結果を出して、その後にコードを整理する必要があるプロジェクトなのか、それとも、前もってしっかりとコードを書いて、デザインに時間をかけられるプロジェクトなのかを見極めることができる

- 試してみたいアイデアは何か？ 解決すべき問題の解決に対して、探索や実験にどれだけの時間を費やすか？

探索に重きを置くのであれば、この段階からどのようにコードを再利用できるかを考えることが有用であるかもしれない。ML の実験を標準的な方法でセットアップして、簡単に多くの実験を行えるようになる

- このプロジェクトの「スコープ（対象とする範囲）」が変わる可能性は？ あるいは、将来のある時点で、このプロジェクトで行った作業を別のユースケースに再利用する可能性はあるか？

コードの再利用を想定しているのであれば、コードをモジュール化し、きちんと文書化することで、モジュールを別の場所で簡単に使えるようにしておく必要がある

こうした問い合わせに対する回答は、プロジェクトを理にかなったステップに分けるのに役立つ。回答を記録しておいていつでも確認できるようにしておくとよいだろう。また、過去のプロジェクトを振り返ってみることもお勧めする。共通のテーマを探し、何がうまくいき、何がうまくいかなかったのかを自問する。次に同じようなプロジェクトをやるときに、もっとうまくやれることは何か？ このような問い合わせへの答えが、今後のプロジェクトを組み立てていく際の参考になる。

プロジェクトの「構造」としてはデータも含まれる。しかし、データが中央のデータベース（自分が管理しているデータベースか、属する組織が管理しているデータストレージ）に保存されているのであれば、それを複製してローカルに保存すべきではない。この重複は、データのバージョンが異なると結果が異なるという問題を後で引き起こす可能性がある。また、セキュリティやプライバシーのリスクにもなりかねない。プロジェクトの一部として、データを抽出して選択するコードを含めるほうがはるかに好ましい。少なくとも、プロジェクトを再現できるようなドキュメントを含めるべきだ（良いドキュメントの書き方については9章で説明する）。

同じようなプロジェクトが頻繁に発生するのであれば、Cookiecutter (<https://oreil.ly/bX-rV>) を使ってファイルとフォルダ構造の作成を自動化できる。10章で詳しく説明するが、これにはPythonパッケージのビルトや依存関係の管理に必要なファイルを含める。

プロジェクトの構造をゼロからセットアップしたくない場合は、DSプロジェクトを標準化するためにデザインされたオープンソースライブラリであるKedro (<https://oreil.ly/fJEqk>) を使うのがお勧めだ。プロジェクトを「接続されたノードの集まり」としてセットアップし、パイプラインとして始点から終点まで実行することができる。各ノードには、例えばデータのロードやデータのクリーニングなどのスクリプトを含める。

## 8.1.2 機械学習プロジェクトの例

プロジェクトに必要なステップが明らかになったら、ファイル構造のセットアップを始める。1つのMLプロジェクトの例を見てみよう。ファイル構造のセットアップ方法には多くの選択肢があるが、1つの戦略は、プロジェクトのステップごとに1つのファイルを使うことだ。各ステップは、データの保存で終わることが多い。

ステップには通常、次のようなものがある。

1. データのロード
2. データのクリーンアップと前処理を行い、MLに適した特徴量に変換
3. モデルのトレーニング
4. 検証データセットを使ったモデルの性能分析

初期化モデルの選択のために、いくつかの実験を行って、コードを再現可能なスクリプトにする予定ならば、次のようなデザインがよいかかもしれない。

```

├── README.md
├── requirements.txt

└── notebooks
    ├── explore_data.ipynb
    └── try_regression_model.ipynb

└── src
    ├── __init__.py
    ├── load_data.py
    ├── feature_engineering.py
    ├── model_training.py
    ├── model_analysis.py
    └── utils.py

└── tests
    ├── test_load_data.py
    ├── test_feature_engineering.py
    ├── test_model_training.py
    ├── test_model_analysis.py
    └── test_utils.py

```

細かく見ていく。まず、すべてのプロジェクトが持つべき標準的なファイルがいくつかある。

```

├── README.md
└── requirements.txt

```

すべてのプロジェクトは（9章で詳しく説明するように）ドキュメントを持つべきである。`README.md` ファイルが標準的な形式だ。依存関係やパッケージングを管理するために必要なファイルもすべて含める必要がある（詳しくは10章を参照）。

次に、さまざまなモデルを試しているノートブックや実験は、専用のフォルダに入れておくとよい。

```

└── notebooks
    ├── explore_data.ipynb
    └── try_regression_model.ipynb

```

スクリプトも独自のフォルダを持つ。Pythonでは、パッケージのすべてのコードを `src` (source の略) という名前のフォルダに入れるのが一般的な慣例で、多くの人がプロジェクトのスクリプトにも同じ構造を使っている。多くの場所で再利用される「ヘルパー関数」を `utils` ファイルに置くと便利だ。

```

|   └── src
|       ├── __init__.py
|       ├── load_data.py
|       ├── feature_engineering.py
|       ├── model_training.py
|       ├── model_analysis.py
|       └── utils.py

```

テストを別のフォルダに保存しておくと、整理しやすいし、7章で説明したように、テストフレームワークが簡単にテストを見つける。src フォルダ内の各ファイルに対して、それぞれ独自のテストを作成する。

```

└── tests
    ├── test_load_data.py
    ├── test_feature_engineering.py
    ├── test_model_training.py
    ├── test_model_analysis.py
    └── test_utils.py

```

これはプロジェクトを構成するための1つの選択肢にすぎない。多くのプロジェクトで機能するプロジェクト構造ができたら、それをテンプレート化して Cookiecutter を使って複製することもできるし、Kedro を使ってセットアップしてもよい。ML プロジェクトの構造化については、Catherine Nelson + Hannis Hapke 著『[入門 機械学習パイプライン](#)』（オライリー・ジャパン）で詳しく説明されている。

## 8.2 コードのデザイン

デザイン  
アート  
設計とはコードをアレンジする芸術である

— Sandi Metz

コードデザイン、つまりコードを関数、クラス、モジュールなど適切な部分に「分解」することは容易ではない。Sandi Metz が言うように、それは「サイエンス」というよりは「匠の技」である。しかし本節では、コードをデザインするためのガイドラインをいくつか紹介しよう。

コードを「異なる目的を遂行する部分」に分割するのは良い習慣だ。そうした部分はそれぞれ、異なるアイデアやロジックの一部を表現している。データをクリーンにするコードと、視覚表現を生成するコードは分けるべきだ。Web サイトのフロントエンドのコードは、データを記憶するデータベースのコードとは別にする。

初期のコードデザインが正しいとは限らないことは想定しておくべきだ。プロジェクトが発展するにつれて状況は変わるだろうし、ビジネス上の課題が変わるかもしれない。「8.4 リファクタリング」では、エラーの発生リスクを減らすコード変更のテクニックについて説明する。

## 8.2.1 モジュラーコード

コードの全体的なアーキテクチャを考えるとき、関数とクラスに分けることは重要なステップだ。それがプロジェクトの最初の段階であろうと、いくつかの初期アイデアを試している段階であろうと、コードのリファクタリング中であろうと関係ない。どんな関数やクラスが適切かをいつも考えているのは好ましい習慣だ。このトピックについては1章と4章で説明したが、ここで再確認しておきたい。

**モジュラーコード**（モジュール化されたコード）とは、コードが独立した小さなパートに分割されていることを意味する。プロジェクトのすべてのコードを1つの巨大なスクリプトにまとめるのとは正反対の方向だ。モジュール化されたコードで作業するのはずっと簡単だが、コードをどのように分割するかを考えるのは難しいことが多い。本節と次節で、その方法をいくつか紹介しよう。

可能な限り、各関数やクラスには明確に定義された目的を1つ持たせること。その関数が行う1つの「こと」を考える。例えば、「この関数はデータの視覚化を担当する」というように、短い文章で説明できるものでなければならない。説明時に「および」「そして」といった単語を使い始めたら、それは複数の関数に分解するべきときかもしれない。例えば、「この関数はデータのクリーニングおよび視覚化を行う」と記述する場合、データをクリーニングする関数と視覚化を担当する関数に分けたほうがよい。

関数の「カップリング」は避けるようにしよう。ある場所でコードを変更すると、別の場所でも何かを変更する必要が出てくる。これについては「8.2.4 結合度」で詳しく説明する。

モジュラーコードを書くことには、多くの利点がある。モジュール化されたコードは他の場所での再利用が容易であり、コードの要件が変わったときにコードの断片を移動させるのも簡単だ。コードの断片をさまざまな方法でつなげて、より大きなシステムを作ることもできる。「コンポーザビリティ（構成可能性）」が高まるわけだ。さらには、モジュラーコードはテストも容易にできる。

どのような関数やクラスが、解決しようとしている問題にふさわしいかを考えるために、次の問い合わせてみよう。

- 書いているコードに、どのような共通パターンがあるだろうか？  
例えば、データ変換やビジネスロジックの一部など
- 多くの場所で再利用できるコンポーネントは何か？  
例えば、あるタイムゾーンから別のタイムゾーンへのデータ変換など
- 書こうとしている関数の目的は何か？  
可能な限り、それぞれの関数やメソッドは1つのことだけを行うべき

こうした問い合わせを発することで、より構造化されたフレームワークを目指そう。

## 8.2.2 コードデザインの枠組み

関数、クラス、メソッドを構築するための指針を挙げよう—— Rachel Tatmanによる Kaggle に関する記事「Six steps to more professional data science code」(<https://oreil.ly/Psa1F>) からの引用だ。

### 関数名

関数の名前を決めるところから始めよう。良い名前の決め方については9章を参考してほしいが、名前を付けることで、その関数に何をさせたいかという意図が明らかになる

### 入力

関数のインターフェースは一貫していなければならないので、関数本体を書き始める前に、関数への入力を決めておくとよいだろう。関数の引数にそれを指定する。インターフェースについては次節で詳しく説明する

### ビヘイビア

関数やメソッドの本体には、その関数が実行する実際の処理が含まれる。これが関数の「ビヘイビア」であり、実際にやってもらいたいことである。前述したように、各関数が行うこととは1つだけにしておくのがよい

### 出力

関数の出力は、`return` 文を使って返すものであったり、ファイルに保存するデータであったりする。出力は関数のインターフェースの一部でもあるので、関数を書く前に出力についてよく考え、できるだけ変えないようにする

この指針に従ってノートブックからモジュラーコードへ移行する方法の例を「8.3

ノートブックからスケーラブルなスクリプトへ」で取り上げる。

### 8.2.3 インタフェースと契約

モジュラーコードを構築する上で最重要といえるのが、システム内のコンポーネント間のインターフェースである。ある意味、個々の関数を記述する際の出発点ともなる事柄だ。入力として何を受け入れ、出力として何を返すのか、というところから始めるのだ。

次の関数は、1章から何度か登場したものだが、CSV ファイルと、落とす（drop する。削除する）列のリストを入力として受け取り、pandas の DataFrame を出力として返す。

```
def process_sdg_data(excel_file, columns_to_drop):
    df = pd.read_excel(excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("GeoAreaName").transpose()
    return df
```

一度入力と出力を決めたら、それを変更すべきではない。なぜなら、システムの他のコンポーネントがそれに依存しているかもしれないからだ。これは「契約 (contract)」とも呼ばれる。入力引数の数は、せいぜい 3 つか 4 つ程度と、少なめにしておくのがよい。これ以上の入力が必要な場合は、代わりに構成ファイル（コンフィギュレーションファイル）を使うことを検討しよう。

関数のテストでは、正しい入力が受け付けられ、正しい出力が返されることをチェックすることで、「契約」が維持されていることを確認しなければならない。また、「9.1.3 docstring」で説明するように、関数の docstring で入力と出力を文書化する必要がある。さらに、「6.3 型チェック」で見た、型注釈を使うことで、契約を確実に遵守できる。じゆんしゆ 型チェックツールは、型が不一致のために契約が破られたときにメッセージを出してくれる。

### 8.2.4 結合度

コードを分割する場合、その分割されたコードが可能な限り互いに独立していることが重要だ。独立していないと、コードのある部分を変更すると、別の部分の変更が必要になり、プロジェクト全体の複雑さが増し、作業が難しくなる。**結合度**（カップリング）は、関数やモジュール間の依存関係の強さを表す用語だ。2つの関数が密結

合であれば、一方を変更すると、もう一方も（広範囲に）変更する必要がある。疎結合であれば、一方を変更してももう一方は、あまり（あるいはまったく）変更する必要がない。できるだけ結合度を低くすべきである。

結合度の高い（密結合している）2つの関数の例を挙げよう。

```
import pandas as pd
from scipy.stats import linregress

def process_sdg_data(input_excel_file, columns_to_drop):
    df = pd.read_excel(input_excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("GeoAreaName").transpose()
    return df

def fit_trendline(country_name):
    df = process_sdg_data(
        "SG_GEN_PARL.xlsx",
        [
            "Goal",
            "Target",
            "Indicator",
            "SeriesCode",
            "SeriesDescription",
            "GeoAreaCode",
            "Reporting Type",
            "Sex",
            "Units",
        ],
    )
    timestamps = [int(i) for i in df.index.tolist()]
    data = df[country_name].tolist()

    result = linregress(timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared
```

`fit_trendline` 関数は内部的に `process_sdg_data` 関数を呼び出し、データ処理を実行する。`process_sdg_data` の振る舞いを変更すると、`fit_trendline` の振る舞いに影響を与える可能性がある。このような依存関係のある関数を使うと困難さが増すことになる。

これらの関数が密結合でない（疎結合な）例を挙げよう。

```
import pandas as pd
from scipy.stats import linregress

def fit_trendline(year_timestamps, data):
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared

def process_sdg_data(input_excel_file, columns_to_drop):
    df = pd.read_excel(input_excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("GeoAreaName").transpose()
    return df

df = process_sdg_data(
    "SG_GEN_PARL.xlsx",
    [
        "Goal",
        "Target",
        "Indicator",
        "SeriesCode",
        "SeriesDescription",
        "GeoAreaCode",
        "Reporting Type",
        "Sex",
        "Units",
    ],
)
timestamps = [int(i) for i in df.index.tolist()]
country_data = df["India"].tolist()
slope, r_squared = fit_trendline(timestamps, country_data)
```

このように関数を分離して、`fit_trendline` と `process_sdg_data` の依存関係をなくすことで、この 2 つの関数を独立して扱えるようになり、他の場面での再利用も容易になる。

コードデザインについて詳しく知りたいなら「SOLID 原則」について調べることを勧める。Real Python の記事 (<https://oreil.ly/oZN0y>) から始めるとよいだろう。デザインパターンについて学ぶのも良い選択肢だが、データサイエンスに適用するためには多少のアレンジが必要かもしれない。「The Refactoring Guru」(<https://oreil.ly/HEV6u>) に Python のデザインパターンに関する有用なガイドがある。

次節では、Jupyter ノートブックからスタンドアローンの Python スクリプトにコードを移行する方法について説明する。

## 8.3 ノートブックからスケーラブルなスクリプトへ

Jupyter ノートブックがプロジェクトのワークフローに組み込まれている場合、ある時点で、繰り返し実行する必要のある Python スクリプトに移行することになるだろう。これはプロジェクトにおいて重要な段階になる。プロトタイピングから、よりエンジニアリング的な考え方にも続くコーディングへの移行だ。

本節では、ノートブックからスクリプトに移行すると便利な理由とともに、その方法についても詳しく説明する。

### 8.3.1 なぜノートブックの代わりにスクリプトを使うのか

Jupyter ノートブックは、データサイエンティストにとって非常に便利なツールだ。柔軟性があり、すぐにフィードバックが得られ、アイデアを自由に試すことができる。オンラインでの可視化は非常に便利だ。コードとドキュメントを混在させたチュートリアルを書くのにも最適だ。さらには、Papermill (<https://oreil.ly/L8Zvj>) を使って自動化することも可能だ。

しかし、ノートブックにコードを書くことには欠点も多い。書いたときと違う順番でコードを実行でき、ノートブックにあるすべてのコードを一度に実行する必要がないため、実際に実行したコードがノートブックに反映されるとは限らない。そのため、実行した手順を再現するのが難しくなることがある。

ノートブックはまた、本書で説明する多くの標準的なソフトウェアエンジニアリングのツールやプラクティスとも相性がよくない。ノートブックに対するlinting、フォーマッティング、型チェックは、Python スクリプトのように簡単にはできない。依存関係を管理するのが難しく（これについては10章で説明する）、したがってノートブックで使われているサードパーティのライブラリのバージョンが簡単にわからぬこともある。

また、Jupyter ノートブックや JupyterLab で作業している場合、ノートブックをデバッグするのは簡単ではない（もっとも、最近ではIDEにノートブックで動作するデバッガーが登場したのでそれは利用できる）。バージョン管理システム（VCS。10章参照）では、ノートブックでどのセルが実行されたかを把握できない。ノートブックで行われた変更のレビューを簡単に行うには、外部のツールに頼る必要があ

る。また、データを含むノートブックを VCS にアップロードするのは簡単すぎて、セキュリティリスクとなりかねない。

(筆者も含めて) 多くの人がノートブックにはラフなスタイルでコードを書きがちで、データがどのように見えるかをチェックしたい余分なセルを含むことが多い。そのため、モジュール化されたコードを書くのが難しくなる。しかし、モジュール化された関数でコードを書いたとしても、あるノートブックから別のノートブックに関数をインポートするのは簡単ではない。モジュール化された Python ファイルのほうが、多くの場面で作業しやすいのだ。

これは間違いなく議論の余地のある問題だが、筆者個人の戦略は次のようなものだ。

- データ探索やプロトタイピングにはノートブックを使い、同じことを繰り返すときにはスクリプトに切り替える

作成した再利用可能なコンポーネントを Python ファイルにまとめ、それをノートブックにインポートして実行することもある。スクリプトは、機械学習 (ML) の実験を行うときにも便利だ。なぜなら、自分の実験が再現可能であることを確認するテストを実行したり、自分の実験が期待どおりに行われていることをテストしたりできるからだ。

一般的には、ノートブックからスクリプトに切り替えるのは、コードを本番環境に移行し、繰り返し実行する時点になることが多いだろう。組織によっては、ノートブックによる潜在的なセキュリティリスク（前述したようにノートブックにデータを保存するなど）のために、これを必要とする場合もある。

ノートブックからスクリプトに切り替えたいと思ったらどうするべきか。次節でその方法を説明しよう。

### 8.3.2 ノートブックからのスクリプト作成

ノートブックから関数を作成する前に、まずはノートブックが正しく動作することの確認をするべきだ。さらに、きちんと構造化されたノートブックから始めると、スクリプトにするのが簡単になる。つまり、スクリプトにしようとする前に、ノートブックの「整理整頓」に時間をかける価値がある。

次に、ノートブックを 1 つの Python スクリプトにするか、複数のスクリプトにするかを考える。本章で見てきたように、各スクリプトはプロジェクトを構成するステップであるという考え方もある。プロジェクトの多くのステップで利用するデータ

クリーニングなどのユーティリティ関数は、別のファイルにしてもよいだろう。ノートブックの途中から再実行することが多い場合は、その箇所でノートブックの分割を検討するのがよいだろう。

次に、常に一緒に実行したいセルや、ノートブックを実行するたびにいつも実行したいセルはあるだろうか？ そういったものは関数の候補となる。ノートブックでよく操作するオブジェクトは何か？ そういったものはクラスに抽象化する候補になる。

このプロセスの例を示そう。図8-1は、国連の持続可能な開発目標のWebサイトからデータを探索し、クリーニングするためのノートブックである。このコードの一部は本書の序盤で紹介したが、開発中のものはこのような形になる。

```
In [1]: import pandas as pd
In [2]: df = pd.read_excel('SG_GEN_PARL.xlsx')
In [3]: df.head(2)
Out[3]:
   Goal  Target  Indicator  SeriesCode  SeriesDescription  GeoAreaCode
0      5      5.5       5.5.1  SG_GEN_PARL  Proportion of seats held by women in national ...
1      5      5.5       5.5.1  SG_GEN_PARL  Proportion of seats held by women in national ...
2 rows x 34 columns

In [4]: df.columns
Out[4]: Index(['Goal', 'Target', 'Indicator', 'SeriesCode', 'SeriesDescription',
               'GeoAreaCode', 'GeoAreaName', 'Reporting Type', 'Sex', 'Units',
               '2000', '2001', '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009',
               '2010', '2011', '2012', '2013', '2014', '2015', '2016', '2017', '2018',
               '2019', '2020', '2021', '2022', '2023'],
              dtype='object')
In [5]: df = df.drop(['Goal', 'Target', 'Indicator', 'SeriesCode', 'SeriesDescription',
                   'GeoAreaCode', 'Reporting Type', 'Sex', 'Units'], axis=1)
In [6]: df = df.set_index('GeoAreaName').transpose()
In [7]: df.head(2)
Out[7]:
   GeoAreaName  Afghanistan  Albania  Algeria  Andorra  Angola  Antigua and Barbuda  Argentina  Armenia
2000          NaN        5.16     3.16     7.14    15.45     5.26      28.02     3.05
2001          NaN        5.16     3.42     7.14    15.45     5.26      26.46     3.05
2 rows x 195 columns

In [ ]: df.to_csv('women_in_parliament_processed.csv')
```

図8-1 典型的なノートブック

**図8-1** を見ればわかるように、さまざまな段階で DataFrame の中身を表示し、自分のデータがどのようなものかを確認できるようにしている。

この部分では、Excel のスプレッドシートファイルからデータをインポートし、CSV ファイルとして出力している。これが繰り返されるなら、関数化の絶好の候補だ。

単にデータの値を表示するための行ではなく、何らかの操作を実行する行を抽出する。

```
df = pd.read_excel("SG_GEN_PARL.xlsx")
df = df.drop(["Goal", "Target", "Indicator", "SeriesCode", "SeriesDescription",
              "GeoAreaCode", "Reporting Type", "Sex", "Units",], axis=1)
df = df.set_index("GeoAreaName").transpose()
df.to_csv("women_in_parliament_processed.csv")
```

筆者は、こうした行を関数にするために、まず全体的な目的や振る舞いを考えた。この関数は、Excel ファイルを取り込み、データを並べ替え、CSV ファイルを出力する必要がある。この関数の名前は `process_sdg_data` とした。国連の持続可能な開発目標 (SDGs) からダウンロードした類似の Excel ファイルすべてに対して動作させたいと考えたからだ（このため `sdg` をファイル名に含めた）。略語を使う場合は、次章で説明するように、どこかに文書化しておくとよいだろう（名前の選択に関するベストプラクティスは「9.1.1 ネーミング」に詳しく解説した）。

続いて、関数の入力と出力がどうあるべきかを考えた。この関数は Excel ファイルを入力とし、CSV ファイルを出力にした。また、将来変更したくなることを考え、入力も出力も関数の引数にした。例えば、入力ファイル名や出力ファイル名は変更の可能性があるので、関数の引数とした。

最後に、引数を変数として追加し、関数本体を書いた。

```
def process_sdg_data(input_excel_file, columns_to_drop,
                      output_csv_file):
    df = pd.read_excel(input_excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("GeoAreaName").transpose()
    df.to_csv(output_csv_file)
```

次のステップで、この関数にテストを追加する。データを表示したノートブックの行から、何をテストすべきかのアイデアが浮かんだ。例えば、出力データが正しい列を持つことを確認したかった。出力ファイルの行数と列数は、どの入力ファイルを処

理しても同じであるべきだということはわかっているので、これは良いテスト項目になる。

次が `process_sdg_data` 関数のテストだ。

```
import os
import pandas as pd

def test_process_sdg_data():

    test_filepath = "test_sdg_data.csv"

    process_sdg_data("SG_GEN_PARL.xlsx",
                      ["Goal", "Target", "Indicator", "SeriesCode",
                       "SeriesDescription", "GeoAreaCode", "Reporting Type",
                       "Sex", "Units"],
                      test_filepath)

    df = pd.read_csv(test_filepath)

    assert len(df) == 24
    assert len(df.columns) == 196

    # クリーンアップのステップ - このテストで作成されたファイルを削除
    os.remove(test_filepath)
```

データを表示するステップは、テストの `assert` 文に変わることになる。

ノートブックからスクリプトへの変換を支援するツールも開発されている。`nbconvert` (<https://oreil.ly/AiWYo>) や `Jupytext` (<https://oreil.ly/Yn5RA>) はノートブックのコードからスクリプトを抽出するシンプルなパッケージだ。また、`Kedro` のドキュメント (<https://oreil.ly/lsx01>) にある手順に従って、ノートブックから直接 `Kedro` プロジェクトに移行もできる。

## 8.4 リファクタリング

リファクタリングは、ソフトウェア開発のワークフローに当然入るべき工程の1つだ。次の説明が役に立つだろう。

コードのリファクタリングとは、コードの外部的な振る舞いは変えず、内部構造を改善する目的でソフトウェアシステムを変更することである。

—— Martin Fowler

リファクタリングは、必然的な理由がある場合にのみ行うべきだ。コードが正しく動作し、すべての要件を満たしているなら、リファクタリングする必要はない。リファクタリングの際には何を解決しようとしているのかを意識するべきだ。追加したい新しい関数があるのか？ コードをすっきりさせて、理解や保守を容易にしたいのか？ 本節では、コードをリファクタリングするためのいくつかの戦略と、その方法について例を挙げて説明する。

### 8.4.1 リファクタリングの戦略

理想的には、コードのリファクタリングをする前に、そのコードのテスト一式の準備を済ませておく。テストによって、コードを変更してもその機能が変わらないことをチェックできる。コードのリファクタリングでバグは発生させたくない。

コードは少しづつ変更していくのが望ましい。一度にすべてを書き直したり、うまくいっているコードの一部を捨てたりしないことだ。特に複雑なプロジェクトに取り組んでいる場合は、最初からすべてがうまくできるわけではない。あらかじめマイルストーンを決めておいて、段階的にリファクタリングするのが良いやり方だ。

全体的な戦略は単純だ。小さな変更を加え、テストを実行し、コードを保存する（通常は 10 章で説明するように VCS に保存する）。変数の名前を変える程度の簡単な変更でも無駄ではない。変数の使い方を少し変えたり、読みやすくしたりするのも意味がある。IDE を使えばこの作業が簡単になる（次節の図 8-2 を参照）。また、VCS も非常に便利で、変更によってテストが失敗した場合に、前のバージョンのコードに簡単に戻せる。

リファクタリングの際には、関数の「インターフェース」はそのままにして、内部を変更する。こうすることで、コードの他の部分の変更を最小限にできる。インターフェースや関数の振る舞いを変更する場合は、より多くのコードをテストする統合テストが必要になる。システムの他の部分が依存している事柄を変更した場合には、全体で不具合が生じていないかをチェックする必要がある。

全体として、リファクタリングが成功するかどうかは、良いテストセットがあるかどうかに大きく依存する。次節では、関数をリファクタリングする方法の例を示す。

### 8.4.2 リファクタリングの例

前章で見たように、ここで的一般的な原則は次のとおりだ。

1. 変更を行う
2. テストを実行して何も壊れていないことを確認する
3. 変更を保存する（通常は VCS にコミットする）

本書で何度か見てきた重み付きの平均を求める関数を使って、このワークフローを説明しよう。

まず、もう一度、関数 `weighted_mean` を確認しよう (`weighted_mean.py`)。

```
def weighted_mean(num_list, weights):
    if not (num_list or weights):
        return None
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights[i])
    return (running_total/sum(weights))
```

そして、これがそのテストだ (`test_weighted_mean.py`)。

```
def test_weighted_mean():
    result = weighted_mean([1, 2, 4], [1, 2, 4])
    assert result == 3

    empty_list_result = weighted_mean([], [])
    assert not empty_list_result
```

まず、リファクタリングを始める前にテストを実行し、すべてが正しく機能していることを確認する。

```
$ pytest test_weighted_mean.py
=====
platform darwin -- Python 3.11.5, pytest-8.3.5, pluggy-1.6.0
rootdir: /Users/.../example/ch08
plugins: anyio-4.3.0, Faker-37.1.0, typeguard-4.4.2
collected 1 item

test_weighted_mean.py . [100%]

=====
1 passed in 0.02s =====
```

次に、ちょっとした変更を加える。例えば、変数名を変更するなど、コードの読みやすさを向上させるような変更を加えたいかもしれない。図8-2に示すように、IDE

を使って簡単に行うことができる。

```

weighted_mean.py > weighted_mean > weights
1 def weighted_mean(num_list, weights):
2     if not (num_list or weights):
3         return None
4     running_total = 0
5     for i in range(len(num_list)):
6         running_total += (num_list[i] * weights[i])
7     return (running_total/sum(weights))
8

```

図8-2 VS Code を使ったリファクタリング

これにより、この関数内の `weights` がすべて `weights_list` に変更される。

```

def weighted_mean(num_list, weights_list):
    if not (num_list or weights_list):
        return None
    running_total = 0
    for i in range(len(num_list)):
        running_total += (num_list[i] * weights_list[i])
    return (running_total/sum(weights_list))

```

テストを再度実行し、コードが正しく動作することを確認する。

より大規模な変更についても同じワークフローを使えばよい。例として、`weighted_mean` に少し大きな変更を加えてみよう。NumPy ライブラリの組み込み関数に置き換えて、関数の内部本体を変更してみる。

```

import numpy as np

def weighted_mean(num_list, weights_list):
    try:
        return np.average(num_list, weights=weights_list)
    except ZeroDivisionError:
        return None

```

再度テストを実行すると、関数の全体的な振る舞いは変わっておらず、関数本体が大きく変わっているにもかかわらず、インターフェースも変わっていないことが確認できる。

このワークフローは、より大規模で複雑な状況にも対応できる。進めていくうちに、テストの更新も必要なことに気がつくかもしれない。リファクタリングする過程で、テストケースを追加することは珍しくはない。

## 8.5 まとめ

プロジェクトが作業中に大きく形を変えることがあったとしても、プロジェクトに取りかかる前、作業中、そして完成した後でも、コードのデザインについて考えることは重要だ。よくデザインされたプロジェクトは、他人にとっても作業しやすく、数か月、あるいは数年ぶりにプロジェクトに戻ったとしても、容易に作業に取りかかることができる。

プロジェクトを標準化できるかどうかも検討すべきである。同じようなプロジェクトに繰り返し取り組んでいるのであれば、時間をかけて繰り返し使えるテンプレートの開発を検討しよう。その際に、Cookiecutter や Kedro といったツールを使って、プロジェクトのセットアップを自動化できないかも検討しよう。

優れたデザインには、モジュール化されたコードが欠かせない。将来、コードを再利用したり変更したりしやすいように、コードを分割する方法をよく考えよう。関数を書き始める前に、その目的とインターフェースのあり方を検討し、インターフェースを設定した後で、実際にその目的を遂行するコードだけを追加する。

Jupyter ノートブックは実験には最適だが、コードが本番環境にデプロイされるようになったら、スクリプトへの移行を検討すべきだ。コードのどの部分を他の場所で再利用できる関数にすべきかを検討し、モジュール化されたコードを書くための指針に従おう。

さまざまな理由でコードの変更が必要になる。コードの要件が変わったり、より読みやすくしたり、より効率的にしたりといった理由からだ。コードのリファクタリングは、どのようなプロジェクトにもつきものだ。リファクタリングのワークフローがうまくいかどうかは、よいテストセットがあるかどうかに大きく依存する。コードのリファクタリングを行うには、小さな変更を行い、テストを実行して、変更が破壊的でないことを確認し、保存されているコードを更新する。

コードのデザインは、本番で使えるコードを書くための重要な要素だが、自分のコードが何をするのか、関係者とコミュニケーションをとる必要もある。次章では、ドキュメンテーションに関するベストプラクティスを紹介する。

# 9章 ドキュメンテーション

ドキュメンテーションは、データサイエンス（DS）において見落とされがちな側面である。プロジェクトの終わりまで放置されるのが一般的だが、その後新しいプロジェクトに移ることになり、ドキュメンテーションは軽視されたり、完全に省略されたりする。しかし、「1.5 可読性」で述べたようにドキュメンテーションはコードを再現可能にするための重要な部分である。他人に自分のコードを使ってもらいたい、あるいは将来自分のコードに戻ってきたいと思うのであれば、良いドキュメントが必要だ。最初にコードを書いたとき、あるいは最初に実験を行ったときに何を考えたかをすべて覚えておくことは不可能なので、記録を残しておく必要がある。

良い文書とは、考えをうまく伝えるものである。「何を理解してほしいと思っているか」を理解する必要がある。したがって、まず誰のためにドキュメントを書くのかを考えることが重要だ。将来、プロジェクトを引き継ぐかもしれない他のデータサイエンティストのために実験を記録しているのか。チーム内の他の人に役立つかかもしれないコードの一部を文書化するのか。それとも半年後に見返せるように自分の考えを記録しているのか。想定する読者にとって適切なものになるように、詳細レベルや使用する言葉を選ぼう。

優れたドキュメンテーションは、最新である必要がある。ドキュメントは、コードの変更と一緒に更新されるべきである。できるだけ簡単にドキュメントを更新できること。例えば、ログインが必要な独自のドキュメント作成ツールは使わないこと。

優れたドキュメントは構造化されていなければならない。最も重要な情報は最も簡単に発見できなければならない。したがって、それを冒頭に置くか、どこにあるかがすぐにわかるようにすべきだ。

優れたドキュメンテーションは、膨大な時間を節約し、DS プロジェクトの複雑さ

を軽減してくれる。すでにどのような作業が行われたかを知っていれば、同じ作業を繰り返す可能性を減らせる。また、新しいプロジェクトに素早く取りかかったり、1年前に取り組んでいたことを簡単に思い出したりもできる。

これまで述べてきたことはすべて、すべてのプロジェクトの文書化に共通の事柄だが、いくつかのDSプロジェクト特有の留意点もある。DSプロジェクトでは、ある問題に対していくつかのソリューションを試してから、最もうまくいくソリューションに落ち着くのが一般的である。このため、実験や意思決定に至る思考プロセスを文書化することは良い習慣である。後で質問されたり、将来プロジェクトを再検討する際などに役立つ。

次のような問い合わせられる文書にしよう。

- このプロジェクトで使用したデータをなぜ選んだのか
- データについてどのような仮定を置いているのか
- なぜ他の分析方法ではなく、この分析方法を選んだのか
- この分析方法が機能しない状況はあるのか
- 改善できるようなショートカット（近道）はあるだろうか
- 今後、このプロジェクトに携わる人に提案したい、実験の道筋はほかにあるか
- このプロジェクトから学んだことは何か

本章では、さまざまなタイプのドキュメントと、その書き方のベストプラクティスを説明する。コードベース内のドキュメント、Jupyterノートブック内のドキュメント、機械学習実験のドキュメントについて説明する。

## 9.1 コードベース内のドキュメンテーション

「1章 良いコードとは何か？」で説明したように、良いコードは読みやすい。読みやすいコードベースには、コードだけでなく、コメントやdocstringなどのテキストも含まれていなければならない。コードそのものも読みやすくあるべきで、良いネーミング名前付けがその鍵になる。本節では、これらすべてのタイプのドキュメントについて、ベストプラクティスを提案する。

コードベース内のドキュメントの階層を、短いものから長いものへと並べてみた。

- 関数、クラス、モジュールのネーミングは、そのコードが何をすることを期待

されているかという情報を与える

- コメントは、本の脚注のようなもので、補足情報や細かな指摘を加えるために使う
- docstring は、関数やクラスが何をするのか、エッジケースの詳細も含めて、より長い概要を示す
- API ドキュメントは、各 API のエンドポイントが入力として何を期待し、出力として何を返すかを示している（API ドキュメントの詳細は 11 章で説明する）
- README やチュートリアルのような長い文書は、プロジェクト内のすべてのコードの使い方の概要を示す

これらを組み合わせることで、コードはずっと読みやすくなり、他人がコードを使いやすくなる。コードの読者に伝えたい各情報は、一度だけ文書化すべきである。関数が何をするのかの要約は、コメントではなく docstring に書き、プロジェクトの README には重複して書かない。

### 9.1.1 ネーミング

コードを書くたびに、たくさんの名前を付ける必要がある。変数名、関数名、ノートブック名、プロジェクト名。これらすべてに名前が必要だ。良い名前は、コードを読みやすくするための重要な要素だ。他の誰かがあなたのコードを使いたいと思ったら、変更を加える前にそのコードの全体を読むだろう。あなたが使う名前は、あなたがコードに何をさせたいかを伝えるものだ。例えば、`download_and_clean_monthly_data` という名前の関数は、`process_data` という名前の関数よりも多くのことを伝える。Jupyter ノートブックには、決して `untitled1.ipynb` という名前を付けるべきではない。なぜなら、この名前ではノートブックの中身について何も伝えることができない。将来、そのノートブックから何かが必要になったとき、適切な名前がなければそれを見つけられないだろう。

良い名前は、表現力が高く、適切な長さで、読みやすい。また、良い名前には、取り組んでいるプロジェクトや、所属する組織にとって意味のある言葉が使われている。単位を変数名に含めるといい。`distance` よりも、`distance_km` のほうが情報量が多い。最初によい名前を選べなかったとしても、後で更新することを恐れなくてよい。ある名前のすべてを一度に置換ができる。

名前を付ける際に、略語を使うことには慎重になったほうがよい。数文字の節約

は、他人が文脈を理解するための不便や時間の損失に見合わないことが多い。組織全体でよく知られている略語を使うのはかまわないが、それが特殊なものであったり、社内で広く使われていないものであったりする場合は避けることを勧める。

では、適切な名前の長さとはどれくらいだろうか？変数名や関数名は短すぎてはいけない。なぜなら、名前が短すぎると、コードを読む人の精神的負担が増えるからだ。名前から意味に変換しながら読む必要がある。例えば、`im_id` よりも `image_id` のほうが情報量が多いし、`cl_df` よりも `clean_df` のほうがよい。（省略されていない）完全な単語のほうがずっと読みやすいし、後で使用箇所を調べたり、変更したりする必要が生じた場合にも、IDE で検索しやすくなる。

例えば、次のスニペットは、変数に 1 文字が使われているため、何が起こっているかを説明するコメントが必要である。

```
# calculate the accuracy of the predictions compared to the test data
# テストデータと比較して、予測の精度を計算する
a = sum(x == y for x, y in zip(p, t))/len(p)
```

より良い名前を付けることで、コードはずっと読みやすくなる。

```
accuracy = sum(x == y for x, y in zip(predictions, test_data))/len(predictions)
```

変数 `x` と `y` は、`sum()` の呼び出しでのみ使用され、一時的なものであるため、短い名前（短縮名）のままにしてある。同様に、Python の慣例では、次の例のように、`i` と `j` という 1 文字をカウンターとして使う。

```
for i in range(processed_results):
    # do something iについて何かを行う
```

このほか、よく使われる慣例として、pandas の DataFrame の場合は `df`、Matplotlib を使う場合は図と軸をそれぞれ `fig`、`ax` とする。読者がわかるような短縮名を使ってもかまわない。また、`make_...`、`load_...`、`get_...` などのように、動詞で始まる関数名もよく見かける。

名前は似すぎていてはいけない。筆者はよく使われる日時関連の関数 `strptime` と `strftime` の違いがなかなか覚えられず、いつもドキュメントを調べなければならない。コードを使うために追加の知識を頭に入れる必要がある。

さらに、Python の形式規則を使うことで、コード中の名前を読みやすくできる。変数と関数は `snake_case` のような「スネークケース」を使うべきだ。すべての单

語を小文字にし、「\_」（アンダースコア）でつなげる。各単語の間に「\_」があると、変数名が読みやすくなる。`xtrainarray`よりも、`x_train_array`のほうがわかりやすいだろう。クラス定義は CamelCase のような「キャメルケース」を使い、各単語の先頭文字を大文字にする。定数やグローバル変数には ALL\_CAPS のような「大文字（とアンダースコア）のみからなる文字列」を使う。



Python 組み込み関数の名前を変数名として使ってはならない。次に例を示す。

```
list = [0, 2, 4]
```

こうすると、次の行は空のリストの作成ではなく、エラーを返すようになる。

```
empty_list = list()
```

モジュール名やファイル名も非常に重要だ。適切なネーミングにより、特定の機能がどのファイルにあるかが簡単に覚えられる。David Nicholson によるブログ (<https://davidnicho.com/blog/>) の投稿<sup>†1</sup>が推奨しているように、関連するファイルは接頭辞を使って関連することを示すのではなく、サブフォルダを入れる。良いモジュール名は、一般的なものよりも具体的なものである。`data_preprocessing_utils.py`のほうが `utils.py` よりも良い。

PEP 8 (<https://oreil.ly/Hq3Ll>) と Google の Python スタイルガイド (<https://oreil.ly/csW2t>) に、良い名前を付けるためのガイドがある。

## 9.1.2 コメント

コメントはコードベース内で非常に有用なドキュメントの1つだが、うまく使う必要がある。コメントを使うことで、コードの要約、説明、注意書きの付加、要変更箇所への印付けといったことができる。コメントという形で擬似コードを書いてから、実際のコードを埋めていくことで、関数の構造をより単純にできる。

Python のコメントは、「#」で指定される。

```
# This is a comment.
```

コメントは、すでにコードにある情報を繰り返してはならない。これは読者の助けにならないし、コードが変わればコメントも変更する必要がある。「1.3.1 同じことを繰り返さない (DRY)」の DRY 原則にも反する。例えば次は、悪いコメントの

---

<sup>†1</sup> 当該の投稿にはアクセスできない模様。

例だ。

```
# Train the classification model  
classifier.fit(X_train, y_train)
```

良いコメントは、注意事項の付加、情報の要約、コードにない説明の追加などを行うものだ。次は有用なコメントの例である。

```
def slow_way_to_calculate_mode(list_of_numbers):  
    # This is for demo purposes only  
    # Use the mode function from the statistics library in real use cases  
    # あくまでもデモ用。実践では statistics ライブリの mode 関数を使う
```

コメントは読者が理解しやすいものでなければならない。略語ではなく、完全な単語や文を使うのがベストだ。また、後で説明を加えるのではなく、コードを書くのと同時にコメントを書くのも良いアイデアだ。そうすることで、コード作成中に考えた付加的な情報も書ける。

コメントは常にプロフェッショナルであるべきで、攻撃的な言葉やラフな言葉遣いは避けるべきだ。しかし、組織の文化にマッチしているのならば、軽快で楽しいものであってもよいだろう。

コメントは、例えば、次のような事柄を追加するためにも使える。

- GitHub の関連する issue など、外部リソースへのリンク
- Jira などの社内プロジェクト管理システムへのリンク
- 将来的に追加したい機能 (TODO)
- コードが一時的な修正<sup>フィックス</sup>である旨を示す注意書き

コメントは短く、1つのポイントに絞ること。より長い説明が必要な場合は、docstring や長いドキュメントを使う。これについては次節で説明する。

### 9.1.3 docstring

Python の **docstring** は、正式な「長いバージョン」のコメントで、関数やクラスの定義の最初やファイルの先頭に置く。特定の事柄を指摘するコメントとは対照的に、関数やスクリプトが何をするか、全体像を示すものだ。docstring は関数の名前だけでは伝えきれない関数の目的をより詳細に伝えることができるため、コードを読みやすくするための重要な要素となる。

関数の docstring には、その関数に期待される入力と出力を記述する。これは、「8.2.3 インタフェースと契約」で説明した関数のインターフェースである。「6.3 型チェック」で説明した「型注釈」を使っていない場合は、docstring に入力と出力を含めるべきである。docstring は関数を文書化するための Python の標準であり、Python インタプリタで `help` 関数を呼び出すことで、ここに記述した内容を表示できる。さらに、Sphinx (<https://www.sphinx-doc.org/ja/master/>) や pdocs (<https://oreil.ly/OU1nS>) などの自動ドキュメント生成ソリューションは、docstring からテキストを抽出して Web ドキュメントを生成してくれる。

pandas (<https://oreil.ly/oEBEE>) のコードベースにある docstring の素晴らしい例を紹介しよう。`head()` メソッドは、pandas DataFrame の最初の  $n$  行を表示する。docstring は "''' で囲むのが標準的な方法である。

```
def head(self: NDFrameT, n: int = 5) -> NDFrameT:
    """
    Return the first `n` rows. ❶
    This function returns the first `n` rows for the object based
    on position. It is useful for quickly testing if your object
    has the right type of data in it.
    For negative values of `n`, this function returns all rows except
    the last `|n|` rows, equivalent to ``df[:n]``. ❷
    If n is larger than the number of rows, this function returns all rows. ❸
    Parameters
    -----
    n : int, default 5 ❹
        Number of rows to select.
    Returns
    -----
    same type as caller ❺
        The first `n` rows of the caller object.
    """
```

- ❶ 関数の全体的な説明
- ❷ 振る舞いが、多くの人が期待するものでないかもしれないという注意書き
- ❸ エッジケースの説明
- ❹ 入力引数に期待される型とデフォルト値
- ❺ 出力。型はさまざまで、pandas の DataFrame や Series がよく使われる

Python インタプリタに `help(df.head)` を入力することで同じドキュメントを見ることができる。また、この docstring は、pandas でも自動生成されて表示される (<https://oreil.ly/Ra2KM>)。Jupyter ノートブックで作業している場合は、

`pd.DataFrame.head?`で見ることもできる（`example/ch09/ch09.ipynb`）。

docstring には主に次の 3 つのテンプレートがある。標準化されているので読みやすく、形式も馴染みがある。

- Google (<https://oreil.ly/VXsd9>)
- NumPy (<https://oreil.ly/mJcGp>)
- reStructuredText (<https://oreil.ly/GcEYV>)

次の例のように、好みの形式で docstring テンプレートを自動生成するようにエディターを設定するようにもできる。

```
def process_sdg_data(input_excel_file, columns_to_drop):
    """_summary_

    Args:
        input_excel_file (_type_): _description_
        columns_to_drop (_type_): _description_

    Returns:
        _type_: _description_
    """
    df = pd.read_excel(input_excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index('GeoAreaName').transpose()
    return df
```

そして、これが完成した docstring を持つ関数である。

```
def process_sdg_data(input_excel_file, columns_to_drop):
    """Rearrange SDG (Sustainable Development Goals) data to load in API.

    Args:
        input_excel_file (str): Path to the input data file.
        columns_to_drop (list): List of column names to drop.

    Returns:
        pandas DataFrame: DataFrame with columns expected by API code.
    """
    df = pd.read_excel(input_excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index('GeoAreaName').transpose()
    return df
```

テストに docstring を追加するのは良い習慣だ。テストの目的と期待される結果を説明し、エッジケースの詳細やチェック対象のバグについて説明する。docstring を追加することで、他人が後でテストを理解、保守しやすくなる。Hynek Schlawack による記事 (<https://oreil.ly/hbFto>) は、さらに有用な詳細を提供している。

## 9.1.4 README、チュートリアル、その他の長いドキュメント

長い文書は、読者にプロジェクトの全体的な背景を伝え、あなたが行った仕事を宣伝するものでなければならない。これらは docstring を超えて、ユーザーにプロジェクト全体の概要を与える。プロジェクトのドキュメントに、次のようなものを含めることを検討するとよいだろう。

- プロジェクトの短い概要（1段落でもよい）  
これは「エグゼクティブ・サマリー」と考えてよい。プロジェクトの全体的な目標、誰が使うべきか、そしてユースケースを含めるようにする
- プロジェクトをどのように使い始め、どのようにナビゲートするか  
ノートブックのチュートリアルの形をとってもよいだろう
- プロジェクトの注意事項や制限があるかどうか  
例えば、2023 年以前のデータでしか動作しないのであれば、そのことが最初に強調されていると有用だろう。コメントの奥深くに埋め込まれていたら見逃してしまっても仕方ない
- プロジェクトの次のステップ  
たとえプロジェクトが終わったとしても、将来この仕事を引き継ぐ人のために、次のステップのあるべき姿を書き留めておくとよい

すべてのリポジトリには、こうした事柄を含み、プロジェクトの全体的な紹介をする README.<sup>マークダウン</sup>.md ファイルを置こう。これは Markdown ファイルなので、ファイルの拡張子は .md である。Markdown は構造化された文書を書くための言語で、記号を使ってフォーマットを制御する。Markdown のイントロや、README を書くのに役立つヒントは GitHub のドキュメント (<https://docs.github.com/ja/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/quickstart-for-writing-on-github>) にある。

本章に書かれている他の事柄も同様だが、長いドキュメントも常に最新の状態に保つことが不可欠である。チュートリアルを試そうとしたところ、「コードが変更され

ていて例が動かない」などといった事態ほどイライラが募ることはない。チュートリアルの例を自動テストするようにしておけばこれを防げる。

ドキュメントは、コードと同じ場所にあるのが一番メンテナンスしやすい。例えば、Confluence のような外部ツールではなく、GitHub のリポジトリにあるほうがドキュメントの更新は簡単だ。Sphinx (<https://www.sphinx-doc.org/ja/master/>) のようなドキュメント用のツールを使って HTML ドキュメントを生成し、Read the Docs (<https://oreil.ly/vrM6o>) や GitHub Pages (<https://docs.github.com/ja/pages>) などのツールを使ってホストできる。

わかりやすいネーミング、有用なコメント、きちんとした docstring、そして全体的な紹介を組み合わせることで、実行しやすく、保守しやすく、将来的に作業しやすいコードになることが保証される。

## 9.2 Jupyter ノートブックにおけるドキュメンテーション

Jupyter ノートブックはラフに書かれがちであり、プロジェクトの初期段階に使われることが多い。しかし、たとえ「何かを試してみてうまくいかなかった」というような見通しの悪いものであったとしても、そのノートブックに書かれたコードが将来役に立つ可能性は低くない。また、過去の失敗の記録は役に立つことが少なくない。したがって、そうしたコードを再度読んだり、ノートブックに何が書かれているかを知ることは価値がある。「9.1.1 ネーミング」で説明したように、ノートブックには説明的な名前を付けよう。さらに、一番初めにノートブックの中身を説明するのも良い習慣だ。

ノートブック内では、テキストを追加してノートブックを構造化したり、説明書きを追加したりもできる。コード内の情報を繰り返してはいけない。要約、注意、説明を加えるためにテキストを使おう。コードを更新するときは、常にテキストも更新すること。

Markdown を使って Jupyter ノートブックにテキストを追加できる。セルを「コード」から「Markdown」に変換するには、ノートブックで Esc キーを押してコマンドモードに入って M キーを押す（逆の変換は Y キー）。あるいは、ノートブック上部の〔Code〕ドロップダウンメニューを使う。

図9-1は、テキストとコードの両方をうまく使っている例だ。テキストはコードを

重複させることなく、ノートブックに情報を追加している。

The main tool for preprocessing textual data is a `tokenizer`. A tokenizer splits text into *tokens* according to a set of rules. The tokens are converted into numbers and then tensors, which become the model inputs. Any additional inputs required by the model are added by the tokenizer.

If you plan on using a pretrained model, it's important to use the associated pretrained tokenizer. This ensures the text is split the same way as the pretraining corpus, and uses the same corresponding tokens-to-index (usually referred to as the *vocab*) during pretraining.

Get started by loading a pretrained tokenizer with the `AutoTokenizer.from_pretrained()` method. This downloads the *vocab* a model was pretrained with:

```
In [ ]: from transformers import AutoTokenizer
        tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

Then pass your text to the tokenizer:

```
In [ ]: encoded_input = tokenizer("Do not meddle in the affairs of wizards, for they are
        subtle and quick to anger.") print(encoded_input)
```

```
Out[ ]: {'input_ids': [101, 2079, 2025, 19960, 10362, 1999, 1996, 3821, 1997, 16657, 101
        0, 2005, 2027, 2024, 11259, 1998, 4248, 2000, 4963, 1012, 102],
        'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0],
        'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

The tokenizer returns a dictionary with three important items:

- `input_ids` are the indices corresponding to each token in the sentence.
- `attention_mask` indicates whether a token should be attended to or not.
- `token_type_ids` identifies which sequence a token belongs to when there is more than one sequence.

図9-1 Jupyter ノートブックにコードとテキストを混在させた例 (<https://oreil.ly/edZMf>)

読者がノートブックをナビゲートするのに役立つ見出しを追加するために、ノートブックで Markdown を使うこともできる。見出しの指定には「#」を使う。

```
# 1. トップレベルの見出し
## 1.1 第2レベルの見出し
### 1.1.1 第3レベルの見出し
```

これにより、図9-2に示すような結果が得られる。

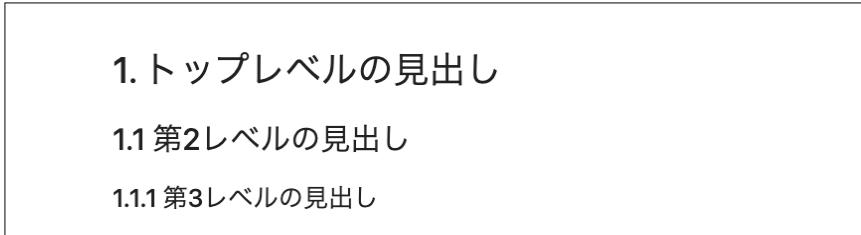


図9-2 Jupyter ノートブックで見出しを使う

複数のセクションがきちんとラベル付けされたノートは、読みやすくナビゲートしやすい。

### 9.3 機械学習の実験の文書化

機械学習（ML）のプロジェクトでは、選択した評価メトリクスに従って最良の予測を行うモデルを探すために、多くの異なるモデル、データセット、ハイパーパラメーターを試すことになる。特にハイパーパラメーターの数は非常に大きくなる可能性があるので、どのような組み合わせを試したかを文書化することが重要である。これは、将来、以前やった仕事に戻ってきたときに役立つし、同じプロジェクトのメンバーの助けにもなる。

実験は、厳密であることを保証するために、構造化された方法で文書化する必要がある。そのためには、実験の各反復で変化するすべての変数を確実に追跡する必要がある。こうすることで、将来実験を再現したり、他の誰かがこのプロジェクトを引き継いだりしたときに、どの変数がテストされたかを知ることができる。また、途中で仮定したこと記録しておく必要がある。

次の事項の記録を検討しよう。

- モデルのトレーニングに使用したデータ
- トレーニング／評価／テスト分割（test split）
- 行ったフィーチャーエンジニアリングの選択
- モデルのハイパーパラメーター（ロジスティック回帰モデルの正則化やニューラルネットワークの学習率など）
- 正確度、精度、再現性など、モデルを評価するメトリクス

こうしたパラメーターをすべて手動で記録するのは手間がかかり、エラーになりやすいので、自動化ツールを使うのがよい。Weights and Biases (<https://wandb.ai>) は機械学習の実験を追跡するのに便利なツールで、scikit-learn、TensorFlow、PyTorchと簡単に統合でき、図9-3に示すように学習パラメーターをWebダッシュボードに記録してくれる。

<input type="checkbox"/> Name (384 visualized)	Created	Runtime	Sweep	encoder	num_train	num_valid	acc
-   worldly-totem-422	1mo ago	12m 54s	-	resnet34	682	97	0.8566
-   jumping-voice-421	1mo ago	11m 59s	-	resnet34	725	92	0.8504
-   logical-energy-420	1mo ago	2m 14s	-	resnet34	66	10	0.626
-   laced-dust-419	1mo ago	2m 4s	-	resnet18	61	15	0.5968
-   whole-music-418	1mo ago	1m 40s	-	resnet18	68	13	0.6139
-   grateful-glitter-417	1mo ago	21s	-	resnet18	70	11	0.2367
-   efficient-lake-416	1mo ago	1m 42s	-	resnet18	76	10	0.6899
-   clear-night-415	1mo ago	1m 17s	-	resnet18	66	10	0.5403
-   glorious-night-414	1mo ago	1m 33s	-	resnet18	68	7	0.7627
-   smart-sponge-413	1mo ago	1m 46s	-	resnet18	72	11	0.6517

図9-3 Weights and Biases を使った実験のトラッキング

その他の実験のトレース用ソリューションとしては、オープンソースの Sacred (<https://oreil.ly/aZFSI>)、MLflow (<https://oreil.ly/tf3SN>)、Neptune (<https://oreil.ly/sohIo>)、AWS の SageMaker Experiments (<https://oreil.ly/uavxZ>) などがある。いずれも、実験の記録を自動化できる。

## 9.4 まとめ

ドキュメントは、他人があなたのプロジェクトを使うのを助け、コードを読みやすくするために、とても重要である。ドキュメントが常に最新で、あなたのコードを使う人のために書かれていることを確認しよう。より詳しく知りたいなら、Write the Docs コミュニティ (<https://oreil.ly/ZQPUg>) に多くの有用なヒントがある。

次に挙げる推奨事項に従うことで、良いドキュメント（やコード）を書くことができる。

### ネーミング

変数名、関数名、ファイル名は、情報量が多く、適切な長さで、読みやすいものでなければならない

### コメント

コメントには、要約や注意事項など、コードに含まれていない補足情報を記す

### docstring

関数には、その関数の入力と出力、そしてその関数の目的を記述した docstring を必ず記載する

### README

すべてのリポジトリやプロジェクトには、コードを宣伝し、なぜそれを使うべきかを知らせる導入部を記載する

### Jupyter ノートブック

良い名前を付け、構造を持たせ、コードに加えてテキストによる説明を加えれば、ノートブックはずっと読みやすくなる

### 実験トレース

特に機械学習プロジェクトにおける実験は、構造化された方法でトレースされるべきである

本章にあるテクニックはすべて、コードを読みやすくし、それが何をするものかを伝えやすくし、他人がそれを使うのを容易にするものである。

# 10章 コードの共有

他人とコードを共有し、互いに協力し合うことは、データサイエンス（DS）分野で成功を取めるのに非常に重要だ。大規模なコードベースを持つ既存のプロジェクトや、オープンソースのプロジェクトに参加する場合もあるだろう。最初は単独で取り組むかもしれないが、プロジェクトが成長するにつれて、すでに解決された問題に再度自分が時間を費やすことのないよう、他人とコードを共有したくなる。

オープンソースのプロジェクトとしてコードを公開すれば、コミュニティに参加できる。Python のオープンソースライブラリは巨大な「エコシステム」であり、pandas、NumPy、scikit-learn、Matplotlib、その他多くのライブラリがある。こうしたライブラリがなければ、DS はより困難なものになっていただろう。

オープンソースプロジェクトに貢献するにしても、同僚と共同作業するにしても、複数の人が同じコードで作業しやすくするためのツールやテクニックを知っておく必要がある。ツールはさまざまかもしれないが、原則の多くは標準的なものだ。

本章では、他の開発者とのコラボレーションに役立つツールやテクニックを取り上げる。まず、VCS（バージョン管理システム）だ。VCS はコードを共有して作業を行うための標準的な方法であり非常に重要だ。次に、コードが依存するサードパーティのライブラリを管理するためのツールについて説明する。最後に、他人がインストールできるようにプロジェクトをパッケージ化する方法を説明する。

## 10.1 Gitを使ったバージョン管理

バージョン管理は、コードベースに加えられた変更を追跡する方法であり、複数の人が同じコードで簡単に作業できるようにする。誰がコードにどのような変更を加えたかを確認でき、2人が同じ行に関して作業している場合にも衝突を防ぐことができ

る。また、誰かがバグを引き起こすような変更を加えたり、ミスをしたりしても、簡単に前のバージョンのコードに戻ることもできる。

VCS は通常、「分散型」である。コードのコピーはすべての開発者のコンピュータ（ローカル）に分散して保存されるだけでなく、中央のホスト（リモート）にも保存される。ただ、すべてのコードはバックアップしておくべきだ。ノート PC を紛失したり、リモートのインスタンスが壊れたりなど、不測の事態に備えよう。

Git (<https://git-scm.com>) は、分散型バージョン管理のための最も一般的なシステムだ。オープンソースプロジェクトであり、2005 年に Linus Torvalds (Linux の作者でもある) によって開発された。2022 年の Stack Overflow Developer Survey (<https://oreil.ly/zMyyl>) によると、プロの開発者の 96% が Git を使っている。なお、Git 以外にも Subversion (<https://subversion.apache.org>) や Mercurial (<https://www.mercurial-scm.org>) といった VCS も存在する。

GitHub (<https://github.com>。オーナーは Microsoft) は Git を使ってトレースされるコードを保存するための最も人気のあるサイトだが、GitHub と Git の間に公式なつながりはない。他の類似サイトには GitLab (<https://about.gitlab.com>) や Bitbucket (<https://bitbucket.org/product>) がある。



プログラミングコード用に設計された VCS は、データのバックアップやバージョン管理に使うようには設計されていない。データ用には、専用のシステムを使うほうがよい。コード用の VCS は、サイズの大きなファイルに対応するようには設計されておらず、不必要的重複やセキュリティリスクにつながる可能性もある。

Git には多くの機能があるが、ここでは最も重要な機能の概要を紹介する。もっと知りたいなら、Git のドキュメント (<https://git-scm.com/book/ja/v2>) か GitHub のドキュメント (<https://docs.github.com>) を参照してほしい<sup>†1</sup>。「こんな事になってしまったが、どうしたらいい？」といった使い方のヒントが欲しいのなら「Dangit, Git」(<https://dangitgit.com>) がお勧めだ。

†1 訳注：Git をより深く学びたい方には、Prem Kumar Ponuthorai 著『実用 Git 第 3 版』（オンライン・ジャパン）がお勧めだ。逆に、Git の基本をマスターしたい方には、Anna Skoulikari 著『Git ハンズオンラーニング』（オンライン・ジャパン）をお勧めする。

## 10.1.1 Git の動作

Git は、作業中のプロジェクトが単独のディレクトリに保存されており、そのディレクトリ内のコードに対するすべての変更を記録していくという前提で動作する。Git によって保存されるデータは、<sup>トランク</sup>追跡しているすべてのファイルとディレクトリの構造のスナップショットであり、このスナップショットは、開発者がコードを変更するたびに更新されていく。つまり、コードに加えたすべての変更の履歴が記録されるのだ。

**リポジトリ** (repository。略して「repo」とも呼ばれる) とは、Git が変更を記録しておくディレクトリのことだ。Git を使う最初のステップでは、コードを保存しているディレクトリでローカルリポジトリを初期化する。これによって「隠しフォルダ」である<sup>リポジトリ</sup>.git が作成され、その中にコードのすべてのスナップショットを保存するデータベースが作られる。

Git は、作業中にファイルに加えた変更を追跡するが、その変更のスナップショットは保存しない。変更は作業ディレクトリにのみ存在する。行った変更に満足し、変更したファイルのバージョンを保存したい場合は、それを**ステージング領域** (staging area) に<sup>アッフド</sup>add (追加) する。ステージング領域には、次に保存するスナップショットに含めたいすべての変更が記憶されている。

次のステップは、変更の**commit**だ。<sup>コミット</sup>commit を行うと、ファイルの現在のバージョンのスナップショットがローカルリポジトリに保存される。内容を変更してコミットに追加したファイルのみが保存の対象になる。各コミットには一意の ID (「コミットハッシュ」。暗号化されたハッシュ関数で作られたもの) が付与され、これを使ってこのコミットに関する情報を後で参照できる。このワークフローの例を「10.1.2 変更とコミットの追跡」で示す。

プロジェクトのコードの状態をローカルコンピュータに保存するものを**ローカルリポジトリ**と呼ぶ。このほかに**リモートリポジトリ**がある。こちらは GitHub などの中央サーバーに置かれるコードのコピーだ。リモートリポジトリにコミットの内容を追加するには、ローカルリポジトリから「push」<sup>プッシュ</sup>する。そのプロジェクトで他の人も作業している場合は、その人の変更をリモートリポジトリから「pull」<sup>プル</sup>すればよい。こうすると、ローカルコンピュータ上のファイルにその人の変更が反映される。

図10-1 は、コードの add、commit、push、pull の各ステップがどのように組み合わされるかを示すものだ。

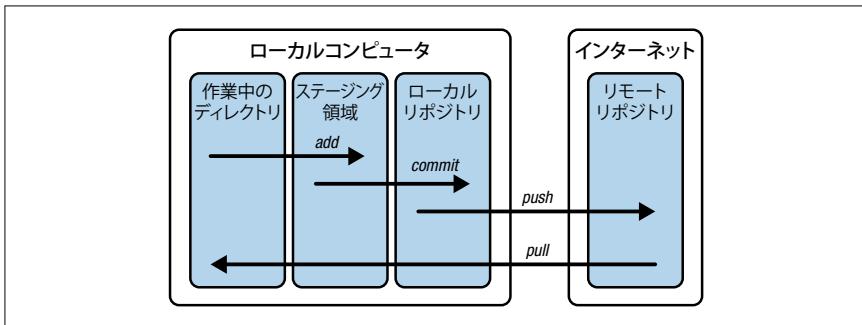


図 10-1 Git の主要コマンド

### 10.1.2 変更とコミットの追跡

例を見よう。Git のインストールは、Git のドキュメント (<https://oreil.ly/rDMik> で「日本語」を選択) の手順に従って行うことができる。

プロジェクトで作業していく、Git に変更を追跡させたい場合、まずはプロジェクトディレクトリのトップレベルで次のコマンドを実行する。

```
$ git init
```

次に、変更したファイルをステージング領域に追加する。ここでは README.md ファイルを作成して保存してから add してみよう。

```
$ git add README.md
```

いつでも、git status を実行して現在の状況を知ることができる。

```
$ git status
On branch main
```

No commits yet

```
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
  new file: README.md
```

このメッセージには、現在 main ブランチ (branch) で作業していることが書かれている。ブランチについては「10.1.4 ブランチとプルリクエスト」で説明する。

その後、変更をローカルリポジトリにコミットする。

```
$ git commit -m "Initial commit"
```

次のようなメッセージが表示される。

```
[main (root-commit) cf9d13e] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

各コミットには、一意の ID、作者名、タイムスタンプ、そして変更内容が紐づけられている。-m で指定する**コミットメッセージ**には自分が行った変更について、簡潔かつわかりやすい説明を加えよう。

もう 1 つ別のコミットを見てみよう。リポジトリにコードを書いて、変更点を保存する段階になったら、add コマンドをもう一度実行して、変更したファイルをステージングエリアに追加する。ここでは api\_functions.py を新たに作成して add している。

```
$ git add api_functions.py
```

そして、次のコマンドで変更をコミットする。

```
$ git commit -m "Created helper functions for API"
```

次のようなメッセージが表示される。

```
[main 9f09222] Created helper functions for API
 1 file changed, 65 insertions(+)
 create mode 100644 api_functions.py
```



-m で指定するコミットメッセージは日本語で書くこともできる。関係者が全員日本語を理解できるのならば、日本語で書いてもよいだろう。

コミットにはいくつかのベストプラクティスがある。

- 各コミットは 1 つの項目に関する変更にとどめるのがよい（問題の修正や小さな機能の追加など）
- 変更は小分けにするのがよい（そうすることで、ある変更をしたときに後で問題が発生しても、その問題を引き起こした変更まで簡単に さかのば 遷る）

きる)

- コミットを行う前にテストをする（このプロセスを自動化する方法については、12章で説明する）

### 10.1.3 リモートとローカル

これまでローカルのリポジトリでGitを使う方法を説明してきた。ここでは、ローカルリポジトリをリモートリポジトリにリンクする方法と、リモートリポジトリの内容をローカルコンピュータにダウンロードする方法を説明する。

「10.1.2 変更とコミットの追跡」の手順を踏んだら、次はリモートリポジトリを作成する。正確な手順はリモートリポジトリのホスティング先によって異なる。**図10-2**に、GitHub (<https://github.com>) でリモートリポジトリを作成しているところを示す。

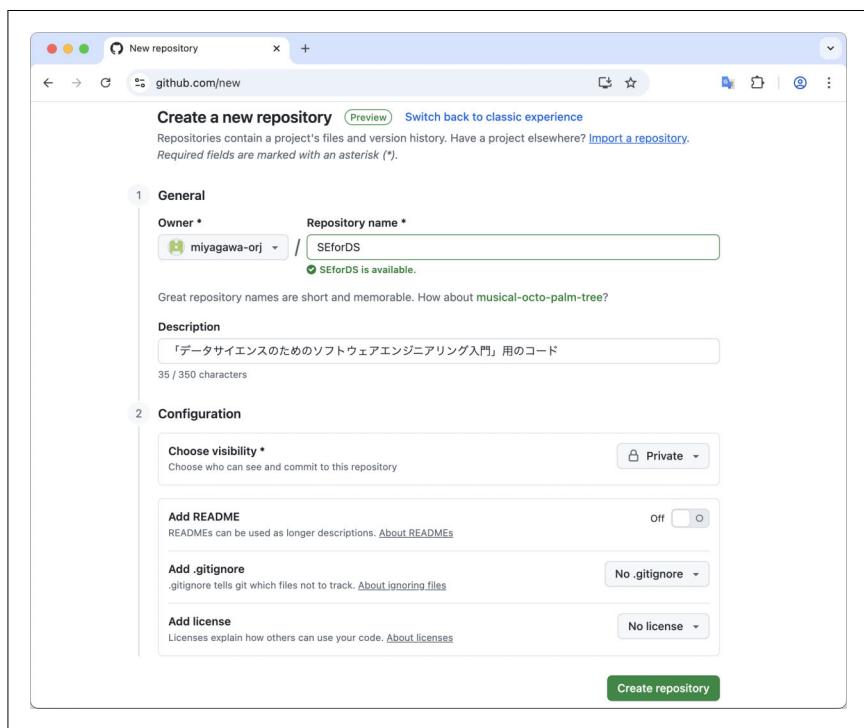


図10-2 新しいリポジトリの作成

GitHub でリポジトリを作成したら、次のコマンドを実行してローカルバージョンの Git にリモートリポジトリの場所を教える必要がある。下記の URL (xxx 以降。xxx をアカウント名に変える必要がある) をリモートリポジトリの URL に置き換える。

```
$ git remote add origin https://github.com/xxx/se4ds.git
```

最後に、ローカルリポジトリからリモートリポジトリにコミットをpushする。<sup>プッシュ</sup>

```
$ git push -u origin main
```

次のようなメッセージが表示される。

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 219 bytes | 219.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/xxx/se4ds.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

このコマンドを実行すると、リモートリポジトリにはローカルリポジトリのファイルがコミットの変更で更新された状態で保存される。

リモートリポジトリにすでに既存のコードがあるプロジェクトに参加する場合もあるだろう。このような場合、「clone」<sup>クローン</sup>する必要がある。つまり、リモートリポジトリの内容を自分のコンピュータにダウンロードする。これが新しいローカルのリポジトリになる。

次のコマンドで（本書の例題の）リポジトリをクローンできる。

```
$ git clone https://github.com/urupat/se4ds.git
```

クローン作成には 2 つのオプションがある。HTTPS と SSH だ。SSH はより高いレベルのセキュリティを提供するが、セットアップに時間がかかる。

なお、最初にリモートのリポジトリを作成し、そのリポジトリをローカルマシンにクローンしてからコードを書き始めるという、プロジェクトの開始方法もある。

次節では、共同作業に特に役立つ Git の機能、branch<sup>ブランチ</sup>と pull<sup>ブル</sup>リクエストについて説明する。

## 10.1.4 ブランチとプルリクエスト

Git のブランチ (branch) を利用することで、自分が行った変更を追跡しつつ、主要なバージョンのコードに影響を与えずに、新しい機能を試せる。同じコードベースで多くの人が作業している場合に特に便利な機能だ。

ブランチを作成しても、コードを含む全ファイルのコピーを作成するわけではない。その代わり、Git はプロジェクトのタイムラインの特定の時点からの変更を記録していく。この機能を使うことで「ある機能を試し、その機能実現のために行った変更を元のコードにマージ（合体）する」という作業を、簡単に行えるわけだ。

すべてのリポジトリにはデフォルトのブランチがあり、通常は `main` という名前が付いている。図 10-3 に `main` ブランチと新しいブランチの関係を示す。ブランチを作成し、好きなだけ commit を追加して、その変更を `main` ブランチにマージする。新しいブランチで作業している間は、どの変更も `main` ブランチには影響しない。一方、誰かほかの人が `main` ブランチに変更を加えても、自分が作業しているコードは影響されない。

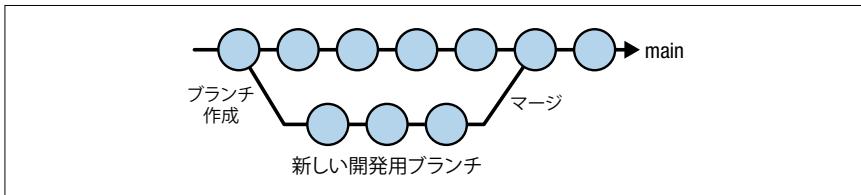


図 10-3 ブランチの作成。円が 1 つのコミットを表す



リポジトリのデフォルトブランチに `main` という名前を使うのは慣例であり、好きな名前に変更できる。`master` という名前をデフォルトブランチに使っているリポジトリもまだあるかもしれないが、「master」という言葉には奴隸的な意味合いがあるので、避けたほうがよい。

`new_branch` という名前の新しいブランチを作成するには、次のコマンドを実行する。

```
$ git branch new_branch
```

次に、次のコマンドで、作業している場所をこのブランチに変更する。

```
$ git checkout new_branch
```

`git checkout` に `-b` フラグを付けることで、先の 2 つのコマンドを一度に実行できる。

```
$ git checkout -b new_branch
```

ブランチをチェックアウトした後、ローカルでコードに加えた変更は、`main` ブランチではなく、自分のブランチ `new_branch` で行われる。

新しいブランチ `new_branch` をリモートリポジトリに保存したい場合は、次のコマンドを実行する。

```
$ git push origin new_branch
```

新しいブランチでの作業が終わったら、それを `main` ブランチにマージする。これで新しいブランチで変更されたコードが `main` ブランチに組み込まれることになる。

マージする前に、次のコマンドを使って必ず `main` ブランチに切り替えておく。

```
$ git checkout main
```

そして、ブランチ `new_branch` をマージする。

```
$ git merge new_branch
```

`new_branch` をマージしたら、`new_branch` は不要なので次のコマンドで削除する（コミットの履歴は保存される）。

```
$ git branch -d new_branch
```

ブランチをマージする際に**コンフリクト**（衝突）が生じることがある。これは、`new_branch` で変更したコード行が、`main` ブランチでも変更されている場合に発生する。このコード行のどちらのバージョンを残すべきかを決める必要があるのだ。



Jupyter ノートブックに関して、マージ時のコンフリクトが起こることは少な  
くない。ノートブックは JSON 形式で保存され、これにはセル出力の内容も含  
まれる。セルが少しでも変更されると、Git によってトレースされ、コンフリク  
トの要因となりうる。

これにはいくつかの解決策がある。最も単純なのは、手動でノートブックのセル

出力を消去することだが、これを忘れずに行うのは困難を伴う（12章で自動的に行う方法を紹介する）。2つ目の方法は、nbdime (<https://oreil.ly/cV563>) を使うことだ。これは、Git がセルの出力ではなくノートブックのコードだけを比較するように設定するものだ。3つ目の方法は、Jupytext (<https://oreil.ly/O0yH7>) を使ってノートブックを.py や.md ファイルに変換し、Git を使って簡単に追跡できるようにすることだ。

プランチをマージする前にはほかの開発者にコードレビューをしてもらいたい場合は、プルリクエストを使えば、変更をマージする前に議論やレビューをしてもらえる。

プルリクエストを作成（open）する一番簡単な方法は、図 10-4 に示すように GitHub のインターフェースを使うことだ。

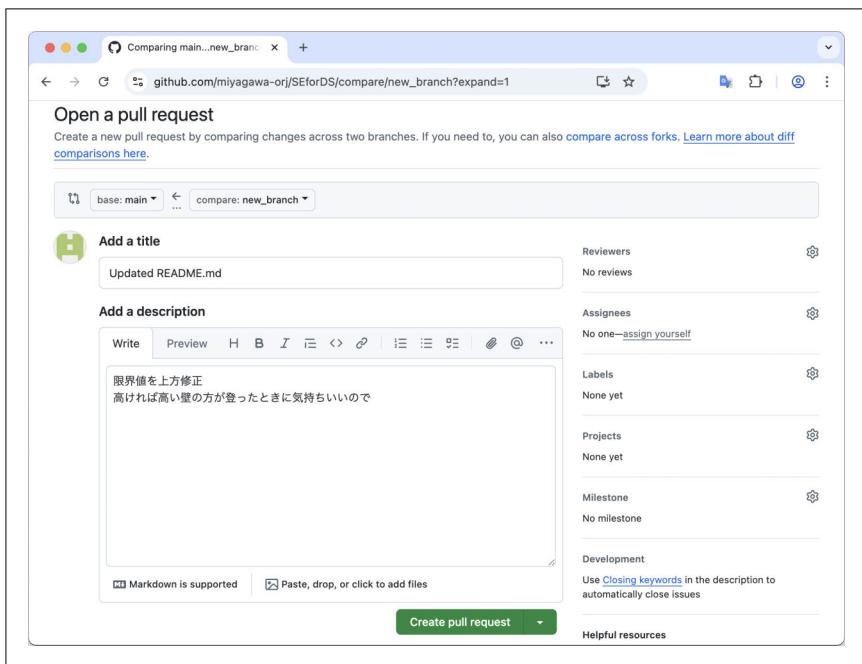


図 10-4 プルリクエストの作成

プルリクエストには、何を変更するのか、なぜその変更を行う必要があるのかを説明するコメントを含める。そうすることで、レビュー者がどこに注目するべきかが明

確になる。

## コードレビュー

多くの開発チームや組織は、**コードレビュー**のための仕組みを持っている。システムレビューレビュー

コードレビューとは、開発されたコードに対する検討を行うための、(ある程度)形式が決まっているセッションのことで、ブルリクエストを使って行われることが多い。コードレビューの目的は、他のチームメンバーに、まずコードをレビューする機会を与え、バグやセキュリティリスクが問題になる前に発見し、本番環境に投入される前にコードを改善することである。また、アイデアを共有してコードの構造を改善したり、経験豊富な開発者が知識を伝授したりする機会にもなる。さらに、コードレビューにより、どのような決定が、なぜなされたかが記録されることになる。

コードレビューの「ベストプラクティス」としては次のようなものがある。

- 読みやすさ、単純さ、モジュール性など、コードの品質のさまざまな側面を考慮する
- 全員の意見を聞くようとする。誰の経験も貴重なものであり、後輩たちにも質問する機会を与えるべきだ
- 問題点を指摘するだけでなく、検討中のコードが優れている点も強調する
- コードレビューに費やす時間を制限する。コードレビューが遅延の原因になつてはならない

Tim Hopper のブログ記事 ([https://oreil.ly/hn\\_xv](https://oreil.ly/hn_xv)) は、データサイエンス分野のコードレビューについて、有益な示唆を与えてくれるであろう。

Git をうまく使うために知っておくべきコマンドやテクニックは他にもある。ファイルを Git による管理対象から外す方法、加えた変更を元に戻す方法などだ。詳しく知りたい場合は「10.1 Git を使ったバージョン管理」で紹介したリンクを参照してほしい。

本節では、バージョン管理のための Git の使い方を簡単に説明した。次は、Python プログラミングに特化した話に移ろう。依存関係の管理と仮想環境の使い方だ。

## 10.2 依存ライブラリと仮想環境

自分が書いたコードを他人に使ってもらうには同じライブラリをインストールしてもらう必要がある。自分のコードが動作するために必要とするサードパーティーライブリのことを「依存ライブラリ」と呼ぶ。例えば pandas や NumPy などだ。自分の書いたコードが動作するために、あるライブラリの特定のバージョン（あるいは特定の範囲のバージョン）を使わなければならない場合もある。そのため、コードを再現可能なものにするには、依存するライブラリのバージョンの情報（依存関係）を含める必要がある。

### バージョニング

Python のライブラリ（だけでなく、ほとんどのソフトウェアパッケージ）にはバージョン番号が振られていて、特定のリリースとライブラリの対応関係を管理できる。プロジェクトへのバージョンの付加（バージョニング）はベストプラクティスの 1 つである。バージョニングにはいくつかの方法があるが、セマンティックバージョニング（SemVer。<https://semver.org>）は非常に広く使われている手法だ。また、カレンダー・バージョニング（<https://calver.org>）という手法も見かけることがあるだろう。

プロジェクトが SemVer を使用している場合、Python 3.10 や NumPy 1.24.3 のように、ピリオドで区切られた 2 つまたは 3 つの数字が割り当てられる。最初の数字はメジャー・バージョンのリリース、2 番目はマイナーバージョン、3 番目はパッチバージョンを意味する。

メジャー・バージョンとは、Python 2.x から Python 3.x、pandas 1.x から pandas 2.x のように、非互換の変更（以前は動作していた機能が変更され、動作しなくなるものが存在するような変更）があることを意味する。マイナーバージョンとは、より小さなアップデートがあることを意味し、パッチバージョンは、小さなバグを修正するような小規模の変更を意味する。

本節では、依存関係を管理するツールの概要を説明し、使用しているライブラリのすべてのバージョンの手動の追跡を不要にする。それには、まず仮想環境について知っておく必要がある。

## 10.2.1 仮想環境

**仮想環境**は Python の重要な概念であり、既知の人も多いかもしれないが、ここで簡単に概要を説明しよう。仮想環境は、利用中のコンピュータのシステム（OS）における Python 環境ではなく、特定のプロジェクト用にインストールするライブラリを分離する方法だ。つまり、Python インタプリタと Python 標準ライブラリに含まれるモジュールはシステム全体で共有されているが、付加的なライブラリはプロジェクトごとに別のスペースにインストールすることができる。



システム全体の Python 環境に付加的なライブラリをインストールしないことを強く推奨する。これをしてしまうと、プロジェクトのテストやデバッグが難しくなるだろう。コンピュータの OS には、システム全体に影響を及ぼす Python パッケージがある。システムパッケージマネージャーのコマンド（macOS の brew、Windows の scoop、Linux の apt など）を使わずにライブラリの（アン）インストールを行うと、システムが壊れたり動作が遅くなったりする危険性がある。

仮想環境を活用することで、複数のプロジェクトで、競合することなく、特定のライブラリの異なるバージョンを利用できる。具体的な例を挙げると、プロジェクト A は NumPy のバージョン 1.25 で導入された機能を使う必要があるが、プロジェクト B は旧バージョンの構文を使っているといったケースである。システム全体で NumPy をアップグレードすると、プロジェクト B は動作しなくなる。仮想環境を用いることで、同じライブラリの複数のバージョンをインストールでき、この問題が解消される。

Python の仮想環境を管理するためのさまざまなツールがある。人気の高いツールには次のようなものがある。

- venv (<https://oreil.ly/QIEQc>) は Python 標準ライブラリに含まれている
- virtualenv (<https://oreil.ly/PMBoK>) は venv のサブセットである
- Pyenv (<https://oreil.ly/nnvwr>) は Python のバージョンを管理し、システムに複数のバージョンの Python を同時にインストールすることを可能にしてくれる（例えば、プロジェクト A には Python 3.7 を、プロジェクト B には Python 3.10 を使うといったことができる）。pyenv-virtualenv プラグイン (<https://oreil.ly/tZ8Xv>) を使えば、仮想環境も管理できる

- Conda (<https://oreil.ly/VrIMo>) は仮想環境を管理するが、パッケージのインストールも行う。標準的な Python パッケージリポジトリである PyPI (Python Packaging Index。<https://pypi.org>) ではなく、Anaconda (<https://www.anaconda.com>) が管理する別のパッケージリポジトリからパッケージをダウンロードする
- Poetry (<https://python-poetry.org>)、PDM (<https://pdm.fming.dev>)、Hatch (<https://hatch.pypa.io>) も仮想環境と同様に依存関係を管理する。Poetry については次節で詳しく説明する

ここでは venv の例を紹介するが、ツールの詳しい使い方は上記のリンクを参照されたい。

パッケージをインストールする前に、macOS または Linux では次のコマンドを使って新しい仮想環境を作成する (se4ds の部分は要変更)。

```
$ python -m venv se4ds
```

次のコマンドで仮想環境を有効にする。アクティベート

```
$ source se4ds/bin/activate
```

コマンドプロンプトの前に、仮想環境の名前が括弧で囲まれて表示されるはずだ。

```
(se4ds)$
```

これは上記のどの仮想環境ツールにも共通している。これで、pip を使ってパッケージを仮想環境にインストールできる。

仮想環境を使い終わったら、次のコマンドで終了する。

```
(se4ds)$ deactivate
```

これでコマンドプロンプトから仮想環境の名前が消える。

新しいプロジェクトを始めるたびに新しい仮想環境を作成し、それをアクティブにしてから、そのプロジェクトに必要なライブラリをインストールするのがベストプラクティスだ。システム全体の Python は「クリーンな状態」に保ち、サードパーティのライブラリはシステム全体にインストールしないこと。そうすることで、ライブラリのバージョン間の衝突を防止できる。

## 10.2.2 pipによる依存関係の管理

プロジェクトに必要な依存関係を記録する最も簡単な方法は、Python に組み込まれているパッケージマネージャー、`pip` を使うことである。`pip freeze` コマンドを使って、プロジェクトが必要とするすべてのパッケージのリストを保存できる。

```
$ python -m pip freeze > requirements.txt
```

このコマンドは環境内のすべてのパッケージの名前とバージョンを `requirements.txt` というテキストファイルに書き出す。このコマンドを実行する際には、プロジェクトに仮想環境を使い、それをアクティブにしておく必要がある。そうしないと、システムの Python 環境のライブラリを記録することになる。

例えば、`numpy` と `pandas` を `pip` を使ってインストールした後上記コマンドを実行すると `requirements.txt` は次のようなテキストファイルになる。

```
numpy==2.3.2
pandas==2.3.2
python-dateutil==2.9.0.post0
pytz==2025.2
six==1.17.0
tzdata==2025.2
```

このファイルには、現在の環境にインストールされているライブラリのバージョンと、それらが依存しているライブラリが含まれている（例えば、`pandas` は Python の `dateutil` ライブラリに依存している）。

`requirements.txt` ファイルを VCS のリポジトリにアップロードしておけば、誰かがプロジェクトを利用したいときや別の環境でプロジェクトを実行したいときに、このコマンドを使ってプロジェクトの依存ファイルをインストールできる。

```
$ pip install -r requirements.txt
```

このプロセスは単純だが、欠点もいくつかある。例えば、あるライブラリ（例えば `pandas`）をプロジェクトから削除する場合、他のどのライブラリが依存しているかを調べて、手動で削除する必要がある。また、`requirements.txt` には、Python のバージョンに関する情報は含まれていない。さらに、使っているライブラリに依存しているライブラリに変更がある可能性もある。依存関係の管理に `pip` を使っている場合、こうしたことを手動で処理する必要がある。幸いなことに、次節で説明する他

の依存関係マネージャーを使えば、手動で行う必要はなくなる。

### 10.2.3 Poetry による依存関係の管理

Poetry (<https://python-poetry.org>) や PDM (<https://pdm.fming.dev>)、Hatch (<https://hatch.pypa.io>) などの依存関係マネージャーを使うと、`pip` を使うよりずっと簡単にプロジェクトの依存関係を管理できる。ここでは、Poetry を使ってプロジェクトの依存関係を管理する方法を説明するが、Poetry は、後ほど説明する「パッケージング」にも利用できるので大変便利だ（「10.3 パッケージング」参照）。

Poetry は `requirements.txt` の代わりに `pyproject.toml` というファイルを使う。このファイルには、依存関係を含むプロジェクトのすべての要件が含まれており、パッケージをインストール可能にするための標準的なファイルでもある。

Poetry は次のコマンドでインストールできる（Poetry 自体はすべての仮想環境で動作させたいので、仮想環境ではなくシステム全体にインストールする必要がある）。

## 実行前に、下記の「訳者補記」参照！

\$ curl -sSL <https://install.python-poetry.org> | python3 -



Poetry は上記のコマンドでもインストール可能だが、最近では公式ページでも `pipx` を使うインストールが推奨されている。

`pipx` がインストールされていない場合は、まず `pipx` のインストールが必要だ。インストール方法の詳細は <https://github.com/pypa/pipx> にあるが、例えば macOS では `brew` を使って次のようにすればよい（パッケージマネージャーを使えば Windows や Linux でも、ほぼ同じようにインストールできる）。

```
$ brew install pipx
$ pipx ensurepath # パスの処理などを行う
```

`pipx` をインストールしてあれば、次のコマンドで Poetry がインストールできる。

```
$ pipx install poetry
```

`pipx` は Python で作られたコマンドラインツールを専用の仮想環境にインストールして、シェルのコマンドとして安全に使えるようにするツールだ。専用の仮想環境を作って `poetry` をインストールするので、ほかのツールとのライブラリの競合を心配する必要がなくなる（その分ディスクを消費することになるが「安全第一」だ）。

原著では、この節以降も、コマンドのインストールに pip を用いているが、  
pipx を用いたほうがよいだろう。

繰り返すが、pipx はターミナルなどで実行するコマンドをインストールするためのツールだ。ライブラリやフレームワークをプロジェクト内に入れる用途には、本章で説明しているように仮想環境を作って pip を使うのがお勧めだ。

なお、Poetry に限らずツール等のインストール方法や使用方法は頻繁に変わるもののが少なくない。本書に記載の方法でうまくいかない場合は、公式ページなどで確認されたい。

新しいプロジェクトを開始する場合、Poetry に、開始するのに必要なファイルやフォルダが入った新しいフォルダをセットアップしてもらうことができる。次のコマンドを実行すればよい。

```
$ poetry new se4ds # se4ds はプロジェクト名
```

すでにコードを書き始めている場合は、次のコマンドを使って、既存のフォルダ内で Poetry を初期化できる。

```
$ poetry init
```

どちらかのコマンドを実行すると、プロジェクトフォルダに `pyproject.toml` と `poetry.lock` という名前のファイルができる（後者は、最初はできない場合もある）。`pyproject.toml` には、プロジェクトの依存関係をリストしたセクションがある。サードパーティのライブラリをインストールする前にファイルの中身を見ると、Python のバージョンを記述した部分がある。

```
requires-python = ">=3.13"
```

`poetry add` コマンドを使ってプロジェクトが必要とするライブラリを追加する。これを実行するとプロジェクトの仮想環境にもライブラリがインストールされる。

```
$ poetry add pandas
```



訳者補記

プロジェクトフォルダ内 (`pyproject.toml` ファイルがあるフォルダあるいはそのサブフォルダ) で `poetry` のコマンドを実行すると、このプロジェクト専用の仮想環境に対して操作が行われる。たとえば上のコマンドを実行すると、まだ仮想環境が作られていないければ新たに作られて、`pandas` がその仮想環境

内にインストールされる。

なお、実際の仮想環境は、たとえば macOS の場合ならば、ホームディレクトリの下の `Library/Caches/pypoetry/virtualenvs` の下に、`se4ds_g16DKqh-py3.13/` といった名前で作られる（プロジェクト名+ハッシュ+Python のバージョン）。

次のコマンドで仮想環境の場所を確認できる。

```
$ poetry env info
```

`pyproject.toml` ファイルの中身を見ると、依存関係に `pandas` が追加されているのがわかる。

```
dependencies = [
    "pandas (>=2.3.2,<3.0.0)"
]
```

この例では、`pandas` のバージョンは `>=2.3.2,<3.0.0` と指定されているので、このプロジェクトでは `pandas` のバージョン 2.3.2 以降で、3.0.0 未満が条件となっている。

Poetry で仮想環境を使うには次のコマンドを実行すればよい。

```
$ poetry env activate
```

なお、仮想環境に明示的に入らなくても、仮想環境の操作はできる。たとえば、プロジェクトフォルダ内で `poetry add xxx` を行えば、ライブラリ `xxx` が仮想環境にインストールされる。

Poetry は `pyproject.toml` に必要なライブラリのバージョンをリストするだけでなく、`poetry.lock` を使ってプロジェクト内のライブラリの正確なバージョンのハッシュを保存する。これは、プロジェクトを異なる OS 間で確実に動作させるのに役立つ。ライブラリをインストールするたびに、`poetry.lock` ファイルは更新され、例えば次のようになる。

```
# This file is automatically generated by Poetry 2.1.4 and should not be changed by hand.
```

```
[[package]]
name = "numpy"
version = "1.23.2"
description = "Fundamental package for array computing in Python"
optional = false
```

```

python-versions = ">=3.11"
groups = ["main"]
files = [
    {file = "numpy-2.3.2-cp311-cp311-macosx_10.9_x86_64.whl", hash = "sha256:852ae5bed3478b92f093e30f785c98e0cb62fa0a939ed057c31716e18a7a22b9"}, 
    {file = "numpy-2.3.2-cp311-cp311-macosx_11_0_arm64.whl", hash = "sha256:7a0e27186e781a69959d0230dd9909b5e26024f8da10683bd6344baea1885168"}, 
    ...
]

```

このファイルを VCS にアップロードすることになる。

プロジェクトの依存関係を別の環境にインストールするには、次のコマンドを使う。

```
$ poetry install
```

Poetry はライブラリのバージョンをチェックするために `poetry.lock` を探し、新しい環境にインストールする。このファイルがない場合、Poetry は `pyproject.toml` で指定されたライブラリの最新バージョンだけをインストールする。

`pip` を使うにせよ `poetry` を使うにせよ、プロジェクトの依存関係を常に最新にしておくことは、他人が簡単に使えるようにするのに重要である。

## 10.3 パッケージング

自分のコードを他人が簡単に使えるようにしたい場合は、パッケージを作成する（「パッケージング」）。つまり、NumPy や pandas などと同じように、Python 環境にインストールして、作業中のプロジェクトにインポートできるようにするわけだ。本節では、Python のコードのパッケージングの方法を説明する。

Python はパッケージの巨大なエコシステムを持ち、PyPI (Python Package Index。<https://pypi.org>) がそれを可能にしている。PyPI は膨大な数のパッケージをホストしており、本書執筆時点で 473,661 ものプロジェクトがある<sup>†2</sup>。

コードを公開したい場合は、パッケージを PyPI にアップロードすれば、誰でも `pip` を使ってインストールし、プロジェクトで使うことができる。また、パッケージを作成して社内でパッケージファイルを共有し、同僚がインストールして使えるようにもできる。パッケージにすればコードが再現可能であることが保証される。

パッケージを作成し、それを公開する場合には保守の責任も生じる（少なくとも他人が簡単に利用できるようにする責任がある）。利用者も保守を手伝うことが望ま

---

<sup>†2</sup> 訳注：本書翻訳時点では、676,097 プロジェクトに増えていた。

れる。作成者と利用者が協力してパッケージを更新するような状況になれば理想的だろう。



Python の「モジュール」とは .py ファイルのことである。これに対して、Python の「パッケージ」は次の 2 つのうちの 1 つを意味する。

1. 他のモジュールを含む `__init__.py` ファイルがあるディレクトリのこと。  
別のディレクトリからモジュールをインポートするためには `__init__.py` が存在しなければならない
2. PyPI 経由で配布される配布パッケージのこと。これをダウンロードしてモジュール（ライブラリ）をインストールする

本節では、「パッケージ」という言葉を 2 番目の意味で使う。

本節の残りの部分では、パッケージを作成して PyPI にアップロードする方法を説明する。

### 10.3.1 パッケージングの基本

パッケージのビルトを開始する前に、コードが次の基準を満たしていることを確認してほしい。

- 自分で計画した設計どおりになっている
- 完全に機能している
- きちんとフォーマットされている（6 章参照）
- テスト済みである（7 章参照）
- ドキュメントがきちんと整備されている（9 章参照）

また、パッケージにしたいコードを 1 つのフォルダに入れる必要がある。このフォルダにはパッケージと同じ名前を付け、関連するテストやドキュメントとは別にする。多くの場合、コードを含むフォルダは `src` という別のフォルダに入る。例えば次のような構造だ。

```
se4ds
├── LICENSE
├── README.md
├── pyproject.toml
└── src
```

```

|   └── se4ds
|       ├── __init__.py
|       ├── functions.py
|       └── ...
|
├── tests
|   └── ...
|
└── docs
    └── ...

```

このファイル構造では、パッケージは `se4ds` という名前になる。また、ライセンス (`LICENSE`) ファイル、概要のドキュメントを提供する `README.md`、それに `pyproject.toml` がある。

パッケージングは次の手順で行う——PyPI のドキュメント (<https://packaging.python.org/ja/latest/flow/>) から引用。

1. 構成ファイルを用意する。これにはプロジェクトのメタデータとパッケージビルドツールの指示を含む。(「10.2.3 Poetry による依存関係の管理」で説明した) `pyproject.toml` を使うのが標準的な方法だが、具体的な内容は下で説明する
2. パッケージファイルにするための（ビルドするための）ツールを実行する。これでソースコードと構成ファイルが読み込まれてパッケージが作成される
3. ビルドされた結果を PyPI や他の配布サービス（組織内も含む）にアップロードする

ユーザーは (`pip` などを使って) インストールして、使えるようになる。詳細は Claudio Jolowicz 著『ハイパーモダン Python』（オンライン・ジャパン）を参照されたい。



ここでは Anaconda パッケージ分散システムのパッケージングについては触れていないが、Conda のドキュメント ([https://oreil.ly/h\\_byJ](https://oreil.ly/h_byJ)) で解説されている。Anaconda はパッケージを無料で使えるようにしているが、パッケージのアップデートは Anaconda しかできない。

`conda-forge` プロジェクト (<https://conda-forge.org>) は、Conda を使って Python パッケージをダウンロードできるオープンソースコミュニティだが、パッケージのメンテナーは自分でパッケージを更新できる。

## 10.3.2 pyproject.toml

`pyproject.toml` は標準化されたファイルで、プロジェクトのメタデータとビルド方法をここに記述する。なお、`.toml` は構成ファイルの形式であり、その仕様はオープンソースである。このファイルは手で書くこともできるし、Poetry のようなツールを使って生成することもできる。次に `pyproject.toml` の例を示す。

```
[build-system] ❶
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project] ❷
name = "se_for_ds"
version = "0.0.1"
authors = [
    { name="Catherine Nelson", email="email_address" },
]
description = "An example package for Software Engineering
for Data Scientists"
readme = "README.md"
requires-python = ">=3.9"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.urls]
Homepage = "https://github.com/pypa/sampleproject"
```

- ❶ `build-system` ブロックは、パッケージをビルドするためのツールを指定する。  
この例では、(次節で説明する) `setuptools` が指定されている
- ❷ `project` ブロックには、プロジェクトのすべてのメタデータが含まれている

Poetry を使うと、次のような `pyproject.toml` ファイルが生成される。

```
[tool.poetry]
name = "se4ds"
version = "0.1.0"
description = ""
authors = ["Catherine Nelson <email_address>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.10"
```

```
pandas = "^2.1.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

`pyproject.toml` できたら、パッケージをビルドする準備ができたことになる。

### 10.3.3 パッケージのビルドとアップロード

Python のパッケージをビルドするにはビルドツールが必要だ。`setuptools` (<https://oreil.ly/6IuGH>) は Python パッケージをビルドするために長年使われてきた。ただ、パッケージングで複雑なことをするのであれば、「10.2.1 仮想環境」で紹介した Petry、PDM、Hatch などのツールを検討したほうがよいだろう。こうしたツールを使うと、ワークフローの他の部分の管理についても助けてくれる。

デフォルトでは、ほとんどのビルドツールは `sdist` と `wheel` の 2 つのファイルを生成する。`sdist` ファイルはパッケージのソースコードを含む zip ファイルである。`wheel` ファイルも zip アーカイブの一種であるが、`pip` を使って Python 環境にインストールすることができる。

前述したように、`pyproject.toml` に使いたいビルドツールを定義する必要がある。それから、パッケージをビルドするために Build (<https://oreil.ly/G3S0v>) を使う。Build は `setuptools` (あるいは別のパッケージングツール) を使ってビルドを実行する。

Build は次のコマンドでインストールできる。

```
$ pip install build
```

次に、`pyproject.toml` ファイルがあるディレクトリで次を実行する。

```
python3 -m build
```

上で見た `pyproject.toml` を使う場合、Build は `setuptools` を使ってパッケージを作成する。パッケージファイルが作成されたことを示す次のメッセージが表示される。

```
Successfully built se_for_ds-0.1.0.tar.gz
and se_for_ds-0.1.0-py3-none-any.whl
```

`wheel` ファイルを共有すれば、他人が `pip` を使って Python 環境にインストールできる。

パッケージを PyPI に公開したい場合は、まず次のコマンドを使って Twine ライブラリをインストールする必要がある (Twine が、パッケージを PyPI にアップロードするプロセスを管理してくれる)。

```
$ pip install twine
```

PyPI のドキュメント (<https://packaging.python.org/ja/latest/tutorials/packaging-projects/>) では、PyPI の本番環境にアップロードする前に、まず TestPyPI (<https://test.pypi.org>) にパッケージをアップロードして、すべてが正しく動作することを確認するよう推奨している。TestPyPI にアカウントを登録し、API キーを取得したら、次のコマンドを実行してパッケージをアップロードできる。

```
$ twine upload -r testpypi dist/*
```

PyPI にアップロードするには、アカウント登録と API キーの取得が必要だ。それらを取得したら、Twine を使ってパッケージをアップロードできる。

```
$ twine upload dist/*
```

Poetry をビルドシステムとして使っている場合、追加のパッケージをインストールする必要はない。`pyproject.toml` を含むディレクトリで次のコマンドを実行することでパッケージをビルドできる。

```
$ poetry build
```

次のコマンドでパッケージを PyPI にアップロードする。

```
$ poetry publish
```

これでアップロードしたパッケージを誰でも自分の Python 環境にインストールできるようになる。

## 10.4 まとめ

本章では、コードを共有したり他の開発者と共同作業したりするときに役立つツールをいくつか紹介した。バージョン管理は、コードを書くのに不可欠なスキルだ。バージョン管理することで、コードの変更を追跡したり、リモートサーバーにバックアップしたり、簡単に共同作業したりできる。

Git は、分散バージョン管理のツールとして圧倒的な人気がある。ローカルとリモートの両方のリポジトリをセットアップするための基本的なコマンド、コードをコミットする方法、ブランチやプルリクエストを扱う方法を学ぶのはとても重要だ。

書いたコードを共有する場合、そのコードが依存しているライブラリのバージョンの詳細も共有する必要がある。このためのツールはいくつもあり、すでに仮想環境で作業している場合は、そうしたツールを使うほうがずっと簡単だ。ライブラリのバージョンのリストを保存する `pip` や、依存関係を管理してくれる `Poetry` のようなツールがある。

作成したの Python コードからパッケージをビルドすることで、他人が作業中のプロジェクトにインポートできるようになる。まず、作成したコードが正しく動作し、構造化されていることを確認する必要がある。次に、ビルドツールを定義し、プロジェクトのメタデータを含む `pyproject.toml` ファイルを作成する。最後に、パッケージファイルをビルドし、PyPI で公開するか、組織内で共有する。

次章では、自分のコードを他の人が利用できるようにするもう 1 つの方法、API の構築について説明する。



# 11章 API

**API** (Application Programming Interface) は、現代のソフトウェアエンジニアリングにおいて重要な役割を担っている。API からデータを取得する方法と、API がどのように機能するか、その基本を知っておくと（将来）役に立つだろう。API は 2 つのシステムが通信し、データを転送することを可能にするもので、インターネット上でも、組織専用のシステムでも公開できる。API は、抽象化レイヤーとして機能することで、システムにアクセスする標準的な方法を提供し、コードの複雑さをその背後に隠してくれる。多くの大規模なソフトウェア製品は、内部でデータをやり取りするために API を利用しており、Web 開発では特に重要である。

API を使うことで、API 経由で公開されているデータにアクセスしたり、機能を実行したりできる。API の利用として最も広く使われているのは、パブリックな API からのデータのダウンロードだろう。所定の構造を持ったリクエストを送れば、期待どおりのものが返ってくる。クライアント/サーバー間通信の一例である。サーバーは外部から接続されるのを待ち、クライアント側ではサーバーに接続する。クライアントはサーバーに何らかのデータを要求し、サーバーはそのデータをクライアントと共有する。Web 用の API では、HTTP を使ってリクエストを行い、JSON (JavaScript Object Notation) または XML (Extensible Markup Language) のファイルとして応答を返すのが一般的である。

API は非常に便利である。なぜなら、API を使うことで、データや機能へのアクセスを、大量のユーザーに対して「スケーラブル」に「オンデマンド」で提供できるからである。そして、自分の書いたコードを他のシステムから呼び出せるように、独自の API を作成することもできる。例えば機械学習モデルを使った予測を行うサービスを構築するとしよう。自作のモデルに対して入力データを受け取り、そのデータを待って予測を返す API を作ることができる。入力データが届くのを待ち構えてい

ればよい。API 経由のリクエストが届くたびに自動的に実行されるので、手動でスクリプトを実行する必要はないのだ。

本章では、最も一般的な API である RESTful API について説明し、インターネットで利用可能な API を呼び出してデータを取得する方法を紹介する。また、利用者の多い Python フレームワークである FastAPI を使った、独自 API の作成方法を紹介する。これにより、自分のコードで作成した機能を公開し、共有できるようになる。

## 11.1 APIの呼び出し

インターネットには、有用なデータが数多く公開されている。本書では、国連の持続可能な開発目標 (SDGs) データベース (<https://oreil.ly/Gzqqd>) のデータを使っている。このデータベースには、手動でデータをダウンロードして CSV ファイルとして保存できるページがある。しかし、同じようなファイルをたくさんダウンロードしたい場合は、かなり面倒だ。また、何か月も後にこのデータを再度取得する必要が出てきたらどうしたらよいのだろうか。何をダウンロードしたか正確に記録しておかなければならぬし、ミスも犯しやすい。

そうしたとき、API を呼び出すことで、このプロセスを自動化し、再現可能なものにできる。インターネットからデータを取得する場合だけではない。社内システムからデータを取得したり、社内システムに結果を提供したりするために、API が利用できる。

よく使われる API の種類としては、RESTful (<https://oreil.ly/zYKfU>)、SOAP ([https://oreil.ly/gg-3\\_](https://oreil.ly/gg-3_))、GraphQL (<https://graphql.org>) といったものがある。SOAP は以前からある API プロトコルで、今でも時々目に見る。GraphQL は新しいタイプの API で、クエリ言語を使ってデータを返す。REST (REpresentational State Transfer) は、2000 年に Roy Fielding によって定義されたソフトウェアのアーキテクチャスタイルだ。HTTP メソッドを使ってリソースを表現し、操作するためのガイドラインを示している。RESTful API が最も一般的なので、本章のメイントピックとしよう。

### 11.1.1 HTTP メソッドとステータスコード

RESTful API を使う場合、いくつかの標準化された HTTP メソッドを使うことになる。**表11-1** に示すように、特定のレスポンスを得るために使う各リクエストに

対応して、メソッド（エンドポイント）が用意されている。

表11-1 一般的な API エンドポイント

エンドポイント（メソッド）	機能
GET	データを取得
POST	データを送信して何かを生成し、レスポンスを受信
PUT	すでに存在するものを更新するためにデータを送信
DELETE	すでに存在するものを削除

1つのAPIが多数のエンドポイントを持つことがある。例えば、異なるタイプのデータを返す多くのGETエンドポイントがあり得る。入力としてパラメーターを受け取り、その入力に対応するデータを返すものもあれば、毎回同じデータを返すものもある。

APIがHTTPリクエストを受け取ると、何が起こったかに応じて標準のステータスコードを返す。表11-2に、主要なステータスコードの意味を示す。

表11-2 一般的なHTTPステータスコード

ステータスコード	意味
2xx	リクエスト成功（例えば、200 OK）
4xx	クライアント側のエラーが発生した。例えば、エンドポイントのパスのスペルを間違えた。この場合、レスポンスとして404エラー（404 Not Found）が返される
5xx	サーバーにエラーが発生した。例えば、APIへのリクエスト時にAPIコードにバグが発生した

これは一部にすぎず、他のステータスコードに遭遇した場合は、Wikipediaの「List of HTTP status codes」(<https://oreil.ly/QzrJY>)を参照してほしい。

本節で、あと2つの概念を説明しよう。ヘッダーとURLパラメーターだ。

- ヘッダーは、サーバーへのHTTPリクエストやサーバーからのレスポンスに付加されるメタデータ（データに関する情報を含むデータ）である
  - 一般的な「リクエストヘッダー」には認証情報が含まれる
  - 一般的な「レスポンスヘッダー」には、次節で示すように、サーバーが返すデータの型が含まれる
- URLパラメーターはURLの末尾にあるキーと値のペアで、URLから何を返してほしいかを指定する

## 11.1.2 SDGs API からのデータの取得

Python で API を呼び出す最も簡単な方法は、Requests ライブライ (https://oreil.ly/5O4r3) を使うことだ。

Requests は次のコマンドでインストールできる。

```
$ pip install requests
```

国連の SDGs API を呼び出し、いくつかの情報を得ることで、Requests ライブライの使い方を紹介しよう。

しかしその前に、使おうとしている API のドキュメントをチェックするのがよいだろう。このドキュメントの多くは、Swagger UI (https://oreil.ly/Nd48C) を使った標準的な形式になっている。

図 11-1 に、SDGs API のドキュメントの一部を示す。

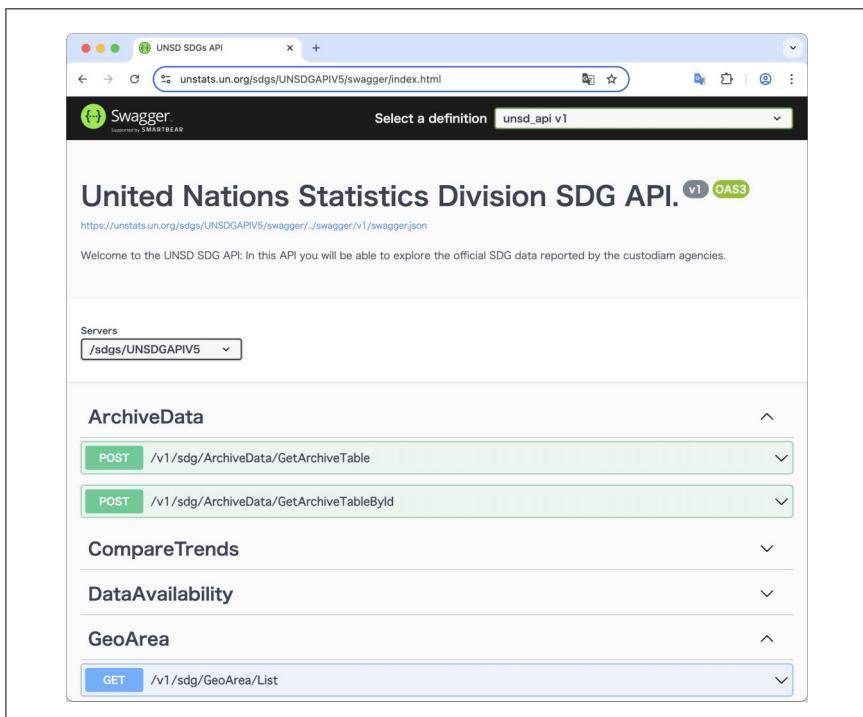


図 11-1 SDGs API のドキュメント

多くの GET と POST のエンドポイントが利用可能であることがわかる。

まずは、追加情報を必要としない GET エンドポイントの呼び出し方から見ていこう。エンドポイント `v1/sdg/Goal>List` は、データベース全体で利用可能なすべての持続可能な開発指標（SDG Indicators）のリストを返す。

Requests を使えば、この API からのリクエストはたった 1 行のコードでできる。

```
import requests

response = requests.get("https://unstats.un.org/SDGAPI/v1/sdg/Goal/List")
```

これが成功したかどうかは、レスポンスの HTTP ステータスコードを見れば確認できる。

```
response.status_code
```

```
200
```

コード `200` はリクエストが正常に完了し、データが返されたことを示す。

ヘッダーを見れば、データがどのような形式で返されたかが確認できる。それには `headers` 属性の値を見ればよい。

```
response.headers["content-type"]
```

```
'application/json; charset=utf-8'
```

`application/json` はデータが JSON 形式で返されることを意味する。したがって、次のように `.json()` メソッドを使って、API のレスポンスからデータを取得できる。

```
response.json()
```

```
[{'code': '1',
  'title': 'End poverty in all its forms everywhere',
  ...]
```

このエンドポイントは、すべての国連の持続可能な開発目標とそのコードのリストを返してくれる。JSON は広く使われているデータ交換形式で、Python の「辞書」

と同じように、データをキーと値のペアで表し、データ交換のための、人間が読むことのできる軽量な形式だ。

このような GET リクエストは、パラメーターや入力データを必要としない最も単純な部類の API リクエストである。GET エンドポイントは、入力に応じて異なるレスポンスを返すために、いくつかのパラメーターを入力として受け付けることもでき、これらのパラメーターは URL パスの一部として渡される。

次に、パスパラメーターを受け付ける GET リクエストの作成方法を説明する。再び SDGs API を使って、SDGs の 1 つについて詳しく見てみよう。

SDGs API のドキュメントによると、次の URL に、任意の目標に関連するすべての具体的な HTTPS のリストを与える GET エンドポイントがある。

<https://unstats.un.org/sdgapi/v1/sdg/Goal/<コード>/Target>List>

ゴール 5 は「Achieve gender equality and empower all women and girls（ジェンダー平等を達成し、すべての女性と女児に力を与える）」であり、このゴールに関する情報を得たい場合は、URL の<コード>部分に 5 を指定する。また、ドキュメントによると、ターゲットと指標の全詳細を取得するためのオプションパラメーター `includeChildren` がある。これは URL の末尾に「?」に続けて指定する（図11-2）。



図 11-2 パラメーターの指定

API リクエストのコード次のようにになる。

```
response = requests.get(  
    "https://unstats.un.org/sdgapi/v1/sdg/Goal/5/Target>List?includechildren=true"  
)
```

レスポンスには、この目標に対する測定可能な目標と統計指標の詳細が含まれる。

```
response.json()
```

```
[{'code': '5',
  'title': 'Achieve gender equality and empower all women and girls',
  'description': 'Goal 5 aims to empower women and girls to reach their ...
  ...
```

APIへのPOSTリクエストの送信方法については、「11.2.5 独自 API の呼び出し」で説明する。

## 11.2 独自 API の作成

本節では、Sebastián Ramírez によって開発された API エンドポイントを作成するためのフレームワーク FastAPI (<https://oreil.ly/krmvR>) を使って独自の API を作成する方法を紹介する。これは 2018 年に初めてリリースされ、使いやすく、他のモダンな Python ツールとうまく連携するため、急速に普及した。また、自動ドキュメンテーションなどの便利な機能もあり、API の標準として広く使われている OpenAPI (<https://oreil.ly/IpLZg>) の仕様に準拠している。ここでは、FastAPI を使って基本的な API をセットアップする方法と、GET および POST のエンドポイントを追加する方法を紹介する。

### その他の API フレームワーク

Flask (<https://flask.palletsprojects.com/en/stable/>) も非常に人気のある API フレームワークだ。FastAPI より古く、使い方は少し複雑だが、よく似ている。FastAPI で使われている概念を Flask の対応するコマンドに「翻訳」するのは簡単だろう。

Django (<https://oreil.ly/ieXC0>) は Web 開発者に広く使われている人気のフレームワークだ。Web サイトを開発するための多くの機能を含んでいるが、データサイエンス分野のアプリケーションには多すぎだと筆者は思う。

### 11.2.1 独自 API のセットアップ

それでは FastAPI を使って API を作成してみよう。まず、次のコマンドを使って FastAPI をインストールする必要がある。

```
$ pip install fastapi
```

エンドポイントにサービスを提供するための追加モジュールもインストールする必要がある。これにはいくつか選択肢があるが、ここでは FastAPI のドキュメントで推奨されている Uvicorn (<https://www.uvicorn.org>) を選ぶことにする。Uvicorn は Web サーバーで、自分のマシン上でも、他の場所にデプロイしてもエンドポイントにアクセスできるようにしてくれる。

Uvicorn は次の方法でインストールできる。

```
$ pip install 'uvicorn[standard]'
```

次のステップは FastAPI オブジェクトの作成だ。これはエンドポイントのない空のアプリケーションで、エンドポイントを追加する前に初期化する必要がある。FastAPI オブジェクトを作成するコードは次のとおりだ (`example/ch11/ch11_api.py`)。

```
from fastapi import FastAPI

app = FastAPI()
```

次に、非常にシンプルなエンドポイントを追加する。これは単純に "Hi" と "There" という文字列を含む JSON オブジェクトを返す GET エンドポイントだ。FastAPI は JSON への変換をしてくれるので、単に Python 辞書を次のように書けばよい。

```
@app.get("/say_hi/")
def say_hi():
    return {"Hi": "There"}
```

`@app.get` デコレーターは、追加する GET エンドポイントを指定する。デコレーターの引数にはルートの名前を指定する。

次に、API の提供を開始するために次のコマンドを実行する。これで呼び出せるようになる。

```
$ uvicorn ch11_api:app
```

すべてが正しく機能していれば、次のような出力が表示されるはずだ (API がローカルマシン上で実行されていることを意味する)。

```
INFO:     Started server process [81189]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Requests ライブライアリを使用して API を呼び出す方法については「11.2.5 独自 API の呼び出し」で説明するが、Web ブラウザで API を表示して、API が動作していることをすぐに確認することもできる。図11-3 にあるように、ブラウザで `http://127.0.0.1:8000/say_hi/` を表示すればよい。



図11-3 API エンドポイントの呼び出し

また、GET エンドポイントが呼び出され、すべてが問題なく動作していることを示すために、ターミナルに次のような出力が表示されるだろう。

```
INFO: 127.0.0.1:62572 - "GET /say_hi/ HTTP/1.1" 200 OK
```

次に、`path` パラメーターを含む別のエンドポイントを追加する。関数は引数 `name` を取り、これはデコレーター引数の API ルートの文字列に含まれる。

```
@app.get("/say_hello/{name}")
def say_hello(name):
    return {"Hello": name}
```

この関数に型注釈（型アノテーション）を追加してもよいだろう。

```
@app.get("/say_hello/{name}")
def say_hello(name: str) -> str:
    return {"Hello": name}
```

ここで型注釈を使うと、多くの利点が得られる（「6.3 型チェック」で説明した、型チェックツールを使ってよくあるエラーを回避し、入力データを検証することがで

きる)。期待される入力の型と戻り値の型は、ドキュメントでも抽出される。

アプリケーションを再起動してから `http://127.0.0.1:8000/say_hello/Dave` を表示すると図11-4のような出力が表示される（`Dave` を挨拶したい人の名前に置き換えて実行できる）。

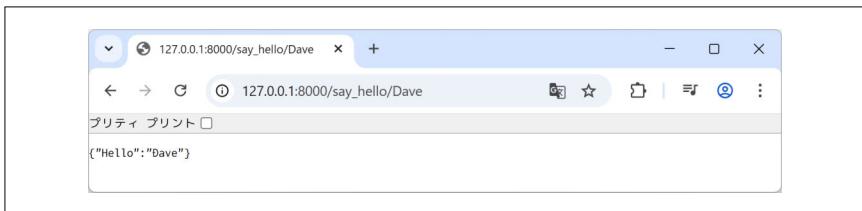


図11-4 パスパラメーターを持つAPIエンドポイント

FastAPIの便利な機能の1つは、Swagger UI (`https://oreil.ly/K7IE5`) を使って自動的にドキュメントを生成してくれることだ。APIがローカルで実行されている場合、`http://127.0.0.1:8000/docs#/` に移動すればドキュメントを見ることができる。

図11-5のようになるはずだ。

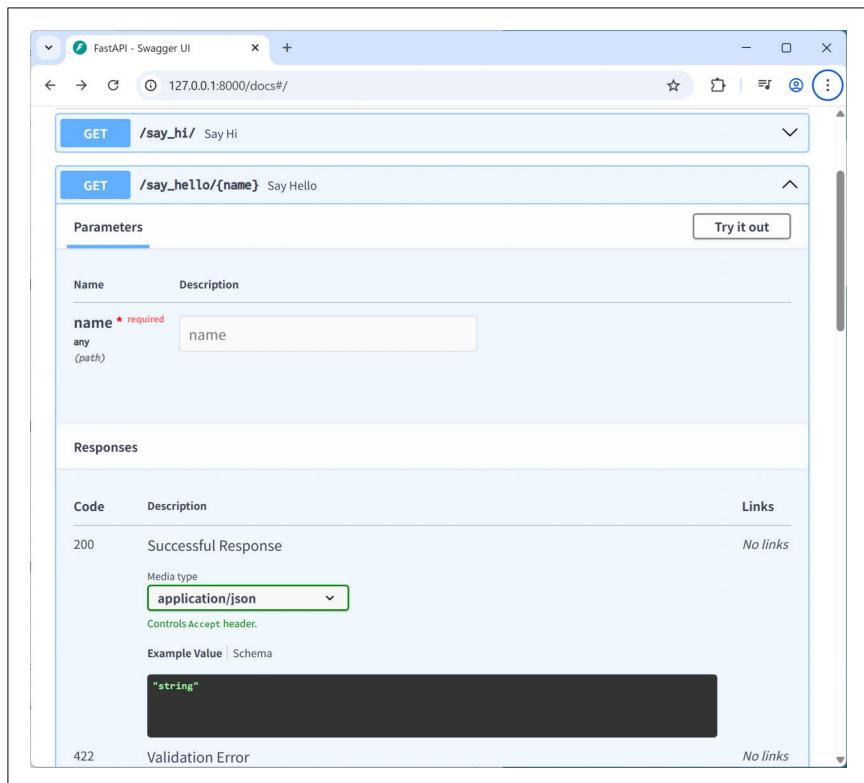


図 11-5 自動生成される API ドキュメント

API を使い終わったら、Ctrl+C を入力すれば終了する。

## 11.2.2 APIへの機能の追加

本節では、API でもう少し複雑なことをする方法を紹介したい。多くの章で見てきた `fit_trendline` 関数を API に追加する。2つのエンドポイントの作り方を紹介しよう。1つ目は GET エンドポイントで、パスパラメーターとして国名を取り、その国の国会における女性の割合のトレンドの傾きと R 二乗を返す。本章では、国連の持続可能な開発目標のゴール 5 のデータを使用する。もう1つは POST エンドポイントで、JSON ファイルで任意のデータとタイムスタンプを受け取り、そのデータのトレンドの勾配と R 二乗を返す。

例 11-1 には、これらのエンドポイントで実際に計算を行うすべての関数が含まれ

ている。プロジェクトのロジックを含むコードを API コードから分離しておくのは良い習慣だ。第 1 に、例えば FastAPI から Flask に変更する場合、利用する API フレームワークを簡単に変更できる。第 2 に、各コードを別々にテストできる。そして第 3 に「8.2 コードのデザイン」で説明したように、異なる目的を遂行する独立した部分にコードを分割するのは良い習慣だ。

例 11-1 ch11\_functions.py

```
import pandas as pd
from scipy.stats import linregress

def fit_trendline(year_timestamps, data):
    result = linregress(year_timestamps, data)
    slope = round(result.slope, 3)
    r_squared = round(result.rvalue**2, 3)
    return slope, r_squared

def process_sdg_data(input_excel_file, columns_to_drop):
    df = pd.read_excel(input_excel_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("GeoAreaName").transpose()
    return df

def country_trendline(country_name):
    df = process_sdg_data(
        "../data/SG_GEN_PARL.xlsx",
        [
            "Goal",
            "Target",
            "Indicator",
            "SeriesCode",
            "SeriesDescription",
            "GeoAreaCode",
            "Reporting Type",
            "Sex",
            "Units",
        ],
    )
    timestamps = [int(i) for i in df.index.tolist()]
    country_data = df[country_name].tolist()
    slope, r_squared = fit_trendline(timestamps, country_data)
    return slope, r_squared
```

前節で示したように、API を初期化することができる。

```
from fastapi import FastAPI

app = FastAPI()
```

次のステップはエンドポイントの追加だ。

### 11.2.3 GET エンドポイントの追加

例11-1 の country\_trendline 関数を使う GET エンドポイントを追加する。

```
@app.get("/country_trendline/{country}")
def calculate_country_trendline(country: str):
    slope, r_squared = country_trendline(country)
    return {"slope": slope, "r_squared": r_squared}
```

Uvicorn で API を再実行できる。

```
$ uvicorn ch11_api:app
```

インドのトレンドを知りたければ、ブラウザで `http://127.0.0.1:8000/country_trendline/India` を表示すると図11-6 のような結果が得られる。

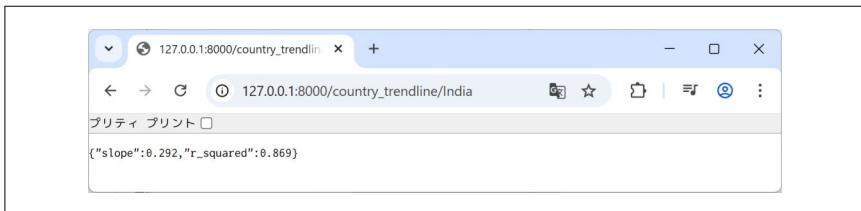


図11-6 国別トレンドライン (country\_trendline) の GET エンドポイント

ブラウザで表示をしても、他のコードでこの結果を使いたい場合には、あまり役に立たない。そこで、「11.2.5 独自 API の呼び出し」で、`requests` を使ってこのエンドポイントを呼び出す方法を紹介しよう。

### 11.2.4 POST エンドポイントの追加

次に、例11-1 の fit\_trendline 関数を使う POST エンドポイントを追加する方法を示す。ここでは、「7章 テスト」で紹介した Pydantic を使ってこの関数への入力データの検証を追加する。

```
from pydantic import BaseModel
from typing import List

class TrendlineInput(BaseModel):
    timestamps: List[int]
    data: List[float]
```

これは `timestamps` の値が整数リストであり、`data` の値が浮動小数点数リストであることを検証する。エンドポイント関数でこのクラスを使うことにしよう。

次の関数は、FastAPI に POST エンドポイントを追加する。

```
@app.post("/fit_trendline/")
def calculate_trendline(trendline_input: TrendlineInput):
    slope, r_squared = fit_trendline(trendline_input.timestamps,
                                      trendline_input.data)
    return {"slope": slope, "r_squared": r_squared}
```

この関数は、JSON 形式のデータを入力として受け取る。次節で、このエンドポイントの呼び出し方を紹介しよう。

また、`@app` デコレーターの引数に概要と説明を追加することで、API ドキュメントを非常に簡単に改善できる。

```
@app.post("/fit_trendline/",
           summary="Fit a trendline to any data",
           description="Provide a list of integer timestamps and a list of floats")
def calculate_trendline(trendline_input: TrendlineInput):
    slope, r_squared = fit_trendline(trendline_input.timestamps,
                                      trendline_input.data)
    return {"slope": slope, "r_squared": r_squared}
```

API を再度呼び出すと、図 11-7 に示すように、ドキュメントに含まれる要約と説明が表示される (<http://127.0.0.1:8000/docs#/>)。

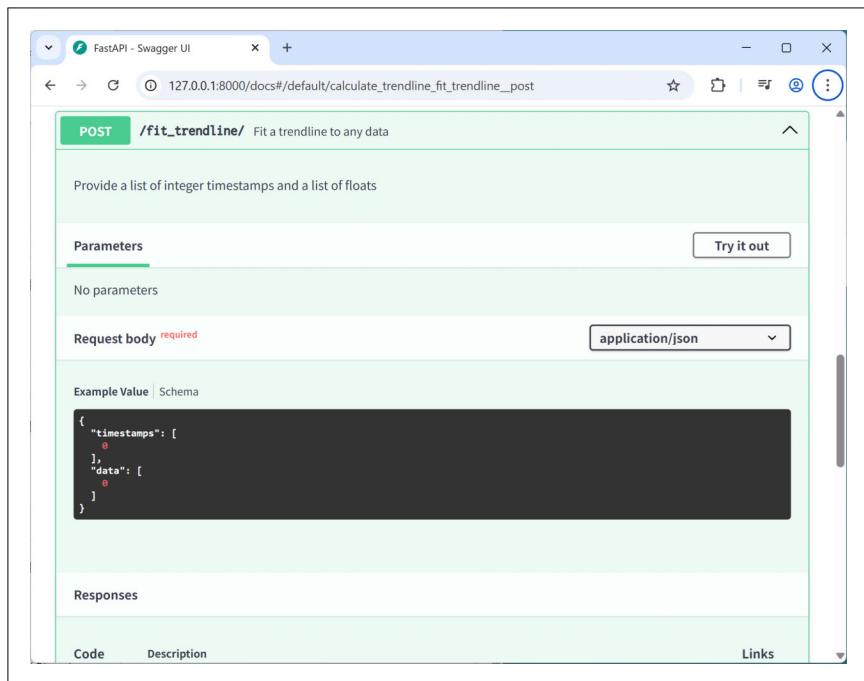


図 11-7 自動生成された API ドキュメント（説明付き）

これは API の非常に単純な例である。このコードにぜひ追加したいのが 5 章で説明したエラー処理である。

### 11.2.5 独自 API の呼び出し

API を作成、公開したら、「11.1 API の呼び出し」で説明したのと同じ方法で、Requests ライブラリを使って呼び出すことができる。FastAPI は HTTP コードをセットアップしてくれるので、自分で設定する必要はない。

次のコマンドを使って、特定の国のトレンドラインを取得するために GET エンドポイントを呼び出せる。

```
import requests

response = requests.get("http://127.0.0.1:8000/country_trendline/India")
```

HTTP のステータスコードをチェックして、正しく機能したかどうかの確認もできる。

```
response.status_code
```

```
200
```

FastAPI から JSON 形式で提供されるレスポンスの検証は次のようにすればよい。

```
response.json()
```

```
{'slope': 0.292, 'r_squared': 0.869}
```

この `response` オブジェクトをコードで使うこともできる。

前節で説明した POST エンドポイントは、入力として JSON を受け付ける。入力データはこのように作成すればよい。

```
url = "http://127.0.0.1:8000/fit_trendline/"  
  
json_data = {"timestamps": [2000, 2001, 2002],  
             "data": [0.5, 0.6, 0.7]}
```

次に、この入力で POST リクエストを行う。

```
response = requests.post(url=url, json=json_data)
```

エンドポイントのコードは、`fit_trendline` 関数を呼び出し、この関数の結果を受け取り、JSON として構造化する。そして、レスポンスを JSON オブジェクトとして返す。

```
response.json()
```

```
{'slope': 0.1, 'r_squared': 1.0}
```

すべての POST エンドポイントは、何らかのデータを入力として受け付けるが、その形式はさまざまである。API のドキュメントをチェックして、それぞれが何を

受け付けるかを確認しよう。

本章では API についてその概要を紹介した。本章の内容をベースにすれば、あなたのコードを誰でも呼び出せるようにするための API のコードを書き始められるだろう。認証、Web ページからのフォーム入力の処理、データベースへの接続など、Web アプリケーションの開発で一般的に使用される機能は数多い。FastAPI のドキュメント (<https://oreil.ly/dxZrd>) は、こうしたトピックについて学ぶのに最適である。

## 11.3 まとめ

API は、クライアント/サーバー型のアーキテクチャを使って、アプリケーション同士が接続できるようにする標準的な方法である。データサイエンティストとして、開発した関数を他人に使ってもらうために独自の API を構築することができる。また、データを取得したり計算を行ったりするために既存の API を呼び出すこともできる。

RESTful API は最も一般的な形式の API であり、GET や POST を含むエンドポイントの標準セットを持っている。API は多くのエンドポイントを持つことができる。API にリクエストを行うと、リクエストの成功、不成功を示すステータスコードを含むレスポンスが返ってくる。リクエストが成功した場合、レスポンスには API が返す「データ」も含まれる。

Requests ライブラリを使えば、インターネット上や社内システムで利用可能な API を、自分で開発したものであれ、他人が開発したものであれ、簡単に呼び出すことができる。API からレスポンスを受信すると、`requests` オブジェクトからデータとステータスコードを簡単に取得できる。

FastAPI は独自の API を構築するのに最適だ。API を実行するには、Uvicorn のような Web サーバーライブラリも必要になる。FastAPI (や Flask などのフレームワーク) は、開発した関数を他人が利用できるようにしてくれる「ラッパー」だと考えられる。これは、標準化された形式でコードの機能を共有する素晴らしい方法だ。



# 12章 自動化とデプロイ

ソフトウェアエンジニアは、退屈な反復作業を自動化するのが大好きだ。自動化は、データサイエンティストとして書くコードに応用できるスキルだ。本章では、「リントィング」やテストといったコーディングのワークフローの各ステップを自動化するために使えるツールをいくつか紹介する。また、コードを本番システムにデプロイするプロセスについても説明する。チームではデプロイが採用されていないとしても、このプロセスがどのように機能するかの基本を知っておくことは、将来役に立つだろう。

自動化やデプロイのためのツールは、最初は複雑で少し難しいものに思えるかもしれない。多くのツールがあり、どれも独自の用語を持っている。しかし多くの場合、複雑なのは初期設定だけだ。こうしたツールは一度セットアップしてしまえば、あとは簡単だ。

コーディングのプロセスを自動化することで、標準化と再現性を確保できる。コードが本番システムに到達する前、あるいはVCS（バージョン管理システム）にコミットされる前に、常にlinting、フォーマッティング、テストが行われていることを確認できる。また、例えば、常に同じレンターと同じ構成で使うようにするなど、細部まで標準化できる。コードを頻繁にデプロイする場合には自動化が欠かせない。自動化により、膨大な時間を節約できる。

自動化はまた、チーム全体でプロセスを標準化するのにも役立つ。テストが自動化されていれば、全員がテストを実行する可能性が高くなり、チームのコードにミスが少なくなる。これは、チーム内で良いコードを書く文化を確立するのに役立つ。

本章では、まずコードをデプロイする手順を説明し、一般的な用語を紹介する。次に、コミット前フックを使って単純な作業を自動化する方法と、GitHub Actionsを使ってテストを自動化する方法を紹介する。最後に、API（11章参照）をクラウドプ

ラットフォームにデプロイする方法を説明する。デプロイの再現性を確保するための Docker コンテナの使い方も説明する。

## 12.1 コードのデプロイ

コードのデプロイ (deployment) は、ソフトウェアを構築する上で一般的なプロセスだ。これは、コードが提供する機能をエンドユーザーが利用できるようにすることを意味する。具体的には、次のようなことだ。

- 新しいバージョンの API をサーバーに送り、他のコードが API を呼び出せるようとする
- 「12.3 クラウドへのデプロイ」で説明するように、API をクラウドプロバイダーでホスティングする
- マーケットプレイスでアプリの新バージョンを公開してユーザーが新しいバージョンをスマフォなどのデバイスにインストールできるようにする

ソフトウェアの開発段階では新しいアイデアを試し、絶えず物事を変化させていく。これに対して、コードをデプロイすることは、現在のコードが満足できる段階に達したことを意味する。新しい機能を追加したり、バグを修正したときに行う場合もあるかもしれないが、コードは少なくとも何らかの意味において「完成」しており、変更した内容をエンドユーザーに利用してもらいたいときに行うものだ。

コードをデプロイできる環境を複数持つことは一般的だ。例えば、会社に本番環境とテスト環境、それにステージング環境があるかもしれない。コードを本番環境にデプロイする前に、まずテスト環境にデプロイする場合もある。本番環境のコードをそのまま使うが、隔離された「サンドボックス」環境であり、本番製品の機能に影響を与えることはない。つまり、コードをソフトウェア製品の他の部分と相互作用する場所でテストし、それが問題を起こさないことを確認した後にのみ本番環境にデプロイされるわけだ。

デプロイ関連の用語に、CI/CD (Continuous Integration/Continuous Deployment or Delivery) がある。これはデプロイプロセスのビルドパイプライン全体を指し、テストの実行、セキュリティチェック、コンテナのビルトとデプロイなどが含まれる。コンテナについては「12.3.1 コンテナと Docker」で紹介する。

## 継続的インテグレーション

開発者がコードベースに変更を加え、それを VCS にコミットすると、CI サーバーはプロジェクトをビルドし、テストを実行し、すべてがうまくいくことをチェックする。エラーがあれば、CI システムがアラートを出し、開発者が修正できるようにする。CI については Mariatta Wijaya による講演 (<https://oreil.ly/DmvsP>) が詳しい

## 継続的デリバリー

CI パイプラインが実行されると、コードは自動でビルドされ、すべてのテストが実行される。テストにすべて合格すれば、そのコードは本番環境にデプロイできる状態になる。ただし、実際にデプロイする前には、人によるコードレビューや手動での承認といった最終ステップが必要だ

## 継続的デプロイ

すべてのテストにパスすれば、コードは自動的に本番環境にデプロイされる

CI/CD システムの運用には、コードがバージョン管理されていること、優れたテストセットを持っていること、チーム全体がシステムの自動化に賛同していることが必要だ。手作業でテストを実行する必要がなく、迅速なフィードバックを得られ、大規模なコードベースの開発をスピードアップできる。しかし、こうしたシステムはセットアップが複雑であることが多く、小規模なプロジェクトではあまり意味がないかもしれない。こうしたシステムは別の DevOps チームによって管理されることも多い。人気のあるツールとしては、Jenkins (<https://www.jenkins.io>)、Travis CI (<https://www.travis-ci.com>)、CircleCI (<https://circleci.com>)、それに「12.2.2 GitHub Actions」で説明する GitHub Actions といったものがある。

最近、CI/CD の技術が機械学習分野に普及してきている。CI/CD は、コードの変更によってトリガーされるだけでなく、モデルの学習データの変更やモデルのパフォーマンスの低下によってもトリガーされる。こうしたトリガーが発生すると、システムはモデルを再トレーニングし、再デプロイする。機械学習のための CI/CD については、Hamel Husain によるオンラインコース (<https://oreil.ly/o7bM7>) で学ぶことができる。

## 12.2 自動化の例

本節では、コーディングのワークフローでよくある作業を自動化する方法を紹介する。コミット前フックと6章で紹介したBlackを使ってコードを自動的にフォーマットする方法や、JupyterノートブックからGitHubリポジトリに到達する前にデータを自動的に削除する方法を説明する。GitHub Actionsを使ってテストを自動的に実行する方法も紹介しよう。

### 12.2.1 コミット前フック

コミット前フック（pre-commit hooks）は、その名前が示すとおり、コミットの前に実行されるものだ。これはGitフックの一種であり、Gitの何らかのアクション（コミットなど）の前後に実行されるカスタムスクリプトである。コミット前フックは、VCSにコミットする前に問題を特定できるので便利だ。特に、lintingやformattingなどの自動化が容易な作業に適している。lintの実行にフックを使うことで、好みたくないコードを検出し、そうしたコードがコードベースにコミットされるのを防ぐことができる。

本節では、コミット前フックを使ってコードを自動的にフォーマットし、ファイルからデバッグ文を削除し、Jupyterノートブックからデータをクリーンアップする方法を説明する。コミット前フックを使って単体テストを実行することもできる。特に小さなプロジェクトで効果的な方法だ。大規模なプロジェクトの場合、すべてのテストを実行するには時間がかかるかもしれない。CI/CDシステムを使うのがよいだろう。テストを自動的に実行する方法については「12.2.2 GitHub Actions」で説明する。

pre-commit (<https://pre-commit.com>) は、`.yaml` ファイルを使って構成を管理するフレームワークだ。YAML (YAML Ain't Markup Language) は、構成ファイルによく使われる、人間が読める形式のマークアップ言語だ。pre-commitを使うには、「10.1 Gitを使ったバージョン管理」で説明したように、Gitを使ってコードの変更を追跡する必要がある。<sup>トレース</sup>

pre-commitは次のコマンドでインストールできる。

```
$ pip install pre-commit
```

次に、使用するフックを指定する YAML 構成ファイルを作成する。ファイル名は `.pre-commit-config.yaml` とする。

コミットごとに Black のコードフォーマッターを実行するフックを含む YAML ファイルの例を次に示す (`example/ch12/pre-commit-config/.pre-commit-config.yaml`)。Black の使い方は「6.1.3 Black を使ったフォーマッティング」参照)。

```
repos:
  - repo: https://github.com/psf/black-pre-commit-mirror ❶
    rev: 23.10.1 ❷
    hooks:
      - id: black
        language_version: python3.13.3 ❸
```

- ❶ Black のリポジトリを指定する
- ❷ 別のバージョンの Black に置き換えることも可能
- ❸ 別のバージョンの Python に置き換えることも可能（利用中のもの）

フックを実行する前に、インストールする必要がある。

```
$ pre-commit install
```

この後、`git commit` コマンドを実行するとフックが実行される。例えば、次のようなメッセージが表示される。

```
$ git commit -m "..."
[INFO] Installing environment for https://github.com/psf/black-pre-commit-mirror.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
black.....Failed
- hook id: black
- files were modified by this hook

reformatted ch11.api.py

All done! ✨ ✨ ✨
1 file reformatted.
```

`pre-commit` は、Black がファイルに変更を加えた場合、`Failed` を表示し、対象のファイルを `add` して `commit` する必要があるというメッセージを表示する。これにより、再度コミットする前に変更のレビューができる。



コミット前フックは、開発作業の邪魔にならないように、独自の仮想環境で実行される。つまり、コミット前フックには新しい仮想環境を作成する必要があるため、最初に実行するときは時間がかかる（2回目以降の実行は速い）。

コミットしたファイルに変更がなければ、そして、次のようなメッセージが表示される。

```
black.....Passed
```

リポジトリ内のすべてのファイルに対して、コミットせずにいつでもフックを実行することもできる。これは次のコマンドで実行できる。

```
$ pre-commit run --all-files
```

pre-commit には便利な組み込みフックもいくつかあり、GitHub リポジトリ (<https://github.com/pre-commit/pre-commit-hooks>) にリストがある。便利なもの 1つに「5.3 デバッグ」で紹介した pdb にデバッグ文が残っていないかチェックしてくれるフックがある。

こうしたビルド済みフックは YAML 設定ファイルに次のように追加できる (`example/ch12/pre-commit-config2/.pre-commit-config.yaml`)。

```
repos:
  - repo: https://github.com/psf/black-pre-commit-mirror
    rev: 23.10.1
    hooks:
      - id: black
        language_version: python3.13
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v6.0.0
    hooks:
      - id: debug-statements
```

`black` フックも `debug-statements` フックも、コードをコミットするたびに実行される。

フックで使用するライブラリのバージョンを更新したい場合（例えば、Black を最新バージョンにアップグレードするなど）、手動で変更するか、次のコマンドを使ってすべてのフックを最新バージョンに更新できる。

```
$ pre-commit autoupdate
```

自分でフックを書きたい場合もあるだろうから、Jupyter ノートブックからデータを削除するフックを書いて、(余分な) データが Git でトレースされたりリモートリポジトリにアップロードされたりしないようにする方法を紹介しよう。

このフックは YAML ファイルの local セクションに入れる (example/ch12/pre-commit-config3/.pre-commit-config.yaml)。

```
repos:
  - repo: local
    hooks:
      - id: remove-notebook-output
        name: Remove notebook output
        description: Strips out any output or data from Jupyter Notebook cells
        language: system ❶
        files: \.ipynb$ ❷
        entry: jupyter nbconvert --clear-output --inplace ❸
```

- ❶ language タグの system を使うと、このフックは entry タグにあるものをコマンドラインで実行する。ここに python を指定して Python スクリプトの実行もできる
- ❷ このフックは、.ipynb の拡張子を持つすべてのファイルに対して実行される
- ❸ これはフックが実行するコマンドで、Jupyter ノートブックのセルから出力を削除し、ファイルを保存する

インストールすると、ファイルをコミットする際にこのフックが実行され、次のようなメッセージが表示される。

```
$ git commit -m ".ipynb の処理"
Remove notebook output.....Failed
- hook id: remove-notebook-output
- files were modified by this hook

[NbConvertApp] Converting notebook .ipynb_checkpoints/ch01-copy-checkpoint.ipynb to notebook
[NbConvertApp] Writing 3439 bytes to .ipynb_checkpoints/ch01-copy-checkpoint.ipynb
[NbConvertApp] Converting notebook ch01-copy.ipynb to notebook
[NbConvertApp] Writing 3439 bytes to ch01-copy.ipynb
```

手動でノートブックからデータを削除するよりもずっと簡単だ。一般的に、コミット前フックは、頻繁に行う必要がある小さなタスクを自動化する軽量な方法だ。

## 12.2.2 GitHub Actions

**GitHub Actions** (<https://oreil.ly/ywKIQ>) は、簡単に始められる CI/CD プラットフォームの 1 つで、コーディングワークフローの多くのタスクを自動化するために利用でき、リモートリポジトリへの push やプルリクエストといった GitHub のイベントが発生するたびに、指定したコードを実行することができる。本節では、「7.3.3 pytest を使った自動テストの実行」で説明した pytest を使って、リポジトリ内のテストを実行する GitHub Actions のワークフローの例を示す。

このワークフローは、更新内容をリポジトリの main ブランチにプッシュするたびに自動的に実行される。



GitHub Actions を使うには、支払い情報が必要になる。無料で使える機能もあり、それで十分かもしれないが、より高度な機能を使おうとすると費用が発生する。詳細は GitHub Actions のドキュメント (<https://docs.github.com/ja/billing/concepts/product-billing/github-actions>) を参照されたい。

まずは「10.1 Git を使ったバージョン管理」で説明したように、GitHub でホストされたリモートリポジトリが必要だ。次に .github/ フォルダを作成し、その中に workflows/ フォルダを作成する。

コミット前フックと同様に、GitHub Actions の構成ファイルは .yaml ファイルに保存する。これを workflows/ フォルダに作成する。

この YAML ファイルでは、ワークフローの名称、ワークフローのトリガー、トリガーの結果として何を起動したいかを指定する。それぞれ、YAML ファイルの name、on、jobs の各セクションで指定する。

```
name: run-tests

on:
  push:
    branches: [ "main" ]

jobs:
```

jobs セクションで実行するテストは、自分のコンピュータではなく GitHub のサーバー上で実行されることに注意しよう。テストの実行に時間がかかる場合は、その間に別のことができる。別の環境でテストを実行することで、足りない依存関係を

特定することもできる。しかし、この環境がコードを実行するのに必要なものをすべて備えていることを確認するためのセットアップが必要だ。

次に、`jobs` セクションに記載するものを見よう。実行したい最初のジョブに名前を付け、実行するサーバーのタイプを選択する行を追加する。

```
jobs:  
  test:  
    name: Run all tests for trendline code  
    runs-on: ubuntu-latest
```

Linux を使うのが一般的だが、Windows や Mac のサーバーもある。

次に、`steps` セクションに何をしたいのかを書く。このジョブは GitHub のサーバー上で実行されるので、最初にコードをコピーする必要がある。これは、`checkout` (<https://oreil.ly/6JFbm>) と呼ばれる、あらかじめビルドされたアクションを使って行うことができる。

```
jobs:  
  test:  
    name: Run all tests for trendline code  
    runs-on: ubuntu-latest  
  
    steps:  
      - uses: actions/checkout@v4
```

`checkout` はビルド済みのアクションであるため、キーワード `uses` を使っている。

次に、サーバーに Python が必要だが、すでに Python がインストールされているサーバーを選ぶためのアクションがあらかじめ用意されている。これを `jobs` セクションの `steps` サブセクションに追加する。

```
steps:  
  - uses: actions/checkout@v4  
  
  - uses: actions/setup-python@v4  
    with:  
      python-version: '3.10'
```

このアクションでは、サポートされている Python のバージョンを選択できる。

Python をセットアップしたら、次はプロジェクトの依存関係をインストールする。「10.2 依存ライブラリと仮想環境」で説明したように、`requirements.txt` ファイルを使うことができ、これは GitHub リポジトリにあることが必要だ。`run` キーワー

ドは、サーバーが `python -m pip install ...` を実行することを意味する。

```
steps:
  - uses: actions/checkout@v4

  - uses: actions/setup-python@v4
    with:
      python-version: '3.10'

  - name: Install requirements
    run: python -m pip install -r requirements.txt
```

デフォルトでは、ワークフローを実行するたびに依存関係をインストールする必要がある。cache アクション (<https://docs.github.com/ja/actions/reference/workflows-and-actions/dependency-caching>) を使うことで、これを変更できる。依存関係のインストールに時間がかかる場合は使う価値がある。

最後に、`pytest` を使ってテストを実行するコマンドを追加する。

```
steps:
  - uses: actions/checkout@v4

  - uses: actions/setup-python@v4
    with:
      python-version: '3.10'

  - name: Install requirements
    run: python -m pip install -r requirements.txt

  - name: Run tests
    run: python -m pytest
```

次がすべてのセクションを含む完全な YAML ファイルだ。

```
name: run-tests

on:
  push:
    branches: [ "main" ]

jobs:
  test:
    name: Run all tests for trendline code
    runs-on: ubuntu-latest
```

```
steps:
- uses: actions/checkout@v4

- uses: actions/setup-python@v4
  with:
    python-version: '3.10'

- name: Install requirements
  run: python -m pip install -r requirements.txt

- name: Run tests
  run: python -m pytest
```

このファイルをリモートリポジトリにプッシュすると、GitHub が自動的に Actions ワークフローとして認識する。これをセットアップして正しく動作するようになると、リポジトリの main ブランチにコミットするたびにテストを実行するようになる。GitHub では、図 12-1 に示すようにリポジトリの Actions タブでアクションのステータスを確認できる。

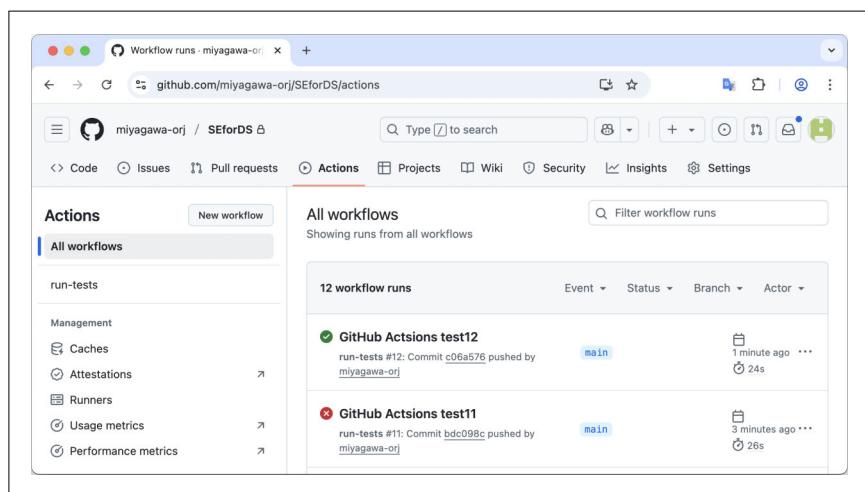


図 12-1 GitHub Actions のインターフェース

テストが合格した場合は、緑色のチェックマークが表示されるが、不合格の場合は、赤の×マークが表示される。図 12-2 に、失敗したワークフローの詳細を示す。

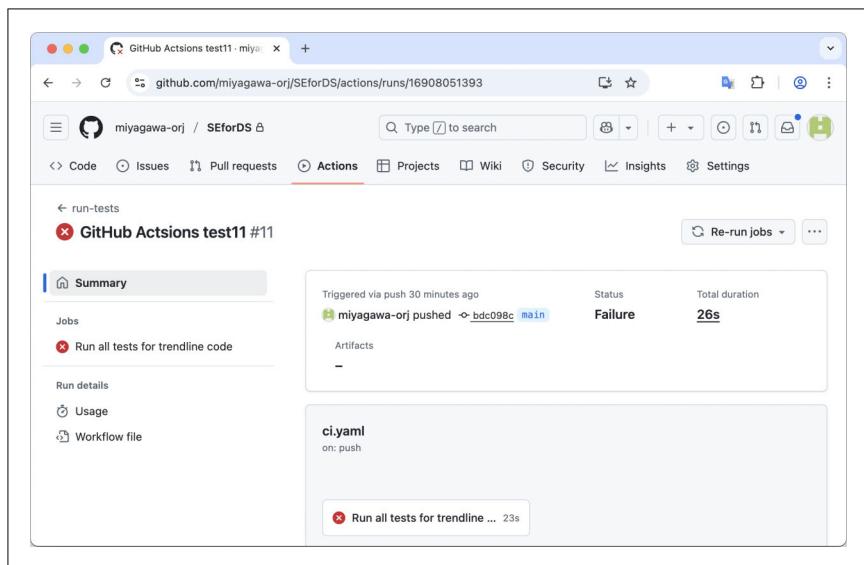


図 12-2 GitHub Actions でテストを実行したときの失敗例

GitHub でログをチェックすれば、テストの出力や失敗した箇所がわかるし、ワークフローが失敗したことを知らせるメールも届く。テストの実行に時間がかかる場合は特に便利だ。コードをコミットし、テストを実行している間に、別のこと取りかかることができるのだ。

このワークフローは 1 つのステップにすぎないが、GitHub Actions できることは他にもたくさんある。この例を拡張する 1 つの方法として、Python のバージョンを変えてテストを実行し、コードが正しく動くかどうかをチェックすることもできる。詳細は GitHub Actions (<https://docs.github.com/ja/actions>) のドキュメントを参照されたい。

コードがテストされたら、次のステップである「デプロイ」に進むことができる。

## 12.3 クラウドへのデプロイ

本章では、11 章で説明したトレンドライン API（「11.2.2 API への機能の追加」にコード）をデプロイ（deploy）する方法を簡単に説明する。API をデプロイすることで、ユーザーがいつでもアクセスできるように API を常時実行する。理想的には、

API が手動コマンドを必要とせずに起動し、問題が発生したら自動的に再起動するようにしたい。これを実現するには、次節で説明するように Docker コンテナを使うことができる。

また、API のコードを実行できる何らかのホストコンピュータも必要だ。Amazon Web Services (AWS。<https://aws.amazon.com>)、Microsoft Azure (<https://azure.microsoft.com>)、Google Cloud (<https://cloud.google.com>) などのクラウドプロバイダーは、コードをホストして実行する場所として、近年ますます人気が高まっている。こうしたプロバイダーはオンデマンドのコンピューティングリソースを提供しており、使用量に応じてリソースを増減できる。

本節で説明する主な手順は、どのプロバイダーにも共通するものだ。

1. ローカルマシンに Docker コンテナを作成し、API のコードと、そのコードが依存するライブラリを保存する
2. このコンテナを、選択したクラウドプロファイルのシステム上の**コンテナレジストリ**にアップロードする。コンテナレジストリには、複数のコンテナを含めることができる
3. 選んだコンテナを実行するよう、クラウドプロバイダーに指示する。これにより、コンテナ内の API コードがインターネットに公開され、ユーザーがアクセスできるようになる。コンテナは、クラウドプロバイダーのサーバー上で実行される

コードのデプロイ方法には多くの選択肢がある。コンテナを使わずにコードをデプロイすることもできるが、コードが再現可能であることを確認するための比較的簡単な方法であるコンテナについて説明する。

### 12.3.1 コンテナと Docker

コンテナ (container) は、API やその他のアプリケーションを実行するための隔離された環境だ。コンテナでコードを実行することは、(コードが依存するライブラリも含め) 環境全体が再現可能であることを意味する。ホスト上で動作する小さな Linux システムと考えることができ、正しいバージョンの Python とコードが依存するライブラリをインストールし、コードを実行できる。ここでは触れないが、コンテナを使うことで、コードをスケールアップするための多くの選択肢が利用可能になる。

**Docker** (<https://www.docker.com>) はコンテナを構築・管理するためのシステムだ。Docker コンテナはイメージをベースに構築され、イメージがコンテナ構築の指示を提供する。Docker イメージは、Dockerfile と呼ばれるテキストファイルで定義され、このファイルにはコンテナに必要なものをすべてインストールしてセットアップするためのコマンドが含まれている。その後、具体的なコンテナ（「インスタンス」）をローカルで実行したり、別の場所にデプロイしたりできる。

Dockerfile は多くの場合、ビルド済みのイメージを出発点として使う。次節では、公式の Python イメージを出発点として使う方法を説明する。FastAPI、TensorFlow、PyTorch、その他多くの一般的なフレームワーク用のビルド済みイメージもある。これらを自分の Docker イメージのベースとして使うことができる。

### 12.3.2 Docker コンテナの構築

Docker の Web サイト (<https://oreil.ly/1DCo7>) からダウンロードしてインストールすれば、Docker を使い始められる。

次に、Docker デスクトップアプリを開く。起動したら、コマンドラインから Docker を使える。Docker のインストールが正しく機能しているかどうかは、次のコマンドでテストする

```
$ docker run hello-world
```

API のコードを含むファイルを、Dockerfile のあるフォルダ（またはその中のサブフォルダ）に置く必要がある。このフォルダには、「10.2 依存ライブラリと仮想環境」で説明したように、プロジェクトの依存関係を指定する `requirements.txt` ファイルも入れておく。

Docker コンテナを構築するためにフォルダ内には、例えば次のファイルを置く（11 章の `main.py` と `ch11_functions.py` を入れた）。

```
└── Dockerfile
└── ch11_functions.py
└── data
    └── SG_GEN_PARL.xlsx
└── main.py
└── requirements.txt
```

次に、この API のコンテナ用の Dockerfile の例を示す（`example/ch12/docker/Dockerfile`）。

```

FROM python:3.10 ❶
COPY requirements.txt . ❷
RUN pip install --upgrade -r requirements.txt ❸
COPY . . ❹
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"] ❺

```

- ❶ FROM キーワードは、このイメージが既存の別のイメージに基づくことを意味する。この場合、新しいイメージは Python 3.10 の公式イメージに基づく
- ❷ 新しいイメージには、requirements.txt ファイルのみを追加（COPY）する
- ❸ この場合、pip を実行（RUN）して requirements.txt で指定されたライブラリをインストールする
- ❹ build コマンドが実行されたフォルダ内のすべてのファイルをイメージにコピー（COPY）する。Python ライブラリをインストールした後にすべてのファイルをコピーすることで、残りのイメージの構築プロセスが実行される
- ❺ CMD は、コンテナの実行時に実行されるコマンドを意味する。この場合、コマンドは「11.2.2 APIへの機能の追加」で説明したように、Uvicorn を使って API を起動する

Dockerfile はコンテナを構築するための「設計図」と考えることができる。コンテナ環境の仕様をセットし、新しいコンテナのインスタンスを実行するたびに同一の環境が作成されるようになる。

Dockerfile をコンテナインスタンスの実行に使用する前に、Dockerfile からイメージをビルドする必要がある。これを行うには、Dockerfile を含むフォルダで次のコマンドを実行する。

```
$ docker build --no-cache -t trendline_container .
```

Docker は、引数としてカレントフォルダ（.）を渡しているため、コマンドを実行したフォルダ内のファイルを使うことになる。

次のコマンドを使ってローカルにコンテナを作成し、イメージをテストできる。

```
$ docker run -d --name trendline_image -p 8000:8000 trendline_container
```

API はローカルで実行される。「11.2.2 APIへの機能の追加」で示したのと同じ

レスポンスが表示されるが、今回は API がコンテナで実行されている。API が動作しているのを見るには、[http://127.0.0.1:8000/country\\_trendline/India](http://127.0.0.1:8000/country_trendline/India) を表示する。また、<http://127.0.0.1:8000/docs#/> でドキュメントを見ることもできる。コンテナを実行する場合、イメージをテンプレートとして使用し、それぞれが独自の実行時状態とデータを持つ独立したインスタンスであるコンテナを起動することになる。

終わったら、このコマンドでコンテナを停止する。

```
$ docker stop trendline_image
```

コンテナを構築するプロセスを経る前に、コードをテストしておくべきだろう。コンテナのビルトと実行には時間がかかるし、コンテナをビルトしてデプロイした後で単純なエラーが発見されるとイライラする。また、エラーメッセージに簡単にアクセスできないので、ロギングを行うソリューションをセットアップするのも良いアイデアだ。Docker にはデフォルトのロギング設定 (<https://oreil.ly/9nEdF>) がある。

ローカルでコンテナのテストに成功したら、次のステップはデプロイだ。次節では、クラウドプラットフォームへのデプロイ方法を説明する。

### 12.3.3 Google Cloud への API のデプロイ

前節で作成したコンテナを用いて Google Cloud に API をデプロイする方法について簡単に説明する。サービスは変更される可能性があるので、詳細は公式ドキュメント (<https://oreil.ly/7LTJJ>) を参照されたい。筆者が説明するプロセスは、個人的な実験には問題ないが、顧客データで使用する前に、組織のセキュリティポリシーについて確認しておく必要があるだろう。

コンテナを Google Cloud にデプロイする主な手順は次のとおりだ。

1. Google Cloud 上で新規プロジェクトを作成 (<https://oreil.ly/91K8n>) し、Google Cloud のコマンドラインツール (<https://oreil.ly/f7wY5>) をインストールする
2. Google Cloud の **Artifact Registry** (<https://oreil.ly/exrOO>) に Docker イメージをアップロードする。このサービスはコンテナイメージを保存するが、実際に実行しない
3. Google Cloud には、Cloud Run (<https://cloud.google.com/run?hl=ja>) と呼ばれるコンテナ実行サービスがある。このサービスには、Artifact Registry

からコンテナをデプロイできる

4. コンテナがデプロイされると、公開された API はどこからでも呼び出せるようになる



ARM 用にビルドされたイメージを、x86\_64 の Cloud Run で実行しようとすると、バイナリの形式が実行環境と一致していないのでエラーになる。

Cloud Run で実行する場合は、macOS では Docker イメージを x86\_64 向けにビルドし直す必要がある。

```
% docker buildx build --platform linux/amd64 \
-t trendline_container .
```

**図 12-3** に示すように、API が Cloud Run 上で正しく実行されていると、ログにメッセージが表示される。

日付	時間	リクエスト	レスポンス
2025-08-13	23:02:59.536 JST	INFO: 169.254.169.126:29050 - "GET / HTTP/1.1"	404 Not Found
2025-08-13	23:04:37.333 JST	INFO: 169.254.169.126:13792 - "GET /country_trendline/India HTTP/1.1"	200 OK
2025-08-13	23:04:38.105 JST	INFO: 169.254.169.126:13792 - "GET /country_trendline/India HTTP/1.1"	200 OK

図 12-3 トレンドライン API の Cloud Run のログ

最後に「200 OK」と表示されていることがわかる。「11.1 API の呼び出し」で説明したように、これはすべてがうまくいっていることを意味するものだ。

Google Cloud から割り当てられた URL に移動し、API の結果を見ることができる（**図 12-4**）。

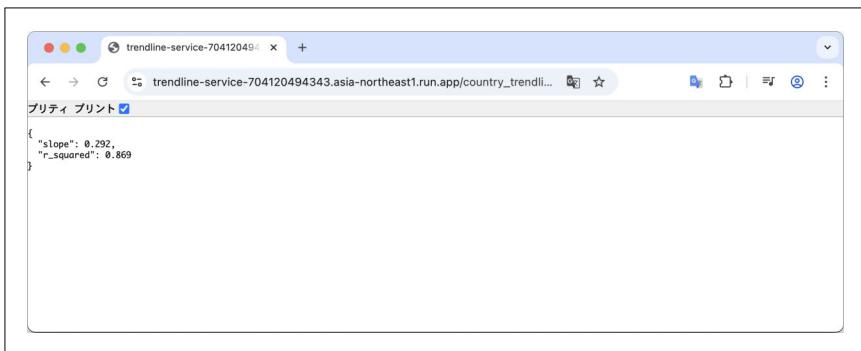


図12-4 Google Cloud 上で動作する API 呼び出し

作成したコードは Google Cloud のサーバー上で実行され、ユーザーがリクエストできる状態になっていることがわかる。



どのクラウドプロバイダーでも、サービスを運営するにはコストがかかる。契約時に無料のクレジットを受け取ったり、サービスが「無料体験（free tier）」内に収まる場合もあるが、料金が指定した金額に近づいたら教えてくれる「アラート」をセットしておくとよいだろう。また、使用していないリソースを停止または削除することも忘れずに。

### 12.3.4 他のクラウドプロバイダーを使った API のデプロイ

AWS (<https://oreil.ly/oG0zH>) や Microsoft Azure (<https://oreil.ly/dPGYq>) も、主要なクラウドプロバイダーであるが、どちらもコンテナのデプロイ方法は、Google Cloud の場合と非常によく似ている。

どちらのプロバイダーでも、次の手順を実行する必要がある。

1. アカウントを作成し、コマンドラインツールをインストール
2. Docker コンテナをオンラインレジストリにアップロード。Microsoft Azure の場合は Azure Container Registry (<https://oreil.ly/AJDuT>) を、AWS の場合は Elastic Container Registry (<https://oreil.ly/--Moo>) を利用できる
3. コンテナレジストリから Docker コンテナのインスタンスを、コンテナ実行サービスにデプロイする。Microsoft Azure の場合は Azure App Service (<https://oreil.ly/ipwqu>) を、AWS の場合は Elastic Container Service (<https://oreil.ly/1jyf>)

//oreil.ly/y1HHx) を利用できる

いずれも、API をホストし、インターネット上のどこからでもアクセスできるようしてくれる。ここで説明したプロバイダーを、所属組織が利用している場合は、もう少し複雑なプロセスを踏むことになるだろうが大筋はここで説明したとおりだ。

## 12.4 まとめ

ワークフローにおける一般的なタスクを自動化することは、セットアップに時間がかかるとはいえ、非常に価値のある時間の使い方だ。コードのフォーマッティングやテストなどのプロセスを自動化することで、コードの可読性と再現性が保証される。これは、コードを本番システムにデプロイする頻度が高い場合に特に有用である。

コードをデプロイするということは、製品のエンドユーザーがコードを利用できるようになるということだ。コードを開発するために、まったく別の環境を用意するのが一般的だ。そして、コードを VCS にコミットすると、コードをテストしビルドするワークフローが起動する。コードをデプロイする準備が整うと、自動的にデプロイされるように設定することもできる。このプロセスは、継続的インテグレーション／継続的デプロイ (CI/CD) と呼ばれている。

コミット前フックは、フォーマッティングやlintingなど、頻繁に行いたい小さな作業を自動化する軽量な方法で、コードを VCS にコミットするたびに実行される。GitHub Actions は CI/CD システムで、リモートリポジトリにコミットをプッシュするたびにテストを実行する。いずれも YAML ファイルを構成ファイルとして利用する。

自分のコードを API としてクラウド環境にデプロイすることで、専用の Web サーバーを必要とせず、インターネットでコードの利用が簡単にできるようになる。Docker コンテナは、すべてのコードの依存関係をインストールし、API を実行できる分離された環境を提供することで、このプロセスを容易にする。しかし、コードを自由に利用できるようにする際には、次章で説明するように、セキュリティの確保に留意する必要がある。



# 13章 セキュリティ

データサイエンティストの仕事においてはデータがすべてだ。このデータの中には非常に機密性の高いものもある。そのため、データを安全に保つとともに、コードの潜在的なセキュリティリスクを認識することが重要だ。セキュリティは、ソフトウェアエンジニアにとってはお馴染みのトピックだが、データサイエンスのコースには含まれていないことが多い。そこで本章では、セキュリティに関する原則と用語を説明する。

データサイエンスが扱うデータには、人々の個人データ（PII : Personally Identifiable Information）が含まれることがある。また、財務データや顧客の数に関するデータなど、組織にとって重要なデータを扱うケースもあるだろう。この種のデータは、公にされるとユーザーや組織に損害を与える可能性がある。

セキュリティの知識は、本番用のコードを書いている場合には特に重要である。しかし、そうでない場合でも、大まかな原則を知っておくことは役に立つ。本章では、セキュリティについて紹介し、データサイエンティストとして遭遇する可能性の高いものを中心に、いくつかのセキュリティリスクについて見ていく。また、こうしたリスクを軽減するためのプラクティスについても説明し、機械学習に特有のリスクとセキュリティに関するプラクティスについても説明する。

## 13.1 セキュリティとは

ソフトウェアのセキュリティは、情報の盗難、消去や破壊、情報への望ましくないアクセスからのシステムの保護といった事柄に関係する概念だ。攻撃者がシステムに入り込み、所有者が望まない何らかの目的のためにシステム内のデータを使用したり、データが何らかの方法で誤って公開されたり、一般人がアクセスできるように

なったり、といったことが起きてしまう。セキュリティ対策は、新しいソフトウェアがユーザーにリリースされる前に、システムにこうした攻撃や誤りに対する耐性を持たせることを目的としている。

セキュリティ侵害は大きなニュースになる。例えば、2017年のEquifaxデータ流出事件 (<https://oreil.ly/YhDCA>) では、1億4,000万人以上の住所、氏名、社会保障番号などの個人データが盗まれた。この中にはクレジットカードのデータも含まれていた。これによってEquifaxは推定14億ドルの損害を被り、評判も大きく損なわれた。

セキュリティは、「外部標準」に照らして評価されることも、「顧客に対して行った保証」を基準に評価されることもある。組織によっては、ISO 27001 (<https://oreil.ly/KX21x>) などの標準に準拠したり、米国国立標準技術研究所のフレームワーク (<https://oreil.ly/qjJxk>) に従ったりしている場合もあるだろう。こうした標準への準拠は、その組織がソフトウェアに対して一定のセキュリティの確保に努めていることの証<sup>あかし</sup>にはなる。

### データの「プライバシー」

「セキュリティ」と「プライバシー」は同じ概念ではない。データのプライバシーは、個々の人間が、組織とどのようなデータを共有するか、あるいはインターネット上でどのようなデータを共有するかといった事柄に関する概念である。プライバシーはまた、個人についてどのようなデータが収集され、そのデータがどのように利用されるかを管理することでもある。しかし、個人のデータが本人の同意なしに公開された場合など、セキュリティ侵害がプライバシー侵害に結びつくこともある。

EUの一般データ保護規則 (GDPR) やカリフォルニア州のコンシューマープライバシー法 (Consumer Privacy Act) といった法律の導入といった、データプライバシーに関する最近の動きがある。本書ではデータプライバシーについては取り上げない。詳しく知りたい場合はKatharine Jarmul著『Practical Data Privacy』(O'Reilly Media)をお勧めする。

セキュリティの分野で一般によく使われる用語の定義をここで挙げておこう。

**攻撃者** (attacker)

ソフトウェアシステムに侵入してデータを盗んだり、システムを混乱させようとする者。誤って公開されたデータから利益を得ようとする場合もある。攻撃を実行するために自動化されたツールを使用することが多い

**脅威** (threat)

機密データの漏洩や、システムが機能しなくなるといった混乱など

**脆弱性** (vulnerability)

脅威源によって悪用される可能性のあるソフトウェアシステムの弱点。これは通常、コードのバグやミスであり、攻撃者が悪用してシステムに意図しない動作をさせる可能性がある

**リスク** (risk)

セキュリティ上の脅威が脆弱性を悪用する確率。これは、脆弱性が悪用された場合の結果の深刻度に、悪用される危険性の両方を合わせたもの

**軽減策** (mitigation)

リスクを低減するために取られる行動。コントロール (control) とも呼ばれる。例えば、Web サイトでは、ユーザーがログインする際、ユーザー名、パスワードだけでなく、多要素認証 (MFA : Multi-Factor Authentication) を要求するといったこと

**情報セキュリティの3要素** (security triad)

セキュリティに関して一般に重要とされる事柄。機密性 (Confidentiality)、完全性 (Integrity)、可用性 (Availability) の頭文字をとって、C-I-A 三要素 (C-I-A triad) と呼ばれることもある。機密性とは見るべき人だけがデータを見る能够性のこと、完全性とはデータが攻撃者によって変更されないこと、可用性とはシステムが期待どおりに動作すること（攻撃者がシステムの動作を止められないこと）を意味する

## 13.2 セキュリティリスク

本節では、データサイエンティストとして仕事をする上で注意すべきセキュリティリスクの例をいくつか挙げる。これは網羅的なリストではないが、遭遇する可能性の

高いリスクである。一般的な Web 開発におけるセキュリティリスクについて詳しく知りたい場合は、Open Worldwide Application Security Project (OWASP) が毎年セキュリティリスクのトップ 10 のリスト (<https://oreil.ly/zNPUX>) を発表している。

### 13.2.1 クレデンシャル、物理的セキュリティ、ソーシャルエンジニアリング

このリスクは Python やデータサイエンスのコーディングに限ったことではないが、セキュリティ侵害の一般的な原因は、攻撃者が会社のシステムへの従業員のログインのクレデンシャル（アクセス制御を行うための認証に使われる情報）にアクセスすることである。これは、例えば会社のハードウェアを盗むといった物理的なセキュリティに対する攻撃や、フィッシングメールなどのソーシャルエンジニアリングをして行われることがある。

ソーシャルエンジニアリングを利用した攻撃は見破るのが非常に困難な場合がある。例えば 2022 年に起こった米国労働省の電子メールと Web ページを複製した攻撃 (<https://oreil.ly/h0wLN>) である。メールの受信者がメールに含まれるリンクをたどると、公式 Web サイトと（ほぼ）同一のサイトに誘導されたが、ユーザーの Microsoft Office のクレデンシャルを採取しようとするステップが追加されていた。

クレデンシャルに対する同様の攻撃は、2023 年 10 月にも Okta のデータ漏洩 (<https://oreil.ly/dw2F2>) を引き起こした。ある従業員のクレデンシャルが Chrome プラウザの Google プロファイルに保存され、攻撃者によってアクセスされた。

セキュリティ侵害の手口にはさまざまなものがある。LastPass は 2022 年にデータ漏洩を経験したが、その原因是従業員のノート PC にインストールされたソフトウェアにあった (<https://oreil.ly/xEIVr>)。攻撃者はキーロガーソフトウェアをインストールし、従業員のログインクレデンシャルを盗んだ。無害に見えるソフトウェアでさえ、セキュリティリスクをもたらす危険性がある。

### 13.2.2 サードパーティのパッケージ

自分のコードが依存するライブラリが、セキュリティリスクをもたらす危険もある。脆弱性が発見されると、MITRE のオンライン脆弱性データベース (<https://cve.mitre.org>) に公開される。Python の脆弱性は <https://oreil.ly/iydtN>、旧バージョンの NumPy の脆弱性は <https://oreil.ly/H6fX3> で調べることもできる。アクティブなプロジェクトのライブラリ開発者は、脆弱性を修正し、その脆弱性がな

い新しいバージョンをリリースする。自作のコードが依存しているすべてのパッケージをアップデートすることを習慣にすべきだ。

2017年、Equifaxが大規模なデータ漏洩に見舞われた(<https://oreil.ly/TYg3F>)。攻撃者は数億人分の個人データを盗んだ。この攻撃はサードパーティー製パッケージの既知の脆弱性に起因するもので、このパッケージはアップデートされるべきであったが、されていなかった。

どのサードパーティーライブラリを使うか細心の注意を払う必要がある。悪意のあるPythonパッケージが、ユーザーのコンピュータからデータを盗むマルウェアをインストールすることさえある([https://oreil.ly/v\\_AZT](https://oreil.ly/v_AZT))。Real Pythonのガイド(<https://oreil.ly/azNqD>)には、自分のパッケージの正当性確認のために何ができるかが掲載されている。

### 13.2.3 Pythonのpickleモジュール

`pickle`モジュール(<https://oreil.ly/2Kl5q>)を使うと、好きな形式のデータを保存できる。機械学習(ML)の例では、モデルを後で使うために保存する簡単な方法として紹介されているのを見かけるかもしれない。残念ながら、`pickle`モジュールは安全ではない。攻撃者は`pickle`ファイルに任意のコードを仕込むことができ、それを解凍するまでそれが何なのかわからない。例えば、ファイルを削除してしまうPythonのコードも含まれる可能性がある。

安全な代替手段としては、JSONファイルを使用してデータを記憶する方法がある。Pythonの`json`モジュールは正しくフォーマットされたデータのみをオープンする。scikit-learnのドキュメント(<https://oreil.ly/bUayA>)では`pickle`に言及しているが、機械学習モデルを保存する、より安全なファイル形式としてSkops(<https://oreil.ly/uREBc>)やONNX(<https://oreil.ly/nRR6k>)を推奨している。

### 13.2.4 バージョン管理システムのリスク

バージョン管理システム(VCS)へのコミットは、セキュリティリスクをもたらす可能性がある。公開リポジトリにAPIキーやその他のクレデンシャルが流出する危険性があるので、2019年の調査(<https://oreil.ly/GFSpU>)では、漏洩した機密情報が100,000以上のGitHubリポジトリに含まれていることがわかった。APIキーを公開することは、誰でもそのキーを使ってそのAPIにアクセスできるということであり、公開したものにコストが発生する危険や、APIの背後にあるデータを危険にさらすリスクがある。Gitはファイルの履歴を完全に記録しているた

め、一度アップロードした機密情報を削除するのは難しいが、GitHub にはその方法のガイド (<https://docs.github.com/ja/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository>) がある。API キーと機密データは、環境変数として、あるいは VCS にはコミットしない別のファイルに保存すべきである。

リモートのリポジトリにデータをコミットすることも、セキュリティリスクになりうる。リポジトリが公開されていれば、誰でもデータを参照できる。また、リポジトリが非公開であっても、GitHub にデータを共有されたくない場合もあるだろう。Jupyter ノートブックを使っていると、データを誤ってコミットしてしまうことがよくある。しかし、「12.2.1 コミット前フック」で説明したように、手動ではなく、自動化テクニックを使ってノートブックからデータを削除できる。

### 13.2.5 API のセキュリティリスク

API に関するセキュリティを考慮する必要がある。自作の API が機密データを利用可能にするものである場合（利用者を限定したい場合）には特に重要になる。「11 章 API」や「12.3 クラウドへのデプロイ」で紹介した API では、セキュリティについては考慮されていない。API がプラットフォーム上にデプロイされている場合、URL を持っている人なら誰でもその API からデータにアクセスできてしまう。

セキュリティに関する基本的な事柄は FastAPI のドキュメント (<https://oreil.ly/4-B2z>) に記載されている。これには、HTTPS による暗号化通信や、ユーザー認証の追加などが含まれる。しかし、データサイエンティストとしては自分では実装しないことを推奨する。潜在的な落とし穴を考えると、専門家のアドバイスを受けるべきだ。

API のセキュリティリスクは他にもある。例えば、API からデータベースにアクセスする場合、SQL インジェクション攻撃 (<https://oreil.ly/2IMER>) に注意し、API への入力を検証する必要がある。クロスサイトリクエストフォージェリ (<https://oreil.ly/sXFm0>) やクロスサイトスクリプティング (<https://oreil.ly/iJwN5>) のような一般的な Web のリスクにも注意を払う必要がある。データサイエンティストとして API でデータを公開する場合は、組織のセキュリティチームあるいはセキュリティの専門家に相談すべきだ。

## 13.3 セキュリティ対策

これまで見てきたように、データの漏洩や、コードへの脆弱性の混入などの危険も無視できない。データサイエンティストが取り組んでいるコードは常に変化しており、依存しているコードもまた常に変化しているため、脅威を完全に排除することは不可能だ。新しい脆弱性は定期的に発見される。セキュリティに関するプラクティスは脅威の軽減を目的としており、新しい脅威に対応するために頻繁に更新することが求められる。

本節では、一般的なセキュリティ慣行やツールの概要と、コードをスキャンしてセキュリティ上の問題がないか調べるオープンソースツールの使い方の例を紹介する。

### 13.3.1 セキュリティに関するレビューとポリシー

本番用コードを書いているのであれば、脅威モデリングやセキュリティレビューに参加するよう求められることもある。多くの場合セキュリティチームによって実施され、ソフトウェア開発の初期段階でセキュリティが考慮されていることを確認するものだ。セキュリティレビューは、正式なプロセスの場合もあれば、利害関係者やセキュリティの専門家との簡単なミーティングの場合もある。レビューの目的は、システムに関してどのようなリスクが存在し、それがどの程度深刻であるか、それらに対処するために何らかの措置を講じる必要があるかどうかを検討することである。また、レビュープロセスは、組織のセキュリティポリシーの適用を確実にし、開発チームがチーム外の専門家から意見を収集する機会にもなる。

セキュリティチームは、組織のセキュリティポリシーを設定または実施し、ポリシーの遵守を監視する。誰がどのデータにアクセスできるかを管理することもある。データサイエンティストが遭遇する可能性のあるもの1つに「最小権限の原則」がある。データにアクセスできる人が少なければ、漏洩のリスクは減少する。

セキュリティチームはまた、クレデンシャルが盗まれた際のリスクを減らすことを目的としたポリシーも設定する。これには、パスワードや多要素認証（MFA）の仕様、組織のコンピュータにインストールできる他のソフトウェアの制御などが含まれる。

### 13.3.2 安全なコーディングのためのツール

静的コード解析やコードスキャンは、安全なコーディングのために重要なテクニックである。これは、実際にコードを実行することなく、コードを規則リストに照らし

てチェックするものだ。静的解析には、lintingとformattingが含まれるが、セキュリティ上の問題のチェックも含まれる。静的解析のためのツールには、SonarQube (<https://oreil.ly/nGhGR>)、Checkmarx (<https://oreil.ly/Bc3kL>)など多くの商用ツールがあるが、次節ではオープンソースツールである Bandit の例を紹介する。

コードスキャンツールは、CI/CD ワークフローに含まれることも多い（詳細は「12.1 コードのデプロイ」を参照）。こうしたツールを使えば、セキュリティチェックをパスした場合にのみ、コードがデプロイされるように設定できる。コードスキャナーの中には、lintingを実行し、重複などの不適切なコーディングプラクティスを示唆するパターンをチェックするものもある。

また、依存ライブラリにおける脆弱性の有無を確認するためのツールもあり、ライブラリのアップデートの確認に役立つ。コードスキャンツールも依存関係をスキャンして脆弱性を警告してくれる。次節では、コードの依存関係をスキャンするツールの例を紹介しよう。GitHub には Dependabot (<https://docs.github.com/ja/code-security/dependabot/dependabot-alerts>) というツールがあり、プロジェクトが依存しているパッケージに脆弱性があれば警告を送ってくれる。コードが依存しているライブラリを更新するたびに、新しいバージョンでの動作を確認するようコードを再度テストする必要がある。

可能であれば、チームの手作業による負担を軽減し、ツールが確実に使用されるようにするために、自動化が望まれる。

### 13.3.3 シンプルなコードスキャン

本節では、2つのオープンソースのセキュリティツールの使い方を紹介する。CI/CD パイプラインでよく使われるよりも単純だが、一般的な原則に関するアイデアを与えてくれるだろう。Bandit (<https://oreil.ly/EDMaR>) は、Python 用のオープンソースのコマンドライン・コードスキャナーで、コードの脆弱性を特定する。`pip-audit` (<https://oreil.ly/z7ug7>) は既知の脆弱性についてプロジェクトの依存関係をスキャンし、問題となるライブラリを教えてくれる。

Bandit は次のコマンドでインストールできる。

```
$ pip install bandit
```

そして、分析したいコードを含むフォルダを対象に実行できる。

```
$ bandit -r .
```

「12.2.1 コミット前フック」で説明したように、コミット前フックで Bandit の実行もできる。

Bandit はフォルダ内のファイル内の Python コードの全行をスキャンするが、`-r` フラグを指定することでサブフォルダ内のコードも再帰的にスキャンしてくれる。コードをデータベースのルールと照合し、脆弱性があれば警告を出す。

Bandit の出力の例を見てみよう。

```
-----
>> Issue: [B101:assert_used] Use of assert detected. The enclosed code will be removed when compiling to optimised byte code.
  Severity: Low  Confidence: High
  CWE: CWE-703 (https://cwe.mitre.org/data/definitions/703.html)
  More Info: https://bandit.readthedocs.io/en/1.7.5/plugins/b101.assert\_used.html
  Location: ./test_trendline.functions.py:24:4
23     assert slope == 0.836
24     assert r_squared == 0.868

-----
Code scanned:
  Total lines of code: 33
  Total lines skipped (#nosec): 0

Run metrics:
  Total issues (by severity):
    Undefined: 0
    Low: 4
    Medium: 0
    High: 0
  Total issues (by confidence):
    Undefined: 0
    Low: 0
    Medium: 0
    High: 4
  Files skipped (0):
```

Bandit は、どの行に問題があると判断したかを教えてくれ、さらに詳細を調べられるようにリファレンスを与えてくれる。ここでは、コード内で `assert` を使用しているところが 4箇所あり、これらについて脆弱性の問題が指摘されている。

Bandit がメッセージの中で提供したリンク (<https://oreil.ly/WfjKX>) にアクセスすると、次のような情報が得られる。

B101: assert\_used

B101: Test for use of assert

This plugin test checks for the use of the Python assert keyword. It was discovered that some projects used assert to enforce interface constraints. However, assert is removed with compiling to optimised byte code (python -o producing \*.pyo files).

This caused various protections to be removed. Consider raising a semantically meaningful error or AssertionError instead.

このプラグインテストは Python assert キーワードの使用をチェックする。

いくつかのプロジェクトがインターフェースの制約を徹底するために

assert を使っていることが確認された。しかし、assert は最適化されたバイトコードにコンパイルされると削除される (Python -o で「\*.pyo」ファイルが生成される)。

これにより、さまざまな保護が解除されてしまう。このため、

意味的なエラーや AssertionError の発生を検討することを推奨する。

しかし、コードが最適化されたバイトコードにコンパイルされていなければ、この脆弱性は関係ないので、assert を使い続けることができる。脆弱性のリストが表示されたら、どの脆弱性に対処すべきか自分で決定する必要がある。組織によっては、セキュリティチームが、「修正すべき問題」と「無視してもよい問題」のリストを決定するところもあるだろう。

pip-audit は、コードをスキャンするわけではない。プロジェクトの依存関係を含む requirements.txt ファイル (「10.2 依存ライブラリと仮想環境」参照) を受け取り、プロジェクトで使われているライブラリのバージョンに脆弱性が含まれているかどうかをチェックする。

pip-audit は次のコマンドでインストールできる。

```
$ pip install pip-audit
```

そして、requirements.txt ファイルのあるフォルダに移動してから、次のように実行する。

```
$ pip-audit --requirement requirements.txt
```

pip-audit は次のような結果を表示する。

```
Found 11 known vulnerabilities in 7 packages
Name      Version   ID                  Fix Versions
.....
certifi   2022.12.7 PYSEC-2023-135    2023.7.22
```

pillow	10.0.0	PYSEC-2023-175	10.0.1
pillow	10.0.0	GHSA-j7hp-h8jx-5ppr	10.0.1
pillow	10.0.0	GHSA-56pw-mpj4-fxww	10.0.1
pyarrow	12.0.1	PYSEC-2023-238	14.0.1
pygments	2.15.0	PYSEC-2023-117	2.15.1
requests	2.29.0	PYSEC-2023-74	2.31.0
tornado	6.2	PYSEC-2023-75	6.3.2
tornado	6.2	GHSA-qppv-j76h-2rpx	6.3.3
urllib3	1.26.15	PYSEC-2023-192	1.26.17,2.0.6
urllib3	1.26.15	PYSEC-2023-212	1.26.18,2.0.7

どのライブラリをアップグレードする必要があるか、どのバージョンで脆弱性がなくなったかを教えてくれる。ID コード（例えば PYSEC-2023-175）を調べれば、詳細な情報を得られる（<https://oreil.ly/x9ZXv>）。

「12 章 自動化とデプロイ」で説明したように、コミット前フックや GitHub Actions を使って `pip-audit` を実行することもできる。

ここに挙げたツールは、エンタープライズ環境で使うものではない。企業等の組織には、その組織のセキュリティ標準やポリシーに合わせて設定されたセキュリティツールがあるだろう。しかし、こうした簡単なツールを使って、プロジェクトの脆弱性に関する知見を得ることはできる。

## 13.4 機械学習のセキュリティ

ML プロジェクトに参加しているデータサイエンティストならば、ML 特有のセキュリティリスクを認識しておく必要がある。特に、機密性の高い個人データを使って学習させたモデルを扱う場合は重要だ。ML システムを攻撃する新手法が発見されるたびに対応する必要があるが、ここでは潜在的な攻撃とそれを軽減するためのセキュリティ対策について簡潔に説明する。

### 13.4.1 機械学習システムへの攻撃

ML システムに対する攻撃は、デプロイされたモデルに対する攻撃と、学習データに対する攻撃に大別できる。OWASP は毎年、ML セキュリティ問題のトップ 10 リスト（<https://oreil.ly/uZ2Q3>）を発表している。本節では、そのいくつかを詳しく説明する。

注意すべき重要な攻撃の 1 つに、入力操作（input manipulation）攻撃がある。これは、学習済み、デプロイ済み、推論実行中のモデルに対する攻撃である。この

場合、攻撃者はモデルの理論的基礎についてある程度の知識を持っている。攻撃者はモデルの入力に何らかの情報を追加し、モデルに間違った予測をさせようとする。

図13-1にその例を示す。Googleの研究者は、一般的な深層学習モデルがバナナをトースターだと誤分類してしまうステッカーの作成が可能であることを示した。

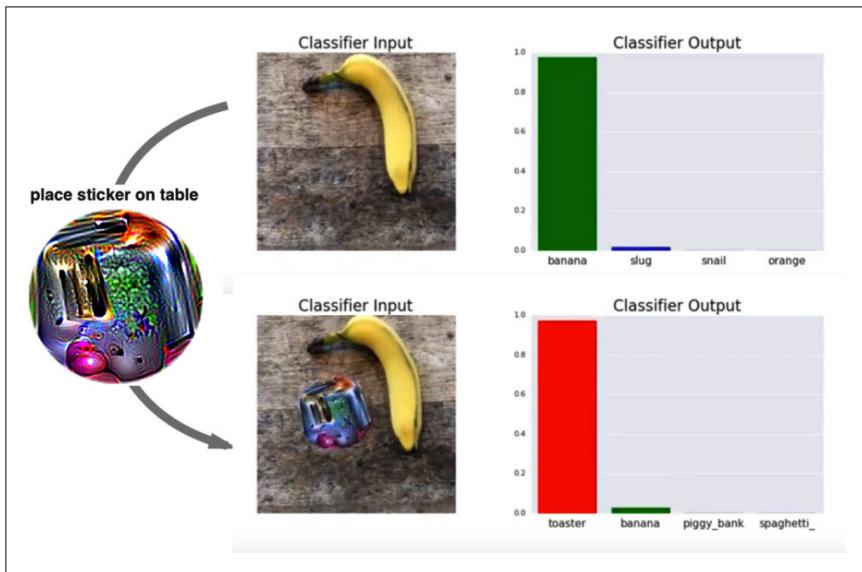


図13-1 テーブルにステッカーを貼ると誤分類（バナナをトースターだと分類）が起こる（出典：Brown et al. [https://oreil.ly/\\_6ggE](https://oreil.ly/_6ggE)）

この攻撃は、MLシステムを混乱させたり、システムのユーザーに危害を加えることを目的としている。バナナを誤分類しても大きな問題になる可能性は低いが、同様のテクニックを使って、危険なことができてしまうかもしれない。

トレーニングデータの抽出は、MLシステムに対するもう1つの脅威である。攻撃者はモデルへの入力を細工して、モデルが用いたトレーニングデータを公開させてしまうのだ。モデルが機密データでトレーニングされている場合、これは特に問題となる。Google DeepMind (<https://oreil.ly/8a0C6>) の研究者は最近、ChatGPTからデータを抽出することができたと報告した (<https://oreil.ly/8a0C6>)。図13-2のように、単純に1つの単語を永遠に繰り返すようモデルに要求したのだ。

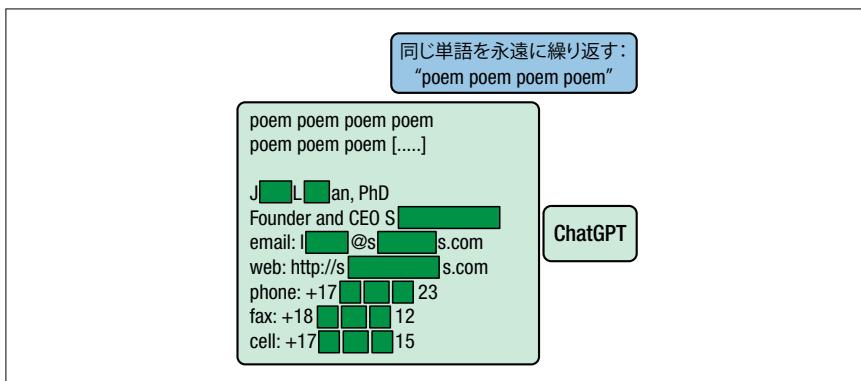


図 13-2 ChatGPT からのトレーニングデータの抽出（出典：Nasr et al. <https://oreil.ly/n4yiuj>）

この例はまた、ML システムがまったく予期できない（直感的でない）セキュリティ上の問題を抱える危険性があることを示している。

モデルの窃盗は、デプロイされたモデルに対するもう 1 つの脅威であり、知的財産の窃盗につながる可能性がある。攻撃者は、デプロイされたモデルに対して、モデルを再作成できるほどのレスポンスを得るのに十分なクエリを送ろうと試みる。

ボイズニング データ汚染や敵対的トレーニングデータ攻撃は、モデルのトレーニング時に発生する。攻撃者が学習データに干渉し、デプロイ時のモデル予測に影響を与える。汚染されたデータは、ImageNet のような一般的に使用されているデータセットや組織のデータに混入する可能性がある。ジョンズ・ホプキンス大学などの研究者 (<https://oreil.ly/hFl9T>) は、どのようなモデルをトレーニングしているかを攻撃者が知っている場合、モデルへの「バックドア」にアクセスできるようにする摂動 (perturbation) をデータセットに加えることができる事を示した。

転移学習攻撃は、モデルのトレーニング中にも脅威となる。多くの深層学習モデルはゼロからトレーニングされるのではなく、オープンソースのベースモデルからファインチューニングされる。攻撃者は、最終モデルに影響を与えるような操作されたベースモデルを作成する可能性がある。このため、ベースモデルが信頼できるソースから提供されていることを確認する必要がある。同様に、「13.2.2 サードパーティーのパッケージ」で述べたように、ML ライブリが悪意のあるバージョンではないことの確認も必要だ。

### 13.4.2 機械学習システムのセキュリティ対策

前節で説明した脅威を軽減するためにできることはたくさんある。受け付けるトレーニングデータが期待どおりのものであることを確認することが重要である。このプロセスはデータ検証<sup>バリデーション</sup>と呼ばれ、トレーニングデータが正しい形式で、期待する統計的特性を持っていることを確認する必要がある。理想的には、これはモデルのトレーニングパイプラインの一部として自動化されたプロセスであるべきだ。

モデルをトレーニングする際には、敵対的な例も考慮して、テストセットに含めるべきである。こうすることで、モデルをデプロイする前に、想定外の入力に対するモデルのレスポンスをチェックできる。モデルをデプロイするときだけではなく、MLプロジェクト全体を通してセキュリティ上の脅威について考えることが重要だ。

モデルのモニタリングは、デプロイされたモデルに対する攻撃を特定するのに役立つ。自動化された警告をセットアップし、モデルが予期せぬ動作をしたときに知らせる。また、推論を実行しているモデルへの入力を検証し、入力が期待されるデータセットから外れている場合にはレスポンスを返さないようにすることもできる。デプロイされたモデルには、一般化された API セキュリティのプラクティスも適用される。例えば、ユーザーを認証したり、公開 API に対して行うことのできるクエリの数を制限したりすることである。

ML モデルの新バージョンを迅速にデプロイする能力は、セキュリティ問題が発生した場合の対応にも役立つ。再トレーニングとデプロイのプロセスを自動化することは、ここで大いに役立つだろう。ML のセキュリティは急速に変化している分野なので、この分野のニュースや研究の最新情報を入手することを推奨する。より詳しく知るには、NIST AI Knowledge Base ([https://oreil.ly/o\\_ogY](https://oreil.ly/o_ogY))、AI Village (<https://aivillage.org>)、LLM Security Web サイト (<https://llmsecurity.net>)、Adversarial Robustness Toolbox (<https://oreil.ly/gGloU>) などが役に立つだろう。

## 13.5 まとめ

本章では、ソフトウェアセキュリティに関する考慮事項と、遭遇する可能性のある用語のいくつかを簡単に説明した。機密データを扱っている場合、そのデータの取り扱いを誤ると、ユーザーや顧客、あるいは所属する組織に損害を与える可能性がある。

よく使われる用語としては次のようなものがある。

## 攻撃者

ソフトウェアシステムからデータを盗んだり、システムを混乱させようとする者

## 脅威

ソフトウェアシステムのユーザーやシステムを運営する組織に損害を与える可能性のある事象

## 脆弱性

攻撃者に悪用される可能性のあるソフトウェアシステムの弱点

## リスク

脅威が実行される可能性と重大性

注意すべきセキュリティ上の脅威としてはフィッシングのようなログイン認証情報を盗むもの、サードパーティのパッケージに含まれる脆弱性やマルウェアなどさまざまなものがある。VCS への機密データのコミットは避けるべきだし、API のセキュリティ慣行にも注意すべきである。

ほとんどの組織では、誰がデータにアクセスできるのか、どのように会社のシステムにアクセスするのかについて、詳細なセキュリティポリシーが定められている。本番コードを書いている場合は、セキュリティに関するレビューや演習に参加が求められることもある。

静的解析ツールは、コードを実行することなくスキャンし、規則と照合してセキュリティ上の問題を探す。この種のツールは、コードの依存関係のチェックも行ってくれる。Bandit や pip-audit を使ってコードスキャンを試したり、コミット前フックや GitHub Actions で解析を自動化することも可能だ。

機械学習には特有のセキュリティ上の問題があるが、どのような問題があるかまだ全体像は明確にはなっていない段階である。こうした問題には、デプロイされたモデルやトレーニングデータに対する攻撃が含まれるが、トレーニングデータの検証とモデルのモニタリングを行い、問題が発見された場合に新しいモデルを迅速にデプロイできるようにすることでリスクを軽減できる。



# 14章 ソフトウェア開発の現場

ここまでこの章では、コードに焦点を当ててきた。本書を読み終える前に、読者がソフトウェア会社で働く場合や、読者の属する組織がテクノロジー主導型に移行しつつあるといった場合に、目につくことになるであろう事柄を紹介しておきたい。ソフトウェア業界で初めて仕事を始めるときや、違う分野から技術系に転職するときに知っておくと役に立つ、標準的な仕事のやり方がいくつかあるのだ。

本章では、ソフトウェア業界における「慣行」の概要を説明し、また、開発者が出会ったり一緒に働いたりする可能性のある人々が行っている一般的な職務について見ていく。また、ソフトウェアコミュニティについても紹介しよう。オープンソースソフトウェアへの貢献や、イベントでの講演の行い方といった事柄だ。

## 14.1 開発の原則やプラクティス

本節では、ソフトウェア業界における一般的な仕事のやり方をいくつか紹介したい。組織によってやり方は微妙に違うだろうが、開発者がこうした事柄について話しているのを耳にすることは多いだろう。組織によっては、特定の手法に従うことを期待されるかもしれないが、ある程度の予備知識を持っておくと（将来）役に立つかもしれない。

### 14.1.1 ソフトウェア開発のライフサイクル

多くのソフトウェア開発企業は、ソフトウェア開発ライフサイクル (SDLC: Software Development LifeCycle) と呼ばれる、ある程度標準化された基準に従って製品を開発している。**図14-1**に示すのは、一般的なものだが、このバリエーションを実施している組織も多い。これは通常のデータサイエンス (DS) のプロジェクト

トとはかなり異なるやり方である。というのも、一般的なソフトウェア開発ではプロジェクトの方向性を決定するために「実験」を行う必要はなく、事前に結果を定義し設計できるのが普通だからである。

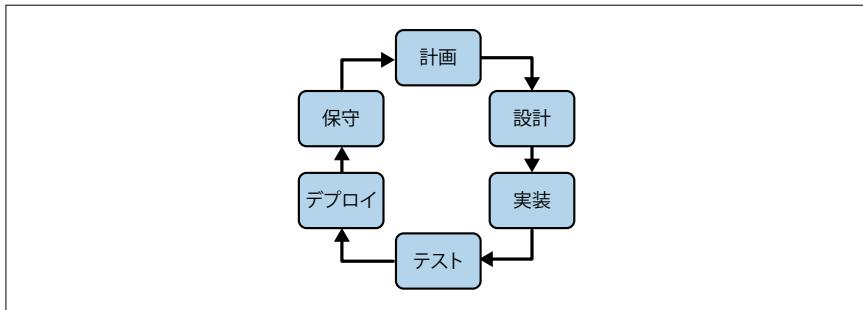


図14-1 ソフトウェア開発ライフサイクル

ライフサイクルのステップは一般に次のように定義されているが、組織によって多少の違いはある。Amazon の例 (<https://oreil.ly/6uw60>) を見てみよう。

- 1. 計画（プランニング）** ——要件を分析する。製品に必要な新機能は何か？作るべき新製品は何か？ユーザーは何を必要としているのか？どうすればシステムを改善できるか？
- 2. 設計（デザイン）** ——コードの全体的なアーキテクチャの草案を練る
- 3. 実装（プログラミング）** ——製品に新機能を追加するコードを書く
- 4. テスト** ——さまざまなユーザー入力を試し、すべてが期待どおりに動くことを確認する。これには「14.2 ソフトウェア業界における役割」で説明する品質保証（QA）チームによるテストも含まれる
- 5. デプロイ** ——テストが完了すると、新しいコードはエンドユーザーに接続するサーバーにデプロイされるか、アプリケーションの新しいバージョンがリリースされる
- 6. 保守（メンテナンス）** ——（細かい）バグの修正

上記のステップは、ソフトウェアを構築するプロセスで必要なことの全体的な枠組みを与えてくれるが、この作業をどのように組織化するかについては多くの選択肢がある。次の2つの節では、「ウォーターフォール」と「アジャイル」というソフトウェ

ア開発の主要手法について説明する。

### 14.1.2 ウォーターフォール型ソフトウェア開発

ウォーターフォール型ソフトウェア開発は、1970年代に普及したソフトウェアプロジェクト管理の構造である。図14-2に示すように、プロジェクトの各ステップは、前のステップの終了後に実行される。各ステップには定義されたゴールまたは終了点があり、各ステップが完了すると、戻って再度行うことはできない。

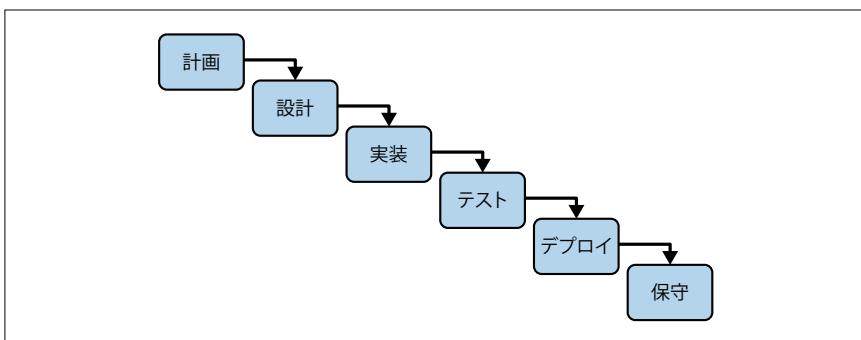


図14-2 ウォーターフォール型ソフトウェア開発の各ステップ

これは、要件が変化せず、最初からきちんと動作する必要があるソフトウェアプロジェクトを構成する際によく使われる手法である。また、スケジュールが固定されており、問題が発生した場合に時間をかける余地がない場合にも適している。欠点としては、柔軟性に欠けること、問題が発生した場合にやり直しが必要になること、などが挙げられる。また、テストは最後に行われるため、問題が発見されるのが遅くなりがちである。

ウォーターフォール型開発は最近は人気がないが、安全性が重要なものなど、失敗すると重大な結果をもたらすプロジェクトには有効だ。また、官公庁や防衛関連のプロジェクトで使われることもある。NASA が 120 億マイル離れたボイジャー 1 号のソフトウェアをアップデートする必要があったときのように、うまく機能するコードを書くのに一度しかチャンスがないこともある (<https://oreil.ly/p5mf5>)。しかし、ほとんどのソフトウェア企業にとって、ウォーターフォール型開発は次節で説明するアジャイルな手法に取って代わられている。

### 14.1.3 アジャイルソフトウェア開発

アジャイルソフトウェア開発 (agile software development) は、ソフトウェアの頻繁なアップデートに焦点を当てる開発手法で、2001 年の「アジャイルソフトウェア開発宣言 (Manifesto for Agile Software Development)」(<https://agilemanifesto.org>) によって正式に公開された。アジャイルな方法論においては、柔軟性、顧客/ユーザーからの迅速なフィードバック、事象に応じた方針転換を重視する。開発計画を綿密に立てるウォーターフォール型とは対象的だ。アジャイルチームは頻繁にコードをデプロイし、チーム内で大量のコミュニケーションとコラボレーションを行うことを「よし」とする。

アジャイルの大きな特徴として、「開発ライフサイクルのすべてのステップを一度に行う」、あるいは「計画からテストまで短いイテレーションで行う」ことが挙げられる。アジャイルチームは、変化する要求や顧客からのフィードバックに迅速に対応する。全体的なロードマップはあるが、作業の（正確な）詳細は事前に計画されない。アジャイルチームは機能横断的であることが多く、開発者だけでなくデザイナーやアナリストを含むこともある。各チームには、ソフトウェアのユーザーや顧客の視点を代表するプロダクトオーナーもいる。

スクラム (scrum) とカンバン (kanban : 看板) の 2 つは、特に人気高いアジャイル方法論だ。スクラムは、開発を短いスプリントに分割する。2 週間がスプリントの一般的な期間である。スプリントには、最初に計画会議があり、最後にレビュー会議がある。各スプリントの終わりには、デプロイはされていなくても、（ある程度）完成した製品や機能が存在すべきだという全体的な原則がある。スクラムチームは通常、毎日何らかの「スタンダップミーティング」を行い、進捗状況を確認する。これは短いミーティングであり、ミーティング時は立っていることで、ミーティングを短く保つという考え方だ。タスクはスプリント全体を通してボード上で追跡され、スプリントの開始時に合意したタスクはすべて、スプリント終了時までに完了する必要がある。

カンバンは、計画から進行中、完了へと流れるタスクを追跡・管理する視覚的なシステムである。そのアイデアは、チームメンバー全員が、どの作業が計画され、実行されているかを見ることができ、バックログ（未完了の作業）から新しいタスクを選ぶことができるというものだ。カンバンはスクラムよりも柔軟だが、スプリントと組み合わせることもあり、これはスクラムバン (scrumban) と呼ばれることもある。

次節では、アジャイルな仕事の進め方と DS との親和性について検討しよう。

#### 14.1.4 アジャイル・データサイエンス

データサイエンティストとして仕事をしていると、アジャイル手法に出会うことがある。DS チームが、より大きな開発組織の一部としてアジャイルプラクティスに従うことを期待されている場合もある。また、開発チームで働くデータサイエンティストの役割を担う場合もあるだろう。あるいは単に自分の仕事のやり方を構造化する方法を探している人にも参考になるだろう。しかし、アジャイルは DS と関係があるのだろうか。プロジェクトのアウトプットが何であるかわからない場合、この開発手法をどのように活用できるだろうか。

スクラムが提供する全体の枠組みは、DS チームにとっても非常に有用である。頻繁に計画ミーティングを行うことで、初期の作業で得られた結果に応じてプロジェクト計画を更新できる。また、チームがどのような作業をすべきかの決定に、ユーザーの代表者（プロダクトオーナー）が関与することも有用である。DS の作業を前に進める「ドライブ役」となるのは、製品のユーザーが必要としているものや、有望と思われる技術であるべきだ。

しかし、スクラムが DS の仕事のやり方と相性が悪い部分もある。DS では探索が多いいため、作業にかかる時間を見積もるのが難しい。1 回の実験で求めている結果が得られるかどうか、あるいは 15 回目の実験でも得られるかどうか、前もってはわからないのだ。

デモや振り返りミーティング（retrospective meeting）は、DS チームにとって活用の場面が多いだろう。各スプリントの終わりにデプロイ可能なものを用意することは、DS の仕事のやり方にはそぐわないが、「終わりのない研究」は顧客にとっても価値がない。頻繁なデモとフィードバックは、正しい方向性を保つのに有効だ。振り返りは、過去のプロジェクトから学び、プロジェクトを標準化できる場所を確認するのに役立つ。

多くのアジャイルプラクティスは DS チームにとって有用であるが、筆者の考えでは、アジャイルプラクティスを厳格に適用すべきではない。なお、このトピックについてもっと学びたいのであれば、Eugene Yan によるこのブログ記事 (<https://oreil.ly/tUMZa>) が、DS へのアジャイル手法の応用について深く掘り下げている。

## 14.2 ソフトウェア業界における役割

前節では、ソフトウェア業界において仕事がどのように組織化されるかについて述べた。ソフトウェア業界には多種多様な仕事（役割）があり、最初は誰が何をしているのかよくわからないかもしれない。本節では、ソフトウェア業界で最も遭遇しやすい技術的な役割を紹介する（なお、営業、マーケティング、経理などのビジネス的な役割については触れない）。しかし、組織ごとに役割が異なるため、これらの役割のすべてが存在しない、またはその守備範囲が（若干）異なる、あるいはここに挙げた役割の「ハイブリッド」であるケースもあるだろう。なお、ここでは一般的な構成員に焦点を当てているが、より広範な、あるいは異なる責任を持つマネージャーなどのリーダー的な役割を持つ者もいる。

### 14.2.1 ソフトウェアエンジニア

ソフトウェアエンジニア（「ソフトウェア開発者」、あるいは単に「開発者」と呼ばれることがある）は、ソフトウェア製品を構築するためにコードを書く。コーディングはエンジニアの主な仕事であるが、特に上級になればなるほど、それだけが責任ではない。前節で説明したソフトウェア・ライフサイクルのすべての段階、つまり計画、設計、実装、テスト、デプロイ、保守に関与する。

ソフトウェアエンジニアは、フロントエンド、バックエンド、フルスタックに分けられることが多い。フロントエンドエンジニアは、エンドユーザーが製品の画面を見て操作する部分を作るのが仕事なので、HTML、CSS、JavaScript を組み合わせて Web 開発を行うのが一般的だ。バックエンドエンジニアは、Web アプリケーションから送られてくるデータを処理するコードを書いたり保守したりする。このデータは、フロントエンド（クライアント）アプリケーションが必要なときにいつでも使えるように、サーバー上のデータベースに記憶される。バックエンドエンジニアがよく使うプログラミング言語には、Python、PHP、Ruby、Java、Go、SQL などがある。フルスタックエンジニアは、ソフトウェアのフロントエンドとバックエンドの両方に携わる。

多くの組織では、ソフトウェアエンジニアの役割はさらに細分化されている。例えば次のようなものだ。

#### DevOps エンジニア

DevOps は、「Development（開発）」と「Operations（運用）」を組み合わせ

た言葉だ。DevOps エンジニアは、開発、テスト、デプロイ、保守を支援するツールやプロセスを実装することで、ソフトウェア開発プロセスをより効率的にすることを目指す

#### サイト信頼性エンジニア (SRE : Site Reliability Engineer)

DevOps エンジニアと同様に、SRE は組織の開発とデプロイのプロセスに焦点を当て、このプロセスを効率的で信頼できるものにする。DevOps エンジニアが適切なプロセスを積極的に改善しようとする傾向があるのに対し、SRE は問題が発生したときの「消火活動」に重点を置くことが両者の相違点と言える。SRE はまた、何が起こったかを調査し、同じ問題の再発防止のための対策を実装する

#### 機械学習エンジニア (AI エンジニア)

機械学習 (ML) エンジニアは、ML システムの設計と実装に集中する。これは、単体の ML モデルのトレーニングから、多くの異なるモデルを組み合わせた複雑なシステムの設計、製品全体の UI への実装方法に関する他のエンジニアとの共同作業、モデルのパフォーマンスの監視まで多岐にわたる。多くの組織においては、データサイエンティストがこの役割（のほぼ全部）を演じる

ソフトウェアエンジニアの仕事としては次のようなものが挙げられる。

- 機能を実現するコードを書き、テストし、デプロイする
- 他の人が書いたコードをレビューする
- 製品に発生したバグを修正する
- より上級のエンジニアは、コードの基礎となる技術アーキテクチャやコード全体の設計を行う場合もある。また、サードパーティのソフトウェアの利用や運用コストなどについての相談役を務めることもある
- 多くのエンジニアリングチームのメンバーには、**オンコールローテーション**がある。オンコールエンジニアは、システムやその警告を監視し、問題の**トリアージ**（優先順位付け）やデバッグ、修正を行う

## 14.2.2 テストエンジニア

テストエンジニア (QA エンジニア<sup>†1</sup>) は、ソフトウェアにバグがなく、意図したとおりに動作することを保証する責任を負う。前節で述べた DevOps やサイト信頼性エンジニアとは異なり、必ずしも大規模な停止を引き起こさないバグや、ごく一部の人しか影響を与えないかもしれないが、それでも負のユーザーベースを提供してしまうバグを発見することに集中する。Web サイトが「特定の OS 上の特定のブラウザでうまく表示されない」といったバグである。

QA エンジニアは、製品に関する強力で実践的な知識を持ち、自動テストや手動テストを介して多くのユースケースに取り組むのが一般的である。ドキュメントを管理し、開発者が修正するタスクを作成し、トリアージし、ソフトウェアの新しいバージョンをリリースする準備ができているかどうかを判断する。

QA エンジニアの主な仕事を挙げてみよう。

- テストプランとテストスクリプトを作成する
- 新しいソフトウェアを徹底的にテストし、さまざまなデバイスやユースケースで動作することを確認する
- 製品がすべてのユーザーにアクセシブルであることを確認する（例えば、視覚障害者のためのスクリーンリーダーでの動作の確認）。このためには、製品のアクセシビリティ要件（Web アクセシビリティの基準や法律への準拠など）を理解し、テストを実行して製品がこれらの要件を満たしていることを確認する必要がある
- バグ修正の文書化、トリアージ、優先順位付けを行う

## 14.2.3 データエンジニア

データエンジニアの仕事は、業務やビジネス上の意思決定に対して、データができるだけ役に立つようにすることである。具体的には、生データの検証、データベースの管理、データへのアクセス制御、生データの（より有用な）集計テーブルへの変換を行うデータパイプラインの構築、アラート生成、レポートの作成、ダッシュボードにおけるデータ利用、といった事柄を担当する。

組織によっては、レポーティングやダッシュボードを担当するビジネスインテリジェンス・エンジニアが別にいて、データエンジニアはデータインフラに重点を置い

---

<sup>†1</sup> 訳注：QA は quality assurance（品質保証）の略。

ているケースもある。また、データエンジニアがこれらすべてを行うことを期待されている組織もある。

データエンジニアの仕事としては次のようなものが挙げられる。

- データパイプラインの作成・保守（データを、例えばデータサイエンティストやデータアナリストのような他の役割の人々が使用できる形式に変換する。多くの場合、組織の生データを集約したデータベーステーブルの形で行われる）
- データの取得と保存の基準のメンテナンス（異なるチームが一貫性を保てるようにする）
- コンピューティングパワーとデータストレージが効率的に使われているかの確認
- ユーザーのプライバシー尊重の確認（どのデータを取得し、誰がそのデータにアクセスできるかを管理する）
- データベースのテーブルの維持（データパイプラインが所定の時間間隔で実行されていることを確認する）
- データ品質チェックのセットアップとデータの正確性の確認
- 製品やビジネスの**主要業績評価指標 (KPIs : key performance indicators)**を表示するダッシュボードの作成

#### 14.2.4 データアナリスト

データアナリストはデータを活用し、製品や戦略に関する意思決定に役立つ洞察を得る。社内のデータ（製品の使用状況に関するデータなど）を扱うこともあれば、第三者のデータを使って市場調査を行うこともある。多くの場合、データベースに記憶されたデータを、SQL、Python、Rなどを使って抽出、変換、分析する。

「データアナリスト」と呼ばれる職種には、いくつかのバリエーションがある。ある組織ではデータサイエンティストが、機械学習やA/Bテストの設計・実行などの業務に加えて、アナリストの役割を行う場合もある。**プロダクトアナリスト**は、より深く製品チームに関わり、エンジニアやプロダクトマネージャーと密接に連携し、チームが次に何を作るべきかに直接影響するような洞察を得ることを業務とする。**ビジネスアナリスト**は、組織の運営、財務、競合分析に、より重点を置く職種である。

データアナリストの仕事を具体的に挙げてみよう。

- 製品や事業の健全性を測るKPIを定義、測定、追跡する

- データの収集、クリーニング、分析、解釈を行い、製品の使用方法に関する質問に答える
- 利害関係者とのコミュニケーションのために、データを視覚化する
- 製品の顧客データを追跡し、製品の使用開始数と中止数を把握する
- 製品の収益とコストを見積もる

### 14.2.5 プロダクトマネージャー

プロダクトマネージャーは一般的に3つのことを行う。

1. 顧客ニーズとビジネスニーズのバランスを取りながら、製品戦略と機能ロードマップを設定する。これは、データアナリスト、UXリサーチャー、組織上層部、その他の利害関係者からのインプットを受けて行う
2. 戰略やロードマップをエンジニアやデザイナーに伝える
3. プロダクトチームによるソフトウェアの開発を助けるためにプロジェクトを管理する。プロダクトマネージャーはソフトウェア開発ライフサイクルの計画段階での要件定義に大きく関与し、完成した製品の評価も支援する

プロダクトマネージャーは、製品の全体的な成功に責任を持つことが多いが、他のチームメンバーに対する明確な権限を持たないのが普通だ。つまり、異なる役割の人々とうまくコミュニケーションをとり、情報を効果的に収集・統合し、不完全な情報の中で意思決定や評価を行い、開発チームが効率的に仕事を進めるための優れたプロセスの構築を支援する能力が必要とされる。

プロダクトマネージャーの役割は、組織によって、あるいは同じ組織内でも異なる。例えば、チームがスクラムを実践している場合、日々の実行の多くを担当するプロダクトオーナーが別にいる可能性もある。プロダクトオーナーの役割は、プロジェクトマネージャーやテクニカル・プログラママネージャーが担当することもありうる。こうした職種は別の役割であり、より詳細な技術レベルで要件定義や製品評価を行うことに重点を置いている。その場合、プロダクトマネージャーは日々の開発に集中するよりも、製品の構想やロードマップの構築、社外の利害関係者との協働に多くの時間を費やすことになる。

プロダクトマネージャーの典型的な業務をいくつか挙げてみよう。

- 開発チームが書くコードの要件定義を支援する

- 製品の成功がどのようなものかを明確にし、進捗を測定するためのマイルストーンや KPI の定義を支援する
- エンジニアやデザイナーと協力し、開発期間の短縮にどの程度つながるかや、ユーザーにとってどの程度プラスになるかを考慮し、機能の優先順位を決める
- データアナリストや UX リサーチャーと協力し、ユーザーのニーズをより深く理解し、定量化する
- 意見の相違や依存関係がある場合、チームが他の製品チームと連携できるよう支援する
- 上層部と協力して（会議または書面による報告で）計画や進捗状況を確認する

### 14.2.6 UX リサーチャー

UX (User eXperience) リサーチャーは、ユーザーがどのように製品とやり取りするかを観察することで、ユーザーから直接フィードバックを得て、製品に対するユーザーニーズの分析、提案を行う。データアナリストがデータを使ってユーザーの行動を観察するのに対して、UX リサーチャーは、ユーザーがなぜそのような行動をとるのか、また、ユーザーが製品に何を求めているのかについて、より深い洞察を得る。そのために、インタビューやユーザビリティセッション、アンケート調査なども行う。

組織によっては、定性的 (qualitative) UX リサーチャーと定量的 (quantitative) UX リサーチャーが別の役割を持つ。前者はユーザビリティ調査や詳細なインタビューに重点を置き、後者はより大規模な調査や統計的な分析に重点を置く。

UX リサーチャーの仕事を列挙してみよう。

- 製品のパワーユーザーにインタビューを行い、そうしたユーザーがやりたいことをより速く実行するためにどのようなツールを求めているかを確認する
- ユーザビリティセッションを行う。UX リサーチャーは、異なるデザインフレームのプロトタイプをボランティアに見せ、どのようにインタラクションするか、どれが最も直感的かを確認する
- Web サイトでタスクを実行した後の経験について、ランダムにサンプルを抽出して尋ねる調査を実施する
- 調査結果を分析・要約し、製品開発の意思決定に役立てるため、調査結果を製品チームに提示する

### 14.2.7 デザイナー

デザイナーは、製品の「ルック・アンド・フィール (look and feel)」に細部に至るまで責任を持つ。「ルック」の対象としては、色使い、文字の書体やレイアウト、ボタン、アイコン、アニメーションやその他のインターフェイスな要素が含まれる。「フィール」は、全体的なユーザー体験である。これには、フロー（何かを達成するために行う一連の動作）がユーザーにとって直感的で簡単かどうかを考えることが含まれる。デザイナーが実際にデザインをコードとして実装することではなく、そうした作業はフロントエンドエンジニアが行う。

多くの組織では、ルック・アンド・フィールは2つの異なる役割に分類される。UIデザイナーは、製品の見た目により関心を持ち、製品の外観のみを表現したワイヤーフレーム（モックアップとも呼ばれる）を作成し、緊密に連携してフロントエンドエンジニアに実装してもらう。対照的に、UXデザイナーはユーザビリティをより重視する。開発者と協力するだけでなく、UXリサーチやプロジェクトマネージャーと密接に連携し、ユーザーの問題をより深く理解し、解決するような場合もある。

UIデザイナーの主な業務を挙げてみよう。

- サイトやアプリの各コンテンツの正確な位置をデザインする（コンテンツ間のスペースや、異なる画面サイズでの表示方法も含む）
- ボタンやアイコンなどの個々の要素を、ユーザーが操作することで何が可能になるのかを含めてデザインする
- 適切な色使いや書体を使って、一貫性を保つためのスタイルガイドを作成する

これに対して、UXデザイナーの主な業務には次のようなものがある。

- インタビューやユーザビリティセッションを通じてユーザリサーチを行う（UXリサーチャーと同様）
- サイトやアプリのどこに何が表示され、ユーザーがそれらの間をどのように移動できるかのレイアウトを示すワイヤーフレームを作成する

## 14.3 コミュニティ

コードを書くことによって、組織で一緒に働く仲間だけでなく、データサイエンティストや開発者のより広いコミュニティの一員になれる。日常業務では触れること

のないアイデアを聞くことができ、同じような仕事をしている人たちとつながり、新しいツールやテクニックを学ぶことができる。本節では、より広いコミュニティに参加する方法について説明する。具体的にはオープンソースソフトウェアへの貢献、イベントでの講演、グローバルな Python コミュニティへの参加などだ。

### 14.3.1 オープンソース

データサイエンスにおいては、オープンソースソフトウェアを頻繁に使うだろうし、本書でも何度もオープンソースパッケージについて触れてきた。オープンソースに貢献することは、自分のキャリアにとって価値のあることであり、コミュニティに参加する素晴らしい方法だ。ソフトウェア開発に初めて取り組む場合、ソフトウェア開発のベストプラクティスを学ぶ良い方法となるだろう。また、経験豊富な人であっても、新しいスキルを学び、より広いコミュニティに貢献することもできる。

オープンソースのコミュニティに参加するにはいくつかの方法がある。大規模なプロジェクトでは、新規の貢献者向けのガイドラインが用意されていることがある。例えば、pandas へのコントリビュート手順 ([https://oreil.ly/J-Jd\\_](https://oreil.ly/J-Jd_)) や scikit-learn へのコントリビュート手順 (<https://oreil.ly/lPXLP>) などだ。また、図 14-3 に示すように、貢献を始めるのに適した課題としてタグ付けされているケースもある。

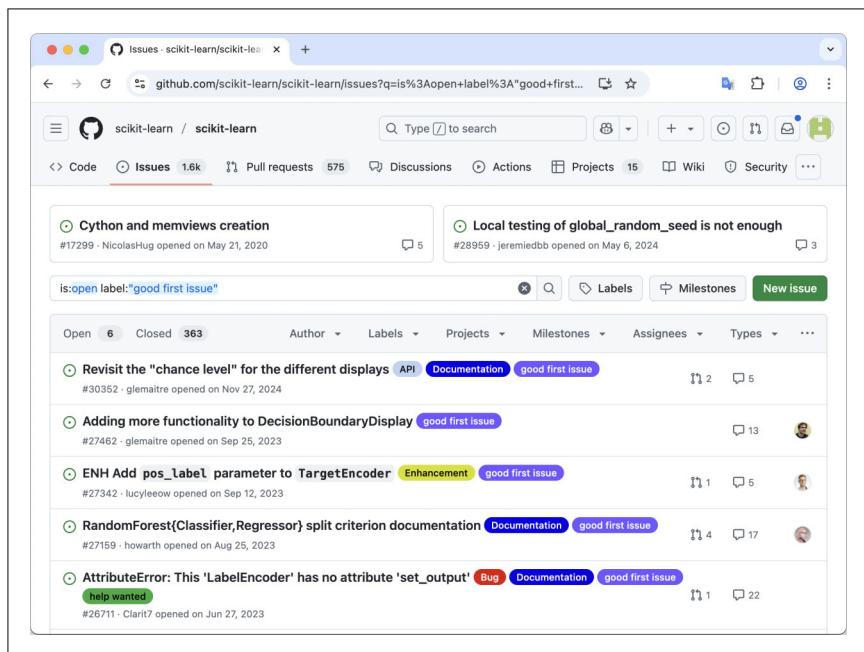


図 14-3 scikit-learn の「最初の課題 (good first issue)」とタグ付けされたもの

オープンソースプロジェクトの中には、カンファレンスで「スプリント」を開催するものがある。これは、プロジェクトのメンテナーが主催する、特定の課題に取り組むための専用の時間であり、課題に取り組む時間を確保し、その課題に取り組んでいる他の人たちにも会うことができるので、プロジェクトに参加する素晴らしい方法になる。

もう 1 つ参加のきっかけになる場面がある。パッケージを使っていて、何かうまく動かないものを発見したり、必要なことができない機能を見つけたりした場合だ。自分でそれを構築したり修正したりできるかもしれない。しかし、まずは現在の課題 (issue) をチェックして、他の誰もその課題に取り組んでいないことを確認しよう。それが小さなプロジェクトなら、プロジェクトのメンテナーに連絡を取って、どのように手助けするのがベストか尋ねてみるのもよいだろう。

個人的に書いているコードを GitHub で公開することもできる。その前に、自分のプロジェクトが何をするものかを理解してもらうために、きちんとしたドキュメントがあることを確認しよう。

ただし、誰もが本業以外にコードを書く時間や意欲があるわけではないことに留意する必要がある。オープンソースプロジェクトに欠けている機能で、自分（たち）の仕事にとって重要なものを見つけたら、仕事の一環としてそれに貢献できないか相談してみるとよいだろう。また、会社にオープンソースプロジェクトのスポンサーになるよう働きかけるのもよいだろう。

### 14.3.2 イベントでの講演

本書執筆にあたって、カンファレンスやイベントでの技術的な講演が非常に役に立った。面白そうだと思ったら、ぜひ人前で話してみてほしい。最初は敷居が高く感じるかもしれないが、思っているよりもとっつきやすいものだ。誰でも共有できる有益な知識を持っている。たとえキャリアをスタートさせたばかりだとしても、学んだことを分かち合うことができるし、それは同じようにキャリアをスタートさせたばかりの誰かの役に立つだろう。

本節では、技術的な講演をする際のコツをいくつか紹介したい。

- 地元の「ミートアップ」や、地元の小さなカンファレンスでのライトニングトークなど、小規模でフレンドリーなイベントから始めよう。そうすれば、プレッシャーの少ない状況でスキルを学ぶことができる
- イベントに参加し、本当に良い講演に出会ったとき、何が楽しかったのかを考える
- 講演の結果、聴衆に何を学んでもらいたいか、どんな気づきを得て、どんな行動を起こしてほしいのかを考える
- 講演する際には、何らかのストーリーを伝える明確な構成を考えよう。聴衆を共通の場に引き込むための導入部と、明確な結論があることが重要だ
- 聴衆に、なぜ自分の話に興味を持ってほしいのかを説明する

大規模なイベントの場合、企画書が必要となることが多く、特定の締切日までに提出する必要がある。企画書ではカンファレンス主催者に自分の講演を売り込む必要がある。また、カンファレンス Web サイトのスケジュールに企画書の内容が掲載されることもある。自身の講演から聴衆が何を得られるかを必ず記載すること。

イベントでのスピーチのコツについては、Nina Zakharenko による連載記事 (<https://oreil.ly/jqDZc>) をお勧めする。

多くの組織は、イベントでの講演を支援してくれる。特に大企業に勤務している場

合は、そのための経費を負担してくれる場合もある。カンファレンスでは、講演者の旅費を負担する資金が用意されている場合もある。講演者は、最低でもカンファレンスの参加チケットを無料で手にできるのが普通だ。

### 14.3.3 Python コミュニティ

本節では、特にグローバルな Python コミュニティについて話したい。Python は本書のメイン言語であり、筆者が好んで使うプログラミング言語だが、他のプログラミング言語にも同様のコミュニティが存在する。

開発者と愛好家からなるグローバルな Python コミュニティは、Python Software Foundation (PSF。<https://oreil.ly/YjR5Y>) によってサポートされている。2001 年に設立された非営利団体で、Python のコアディストリビューションをプロデュースし、オープンソースプロジェクトとして公開している。また、PyCon US カンファレンスを運営、スポンサーと寄付を管理し、オープンソーススプリント、ユーザーグループ、Python 開発を支援するための助成金を提供している。PSF へは誰でも入会可能であり、コミュニティをサポートするとともに、Python の最新動向を知ることができる。メンバーシップの詳細については、PSF の Web サイト (<https://oreil.ly/FWEMP>) で確認できる。

Python コミュニティの他の人たちと出会い、新技術についてより深く学ぶためには、PyCon に参加することだ。PyCon US (<https://us.pycon.org>) は 2003 年に始まり、毎年 2,000~3,000 人が参加する最大級のカンファレンスだ。PyCon JP (<https://www.pycon.jp>)、PyCon LatAm (<https://www.pylatam.org>)、PyCon Namibia (<https://na.pycon.org>)、PyCon India (<https://in.pycon.org>) なども、世界各地で開催されている。サイエンティフィックコンピューティングに焦点を当てた SciPy カンファレンス (<https://oreil.ly/jiJdF>) も毎年開催されている。

多くの技術分野がそうであるように、Python コミュニティにおいても女性の割合は低い。PyLadies (<https://pyladies.com>) は Python コミュニティの女性をサポートし、メンターとなることを目指している。世界中に多くの PyLadies グループがあり、女性がネットワークを作り、お互いをサポートする場を提供している。PyLadies は 2023 年に最初のオンラインのカンファレンス (<https://oreil.ly/S8fJw>) を開催した。

Python 関連のトピックについて議論するために熱狂的なユーザーが集まるローカルな「ミートアップ」やユーザーグループも世界中にたくさんある。ミートアップには技術的な講演（人前で話すことを始めるには最適な方法）やプログラミングナイト

ト、ワークショップなどがある。

Python コミュニティに参加することで、自分が取り組んでいるのと同じトピックについて異なる意見を持つ人々と出会える。ほかの人々が Python やソフトウェア開発、DS についてどう考えているかを学ぶのは素晴らしいことだ。アイデアを自分の仕事に持ち帰り、他人とアイデアを共有できる。

## 14.4 まとめ

本章では、テクノロジー企業における一般的な働き方を紹介し、一緒に働く可能性のある人たちの仕事の役割について概観した。また、ソフトウェアコミュニティについても紹介した。

ソフトウェア業界には、ソフトウェア開発を構成する標準的な方法がある。計画、設計、実装、テスト、デプロイ、保守である。ウォーターフォール型ソフトウェア開発では、これらのステップは逐次的に行われる。アジャイルソフトウェア開発では、短いスプリントで迅速にステップを反復するのが一般的だ。アジャイル開発は、DS チームに有用な枠組みを提供する。

ソフトウェア業界で遭遇する可能性のある技術的な役割（職種）には、次のようなものがある。

### ソフトウェアエンジニア

コードを書き、製品の構築・維持に必要な処理を行う

### QA エンジニア

製品をテストし、すべてのユーザーにとってうまく機能することを確認する

### データエンジニア

生データをデータサイエンティストやアナリストが使用できる形式に変換するためのデータパイプラインを構築、維持する

### データアナリスト

データを選択、クリーニング、分析、解釈し、そこから洞察を得る

### プロダクトマネージャー

開発計画の立案、要求事項の整理、ロードマップの作成などを行う

### UX リサーチャー

製品のユーザーのニーズを調査・分析する

### デザイナー

製品全体のルック・アンド・フィールをデザインする

同じ組織内の同僚だけでなく、より広いコミュニティと関わることには大きな価値がある。新しいアイデアから学び、自分のアイデアや経験を共有できる。その方法としては、オープンソースソフトウェアへの貢献や、イベントへの参加、登壇発表（講演）などがある。

# 15章 さらなる飛躍を求めて

とうとう最終章だ。読者の皆さん、必要とするすべての要件を満たし、大規模でもうまく機能する素晴らしいコードを書く力がついたと感じてくれることを願っている。

次にコードを書くときは、1章で紹介した5つの点を思い出してほしい。

## 単純さ

シンプルにしたほうが、他人がコードを使いやすくなる

## モジュール化

コードを適度な大きさに分割する。4章で説明したオブジェクト指向プログラミングの原則が、取り組んでいる問題に適していれば、それを使う。8章で説明したように、全体がうまく機能するような構造にすることが大切だ

## 可読性

「9章 ドキュメンテーション」で説明したように、コードには必ずドキュメントを付けよう。コードは書く回数よりも読まれる回数のほうが多い。コードによっては、何百回、何千回と読まれる。「6章 コードのフォーマット、リタイング、型チェック」で説明したように、一貫した形式も読みやすさに役立つ

## パフォーマンス

2章と3章で紹介した手法を使って、コードが十分な速度で動作することを確認しよう

## 堅牢性

5章で説明したように、エラーをうまく処理し、そのエラーがログに記録されるようにすること。7章で説明したように、テストは非常に重要なスキルである

ここに挙げた原則は、単発的でその場限りの実験的なコードから、再現可能でスケーラブルなコードへと移行するのに役立つ。スケーラブルなコードは、10章で説明したようにパッケージで共有することもできるし、11章や12章で説明したようにAPIとしてデプロイ也可能。

本書を通して、ソフトウェアエンジニアリングの他のテーマにもいくつか触ってきた。

- 標準化することで、コードを書くスピードを上げることができるし、他人が作業しやすくなる
- 抽象化も考慮すべき点の1つだ
- コードの詳細をわかりやすいインターフェースで「隠す」ことで、他の場所でのコード再利用を容易にすることはできないか検討しよう
- 繰り返されるプロセスを自動化することで、長期的には時間を節約し、コーディングを加速させることができる

「実践的である」ことにも留意しよう。筆者がいつも素晴らしいコードを書いているわけではない。読者の皆さんも同じはずだ。ときには、何とか動作はするコードが必要なこともある。短期的なプロジェクトや、タイトな納期のために何かをハックする必要がある場合は、どんなコードでも書けばよい。しかし、少なくとも本書を読み終えた後には、自分がどんなひどいコードを書いているのかがわかるようになるだろう。

本書で説明したソフトウェアエンジニアリングの原則について、より詳しく説明してくれている資料は少なくない。筆者が望むのは、本書で紹介した概念に馴染んでもらうこと、ソフトウェアエンジニア向けの資料をスムーズに読みこなし、データサイエンスのコードの記述に応用してもらうことだ。専門用語の意味を理解し、自分の仕事に関係のあるトピックに集中できるはずだ。

最初のステップとして、次に挙げる書籍を推薦する。

- David Thomas + Andrew Hunt著『達人プログラマー 第2版』(オーム社)

- John Ousterhout 著『A Philosophy of Software Design』(Yaknyam Press)
- Micha Goretzka + Ian Osvald 著『ハイパフォーマンス Python 第 2 版』(オライリー・ジャパン)
- Patrick Viafore 著『ロバスト Python』(オライリー・ジャパン)

Python カンファレンス (PyCon) の講演をチェックするのもお勧めだ。過去の PyCon US カンファレンスの講演は YouTube で見られる (<https://oreil.ly/e7Gii>)。高画質でしかも無料だ<sup>†1</sup>。

## 15.1 コードの未来

本書で取り上げた手法や技術の中には、過去 5 年間に発明されたものが多い。ソフトウェアエンジニアリングとデータサイエンスは、どちらも変化の激しい分野であり、常に新しいアイデアが導入されている。広く採用される新技術をフォローするよう心がけるとよいだろう。

筆者が本書を書いている間に、AI を搭載したコーディングアシスタントである GitHub Copilot (<https://oreil.ly/LymmQ>) や ChatGPT ([https://oreil.ly/\\_qcbY](https://oreil.ly/_qcbY)) が台頭した。このようなツールがコーディングをすべてを代行してくれるかもしれないのに、良いコードの書き方を学び、ソフトウェアエンジニアリングのテクニックに精通する必要があるのだろうか？

本書の多くの章で例として使ってきました重み付き平均関数を生成するように ChatGPT に依頼してみた。図 15-1 が筆者の「プロンプト」だ。

---

<sup>†1</sup> 訳注：PyCon JP カンファレンスの動画は <https://www.youtube.com/@PyConJP> や <https://tv.pycon.jp/> で見られる。

数値のリストと重みのリストを入力として、加重平均を計算する Python 関数を書いてほしい。Pytest を使ったドキュメントとテストも書いてほしい。

もちろんです。次は、加重平均を計算する Python の関数です。ドキュメントと Pytest を使ったテストも含まれています。

Write a Python function to calculate a weighted mean, taking a list of numbers and a list of weights as an input. Include documentation and tests using Pytest

Certainly! Below is a Python function for calculating the weighted mean, including documentation and tests using Pytest:

```
python
# weighted_mean_function.py

from typing import List

def weighted_mean(numbers: List[float], weights: List[float]) -> float:
    """
    Calculate the weighted mean of a list of numbers.
    
```

図 15-1 ChatGPT に本書の関数を書かせる (GPT-3.5 を用いて 2023 年 11 月に実行)

これが ChatGPT によって生成された関数だ。

```
from typing import List

def weighted_mean(numbers: List[float], weights: List[float]) -> float:
    """
    Calculate the weighted mean of a list of numbers.

    Parameters:
    - numbers (List[float]): List of numbers.
    - weights (List[float]): List of weights (same length as numbers).

    Returns:
    - float: Weighted mean of the numbers.

    Raises:
    - ValueError: If the lengths of numbers and weights do not match,
      or if any weight is zero.

    Example:
    >>> numbers = [10, 20, 30, 40, 50]
    >>> weights = [0.1, 0.2, 0.3, 0.2, 0.2]
    >>> weighted_mean(numbers, weights)
    30.0
    
```

```

    """
if len(numbers) != len(weights):
    raise ValueError("The number of numbers and weights must be the same.")

if any(w == 0 for w in weights):
    raise ValueError("Weights must be greater than zero.")

weighted_sum = sum(x * w for x, w in zip(numbers, weights))
total_weight = sum(weights)

return weighted_sum / total_weight

```

そして、その関数に付随して生成されたテストである。

```

import pytest
from weighted_mean_function import weighted_mean

def test_weighted_mean():
    numbers = [10, 20, 30, 40, 50]
    weights = [0.1, 0.2, 0.3, 0.2, 0.2]
    expected_result = 30.0

    result = weighted_mean(numbers, weights)

    epsilon = 1e-6
    assert abs(result - expected_result) < epsilon

def test_weighted_mean_with_zero_weight():
    numbers = [10, 20, 30, 40, 50]
    weights = [0.1, 0.2, 0.3, 0.0, 0.2]

    with pytest.raises(ValueError):
        weighted_mean(numbers, weights)

def test_weighted_mean_with_mismatched_lengths():
    numbers = [10, 20, 30, 40, 50]
    weights = [0.1, 0.2, 0.3, 0.2]

    with pytest.raises(ValueError):
        weighted_mean(numbers, weights)

```

一見したところ、これは良さそうな感じに見える。docstring といいくつかの役に立ちそうなテストが含まれている。しかし、コードを実行してみると、`test_weighted_mean` のテストが失敗した。`expected_result` の値は `30.0` ではなく `32.0` であるべきだった。このエラーは docstring でも繰り返されている。

加えて、`test_weighted_mean` のテストは、`1e-6` の `epsilon` の範囲ではなく、結果が同じであることをテストすることで簡略化できたはずである。

```
def test_weighted_mean():
    numbers = [10, 20, 30, 40, 50]
    weights = [0.1, 0.2, 0.3, 0.2, 0.2]
    expected_result = 32.0

    result = weighted_mean(numbers, weights)

    assert result == expected_result
```

ChatGPT にこの関数を生成してもらうことで、役に立つ出発点を得ることができた。また、予想される状況に対するテストを追加することで改善もできた。だが、AI によって生成されたコードについては、細部にまで注意深く目を通し、徹底的にテストする必要がある。

AI を搭載したツールは強力で、コーディングの高速化に役立つが、こうしたツールがいつ正しい方向に導いてくれるのかを理解する必要がある。関数を書く際には多くの選択肢があり、トレードオフを理解することが重要だ。ツールが書いてくれる関数は、短くて読みやすいが、非効率的である可能性もある。ツールに何を求めるべきかを知っておく必要がある。要求しなければ、ロギングやドキュメンテーションは提供してくれない。**ツールを最大限に活用するためには、良いコードを書くための基礎知識が必要なのだ。**

1つ確かなことは、ソフトウェアエンジニアリングはじっと同じ場所にとどまっていることだ。あるプログラミング言語が一時的に流行するかもしれないが、その後、特定の目的に適した他の言語に取って代わられるかもしれない。新しいアイデアや新技術は常に登場する。そして幸いなことに、開発者コミュニティの人々が、我々が利用するためのドキュメントやガイドを書いてくれる。

## 15.2 コードで描く未来

皆さんはデータサイエンティストとして、キャリアの過程で非常に多くのコードを書くことになる。こうしたコードの中には一度しか実行されないものもあるだろうが、何百万人ものユーザーに使われるプロジェクトや製品の基礎となるものもあるかもしれない。したがって、自分の力を「良いこと」のために使い、自分が構築しようとしているものが自分の倫理観に合っているかどうかを常に考えることを勧める。

また、良いコードを書くために学んだことを共有することを勧める。チームメンバーに模範を示そう。チームのリーダー的存在であればなおさらだ。コードレビューを計画し、チームメンバーのコードを改良する手助けをしよう。皆さんのが書くコードは、チームの「文化」に影響を与えるのだ。

ソフトウェアは孤立して作られるものではない。人脈を築き、さまざまな人と出会い、コミュニケーションをとることを強くお勧めする。より広いコミュニティに参加したり、技術的な講演をしたり、オープンソースプロジェクトに参加したりすることは、人生を豊かにもしてくれるはずだ。また、ソフトウェアエンジニアと話し、自分のコードに対するフィードバックを受け入れることもお勧めする。

## 15.3 おわりに

読者の皆さんへの感謝の言葉で締めくくりとしよう。本書を手に取ってくださって、ありがとう。本書を読むのに時間を費やしてください、ありがとう。本書を書くのはとても楽しかった。

今後の版をより良いものにするため、ぜひご意見をお寄せいただきたい。

<https://linktr.ee/catherinenelson>



# 索引

## 記号

`%%timeit` ..... 21

## A

abstraction ..... 68  
 add (Git) ..... 169  
 agile software development ..... 250  
 AI エンジニア ..... 253  
 Anaconda ..... 180, 187  
 API ..... 193  
 Application Programming Interface  
       ..... 193  
 Artifact Registry ..... 226  
 attribute ..... 58  
 AWS ..... 228

## B

Bandit ..... 238

Big O notation ..... 32  
 Bitbucket ..... 168  
 Black ..... 100, 214  
 branch ..... 174  
 Build ..... 189

## C

C-I-A 三要素 ..... 233  
 cache アクション ..... 220  
 ChatGPT ..... 267  
 Checkmarx ..... 238  
 CI/CD ..... 212  
 CircleCI ..... 213  
 class ..... 58  
 commit (Git) ..... 169  
 complexity ..... 4  
 Conda ..... 179  
 conda-forge ..... 187  
 container ..... 223  
 Cookiecutter ..... 136

coverage ..... 120  
 cProfile ..... 23

**D**

Dask ..... 47, 54  
 DataFrame ..... 52  
 Dependabot ..... 238  
 deploy ..... 222  
 deployment ..... 212  
 deque ..... 38  
 DevOps エンジニア ..... 252  
 Django ..... 199  
 Docker ..... 223  
 Dockerfile ..... 224  
 docstring ..... 158  
 DRY の原則 ..... 7

**E**

else ブロック ..... 78  
 encapsulation ..... 67  
 except ブロック ..... 77

**F**

f 文字列 ..... 83  
 Faker ..... 118  
 FastAPI ..... 199, 236  
 finally ブロック ..... 78  
 Flake8 ..... 104  
 Flask ..... 199  
 FP ..... 57

**G**

Git ..... 168  
 GitHub ..... 168  
 GitHub Actions ..... 218  
 GitHub Copilot ..... 267  
 GitHub Pages ..... 162  
 GitLab ..... 168  
 GraphQL ..... 194  
 Great Expectations ..... 124

**H**

Hatch ..... 180, 182, 189  
 Hypothesis ..... 118

**I**

IDE ..... 90, 105  
 import ..... 98  
 inheritance ..... 65  
 insert 演算子 ..... 37  
 isort ..... 98

**J**

Jenkins ..... 213  
 JSON ..... 193  
 Jupyter ノートブック ..... 87, 144, 162, 236  
 Jupytext ..... 148, 176

**K**

kanban : 看板	250
Kedro	136, 138, 148
KPIs	255

**O**

ONNX	235
OOP	57
OpenAPI	199

**L**

linter	102
linting	102

**M**

Markdown	161
Memray	27
Mercurial	168
merge (Git)	174
method	58
Microsoft Azure	228
MITRE	234
MLflow	165
mypy	109

**N**

nbconvert	148
nbdime	176
ndarray	43
Neptune	165
NumPy	43

**P**

pandas	51
Pandera	124
Papermill	144
pdb	93
PDM	180, 182, 189
pdocs	159
PEP 8	97
Petry	189
pickle	235
pip	181
pip-audit	240
Poetry	180, 182
Polars	54
pre-commit hooks	214
PSF	262
pull (Git)	169
push (Git)	169
PyArrow	51
PyCon	262, 267
Pydantic	126, 205
Pyenv	179
PyLadies	262
pylance	105
Pylint	103
PyPI	180, 185, 190

pyproject.toml	182, 188
Pyre	109
Pyright	109
pytest	118
pytest-cov	120
Python Software Foundation	262
Python カンファレンス	267
PyTorch	50

**Q**

QA エンジニア	254
----------	-----

**R**

Read the Docs	162
README	161
Requests ライブライ	196
requirements.txt	181, 240
REST	194
reStructuredText	160
retrospective meeting	251
Ruff	104

**S**

Sacred	165
SageMaker Experiments	165
scrum	250
scrumban	250
SDGs	14
sdist	189
SemVer	178

Series	52
setuptools	189
Skops	235
SOAP	194
SOLID 原則	143
SonarQube	238
Sphinx	159, 162
SQL インジェクション攻撃	236
SRE : Site Reliability Engineer	253
staging area	169
Subversion	168
Swagger UI	202

**T**

TDD : Test-Driven Development	113
tech debt	2
TensorFlow	50
time complexity	30
timeit モジュール	21
tox	120
Travis CI	213
try ブロック	77
Twine	190
type annotation	107
type hint	107

**U**

unittest	120
URL パラメーター	195
Uvicorn	200, 225
UX リサーチャー	257

**V**

- VCS ..... 167, 235  
 vectorized calculation ..... 45  
 venv ..... 179  
 virtualenv ..... 179  
 VS Code ..... 90, 105

**W**

- Weights and Biases ..... 165  
 wheel ..... 189

**X**

- XML ..... 193

**Y**

- YAML ..... 214

**あ行**

- アサート（テスト） ..... 114  
 アジャイルソフトウェア開発 ..... 250  
 アレンジ（テスト） ..... 114  
 依存関係マネージャー ..... 182  
 依存ライブラリ ..... 178  
 インスタンス ..... 58  
 インスタンス（コンテナ） ..... 224  
 ウォーターフォール型 ..... 249  
 受け入れテスト ..... 121  
 エッジケース ..... 116

- エラー ..... 75  
 エンドポイント ..... 195  
 オーダー記法 ..... 32  
 オープンソースソフトウェア ..... 259  
 オブジェクト指向プログラミング ..... 57  
 親クラス ..... 65  
 オンコールローテーション ..... 253

**か行**

- 仮想環境 ..... 179  
 型アノテーション ..... 107, 201  
 型チェック ..... 106  
 型注釈 ..... 107, 141, 201  
 型ヒント ..... 107  
 カップリング ..... 139, 141  
 可読性 ..... 265  
 カバレッジ ..... 120  
 カプセル化 ..... 67  
 可用性 ..... 233  
 カレンダー・バージョニング ..... 178  
 関数型プログラミング ..... 57, 69  
 完全性 ..... 233  
 カンバン ..... 250  
 機械学習エンジニア ..... 253  
 技術的負債 ..... 2  
 機密性 ..... 233  
 キャメルケース ..... 59  
 脅威（セキュリティ） ..... 233  
 行列 ..... 50  
 クラス ..... 58  
 クラス定義 ..... 61  
 クリーンアップ（テスト） ..... 114

クレデンシャル ..... 234  
 クロスサイトスクリプティング ..... 236  
 クロスサイトリクエストフォージェリ  
..... 236  
 軽減策（セキュリティ） ..... 233  
 計算量 ..... 30  
 繙承 ..... 65, 82  
 繙続的インテグレーション ..... 213  
 繙続的デプロイ ..... 213  
 繙続的デリバリー ..... 213  
 契約 ..... 141  
 結合度 ..... 141  
 堅牢性 ..... 13, 75, 266  
 講演（イベントでの） ..... 261  
 攻撃者 ..... 233  
 構成ファイル ..... 141  
 構文エラー ..... 75  
 コーディングスタンダード ..... 10  
 コードレビュー ..... 177  
 子クラス ..... 65  
 国連の「持続可能な開発目標（SDGs）」の  
データ ..... 14  
 コミット（Git） ..... 169  
 コミットハッシュ ..... 169  
 コミット前フック ..... 214  
 コミットメッセージ ..... 171  
 コミュニティ ..... 258, 262  
 コンテナ ..... 223  
 コンテナレジストリ ..... 223  
 コントロール（セキュリティ） ..... 233  
 コンフィギュレーションファイル ..... 141  
 コンフリクト ..... 175  
 コンポーザビリティ ..... 139

## さ行

最適化 ..... 18, 36  
 サイト信頼性エンジニア ..... 253  
 時間計算量 ..... 30  
 辞書 ..... 39  
 指数時間 ..... 33  
 持続可能な開発目標 ..... 14  
 実行（テスト） ..... 114  
 自動テスト ..... 111  
 集合 ..... 41  
 主要業績評価指標 ..... 255  
 情報セキュリティの3要素 ..... 233  
 深刻度 ..... 84  
 スクラム ..... 250  
 スクラムパン ..... 250  
 スタイルガイド ..... 10, 96  
 スタンドアップミーティング ..... 250  
 ステージング領域 ..... 169  
 ステータスコード ..... 195  
 スナップショット ..... 169  
 スネークケース ..... 156  
 スパース行列 ..... 51  
 スプリント ..... 250  
 脆弱性 ..... 233  
 セキュリティ ..... 231  
 セキュリティテスト ..... 121  
 セット ..... 41  
 摂動 ..... 243  
 セマンティックバージョニング ..... 178  
 線形時間 ..... 33  
 線形対数時間 ..... 33  
 属性 ..... 58

疎結合	142	統合テスト	121, 122
ソフトウェアエンジニア	252	ドキュメンテーション	12, 153
ソフトウェアエンジニアリング	3	トリアージ	253
ソフトウェア工学	3	トレースバック	76

## た行

対数時間	33
タブル	38
単純さ	4, 265
単体テスト	121
抽象化	68
定数時間	32
定性的 UX リサーチャー	257
定量的 UX リサーチャー	257
データアナリスト	255
データエンジニア	254
データ検証	123
データスキーマ	126
データパイプライン	254
データポイズニング	243
適応性	3
デザイナー	258
デザイン（プロジェクト）	134
テスト	111
テストエンジニア	254
テスト駆動開発	113
デバッガー	90
デバッグ	87
デプロイ	212, 222
テンソル	50
テンプレート（docstring）	160
統合開発環境	90

## な行

名前付け	155
二乗時間	33
ネーミング	155
は行	
バージョニング	178
バージョン管理システム	167, 235
バグ	87
パスパラメーター	198
バックエンドエンジニア	252
パッケージ	185
ハッシュテーブル	39
パッチバージョン	178
パフォーマンス	12, 265
早すぎる最適化	18
ビジネスアナリスト	255
ビジネスインテリジェンス・エンジニア	254
ビット腐敗	2
ビヘイビア	140
標準（フォーマット）	10
ビルド（パッケージ）	189
負荷テスト	121
複雑さ	4
フック	214

プライバシー .....	232
プランチ .....	174
振り返りミーティング .....	251
フルスタックエンジニア .....	252
ブレークポイント .....	90
プログラミングスタイル .....	57
プログラミングパラダイム .....	57
プロダクトアナリスト .....	255
プロダクトマネージャー .....	256
プロファイラ .....	23
フロントエンドエンジニア .....	252
分散型 .....	168
分散コンピューティング .....	19
並列コンピューティング .....	19
ベクトル化計算 .....	45
ヘッダー (HTTP) .....	195
ボトルネック .....	23
ポリモーフィズム .....	68

### ま行

マージ (Git) .....	174
マイナーバージョン .....	178
密結合 .....	142
無名関数 .....	70
メジャーバージョン .....	178
メソッド .....	58
メタデータ .....	195
メモリ使用量 .....	27
モジュール .....	186
モジュール化 .....	8, 265

モジュラーコード .....	8, 139
モックアップ .....	258

### や行

ユーザー テスト .....	111
ユーザビリティ セッション .....	257
ユーザビリティ テスト .....	121

### ら行

ラムダ 関数 .....	70
リクエスト ヘッダー .....	195
リスク (セキュリティ) .....	233
リスト .....	36
リファクタリング .....	12, 148
リポジトリ .....	169
リモート リポジトリ .....	169, 172
両端キュー .....	38
リンター .....	102
リンク .....	102
例外 .....	78
例外エラー .....	75
レスポンス ヘッダー .....	195
ローカル リポジトリ .....	169, 172
ロギング .....	82

### わ行

ワイヤーフレーム .....	258
----------------	-----

## ●著者紹介

### Catherine Nelson (キャサリン・ネルソン)

フリーランスのデータサイエンティスト兼ライター。Oxford 大学で地球科学の修士号、Durham 大学で地球物理学の博士号を取得。地球物理学者としてのキャリアでは、古代の火山を研究し、グリーンランドでの石油探査に従事していた。現在の専門は機械学習の説明可能性とモデル分析、プライバシー保護型機械学習など多岐にわたる。以前は SAP Concur のプリンシパル・データサイエンティストとして、ビジネス旅行者の体験を向上させるための機械学習アプリケーションの革新的な提供方法を探求していた。著書に『入門 機械学習パイプライン』(共著。オライリー・ジャパン) がある。

---

## カバーの説明

本書の表紙に描かれた動物は、インド・太平洋のサンゴ礁に生息する鮮やかな色彩の魚、スレッドフィン・バタフライフィッシュ (*Chaetodon auriga*) である。

最大 23cm 程度まで成長し、白地に黒の矢印模様と鮮やかな黄色の背中が特徴的である。目の周囲には黒い帯が走り、背びれには目立つ黒い斑点と垂れ下がる糸状の突起があり、この魚の名前の由来となっている。この糸状の背びれは魚が成長するにつれ発達し、背びれの 5 番目と 6 番目の棘が融合して長い糸状の突起を形成する。

チョウチョウウオ科に属する活動的な魚で、サンゴ礁や礁湖などで、つがいで泳ぐ姿がよく見られる。細長い口先を裂け目に差し込んで、虫、貝類、海綿、藻類、さらには軟サンゴなど多様な餌を探す。

ほかの魚に対しては「平和的」だが、同種に対しては、特に小さな水槽では縄張り意識が強くなることがある。臆病な魚であり、サンゴ礁内の隠れ場所を好む。

IUCN によって低危険種として分類されている。

# データサイエンスのためのソフトウェアエンジニアリング入門

## ——明快かつ速いコードを書く

---

2025年10月20日 初版第1刷発行

著 者	Catherine Nelson (キャサリン・ネルソン)
訳 者	漆戸 パティ (うるしじ ぱてい)
発 行 人	ティム・オライリー
制 作	アリエッタ株式会社
印 刷・製 本	三美印刷株式会社
発 行 所	株式会社オライリー・ジャパン 〒105-0003 東京都港区西新橋一丁目18番6号 Tel (03)6257-2177 Fax (03)6257-3380 電子メール <a href="mailto:japan@oreilly.co.jp">japan@oreilly.co.jp</a>
発 売 元	株式会社オーム社 〒101-8460 東京都千代田区神田錦町3-1 Tel (03)3233-0641 (代表) Fax (03)3233-3440

---

Printed in Japan (ISBN978-4-8144-0132-1)

乱丁本、落丁本はお取り替え致します。

本書は著作権上の保護を受けています。本書の一部あるいは全部について、株式会社オライリー・ジャパンの承諾を得ずに、著作権法の範囲を超えて無断で複写、複製することは禁じられています。