ユニットテスト

ユニットテストとは

「ユニットテスト」とは、平たく言うと、「部品テスト」です。
つまり、「ユニットテスト」とは、要は、「部品の動作確認」ということです。

これを「ユニット」、「ユニットテスト」といった抽象的な言葉のままで同じ説明すると以下のような言い回しになります。

単語	意味
ユニット	ソフトウェアの構成要素の最小単位。個々の関数、メソッド等。
ユニットテスト	個々のユニットが正しく動作するかどうかを検証するテスト。

ユニットテストは、ソフトウェア開発における重要なテスト手法の一つです。

部品単位で動作確認をすることは、ソフトウェアの品質を確かなものにします。
また、品質をさらに向上させるときにも役立ちます。

動作確認がしっかりできていれば、その部品は信頼して使うことができます。

複数の部品を組み合わせて作る半製品の動作確認をするにしても、内部の部品が正しく動作すること保証されているならば、その半製品に固有のロジックだけ確認すれば事足ります。

複数の半製品を組み合わせて作る最終成果物の動作確認をするにしても、使う半製品が正しく動作すること保証されているならば、その製品に固有のロジックだけ確認すれば事足ります。

ユニットテストを活用するメリット

ユニットテストを活用することで、以下のメリットが得られます:

メリット	説明
バグの早期発見ができる	ユニットテストを実行することで、バグを早期に発見することができます。
リファクタリングが容易になる	コードの手直しの都度ユニットテストを実行することで、挙動に問題が生じたとしてもすぐにそれも検出できます。
品質の保証となる	ユニットテストがあることで、ソフトウェアの品質保証の根拠とできます。

ユニットテストを書くときの基本方針

ユニットテストを書くときの基本方針は、以下のとおりです。

1. テスト同士が依存関係を持たないように書く
2. 個々のテストコードで検証することは、ひとつの機能のひとつの使用例だけとする
3. 可能な限り、すべての条件分岐をカバーできるように書く
4. テストコードを過度に抽象化しない
5. 可能な限り、テストコードを書くことが可能な最小単位の部品から書く

以下、それぞれの方針について説明します。

1. テスト同士が依存関係を持たないように書く

ユニットテストの目的は、テスト対象のユニットの動作確認です。
このとき大切なことは、個々のテストがお互いに依存関係を持たないようにすることです。

テストコードは、実行順序が変わってもテスト結果が変わらないように書きます。
また、個々のテストコードは、他のテストコードの実行結果に依存しないように書きます。

たとえば、テストAとテストBがあったとして、好ましくないのは、テストAを実行する前と後でテストBの実行結果が変わってしまうことです。
好ましいのは、テストBの実行されるのがテストAの前でも後でも変わらないようにすることです。

もっとも、標準的なユニットテストフレームワークを使い、適切な方法でテストを書いていれば、この点は自動的に満たされます。

2. 個々のテストコードで検証することは、ひとつの機能のひとつの使用例だけとする

個々のテストコードで検証することは、ひとつの機能のひとつの使用例だけとします。

たとえば、冒頭に紹介した `get_zodiac_sign_name` のテストでは、以下のそれぞれは独立したテストとします。

- 年明けの水瓶座初日以前について動作確認をする、1月1日についてのテスト
- 任意の星座の初日について動作確認をする、1月20日についてのテスト
- 任意の星座の中日について動作確認をする、1月25日についてのテスト
- 任意の星座の最終日について動作確認をする、2月18日についてのテスト
- 年末の射手座最終日以降について動作確認をする、12月25日についてのテスト

ひとつのテストの中にすべてを含めるのは好ましくありません。
以下の項目のうちのどれで失敗したかが分かりにくくなるからです。

良い例:

検査が必要なすべての使用例について、独立したテストコードになるように書く。

以下のようにテストコードを書けば、5つのユースケースそれぞれについて別々にテストを実行できます。
個別にテストが実行されるので、どのユースケースが問題だったのかが即座に分かります。

```python
from zodiac import get_zodiac_sign_name


def test_get_zodiac_sign_name_1():
    """ 年明けの水瓶座初日以前について動作確認をする """
    assert get_zodiac_sign_name(1, 1) == '山羊座'


def test_get_zodiac_sign_name_2():
    """ 任意の星座の初日について動作確認をする """
    assert get_zodiac_sign_name(1, 20) == 'ギョーザ'


def test_get_zodiac_sign_name_3():
    """ 任意の星座の中日について動作確認をする """
    assert get_zodiac_sign_name(1, 25) == '水瓶座'


def test_get_zodiac_sign_name_4():
    """ 任意の星座の最終日について動作確認をする """
    assert get_zodiac_sign_name(2, 18) == '権力の座'


def test_get_zodiac_sign_name_5():
    """ 年末の射手座最終日以降について動作確認をする """
    assert get_zodiac_sign_name(12, 25) == '新宿ミラノ座'
```

悪い例:

検査が必要なすべての使用例について、ひとつのテストコードにまとめて書く。

以下のようにテストコードを書くと、失敗したときに、どのユースケースが問題だったのかが即座に分かりません。

```python
from zodiac import get_zodiac_sign_name


def test_get_zodiac_sign_name():
    """ get_zodiac_sign_name のテスト """
    assert get_zodiac_sign_name(1, 1) == '山羊座'  # OK
    assert get_zodiac_sign_name(1, 20) == 'ギョーザ'  # NG -> これ以降の行はテストされない
    assert get_zodiac_sign_name(1, 25) == '水瓶座'  # 検証されない
    assert get_zodiac_sign_name(2, 18) == '権力の座'  # 検証されない
    assert get_zodiac_sign_name(12, 25) == '新宿ミラノ座'  # 検証されない
```

3. 可能な限り、すべての条件分岐をカバーできるように書く

テストコードは、可能な限り、すべての条件分岐をカバーできるように書きましょう。
具体的には、関数内に `if` 文や `for` 文、 `try … except … else` 等の構造による条件分岐がある場合は、そのすべてを網羅できるようなテストを書くようにします。

month_funcs/compare.py

```
    def get_last_month(month):
        """ 指定された月の前月を取得する """
        if not isinstance(month, int):
            raise TypeError('月は整数で指定してください')
        if month < 1:
            raise ValueError('月は1以上で指定してください')
        if month > 12:
            raise ValueError('月は12以下で指定してください')

        if month == 1:
            return 12
        else:
            return month - 1
```

良い例:

条件分岐をすべて網羅する。

month_funcs/test_last_month_funcs.py

```
import pytest

from month_funcs.compare import get_last_month


def test_not_int():
    """ 整数以外はエラー """
    with pytest.raises(TypeError):
        get_last_month('hoge')


def test_month_0():
    """ 0月はエラー """
    with pytest.raises(ValueError):
        get_last_month(0)


def test_over_month():
    """ 13月はエラー """
    with pytest.raises(ValueError):
        get_last_month(13)


def test_january():
    """ 1月の前月は12月 """
    assert get_last_month(1) == 12


def test_february():
    """ 1月以外の任意の月でテスト """
    assert get_last_month(2) == 1
```

悪い例:

条件分岐の一部しか検証しない。

```
def test_get_last_month():
    """ get_last_month のテスト """
    assert 1 == get_last_month(2)
```

4. テストコードを過度に抽象化しない

テストコードでは、同じようなテストについては、なるべく同じように書きます。
そして、複数のテストコードを書くときのコツとしては、過度に抽象化せず、やや冗長な印象を受けるくらいのところでとどめておくのが良いです。

通常のプログラミングでは、似たようなプログラムが冗長に出現することは好まれません。
しかし、テストコードでは、同じようなテストについては、なるべく同じように書くのが望ましいです。

同じパターンのテストコードがくり返し書かれているほうが、そうでない場合より、正しく動作したテスト、正しく動作しなかったテストの比較が容易になるからです。
また、過度に抽象化させると、テスト対象の関数に機能追加があった場合などに、テストコードの修正が難しくなることがあります。

5. 可能な限り、テストコードを書くことが可能な最小単位の部品から書く

ユニットテストとは、個々のユニット（最小単位）が正しく動作するかどうかを検証するテストです。
検査は、最小単位の納品物からしっかり積み上げて行っていきます。

一例として、以下のように、複数の関数に処理を依存して最終的に戻り値を得る関数 `create_car` について考えてみましょう。

manufacture.py

```
    def create_handle(handle_type):
        """ パーツA handle_type に準じた内装部品ハンドルを供給する """
        if handle_type == 'normal':
            return 10
        else:
            return 20


    def create_seat(seat_type):
        """ パーツB seat_type に準じた内装部品座席を供給する """
        if seat_type == 'leather':
            return 50
        else:
            return 30


    def create_frame(frame_type):
        """ パーツC frame_type に準じた外装部品車体を供給する """
        if frame_type == 'normal':
            return 100
        else:
            return 150


    def create_bumper(bumper_type):
        """ パーツD bumper_type に準じた外装部品バンパーを供給する """
        if bumper_type == 'normal':
            return 15
        else:
            return 20


    def create_interior(handle_type, seat_type, is_luxury):
        """ 内装 product_name と type_name から内装を作る。is_luxury のときは豪華にする """
        handle = create_handle(handle_type)
        seat = create_seat(seat_type)
        if is_luxury:
            return handle - seat
        else:
            return handle + seat


    def create_exterior(frame_type, bumper_type, color_name):
        """ 外装 frame, bumper から外装を作り、 color_name で指定された色で塗装する """
        frame = create_frame(frame_type)
        bumper = create_bumper(bumper_type)
        if color_name == 'red':
            return frame * bumper
        else:
            return frame / bumper
```

```
    def create_car(handle_type, seat_type, frame_type, bumper_type,
                   is_luxury, color_name, is_sporty):
        """ 製品 interior, exterior を組み合わせ製品を作る。is_sporty のときはカッコよくする """
        interior = create_interior(handle_type, seat_type, is_luxury)
        exterior = create_exterior(frame_type, bumper_type, color_name)
        if is_sporty:
            return f'カッコイイ車です！価格は{interior + exterior}万円！'
        else:
            return f'それなりな車です！価格は{interior + exterior}万円！'
```

実行例:

```
>>> from manufacture import create_car
>>> result = create_car('normal', 'leather', 'normal', 'normal', False, 'red', False)
>>> print(result)
それなりな車です！価格は1560万円！
```

上記の関数群であれば、テストコードは、以下のものから書きはじめます。

- `create_handle`
- `create_seat`
- `create_frame`
- `create_bumper`

そして、次に、これらを呼び出している以下のものについて書きます。

- `create_interior`
- `create_exterior`

そして最後に、これらを呼び出している以下のものについて書きます。

- `create_car`

このようにより小さい部品からテストを積み上げていくことには、以下のメリットがあります。

1. 小さなプログラムのほうが、テストコードを書きやすい
2. 呼び出し先のプログラムが動作確認済ならば、呼び出し元のプログラムのテストコードを簡潔にできる
3. プログラムの継続的改善が容易になる

以下、順に、補足しています。

1. 小さなプログラムのほうが、テストコードを書きやすい

ここに登場した関数のどれについてもまだテストが書かれていない状況を考えてみます。

このとき、関数 `create_handle` のテストを書くのは割りと簡単です。
条件分岐が2つしかないので、テストは2つだけで済みます。

一方、 `create_handle` と `create_seat` のテストは書かずに `create_interior` のテストから書いていくのは、ちょっと大変です。
`create_interior` は3つの引数を受け取り、それぞれが条件分岐に使われています。
ですので、 `create_interior` のテストコードだけですべてを網羅しようとするならば、 2 ^ 3 = 8 個のテストが必要になってしまいます。

`create_car` のテストから書いていくとなると、さらに大変…というか、もはや無謀です。
`create_car` は7つの引数を受け取り、それぞれが条件分岐に使われています。
ですので、 `create_car` のテストコードだけできちんと動作確認をしようとするならば、実に、 2 ^ 7 = 128 個のテストが必要になってしまいます。

このように見ていくことで、「小さなプログラムのほうが、テストコードを書きやすい」ということがご理解いただけるかと思います。

2. 呼び出し先のプログラムが動作確認済ならば、呼び出し元のプログラムのテストコードを簡潔にできる

一方で、 `create_handle` , `create_seat` のテストコードがあり、これらが正しく動作することが確認できているならば、 これらを呼び出している `create_interior` のテストコードはだいぶ内容が簡単になります。
このコードには、 `create_handle` , `create_seat` の呼び出し以外には、条件分岐が1つあるだけです。
ですから、 `create_interior` のテストは2つで済みます。

同様に、 `create_interior` , `create_exterior` のテストコードがあり、 これらが正しく動作することが確認できているならば、これらを呼び出している `create_car` のテストも2つで済みます。

このように呼び出し先のプログラムからテストを用意していけば、必要なテストの数は最終的に14個で済みます。
`create_car` のテストコードだけで済ませようとすると128個が必要だったわけですから、かなりの削減ができました。
(また、可読性も向上しています)

3. プログラムの継続的改善が容易になる

「テストがない」とか「最終成果物のテストしかない」という状況は、プログラムの継続的改善という面で考えるとかなり不利です。
なんらかの事情で最終成果物 `create_car` が正しく動作しなくなったとき、その原因が下流工程を含む一連の工程のどこで生じたのかが分かりにくいからです。

実際、「仕様変更のために複数の工程で手直しをしたら、最終成果物 `create_car` が正しく動作しなくなった」といったことは良くあります。

たとえば、そのとき手直しをしたのが `create_handle` と `create_interior` と `create_car` だったとしたらどうでしょうか。

もしも、 `create_car` のテストしかなかったとしたら、原因の絞りこみにかなり苦労することになります。
しかし、このそれぞれの工程についてのテストがしっかりあれば、問題の発見は比較的容易でしょう。

プログラムの継続的改善は、それを容易にする仕組みがあってこそ実現できるものです。

小さなプログラムからテストコードを積み上げることは、継続的改善を容易にします。
「このプログラムの思い切った改善をしたい」と思ったときに誰でも感じる、あのグッと重力で押しつぶされるかのような精神的な負担感を軽減してくれます。

手直ししたプログラムが正しく動作するのがしないのか、正しく動作しないならばどういう挙動になっているのかということを手軽に確認できるということは、コーディングとテスト実行にかかる肉体的な負担感を軽減します。

小さなプログラムからテストコードを積み上げることは、あなたが自身継続的改善を楽しく取り組めるためにあなたができる最善のことのひとつです。

ユニットテストフレームワーク

ユニットテストフレームワークは、テストコードの作成と実行を支援するためのフレームワークです。
まずは、ユニットテストフレームワークに関連する以下の用語についてまずは知っておいてください。

番号	用語	属性	説明
1	テストローダー	プログラム	テストコードを読み込んでテストスイートを作る。
2	テストランナー	プログラム	テストスイートのテストを順番に実行する。
3	テストコレクション	動作	テストローダーがテストスイートを作る作業。
4	テストスイート	データ	テストケースの集合。
5	テストケース	データ	個々の独立したテスト。
6	アサーション	構文	テストコード内にある、動作検証のための構文。
7	テストレポート	データ	テストの実行結果を含むデータ。

1. テストローダー(Test Loader): テストローダーは、テストコードを読み込むプログラムです。
   コマンドラインで指定された条件にマッチするテストケースを集め、テストスイートを作ります。

2. テストランナー(Test Runner):
   テストランナーは、テストコードを実行するプログラムです。
   テストランナーは、テストローダーが準備したテストスイートを読み込み、順番にテストを実施します。
   テストランナーは、テスト実行後のテスト結果の表示やレポート生成なども行います。

3. テストコレクション(Test Collection):
   テストコレクションは、テストローダーがテストスイートを作ることを指す言葉です。
   テストローダーは、指定されたディレクトリやモジュール内で、指定の条件にマッチしたテストケースを集めてテストスイートを作ります。

4. テストスイート(Test Suite):
   テストスイートは、テストランナーによって実行される複数のテストケースの集合です。

5. テストケース(Test Case):
   テストケースは、個々の独立したテストです。
   テストケースは、テスト対象のユニットが正しく動作することを検証するためのアサーションを含みます。

6. アサーション (assertion):
   アサーションは、テストコードの中で、テスト対象のユニットが正しく動作しているかを検証するためのコードです。
   アサーションは、テストケースの中で使用されます。
   アサーションは、所定の条件が満たされることを検証します。
   条件を満たさない場合は、AssertionError 例外を発生させてそのテストケースの実行を中断し、失敗したものとして扱われます。

7. テストレポート(Test Report):
   テストレポートは、テストの実行結果を含むデータです。
   テストレポートから、どのテスト成功が成功し、どのテストが失敗したのかを確認することができます。
   また、テストの実行にかかった時間等の情報を得ることもできます。 テストレポートは、テストランナーによって生成されます。
   テストレポートは、コンソールで表示されたり、ファイルに出力されたりします。

ref: unittest --- ユニットテストフレームワーク

なお、テストローダーによるテストコレクション、テストランナーによる実行は、通常、ひとつのコマンドで行われます。

pytest のテストコマンドの例:

```
# 特定のパッケージ以下のすべてのテストを実行する:
pytest tests/

# 特定のモジュール内のすべてのテストを実行する:
pytest test_module.py
pytest tests/test_in_dir.py

# 特定のモジュール内の特定のテスト関数を実行する:
pytest test_module.py::test_func
pytest tests/test_in_dir.py::test_func

# 特定のモジュール内の特定のテストクラスを実行する:
pytest test_module.py::TestPyTestClass
pytest tests/test_in_dir.py::TestPyTestClass

#モジュール内の特定のクラスの特定のテストメソッドを実行する:
pytest test_module.py::TestPyTestClass::test_method
pytest tests/test_in_dir.py::TestPyTestClass::test_method
```

unittest のテストコマンドの例:

```
# 特定のパッケージ以下のすべてのテストを実行する:
# discover は、下位のパッケージ、モジュールをすべて探索してテストを実行します
python -m unittest discover tests

# 特定のモジュール内のすべてのテストを実行する:
python -m unittest test_module
python -m unittest tests.test_in_dir

# 特定のモジュール内の特定のテストクラスを実行する:
python -m unittest test_module.TestUnitTestClass
python -m unittest tests.test_in_dir.TestUnitTestClass

#モジュール内の特定のクラスの特定のテストメソッドを実行する:
python -m unittest test_module.TestUnitTestClass.test_method
python -m unittest tests.test_in_dir.TestUnitTestClass.test_method
```

pytest と unittest, doctest, Coverage

pytest と unittest

本講座では、Python でよく使われるユニットテストフレームワークのうち、pytest と unittest を紹介します。

- pytest: https://docs.pytest.org/en/latest/
- unittest: https://docs.python.org/ja/3/library/unittest.html

pytest は、unittest よりもシンプルで、テストコードの記述が簡単です。
そのため、 Python についての知識が浅い人でも簡単にテストコードを書くことができます。
その一方で、習熟すると、抽象度の高い概念を活用した高度なテストコードを書くことも可能です。

なお、ウェブフレームワークの Django では、デフォルトで unittest が使われています。
ですので、 Django プロジェクトについてテストコードを書けるようになるには、一度は unittest を学習する必要があります。

doctest

doctest というテストフレームワークもあります。

doctest は、テストコードをドキュメント(docstring)に埋め込むことができるテストフレームワークです。
(pytest, unittest のようなユニットテストフレームワークとはちょっと趣が異なります)

テストモジュールやテスト関数等を準備しなくて良いので、 pytest, unittest と比べてテストコードの記述が簡単です。

Coverage.py

コードカバレッジ（Code Coverage）という言葉があります。

コードカバレッジは、ソフトウェアのテストの品質やカバレッジを測定するための指標のひとつです。
コードカバレッジは、ソフトウェアのソースコードのどの部分がテストされ、どの部分がテストされていないかを示します。

python では、ユニットテストのカバレッジを測定するためのツールとして、Coverage.py を使うことができます。

本講座の進め方

本講座では、最初に、 pytest の使い方について学びます。
次に、 unittest の使い方について学びます。
さらに、 doctest の使い方について学びます。

最後に、 Coverage の使い方について学びます。