自走プログラマー【抜粋版】

清原 弘貴、清水川 貴之、tell-k、株式会社ビープラウド（監修）

2022 年 05 月 25 日

目次

第 1 章

前書き

プログラミング「迷子」になったことはありませんか？　プログラミング迷子には次のような特徴があります。

- 特徴 1: 何から手を付けて良いのかわからない
- 特徴 2: 正しい方向に進んでいるか自信がもてなくて考え込んでしまう
- 特徴 3: どのファイルに何のコードを書くか定まらず、気がつくと一貫性がなくなっている
- 特徴 4: ミドルウェアや外部ライブラリをどう選んだら良いかわからないし、質問もできない
- 特徴 5: 検索で見つけたコードを貼ってみたけど、なぜかうまく動かない
- 特徴 6: テストコードを書くのにものすごい時間がかかった割に、肝心なところがバグだらけ
- 特徴 7: バグをなおそうとして、別のバグを生んでしまった
- 特徴 8: 完璧に仕上げようと時間をかけたのに、レビューしてもらったら見当違いだと指摘された
- 特徴 9: 作ってみたは良いけど、ごちゃごちゃしていて今後も開発を続けるのが大変
- 特徴 10: ログを出しているのは知ってるが、役に立ったことはない

自分 1 人で走れないのは、進む方向がわからないからです。つまり「地図」を持っていないからです。本書はプログラミング迷子に向けて、絶対に知ってほしい「ソフトウェア開発の地図」を伝えるものです。さあ、地図の作り方を身につけて、「自走プログラマー」になりましょう！

1.1. 本書の構成と読み進め方

本書は、「プログラミング入門者が中級者にランクアップ」するのに必要な知識をお伝えする本です。扱っている 120 のトピックは、実際の現場で起こった問題とその解決方法を元に執筆しています。このため、扱っているプロジェクトの規模や、失敗パターンのレベル感もさまざまです。各トピックでは具体的な失敗とベストプラクティス、なぜそれがベストなのかを解説します。

本書は、プログラミング言語 Python を使って設計や開発プロセスのベストプラクティスを紹介します。Python にくわしくない方でも、プログラミング言語の文法を知っている方であれば理解できるようにしています。逆に、プログラミング自体が何かわからない人のための本ではありません。すでにプログラミング言語の文法や書き方、役割を知っている人が、より効率的かつ効果的にプログラムを書く、価値を創る方法をお伝えする本です。

本書は、大きく 5 つの章に分かれています。

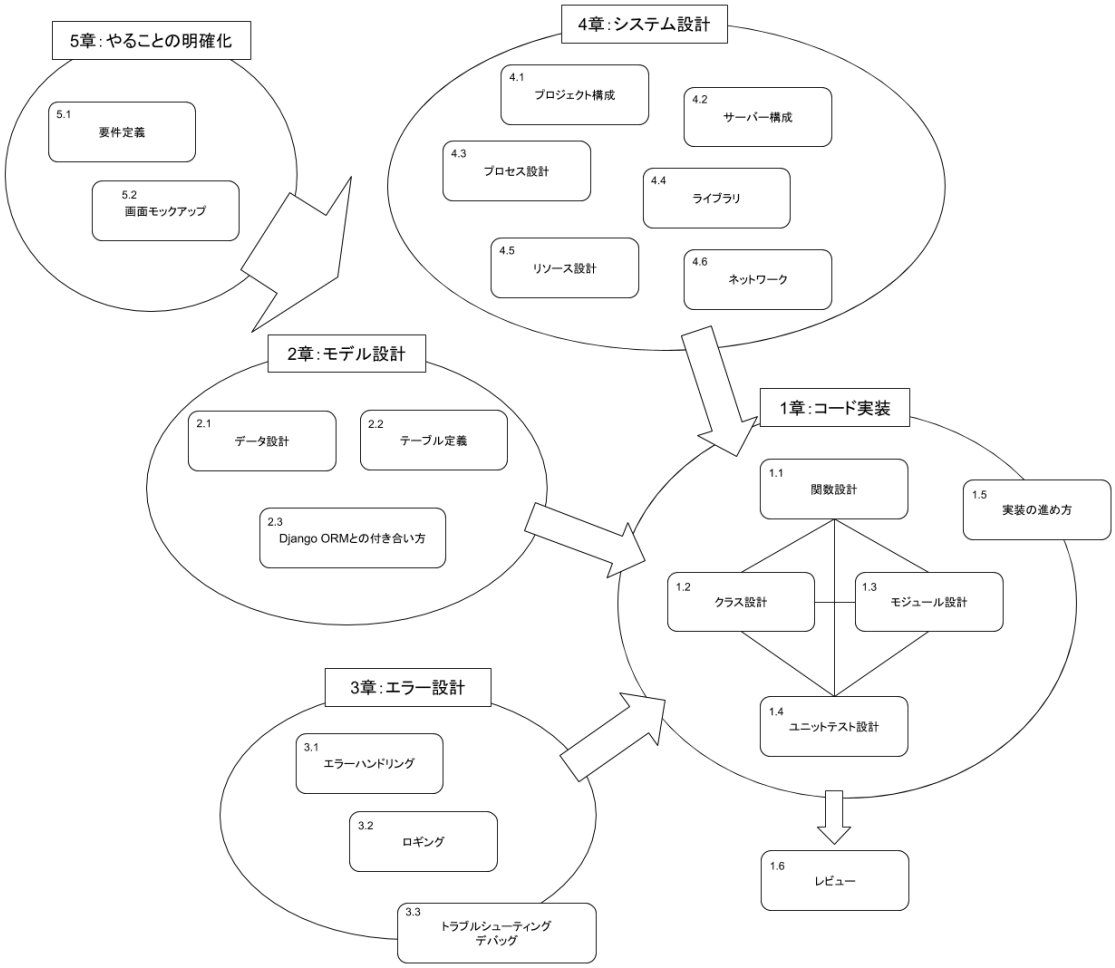


図 1.1　本書で扱っているトピックの地図

1 章「コード実装」では、コードを書く話を扱っています。プログラマーがもっとも興味のある部分かもしれません。関数設計、クラス設計、モジュール設計、ユニットテストの実装、そして GitHub の PR（Pull Request）を使ったレビューの進め方について、具体的なプラクティスを紹介します。また、「迷わない実装の進め方」を実現するソフトウェア開発の地図「開発アーキテクチャドキュメント」の作り方について説明します。

2 章「モデル設計」では、アプリケーション開発で必ず必要になるデータの扱いについて具体的なプラクティスを紹介します。テーブル定義やデータの扱い方は、開発速度を左右する重要な知識です。また、ORM（Object-Relational-Mapping）との良い付き合い方についても紹介します。

3 章「エラー設計」では、プロダクション環境へのリリース後に必要となる技術について紹介します。エラーが発生する処理をどのように実装するべきか、ログ出力は何のためにどんな内容にするべきかといった

プラクティスを紹介します。また、トラブルシューティングとデバッグについても紹介します。

4 章「システム設計」では、プロジェクト全体のシステム構成を組み立てていきます。Python 環境の選び方、サーバー構成、プロセス、ライブラリ、リソース、ネットワークといった、プログラムを実際に動作させるのに必要となる環境に焦点をあてています。

5 章「やることの明確化」は、コードを書く前の話です。これから作ろうとしているモノがどのような特徴を持っていて、どのように使われるのか明確にしていきます。何を作ろうとしているのかをあいまいなまま進めてしまうと、実装中やレビューの段階で大前提からやり直しになってしまいます。どのくらい「明確化」に時間をかければ手戻りをおさえつつ、次の段階に進めるのかを解説します。

本書は、章が進むにつれて「プログラム」から「周辺の技術」に話題が移っていきます。各トピックは独立しているため、どこから読み始めてもかまいません。

- コードに関する要素から読みたい人は、1 章から読んでいくと良いでしょう。
- そもそも何を作るかを明確にしてから進めたい人は、5 章から 1 章に向けて読むと良いでしょう。

1.2. 対象読者

- チョットした便利なコードを書けるけど、中〜大規模のシステムを上手に作れない人
- プログラムを書けるけど、レビュー指摘などで手戻りが多い人
- 優れたエンジニアになりたい人
- Python で Web アプリケーションの開発をするときの指針が欲しい人
- Python 入門を果たしたプログラマーで、仕事で Python をやっていこうという人
- 設計の仕方や、メンテナンス性の高いプログラムの書き方を知りたい人
- ライブラリの選定を、確信を持ってできるようになりたい人

1.3. プログラミングブームにおける本書の価値

プログラミングは、パソコンがあれば無料で始められます。初学者向けの本もたくさんあり、特にここ数年は今まで以上に多くの人がプログラミングを始めています。裾野はどんどん広がっていき、2020 年からの初等教育でのプログラミングの必修化もそれを後押ししていくでしょう。あと何年か後には、プログラミングが今よりももっと日常的に行われる世の中になっているかもしれません。

こうした状況はとても喜ばしいことですが、プログラミングが一部の人のだけが持つスキルでなくなれば、仕事でプログラミングする人にはより高いスキルが求められることになります。競争が激しくなっていくなかで、より秀でたプログラマーになるためには、何が求められるでしょうか？

プログラミングで何かを作るには、文法の他にアプリケーションを設計するスキルや、ライブラリを選定するスキル、Web アプリケーションなら本運用し続ける環境を整えるスキル、運用するスキル、などさまざまなスキルが必要です。これらのスキルのうち、「プログラミングで何かを作るプロセスとスキル」「プログラミングで作りたいものを設計するスキル」をお伝えするのが本書です。本書を読み終わったとき、次のようになれたらすばらしいと思いませんか？

- 自分の作りたいものを着実に作るプロセスがわかっている
- どう設計すればアプリケーションとして良いものができるかがわかっている
- どういう場合にどのライブラリを使えば良いかわかっている

本書はそんな、単にプログラミング言語だけでないプログラミングの「中級な」「設計を含んだ」「うまく作るための」内容をお伝えする本です。少し読んでみると「プログラミングそのものの話題が少ない」と少し戸惑うかもしれません。ですがそれこそがプログラミング能力を活かして何かを作るために必要なことです。

1.4. 著者の思い

我々ビープラウドは、在籍するスタッフの多くがコードを扱います。代表取締役も、営業の役割を担っている人間も、コードを読み書きします。現在の仕事は多くが受託開発で、大量のトラフィックを捌く必要があるコンシューマ向けの Web システムや、ビジネス向けの Web 業務システム、機械学習によるデータ処理と Web の連携などが多くを占めている Web 系ソフトウェア開発という職種です。

ビープラウドでの開発プロジェクトは 2-3 ヶ月という短いスパンのプロジェクトを 2-3 名で開発することが多いため、1 人が関わる範囲が広く、必然的に書くコードの量も多くなります。私たちにとってプログラマーとは、設計書をコードにする単純作業者のことではなく、やりたいことをまとめ、設計からコードにし、そしてリリースするまでをすべて 1 人でできる人のことを指しています。本当にすばらしいサービスやアプリケーションをつくり出すには、自走できるプログラマーが必要です。

とはいえ、すべてのプログラマーがはじめから自走できたわけではなく、組織のメンバは常に入れ替わっていき、新しく参加するメンバの中にはこれからいろいろなことを学んでいく人もいます。それは、技術的なつまづきと学びを繰り返して、その背景にある原理原則をメンバそれぞれが見つけていく、長い旅のようなものです。ビープラウドには、この学びの旅をサポートする「教え合う文化」が根付いており、つまづいたときには先輩が親身になって助けてくれます。そこで先輩達が教えてきた履歴をみると、新しいメンバーがなぜか必ずつまづいてしまうパターンがいくつもあることがわかってきました。こういった、設計からコードまで書けるようになるために知っておいて欲しい技術的なトピックを集め、この本にまとめました。

本書は、プログラミング入門ならぬ、脱入門者を目指す開発者向けの指南書です。自走できるプログラマーであれば知っているであろういろいろな手法や観点を元に、「プロジェクトの各段階でプログラマーがやること」「その選択をどう判断するのか」「どうコードを実装して実現していくのか」を紹介します。一部の最新技術に注目するのではなく、実際のプロジェクトに適用して、プロジェクトを完成させるための指針をまとめました。

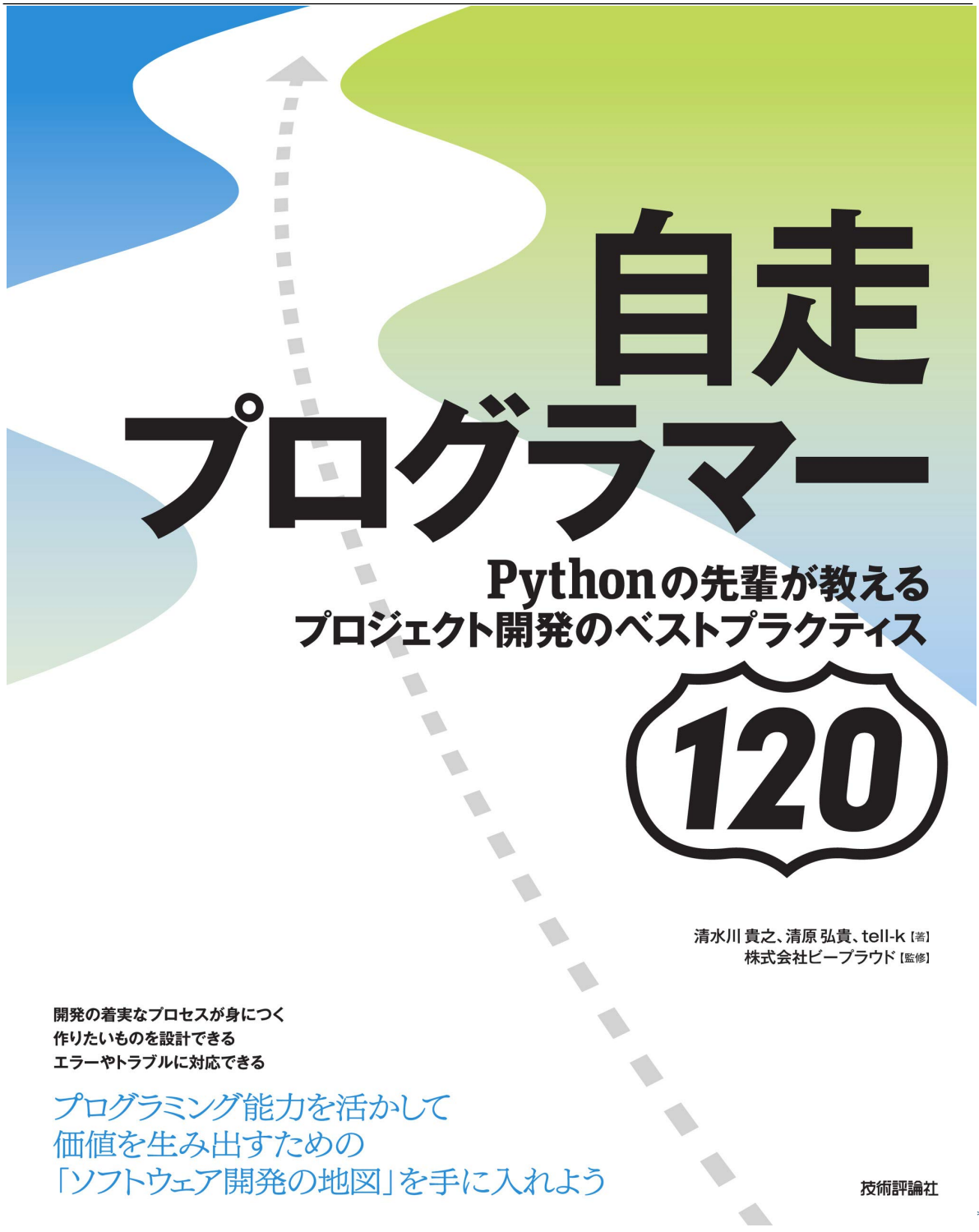
第 2 章

コード実装

2.1. 関数設計

2.1.1 1:関数名は処理内容を想像できる名前にする

プログラミングにおいて関数化と関数名はとても大切です。良い関数名をつけることで動作や仕様が想像できるので、プログラムを読む人は関数の実装を詳しく見る必要がなくなります。



[*1]

[*1] https://gihyo.jp/book/2020/978-4-297-11197-7

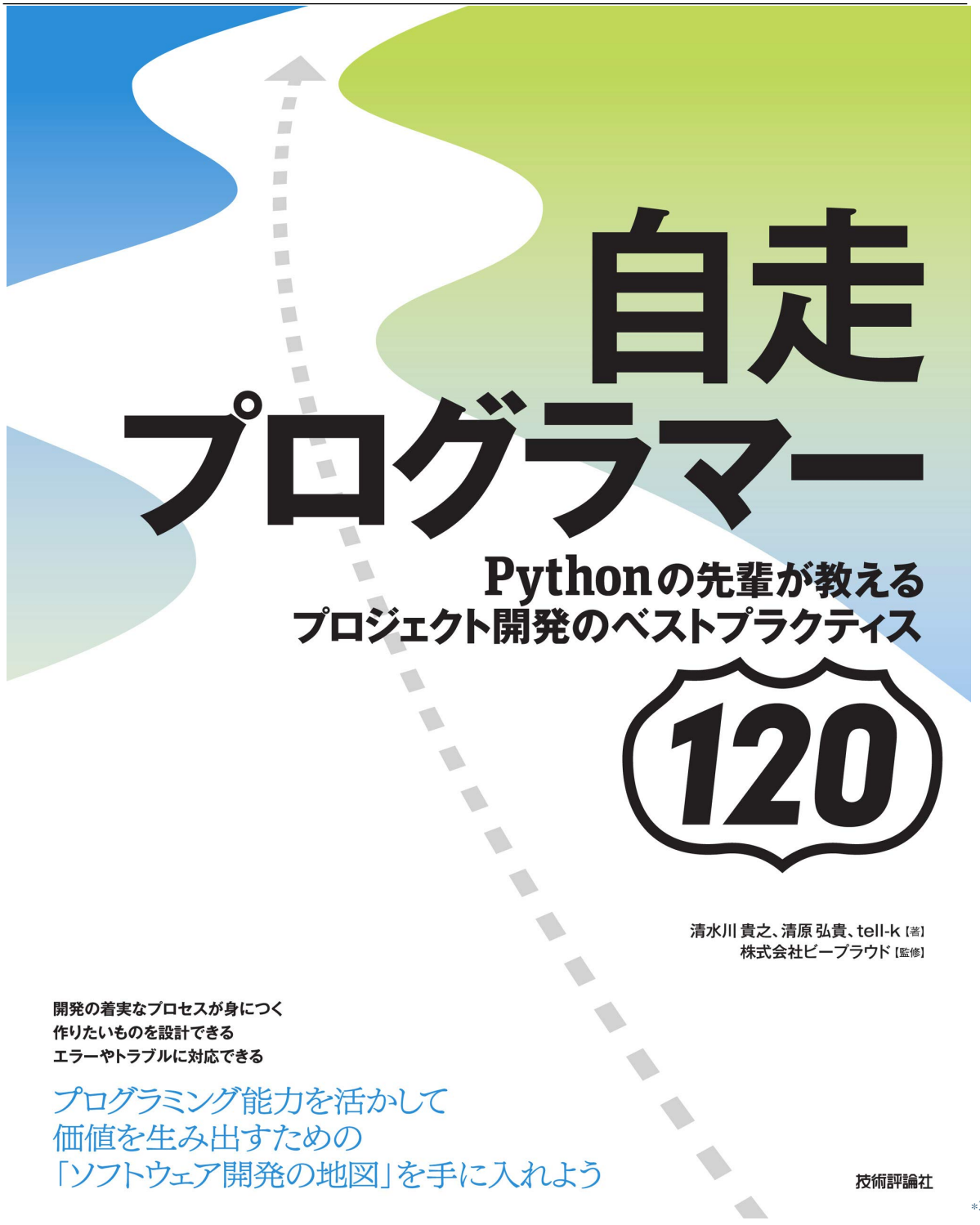
（中略）詳細は書籍 自走プログラマー[*2] をご参照ください

ベストプラクティス

動詞を関数名の頭につけるか、取得できるものや役割の名詞にしましょう。

- 動詞にする例： `get_item_list` 、 `calc_tax` 、 `is_member` など
- 取得できるものの名詞にする例： `current_date` 、 `as_dict` など
- 役割で関数名にする例： `json_parser` など

[*2] https://gihyo.jp/book/2020/978-4-297-11197-7



[*3]

[*3] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*4] をご参照ください

関連

- *2:*関数名ではより具体的な意味の英単語を使おう (ページ 14)

2.1.2 2:関数名ではより具体的な意味の英単語を使おう

関数名からはより具体的な意味を類推できることが重要です。関数名に使う言葉を「より狭い意味」の単語に置き換えることで意味が伝わりやすくなります。

関数名に get_ ばかりを使ってしまった例を見てみましょう。

具体的な失敗

```
def get_latest_news():
    ...

def get_tax_including(price):
    ...

def get_sum_price(items):
    ...
```

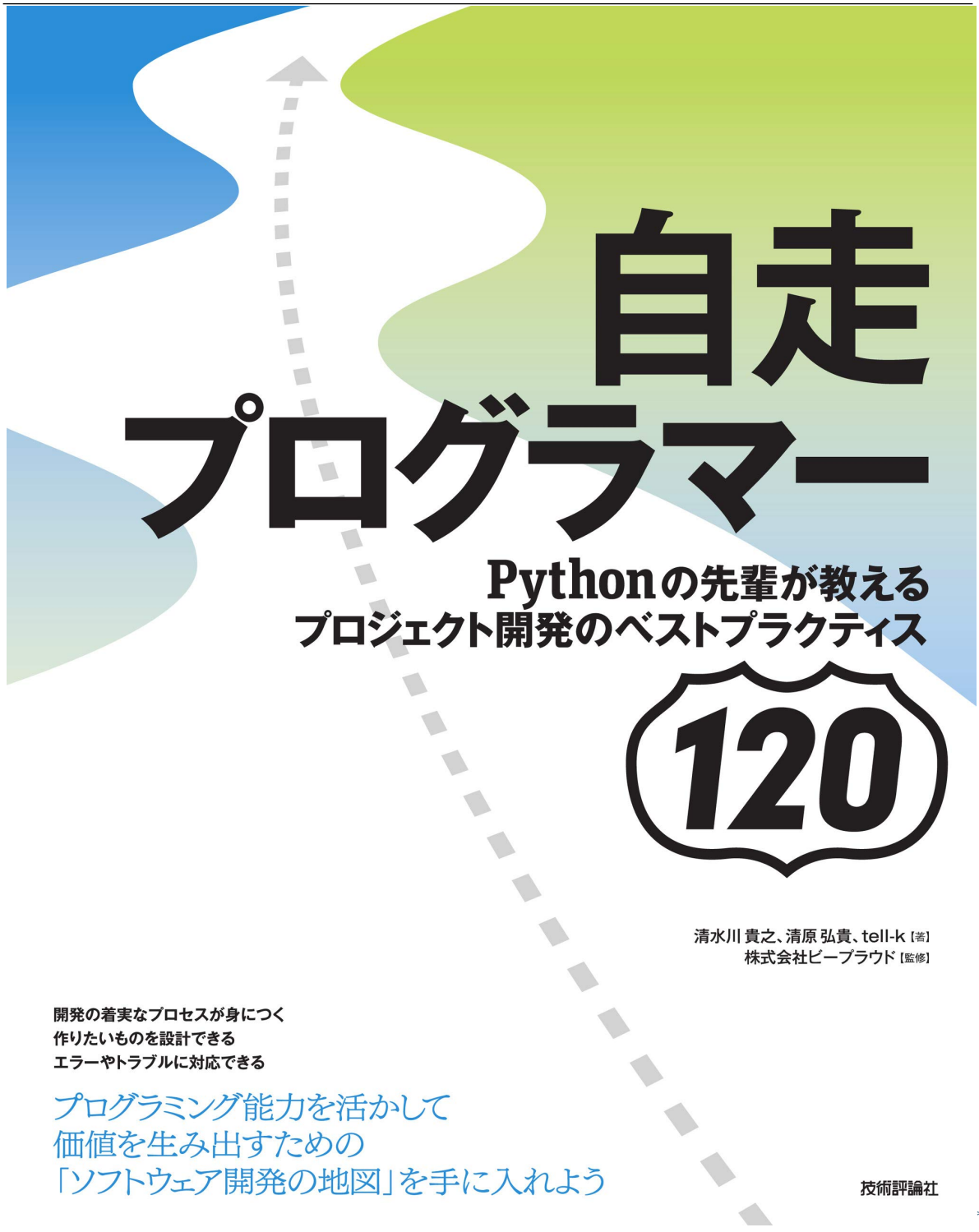
この失敗では「何かを取得する」という意味ですべての関数名が get_ になっています。ですが、関数が具体的にどういう動作をするかまで、 get という英単語からは想像できません。外部へのアクセスがどれだけ発生するのか、どれだけ計算処理が発生するのか、データベースへのアクセスがあるのかわからないのは問題です。

[*4] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

より狭い意味の英単語を使いましょう。処理の内容を想像できる「より狭い」英単語を使います。

```
def fetch_latest_news():
    ...

def calc_tax_including(price):
    ...

def aggregate_sum_price(items):
    ...
```



*5

*5 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*6] をご参照ください

関連

- *1:*関数名は処理内容を想像できる名前にする (ページ 10)
- *3:*関数名から想像できる型の戻り値を返す (ページ 17)

2.1.3 3:関数名から想像できる型の戻り値を返す

変数名、関数名から「想像できること」はとても大切です。特にプログラミングにおいては「型」が想像できることがとても重要です。特に Python は 動的型付け言語 なので、関数がどんな型で戻り値を返すかを制限できません。

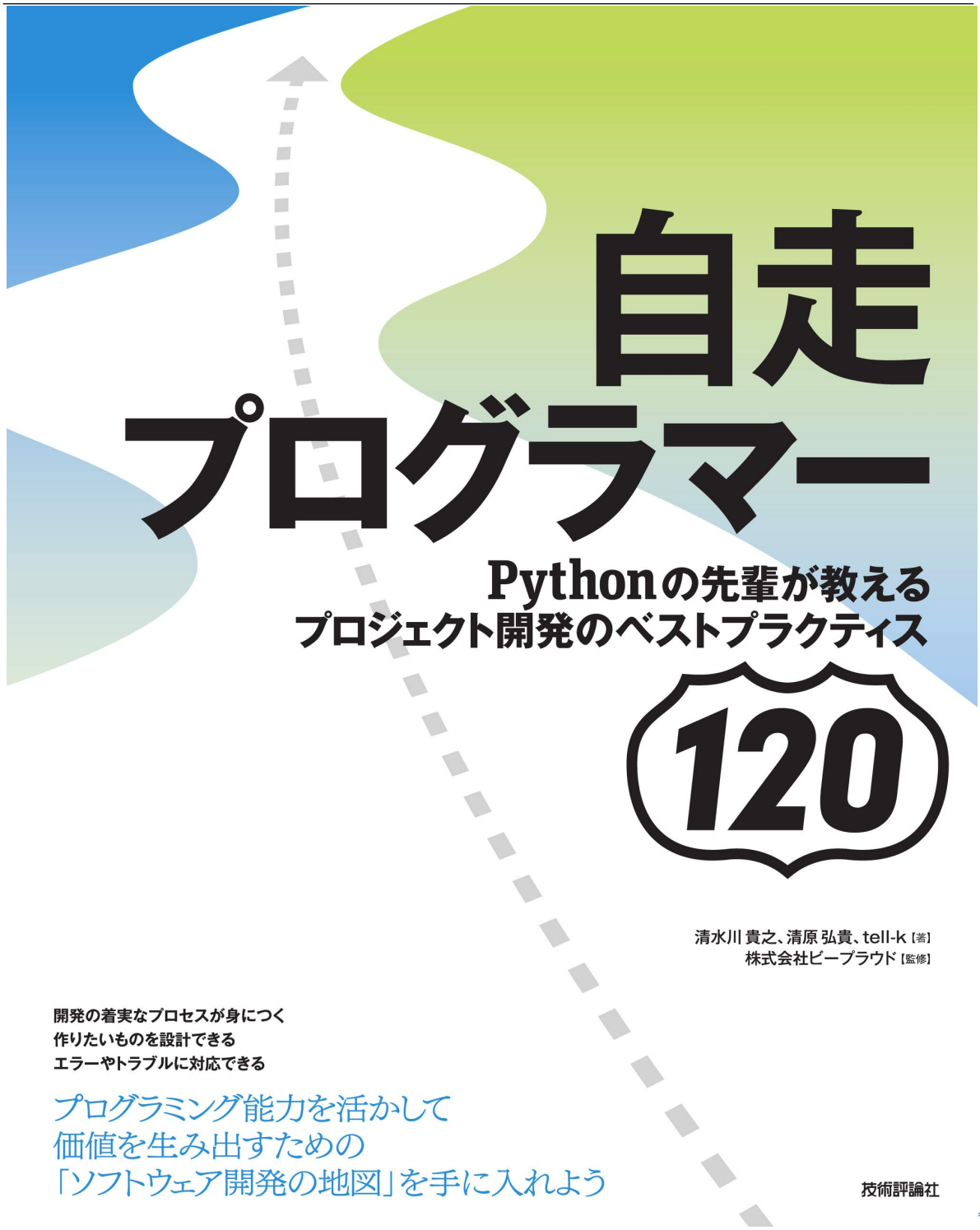
次の関数にはどんな問題があるか考えてみましょう。

具体的な失敗

```
def is_valid(name):
    if name.endswith(".txt"):
        return name[:-4] + ".md"
    return name
```

この is_valid 関数の問題は、関数名から想像できる「戻り値の型」と実装が違うことです。is_ や has_ で始まる関数名の場合は bool が返る関数のように思えてしまいます。

[*6] https://gihyo.jp/book/2020/978-4-297-11197-7



*7

*7 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*8] をご参照ください

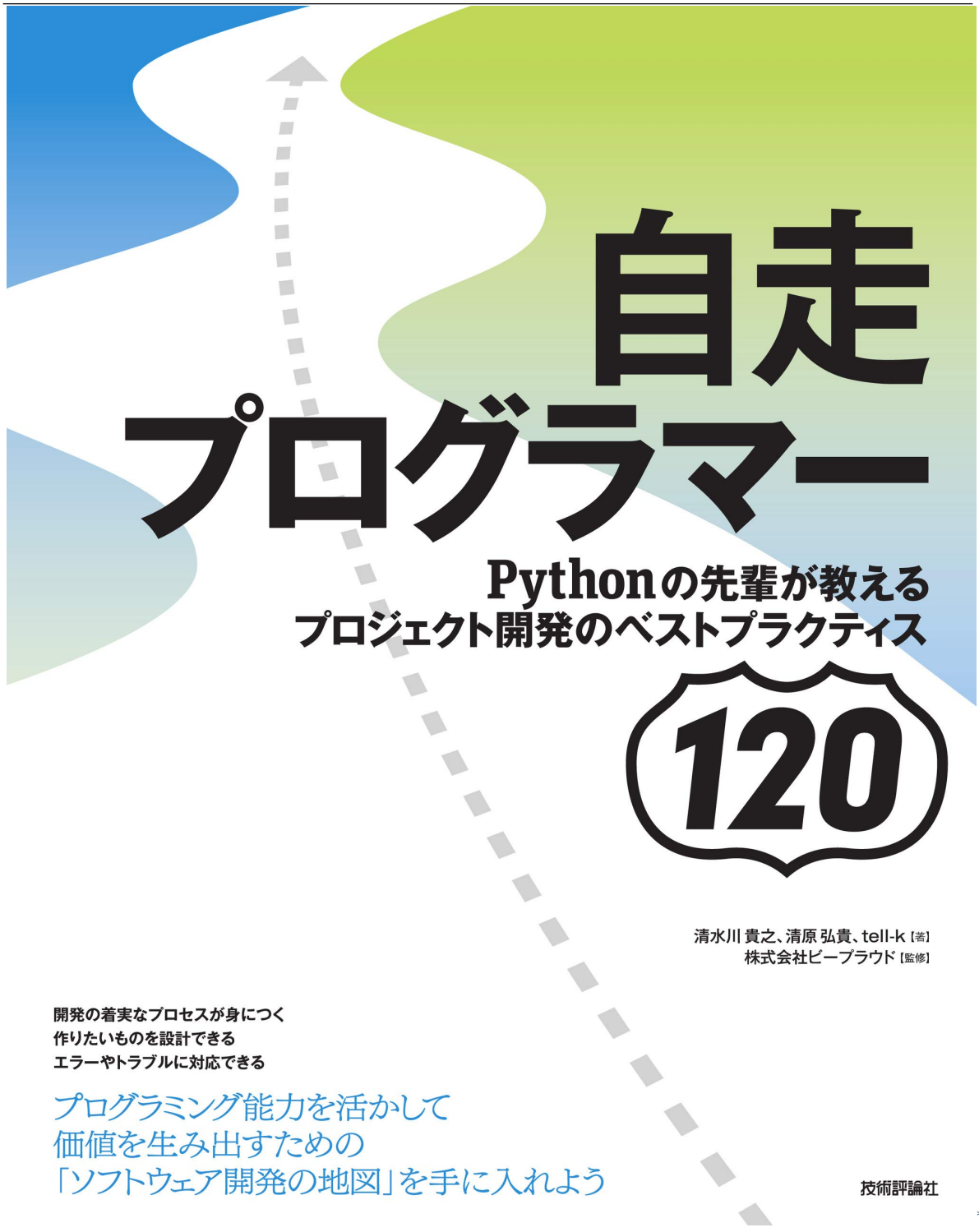
ベストプラクティス

is_ 、 has_ で始まる変数名、関数名の場合は bool を扱うようにします。

```
def is_valid(name):
    return not name.endswith(".txt")
```

これで if is_valid(): のように if 文で正しく扱えます。大切なのは関数名から期待される動作や戻り値の型と、実装を一致させることです。

[*8] https://gihyo.jp/book/2020/978-4-297-11197-7



[*9]

[*9] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*10] をご参照ください

2.1.4 4:副作用のない関数にまとめる

プログラミングにおいて「副作用」を意識することはとても大切です。副作用とは、プログラムが実行された結果に何かしらの状態が変更されることを言います。関数が外部にある変数や状態に影響されない場合、同じ入力を与えると常に同じ出力をするはずです。テストがしやすく、状態に影響されない関数ができます。

副作用のある関数はどんなもので、どういう注意点があるのでしょうか？

具体的な失敗

```
def is_valid(article):
    if article.title in INVALID_TITLES:
        return False

    # is_valid 関数が article.valid の値を書き換えている
    article.valid = True
    # .save() を呼び出すことで、外部にデータを保存している
    article.save()
    return True

def update_article(article, title, body):
    article.title = title
    article.body = body
    if not is_valid(article):
        return
    article.save()
    return article
```

この場合 `is_valid` 関数を呼び出しただけで `article` の `.valid` 値が変更されてしまいます。いろいろな関数から副作用があると、開発者が予期しないところでデータが変更されてしまう問題があります。予期しないところでデータが変更されると、バグの元になったり トラブルシューティング が難しくなったりします。

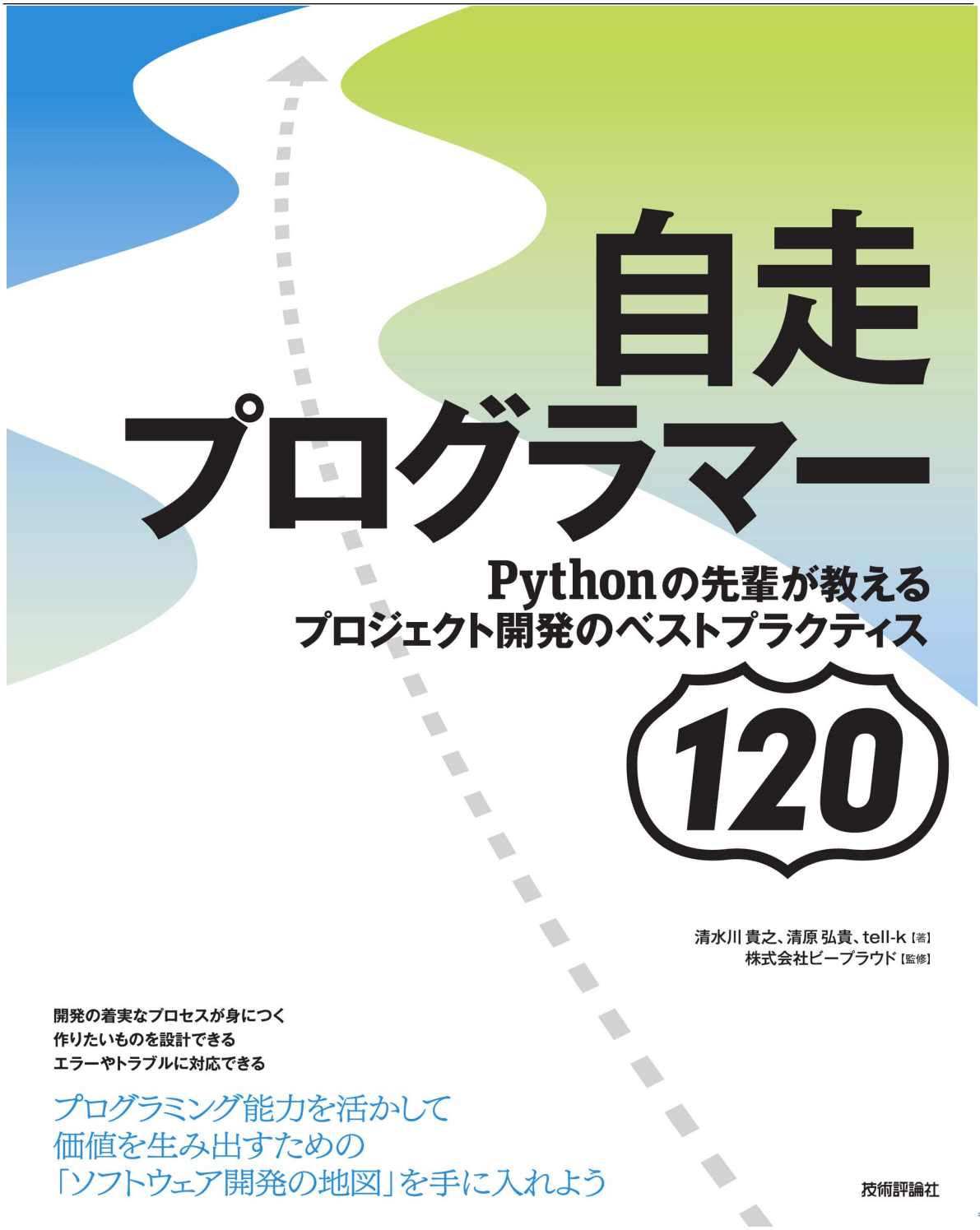
[*10] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

この場合は is_valid 関数では副作用を起こさないほうが良いでしょう。関数名を is_valid_title として「正しいタイトルかどうか」を確認する関数に留めましょう。

```
def is_valid_title(title):
    return title not in INVALID_TITLES:

def update_article(article, title, body):
    if not is_valid_title(title):
        return
    article.title = title
    article.body = body
    article.valid = True
    article.save()
    return article
```

こうすると is_valid_title では副作用がないので他の処理からも再利用しやすくなります。また、「update_article を呼び出したときは副作用がある」というのが明確になります。



[*11]

[*11] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*12] をご参照ください

2.1.5 5:意味づけできるまとまりで関数化する

無思慮に関数をまとめていませんか？　関数に分離するときは処理のまとまりで分けてはいけません。

具体的な失敗

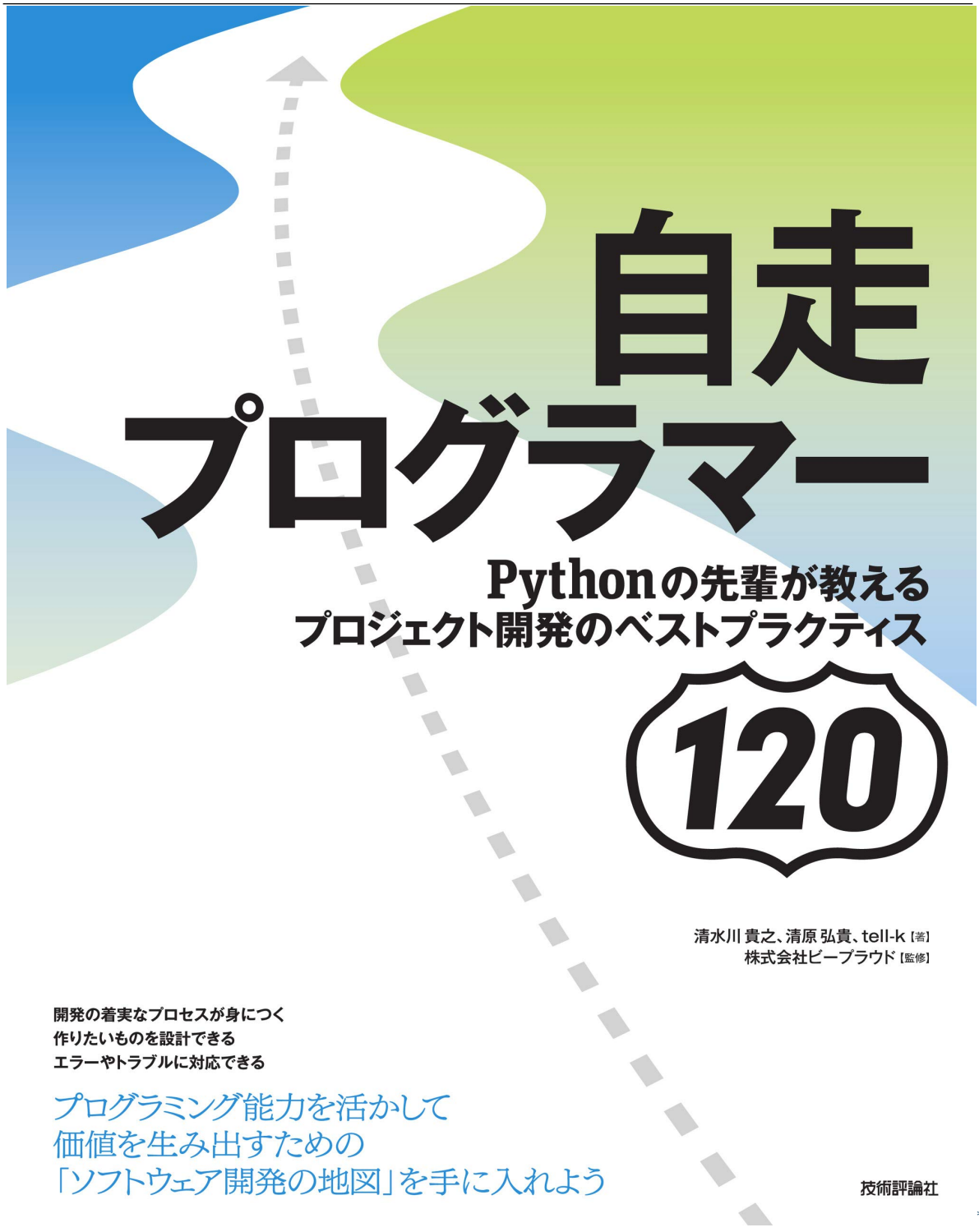
```
def main():
    with open(...) as f:
        reader = csv.reader(f)
        for row in reader:
            i = row[1]
            if i < 100:
                print(row[0])
```

この関数は単に「主な処理」として main() 関数にまとめられているだけです。これを「処理のまとまり」で分離してしまうと以下のようになります。

```
def print_row(row):
    i = row[1]
    if i < 100:
        print(row[0])


def main():
    with open(...) as f:
        reader = csv.reader(f)
        for row in reader:
            print_row(row)
```

関数化することで改善した気持ちになってしまいますが、この分離は問題があります。

[*12] https://gihyo.jp/book/2020/978-4-297-11197-7



*13

*13 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*14] をご参照ください

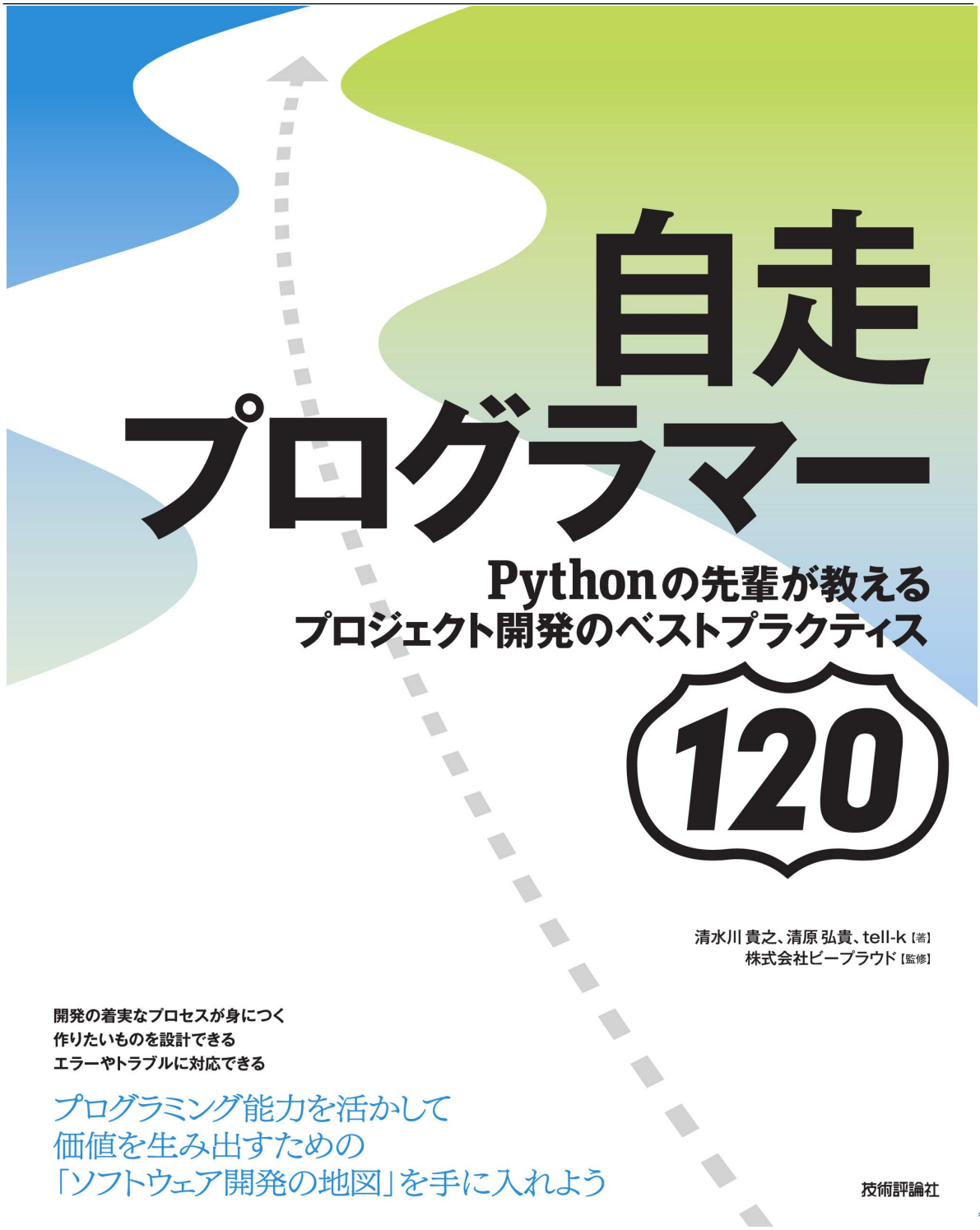
ベストプラクティス

処理の意味、再利用性で関数や処理は分離しましょう。「関数に分離しよう」と意気込む前に、処理をそれぞれどう意味づけできるかを考えることが大切です。CSV 読み込み、 100 との比較という処理が、どういう意味なのかを関数で表します。今回は、「価格が 100 円未満の場合は、買い合わせ対象商品である」という仕様があったとします。

```
import csv


def read_items():
    """ 商品一覧の CSV データを読み込んでタプルのジェネレーターで返す
    各商品は「商品名、価格」のタプルで返される
    """
    with open(...) as f:
        reader = csv.reader(f)
        for row in reader:
            name = row[0]
            price = int(row[1])
            yield name, price


def is_addon_price(price):
    """ 価格が「買い合わせ対象商品」の場合 True を返す
    """
    return price < 100


def main():
    items = read_items()
    for name, price in items:
        if is_addon_price(price):
            print(name)
```

[*14] https://gihyo.jp/book/2020/978-4-297-11197-7

処理でなく意味でまとめることで、それぞれの処理が「何のために行われているか」が自明になりました。



*15

*15 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*16] をご参照ください

2.1.6 6:リストや辞書をデフォルト引数にしない

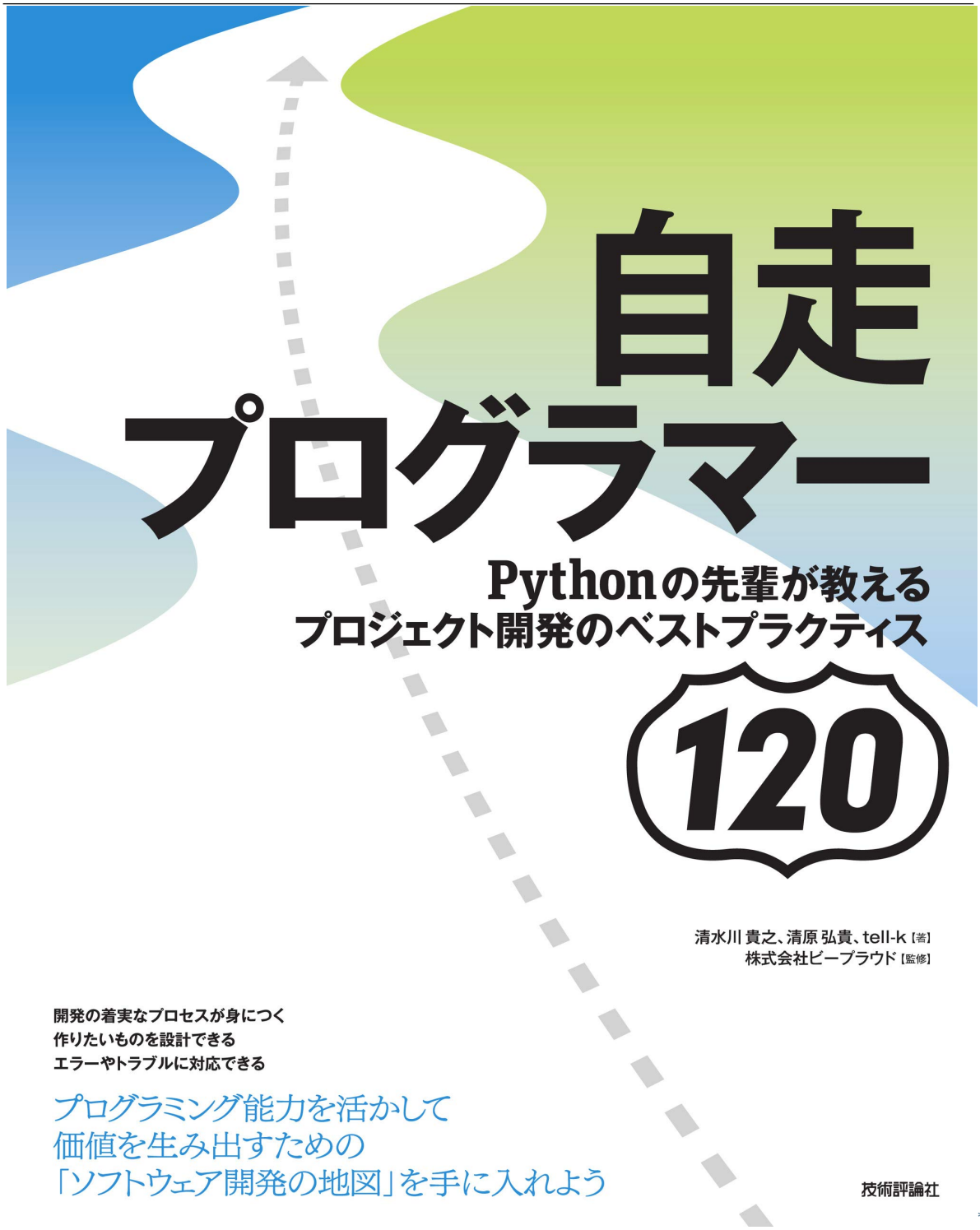
Python のデフォルト引数は便利な機能ですが、使ううえでの罠があります。次の例のようにプログラムを書いたことはありませんか？

具体的な失敗

```
def foo(values=[]):
    values.append("Hi")
    return values
```

引数 values をデフォルトで空のリストにしたい場合に values=[] と書いてはいけません。

[*16] https://gihyo.jp/book/2020/978-4-297-11197-7



*17

*17 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*18] をご参照ください

ベストプラクティス

更新可能（mutable）な値はデフォルト引数に指定してはいけません。リスト、辞書、集合をデフォルト引数にしてはいけないと覚えておきましょう。デフォルト引数に None を設定しておいて、関数内で None の場合に空のリストや辞書を指定しましょう。

```
def foo(values=None):
    if values is None:
        values = []
    values.append("Hi")
    return values
```

これで foo() を何回呼び出しても常に ["Hi"] が返ります。

2.1.7 7:コレクションを引数にせず int や str を受け取る

関数の引数にはどういった値を期待するのが良いでしょうか？関数の引数を考えることは、関数の入力仕様を決めることなのでとても重要です。

次の関数は何が問題でしょうか？

具体的な失敗

```
def calc_tax_included(item, tax_rate=0.1):
    return item['price'] * (1 + tax_rate)
```

この calc_tax_included は引数に item （商品を表す辞書）を期待しています。これでは単に「消費税を計算したい」という場合にも、毎度「 'price' キーを持つ辞書」を用意する必要があります。関数の再利用性が低くなってしまいます。

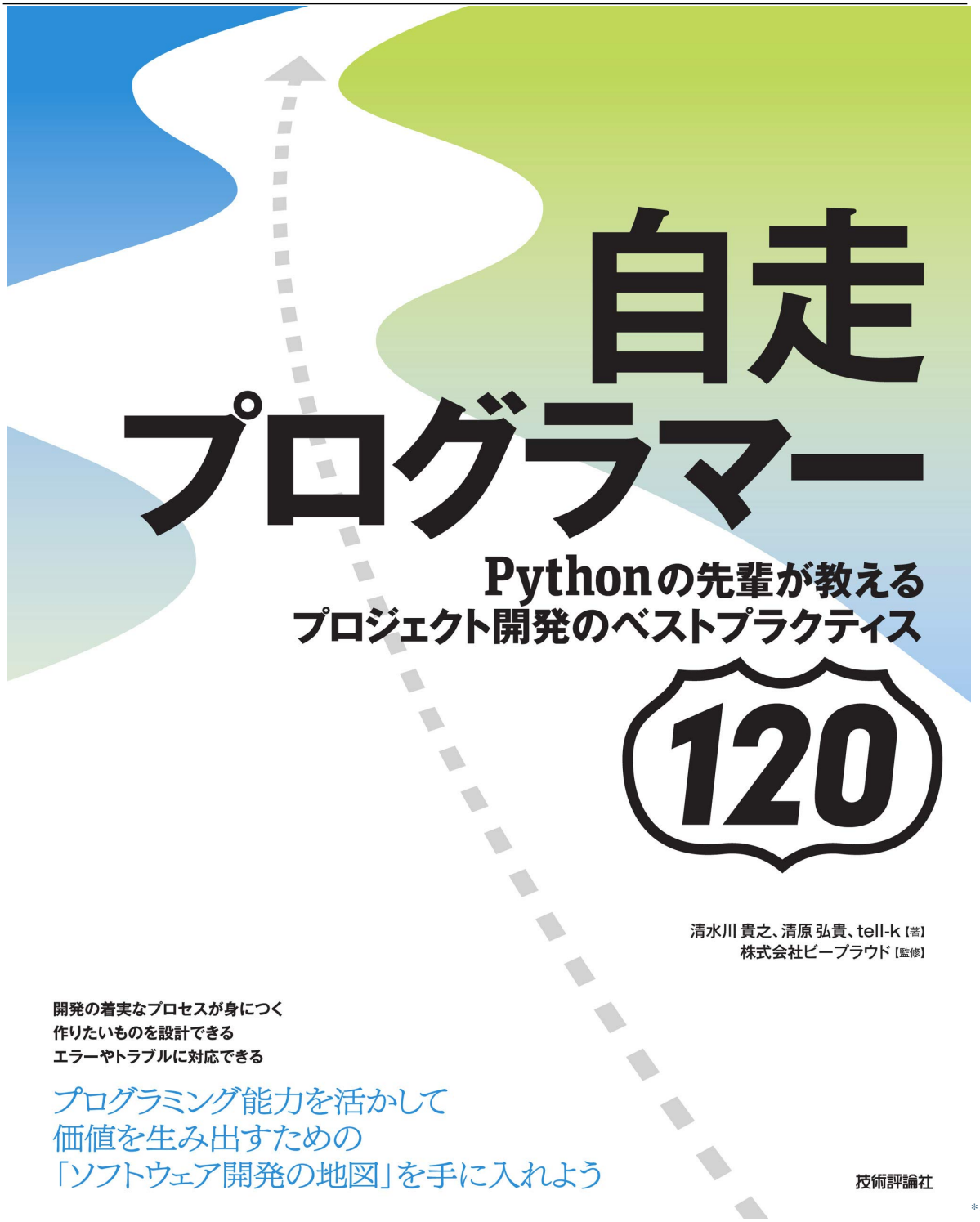
[*18] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

関数の引数は数値（int）や浮動小数点数（float）、文字列（str）など、 コレクション でない値を受け取るのが良いでしょう。

```
def calc_tax_included(price, tax_rate=0.1):
    return price * (1 + tax_rate)
```

辞書を受け取らずに数値で受け取る関数にすることで、単に消費税込みの金額を計算したい場合にも `calc_tax_included` 関数が使えます。



*19

*19 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*20] をご参照ください

2.1.8 8:インデックス番号に意味を持たせない

Python でリストやタプルの インデックス番号 を使ったほうが良いプログラムになる場合は、非常に稀です。インデックス番号に意味を持たせてしまうとどうなるでしょうか？

具体的な失敗

```
from .item import item_exists


def validate_sales(row):
    """
    row は売上を表すタプル
    1 要素目: 売上 ID
    2 要素目: 商品 ID
    3 要素目: ユーザー ID
    4 要素目: 個数
    5 要素目: 売上日時
    """
    # ID のチェック
    if not item_exists(row[1]):
        raise ...

    # 個数のチェック
    if row[3] < 1:
        raise ...
```

たとえば辞書であれば `row['item_id']` のように、処理そのものが意味を表してくれます。しかしこの例では処理の中で `row` のインデックス番号が意味を持っているので、プログラムが読みにくくなっています。プログラムを読んでいるときに `row[1]` が商品 ID であると覚えておかないといけません。

またインデックス番号で処理すると、間に新しい値が入ると処理が壊れます。たとえば `row` の仕様が変わって 2 要素目に「販売店 ID」が入るようになったとすると、それ以降の要素を指定する処理を書き換える必

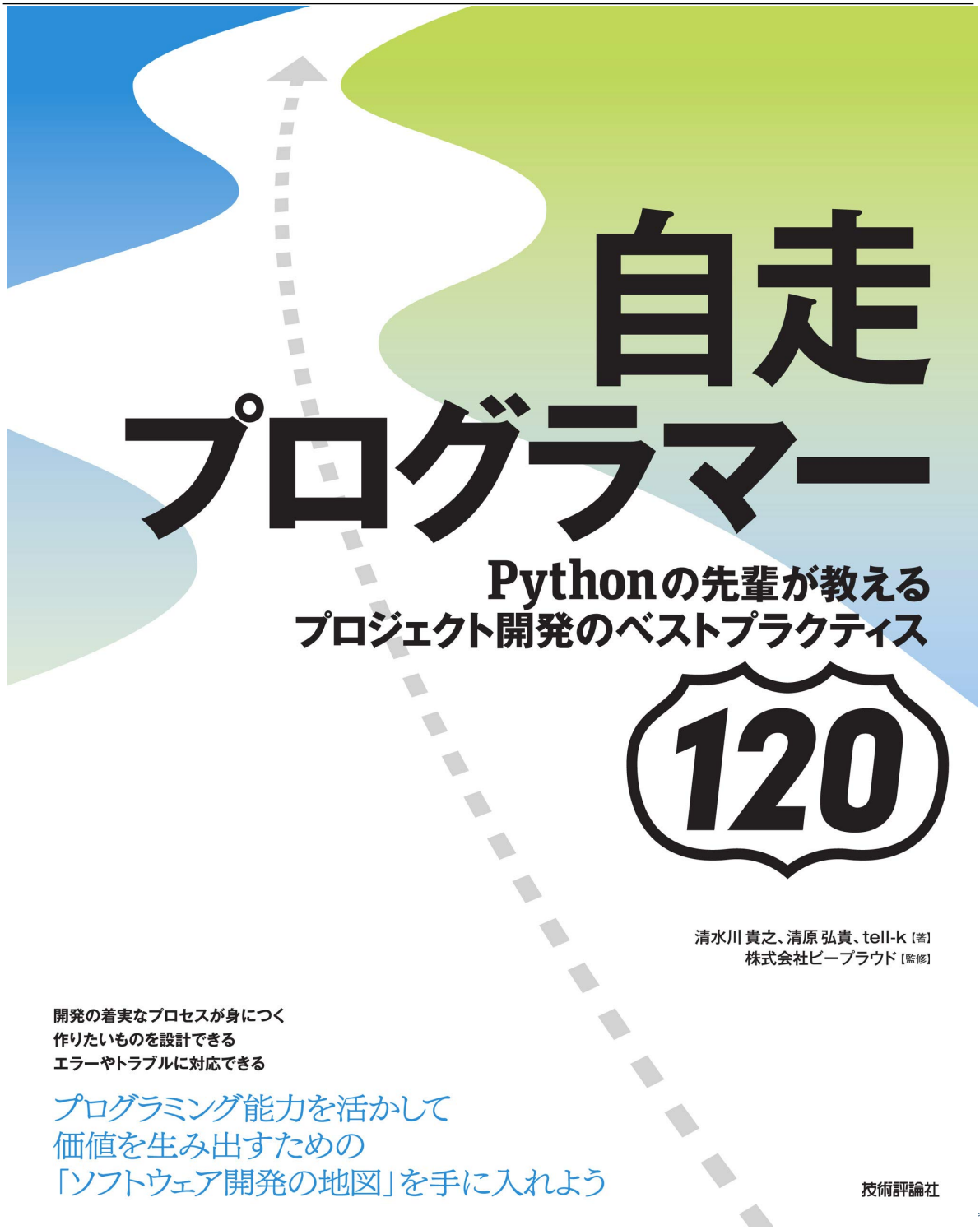
[*20] https://gihyo.jp/book/2020/978-4-297-11197-7

要があります。その場合は、 row[3] を row[4] にする必要があります。

ベストプラクティス

タプルで管理せず辞書やクラスにしましょう。row のタプルを Sale というクラスに置き換えると、validate_sales 関数がとても読みやすくなります。

```
@dataclass
class Sale:
    sale_id: int
    item_id: int
    user_id: int
    amount: int
    sold_at: datetime


def validate_sales(sale):
    """ 売上 sale が不正なデータの場合エラーを送出する
    """
    if not item_exists(sale.item_id):
        raise ...

    if sale.amount < 1:
        raise ...
```



[*21]

[*21] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*22] をご参照ください

2.1.9 9:関数の引数に可変長引数を乱用しない

Python の便利な機能である 可変長引数 の *args 、 **kwargs ですが、無思慮に使いすぎるとバグを仕込みやすいプログラムになります。

どういった問題があるのでしょうか？　プログラムを見ながら考えてみましょう。

具体的な失敗

```
class User:
    def __init__(self, **kwargs):
        self.name = kwargs['name']
        self.mail = kwargs.get('mail')
```

この User は以下のように、クラスが期待していない email= 引数を受け取れてしまいます。email= と勘違いしてプログラムした場合に、エラーになりません。

```
>>> user = User(name="hiroki", email="hiroki@example.com")
```

ここで user.mail は None になります。予期しないデータが作成されているのにエラーにならないので、プログラムの別の場所でエラーになったり、必要なデータが保存されない問題があります。

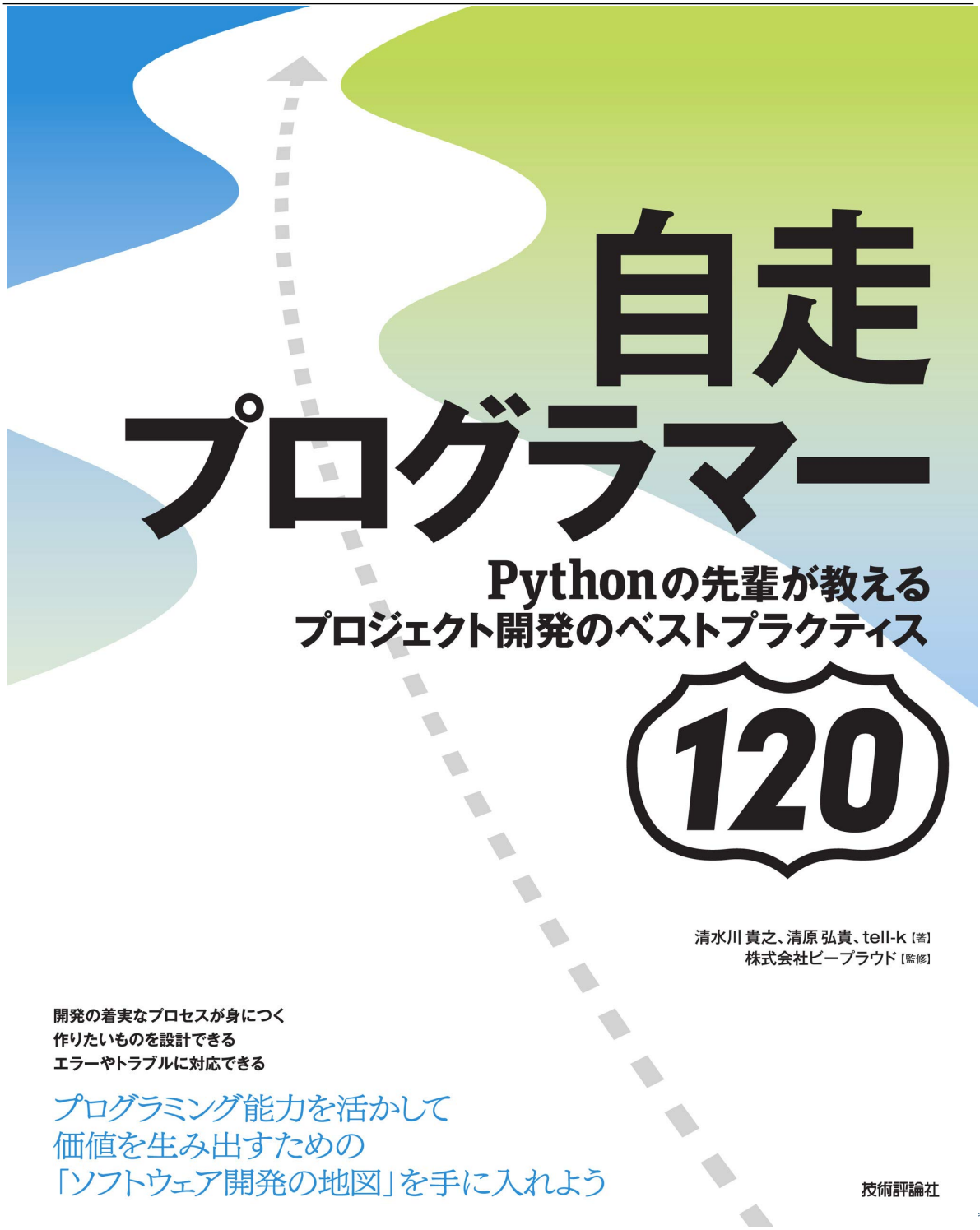
ベストプラクティス

不用意に *args 、 **kwargs を使わずに個別の引数で指定しましょう。

```
class User:
    def __init__(self, name, mail=None):
        self.name = name
        self.mail = mail
```

この場合、存在しない引数を指定すればエラーになります。

[*22] https://gihyo.jp/book/2020/978-4-297-11197-7



*23

*23 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*24] をご参照ください

2.1.10 10:コメントには「なぜ」を書く

プログラムの コメント には何を書いていますか？　コメントを書くときにも、抑えておくべきポイントがあります。

具体的な失敗

```
def do_something(users):
    # 〜〜をする処理
    # users には User の QuerySet を受け取る

    # 引数の users を 1 つひとつループして処理をする
    # users がループするときにバックエンドに SQL が実行される
    for user in users:
        ...
    return users  # SQL はループでの 1 回しか実行されない
```

このプログラムには無駄なコメントが多いのが問題です。プログラムを読めば、処理は理解できます。コメントがないと理解しにくいコードの場合、コメントで説明を補う前に簡単なコードに書き直せないか考えてみましょう。

ベストプラクティス

コメントには「なぜ」を書きましょう。関数の仕様を書く場合はコメントでなく、docstring に書きましょう。

```
def do_something(users):
    """ 〜〜をする処理

    複数のユーザーに対して <do_something> を行う。
    〜〜の場合に〜〜なので、ユーザーのデータを変更する必要がある。
    """
    # SQL の実行回数を減らすために、このループは別関数に分離せずに処理する
```

(次のページに続く)

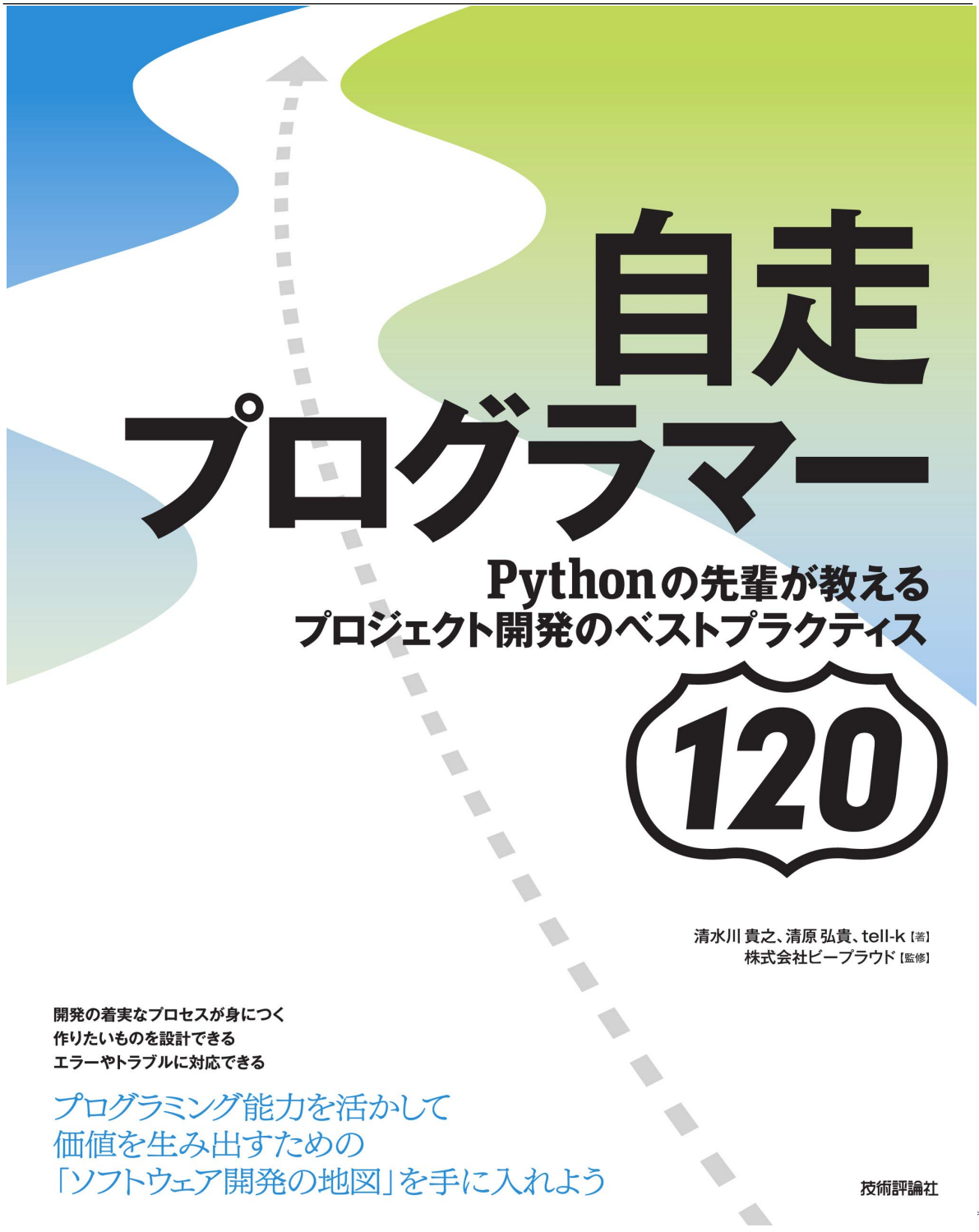
[*24] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
    for user in users:
        ...
    return users
```

コードを読むだけでは理解しにくいプログラムの場合、処理の意味と、なぜそう書くのか をコメントに書きましょう。

コメントは 「なぜこう処理しないのか」の説明 と考えても良いでしょう。プログラムを読んだ人が「当たり前に考えた」ときに違和感があるような処理に「なぜこのような処理をしているのか」を注釈する場合などに使うと効果的です。



[*25]

[*25] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*26] をご参照ください

2.1.11 11:コントローラーには処理を書かない

main() 関数や Web フレームワーク の コントローラー （Django の View）に処理を書きすぎてはいませんか？

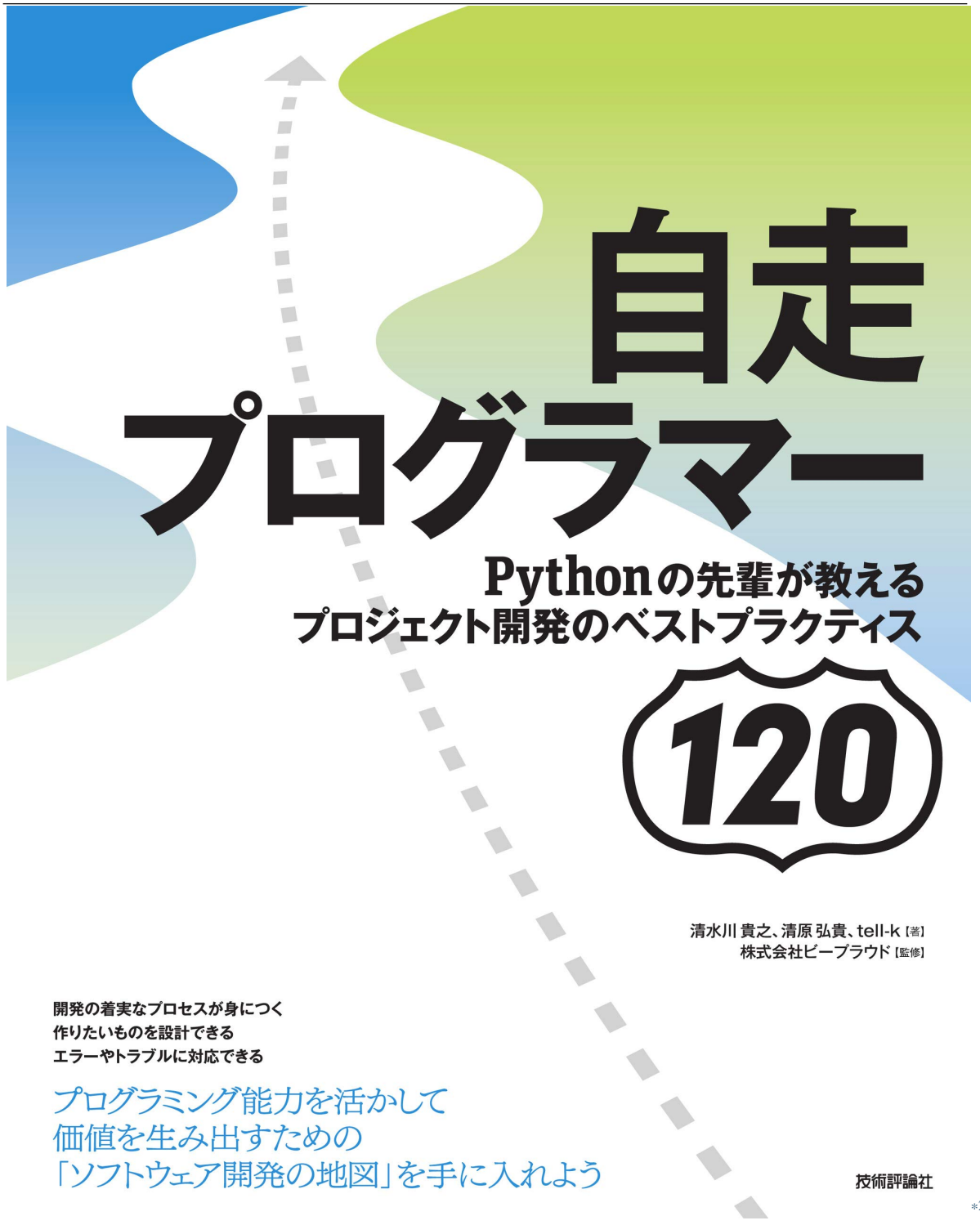
具体的な失敗

ここでは Web フレームワーク Django で、 View 関数 に書かれた処理を例に説明します。

```
@login_required
def item_list_view(request, shop_id):
    shop = get_object_or_404(Shop, id=shop_id)
    if not request.user.memberships.filter(role=Membership.ROLE_OWNER, shop=shop).
↪exists():
        return HttpResponseForbidden()

    items = Item.objects.filter(shop=shop,
                                published_at__isnull=False)
    if "search" in request.GET:
        search_text = request.GET["search"]
        if len(search_text) < 2:
            return TemplateResnponse(request, "items/item_list.html",
                                     {"items": items, "error": "文字列が短すぎます"})
        items = items.filter(name__contains=search_text)
    prices = []
    for item in items:
        price = int(item.price * 1.1)
        prices.append(f"{price:,}円")
    items = zip(items, prices)
    return TemplateResponse(request, "items/item_list.html", {"items": items})
```

このプログラムは item_list_view 関数に処理を書きすぎています。

[*26] https://gihyo.jp/book/2020/978-4-297-11197-7



*27

*27 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*28] をご参照ください

ベストプラクティス

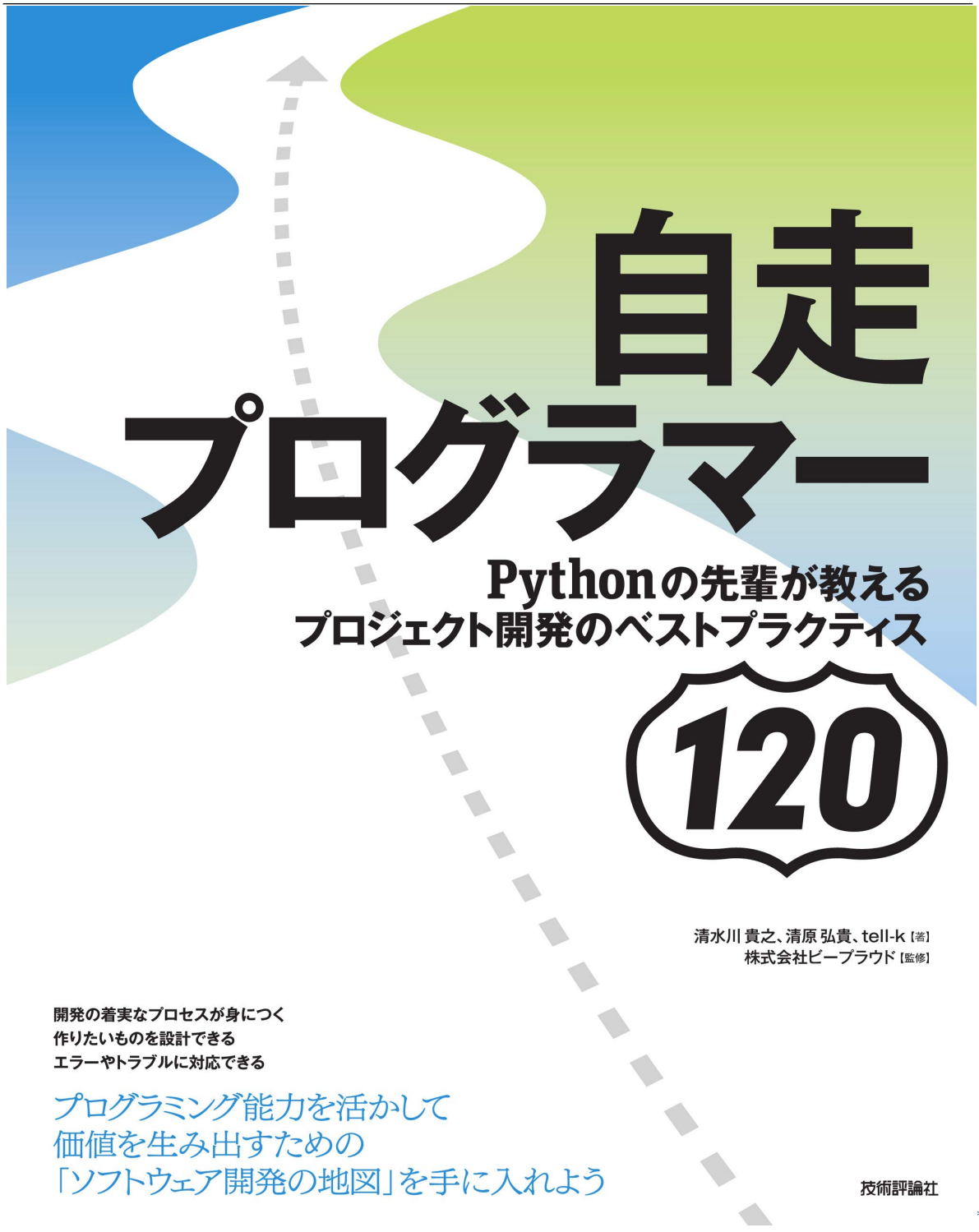
コントローラーでは値の入出力と、処理全体の制御のみ行うべきです。コントローラーに細かい処理まで実装すると、コントローラーに書かれるプログラムが多くなりすぎます。それでは処理全体の見通しが悪くなるだけでなく、 単体テスト もしにくくなります。上記の `item_list_view` View 関数内の処理もほとんどは別の関数や コンポーネント に分離して実装すべきです。

処理をそれぞれ分離したあとの `item_list_view` 関数は以下のようになります。

リスト 2.1　views.py

```
@login_required
def item_list_view(request, shop_id):
    shop = get_object_or_404(Shop, id=shop_id)
    validate_membership_permission(request.user, shop, Membership.ROLE_OWNER)

    items = Item.objects.filter(shop=shop).published()
    form = ItemSearchForm(request.GET)
    if form.is_valid():
        items = form.filter_items(items)
    return TemplateResponse(request, "items/item_list.html",
                            {"items": items, "form": form})
```

[*28] https://gihyo.jp/book/2020/978-4-297-11197-7



*29

*29 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*30] をご参照ください

[*30] https://gihyo.jp/book/2020/978-4-297-11197-7

2.2. クラス設計

2.2.1 12:辞書でなくクラスを定義する

クラスを作るのに抵抗感がありませんか？　積極的にクラスを定義する利点と、辞書で処理し続ける問題は何でしょうか。

具体的な失敗

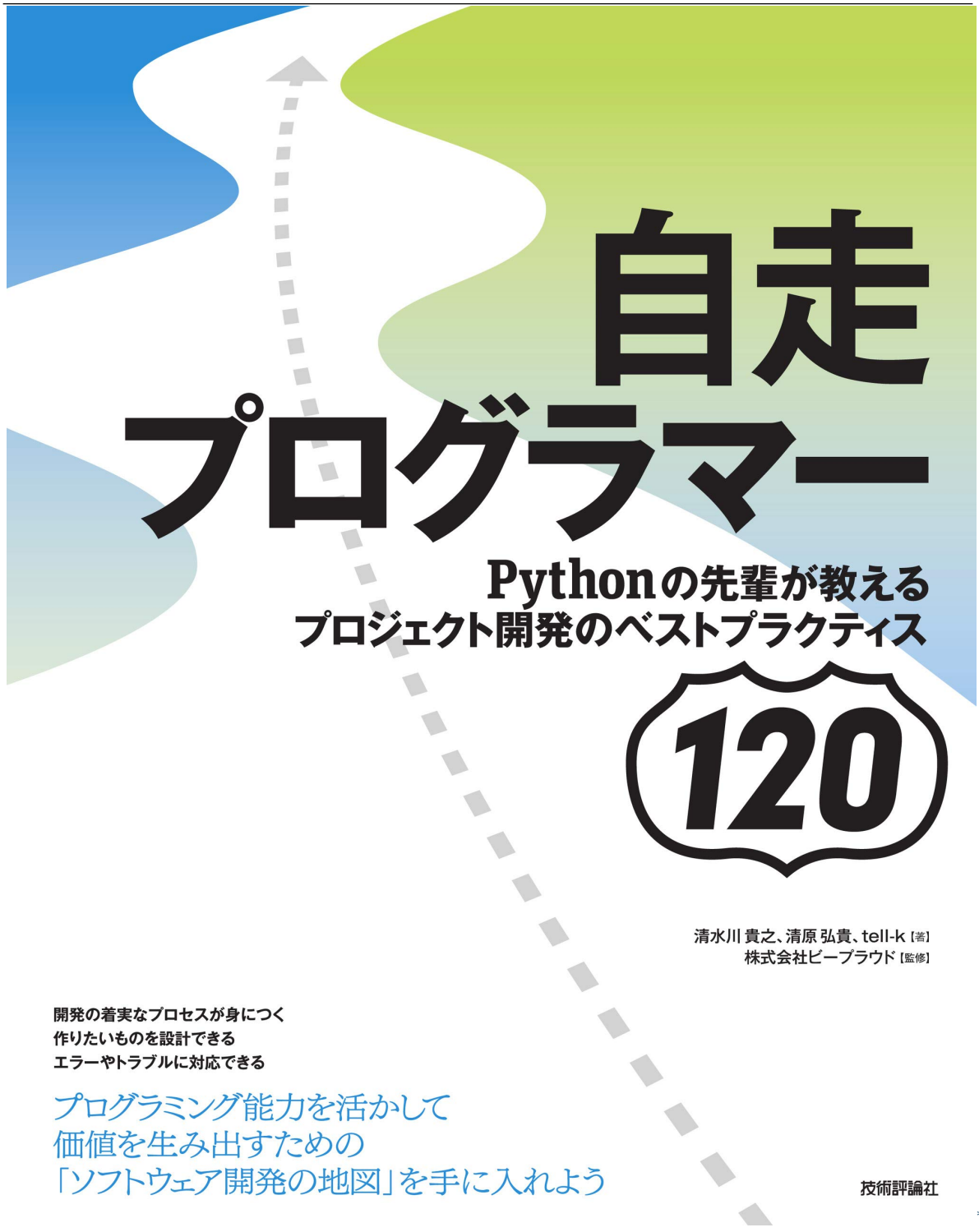
```
import json
from datetime import date


def get_fullname(user):
    return user['last_name'] + user['first_name']


def calc_age(user):
    today = date.today()
    born = user['birthday']
    age = today.year - born.year
    if (today.month, today.day) < (born.month, born.day):
        return age - 1
    else:
        return age


def load_user():
    with open('./user.json', encoding='utf-8') as f:
        return json.load(f)
```

この処理の問題は get_fullname などの関数が「ユーザー」という意味を持つ辞書を期待していることです。関数が「 特定のキーをもつ辞書 」に縛られるので、他の形式の辞書を渡しても正しく動作しません。関数にするのであれば、辞書でなく個別の引数として期待するべきです（ *7:*コレクションを引数にせず *int* や *str* を受け取る (ページ 31) 参照）。

ベストプラクティス

特定のキーを持つ辞書を期待するなら、クラスを定義しましょう。

```
import json
from dataclasses import dataclass
from datetime import date


@dataclass
class User:
    last_name: str
    first_name: str
    birthday: date

    # 解説:
    #     クラスにすることで、それぞれの処理をクラスのメソッドやプロパティーとして実装できます。
    #     ``user.fullname`` のように簡潔にプログラムを書けます。
    @property
    def fullname(self):
        return self.last_name + self.first_name

    @property
    def age(self):
        today = date.today()
        born = self.birthday
        age = today.year - born.year
        if (today.month, today.day) < (born.month, born.day):
            return age - 1
        else:
            return age


def load_user():
    with open('./user.json', encoding='utf-8') as f:
        return User(**json.load(f))
```



*31

*31 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*32] をご参照ください

2.2.2 13:dataclass を使う

クラス化したときの問題は、引数の多いクラスを定義するのが面倒な点です。こういった場合はどのように実装するのが良いでしょうか。

具体的な失敗

```
class User:
    def __init__(self, username, email, last_name, first_name, birthday, bio, role,␣
↪mail_confirmed=False):
        self.username = username
        self.email = email
        self.last_name = last_name
        self.first_name = first_name
        self.birthday = birthday
        self.bio = bio
        self.mail_confirmed = mail_confirmed
```

このプログラムが「問題」というわけではありませんが、冗長な印象があります。

ベストプラクティス

Python3.7 から使える dataclass を使いましょう。

```
from dataclasses import dataclass
from datetime import date


@dataclass
class User:
    username: str
    email: str
```

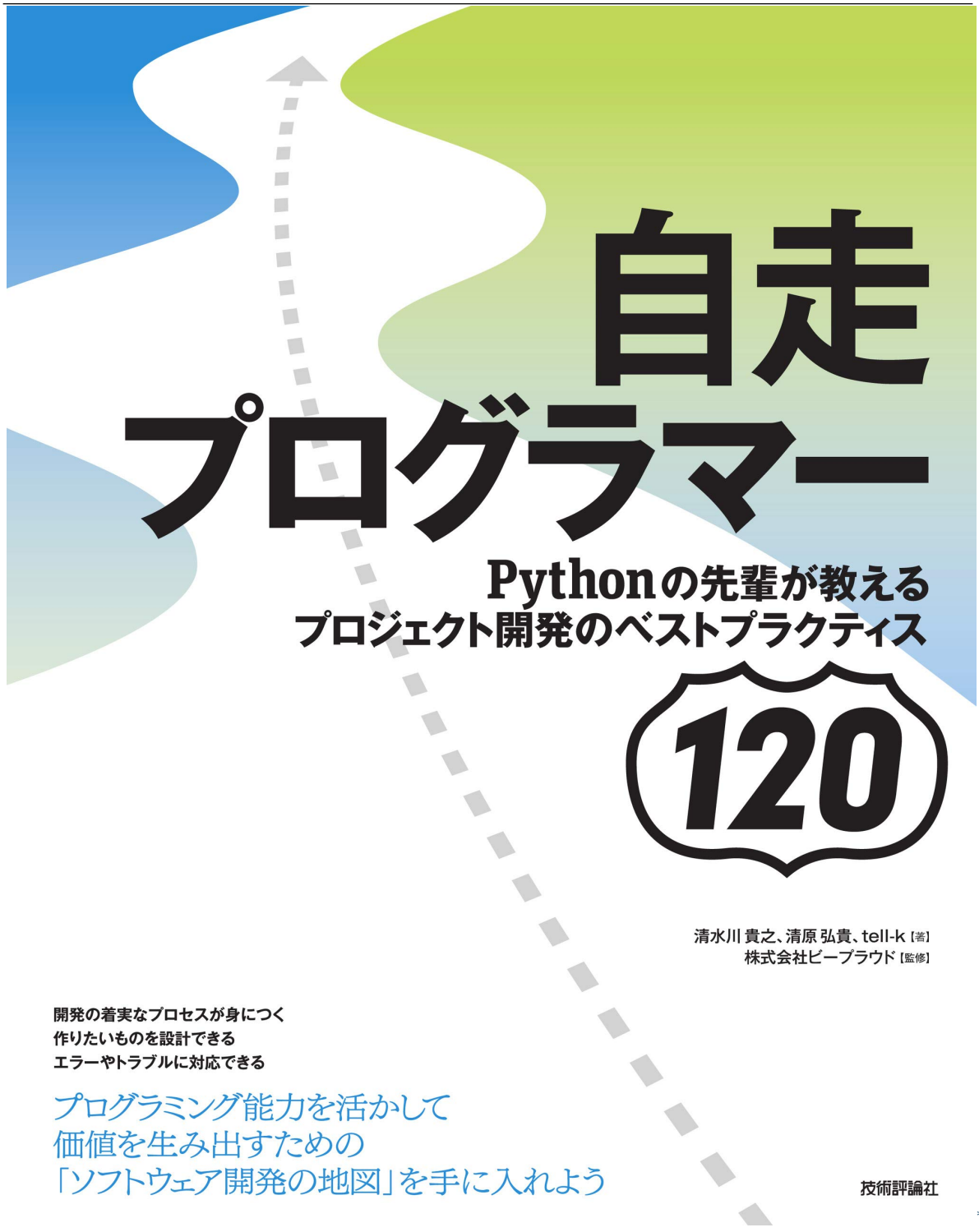
(次のページに続く)

[*32] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
    last_name: str
    first_name: str
    birthday: date
    role: str
    mail_confirmed: bool = False
```

__init__ メソッドの引数が多いクラスは dataclass を使うと良いでしょう。各引数の型と デフォルト引数を可読性高く設定できます。



*33

*33 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*34] をご参照ください

2.2.3 14:別メソッドに値を渡すためだけに属性を設定しない

クラスはとても有効ですが、 self の扱い方を間違えるとクラス内の処理が読みにくくなります。たとえば、次の例を見てください。

具体的な失敗

```
from datetime import date


class User:
    def __init__(self, username, birthday):
        self.username = username
        self.birthday = birthday
        self.age = None

    def calc_age(self):
        today = date.today()
        age = (self.birthday - today).years
        if (self.birthday.month, self.birthday.day) < (today.month, today.day):
            age -= 1
        self.age = age

    def age_display(self):
        return f"{self.age}歳"
```

このクラスでは self.age 属性を介して、 age_display メソッドが calc_age に依存しています。calc_age メソッドの前に age_display を呼び出してしまうと "None 歳" という文字が返されてしまいます。

__init__ 内で calc_age を呼び出すようにした場合も、 birthday を変更すると calc_age を呼び出す必要があります。そもそも「事前に他のメソッドを呼び出す必要がある」という設計にするのが良くありません。

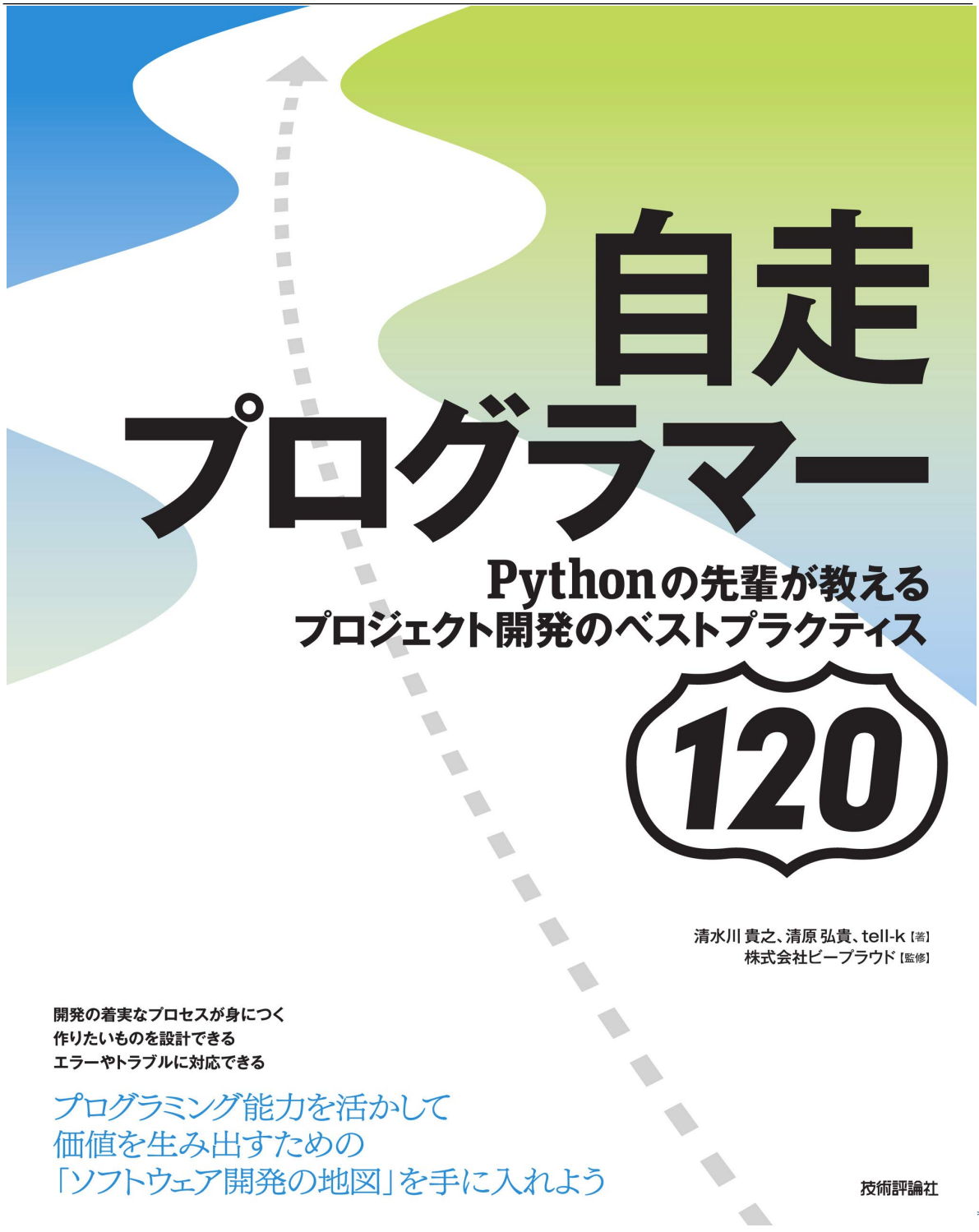
[*34] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

別のメソッドに値を渡すためだけに属性を設定するのはやめましょう。

```
from datetime import date


class User:
    def __init__(self, username, birthday):
        self.username = username
        self.birthday = birthday

    @property
    def age(self):
        today = date.today()
        age = (self.birthday - today).years
        if (self.birthday.month, self.birthday.day) < (today.month, today.day):
            age -= 1
        return age

    def age_display(self):
        return f"{self.age}歳"
```

age を属性という状態にするのでなくて、 @property に実装するほうが良いです。変数や属性という「状態」を減らすことで、考えるべきこと、覚えておくべきことが減らせるからです。



[*35]

[*35] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*36] をご参照ください

2.2.4 15:インスタンスを作る関数をクラスメソッドにする

クラスメソッドの使いどころは少し難しいかもしれません。具体的にどのようなクラスと関数の場合にクラスメソッドにできるか考えてみましょう。

具体的な失敗

```
from dataclasses import dataclass


@dataclass
class Product:
    id: int
    name: str


def retrieve_product(id):
    res = requests.get(f'/api/products/{id}')
    data = res.json()
    return Product(
        id=data['id'],
        name=data['name']
    )
```

このクラスと関数の実装は問題ありませんが、このクラスを使う別のモジュールから、 Product クラスと retrieve_product 関数をインポートする必要があります。

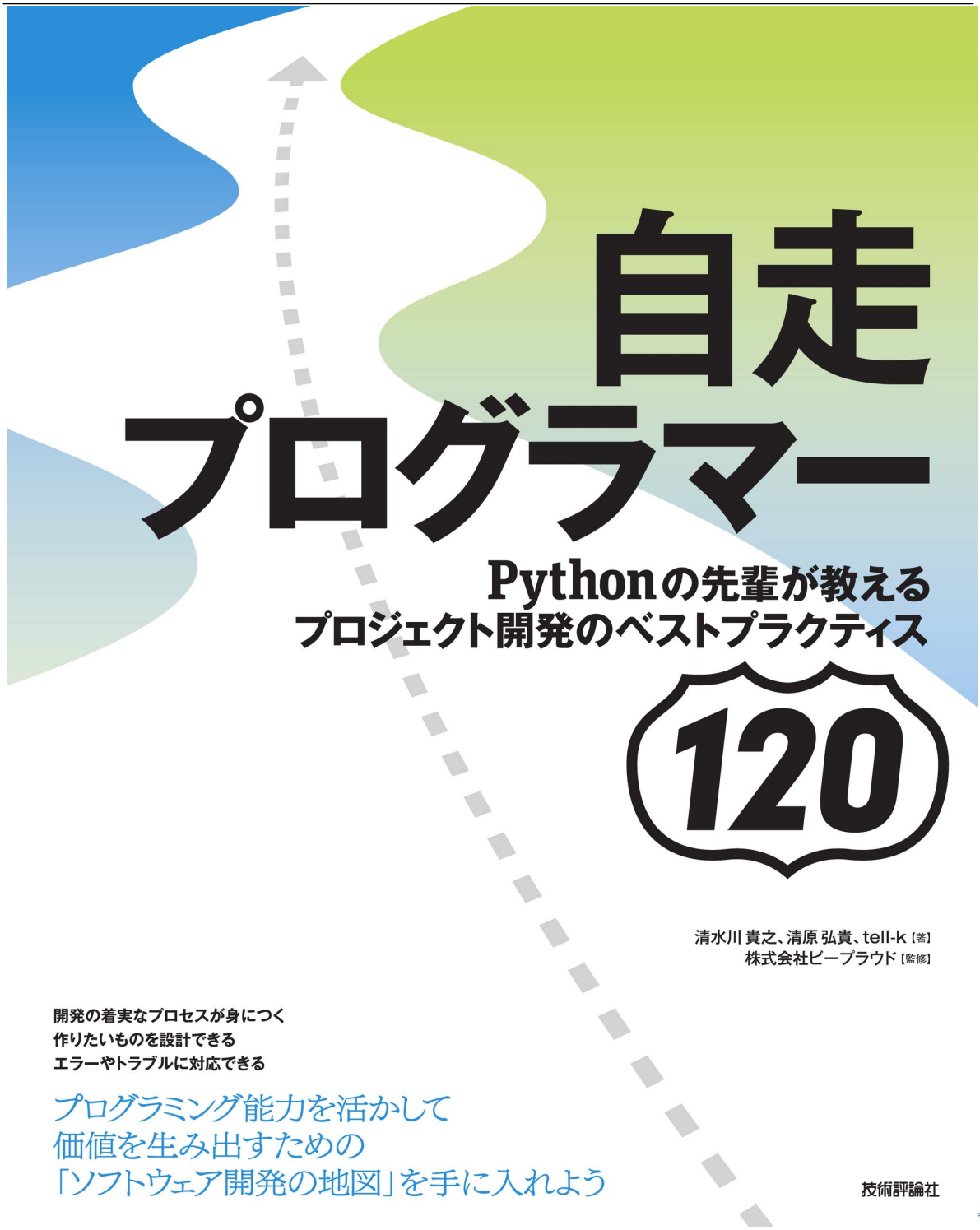
[*36] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

外部 API から requests で情報を取得する処理を retrieve_product_detail 関数に分離して、以下のように実装します。

```
from dataclasses import dataclass

from .dataapi import retrieve_product_detail


@dataclass
class Product:
    id: int
    name: str

    @classmethod
    def retrieve(cls, id: int) -> 'Product':
        """ データ API から商品の情報を取得して、インスタンスとして返す
        """
        data = retrieve_product_detail(id)
        return cls(
            id=data['id'],
            name=data['name'],
        )
```

このように実装する利点は、 Product をインポートすれば値を取得する処理も使えることです。



*37

*37 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*38] をご参照ください

関連

- *18:* **モジュール名のオススメ集** (ページ 67)

[*38] https://gihyo.jp/book/2020/978-4-297-11197-7

2.3. モジュール設計

2.3.1 16:utils.py のような汎用的な名前を避ける

Python のモジュール（Python ファイル）を分割するとき、とりあえずで utils.py という名前にしていませんか？

具体的な失敗

以下のような関数をすべて utils.py にまとめるのはやめましょう。

リスト 2.2　utils.py

```
from datetime import timedelta
from urllib.parse import urlencode

from payment.models import Purchase


def get_purchase(purchase_id):
    return Purchase.objects.filter(published_at__isnull=False).get(id=purchase_id)


def takeover_query(get_params, names):
    return urlencode({k: v for k, v in get_params.items() if k in names})


def date_range(start, end, step=1):
    current = start
    while current <= end:
        yield current
        current += timedelta(days=step)
```

utils.py というモジュール名はなるべく使わないのが良いでしょう。ビジネス上の仕様に深く関わる処理や、データの仕様などに関係する処理を、「ユーティリティ」というモジュールにまとめるのは不適切です。ユーティリティには「有益なもの」「便利なもの」くらいのニュアンスしかありません。

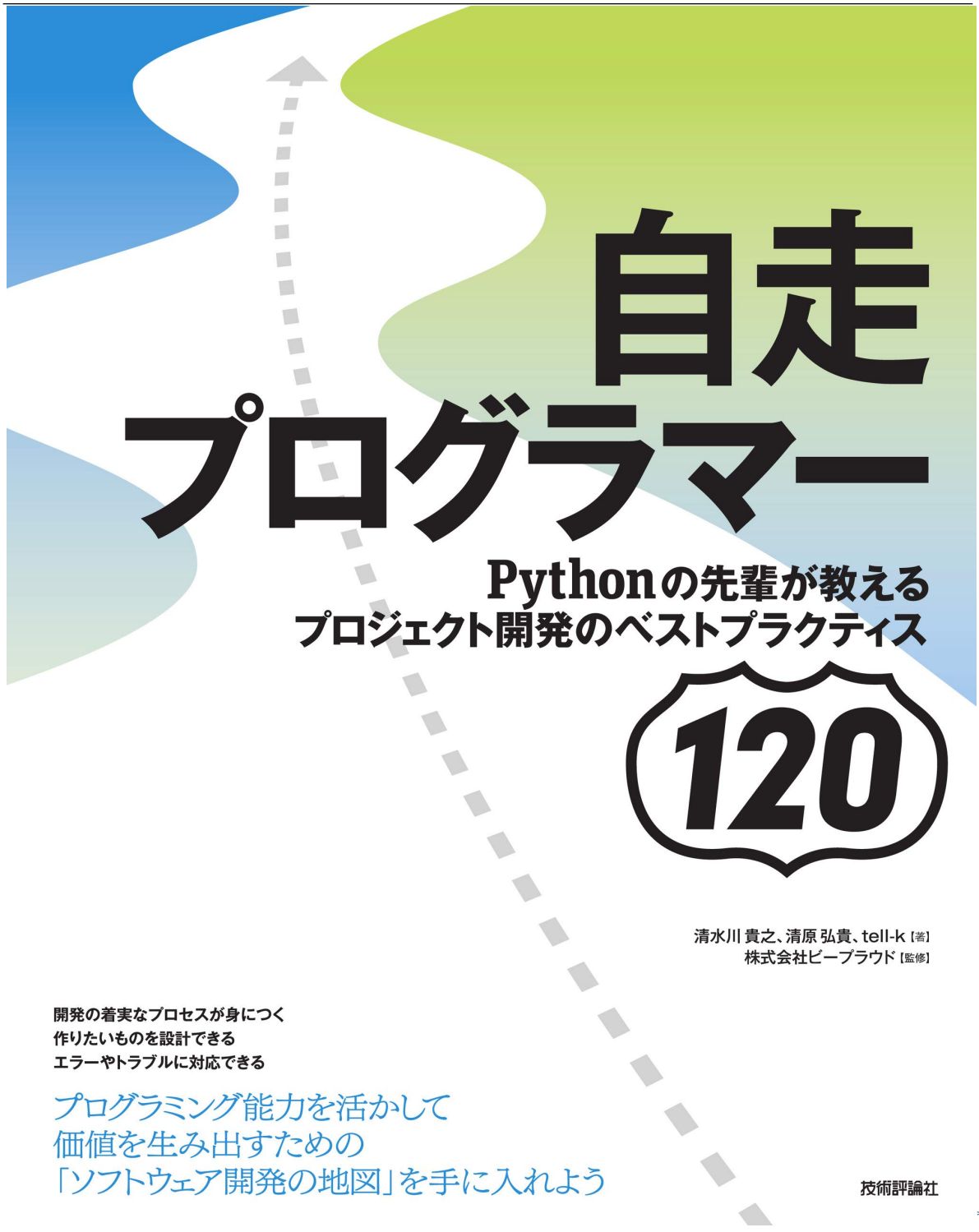
ベストプラクティス

まずデータをフィルターする処理は models.py などにまとめるのが良いです。Django フレームワークを使う場合は QuerySet のメソッドに実装できます。

リスト 2.3　models.py

```
from django.db import models


class PurchaseQuerySet(models.QuerySet):
    def filter_published(self):
        return self.filter(published_at__isnull=False)


class Purchase(models.Model):
    ...
    objects = PurchaseQuerySet.as_manager()
```

また、「リクエスト」に関係する処理であれば、 request.py など別のモジュールを作るのが良いでしょう。



*39

*39 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*40] をご参照ください

関連

- *17:*ビジネスロジックをモジュールに分割する (ページ 63)

2.3.2 17:ビジネスロジックをモジュールに分割する

モジュールを分割する際は「 ビジネスロジック 」を意識することが大切です。ビジネスロジックとは具体的な業務に必要な処理のことです。たとえば商品、購入、在庫などを扱うプログラムのことを言います。

ビジネスロジックとモジュール分割がどう関係するのでしょうか？

具体的な失敗

以下の例では コントローラー （View 関数）をまとめる `views.py` モジュールに、View 関数でない関数も記述してしまっています。

リスト 2.4　views.py

```
from some_payment_asp import purchase_item


def render_purchase_mail(item):
    return render_to_string('payment/item_purchase.txt', {'item': item})


def purchase(user, item, amount):
    purchase_item(user.card.asp_id, item.asp_id, amount=amount)
    PurchaseHistory.objects.create(item=item, user=request.user)
    body = render_purchase_mail(item)
    send_mail(
        '購入が完了しました',
        body,
        settings.PAYMENT_PURCHASE_MAIL',
        [user.email],
```

(次のページに続く)

[*40] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
        fail_silently=False,
    )


def item_purchase(request, item_id):
    item = get_object_or_404(Item, id=item_id)
    purchase(request.user, item, amount=1)
```

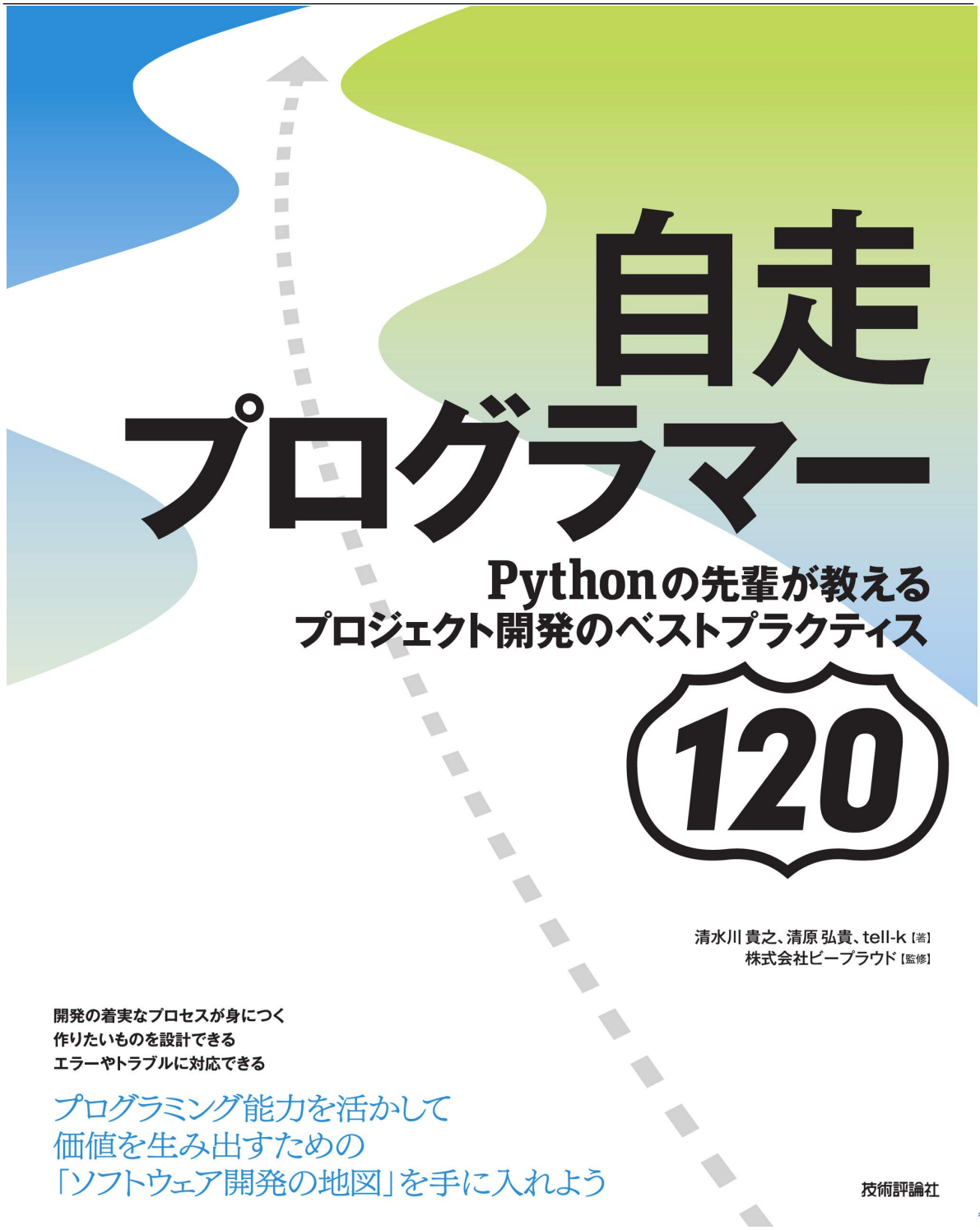
この場合 item_purchase だけが View 関数なのに、他の関数も View 関数のように見えてしまいます。より適切な別のモジュールに分割すべきです。

ベストプラクティス

ビジネスロジックを専用のモジュールに分割しましょう。モジュール名はこの場合、payment.py とするのが良いでしょう。

リスト 2.5　payment.py

```
from some_payment_asp import purchase_item


def render_purchase_mail(item):
    return render_to_string('payment/item_purchase.txt', {'item': item})


def purchase(user, item, amount):
    purchase_item(user.card.asp_id, item.asp_id, amount=amount)
    PurchaseHistory.objects.create(item=item, user=request.user)
    body = render_purchase_mail(item)
    send_mail(
        '購入が完了しました',
        body,
        settings.PAYMENT_PURCHASE_MAIL',
        [user.email],
        fail_silently=False,
     )
```



[*41]

[*41] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*42] をご参照ください

2.3.3 18:モジュール名のオススメ集

「モジュールを分けましょう」と指摘されても、具体的にどんな名前で分割すべきなのでしょう。ここでは失敗例と、オススメのモジュール名を説明します。

具体的な失敗

```
.
├── common.py
├── utils.py
└── main.py
```

モジュールの分割が少なく、一部のモジュールが大きくなりすぎるのは問題です。「商品を購入する関数はどこにある？」と疑問になったときに、探すのが難しくなります。また、1 つのファイルが大きすぎるとエディターが遅くなる問題や、複数人で開発したときに変更が衝突しやすくなる問題もあります。

ベストプラクティス

モジュールは、意味でまとめられるときに積極的に分割しましょう。さらに、モジュールが大きくなった場合はパッケージ（__init__.py のあるディレクトリー）にまとめると良いでしょう。

たとえば、商品（item）の一覧や購入をするプログラムのパッケージとモジュールの構造は以下のようになります。

```
.
├── api                 # 外部 API にアクセスする処理をまとめる
│   ├── __init__.py
│   ├── item.py          # 商品に関する API 処理をまとめる
│   └── user.py          # ユーザーに関する API 処理をまとめる
├── commands            # コマンドラインツールのサブコマンドをまとめる
│   ├── __init__.py
│   ├── list.py          # 商品の一覧を表示するコマンドの入出力を扱う処理をまとめる
```

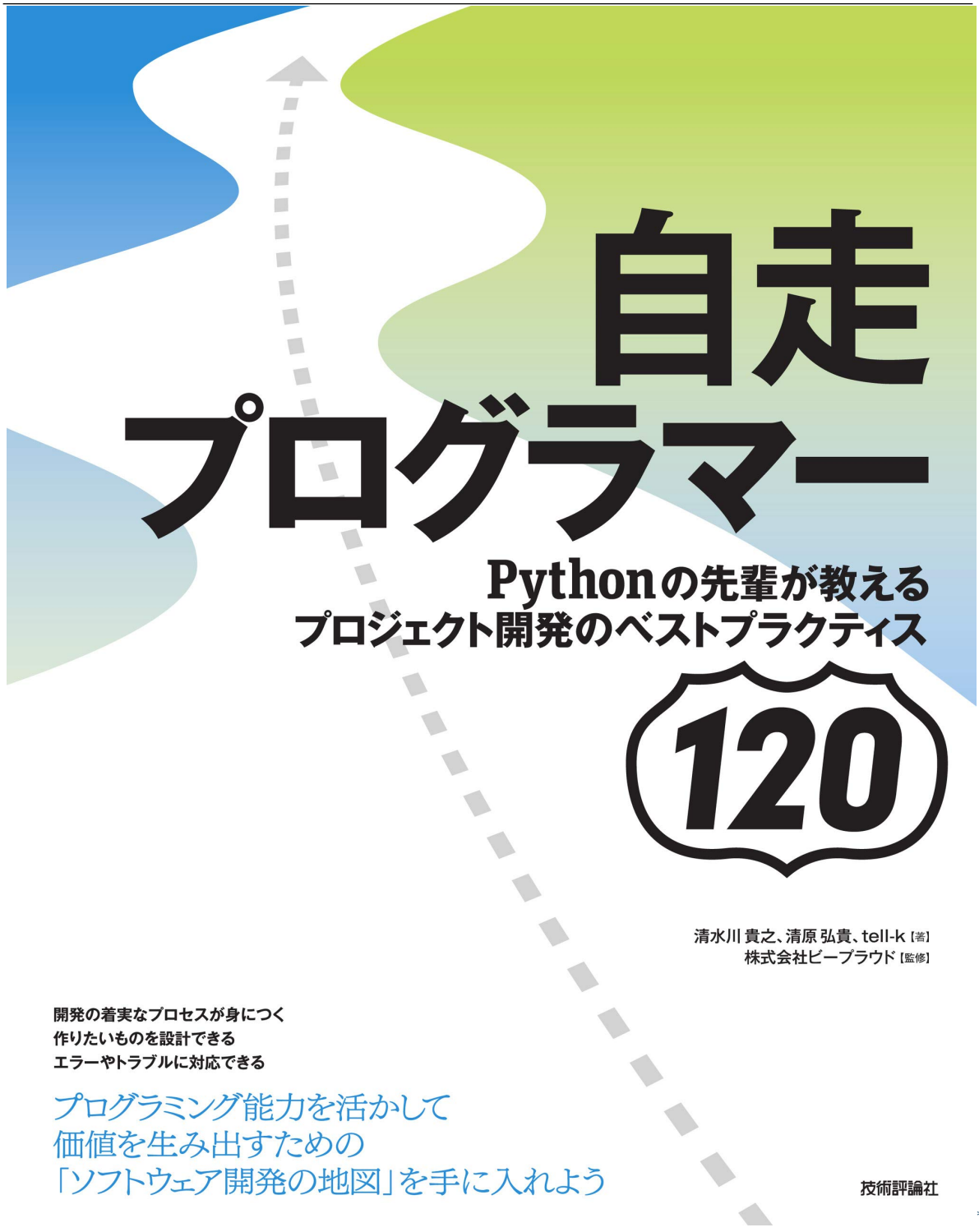
(次のページに続く)

[*42] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
|   └── purchase.py     # 商品の購入をするコマンドの入出力を扱う処理をまとめる
├── consts.py           # バックエンド API のホストなど定数をまとめる
├── main.py             # ツールのエントリーポイントの main 関数を置く
├── models.py           # 商品やユーザーのデータを永続化するクラスをまとめる
├── purchase.py         # 商品を購入する処理をまとめる
└── validators.py       # コマンドラインからの入力をチェックする処理をまとめる
```

フレームワーク の制約がある場合は基本的に従いましょう。たとえば Django の `views.py` 、 `models.py` 、 `urls.py` や `middlewares.py` 、Scrapy の `spiders.py` 、 `items.py` 、 `middlewares.py` があります。



[*43]

[*43] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*44] をご参照ください

関連

- *99:* **フレームワークの機能を知ろう** (ページ 360)

[*44] https://gihyo.jp/book/2020/978-4-297-11197-7

2.4. ユニットテスト

2.4.1 19:テストにテスト対象と同等の実装を書かない

テストを書けと言われるが、どう書けば良いかピンとこない という方は多いのではないでしょうか。次のような例はとてもありがちな失敗です。

具体的な失敗

以下の MD5 を計算する関数 calc_md5 の単体テストを考えましょう。

リスト 2.6　main.py

```
import hashlib


def calc_md5(content):
    content = content.strip()
    m = hashlib.md5()
    m.update(content.encode('utf-8'))
    return m.hexdigest()
```

この実装の単体テスト内で、実装内でも使われている hashlib.md5 を使ってはいけません。

リスト 2.7　tests.py

```
import hashlib
from main import calc_md5


def test_calc_md5():
    actual = calc_md5(" This is Content ")
    m = hashlib.md5()
    m.update(b"This is Content")
    assert actual == m.hexdigest()
```

よく見ると、テストの中に calc_md5 の実装と全く同じ処理が含まれています。これではテストが成功することは間違いないので、テストの意味がありません。実装で根本的に処理が間違っていても、テストが同じ

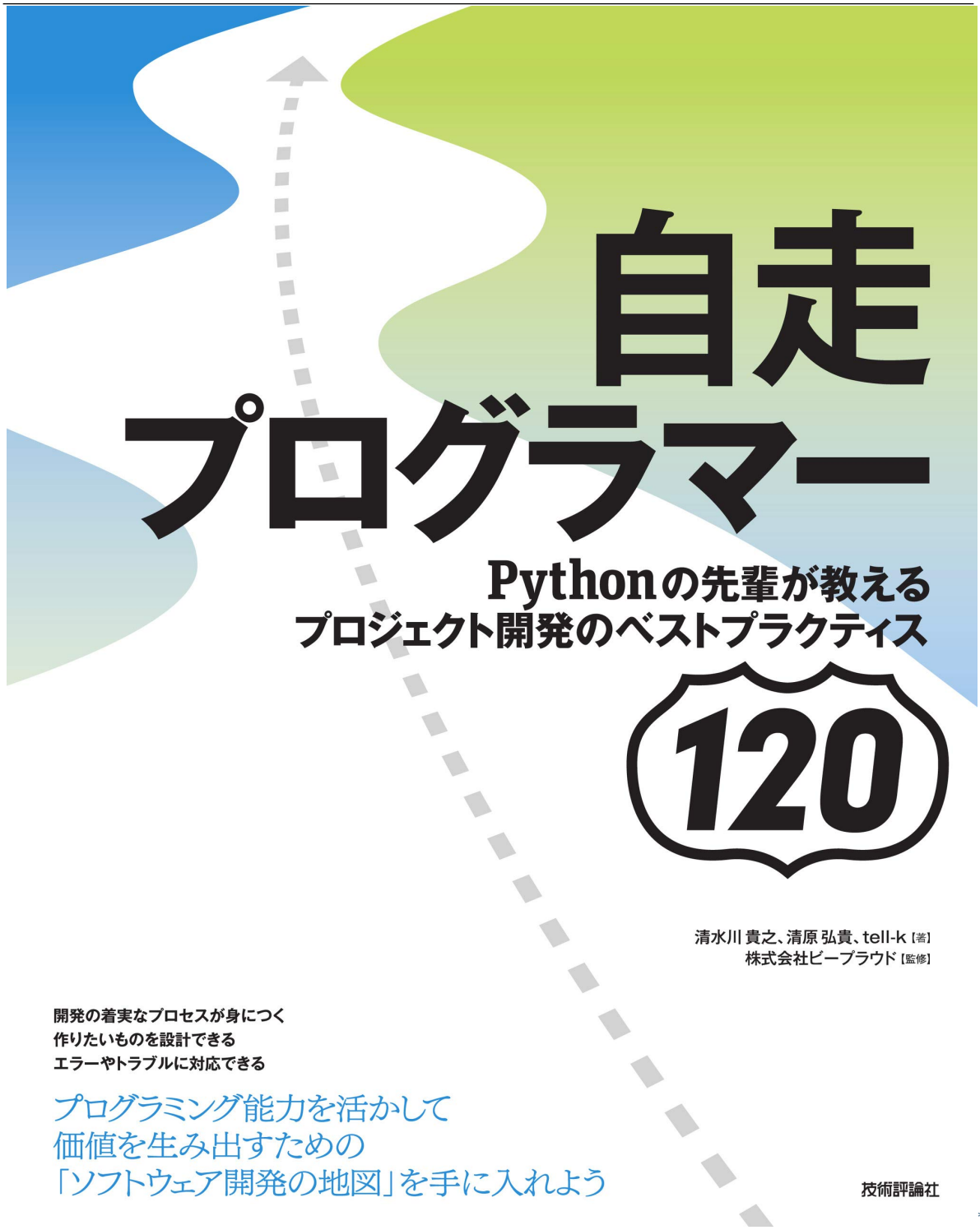
結果になるので間違いには気づけません。

ベストプラクティス

テスト内で入出力を確認するときは、文字列や数値などの値をテスト内に直接書きましょう。テスト内に、テスト対象 とほぼ同等の実装を書いてはいけません。

リスト 2.8　tests.py

```
from main import calc_md5


def test_calc_md5():
    actual = calc_md5(" This is Content ")
    assert actual == b"e61994e96b20e3965b61de16077e18c7"
```



[*45]

[*45] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*46] をご参照ください

2.4.2 20:1 つのテストメソッドでは 1 つの項目のみ確認する

単体テストの書き方がわかっても、どのように各単体テストを分割すればよいかを考えるのは難しいでしょう。次のような単体テストを書いてしまった経験はないでしょうか？

具体的な失敗

今回は次のような、単純な関数のテストを考えます。

```
def validate(text):
    return 0 < len(text) <= 100
```

1 つの単体テストに動作確認を詰め込みすぎてはいけません。

```
class TestValidate:
    def test_validate(self):
        assert validate("a")
        assert validate("a" * 50)
        assert validate("a" * 100)
        assert not validate("")
        assert not validate("a" * 101)
```

こうすると、 test_validate は「 validate 関数の何を確認しているのか」がわからなくなります。単体テストを実行してエラーになったときも「 test_validate でエラーがあった」と表示されるので、具体的にどういうケースでエラーがあったのかがわかりません。

ベストプラクティス

1 つの テストメソッド では、1 つの項目のみ確認するようにしましょう。

```
class TestValidate:
    def test_valid(self):
        """ 検証が正しい場合
```

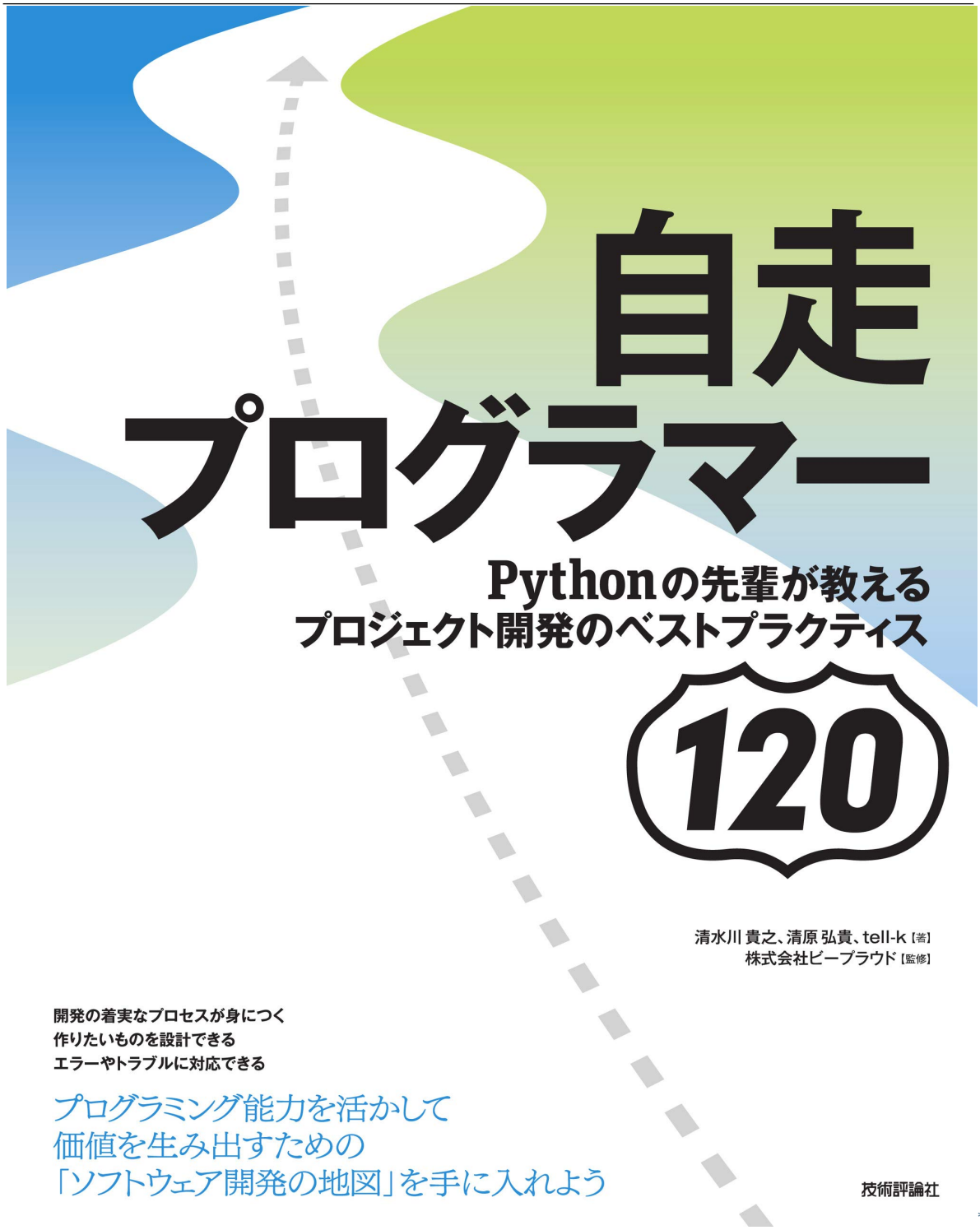
(次のページに続く)

[*46] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
        """
        assert validate("a")
        assert validate("a" * 50)
        assert validate("a" * 100)

    def test_invalid_too_short(self):
        """ 検証が正しくない: 文字が短すぎる場合
        """
        assert not validate("")

    def test_invalid_too_long(self):
        """ 検証が正しくない: 文字が長すぎる場合
        """
        assert not validate("a" * 101)
```

このテストでは 3 つのメソッドに分割しています。テストメソッドの名前を明確にすると、その名前からテストしている内容がわかります。docstring も書くとよりわかりやすくなります。



*47

*47 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*48] をご参照ください

2.4.3 21:テストケースは準備、実行、検証に分割しよう

他人の書いたテストコードを見たときに漠然と理解しづらいと思ったことはありませんか？ここではテストケースの見やすい書き方について紹介します。

プログラミング迷子**:** テストコードはゴチャっとしてるもの？

- 後輩 W：んー？
- 先輩 T：どうしたの？
- 後輩 W：自分で以前書いたテストコードを見直したいんですけど、ぱっと見どこを直していいのかわからないんですよね。
- 先輩 T：なるほどーちょっと見てみよう。
- 後輩 W：はい。
- 先輩 T：これはもう少しテストケースのコードを見やすく分けたほうがいいね。
- 後輩 W：どういうことですか？
- 先輩 T：だいたいユニットテストのテストケースでやることって、テスト対象を実行するための準備と、対象の実行、最後に検証（アサート）っていう 3 段階に分かれるんだよ。だからその 3 つに分けてテストケースのコードを書いておくと、あとで他の人が見てもわかりやすいってことだね。
- 後輩 W：ふむふむ。わかりました。やってみます。

具体的な失敗

これは Django アプリで会員登録をする API のテストコードです。どこまでがテストの準備で、どこからがテスト対象の実行か区別がつきますか？

```
class TestSignupAPIView:
```

(次のページに続く)

[*48] https://gihyo.jp/book/2020/978-4-297-11197-7

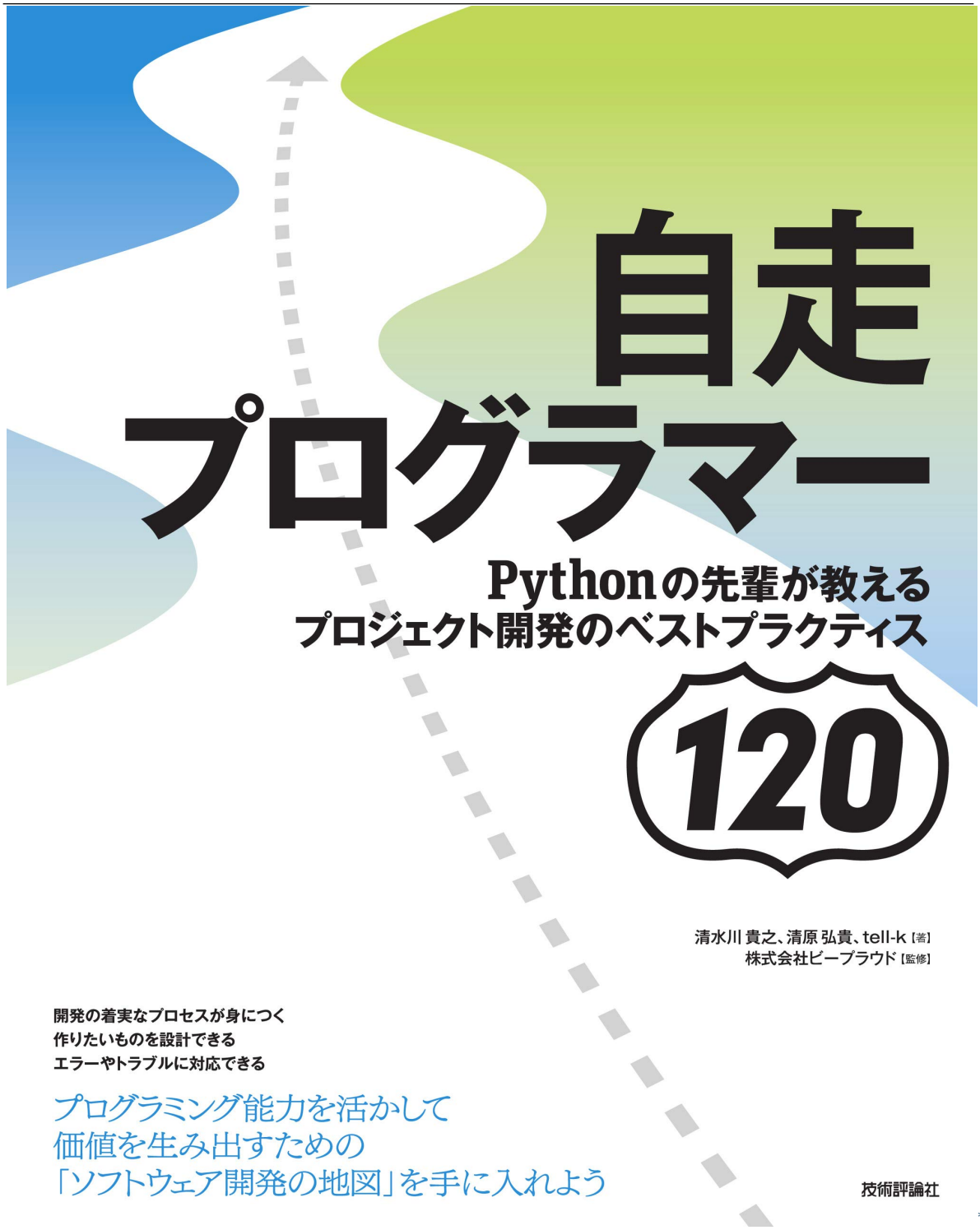
(前のページからの続き)

```
    @pytest.fixture
    def target_api(self):
        return "/api/signup"

    def test_do_signup(self, target_api, django_app):
        from account.models import User
        params = {
            "email": "signup@example.com"
            "name": "yamadataro",
            "password": "xxxxxxxxxxx",
        }
        res = django_app.post_json(target_api, params=params)
        user = User.objects.all()[0]
        expected = {
            "status_code": 201,
            "user_email": "signup@example.com",
        }
        actual = {
            "status_code": res.status_code,
            "user_email": user.email,
        }
        assert expected == actual
```

開発者はテストコードを手がかりにテスト対象処理の用途や仕様を確認します。テストコードが見づらかったり、理解しづらかったりすると、リファクタリングやテストの修正にも無駄に時間を費やしてしまいます。

ベストプラクティス

読みやすくするために、テストコードを 準備（**Arrange**）と 実行（**Act**）と 検証（**Assert**）に分けましょう。



[*49]

[*49] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*50] をご参照ください

コラム： **Arrange Act Assert** パターン

ここで紹介したテクニックは、Arrange Act Assert パターンとして知られています。興味のある人はぜひ原文を読んでみてください。

- Arrange Act Assert http://wiki.c2.com/?ArrangeActAssert

2.4.4 22:単体テストをする観点から実装の設計を洗練させる

単体テストの意味は何でしょうか？もちろんテスト対象の動作を保証することも大切ですが、「単体テストしやすいか？」という観点から実装の設計を洗練させることも大切です。「テストしにくい実装は設計が悪い」という感覚を身につけましょう。

具体的な失敗

まずテスト対象になる、イマイチな設計の関数を見てみましょう。この関数は `sales.csv` を読み込んで、合計の金額と、CSV ファイルから読み込んだデータのリストを返します。

リスト 2.9　sales.py

```
import csv


def load_sales(sales_path='./sales.csv'):
    sales = []
    with open(sales_path, encoding="utf-8") as f:
        for sale in csv.DictReader(f):
            # 値の型変換
            try:
                sale['price'] = int(sale['price'])
                sale['amount'] = int(sale['amount'])
            except (ValueError, TypeError, KeyError):
                continue
```

(次のページに続く)

[*50] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
            # 値のチェック
            if sale['price'] <= 0:
                continue
            if sale['amount'] <= 0:
                continue
            sales.append(sale)

    # 売上の計算
    sum_price = 0
    for sale in sales:
        sum_price += sale['amount'] * sale['price']
    return sum_price, sales
```

この関数をテストしようとすると、以下のようになります。

リスト 2.10　tests.py

```
class TestLoadSales:
    def test_invalid_row(self, tmpdir):
        test_file = tmpdir.join("test.csv")
        test_file.write("""id,item_id,price
1,1,100
2,1,100
""")
        sum_price, actual_sales = load_sales(test_file.strpath)
        assert sum_price == 0
        assert len(actual_sales) == 0

    def test_invalid_type_amount(self, tmpdir):
        # 解説: テストのたびに CSV ファイルを毎度用意する必要がある

        test_file = tmpdir.join("test.csv")
        test_file.write("""id,item_id,price,amount
1,1,100,foobar
2,1,200,2
""")
        sum_price, actual_sales = load_sales(test_file.strpath)
```

(次のページに続く)

(前のページからの続き)

```
        assert sum_price == 400
        assert len(actual_sales) == 1

    def test_invalid_type_price(self):
        ...

    def test_invalid_value_amount(self):
        ...

    def test_invalid_value_price(self):
        ...

    def test_sum(self):
        ...
```

load_sales 関数をテストするときは、毎度 CSV ファイルを用意する必要があり面倒です。無効な行がある場合を確認するとき、値が無効なとき、価格が無効なときなど、個別の確認をするために CSV ファイルの用意が必要です。小さな違いの確認のために、たくさんコードを書く必要があります。

ベストプラクティス

単体テストを通して、テスト対象コードの設計を見直しましょう。

- 関数の引数や フィクスチャー に大げさな値が必要な設計にしない
- 処理を分離して、すべての動作確認にすべてのデータが必要ないようにする
- 関数やクラスを分離して、細かいテストは分離した関数、クラスを対象に行う（分離した関数を呼び出す関数では、細かいテストは書かないようにする）

元の処理も以下のように改善しました。

リスト 2.11　sales.py

```
import csv
from dataclasses import dataclass
from typing import List
```

(次のページに続く)

(前のページからの続き)

```
# 解説: 売上（CSV の各行）を表すクラスに分離する
@dataclass
class Sale:
    id: int
    item_id: int
    price: int
    amount: int

    def validate(self):
       if sale['price'] <= 0:
           raise ValueError("Invalid sale.price")
       if sale['amount'] <= 0:
           raise ValueError("Invalid sale.amount")

   # 解説: 各売上の料金を計算する処理を Sales に実装
    @property
    def price(self):
        return self.amount * self.price


@dataclass
class Sales:
    data: List[Sale]

    @property
    def price(self):
        return sum(sale.price for sale in self.data)

    @classmethod
    def from_asset(cls, path="./sales.csv"):
        data = []
        with open(path, encoding="utf-8") as f:
            reader = csv.DictReader(f)
            for row in reader:
                try:
                    sale = Sale(**row)
```

(次のページに続く)

(前のページからの続き)

```
                    sale.validate()
                except Exception:
                    # TODO: Logging
                    continue
                data.append(sale)
        return cls(data=data)
```

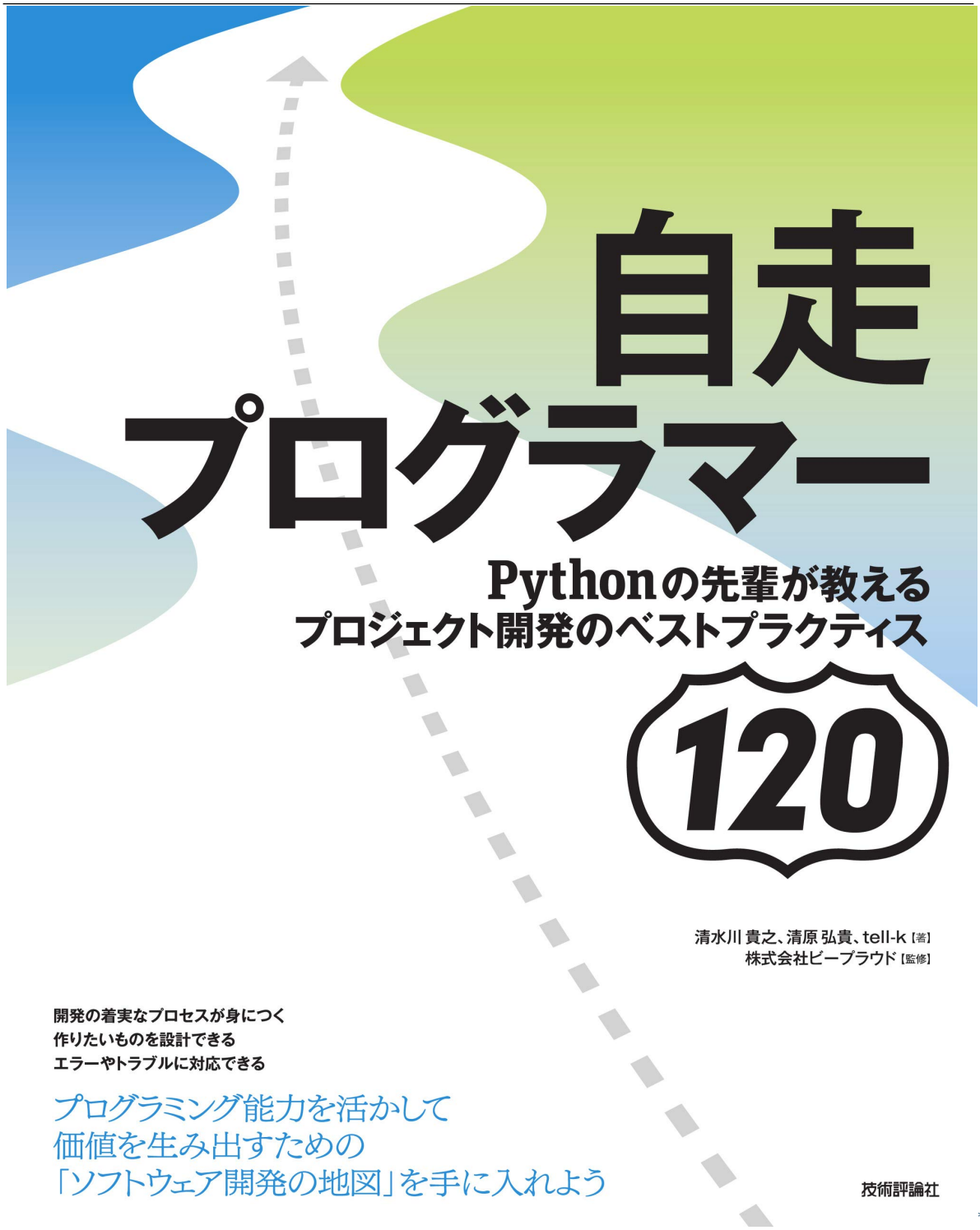
プログラムの行数は少し長くなりましたが、テストのしやすさ、 再利用性 、 可読性 が向上しています。単体テストも、各クラス Sale や Sales ごとに細かく書けます。

```
import pytest


class TestSale:
    def test_validate_invalid_price(self):
        # 解説: 値の確認をするテストで CSV を用意する必要がなくなった
        sale = Sale(1, 1, 0, 2)
        with pytest.raises(ValueError):
            sale.validate()

    def test_validate_invalid_amount(self):
        sale = Sale(1, 1, 1000, 0)
        with pytest.raises(ValueError):
            sale.validate()

    def test_price(self):
        ...


class TestSales:
    def test_from_asset_invalid_row(self):
        ...

    def test_from_asset(self):
        ...
```

(次のページに続く)

(前のページからの続き)

```
    def test_price(self):
        ...
```



[*51]

[*51] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*52] をご参照ください

関連

- *26:*テストケース毎にテストデータを用意する (ページ 99)
- *31:*過剰な *mock* を避ける (ページ 117)

2.4.5 23:テストから外部環境への依存を排除しよう

単体テストを書くときは、テストが外部環境に依存しないように注意しましょう。

次のような単体テストを書いたことはありませんか？

具体的な失敗

以下の実装のテストを考えましょう。

リスト 2.12　api.py

```
import requests


def post_to_sns(body):
    # 解説: この行で外部にアクセスしている
    res = requests.post('https://the-sns.example.com/posts', json={"body": body})
    return res.json()


def get_post(post_id):
    res = requests.get(f'https://the-sns.example.com/posts/{post_id}')
    return res.json()
```

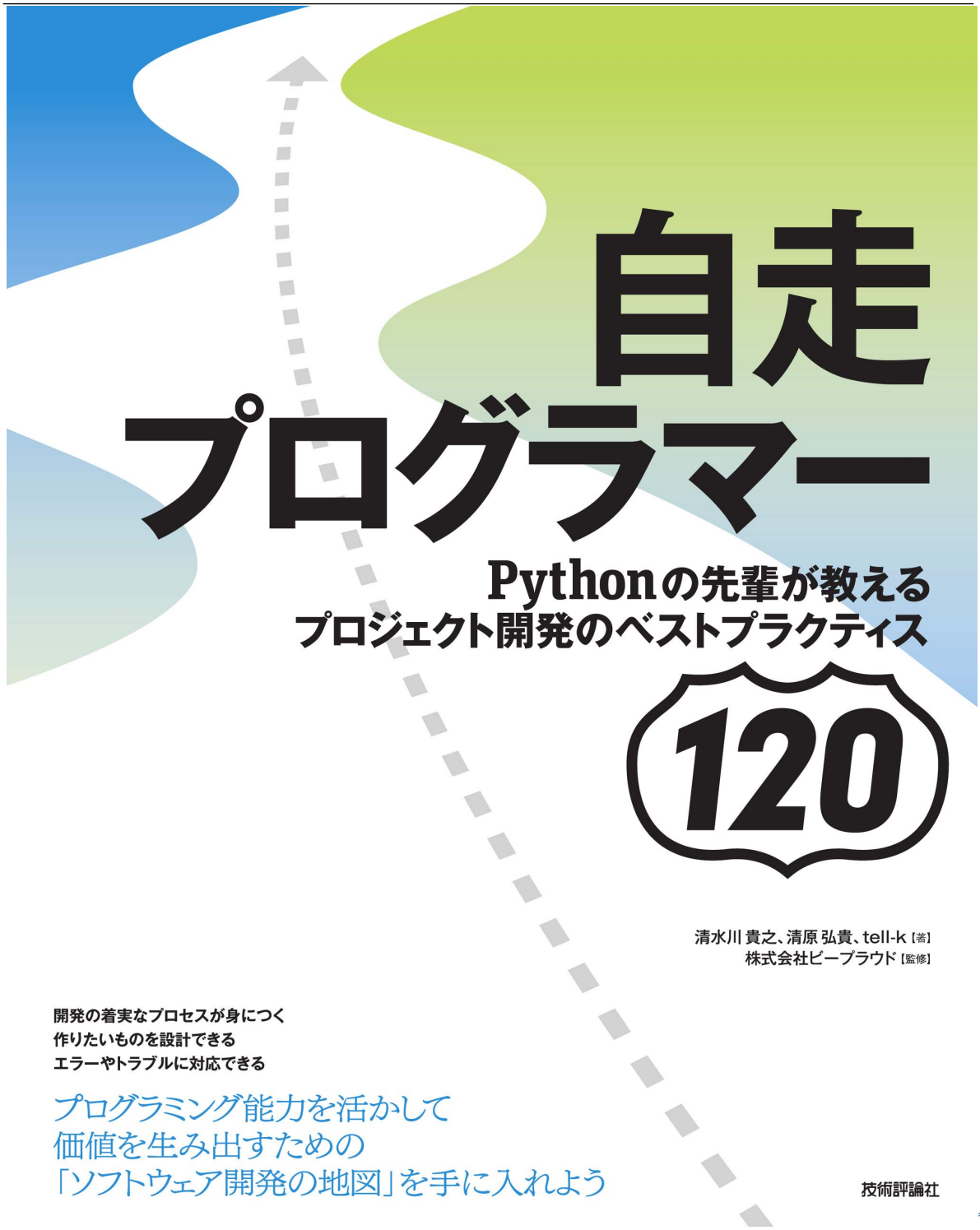
このテスト対象のように、外部へのアクセスが発生する処理を単純にテストしてはいけません。

[*52] https://gihyo.jp/book/2020/978-4-297-11197-7

リスト 2.13　tests.py

```
import requests

from .api import get_post, post_to_sns


class TestPostToSns:
    def test_post(self):
        data = post_to_sns("投稿の本文")
        assert data['body'] == "投稿の本文"

        data2 = get_post(data['post_id'])
        assert data2['post_id'] == data['post_id']
        assert data2['body'] == "投稿の本文"
```

外部へアクセスするテストを避けるべき理由は以下です。



*53

*53 https://gihyo.jp/book/2020/978-4-297-11197-7

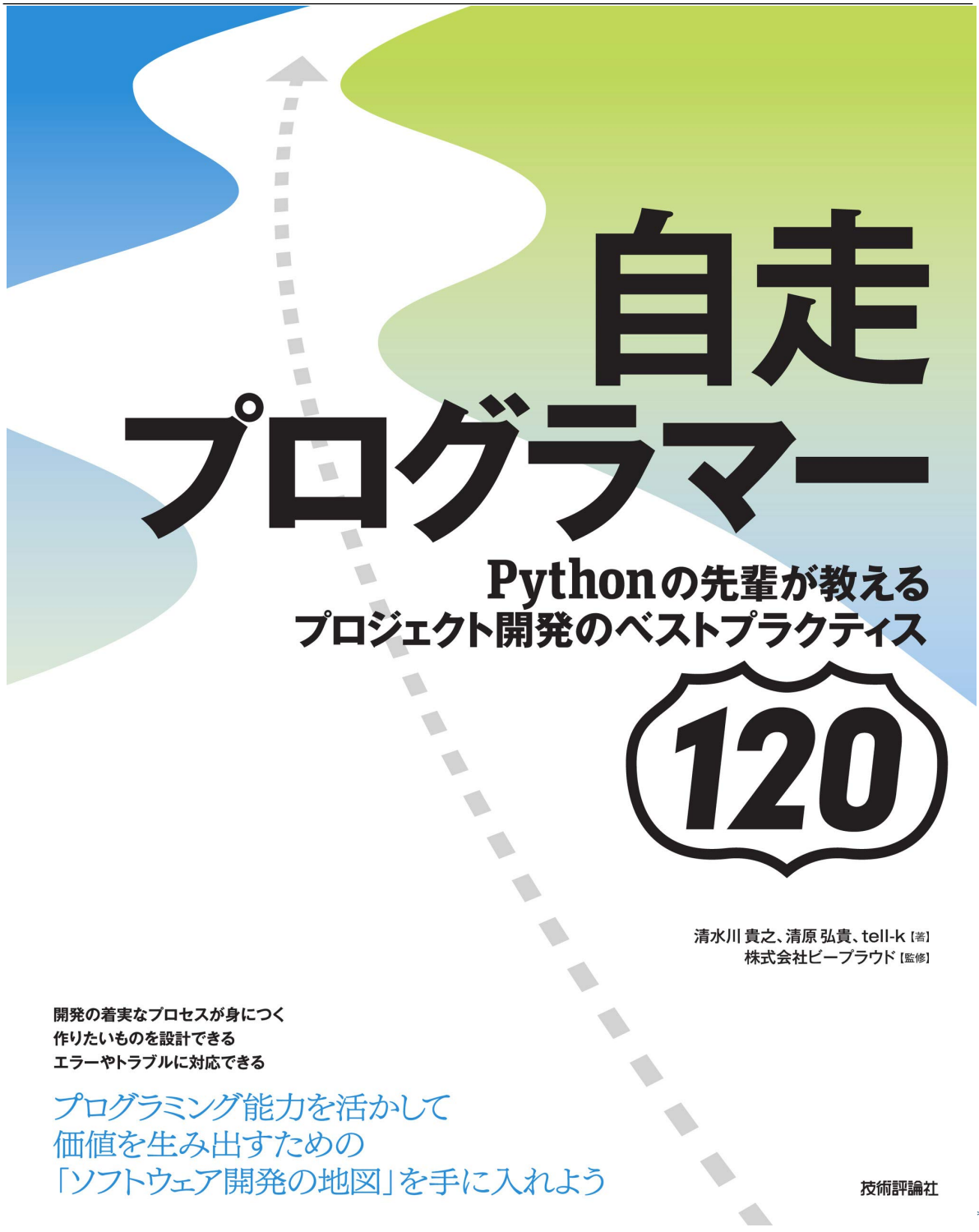
（中略）詳細は書籍 自走プログラマー[*54] をご参照ください

ベストプラクティス

単体テストから外部環境への依存を排除しましょう。requests がバックエンドサーバーへアクセスするのを、 responses[*55] を使ってモックしましょう。

```
import responses

from .api import get_post, post_to_sns


class TestPostToSns:
    @responses.activate
    def test_post(self):
        # 解説: 外部環境へのアクセスを、responses を使ってモックしている
        responses.add(responses.POST, 'https://the-sns.example.com/posts',
                      json={"body": "レスポンス本文"})

        data = post_to_sns("投稿の本文")

        assert data['body'] == "レスポンス本文"

        # 解説: 正しく外部アクセスが呼び出されたことを確認する
        assert len(responses.calls) == 1
        assert responses.calls[0].request.body == '{"body": "投稿の本文"}'
```

[*54] https://gihyo.jp/book/2020/978-4-297-11197-7
[*55] https://github.com/getsentry/responses



*56

*56 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*57] をご参照ください

2.4.6 24:テスト用のデータはテスト後に削除しよう

テスト用に生成されたテスト用のファイルが原因で、別のテストが失敗してしまったり不必要にマシンのディスク容量を占有してしまったりします。ここでは、テスト用のデータやファイルをどう扱うべきかについて説明します。

具体的な失敗例

事前準備として pytest の setup_method でテスト用の CSV を生成するコード例です。このコードは実行時にテスト用の CSV が用意されますが、ずっとファイルが残り続けます。

```
class TestImportCSV:

    def setup_method(self, method):

        self.test_csv = 'test_data.csv' # <- 削除されないのでテスト実行後も永遠に残り続ける
        with open(self.test_csv, mode="w", encoding="utf-8") as fp:
            fp.writelines([
                'Spam,Ham,Egg\n',
                'Spam,Ham,Egg\n',
                'Spam,Ham,Egg\n',
                'Spam,Ham,Egg\n',
                'Spam,Ham,Egg\n',
            ])

    def test_import(self):
        from spam.hoge import import_csv
        from spam.models import Spam

        import_csv(self.test_csv)
```

(次のページに続く)

[*57] https://gihyo.jp/book/2020/978-4-297-11197-7

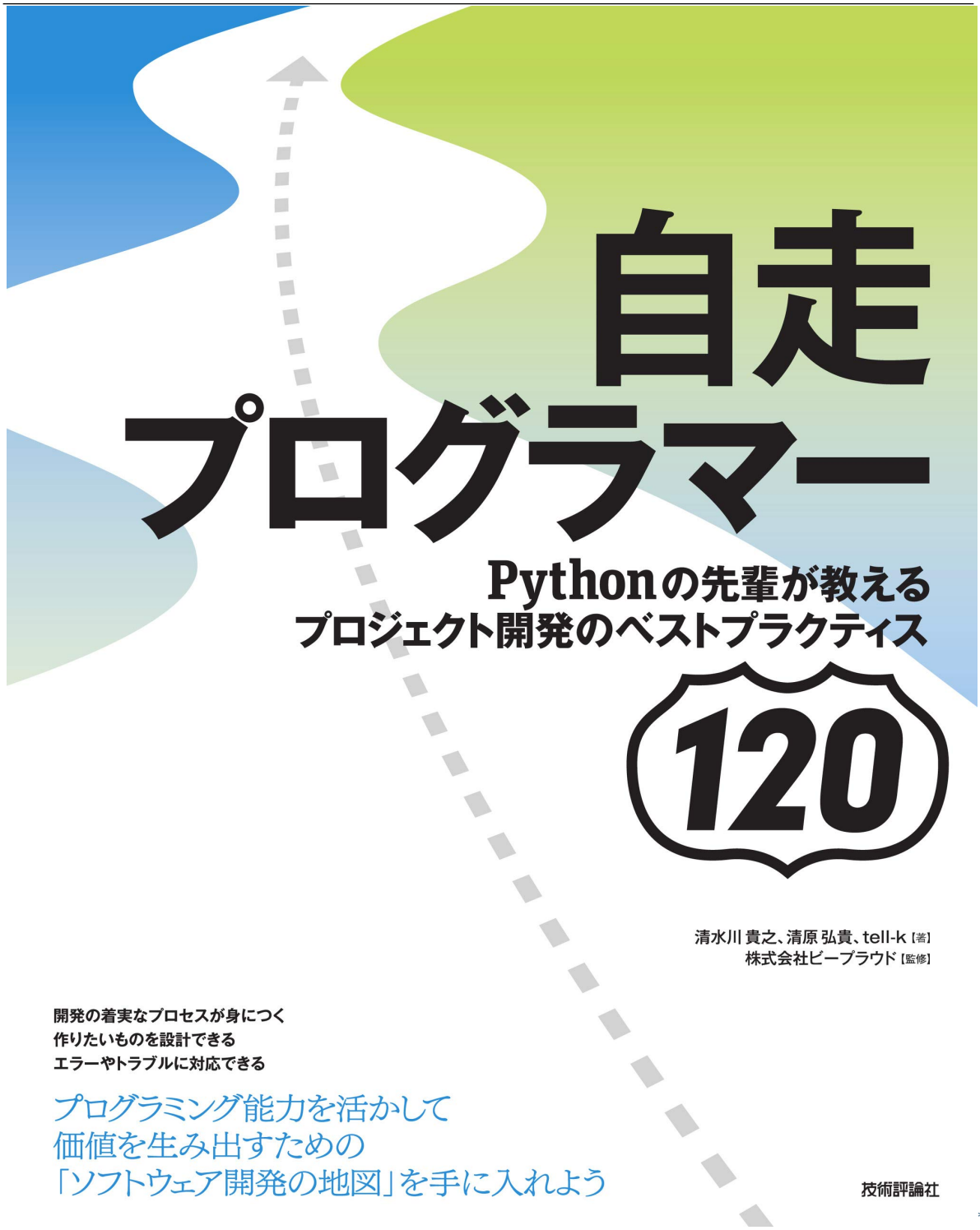
(前のページからの続き)

```
        assert Spam.objects.count() == 5
```

この CSV はわずか 5 件ですが、それでも意識的に削除しない限りマシンのディスク容量を占有してしまいます。また削除されていないので、誤って他のテストケースが参照した場合、テストが失敗してしまう可能性があります。

ベストプラクティス

テスト用の一時的なファイルを作ったときは、テストケースが終わるタイミングで削除しましょう。なるべく他のテストケースに影響を与えない状態を作れるように工夫できると良いです。



*58

*58 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*59] をご参照ください

2.4.7 25:テストユーティリティーを活用する

テストを書くときは、なるべく便利なユーティリティーを活用しましょう。オープンソースとして公開されているライブラリがたくさんあるので、手持ちの知識を蓄えておきましょう。

具体的な失敗

Django の View 関数をテストするプログラムから、その失敗を学びましょう。

```
import pytest

from .models import Organization, Post, User


class TestPostDetailView:
    @pytest.mark.django_db
    def test_get(self, client):
        organization = Organization.objects.create(
            name="beproud",
        )
        author = User.objects.create(
            username="theusername",
            organization=organization,
        )
        post = Post.objects.create(
            title="ブログ記事のタイトル",
            body="ブログ記事の本文",
            author=author,
            published_at="2018-11-05T00:00:00+0900",
        )

        res = client.get(f"/posts/{post.id}/")
```

(次のページに続く)

[*59] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
        assert res.context_data["title"] == "ブログ記事のタイトル"
        assert res.context_data["body"] == "ブログ記事の本文"
        assert res.context_data["author_name"] == "theusername"
```

この単体テストは悪くはありませんが、テスト対象に関係しない Organization のデータまで作成しています。User が Organization に依存しているので仕方なく用意していますが、検証したい項目には関係しないので省いたほうがより良いでしょう。

しかし、複数のテストで使い回す organization を作ることは推奨しません。詳しくは *26:* **テストケース毎にテストデータを用意する** (ページ 99) で説明します。

ベストプラクティス

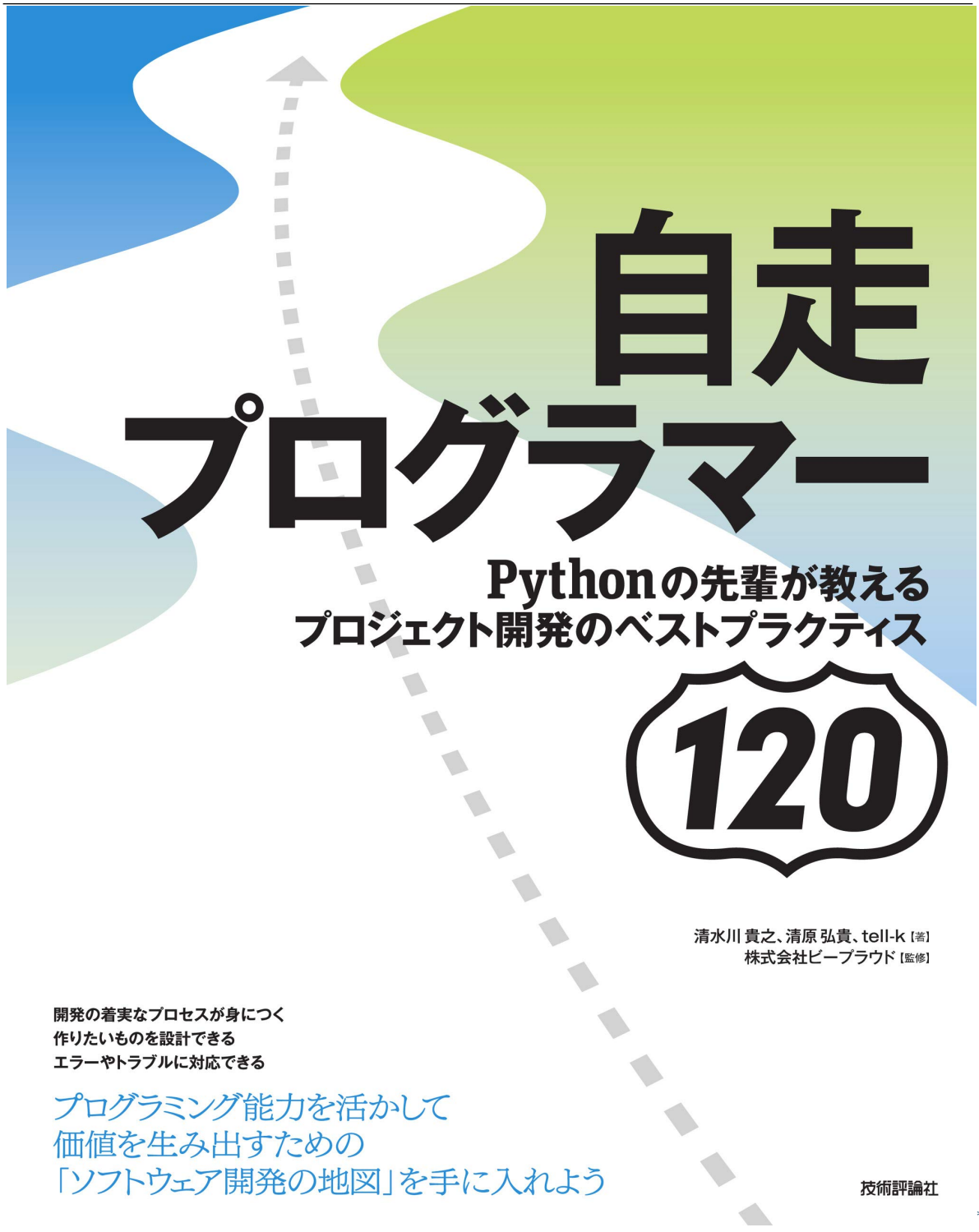
factory-boy[*60] を使いましょう。不要な フィクスチャー の作成が不要になります。

リスト 2.14　tests.py

```
import pytest

from .factories import OrganizationFactory, PostFactory, UserFactory


class TestPostDetailView:
    @pytest.mark.django_db
    def test_get(self, client):
        post = PostFactory(
            title="記事タイトル", body="記事本文",
            author__username="theusername"
        )

        res = client.get(f"/posts/{post.id}/")

        assert res.context_data["title"] == "ブログ記事のタイトル"
        assert res.context_data["body"] == "ブログ記事の本文"
        assert res.context_data["author_name"] == "theusername"
```

[*60] https://factoryboy.readthedocs.io/en/latest/

このテストからインポートしている factories.py は以下のようになります。



*61

*61 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*62] をご参照ください

2.4.8 26:テストケース毎にテストデータを用意する

テストコードを修正したら、関係のないところでテストが失敗してしまって困ったことはありませんか？テストデータを使い回すと、意図しないテストの失敗を招いてしまいます。

具体的な失敗

square_list という整数のリストの各要素を 2 乗してまたリストとして返す関数があるとします。

```
# spam.py ----

def square_list(nums):
    return [n * n for n in nums]

# 実行イメージ
# square_list([1, 2, 3]) => [1, 4, 9]
```

この関数に対して、下記のようなテストコードを書いたとします。

```
# 本来は別のテスト使うテストデータ生成関数を import
from spam.tests.other_fixtures import get_other_fixtures

class TestSquareList:

    def test_square(self):
        # Arrange --
        from spam import square_list

        test_list = get_other_fixtures() # => [1, 2, 3] がテストデータとして取得できる

        # Act --
        actual = square_list(test_list)
```

(次のページに続く)

[*62] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
        # Assert --
        expected = [1, 4, 9]
        assert actual == expected
```

square_list という関数をテストするために、たまたま他のテストで用意した整数のリストを返す関数 *get_other_fixtures* を利用しています。

この状態で、他の開発者が 別のテストを修正する目的 で `get_other_fixtures` 関数の戻り値を変更したらどうなるでしょうか？　もちろんこの `TestSqureList` のテストケースは失敗してしまいます。

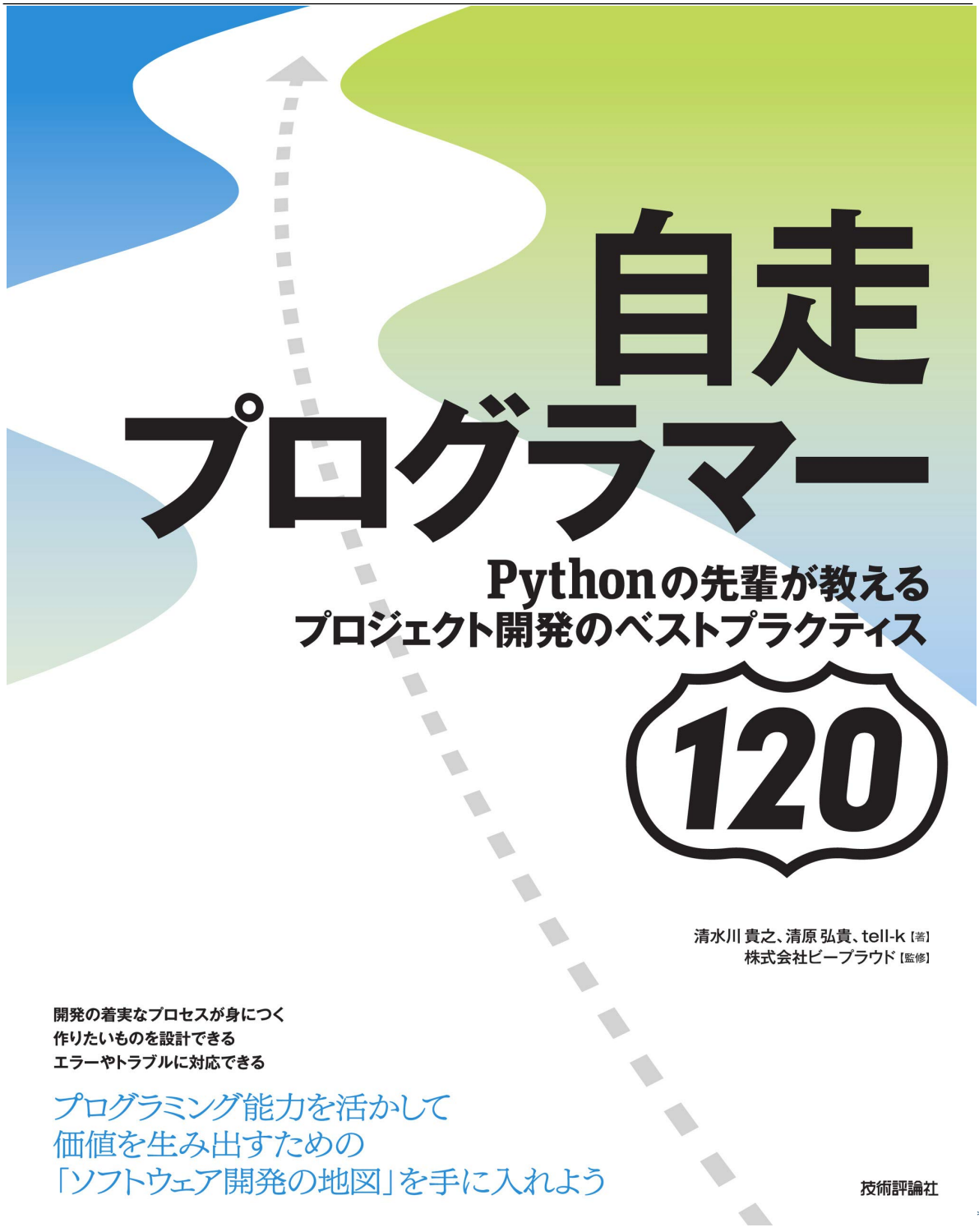
このテストを書いた本人ならば、すぐに原因もわかるかもしれませんが、他の開発者は別のテストを修正しているつもりなので、原因がわかるまでに時間が掛かってしまうでしょう。

ベストプラクティス

上記のようなトラブルを避けるためにも、 フィクスチャー を複数のテスト間でに使い回すのを極力避けましょう。理想的には個々のテストケースの中でのみ有効なフィクスチャーを用意して、他のテストには影響を与えないようにしましょう。

```
class TestSquareList:

    def test_square(self):
        # Arrange --
        from spam import square_list

        test_list = [1, 2, 3]  # 専用のテストデータを用意

        # Act --
        actual = square_list(test_list)

        # Assert --
        expected = [1, 4, 9]
        assert actual == expected
```

フィクスチャーを使い回さないという考えは、 factory-boy などのフィクスチャーを自動生成するライブラリを使うときにも適用できます。たとえばよくあるのが、Factory クラスのデフォルト値に依存したテストを書いてしまうケースです。



*63

*63 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*64] をご参照ください

2.4.9 27:必要十分なテストデータを用意する

ユニットテストを実行しているときに徐々に伸びていくテスト実行時間に不安を覚えたことはありませんか？ここではテスト実行時間を短くできる方法について説明します。

プログラミング迷子**:** 境界値テストのために **1** 万レコード必要なんです

- 後輩 W：テストのレビューお願いします。
- 先輩 T：ほいほい。なんかこのテスト実行するのに時間かかるね。
- 後輩 W：あー、あそこのテストで、テスト用のデータいっぱい生成してるからですかね？
- 先輩 T：ふむふむ。なるほど。こんなにデータ作らなくてもいいかもね。
- 後輩 W：どういうことです？　そこは 1 万件データがあると if 分岐して XX の処理を挟むんですが。
- 先輩 T：そうだね、本当にテストしたいのはその if 分岐が条件に従って実行されるかってことだよね？
- 後輩 W：はい。
- 先輩 T：その条件となる 1 万件はテストのときには任意の数に変更できるようにすれば、無駄にテストデータを生成せずにテストできるよね。
- 後輩 W：なるほどー。

具体的な失敗

Django の ORM から Spam モデルの件数をカウントして、件数が 10,000 件を超えるかどうかで結果が変わるようなコードがあるとします。このテストコードを書く場合、10,000 件の Spam モデルのデータを用意しないと、if の分岐をテストできません。テストを実行するたびに、毎回 10,000 件のデータを生成していては時間がかかりすぎます。

[*64] https://gihyo.jp/book/2020/978-4-297-11197-7

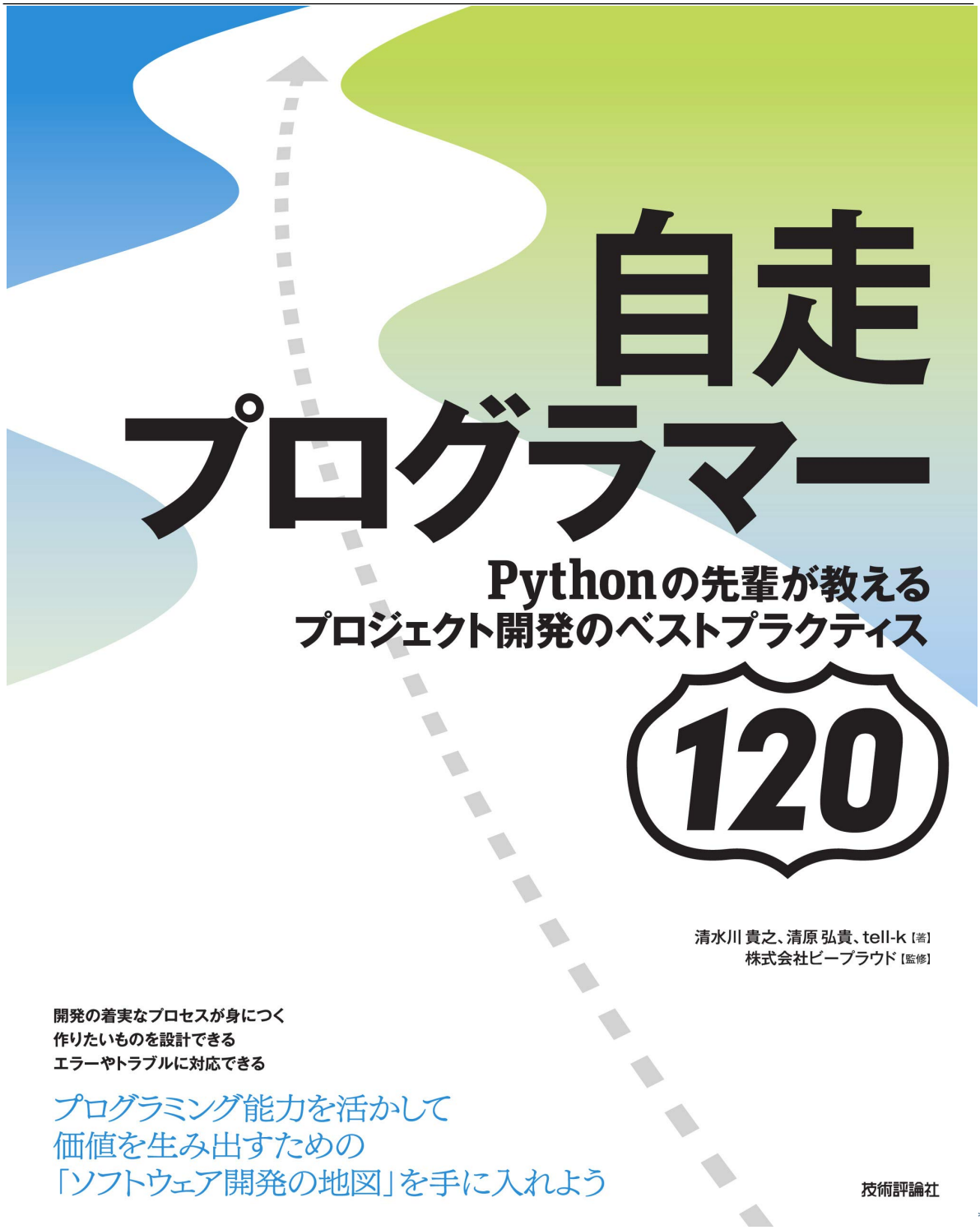
```
def is_enough_spam(piyo_id):
    if Spam.objects.filter(piyo_id=piyo_id).count() > 10000:
        return True
    else:
        return False
```

ベストプラクティス

上記コードでは、True と False を返すことがテストできれば良いので、テストをしやすいように、条件となる数値を引数として用意しましょう。テストのときに num_of_spam を 任意の数、たとえば 1 に変えてテストができます。

```
def is_enough_spam(piyo_id, num_of_spam=10000):
    if Spam.objects.filter(piyo_id=piyo_id).count() > num_of_spam:
        return True
    else:
        return False
```

引数で渡せないのであれば条件となる数字を定数化して、それをモックで置き換えるのでも良いでしょう。



*65

*65 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*66] をご参照ください

2.4.10 28:テストの実行順序に依存しないテストを書く

「なぜかテストが落ちるようになった」、そんなことはありませんか？各テストメソッドが他のテストメソッドに依存していると、実行の順序が変わったタイミングでテストが失敗するようになります。

テストの実行順序に依存したテストにはどういった問題があるのでしょうか？

具体的な失敗

リスト 2.15　tests.py

```
import pytest


class TestSum:
    def setup(self):
        self.data = [0, 1, 2, 3]

    def test_sum(self):
        self.data.append(4)
        actual = sum(self.data)
        assert actual == 10

    def test_negative(self):
        self.data.append(-5)
        actual = sum(self.data)
        assert actual == 5

    def test_type_error(self):
        self.data.append(None)
        with pytest.raises(TypeError):
            sum(self.data)
```

このテストは、各テストメソッドが他のメソッドに依存しています。self.data の中身を変更し続けてい

[*66] https://gihyo.jp/book/2020/978-4-297-11197-7

るので、テストが上から順番に実行されないと成功しません。

問題は、1 つのテストメソッドとして「正しさ」の保証ができない点です。1 つのテストとして、何をもって「正しさ」を保証しているかが曖昧になります。テストメソッドが独立していれば、そのテストメソッドだけで正しさを保証できます。他のテストに依存していると、テストを分離したり、移動したり、足したり、消したりするとテストが壊れてしまいます。

また、テストが依存し合っていると、単純に読みにくくなる場合が多いでしょう。単一のテストメソッド以外のデータや処理も見る必要があるからです。

ベストプラクティス

まずデータを使い回さないようにしましょう。

```
import pytest


class TestSum:
    def test_sum(self):
        assert sum([0, 1, 2, 3, 4]) == 10

    def test_negative(self):
        assert sum([0, 1, 2, 3, 4, -5]) == 5

    def test_type_error(self):
        with pytest.raises(TypeError):
            sum([1, None])
```

見た目の記述量が多く、冗長になっている印象を受けるかもしれません。ですが単体テストは少し冗長なくらいが良いです。単体テストで確認する内容に関係しないコードはなくすべきですが、メソッド間で共通のデータを持つのはやめましょう。

関連

- *23:*テストから外部環境への依存を排除しよう (ページ 87)
- *24:*テスト用のデータはテスト後に削除しよう (ページ 92)
- *26:*テストケース毎にテストデータを用意する (ページ 99)

2.4.11 29:戻り値がリストの関数のテストで要素数をテストする

単体テストで結果の確認するとき、よく陥る罠があります。リスト（正確には Iterable）のテストをするときに、要素数を確認しないことです。

具体的な失敗

テスト対象として、以下の関数を考えます。

リスト 2.16　items.py

```
def load_items():
    return [{"id": 1, "name": "Coffee"}, {"id": 2, "name": "Cake"}]
```

この `load_items` の動作確認をするとき、以下のように書いてしまっていませんか？

リスト 2.17　tests.py

```
class TestLoadItems:
    def test_load(self):
        actual = load_items()

        assert actual[0] == {"id": 1, "name": "Coffee"}
        assert actual[1] == {"id": 2, "name": "Cake"}
```

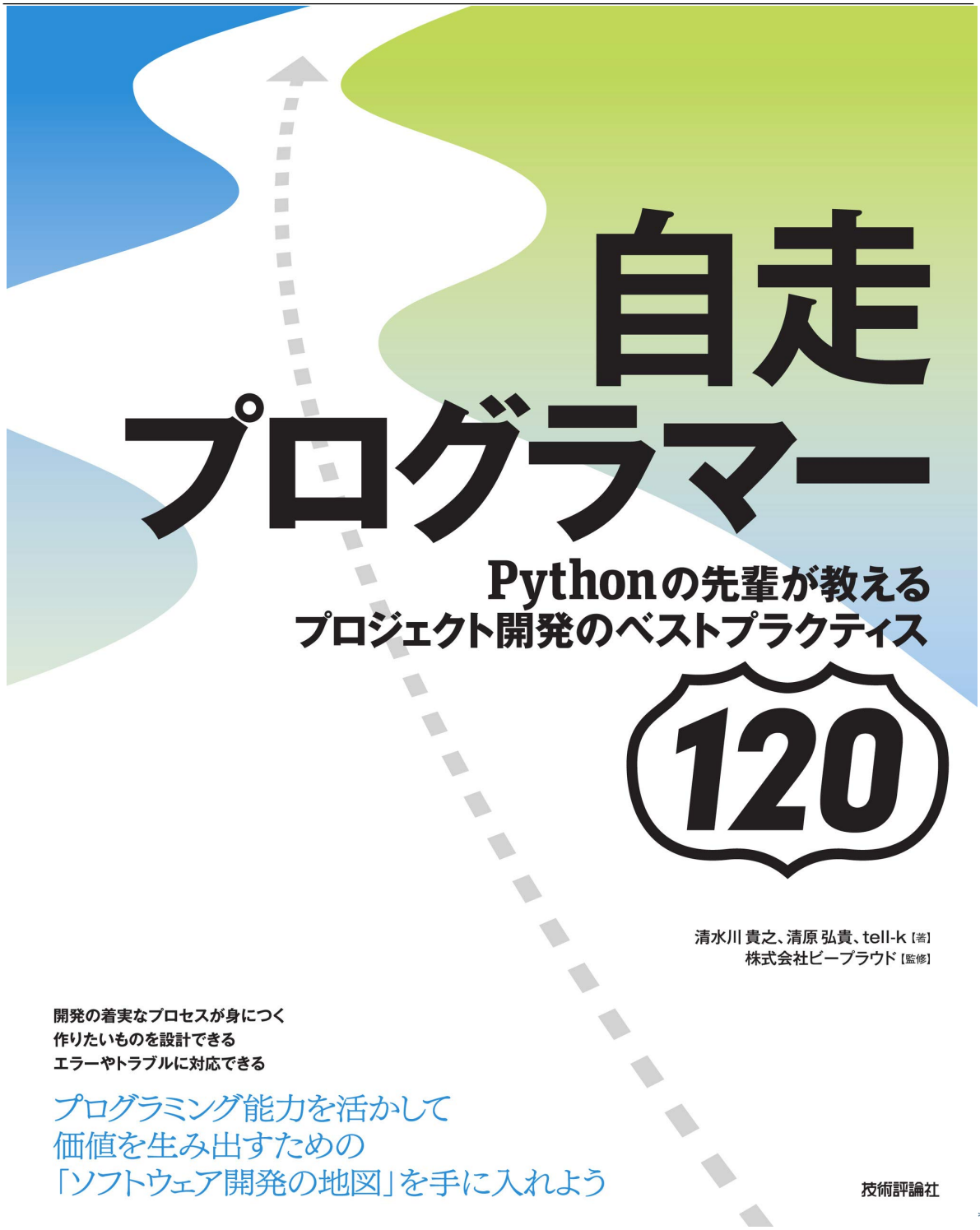
要素数を確認しないと、リストに 3 つ目の値がある可能性があるのが問題です。予期しないデータが追加で返されていてもバグに気づけません。たとえば load_items のバグで、常にリストの最後に空の辞書が入ってしまうなどが考えられます。

ベストプラクティス

リスト actual の長さを必ず確認しましょう。

リスト 2.18　tests.py

```
class TestLoadItems:
    def test_load(self):
        actual = load_items()

        assert len(actual) == 2
        assert actual[0] == {"id": 1, "name": "Coffee"}
        assert actual[1] == {"id": 2, "name": "Cake"}
```



*67

*67 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*68] をご参照ください

2.4.12 30:テストで確認する内容に関係するデータのみ作成する

テストが無駄に長くなる原因として、無駄にデータを作成しすぎることがあります。その失敗と、ちょうど良い解法を見ていきましょう。

具体的な失敗

Django のモデルと、モデル用のファクトリー、そしてモデルを絞り込んで取得する関数を考えます。

リスト 2.19　factories.py

```
import factory

from . import models


class BlogFactory(factory.django.DjangoModelFactory):
    name = "ブログ名"

    class Meta:
        model = models.Blog


class PostFactory(factory.django.DjangoModelFactory):
    blog = factory.SubFactory(BlogFactory)
    title = "タイトル"
    body = "本文"

    class Meta:
        model = models.Post
```

テスト対象になるのは、以下の Post モデルを絞り込む関数です。

[*68] https://gihyo.jp/book/2020/978-4-297-11197-7

リスト 2.20　posts.py

```
from .models import Post


def search_posts(text):
    if ':' in text:
        blog_name, post_text = text.split(':', 1)
        return Post.objects.filter(
            blog__name__contains=blog_name,
            title__contains=post_text,
            body__contains=post_text,
        )
    else:
        return Post.objects.filter(
            title__contains=text,
            body__contains=text,
        )
```

まず、単体テストで過剰に値を指定している例を見てみましょう。

リスト 2.21　tests.py

```
from .factories import BlogFactory, PostFactory
from .posts import search_posts


class TestSearchPosts:
    def test_search_post(self):
        """ 検索条件から記事を検索する（ブログ名の指定はしない）
        """
        blog = BlogFactory(name="ブログ名")
        post1 = PostFactory(blog=blog, title="八宝菜の作り方", body="しいたけが美味しい
↪")
        PostFactory(blog=blog, title="プラモデルのイロハ", body="合わせ目消しの極意その
1...")

        actual = search_posts("しいたけ")
```

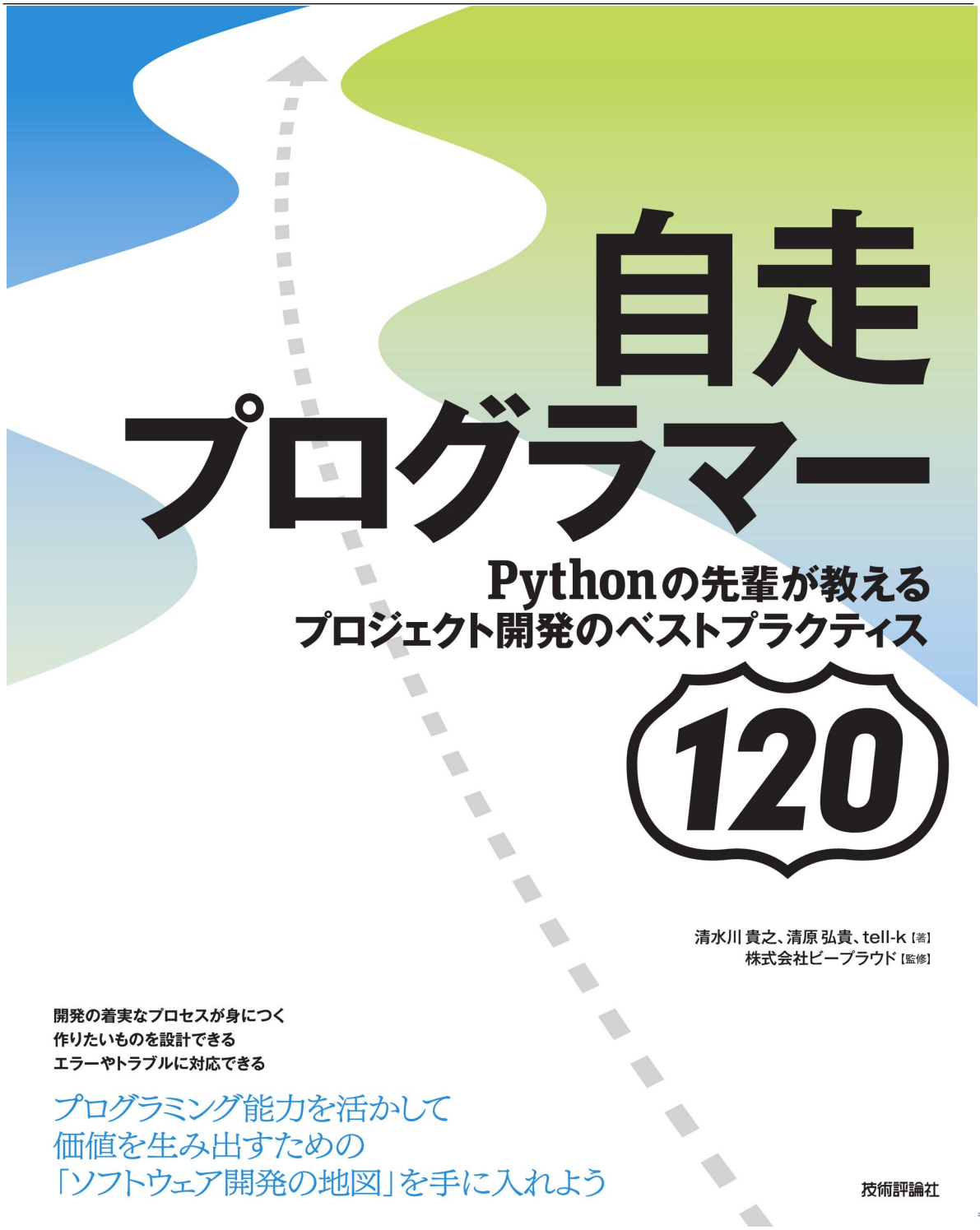
(次のページに続く)

(前のページからの続き)

```
        assert len(actual) == 1
        assert actual[0] == post1
```

このテストメソッドでは「しいたけ」という文字列で Post を検索しています。検索対象になる Post を作るのであれば `title` か `body` のどちらかに「しいたけ」という文字列が含まれていれば十分です。ここでは `body` に「しいたけ」が含まれているので、十分検索の対象になっています。ですが `post1` には不要な `title` が指定されています。また、このメソッドではブログ名での検索はしていないので、 `blog` の指定も不要です。

次は、デフォルト値に頼り過ぎている例を紹介します。



*69

*69 https://gihyo.jp/book/2020/978-4-297-11197-7

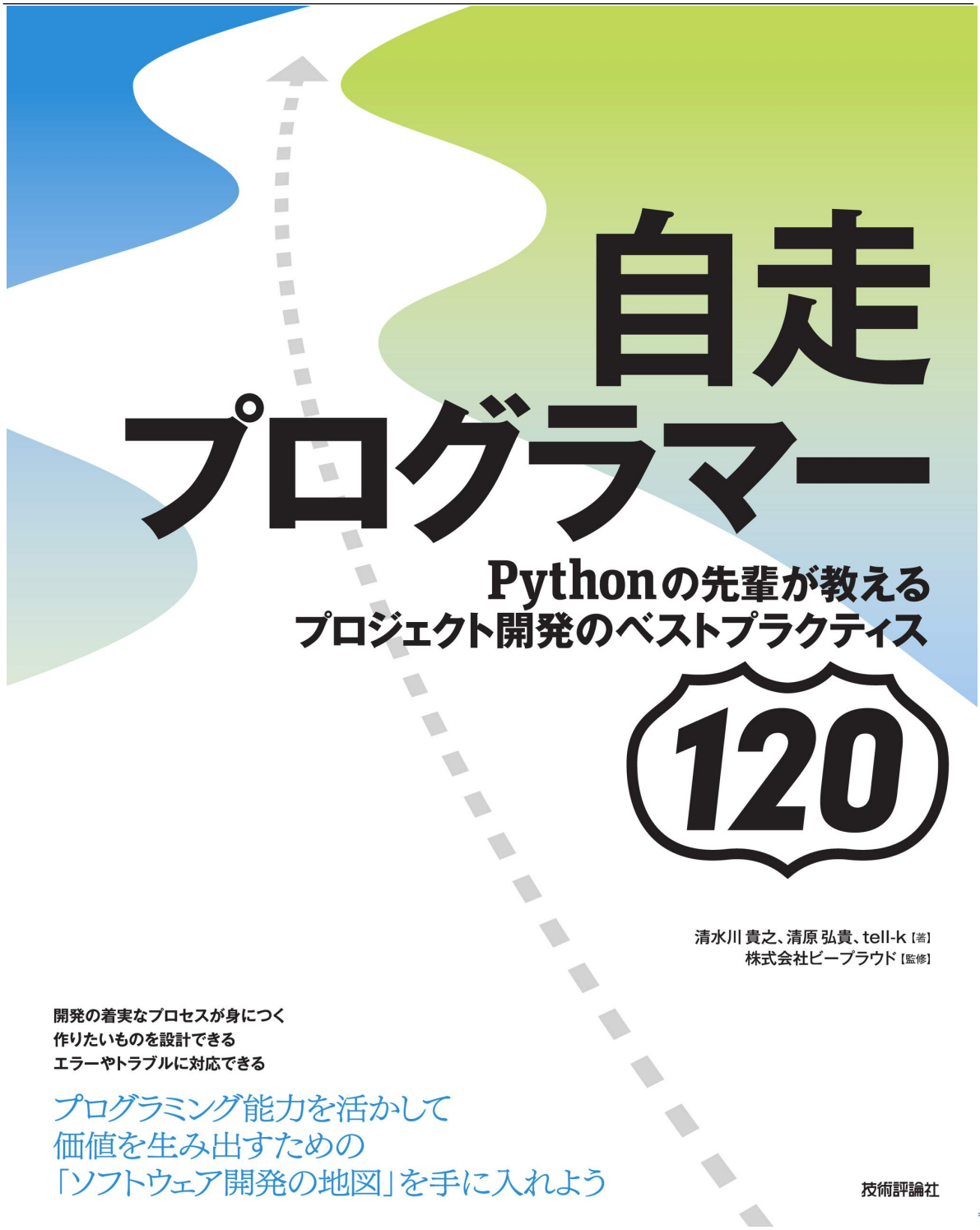
（中略）詳細は書籍 自走プログラマー[*70] をご参照ください

ベストプラクティス

以下のポイントを守りましょう。

- テストで確認する内容に関係するデータのみ作成する
- テストに関係しないデータ、パラメーターを作らない、指定しない
- テストに関係するデータ、パラメーターを作る、指定する（デフォルトに依存しない）

[*70] https://gihyo.jp/book/2020/978-4-297-11197-7



*71

*71 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*72] をご参照ください

関連

- *26:*テストケース毎にテストデータを用意する (ページ 99)

2.4.13 31:過剰な mock を避ける

mock[*73] は便利なライブラリですが、使いすぎには要注意です。

mock でよくある失敗から、ベストプラクティスを学びましょう。

具体的な失敗

Django の View 関数をテスト対象として考えます。

```
from .forms import PostSearchForm
from .posts import search_posts


def post_list(request):
    if request.GET:
        form = PostSearchForm(request.GET)
        if form.is_valid():
            text = form.cleaned_data['search']
            posts = search_posts(text)
    else:
        form = PostSearchForm()
        posts = Post.objects.all()
    return TemplateResponse(request, 'post_list.html',
                            {'posts': posts, 'form': form})
```

この View 関数のテストとして mock を使いすぎると、次のようになります。

[*72] https://gihyo.jp/book/2020/978-4-297-11197-7
[*73] https://docs.python.org/ja/3/library/unittest.mock.html

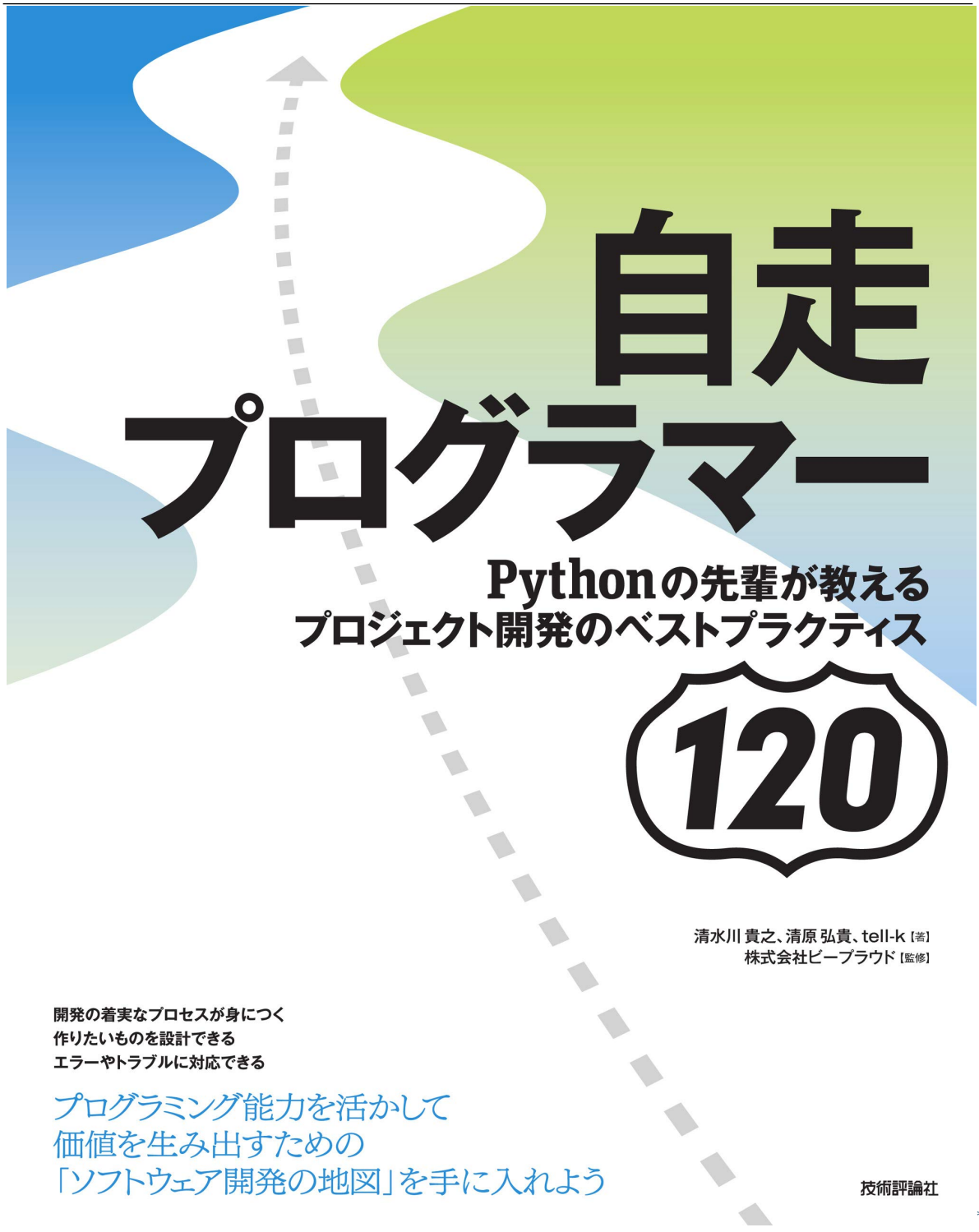
```
from unittest import mock
from django.test import TestCase

class TestPostList(TestCase):
    @mock.patch('posts.search_posts',
                return_value=[{'title', 'タイトル', 'body': '本文'}])
    @mock.patch('forms.PostSearchForm')
    def test_search(self, m_search, m_form):
        with mock.patch.object(m_form, 'is_valid', return_value=True):
            res = self.client.get('/posts', data={'search': '本文'})

        assert '本文' in res.content.decode()
```

この例では View 関数の動作のみをテストするために mock を乱用しています。しかし、このテストから確認できることは、ほぼありません。search=**本文** のように指定されたクエリーパラメーターが正しくフォームで解釈されて、検索に使われて、テンプレートに描画されるつながりを確認できないからです。

ベストプラクティス

mock を使いすぎるよりも、単純にデータを作成して動作確認をするほうが良いでしょう。

```
class TestPostList(TestCase):
    def test_search(self):
        PostFactory(title='タイトル')
        PostFactory(title='テスト')

        res = self.client.get('/posts', data={'search': 'タイトル'})

        assert "タイトル" in res.content.decode()
        assert "テスト" not in res.content.decode()
```



*74

*74 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*75] をご参照ください

関連

- *22:*単体テストをする観点から実装の設計を洗練させる (ページ 80)

2.4.14 32:カバレッジだけでなく重要な処理は条件網羅をする

具体的な失敗

テスト対象として以下の関数を考えます。この処理はユーザーを認証する重要なプログラムです。

```
def find_auth_user(username=None, password=None, team_name=None):
    try:
        users = User.objects.select_related('team').filter(
            (Q(username=username) | Q(email__iexact=username)),
        )
        if team_name:
            # チームユーザー
            users = users.filter(team__name=team_name)
        else:
            # 個人ユーザー
            users = users.filter(team_id=Team.PERSONAL_USERS_ID)
        return users.get()
    except User.DoesNotExist:
        return None
```

この関数のテストとして分岐網羅をすると、以下 3 つのメソッドが必要です。

```
class TestFindAuthUser:
    def test_team(self):
        """ チームユーザーの場合 """

    def test_personal(self):
        """ 個人ユーザーの場合 """
```

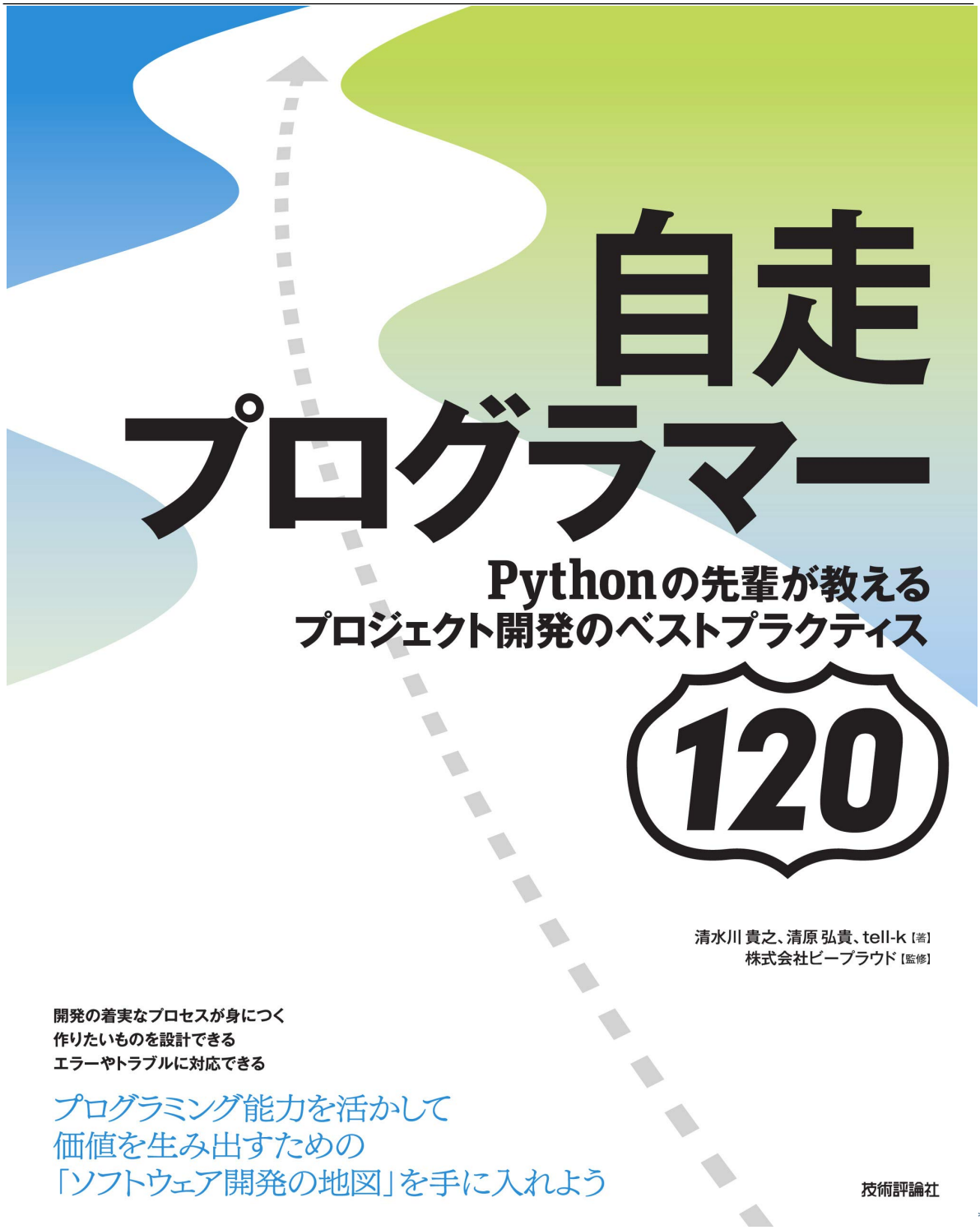
(次のページに続く)

[*75] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
    def test_not_exist(self):
        """ ユーザーが存在しない場合 """
```

この 3 つだとユーザーを取得する細かい処理までは確認できません。たとえば find_auth_user は username と email の両方でユーザーを指定できますが、分岐網羅では片方だけしか網羅できません。



*76

*76 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*77] をご参照ください

ベストプラクティス

上記のテストメソッドに、2 つ追加しましょう。

```
class TestFindAuthUser:
    ...  # 迷子コードのテストにさらに追加して

    def test_email(self):
        """ メールアドレス指定で取得 """

    def test_email_case_insensitive(self):
        """ メールアドレスでは大文字小文字を区別しない """
```

今回の場合は username で認証する場合と、 email で認証する場合それぞれのテストを書きましょう。また email には「大文字小文字を区別しない確認」も書きます。

次のような処理も条件網羅すべきです。

- 支払い
- 認証
- 引当て
- データの変更、削除（変更や削除は後戻りできない作業な場合が多いので重要）

[*77] https://gihyo.jp/book/2020/978-4-297-11197-7

2.5. 実装の進め方

2.5.1 33:公式ドキュメントを読もう

プログラミング迷子**:** 公式ドキュメントは難しいので正解を **Web** で検索しました

- 先輩 T：この Django ModelForm のコード、なんかすごい不思議な書き方になってるんだけど、どうしてこうなったw
- 後輩 W：ModelForm の使い方がよくわからなくて、いろいろ調べて書きました。
- 先輩 T：うーん、それはどうやって調べたの？
- 後輩 W：いろいろググって調べたんですが、良い感じの情報がなくて……。
- 先輩 T：そっかー。で、この書き方はどこに書いてあったやつ？
- 後輩 W：すいません、ちょっと覚えてません。
- 先輩 T：公式ドキュメントは読んだ？
- 後輩 W：ちょっとは読んだんですが、よくわからなくていろいろググってました。
- 先輩 T：公式ドキュメントのここに、そのまんまの使い方が書いてあるで。
- 後輩 W：あれ……ほんとだ……。

よく使われている言語やライブラリのドキュメントには、いろいろなことが書かれています。利用者が多ければ多いほど、利用者それぞれの疑問に応えるドキュメントが用意されています。ただし、利用者すべての個別の使い方向けのサンプルを用意するのではなく、抽象度の高い 原理や概念 を説明するドキュメントが用意されることもよくあります。

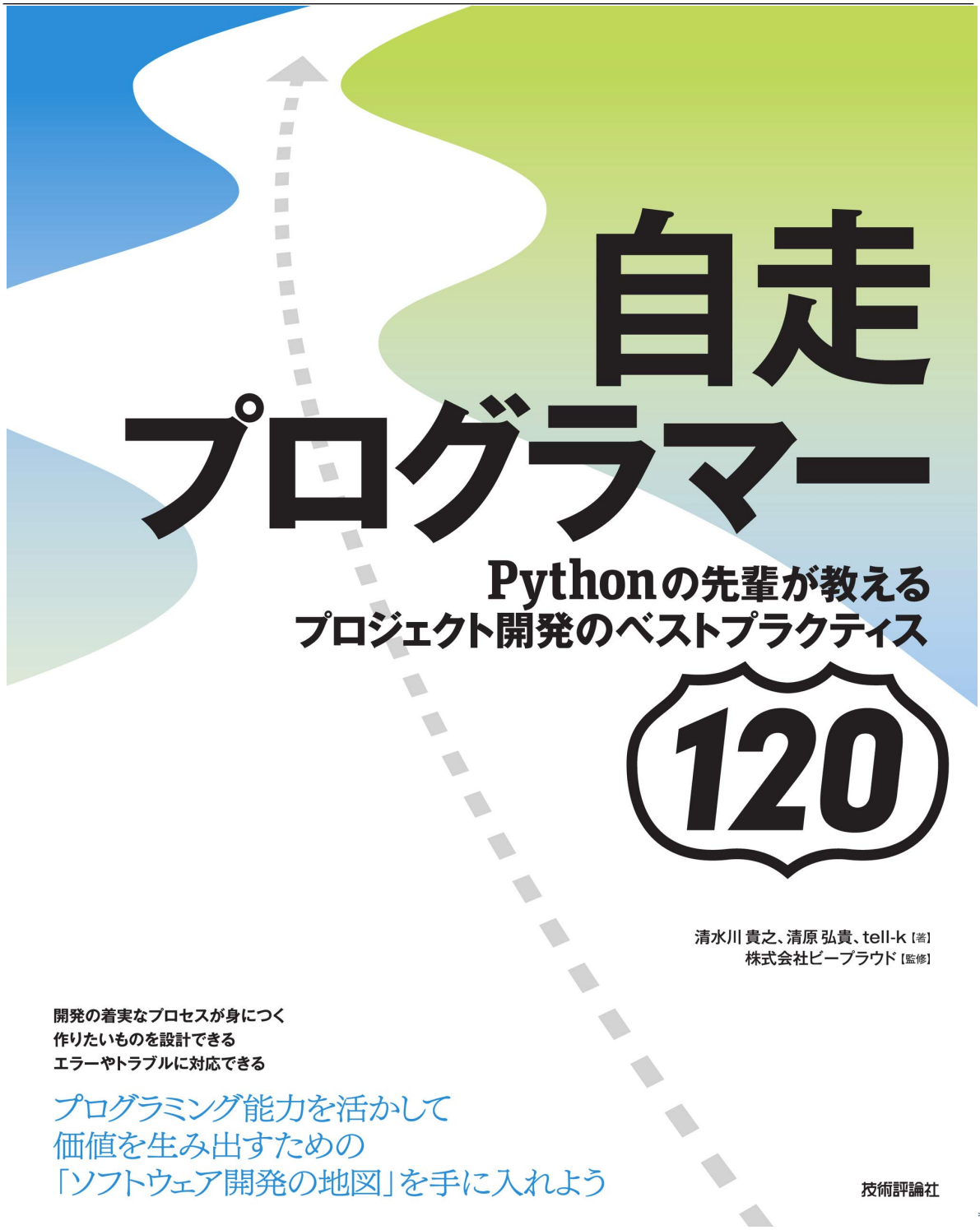
しかし、「迷子」はこういった抽象度の高いドキュメントを読み解くよりも、自分と全く同じケースの課題を解決した「具体的な正解」を検索して見つけようとしてしまいます。そのため、誰かの blog 等で近いケースを見つけると、 原理や概念 を理解しないままコードを使おうとして、うまく動作しなかったり、解釈が難しい複雑なコードを書いてしまいます。あとから振り返って、公式ドキュメントを読み解くことがゴールへの最短ルートだった、ということがよくあります。

ベストプラクティス

公式ドキュメント（原典）を読みましょう。このとき、時間を制限して取り組むのが大事です。

調べるための キーワード がわかっていれば、そのキーワードで検索することで公式ドキュメントの読むべき箇所を見つけられます。キーワードがわからない、見つからないときは、 用語集 を探す、といったアプローチが有効です。仕事のプロジェクトでは用語集をまとめることで曖昧になりがちな言葉の定義を明確にします。用語集になければ、検索を活用して、徐々にキーワードに近づいていく、という方法が使えます。[*78]

[*78] エンジニアの「プロの所作」01. まず自分で調べる ：「自分主体で考えて作る」第 1 歩。わからないことを調べる所作を伝えます - Python 学習チャンネル by PyQ: https://blog.pyq.jp/entry/professionalism_of_engineer_01



*79

*79 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*80] をご参照ください

2.5.2 34:一度に実装する範囲を小さくしよう

プログラミング迷子**:** 一気に遅れを取り戻すためにタスク分割を省略？

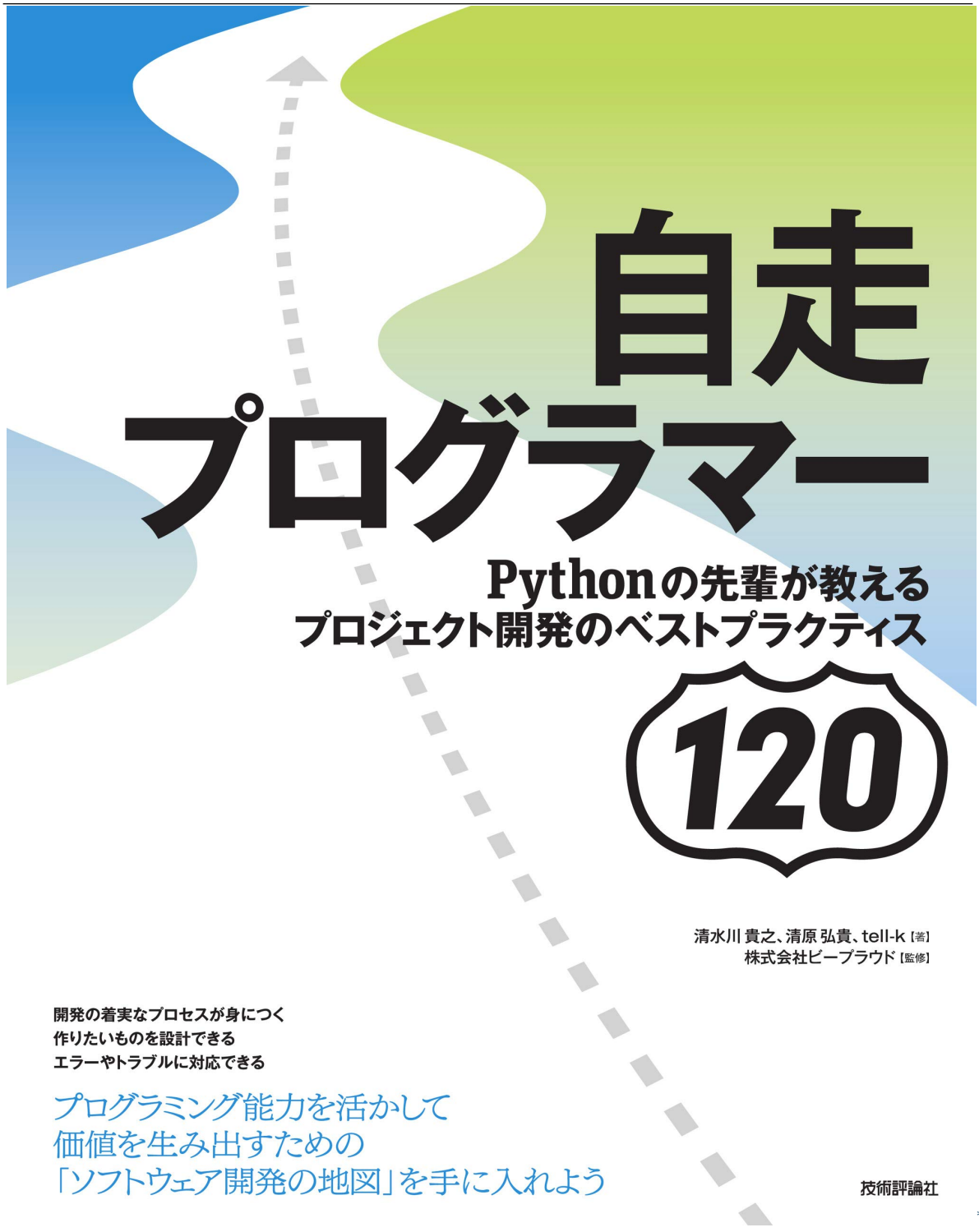
- 先輩 T：SNS 連携の実装って今どうなってる？　なかなかレビュー依頼が来ないけど何かハマってる？
- 後輩 W：SNS への通知機能、予定より遅れていますが、もう 1 週間くらいかかりそうです。
- 先輩 T：え、どうしたの、けっこうかかってるよね。
- 後輩 W：SNS の認証が必要で、そのライブラリの使い方に手間取りました。コメント欄に SNS に通知するための機能がまだ途中で、ここにも SNS のメンションの自動補完が必要だし、あと、連携解除する機能が必要なこともわかったのでそれも……。
- 先輩 T：ちょっとまって！　それだとやることが増えていって、いつまでも終わらなそう。だから、全部別々に分けてレビュー依頼しましょう。 機能の粒度 が大きくて、作るのもいろいろ考えることが多くて大変だよね。
- 後輩 W：はい、実はすごい大変で……。
- 先輩 T： 大きいチケットは分割しよう 、っていうのはこのあいだ『Python プロフェッショナルプログラミング』[*81] 5 章の「チケットを分割しよう」を読んで納得してたと思うんだけど、今回はどうして分割できなかったの？
- 後輩 W：粒度が大きすぎるのは認識していたんですが、もう遅れているので 分割に時間かけてる場合じゃない と思って……。
- 先輩 T：なるほど、一気に実装して遅れを取り戻そうとしたのか。

*35:**基本的な機能だけ実装してレビューしよう*** (ページ 132) に続く

SNS 連携のような機能を実装する場合、見た目は SNS に投稿するだけの簡単なものでも、内部では OAuth 等による認証が必要だったり、トークンをデータベースに保存しておくといった 多くの前準備 が必要にな

[*80] https://gihyo.jp/book/2020/978-4-297-11197-7
[*81] 『Python プロフェッショナルプログラミング第 3 版』（ビープラウド著、秀和システム刊、2018 年 6 月）

ります。こういったコードを書いたことがないと、実装にどのくらいの時間が必要なのかの 見積もり ができません。



[*82]

[*82] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*83] をご参照ください

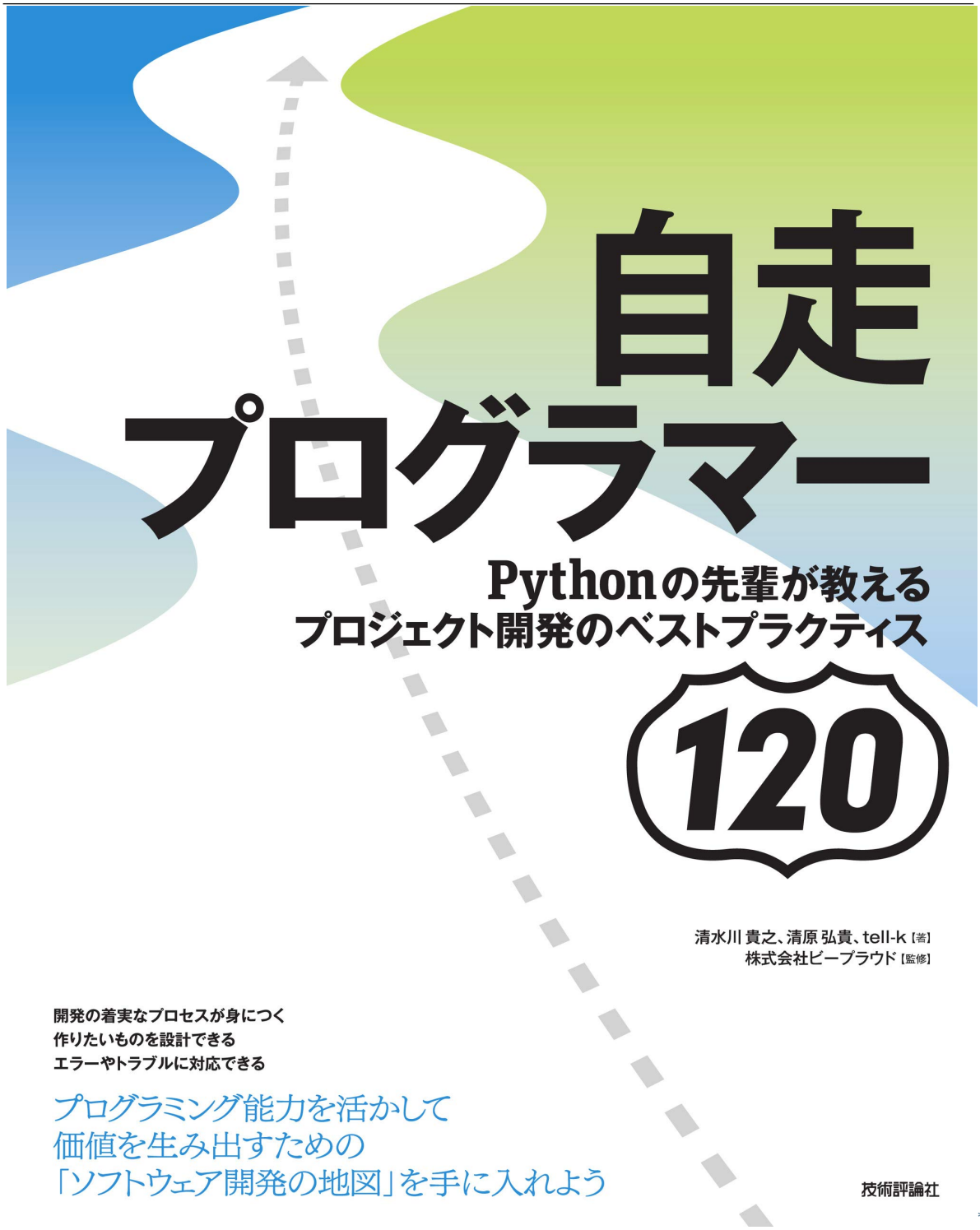
ベストプラクティス

一度に実装する範囲を小さくしましょう。

「SNS 連携機能」のような一言で済む機能であっても、実装する内容は多岐にわたります。あるいはちょっとした機能だと思っていたものでも「実装し始めると芋づる式にやることが増えていく」というのはよくあるハマリパターンです。あれもこれも、と手を広げる前に タスクばらし をしましょう[*84] 。

[*83] https://gihyo.jp/book/2020/978-4-297-11197-7

[*84] 『管理ゼロで成果はあがる〜「見直す・なくす・やめる」で組織を変えよう』（倉貫義人著、技術評論社刊、2019 年 1 月）または著者のブログ記事 https://kuranuki.sonicgarden.jp/2016/07/task-break.html



[*85]

[*85] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*86] をご参照ください

関連

- *39:*開発アーキテクチャドキュメント (ページ 144)

2.5.3 35:基本的な機能だけ実装してレビューしよう

プログラミング迷子**:** 目に見える機能でタスクを分割

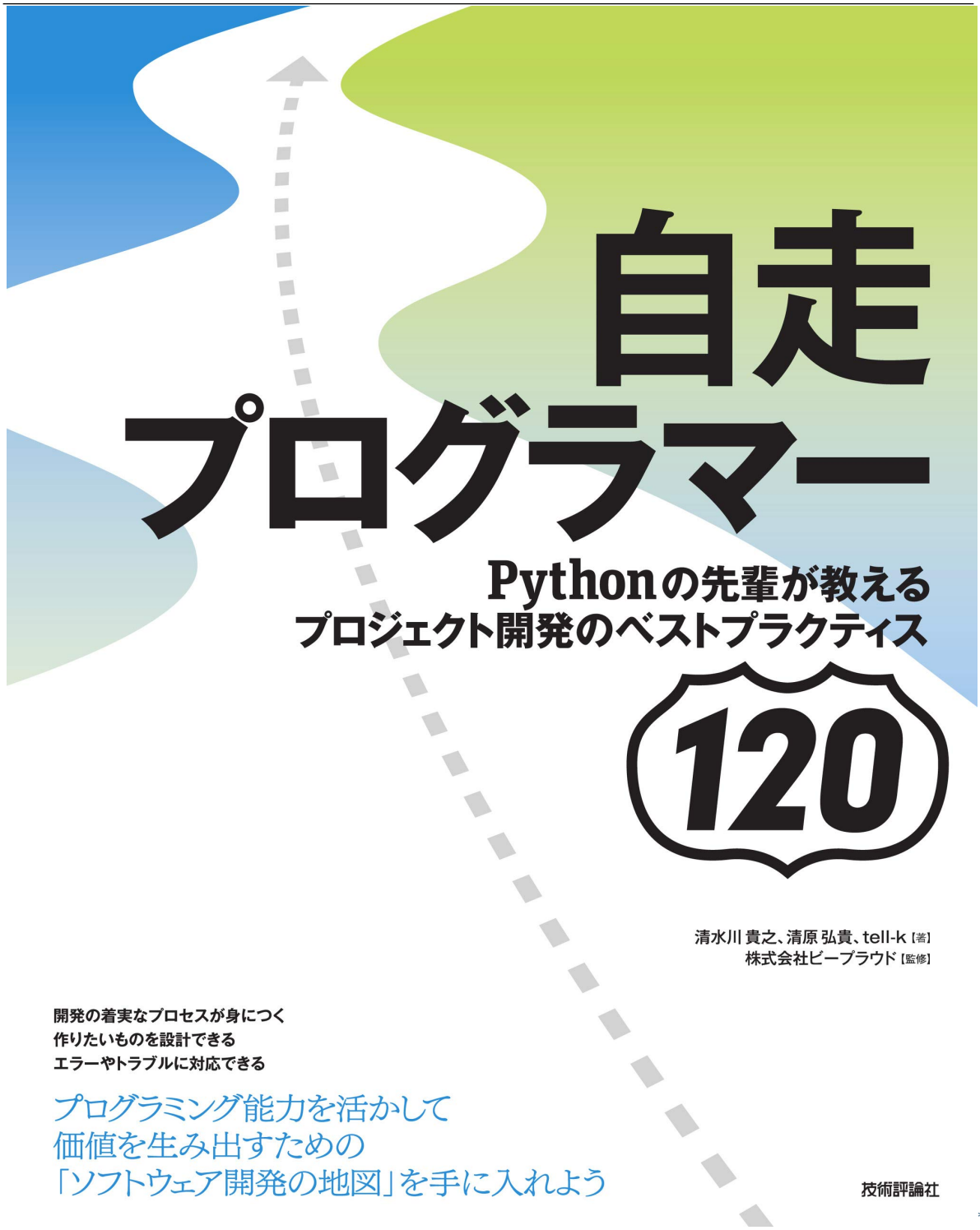
- 先輩 T：一度に実装する範囲を小さくするにはどう分けるといいと思う？
- 後輩 W：え、はい。ええっと、じゃあ「認証と解除」「メンションの記法と自動補完」「SNS 投稿」……。
- 先輩 T：認証と解除も分けましょう。記法と補完も分けようか。
- 後輩 W：そこもですか？
- 先輩 T：はい。 機能の粒度 が大きいと、レビューも大変だから分けよう。認証まわりは、「目に見える機能は何もないけど、内部では SNS 連携が通信レベルで動作するようになったよ」という段階でレビューに出そう。

*36:*実装方針を相談しよう (ページ 135) に続く

[*86] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

ごく基本的な機能だけを実装して、その段階で レビュー してもらいましょう。



*87

*87 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*88] をご参照ください

2.5.4 36:実装方針を相談しよう

いざコードを書こうと思ったときに、どのように実装したら良いか 1 人で迷ったことはありませんか？または、言われたとおりに実装にしたつもりでも、コードレビューに出したら、実装方法が間違っていたなんて経験はありませんか？

プログラミング迷子**:** 多分これで伝わるでしょ **vs** 多分こういう意味かな

- 先輩 T：これなんでこんな修正になったの？
- 後輩 W：ちょっと悩みましたが、レビューで指摘されたのでそう直しました。
- 先輩 T：けど修正されてる内容は私の期待とだいぶ違うよ？
- 後輩 W：えー？
- 先輩 T：えー？

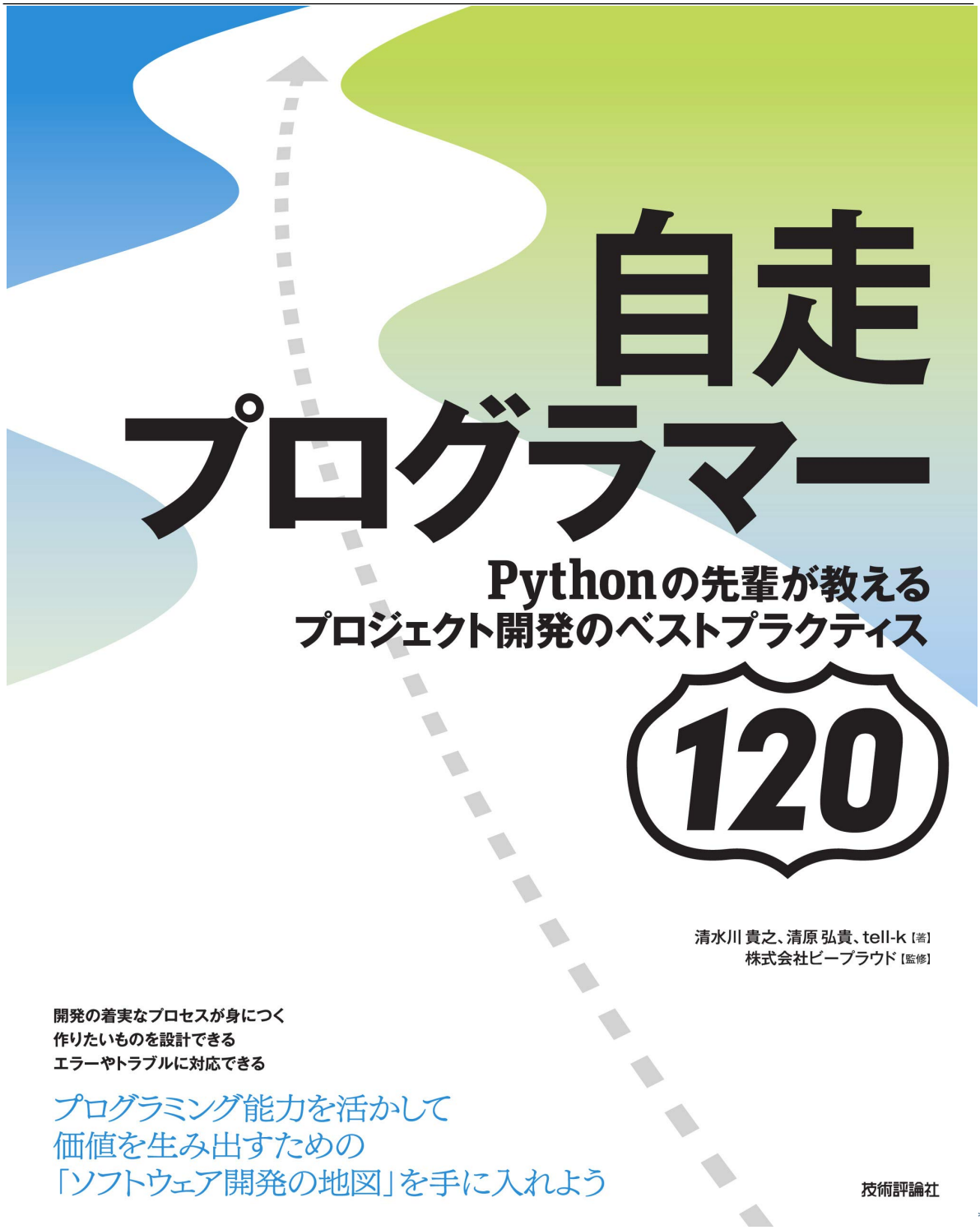
（T も W も迷子）

ベストプラクティス

仕様や設計がどれだけしっかり書かれていても、どんなコードを実装するかは開発者によって異なります。そして、些細な認識違いで仕様や設計意図とは全く異なる実装をしてまうこともあります。そういった場合、コードレビューのタイミングになって、間違いに気づき作業が無駄になってしまうなどのトラブルがあります。

そのような悲しい状況を避けるためにも、事前にお互いの認識を合わせることが重要です。認識を合わせるというのは、一方的に相手に伝達するのではなく、1 つひとつお互いがわかっているかどうか確認し合うことです。

[*88] https://gihyo.jp/book/2020/978-4-297-11197-7



[*89]

[*89] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*90] をご参照ください

関連

- *34:*一度に実装する範囲を小さくしよう (ページ 127)

2.5.5 37:実装予定箇所にコメントを入れた時点でレビューしよう

プログラミング迷子**:** 自分の言葉で設計を説明しよう

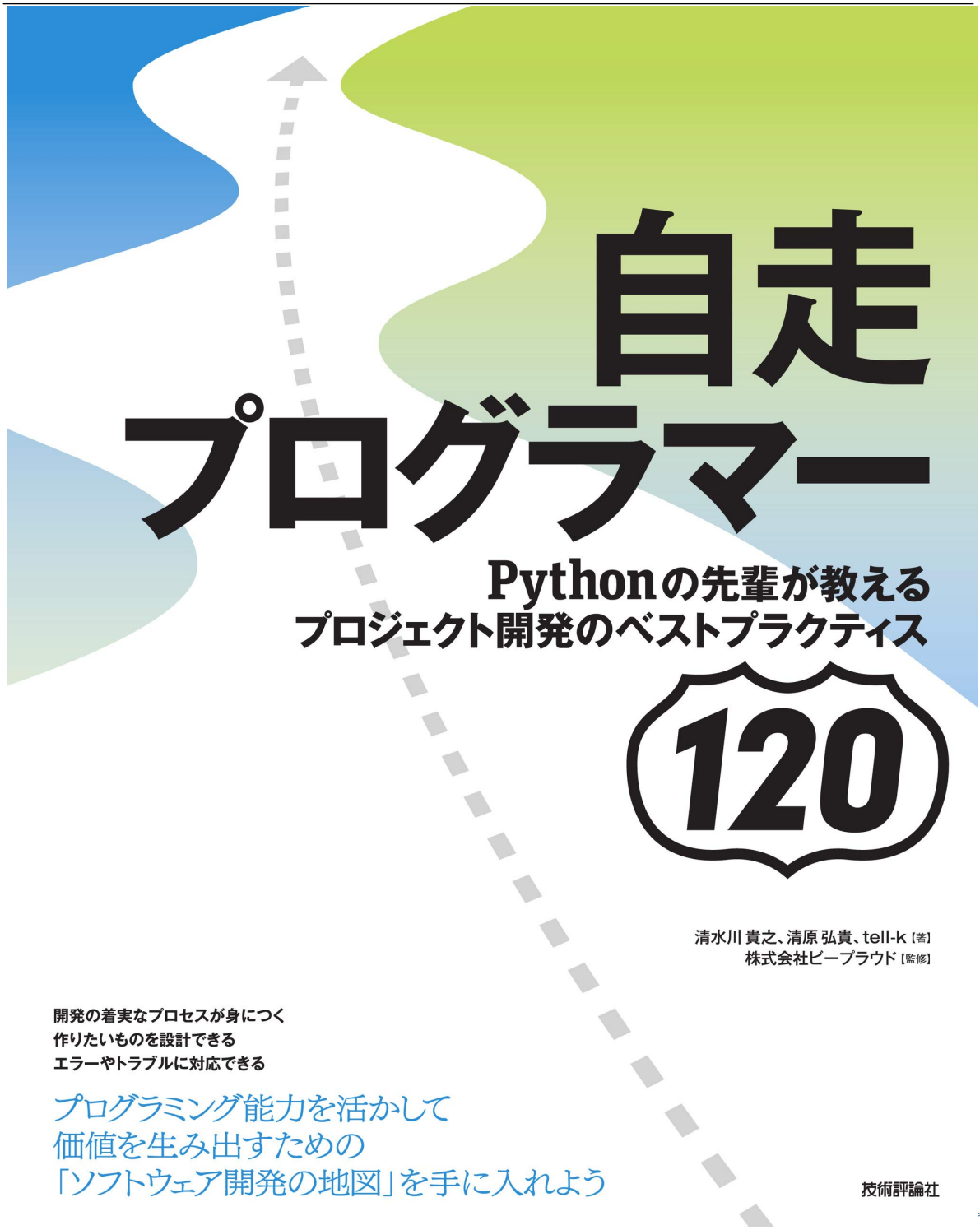
- 後輩 W：レビューで指摘されたことを今読み返すと、ちゃんとわかってないまま実装進めちゃってました。次から、気をつけます。
- 先輩 T：それは今だからそう言えるけど、最初読んだときはわかったと思ったんだよね？　だとししたら気をつけようがない気がするよ。
- 後輩 W：設計の意図を実装前にちゃんと把握するには、どうすればいいですかね？
- 先輩 T：コードを書き始める前に、設計を元に実装予定箇所にコメントでやることを書いていこう。で、それを先にレビューしましょう。

あとからなら間違っていたことに気づけても、それを事前に自分で気づくのは難しいものです。また、自分が理解したことを文章で書いて依頼者に確認してもらうとしても、もともとの要件が文章になっている場合、それを改めて書き直してもあまり効果がありません。自分で気づけないことであれば、先に自分の理解を実装コード上で表現してみましょう。

ベストプラクティス

ソースコードの実装予定箇所に TODO コメントを書きましょう。Pull Request（PR）でレビューしているチームであれば、その時点で PR を作ってレビューしてもらいましょう。コメントには、実装の根拠になるチケット番号やドキュメントの URL を書きましょう。

[*90] https://gihyo.jp/book/2020/978-4-297-11197-7



*91

*91 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*92] をご参照ください

関連

- *40:PR* **の差分にレビューアー向け説明を書こう** (ページ 147)

2.5.6 38:必要十分なコードにする

機能を開発中についつい気分が乗って余計な実装まで盛り込んでしまった経験はありませんか？シンプルに必要十分なコードを書くことがなぜ大切なのか考えてみましょう。

具体的な失敗

あるユーザー情報のつまった辞書のリスト中から 特定の性別のデータだけを抜き出だすような関数を実装するという開発タスクをアサインされたとします。

```
# ユーザー情報のデータ
data_list = [
    {'name': 'shimizukawa',  'gender': 'male',  'age': 40 },
    {'name': 'spam',  'gender': 'female',  'age': 10 },
    {'name': 'ham',  'gender': 'none',  'age': 20 },
    {'name': 'egg',  'gender': 'male',  'age': 70 },
]
```

ここからデータの抽出ロジックを書いていくうちに、将来的にもっといろんなパターンが必要になるんじゃないかと考え、いろんなパターンで検索できる関数を実装してしまいました。

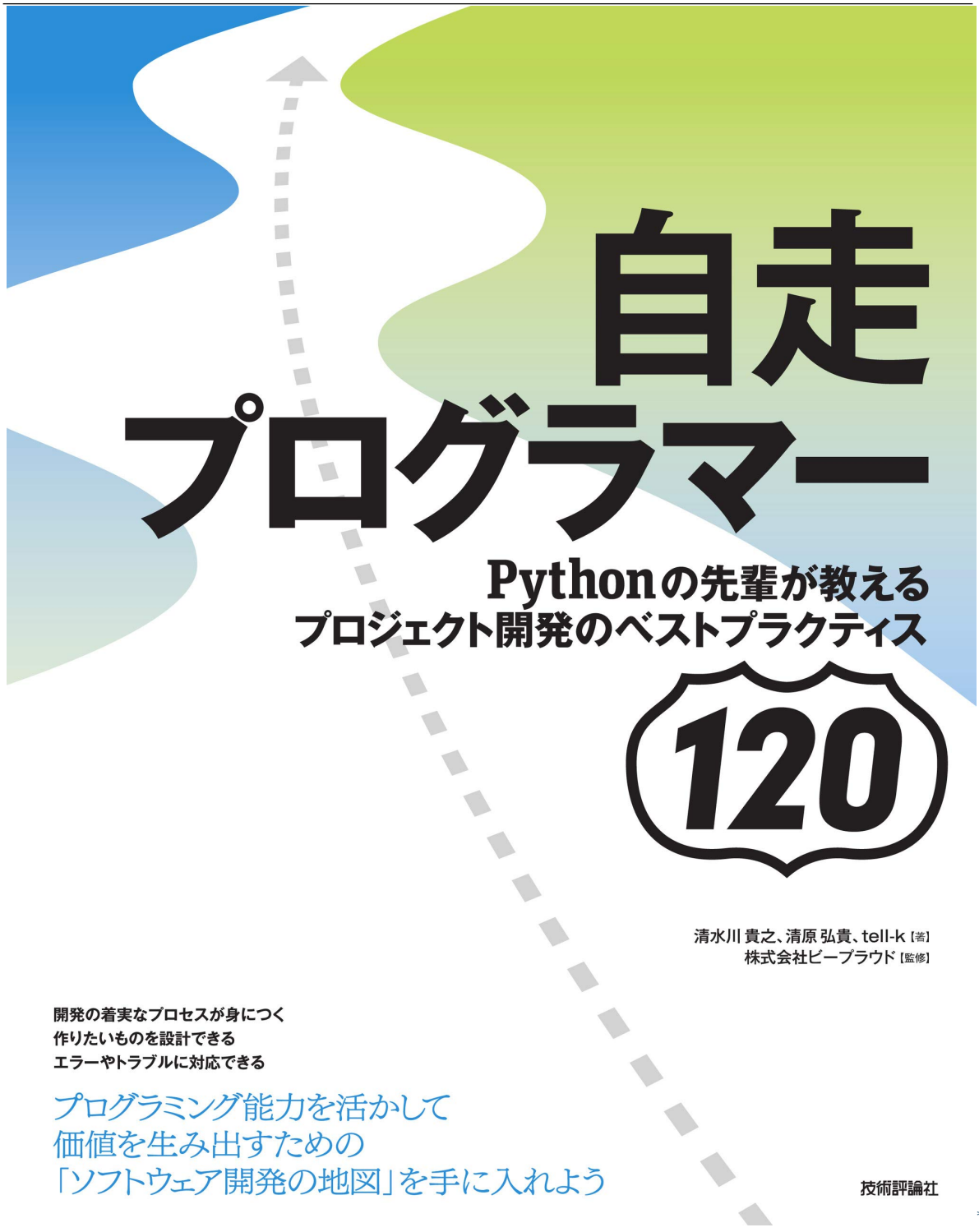
```
def filter_various_pattern(data_list, search_key, search_value, search_op):
    """ とにかくいろんなパターンで絞り込みができる関数 """

    result = []
    for data in data_list:
        target_value = data[search_key]
        if search_op == 'eq' and target_value == search_value:
```

(次のページに続く)

[*92] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
            result.append(data)
        elif search_op == 'gt' and target_value > search_value:
            result.append(data)
        elif search_op == 'gte' and target_value >= search_value:
            result.append(data)
        elif search_op == 'lt' and target_value < search_value:
            result.append(data)
        elif search_op == 'lte' and target_value <= search_value:
            result.append(data)
        elif search_op == 'is' and target_value is search_value:
            result.append(data)
        elif search_op == 'startswith' and target_value.startswith(search_value):
            result.append(data)
        elif search_op == 'endswith' and target_value.endswith(search_value):
            result.append(data)
    return result

# Use function
filter_various_pattern(data_list, 'gender', 'male', 'eq')
```



*93

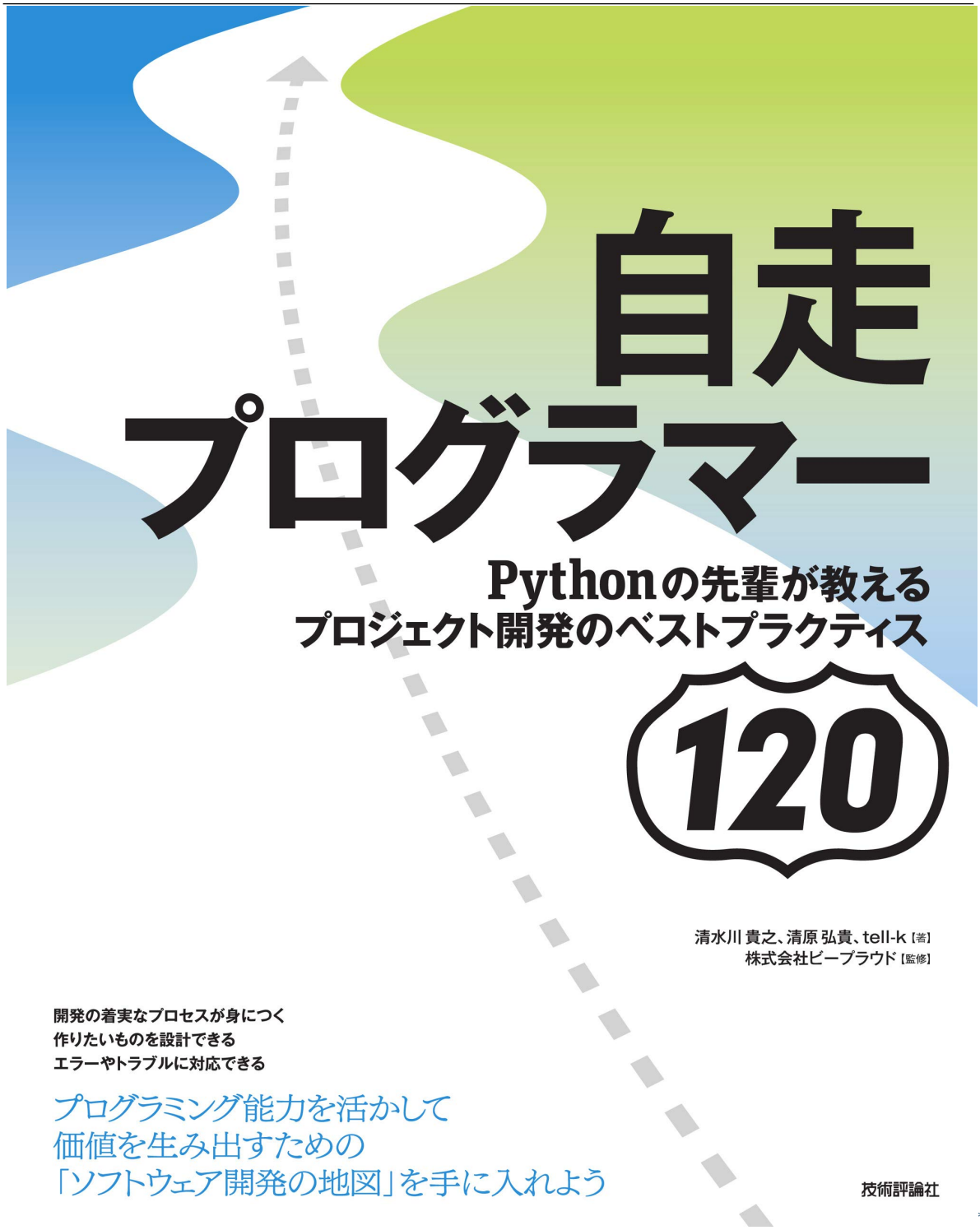
*93 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*94] をご参照ください

ベストプラクティス

あらゆるパターンに対応できるものよりも、目的を絞った実装をするほうが、将来的な保守性や、拡張のしやすさを維持できます。今回の例で言えば 特定の性別のデータだけを抜き出す という目的を満たす最小限のコードは以下のようになります。

[*94] https://gihyo.jp/book/2020/978-4-297-11197-7



*95

*95 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*96] をご参照ください

2.5.7 39:開発アーキテクチャドキュメント

プログラミング迷子**:** 決めた指針はどこにいった？

- 先輩 T：最近、また関数名に `is_` をつけるとか、他にもいくつか、決めたことが守られてないようだよ。
- 後輩 W：あー、そういえば前に言われたような気もします。
- 先輩 T：決めたときに、コーディング規約のページに書いたと思うんだけど、見ていない？
- 後輩 W：すみません、そこは最近見てませんでした。DB やテストのページは見てたんですけど、見るページが多くて毎回全部は見れなくて……。
- 先輩 T：なるほど、じゃあ 1 箇所にまとめようか。

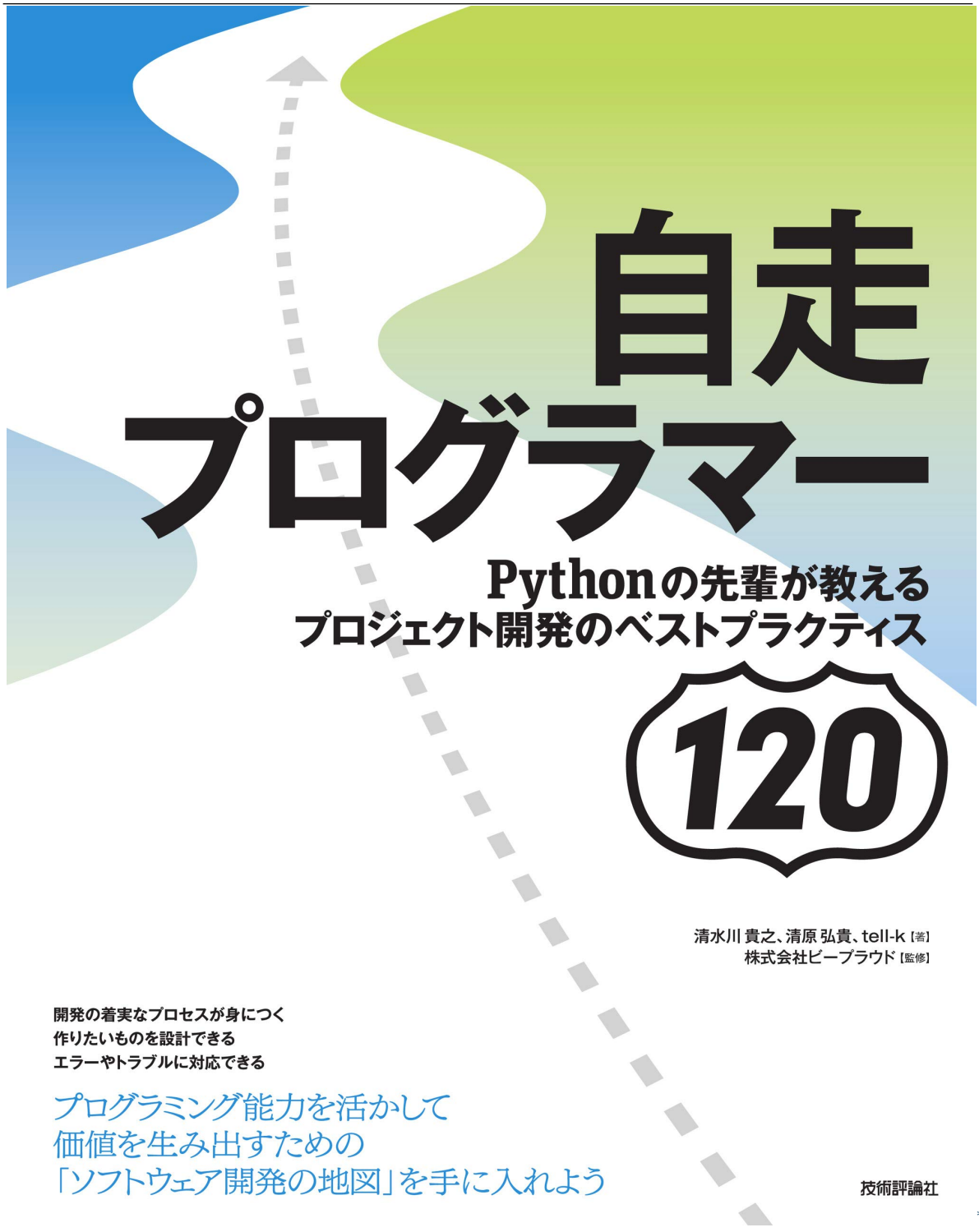
開発中に決めたチームの開発ルールやベストプラクティスは、開発期間とともに増えていきます。内容も、コーディング規約だけでなく、モジュール設計、関数引数の扱い、バリデーション方法、テーブルカラムの扱い、テストコードの書き方、Git 等のブランチの扱い、リリース手順、等々、多岐にわたっていきます。

決めたルールそれぞれをカテゴリ毎にドキュメントや Wiki ページにまとめていき、いつでも参照できるようにするべきです。しかし、そういったドキュメントには、具体的なコードの書き方やコマンド例、その方法を採用した経緯や哲学などが追記され、重要なポイントを押さえづらくなります。そこで、ルールの詳細や具体例とは別に、ルールの決定事項だけをまとめたドキュメントを用意して、開発ルール全体を俯瞰して確認できるようにしましょう。

ベストプラクティス

チームの開発運用ルールを 開発アーキテクチャドキュメント に書いて、更新していきましょう。開発アーキテクチャドキュメントはあらかじめ用意できるものではなく、プロジェクトの結果として完成していきます。チームで合意した選択基準や開発ルールを明文化し、気分や時間経過によって起こる実装方針のぶれをなくすのが、開発アーキテクチャドキュメントの役割です。

[*96] https://gihyo.jp/book/2020/978-4-297-11197-7



*97

*97 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*98] をご参照ください

[*98] https://gihyo.jp/book/2020/978-4-297-11197-7

2.6. レビュー

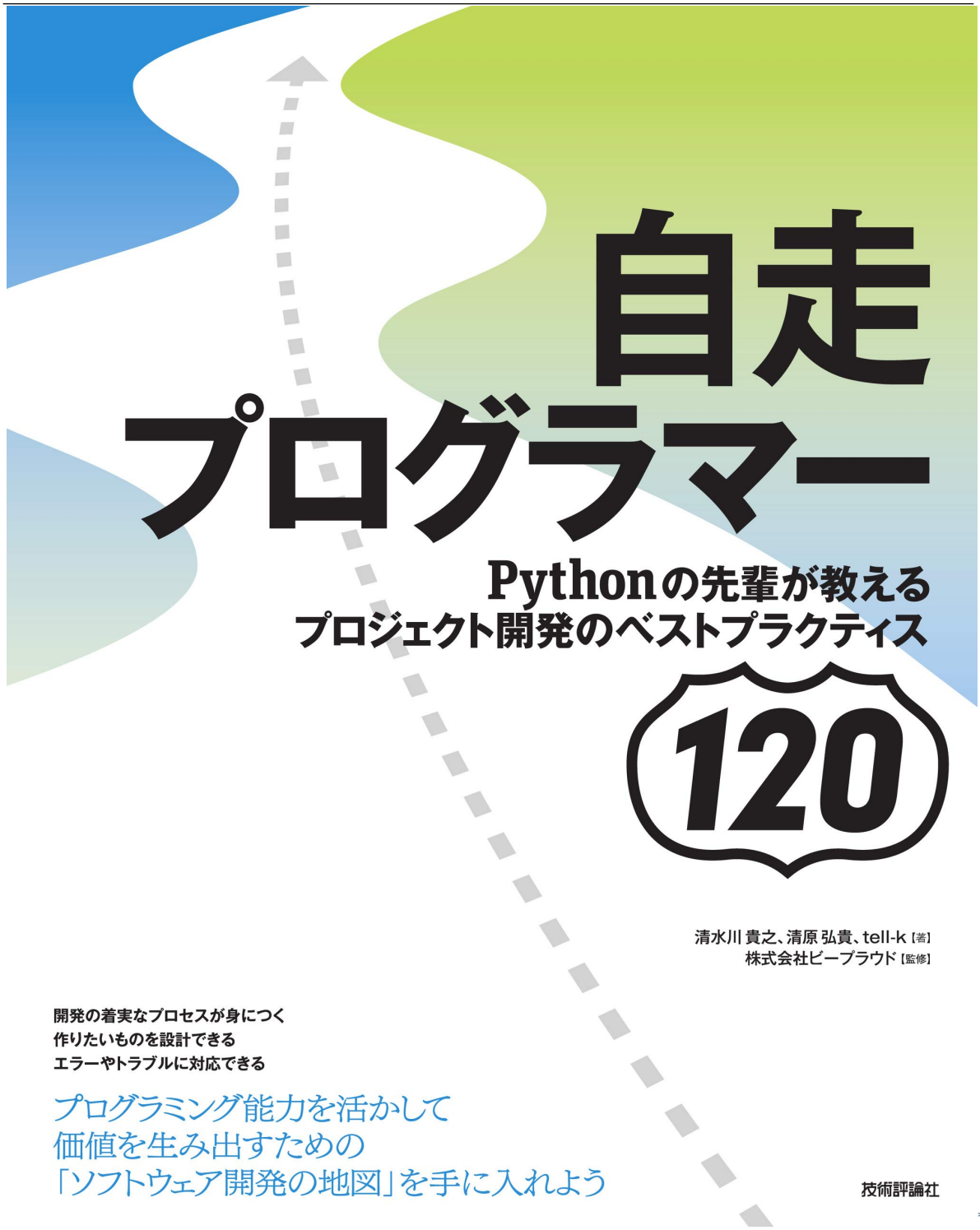
2.6.1 40:PR の差分にレビューアー向け説明を書こう

プログラミング迷子**:** 私が知ってることは先輩なら全部知ってるはず？

- 先輩 T：さっき頼まれたコードレビューなんだけど、ちょっといろいろ情報が足りてなくて、これだとレビューできないよ。
- 後輩 W：え、何が足りないですか？
- 先輩 T：まず、この新機能の仕様はどこにまとまってる？
- 後輩 W：このチケットにまとまってます。途中確認や変更がいろいろあったので、読む必要があるコメントはここと、ここと……あとこの添付ファイルと……。
- 先輩 T：それを読み解いてコードレビューするのは無理だなあ。まず仕様を 1 箇所見ればわかるようにまとめてください。チケットなら、最終的に決定した仕様をチケット本文に書くといいね。コメントだと流れて行っちゃうから。
- 後輩 W：わかりました。
- 先輩 T：そして、PR の差分それぞれに自分で「その差分が何のためのものなのか、どの仕様のためなのか」を書いてください。レビューアーがその差分を見たときにそれが書いてあれば質問せずに済むので、お互いに質問と回答を書く時間が減らせるよ。そしてもっと重要なのは、実装者自身で説明を書くことで自分の勘違いに気づくチャンスができることだね。
- 後輩 W：なるほどー。

GitHub の Pull Request（PR）[*99] 機能が登場して以来、コードレビューは格段に行いやすくなりました。だからといって、PR を作って渡せばわかってもらえる、という訳にはいきません。レビューアーがレビューしにくい原因は「説明不足」にあります。説明が不足していると、レビューアーは レビューするべき重要な箇所 に集中できず、些細な問題に気を取られてしまい、重要な問題を見逃してしまいます。レビューアーから見てわかりやすい依頼にするには、前提になる知識の差を埋める必要があります。

[*99] https://help.github.com/ja/github/collaborating-with-issues-and-pull-requests/about-pull-requests



*100

*100 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*101] をご参照ください

ベストプラクティス

セルフレビューを行い、レビューアー向けに知識の差を埋めるための説明を書きましょう。

セルフレビューを効果的に行う、6 つのプラクティスを紹介します。

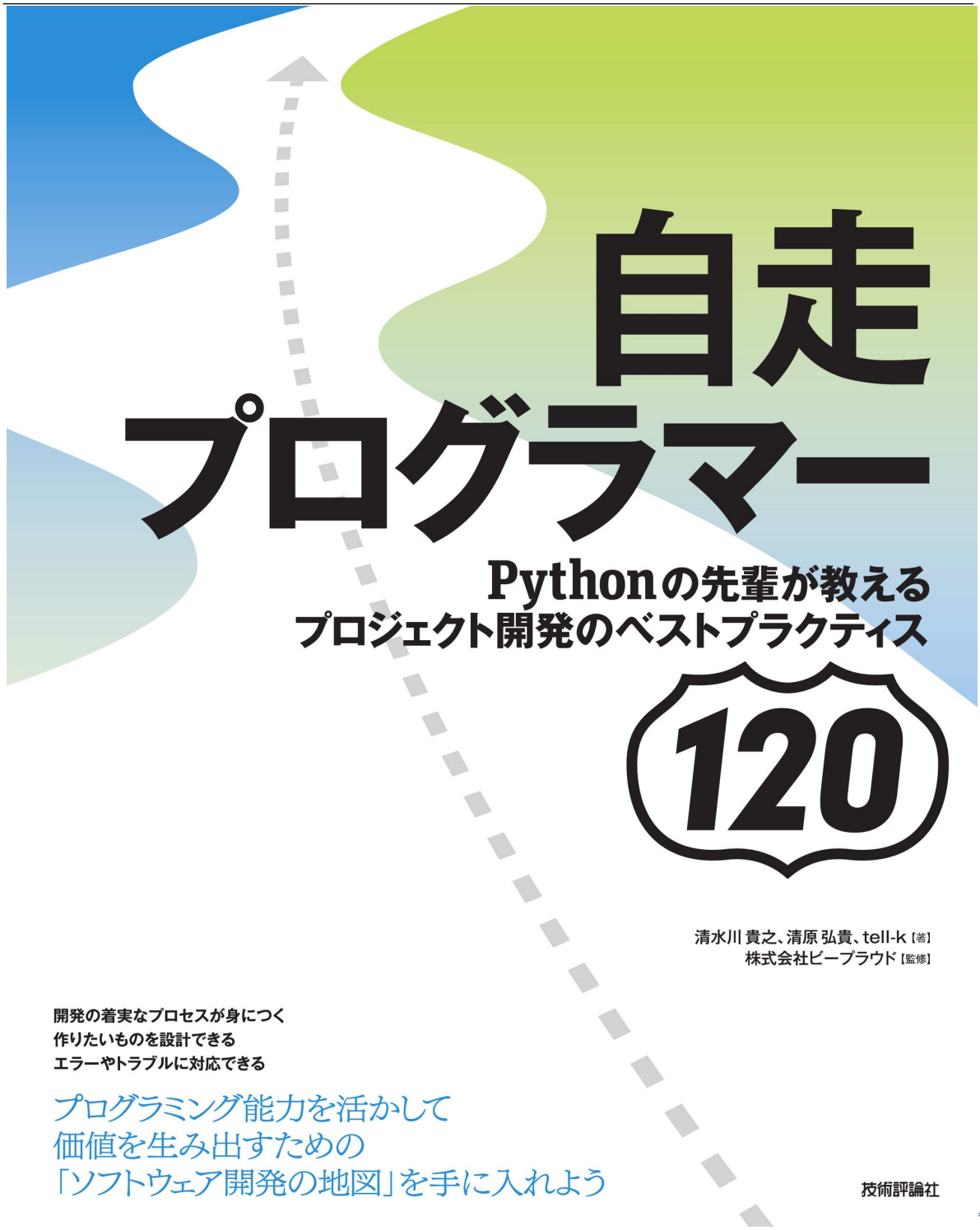
1. 実装の根拠を書く
2. 仕様や設計をまとめる
3. コードを改善する
4. 差分の説明をしてみる
5. PR の本文（description）に、「PR の目的」「仕様をまとめたページへのリンク」「レビューで確認してほしいこと」を書く
6. 差分コメントを恒久的な情報に移す

1. **実装の根拠を書く**　コード差分のコメントに、実装の根拠となる「仕様や設計へのリンク」を書くことで、レビュー依頼者がチェックできます。この作業で、レビュー依頼者は具体的にどの仕様を元に実装したのか、仕様と実装内容が一致しているのか確認できます。実装コードの中には、効率的なアルゴリズムや、普段使い慣れていない API 利用などもありますが、レビューアーはその知識がないかもしれません。レビュー依頼者がそのとき実装して初めて知った情報、初めて使った API についてもコメントに書いておきましょう。ここで、仕様の理解や API の使い方についての認識違いがあれば、レビュー依頼前に気づくことができます。

   以下を実践しましょう。

   - どの仕様のためのコードなのか、仕様へのリンクを書く
   - 仕様を満たしているか確認する
   - コードを書くときに調べたことがあれば、レビューアーへ伝えるためコメントに書く

[*101] https://gihyo.jp/book/2020/978-4-297-11197-7



*102

*102 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*103] をご参照ください

関連

- *10:* **コメントには「なぜ」を書く** (ページ 39)
- *39:* **開発アーキテクチャドキュメント** (ページ 144)

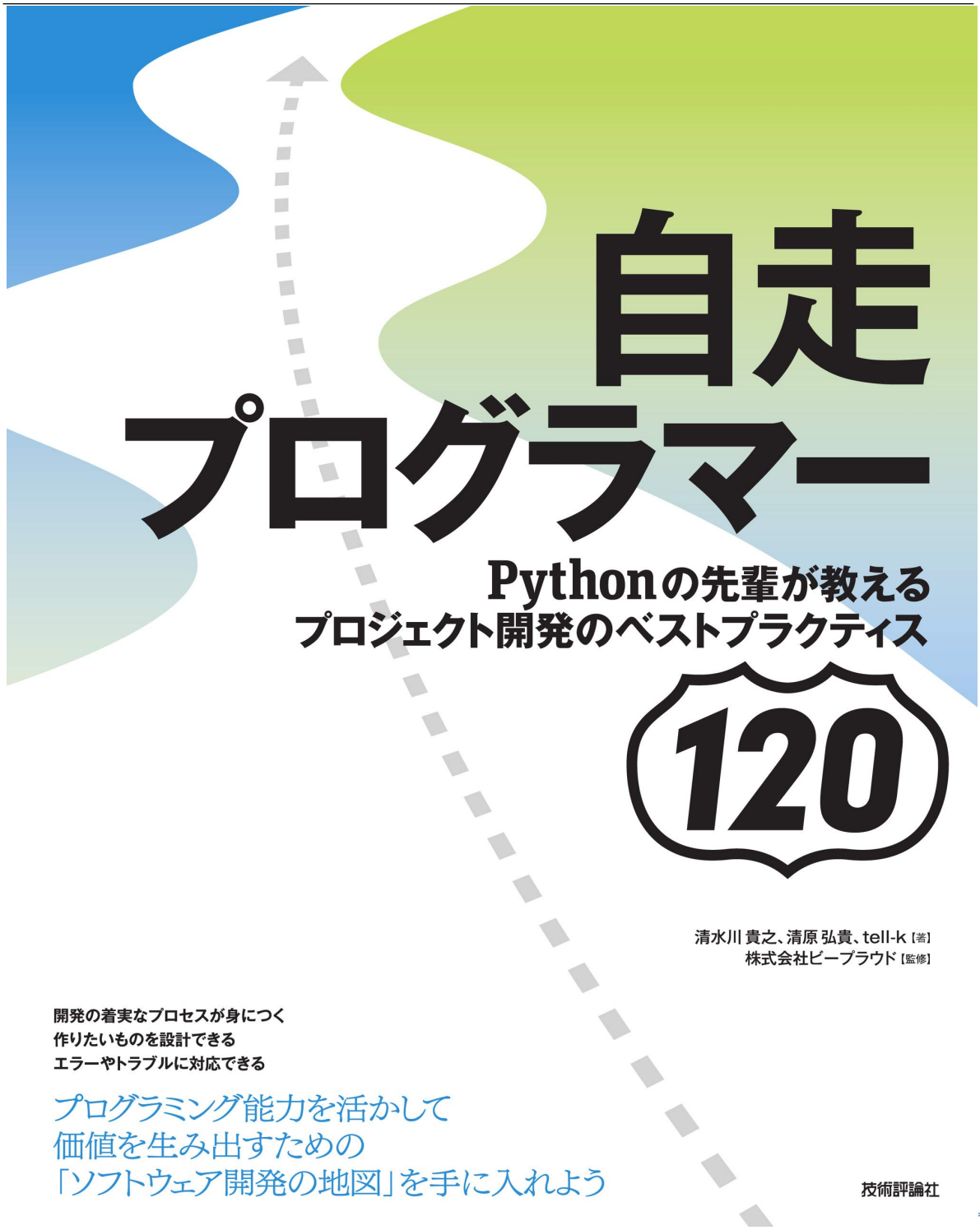
2.6.2 41:PR に不要な差分を持たせないようにしよう

プログラミング迷子**:** 私の手間は少ないほうが良い

- 先輩 T：さっきレビュー依頼された PR だけど、100 行近くある差分のほとんどが行末のスペース削除みたいだね。
- 後輩 W：はい、気になったのでついでに直しました。
- 先輩 T：実際にレビューするべき箇所を探すのが大変なんだけど……。
- 後輩 W：えっ（せっかく直したのに）、修正しないほうがいいですか？
- 先輩 T：そうですね、この PR の目的ではないので、こういう別の目的の修正は、別の PR にしてください。
- 後輩 W：でもそれだとブランチ作ったり PR 書いたり手間じゃないですか。
- 先輩 T：その手間を実装者がやらなかった分、レビュー時間が延びてレビューアーの時間が使われていくんだよ。実装者なら簡単に分割できるけど、レビューアーは目的が混ざった状態から見始めるので、頭の中で分類しながらレビューするのはとても大変で時間がかかるんだよ。

ちょっとした問題に気づいて、それを修正するのは良いことのように思えます。しかし、前述の例ではちょっとした修正がレビューの邪魔になってしまっています。1 つの PR に複数の目的 が含まれていると、レビューで確認するべきことを見落としてしまいます。

[*103] https://gihyo.jp/book/2020/978-4-297-11197-7



*104

*104 https://gihyo.jp/book/2020/978-4-297-11197-7

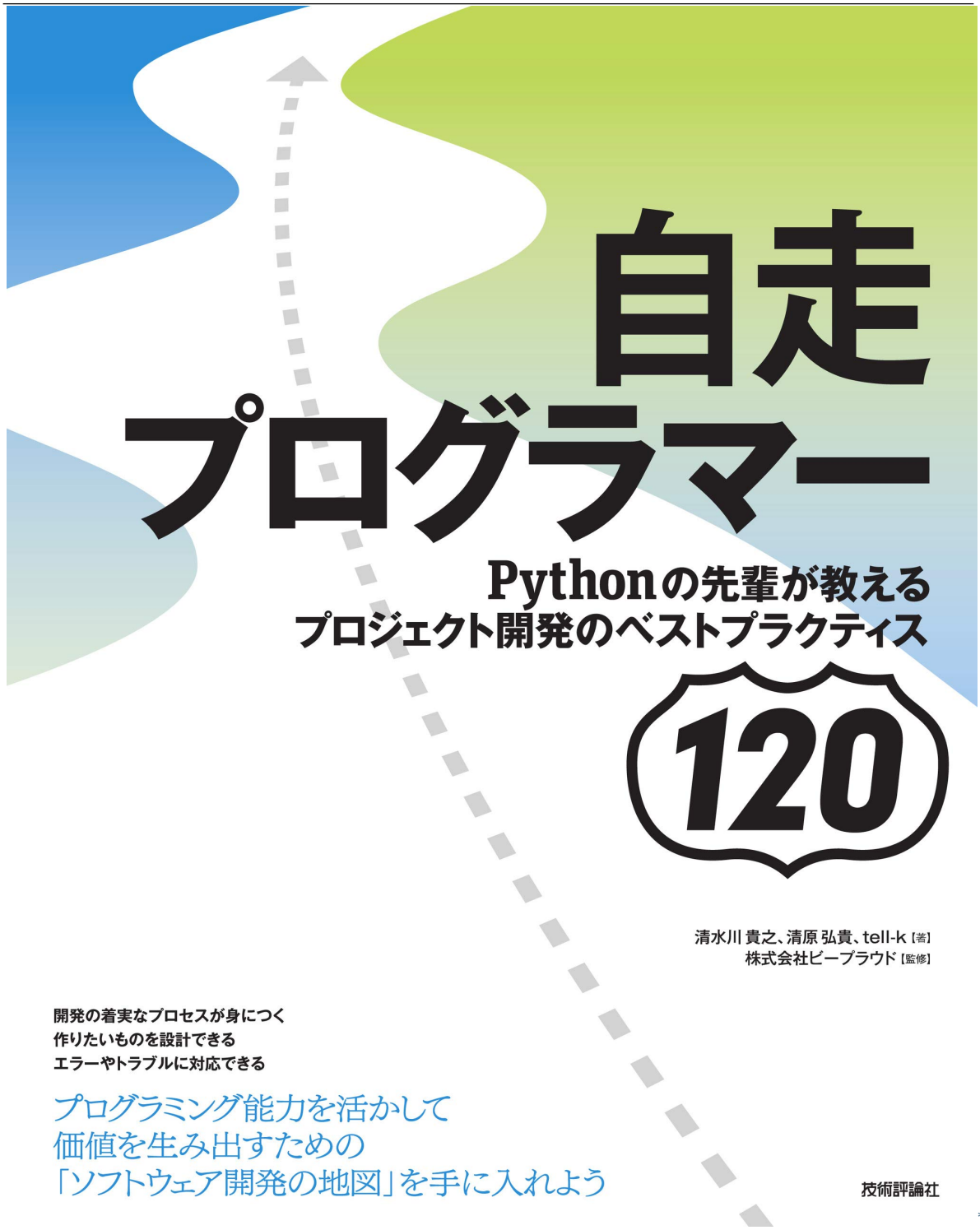
（中略）詳細は書籍 自走プログラマー[*105] をご参照ください

ベストプラクティス

以下のポイントを守りましょう。

- PR の目的を 1 つに絞る。たとえば、ロジックの変更と単純なフォーマット変更は別の PR に分ける
- ...

[*105] https://gihyo.jp/book/2020/978-4-297-11197-7



*106

*106 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*107] をご参照ください

関連

- *34:* **一度に実装する範囲を小さくしよう** (ページ 127)
- *40:PR* **の差分にレビューアー向け説明を書こう** (ページ 147)

2.6.3 42:レビューアーはレビューの根拠を明示しよう

プログラミング迷子**:** 先輩、それ先に言ってよ

- 後輩 W：この PR をレビューお願いします（今回は *40:PR* **の差分にレビューアー向け説明を書こう** (ページ 147) を実践して説明を書いたから、バッチリだぞ！）
- 先輩 T：はい（1 分後）さっきの PR だけど、コーディング規約に準拠してないのでレビューできないよ。ざっと見た感じ、ログ出力が他のところと合ってないようです。クラス継承による差分実装を多用しているようだけど、このプロジェクトではできるだけ避けてください。使用する場合も、デメテルの法則に違反しないようにしてください。
- 後輩 W：コーディング規約、PEP8 には準拠しているはずですが……、あと、デメテルの法則って初めて聞いたんですが何ですか？
- 先輩 T：あれ、このプロジェクトのコーディング規約とかって言ってなかったっけ？　この Wiki に書いてあるので読んでみてください。
- 後輩 W：（何かいろいろ書いてある……先に言ってほしかった……）ログ出力はどこを見たらわかりますか？
- 先輩 T：あれ、書いてない？　ごめん今から書くわ。
- 後輩 W：（Wiki にも書いてなかったことを実装するのは無理では……もしかして思いつきで指摘してるのでは？）

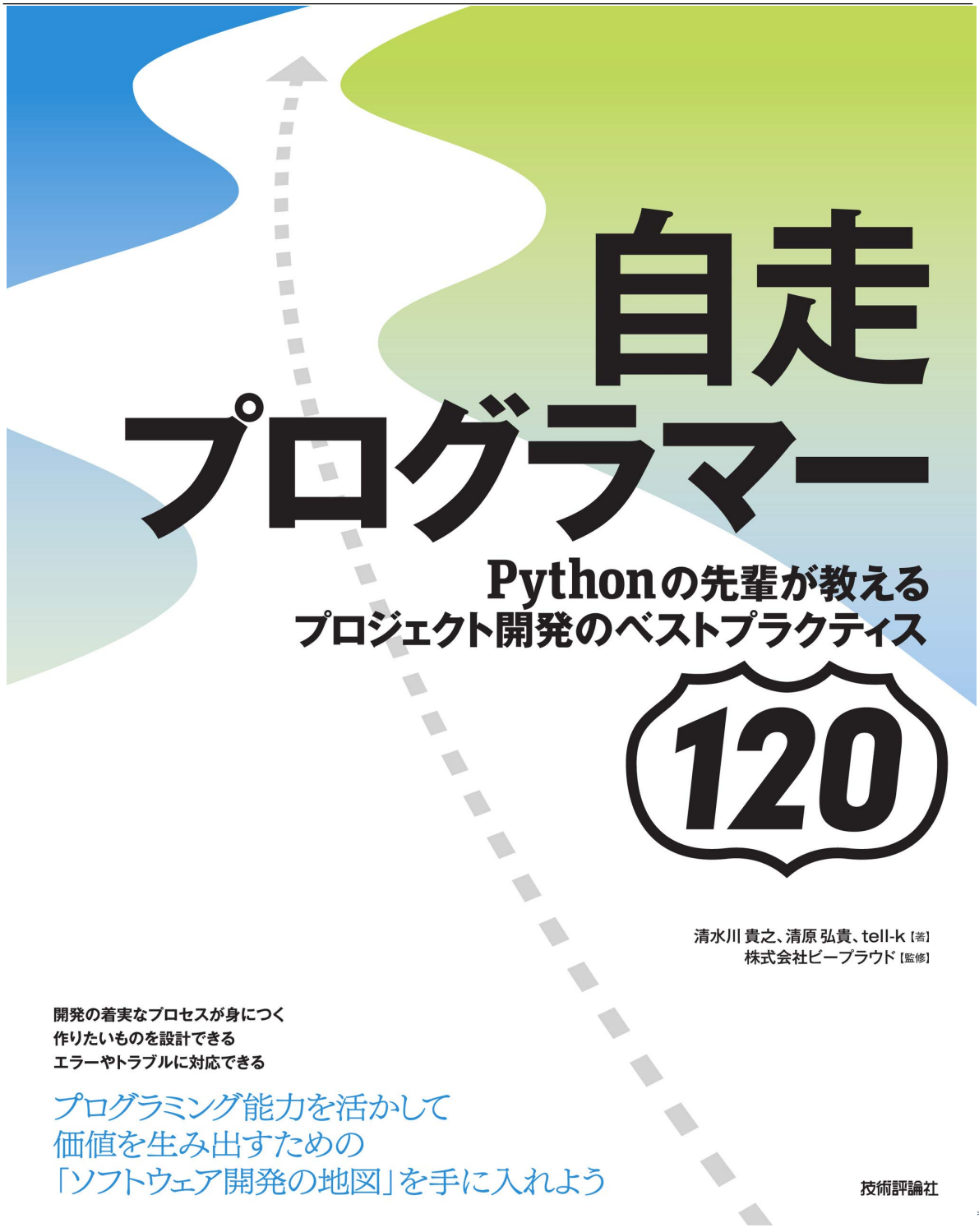
レビュー観点をまとめず、「ルールなど、わからないことがあったら聞いて」という進め方ではうまくいきません。どんなルールを採用しているのかわからないと、聞けないこともたくさんあります。「継承を使って良いですか」と質問する人はあまりいないでしょう。

[*107] https://gihyo.jp/book/2020/978-4-297-11197-7

レビュー観点がないと、レビュー指摘の根拠がありません。先輩だから、世の中がそうだから、というのは根拠にならないですし、そのような指摘に対して根拠を求めたり同じ話題で毎回議論するのも時間の無駄です。議論をするのであれば、決めておいた観点を変更するための議論のほうが建設的です。

ベストプラクティス

プロジェクトメンバー全員でレビュー観点をまとめて、合意しておきましょう。レビューアーは、合意された観点を元にレビューしましょう。



[*108]

[*108] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*109] をご参照ください

2.6.4 43:レビューのチェックリストを作ろう

プログラミング迷子**:** どこまで確認したの？

- 後輩 W：この PR をレビューお願いします（レビュー観点を自分でも確認したから、今度こそバッチりだぞ！）
- 先輩 T：はい（1 分後）うーん、だいたいは良いと思うんだけど、規約に合っていない部分がいくつかあるみたいだね。セルフレビューで確認してる？
- 後輩 W：もちろんです。どこが合ってないですか？
- 先輩 T：このファイルなんだけど、これだとログに個人情報が出てしまうんじゃないかな？
- 後輩 W：あっ、ほんとうだ。おかしいな、ちゃんとレビュー観点を元に確認したんですが、見逃したみたいです。すみません。
- 先輩 T：なるほど。どこまで確認したか、ちょっとまとめて教えてくれる？

レビュー観点が用意されていてその観点を確認していたとしても、どこまで確認したかが不明確だとレビューアーは効率良くレビューを進められません。それに、確認項目の見逃しはどんなに開発に慣れてきても発生してしまうものです。記憶や慣れに頼らずに、漏れなく確認する方法を検討しましょう。

ベストプラクティス

レビューチェックリスト を作っておき、レビューを依頼する前にチェックしましょう。

チェックリストがあればチェック漏れをなくせますし、レビュー依頼される人も「ここまでは自分でも確認しているんだな」ということがわかります。GitHub であれば、PR のテンプレートを用意できるので、以下のようにチェック項目として観点を記載しておきましょう。レビュー依頼前のチェック項目があれば、「先に言ってよ」という問題も回避できます。

[*109] https://gihyo.jp/book/2020/978-4-297-11197-7

リスト 2.22　.github/PULL_REQUEST_TEMPLATE.md

```
PR 作成時のチェック項目

- [ ] チケットタイトルを次の書式で記載したか？ `refs #<issue-id> <チケット名>`
- [ ] label に `WIP` を指定したか?（レビューが必要になるまで付けておく）
- [ ] label の `WIP` を解除したか?（レビューが必要になったら）
- [ ] reviewers にレビューアーを指定したか？

チケット URL

- [ ] https://github.com/<org>/<proj>/issues/99999999

このレビューで確認してほしい点

- [ ] 機能 xxx をクリックしたら xxxx できること <仕様 1 リンク>
- [ ] 機能 yyy をクリックしたら xxxx できること <仕様 2 リンク>

レビュー提出前 規約セルフチェック

- [ ] C1 各種機能に適切なパーミッションが設定されているか
- [ ] C2 変更が発生するリクエストでは CSRF トークンを使用しているか
- [ ] C3 トークンは適切な時間で破棄されているか
- [ ] C4 エラーログ、スタックトレースに重要情報が含まれていないか
- [ ] C5 /tmp にファイルを書いていないか
- [ ] C6 SQL を文字列操作で組み立てていないか
- [ ] C7 システム外部から渡ってくる入力はバリデーションしているか
- [ ] D1 モデルの構造に着目したチェック
- [ ] D2 機能単体に着目したチェック
- [ ] D3 機能の結合に関連したチェック
- [ ] E1 処理の長さで関数を分割しない
- [ ] E2 引数の数を減らす
- [ ] E3 継承の利用を最小限にする（Flat is better than nested）
- [ ] E4 継承で挙動を変えていないこと（リスコフの置換原則）
- [ ] E5 型ヒントが書かれてること
- [ ] E6 ログ出力は規約<link>に合っていること
- [ ] E7 実装されている変更は仕様（Wiki）に記載、反映されていること
```

(次のページに続く)

(前のページからの続き)

レビュー提出前 動作セルフチェック

- [ ] UnitTest **はすべて通っているか**
- [ ] **差分は期待どおりに動作しているか**

レビューアーからの確認項目

- [ ] <**確認内容**> <PR **コメント** URL>
- [ ] <**確認内容**> <PR **コメント** URL>

関連

- *42:* **レビューアーはレビューの根拠を明示しよう** (ページ 155)

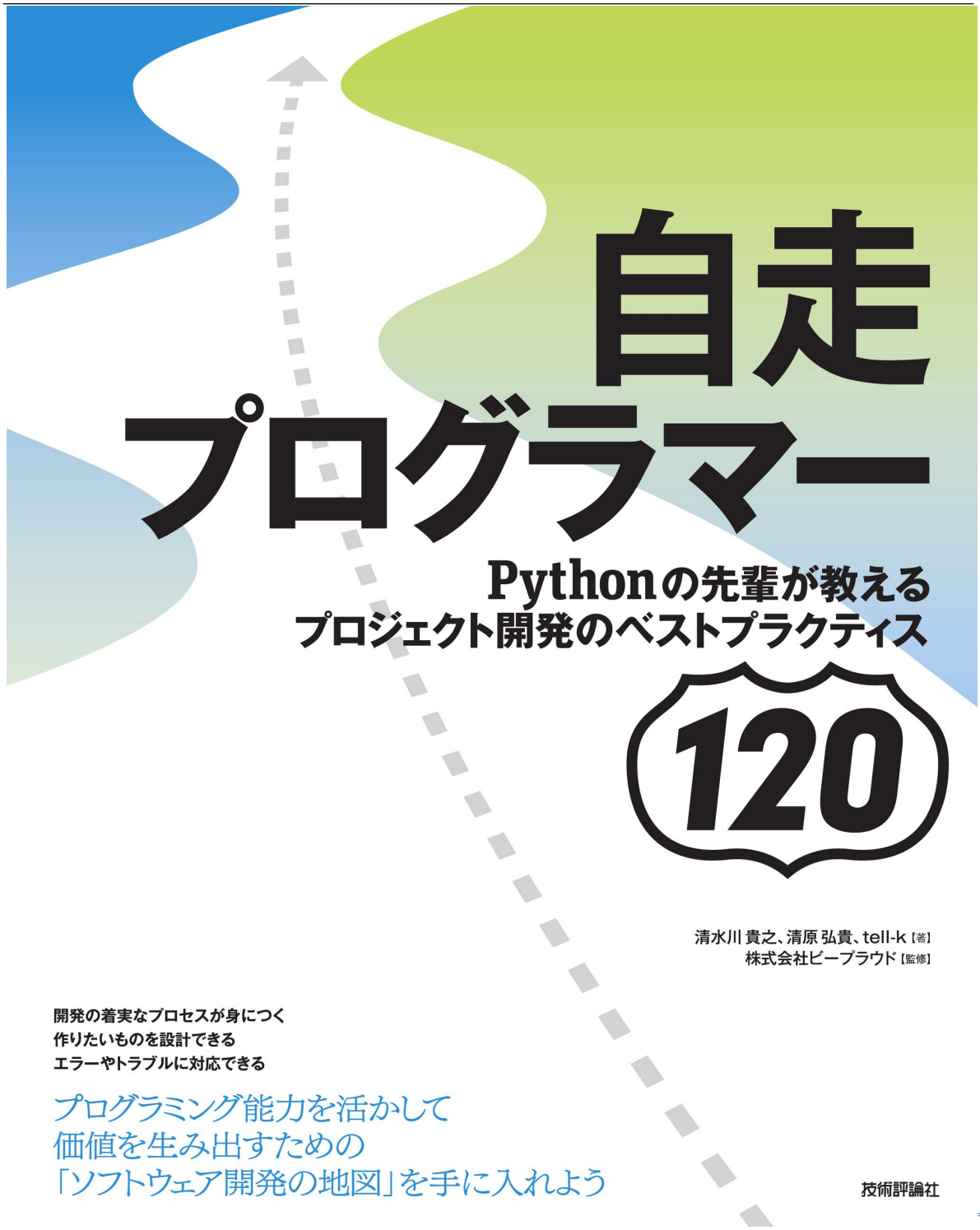
2.6.5 44:レビュー時間をあらかじめ見積もりに含めよう

プログラミング迷子**:** レビューの時間なんて見積もりに含めてなかったんだけど

- 先輩 T：ごめん、仕事が終わらなそうなんで今日のイベント欠席するよ。
- 同僚 A：そっか。だいぶ疲れているみたいだけど、そんなに大変なプロジェクトなの？
- 先輩 T：機能はそうでもないんだけど、レビューにかなり時間を取られてしまって、自分がコードを書く時間が足りないんだ。
- 同僚 A：へー。レビューにどのくらい時間を使ってるの？
- 先輩 T：1 日 3、4 時間かな……。
- 同僚 A：えっ、1 日の半分以上？　それはレビューに時間かけすぎだよ。
- 先輩 T：しかも、実装とユニットテストは見積もり時間に入れてあったけど、レビュー時間は含めてなかったんだ。
- 同僚 A：たしかに、見積書に「レビュー時間」という項目はないからなあ……。

見積もりにコードレビューの時間を含めずにいると、レビューにかける時間の分だけスケジュールが遅れて

いきます。過去に同じような見積もり方法で問題がなかったとしても、新しいメンバー、新しく利用するライブラリ、難しい機能の実装など、十分なレビュー時間を必要とする状況はいろいろあります。



*110

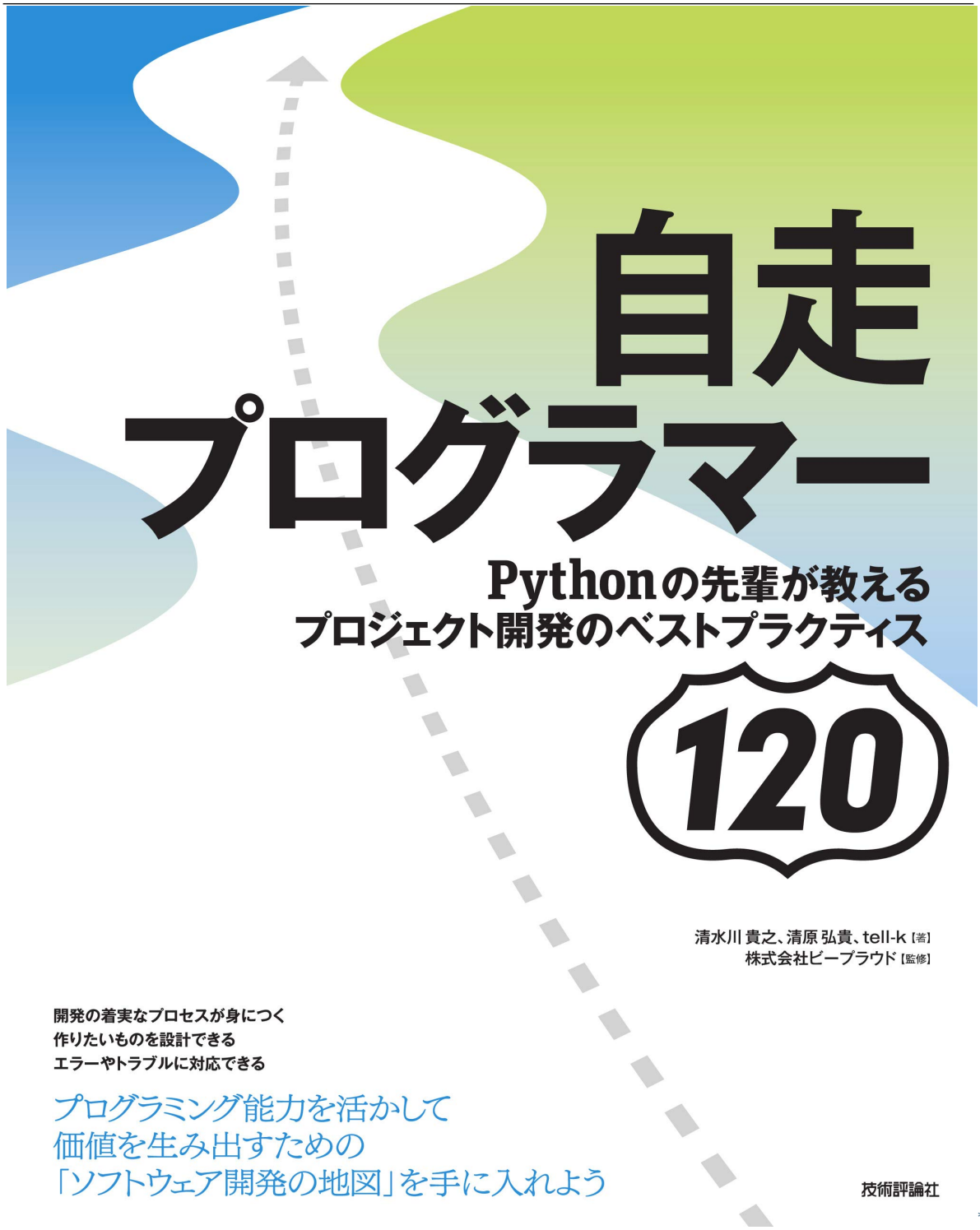
*110 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*111] をご参照ください

ベストプラクティス

工数見積もり時に、レビュー時間も工数として明示的に見積もりましょう。

[*111] https://gihyo.jp/book/2020/978-4-297-11197-7



*112

*112 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*113] をご参照ください

品質の担保という重要な目的を達成するために必要な時間は、見積書に工数として明示しておくべきです。そのうえで、スケジュールや金額が問題になるのであれば、一部の品質を下げるか機能を減らすかを相談しましょう。

2.6.6 45:ちょっとした修正のつもりでコードを際限なく書き換えてしまう

プログラミング迷子**:** ちょっと修正、のついでに

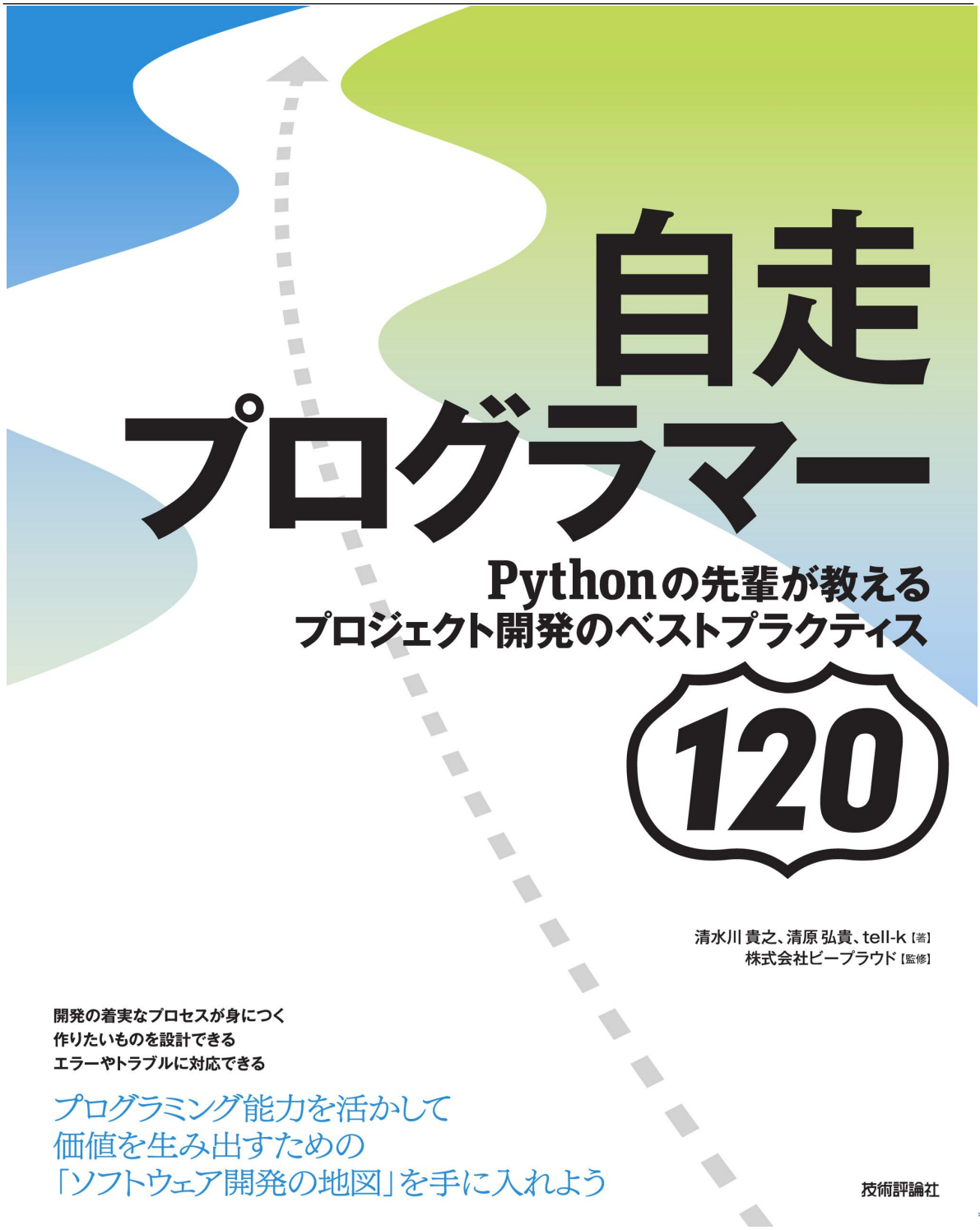
- 後輩 W：昨日レビューしてもらった PR なんですが、問題があったのでちょっと修正しました。修正した問題はちょっとなのでレビューなしで大丈夫です。
- 先輩 T：お、了解（どれどれ、チラッと見ておこうかな……ちょっとじゃない、がっつり書き換わってる！！）がっつり書き直されてるんだけど、どのへんが「ちょっと修正」なの？
- 後輩 W：特定の組合せのときだけエラーになる、ちょっとした問題を修正しました。
- 先輩 T：いやそうじゃなくて、コードレビューしたところがあらかた書き換わってるじゃない。
- 後輩 W：直しながら、どうせなら設計変えたほうがいいと気づいたので、ついでに修正しました。

「修正によって直した動作はちょっとしたもの」だとしても、コードを大幅に書き換えているのであれば再度レビューするべきです。変更した挙動の大小でレビューするかどうかを決めてしまうと、動作が変わらないリファクタリングはレビュー不要、ということになってしまいます。

ベストプラクティス

挙動が変わるなら、レビューしましょう。挙動が変わらなくても、変更範囲が大きいならレビューしましょう。

[*113] https://gihyo.jp/book/2020/978-4-297-11197-7



[*114]

[*114] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*115] をご参照ください

[*115] https://gihyo.jp/book/2020/978-4-297-11197-7

第 3 章

モデル設計

3.1. データ設計

3.1.1 46:マスターデータとトランザクションデータを分けよう

RDB（Relational DataBase）についての知識をひととおり学び、現実世界のデータを元にデータ設計をしようとしたときに、どこから手をつけて良いのかわからないと思ったことはありませんか？

ここではデータの種類と利用目的に応じてテーブルを分類する方法について説明します。

ベストプラクティス

実世界のデータの塊を RDB で扱う場合、マスターデータとトランザクションデータの 2 種類に大別して考えるとデータ設計がスムーズに進みます。この 2 種類を区別して考えないと無駄に多くデータを増やしてしまったり、*47:* **トランザクションデータは正確に記録しよう** (ページ 173) で紹介する失敗のように、過去のデータが意図せずに復元できなくなります。

マスターデータとは、データの中でも基礎となるもので、商品情報や従業員情報など 1 つひとつの基礎的な情報を記録します。たとえば、商品マスターであれば、商品名、型番、仕様など個々の商品の情報を扱います。

一方でトランザクションデータとは、システム上で発生した取引などの出来事を記録したデータのことで、一般に履歴と呼ばれるものを指します。たとえば、商品の購買履歴や、従業員への給与支払い履歴などです。イメージしやすいように図で考えてみましょう。

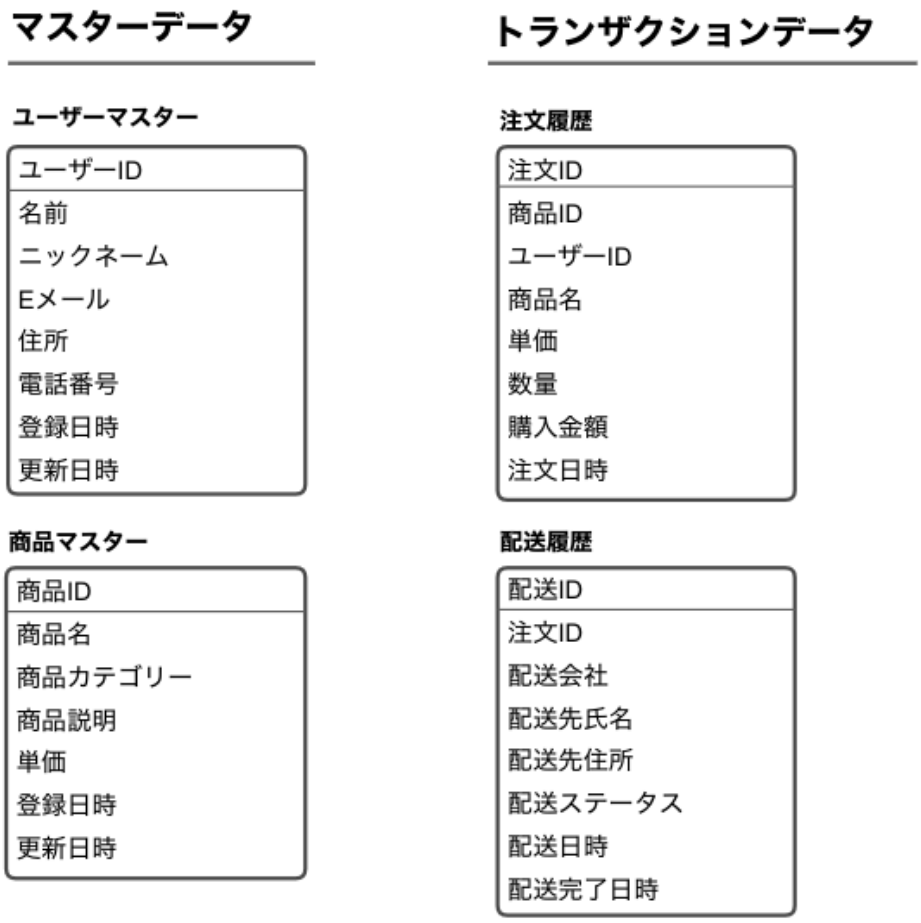
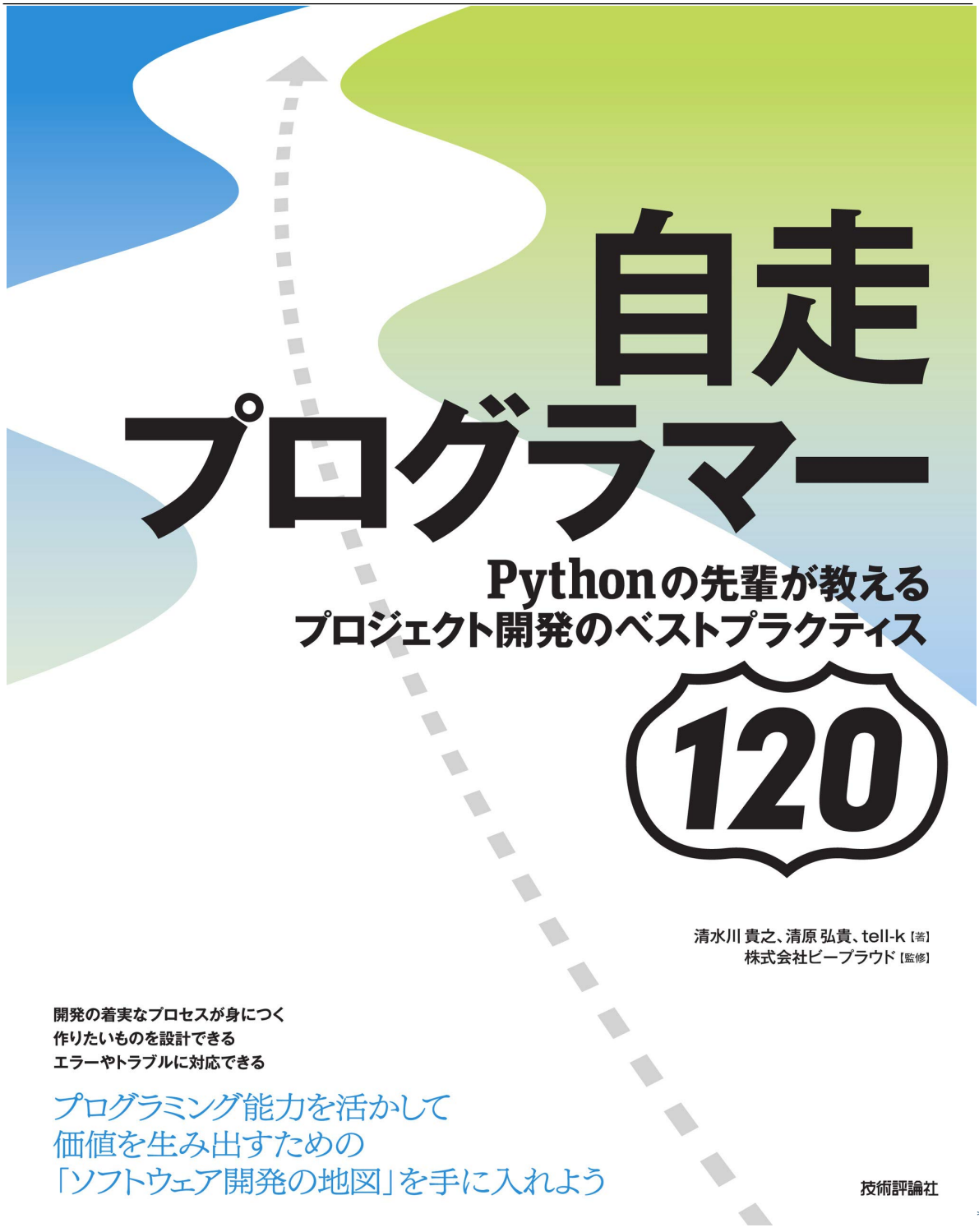


図 3.1　マスターデータとトランザクションデータの例

マスターデータとトランザクションデータのイメージはなんとなくできたと思いますが、実際に何を基準として分類していけば良いでしょうか。



*116

*116 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*117] をご参照ください

3.1.2 47:トランザクションデータは正確に記録しよう

履歴系のデータを設計したときに、システムの運用が始まってからカラムが足りないとか、当時のデータが再現できない等のトラブルになることがあります。そのようなトラブルはどうすれば避けられるのでしょうか。

具体的な失敗

ある EC サイトでユーザーの商品と注文履歴を以下のように管理していました。

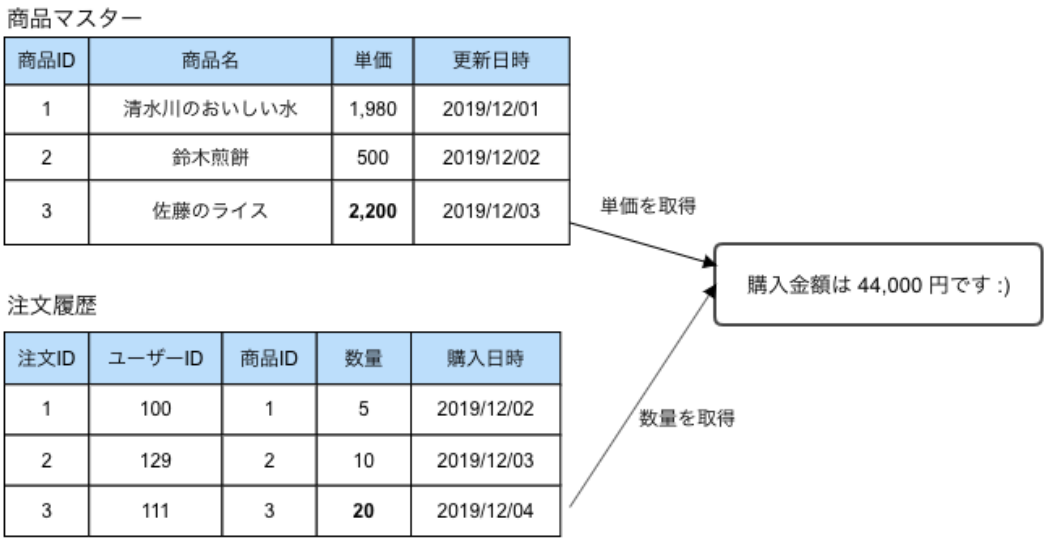


図 3.2　商品マスターと注文履歴

このサイトでは購入金額を注文履歴で閲覧できるようにするために、都度計算して表示していました。

商品マスターの単価×注文履歴の数量＝購入金額

[*117] https://gihyo.jp/book/2020/978-4-297-11197-7

ところがある日「佐藤のライス」の単価を変更したら、過去の購入履歴の金額まで変わってしまう というトラブルが発生してしまいました。

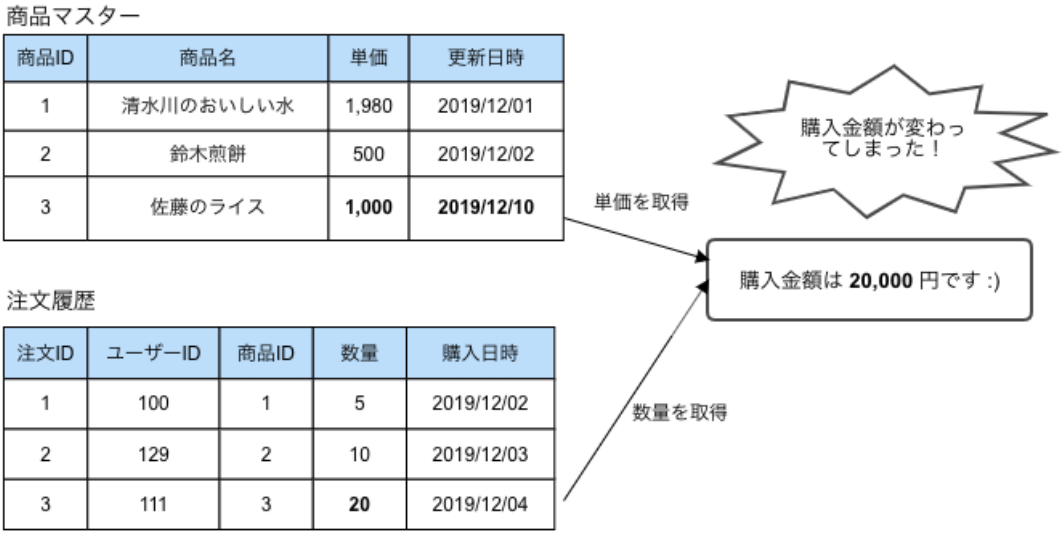


図 3.3　過去の購入金額が変動してしまう例

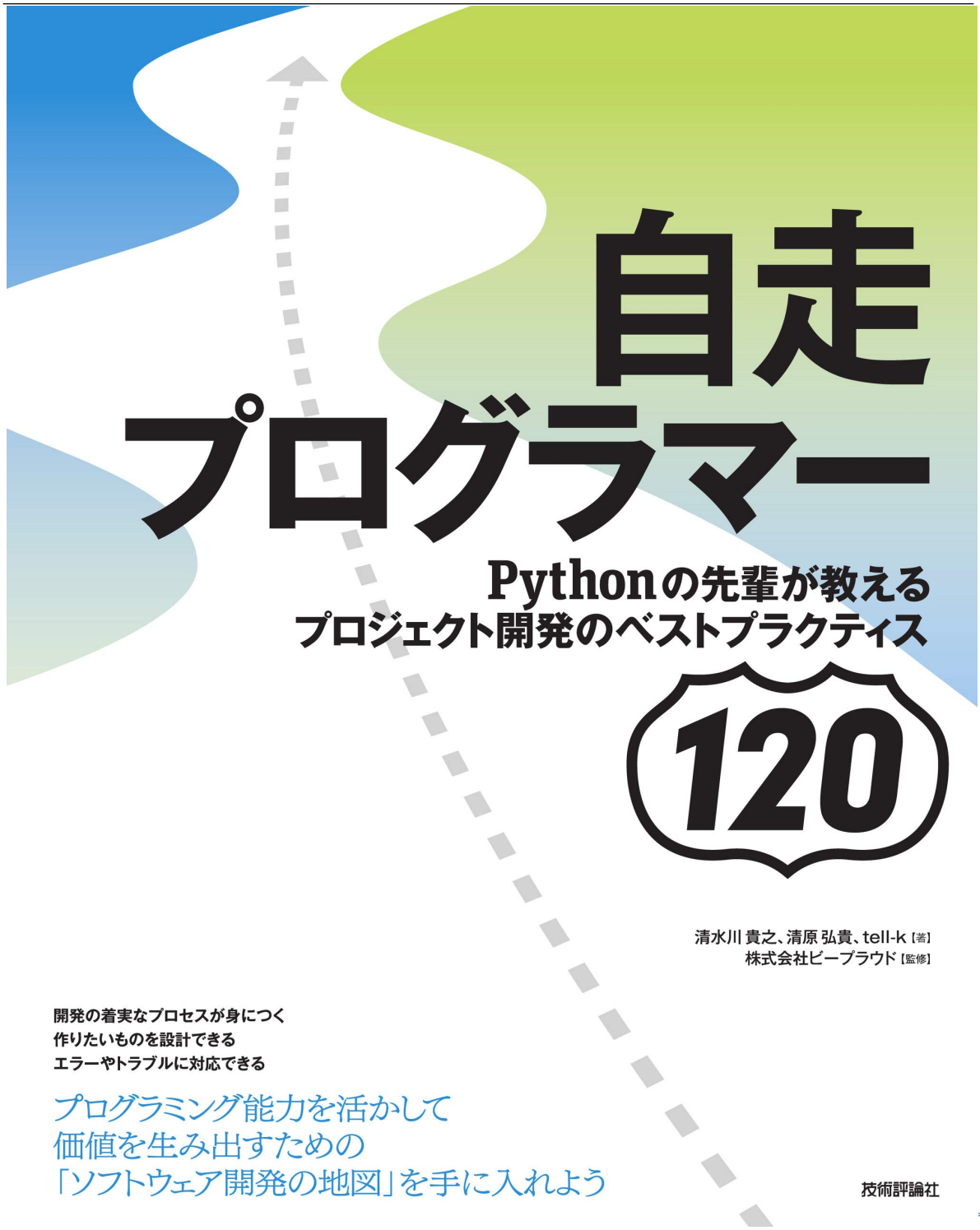
これは何がいけなかったのでしょうか？

商品マスターの「現在の単価」を、購入金額を計算するために必要な「購入当時の単価」として使用してしまったのが原因です。そのため過去の事実が失われてしまったのです。

ベストプラクティス

トランザクションデータに「そのときの行為」をデータとして正確に記録しましょう。安易に正規化して重複を排除して、必要なデータまで削ってしまわないように気をつけましょう。

今回の場合は、単純に注文履歴に購入当時の単価を追加してあげれば良さそうです。



[*118]

[*118] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*119] をご参照ください

関連

- *46:*マスターデータとトランザクションデータを分けよう (ページ 170)

3.1.3 48:クエリで使いやすいテーブル設計をする

RDB を運用していて、大量のデータがあるテーブルにあとからカラムを追加しなければならなかったり、無駄に複雑なクエリが必要になったりして困ったことはありませんか？

具体的な失敗

以下のような注文履歴テーブルと注文明細テーブルがあるとします。2 つのテーブルは良く正規化されています。たとえば、注文履歴では購入金額のカラムは持っていません。注文明細では注文日のカラムを持っていません。

注文履歴

注文ID	ユーザーID	配送先ID	注文日
1	100	1	2019/12/01
2	101	2	2019/12/01
3	102	3	2019/12/10

注文明細

注文明細ID	注文ID	商品ID	商品単価	注文個数
1	1	20	100	4
2	1	21	200	5
3	1	22	300	6
4	2	20	100	5
5	2	23	900	10
6	3	21	200	7

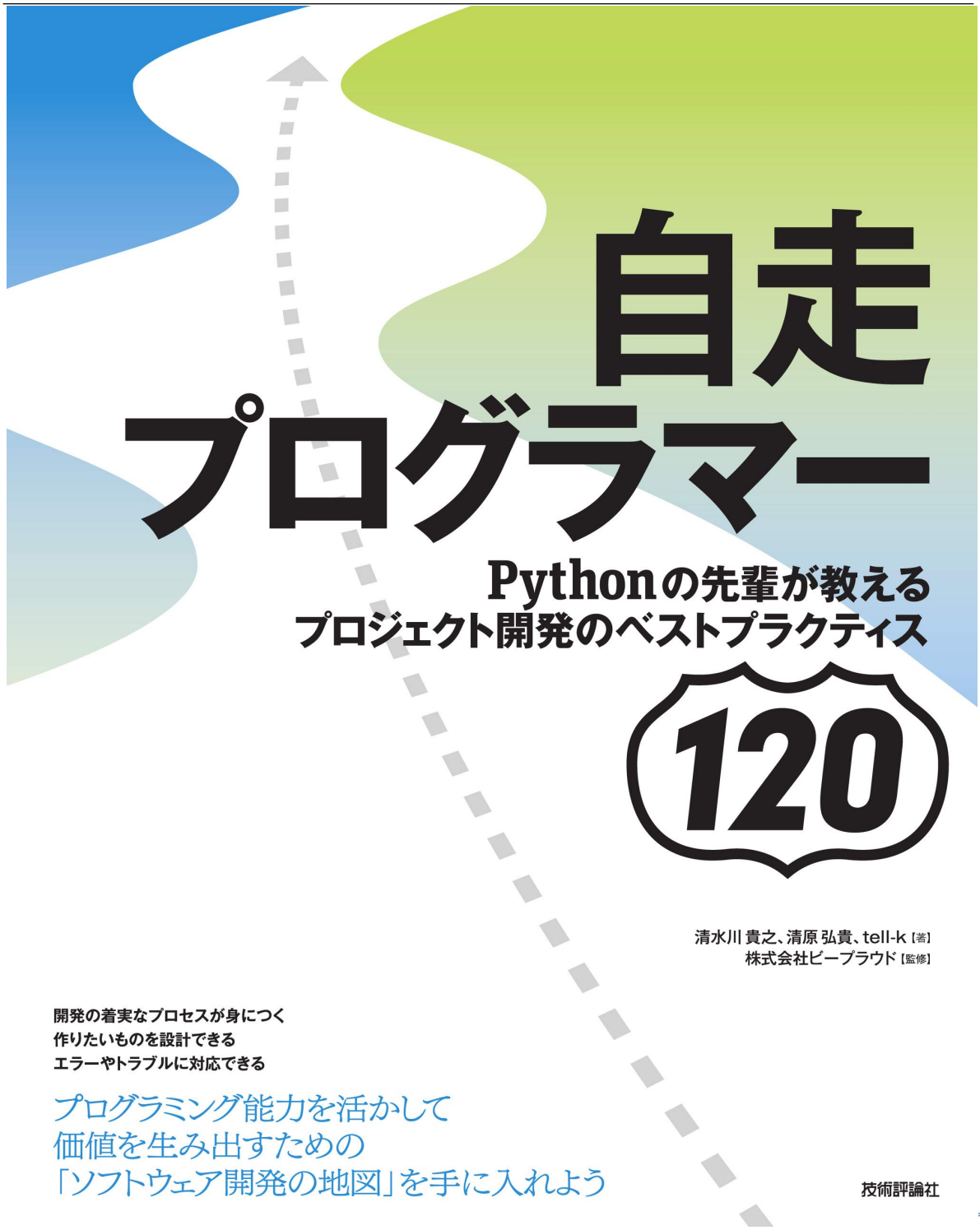
図 3.4　注文履歴テーブルと注文明細テーブル

このときに以下のような条件でデータを検索するよう依頼されたとします。

[*119] https://gihyo.jp/book/2020/978-4-297-11197-7

1. 注文日毎の売上がいくらか表示したい
2. 特定の期間に購入された商品 ID とその個数を表示したい

それぞれの要件のデータを抽出できるように下記のクエリを発行するようプログラムを開発しました。



*120

*120 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*121] をご参照ください

当初は集計した結果がすぐに表示されることを確認していました。しかし、時間が経ち、データ量が増えていく過程で徐々に集計に時間がかかるようになりました。

なぜ時間がかかるようになったのでしょうか？原因は大量のデータが入ったテーブルに対して JOIN を含む SQL が頻繁に実行されたことです。

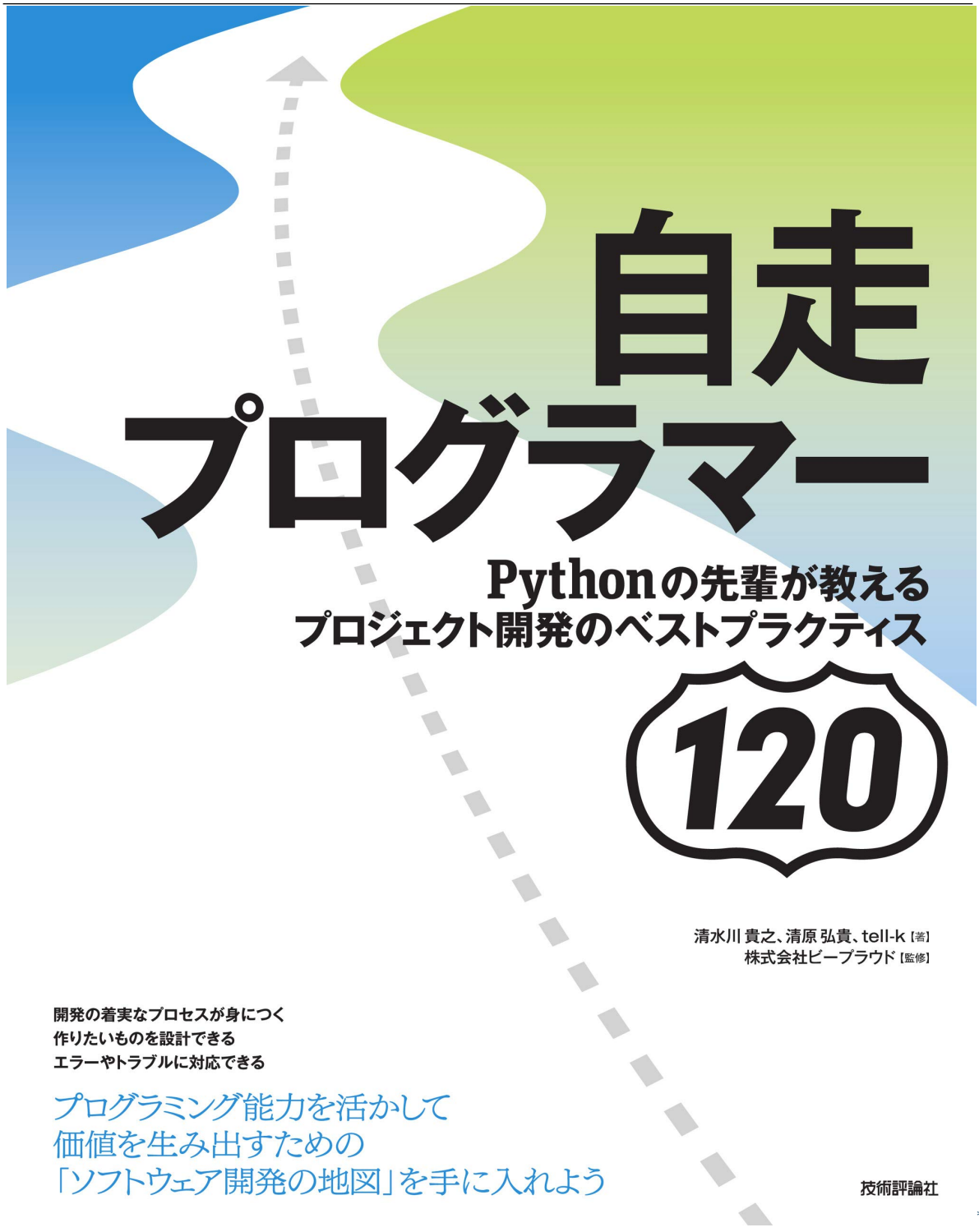
ベストプラクティス

クエリで使いやすいテーブル設計をしましょう。RDB でテーブル設計するときは往々にして正規化をします。しかし、正規化だけに着目してテーブル分割を進めると、パフォーマンスの劣化を伴うことがあります。ほしい結果を得るために、たくさんのテーブルを JOIN して無駄に複雑なクエリを作り出してしまうからです。

具体的な失敗では、機能的な要件を満たしていましたが将来的なデータ量を考慮した性能の要件は満たせていませんでした。

JOIN によるテーブルの結合は、対象となるテーブルのデータ量が大きくなればなるほど、性能が劣化していきます。同様の結果を得つつ、性能を改善するためには、あえて正規化を崩して冗長にデータを持たせます。

[*121] https://gihyo.jp/book/2020/978-4-297-11197-7



[*122]

[*122] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*123] をご参照ください

[*123] https://gihyo.jp/book/2020/978-4-297-11197-7

3.2. テーブル定義

3.2.1 49:NULL をなるべく避ける

テーブル定義で最も重要になることは、いかに「制約をつけるか」ということです。次のような、「寛容な」設計にしていませんか？

この節では、テーブル定義については Django のモデルで説明します。

具体的な失敗

```
class Product(models.Model):
    name = models.CharField("商品名", max_length=255, null=True, blank=True)

    @property
    def name_display(self):
        if not self.name:
            return "<商品名なし>"
        return name
```

この商品（Product）モデルは商品名がないデータを許容しています。ですが本当に「商品名がない商品」を受け入れる必要があるのでしょうか？

ベストプラクティス

テーブルのカラムをなるべく NULL 可能 にしないようにします。NULL 可能にする前に、本当に必要か、他の方法で解決できないかを立ち止まって考えることが大切です。

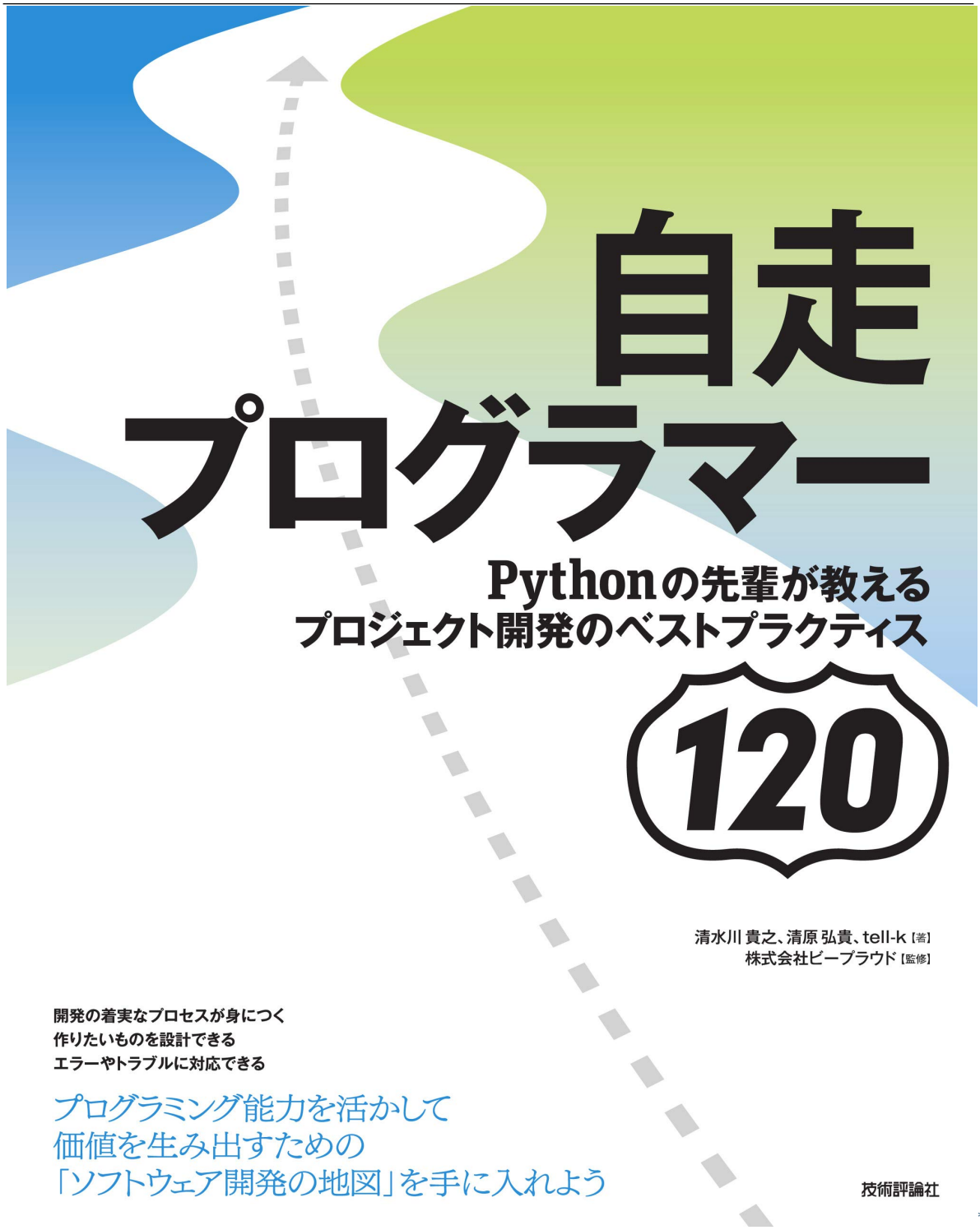
商品名であれば単に「NULL にはできない」という仕様にします。

```
class Product(models.Model):
    name = models.CharField("商品名", max_length=255)
```

NULL を許容するとアプリケーション側で「NULL の場合」を扱う必要が出ます。NULL を扱う処理や仕様が必要になり、プログラムが煩雑になります。制約が少なくなるとアプリケーションで想定するケースが増えるのが問題です。今回の失敗では「Product.name が NULL（None）のとき」を扱う必要があります。

特に「商品名がない商品」という仕様が求められないのであれば、NULL 不可が良いです。「不用意な親切心」で甘い制約のテーブル設計にしないようにしましょう。

とはいえ「何でも NULL 不可にはできない」という場合もあります。デフォルト値を使った NULL 可能の回避方法を紹介します。



*124

*124 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*125] をご参照ください

3.2.2 50:一意制約をつける

「本番環境で想定しないデータが入ってしまい、エラーになったようです」

このような障害報告を聞いたことがある人は、少なくないと思います。その問題点と、解決方法を説明します。もし「まだ聞いたことがない」という方は先に勉強して、将来の問題を回避しましょう。

具体的な失敗

```
class Product(models.Model):
    ...


class Review(models.Model):
    product = models.ForeignKey(Product)
    user = models.ForeignKey(User)
```

このテーブル設計では、1 つの商品に対して同じユーザーが複数のレビューを投稿できてしまいます。1 人のユーザーが評価を上げる（下げる）ために複数投稿できる問題があります。

ベストプラクティス

仕様上、想定しないデータであればできるだけ一意制約をつけておきましょう。

```
class Review(models.Model):
    product = models.ForeignKey(Product)
    user = models.ForeignKey(User)

    class Meta:
        constraints = [
            models.UniqueConstraint(
                fields=["product", "user"],
```

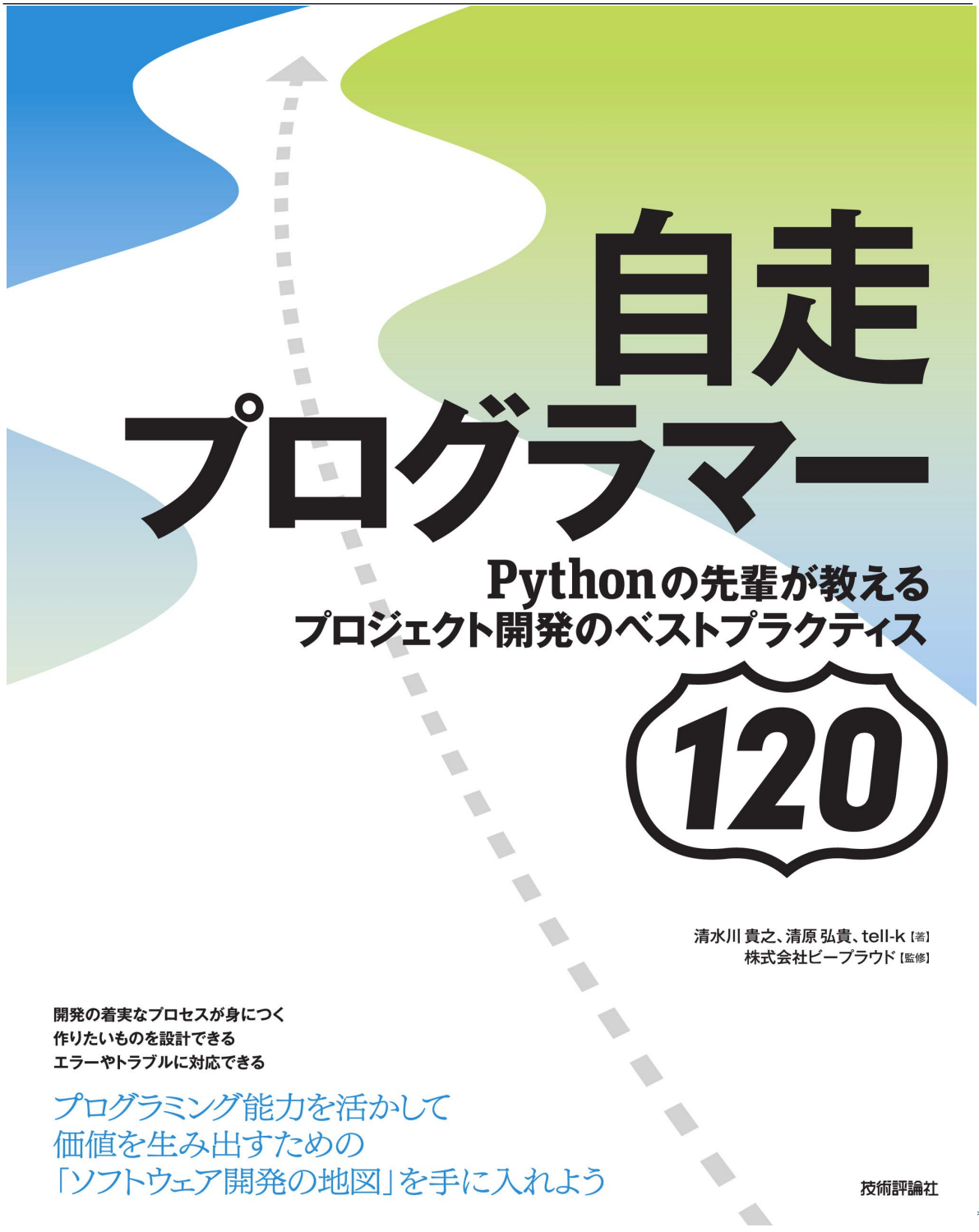
(次のページに続く)

[*125] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
                name="unique_product_review"
            ),
        ]
```

一意制約 があればアプリケーション側で扱う状態を減らせます。



*126

*126 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*127] をご参照ください

3.2.3 51:参照頻度が低いカラムはテーブルを分ける

必要なデータすべてを 1 つのテーブルに押し込めていませんか？テーブルが肥大化する問題と解決方法を説明します。

具体的な失敗

```
class User(models.Model):
    username = models.CharField(...)
    email = models.EmailField(...)
    ...

    enable_notification_release = models.BooleanField(..., help_text="リリースのお知らせを受け取る場合 True")
    enable_notification_security = ...
    enable_notification_mailmagazine = ...
    enable_notification_important = ...
```

参照頻度の低い「リリースのお知らせを受け取るかどうか」という情報を、User というユーザーアカウントを表すテーブルに保持しています。大きな問題ではありませんが、より良いテーブル設計の方法があるはずです。

ベストプラクティス

「通知の設定」に関する情報を、 UserNotificationSettings という別のテーブルに保持させます。

```
class User(models.Model):
    username = models.CharField(...)
    email = models.EmailField(...)
    ...
```

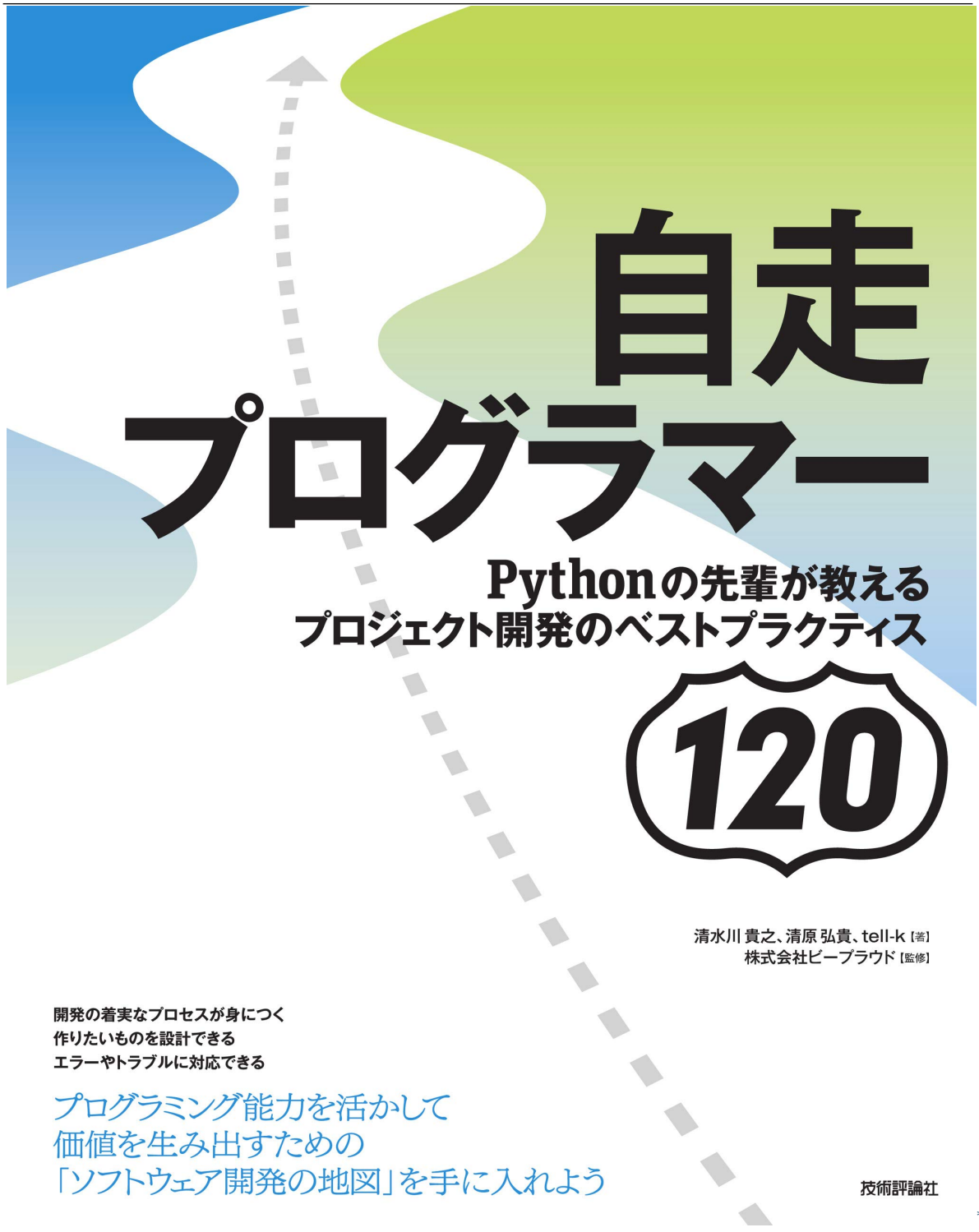
(次のページに続く)

[*127] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
class UserNotificationSettings(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)

    enable_release = models.BooleanField(..., help_text="リリースのお知らせを受け取る場合 True")
    enable_security = ...
    enable_mailmagazine = ...
    enable_important = ...
```

テーブルのカラムが増えると参照や JOIN が遅くなる問題があります。参照したときのデータ転送時に、データ量が多くなり、JOIN する際に、必要な一時テーブルの容量が多くなるためです。



*128

*128 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*129] をご参照ください

3.2.4 52:予備カラムを用意しない

プログラミング迷子: 社内フローのシワ寄せで生まれてしまう予備カラム

- 後輩 W：将来的に、データベースにカラムが必要になるかもしれません。
- 先輩 T：たしかにそうだね。
- 後輩 W：ですので、今のうちに予備用のカラムをいくつか作っておこうと思います。
- 先輩 T：それは良くないよ。あとから追加すれば十分じゃない？
- 後輩 W：社内の運用上、私がデータベースの操作をする権限がないので、先に十分な量を作っておいたほうがいいかなと。
- 先輩 T：必要なときにカラムを足すほうが良いよ。アプリケーションの開発が大変になってしまうよ。

「予備カラム」という言葉が聞こえたら、できる限り避けることを考えましょう。

具体的な失敗

```
class Sale(models.Model):
    product = models.ForeignKey(...)
    bought_by = models.ForeignKey(...)

    yobi_001 = models.CharField("予備 1", max_length=1023)
    yobi_002 = models.CharField("予備 2", max_length=1023)
    yobi_003 = models.CharField("予備 3", max_length=1023)
    yobi_004 = models.CharField("予備 4", max_length=1023)
    yobi_005 = models.CharField("予備 5", max_length=1023)
```

この例では今後のことを考えて yobi_ という予備カラムが 5 つあります。将来的に予備カラムが使われるようになったとして、以下の問題があります。

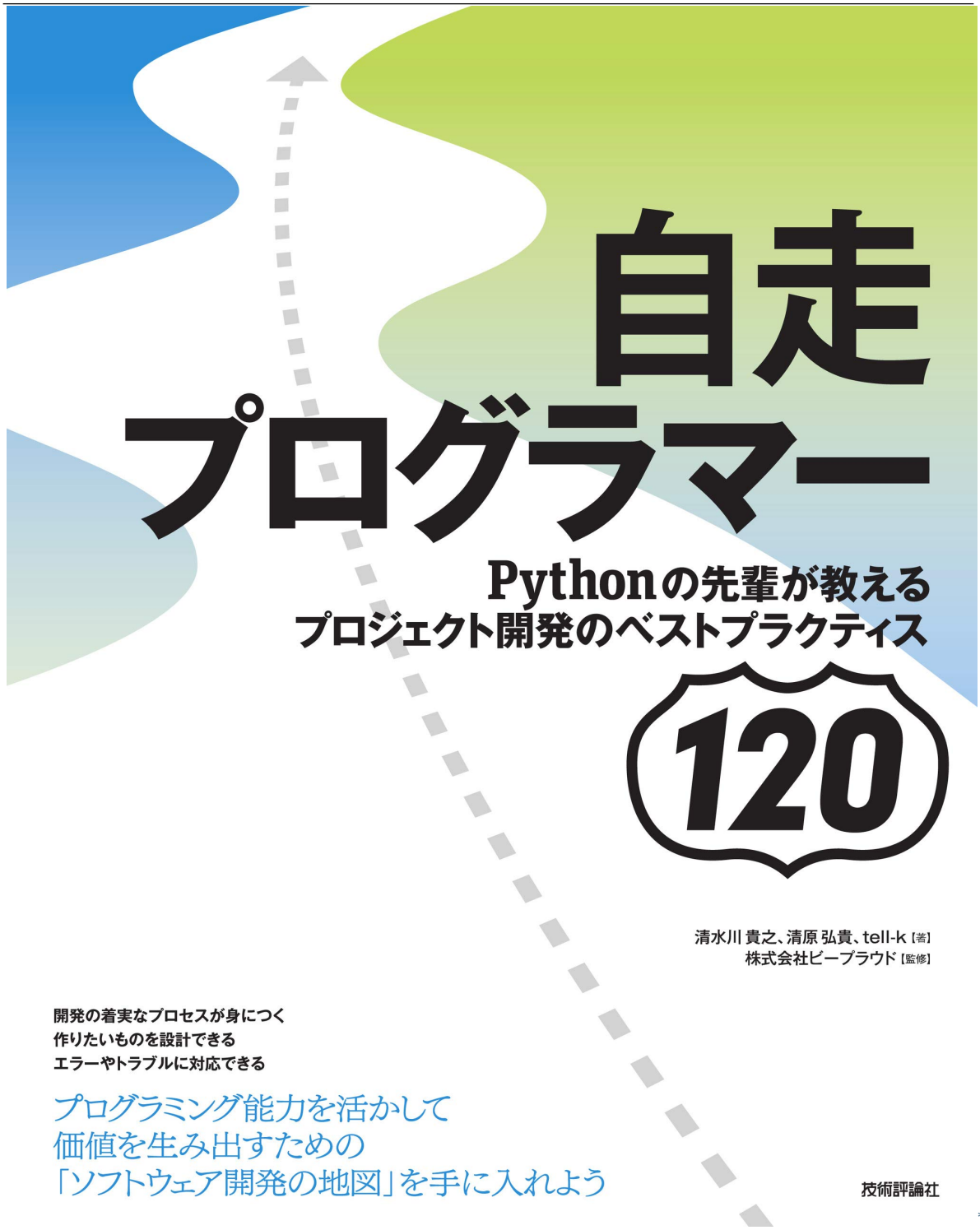
[*129] https://gihyo.jp/book/2020/978-4-297-11197-7

- カラム名が意味を説明できない
  - 「yobi_001 はキャンペーン ID が入っている」と直感的にわからない
- 文字列型など事前に決めた型でしか使えない
  - 文字列型として数値や日付を管理する必要が出る
  - 外部キーを貼れない
- 事前に決めたカラムの大きさで使うしかない

ベストプラクティス

単純に、予備カラムを使わないようにしましょう。

```
class Sale(models.Model):
    product = models.ForeignKey(...)
    bought_by = models.ForeignKey(...)
```



*130

*130 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*131] をご参照ください

3.2.5 53:ブール値でなく日時にする

テーブル設計をするとき、ブール値を多く使いがちになります。ですがブール値でなく、日時を使うことでより良い設計にできる場合があります。

具体的な失敗

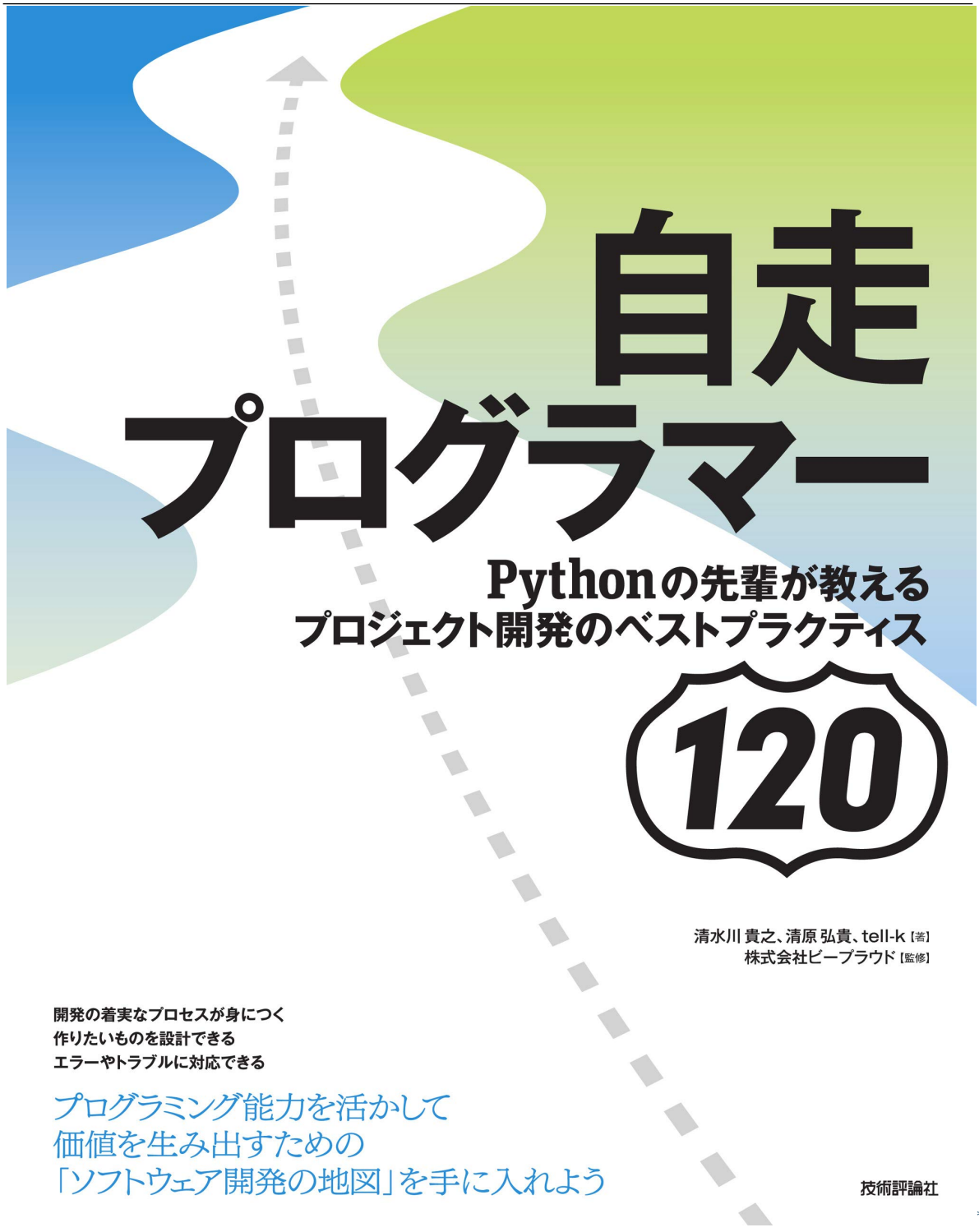
```
class Article(models.Model):
    published = models.BooleanField("公開済みフラグ", default=False)
    published_at = models.DateTimeField("公開日時", default=None, null=True,␣
↪blank=True)
```

この記事（Article）テーブルには、 published というブール値のカラムがあります。published というカラムを用意しなくても、 published_at というカラムを使えば、公開されたかどうかは判定できます。カラムも 1 つ減らせるので、 published_at のみを用意するのが良いでしょう。

ベストプラクティス

「公開済み」など、公開日時をフラグとして使えるデータであれば、ブール値を別途用意する必要はありません。NULL の場合は「非公開」であり、データがある場合を「公開済み」と扱います。

[*131] https://gihyo.jp/book/2020/978-4-297-11197-7



*132

*132 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*133] をご参照ください

3.2.6 54:データはなるべく物理削除をする

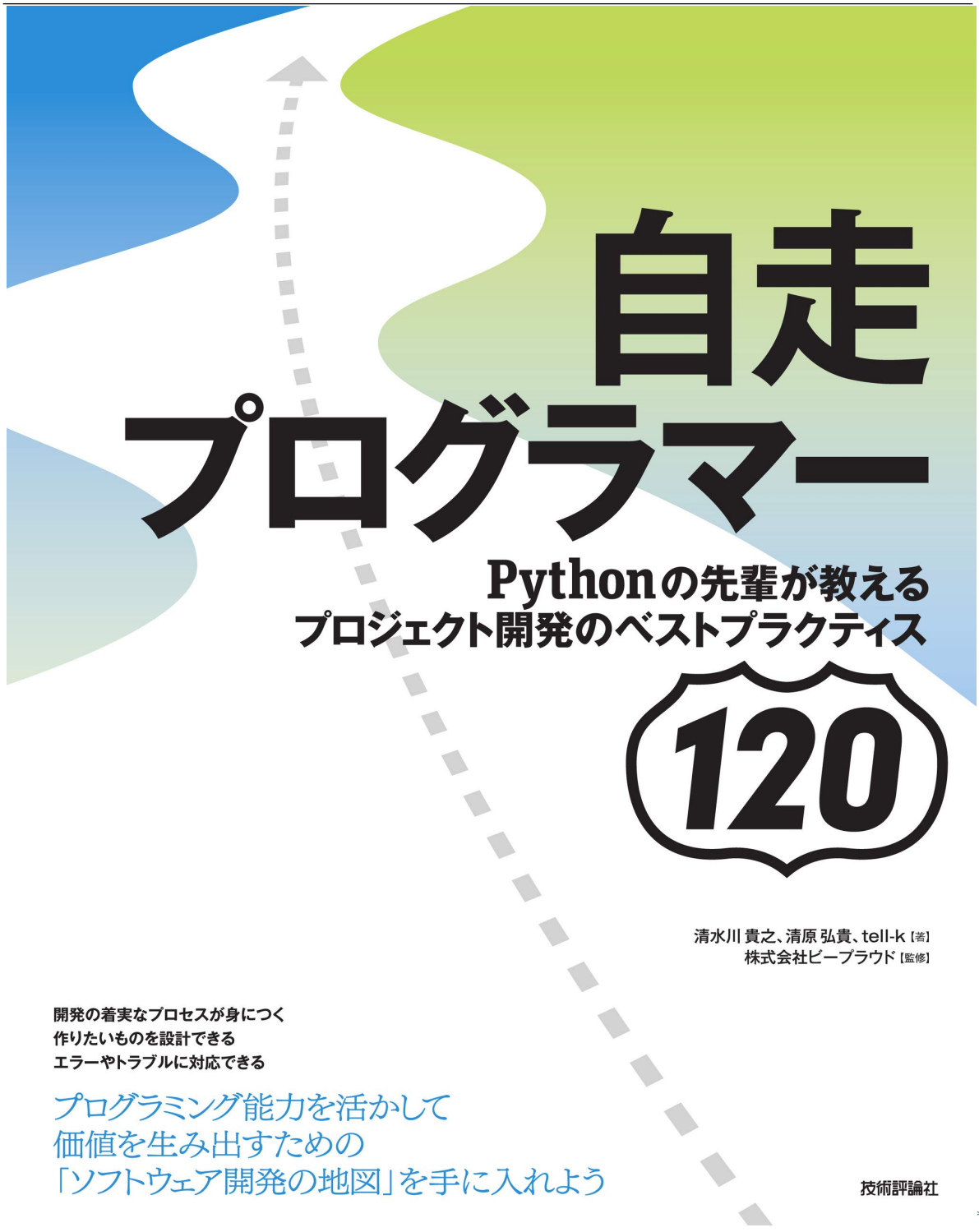
「論理削除 をしたい」という要望はとても多くあると思います。ですが実際には将来的な開発コストや運用コストが大きくなりますので、安易に導入しないほうが良いでしょう。なぜでしょうか？

具体的な失敗

```
class ArticleQuerySet(models.QuerySet):
    def exclude_deleted(self):
        return self.filter(deleted_at__isnull=True)


class Article(models.Model):
    ...
    deleted_at = models.DateTimeField(null=True, blank=True)

    objects = ArticleQuerySet.as_manager()
```

このテーブル設計では、 deleted_at というカラムが設定されていれば「削除された」と扱うようにしています。論理削除はプログラム上扱う状態が増えるのでオススメしません。すべてのデータ取得に「削除済みでない」という条件が必要になります。 JOIN をする際にも条件が常に必要です。開発の際に常に条件を意識する必要がありますし、誤って実装してしまうと大きな問題になります。

[*133] https://gihyo.jp/book/2020/978-4-297-11197-7



[*134]

[*134] https://gihyo.jp/book/2020/978-4-297-11197-7

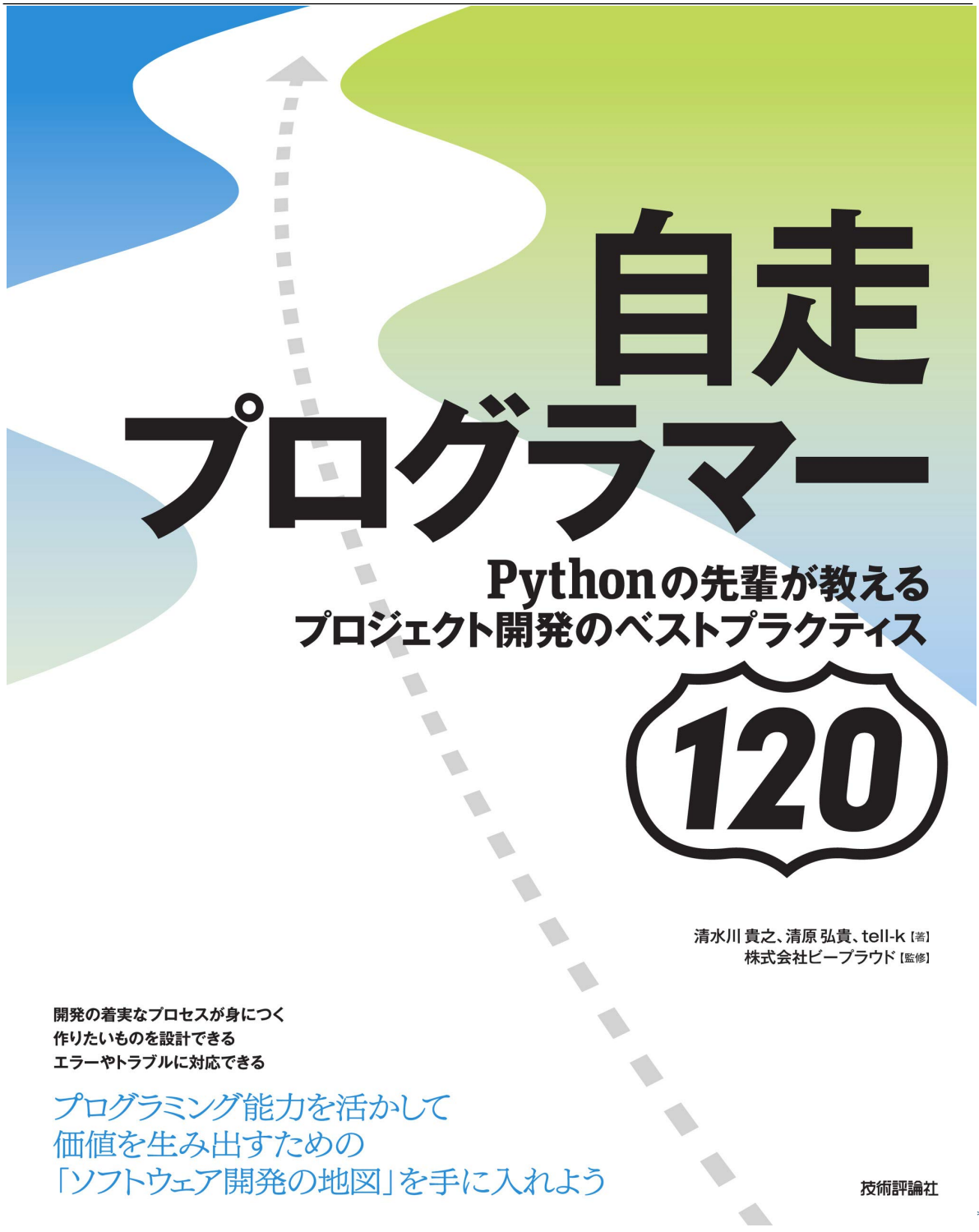
（中略）詳細は書籍 自走プログラマー[*135] をご参照ください

ベストプラクティス

論理削除をしないのが一番です。ほとんどの要望、要件に対して論理削除が必要になることは非常に少ないでしょう。

「論理削除がほしい」という要望の背景としては「データを戻せるようにしたい」や「過去のデータを参照したい」が多いかと思います。その場合、以下のような別の方法で解決できます。

[*135] https://gihyo.jp/book/2020/978-4-297-11197-7



*136

*136 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*137] をご参照ください

3.2.7 55:type カラムを神格化しない

type というカラムも無思慮に作成されがちです。少し複雑な仕様の場合に、うまくやろうとして、失敗してしまう場合が多くあります。

具体的な失敗

ある EC サイトでは商品に対して「コメント」と「レビュー」が残せるようになっているとします。コメントとレビューはそれぞれ「投稿者」「タイトル」「本文」があり、レビューには 5 段階で商品の良し悪しを評価できます。

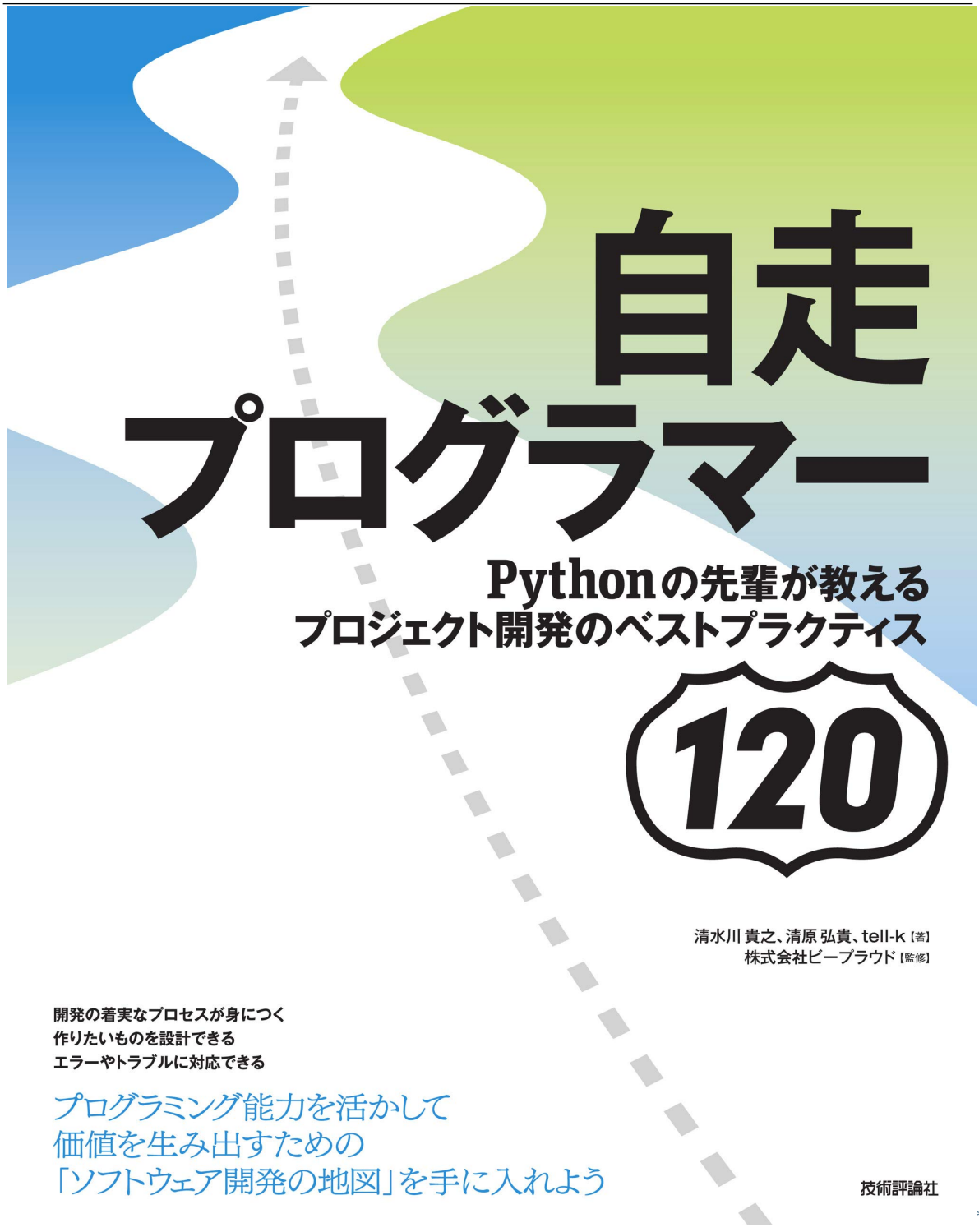
- ユーザーは投稿する際に「レビュー」にするか「コメント」にするかを選べる
- レビューは集計することで平均の評価数を表示する
- レビューとコメントは 1 つの画面でまとめて見られるが、別々のものとしても表示できるようにする

この場合、以下のようなモデル設計にしてはいけません。

```
class Comment(models.Model):
    TYPE_COMMENT = 0
    TYPE_REVIEW = 1
    TYPE_CHOICES = (
        (TYPE_COMMENT, "コメント"),
        (TYPE_REVIEW, "レビュー"),
    )
    posted_by = models.ForeignKey(User)
    title = models.CharField(...)
    body = models.TextField(...)

    type = models.PositiveSmallIntegerField(choices=TYPE_CHOICES, ...)
    star = models.PositiveSmallInteger(null=True, blank=True)
```

type によって挙動が大きく変わるのが問題です。データの内容としては似ているものですが、概念として別のものなので別と扱ったほうが良いです。

[*137] https://gihyo.jp/book/2020/978-4-297-11197-7



[*138]

[*138] https://gihyo.jp/book/2020/978-4-297-11197-7

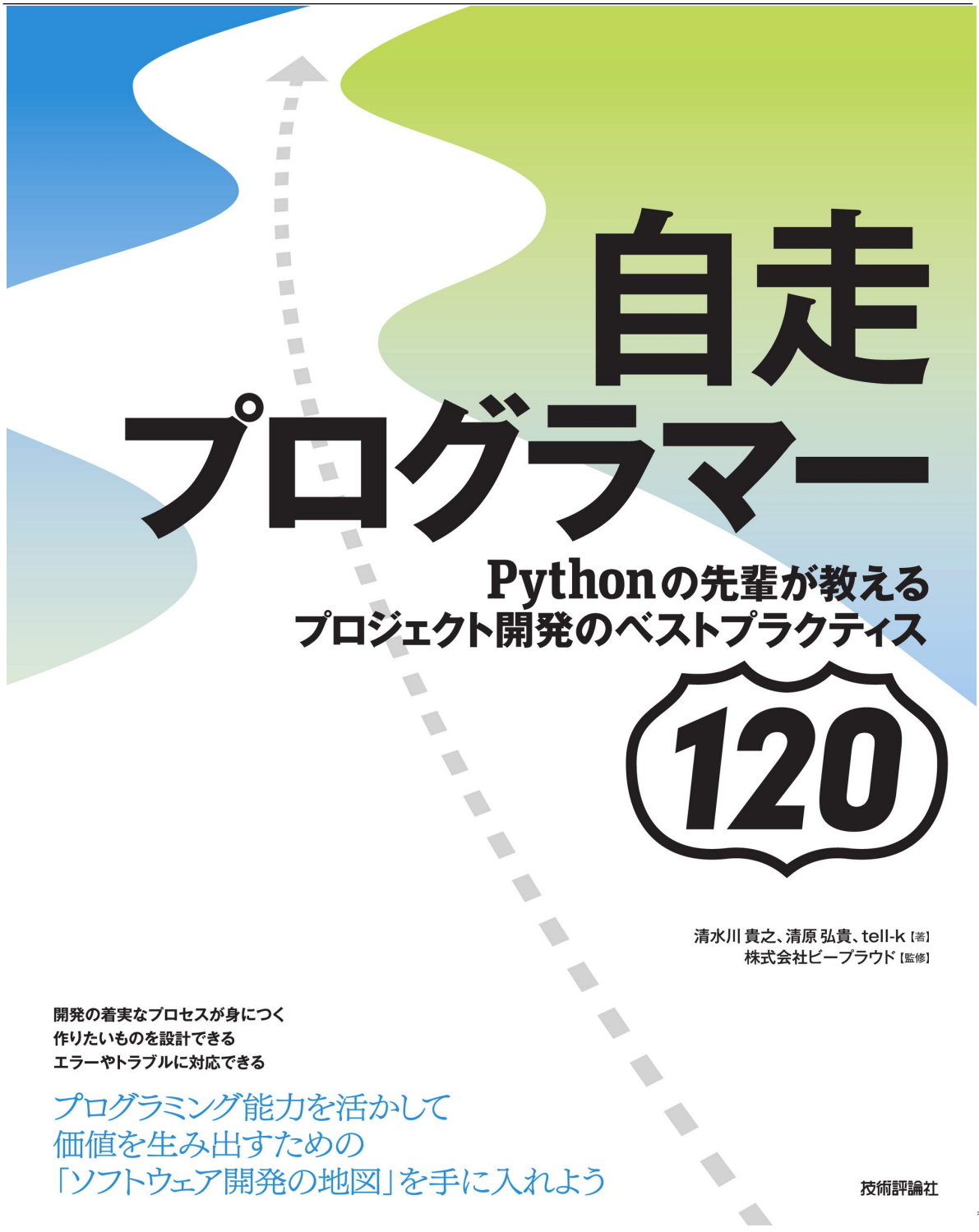
（中略）詳細は書籍 自走プログラマー[*139] をご参照ください

ベストプラクティス

単純にテーブルを分けるのが良いでしょう。

```
class Comment(models.Model):
    posted_by = models.ForeignKey(User)
    title = models.CharField(...)
    body = models.TextField(...)


class Review(models.Model):
    posted_by = models.ForeignKey(User)
    title = models.CharField(...)
    body = models.TextField(...)
    star = models.PositiveSmallInteger()
```

[*139] https://gihyo.jp/book/2020/978-4-297-11197-7



*140

*140 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*141] をご参照ください

3.2.8 56:有意コードをなるべく定義しない

仕様的に必要とされる「 有意コード 」にも罠が潜んでいます。この問題と解決方法を見ていきましょう。

具体的な失敗

ここでは有意コードとは「商品コード」のような一意な値を決めるときに「1 桁目は商品の区分、それ以降が商品ごとの数値」のようにコードの桁数によって複数の意味を持たせることを言います。たとえば次のようなものです。

- FD10001： FD が商品の区分、10001 が商品の番号
- A2019101： A が記事のカテゴリー、201910 が作成の年と月、1 が記事ごとの番号

商品（Item）のカラムとして「商品コード」という値が必要とします。

```
class Item(models.Model):
    code = models.CharField("商品コード", max_length=16, unique=True,
                            help_text="1 桁目が商品区分、2〜7 桁目が登録日、残りが一意な番
号")
```

ここで、以下のように有意コードに依存したプログラムを書いてはいけません。

```
Item.objects.filter(code__startswith="A")  # 商品区分が A（家電）の商品を取り出し
Item.objects.filter(code__contains="201105")  # 20 年 11 月 5 日に登録された商品の取り出し
```

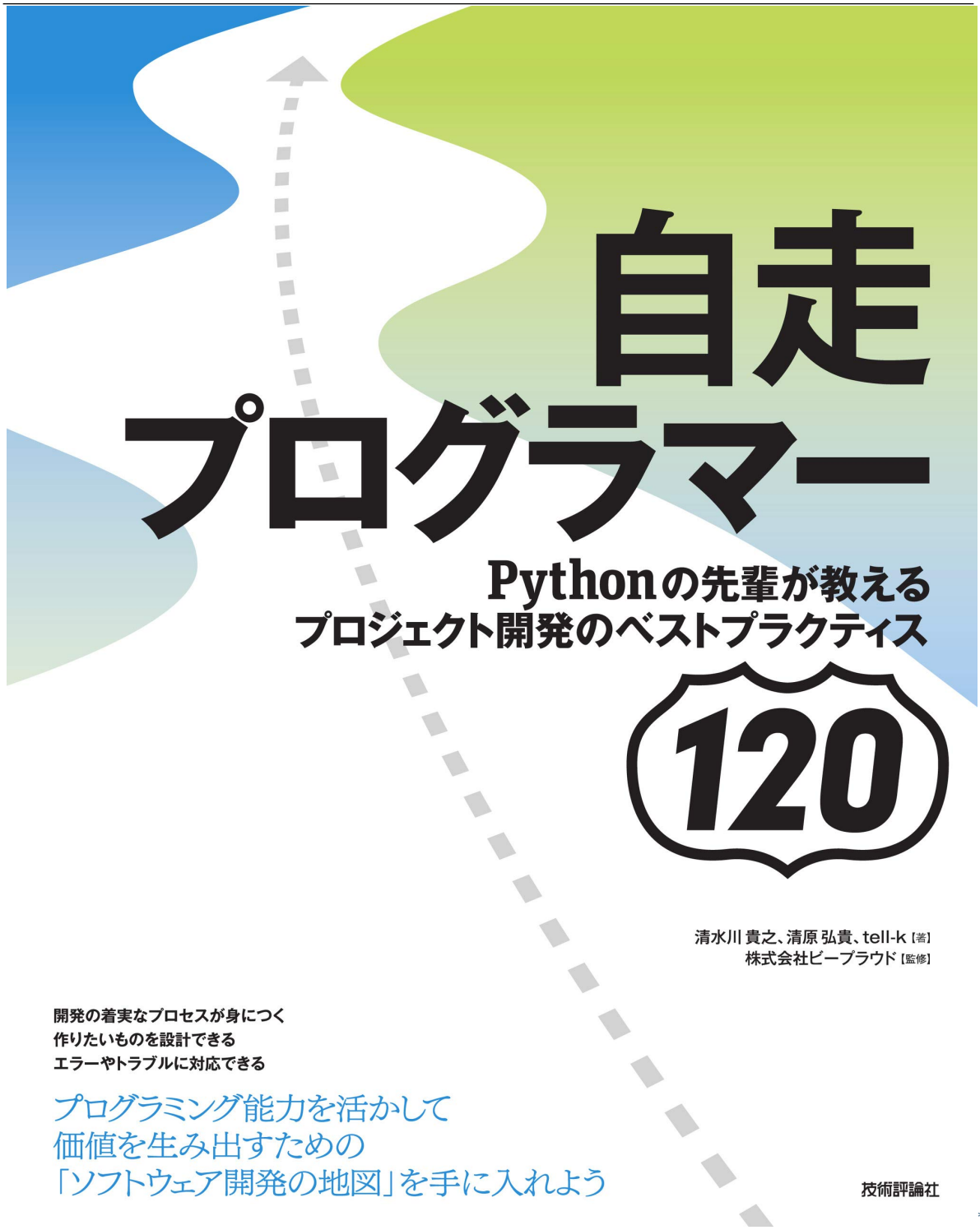
有意コードの「桁の意味」を使って検索すると LIKE 検索になるので遅いのが問題です。INDEX が効かなくなる場合もあるので、プログラムしないよう気をつけましょう。有意コードには外部キー制約が使えないので、「商品区分から商品一覧を取得する」処理も遅くなります。

また単純に、有意コードから値を取り出す処理が頻発してプログラムが汚くなります。商品区分を意味して `item.code[:2]` というプログラムを書かれても、商品コードの仕様を知らない人にはピンとこないでしょう。

[*141] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

アプリケーションの仕様上必要ないのであれば有意コードを定義しないのが理想です。有意コードが必要な場合は、検索や制約の条件として使わないようにしましょう。商品コードは、他に存在する情報から自動で作られる値にします。あくまでシステム運用上、人間が使うためだけに用意します。



*142

*142 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*143] をご参照ください

3.2.9 57:カラム名を統一する

データベースを設計したらカラム名がバラバラ、ということはないでしょうか？小さな範囲でもルールを決めておくことで、開発時にタイプミスや勘違いを減らせます。

具体的な失敗

```
class Item(models.Model):
    name = models.CharField(...)

    reviewed = models.ForeignKey(User, ...)

    item_kbn = models.PositiveSmallIntegerField(...)
    delivery_type = models.PositiveSmallIntegerField(...)

    publish_dt = models.DateTimeField(...)
    created_at = models.DateTimeField(...)
```

このコードには以下のような問題があります。

- reviewed が外部キーかブール値かわかりにくい
- _type と _kbn でブレている
- _dt と _at でブレている

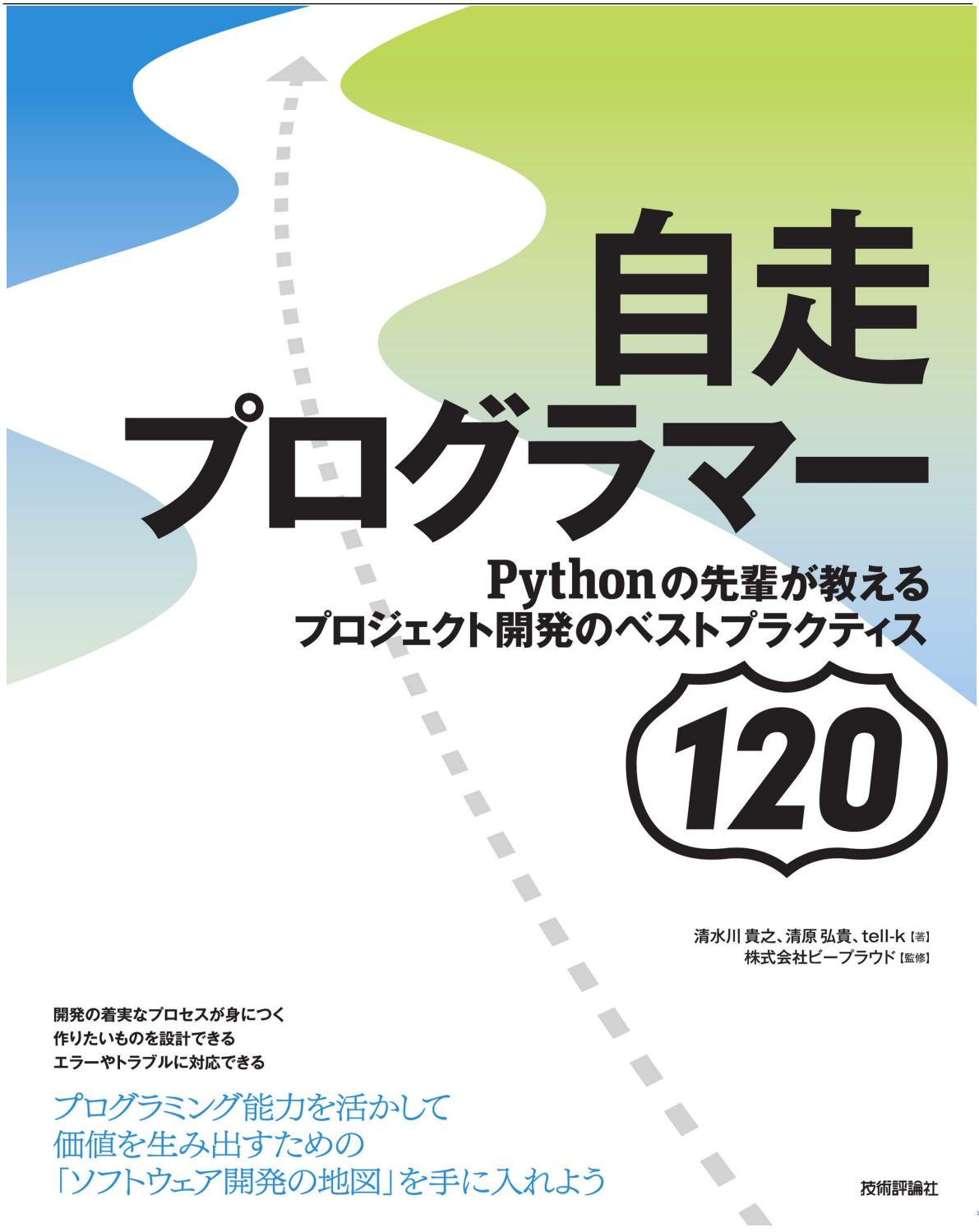
1 つのテーブル内などで表記がブレていると、同じ型のものを類推しにくくなります。また、この場合、たとえば Item.publish_at とタイプミスする確率が上がります。

[*143] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

カラムの型によってある程度揃えたほうが良いでしょう。

```
class Item(models.Model):
    name = models.CharField(...)

    reviewer = models.ForeignKey(User, ...)

    item_type = models.PositiveSmallIntegerField(...)
    delivery_type = models.PositiveSmallIntegerField(...)

    published_at = models.DateTimeField(...)
    created_at = models.DateTimeField(...)
```



*144

*144 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*145] をご参照ください

[*145] https://gihyo.jp/book/2020/978-4-297-11197-7

3.3. Django ORM との付き合い方

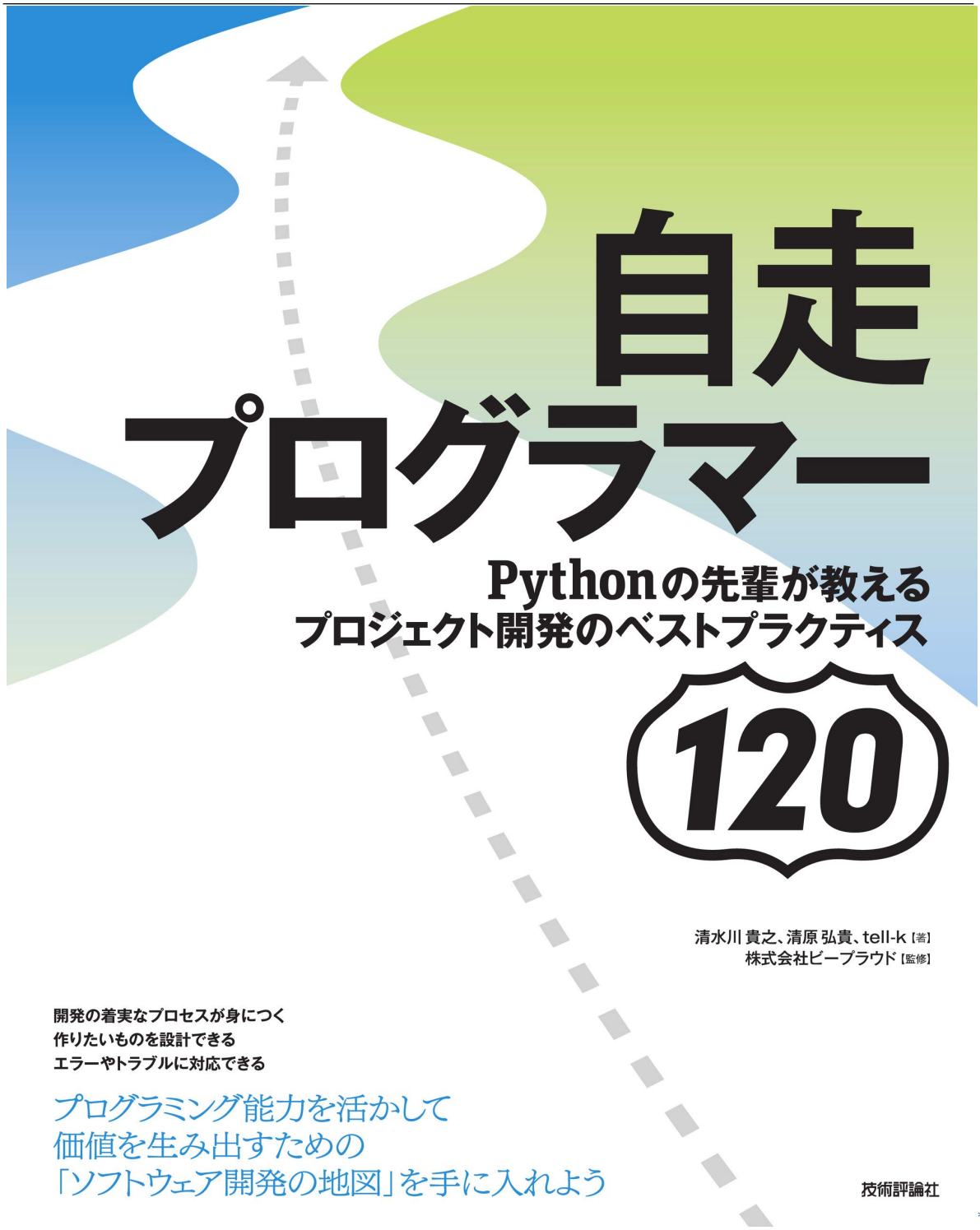
3.3.1 58:DB のスキーママイグレーションとデータマイグレーションを分ける

プログラミング迷子**:** トラブルに弱いマイグレーション実装

- 後輩 W：Django でモデルにフィールドを追加したので マイグレーション したところ、自分の環境と CI では問題なかったんですが、開発サーバーで実行したら途中でエラーになって直せなくなってしまいました。こういう場合、テーブルを直接直したりしていいんでしょうか？
- 先輩 T：途中でエラーっていうと、どんなエラー？
- 後輩 W：追加しようとしたカラムが NULL 不可で、データがある場合にデフォルト値がないせいでエラーになりました。
- 先輩 T：Django の migrate コマンドならバージョン指定することで ロールバック できると思うけど、やってみた？
- 後輩 W：それが、データを移動する処理も同じ migration コードに書いていて、ロールバックしようとするともう必要なカラムがなくてエラーになります。
- 先輩 T：なるほど、 スキーママイグレーション と データマイグレーション を 1 回でやろうとしたのか。MySQL ではスキーマ変更に トランザクション が効かないから、エラーが起きた時点の状態で確定されちゃうんだよね。だからテーブルを直接直すしかなさそう。次からはデータマイグレーションを別のバージョンに分けると良いね。

Django の ORM（Object-Relational-Mapping）はマイグレーション機能も提供しています[*146] 。Django ORM のモデルに定義したフィールドの移動は、実際にはフィールドの削除と新規追加として扱われます。このような変更に対するマイグレーションファイルは、1 つのマイグレーションでカラムの追加と削除を行います。

[*146] 『Python プロフェッショナルプログラミング第 3 版』（ビープラウド著、秀和システム刊、2018 年 6 月）の 14 章で Django のマイグレーション機能について紹介しています。



*147

*147 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*148] をご参照ください

上記のマイグレーションでも基本的には問題ありませんが、もしエラーが発生したら困ることになります。このマイグレーションを実行したとき、「カラム追加」が成功したあと「データマイグレーション」や「カラム削除」で失敗する可能性があります。原因は、データマイグレーションコードの考慮不足かもしれませんし、カラム削除がローカル環境の SQLite でうまくいっても本番環境の MySQL や PostgreSQL でうまくいかないケースなのかもしれません。

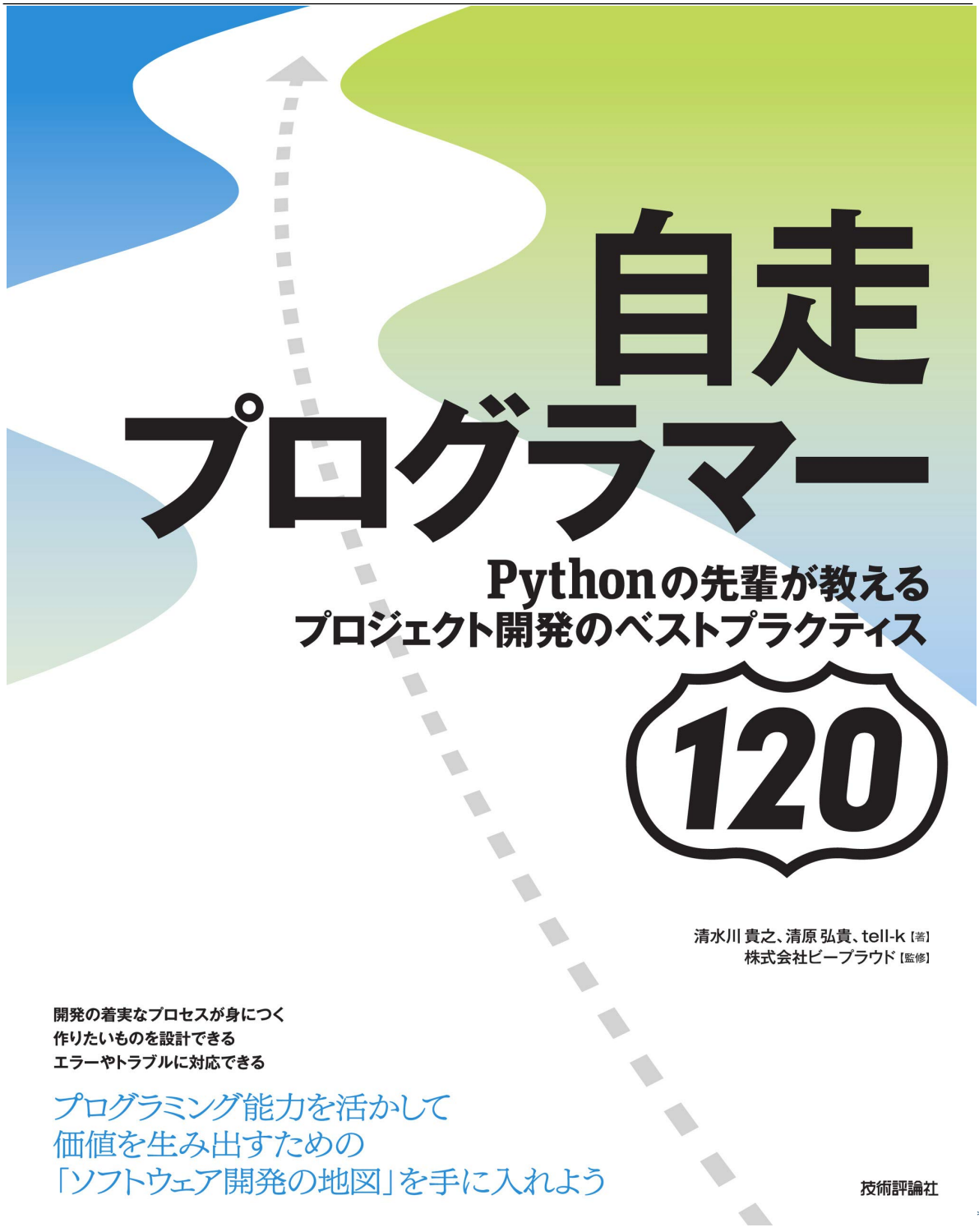
このような失敗が起こると、再実行も ロールバック もできなくなってしまいます。マイグレーションの再実行は、すでに追加済みのカラムをさらに追加しようとして失敗します。ロールバックは、最後の「カラム削除」をロールバックとして「カラム追加」しようとしますが、実際にはカラムはまだ削除されていないため、存在するカラムをさらに追加しようとして失敗します。

ベストプラクティス

スキーママイグレーションとデータマイグレーションは個別に実行できるように用意しましょう。

モデルのフィールドを別のモデルに移動する場合は、3 回のマイグレーションに分けます。

[*148] https://gihyo.jp/book/2020/978-4-297-11197-7



*149

*149 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*150] をご参照ください

関連

- *59:*データマイグレーションはロールバックも実装する (ページ 215)

3.3.2 59:データマイグレーションはロールバックも実装する

プログラミング迷子**:** ロールバックする予定がないからロールバックを実装しなくて良い？

- 先輩 T：この データマイグレーション 、 ロールバック 処理が実装されてないけど、絶対にロールバックしない想定？
- 後輩 W：はい、ロールバックしないです。本番リリース後にこのデータマイグレーションをロールバックするとマズイので。
- 先輩 T：データマイグレーションのロールバックがあれば、実装中に進めたり戻したりして試行錯誤できるのでオススメだよ。そういった試行錯誤で見つかるバグや考慮漏れも見つかるので超オススメです。
- 後輩 W：本番でロールバックしないなら不要と思ってました。ちょっと実装方法を勉強してきます。

データマイグレーションのロールバックを実装するかどうかは、本番環境でロールバックを実行する予定があるかどうかで決めるものではありません。本番環境でマイグレーションをロールバックするということは、本番環境へのリリースで何らかの障害が発生した、ということです。障害が発生してしまったのに本番環境のデータを元に戻せない、という状況は避けるべきでしょう。

データベースマイグレーション機能を持つ Django などの多くのフレームワークでは、スキーママイグレーションのロールバック機能を提供しています。これに対してデータマイグレーションは、正しいデータの状態を人間がプログラムする必要があるため、自動では用意されません。スキーマと同様に、データもロールバックできるように実装しておけば、何かあった場合の最終手段として利用できます。

もし、データマイグレーションのロールバックが用意されていなかったり、どうしてもロールバック処理を実装できないマイグレーションの場合、本番環境への適用はかなり慎重に行う必要があるでしょう。

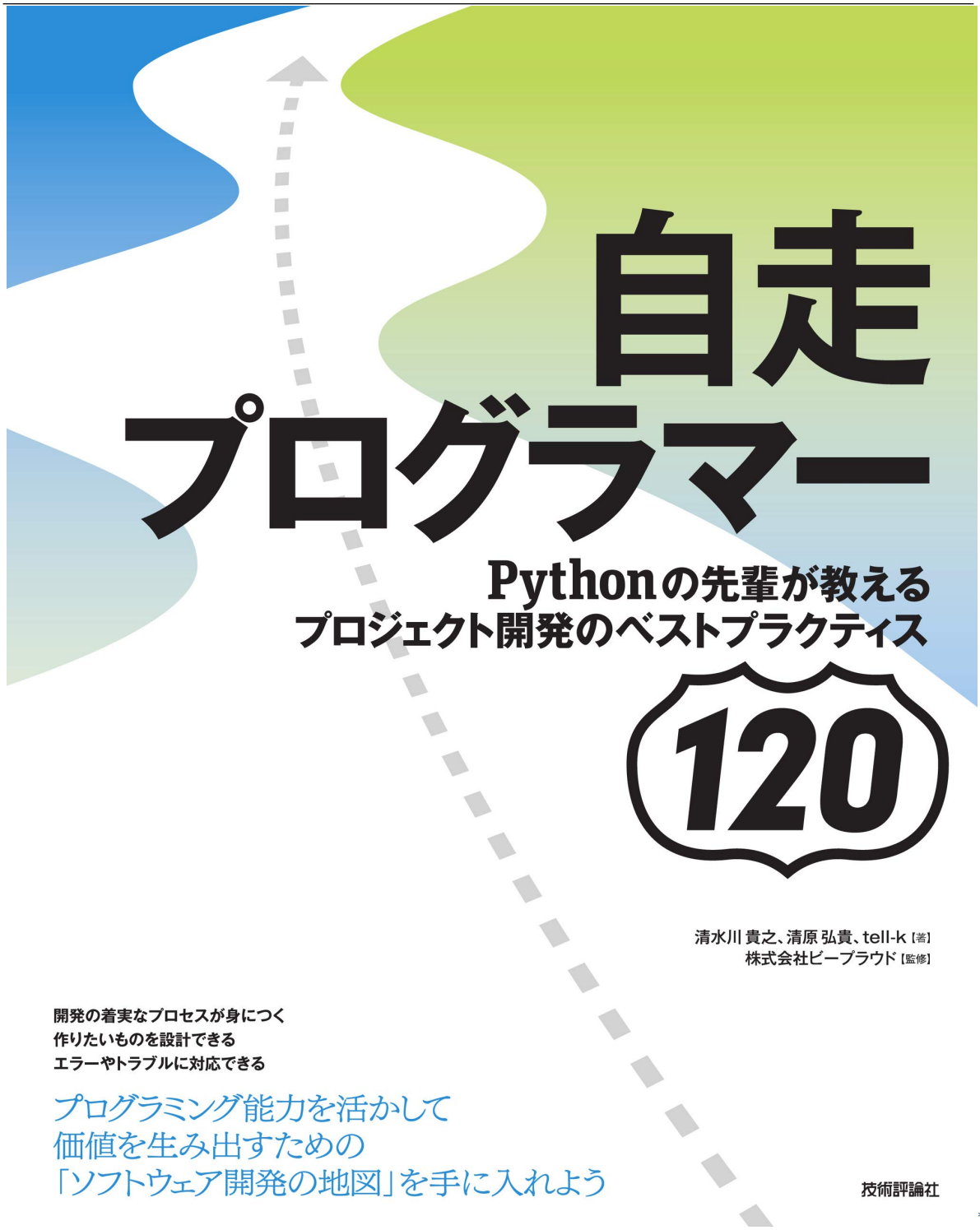
[*150] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

データマイグレーションはロールバックも実装し、動作を確認しましょう。

データマイグレーションのロールバック処理の実装は、本番適用時のトラブルに対する備えであると同時に、データマイグレーションに対するユニットテストでもあります。ロールバック処理を書くことでデータマイグレーションに対する理解が深まり、事前に問題に気づく機会を得られます。また、適用とロールバックを繰り返しながらデータの整合性に問題がないか、確認を繰り返し行えるようになります。

58:DB **のスキーママイグレーションとデータマイグレーションを分ける** (ページ 211) で実装したデータマイグレーションに、ロールバック処理を実装してみましょう。以下のコードにある `reverse_address_data` がロールバック処理です。



[*151]

[*151] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*152] をご参照ください

3.3.3 60:Django ORM でどんな SQL が発行されているか気にしよう

プログラミング迷子**: ORM** を使えば **SQL** を知らなくても良い？

- 後輩 W：新しい機能を実装したらレスポンスがすごい遅い……。
- 先輩 T：どんな SQL が発行されてるか確認してみた？
- 後輩 W：どうやったらわかるんですか？
- 先輩 T： Django Debug Toolbar を使うといいよ。あるいは settings.LOGGING にこんな設定を書いて、DB の SQL 発行をログ出力しよう。

```
LOGGING = {
    'version': 1,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django.db.backends': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': False
        },
    },
}
```

- 後輩 W：ブラウザでアクセスしたら何十行も SQL が出てきました。
- 先輩 T：それは SQL を発行しすぎみたいだね。SQL を発行している実装コードを確認してみよう。
- 後輩 W：これは何か問題があるんですか？

[*152] https://gihyo.jp/book/2020/978-4-297-11197-7

- 先輩 T：そうだね、SQL 発行ごとにデータベースと通信してデータをやりとりするので、意図せず SQL 発行が多くなってるのは問題があるよ。こういうのを **N + 1 問題** って言うんだ。
- 後輩 W：そうなんですね……。そういうのは ORM でうまくやってくれるんだと思ってました……。

残念ながら、ORM は「SQL を知らなくても使える便利な仕組み」ではありません。簡単なクエリであれば SQL を確認する必要はなく、多くの要件は簡単なクエリの発行で済むかもしれません。だからといって、ORM がどんな SQL を発行しているか気にしないままでいると、落とし穴にはまってしまいます。

ORM を使ってクエリを作成していると、どんな SQL が発行されているか見えづらくなります。Python の辞書データを使う感覚で DB へのクエリを発行すると、同じ SQL が何度も発行されたり、Python プログラムとデータベースとの間でデータが往復していたりして、その分アプリは遅くなっていきます。このような問題は ORM で大量のデータを扱ったことがない場合に発生します。

具体的な失敗

1. データベースに格納されているマスターデータ（本のジャンルや企業の営業所名）などの、めったに変更されないけれどよく参照するデータを 1 リクエスト中に何度も取得している
2. SELECT で数十万件の ID をデータベースから取得して、それを少し加工してから次の SQL に渡している
3. 期待するデータを得ようと ORM で複雑なコードを書いた結果、複雑な SQL が組み立てられてしまい、DB での処理コストが非常に高い
4. SELECT で数件の ID を取得して、プログラム側のループ処理で ID それぞれについて別のテーブルから該当するデータを取得しており、件数に比例してクエリ実行回数が増加する

1 番目の問題は、たとえばログ出力に以下のような SQL 発行が短時間のうちに繰り返されている状態です。

```
DEBUG [2019-12-13 03:23:56,373] django.db.backends (0.000) SELECT "genre"."id",
↪"genre"."name", "genre"."created_at" FROM "genre";
...
DEBUG [2019-12-13 03:23:56,374] django.db.backends (0.000) SELECT "genre"."id",
↪"genre"."name", "genre"."created_at" FROM "genre";
...
DEBUG [2019-12-13 03:23:56,375] django.db.backends (0.000) SELECT "genre"."id",
↪"genre"."name", "genre"."created_at" FROM "genre";
...
```

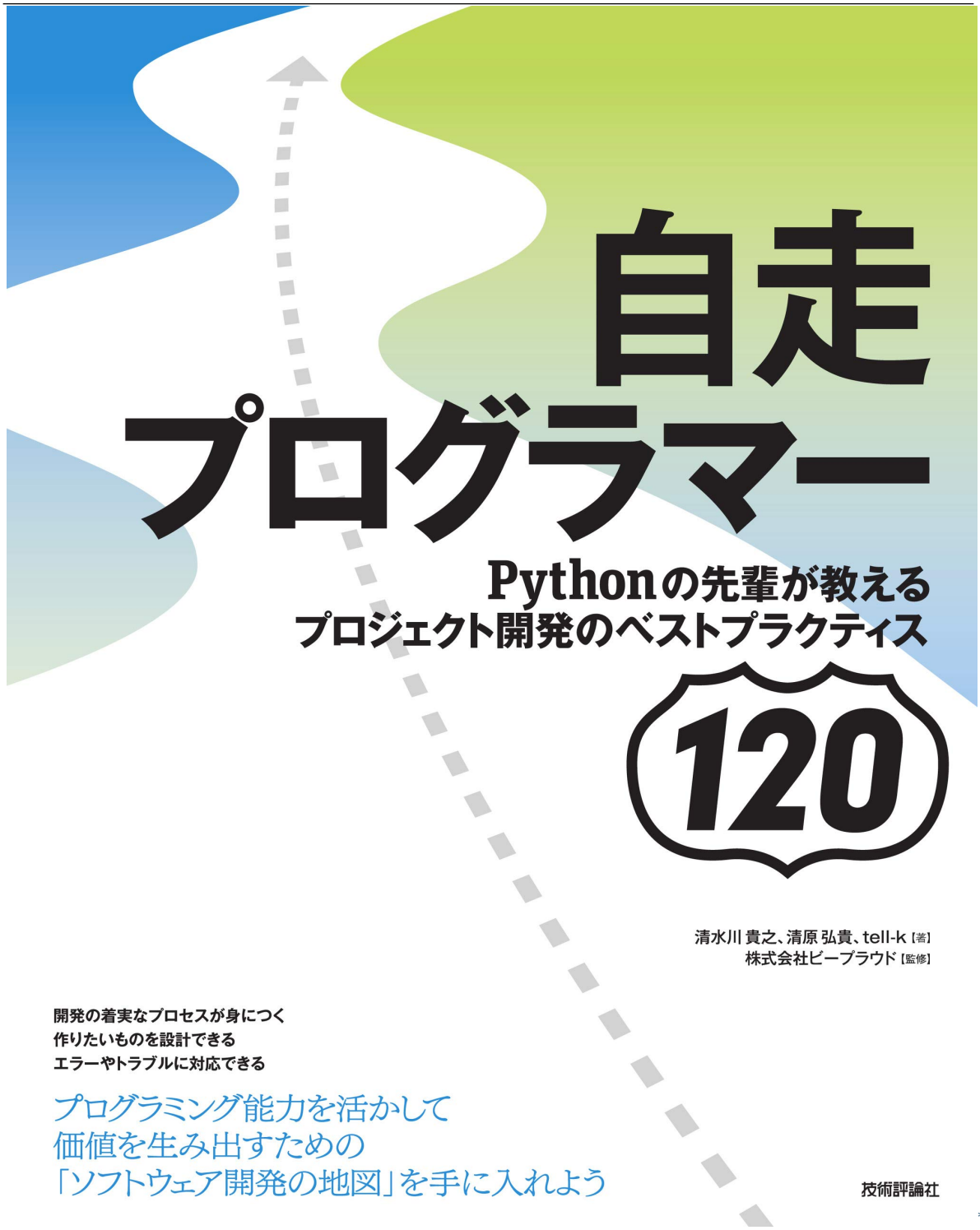
2 番目と 3 番目は *62:SQL* **から逆算して** *Django ORM* **を組み立てる** (ページ 227) で説明します。4 番目は *61:ORM* **の** *N* **＋** *1* **問題を回避しよう** (ページ 222) で説明します。

本項では、そもそも問題に気づくためにはどうすればよいか説明します。

ベストプラクティス

以下のポイントを守りましょう。

- ORM を使ったクエリを新しく書いたら、ORM が生成する SQL を確認する
- 1 回の SELECT で書けるクエリが複数回に分けて実行されていたら、1 つにまとめることを検討する
- 1 つのリクエスト中に何度も同じ SQL が発行されていたら、1 回で済むように修正する



[*153]

[*153] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*154] をご参照ください

関連

- *62:SQL から逆算して Django ORM を組み立てる* (ページ 227)
- *61:ORM の N + 1 問題を回避しよう* (ページ 222)
- *76:シンプルに実装しパフォーマンスを計測して改善しよう* (ページ 283)

3.3.4 61:ORM の N + 1 問題を回避しよう

プログラミング迷子**: N + 1** 問題を回避する **ORM** の書き方は？

- 後輩 W：ログを出して発行される SQL を確認するのはわかったんですが、件数に比例して SELECT がたくさん発行されてしまうのは、どうやって直せば良いんでしょうか？
- 先輩 T：N + 1 問題は、Django の場合、 `select_related` か `prefetch_related` を使えば解決できるよ。
- 後輩 W：それじゃあ、常にそれを使うようにコードを書けば解決するんじゃないですか？
- 先輩 T：いやいや、常に使ってしまうと関連テーブルのデータを全く必要としないときにもデータを取得してデータベースに負荷をかけてしまうことになるよ。

具体的な失敗

プログラムのループ処理で、複数の ID それぞれについてデータベースに SQL を発行すると、件数に比例してクエリ実行回数が増加して、パフォーマンスに影響が出ます。たとえば以下のようなコードです。

```
def process_tasks(ids):
    for pk in my_ids:
        task = Task.objects.get(pk=id)
        ...
```

(次のページに続く)

[*154] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

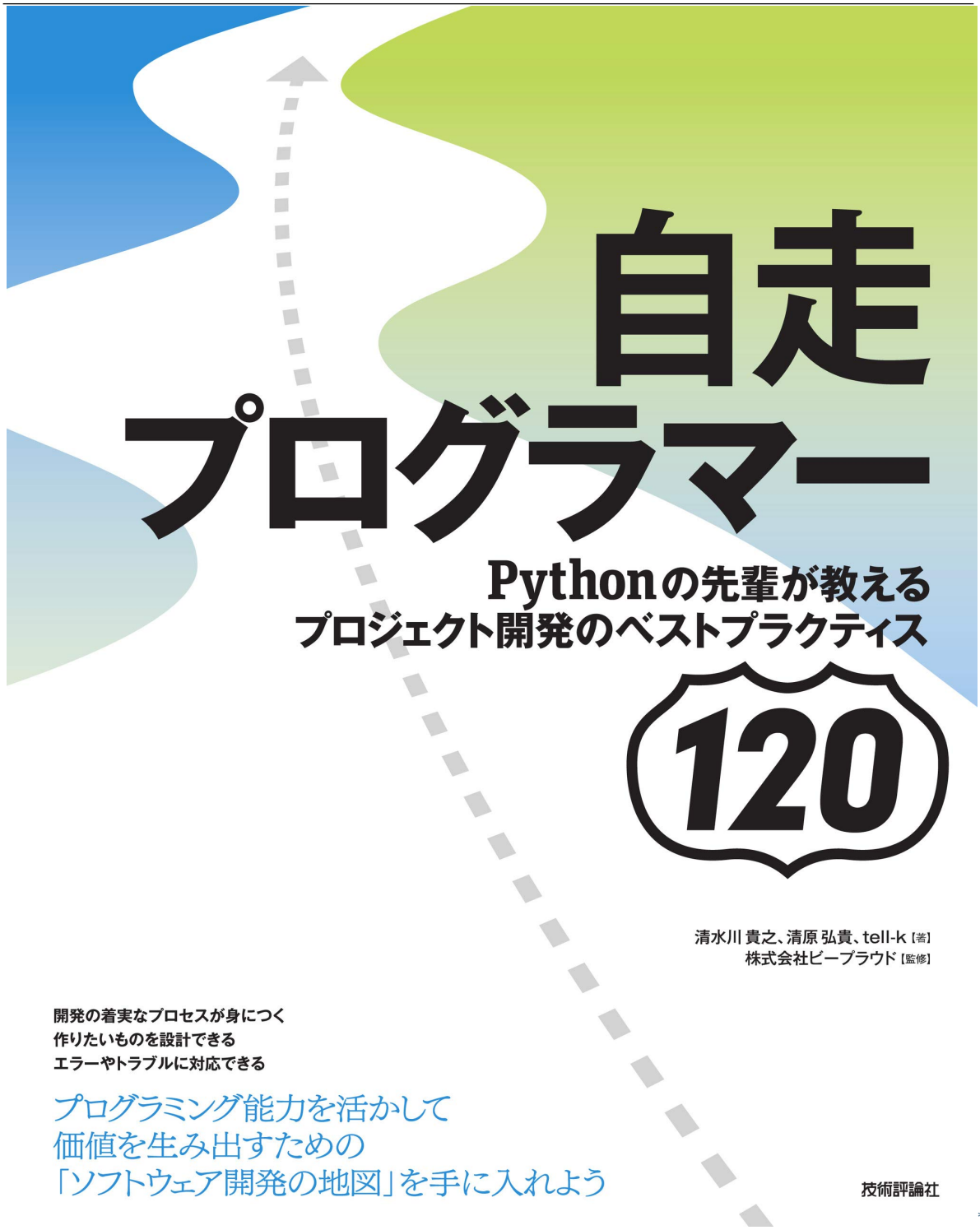
```
my_ids = [1, 2, 3, 4, 5]
process_tasks(my_ids)
```

このようなコードは、コードレビューなどで指摘されて、すぐに修正されるでしょう。では、以下のコードではどうでしょうか。

```
def process_tasks(mail: Mail):
    for attach in mail.mailattach_set.all():
        task = attach.task
        ...

mail = Mail.objects.first()
process_tasks(mail)
```

このコードに登場する、mail, attach, task がどんなオブジェクトなのかは、このコードだけではわかりません。注意深くレビューする人であれば、変数それぞれが何のオブジェクトなのかを調べることで、問題に気づけるかもしれません。

メールに添付された複数のファイルそれぞれからタスク化して業務を進めるシステムの例を考えてみましょう。



[*155]

[*155] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*156] をご参照ください

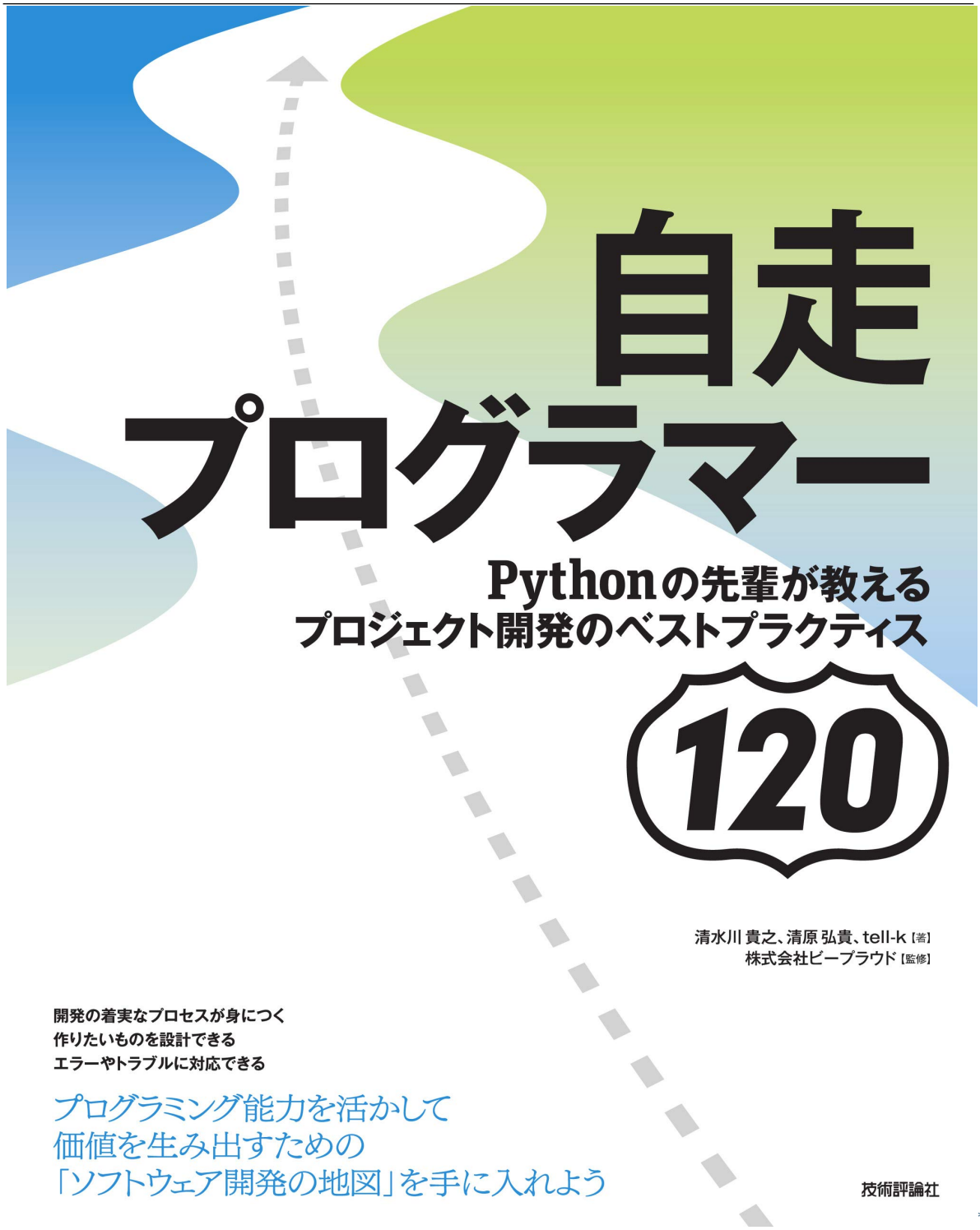
ベストプラクティス

ログを出力して、発行されている SQL を理解しましょう。前述の例のように、ログ出力されていれば、どのような SQL が発行されているかは簡単にわかります。その SQL を読み解いて、それがパフォーマンスに影響を及ぼす SQL だと理解する必要があります。

発行されている SQL を読み解いた後は、Django ORM の知識も必要となります。N ＋ 1 問題を回避する方法は、Django の公式ドキュメントの QuerySet API reference[*157] の `prefetch_related` に記載されています。

ここでは、 `prefetch_related` を使って前述のコードを修正する例を紹介します。

[*156] https://gihyo.jp/book/2020/978-4-297-11197-7
[*157] https://docs.djangoproject.com/ja/2.2/ref/models/querysets/



*158

*158 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*159] をご参照ください

関連

- *60:Django ORM* **でどんな** *SQL* **が発行されているか気にしよう** (ページ 218)
- *76:***シンプルに実装しパフォーマンスを計測して改善しよう** (ページ 283)

3.3.5 62:SQL から逆算して Django ORM を組み立てる

プログラミング迷子**: Django ORM** で組んだ **SQL** のバグが直らない

- 先輩 T：実装中の、タスク一覧に保留コメントを表示する機能が 3 日くらい遅れてるけど、問題は解決しそう？
- 後輩 W：はい、一部うまく表示できない問題が解決できました。ただ、ORM で発行された SQL がテーブルを 2 重に JOIN していて不安なので動作確認中です。
- 先輩 T：ちょっと気になるね。そういうのを残しておくとパフォーマンス悪化や別のバグの原因になったりするので、確認してみるよ。
- 後輩 W：お願いします。
- （1 時間後）
- 先輩 T：ORM 周りのコード、JOIN が 2 重になってるのは直せそうだけど、それ以外にも DB から ID 数千件を取得してからまた DB に渡したり、コメントに「ORM で NULL を取り除けないので Python で除去」って書いてあったりして、だいぶ問題がありそうだね。
- 後輩 W：Django ORM の書き方を変えて試行錯誤したんですけど、1 回のクエリ発行では無理そうだったので、わかりやすい方法にしました。
- 先輩 T：いや全然わかりやすくないってw ちょっとペアプロで一緒に書き直していこうか。

業務系の Web システムを開発していると、プログラミングにかける時間の多くは期待するデータを取得するためにデータベースへのクエリを ORM で実装する時間に充てられます。Web システムがすでに利用中でそこに機能追加を行う場合、すでに実装されている ORM のクエリに処理を追加してしまいがちですが、

[*159] https://gihyo.jp/book/2020/978-4-297-11197-7

そのような進め方ではなかなか期待どおりの結果は得られないばかりか、パフォーマンス悪化やバグの原因になってしまいます（ *60:Django ORM* **でどんな** *SQL* **が発行されているか気にしよう** (ページ 218) ）。

具体的な失敗

実例を紹介するため、 *61:ORM* **の** *N* **＋** *1* **問題を回避しよう** (ページ 222) で使用したコードを使用します。3 つの Django モデル、タスク（Task）、メールの添付ファイル（MailAttach）、メール（Mail）は、それぞれ外部キー参照しています。タスクには状態 `state` があり、添付ファイルがタスク化されると、未処理、処理中、完了、保留、のいずれかの状態を持ち、途中でキャンセルされると `is_cancelled` が True に設定されます。

ここから、メール一覧画面のためのクエリを実装します。メール一覧では、タスク化されていない添付ファイルを含むメールを表示します。そして、2 つの機能「保留のみ表示の指定」「保留の場合は保留コメントも表示する」を追加実装したこととします。

以下のコードは、既存の ORM 実装に試行錯誤してコードを追加した例です。例示のため、ORM で発行される SQL をコメントで並記しました。

```
from app.models import *
from django.db.models import Q

def get_unprocessed_qs(is_pending_only):
    # まだタスク割当のない MailAttach の ID リストを取得
    # MailAttach から以下の条件に当てはまるものを除外し、振り分けのされていないものの ID のみ取得
    # * タスク割当済み ("保留" 以外の Task と紐付いている)
    # * キャンセルされている ("キャンセル" 状態の Task と紐付いている)
    task_ids = Task.objects.filter(
        ~Q(state=State['保留'])|Q(is_cancelled=True)
    ).values_list('mail_attach', flat=True)
    non_null_ids = filter(None, task_ids)  # NULL を除去
    # 上記条件のタスクに割り当てられていない添付ファイルの ID リストを取得
    non_assigned_mail_attach_ids = MailAttach.objects.exclude(
        id__in=non_null_ids
    ).values_list('pk', flat=True)
    # ここでは以降の SQL 例示のため non_assigned_mail_attach_ids == (1, 3, 5) とする

    # タスク割当されていない MailAttach を 1 つでも持つ Mail を取得
```

(次のページに続く)

(前のページからの続き)

```
    qs = Mail.objects.all()
    qs = qs.filter(mailattach__id__in=non_assigned_mail_attach_ids).distinct()

    # ########## タスクを保留にしたユーザ名 (Task.changed_by) の一覧を取得

    task_changed_by_names = qs.filter(
        mailattach__task__is_cancelled=False,
        mailattach__task__state=State['保留'],
    ).order_by(
        '-mailattach__task__id'
    ).values_list(
        'pk', 'mailattach__task__changed_by'
    )
    # SELECT DISTINCT mail.id, task.changed_by
    # FROM mail
    #     INNER JOIN mail_attach ON (mail.id = mail_attach.mail_id)
    #     INNER JOIN mail_attach T3 ON (mail.id = T3.mail_id)
    #     INNER JOIN task ON (T3.id = task.mail_attach_id)
    # WHERE (
    #     mail_attach.id IN (
    #         SELECT U0.id FROM mail_attach U0 WHERE NOT (U0.id IN (1, 3, 5)))
    #     AND task.is_cancelled = 0
    #     AND task.state = 4
    # )
    # ORDER BY task.id DESC;

    # qs.filter で task の changed_by が None の値を isnull で取り除けないため Python で除去する
    non_null_task_changed_by_names = [x for x in task_changed_by_names if x[1]]

    # ########## タスク未割当のメール一覧を取得

    if is_pending_only:  # 「保留のみ」指定の場合
        qs = qs.filter(
            mailattach__task__is_cancelled=False,
            mailattach__task__state=State['保留']
```

(次のページに続く)

(前のページからの続き)

```
        )
        # SELECT DISTINCT mail.id, mail.addr_from, mail.date
        # FROM mail
        #     INNER JOIN mail_attach ON (mail.id = mail_attach.mail_id)
        #     INNER JOIN mail_attach T3 ON (mail.id = T3.mail_id)
        #     INNER JOIN task ON (T3.id = task.mail_attach_id)
        # WHERE (
        #     mail_attach.id IN (
        #         SELECT U0.id FROM mail_attach U0 WHERE NOT (U0.id IN (1, 3, 5)))
        #     AND task.is_cancelled = 0
        #     AND task.state = 4
        # );

    return qs.order_by('date'), non_null_task_changed_by_names
```

コメント以外のコードは短くシンプルなように見えます。しかしコードをよく読むと、要件どおりに動作する実装かどうかわかりやすく書けている、とは言えません。

最初にデータベースから取得している task_ids は、2 行後で除外に使う ID 群ですが、直前のコメントは逆の意味にも読めます。また task_ids には保留以外のほぼすべての Task.id が格納されるため、サービスの運用期間に比例してデータ量が増え、メモリを圧迫し、Web アプリケーションとデータベース間の通信コストが非常に高い状態です。ORM で発行される SQL を見ても、 mail_attach テーブルが 2 回 JOIN されていてそれが適切な SQL かどうかすぐにはわかりません。

コラム: スパゲッティクエリ

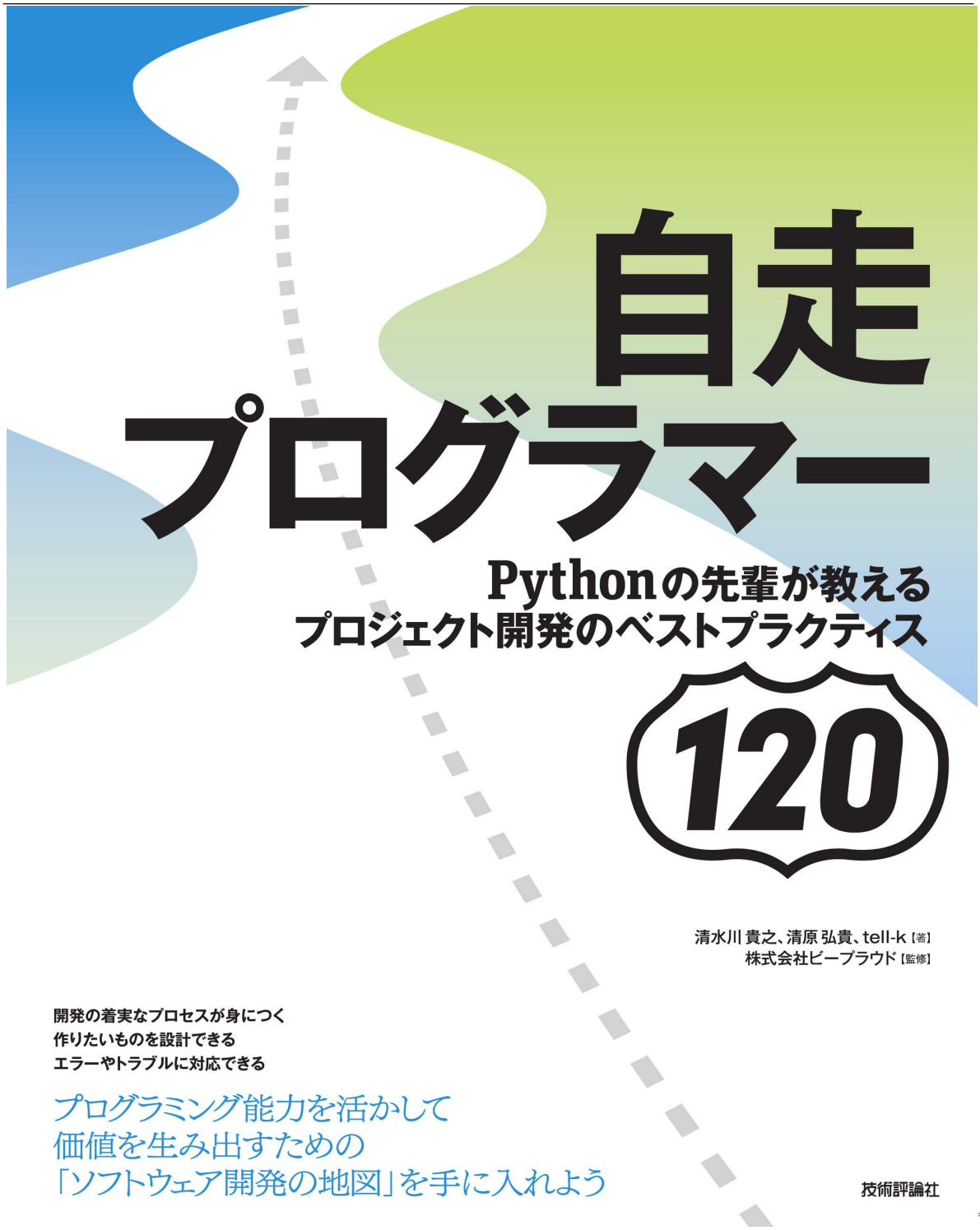
スパゲッティクエリは、複雑な問題を 1 つの SQL で解決しようとするアンチパターンです。『SQL アンチパターン』（Bill Karwin 著、オライリージャパン刊、2013 年）で紹介されているアンチパターンの 1 つで、無理に 1 つの SQL に押し込めようとするあまり、複雑で読み解くことができない SQL を書いてしまう問題を指しています。無理に 1 つの SQL にすることは避けるべきですが、本節の例のようにパフォーマンスに影響が出るような実装もまた避けるべきでしょう。

ベストプラクティス

理想の **SQL** を書いてから、その SQL を ORM で発行するように実装しましょう。

先ほどの例では、データベースから取得した ID のリストをそのまま次の SQL に渡したり、不要な JOIN が行われているという問題がありました。ソースコメントからも、これが意図した結果ではなく ORM をうまく扱えなかった結果だというのが明らかです。ORM をうまく扱うには、使っている ORM ライブラリのクセを把握する必要があります。複雑なクエリを実装するときは先に 理想の **SQL** を書いて、その SQL を使っている ORM で再現できるかを検討するのが良いでしょう。

以下の SQL は、要件から期待される理想の SQL です。



*160

*160 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*161] をご参照ください

関連

- *60:Django ORM* **でどんな** *SQL* **が発行されているか気にしよう** (ページ 218)

[*161] https://gihyo.jp/book/2020/978-4-297-11197-7

第 4 章

エラー設計

4.1. エラーハンドリング

4.1.1 63:臆さずにエラーを発生させる

プログラミング迷子: 例外を発生させたくない

- 先輩 T：この `def validate(data):` 関数の中で `data.get('ids')` っていうコードがたくさんあるんだけど、フレームワークが `data` 辞書を用意して `validate` を呼んでくれるから、 `'ids'` は必ずあるんじゃない？
- 後輩 W：ありますね。
- 先輩 T：じゃあどうして `data['ids']` じゃなく `data.get('ids')` なの？
- 後輩 W： `'ids'` がない場合に例外を発生させないようにするためです。
- 先輩 T：？？？
- 後輩 W： `validate` に必ず `'ids'` を持つ辞書を渡してくれるかわからないですよね。
- 先輩 T：それはフレームワークがよくわからないから過剰防衛してるだけでは。

例外を発生させるのは悪、と考えて、関数に渡される値のさまざまなケースに対応して過剰実装してしまうと、実際にはあり得ない引数のためにコードが複雑化してしまいます。臆病になりすぎず、かつ問題の発生を見逃さないシンプルな方法があるでしょうか？

具体的な失敗

以下のコードは、関数に渡される辞書オブジェクトの中身を心配しすぎています。

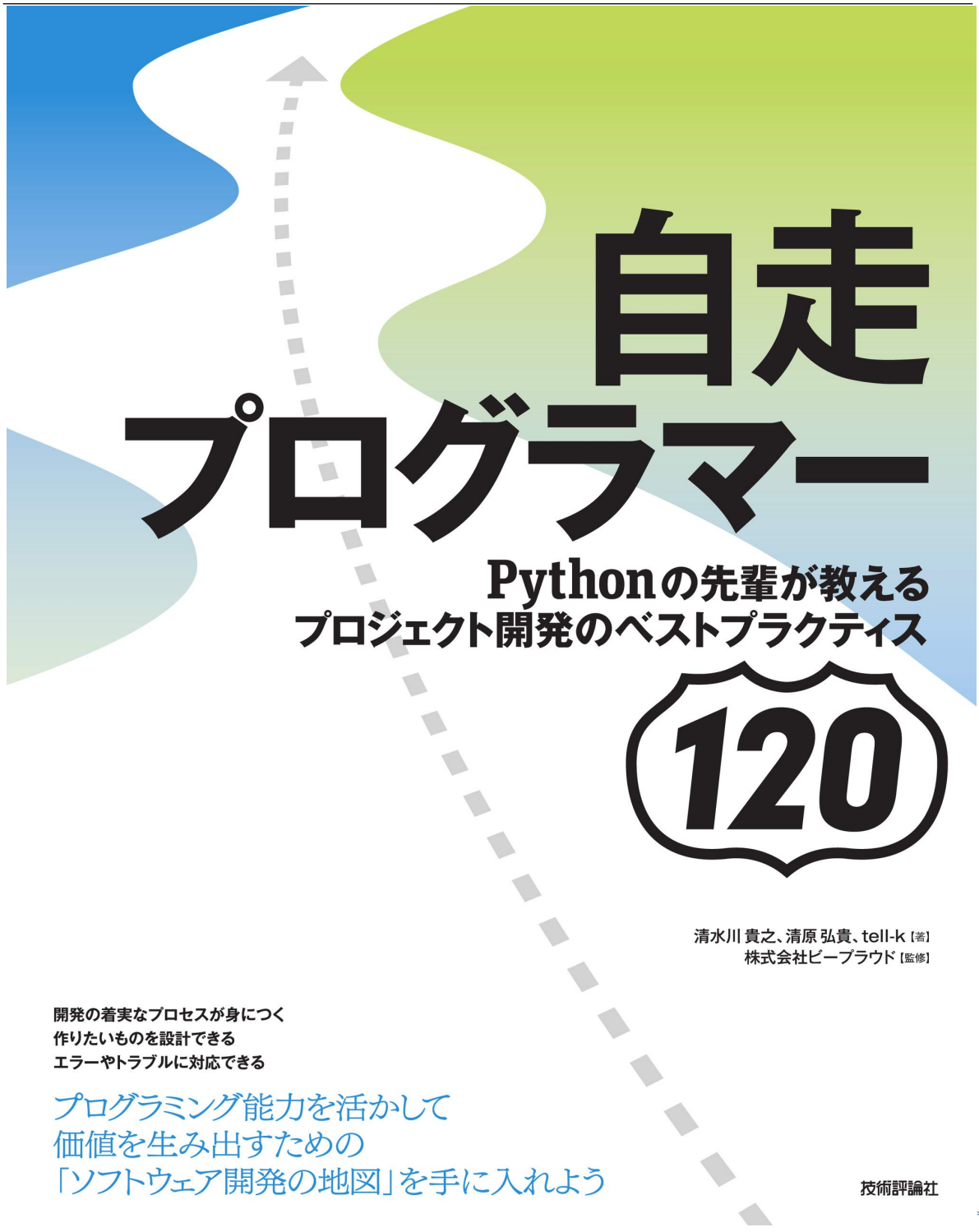
```
def validate(data):
    """data['ids'] を検査して、含まれる不正な id の一覧を返す
    """
    ids = data.get('ids')  # ここが問題
    err_ids = []
    for id in ids:
        if ...:  # id が不正かどうかをチェックする条件文
```

(次のページに続く)

(前のページからの続き)

```
            err_ids.append(id)
    return err_ids
```

辞書オブジェクトが「キーを持っているかどうかわからない」から `data.get('ids')` というコードを書いたケースです。この予防措置によって、 `data.get('ids')` で `None` が返される可能性が生まれてしまっています。もし `None` が返された場合、その 2 行後の `for id in ids` で結局エラーになってしまうため、この予防措置には意味がありません。それどころか、 `data.get('ids')` と書いたために、 `None` が返された場合にどうすれば良いかを心配しながらその先のコードを書かなければいけなくなってしまっています。



*162

*162 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*163] をご参照ください

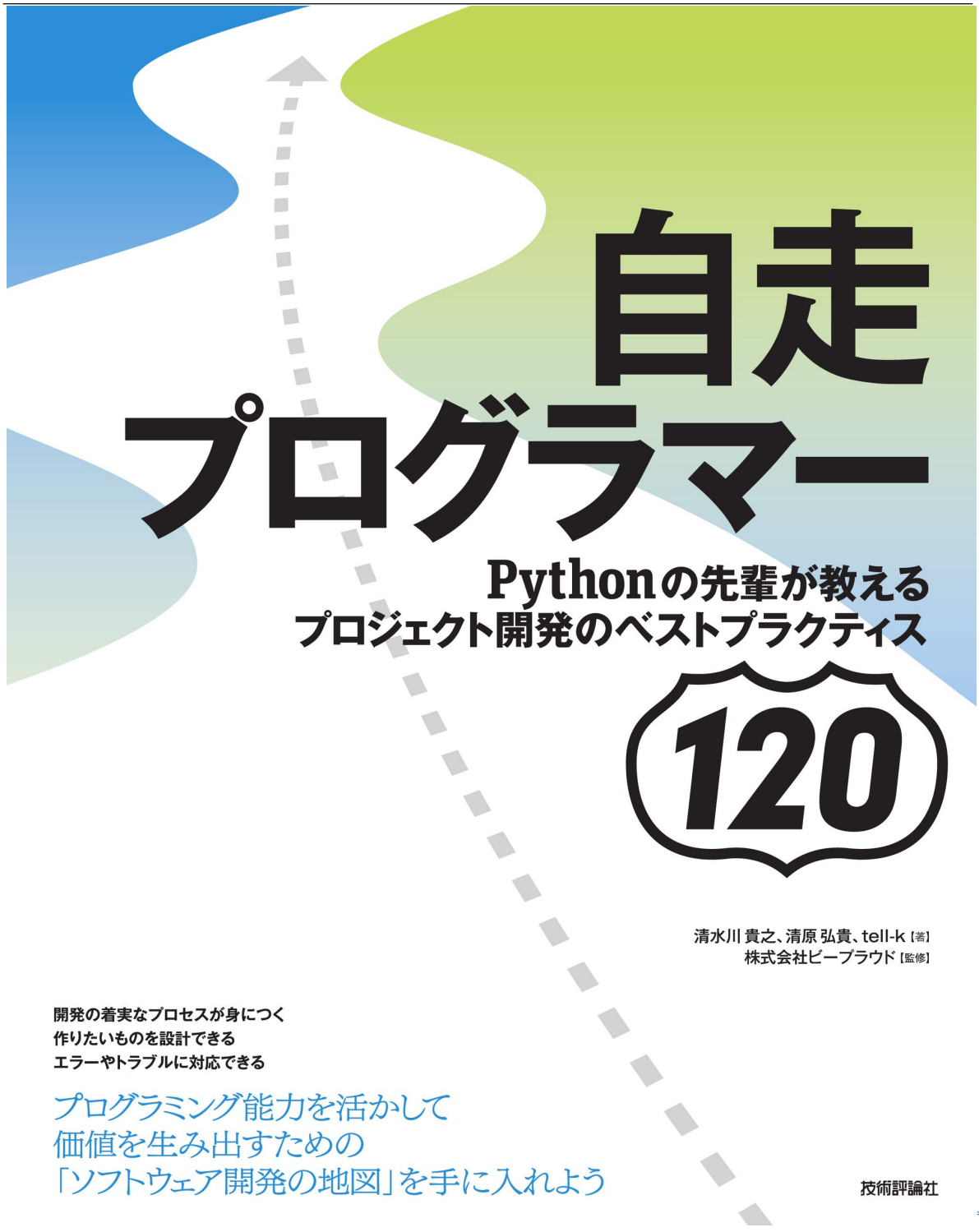
こういったコードは、例外が発生する可能性を気にしすぎて例外を隠蔽してしまったため、 バグに早く気づく ことができません。

ベストプラクティス

例外を隠すのではなく、わかりやすい例外を早く上げるコードを書きましょう。

辞書のキーがあってもなくても動作するコードを書くより、期待するデータが必ず渡される前提でコードを書くとシンプルになります。もし呼び出し方を間違えた場合には、例外が発生するため問題に早く気づけます。

[*163] https://gihyo.jp/book/2020/978-4-297-11197-7



*164

*164 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*165] をご参照ください

また、引数のルールを自分で決められる場合は、 *12:*辞書でなくクラスを定義する (ページ 47) も参照してください。

4.1.2 64:例外を握り潰さない

プログラミング迷子**:** ユーザーに例外を見せるのは絶対避けたい

- 先輩 T：ここの処理で例外を except して `return None` してるけど、そのまま例外起こしたほうがいいね。
- 後輩 W：なんでですか？
- 先輩 T：そもそもここの処理はファイルがあることが前提だから、想定外のことが起こったらそこでエラーになってプログラムは止まってほしい。
- 後輩 W：でもユーザーにエラーが見えちゃうじゃないですか。
- 先輩 T：ファイルがないままプログラムを継続しても、後続の読み込み処理で結局エラーになるから、継続する意味がないんだよ。
- 後輩 W：たしかに継続することに意味がないですね。
- 先輩 T：しかも、下手に例外処理してるから、エラー原因がファイルがないためなのか、あるけど空なのか、traceback を読んでもわからないんだよ。
- 後輩 W：ほーん。
- 先輩 T：プログラムで想定外のことが起こったら、素直に例外を上げて終了してくれたほうがいい。すべての不測の事態に備えてコードを書くことはできないからね。

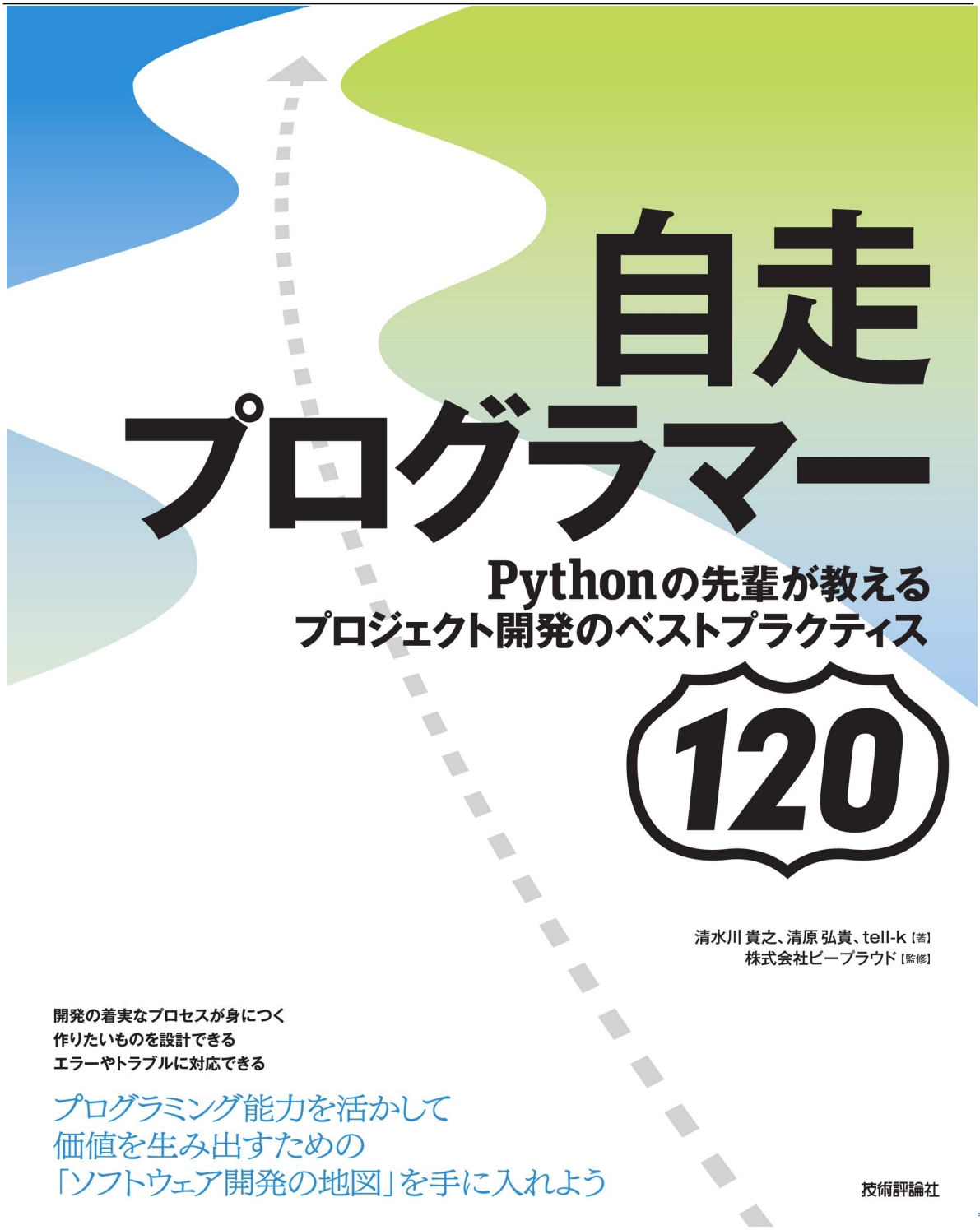
これは「例外を発生させるのは悪、なぜならユーザーに見えてしまうからだ」という発想です。確かに、ユーザーに例外の詳細を見せる必要はないかもしれません。しかし、例外の仕組みはプログラミング言語に組み込まれている機能です。隠蔽するのではなく、活用しましょう。

[*165] https://gihyo.jp/book/2020/978-4-297-11197-7

具体的な失敗

以下の例は、認証が必要な Web API にアクセスするコードですが、例外の発生を避けたために本当の原因がわかりづらくなっています。

```
import requests

def make_auth_header():
    try:
        s = get_secret_key()  # シークレットキーをファイルから読み込み
    except:
        return None
    return {'Authorization': s}

def call_remote_api():
    headers = make_auth_header()
    res = requests.get('http://example.com/remote/api', headers=headers)
    res.raise_for_status()  # ファイルがない場合、ここで認証エラーの例外が発生する
    return res.body
```



*166

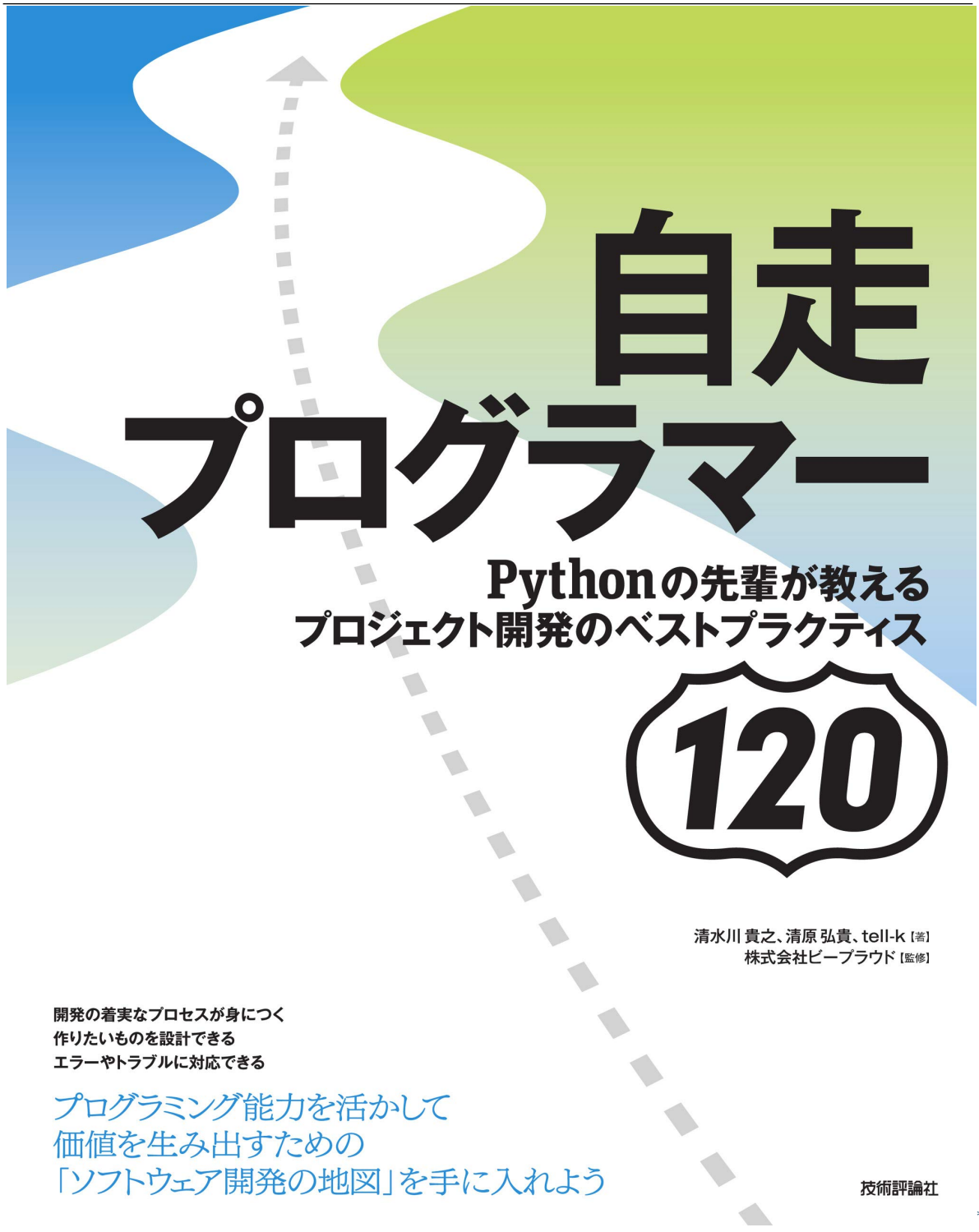
*166 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*167] をご参照ください

ベストプラクティス

想定外の例外を心配して握り潰すのはやめましょう。エラーが起きたとき問題をユーザーから隠すのではなく、簡単に正しい状態に復帰しやすいように適切な情報を提供してくれるシステムこそ「ユーザーにやさしいシステム」と言えます。想定される例外の処理は実装するべきですが、想定外のエラーを隠蔽してはいけません。

[*167] https://gihyo.jp/book/2020/978-4-297-11197-7



*168

*168 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*169] をご参照ください

4.1.3 65:try 節は短く書く

プログラミング迷子**:** 大きい **try** 節は小さい **try** 節を兼ねる？

- 先輩 T：今朝言ってたバグの調査、けっこう手間取ってる？
- 後輩 W：すいません、どこで問題が出てるかまだわからなくて……。
- 先輩 T：どれどれ……うわ、try 節長いなー。これだとどこでバグってるかわからなそうだ。
- 後輩 W：try って長いとだめなんですか？
- 先輩 T：そうだねー、できるだけ短いほうがいいね。

例外の処理を書き慣れていないと、とても長い try 節 を書いてしまいます。このとき、1 つの except 節 ですべてのエラー処理をまとめてしまうと、どの行でどんなエラーが起きたかわからなくなってしまいます。

具体的な失敗

たとえば、以下のような Web アプリケーションのフォームを処理するコードがあるとします。このコードは、エラーが発生した際に問題を切り分けられないというバグを含んでいます。

```
def purchase_form_view(request):
    try:
        product = get_product_by_id(int(request.POST['product_id']))
        purchase_count = request.POST['purchase_count']
        if purchase_count <= product.stock.count:
            product.stock.count -= int(request.POST['purchase_count'])
            product.stock.save()
            return render(request, 'purchase/result.html', {
                'purchase': create_purchase(
                    product=product,
                    count=int(purchase_count),
```

(次のページに続く)

[*169] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

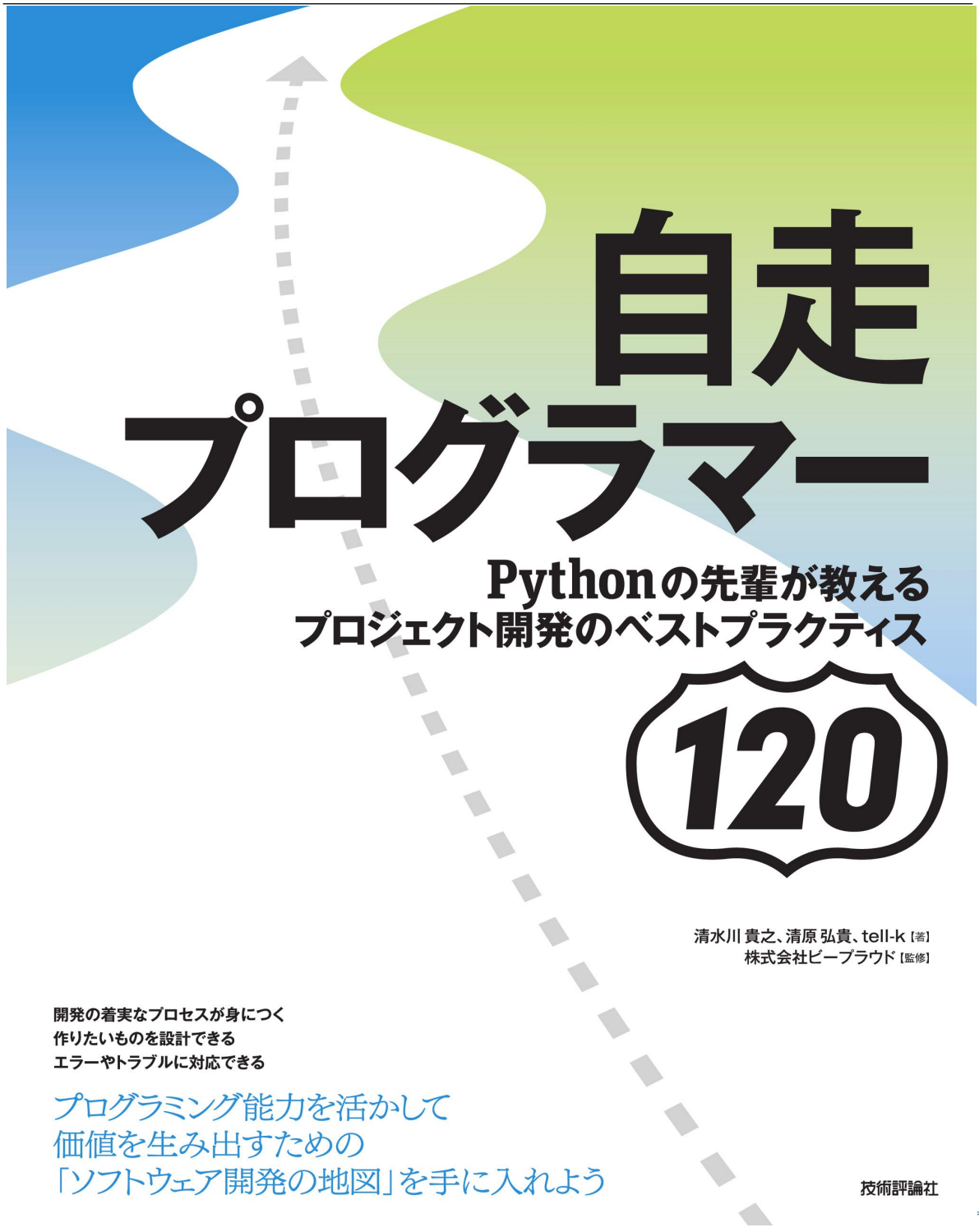
```
                    amount_price=purchase_count * product.price,
                )
            })
    except:
        return render(request, 'error.html')  # エラーが発生しました、と表示
```

このコードは、関数内のすべての処理を try 節に書き、 except 節ですべての例外を捕まえて、エラー処理をしています。ここで、Web アプリケーションの利用中に例外が発生しても、画面には「エラーが発生しました」とだけ表示されるため、ユーザーにも開発者にもエラーの原因はわかりません。エラーの原因の可能性として、ユーザーからのパラメータが想定外、他の処理で DB に保存したデータに問題がある、実装に変数名間違いなど単純なバグがある、ライブラリの更新で動作が変わった……など、多くの可能性があります。このため、開発者が原因を調べて不具合を解消するのにとても時間がかかってしまいます。

ベストプラクティス

try 節のコードはできるだけ短く、1 つの目的に絞って処理を実装しましょう。

try 節に複数の処理を書いてしまうと、発生する例外の種類も比例して多くなっていき、except 節でいろいろな例外処理が必要になってしまいます。次のコードは、try 節の目的を絞ってそれぞれ個別の例外処理を行うことで、わかりやすいエラーメッセージをユーザーに伝えています。これによって、ユーザーが正しい状態に復帰できるようにしています。



[*170]

[*170] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*171] をご参照ください

4.1.4 66:専用の例外クラスでエラー原因を明示する

プログラミング迷子: エラー理由がわからない

- 後輩 W：ユーザーから、メールがあるはずなのに表示されないっていう問い合わせが来てるんですが、いま別件対応中なので見てもらえますか？
- 先輩 T：いいよ。問い合わせにエラーメッセージとか書かれてた？
- 後輩 W：はい。「メールを受信できません」と表示されたみたいです。
- ― 10 分後―
- 先輩 T：実装コードはすぐ見つかったけど、これじゃあ何が原因でエラーになったのかわからないぞ……。

```
mail = mail_service.get_newest_mail()
if isinstance(mail, str):
    return mail  # <-- 文字列のときは常に"メールを受信できません"だった（先輩 T）
```

- 先輩 T：この実装、`mail_service.get_newest_mail()` で異常があったことはわかるんだけど、何が起きても「メールを受信できません」と返しているから異常の原因がわからないよ。原因にあわせて文面を変えるべきだし、異常時には例外を上げるべきじゃないかな？
- 後輩 W：そう思ったんですけど、ちょうど良い例外クラスが Python になかったんです。
- 先輩 T：そういうときは、例外クラスを自分で定義して使えばいいよ。

エラー発生時や期待どおりに動作しないときなどに、ユーザーから問い合わせを受けて調査を行うことがあります。このとき、画面表示にユーザー向けの情報が不足していると、調査が難しくなります。

[*171] https://gihyo.jp/book/2020/978-4-297-11197-7

具体的な失敗

問題のあるコードは以下のように実装されています。

views.py

```
from . import service

def get_newest_mail(user):
    """
    ユーザーのメールアドレスに届いている 1 時間以内の最新のメールを取得する
    """
    mail_service = service.get_mail_service()
    if not mail_service.login(user.email, user.email_password):
        return 'ログインできません'
    mail = mail_service.get_newest_mail()
    if isinstance(mail, str):
        return mail
    if mail.date < datetime.now() - timedelta(hours=1):
        return 'メールがありません'
    return mail

def newmail(request):
    mail = get_newest_mail(request.user)
    if isinstance(mail, str):
        return render(request, 'no-mail.html', context={'message': mail})
    context = {
        'from': mail.from_, 'to': mail.to,
        'date': mail.date, 'subject': mail.subject,
        'excerpt': mail.body[:100],
    }
    return render(request, 'new-mail.html', context=context)
```

get_newest_mail 関数やそこから呼び出している mail_service.get_newest_mail() は、例外を握り潰してはいませんが、エラーが発生した場合に文字列を返してしまっています。このため、呼び出し元では if isinstance で文字列かどうかを判定して場合分けの処理が必要です。また、「文字列が返されたときは常にエラー」というわけでもなく、正常系と異常系の処理の見分けがつかない実装コードになっているため、コードを読み解くのが難しくなっています。

ベストプラクティス

専用の例外クラスを自作して、エラーを明示的に実装しましょう。

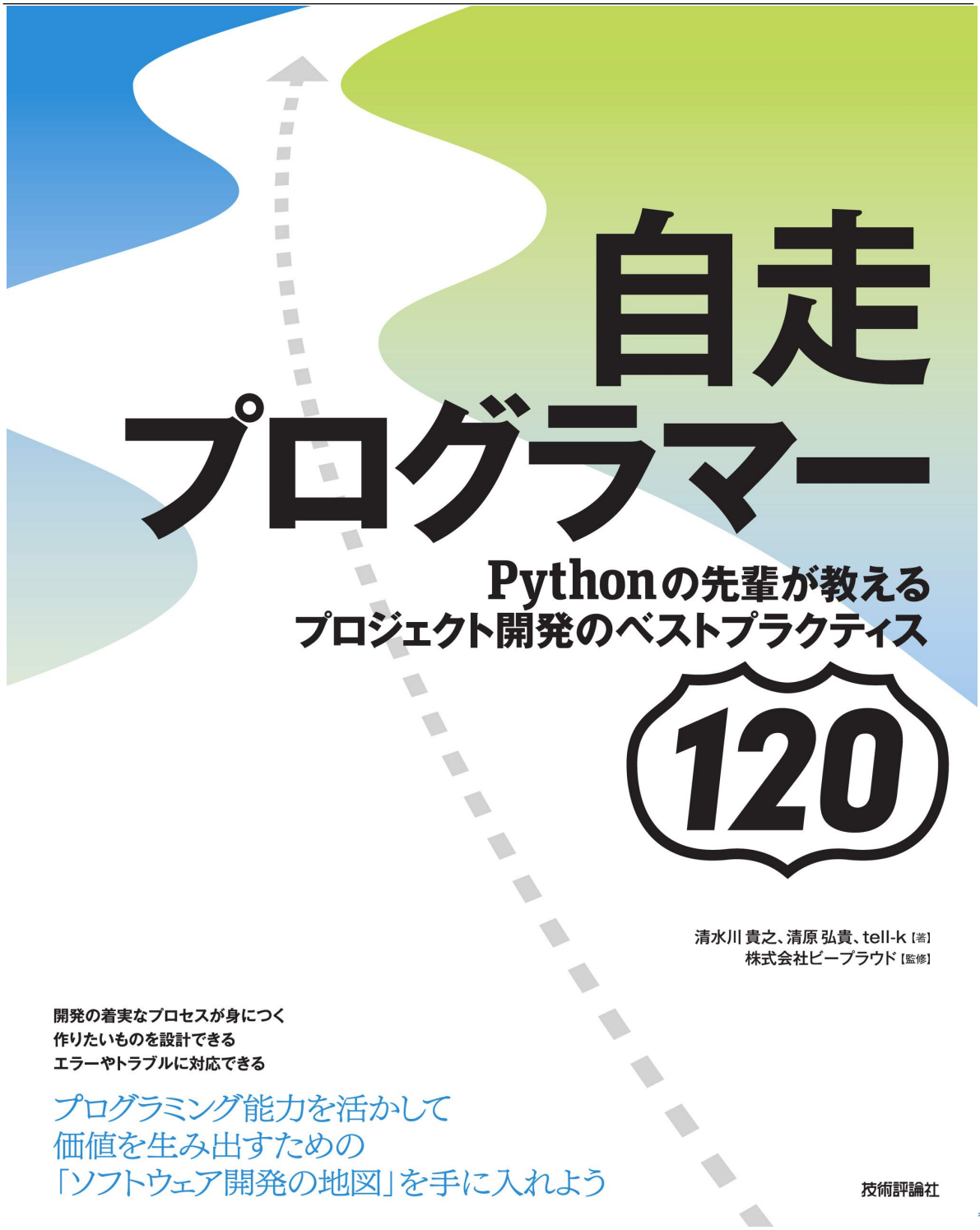
発生するエラーの種類ごとに専用の例外クラスを定義して、それぞれ異なるエラーメッセージを表示するように実装します。また、各例外の親クラスを定義しておけば、例外処理を行うコードで同系統の例外をまとめて捕まえられるため、簡潔でわかりやすい実装になります。前述のコード用に例外クラスを実装すると、以下のようになります。

exceptions.py

```
class MailReceivingError(Exception):
    pretext = ''
    def __init__(self, message, *args):
        if self.pretext:
            message = f"{self.pretext}: {message}"
        super().__init__(message, *args)

class MailConnectionError(MailReceivingError):
    pretext = '接続エラー'

class MailAuthError(MailReceivingError):
    pretext = '認証エラー'

class MailHeaderError(MailReceivingError):
    pretext = 'メールヘッダーエラー'
```

このように実装した例外クラスは、以下のように動作します。

```
>>> e = MailHeaderError('Date のフォーマットが不正です')
>>> str(e)
'メールヘッダーエラー: Date のフォーマットが不正です'
>>> raise e
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
exceptions.MailHeaderError: メールヘッダーエラー: Date のフォーマットが不正です
```



*172

*172 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*173] をご参照ください

関連

- *64:*例外を握り潰さない (ページ 241)
- *71:info*、*error* だけでなくログレベルを使い分ける (ページ 267)
- *75:Sentry* でエラーログを通知／監視する (ページ 279)

[*173] https://gihyo.jp/book/2020/978-4-297-11197-7

4.2. ロギング

4.2.1 67:トラブル解決に役立つログを出力しよう

プログラミング迷子: ログ出力は何のため？

- 後輩 W：ログ出力って要りますか？
- 先輩 T：要るね。ログ出力は開発者をトラブルから守ってくれる大事な武器だよ。
- 後輩 W：そうなんですか。トラブルが起きてもログが役立ったことがなかったんで実感がないんですけど……。
- 先輩 T：ん？　たとえばどんなログが出力されてたの？
- 後輩 W：今運用しているシステムでは、ログファイルに処理の開始と終了を出力してます。でも、それを見ても「購入できない」という問い合わせの原因を調べる役には立ちませんでした。
- 先輩 T：なるほど。それなら、「購入できない」状況を詳しくログ出力すれば良いんじゃないかな。
- 後輩 W：稼働してるシステムにログ出力を追加するんですか？　予算がなくてできないって言われちゃいませんか？
- 先輩 T：あとからでも追加したほうが良いね。そうしないと、トラブルのたびに問い合わせ対応や調査でお金も時間もかかっちゃうよ。

ログ出力（ロギング）を実装しているかどうかで、システムの保守性やトラブルシューティングにかかる時間は格段に変わってきます。ただし、処理の開始と終了しかロギングしていなかったり、処理フローで重要な値をロギングしていないようでは、トラブルの解決にはほとんど役立ちません。トラブルシューティングに時間がかかれば、お金と時間を浪費するだけでなく、サービス自体の機会損失にもつながってしまいます。

具体的な失敗

たとえば、以下のようなログ出力では困ります。

```
INFO: 購入処理開始
INFO: 在庫確認 API 呼び出し
INFO: 在庫引き当て NG
```

(次のページに続く)

(前のページからの続き)

```
INFO: 購入処理開始
INFO: 在庫確認 API 呼び出し
INFO: 在庫引き当て OK
INFO: 購入完了
```

このログには各行の日時情報がなく、「誰がどの商品をいくつ購入しようとしているのか」といった購入処理フローの重要な値も出力されていません。「在庫引き当て NG」というログからは在庫不足のようにも見えますが、在庫確認 API 呼び出しでエラーが起きていてそのエラーが出力されていないのかもしれません。このようにログ出力が不足していると、トラブルシューティングに苦しむことになります。特に外部システムとの結合テストや本番リリース後の調査では、問題発生時に素早く、正確に状況を把握することが重要です。状況を正確に把握できなければ、エラー原因の可能性は無数にありえますし、解決までの暫定的な対策も検討できません。

ベストプラクティス

トラブル解決に役立つログを出力しましょう。問題発生時に状況を正確に把握できるロギングを実装するには、ログ出力の内容からプログラムの動作を把握できるようにすることが大事です。状況を正確に把握できれば、どうやって問題を解決するかに集中できますし、根本解決に時間がかかるとしても暫定的な対策を検討できます。

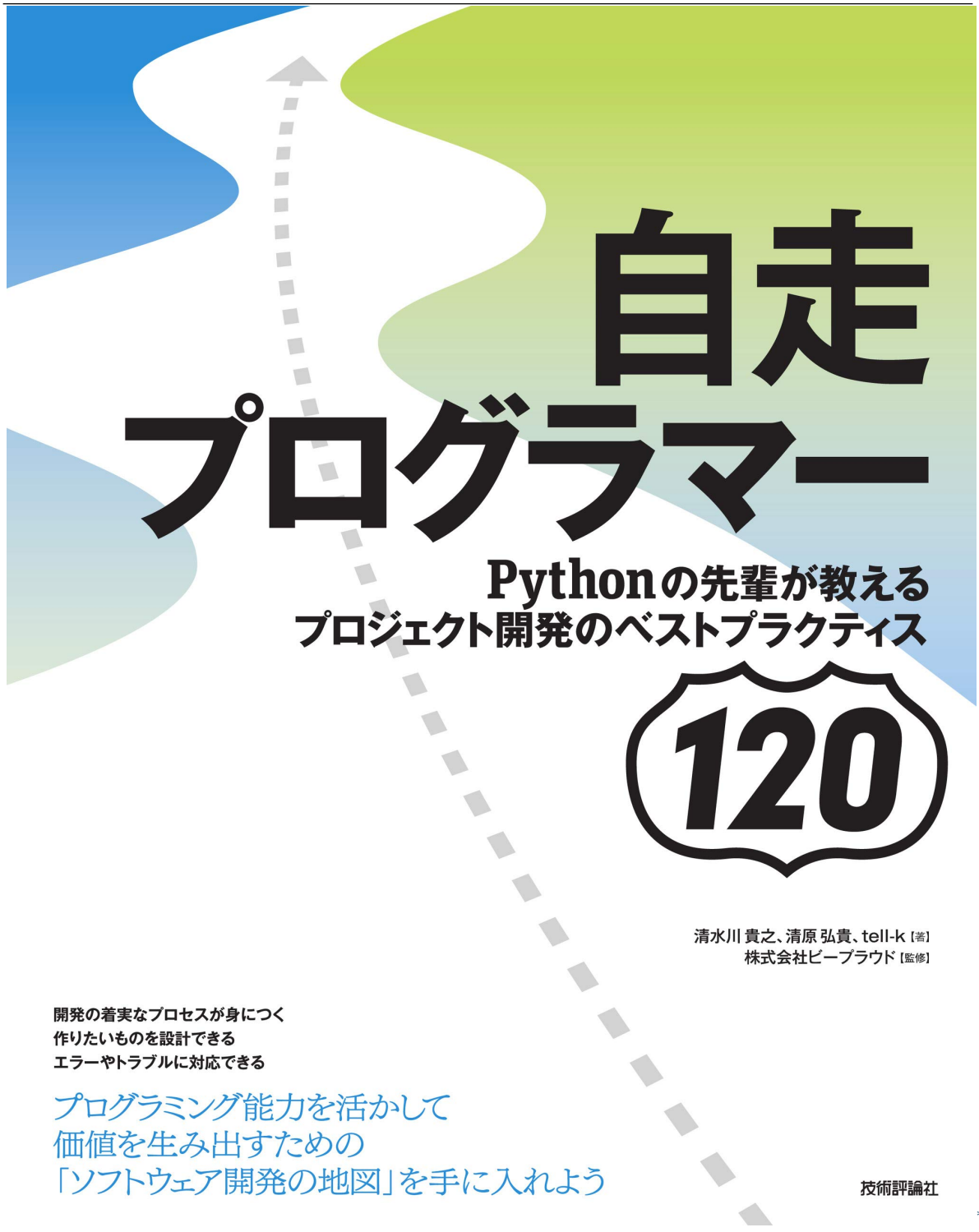
たとえば、以下のようなログ出力であれば、先ほどの例よりも状況が正確に把握できます。

```
[19/Jan/2020 07:38:41] INFO: user=1234 購入処理開始: 購入トランザクション=2345, 商品 id␣
↪111(1 個),222(2 個)
[19/Jan/2020 07:38:41] INFO: user=1234 在庫引き当て API: POST /inventory/allocate␣
↪params=...
[19/Jan/2020 07:38:42] INFO: user=1234 在庫引き当て API: status=200, body=""
[19/Jan/2020 07:38:42] ERROR: user=1234 在庫システム API でエラーのため、担当者へ連絡してく
ださい
Traceback (most recent call last):
  File "/var/www/hanbai/apps/inventry/service.py", line 162, in allocate
    return r.json()
  ...
  File "/usr/lib64/python3.6/json/decoder.py", line 357, in raw_decode
    raise JSONDecodeError("Expecting value", s, err.value) from None
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
[19/Jan/2020 07:38:42] INFO: user=1234 購入 NG status=500
```

(次のページに続く)

(前のページからの続き)

```
[19/Jan/2020 07:40:07] INFO: user=5432 購入処理開始: 購入トランザクション=2346, 商品 id␣
↪222(3 個),333(1 個)
[19/Jan/2020 07:40:07] INFO: user=5432 在庫引き当て API: POST /inventory/allocate␣
↪params=...
[19/Jan/2020 07:40:08] INFO: user=5432 在庫引き当て API: status=200, body="{...}"
[19/Jan/2020 07:40:08] INFO: user=5432 在庫引き当て OK: 商品 id 222(3 個),333(1 個)
[19/Jan/2020 07:40:09] INFO: user=5432 購入確定: 購入トランザクション=2346
[19/Jan/2020 07:40:10] INFO: user=5432 購入完了 status=200
```



*174

*174 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*175] をご参照ください

関連

- *63:* **臆さずにエラーを発生させる** (ページ 236)
- *64:* **例外を握り潰さない** (ページ 241)
- *68:* **ログがどこに出ているか確認しよう** (ページ 258)
- *71:info*、*error* **だけでなくログレベルを使い分ける** (ページ 267)
- *73:* **ログには** *5W1H* **を書く** (ページ 274)
- *75:Sentry* **でエラーログを通知／監視する** (ページ 279)

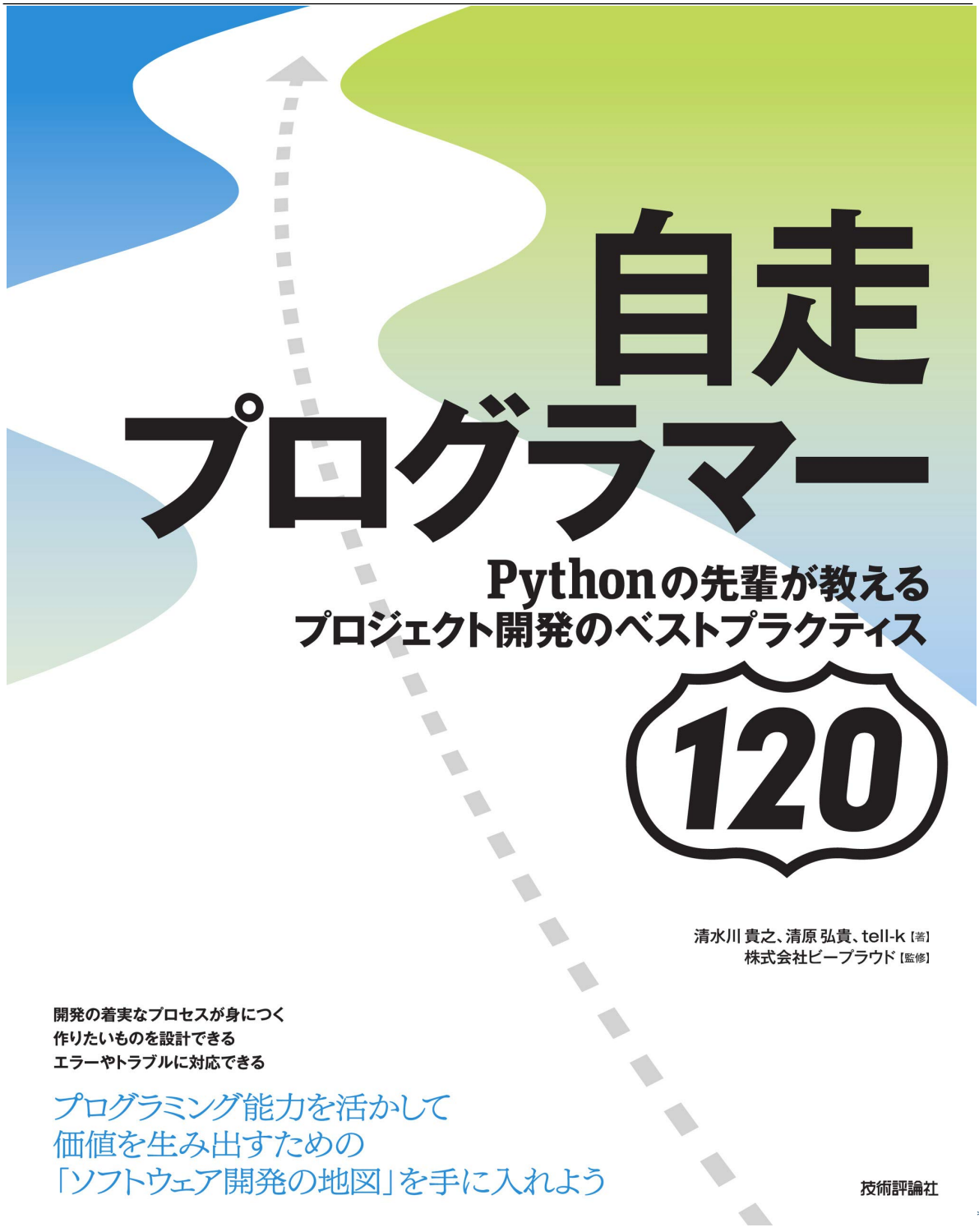
4.2.2 68:ログがどこに出ているか確認しよう

保守を引き継いだプロジェクトなどで、アプリケーションのログが全く出力されていない、といったことはありませんか？利用者から「画面にエラーが発生しました、と表示されます」と連絡をもらい、調べてみたらログがどこにも出ていないということもよくある話です。ログが出力されていないのは論外ですが、実装者がログの重要性がわかっていないと、たびたびこういった問題が起こります。

[*175] https://gihyo.jp/book/2020/978-4-297-11197-7

具体的な失敗

Django の場合、開発中は `manage.py runserver` で Web アプリケーションを実行します。Django のデフォルトの設定では、ページにアクセスするたびにコンソールにアクセスログが出力されます。しかし「ログ出力を実装する」と言った場合、アクセスログのことではなく、明示的に実装したログのことを指すのが一般的です。アクセスログを見て「ログが出ている」と考えてはいけません。



*176

*176 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*177] をご参照ください

ベストプラクティス

ログがどこに出力されるのか、調査しやすい情報が出力されているか、早い段階で確認しておきましょう。そのために、以下の情報を確認しましょう。

- `settings.py` の `LOGGING` が設定されていること
- ファイルに出力する設定の場合、ログがファイルに記録されていること
- 標準出力に出力する設定の場合、Gunicorn 等を起動しているサービスマネージャーのログに記録されていること
- 記録されているログに、ログレベルや時刻など期待する情報が出力されていること

どのような情報がログに出力されていると良いのかについては、以降のプラクティスで説明します。また、サービスマネージャーについては *93:*サービスマネージャーでプロセスを管理する (ページ 338) を参照してください。

関連

- *60:Django ORM* でどんな *SQL* が発行されているか気にしよう (ページ 218)
- *107:*リバースプロキシ (ページ 390)

4.2.3 69:ログメッセージをフォーマットしてロガーに渡さない

Python ではロギングの書き方に注意が必要です。ログメッセージを フォーマット してからログに残していませんか？

[*177] https://gihyo.jp/book/2020/978-4-297-11197-7

具体的な失敗

```
import logging

logger = logging.getLogger(__name__)


def main():
    items = load_items()
    logger.info(f"Number of Items: {len(items)}")
```

ロガーにログメッセージを渡すときは、フォーマットしてはいけません。Python の *f""* を使って文字列をフォーマットするのは便利ですが、ロギングのときは使わないでください。

ベストプラクティス

ログのフォーマットにするときは以下のように、フォーマットせずに使いましょう。

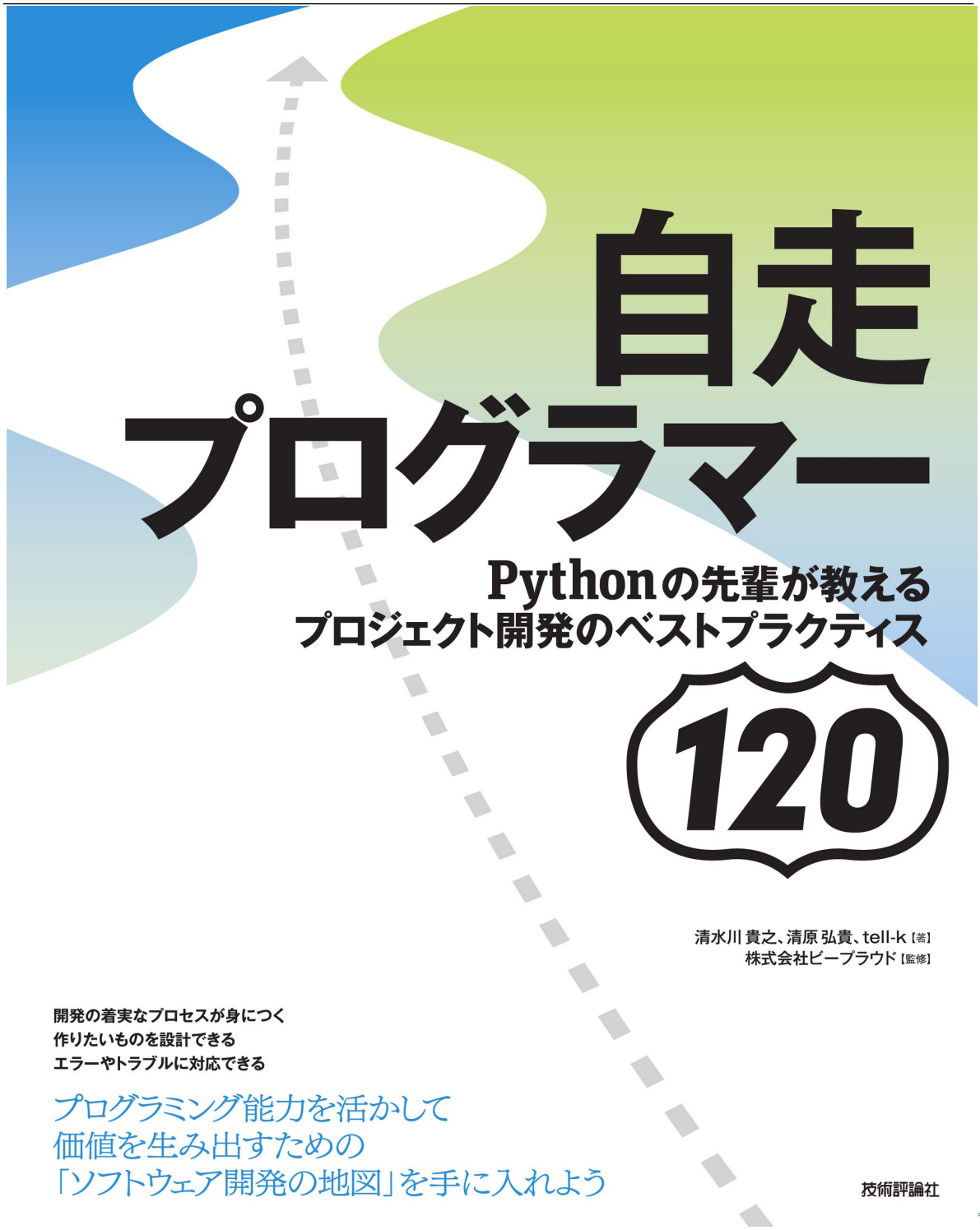
```
def main():
    items = load_items()
    logger.info("Number of Items: %s", len(items))
```

フォーマットしてロガーに渡さない理由は、ログを運用する際にメッセージ単位で集約することがあるからです。たとえば Sentry はログのメッセージ単位で集約して、同一の原因のログを集約、特定します。ここで事前にフォーマットしてしまうと、全く別々のログメッセージと判断されてしまいます。

Python のロギングは内部的に「メッセージ」と「引数」を分けて管理しているので、分けたままログに残すべきです。logger.log の第一引数がメッセージ、以降はメッセージに渡される値になります。

ログメッセージを読みやすく装飾したいときは、ロガーの Formatter に設定しましょう[*178] 。Formatter の style 引数に指定するとフォーマットを指定できます。

[*178] https://docs.python.org/ja/3/library/logging.html#logging.Formatter



*179

*179 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*180] をご参照ください

関連

- *75:Sentry* **でエラーログを通知／監視する** (ページ 279)

4.2.4 70:個別の名前でロガーを作らない

ロギングの設定が上手に書かれていないと、煩雑になりがちです。ここではロガーの効果的な設定方法を学びましょう。

具体的な失敗

```
logging.config.dictConfig({
    ...
    "loggers": {
        "product_detail_view": {},
        "product_edit_view": {},
        "import_products_command": {},
        "export_sales_command": {},
        "sync_ma_events": {},
        "sync_payment_events": {},
        ...
    }
})
```

この設定の場合、ロガーを 1 つ増やすたびにロギングの設定を足す必要があります。

[*180] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

ロガーはモジュールパス __name__ を使って取得しましょう。

```
import logging

logger = logging.getLogger(__name__)
```

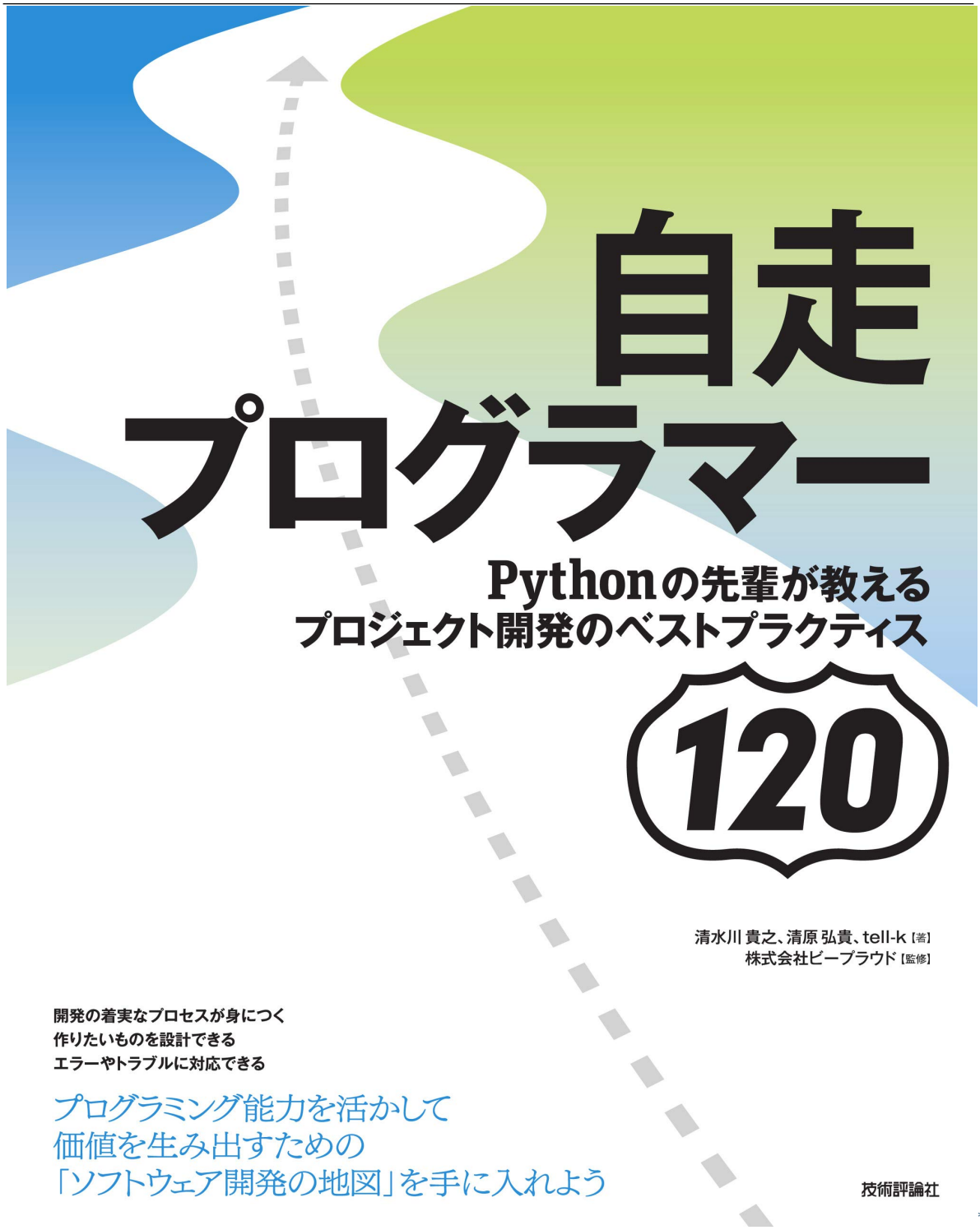
こうするとロギングの設定はまとめて書けるようになります。

```
logging.config.dictConfig({
    ...
    "loggers": {
        "product.views": {},
        "product.management.commands": {},
    }
})
```

Python では「 . 」区切りで「上位」（左側）のロガーが適応されます。ロガーの名前が product.views.api のときは product.views.api 、 product.views 、product `` と順にログの設定を探して、設定があれば使われます。

Python は __name__ で現在のモジュールパスが取得できるので、product/views/api.py というファイルでは product.views.api になります。

ロガーすべてに毎度名前をつけていると、ロガーごとに設定が必要になり面倒です。まとめて設定することで設定の数を減らせます。Python のモジュール名にすることでロガーの命名規則を考える必要もなくなります。



*181

*181 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*182] をご参照ください

4.2.5 71:info、error だけでなくログレベルを使い分ける

ログを書くときに logger.info と logger.error 以外を使っていますか？ログレベルを使い分けることで、ログの集約と通知がより効果的に行えます。

具体的な失敗

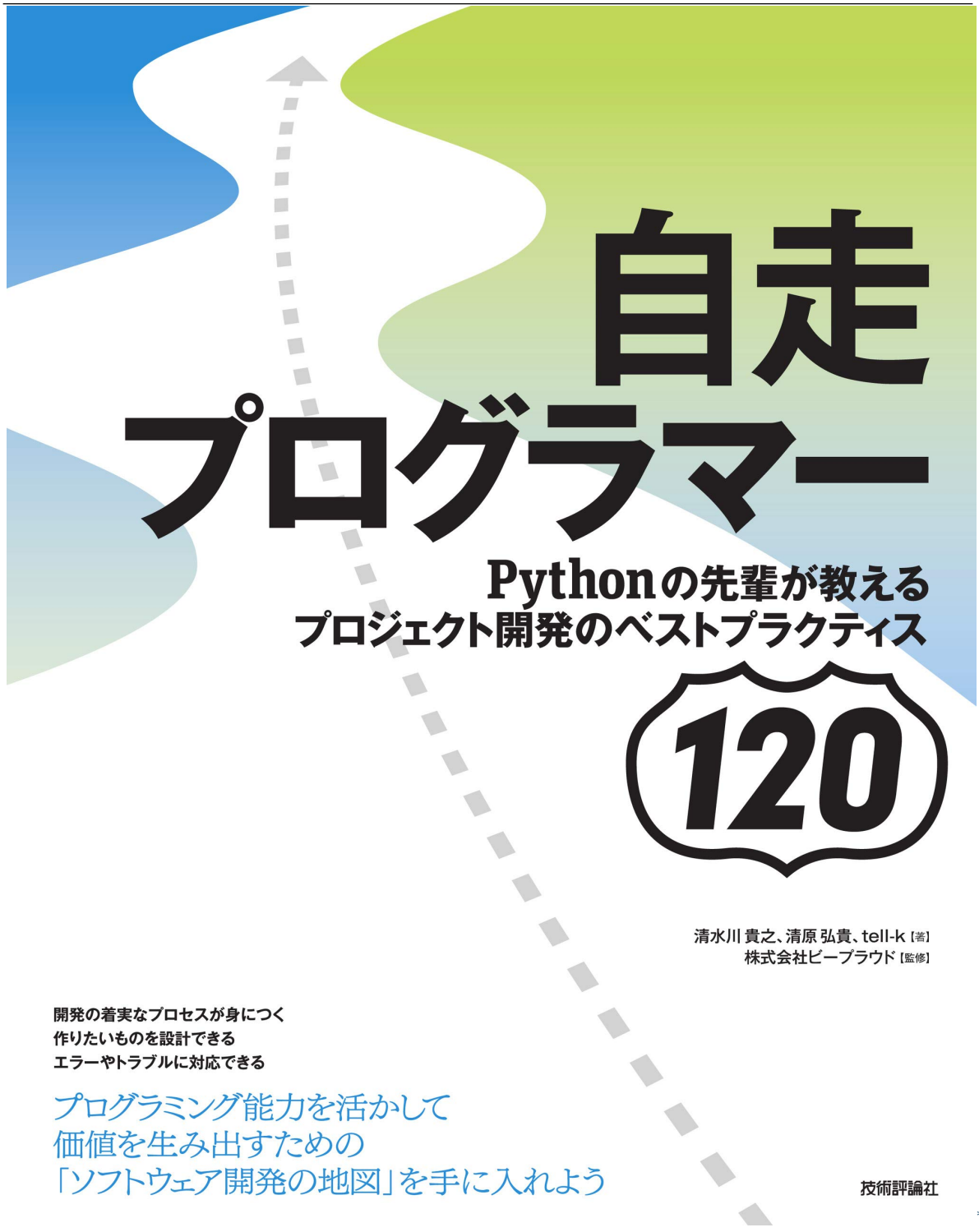
```
import logging

logger = logging.getLogger(__name__)


def main():
    ...
    for row in data:
        if not validate_product_data(...):
            logger.info("Skipped invalid sales data %r", row["id"])
        ...
```

この場合、商品のデータが不正な場合に logger.info でログ出力してしまっています。「エラーほどではない」という理由でインフォレベルのログにすると、何かしらのアクションが必要な場合でも気づけないことが多いでしょう。

[*182] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

このようにエラーとも言い切れない場合は `logger.warning` レベルを使いましょう。



*183

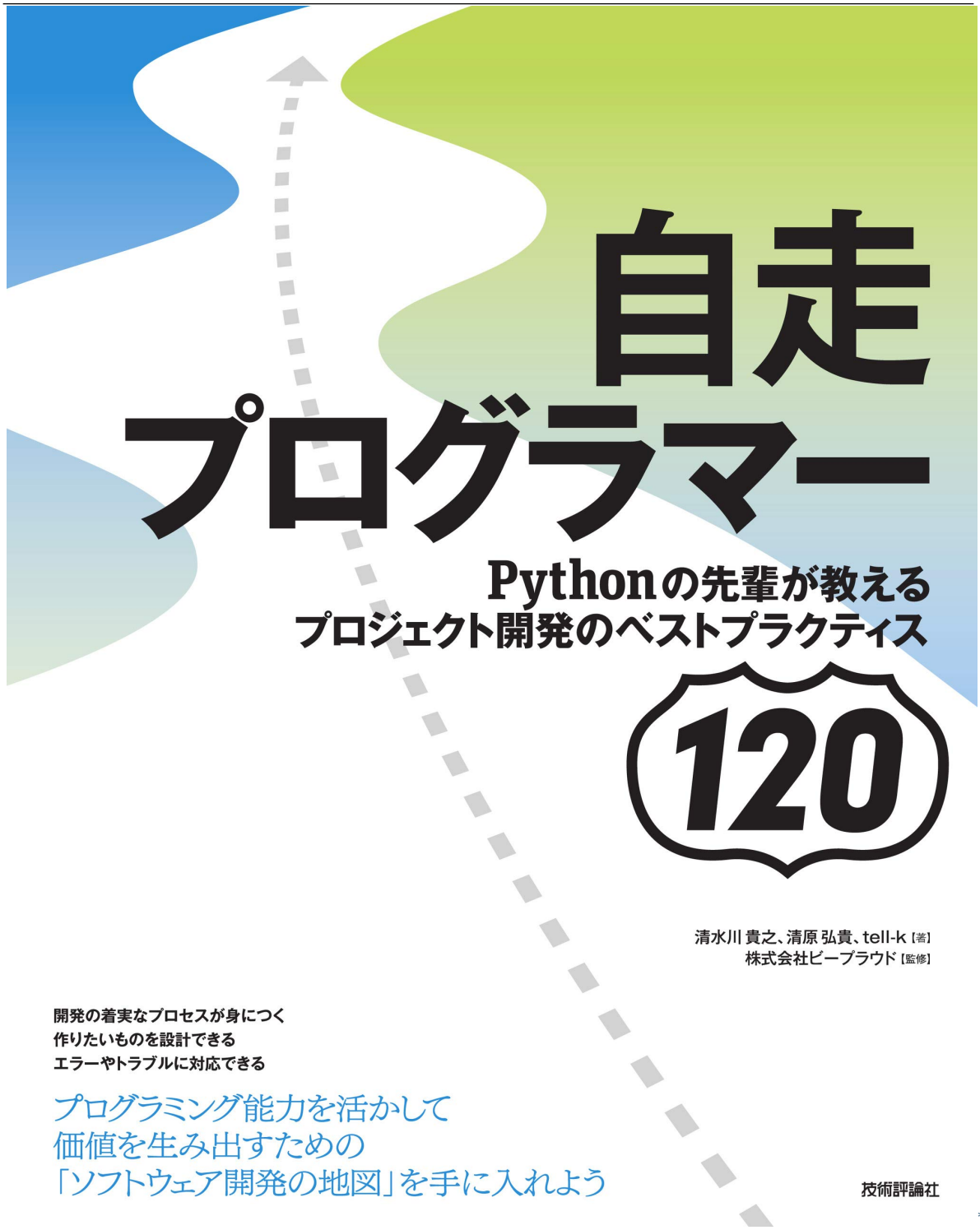
*183 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*184] をご参照ください

ではログレベルはどのように設定すべきでしょうか？ログレベルは以下を参考にしてください。

- デバッグ（ debug ）：ローカル環境で開発するときだけ使う情報
- インフォ（ info ）：プログラムの状況や変数の内容、処理するデータ数など、後から挙動を把握しやすくするために残す情報
- ワーニング（ warning ）：プログラムの処理は続いているが、何かしら良くないデータや通知すべきことについての情報
- エラー（ error ）：プログラム上の処理が中断したり、停止した場合の情報
- クリティカル（ critical ）：システム全体や連携システムに影響する重大な問題が発生した場合の情報

[*184] https://gihyo.jp/book/2020/978-4-297-11197-7



*185

*185 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*186] をご参照ください

関連

- *75:Sentry* **でエラーログを通知／監視する** (ページ 279)

4.2.6 72:ログには print でなく logger を使う

とりあえずで print を仕込んでデバッグしていませんか？ Python のロギングの仕組みを使ってより良い書き方を学びましょう。

具体的な失敗

```
def main():
    print("売上 CSV 取り込み処理を開始")
    sales_data = load_sales_csv():
    print(f"{len(sales_data)}件のデータを処理します")
```

print でのデバッグや print での実行ログも悪くはありません。ですが、環境によって切り替えができない点が不便です。

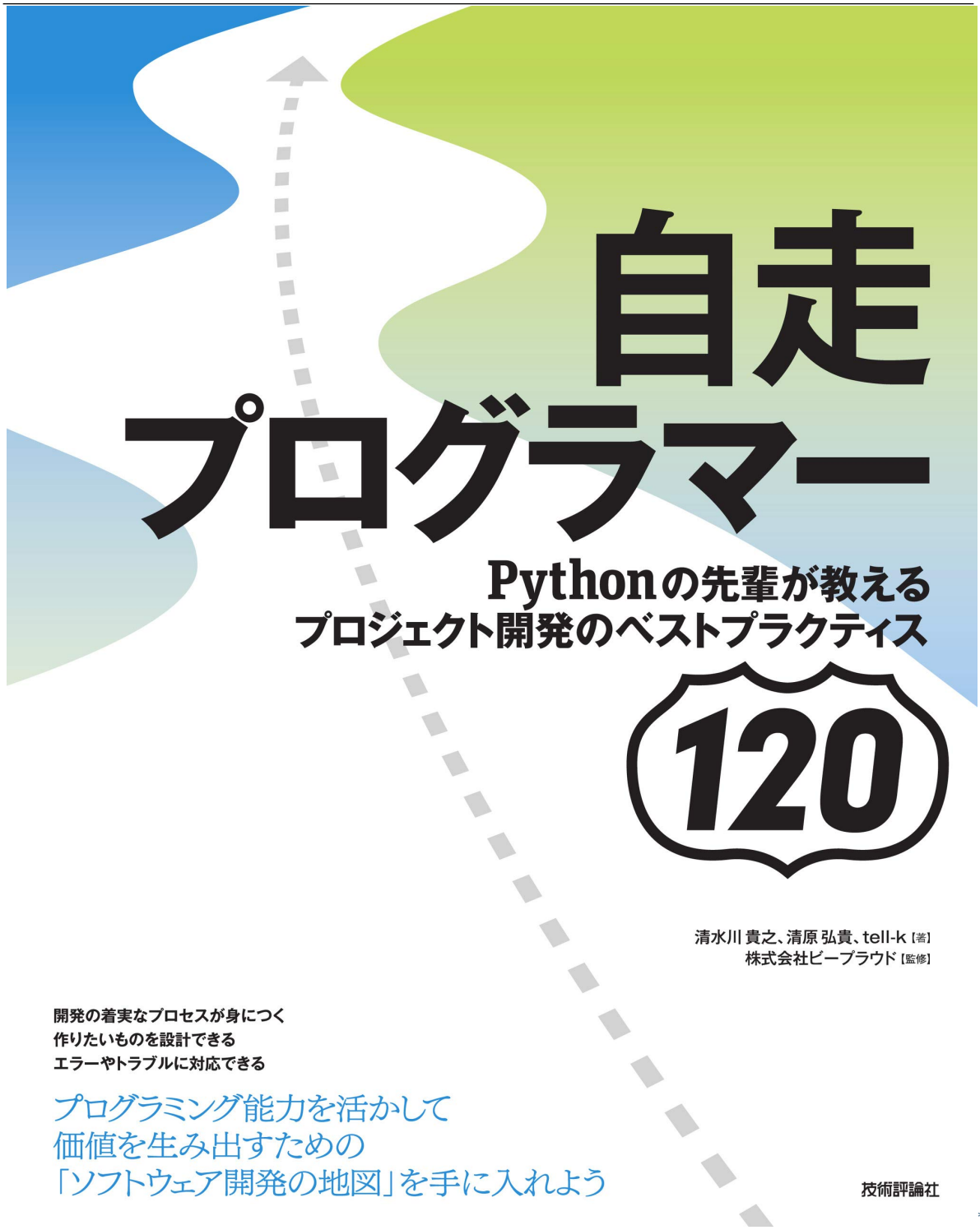
ベストプラクティス

ロギングを使うことで、より便利になります。

```
def main():
    logger.info("売上 CSV 取り込み処理を開始")
    sales_data = load_sales_csv():
    logger.info("%s 件のデータを処理します", len(sales_data))
    ...
```

ロギングを使えば、表示をやめたり、ファイルに出力したり、ログを残した日時を残したりできます。

[*186] https://gihyo.jp/book/2020/978-4-297-11197-7



*187

*187 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*188] をご参照ください

4.2.7 73:ログには 5W1H を書く

プログラミング迷子: どんな情報が必要かを知らず「とりあえず」で書かれてしまうログ出力

- 後輩 W：どこまで処理が実行されたかをログに残すように、って言われたんですけど、とりあえず関数の開始と終了をログに出したら良いですか？
- 先輩 T：うーん。関数の呼び出しだけわかっても、知りたいことはわからないよ。5W1H を書くようにしよう。

「ログに何を書くべきか」は、ロギングにおいて一番難しく、一番大切なことです。次のエラーログの問題を考えましょう。

具体的な失敗

```
def main():
    logger.info("取り込み開始")

    sales_data = load_sales_csv()
    logger.info("CSV 読み込み済み")

    for code, sales_rows in sales_data:
        logger.info("取り込み中")
        try:
            for row in sales:
                # 1 行 1 行、データを処理する
                ...
        except:
            logger.error("エラー発生")

    logger.info("取り込み処理終了")
```

[*188] https://gihyo.jp/book/2020/978-4-297-11197-7

このロギングでは、実際にエラーが発生したときに原因の特定は難しいでしょう。ログが開始と終了しか残っておらず、処理全体でエラー処理がされているからです。

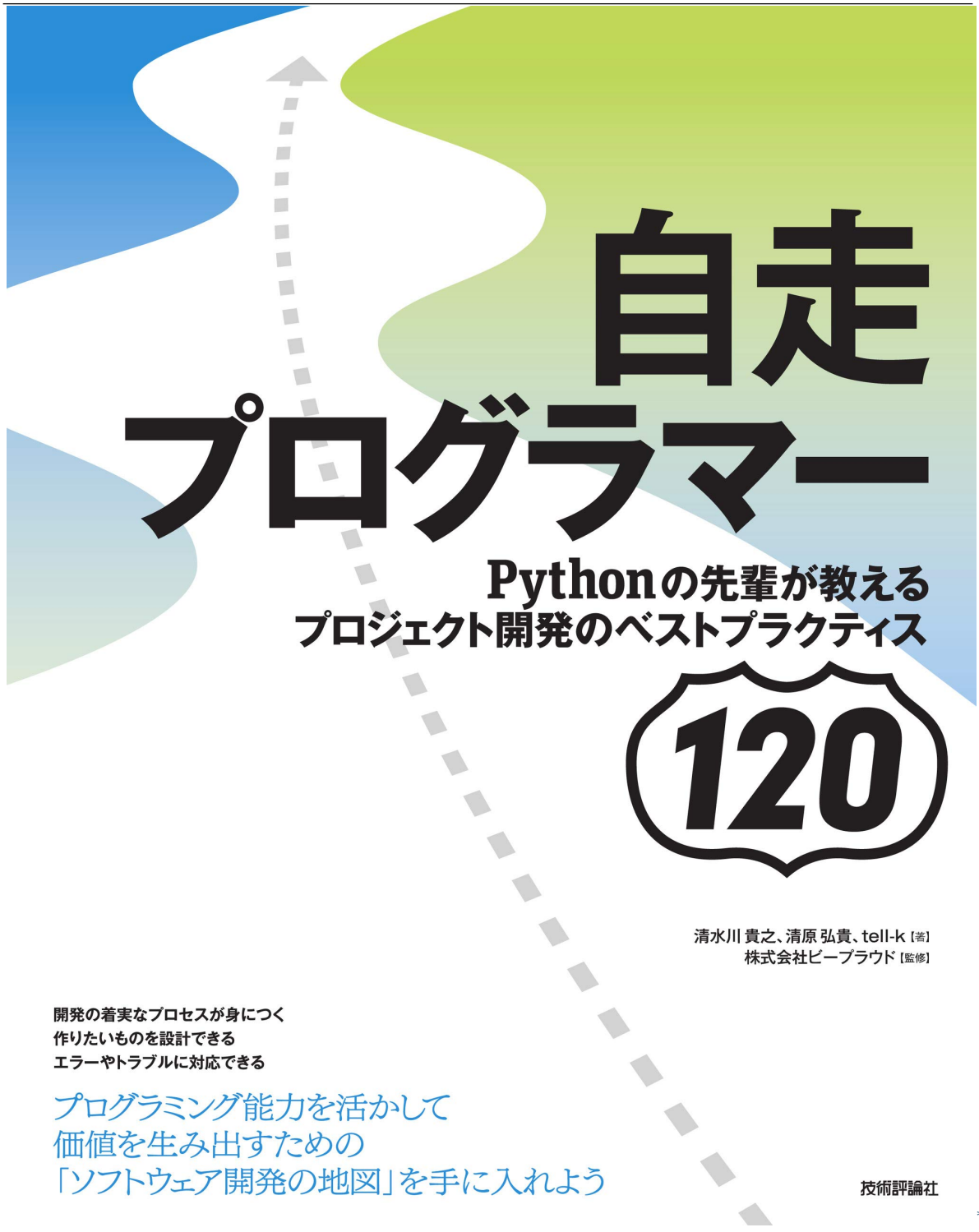
ベストプラクティス

特に長時間実行されるコマンドや、夜間実行される バッチ処理 は細かめにログを残すべきです。エラーがあった際に原因の特定が格段にやりやすくなります。

```
def main():
    try:
        logger.info("売上 CSV 取り込み処理開始")

        sales_data = load_sales_csv()
        logger.info("売上 CSV 読み込み済み")

        for code, sales_rows in sales_data:
            logger.info("取り込み開始 - 店舗コード: %s, データ件数: %s", code,␣
↪len(sales_rows))
            try:
                for i, row in enumerate(sales_rows, start=1):
                    logger.debug("取り込み処理中 - 店舗 (%s): %s 行目", code, i)
                    ...
            except Exception as exc:
                logger.warning("取り込み時エラー - 店舗 (%s) %s 行目: エラー %s", code,␣
↪i, exc, exc_info=True)
                continue
            logger.info("取り込み正常終了 - 店舗コード: %s", code)

        logger.info("売上 CSV 取り込み処理終了")
    except Exception as exc:
        logger.error("売上 CSV 取り込み処理で予期しないエラー発生: エラー %s", exc, exc_
↪info=True)
```

細かくログを残すように変更していますが、重要なバッチ処理であればこの程度は必要です。各店舗の処理毎にインフォログを（店舗コード付きで）残したり、行単位のログをワーニングログとして残すなどの工夫に注目してください。処理の トレーサビリティ を常に意識しましょう。

ログメッセージに何を書けば良いかわからないときは、次のような 5W1H を意識しましょう。



*189

*189 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*190] をご参照ください

4.2.8 74:ログファイルを管理する

自分が担当するシステムで障害やエラーが発生したときにどこのログを調査したら良いかわからないといった経験はありませんか？　ログファイルと一口に言っても、システムが扱うログファイルにはいろんな種類があります。

Web アプリケーションをサーバーまで含めて自分で管理した場合、パッと思いつくだけでも以下のようなログファイルがあるでしょう。

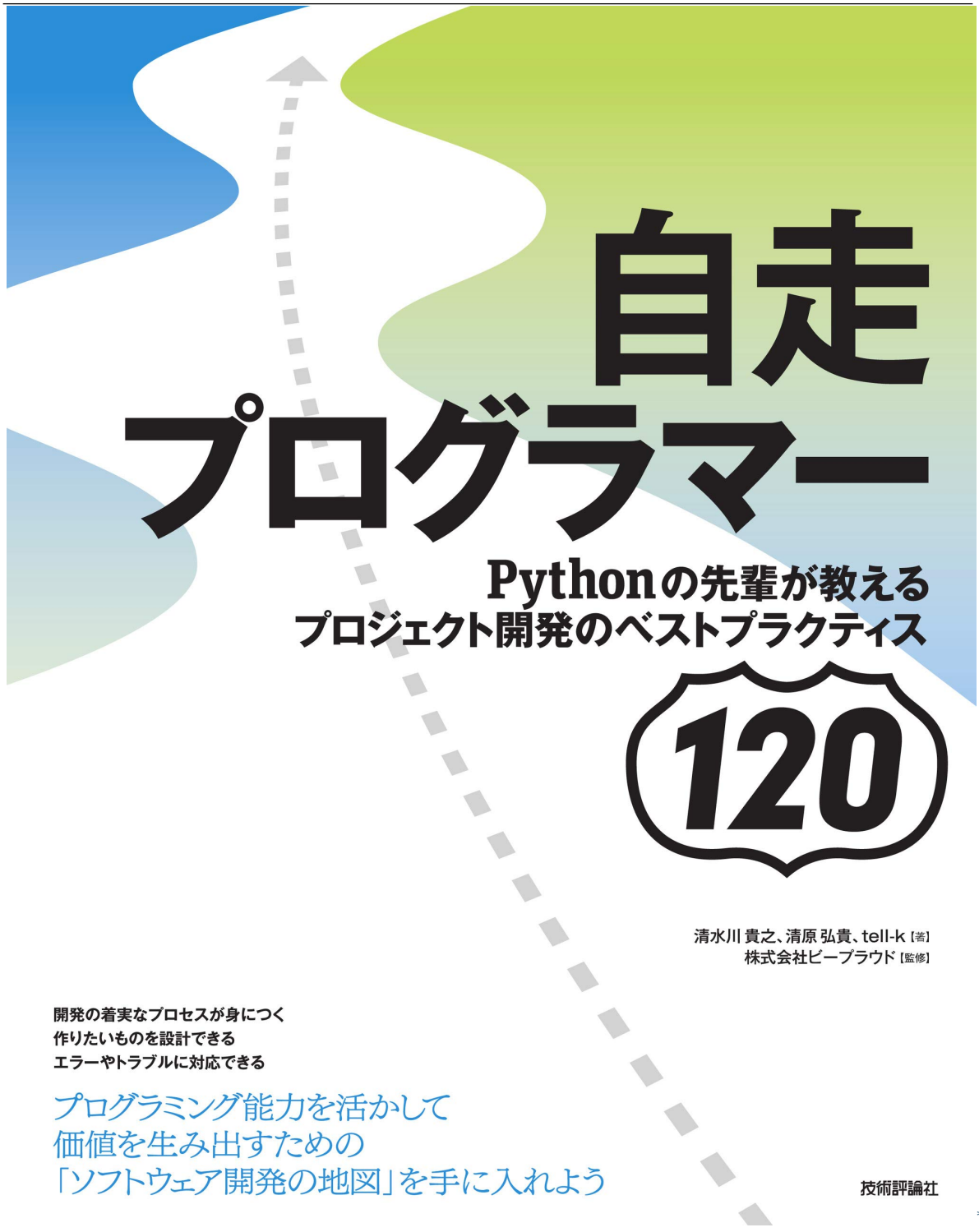
- Nginx や Apache などの Web サーバーのアクセスログ、エラーログ
- Web アプリケーションのログファイル、エラーログ
- systemd などで稼働している各種サービス、ミドルウェアのログ

システムが吐き出すログファイルにどのようなものがあるか把握することは、管理、運用するためにも大切です。

ベストプラクティス

Web アプリケーションの運用では、障害やエラーが発生したときにログを調査します。そのため障害時にも慌てないように、ログファイルがどのように管理されているのか把握しておきましょう。

[*190] https://gihyo.jp/book/2020/978-4-297-11197-7



*191

*191 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*192] をご参照ください

関連

- *75:Sentry* **でエラーログを通知／監視する** (ページ 279)

4.2.9 75:Sentry でエラーログを通知／監視する

ログを収集したものの、大量のログから必要な情報を見つけられない、といったことはありませんか？

あるいは、ログに ERROR が記録されたときに通知するように設定したために、大量の通知でメールボックスが埋め尽くされたことはありませんか？ Django にはエラー発生時に管理者にメール通知を行う機能がありますが、メールを送信しないシステムの場合は通知のためにメールサーバーを用意する必要があります。また、このエラー通知メールはエラー発生ごとに毎回送信されてしまうため、1,000 件のメールの中に非常に重要なエラー通知が 1 件紛れ込んだ場合に、その 1 通を見逃してしまうことがあります。

ベストプラクティス

エラートラッキングサービスを使いましょう。

Sentry[*193] を利用すれば、連続する同じエラーをまとめて 1 回だけ通知してくれるため、障害が発生したときに必要な情報に素早く到達できます。また、Sentry サービスにはログだけでなく、エラー発生回数や頻度、ユーザーのブラウザ情報、ブラウザから POST されたデータ、発行された SQL など、多くの情報が通知されます。こういった情報を Sentry サービス上で参照できるため、状況を素早く把握でき、問題の切り分けがスムーズに進みます。特に、DB トランザクションを使用しているシステムでは、エラーでデータがロールバックされてしまうとデータベースやログにデータの状態が残らないため問題追跡が難しくなってしまいますが、Sentry を使用していれば、POST データと発行した SQL の記録から状況を再現することも可能です。

[*192] https://gihyo.jp/book/2020/978-4-297-11197-7
[*193] https://sentry.io/

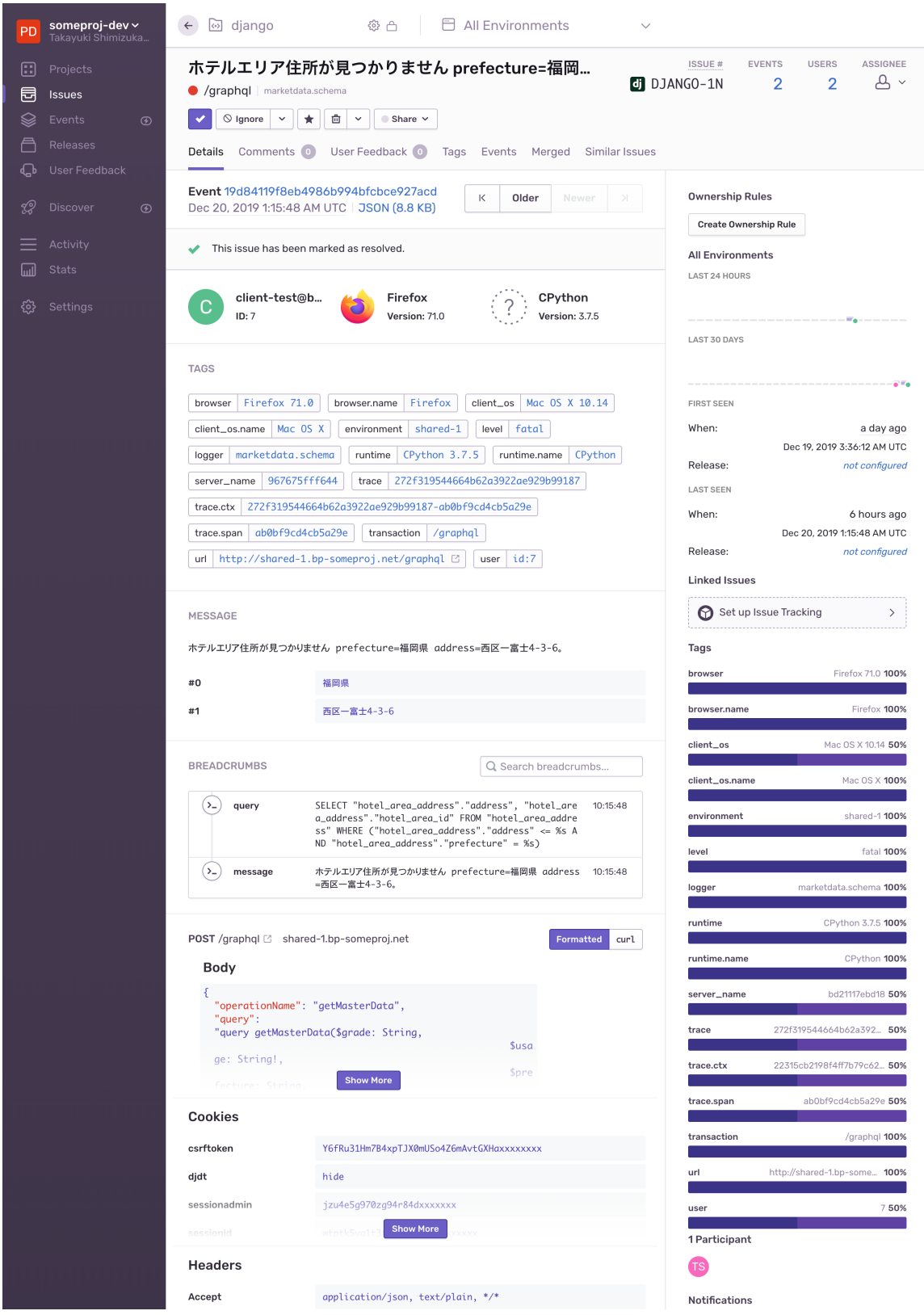
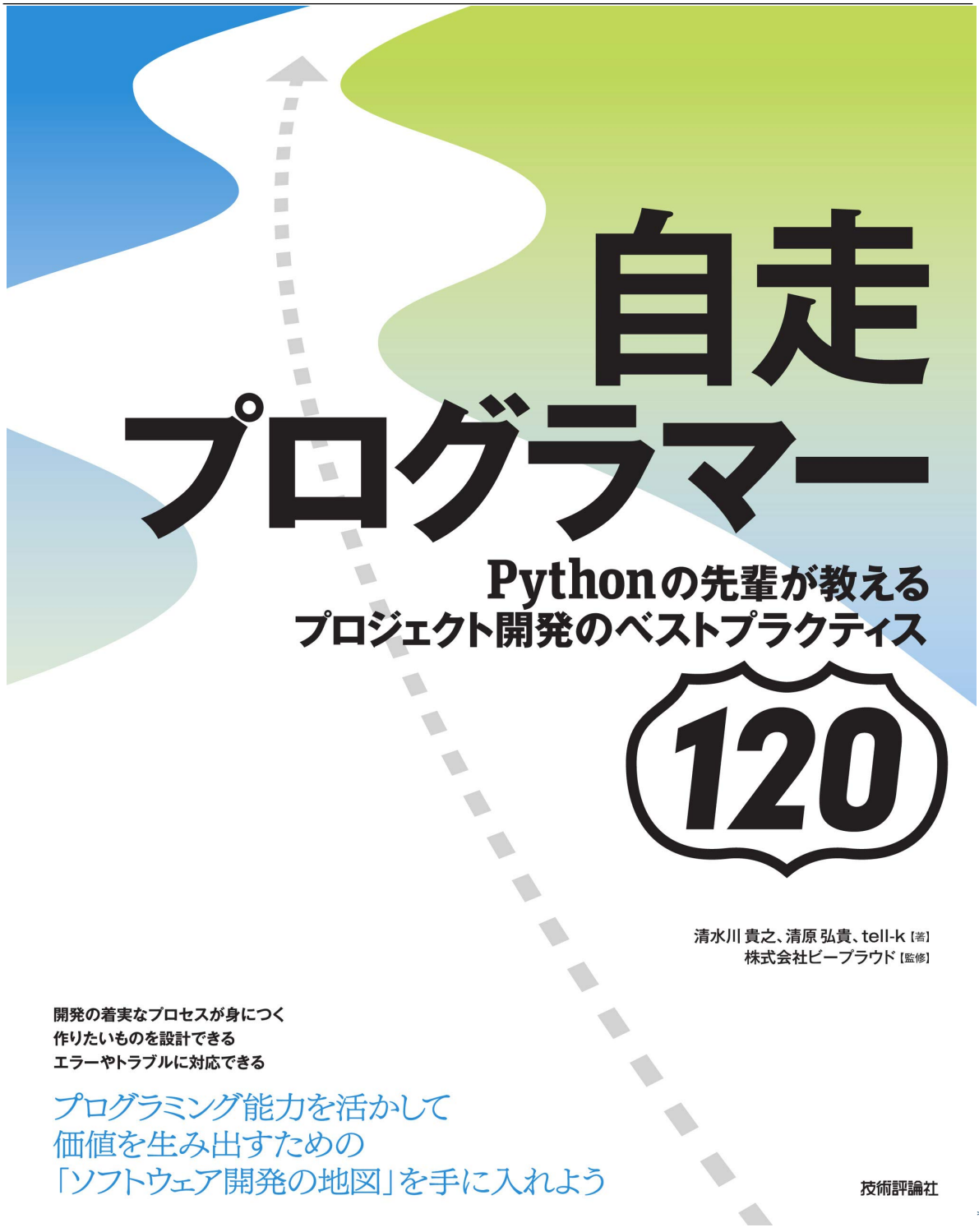


図 4.1　Sentry の通知画面例



*194

*194 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*195] をご参照ください

関連

- *69:* **ログメッセージをフォーマットしてロガーに渡さない** (ページ 261)
- *71:info*、*error* **だけでなくログレベルを使い分ける** (ページ 267)
- *72:* **ログには** *print* **でなく** *logger* **を使う** (ページ 272)

[*195] https://gihyo.jp/book/2020/978-4-297-11197-7

4.3. トラブルシューティング・デバッグ

4.3.1 76:シンプルに実装しパフォーマンスを計測して改善しよう

コードの処理速度が予想以上に遅いことはよくあることです。*60:Django ORM* **でどんな** *SQL* **が発行されているか気にしよう** (ページ 218) では、データ量に比例して遅くなる典型例をいくつか紹介しました。他にも、特定の 2 種類のリクエストを同時に受信したときだけ遅くなることもあり、原因を見つけるのがなかなか難しい問題です。

パフォーマンスの問題が発生したとき、闇雲に当たりをつけてコードを書き換えて問題が解決することは、まずありません。運良く問題が解決できても、次に似たような問題が起きたときに解決できるかどうかは運次第となってしまいます。

また、あらかじめ「ボトルネックが発生しないように実装する」のもオススメしません。ボトルネックが起こる場所を予測するのは難しく、机上では見つけづらいものです。実装時に局所的な数百ミリ秒の速度改善をしても、その改善が原因で別のボトルネックを産んでしまうことすらあります[*196] 。

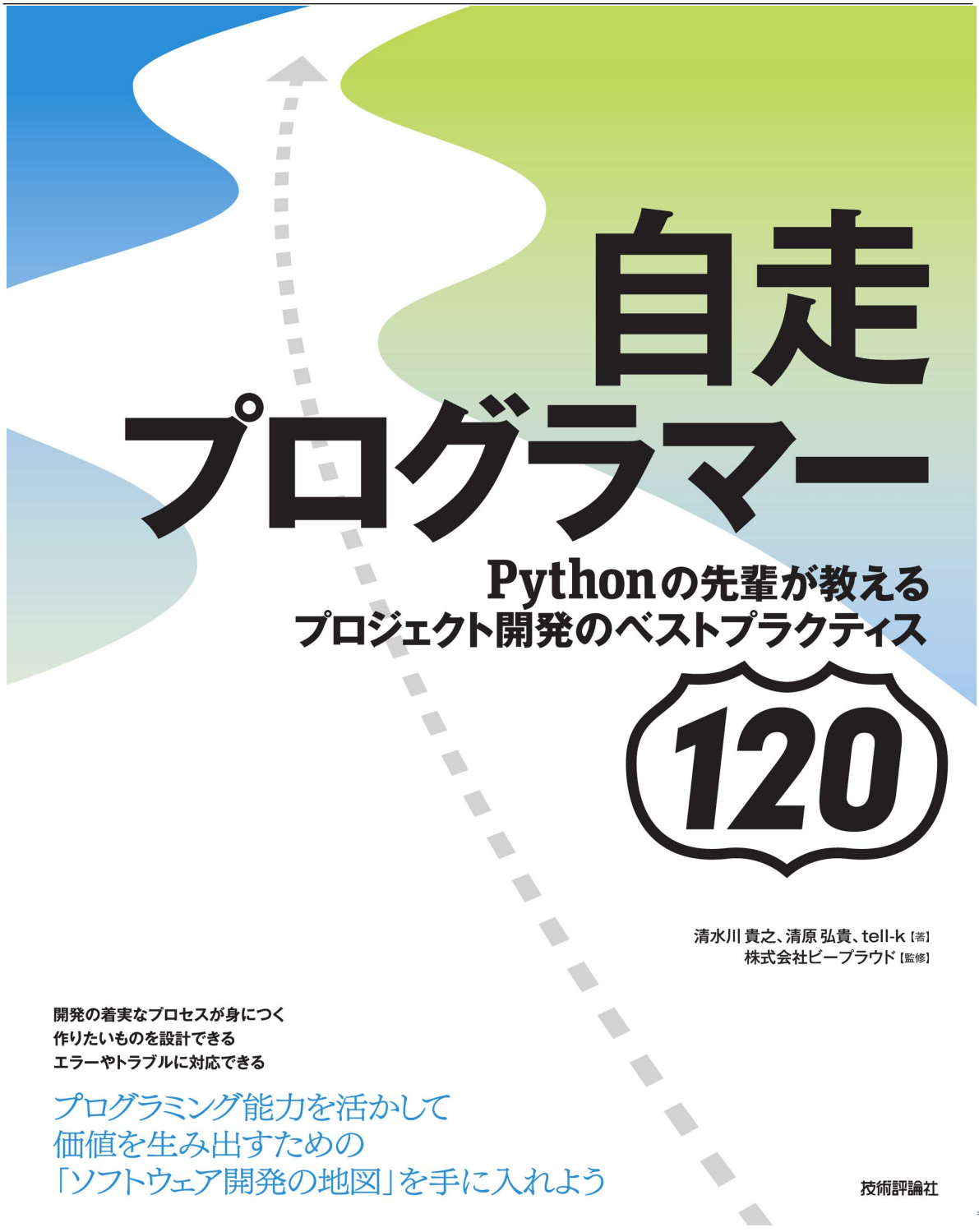
ベストプラクティス

シンプルに実装して、速度を計測して、ボトルネックを改善しましょう。

計測した速度が想定範囲内であれば、多少遅くてもそれ以上改善するべきではありません。他の有意義なことに時間を使いましょう。

速度を改善する必要がある場合、開発環境やより本番に近いデータを持つ検証環境などで実行時の情報を収集し、複数の仮説を立て、可能性を排除していく必要があります。Web アプリケーションの場合それ自体での処理の他、フロントの Web サーバーとデータベースでの処理のどこに時間がかかっているのかを見極める必要があり、これはログやリソース監視を調査することで切り分けできます。ボトルネックの見つけ方については、『Web エンジニアが知っておきたいインフラの基本』（馬場 俊彰著、マイナビ刊、2014 年 12 月）で詳しく解説されています。

[*196] 「早すぎる最適化は諸悪の根源である」『文芸的プログラミング』（ドナルド・E. クヌース著、ASCII 刊、1994 年）



*197

*197 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*198] をご参照ください

4.3.2 77:トランザクション内はなるべく短い時間で処理する

Web アプリケーションの実装で、ブラウザからのリクエスト処理開始時にデータベースのトランザクションを開始してしまうと、さまざまな問題の原因となります。データベースのトランザクションは、何か問題があった場合に中途半端なデータ更新を行わないようにするために利用されます。

具体的な失敗

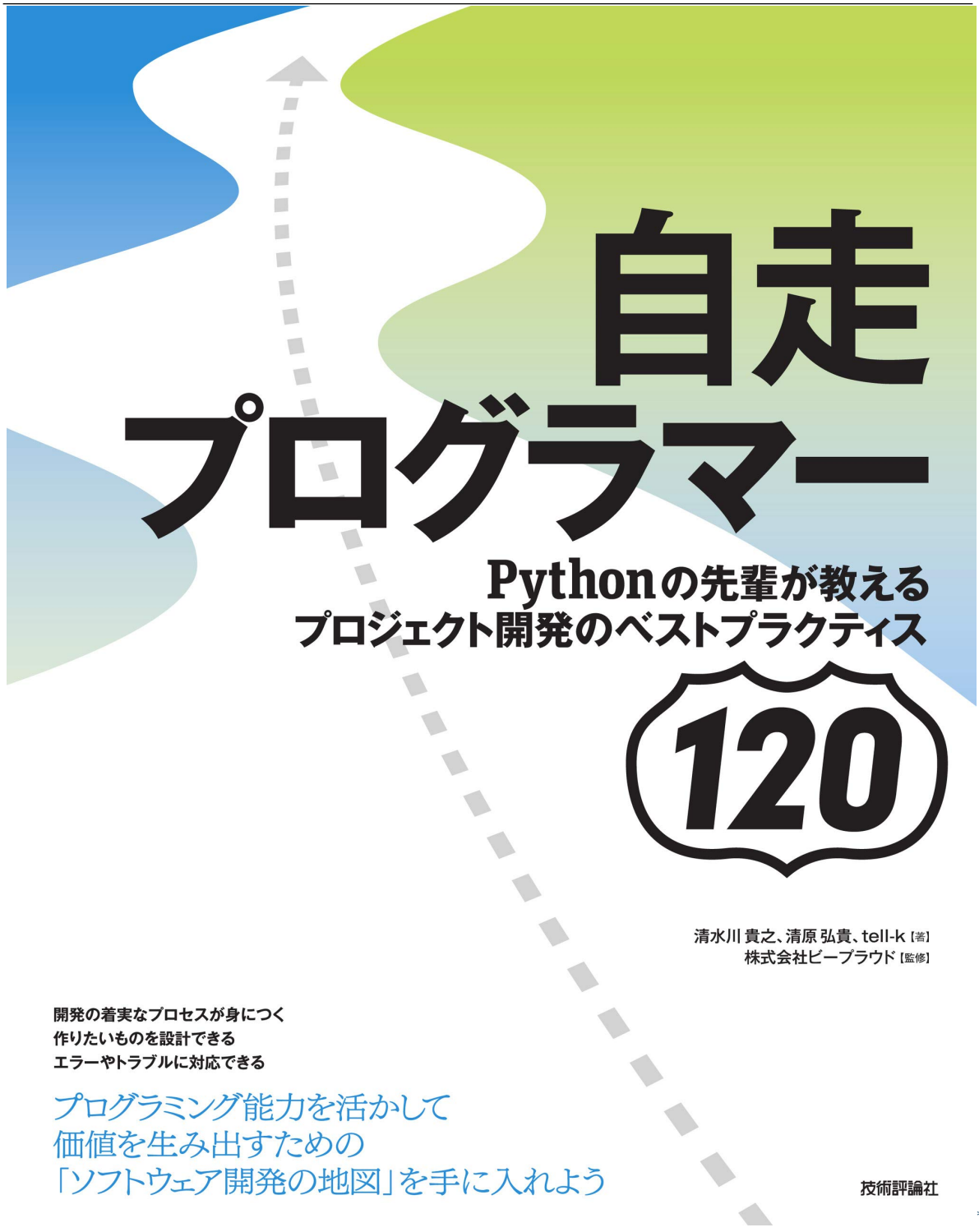
たとえば Web で商品の購入しようとしたとき、内部で何かのエラーが発生して商品の購入が失敗したのに商品の出荷が始まってしまっては困ります。こういった場合、開始したトランザクションを確定せずにロールバックすることで問題を回避します。Django では、トランザクションを開始する関数呼び出しを明示的に実装する方法と、view の呼び出し時にトランザクションを自動的に開始する設定 ATOMIC_REQUESTS があります。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        ...
        'ATOMIC_REQUESTS': True,
    }
}
```

ATOMIC_REQUESTS は便利な設定ですが、これを利用した状態では意図しないテーブルロックが発生することがあります。テーブルがロックされた場合、同時にアクセスしている他の処理ではそのテーブルの更新ができなくなり、ロック解除まで更新が待たされます。また、複数のトランザクション処理がテーブルのロックを奪い合う状況では、 デッドロック によるエラーも発生します。

このシステム障害は、アクセスが集中したり、負荷などによってリクエスト処理時間が長引くことでランダムに発生します 。しかし、開発中やシステム運用開始直後など、アクセス数が少なく負荷が低い状態ではほとんど発生しません。

[*198] https://gihyo.jp/book/2020/978-4-297-11197-7



*199

*199 https://gihyo.jp/book/2020/978-4-297-11197-7

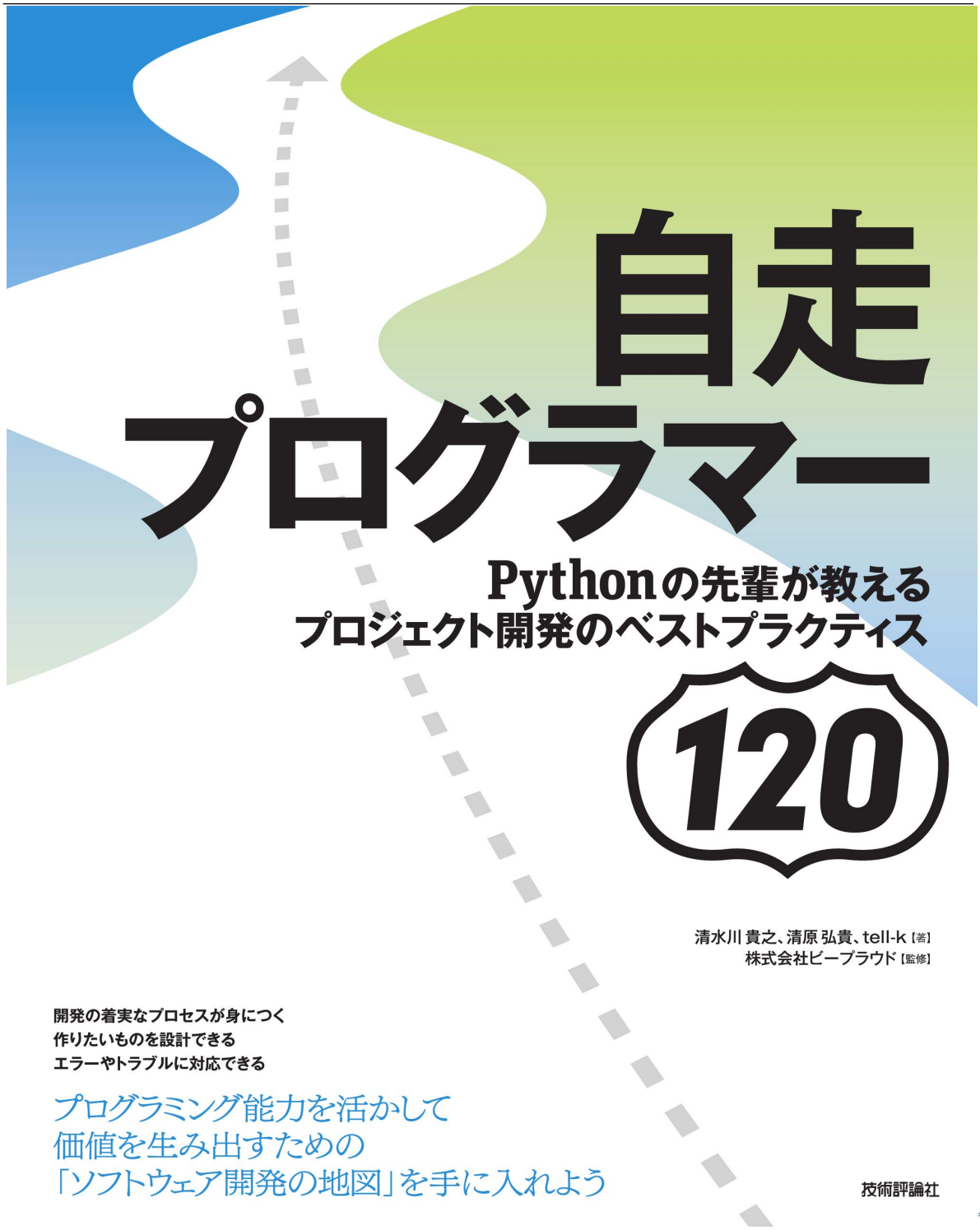
（中略）詳細は書籍 自走プログラマー[*200] をご参照ください

ベストプラクティス

トランザクション内で時間がかかる処理を行わないようにしましょう。具体的には以下の複数の観点で対策します。

- リクエスト全体をトランザクションとする場合、リクエスト処理にかかる時間を短くする
- トランザクション処理を自動にせず、必要最小限の範囲に明示的に設定する
- データベースのトランザクション分離レベルを設計時に選択する

[*200] https://gihyo.jp/book/2020/978-4-297-11197-7



[*201]

[*201] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*202] をご参照ください

関連

- *91:*時間のかかる処理は非同期化しよう (ページ 330)

4.3.3 78:ソースコードの更新が確実に動作に反映される工夫をしよう

リリース作業中のトラブルシューティングなど、時間が限られている状況ではリリース先の環境を直接使って問題の原因を調査することがあります。リリース中に見つかった不具合をその場で修正するなどということは絶対に避けるべきですが、その環境でしか収集できない情報や、修正対応しなければロールバックすらできない状況もごく稀にあるものです。たとえば検証環境特有のデータ不整合が原因と想定される場合、開発環境で問題を再現させるための情報などをその場で収集するため、Python コードを直接書き換えてデバッグログを追加したり、調査結果を元に確認のためにコードを書き換えたりします。しかし、追加したはずのデバッグログが出力されなかったらどうでしょう？「ログを追加した関数には処理が来ていない」と考えるのではないでしょうか。

このような現象に遭遇したときは、基本的なところで間違えている可能性があります。

- 似た名前の別のファイルを編集している
- 修正した `.py` ファイルよりもタイムスタンプが新しい、修正前の `.pyc` ファイルが使われている
- ファイルを修正したあとプロセスを再起動していない
- アクセスしているサーバーが異なる

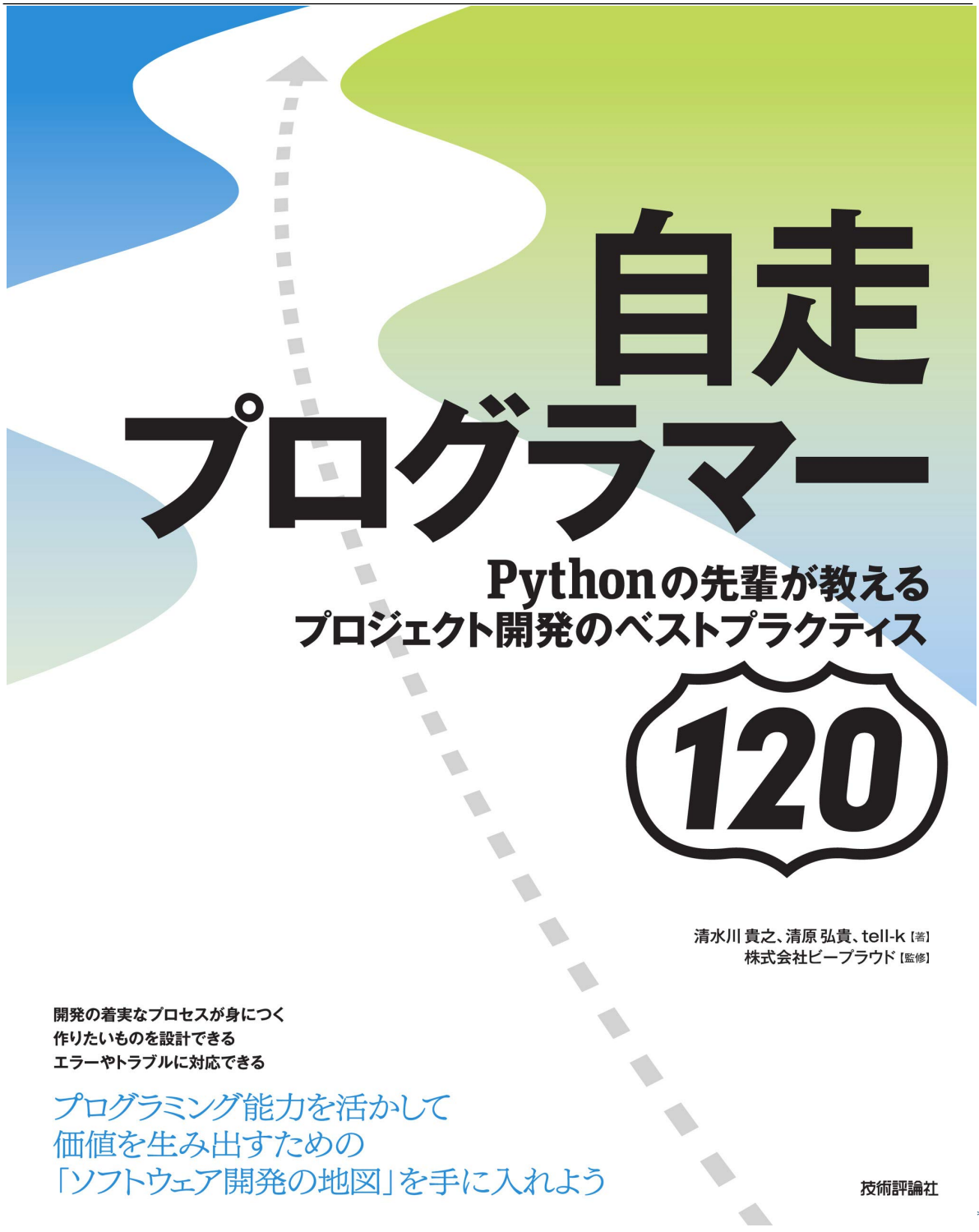
時間が限られている状況では、普段と異なる手順での作業を行うことによる緊張感もあり、ちょっとした見落としをしてしまったり、想定外の動作に惑わされたりします。

ベストプラクティス

つまづかないための工夫をしましょう。

目的とは別のファイルを編集してしまうことは意外とあります。安全のために対象ファイルをバックアップ目的で複製して、間違えて複製したほうを編集していることもあります。落ち着いて、現在編集しているファイルが想定どおりのパスのファイルかを確認しましょう。

[*202] https://gihyo.jp/book/2020/978-4-297-11197-7



*203

*203 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*204] をご参照ください

[*204] https://gihyo.jp/book/2020/978-4-297-11197-7

第 5 章

システム設計

5.1. プロジェクト構成

5.1.1 79:本番環境はシンプルな仕組みで構築する

プログラミング迷子**:** 多機能なツールを選んでおけば安心？

- 後輩 W：Python の環境作るときって、pyenv[*205] と pipenv[*206] のどっちを使ったらいいんですかね？
- 先輩 T：お、その 2 択なんだ？　pyenv や pipenv が必要だと思ったのは何で？
- 後輩 W：Python の環境構築で調べたら、pyenv と pipenv がたくさん見つかったので。
- 先輩 T：なるほど。でも個人環境はともかく、本番環境で pyenv や pipenv を使うのは避けたほうが良いんじゃないかな。
- 後輩 W：えっ、全部の環境で同じツールが使えたほうが楽じゃないですか。
- 先輩 T：なるほど。多機能なツールはどんな問題も解決できる気がしてくるけど、ちょっとそれぞれの目的を考えてみようか。

OS の種類、Python の種類、Python のインストール方法、ライブラリのインストール方法など、Python を使えるように環境構築する組合せは無数にあります。そのため、選択に迷うこともあるでしょう。

選び方として良くないのは、組合せのどれかを使い慣れている、知っているから、という理由ですべての環境でツールを固定してしまうことです。個人の開発環境で使い慣れたものが本番環境に適しているとは限りませんし、多機能なら良いわけでもありません。逆に、機能を制約しすぎると個人の環境が使いにくくなってしまい、開発効率に影響することもあります。

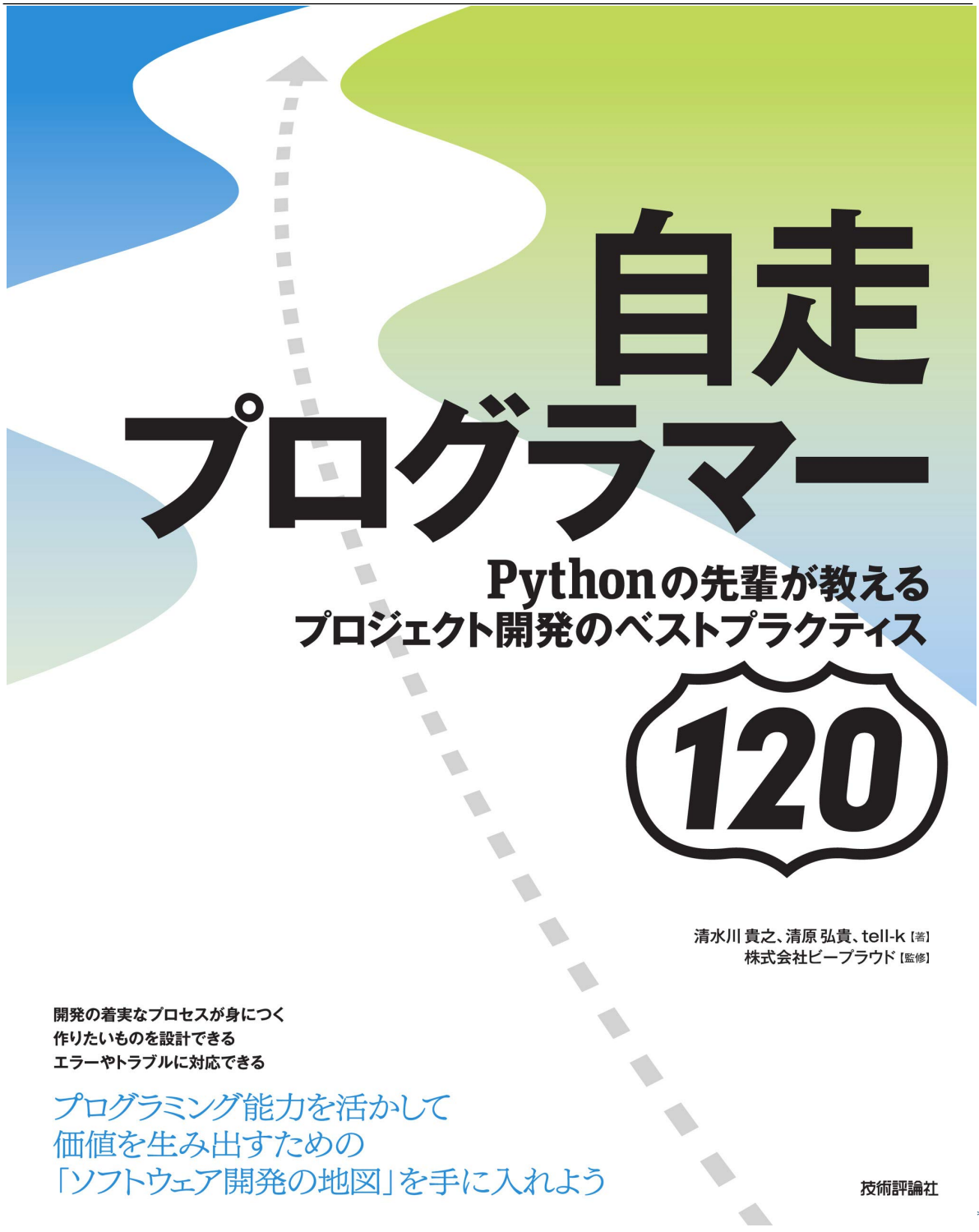
便利さを高めると、シンプルから遠ざかっていきます。pyenv? 、 pipenv[*206] 、 virtualenvwrapper[*207] 、poetry[*208] などが提供する機能が便利でも、便利な機能のために仕組みは複雑化していきます。本番環境を複雑な仕組みで構築してしまうと、トラブル解決にその分時間がかかってしまいます。便利な機能が本番環境にも必要かどうか、シンプルな代替手段がないかはよく検討しましょう。

[*205] https://github.com/pyenv/pyenv
[*206] https://pipenv.kennethreitz.org/
[*207] https://virtualenvwrapper.readthedocs.io/
[*208] https://python-poetry.org/

ベストプラクティス

本番環境は、機能をシンプルに保ち、必要最小限の仕組みで揃えましょう。本番環境にたくさんの機能を持たせると問題発生時の切り分けが難しくなり、セキュリティー上の心配も増えていきます。

このとき、個人環境と本番環境を統一することに固執してはいけません。本番環境や個人環境の目的に合わせて、それぞれ最適な方法を選択しましょう。



*209

*209 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*210] をご参照ください

5.1.2 80:OS が提供する Python を使う

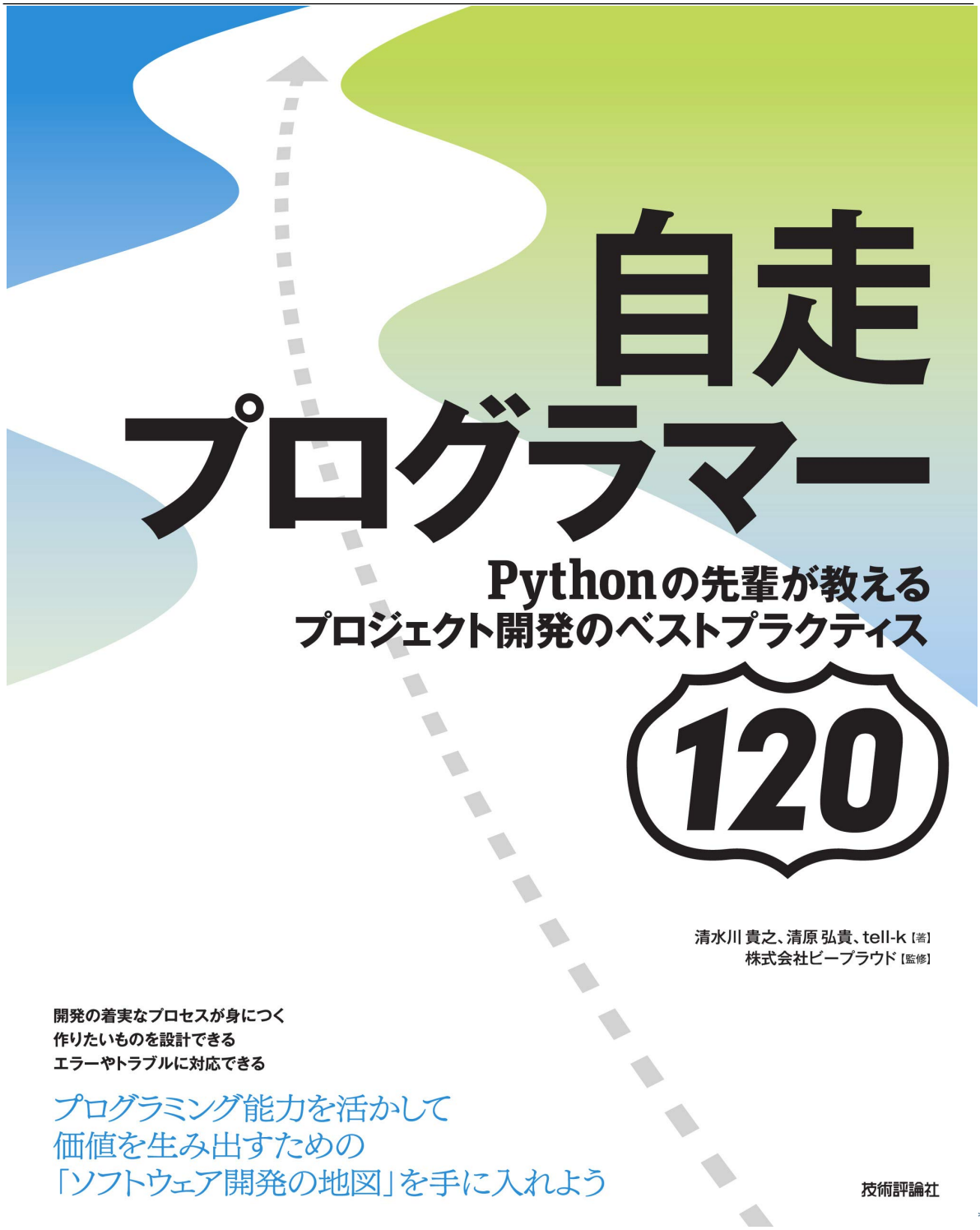
OS が提供する Python を利用するメリット、デメリットは以下のとおりです。

- ○ セキュリティー更新情報が発信されている
- ○ セキュリティー更新があることが apt や yum コマンドでわかるようになっている
- ○ 更新の適用と互換性の確認コストが低い。更新パッチが配布されていて、互換性が維持される
- × Python の最新バージョンを使用できない

ベストプラクティス

OS が提供する Python を使って、運用コストを下げつつ、セキュリティー更新していきましょう。Ubuntu であれば、apt でインストールできる公式の Python を、 RedHat Enterprise Linux（RHEL）であれば、yum や dnf でインストールできる公式の Python を選択します。

[*210] https://gihyo.jp/book/2020/978-4-297-11197-7



*211

*211 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*212] をご参照ください

5.1.3 81:OS 標準以外の Python を使う

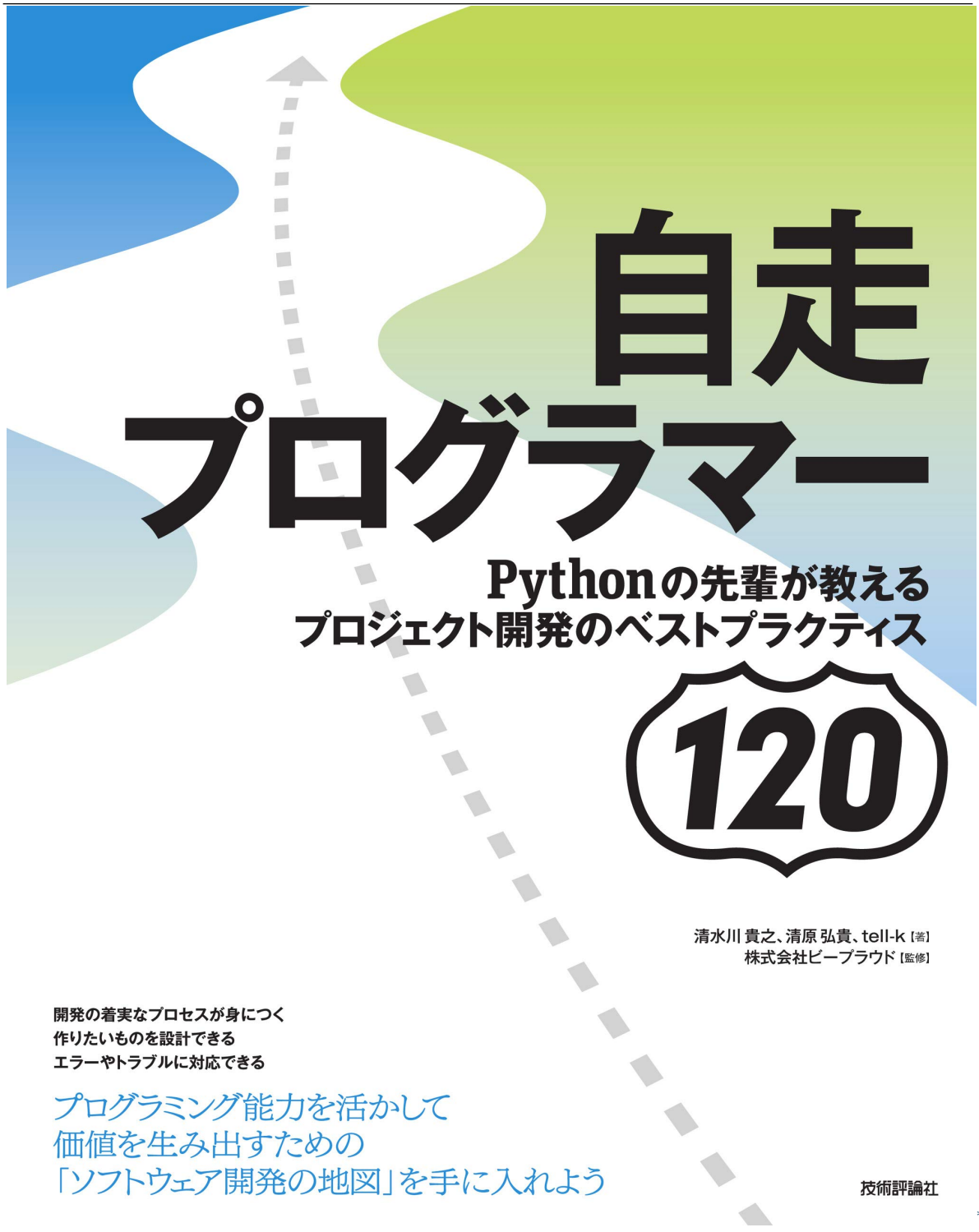
OS 標準以外の Python を利用するメリット、デメリットは以下のとおりです。

- ○ Python の好きなバージョン、配布元を選べる
- △ セキュリティー更新の確認は独自に行う
- × 再インストールと動作確認が必要なため、更新の適用と互換性の確認コストが高い

ベストプラクティス

使用したい Python バージョンが OS で提供されていない場合は、OS 標準以外の Python を選択します。ただし、デメリットに注意してください。

[*212] https://gihyo.jp/book/2020/978-4-297-11197-7



*213

*213 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*214] をご参照ください

5.1.4 82:Docker 公式の Python を使う

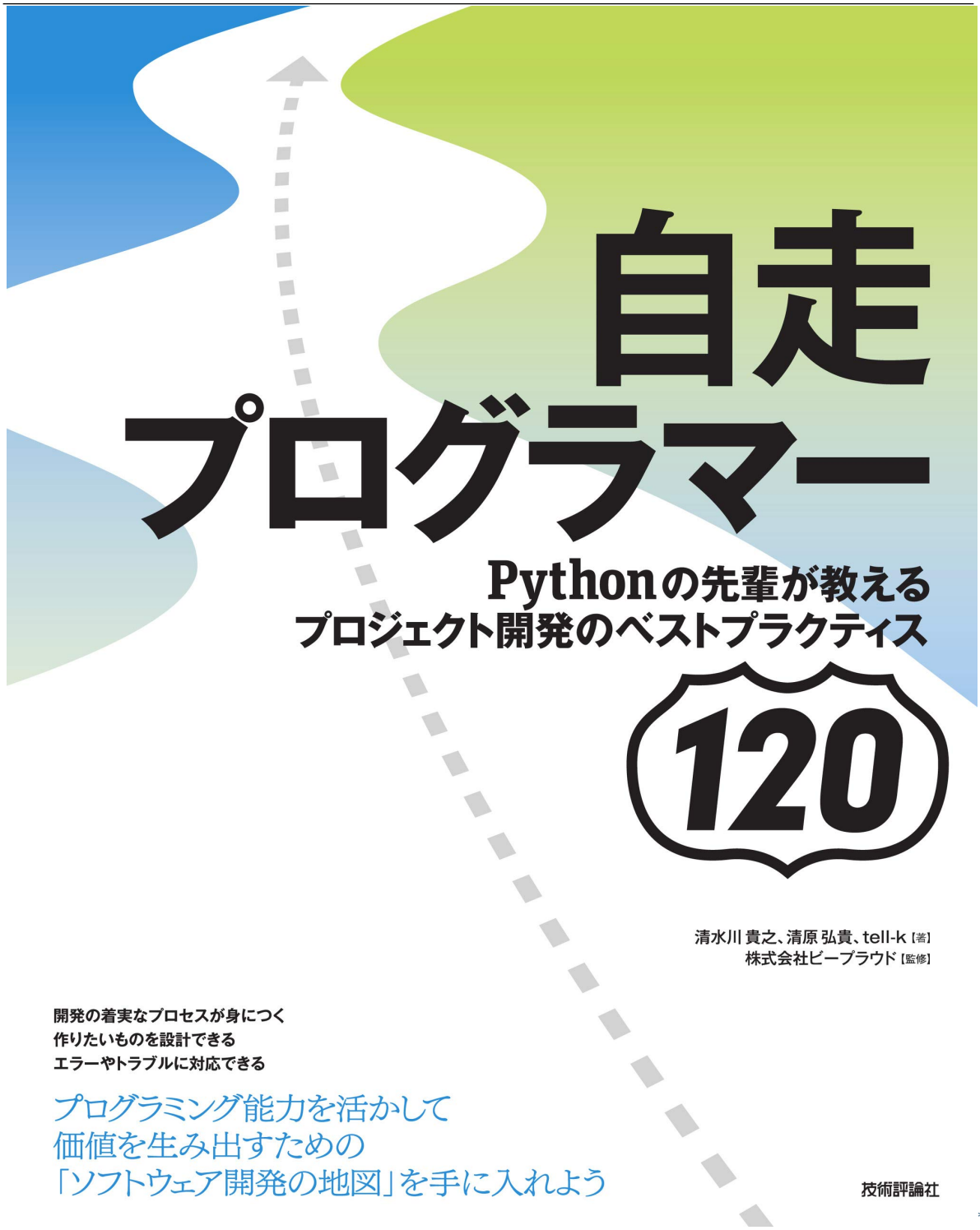
Docker 公式の Python を利用するメリット、デメリットは以下のとおりです。

- ○ Python の好きなバージョンを選べる（2016 年以降のすべてのバージョンが提供されている）
- △ セキュリティー更新の確認は各自で行う
- △ 更新の適用と互換性の確認コストが低〜中程度。コンテナの入れ替え、差分の影響確認が必要

ベストプラクティス

Docker 公式の Python を使って、運用コストを下げつつ、セキュリティー更新していきましょう。Docker 公式の DockerHub[*215] に Python の Docker Image があります。

[*214] https://gihyo.jp/book/2020/978-4-297-11197-7
[*215] https://hub.docker.com/_/python



*216

*216 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*217] をご参照ください

5.1.5 83:Python の仮想環境を使う

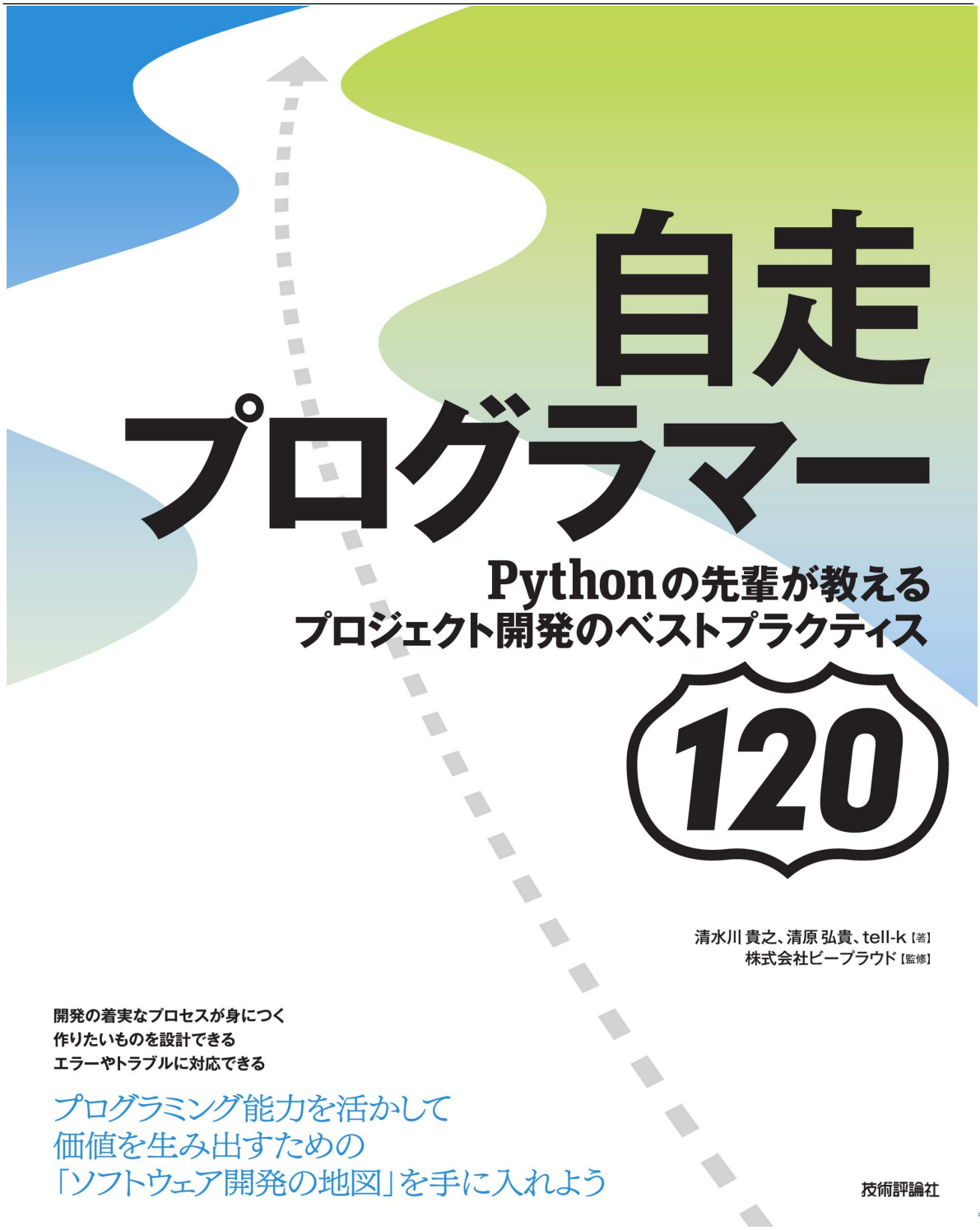
Python の仮想環境を利用するメリット、デメリットは以下のとおりです。

- ○ 仮想化した環境にインストールするため、OS の Python を変更せずに済む
- ○ 仮想化した環境の作り直しは、簡単に行える
- × Docker コンテナを利用する場合は、仮想化が二重化されてしまうため冗長

ベストプラクティス

Python の仮想環境を使って、プログラムの実行環境を Python 本体から切り離しましょう。Python の仮想環境は、Python ライブラリのインストールを独立した環境に閉じ込めて、ライブラリバージョンの競合を避けて、環境の再構築をしやすくする技術です。Python3 標準ライブラリの venv をはじめ、 virtualenv 、 pyenv 、 conda などがこの機能を提供しています。また、 pipenv や poetry など、内部で venv を利用して仮想環境を提供するツールがあります。

[*217] https://gihyo.jp/book/2020/978-4-297-11197-7



*218

*218 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*219] をご参照ください

5.1.6 84:リポジトリのルートディレクトリはシンプルに構成する

プログラミング迷子: リポジトリルートにファイルがたくさんありすぎる

- 後輩 W：先輩、引き継いだプロジェクトのリポジトリなんですけど、ルートディレクトリにファイルがありすぎて何から手をつけて良いかわからないんです。
- 先輩 T：README ファイルはある？　あればそこに説明が書いてあるんじゃない？
- 後輩 W：README にはファイルの説明は書いてなくて、ssh の秘密鍵の作り方と、Vagrant[*220] と Docker[*221] のインストール方法が書いてありました。
- 先輩 T：まじか……。それで、ルートディレクトリにはどんなファイルとディレクトリがあるの？

リポジトリのルートディレクトリは油断すると多くのファイルが置かれてしまいます。特に最近では、多くのツールやサービスがリポジトリのルートディレクトリにある特定のファイル名で動作を設定できるようになっているため、ルートディレクトリは何でも置き場になってしまう傾向があります。

具体的な失敗

リポジトリルートに以下のようなファイルやディレクトリがあると、それぞれの用途を短時間で把握するのは難しいでしょう。

```
.circleci/              config/                 manage.py
CHANGELOG.md            deploy.md               package-lock.json
Makefile                deployment/             package.json
Pipfile                 docker/                 pull_request_template.md
Pipfile.lock            docker-compose.local.yml static/
README.md               docker-compose.yml      templates/
Vagrantfile             file/                   test.md
accounts/               front/                  tests/
```

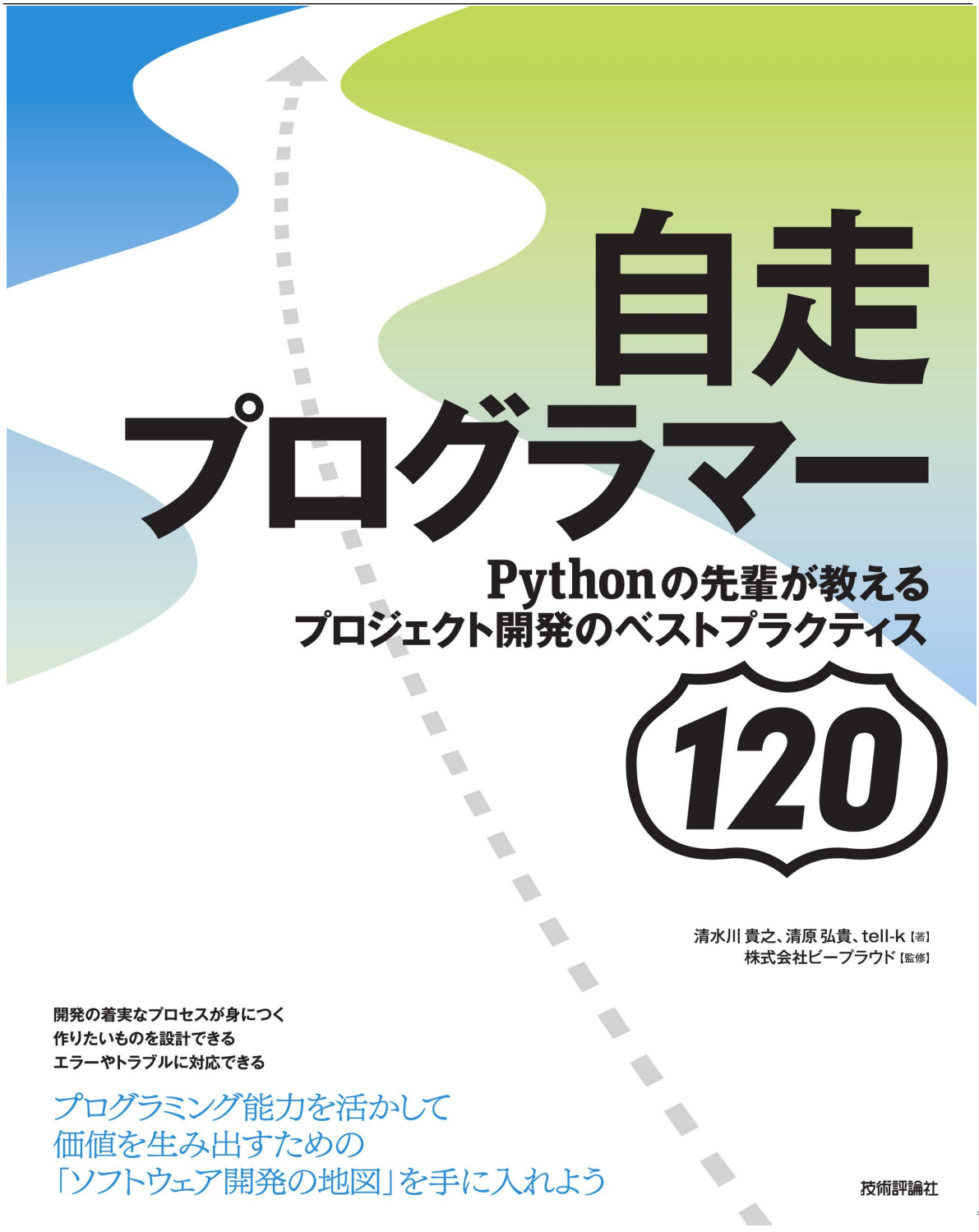
(次のページに続く)

[*219] https://gihyo.jp/book/2020/978-4-297-11197-7
[*220] https://www.vagrantup.com/
[*221] https://www.docker.com/

(前のページからの続き)

```
api/                    help/                       tox.ini
batch/                  issue_template.md
changelog/              log/
```

このリポジトリルートは、いろいろな目的のファイルが全部入り状態になってしまっているため、扱いにくい状態です。この状態でファイル構成の説明を README に書いても、焼け石に水です。一度このような状態になってしまうと変更の影響範囲が予想できないため、構造の整理整頓に手間がかかります。



[*222]

[*222] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*223] をご参照ください

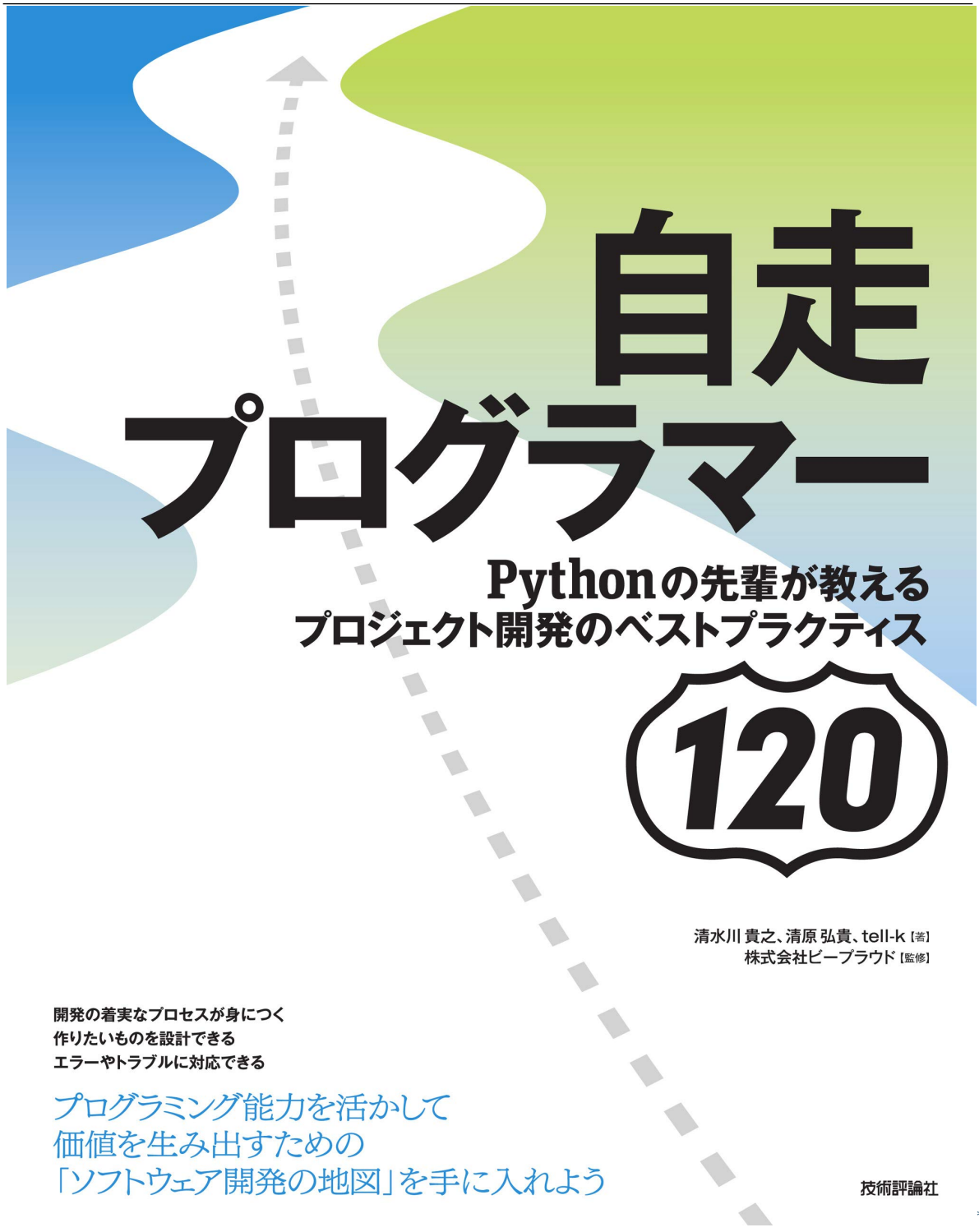
ベストプラクティス

リポジトリのルートディレクトリには、リポジトリの主目的に合った、見た人に注目してほしいファイルやディレクトリだけを置きましょう。たとえば、リポジトリの主目的が PyPI に公開する Python のパッケージであれば、 README と LICENSE の他に、パッケージングに必須となる setup.py や pyproject.toml などの設定ファイルを置くのが一般的です。こういったファイルがルートディレクトリにあれば、リポジトリを見た人は README を詳しく読まなくてもリポジトリの目的を把握できます。

リスト 5.1　重要度に応じて誘導するように整理しましょう

```
.circleci/      Makefile        changelog/      doc/
.github/        README.md       deployment/     docker/
CHANGELOG.md    Vagrantfile     djangoapp/      vueapp/
```

[*223] https://gihyo.jp/book/2020/978-4-297-11197-7



*224

*224 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*225] をご参照ください

5.1.7 85:設定ファイルを環境別に分割する

プログラミング迷子: 設定ファイルが **1** つ

- 後輩 W：Django アプリで環境別の設定を用意するのは、 `settings.py` をコピーして値を変えれば良いんでしょうか？
- 先輩 T：そうだね、Django は使用する設定ファイルをオプションで指定できるからね。でもコピーしちゃうと同じような変更を複数のファイルに書かないといけなくなるんじゃないかな。
- 後輩 W：はい、まさにそれが面倒だなと思って。他に良い方法がありますか？
- 先輩 T： `base.py` に共通の設定を書いて、環境別の設定で継承すると良いよ。

プログラムの設定を環境別に分けて用意することは、Django に限らず他の Web アプリケーションフレームワークや Web 以外のアプリでも行われます。たとえば、本番環境（production.py）と動作確認環境（staging.py）では設定が異なりますし、共有の開発環境（dev.py）や個人開発環境（local.py）、テスト実行時（test.py）などで設定をそれぞれ変える必要があります。

具体的な失敗

プロジェクト開始時は、1 つの設定ファイルから始まります。Django であれば、設定ファイル `settings.py` は `django-admin startproject` で自動生成されます。

リスト 5.2　`settings.py`

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
DEBUG = True
ALLOWED_HOSTS = []
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    ...
```

(次のページに続く)

[*225] https://gihyo.jp/book/2020/978-4-297-11197-7

(前のページからの続き)

```
    'myapp',
]
# MIDDLEWARE = [...]
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
# 以下省略
```

開発が進むにつれて、動作確認環境を用意することになったとします。動作確認環境ではデータベースに PostgreSQL を使い、デバッグ用画面は使わないことにします。このため、 settings.py を複製して settings_staging.py を作成し、 DEBUG と DATABASES の値だけ書き換えます。

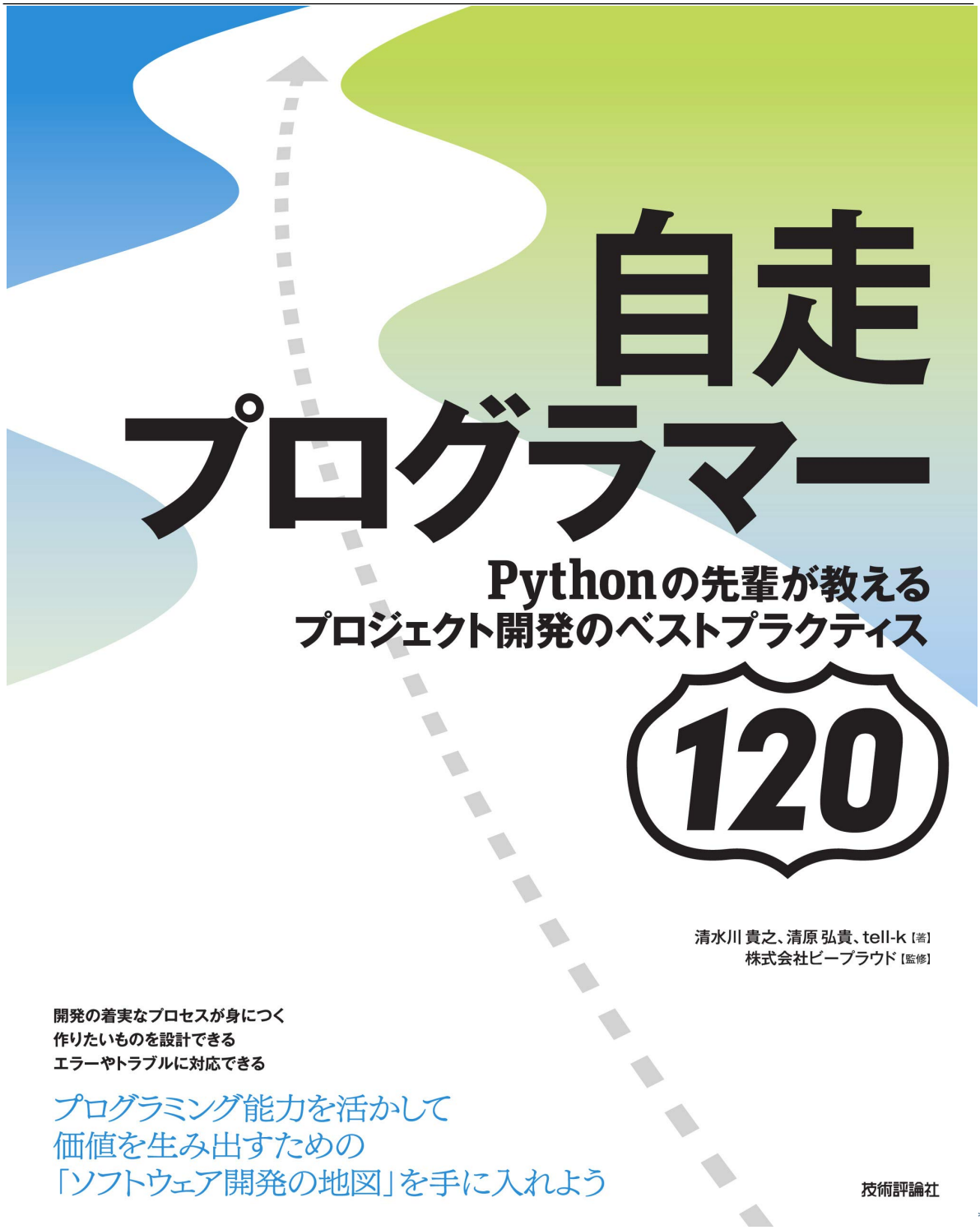
そして、Django が動作確認環境用の設定で起動するように、環境変数 DJANGO_SETTINGS_MODULE=settings_staging.py を設定して起動することにします[*226] 。

この方法はシンプルですが、多くの同じ設定を 2 つのファイルに持つことになります。このため、設定変更を行う場合は 2 つのファイルに同じような変更を行う必要があります。本番環境やテスト用設定など他の環境が増えると、この手間はさらに増えていき、修正漏れなどの原因になってしまいます。

ベストプラクティス

環境別設定 のために、設定ファイルを共通部分と環境依存部分に分割しましょう。

[*226] https://docs.djangoproject.com/ja/2.2/topics/settings/



*227

*227 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*228] をご参照ください

これで、base.py 、 local.py 、 staging.py の 3 つに分割されました。この方針で進めると、他にあと 2 つ、本番環境用の production.py とテスト実行時用の test.py が作られることになるでしょう。こうすることで、設定変更のほとんどは base.py を書き換えるだけで済み、環境別の設定は環境名のファイルを変更すれば済むようになります。たとえば、ローカル環境用に django-silk[*229] を追加するには local.py だけを変更します。

リスト 5.3　settings/local.py

```
from .base import *  # base.py のデフォルト設定を読み込み

INSTALLED_APPS.append('silk')  # 追加
MIDDLEWARE.append('silk.middleware.SilkyMiddleware')  # 追加
INTERNAL_IPS = ['127.0.0.1']
```

本節では、環境依存の設定値を分割管理する方法について説明しました。次の *86:*状況依存の設定を環境変数に分離する (ページ 313) では、状況によって変更したい設定値の扱い方について説明します。

5.1.8 86:状況依存の設定を環境変数に分離する

プログラミング迷子**:** 多段継承した設定ファイル

- 後輩 W：先輩、個人用の環境設定が必要になったら、設定ファイルを追加して from .local import * すれば良いですか？
- 先輩 T：どうして設定を追加したいの？
- 後輩 W：local.py で追加している silk を外すと少し動作が軽くなるので、自分の環境では解除しようかと思ってます。
- 先輩 T：もしかして、継承して INSTALLED_APPS から削除しようとしてる？　多段継承して差分実装を繰り返すのは良くないパターンだよ。別の方法を検討しよう。

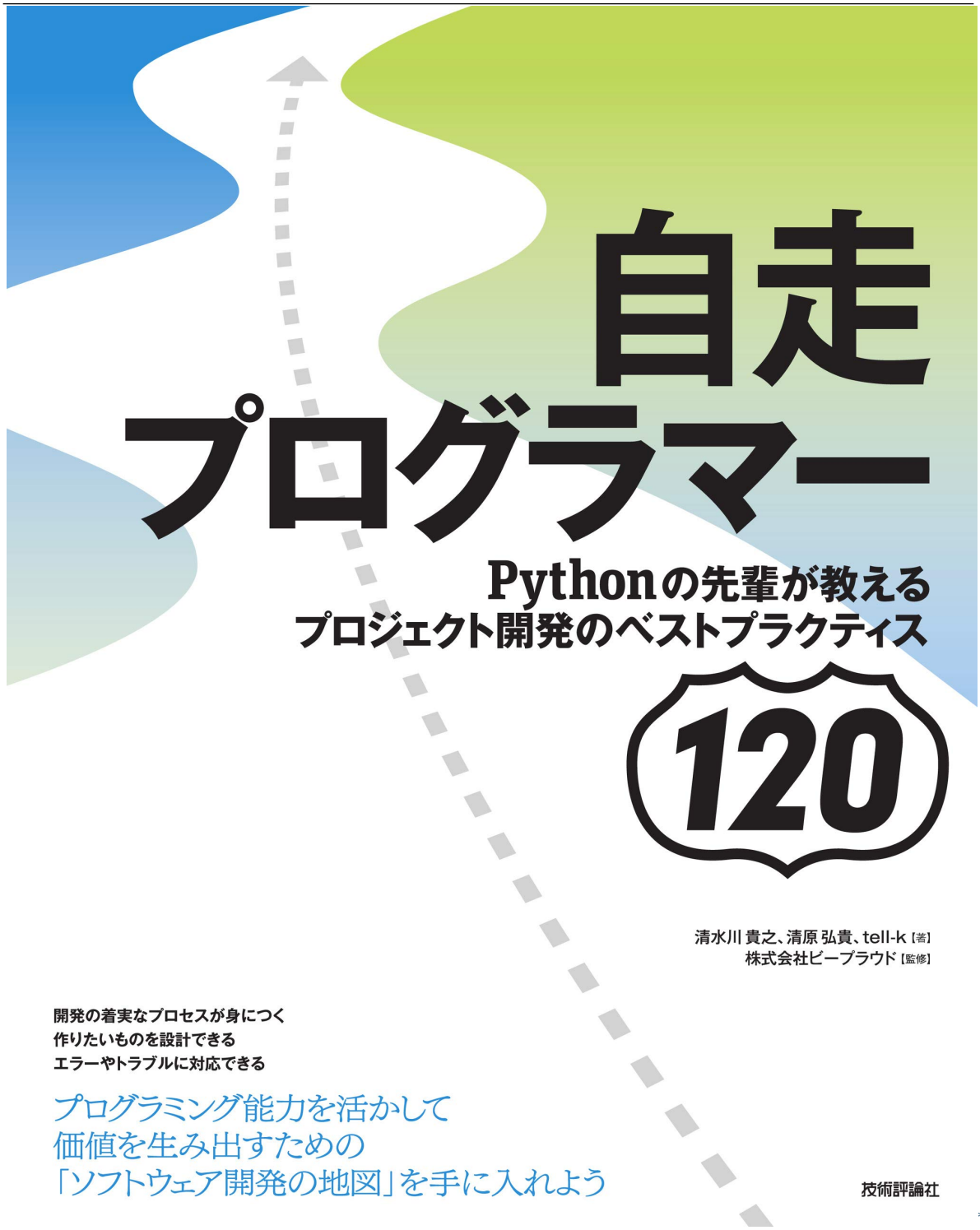
*85:*設定ファイルを環境別に分割する (ページ 310) で settings/ ディレクトリ配下の設定ファイルを base.py 、 local.py 、 staging.py に分割しました。local.py には DEBUG=True と silk のインストー

[*228] https://gihyo.jp/book/2020/978-4-297-11197-7
[*229] *76:*シンプルに実装しパフォーマンスを計測して改善しよう (ページ 283) を参照

ルを指定するなど、各設定ファイルにはその環境で一番よく使う設定を実装しています。しかし、そこからさらに継承した設定ファイルを用意するなど、設定ファイルを多段継承することには問題があります。

具体的な失敗

`local_for_me.py` のような個人用設定ファイルに `from .local import *` を書いてカスタマイズするのは簡単です。このような設定ファイルを共有リポジトリにコミットすると、 `settings/` 配下のファイルが増え、設定内容を把握するのが難しくなってしまいます。また、同じ発想で検証環境用の設定ファイルを複数用意してしまうことには問題があります。このような多段継承による差分実装を繰り返すと、当初はシンプルな方法でうまく対処したように見えても、徐々に設定の複雑化を招いてしまいます。



[*230]

[*230] https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*231] をご参照ください

ベストプラクティス

状況依存の設定値をコードから分離し、環境変数で設定しましょう。DEBUG だけであれば以下のように実装できます。

リスト 5.4　settings.py

```
import os
DEBUG = bool(os.environ.get('DEBUG', False))
```

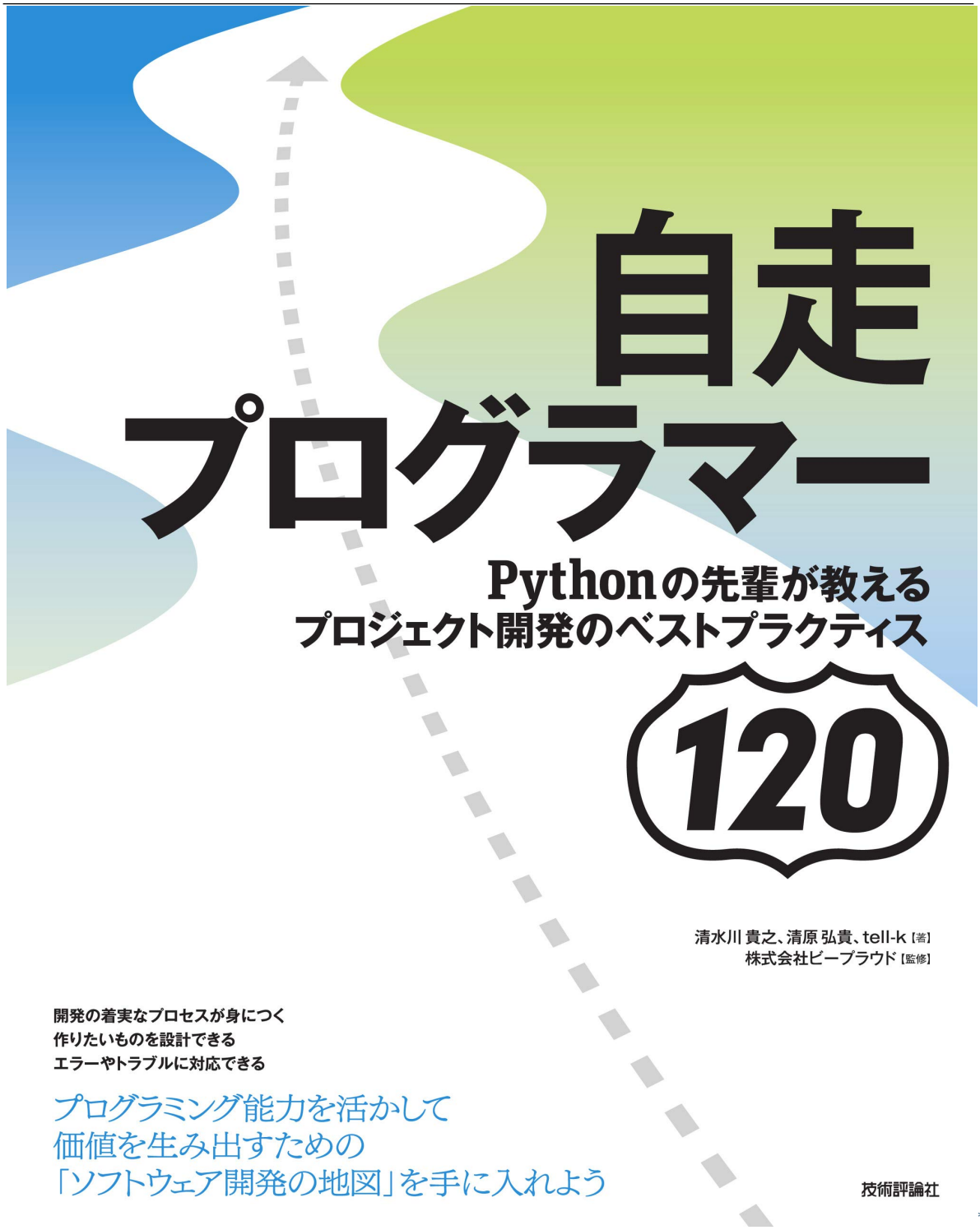
これで、環境変数 DEBUG がなければ DEBUG=False として動作します。True にしたい場合は、 DEBUG=1 python manage.py runserver のように環境変数を指定して実行します。

環境変数を Django の設定に使う場合、 django-environ[*232] パッケージを使うのが便利です。Django 以外でも同じように環境変数を設定に使いやすくするには python-decouple[*233] が利用できます。これらのツールは、環境変数を扱う便利な機能を提供しています。また、OS の環境変数から値を読み取って利用できるだけでなく、 .env ファイルに書いた環境変数設定を読み込んで利用できます。環境変数は、環境別のファイルで用意します。

[*231] https://gihyo.jp/book/2020/978-4-297-11197-7
[*232] https://django-environ.readthedocs.io/
[*233] https://pypi.org/p/python-decouple/

リスト 5.5　.env.local

```
DEBUG=True
ALLOWED_HOSTS=127.0.0.1,localhost
INTERNAL_IPS=127.0.0.1
USE_SILK=True
DATABASE_URL=sqlite:///db.sqlite3
```



*234

*234 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*235] をご参照ください

5.1.9 87:設定ファイルもバージョン管理しよう

Git などでプログラムをバージョン管理することは一般的ですが、プログラムではない設定ファイルをバージョン管理することも大切です。設定ファイルをバージョン管理することで、万が一、元に戻したいときもすぐに対処できます。

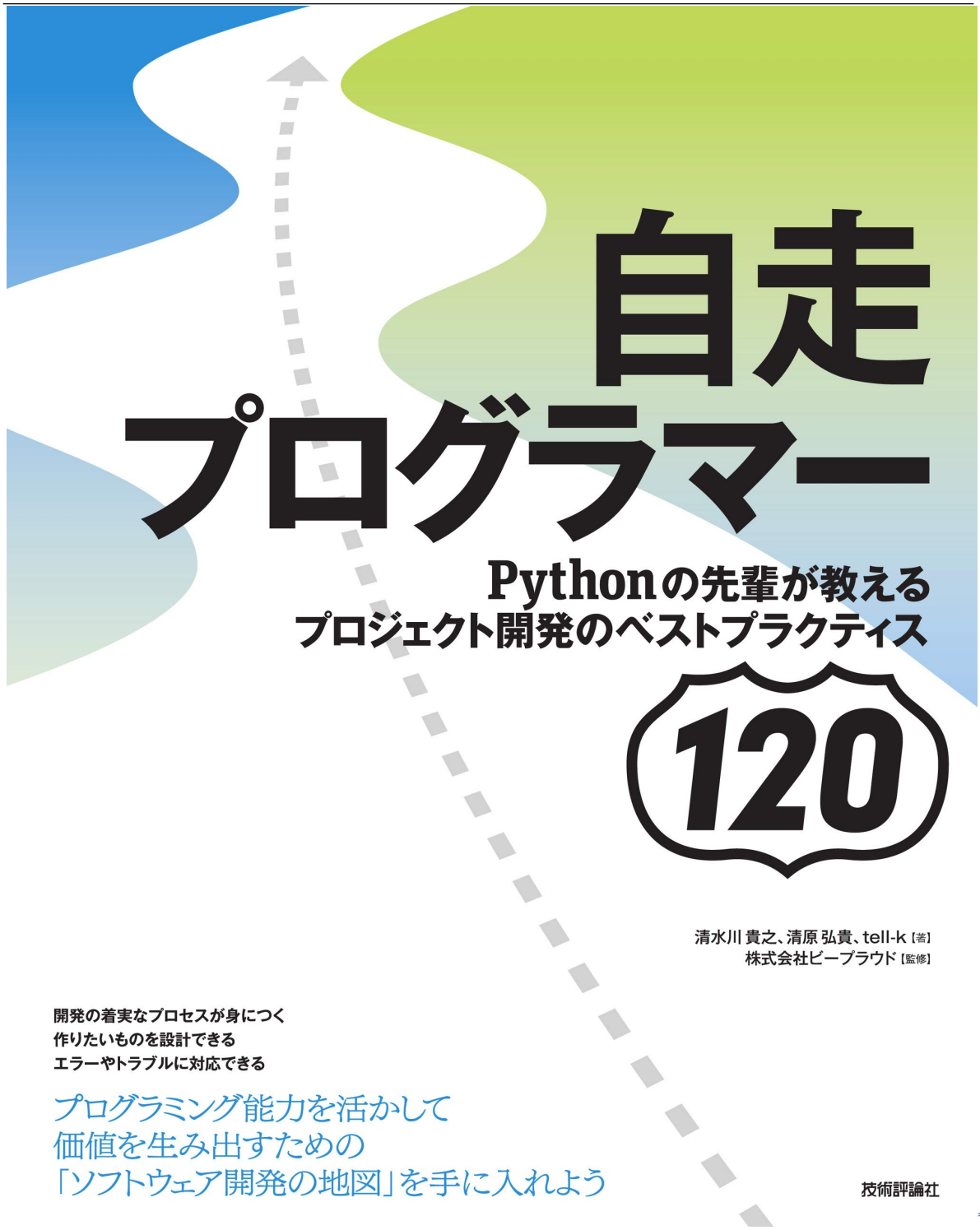
プログラミング迷子**:** サーバー設定の履歴がない

- 後輩 W：たまに直接サーバー上で設定ファイルを編集したいときがあるんですが、バージョン管理とかされてないので不安です。バージョン管理しなくていいんですか？
- 先輩 T：変更する内容とかはどうやってチームと確認してるの？
- 後輩 W：今はチケットに変更内容を書いて見てもらっています。
- 先輩 T：ふむ。変更内容は確認できるし、あとから戻すこともできると言えばできそうではあるが……いざというときにそのチケットを見つけられなさそうだね……。今はバージョン管理するほうが履歴も追いやすいし戻しやすいから、バージョン管理はしたほうがいいよ。
- 後輩 W：なるほどやっぱりそうなんですね。
- 先輩 T：Ansible 等のツールを使うと、設定ファイル群も自然とバージョン管理することになるしね。
- 後輩 W：あぁなるほど。そういう意味でも Ansible とかを使っておくといいんですね。
- 先輩 T：そうだね。サーバーを構築するための手順と設定、あとその履歴を Git 等で管理できるから、いざというときに安心だね。

ベストプラクティス

プログラムと同様に設定ファイルもバージョン管理しましょう。往々にして設定ファイルも単純なテキストデータであるため、一文字間違えただけでもソフトウェアは動かなくなります。git などでバージョン管理をしておけば、間違いがないか事前にチェックしたり、元に戻すことも容易になります。

[*235] https://gihyo.jp/book/2020/978-4-297-11197-7



*236

*236 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*237] をご参照ください

[*237] https://gihyo.jp/book/2020/978-4-297-11197-7

5.2. サーバー構成

5.2.1 88:共有ストレージを用意しよう

サーバーが複数台あったとき、アップロードしたファイルなどの共有データを、どこに置いてどう管理したら良いか悩んだことはありませんか？単一サーバーでは問題にならなかった、複数サーバー間でのファイル共有について考えてみましょう。

具体的な失敗

たとえば複数台のサーバーがあるような Web アプリケーションを作ったとき、以下の図のように単一のサーバーにだけファイルを保存していると、他のサーバーから利用できません。

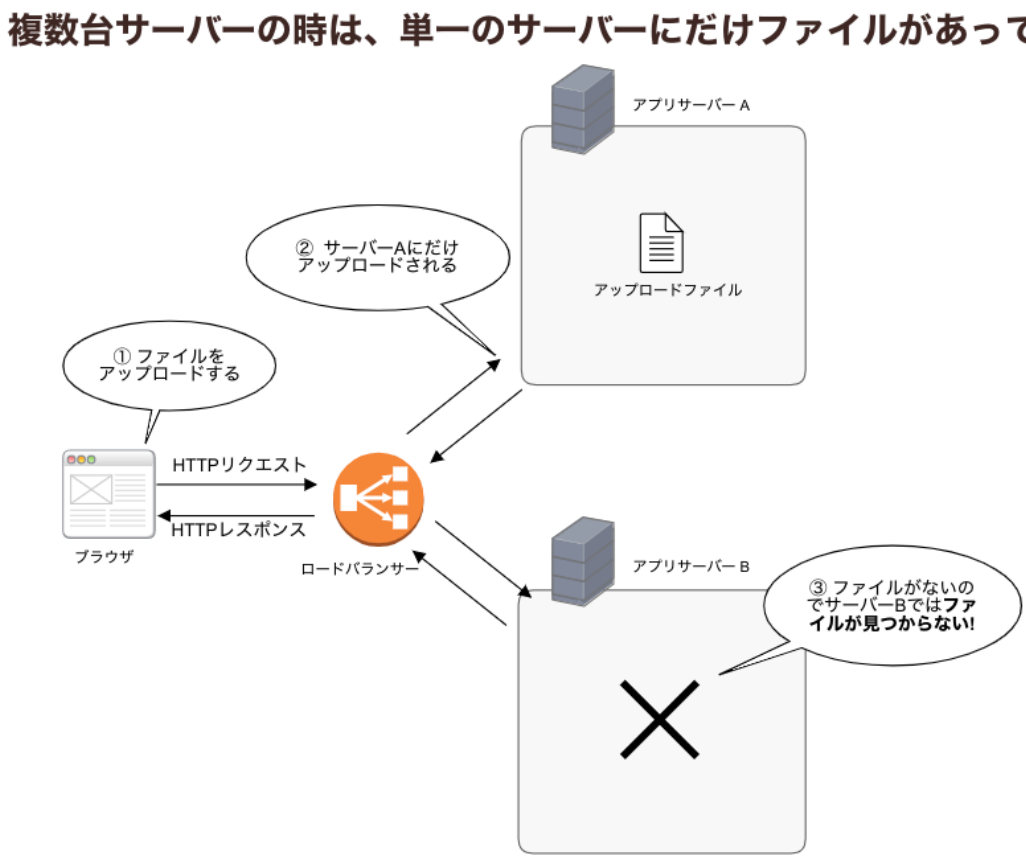


図 5.1　複数台サーバーのときは、単一のサーバーにだけファイルがあってもダメ

またサーバーが故障した場合などに、ファイルが消えてしまうリスクもあります。

ベストプラクティス

アップロードファイルを集約して管理する、共有ストレージとなるようなサーバーを用意しましょう。専用のサーバーが用意できない場合は NFS 等を利用してファイルを共有もします。

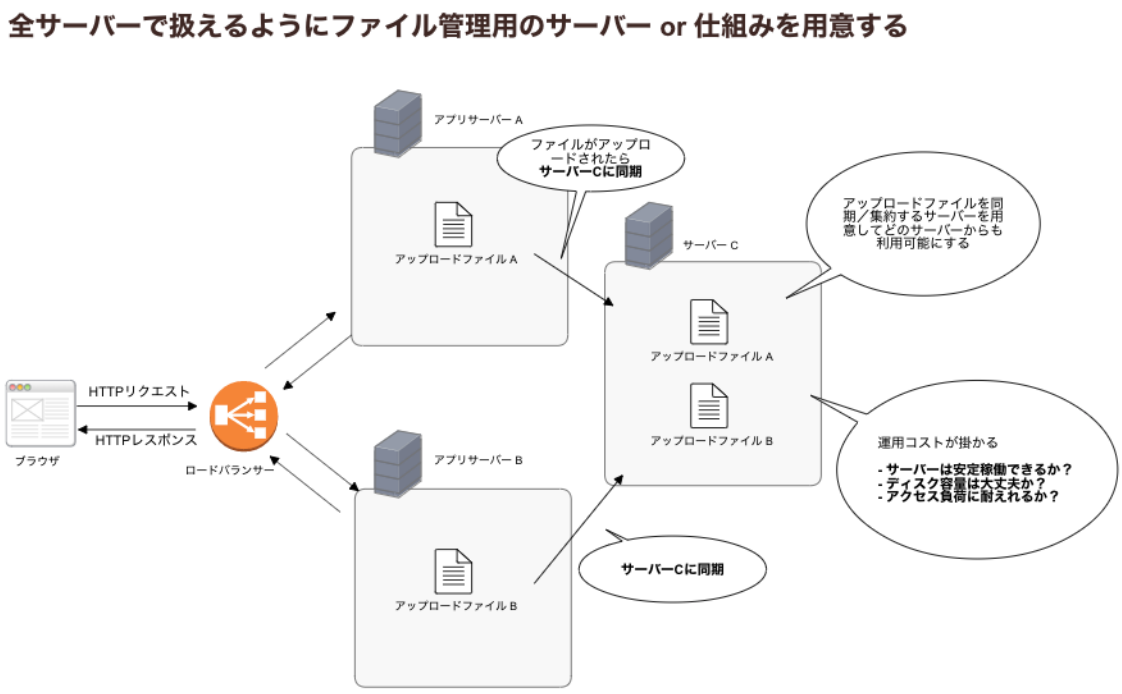
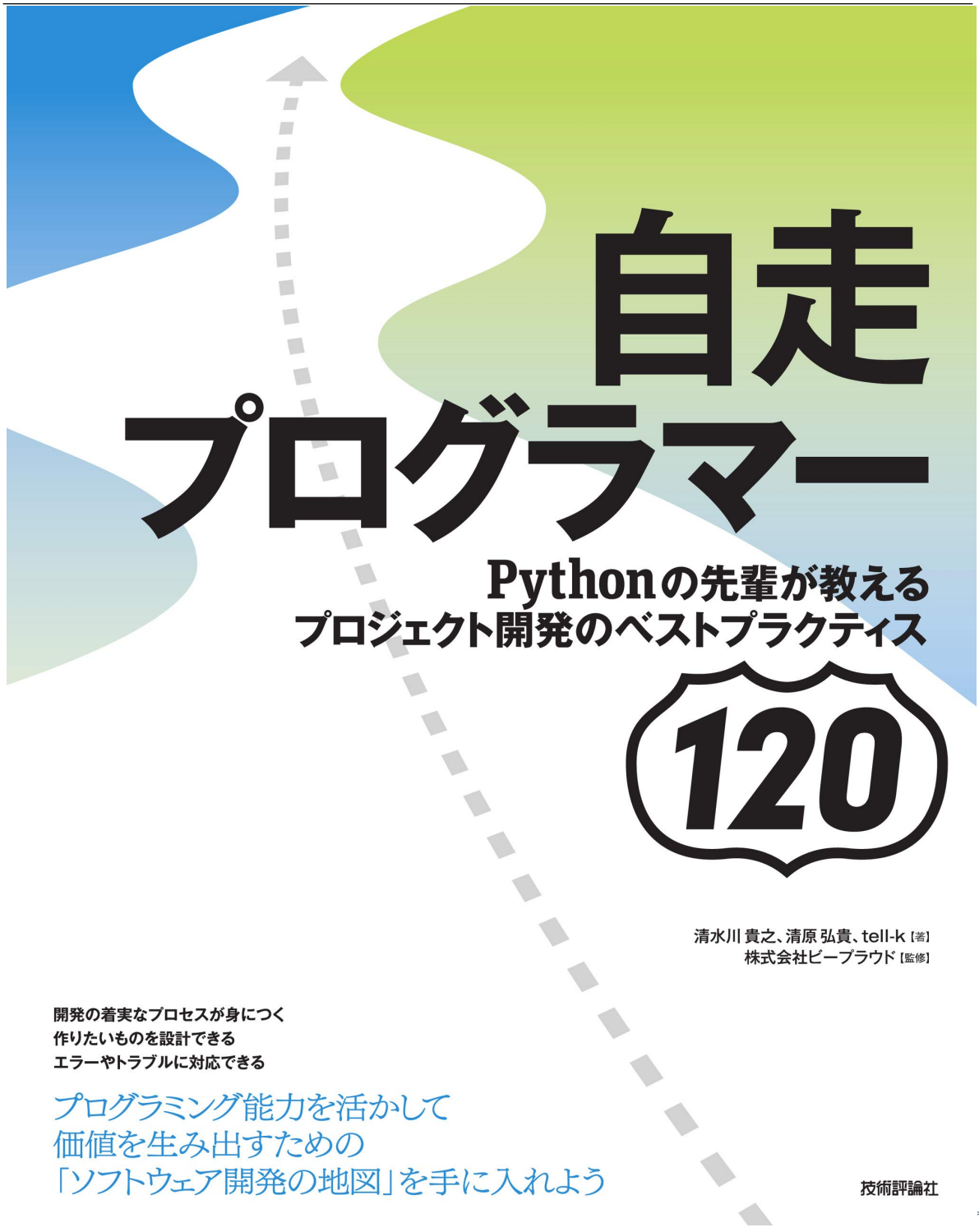


図 5.2　全サーバーで扱えるようにファイル管理用のサーバーか、仕組みを用意する

上記のような自分たちでストレージを管理する場合、サーバーの運用コストはそれなりにかかります。



*238

*238 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*239] をご参照ください

5.2.2 89:ファイルを CDN から配信する

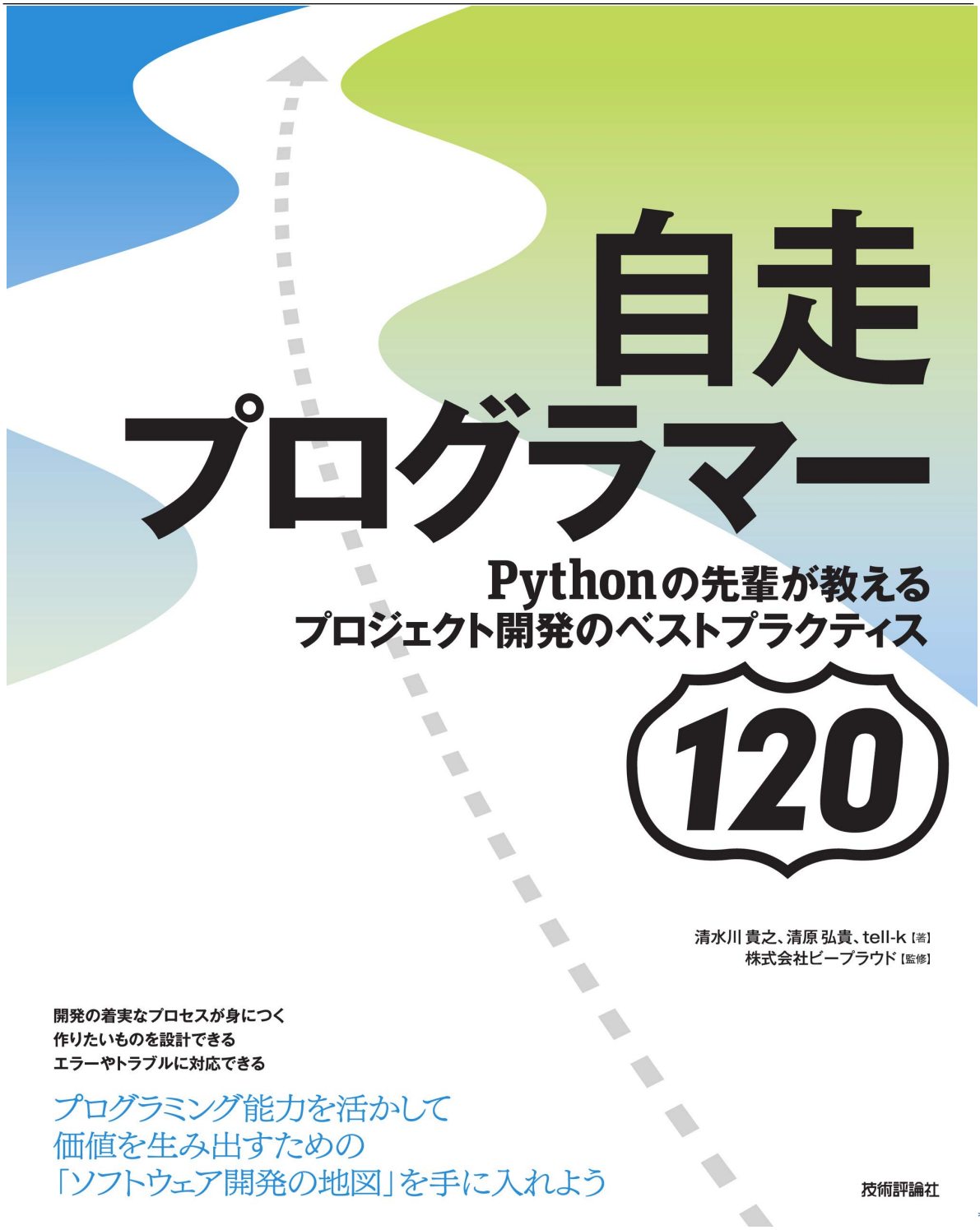
Web アプリケーションを公開したけど、サーバー側が問題がないのに、サイトが表示されるまでに時間がかかって困ったことはないですか？ここでは、静的ファイルを効率良く配信する CDN （Contents Delivery Network）について簡単に紹介します。

ベストプラクティス

Web アプリケーションを公開すると、世界中のユーザーからアクセスがきます。そのため地理的にサーバーから離れた場所にいるユーザーは、単純な画像ファイルや JavaScript ／ CSS ファイルのダウンロードにも遅延を感じるようになります。ユーザーがどこのネットワークからアクセスするかによって、体感する速度は変わってきます。

これらの問題を解決するために CDN（Contents Delivery Network）というサービスが存在します。

[*239] https://gihyo.jp/book/2020/978-4-297-11197-7



*240

*240 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*241] をご参照ください

5.2.3 90:KVS（Key Value Store）を利用しよう

Web アプリケーションを開発していて「大量のデータを RDB から何度も取得して処理が重くなった」「突然サーバーが高負荷になってしまった」というような経験はありませんか？

具体的な失敗

たとえば EC サイトなどを商品一覧ページなどで、「多数のユーザーがアクセスしにきて重くなるので表示速度を改善したい」という要望がきたとします。このとき、RDB から一度取得した商品データをプログラム上でキャッシュして、高速にレスポンスを返すようにしました。

```
from app.models import Item

CACHED_ITEMS = None

def items_view(request):
    global CACHED_ITEMS

    if CACHED_ITEMS:
        # キャッシュがあるときは RDB(Item) からデータを取得しない
        items = CACHED_ITEMS
    else:
        items = Item.objects.all()
        # すべての商品データをグローバル変数（メモリ）にキャッシュする
        CACHED_ITEMS = list(items)

    return render(request, 'items/index.html', {
        "items": items,
    })
```

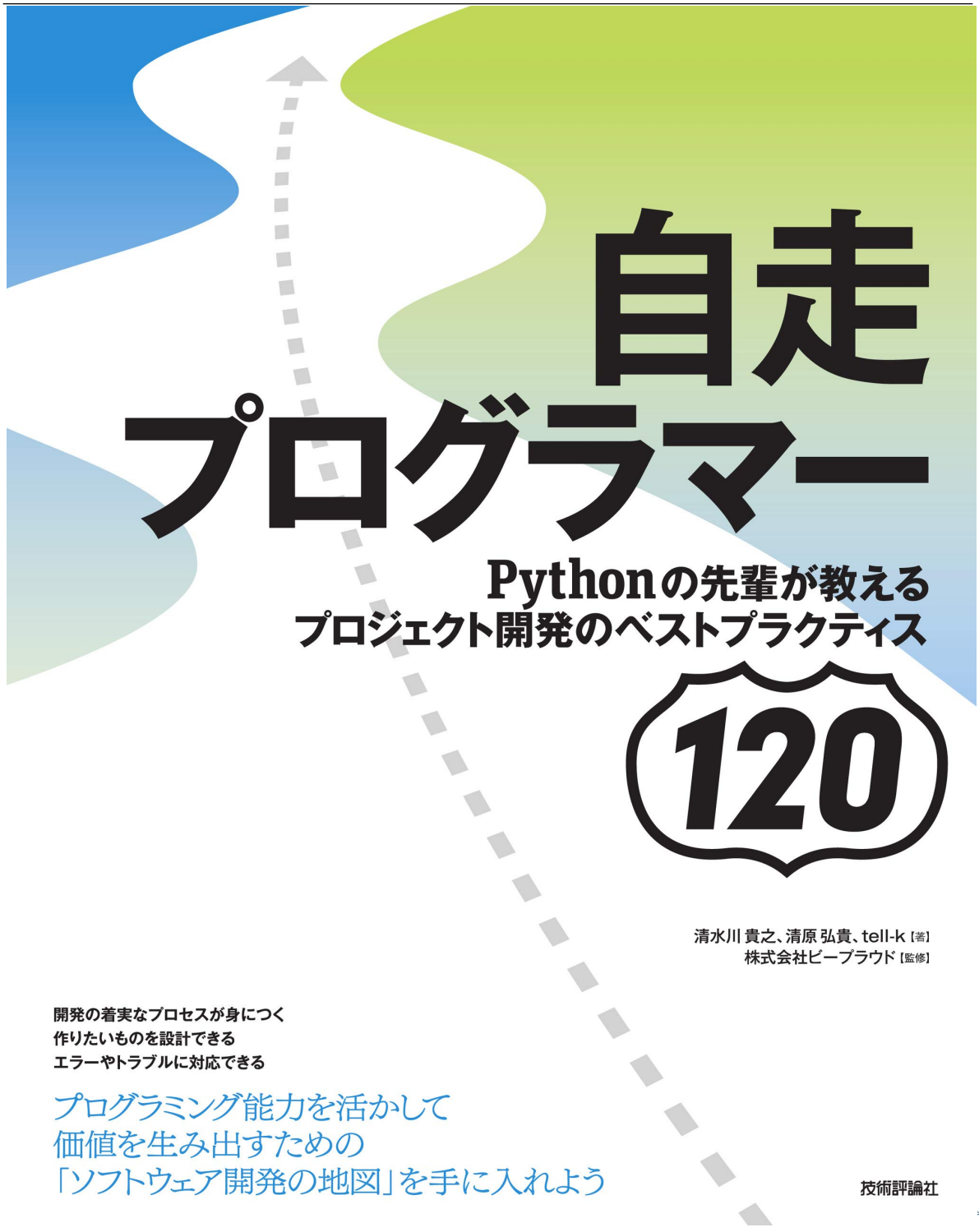
ところが、商品データをメモリにすべて載せてしまったために、逆にサーバーのメモリが枯渇してしまい、サイト全体が重くなってしまいました。

[*241] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

プログラムのメモリ上にキャッシュ用のデータを載せずに、KVS（Key Value Store）を利用しましょう。

KVS は、MySQL、PostgreSQL のような RDB とは違い、単純なキーとそれに紐づく値を管理するデータストアです。



*242

*242 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*243] をご参照ください

5.2.4 91:時間のかかる処理は非同期化しよう

Web アプリケーションを開発していて、1 つの HTTP リクエストの中で、大量のデータを扱い、処理が重くなったことはないですか？ここでは、リアルタイムでの必要のない処理を非同期化することのメリットについて紹介します。

具体的な失敗

たとえば SNS などで複数人の友達に一斉に招待メールを送るような機能を、どう実装しますか？　下記は Django で愚直に書いたコードです。

```
def invite_users_view(request):

    form = InviteForm(request.POST)
    if not form.is_valid():
        return render('error.html')

    emails = form.cleaned_data['emails']
    for email in emails:
        api.send_invite_mail(email) # 1 件 1 件その場で配信してすべて終わるまで処理がブロックされる

    # メールがすべて配信し終わるまで send_end.html 画面は表示されない
    return render('send_end.html')
```

このコードだと、1,000 人同時に招待したら 1,000 人にメール配信が完了するまでユーザーの画面は固まったままです。システム的にもリソースが占有されて他のリクエストを捌けなくなる可能性があります。

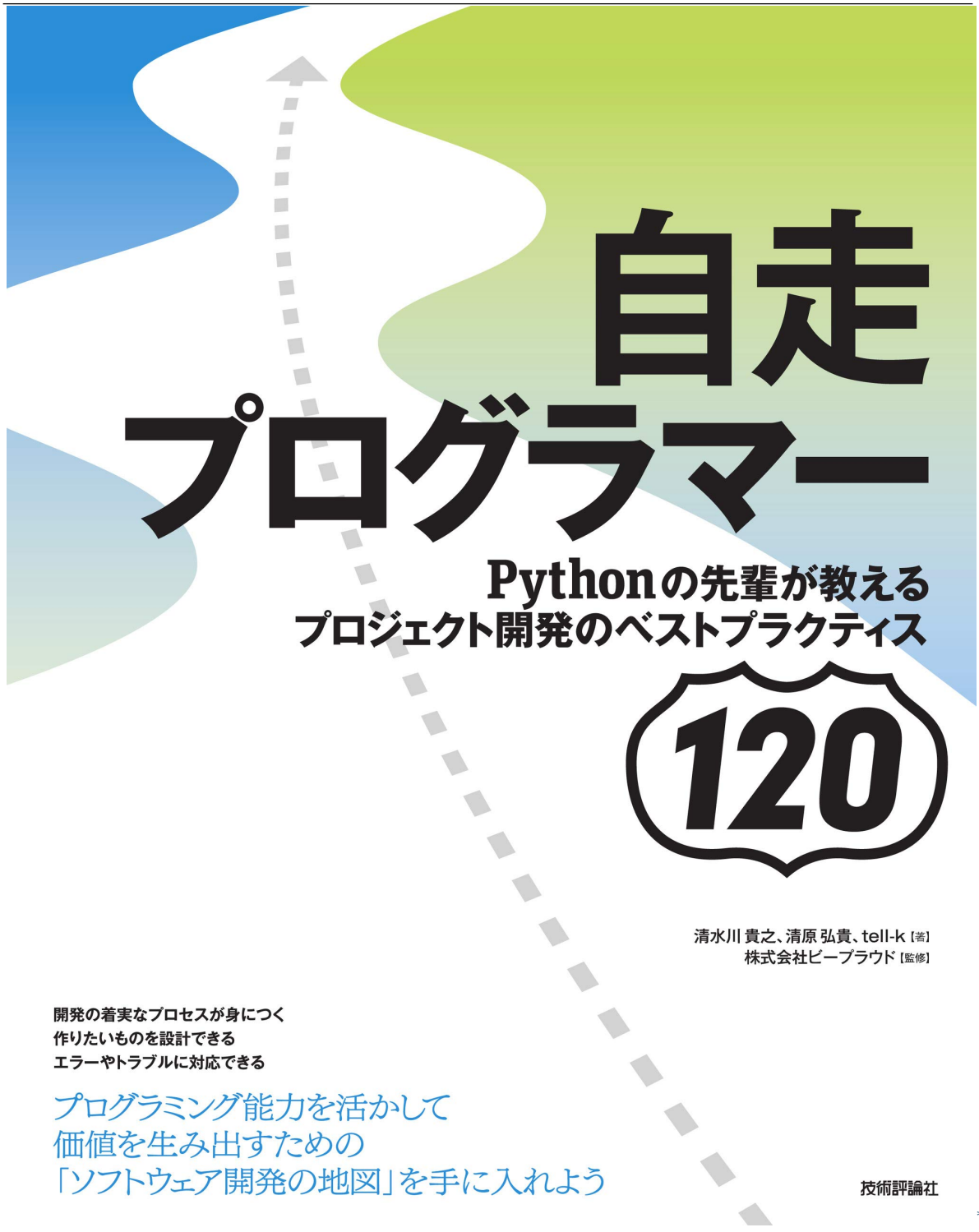
[*243] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

リアルタイムでの処理が必要ない部分で時間がかかるような場合、 非同期化 を検討しましょう。たとえば、メールの送信や、外部システムへの通信は、非同期で処理したほうが良い場合があります。

非同期化と一口に言っても、実現方法はさまざまです。たとえば Python では以下のような方法があります。

- ThreadPoolExecutor 等を用いて別スレッドで処理する
- asyncio を利用する
- Celery などのジョブキューシステムを利用する



*244

*244 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*245] をご参照ください

関連

- *77:* **トランザクション内はなるべく短い時間で処理する** (ページ 285)
- *92:* **タスク非同期処理** (ページ 333)

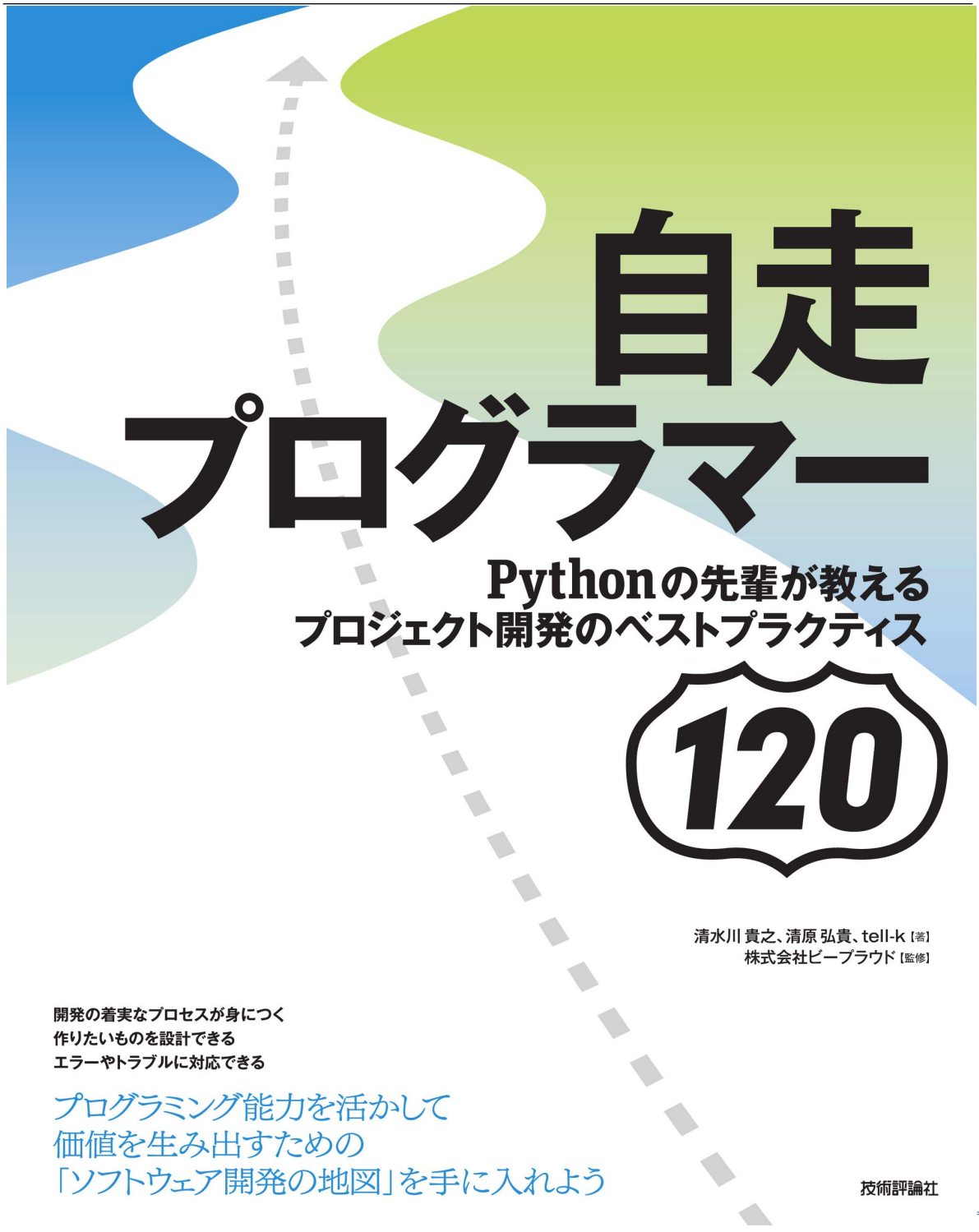
5.2.5 92:タスク非同期処理

プログラミング迷子**:** ワーカープロセスからスレッド起動

- 後輩 W：ちょっとわからない不具合があって、相談に乗ってください。タスクの非同期処理を実装したんですが、たまに処理が行われないことがあるんです。
- 先輩 T：非同期処理、どうしてやりたいんだっけ。
- 後輩 W：Web アプリケーションでボタンを押したときに、時間がかかる処理をやりつつ、ブラウザにはすぐレスポンスを返すためです。
- 先輩 T：んー、なるほど。その非同期処理はどうやって実装したの？
- 後輩 W：スレッドで動かしてます。
- 先輩 T：あー、それが原因だろうね。タスク処理用のスレッドを Gunicorn プロセスから起動したために、Gunicorn のワーカープロセスが自動再起動したときにおかしくなってるんだと思うよ。

Gunicorn のワーカープロセスなど、自動的に再起動されるプロセス上でスレッド起動や子プロセス起動をしてはいけません。Gunicorn のような Web アプリケーションのプロセスは、複数のレスポンスを扱うための機能を提供するためにマルチプロセス、マルチスレッドが使われています。このため各プロセスからさらにスレッドや子プロセスを起動した場合、そういった制御機構と競合してしまい、何が起こるかわかりません。

[*245] https://gihyo.jp/book/2020/978-4-297-11197-7



*246

*246 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*247] をご参照ください

ベストプラクティス

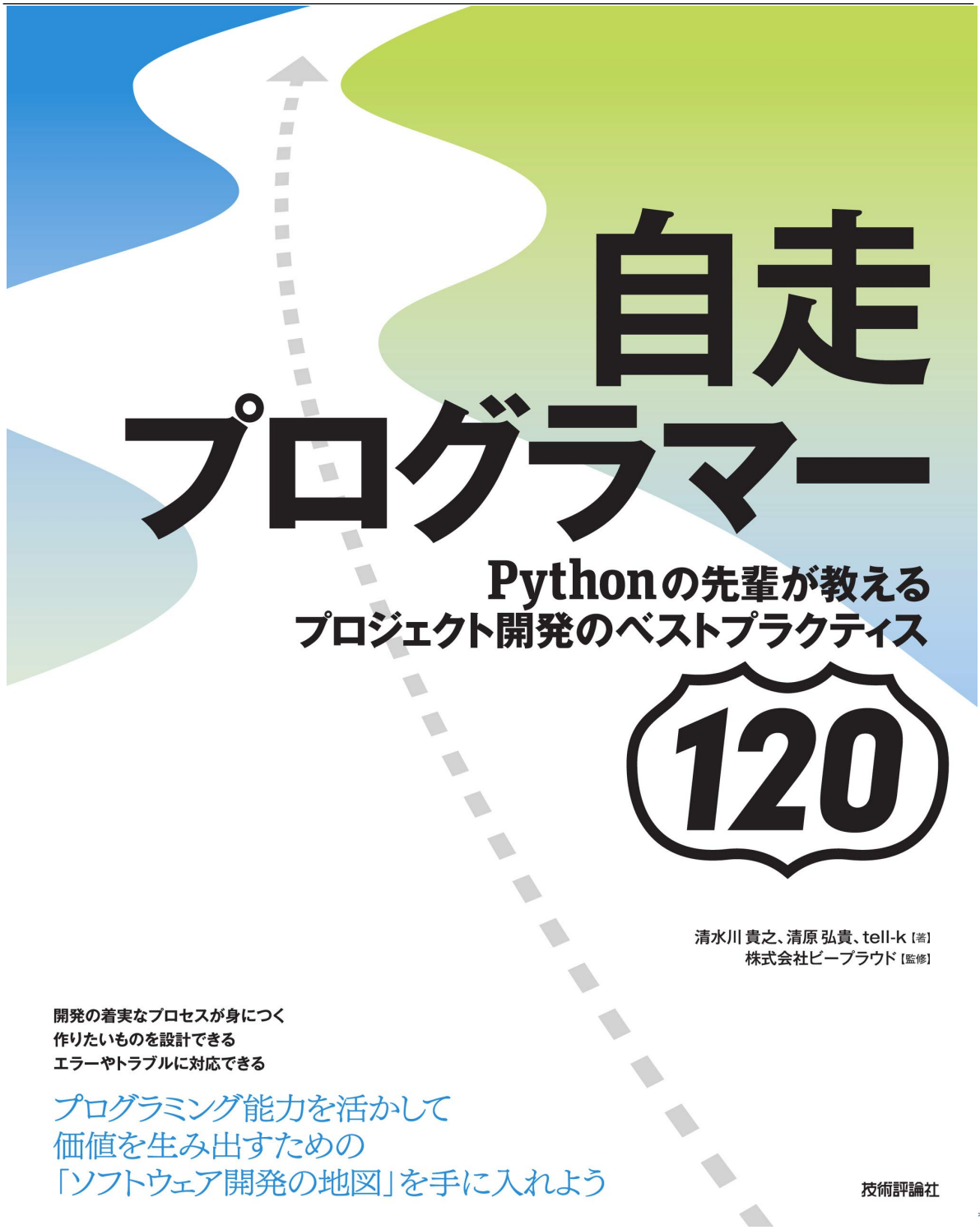
非同期タスク処理が必要な場合は、専用プロセスで処理を行うように設計しましょう。

非同期タスク処理は自作しようとせず、定番フレームワークの利用を検討しましょう。定番フレームワークには、以下のようなものがあります。

表 5.1　非同期タスク処理フレームワークの比較

	Celery[*248]	Django Background Tasks[*249]	APScheduler[*250]
バージョン（リリース日）	4.4.0（2019/12/16）	1.2.5（2019/12/23）	3.6.3（2019/11/5）
インフラミドルウェア追加	Redis	なし	なし
ライブラリの使いやすさ			

[*247] https://gihyo.jp/book/2020/978-4-297-11197-7
[*248] http://www.celeryproject.org/
[*249] https://django-background-tasks.readthedocs.io/
[*250] https://pypi.org/project/APScheduler/



*251

*251 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*252] をご参照ください

関連

- *95:Celery* **のタスクにはプリミティブなデータを渡そう** (ページ 345)

[*252] https://gihyo.jp/book/2020/978-4-297-11197-7

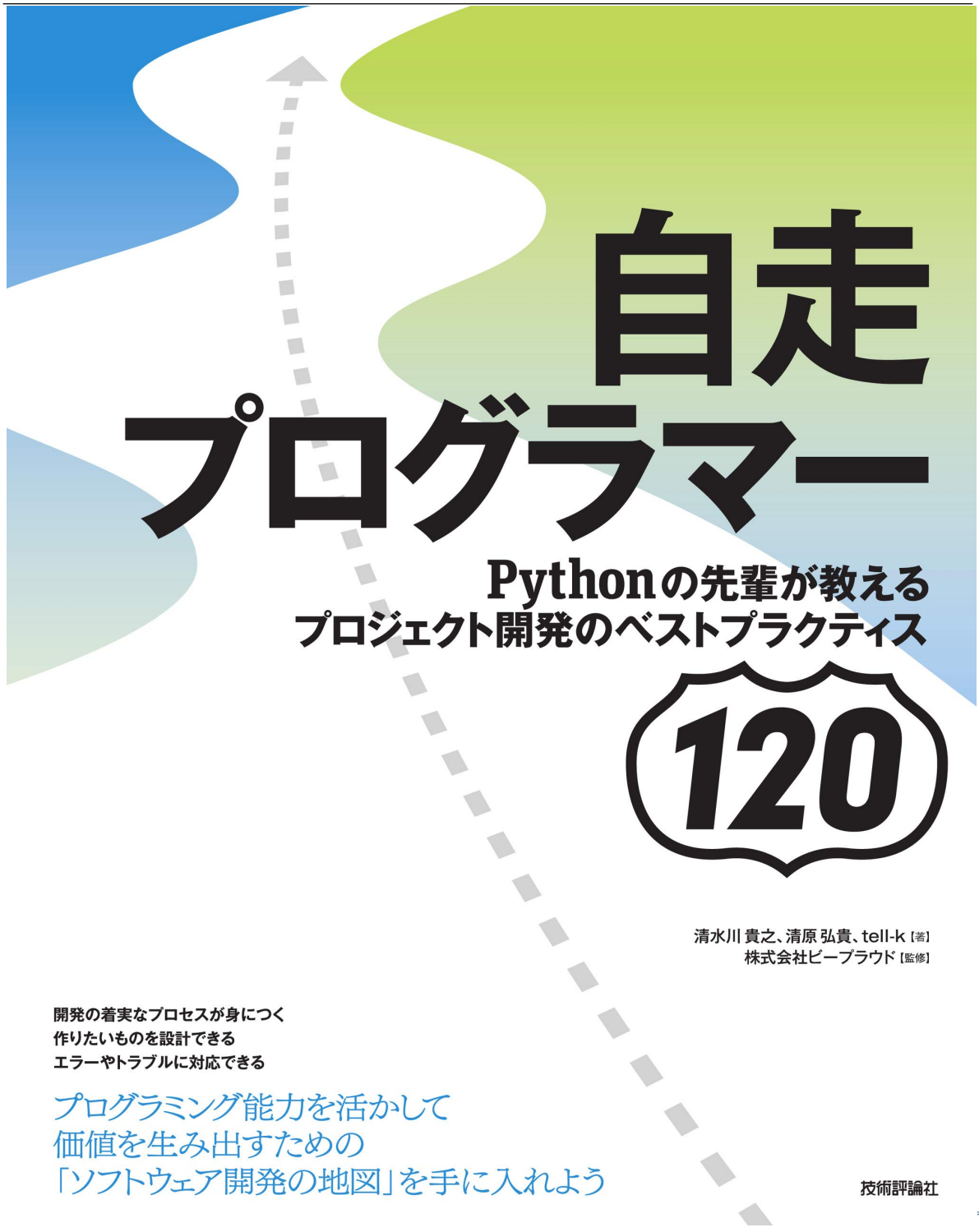
5.3. プロセス設計

5.3.1 93:サービスマネージャーでプロセスを管理する

プログラミング迷子**: Django** サーバーを動かし続ける方法は？

- 後輩 W：Django を実行しているとき、 `Ctrl` + `C` を入力したり、ターミナルからログアウトするとプロセスが終了してしまうんです。
- 先輩 T：そうだろうね。何か困ってるの？
- 後輩 W：ログアウト後も実行し続ける方法を調べてたら `nohup python manage.py runserver < /dev/null &` で起動するっていう方法を見つけたんですけど、これでもときどきプロセスが止まってしまうみたいで、お客さんの動作確認がなかなか進まなくて。どうしたら止まらないようにできるんでしょう？
- 先輩 T：ちょっと待って！　検証環境を runserver で動かしてるの？　Web アプリケーションサーバーとサービスマネージャーは使ってない？
- 後輩 W：Web アプリケーションサーバー、って Django のことじゃないんですか？

Web アプリケーションサーバー は Web アプリケーションを実行するサーバープロセスです。Django は Web アプリケーションフレームワークですが、サーバーではありません。Django が内蔵している **manage.py runserver** コマンドも Web アプリケーションサーバー機能を提供しますが、これは簡易的な機能で本番には不向きです。ソースコードを変更した場合に自動的に再起動したり、画像や CSS などの静的ファイルを配信するといった開発に便利な仕組みを持っていますが、本番環境で必要となるいくつかの機能は持っていません。



*253

*253 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*254] をご参照ください

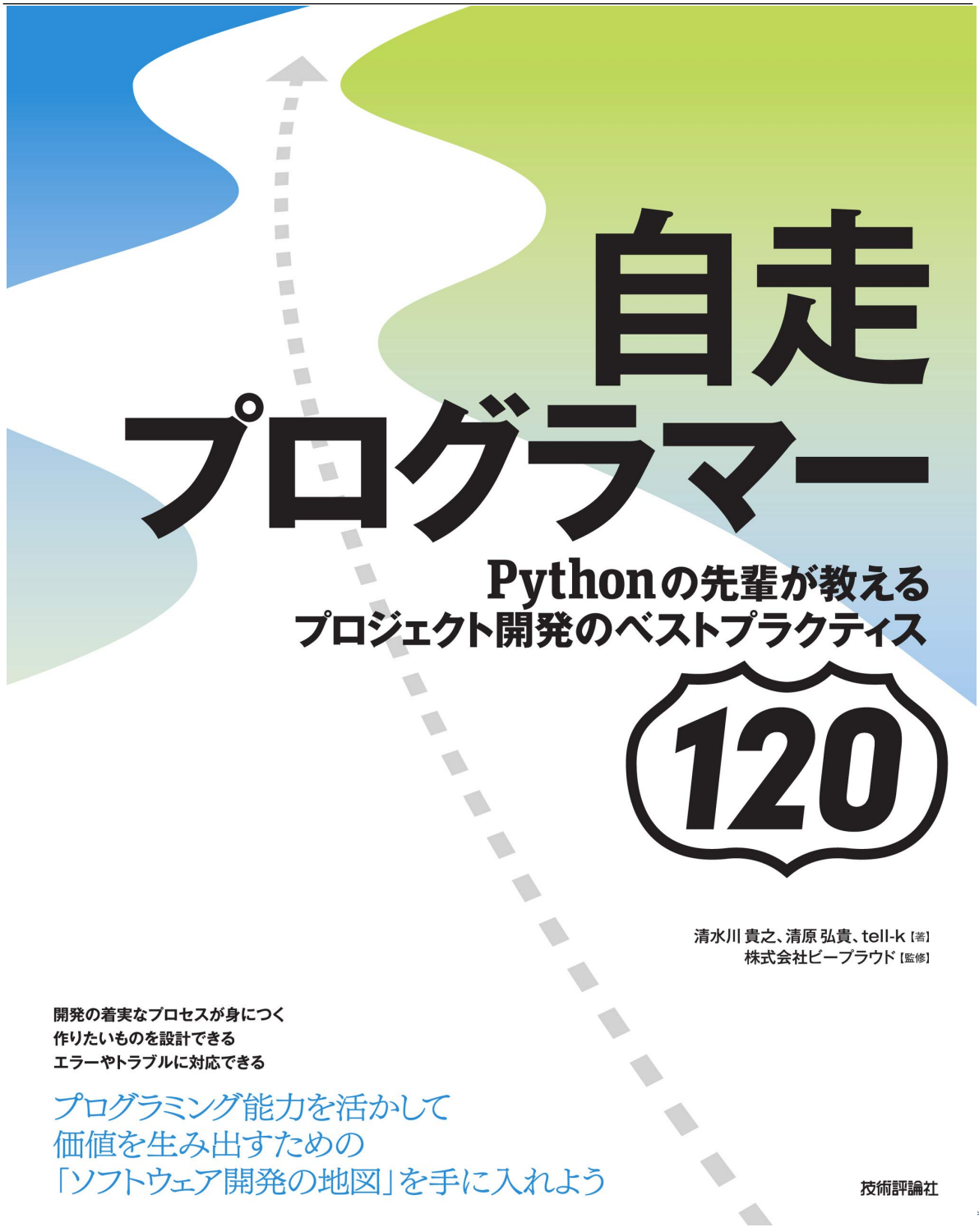
ベストプラクティス

本番環境やそれに近い環境で Django を常駐実行する場合、サービスマネージャーと Web アプリケーションサーバーを使用しましょう。

サービスマネージャーは、常駐するデーモンプロセスの起動や終了を管理し、異常終了時の自動再起動などを行います。最近の Linux では Systemd が標準的に利用されていますが、少し前の Linux では Upstart や SysV init などが使われていました。Systemd はサービス管理のための多くの機能を提供しますが、そのうちの 1 つにログ管理があります。ログ出力は journalctl で確認できます。ログファイル管理について詳しくは *74:*ログファイルを管理する (ページ 277) を参照してください。

Web アプリケーションサーバーとしては、 **Gunicorn**[*255] や **uWSGI**[*256] などが一般的に利用されます。本番利用を想定している Web アプリケーションサーバーは、複数プロセス起動による並列処理機能と死活監視を提供します。さらに、こういった専用のミドルウェアは動作が非常に速く、利用環境に合わせて設定できるさまざまなチューニングオプションを提供しています。

[*254] https://gihyo.jp/book/2020/978-4-297-11197-7
[*255] https://pypi.org/project/gunicorn/
[*256] https://pypi.org/project/uWSGI/



*257

*257 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*258] をご参照ください

関連

- *68:*ログがどこに出ているか確認しよう (ページ 258)
- *94:*デーモンは自動で起動させよう (ページ 342)

5.3.2 94:デーモンは自動で起動させよう

サービスマネージャーでアプリをデーモン化したものの、サーバーを再起動したらアプリが起動せずに困ったことはないですか？デーモン化したものが自動で起動するような設定を紹介します。

具体的な失敗

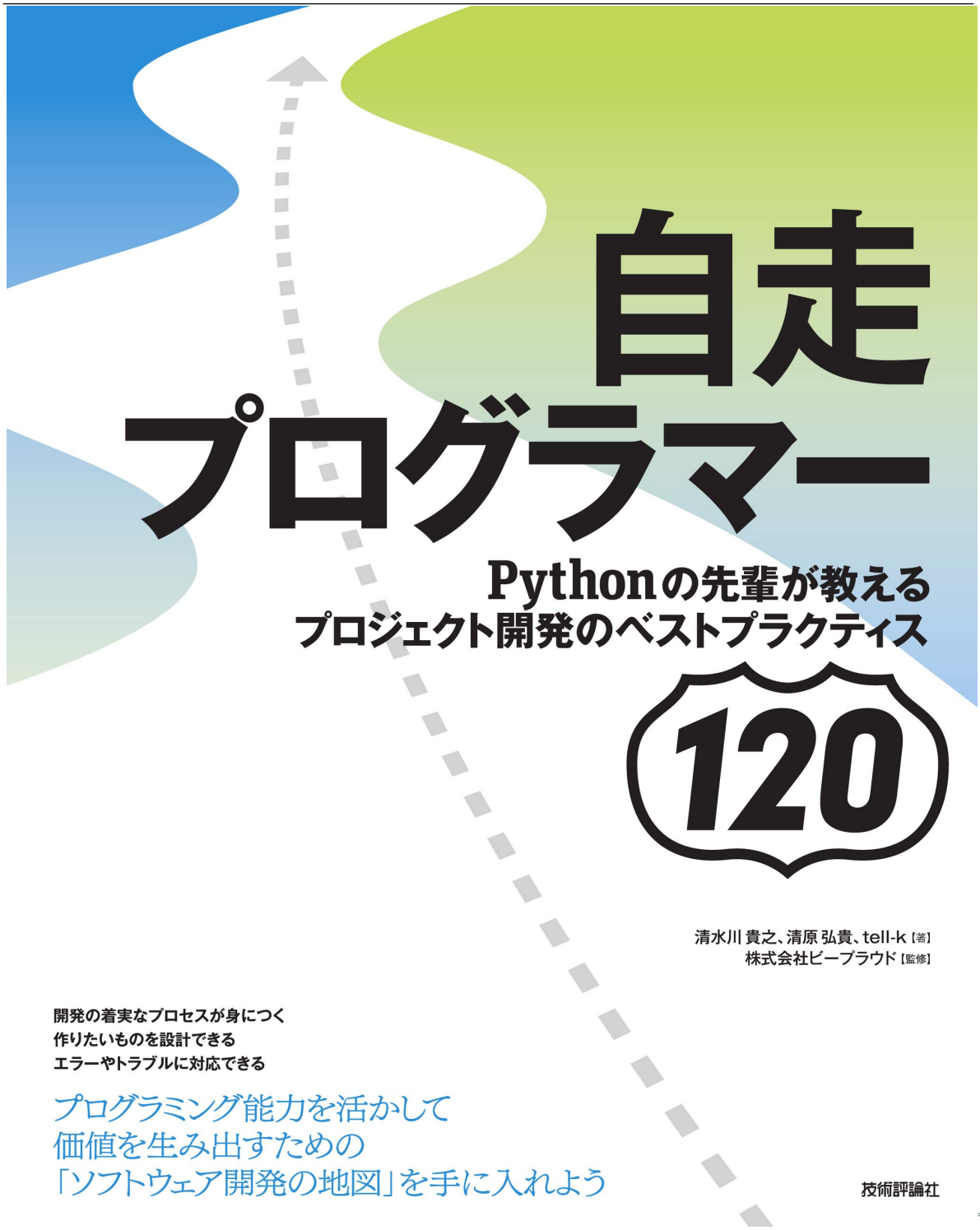
Web アプリケーションを systemd でデーモン化したので、アプリケーションが高負荷状態でプロセスが Kill されても再起動されるという状態は担保できていました。ところがサーバーにセキュリティー更新を当てるために、サーバーを再起動してしばらくしたところ Web アプリケーションが動いていないという事態が発生しました。

原因はごく単純で、サーバーの再起動後の自動起動の設定をしていなかったのです。

[*258] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

サーバー上で `systemd` を使ってデーモン化したら systemctl で自動起動の設定をしておきましょう。サーバーは永久に動き続けるわけではないので、不意の事態に対応できるように備えておくべきです。



*259

*259 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*260] をご参照ください

5.3.3 95:Celery のタスクにはプリミティブなデータを渡そう

Celery のようなジョブキューシステムを利用するとき、ジョブに渡すデータが大きいと、思わぬ不具合に遭遇します。ここでは、なるべく不具合になりにくいデータの渡し方についてご紹介します。

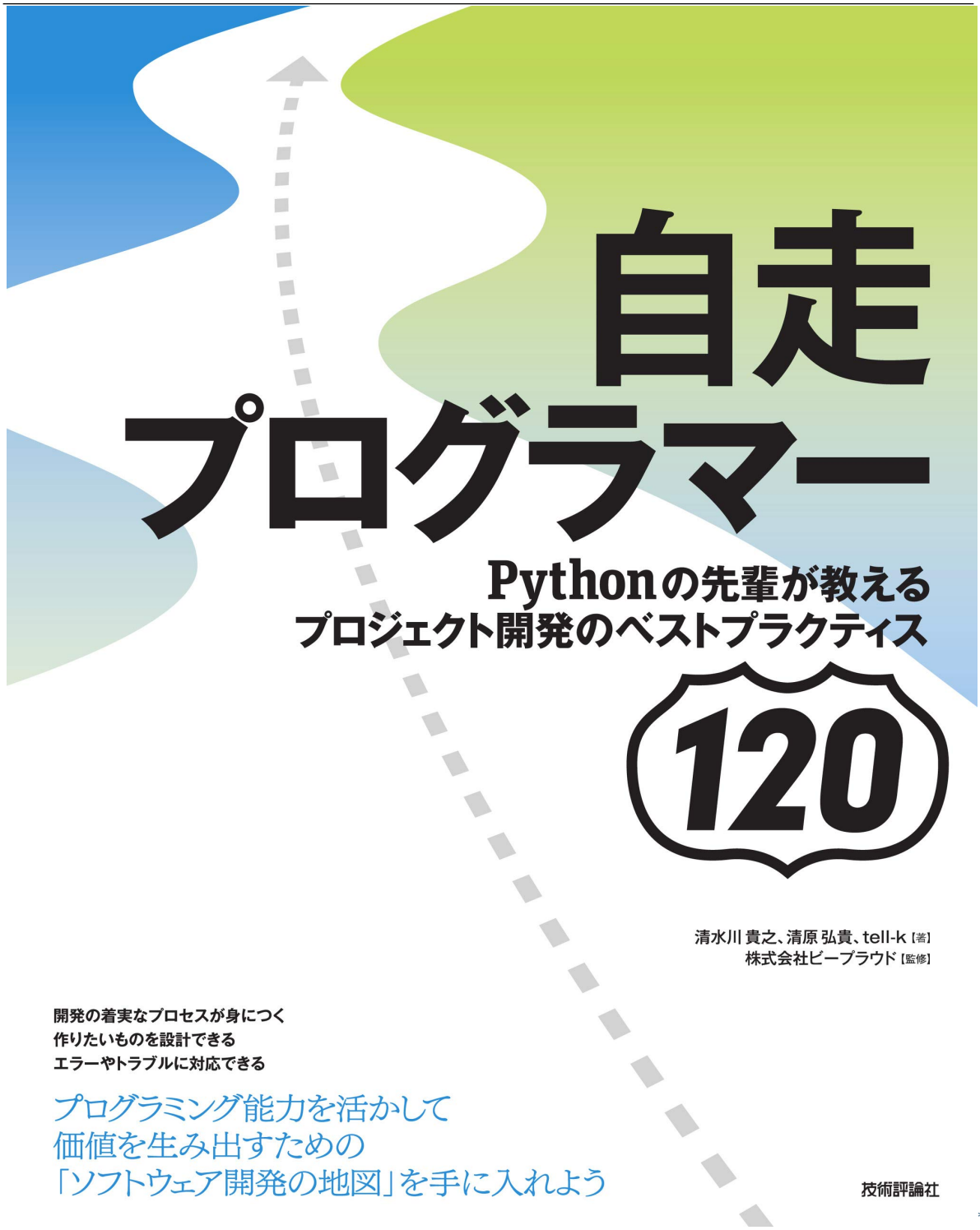
具体的な失敗例

下記のコードは Django の ProductItem というモデルのデータをオブジェクトそのままに Celery のタスクに渡しているコードです。

```
# Celery のタスク
@shared_task
def update_items_task(items, new_attr):
    for item in items:
       if item.attr != new_attr:
          item.attr = new_attr
          item.save()


# タスクの呼び出し元
def some_process(product_item_ids, new_attr):
    target_items = ProductItem.objects.filter(id__in=product_item_ids)
    update_items_task.delay(target_items, new_attr)
```

コードとしてはシンプルですが、Django から Celery への通信コストという点では、複雑なデータ構造を持つ Python のオブジェクトはあまり良くありません。

[*260] https://gihyo.jp/book/2020/978-4-297-11197-7



*261

*261 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*262] をご参照ください

ベストプラクティス

Celery のような専用のデーモンを立ち上げて処理するようなシステムにデータを送るときは、なるべくプリミティブ（原始的）なデータにしましょう。たとえば int や str などのシンプルな値です。受け取った側ではでプリミティブなデータから、本当に必要なデータを取り出して利用しましょう。

```
@shared_task
def update_items_task(item_ids, new_attr):
    for item in ProductItem.objects.filter(id__in=item_ids): # <- 受け取った ID から必要なデータを取得する
        if item.attr != new_attr:
            item.attr = new_attr
            item.save()

def some_process(product_item_ids, new_attr):
    target_items = ProductItem.objects.filter(id__in=product_item_ids)
    update_items_task.delay([t.id for t in target_items], new_attr) # <- id(int) のリストだけを渡す
```

ここでは id のリストだけを Celery に渡し、受け取ったタスク側で id を元に最新のモデル情報を取得しています。こうすることで、送信するデータ量を抑えつつ、常に最新の状態でタスクを処理できます。

[*262] https://gihyo.jp/book/2020/978-4-297-11197-7

5.4. ライブラリ

5.4.1 96:要件から適切なライブラリを選ぼう

OSS などのライブラリを選定するときに何を基準に採用すれば良いか迷ったことはありませんか？ここではライブラリをどのような観点で選び、導入していくと良いのかを説明します。

ベストプラクティス

OSS のライブラリは、ソースが公開されていて無料で利用できるものも多いという利点と引き換えに、開発が突然停止したり、プログラミング言語のバージョンアップに対応してくれなくて利用ができなくなったりと、採用するリスクも存在します。

そういったリスクを完全に回避することはできませんが、導入にあたって気をつけるべきポイントを紹介します。

要件を満たすライブラリを探そう

先行事例を確認しよう

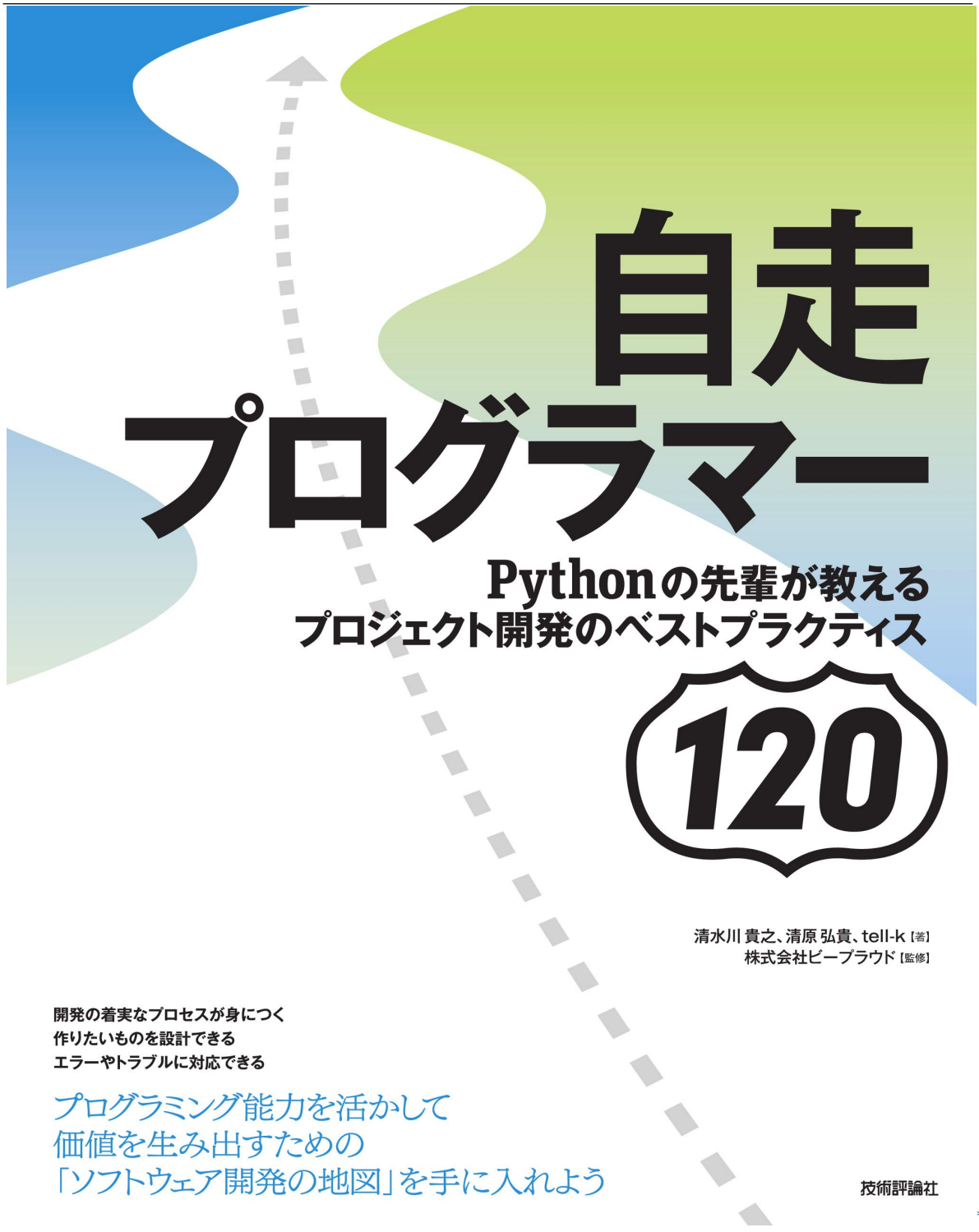
枯れているライブラリを利用しよう

ライセンスを確認しよう

オフィシャルかどうか確認しよう

こんな **OSS** ライブラリはちょっと注意しよう

小さく試そう



*263

*263 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*264] をご参照ください

5.4.2 97:バージョンをいつ上げるのか

Python のライブラリをインストールして利用する場合、バージョンを固定するのが一般的です。バージョンを固定するのは、意図しないタイミングで新しいバージョンのライブラリがインストールされ、API の変更などでプログラムが動作しなくなるトラブルを避けるためです。しかし、バージョンを固定したままでは、今度はセキュリティー上の問題を放置してしまうことになります。

では、いつバージョンを上げれば良いのでしょうか？いつまでも古いバージョンを使い続けると、バージョンアップによる機能や API の変化が大きくなり、バージョンアップによる修正とテストのコストが増大していきます。コストが増大した結果、バージョンアップを諦めざるを得ないプロジェクトもありそうです。

しかし、セキュリティー上の重大な問題が発生した場合、新しいバージョンには修正版が提供されても、古いバージョンは修正されないことがほとんどです。たとえば、Django1.8 は 2018 年 3 月末でセキュリティー更新が終了しました。Django1.8 を使い続けているプロジェクトでは、最低でも 1.11 にバージョンを上げる必要があります。その 1.11 も 2020 年 4 月で更新が終了するため、セキュリティー更新のあるバージョンを使っていくためには次の LTS（Long Term Support）である 2.2 に上げる必要があります。こういった大ジャンプを避けるためにも、定期的にバージョンアップしましょう。

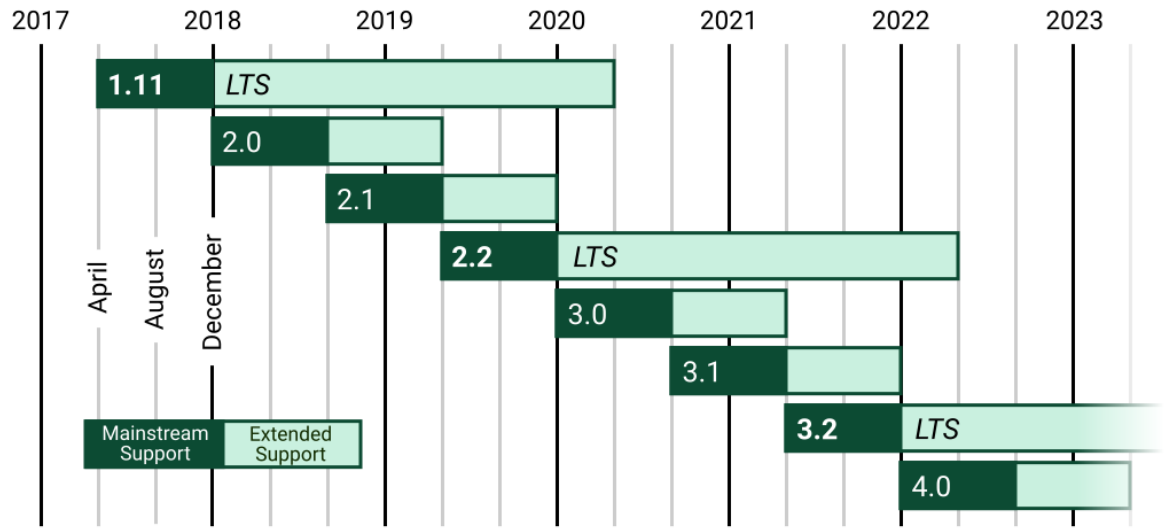


図 5.3　Django リリースロードマップ（公式サイトより）

[*264] https://gihyo.jp/book/2020/978-4-297-11197-7

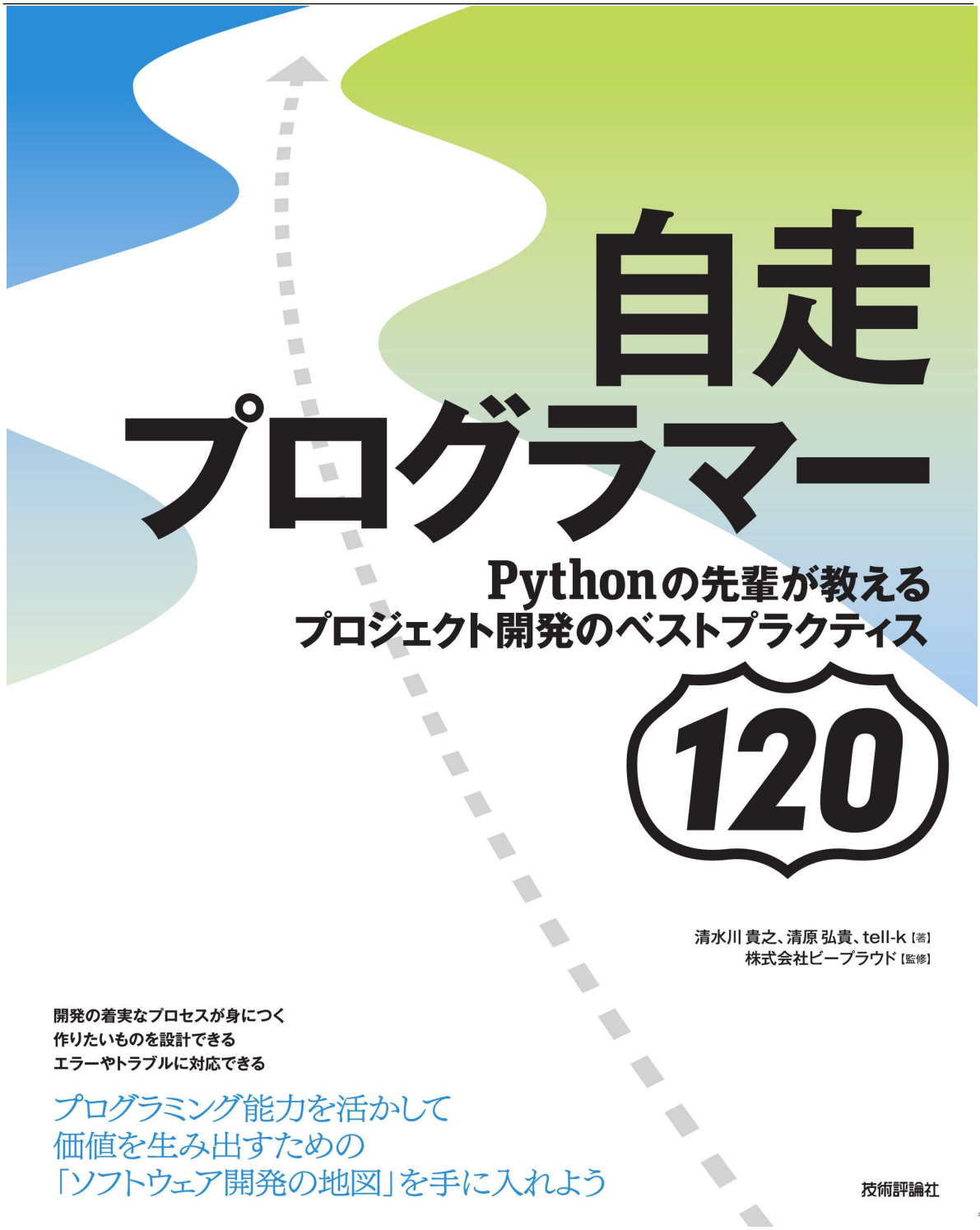
ベストプラクティス

利用しているライブラリのバージョンを上げるタイミングについて、いくつかの観点に分けて説明します。

フレームワーク

Django や Celery といった機能や影響が大きいフレームワークの場合、パッチバージョン（2.2.8→2.2.9）[*265] の適用はこまめに行いましょう。

[*265] https://semver.org/lang/ja/



*266

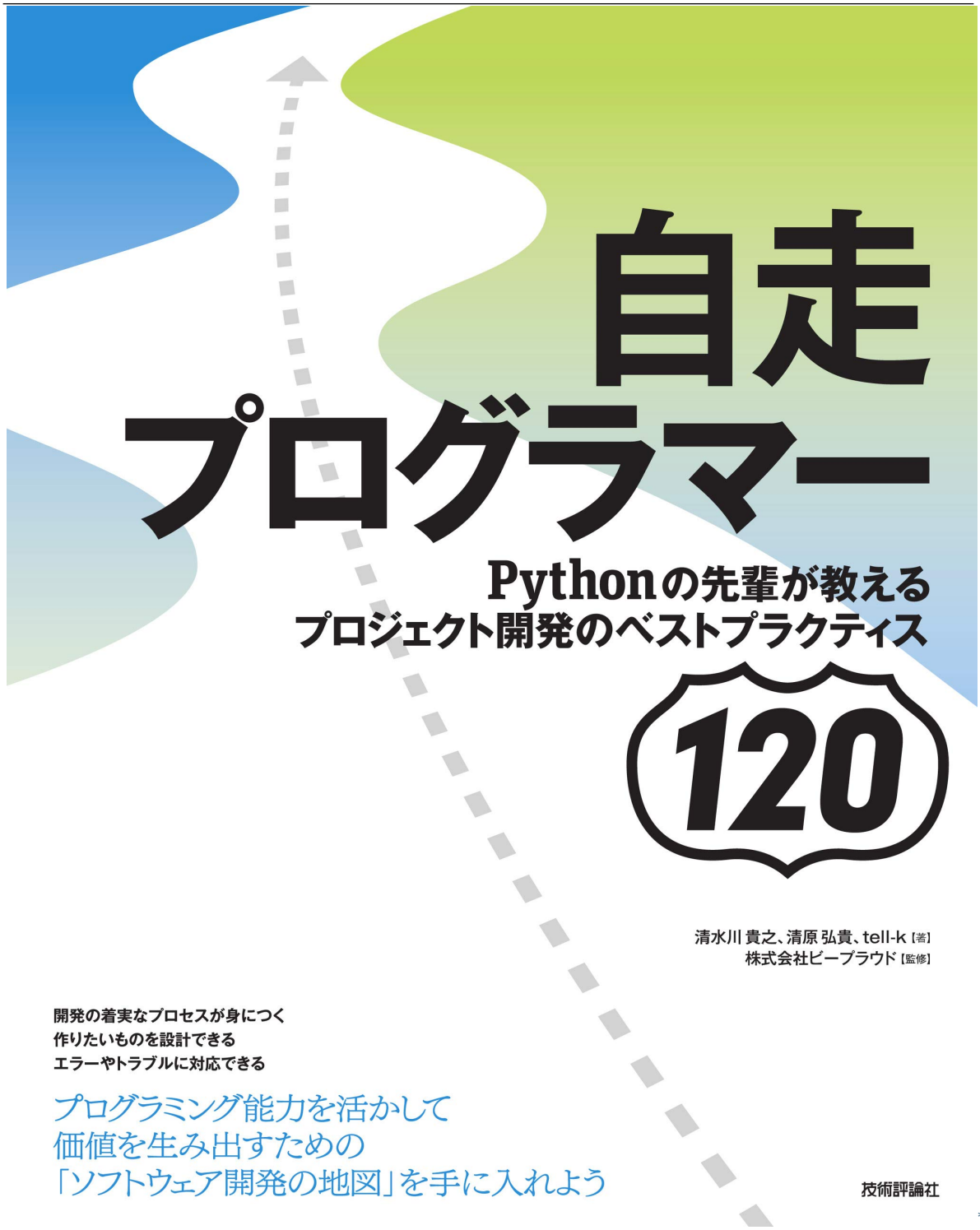
*266 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*267] をご参照ください

フレームワーク以外のライブラリ

1 年に 1 回など、定期的にバージョンを更新していくのが良いでしょう。

[*267] https://gihyo.jp/book/2020/978-4-297-11197-7



*268

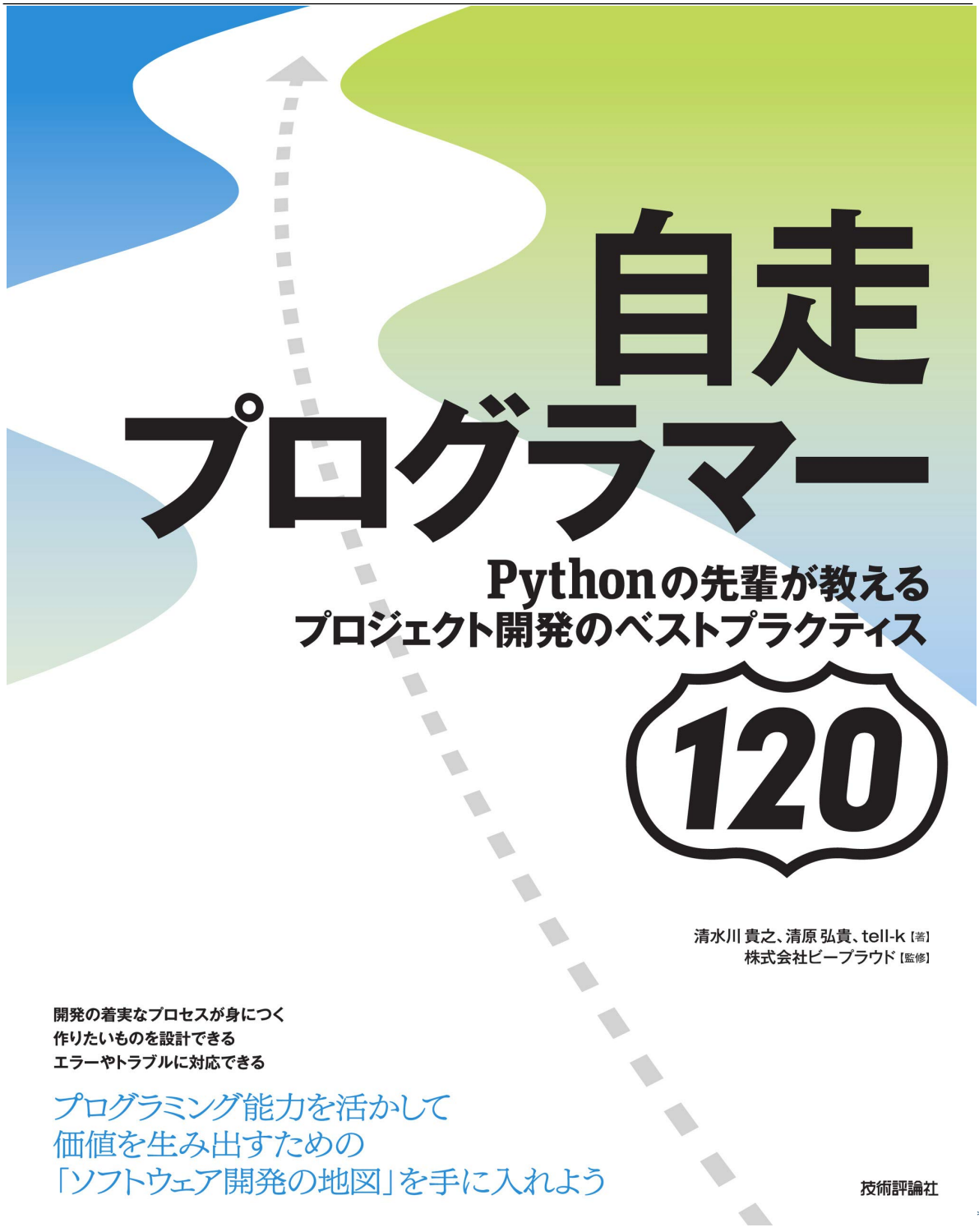
*268 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*269] をご参照ください

開発用のツール

flake8、mypy、pytest、tox といった開発中だけ使用するライブラリは、極端なことを言えばバージョンを上げる必要はありません。半年以上継続する開発プロジェクトであれば、他のライブラリの更新時に合わせてバージョンアップする戦略が良いでしょう。

[*269] https://gihyo.jp/book/2020/978-4-297-11197-7



*270

*270 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*271] をご参照ください

5.4.3 98:フレームワークを使おう（巨人の肩の上に乗ろう）

プログラミング迷子**:** この大量のオレオレフレームワークは何？

- 後輩 W：先輩、今やってるプロジェクトで X さんと一緒に開発してるんですけど、毎日レビューしきれない PR レビューが来てとてもやってられない感じなんです。どうしたらいいんでしょう……。
- 先輩 T：レビューしきれない PR って、どういう状況でそうなったの？
- 後輩 W：今回は Web API のみのサーバーを開発してるんですけど、SQL をたくさん実行する必要があるので、フレームワークを使わなかったんです。
- 先輩 T：うっ、なんだか嫌な予感がする話だね。
- 後輩 W：そしたら、X さんが「必要だから」って新しい仕組みを次々実装しているんですけど、毎日 20 ファイル以上差分のある PR レビュー依頼がバンバン来るんです。でもそれを見てもなんのためにどう動くのかわからないコードの山で、どこから手をつけて良いのかわからなくて。結局レビューが間に合っていなくて、X さん以外はわからないから誰もそのコードに手を出せないんです。
- 先輩 T：それは オレオレフレームワーク っていうやつじゃないかな。使い方のドキュメントは……ないよね？
- 後輩 W：ないですね。「要件に合わせて変更が必要だから、今はドキュメントを書くときじゃない」って、楽しそうに言ってました。

プログラムを書いていると、同じルールを何度も実装することがあります。このとき、たとえばアクセス権限の確認処理があちこちで実装されているとバグが入り込みやすくなります。こういった問題を避けるには、必要な機能を切り出して実装を 1 箇所にまとめる必要があります。

こういった共通化をやり過ぎて「自動的に権限をチェックする BaseView クラスを実装してすべてのビューがクラスを継承して実装する」といったルールが作られることがあります。これが オレオレフレームワーク の始まりですが、それ自体は問題ではありません。大なり小なりどのプロジェクトにもオレオレフレームワークはありますし、Django のようにごく一部で使われていたフレームワークを OSS 化して公開した例はたくさんあります。問題は、公開されていて既に多くのユーザーに使われている良いフレームワークを研究せず、その劣化版を作ってしまうことです。

[*271] https://gihyo.jp/book/2020/978-4-297-11197-7

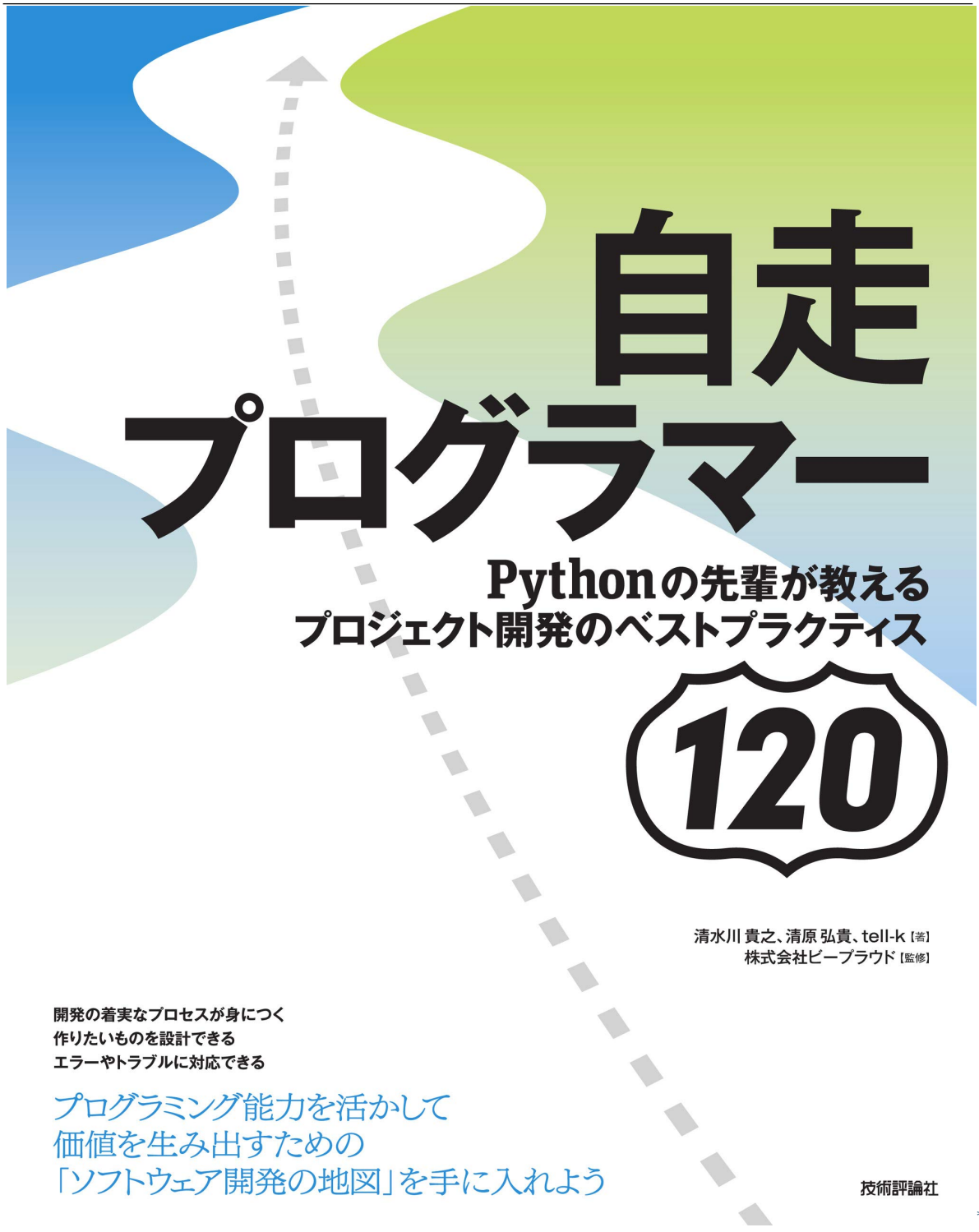
問題を引き起こすオレオレフレームワークは、以下の特徴を持っています。

- 同等機能を持つライブラリを研究していない
- ドキュメントやテストコードがない
- レビューされていない
- 脆弱性があり修正されていない
- DB コネクションやカーソルの解放忘れなど、リソース管理が甘い
- 昨日までの知識で安定的に使えない
- 機能実装よりもフレームワーク修正に時間がかかる

レビューされていない、あるいはレビューが困難なコードが増えていくと、書いた本人以外はコード保守ができないうえに、大量に埋め込まれた潜在的なバグが近い将来顕在化してくることが想像できます。保守が不要な使い捨てコードであればまだしも、通常はこのような オレオレフレームワーク は避けるべきです。

ベストプラクティス

一般的に使われているフレームワークを使いましょう。大きなフレームワークを乗りこなすには時間がかかるかもしれませんが、独自に実装するよりもはるかに少ないコストで課題を解決できます。



*272

*272 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*273] をご参照ください

関連

- *99:* **フレームワークの機能を知ろう** (ページ 360)

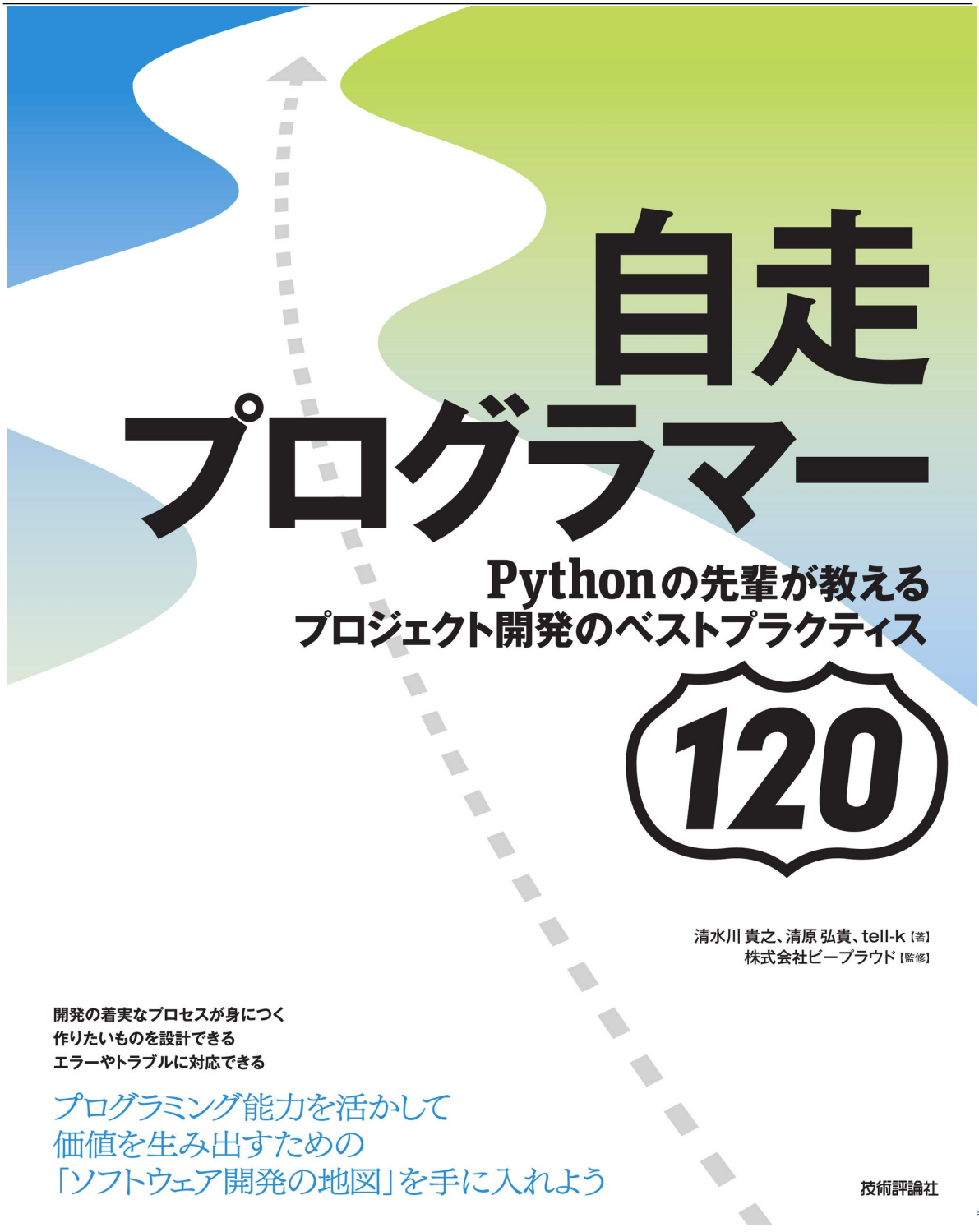
5.4.4 99:フレームワークの機能を知ろう

フレームワークが提供する多くの機能は、安全性が考慮されています。しかしフレームワークをよく知ろうとせずに、似たような機能を独自に実装したり、安全性のための処理を回避する実装をしてしまうと、大きな問題になることがあります。

具体的な失敗

HTML を動的にレンダリングするテンプレートエンジンでは、インジェクション対策として埋め込むデータをエスケープ処理しています。この動作を変更して HTML や JavaScript をそのまま扱うようにしてしまうと、思わぬところから攻撃用のタグやスクリプトを埋め込まれてしまいます。一般に公開するシステムではなく利用者が全員社内のメンバーだとしても、そのメンバーが使い方を間違わない保障はありません。

[*273] https://gihyo.jp/book/2020/978-4-297-11197-7



*274

*274 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*275] をご参照ください

コラム: 安全なウェブサイトの作り方

IPA 独立行政法人 情報処理推進機構 が作成している、以下のドキュメントが参考になります。

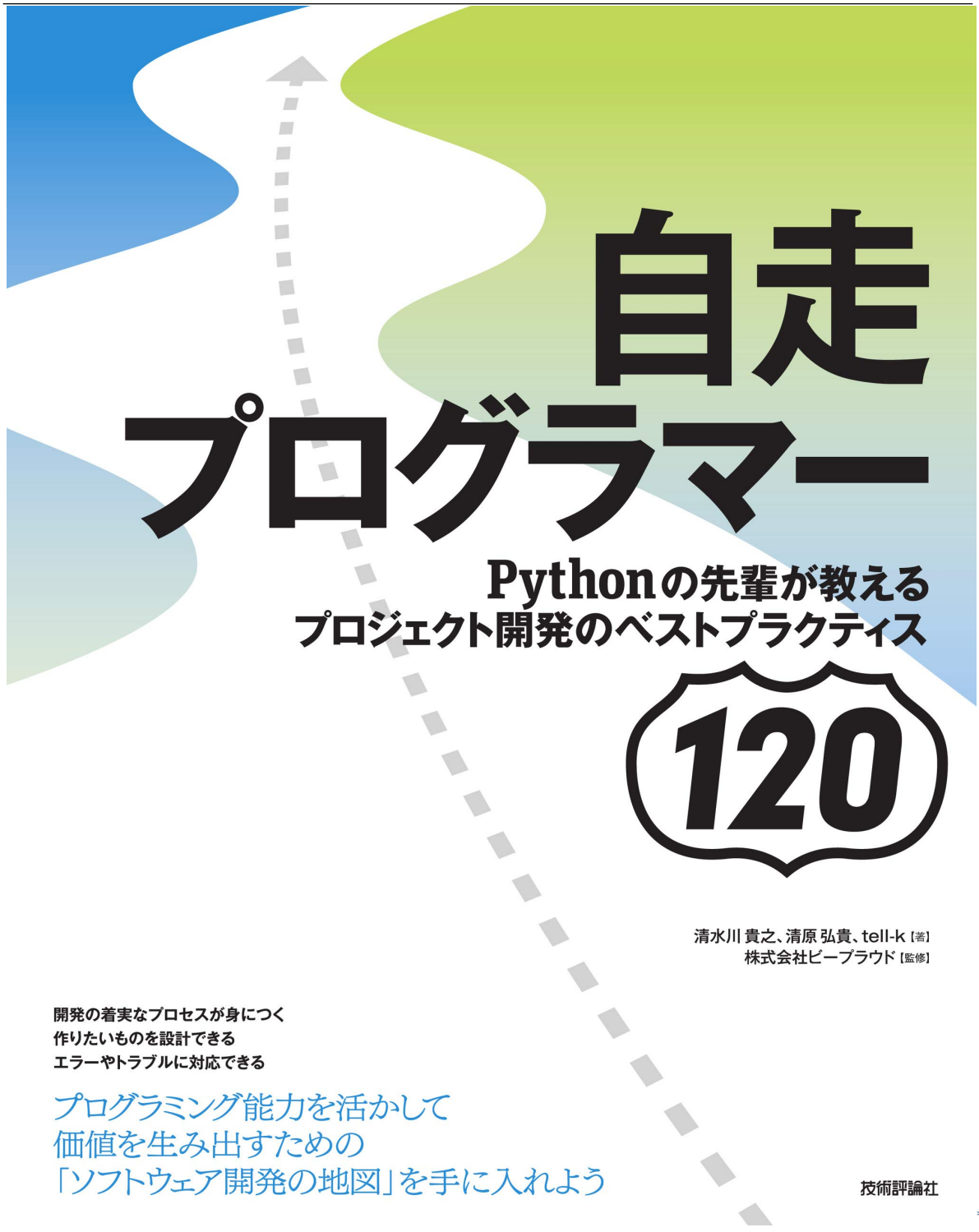
- 安全なウェブサイトの作り方 https://www.ipa.go.jp/security/vuln/websecurity.html
- 安全なウェブサイトの運用管理に向けての 20 ヶ条 〜セキュリティ対策のチェックポイント〜 https://www.ipa.go.jp/security/vuln/websitecheck.html

こういった 制約を回避する実装 や 同等機能の独自実装 は、フレームワークの理解不足によって発生します。Q&A サイトや個人 blog の情報を鵜呑みにして実装してはいけません。役に立つことが多い Q&A サイトにも、間違った情報や安直な回答があることを忘れないようにしましょう。

ベストプラクティス

フレームワークの機能を知りましょう。フレームワークの制約に従いましょう。

[*275] https://gihyo.jp/book/2020/978-4-297-11197-7



*276

*276 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*277] をご参照ください

関連

- *33:* **公式ドキュメントを読もう** (ページ 124)

[*277] https://gihyo.jp/book/2020/978-4-297-11197-7

5.5. リソース設計

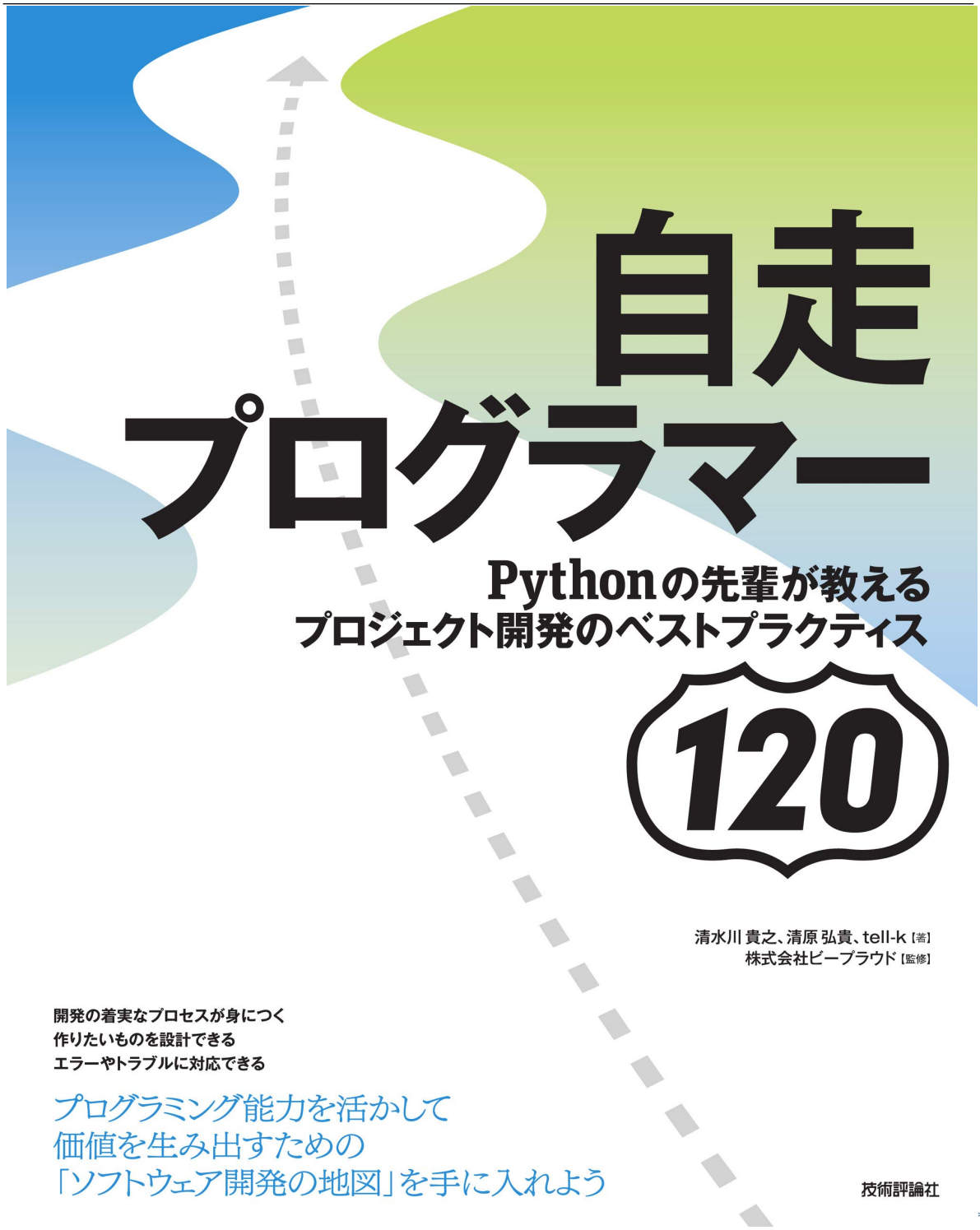
5.5.1 100:ファイルパスはプログラムからの相対パスで組み立てよう

プログラムが外部ファイルを扱うとき、いざ本番にあげたらファイルがあるのにプログラムがファイルを見つけられなくて困ったことはありませんか？プログラムから外部ファイルの位置を指定する方法を見直しましょう。

具体的な失敗

たとえば以下のように CSV ファイルを利用するプログラムがあったとします。

```
# このファイルのパスは「project/scripts/read_csv.py」とする

import csv
from pathlib import Path

CSV_PATH = Path('target.csv')

with CSV_PATH.open(mode='r') as fp:
    reader = csv.reader(fp)
    for row in reader:
        print(row)
```



*278

*278 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*279] をご参照ください

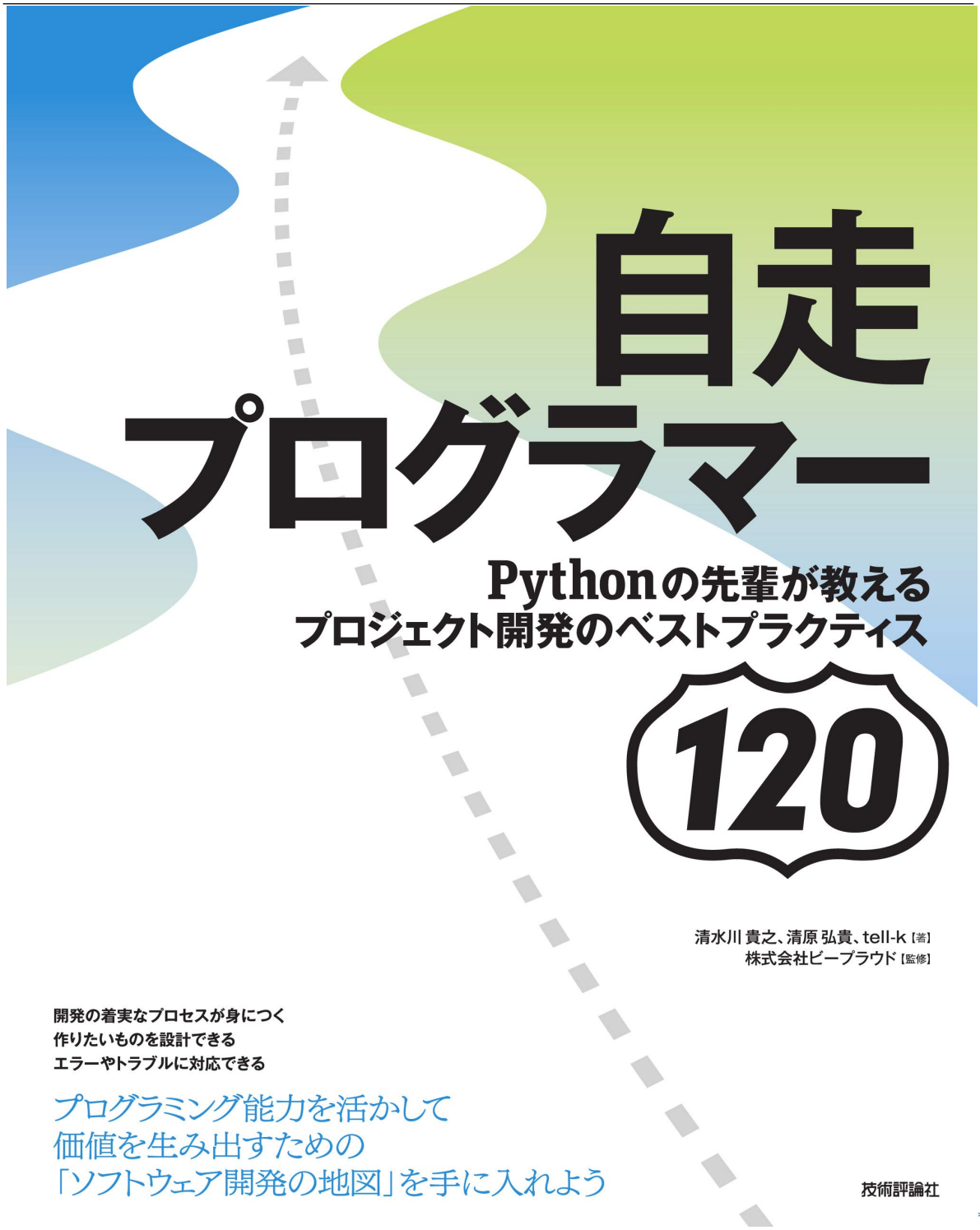
このプログラムは scripts ディレクトリ以外からは実行できないという制限が意図せず生まれてしまっています。

ベストプラクティス

どこからプログラムが実行されても適切に動くようにパスを組み立てましょう。実行されるプログラムを起点したパスを動的に組み立てて利用すると良いでしょう。

```
import csv
from pathlib import Path

# 起点となるプログラムがあるパス
here = Path(__file__).parent
CSV_PATH = here / 'target.csv'

with CSV_PATH.open(mode='r') as fp:
    reader = csv.reader(fp)
    for row in reader:
        print(row)
```

[*279] https://gihyo.jp/book/2020/978-4-297-11197-7



*280

*280 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*281] をご参照ください

関連

- *101:* **ファイルを格納するディレクトリを分散させる** (ページ 369)

5.5.2 101:ファイルを格納するディレクトリを分散させる

プログラミング迷子**: 1** ディレクトリに数万ファイル

- 後輩 W：ユーザーからバグ報告をもらったので調査してるんですが、ls コマンドの結果が表示されるのに数十秒かかってしまって、これってどうにかならないんでしょうか？
- 先輩 T：ls のオプションを指定したり、ls の代わりに find を使う方法とかあるけれど、そもそも、そんなにたくさんのファイルが置かれてるのがまずそうだね。
- 後輩 W：そういうものなんですね……。

たとえば以下のように、作成したすべてのファイルを 1 つのディレクトリに置いてしまうと、パフォーマンスの低下などの問題が発生します。

```
/receipts/receipt-20190718-123456.pdf
/receipts/receipt-20190718-154211.pdf
/receipts/receipt-20190719-081001.pdf
/receipts/receipt-20190720-221020.pdf
```

ファイルが増え続けるシステムの場合、リリース直後は問題になりませんが、ファイル数の増加とともに徐々に影響が出てきます。

[*281] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

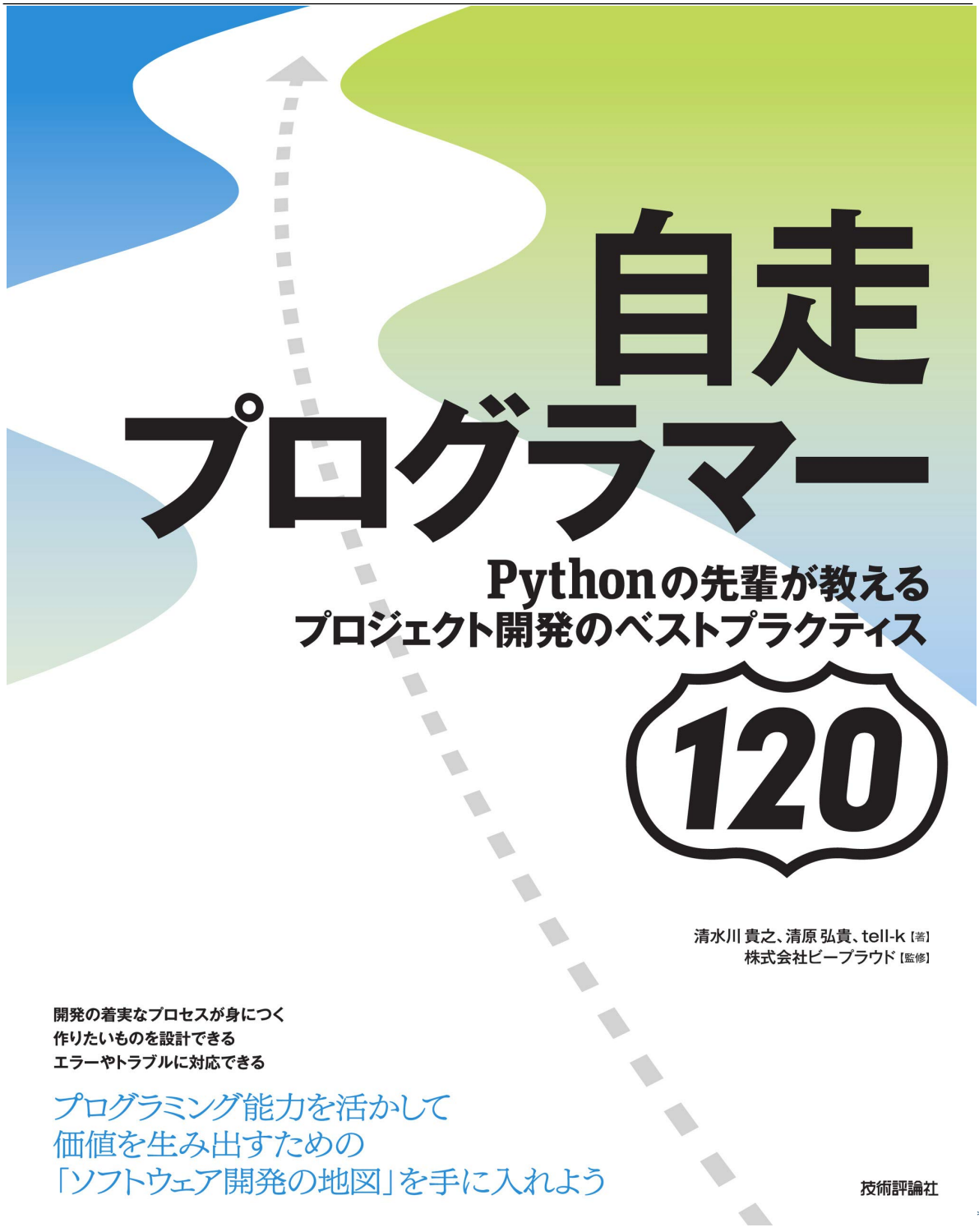
ファイルを格納するディレクトリを分散させましょう。

分散にはいくつかやり方がありますが、元になるデータの ID（データベースで自動採番される ID）を利用してディレクトリを分ける方法がよく使われます。

```
/receipts/123/receipt-20190718-123456.pdf
/receipts/124/receipt-20190718-154211.pdf
/receipts/125/receipt-20190719-081001.pdf
/receipts/126/receipt-20190720-221020.pdf
```

この方法は、レコード単位で複数のファイルを扱う場合などには、直接的でわかりやすい構造です。開発中や障害発生時などには、調査がスムーズに進められます。ただし、デメリットもあります。この方法ではレコード数分だけディレクトリが増えていくため、10 万レコードに対してディレクトリが 10 万個作成され、再び速度低下の原因になってしまいます。

その他にも、ファイル名等の一意な名前からハッシュを生成して、特定の数文字を使ってディレクトリを分ける方法があります。



*282

*282 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*283] をご参照ください

関連

- *102:*一時的な作業ファイルは一時ファイル置き場に作成する (ページ 372)
- *103:*一時的な作業ファイルには絶対に競合しない名前を使う (ページ 374)

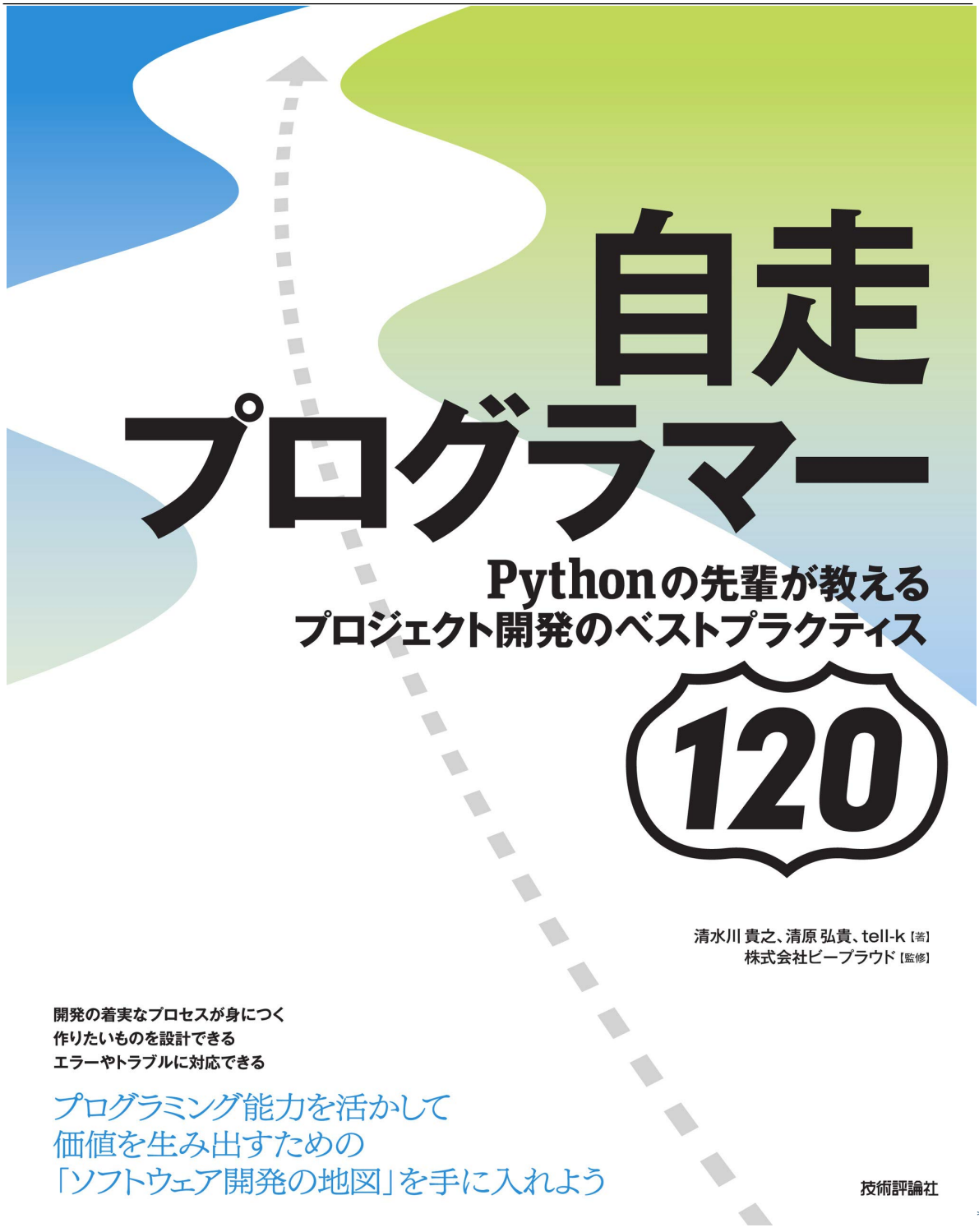
5.5.3 102:一時的な作業ファイルは一時ファイル置き場に作成する

プログラミング迷子**:** 作成済みファイル一覧にゴミファイルが

- 後輩 W：ユーザーから、領収書 PDF 一覧に開けないファイルがあるって連絡が来ました。
- 先輩 T：開けない？　どのファイルなのか聞いた？
- 後輩 W：そのファイルを送ってもらったんですが、`generated_receipt.pdf` というファイル名で、1,024byte しかないんですよ。しかも、一覧ページをリロードしたらそのファイルはなくなったそうです。
- 先輩 T：それ、もしかして書き込み処理中のファイルが見えちゃってたのでは。

作成処理中の一時ファイルを最終的な保存用ディレクトリに作成してはいけません。このとき、作成の途中で別の処理がこのディレクトリにアクセスした場合、書き込みが最後まで完了したファイルと一時ファイル `generated_receipt.pdf` を同等に扱ってしまうと問題になります。

[*283] https://gihyo.jp/book/2020/978-4-297-11197-7



*284

*284 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*285] をご参照ください

ベストプラクティス

一時的な作業ファイルは一時ファイル置き場に作成することで、作成中ファイルへのアクセスの可能性や、エラー時の残骸問題を避けられます。一時ファイル置き場を用意した場合、定期的に不要ファイルをクリーンアップしましょう。

関連

- *103:*一時的な作業ファイルには絶対に競合しない名前を使う (ページ 374)

5.5.4 103:一時的な作業ファイルには絶対に競合しない名前を使う

プログラミング迷子**:** 作業用一時ファイルが競合

- 後輩 W：ユーザーから、領収書をダウンロードしたら別の人の内容の PDF がダウンロードされた、って連絡がありました。
- 先輩 T：えっ、それって大事故じゃない……？　急いで調べよう。
- ― 30 分後―
- 後輩 W：うーん、わからない。一時保存している作業用ファイルが競合してるのかと思ったけど、ちゃんとファイル名に日時を付けてるから大丈夫みたいだし……。
- 先輩 T：ん、日時？　`receipt-20191121-133815.pdf` ってこと？　それだと 1 秒差以内の場合に競合するんじゃない？
- 後輩 W：あっ。

[*285] https://gihyo.jp/book/2020/978-4-297-11197-7

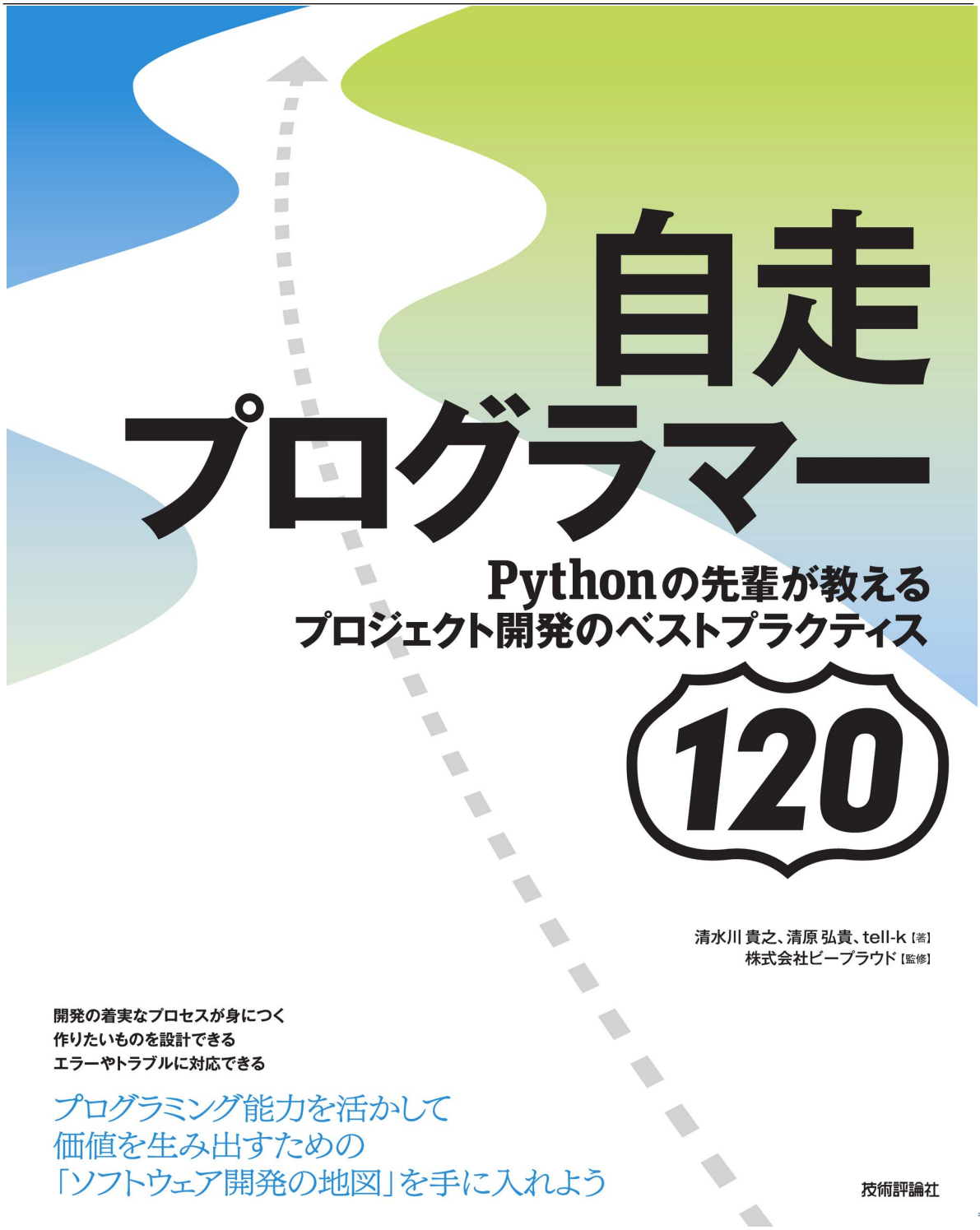
具体的な失敗

この問題は、一時的なファイルが競合する可能性のある名前（例 `receipt-20191121-133815.pdf`）で作成されているために発生します。競合しないようにファイルの命名規則を年月日時分秒で組み立てていますが、秒レベルでの競合は考慮されていませんし、ミリ秒まで指定しても確実とは言えません。

こういった場合、複数のユーザーの操作で 1 つの同じ作業ファイルに上書き保存されてしまいます。その結果、領収書ファイルをダウンロードしてみたら知らない人の領収書だった、という漏洩問題が発生します。

ベストプラクティス

一時的な作業ファイルには絶対に競合しない名前を使いましょう。Python であれば `tempfile` モジュールを使ってください。



*286

*286 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*287] をご参照ください

関連

- *101:*ファイルを格納するディレクトリを分散させる (ページ 369)
- *102:*一時的な作業ファイルは一時ファイル置き場に作成する (ページ 372)

5.5.5 104:セッションデータの保存には RDB か KVS を使おう

プログラミング迷子**: DISK** がいっぱいなので増設します

- 後輩 W： `/tmp` の DISK 容量増やすにはどうすればいいですか？
- 先輩 T：お、なんで `/tmp` の容量を増やしたいの？
- 後輩 W：DISK FULL エラーが出て、調べてみたら `/tmp` がいっぱいで書き込めなくなったみたいなので。
- 先輩 T：それ、先に何が `/tmp` の容量を食ってるか調べたほうがいいよ。たぶん セッション じゃないかな……。
- ― 10 分後―
- 後輩 W： `session-xxxx` っていうファイルがたくさんあったので、これを消します。
- 先輩 T：待って待って、消したらログインしてる人に影響が出るし、消してもまた作られるよ。それに、開発サーバーと違って本番ではサーバーが 2 台あるから、今のままだとユーザーのログイン状態が安定しないような不具合の原因になるよ。

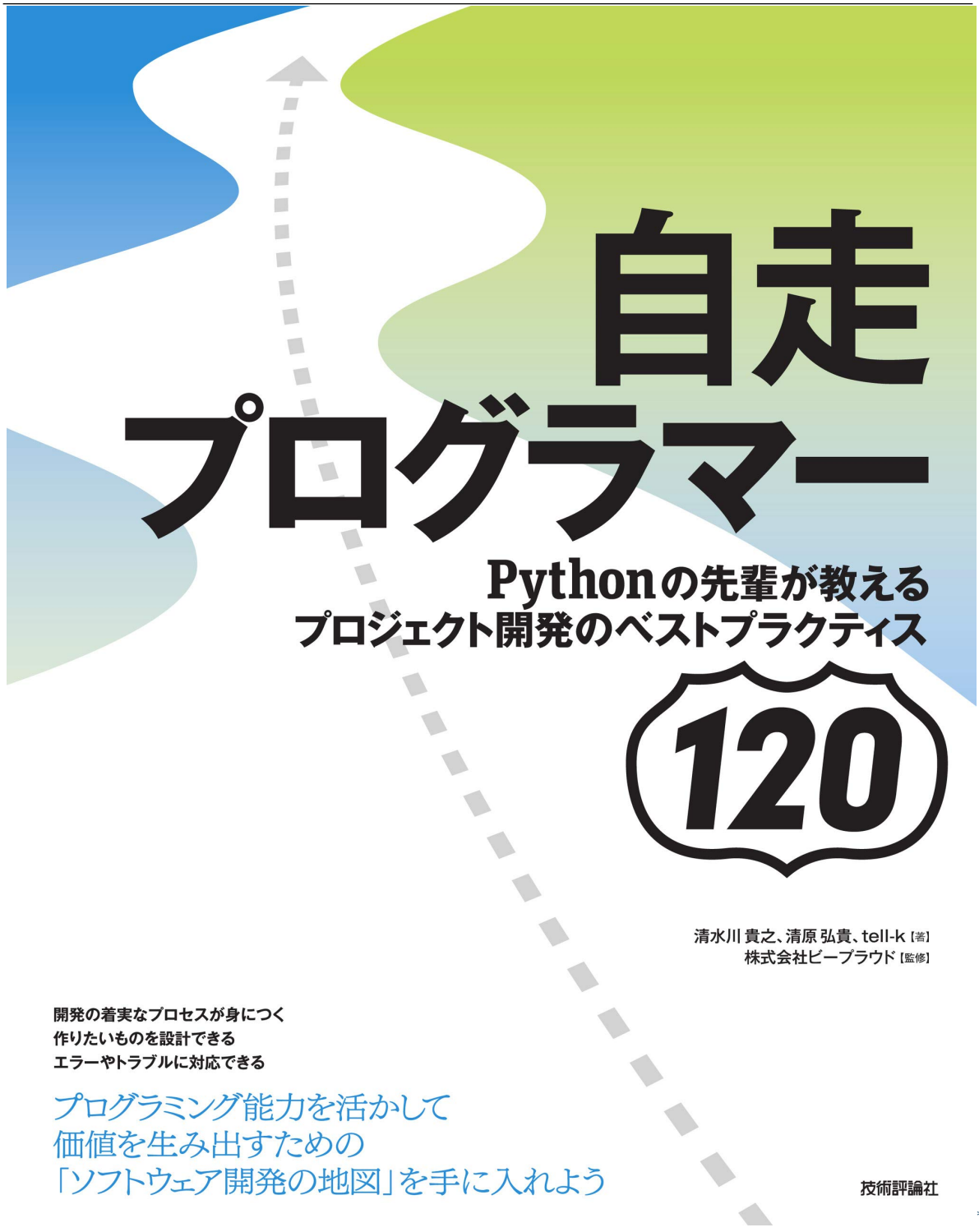
セッションは、ユーザーの一時的な情報を保存するのに使われます。たとえば、ユーザーのログイン状態や、ショッピングカートの内容などです。

セッションのデータをどこに持つかは、Web アプリケーションサーバーで決めることができます。ユーザーのブラウザ上に保存したり、サーバーのファイルシステムやメモリ、データベースなどを選択可能です。Django などの Web アプリケーションフレームワークをよくわからないまま使っていると、サーバーのファイルシステムにセッションを保存するように設定してしまい、セッションを格納したファイルで `/tmp` がいっぱいになってしまうことがあります。

[*287] https://gihyo.jp/book/2020/978-4-297-11197-7

ベストプラクティス

セッションデータの保存には **RDB** か **KVS** を使いましょう。



*288

*288 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*289] をご参照ください

関連

- *101:* **ファイルを格納するディレクトリを分散させる** (ページ 369)

[*289] https://gihyo.jp/book/2020/978-4-297-11197-7

5.6. ネットワーク

5.6.1 105:127.0.0.1 と 0.0.0.0 の違い

コラム: アドレスは合っているのに接続できない

- 後輩 W：開発サーバーで Django を起動したんですが、ブラウザでアクセスできなくて……。
- 先輩 T：お、ブラウザでアクセスしようとしてる URL は何？
- 後輩 W： `http://192.168.99.1:8000/` です。
- 先輩 T：（ `http://localhost:8000/` にアクセスしようとしたわけではないんだな）じゃあ、開発サーバーで Django を起動したときのコマンドとそのあと表示された内容教えてもらえる？
- 後輩 W：こうです。

```
(venv) $ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
April 11, 2019 - 14:03:30
Django version 2.2, using settings 'testproj.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

- 先輩 T：あー、なるほど。その起動方法だと、その Django サーバーは、実行している開発サーバー内からしかアクセスできない状態になっているね。 `python manage.py runserver 0.0.0.0:8000` で起動してみて。
- 後輩 W：こうなりました。

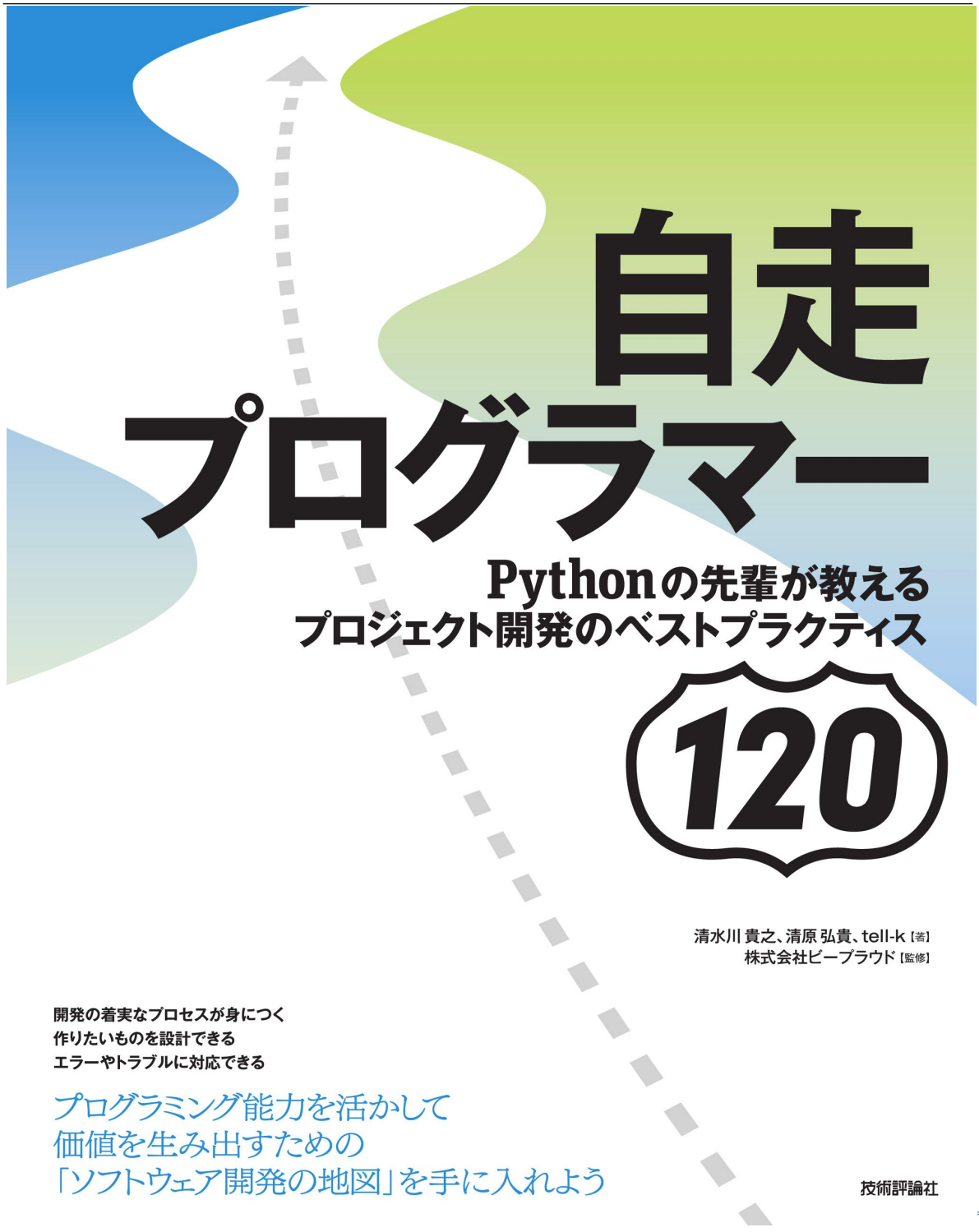
```
(venv) $ python manage.py runserver 0.0.0.0:8000
Performing system checks...

System check identified no issues (0 silenced).
April 11, 2019 - 14:07:53
Django version 2.2, using settings 'testproj.settings'
```

(次のページに続く)

(前のページからの続き)

```
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
```

- 先輩 T： http://192.168.99.1:8000/ にアクセスするとどうなる？
- 後輩 W：できました！　でも **0.0.0.0** って何ですか？



*290

*290 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*291] をご参照ください

ベストプラクティス

サービスを提供したい IP アドレスにバインドしましょう。

開発サーバー や 仮想マシン など、ローカル開発環境 以外で起動した Web サーバーにアクセスする場合は、どのネットワークインターフェースに バインド するか指定が必要です。コンピューター外と直接通信するには、コンピューター外との通信用ネットワークインターフェースの IP にバインドして起動します（【例】`python manage.py runserver 192.168.99.1:8000`）。`0.0.0.0` にバインドすることですべてのネットワークインターフェースと接続できます。

[*291] https://gihyo.jp/book/2020/978-4-297-11197-7

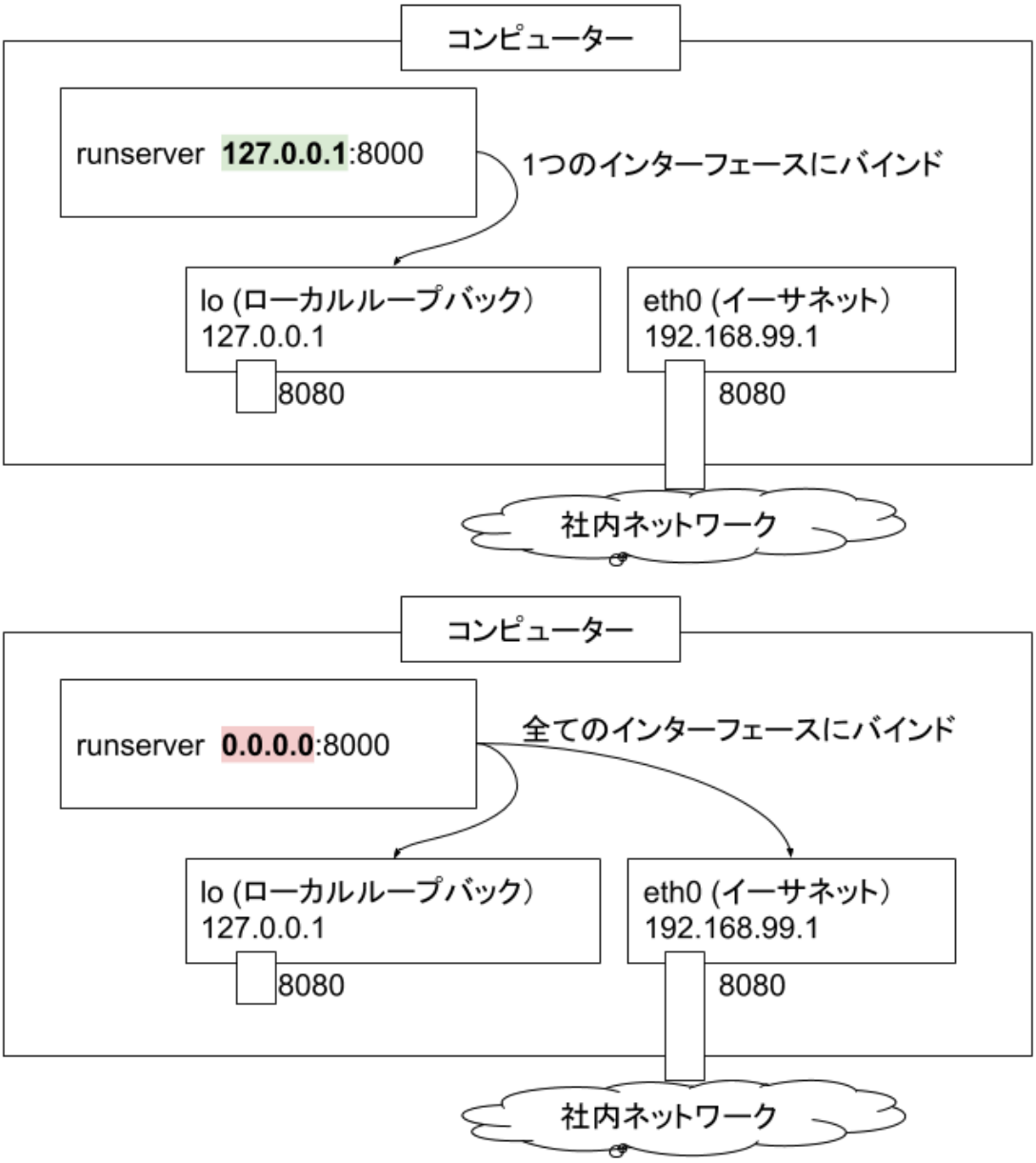
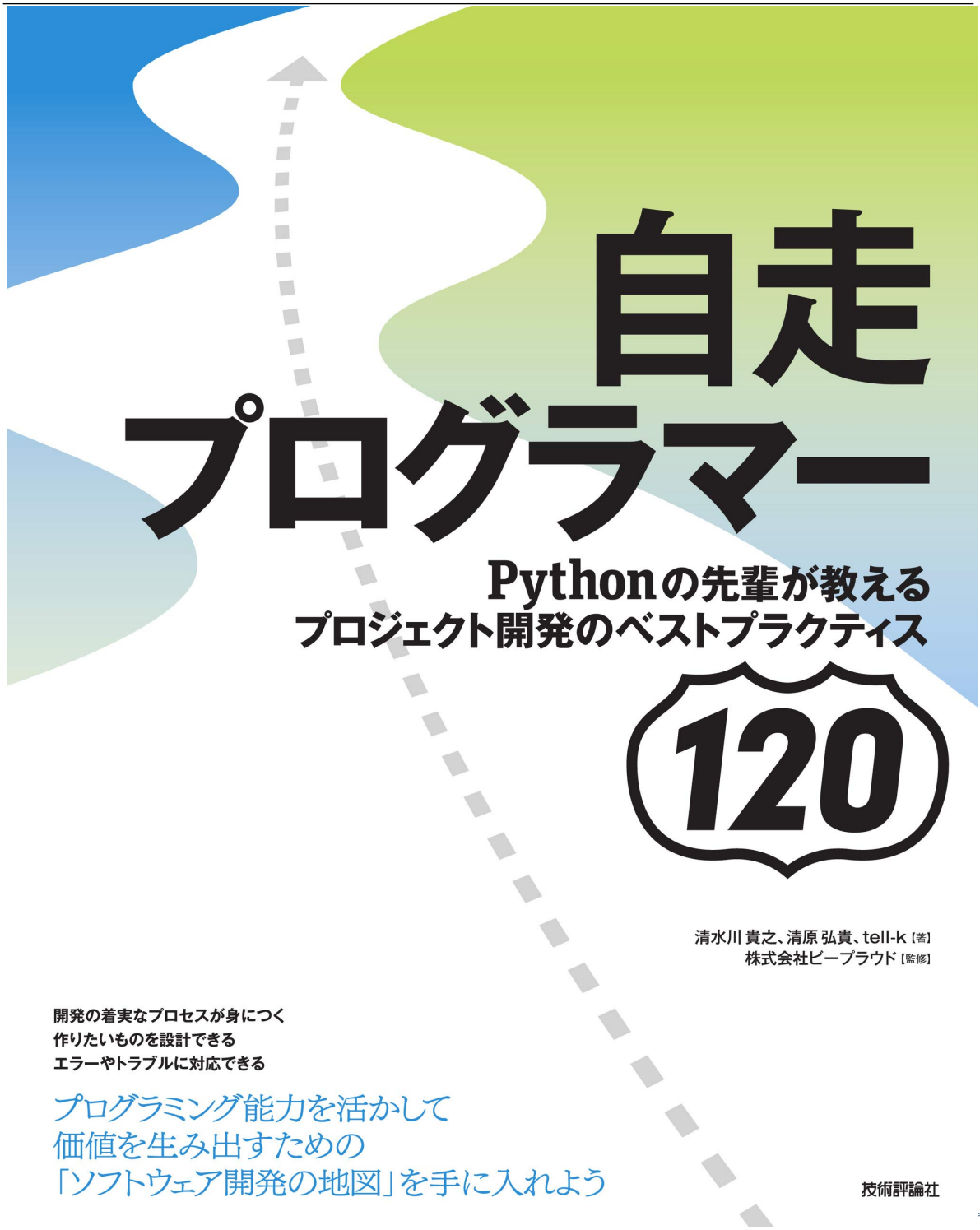


図 5.4　バインド 127.0.0.1 と 0.0.0.0 の違い



*292

*292 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*293] をご参照ください

5.6.2 106:ssh port forwarding によるリモートサーバーアクセス

コラム: インフラ迷子

- 後輩 W：開発サーバーで Django を起動したんですが、ブラウザでアクセスできなくて……。
- 先輩 T：お、以前もそんなこと言ってなかったっけ？（*105:127.0.0.1 と 0.0.0.0 の違い* (ページ 381)）
- 後輩 W：はい、 `http://192.168.99.1:8000/` でアクセスできるようになったんですが、今日は社外からアクセスができなくて……。
- 先輩 T：あー、社外。 `http://192.168.99.1:8000/` は社内のアドレスだから、社外からはつながらないですね。
- 後輩 W：T さんは社外からいつもどうやってつないでるんですか？
- 先輩 T： ssh port forwarding を使ってるよ。開発サーバーに ssh 接続はできてるよね？
- 後輩 W：はい、それはできてます。
- 先輩 T：じゃあその ssh 接続のときのコマンドに `-L 8000:localhost:8000` っていうオプションを付けて ssh 接続してみて。
- 後輩 W：しました。
- 先輩 T： `http://localhost:8000/` にアクセスするとどうなる？。
- 後輩 W：できました！

ベストプラクティス

ssh port forwarding は、ssh 接続を利用して、外部のネットワークから直接通信できないポートへの接続を可能にする技術です。対象のサーバーと直接 http 通信できない場合であっても、そのサーバーに ssh 接続できるのであれば、ssh port forwarding で任意のポートと通信できます。

以下のコマンドは、ssh port forward を行っている例です。

[*293] https://gihyo.jp/book/2020/978-4-297-11197-7

```
$ ssh server.example.com -L 8000:localhost:80
```

このコマンドでの ssh の接続先は server.example.com です。接続元 PC のポート 8000 を接続先の server.example.com から見て localhost:80 に接続するようにトンネルを作成します。

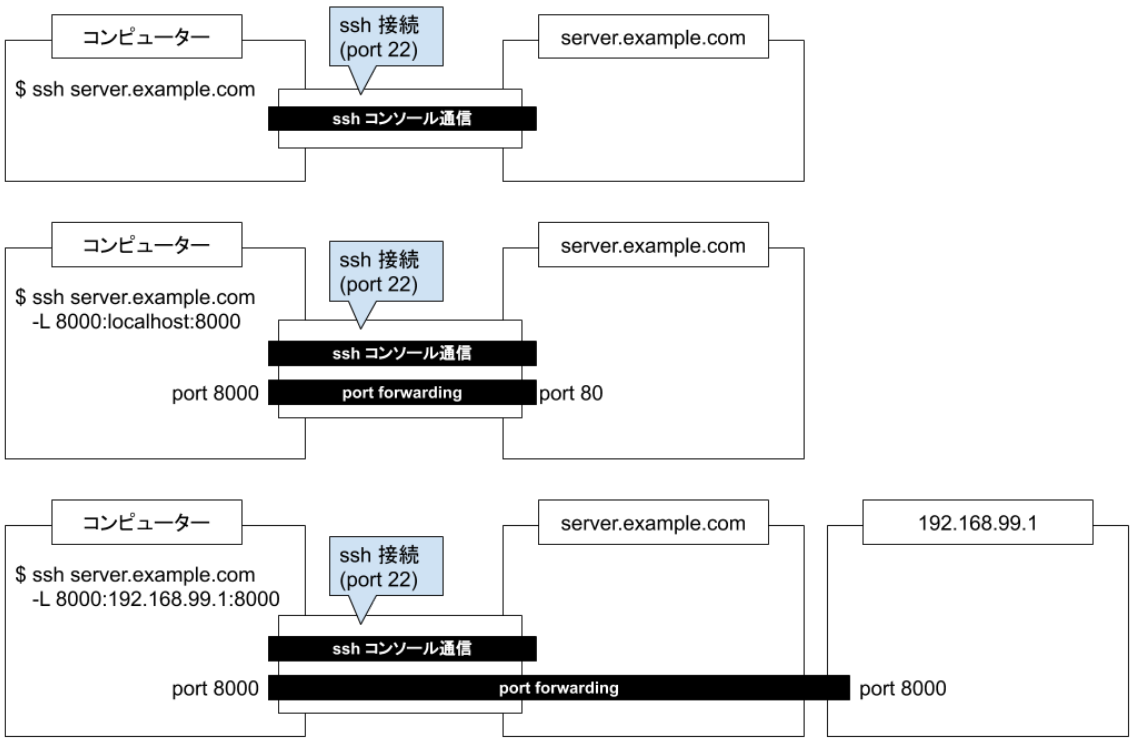
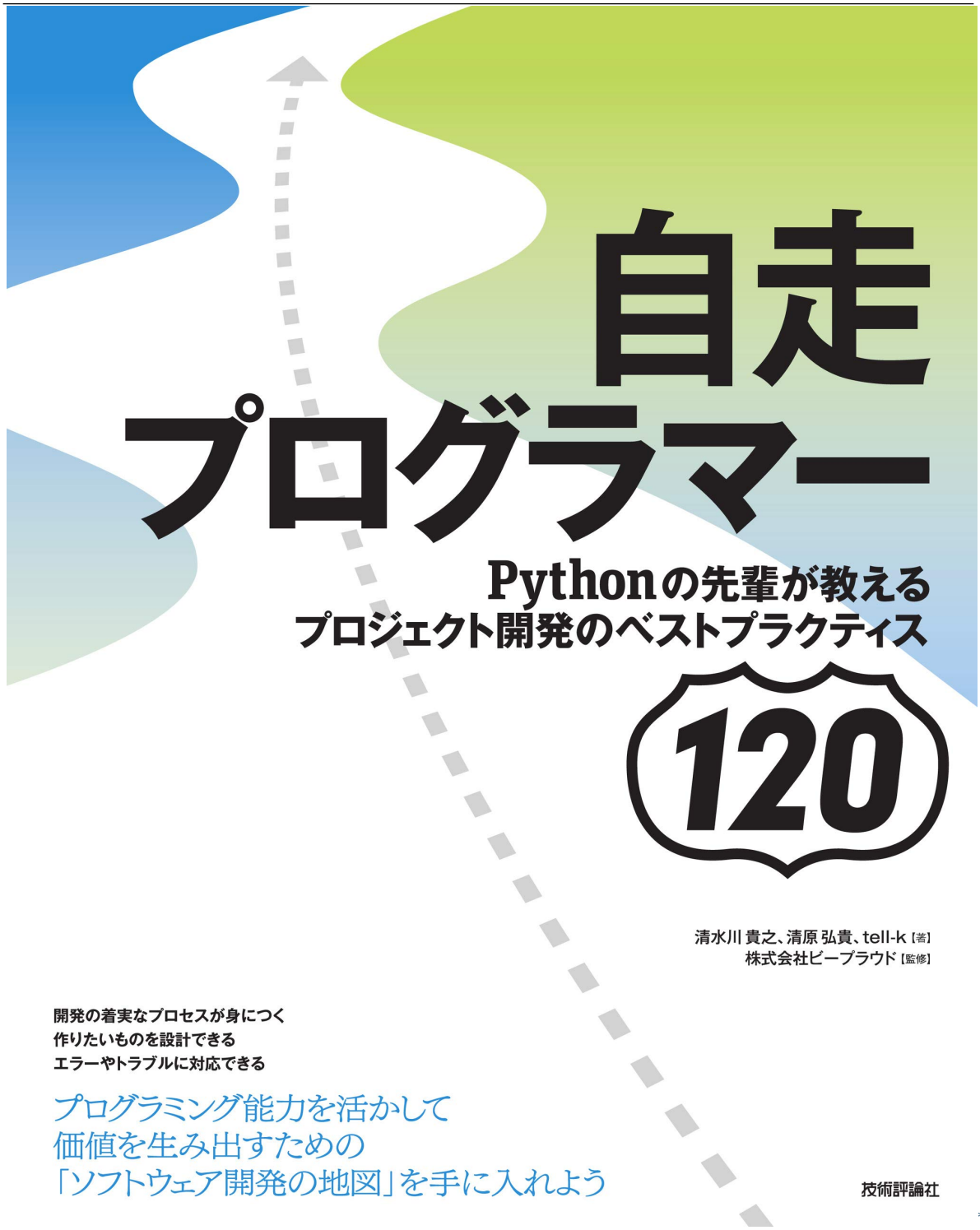


図 5.5　ssh port forwarding のイメージ



*294

*294 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*295] をご参照ください

関連

- *110:hosts* **ファイルを変更してドメイン登録と異なる** *IP* **アドレスにアクセスする** (ページ 400)

5.6.3 107:リバースプロキシ

プログラミング迷子**: Web** アプリケーション開発は覚えることが多い

- 後輩 W：Django を Gunicorn で起動してるんですが、ページの表示が重い気がするんです。アクセスがちょっと増えただけでサーバーの負荷もけっこう高くなってしまうし……。サーバースペック上げたほうが良いんでしょうか？
- 先輩 T：どれどれ……あれ、Gunicorn を直接ネットに公開してるの？　これだと静的ファイルも全部 Django で処理するから、CPU とメモリにかなり負荷がかかるね。Web サーバーを立てて リバースプロキシ するべきだよ。
- 後輩 W：リバースプロキシ……?って何ですか？
- 先輩 T：Web サーバーで受け取ったリクエストをバックエンドの Gunicorn に渡すやつがリバースプロキシだよ。セキュリティーの観点からも、フロントの Web サーバーを立てよう。

Web アプリケーションサーバー（Gunicorn ＋ Django）を直接ネットに公開した場合、すべての HTTP リクエストを Gunicorn ＋ Django で処理して返すことになります。この構成の場合、Django はリクエストされた画面だけでなく、その画面を表示するのに必要な CSS や JavaScript、画像など、動的に処理する必要がない静的ファイルについてもファイル 1 つ毎にリクエストを受けて、返します。リクエストを受けたページで、CSS ファイルを 5 個、JavaScript ファイルを 5 個、画像を 5 個、利用している場合、ブラウザからはページ本体以外に 15 回のリクエストが送られます。こういった静的ファイルのリクエストをすべて Python 等のプログラムで処理すると、どうしても時間がかかってしまいます。

また、インターネットでは Web サイトに対してロングポーリング[*296] や巨大なリクエストを送りつける[*297] といった多種多様な攻撃が日々繰り返されています。こういった攻撃に対抗する仕組みは Gunicorn や Django では提供されていません。

[*295] https://gihyo.jp/book/2020/978-4-297-11197-7

[*296] リクエストデータを 1 秒に 1 文字といった低速でサーバーに送信するリクエストを複数同時に行い、サーバー側の同時接続数を溢れさせ、他の利用者がサービスを利用できなくする攻撃。

[*297] 数百 MB、数 GB といった巨大なリクエストをサーバーに送信することで、サーバーのメモリを溢れさせる攻撃

ベストプラクティス

Web サーバー として Apache や Nginx などを設置し、 Web アプリケーションサーバー にリバースプロキシで接続しましょう。

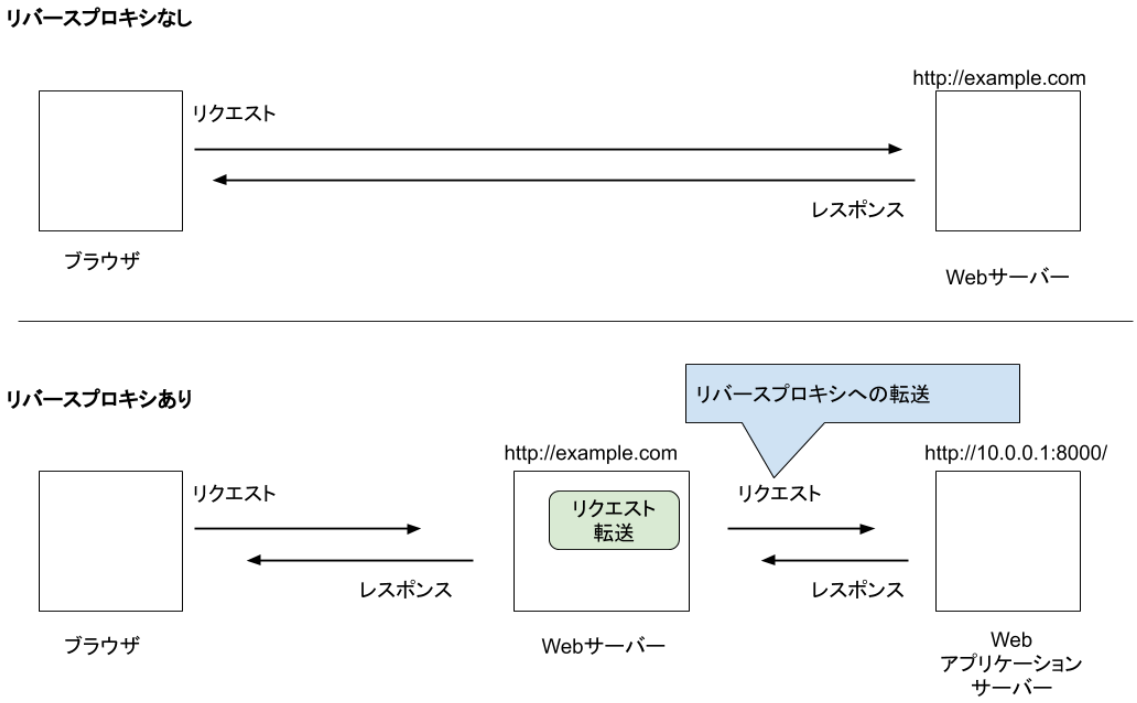
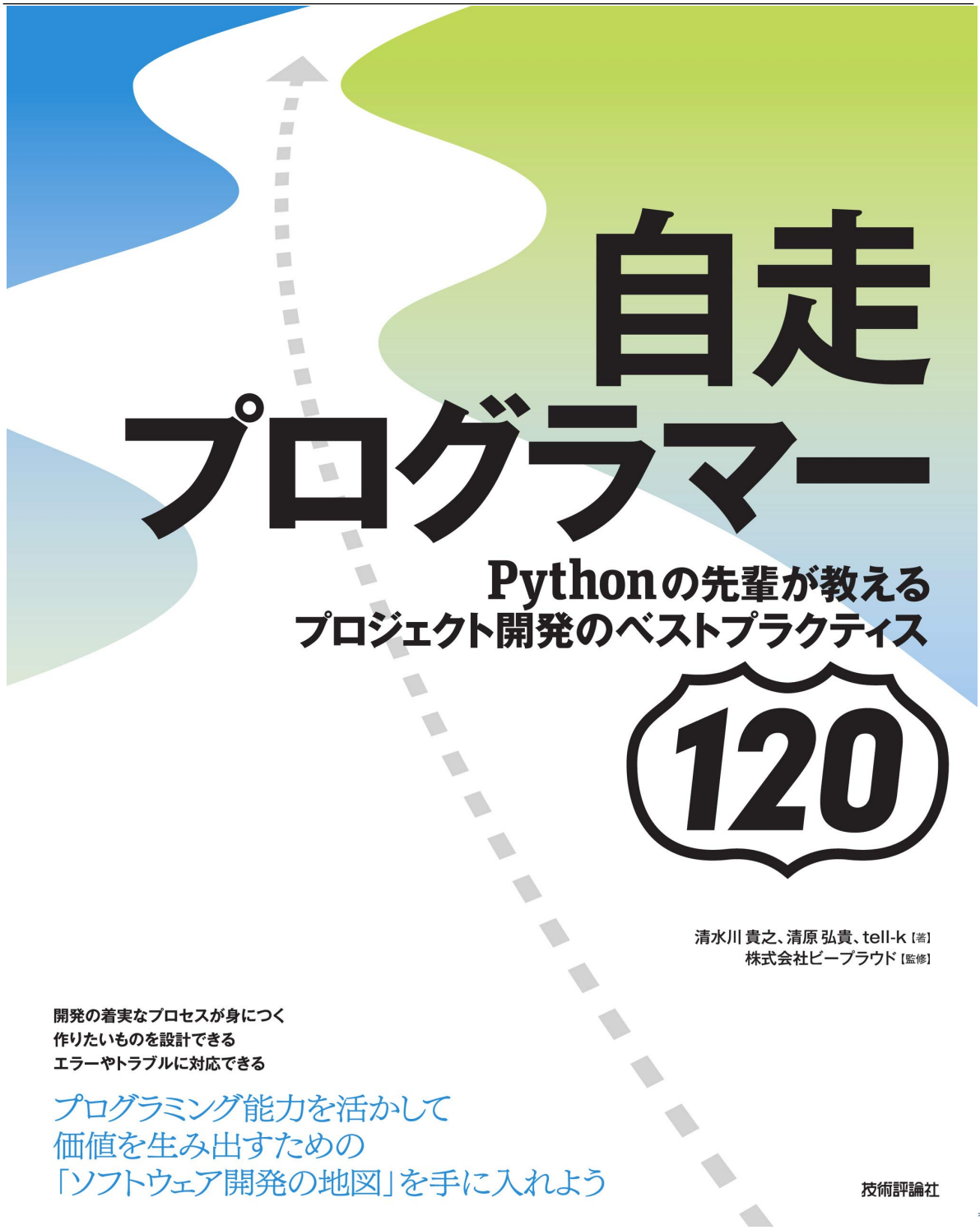


図 5.6　リバースプロキシ



*298

*298 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*299] をご参照ください

関連

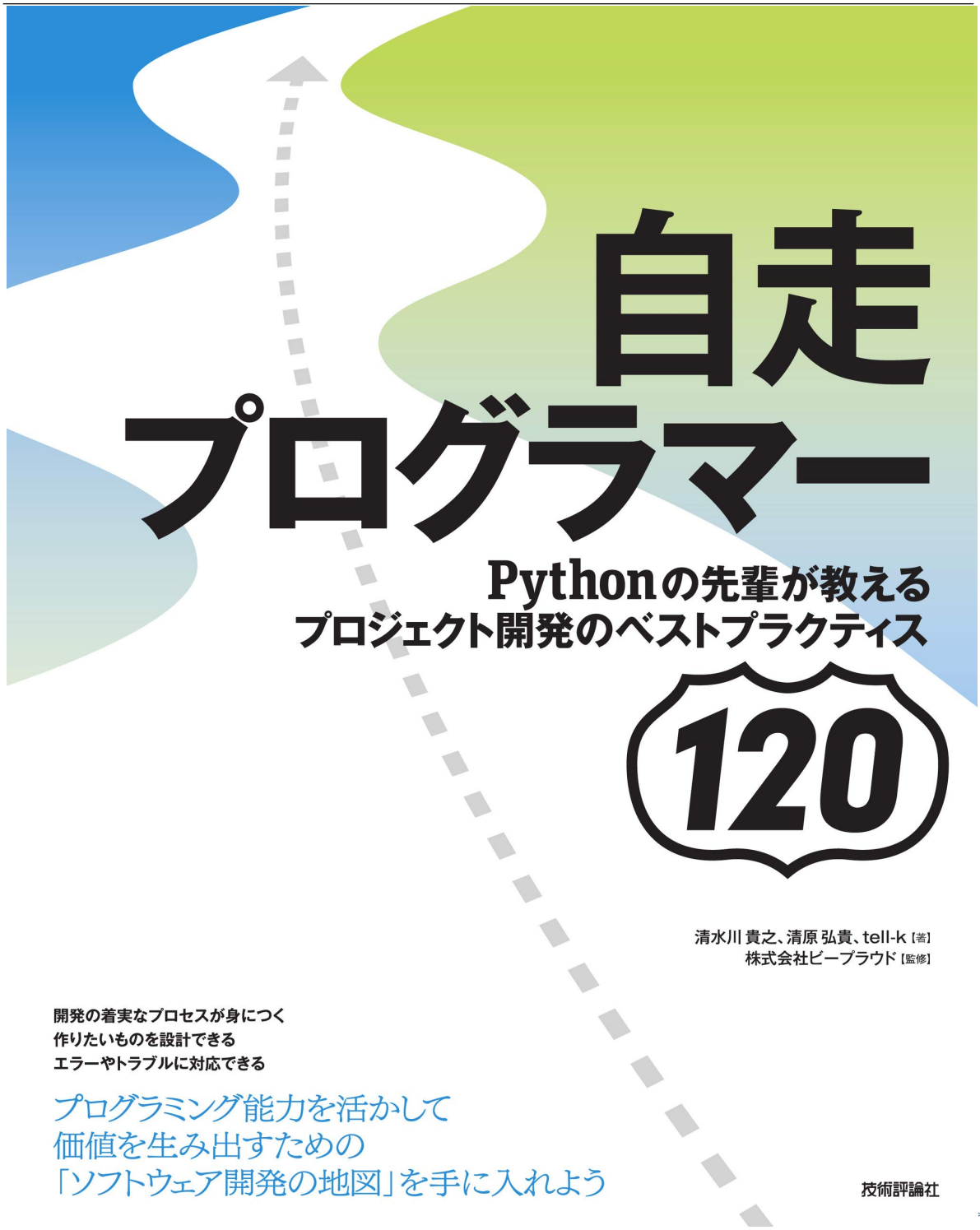
- *93:*サービスマネージャーでプロセスを管理する (ページ 338)

5.6.4 108:Unix ドメインソケットによるリバースプロキシ接続

コラム: 謎のファイル **.sock**

- 後輩 W：Nginx から `unix:/var/run/gunicorn.sock` と指定する手順だったので指定したけれど、`No such file or directory` というエラーが出ました。`ls /var/run/` してみたらファイルがなかったので別の環境から `gunicorn.sock` をコピーしてきたけど、動きません。
- 先輩 T：おっと、`gunicorn.sock` はファイルじゃないからコピーで持ってきてもだめだぞ。
- 後輩 W：ファイルじゃない？？
- 先輩 T：たぶん、Gunicorn が　`gunicorn.sock` を用意する構成だと思うけど、Gunicorn の起動コマンドオプションはどうなってる？
- 後輩 W：systemd で `gunicorn -b 0.0.0.0:8000 apps.wsgi:application` になってます。
- 先輩 T：なるほど、それだと Gunicorn は TCP 8000 で待ち受けしてるのに Nginx が Unix ドメインソケットでリバースプロキシ接続しようとしてエラーになってるんだね。

[*299] https://gihyo.jp/book/2020/978-4-297-11197-7



*300

*300 https://gihyo.jp/book/2020/978-4-297-11197-7

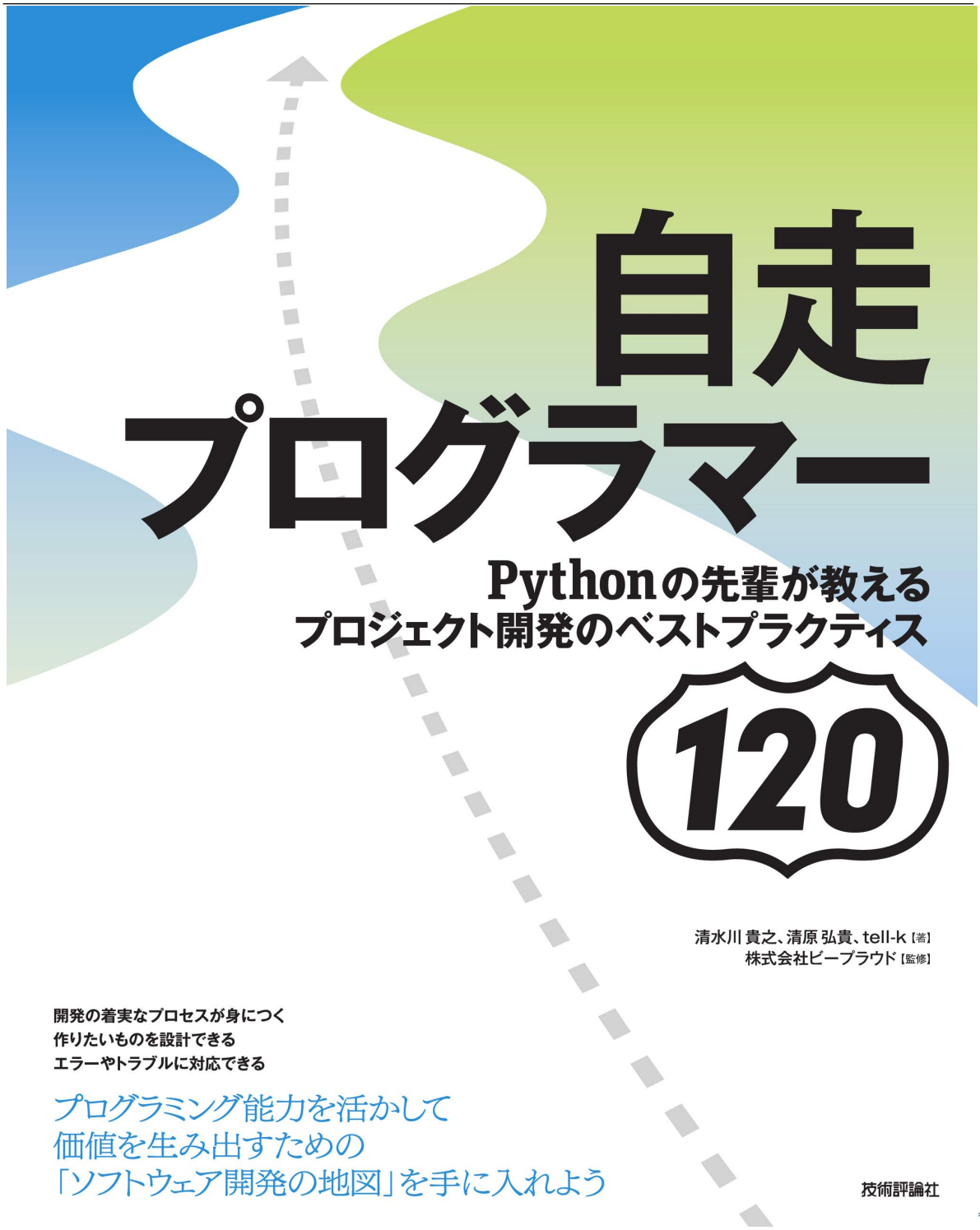
（中略）詳細は書籍 自走プログラマー[*301] をご参照ください

ベストプラクティス

Web サーバーと Web アプリケーションサーバーの通信方式を合わせましょう。可能なら、TCP よりも高速な Unix ドメインソケットによるリバースプロキシ接続を使用しましょう。

Unix ドメインソケット は ソケット[*302] の一種で、ネットワーク通信で使います。ソケットには、Unix ドメインソケットの他に、 TCP/IP や UDP などがあります。ソケット通信を行うには、TCP/IP 通信であれば <IP>:<PORT> を使用しますが、Unix ドメインソケットによる通信では、ファイルパスを使用します[*303] 。待ち受け側と接続側の両方でこのファイルパスを使うことで、ソケット通信ができるようになっています。

[*301] https://gihyo.jp/book/2020/978-4-297-11197-7
[*302] https://docs.python.org/ja/3/howto/sockets.html
[*303] 他に、無名ソケットや、抽象名前空間を使ったソケットをバインドできます。詳しくは次のページを参照してください: https://linuxjm.osdn.jp/html/LDP_man-pages/man7/unix.7.html



*304

*304 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*305] をご参照ください

5.6.5 109:不正なドメイン名でのアクセスを拒否する

プログラミング迷子**: IP** アドレス宛の無差別攻撃

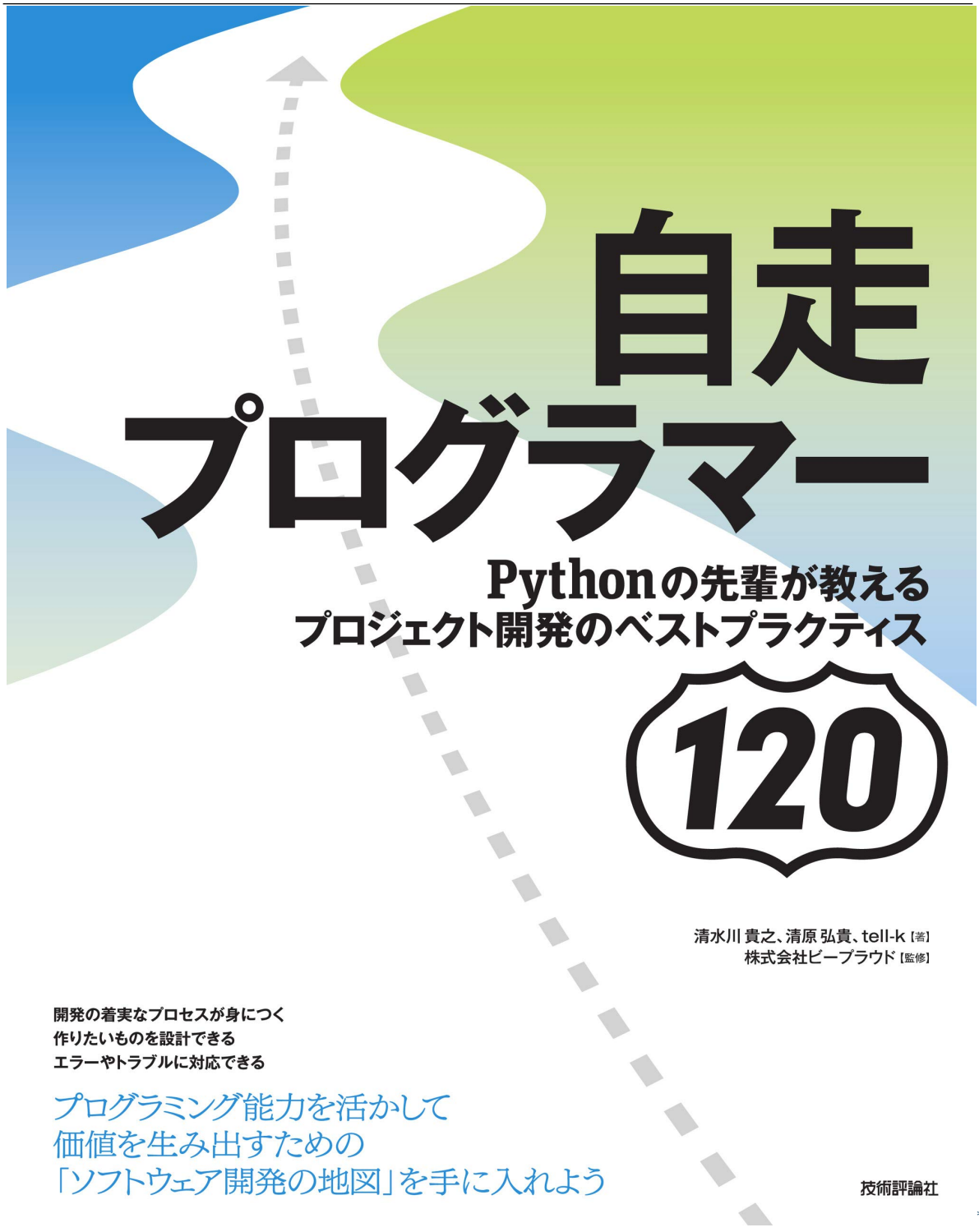
- 後輩 W：Django でエラーが起きて `Invalid HTTP_HOST header: '91.92.66.124'. You may need to add '91.92.66.124' to ALLOWED_HOSTS.` というタイトルのメールがたくさん届くんですけど、どうしたらいいんでしょう？
- 先輩 T：なるほど、bot が IP アドレス直でアクセスしに来てるんだね。どうすれば良いと思う？
- 後輩 W：調べてみます……その IP アドレスにブラウザでアクセスすると 403 エラーになってエラーが再現するので、アクセスできるように `settings.py` の `ALLOWED_HOSTS` に **91.92.66.124** を加えればよさそうです。
- 先輩 T：それはちょっと安直だね。Django の公式ドキュメントには `ALLOWED_HOSTS` の目的が詳しく書いてあるよ[*306] 。
- 後輩 W：読みます……なるほど、 `ALLOWED_HOSTS` は攻撃を防ぐためにあるから、アクセス許可するのは悪手ってことですね。エラーメール通知を完全にオフにするのは良くなさそうだし、今はエラーメールが多いと言っても日に 10 通程度なのでこのままにしておくのが良さそうです。
- 先輩 T：それだと対応が必要なエラー通知が埋もれちゃうだろうね。それに不要なアクセスが Django まで届いているのも良くないよ。

エラーメッセージには、多くの場合エラーの直接の原因が書かれています。しかし、この `ALLOWED_HOSTS` のケースでは指示通りに対処すると、かえって問題を深刻にしていまいます。

[*305] https://gihyo.jp/book/2020/978-4-297-11197-7
[*306] https://docs.djangoproject.com/ja/2.2/ref/settings/#allowed-hosts

具体的な失敗

インターネット上では、ウイルスや bot などによってすべての IP アドレスに対して無差別に攻撃が行われています。`ALLOWED_HOSTS` はそのような攻撃を防ぐことが目的の設定なため、エラーメッセージで `You may need to add '91.92.66.124'` と言われたからといって安直に追加してはいけません。また、こういった攻撃の中には、IP アドレスではなく本来とは異なるドメイン名でアクセスすることでプログラムの脆弱性を突いて侵入しようとするケースもあります。発生件数が少ないからといってエラーを放置してしまうと Django アプリが攻撃に晒され、たとえ攻撃が無効だとしても Django でのリクエスト処理でサーバーリソースが占有されてしまいます。



*307

*307 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*308] をご参照ください

関連

- *107:*リバースプロキシ (ページ 390)
- *33:*公式ドキュメントを読もう (ページ 124)

5.6.6 110:hosts ファイルを変更してドメイン登録と異なる IP アドレスにアクセスする

プログラミング迷子**:** 名前ベースのバーチャルホストを設定したら **ssh port forwarding** 経由でアクセスできなくなった

- 後輩 W：ssh port forwarding で localhost の 8000 番ポートを開発サーバーの 80 番ポートに転送したんですが、ブラウザから、 `http://localhost:8000/` にアクセスしてもサイトが表示されませんでした。
- 先輩 T：Nginx の設定で `localhost` が不正なドメイン名として扱われてるんじゃない？
- 後輩 W：正しい URL は `http://app.example.com/` ですけど、今は IP 制限しているので社外からはアクセスできないんです。こういう場合、どうすればいいですか？
- 先輩 T：端末の hosts ファイルを変更して、ドメインの IP を指定すれば良いよ。

ssh port forwarding は localhost のポートへのアクセスを転送する仕組みです。このため、ブラウザで転送先のサーバーにアクセスしようとした場合、ドメイン名は localhost を指定する必要があります。しかし、通常そのような正式名以外のドメイン名でのアクセスは拒否するよう設定されています。

ベストプラクティス

hosts ファイル を変更して、ドメイン名に任意の IP アドレスを関連づけます。hosts ファイルは DNS よりも先に参照される、IP アドレスとドメイン名の対応を記載したテキストファイルです。今回の例では、以下の内容を `/etc/hosts` ファイルに追記します[*309] 。

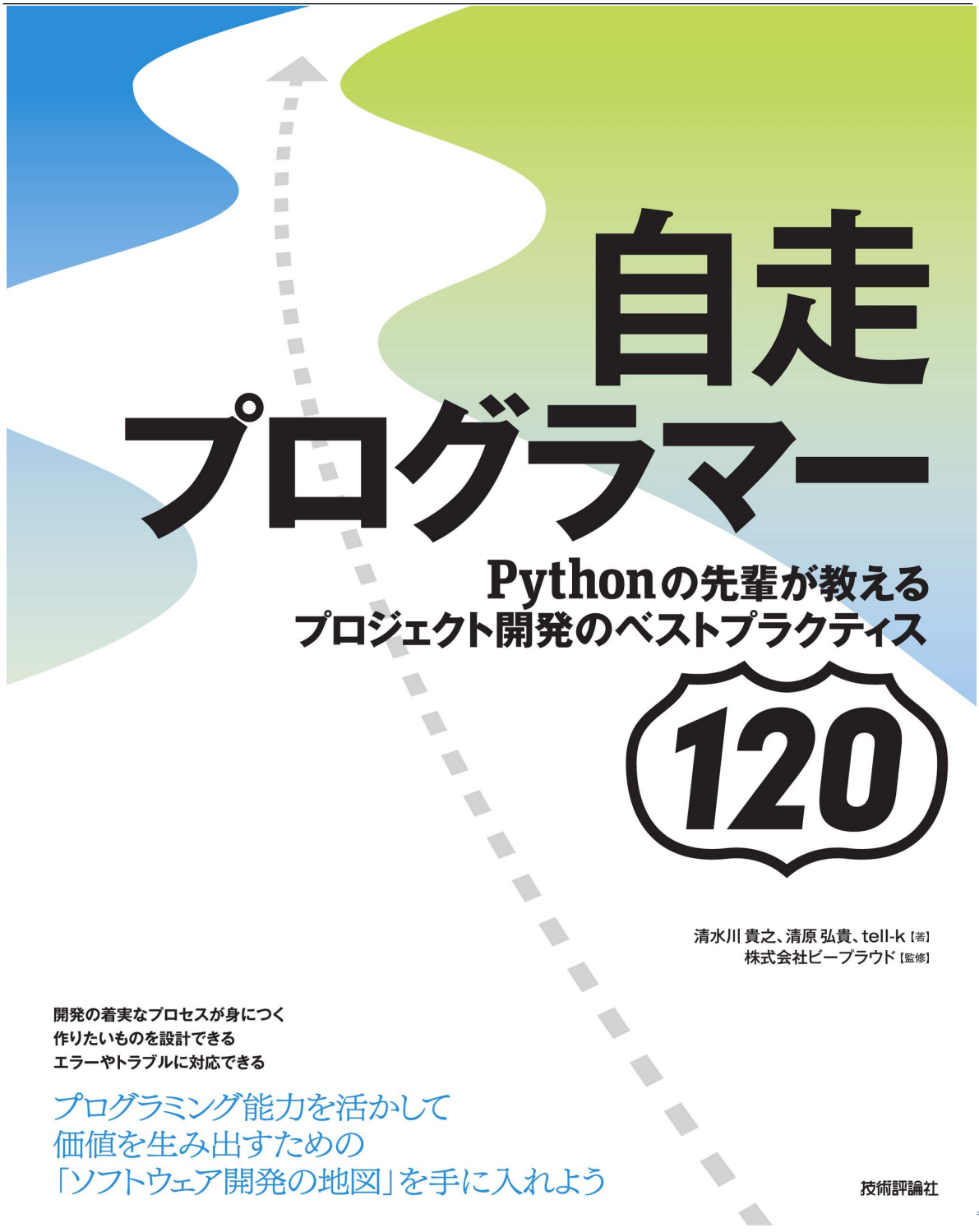
[*308] https://gihyo.jp/book/2020/978-4-297-11197-7
[*309] Windows では `C:\Windows\System32\drivers\etc\hosts` にあります

リスト 5.6　/etc/hosts

```
127.0.0.1 app.example.com
```

これで、 app.example.com への通信は IP アドレス 127.0.0.1 へ送信され、ssh port forwarding 経由でサーバーへリクエストが送られます。このように 閉じられた環境の Web サイトにアクセスするときに使うと便利です。

/etc/hosts を変更すれば、存在しないドメインの定義も行えます。この方法で、DNS に登録される前に正式なドメイン名を使った動作確認をしたり、DNS の切り替え検証などに利用できます。



*310

*310 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*311] をご参照ください

関連

- *106:ssh port forwarding* **によるリモートサーバーアクセス** (ページ 387)
- *109:***不正なドメイン名でのアクセスを拒否する** (ページ 397)

[*311] https://gihyo.jp/book/2020/978-4-297-11197-7

第 6 章

やることの明確化

6.1. 要件定義

6.1.1 111:いきなり作り始めてはいけない

何かを作ろう！　と意気込むとき、まずエディターを起動していませんか？それでは不要な機能を増やしたり、大きな手戻りが発生します。

具体的な失敗

何か Web アプリケーションを作ろうと考えたとき、すぐに使う技術や細かい仕様に注目しがちです。

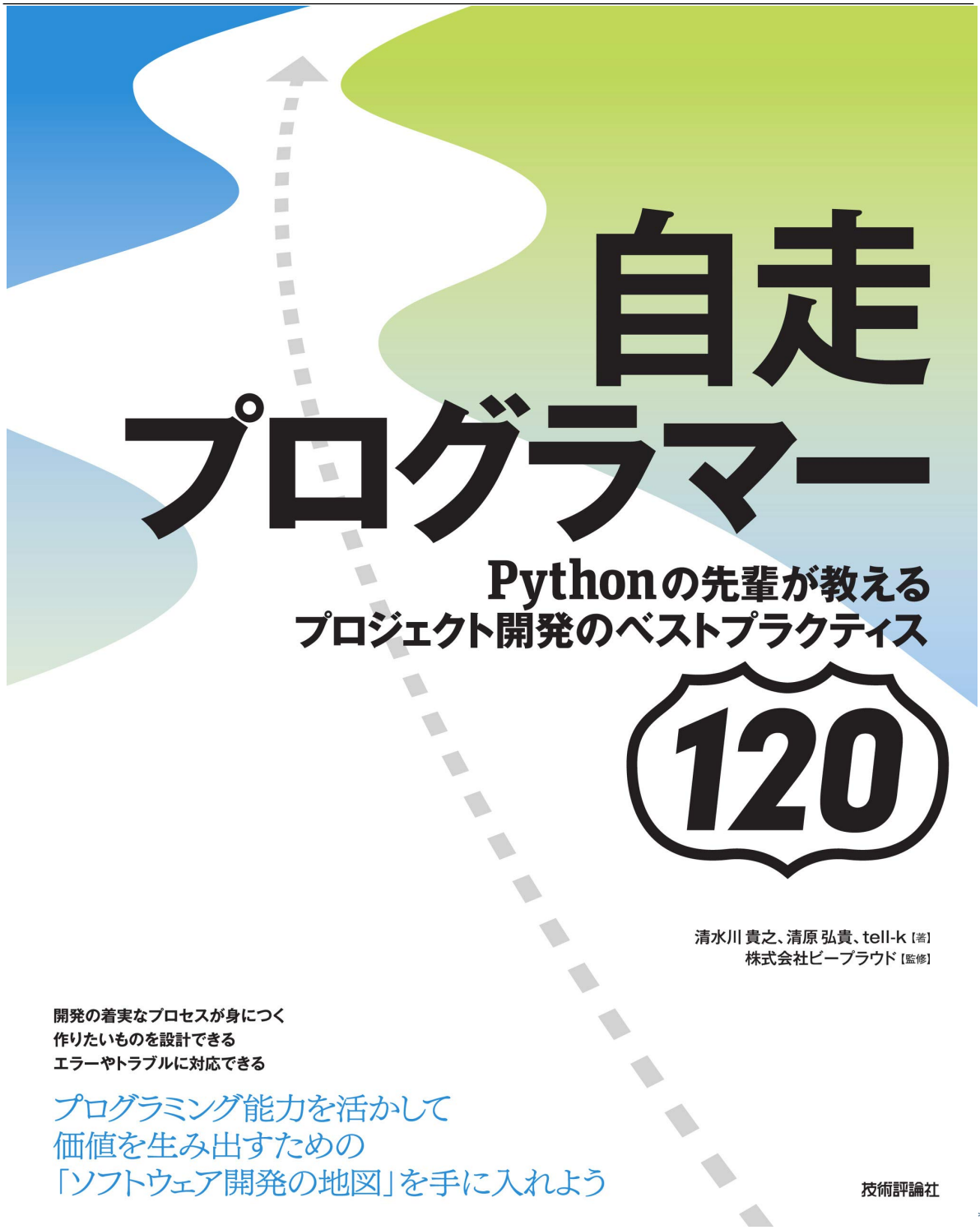
- 作りたい Web アプリケーションではソーシャルログインで Twitter、GitHub、Google、Amazon に対応させようと考えた。実装に 1 ヶ月かかったけど、サービスのメインになる機能はまだ 1 つもできていない。でも実際はリリース当初は限られた人しか使わないので、ソーシャルログインは不要だった。
- 作りたい Web アプリケーションは、スマホアプリと Web サービス両対応をしたいと考えた。でも Web アプリのメインターゲットは日中の会社員なので、最初は Web だけで十分だった。
- 商品のレコメンドやオススメを、協調フィルタリングやディープラーニングで実現しようとした。でも最初のリリース時には商品の数や種類が少ないので、サイト運営者が選んだ「月間のオススメ商品」を表示すれば十分だった。
- 同じ商品でも色、サイズ違いが選べる機能を作ろうとした。でも最初に扱う商品には色、サイズ違いがある商品はほとんどなかった。まずは色、サイズ違いがあっても数種類なので別の商品として扱えば十分だった。

サービスの構想やビジョン、生み出したい価値を練る前に、このような細かい仕様や技術にこだわりすぎていませんか？勢いでプログラムを開始して、よくわからない実験場と化したことはありませんか？それらはすべて時間の無駄になってしまいます。

ベストプラクティス

エディターを開かないことが大切です 。なぜエディターを開いてはいけないのでしょうか？　頭の中には作りたいもののイメージがあることでしょう。今すぐにでもプログラミングを始めるのが賢明なように思えます。ですがそうしてはいけません。「作りたいもののイメージは単なる幻想だから」です。

頭の中にあるイメージはとてもすばらしいものですが、多くの場合は曖昧で、触れられない、価値を検証できないものです。それを一旦書き出して、情報を整理する方法を知る必要があります。



*312

*312 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*313] をご参照ください

6.1.2 112:作りたい価値から考える

「いきなり作り始めてはいけない」と説明しましたが、では何から始めるべきなのでしょうか？「頭の中に構想はあるので、私には不要だ」と思われるかもしれませんが、意外にも人の脳みそというのは不十分なものです。

価値を考えて書き出すことで、客観的に分析する方法を説明します。

具体的な失敗

- 作ったは良いが、誰にも必要のないものだった
- 流行りのものを開発してみるが、本質的に「求められる」ものは作れずヒットしない
- 各チームメンバーは作るべきものをわかっているつもりだったけれど、それぞれの見解は別だった

こういった失敗はよくあることです。共通点は、作ったあとに間違いに気づいてしまうことです。本当に価値があるかどうかは作る前にはわかりません。ですが、作る前にも気づけた問題はあるはずです。どうすれば「作ったあとに必要ないと気づく」確率を減らせるでしょうか？

ベストプラクティス

作りたい価値から考えましょう。ここでは「価値」を、「ある人が嬉しいと感じること」とします。何かをプログラムする前に、それが誰にとって、どう嬉しいかを考えることが大切です。いきなりプログラムしたり、画面設計や要件定義をしようとすると、なぜ作るべきなのか、何を作るべきなのかを見失いがちです。

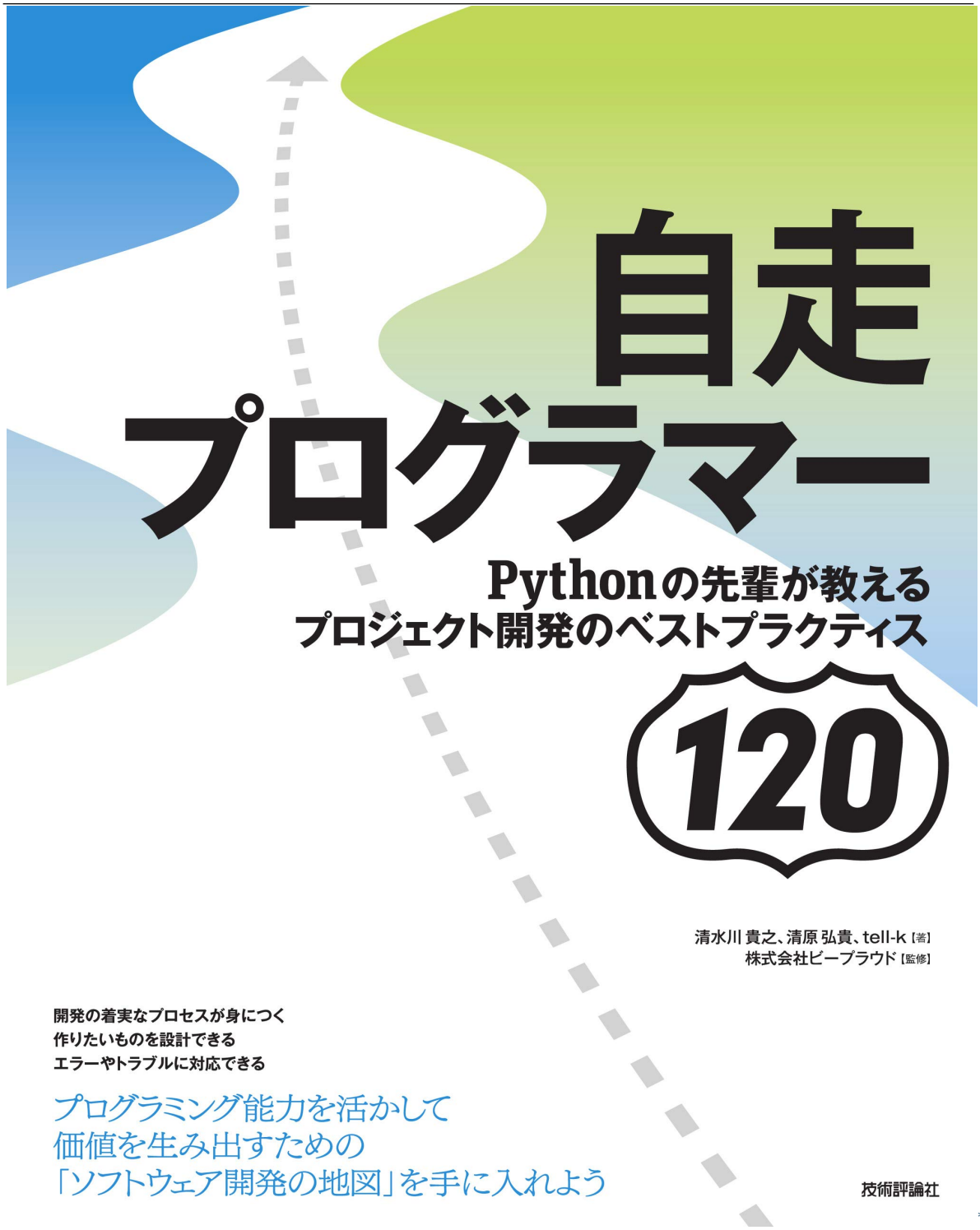
以下の「 価値問診票 」の質問に答えて、作りたい価値をまず明らかにしましょう。

質問 **1.** どんな痛みを解決するもの？

質問 **2.** 痛みの大きさや頻度は？

質問 **3.** 誰の要望、痛み？

[*313] https://gihyo.jp/book/2020/978-4-297-11197-7



*314

*314 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*315] をご参照ください

6.1.3 113:100% の要件定義を目指さない

作るものの要件を決めようとして、決めきれず仕様が曖昧になったことはありませんか？そのまま無理に書き出して、結果良いものにならないことはよくあるかと思います。

具体的な失敗

仕様を決めるとき、はじめから良い答えを求めすぎると失敗しやすいでしょう。

- 決まっていないことを書き出す勇気が出ずに、何も書けなかった
- 100% の要件定義を目指したが、実際に作ったあとは不要な機能だった

ベストプラクティス

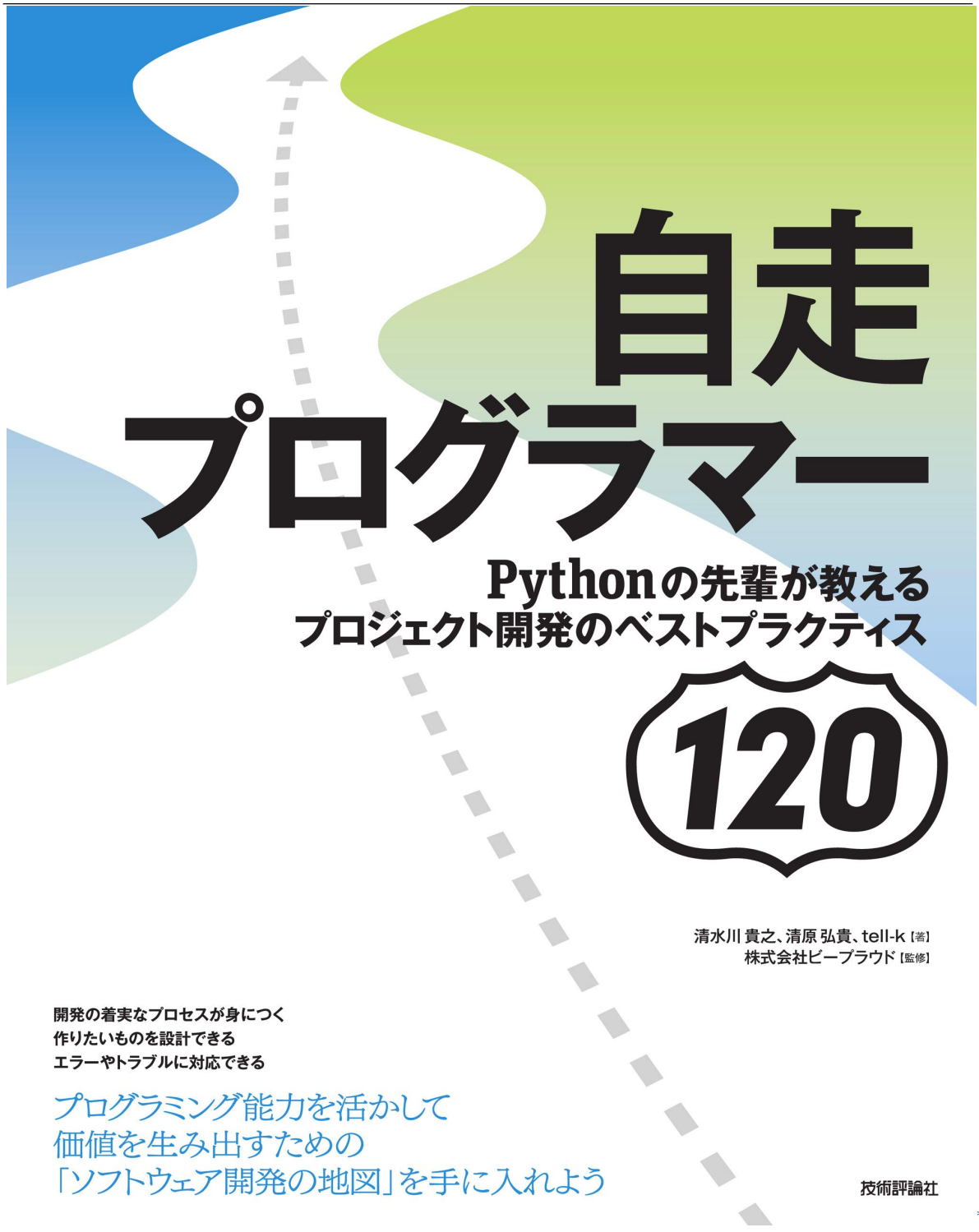
要件の確度を意識、明記しながら書きましょう。

要件定義はどのようなものを作るのか、何を作るのかを明確にするために書き出されます。最終的には（ソフトウェアなので）プログラムのソースコードがわかりやすい成果物になります。

ですがいきなり最終的な成果物を作ろうとすると迷子になってしまいます。たとえば大阪から（行ったことのない）東京に行くために、何も計画せず、考えずに車を発進させるようなものです（人間味のあるドラマは生まれるかもしれませんが）。

その最終的な成果物、価値の実現のために、より抽象的で高い視点から計画、決定していくことが大切です。

[*315] https://gihyo.jp/book/2020/978-4-297-11197-7



*316

*316 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*317] をご参照ください

[*317] https://gihyo.jp/book/2020/978-4-297-11197-7

6.2. 画面モックアップ

6.2.1 114:文字だけで伝えず、画像や画面で伝える

画面の仕様を決める（要件定義をする）ときに、「仕様を文章で書いたは良いが他の人に伝わらない」ということがよくあります。どうすればより他の人に伝えやすい要件定義ができるでしょうか。

具体的な失敗

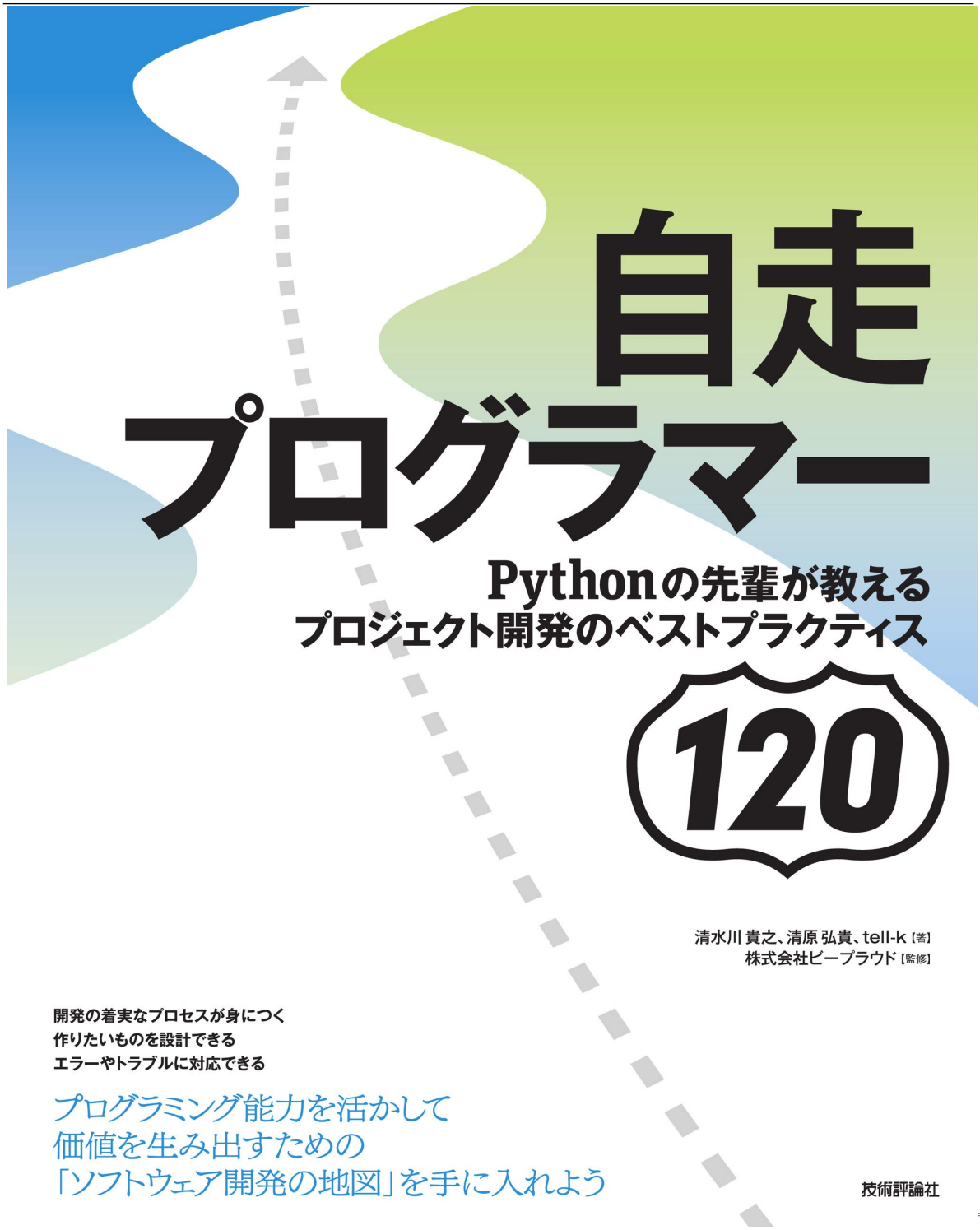
- 画面の仕様を箇条書きで書いたが認識の違いが生まれた
- チームメンバーに共有したときは OK をもらったが画面を作ったときに NG が出た
  - 共有した段階ではどんな画面になるかのイメージが伝わっていなかった
  - 文字情報だけでは目が滑ってしまって深く読んでいる人がいなかった

自分 1 人で作っているとしても、仕様が明確になっていないと作っている間に迷走してしまいます。

ベストプラクティス

文字だけで伝えず、画像で伝えるようにしましょう。

文字で伝えられる情報には限界があります。画像を見れば頭の中のイメージが活性化して、チーム内での議論も活発化します。



*318

*318 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*319] をご参照ください

以下のような白黒を基本とした絵がモックアップです。細かい文言は書かなくて良いので、各画面に必要な要素、おおまかな配置や遷移に注目して描きます。

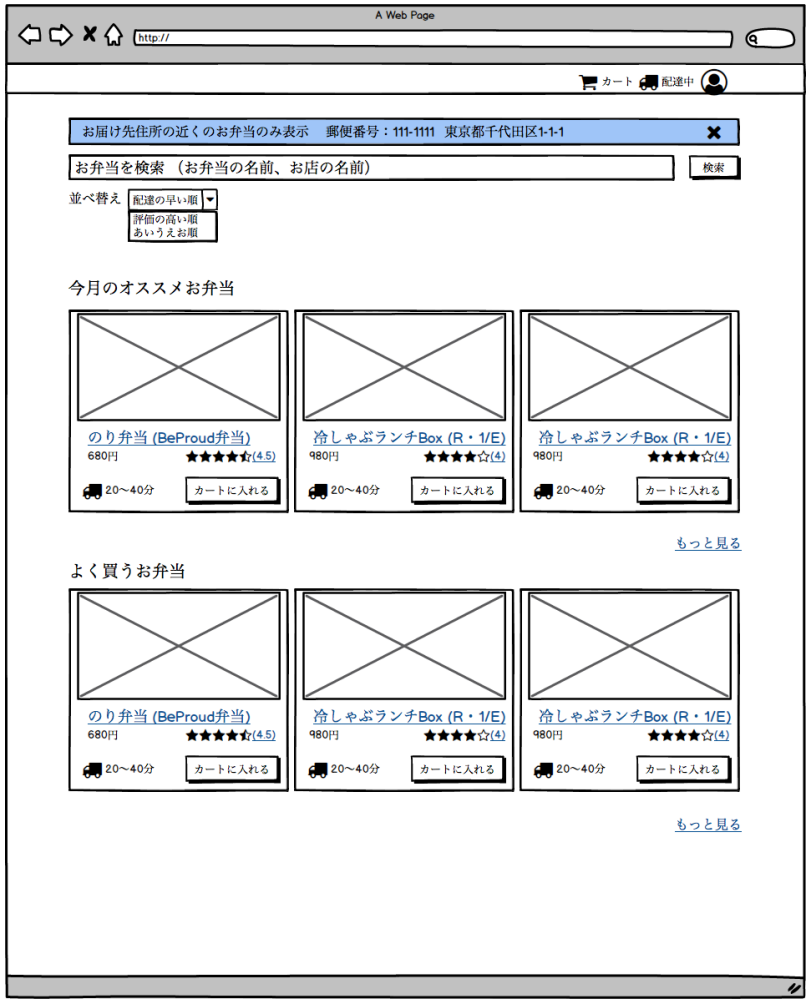
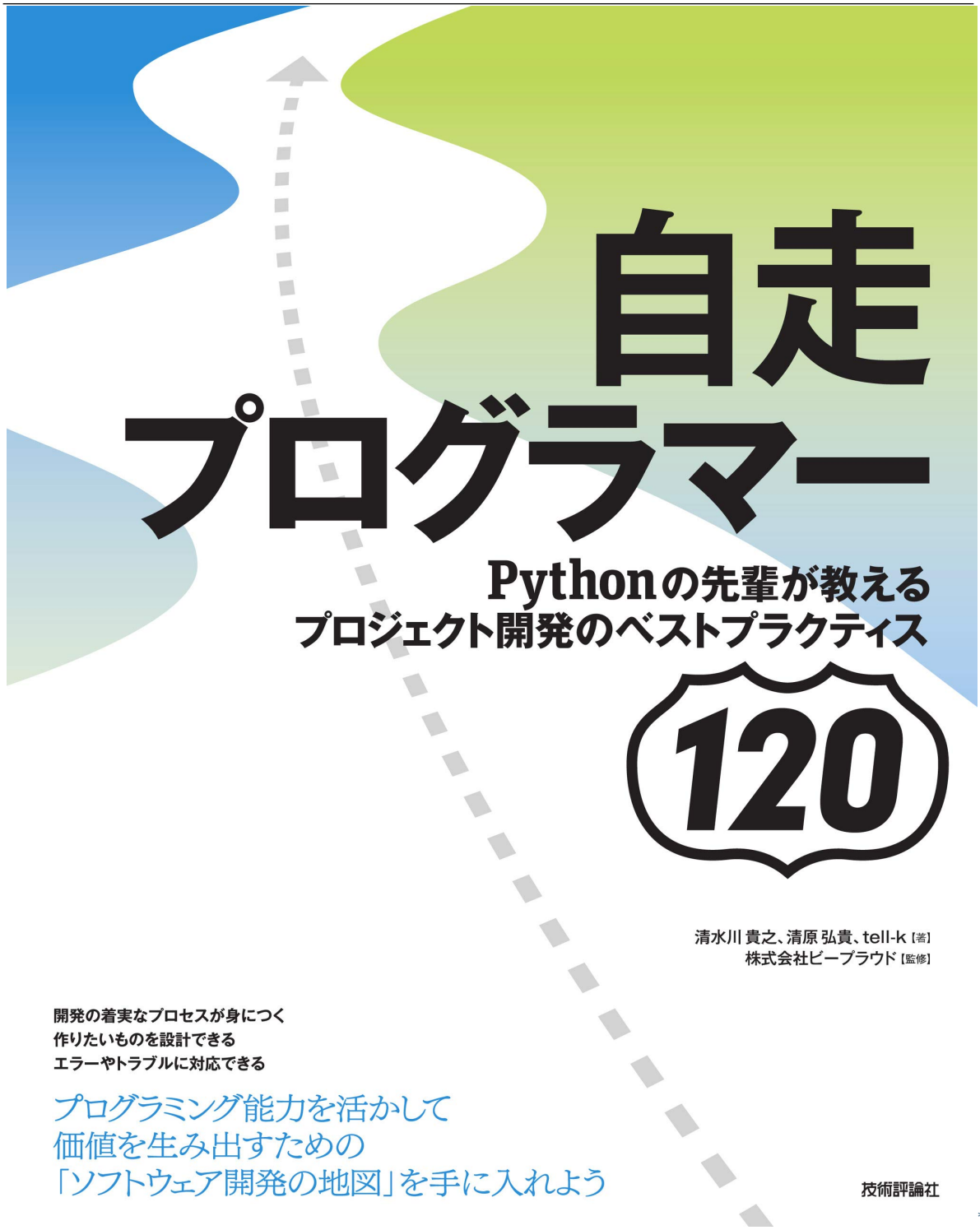


図 6.1　モックアップ：お弁当一覧画面

絵を描くには紙にペンで描いても良いですし、 BalsamiqMockup[*320] のようなツールを使っても良いです。上記の例では BalsamiqMokup を使っています。

[*319] https://gihyo.jp/book/2020/978-4-297-11197-7
[*320] https://balsamiq.com/wireframes/



*321

*321 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*322] をご参照ください

6.2.2 115:モックアップは完成させよう

画面モックアップを描いたは良いものの、やはり意味をなさないという場合は大いにあります。結局わかっていることだけしか描かれていないし、画面設計を見ても無駄だと他の人に思われないように気をつけましょう。

なぜチームの開発に役立たない画面モックアップになってしまうのでしょうか？

具体的な失敗

- 画面のモックアップを描き出してはいたが、実装の段階に入って考慮できていない点が多く見つかった
  - モデル設計にも手戻りが発生した
- モックアップを描き出したが、実装前に設計上重要な点に気づけなかった
  - 画面の表示にデータの集計が必要で、単純に実装すると動作が遅くなった

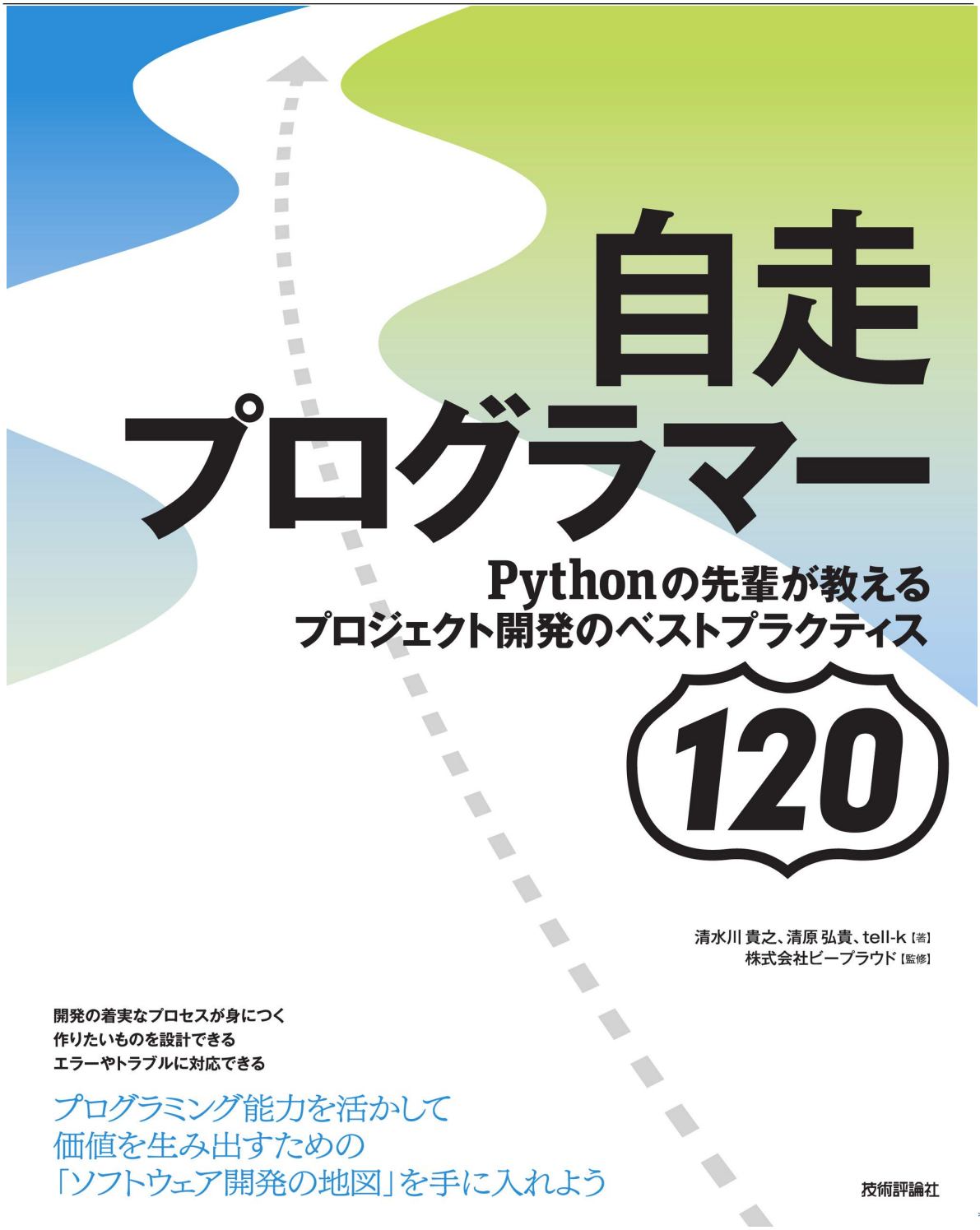
実装の段階になってモデル設計への根本的な修正があると、とても時間がかかってしまいます。あとからモデル設計を勇気をもって変更することは大切ですが、避けられる手戻りは最初から回避しましょう。

ベストプラクティス

モックアップは中途半端にせず完成させましょう。絶対的な完成は難しいので、自分が把握していることをひととおり描き出せるまでは完成させましょう。落書きやメモ程度の完成度にしてはいけません。アイデアを描き出してみるためには良いですが、その状態で「画面仕様」としてはいけません。

将来的に画面仕様が変わることは大いにありますが、 現時点で考えられる仕様は描き出しましょう 。描き出すことで、仕様の曖昧さがないようにしておきましょう。

[*322] https://gihyo.jp/book/2020/978-4-297-11197-7



*323

*323 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*324] をご参照ください

6.2.3 116:遷移、入力、表示に注目しよう

モックアップを描き出すとき、読むときに注目する点はあるでしょうか？無思慮に書いているだけでは、画面モックアップから仕様が定まらなくなります。

具体的な失敗

- よくできたモックアップは作れたが、手戻りは防げなかった
- 画面モックアップを描いているのに装飾やデザインに凝って仕様が固まらなかった

画面モックアップを描き出す理由は、「画面の仕様」を決めることではありません 。その仕様から、本当に必要なものができるか、どういったデータ設計が必要か、システム設計が必要かを読み取ることにあります。

ベストプラクティス

遷移、入力、表示に注目して画面モックアップを描きましょう。

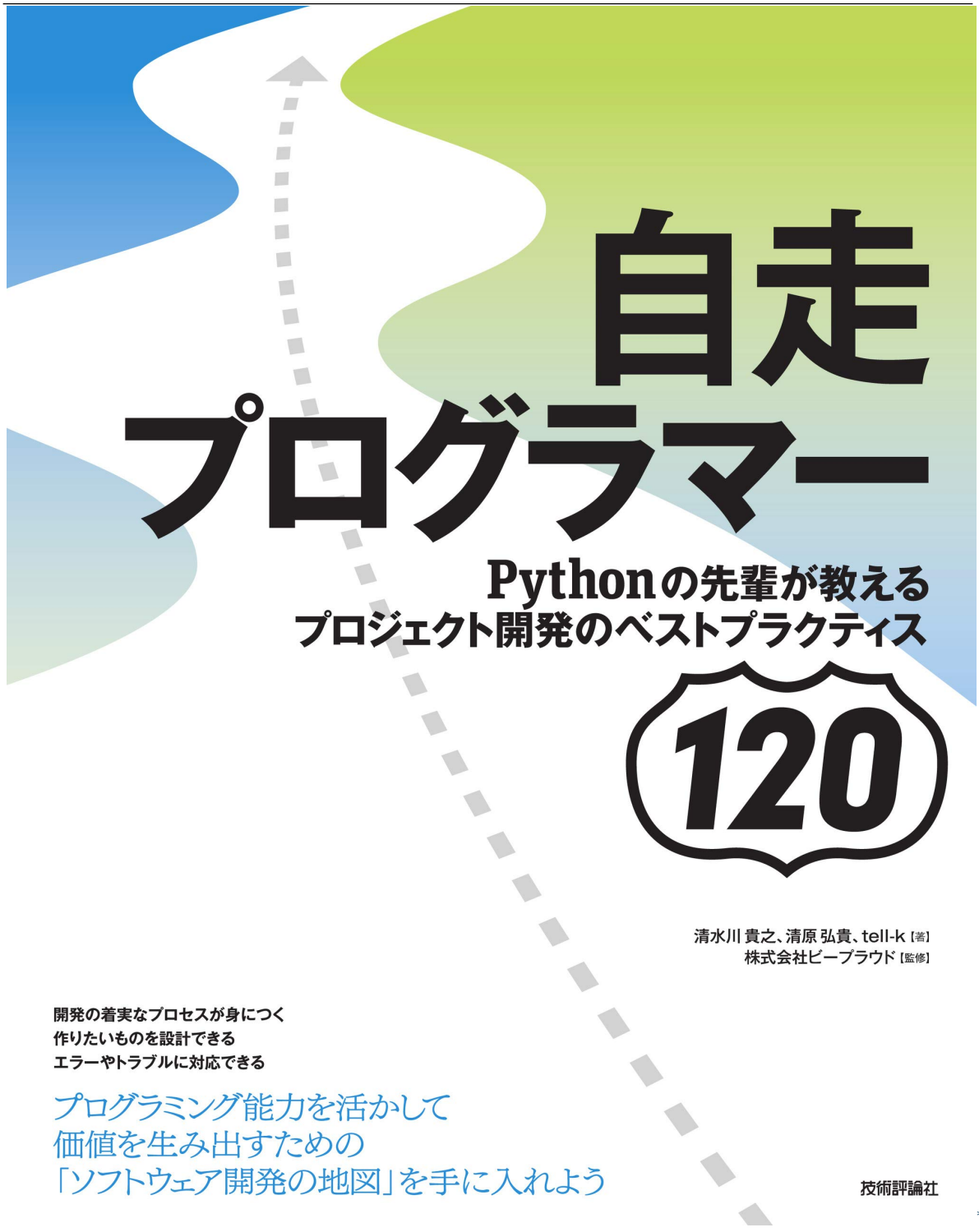
- 遷移：「この画面にはどこから来て、どこに行くのだろう」
- 入力：「この画面ではどんな入力をするのだろう」
- 表示：「この画面ではどんな情報が表示される（表示しなくて良い）のだろう」

遷移と入力、表示に注目することで、以下の仕様が明確になります。

- ユーザーのストーリー、価値に合う画面ができているか
- どんなデータ設計が必要か

画面モックアップの時点で深い洞察を持つことで、「不要なものを作る」「データ設計を失敗する」という大きな手戻りを防ぎましょう。

[*324] https://gihyo.jp/book/2020/978-4-297-11197-7



*325

*325 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*326] をご参照ください

6.2.4 117:コアになる画面から書こう

画面モックアップを書くのは大切なことですが、不用意に書きすぎることにも注意しましょう。特に、設定画面やログイン画面などの重要でない画面のモックアップまで書きすぎていませんか？

具体的な失敗

- 画面仕様をひととおり洗い出したが肝心の画面の仕様は掘り下げられていなかった
- パスワード設定画面やメールアドレス設定画面など不要な画面の仕様ばかりできてしまった

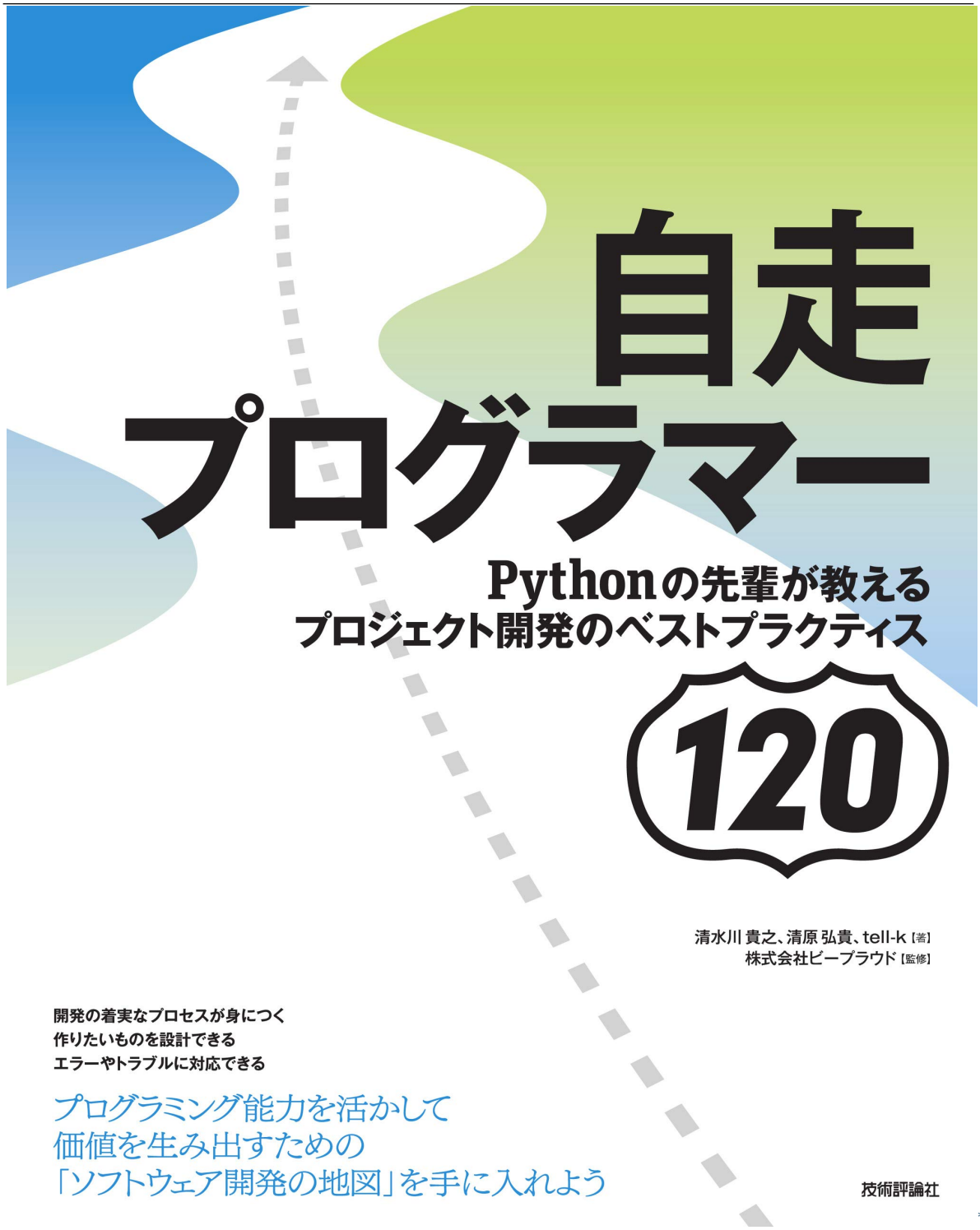
往々にして優先度が低い画面ほど簡単な仕様な画面が多いので、気軽に描き出しやすくなります。ですが優先度の低い画面は後回しにしましょう。

ベストプラクティス

「コア」になる画面から描きましょう。

その Web サービスたらしめる画面から描きましょう。おおむね作りたい Web サービス（やアプリケーション）の中で思いつく画面の順に描き出せば良いでしょう。それは、想定する使い手のストーリーに関わる画面だからです。

[*326] https://gihyo.jp/book/2020/978-4-297-11197-7



*327

*327 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*328] をご参照ください

6.2.5 118:モックアップから実装までをイメージしよう

モックアップは描き出すだけでなく、読むプロセスも大切です。特に頭を使わずに「見た目にもよさそう」と判断していませんか？モックアップを見るときに、システム構成や実装するプログラムをイメージできていますか？たとえば、モックアップを「何となく良さそう」とレビューしてしまうのは要注意です。

ベストプラクティス

モックアップから具体的な実装をイメージしましょう。

実装をイメージできないときは、具体的に仕様を確認しましょう。モックアップと仕様を確認して、できる限り実装をイメージすることで事前に実装が難しい（工数がかかる、複雑になる）箇所を把握しておくのがポイントです。

モックアップは表示する情報や使い心地を検討するだけでなく、仕様書としての意味合いがとても強いです。このような観点を持つことで、画面モックアップは「単なる画面の下書き」でなく、未来に必要な仕様や設計の青写真と捉えられます。

- どのようなデータ（テーブル、モデル）が必要か
- 各画面を表示するために、どうデータを取得する必要があるか
- キャッシュや集計する処理が必要か

EC サイトのトップページから、後々に必要になるモデル、ミドルウェアや集計処理を読み解きましょう。

[*328] https://gihyo.jp/book/2020/978-4-297-11197-7

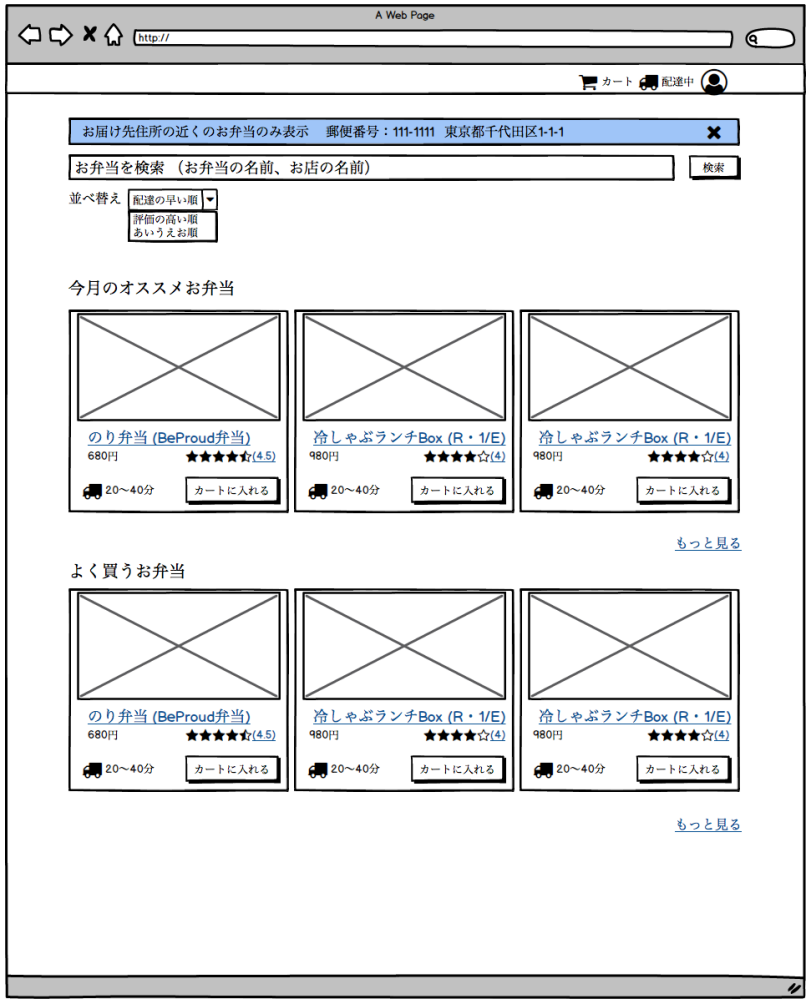
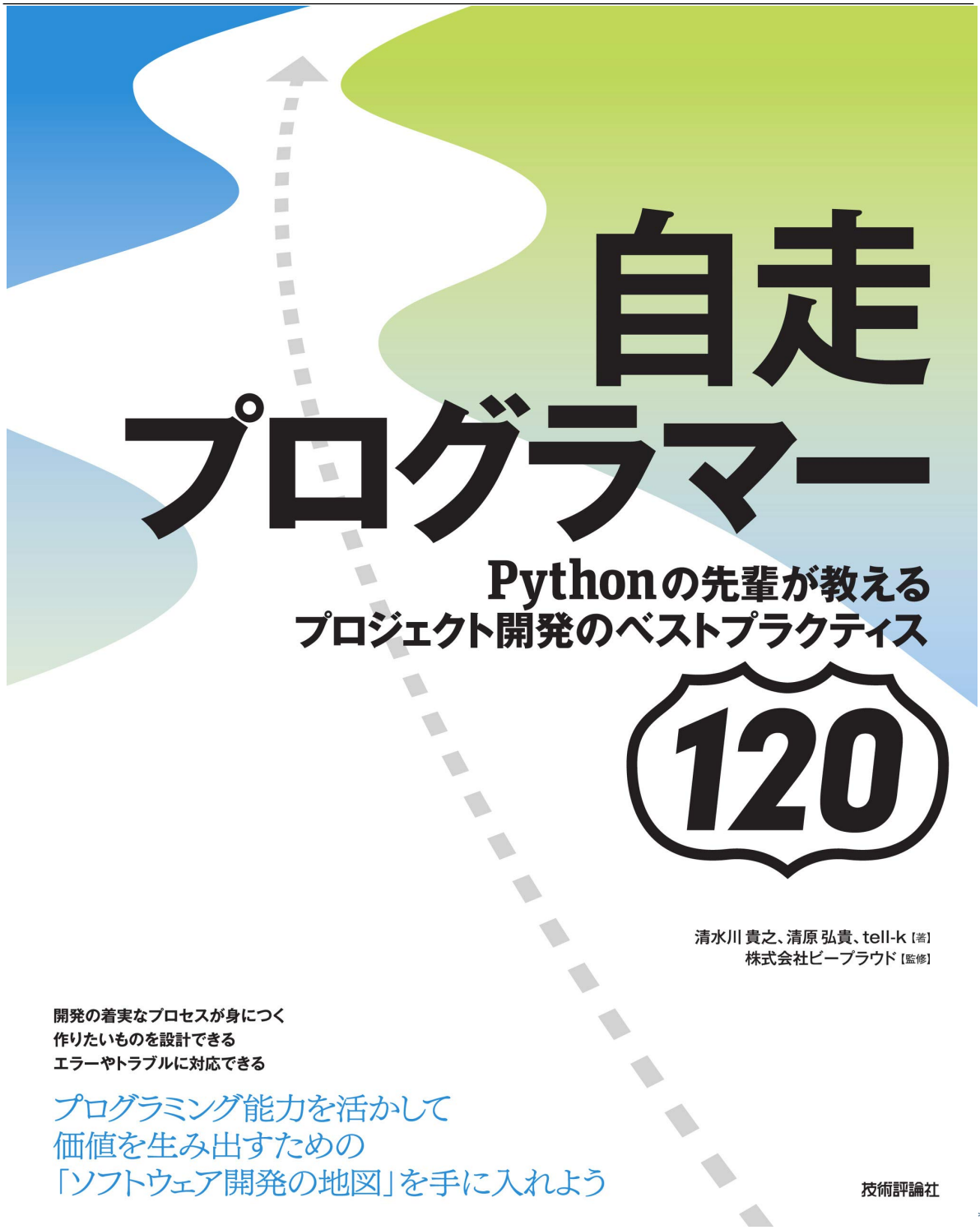


図 6.2　モックアップから実装をイメージする



*329

*329 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*330] をご参照ください

6.2.6 119:最小で実用できる部分から作ろう

何かを作るとき、得てしてリソースは限られた状況にあると思います。その状況では「どうリソースを使って」「どう完成に近づくか」、そして「どう手応えを得るか」が重要です。

次のような失敗をしたことはありませんか？

具体的な失敗

- すべてを作ろうとして、道半ばでやめてしまった
- 技術的に面白い機能から作ってしまった

何かを作るときは、まず小さく使えるものから作りましょう。リソースは限られています。リソースは無限にあると思うのであれば、むしろ良いものはできないでしょう。

ベストプラクティス

最小限の実装で、実際に使えて役に立つ部分から作り始めましょう。すべてを一度に作ろうとせず、最小十分のプログラムを作って、使いながら価値検証をしましょう。

何かを作るうえで、コストと締切りは無視できません。仕事でプログラムする際にはもちろんコストと締切りは存在しますが、仮に自分 1 人で趣味の Web サービスを作る場合にも存在します。

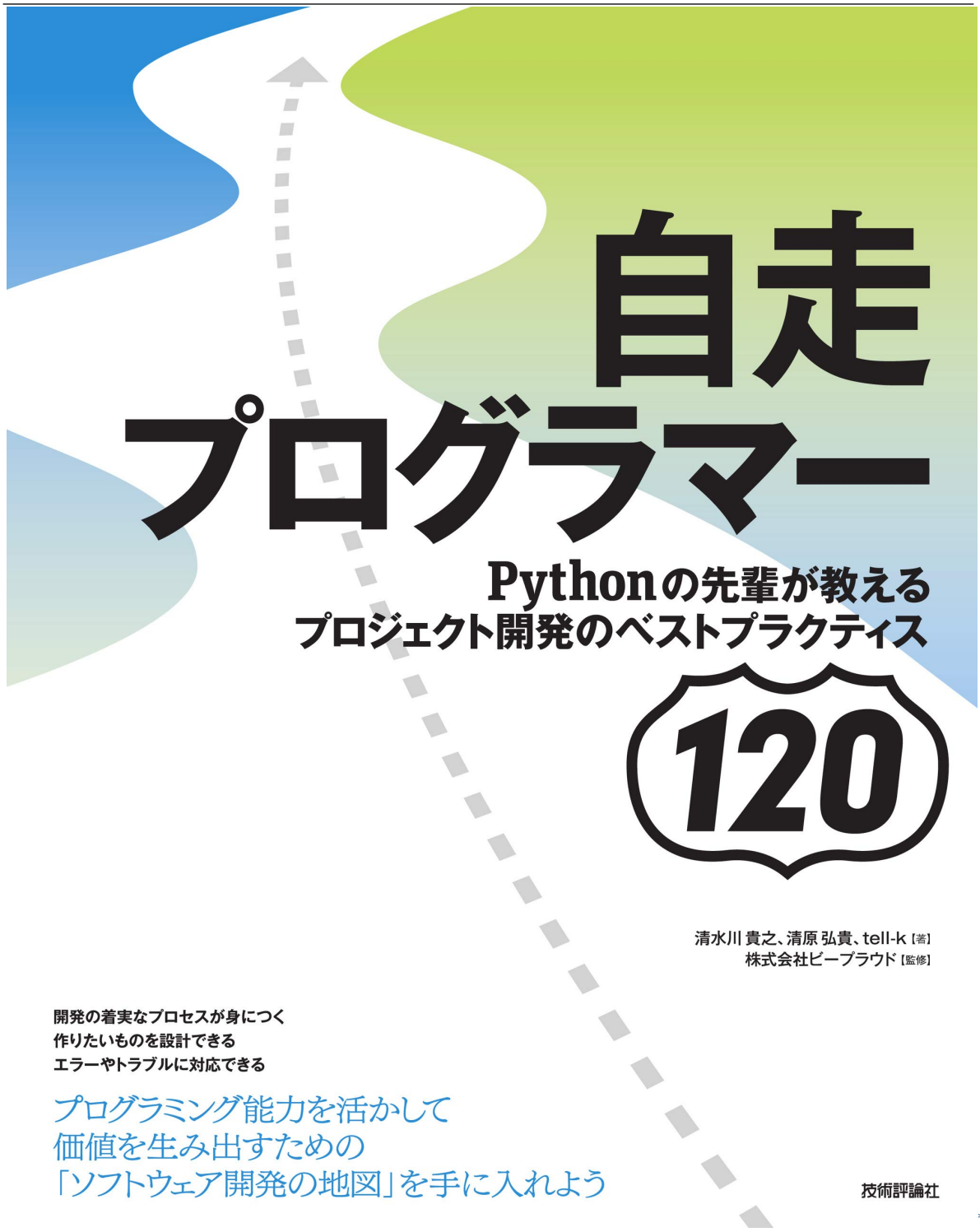
- コスト
  - 自分自身の時間
  - 労力、体力
  - 作り続けるモチベーション
- 締切り
  - 飽きてやめてしまう
  - 似たサービスがローンチされてしまう

[*330] https://gihyo.jp/book/2020/978-4-297-11197-7

締切りというと嫌な印象がありますが、モチベーションを保つうえでとても大切です。途方もなく大きなものを闇雲に作るより、マイルストーンを立てて順に小さく作っていくほうがモチベーションを保てます。

無限にリソースがあると思う場合でも、モチベーションという資源は有限です。どんな場合も、まずは小さく作ることが一番大切なプロセスです。

最小の完成形を見つける



*331

*331 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*332] をご参照ください

6.2.7 120:ストーリーが満たせるかレビューしよう

モックアップが書き終わったら、それでおしまいにしてはいけません。初めに書き出した「価値問診票」に立ち戻って考えましょう。

具体的な失敗

- 良いモックアップができたが、想定する顧客にとって使いやすいものではなかった
- モックアップを書いているうちにブレてしまっていた

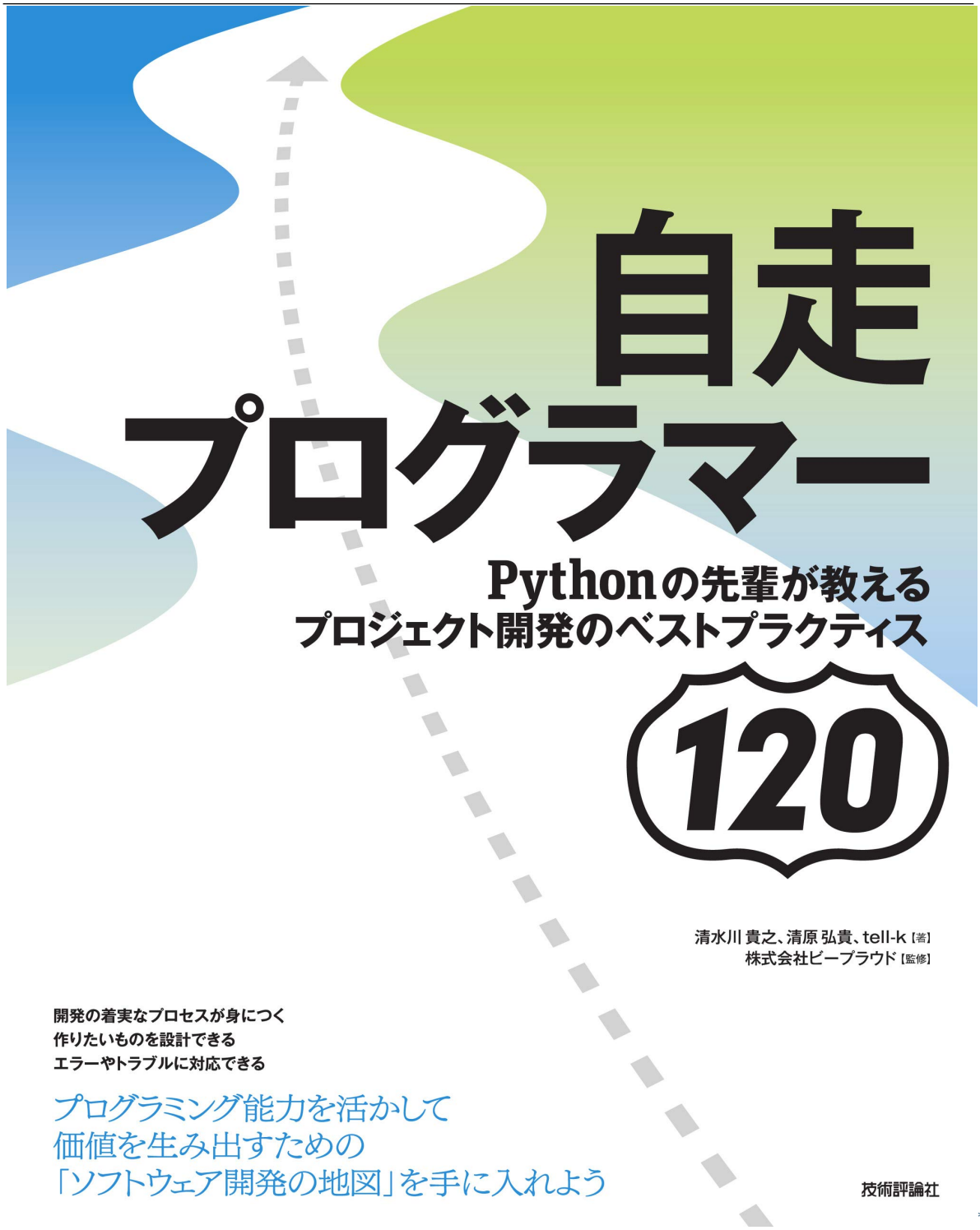
価値検証の段階に戻って考えること、見返すことが大切です。

ベストプラクティス

モックアップがストーリーと価値を満たせるかをレビューしましょう。

初期のリリースする機能のモックアップができれば以下を考え直しましょう。モックアップを作ったところで、もともと必要だった価値を満たすものでなければ意味がありません。

[*332] https://gihyo.jp/book/2020/978-4-297-11197-7



*333

*333 https://gihyo.jp/book/2020/978-4-297-11197-7

（中略）詳細は書籍 自走プログラマー[*334] をご参照ください

[*334] https://gihyo.jp/book/2020/978-4-297-11197-7

第 7 章

参考文献

7.1. 参考書籍

- 『IT エンジニアが覚えておきたい英語動詞 30』（板垣政樹著、秀和システム刊、2016 年 3 月）
- 『Python プロフェッショナルプログラミング第 3 版』（ビープラウド著、秀和システム刊、2018 年 6 月）
- 『SQL アンチパターン』（Bill Karwin 著、オライリージャパン刊、2013 年）
- 『Web エンジニアが知っておきたいインフラの基本』（馬場 俊彰著、マイナビ刊、2014 年 12 月）
- 『xUnit Test Patterns』（Gerard Meszaros 著、Addison-Wesley Professional 刊、2007 年 5 月）
- 『エキスパート Python プログラミング改訂 2 版』（Michal Jaworski、Tarek Ziade 著、アスキードワンゴ刊、2018 年 2 月）
- 『図解でなっとく! トラブル知らずのシステム設計 エラー制御・排他制御編』（野村総合研究所、エアーダイブ著、日経 BP 社 刊、2018 年 3 月）
- 『文芸的プログラミング』（ドナルド・E. クヌース著、ASCII 刊、1994 年）
- 『楽々 ERD レッスン』（羽生章洋 著、翔泳社 刊、2006 年 4 月）
- 『管理ゼロで成果はあがる～「見直す・なくす・やめる」で組織を変えよう』（倉貫義人著、技術評論社刊、2019 年 1 月）
- 『達人に学ぶ DB 設計』（ミック 著、翔泳社 刊、2012 年 3 月）

7.2. 参考サイト

- Arrange Act Assert http://wiki.c2.com/?ArrangeActAssert
- Fragile Test at XUnitPatterns.com http://xunitpatterns.com/Fragile%20Test.html
- Marketing For Developers https://devmarketing.xyz/
- Pull Request https://help.github.com/ja/github/collaborating-with-issues-and-pull-requests/about-pull-requests
- slug https://developer.mozilla.org/ja/docs/Glossary/スラグ
- The Twelve-Factor App （日本語訳） https://12factor.net/ja/config
- 「巨大プルリク 1 件 vs 細かいプルリク 100 件」問題を考える（翻訳） https://techracho.bpsinc.jp/hachi8833/2018_02_07/51095
- エンジニアの「プロの所作」01. まず自分で調べる :「自分主体で考えて作る」第 1 歩。わからないことを調べる所作を伝えます - Python 学習チャンネル by PyQ https://blog.pyq.jp/entry/professionalism_of_engineer_01
- ストーリーとしての競争戦略 https://store.toyokeizai.net/books/9784492532706/
- セマンティック バージョニング 2.0.0 | Semantic Versioning https://semver.org/lang/ja/
- セルフマネジメントの必須スキル「タスクばらし」そのポイント | Social Change! https://kuranuki.sonicgarden.jp/2016/07/task-break.html
- ソフトウェア開発時にどのような基準で OSS ライブラリを選定するのがよいのか https://yoshinorin.net/2019/08/31/how-to-choose-oss-library/
- リーン顧客開発 https://www.oreilly.co.jp/books/9784873117218/
- ローカルなプロセス間通信用のソケット - UNIX https://linuxjm.osdn.jp/html/LDP_man-pages/man7/unix.7.html
- 匠メソッド http://www.takumi-method.biz/
- 安全なウェブサイトの作り方 https://www.ipa.go.jp/security/vuln/websecurity.html
- 安全なウェブサイトの運用管理に向けての 20 ヶ条 〜セキュリティ対策のチェックポイント〜 https://www.ipa.go.jp/security/vuln/websitecheck.html

- 第１回ＣＤＮの仕組み（CDN はどんな技術で何ができるのか）https://blog.redbox.ne.jp/what-is-cdn.html
- 若手開発者の後悔 https://postd.cc/the-sorrows-of-young-developer/

7.3. Python ライブラリ

Python 公式

- enumerate：Python 3 ドキュメント https://docs.python.org/ja/3/library/functions.html#enumerate
- logging - Python 用ロギング機能 - Python 3.8.1 ドキュメント https://docs.python.org/ja/3/library/logging.html#logging.Formatter
- Logging Flow - Logging HOWTO - Python 3.8.1 ドキュメント https://docs.python.org/ja/3/howto/logging.html#logging-flow
- mock https://docs.python.org/ja/3/library/unittest.mock.html
- tempfile https://docs.python.org/ja/3/library/tempfile.html
- TypedDict 仕様提案：PEP-589 https://www.python.org/dev/peps/pep-0589/
- TypedDict：Python 3 ドキュメント https://docs.python.org/ja/3/library/typing.html#typing.TypedDict
- ソケットプログラミング HOWTO - Python 3.8.1 ドキュメント https://docs.python.org/ja/3/howto/sockets.html

Django 公式

- clearsession https://docs.djangoproject.com/ja/2.2/ref/django-admin/#clearsessions
- DEP 0008 https://github.com/django/deps/blob/master/accepted/0008-black.rst
- Django のコーディングスタイル https://docs.djangoproject.com/ja/2.2/internals/contributing/writing-code/coding-style/
- Django の設定 https://docs.djangoproject.com/ja/2.2/topics/settings/
- QuerySet API reference https://docs.djangoproject.com/ja/2.2/ref/models/querysets/
- セッションの使いかた https://docs.djangoproject.com/ja/2.2/topics/http/sessions/#using-file-based-sessions
- テストツール https://docs.djangoproject.com/ja/2.2/topics/testing/tools/
- 複数の値を持つリレーションの横断 https://docs.djangoproject.com/ja/2.2/topics/db/queries/#spanning-multi-valued-relationships
- ALLOWED_HOSTS https://docs.djangoproject.com/ja/2.2/ref/settings/#allowed-hosts

- SESSION_ENGINE https://docs.djangoproject.com/ja/2.2/ref/settings/#std:setting-SESSION_ENGINE

サードパーティーライブラリのドキュメント

- Django Debug Toolbar の設定 https://django-debug-toolbar.readthedocs.io/en/latest/installation.html
- Gunicorn の bind http://docs.gunicorn.org/en/stable/settings.html#bind
- Gunicorn のデプロイ https://docs.gunicorn.org/en/stable/deploy.html
- TypedDict：mypy 公式ドキュメント https://mypy.readthedocs.io/en/latest/more_types.html#typeddict

パッケージ

- aldjemy https://pypi.org/project/aldjemy/
- APScheduler https://pypi.org/project/APScheduler/
- autopep8 https://pypi.org/project/autopep8/
- awesome-python https://github.com/vinta/awesome-python
- black https://pypi.org/project/black/
- Celery http://www.celeryproject.org/
- deform https://docs.pylonsproject.org/projects/deform/en/latest/
- Django Packages https://djangopackages.org/
- django-background-tasks https://django-background-tasks.readthedocs.io/
- django-debug-toolbar https://pypi.org/p/django-debug-toolbar
- django-environ https://django-environ.readthedocs.io/
- django-redis https://niwinz.github.io/django-redis/latest/
- django-silk https://pypi.org/p/django-silk/
- factory-boy https://factoryboy.readthedocs.io/en/latest/
- fakeredis https://pypi.org/project/fakeredis/
- flake8 https://pypi.org/project/flake8/
- flake8-logging-format https://pypi.org/project/flake8-logging-format/
- Gunicorn https://pypi.org/project/gunicorn/

- moto http://docs.getmoto.org/en/latest/
- nplusone https://pypi.org/p/nplusone
- Pipenv https://pipenv.kennethreitz.org/
- Poetry https://python-poetry.org/
- pycodestyle https://pypi.org/project/pycodestyle/
- pylint https://pypi.org/project/pylint/
- python-decouple https://pypi.org/p/python-decouple/
- responses https://github.com/getsentry/responses
- SQLAlchemy https://pypi.org/project/SQLAlchemy/
- uWSGI https://pypi.org/project/uWSGI/
- virtualenvwrapper https://virtualenvwrapper.readthedocs.io/
- WTForm https://wtforms.readthedocs.io/en/stable/

7.4. ミドルウェア

- Anaconda https://www.anaconda.com/
- BEGIN - PostgreSQL https://www.postgresql.jp/document/11/html/sql-begin.html
- COMPOSE_FILE - Docker https://docs.docker.com/compose/reference/envvars/#compose_file
- Docker https://www.docker.com/
- Docker 公式の Python https://hub.docker.com/_/python
- Intel Python https://software.intel.com/en-us/distribution-for-python
- Memcached https://memcached.org/
- proxy_cache_path 設定 - Nginx http://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_cache_path
- pyenv https://github.com/pyenv/pyenv
- Redis https://redis.io/
- Vagrant https://www.vagrantup.com/
- トランザクションの管理 - Oracle http://otndnld.oracle.co.jp/document/products/oracle11g/111/doc_dvd/server.111/E05765-03/transact.htm
- 名前ベースのバーチャルホスト - Apache https://httpd.apache.org/docs/2.4/ja/vhosts/name-based.html
- 名前ベースのバーチャルホスト - Nginx http://nginx.org/en/docs/http/request_processing.html
- 暗黙的なコミットを発生させるステートメント - MySQL https://dev.mysql.com/doc/refman/5.6/ja/implicit-commit.html

7.5. サービス

- AWS CloudFront https://aws.amazon.com/jp/cloudfront/
- Akamai https://www.akamai.com/jp/ja/
- Fastly https://www.fastly.jp/
- GCP Cloud CDN https://cloud.google.com/cdn/
- Sentry https://sentry.io/
- プログラマーのためのネーミング辞書 codic https://codic.jp

7.6. デスクトップツール

- BalsamiqMockup https://balsamiq.com/wireframes/
- Dash https://kapeli.com/dash
- Zeal https://zealdocs.org/

7.7. 標準仕様

- Forwarded - MDN https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Forwarded
- RFC 7239 - Forwarded HTTP Extension https://tools.ietf.org/html/rfc7239
- X-Forwarded-For - MDN https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/X-Forwarded-For
- X-Forwarded-Host - MDN https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/X-Forwarded-Host
- X-Forwarded-Proto - MDN https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/X-Forwarded-Proto

第 8 章

著者紹介

8.1. 清水川 貴之

清水川 貴之（しみずかわ たかゆき）

2003 年から Python を主言語として使い始め、Web アプリケーションの開発を中心に活用してきた。現職のビープラウド[*335] では開発の他、Python 関連書籍の執筆や研修講師も行っている。個人では、一般社団法人 PyCon JP[*336] の理事として日本各地で開催されている Python Boot Camp[*337] で Python 講師を務めている。Python mini Hack-a-thon[*338] など Python 関連イベント運営のかたわら、国内外のカンファレンスへ登壇し Python 技術情報を発信するなど、公私ともに Python とその関連技術の普及活動を行っている。

Twitter　@shimizukawa[*339]

URL　http://清水川.jp/

Amazon **著者セントラル**　清水川貴之[*340]

8.1.1 共著書／共訳書

- Python プロフェッショナルプログラミング 第 3 版 （2018 秀和システム刊）
- エキスパート Python プログラミング改訂 2 版 （2018 アスキードワンゴ刊）
- 独学プログラマー （2018 日経 BP 社刊）
- Sphinx をはじめよう第 2 版 （2017 オライリー・ジャパン刊）
- Python プロフェッショナルプログラミング 第 2 版 （2015 秀和システム刊）
- Sphinx をはじめよう （2013 オライリー・ジャパン刊）
- Python プロフェッショナルプログラミング 第 1 版 （2012 秀和システム刊）
- エキスパート Python プログラミング 1 版 （2010 アスキー・メディアワークス）

[*335] https://www.beproud.jp/
[*336] http://www.pycon.jp/
[*337] https://peraichi.com/landing_pages/view/pycamp
[*338] https://pyhack.connpass.com/
[*339] https://twitter.com/shimizukawa
[*340] https://www.amazon.co.jp/%E6%B8%85%E6%B0%B4%E5%B7%9D%E8%B2%B4%E4%B9%8B/e/B0749GZGZW/

8.1.2 執筆したトピック

8.2. 清原 弘貴

清原 弘貴（きよはら ひろき）

2012 年 10 月より BeProud[*341] 所属。2011 年から本格的に Python を使っている。Django[*342] が好きで、日本で最大級の Django イベント DjangoCongress JP（ https://djangocongress.jp ）の主催をしたり、Web アプリケーションやライブラリを作ったり、Django 本体のソースコードへパッチを送ったりしている。個人で Shodo（ https://shodo.ink ）、dig-en（ https://dig-en.com ）、PileMd（ https://pilemd.com ）、仕事で PyQ（ https://pyq.jp ）など、多数の Web サービス・アプリを企画、開発している。

Twitter @hirokiky[*343]

URL http://hirokiky.org/

Amazon **著者セントラル** 清原弘貴[*344]

8.2.1 共著書

- Python プロフェッショナルプログラミング 第 3 版 （2018 秀和システム刊）
- Python エンジニアファーストブック （2017 技術評論社刊）
- Python プロフェッショナルプログラミング 第 2 版 （2015 秀和システム刊）

8.2.2 執筆したトピック

- *1:***関数名は処理内容を想像できる名前にする** (ページ 10)
- *2:***関数名ではより具体的な意味の英単語を使おう** (ページ 14)
- *3:***関数名から想像できる型の戻り値を返す** (ページ 17)
- *4:***副作用のない関数にまとめる** (ページ 21)
- *5:***意味づけできるまとまりで関数化する** (ページ 24)
- *6:***リストや辞書をデフォルト引数にしない** (ページ 29)

[*341] https://www.beproud.jp/
[*342] https://www.djangoproject.com/
[*343] https://twitter.com/hirokiky
[*344] https://www.amazon.co.jp/%E6%B8%85%E5%8E%9F-%E5%BC%98%E8%B2%B4/e/B00WCKS7X8/

8.3. tell-k

tell-k（てるけー）

2005 年から PHP ／ Perl を利用した Web アプリケーション開発の仕事に従事し、2011 年から本格的に仕事で Python を使い始めた。最近はもっぱらお猫様のお世話に忙しい。

Twitter　@tell_k[*345]

GitHub　https://github.com/tell-k

Amazon **著者セントラル**　tell-k[*346]

8.3.1 共著書

- Python プロフェッショナルプログラミング 第 3 版（2018 秀和システム刊）
- Python プロフェッショナルプログラミング 第 2 版（2015 秀和システム刊）
- Python プロフェッショナルプログラミング 第 1 版（2012 秀和システム刊）

8.3.2 執筆したトピック

- *21:* **テストケースは準備、実行、検証に分割しよう** (ページ 77)
- *24:* **テスト用のデータはテスト後に削除しよう** (ページ 92)
- *26:* **テストケース毎にテストデータを用意する** (ページ 99)
- *27:* **必要十分なテストデータを用意する** (ページ 102)
- *36:* **実装方針を相談しよう** (ページ 135)
- *38:* **必要十分なコードにする** (ページ 139)
- *46:* **マスターデータとトランザクションデータを分けよう** (ページ 170)
- *47:* **トランザクションデータは正確に記録しよう** (ページ 173)
- *48:* **クエリで使いやすいテーブル設計をする** (ページ 176)

[*345] https://twitter.com/tell_k
[*346] https://www.amazon.co.jp/tell-k/e/B084KL4QX9/

第 9 章

著者・関係者による紹介 blog

9.1. hirokiky

著者 **清原 弘貴** (ページ 450)

10 年以上のノウハウを詰め込んだ「自走プログラマー」を執筆しました - Make 組ブログ[*347]　この本は著者陣が仕事の中で他の人に教えたことを中心に執筆しています。 たとえばコードレビューや、社内でのサポート、社内外の研修、コンサルティングの中で伝えてきたことをまとめています。 ビープラウド社内のメンバーも積極的にレビュー、コメントしてくれて、社内のかなりのノウハウが取り込まれています。執筆に際しては社内で半年から 1 年をかけてネタを集めて、選別する時間を設けました。 著者の 3 人が日々の業務で伝えたことや感じたこと、過去に社内で伝えたことをネタ帳として集めてから執筆しています。 たとえば「ログには 5W1H を書こう」という僕が執筆を担当したプラクティスがあるのですが、これは 5 年前（2015 年）から社内では共有していた知識です。これは社内でも好評で、「ロギングって大事だけど、何を書くべきかを学べる場所がなかった」とフィードバックを受けていました。 そのノウハウがやっと日の目を見ることになって、僕個人としてはとても嬉しいです。こんなにうれしいことはない。。

[*347] https://blog.hirokiky.org/entry/2020/02/20/105341

9.2. haru

株式会社ビープラウド代表取締役 佐藤治夫[*348]

「自走プログラマー」は中級以上の Python プログラマーになりたい人のための豊富なレシピ集 - ビープラウド社

「理由を考えるための設計・実装の選択肢」が、前書きに書かれている「プログラミング入門者が中級者にランクアップするのに必要な知識」の本質中級以上のプログラマーが、日々どのようなことにこだわり、思考を積み上げているかを知ることもできるでしょう（中にはそこまで考える必要あるの？というものもあるでしょう）

[*348] https://shacho.beproud.jp/
[*349] https://shacho.beproud.jp/entry/self-propelled-programmer

索引