

# Python コードチェック・テスト超入門: CI/CD のための pylint と pytest

---

このガイドは、Python 初心者でも pylint と pytest を『最短』で使い始め、CI/CD に統合できるよう作成した学習資料です。コードの品質管理とテストの基礎から、自動化パイプラインへの組み込みまで学んでいきましょう。

## 目次

1. [pylint と pytest とは？](#)
  2. [環境のセットアップ](#)
  3. [pylint の基本概念](#)
  4. [基本的な pylint コマンド](#)
  5. [pylint 設定ファイルの作成](#)
  6. [pylint を CI/CD に組み込む](#)
  7. [pytest の基本概念](#)
  8. [基本的なテストの書き方](#)
  9. [テストフィクスチャの活用](#)
  10. [モックとパッチング](#)
  11. [pytest のプラグイン活用](#)
  12. [コードカバレッジの測定](#)
  13. [pytest を CI/CD に組み込む](#)
  14. [GitHub Actions による自動化](#)
  15. [ベストプラクティス](#)
  16. [よくある質問と回答](#)
  17. [次のステップ](#)
- 

## 1. pylint と pytest とは？

pylint と pytest は Python の開発において重要なツールで、コードの品質とテストをサポートします。

pylint とは

pylint は Python のコード解析ツールで、以下のことを助けてくれます:

- **コードの品質チェック:** PEP 8 スタイルガイド準拠の確認
- **エラー検出:** 潜在的なバグやエラーを見つける
- **コードの改善提案:** より良いコーディングパターンを提案
- **コード複雑度の分析:** 複雑すぎるコードを特定

pytest とは

pytest は Python のテストフレームワークで、以下の特徴があります:

- **シンプルな構文:** 直感的かつ読みやすいテスト記述
- **自動検出:** テストファイル・関数を自動で見つけて実行

- **豊富なフィクスチャ**: テスト環境のセットアップ・クリーンアップを簡単に
- **プラグイン機能**: 機能拡張が容易

なぜ pylint と pytest を使うべきか

- **コード品質の向上**: 一貫性があり、読みやすいコードになる
- **バグの早期発見**: テストによって問題を早い段階で見つけられる
- **リファクタリングの安全性**: テストがあれば安心して改善できる
- **CI/CD との連携**: 自動テスト・検証によるデプロイの安全性向上
- **チーム開発の効率化**: コードスタイルの統一とテストによる品質担保

## 2. 環境のセットアップ

pylint と pytest を使い始めるための環境セットアップを行いましょう。

仮想環境の作成（推奨）

まずは仮想環境を作成します。これにより、プロジェクト固有の依存関係を管理できます。

```
# venv による仮想環境の作成
python -m venv myenv

# 仮想環境のアクティベート (Windows)
myenv\Scripts\activate

# 仮想環境のアクティベート (macOS / Linux)
source myenv/bin/activate
```

pylint のインストール

```
pip install pylint
```

pytest のインストール

```
pip install pytest pytest-cov
```

インストールの確認

インストールが正常に完了したか確認しましょう。

```
# pylint のバージョン確認
pylint --version
```

```
# pytest のバージョン確認  
pytest --version
```

## requirements.txt の作成

プロジェクトの依存関係を記録しておくと、他の環境でも同じセットアップが容易になります。

```
pip freeze > requirements.txt
```

### requirements.txt の例:

```
pytest==7.3.1  
pytest-cov==4.1.0  
pylint==2.17.4
```

💡 **初心者向けヒント** : 常に仮想環境を使うことで、プロジェクト間の依存関係の競合を避けられます。また、[requirements.txt](#) を使うことで、チームメンバーや CI/CD サービスでも同じ環境を再現できます。

## 3. pylint の基本概念

pylint を効果的に使うために、基本的な概念と機能を理解しましょう。

### pylint のレポート形式

pylint は実行すると、次のようなレポートを生成します:

```
***** Module example  
C0111: Missing module docstring (missing-docstring)  
C0103: Variable name "a" doesn't conform to snake_case naming style (invalid-name)  
W0612: Unused variable 'a' (unused-variable)
```

### 各メッセージの意味:

- 最初の文字 (C/R/W/E/F) はメッセージの種類
- 数字はメッセージコード
- コロン後は具体的な問題の説明
- 括弧内はメッセージの識別子

### メッセージの種類

C: Convention (コーディング規約違反)  
R: Refactor (リファクタリング推奨)

W: Warning (警告)  
E: Error (エラー)  
F: Fatal (致命的なエラー)

## 評価スコア

pylint は 0.0~10.0 の範囲でコードの品質スコアを表示します。10.0 が最高評価です。

```
Your code has been rated at 8.75/10
```

## pylint が検出する主な問題

### 1. コーディングスタイル:

- インデントのずれ
- 命名規則違反
- 一行の長さ超過

### 2. プログラミング上の問題:

- 未使用の変数・インポート
- 未定義の変数参照
- 重複コード

### 3. ドキュメント不足:

- モジュール・関数のドキュメント文字列がない
- パブリックメソッドの説明不足

### 4. 設計上の問題:

- 複雑すぎる関数
- 多すぎるパラメータ
- クラス設計の問題

💡 **初心者向けヒント** : すべての警告を一度に修正する必要はありません。最初は致命的なエラー (F) と通常のエラー (E) から修正し、徐々に警告 (W) やコーディング規約 (C) に対応していくとよいでしょう。

## 4. 基本的な pylint コマンド

pylint の主要なコマンドとその使い方を見ていきましょう。

### 基本的な使い方

```
# 単一ファイルの解析  
pylint my_file.py
```

```
# モジュール・パッケージの解析  
pylint my_package  
  
# 複数ファイルの解析  
pylint file1.py file2.py  
  
# ディレクトリ内の全Pythonファイルを解析  
pylint *.py
```

## 出力の制御

```
# 詳細な出力を表示  
pylint -v my_file.py  
  
# 特定のメッセージだけを表示  
pylint --disable=all --enable=unused-variable my_file.py  
  
# 特定のメッセージを無視  
pylint --disable=missing-docstring my_file.py  
  
# 結果をファイルに出力  
pylint my_file.py > pylint_report.txt
```

## フォーマット制御

```
# 出力フォーマットの変更  
pylint --output-format=text my_file.py # デフォルト  
pylint --output-format=parseable my_file.py  
pylint --output-format=colorized my_file.py  
pylint --output-format=json my_file.py
```

## 特定の問題を確認する

```
# 未使用変数のみチェック  
pylint --disable=all --enable=unused-variable my_file.py  
  
# 命名規則のみチェック  
pylint --disable=all --enable=invalid-name my_file.py  
  
# 複雑度が高すぎる関数を検出  
pylint --disable=all --enable=too-many-branches my_file.py
```

## ファイル内でのルール無効化

コード内で特定のルールを無効化する方法:

```
# 行末でのルール無効化
x = 1 # pylint: disable=invalid-name

# ブロック内でのルール無効化
# pylint: disable=missing-docstring
def my_function():
    pass
# pylint: enable=missing-docstring

# ファイル全体でのルール無効化 ( ファイル先頭に記述 )
# pylint: disable=missing-docstring
```

💡 **初心者向けヒント** : すべての警告を抑制するのではなく、意図的に規約を破る場合のみ `disable` コメントを使いましょう。例えば、特殊なケースで短い変数名が適切な場合などです。

## 5. pylint 設定ファイルの作成

プロジェクトごとにカスタマイズした pylint 設定を作成する方法を学びましょう。

### 設定ファイルの生成

```
# デフォルト設定を出力してカスタマイズの基礎にする
pylint --generate-rcfile > .pylintrc
```

### 主な設定オプション

.pylintrc ファイルには多くの設定があります。主要なものの見ていきましょう:

#### メッセージ制御

```
[MESSAGES CONTROL]
# 無効化するメッセージ
disable=missing-docstring,invalid-name

# 有効化するメッセージ
enable=unused-variable
```

#### 基本設定

```
[MASTER]
# 並列処理数
jobs=4
```

```
# 解析から除外するファイルやディレクトリ  
ignore=CVS,tests,migrations  
  
# Pythonパスへの追加  
init-hook='import sys; sys.path.append(".")'
```

## 書式設定

```
[FORMAT]  
# 行の最大長  
max-line-length=100  
  
# インデントのスペース数  
indent-string=' '  
  
# 変数名のスタイル  
variable-regex=[a-zA-Z_][a-zA-Z0-9_]{0,30}$
```

## レポート設定

```
[REPORTS]  
# レポートのフォーマット (text, parseable, colorized, json)  
output-format=colorized  
  
# 評価スコアを表示  
evaluation=10.0 - ((float(5 * error + warning + refactor + convention) /  
statement) * 10)
```

## Python での設定

直接ファイルに設定する代わりに、`pyproject.toml` で設定することもできます:

```
[tool pylint]  
max-line-length = 100  
disable = [  
    "missing-docstring",  
    "invalid-name",  
]
```

## 設定の優先順位

pylint は以下の順序で設定を読み込みます:

1. コマンドラインオプション

2. PYLINTRC 環境変数で指定されたファイル
3. ユーザーホームディレクトリの .pylintrc
4. 現在のディレクトリにある .pylintrc または pyproject.toml

💡 **初心者向けヒント** : プロジェクトに最適な設定を見つけるには時間がかかります。最初は厳しそうな設定よりも、必要最低限のルールから始めて、徐々に追加していくアプローチが効果的です。

## 6. pylint を CI/CD に組み込む

pylint を CI/CD パイプラインに組み込む方法を学びましょう。

CI/CD での実行戦略

### 1. エラーレベルに基づく実行:

- 開発中: すべてのチェックを実行
- CI/CD: 重要なエラーだけで失敗させる

### 2. 段階的な導入:

- 既存プロジェクト: まずはエラー (E, F) のみチェック
- 新規プロジェクト: すべてのチェックを有効に

CI/CD での実行例

GitHub Actions の場合:

```
# .github/workflows/pylint.yml
name: Pylint

on: [push, pull_request]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.10"
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pylint
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
      - name: Run pylint
        run: |
          pylint --fail-under=7 $(git ls-files '*.py')
```

## 結果に基づく成功/失敗の制御

```
# スコアが7.0未満なら失敗させる
pylint --fail-under=7 my_file.py

# 特定のエラーのみで失敗させる
pylint --disable=all --enable=unused-variable,undefined-variable --fail-on=unused-
variable my_file.py
```

## 段階的なコード品質向上の進め方

- 最初は重要なエラーのみをチェック:

```
pylint --disable=all --enable=error,fatal --fail-under=9.0 *.py
```

- 警告を追加:

```
pylint --disable=all --enable=error,fatal,warning --fail-under=8.0 *.py
```

- コーディング規約も追加:

```
pylint --fail-under=7.0 *.py
```

## 既存プロジェクトでの対応

大きな既存プロジェクトでは、一度にすべての問題を修正するのは現実的ではありません。以下のアプローチが有効です:

- 現在の状態をベースラインとして記録:

```
pylint my_project > baseline.txt
```

- 新しいコードでは高い基準を適用:

```
git diff origin/main -- '*.py' | pylint --from-stdin
```

**💡 初心者向けヒント :** CI/CD パイプラインでの pylint チェックは、短時間で終わるよう設定しましょう。時間のかかるチェックは別のステップにするか、重要なファイルのみに限定することを検討してください。

## 7. pytest の基本概念

pytest の基本的な概念とテストの仕組みを理解しましょう。

### pytest の特徴

- **シンプルな構文**: `assert` 文だけでテストを書ける
- **自動検出**: `test_` で始まるファイルとメソッドを自動的に探して実行
- **詳細なエラーレポート**: 失敗したテストの詳細情報を表示
- **フィクスチャ**: テスト環境のセットアップとクリーンアップを効率化
- **パラメータ化**: 同じテストを複数のデータセットで実行

### テストファイルの命名規則

pytest は以下の命名パターンのファイルを自動的にテストとして認識します:

- `test_*`
- `*_test.py`

### テスト関数の命名規則

以下の命名パターンの関数や、Test クラスのメソッドが自動的にテストとして実行されます:

- `test_*`
- `*_test`

### テストの基本構造

```
# test_sample.py
def test_addition():
    result = 1 + 1
    assert result == 2

def test_string_length():
    text = "Hello"
    assert len(text) == 5
```

### アサーション (assertion)

pytest では Python の標準的な `assert` 文を使います。失敗した場合は詳細情報を表示します。

```
def test_complex_data():
    data = {"name": "Alice", "scores": [85, 90, 78]}
    assert data["name"] == "Alice"
    assert len(data["scores"]) == 3
    assert sum(data["scores"]) / len(data["scores"]) > 80
```

### テストクラス

関連するテストをクラスにまとめることができます:

```
class TestCalculator:  
    def test_add(self):  
        assert 1 + 1 == 2  
  
    def test_subtract(self):  
        assert 3 - 1 == 2
```

⌚ **初心者向けヒント** : pytest のシンプルさが最大の強みです。複雑なテストフレームワークの文法を覚える必要がなく、Python の標準機能だけでテストが書けます。関連するテストはクラスにまとめることで、コードの整理と共通のセットアップ・クリーンアップが容易になります。

## 8. 基本的なテストの書き方

実際のテストコードの書き方を学びましょう。

### シンプルな関数のテスト

まず、テスト対象の関数を作ります:

```
# calculator.py  
def add(a, b):  
    return a + b  
  
def subtract(a, b):  
    return a - b  
  
def multiply(a, b):  
    return a * b  
  
def divide(a, b):  
    if b == 0:  
        raise ValueError("Cannot divide by zero")  
    return a / b
```

次に、テストコードを書きます:

```
# test_calculator.py  
import pytest  
from calculator import add, subtract, multiply, divide  
  
def test_add():  
    assert add(1, 2) == 3  
    assert add(-1, 1) == 0  
    assert add(0, 0) == 0
```

```

def test_subtract():
    assert subtract(3, 1) == 2
    assert subtract(1, 1) == 0
    assert subtract(1, 2) == -1

def test_multiply():
    assert multiply(2, 3) == 6
    assert multiply(-2, 3) == -6
    assert multiply(0, 5) == 0

def test_divide():
    assert divide(6, 3) == 2
    assert divide(1, 2) == 0.5
    assert divide(-6, 2) == -3

```

## 例外のテスト

例外が発生することを確認するテスト:

```

def test_divide_by_zero():
    with pytest.raises(ValueError) as excinfo:
        divide(1, 0)
    assert "Cannot divide by zero" in str(excinfo.value)

```

## パラメータ化テスト

同じテストロジックを複数のデータセットで実行:

```

@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),
    (0, 0, 0),
    (-1, 1, 0),
    (5, -5, 0),
])
def test_add_parametrized(a, b, expected):
    assert add(a, b) == expected

```

## テストのスキップ

条件付きでテストをスキップ:

```

@pytest.mark.skip(reason="暫定的に無効化")
def test_future_feature():
    # 将来実装予定の機能のテスト
    pass

@pytest.mark.skipif(sys.version_info < (3, 8), reason="Python 3.8以上が必要")

```

```
def test_new_python_feature():
    # Python 3.8以上の機能を使ったテスト
    pass
```

## テストの期待される失敗

まだ修正されていないバグのテスト:

```
@pytest.mark.xfail(reason="このバグはまだ修正されていません")
def test_known_bug():
    # 既知のバグに関するテスト
    assert buggy_function() == expected_result
```

## テストの実行方法

```
# すべてのテストを実行
pytest

# 特定のテストファイルを実行
pytest test_calculator.py

# 特定のテスト関数を実行
pytest test_calculator.py::test_add

# 特定のパラメータ化テストを実行
pytest test_calculator.py::test_add_parametrized[1-2-3]

# 名前でテストをfiltrリング
pytest -k "add or multiply"

# 出力の詳細度を増やす
pytest -v

# テスト実行中にprint文の出力を表示
pytest -v --capture=no
```

💡 **初心者向けヒント** : テストは「AAA」原則に従うと書きやすくなります:

1. **Arrange**: テストの準備（入力データなど）
  2. **Act**: テスト対象の関数を実行
  3. **Assert**: 結果を検証
- この順序で書くと、テストの意図が明確になります。

## 9. テストフィクスチャの活用

テストの前後に実行される準備・後処理のコードを「フィクスチャ」として定義する方法を学びましょう。

## フィクスチャの基本

フィクスチャは `@pytest.fixture` デコレータで定義します:

```
import pytest
import tempfile
import os

@pytest.fixture
def temp_file():
    # セットアップ
    fd, path = tempfile.mkstemp()
    with open(path, 'w') as f:
        f.write("テスト用テキスト")

    # テスト関数にパスを渡す
    yield path

    # クリーンアップ
    os.close(fd)
    os.unlink(path)
```

フィクスチャをテストで使用:

```
def test_read_file(temp_file):
    with open(temp_file, 'r') as f:
        content = f.read()
    assert content == "テスト用テキスト"
```

## フィクスチャのスコープ

フィクスチャの再利用範囲を指定できます:

```
@pytest.fixture(scope="function") # デフォルト: 各テスト関数ごとに実行
@pytest.fixture(scope="class")     # テストクラスごとに1回実行
@pytest.fixture(scope="module")   # テストモジュールごとに1回実行
@pytest.fixture(scope="session")  # テスト実行全体で1回だけ実行
```

例:

```
@pytest.fixture(scope="module")
def expensive_resource():
    # 重い処理 (データベース接続など)
    resource = create_expensive_resource()
    yield resource
```

```
# クリーンアップ
resource.close()
```

## フィクスチャの依存関係

フィクスチャは他のフィクスチャに依存できます:

```
@pytest.fixture
def user():
    return {"id": 1, "name": "Test User"}

@pytest.fixture
def logged_in_user(user):
    # user フィクスチャを利用
    user["logged_in"] = True
    return user
```

テストでの使用:

```
def test_user_profile(logged_in_user):
    assert logged_in_user["logged_in"] == True
    assert logged_in_user["name"] == "Test User"
```

## 共通フィクスチャの定義

プロジェクト全体で使えるフィクスチャは `conftest.py` に定義します:

```
# conftest.py
import pytest
import sqlite3

@pytest.fixture(scope="session")
def db_connection():
    conn = sqlite3.connect(":memory:")
    conn.execute("CREATE TABLE users (id INTEGER, name TEXT)")
    yield conn
    conn.close()

@pytest.fixture
def db_with_data(db_connection):
    cursor = db_connection.cursor()
    cursor.execute("INSERT INTO users VALUES (1, 'Alice')")
    cursor.execute("INSERT INTO users VALUES (2, 'Bob')")
    db_connection.commit()
    yield db_connection
    cursor.execute("DELETE FROM users")
    db_connection.commit()
```

## パラメータ化フィクスチャ

複数のケースを試したい場合:

```
@pytest.fixture(params=["sqlite", "mysql", "postgresql"])
def db_type(request):
    return request.param

def test_database_connection(db_type):
    assert db_type in ["sqlite", "mysql", "postgresql"]
    # db_type に応じた接続テスト
```

## フィクスチャの自動利用

autouse=True で全テストに自動適用:

```
@pytest.fixture(autouse=True)
def setup_environment():
    # 全テスト前に環境変数を設定
    os.environ["TEST_MODE"] = "True"
    yield
    # 全テスト後に環境変数をクリア
    del os.environ["TEST_MODE"]
```

**💡 初心者向けヒント :** フィクスチャは、テストの準備と後処理を効率化します。データベース接続やテストファイルの作成など、繰り返し使う設定はフィクスチャにすると便利です。また、テストコードの可読性も向上します。

## 10. モックとパッチング

外部サービスや重い処理を含む関数をテストする際、モックとパッチングを使うと効率的にテストできます。

### モックとは

モックは実際のオブジェクトの振る舞いを模倣する偽のオブジェクトです。以下の用途があります:

- 外部 API やデータベースに依存せずテストを実行
- 処理時間の長い関数を高速にテスト
- 特定の条件・例外を簡単に再現

### pytest-mock の利用

pytest-mock プラグインを使うと、モックを簡単に作れます:

```
pip install pytest-mock
```

基本的な使い方:

```
def test_function_with_mock(mocker):
    # 関数やメソッドをモック化
    mock_function = mocker.patch('module.function_name')

    # 戻り値の設定
    mock_function.return_value = "モックの戻り値"

    # テスト対象のコードを呼び出す
    result = call_function_that_uses_mocked_function()

    # アサーション
    assert result == "期待する結果"

    # モック関数が呼ばれたことを確認
    mock_function.assert_called_once()
```

## パッチングの例

実際の例を見てみましょう。次のようなコードをテストするとします:

```
# weather.py
import requests

def get_weather(city):
    """指定した都市の天気を取得する"""
    url = f"https://api.example.com/weather?city={city}"
    response = requests.get(url)
    if response.status_code == 200:
        return response.json()["weather"]
    else:
        return "Unable to get weather data"

def is_sunny(city):
    """指定した都市が晴れかどうかを判定する"""
    weather = get_weather(city)
    return weather == "Sunny"
```

このコードをモックを使ってテスト:

```
# test_weather.py
import pytest
from weather import get_weather, is_sunny
```

```
def test_is_sunny_when_sunny(mocker):
    # get_weather 関数をモック化
    mock_get_weather = mocker.patch('weather.get_weather')
    mock_get_weather.return_value = "Sunny"

    # is_sunny 関数をテスト
    result = is_sunny("Tokyo")

    # 検証
    assert result == True
    mock_get_weather.assert_called_once_with("Tokyo")

def test_is_sunny_when_rainy(mocker):
    # get_weather 関数をモック化して"Rainy"を返すように設定
    mocker.patch('weather.get_weather', return_value="Rainy")

    # is_sunny 関数をテスト
    result = is_sunny("London")

    # 検証
    assert result == False
```

## requests をモックする例

外部 API を呼び出す `requests` モジュールをモックする方法:

```
def test_get_weather_success(mocker):
    # Mock Response オブジェクトを作成
    mock_response = mocker.Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {"weather": "Cloudy"}

    # requests.get をモック化
    mocker.patch('requests.get', return_value=mock_response)

    # テスト実行
    result = get_weather("Paris")

    # 検証
    assert result == "Cloudy"

def test_get_weather_failure(mocker):
    # 失敗するレスポンスをモック
    mock_response = mocker.Mock()
    mock_response.status_code = 404

    # requests.get をモック化
    mocker.patch('requests.get', return_value=mock_response)

    # テスト実行
```

```
result = get_weather("InvalidCity")

# 検証
assert result == "Unable to get weather data"
```

## クラスのモック

クラスやそのメソッドをモックする方法:

```
# database.py
class Database:
    def connect(self):
        # 実際はデータベースに接続
        pass

    def query(self, sql):
        # 実際はSQLを実行
        pass

    def close(self):
        # 接続を閉じる
        pass

def get_user_name(user_id):
    db = Database()
    db.connect()
    result = db.query(f"SELECT name FROM users WHERE id = {user_id}")
    db.close()
    return result
```

このクラスをモックするテスト:

```
def test_get_user_name(mocker):
    # Database クラスのモックを作成
    mock_db = mocker.Mock()
    # query メソッドの戻り値を設定
    mock_db.query.return_value = "Test User"

    # Database クラスのコンストラクタがモックを返すよう設定
    mocker.patch('__main__.Database', return_value=mock_db)

    # テスト実行
    result = get_user_name(1)

    # 検証
    assert result == "Test User"
    mock_db.connect.assert_called_once()
    mock_db.query.assert_called_once_with("SELECT name FROM users WHERE id = 1")
    mock_db.close.assert_called_once()
```

## コンテキストマネージャのモック

`with`ステートメントでのコンテキストマネージャをモックする例:

```
from contextlib import contextmanager

@contextmanager
def db_transaction():
    # 実際はトランザクションを開始
    print("Transaction started")
    try:
        yield
    finally:
        # 実際はコミットまたはロールバック
        print("Transaction ended")

def update_user(user_id, name):
    with db_transaction():
        # データベース更新処理
        print(f"Updating user {user_id} to {name}")
    return True
```

コンテキストマネージャのテスト:

```
def test_update_user(mocker, capsys):
    # コンテキストマネージャをモック
    mock_context = mocker.patch('__main__.db_transaction')

    # テスト実行
    result = update_user(1, "New Name")

    # 検証
    assert result == True
    mock_context.assert_called_once()
    captured = capsys.readouterr()
    assert "Updating user 1 to New Name" in captured.out
```

💡 **初心者向けヒント** : モックは強力ですが、過剰に使うとテストが実際の動作から乖離する危険性があります。外部依存がある場合や、テストが遅くなる場合にのみモックを使い、内部ロジックはできるだけ実際のコードでテストしましょう。

## 11. pytest のプラグイン活用

pytest の機能を拡張する便利なプラグインを紹介します。

### 主要なプラグイン

## 1. pytest-cov: コードカバレッジ

テストがコードのどの部分をカバーしているか測定します。

```
pip install pytest-cov
```

使い方:

```
# カバレッジレポートを生成  
pytest --cov=mypackage  
  
# HTMLレポートを生成  
pytest --cov=mypackage --cov-report=html  
  
# XMLレポートを生成 ( CI/CDに便利 )  
pytest --cov=mypackage --cov-report=xml
```

## 2. pytest-xdist: 並列テスト実行

複数プロセスでテストを並列実行し、高速化します。

```
pip install pytest-xdist
```

使い方:

```
# CPUコア数に応じて並列実行  
pytest -xvs -n auto  
  
# 4つのプロセスで並列実行  
pytest -xvs -n 4
```

## 3. pytest-benchmark: パフォーマンステスト

コードの実行速度を計測・比較できます。

```
pip install pytest-benchmark
```

使い方:

```
def test_performance(benchmark):  
    # この関数の実行時間を測定
```

```
result = benchmark(lambda: sum(range(1000000)))
assert result == 499999500000
```

## 4. pytest-mock: モックング補助

前章で説明したモックをより簡単に実装できます。

```
pip install pytest-mock
```

## 5. pytest-timeout: テストのタイムアウト設定

テストが一定時間を超えた場合に失敗させます。

```
pip install pytest-timeout
```

使い方:

```
# すべてのテストに5秒のタイムアウトを設定
pytest --timeout=5

# 特定のテストにタイムアウトを設定
@pytest.mark.timeout(10)
def test_long_running():
    # 長時間実行される処理
    pass
```

## プラグインの組み合わせ例

複数のプラグインを組み合わせて使用する例:

```
# カバレッジ測定 + 並列実行 + HTMLレポート生成
pytest --cov=mypackage --cov-report=html -n auto
```

## カスタムプラグインの作成

独自のプラグインを作成することもできます。基本的な手順:

1. `conftest.py` にフックを追加する方法（簡易的）：

```
# conftest.py
def pytest_runtest_setup(item):
```

```
# 各テスト実行前に呼ばれる
print(f"Setting up {item.name}")
```

## 2. プラグインパッケージとして実装する方法:

```
# myplugin.py
def pytest_addoption(parser):
    parser.addoption("--custom-option", action="store", default="default",
                     help="カスタムオプションの説明")

def pytest_configure(config):
    print(f"カスタムオプション値: {config.getvalue('--custom-option')}"")
```

使用:

```
# -p オプションでプラグインを指定
pytest -p myplugin --custom-option=value
```

⌚ **初心者向けヒント** : プラグインは pytest の機能を大幅に拡張します。しかし、必要なプラグインだけをインストールし、テストの依存関係と複雑性のバランスを取りましょう。カバレッジ (pytest-cov) と並列実行 (pytest-xdist) は特に CI/CD 環境で役立ちます。

## 12. コードカバレッジの測定

コードカバレッジはテストがコードのどの部分を実行しているかを示す指標です。

カバレッジの種類

1. 行カバレッジ (Line Coverage) : コードの各行が実行されたかどうか
2. 分岐カバレッジ (Branch Coverage) : すべての条件分岐 (if/else など) が実行されたか
3. パスカバレッジ (Path Coverage) : すべての可能な実行パスが実行されたか

pytest-cov の使用方法

まずインストール:

```
pip install pytest-cov
```

基本的な使い方:

```
# mypackage パッケージのカバレッジを測定
pytest --cov=mypackage
```

```
# 特定のディレクトリのカバレッジを測定  
pytest --cov=mypackage/utils  
  
# 複数のパッケージ/モジュールのカバレッジを測定  
pytest --cov=mypackage --cov=another_package
```

## カバレッジレポートの形式

```
# ターミナルに結果を表示 ( デフォルト)  
pytest --cov=mypackage  
  
# HTML形式のレポートを生成  
pytest --cov=mypackage --cov-report=html  
  
# XML形式のレポートを生成 ( CI/CDツールと連携しやすい)  
pytest --cov=mypackage --cov-report=xml  
  
# ターミナル + HTML  
pytest --cov=mypackage --cov-report=term --cov-report=html
```

## カバレッジの設定

.coveragerc ファイルで設定をカスタマイズできます:

```
# .coveragerc  
[run]  
source = mypackage  
omit =  
    # 除外するファイルやディレクトリ  
    */tests/*  
    */__init__.py  
  
[report]  
# 出力制御  
exclude_lines =  
    # 常に除外する行  
    pragma: no cover  
  
    # デバッグ用コードを除外  
def __repr__  
  
    # Type checkingブロックを除外  
if TYPE_CHECKING:  
  
    # エラー処理を除外  
except ImportError:  
  
    # 実行されないコードを除外  
if __name__ == '__main__':
```

```
# 最低カバレッジ率
fail_under = 80
```

## 部分的にカバレッジを除外

特定のコード部分をカバレッジ計測から除外:

```
def complex_function():
    # 通常のコード（カバレッジ計測対象）
    result = do_something()

    # このブロックはカバレッジから除外
    if debug_mode: # pragma: no cover
        print("デバッグ情報")
        log_debug_info()

    return result
```

## CI/CD でのカバレッジ確認

GitHub Actions の例:

```
# .github/workflows/test.yml
name: Test with Coverage

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.10"
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pytest pytest-cov
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
      - name: Test with pytest and coverage
        run: |
          pytest --cov=mypackage --cov-report=xml --cov-report=term
      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
```

```
file: ./coverage.xml
fail_ci_if_error: true
```

## カバレッジ目標の設定

最低限のカバレッジ率を設定し、未達の場合にテストを失敗させる:

```
# カバレッジが90%未満なら失敗
pytest --cov=mypackage --cov-fail-under=90
```

## コードカバレッジの効果的な使い方

1. **段階的に目標を設定**: 新規プロジェクトなら高いカバレッジを目指し、既存プロジェクトなら現状から徐々に改善
2. **重要なコードを優先**: すべてのコードが同じ価値ではない。ビジネスロジックを優先
3. **カバレッジだけに固執しない**: 高いカバレッジがあっても、テストの品質が重要
4. **リスクベースのアプローチ**: バグの影響が大きい部分に注力

💡 **初心者向けヒント** : コードカバレッジは重要なメトリックですが、100%を目指す必要はありません。重要なのは、テストの質と重要なコードパスがカバーされていることです。80-90%のカバレッジを目標に設定し、重要なモジュールはより高いカバレッジを目指しましょう。

## 13. pytest を CI/CD に組み込む

テストを CI/CD パイプラインに組み込み、自動的に実行する方法を学びましょう。

### CI/CD でのテスト実行戦略

1. **ファースト・テスト**: コミットごとに高速なテストを実行
2. **フル・テスト**: マージ前やリリース前に全テストを実行
3. **定期的なテスト**: 夜間ビルドなどで時間のかかるテストを実行

### pytest の CI/CD 向け設定

CI/CD 環境で効率的に pytest を実行するオプション:

```
# JUnitXML形式でレポート出力 (CI/CDツールと連携しやすい)
pytest --junitxml=test-results.xml

# テストが1つ失敗したら即座に停止 (早期フィードバック)
pytest -x

# 最初のN個のテスト失敗後に停止
pytest --maxfail=5

# 前回失敗したテストを最初に実行
pytest --failed-first
```

```
# 前回失敗したテストのみ実行  
pytest --last-failed  
  
# テスト実行時間を表示（遅いテストを特定）  
pytest --durations=5
```

## GitHub Actions での実装例

```
# .github/workflows/test.yml  
name: Python Tests  
  
on: [push, pull_request]  
  
jobs:  
  test:  
    runs-on: ubuntu-latest  
    strategy:  
      matrix:  
        python-version: ["3.8", "3.9", "3.10", "3.11"]  
  
    steps:  
      - uses: actions/checkout@v3  
      - name: Set up Python ${{ matrix.python-version }}  
        uses: actions/setup-python@v4  
        with:  
          python-version: ${{ matrix.python-version }}  
      - name: Install dependencies  
        run: |  
          python -m pip install --upgrade pip  
          pip install pytest pytest-cov  
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi  
      - name: Test with pytest  
        run: |  
          pytest --cov=mypackage --cov-report=xml --junitxml=test-results.xml  
      - name: Upload test results  
        uses: actions/upload-artifact@v3  
        with:  
          name: test-results-${{ matrix.python-version }}  
          path: |  
            test-results.xml  
            coverage.xml
```

## マルチ環境テスト

```
# tox.ini  
[tox]  
envlist = py38, py39, py310, py311  
isolated_build = True
```

```
[testenv]
deps =
    pytest
    pytest-cov
commands =
    pytest --cov={envsitepackagesdir}/mypackage {posargs:tests}
```

使い方:

```
# tox をインストール
pip install tox

# すべての環境でテスト実行
tox

# 特定の環境でテスト実行
tox -e py310
```

## CI/CD でのテスト最適化

大規模プロジェクトでのテスト時間を短縮する方法:

### 1. 並列実行:

```
# -n オプションで並列実行
pytest -n auto
```

### 2. テストの分割:

```
# GitHub Actions でテストを分割
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        test-group: [1, 2, 3, 4]
    steps:
      # ...
      - name: Run tests
        run: |
          python -m pytest tests/test_group_${{ matrix.test-group }}/
--cov=mypackage
```

### 3. テストの優先順位付け:

```
# 変更されたファイルに関連するテストのみ実行  
pytest $(git diff --name-only origin/main... | grep -E "^tests/.*\.\py$")
```

#### 4. キャッシュの活用:

```
# GitHub Actions でのキャッシング  
- uses: actions/cache@v3  
  with:  
    path: ~/.cache/pip  
    key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}  
    restore-keys: |  
      ${{ runner.os }}-pip-
```

### テスト結果の可視化

#### 1. テストレポートの生成:

```
# HTMLレポート用プラグインをインストール  
pip install pytest-html  
  
# HTMLレポートを生成  
pytest --html=report.html --self-contained-html
```

#### 2. CI/CD ダッシュボードへの統合:

GitHub Actions の例:

```
- name: Publish Test Report  
  uses: mikepenz/action-junit-report@v3  
  if: always() # テスト失敗時も実行  
  with:  
    report_paths: "test-results.xml"
```

⌚ 初心者向けヒント : CI/CD でのテスト実行は、開発者のフィードバックループを短縮し、品質を維持するために重要です。まずは基本的なテスト実行から始めて、徐々にパフォーマンス最適化やレポート機能を追加していきましょう。並列実行とテストの優先順位付けは、プロジェクトが大きくなつたときに特に役立ちます。

## 14. GitHub Actions による自動化

GitHub Actions を使って pylint と pytest を自動実行する具体的な設定方法を見ていきましょう。

### GitHub Actions の基本

GitHub Actions はリポジトリ内の `.github/workflows/` ディレクトリに YAML ファイルを作成することで設定します。

## 基本的なワークフロー

```
# .github/workflows/python-checks.yml
name: Python Checks

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  lint_and_test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.10"
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pylint pytest pytest-cov
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
      - name: Lint with pylint
        run: |
          pylint --fail-under=7.0 $(git ls-files '*.py')
      - name: Test with pytest
        run: |
          pytest --cov=mypackage --cov-report=xml
```

## 複数の Python バージョンでのテスト

```
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.8", "3.9", "3.10", "3.11"]
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}
      # ... 以下、インストールとテスト
```

## キャッシュを活用した高速化

```
steps:
  - uses: actions/checkout@v3
  - name: Set up Python
    uses: actions/setup-python@v4
    with:
      python-version: "3.10"
      cache: "pip" # pip のキャッシュを有効化
  - name: Install dependencies
    run: |
      python -m pip install --upgrade pip
      pip install -r requirements.txt
```

## コンディショナルステップ

```
steps:
  # ...
  - name: Run lint
    run: pylint $(git ls-files '*.py')
    continue-on-error: true # lint 失敗でも次のステップへ

  - name: Full tests
    if: github.event_name == 'pull_request' # PR時のみ実行
    run: pytest --cov=mypackage

  - name: Fast tests
    if: github.event_name == 'push' # push時は高速テストのみ
    run: pytest -k "not slow"
```

## マトリックスビルドで OS とバージョンの組み合わせをテスト

```
jobs:
  test:
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        python-version: ["3.8", "3.10"]
      runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python ${{ matrix.python-version }} on ${{ matrix.os }}
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}
      # ... 以下、インストールとテスト
```

## テスト結果とカバレッジレポートの表示

```
- name: Run tests
  run: |
    pytest --cov=mypackage --cov-report=xml --junitxml=test-results.xml

- name: Upload coverage to Codecov
  uses: codecov/codecov-action@v3
  with:
    file: ./coverage.xml
    fail_ci_if_error: false

- name: Publish Test Results
  uses: EnricoMi/publish-unit-test-result-action@v2
  if: always()
  with:
    files: test-results.xml
```

## ワークフローの完全な例

```
# .github/workflows/python-package.yml
name: Python Package

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.10"
          cache: "pip"
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pylint
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
      - name: Lint with pylint
        run: |
          pylint --fail-under=7.0 $(git ls-files '*.py')
```

```
test:  
  needs: lint  
  runs-on: ubuntu-latest  
  strategy:  
    matrix:  
      python-version: ["3.8", "3.10"]  
  steps:  
    - uses: actions/checkout@v3  
    - name: Set up Python ${{ matrix.python-version }}  
      uses: actions/setup-python@v4  
      with:  
        python-version: ${{ matrix.python-version }}  
        cache: "pip"  
    - name: Install dependencies  
      run: |  
        python -m pip install --upgrade pip  
        pip install pytest pytest-cov  
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi  
    - name: Test with pytest  
      run: |  
        pytest --cov=mypackage --cov-report=xml --junitxml=test-results.xml  
    - name: Upload coverage  
      uses: codecov/codecov-action@v3  
      with:  
        file: ./coverage.xml  
    - name: Upload test results  
      uses: actions/upload-artifact@v3  
      if: always()  
      with:  
        name: test-results-${{ matrix.python-version }}  
        path: test-results.xml  
  
deploy:  
  needs: test  
  if: github.event_name == 'push' && github.ref == 'refs/heads/main'  
  runs-on: ubuntu-latest  
  steps:  
    - uses: actions/checkout@v3  
    - name: Set up Python  
      uses: actions/setup-python@v4  
      with:  
        python-version: "3.10"  
    - name: Install dependencies  
      run: |  
        python -m pip install --upgrade pip  
        pip install build twine  
    - name: Build and publish  
      env:  
        TWINE_USERNAME: ${{ secrets.PYPI_USERNAME }}  
        TWINE_PASSWORD: ${{ secrets.PYPI_PASSWORD }}  
      run: |  
        python -m build  
        twine upload dist/*
```

## Pull Requestへのコメント機能

```
- name: Comment PR with pylint score
  if: github.event_name == 'pull_request'
  uses: thollander/actions-comment-pull-request@v2
  with:
    message: |
      ## Pylint Score

      Current score: ${{ steps pylint.outputs.score }}/10

      ## Test Coverage

      Line coverage: ${{ steps coverage.outputs.line-rate }}%
      comment_tag: code-quality
```

⌚ **初心者向けヒント** : GitHub Actions は強力ですが、最初から複雑なワークフローを作る必要はありません。まずは基本的なテストと lint から始めて、徐々に機能を追加していきましょう。ワークフローは小さく保ち、必要に応じて複数のワークフローに分割すると管理しやすくなります。

## 15. ベストプラクティス

pylint と pytest を効果的に使うためのベストプラクティスをご紹介します。

### コード品質のベストプラクティス

1. **早期からの導入**: 新規プロジェクトでは最初から pylint を導入し、技術的負債を防ぐ
2. **チーム全体での合意**: 設定は全員で共有し、IDE の自動検出設定を揃える
3. **徐々に厳格化**: まずは基本的なエラーから修正し、徐々に警告も修正していく
4. **コードレビューとの連携**: PR の前に自動チェックを実行し、レビュー負担を減らす
5. **プロジェクトに合わせた設定**: プロジェクトの特性に合わせて .pylintrc をカスタマイズ

### テストのベストプラクティス

1. **テストの独立性**: 各テストは他のテストに依存せず、任意の順序で実行できるようにする

```
# 良い例：テストが独立している
def test_create_user():
    user = create_user("test_user")
    assert user.name == "test_user"

def test_delete_user():
    user = create_user("temp_user")  # テスト内で必要なセットアップをする
    result = delete_user(user.id)
    assert result is True
```

## 2. テスト名の明確化: 何をテストするのか名前から分かるようにする

```
# 良くない例
def test_function1():
    ...

# 良い例
def test_user_registration_with_valid_email_succeeds():
    ...
```

## 3. 一つのテストで一つの概念: 各テストは単一の機能や条件をテスト

```
# 良くない例 : 複数の機能を一つのテストでチェック
def test_user_functions():
    user = create_user("test")
    assert user.name == "test"
    user = update_user(user.id, name="new_name")
    assert user.name == "new_name"
    assert delete_user(user.id) is True

# 良い例 : 機能ごとに分ける
def test_create_user():
    ...

def test_update_user():
    ...

def test_delete_user():
    ...
```

## 4. テスト駆動開発を検討: 実装前にテストを書くことで設計を改善

1. 失敗するテストを書く
2. テストが通るコードを最小限実装
3. コードをリファクタリング

## 5. エッジケースのテスト: 境界値、エラーケース、極端なケースもテスト

```
def test_divide_by_zero_raises_error():
    with pytest.raises(ValueError) as excinfo:
        divide(10, 0)
    assert "division by zero" in str(excinfo.value)

def test_large_numbers():
    assert add(10**9, 10**9) == 2 * 10**9
```

## CI/CD のベストプラクティス

### 1. ファストフィードバック: CI パイプラインは素早く結果を返すように設計

```
# 最初に高速なチェックを実行
jobs:
  quick_checks:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Syntax check
        run: python -m py_compile $(git ls-files '*.py')

  lint:
    needs: quick_checks
    # ...

  tests:
    needs: lint
    # ...
```

### 2. 段階的な検証: 簡単なチェックから詳細なテストへ段階的に進む

### 3. 失敗時の明確なフィードバック: エラーメッセージは具体的で対応策が分かるものに

### 4. テスト成果物の保存: テスト結果やカバレッジレポートを保存

```
- name: Upload test artifacts
  uses: actions/upload-artifact@v3
  with:
    name: test-results
    path: |
      test-results.xml
      coverage.xml
      htmlcov/
```

### 5. 環境の一貫性: 開発環境と CI 環境の一貫性を確保

```
- name: Set up development environment
  run: |
    python -m pip install -e . # プロジェクトをインストール
```

## プロジェクト構成のベストプラクティス

### 1. 明確なディレクトリ構造: テストとソースコードの配置を標準化

```
myproject/
├── mypackage/
│   ├── __init__.py
│   ├── module1.py
│   └── module2.py
├── tests/
│   ├── test_module1.py
│   └── test_module2.py
├── .pylintrc
└── pytest.ini
└── requirements.txt
```

## 2. 共通設定ファイル: 設定を一元管理

```
# pytest.ini
[pytest]
testpaths = tests
python_files = test_*.py
python_functions = test_*
```

## 3. 依存関係の管理: 必要なパッケージを明示

```
# requirements.txt
pytest==7.3.1
pytest-cov==4.1.0
pylint==2.17.4

# requirements-dev.txt
-r requirements.txt
pytest-mock==3.10.0
black==23.3.0
```

## 4. 設定ファイルの最適化: 重複を避け、メンテナンスしやすく

```
# pyproject.toml で一元管理
[tool.pytest]
testpaths = ["tests"]

[tool pylint]
max-line-length = 100
disable = [
    "missing-module-docstring",
]
```

💡 **初心者向けヒント**：完璧を目指すより「継続的な改善」を心がけましょう。すべてのベストプラクティスを一度に導入しようとせず、プロジェクトの状況に応じて徐々に適用していくことがポイントです。最初は「動くテスト」があることが最も重要で、その後で品質やカバレッジを向上させていきましょう。

## 16. よくある質問と回答

pytest と pylint の使用中によく出てくる質問に回答します。

pylint に関する質問

Q: 「pylint で一時的に特定の警告を無効にする方法は？」

A: コード内でコメントを使って無効にできます:

```
# 行末での無効化
variable = 1 # pylint: disable=invalid-name

# ブロックでの無効化
# pylint: disable=unused-argument
def unused_param_function(unused_param):
    return True
# pylint: enable=unused-argument
```

Q: 「プロジェクト全体で特定のルールを無効にするには？」

A: `.pylintrc` ファイルまたは `pyproject.toml` で設定します:

```
# .pylintrc
[MESSAGES CONTROL]
disable=missing-docstring,invalid-name
```

または:

```
# pyproject.toml
[tool pylint.messages_control]
disable = [
    "missing-docstring",
    "invalid-name",
]
```

Q: 「PEP 8 のスタイルチェックだけ実行するには？」

A: pylint の代わりに flake8 を使うのがおすすめです:

```
pip install flake8
flake8 mypackage/
```

Q: 「タイプヒントやドキュメントチェックを含めた静的解析を強化するには？」

A: mypy と pydocstyle を追加します:

```
pip install mypy pydocstyle

# タイプチェック
mypy mypackage/

# ドキュメントスタイルチェック
pydocstyle mypackage/
```

Q: 「違うコード形式ツールの警告が矛盾する場合は？」

A: 優先順位を決めましょう。例えば、自動フォーマットツールは black を使い、その結果に対して pylint を実行します:

```
black mypackage/
pylint mypackage/
```

pytest に関する質問

Q: 「特定のテストだけを実行するには？」

A: いくつかの方法があります:

```
# ファイル名で指定
pytest tests/test_specific.py

# テスト名で指定
pytest tests/test_file.py::test_specific_function

# キーワードで指定
pytest -k "keyword"

# マーカーで指定
pytest -m slow
```

Q: 「テスト実行時に print 文の出力を表示するには？」

A: `-s` オプションまたは `--capture=no` を使用します:

```
pytest -s  
# または  
pytest --capture=no
```

**Q: 「テストの実行時間が遅いときの対処法は？」**

A: いくつかのアプローチがあります:

1. 遅いテストを特定:

```
pytest --durations=10 # 上位10件の遅いテストを表示
```

2. 並列実行:

```
pip install pytest-xdist  
pytest -n auto # 使用可能なCPUコア数で並列実行
```

3. 遅いテストをマークして別途実行:

```
@pytest.mark.slow  
def test_slow_operation():  
    ...
```

```
# 高速なテストだけ実行  
pytest -k "not slow"
```

```
# 遅いテストは別途実行  
pytest -m slow
```

**Q: 「テストごとに毎回データベースをリセットするには？」**

A: フィクスチャを使用します:

```
@pytest.fixture(autouse=True)  
def reset_database():  
    # テスト前の処理  
    db.truncate_all_tables()  
    # または  
    db.create_test_database()
```

```
yield # テストを実行  
# テスト後の処理  
db.drop_test_database()
```

Q: 「CIで失敗したテストだけを再実行するには？」

A: `--lf` (last-failed) オプションを使います:

```
pytest --lf
```

CI/CDに関する質問

Q: 「CIで依存関係のインストールが遅い場合の対処法は？」

A: キャッシュを活用します:

```
# GitHub Actions の例  
- uses: actions/cache@v3  
  with:  
    path: ~/.cache/pip  
    key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}  
    restore-keys: |  
      ${{ runner.os }}-pip-
```

Q: 「マージされたブランチだけで完全なテストを実行するには？」

A: イベントとブランチで条件分岐します:

```
jobs:  
  quick_test:  
    runs-on: ubuntu-latest  
    if: github.event_name == 'pull_request'  
    steps:  
      - name: Run critical tests  
        run: pytest -k "critical"  
  
  full_test:  
    runs-on: ubuntu-latest  
    if: github.event_name == 'push' && github.ref == 'refs/heads/main'  
    steps:  
      - name: Run all tests  
        run: pytest
```

## Q: 「テスト結果を開発者に分かりやすく通知するには？」

A: レポート機能とコメント機能を活用:

```
- name: Generate HTML report
  run: pytest --html=report.html

- name: Comment PR with results
  if: github.event_name == 'pull_request'
  uses: actions/github-script@v6
  with:
    script: |
      const fs = require('fs');
      const summary = fs.readFileSync('test-summary.txt', 'utf8');
      github.rest.issues.createComment({
        issue_number: context.issue.number,
        owner: context.repo.owner,
        repo: context.repo.repo,
        body: `## Test Results\n${summary}`
      });


```

⌚ 初心者向けヒント : エラーメッセージは Google 検索すると解決策が見つかることが多いです。コミュニティのサポートも積極的に活用しましょう。Stack Overflow や各ツールの GitHub Issues は貴重な情報源です。また、ドキュメントを読む習慣をつけると長期的に役立ちます。

## 17. 次のステップ

pylint と pytest の基本を学んだ後、スキルをさらに発展させるための次のステップをご紹介します。

### スキル向上のためのトピック

#### 1. 高度なテスト技術:

- プロパティベーステスト (hypothesis)
- 振る舞い駆動開発 (BDD)
- 負荷テスト・パフォーマンステスト

#### 2. コード品質の更なる向上:

- 複雑度分析と低減
- リファクタリング技術
- デザインパターンの適用

#### 3. CI/CD パイプラインの最適化:

- デプロイ自動化
- 環境管理
- インフラストラクチャのコード化

## 次に学ぶべきツール

### 1. 高度なコード品質ツール:

- **black**: 自動コードフォーマッター

```
pip install black  
black mypackage/
```

- **isort**: インポート文の自動整理

```
pip install isort  
isort mypackage/
```

- **mypy**: 静的型チェック

```
pip install mypy  
mypy mypackage/
```

- **bandit**: セキュリティ脆弱性の検出

```
pip install bandit  
bandit -r mypackage/
```

### 2. 高度なテストツール:

- **Hypothesis**: プロパティベーステスト

```
pip install hypothesis
```

```
from hypothesis import given  
from hypothesis.strategies import integers  
  
@given(integers(), integers())  
def test_addition_commutative(a, b):  
    assert a + b == b + a
```

- **pytest-bdd**: 振る舞い駆動開発

```
pip install pytest-bdd
```

```
# features/calculator.feature
Feature: Calculator
  Scenario: Addition
    Given I have entered 1 into the calculator
    And I have entered 2 into the calculator
    When I press add
    Then the result should be 3
```

- **Locust:** 負荷テスト

```
pip install locust
```

### 3. CI/CD とオートメーションツール:

- **pre-commit:** コミット前の自動チェック

```
pip install pre-commit
```

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/psf/black
    rev: 23.3.0
    hooks:
      - id: black
  - repo: https://github.com/pycqa/isort
    rev: 5.12.0
    hooks:
      - id: isort
  - repo: https://github.com/pycqa/pylint
    rev: v2.17.4
    hooks:
      - id: pylint
```

- **Coverage.py:** 詳細なカバレッジ分析

```
pip install coverage
coverage run -m pytest
coverage html
```

## 1. 一貫したプロジェクト構造:

- **Cookiecutter**: テンプレートからプロジェクト生成

```
pip install cookiecutter
cookiecutter https://github.com/audreyfeldroy/cookiecutter-pypackage
```

## 2. 開発環境の標準化:

- **pyenv**: Python バージョン管理
- **Poetry**: 依存関係とパッケージング管理

```
pip install poetry
poetry init
poetry add pytest pylint --dev
```

## 3. ドキュメント自動生成:

- **Sphinx**: ドキュメント生成

```
pip install sphinx
sphinx-quickstart
```

- **Read the Docs**: ドキュメントホスティング

## 実践的なアプリケーション

### 1. 実際のプロジェクトに適用:

- 小さな個人プロジェクトから始める
- 既存のプロジェクトに段階的に導入

### 2. チームでの採用:

- コーディング規約の策定
- レビュープロセスの確立
- 知識共有セッション

### 3. オープンソースへの貢献:

- 既存のオープンソースプロジェクトにコントリビュート
- テストや lint の改善提案を行う

## 学習リソース

### 1. 本やオンラインコース:

- "Python Testing with pytest" by Brian Okken
- "Clean Code in Python" by Mariano Anaya
- "Test-Driven Development with Python" by Harry Percival

## 2. コミュニティとフォーラム:

- PyTest ドキュメントとブログ
- Python Testing Podcast
- Stack Overflow での Q&A
- PyCon/PyConJP などのカンファレンス講演

⌚ **初心者向けヒント :** 一度にすべてを学ぼうとせず、一つのツールや技術を十分に理解してから次に進みましょう。実際のプロジェクトに適用しながら学ぶことで、理解が深まります。また、コミュニティに参加して質問したり、知識を共有したりすることも大切です。

---

以上で「Python コードチェック・テスト超入門: CI/CD のための pylint と pytest」の解説を終わります。この資料が Python 開発の品質向上に役立つことを願っています。ぜひ実際にコードに適用して、よりクリーンで安定した Python アプリケーションを開発してください！

何か質問があれば、お気軽にコミュニティで共有してください。Happy testing! ☺♦