# ユニットテスト

## 本講座受講にあたって必要なスキルレベル

- やや複雑なロジックの Python コードを書けること。
    - ただし、ロジックに多少破綻があっても良いです。それなりに動くものを書ききれる力があればOKです。
      テストコードの書き方を学ぶことは、そういうスキルレベルの方にとってとても有益です。
- pytest は関数だけで書けます
    - とはいえ、関数のデコレータやクラスについての知識もあるとコーディングの幅が広がります。
- unittest はクラスを使ってテストを書くので、メソッドの書き方についての知識が最低限必要です
    - インスタンスメソッドとクラスメソッドの違いが分かるくらいであればさらに望ましいです。
      unittest について学ぶことは、クラスについての理解を深めるのにもとても有効でしょう。

## 本講座のドキュメント:

docs ディレクトリにある以下の .md ファイルで順に学習してください。

- 10_zodiac_py.md
- 20_about.md
- 30_pytest.md
- 40_unittest.md
- 50_coverage.md
- 60_doctest.md

なお、 上記ドキュメントをそれぞれPDFにしたものと、これらをすべてまとめてPDFにしたものも用意してあります。
/docs/pdf/ ディレクトリに置いてあります。

# 本講座でのテスト対象のプログラムと、解決したい課題

## zodiac.py の概要

ユニットテストについての解説に入る前に、本講座での主たるテスト対象たる `zodiac.py` の仕様について説明します。
( `__doc__` でモジュールやクラス、関数等についての情報を docstring から取得できるので、これも活用してください)

誕生日から、その誕生日の星座の名前を返すプログラムを作りました。
ファイル `zodiac.py` です。

誕生日からその誕生日の星座を返すプログラムを書くのは案外大変です。
大変な理由は、期間が年をまたぐ星座があるからです。

今回採用した各星座の期間は以下のとおりです。(宗派によって違いはあります)
見て分かるとおり、山羊座は12月22日から1月19日までです。

| 星座名 | 開始日 | 最終日 | 年をまたぐか？ |
|---|---|---|---|
| 水瓶座 | 1月20日 | 2月18日 | |
| 魚座 | 2月19日 | 3月20日 | |
| 牡羊座 | 3月21日 | 4月19日 | |
| 牡牛座 | 4月20日 | 5月20日 | |
| 双子座 | 5月21日 | 6月20日 | |
| 蟹座 | 6月21日 | 7月22日 | |
| 獅子座 | 7月23日 | 8月22日 | |
| 乙女座 | 8月23日 | 9月22日 | |
| 天秤座 | 9月23日 | 10月22日 | |
| 蠍座 | 10月23日 | 11月21日 | |
| 射手座 | 11月22日 | 12月21日 | |
| 山羊座 | 12月22日 | 1月19日 | 年をまたぐ |

とはいえ、いちおうきちんと動作しそうなプログラムを作ることができました。
以下の `get_zodiac_sign_name` です。

```
from zodiac import get_zodiac_sign_name

result = get_zodiac_sign_name(month=1, day=1)
print(result)  # 山羊座

result = get_zodiac_sign_name(month=1, day=25)
print(result)  # 水瓶座
```

今回ユニットテストを書く対象は、この関数 `get_zodiac_sign_name` と、これが内部で呼び出している関数など、関連のすべてのプログラムです。

# 実装

## おおまかな方針

この関数 `get_zodiac_sign_name` の参照元として使えるデータとして、以下があります。
これは、星座の名前と、その星座の最終日を CSV 形式で記述したものです。

resouces/zodiac.csv

```
星座名,月,日
山羊座,1,19
水瓶座,2,18
魚座,3,20
牡羊座,4,19
牡牛座,5,20
双子座,6,20
蟹座,7,22
獅子座,8,22
乙女座,9,22
天秤座,10,22
蠍座,11,21
射手座,12,21
```

まずは、このCSVファイルを読みだし、その情報を元にして、まずは以下のような辞書を作ります。

```
zodiac_part_dict = {
    "山羊座": {"month": 1, "day": 19},
    "水瓶座": {"month": 2, "day": 18},
    "魚座": {"month": 3, "day": 20, },
    "牡羊座": {"month": 4, "day": 19, },
    "牡牛座": {"month": 5, "day": 20, },
    "双子座": {"month": 6, "day": 20, },
    "蟹座": {"month": 7, "day": 22, },
    "獅子座": {"month": 8, "day": 22, },
    "乙女座": {"month": 9, "day": 22, },
    "天秤座": {"month": 10, "day": 22, },
    "蠍座": {"month": 11, "day": 21, },
    "射手座": {"month": 12, "day": 21, },
}
```

そして、上記の辞書を元にして、以下のような辞書を作ります。

```
zodiac_full_dict = {
    '山羊座': {'from': {'month': 12, 'day': 22}, 'to': {'month': 1, 'day': 19}},
    '水瓶座': {'from': {'month': 1, 'day': 20}, 'to': {'month': 2, 'day': 18}},
    '魚座': {'from': {'month': 2, 'day': 19}, 'to': {'month': 3, 'day': 20}},
    '牡羊座': {'from': {'month': 3, 'day': 21}, 'to': {'month': 4, 'day': 19}},
    '牡牛座': {'from': {'month': 4, 'day': 20}, 'to': {'month': 5, 'day': 20}},
    '双子座': {'from': {'month': 5, 'day': 21}, 'to': {'month': 6, 'day': 20}},
    '蟹座': {'from': {'month': 6, 'day': 21}, 'to': {'month': 7, 'day': 22}},
    '獅子座': {'from': {'month': 7, 'day': 23}, 'to': {'month': 8, 'day': 22}},
    '乙女座': {'from': {'month': 8, 'day': 23}, 'to': {'month': 9, 'day': 22}},
    '天秤座': {'from': {'month': 9, 'day': 23}, 'to': {'month': 10, 'day': 22}},
    '蠍座': {'from': {'month': 10, 'day': 23}, 'to': {'month': 11, 'day': 21}},
    '射手座': {'from': {'month': 11, 'day': 22}, 'to': {'month': 12, 'day': 21}}
}
```

これで準備ができました。

あとは、誕生日の日付を受け取ると、変数 `zodiac_full_dict` を調べて星座を返す関数を作ります。

## 具体的な実装

上記の方針に基づいた実装を行いました。
`zodiac.py` は、以下のような関数群でを含むモジュールです。

| 項目 | 引数 | 概要 |
|---|---|---|
| `get_zodiac_part_dict` | | resources/zodiac.csv から、<br>データを取得する |
| `get_first_month_day_of_zodiac_sign` | `month` , `zodiac_part_dict` | 各星座の最終日を含む月から、<br>その星座の最初の日を返す |
| `create_zodiac_full_dict` | `zodiac_part_dict` | 各星座の期間を辞書型で返す |
| `get_zodiac_sign_name_dict` | `month` , `day` , `zodiac_full_dict` | 日付と辞書を受け取り、<br>星座を示す文字列を返す |
| `get_zodiac_sign_name` | `month` , `day` | 日付を受け取り、<br>星座を示す文字列を返す |

`get_zodiac_part_dict` , `create_zodiac_full_dict` , `get_zodiac_sign_name_dict` は、 `get_zodiac_sign_name` から呼び出される関数です。
`get_first_month_day_of_zodiac_sign` は、 `create_zodiac_full_dict` から呼び出される関数です。

つまり、全体的な処理の流れは以下のようになります。

- `get_zodiac_sign_name` は、誕生日の月と日を受け取る。
  - `get_zodiac_part_dict` を実行して、各星座の最終日だけの辞書を取得する。
  - `create_zodiac_full_dict` を実行して、各星座の開始日と最終日の辞書を取得する。
    - `get_first_month_day_of_zodiac_sign` を実行して、各星座の開始日を取得する。
  - `get_zodiac_sign_name_dict` を実行して、誕生日に相当する星座を取得する。

## 解決したい課題 - ユニットテストを書きたい理由

ここまでに紹介したような実装のプログラムを書いて、最終的には以下のプログラムを実行すれば求める値を得られるようになったようです。

```
from zodiac import get_zodiac_sign_name

result = get_zodiac_sign_name(month=1, day=1)
print(result)  # 山羊座

result = get_zodiac_sign_name(month=1, day=25)
print(result)  # 水瓶座
```

上に示したとおり、この関数は、少なくとも1月1日と1月25日については正しく動作してるようです。

しかし、ほかの日付については、正しく動作しているかどうかはわかりません。

実際、「ほぼ間違いなく動作するだろう」という確信を持つには、365日すべてとは言わずとも、最低でも以下の類いの日付についてテストする必要があるでしょう。

| テストの趣旨 | 日付の例 | 期待する結果 |
|---|---|---|
| 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 | 山羊座 |
| 任意の星座の初日について動作確認をする | 1月20日 | 水瓶座 |
| 任意の星座の中日について動作確認をする | 1月25日 | 水瓶座 |
| 任意の星座の最終日について動作確認をする | 2月18日 | 水瓶座 |
| 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 | 山羊座 |

しかし、これらすべての日付について正しく動作するかどうかを手動で確認するのはなかなか面倒です。

また、一度動作確認したとしても、関数の内部仕様の変更等をした場合は、再度すべての日付について確認しなくてはなりません。

とはいえ、手直しの都度手動で動作確認するとなると、かなりの手間です。
どうしても、抜け漏れも発生してしてしまうでしょう。

ユニットテストは、このような手動での動作確認の手間を省き、抜け漏れを防ぐためのものです。

たとえば、 `get_zodiac_sign_name` やこれが呼び出している関数のそれぞれについて、正しく動作するかどうかを網羅的に検証するテストコードを書いておきます。

すると、関数の動作確認は、テストコードを実行するだけで簡単にできるようになります。

# 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。

# pytest

Pytest 公式ドキュメント

## インストール

pytest のインストールは、pip で行います。

```
pip install pytest
```

## pytest が実行するテストの要件

pytest が実行するテストは、以下の要件を満たさなくてはなりません(*1)。

| 項目 | 要件 | 例 |
| --- | --- | --- |
| テストモジュール | `test_` で始まるモジュール名にする | `test_example.py` |
| テスト関数 | `test` で始まる関数名にする | `def test_function_name()` |
| テストクラス | `Test` で始まるか終わるクラス名にする(*2) | `class TestExampleFuncClass`, `class ExampleFuncClassTest` |
| テストメソッド | `test` で始まるメソッド名にする | `def test_method_name()` |

> (*1) 後で紹介する `unittest.TestCase` のテストも実行できます。
> (*2) クラス名については、先頭が `Test` からはじまるものに統一するのが可読性も高く好ましいでしょう。

## アサーション

pytest では、アサーションに `assert` 文を使います。
`assert` 文は、条件式が `False` の場合に、 `AssertionError` を発生させます。

`assert` 文は、Python のビルトインステートメントです。
pytest に特有の文ではありません。

```
assert [条件式]
```

`assert` 文の基本的な使用例を紹介します。

```
assert 1 == 1  # エラーは raise されない
assert 1 == 2  # エラーが raise される

assert True  # エラーは raise されない
assert False  # エラーが raise される

assert len([1, 2, 3]) == 3  # エラーは raise されない
assert len([1, 2, 3]) == 4  # エラーが raise される

assert []  # エラーが raise される
assert None  # エラーが raise される

none_var = None
assert none_var is None  # エラーは raise されない
assert none_var is not None  # エラーが raise される
```

### 例外のテスト記述方法

`pytest.raises` を使うと、例外のテストを記述できます。
`pytest.raises` は、通常、 `with` 文と組み合わせて使います。

```
def test_get_zodiac_sign_name_raise():
    """ with を使った 例外テストの書き方 """
    with pytest.raises(ValueError):
        get_zodiac_sign_name(13, 31)
```

以下に示すように `with` 文を使わない書き方もできます。
しかし、どちらかというと、 `with` 文を使った書き方が好まれます。

```
def test_get_zodiac_sign_name_raise_not_with():
    """ with を使わない 例外テストの書き方 """
    pytest.raises(ValueError, get_zodiac_sign_name, 13, 31)
```

## コマンドラインからの実行

pytest は、以下のコマンドで実行できます。

```
pytest
```

> `python -m pytest` でも構いません。

`pytest` コマンドで呼び出されるテストランナーは、主に、以下のような処理を行います。

1. カレントディレクトリからテストファイルを探し、自動的にテストコレクションを行います。
2. テストコレクションでは、pytest の命名規則に従ったテストファイル、テスト関数、テストクラス、テストメソッドが自動的に検出されます。
3. 検出されたテストを実行し、テスト結果を表示します。
4. テスト結果の詳細なレポートを表示します。

テスト対象(テストスイート)を絞りこんで実行することもできます。

```
# 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
```

## テストレポート

テストの実行結果は、以下の例のように表示されます。

以下は、すべてのテストが成功した場合の表示例です。

```
(venv) PS > pytest
========================== test session starts ============================
platform win32 -- Python 3.11.1, pytest-7.3.1, pluggy-1.0.0
rootdir: D:\project_dir
collected 64 items

test_module.py ...                                                   [  4%]
tests\test_in_dir.py ...                                             [  9%]
tests\test_pytests\test_1_func.py ..............                     [ 32%]
tests\test_pytests\test_2_class.py ..................                [ 59%]
tests\test_unittests\test_unittest.py .............                  [ 79%]
tests\test_unittests\test_advanced.py .............                  [100%]

============================ 64 passed in 0.11s ============================
```

以下は、失敗したテストがあった場合の表示例です。

```
(venv) PS D:\project_dir> pytest
========================== test session starts ===========================
platform win32 -- Python 3.11.1, pytest-7.3.1, pluggy-1.0.0
rootdir: D:\project_dir
collected 64 items

test_module.py ...                                                  [  4%]
tests\test_in_dir.py ...                                            [  9%]
tests\test_pytests\test_1_func.py .......F.FF....                   [ 32%]
tests\test_pytests\test_2_class.py ................                 [ 59%]
tests\test_unittests\test_unittest.py ............                  [ 79%]
tests\test_unittests\test_advanced.py ............                  [100%]

================================ FAILURES ================================
___________ test_get_zodiac_sign_name_dict_last_day_of_capricorn ___________

    def test_get_zodiac_sign_name_dict_last_day_of_capricorn():
        zodiac_part_data = get_zodiac_part_dict()
        zodiac_full_dict = create_zodiac_full_dict(zodiac_part_data)

        result = get_zodiac_sign_name_dict(1, 19, zodiac_full_dict)
>       assert result == 'ギョーザ'
E       AssertionError: assert '山羊座' == 'ギョーザ'
E         - ギョーザ
E         + 山羊座

tests\test_pytests\test_1_func.py:83: AssertionError
_____________ test_get_zodiac_sign_name_dict_mid_day_of_aquarius _____________

    def test_get_zodiac_sign_name_dict_mid_day_of_aquarius():
        zodiac_part_data = get_zodiac_part_dict()
        zodiac_full_dict = create_zodiac_full_dict(zodiac_part_data)

        result = get_zodiac_sign_name_dict(1, 25, zodiac_full_dict)
>       assert result == '権力の座'
E       AssertionError: assert '水瓶座' == '権力の座'
E         - 権力の座
E         + 水瓶座

tests\test_pytests\test_1_func.py:101: AssertionError
_____________ test_get_zodiac_sign_name_dict_last_day_of_year _____________

    def test_get_zodiac_sign_name_dict_last_day_of_year():
        zodiac_part_data = get_zodiac_part_dict()
        zodiac_full_dict = create_zodiac_full_dict(zodiac_part_data)

        result = get_zodiac_sign_name_dict(12, 25, zodiac_full_dict)
>       assert result == '新宿ミラノ座'
E       AssertionError: assert '山羊座' == '新宿ミラノ座'
E         - 新宿ミラノ座
E         + 山羊座
```

```
tests\test_pytests\test_1_func.py:110: AssertionError
========================= short test summary info ==========================
FAILED
tests/pytests/test_fixture_basic.py::test_get_zodiac_sign_name_dict_last_day_of_capricorn
- AssertionError: assert '山羊座' == 'ギョーザ'
FAILED
tests/pytests/test_fixture_basic.py::test_get_zodiac_sign_name_dict_mid_day_of_aquarius -
AssertionError: assert '水瓶座' == '権力の座'
FAILED
tests/pytests/test_fixture_basic.py::test_get_zodiac_sign_name_dict_last_day_of_year -
AssertionError: assert '山羊座' == '新宿ミラノ座'
====================== 3 failed, 61 passed in 0.15s ======================
```

# やや高度な手法

## @pytest.fixture

`@pytest.fixture` を使うと、テスト関数の前後で行う処理を定義できます。
複数のテスト関数で共通の前処理、後処理を実装したい場合に便利です。

> 後処理の例としては、以下のようなものが考えられます。
>
> - テスト関数内で作成したファイルを削除する
> - テスト関数内でレコード編集したデータベースのロールバックを行う

### 基本的な処理の流れ

実行例を見ると、処理の流れが分かりやすいかもしれません。
以下のテストコードを実行してみましょう。

tests/test_pytests/test_fixture_basic.py

```
import pytest


@pytest.fixture
def setup():
    print('\nsetup が前処理を開始します')
    # ここでテスト関数の前処理を行う

    yield  # yield は「処理を中断して呼び出し元に戻る」という意味の文です

    print('\nsetup が後処理を開始します')
    # ここでテスト関数の後処理を行う


def test_function1(setup):
    print('test_function1 内部の処理を開始します')
    assert 1 == 1
    print('test_function1 内部の処理が終了しました')


def test_function2(setup):
    print('test_function2 内部の処理を開始します')
    assert 2 == 2
    print('test_function2 内部の処理が終了しました')
```

以下では、 pytest の実行時に `-s` オプションをつけ、print 文の出力を表示しています。

```
(venv) PS D:\project_dir> pytest tests/pytests/test_fixture_basic.py -s
================================= test session starts =================================
platform win32 -- Python 3.11.1, pytest-7.3.1, pluggy-1.0.0
rootdir: D:\project_dir\unit_test_samples
collected 2 items

tests\test_pytest\test_fixture_basic.py
setup が前処理を開始します
test_function1 内部の処理を開始します
test_function1 内部の処理が終了しました
.
setup が後処理を開始します

setup が前処理を開始します
test_function2 内部の処理を開始します
test_function2 内部の処理が終了しました
.
setup が後処理を開始します


================================== 2 passed in 0.01s ==================================
```

なお、 `@pytest.fixture` の後処理は、呼び出し元のテスト関数が assertion 以外の理由で失敗した場合にも実行されます。

以下は、実際のテストコードの例です。

tests/test_pytest/test_deco_func.py

```
import pytest

from zodiac import (
    get_zodiac_sign_name_dict, get_zodiac_part_dict, create_zodiac_full_dict
)


@pytest.fixture
def setup():
    """ get_zodiac_sign_name_dict テスト関数が呼び出す fixture """
    # テストメソッドの前処理がある場合はここに記述します
    zodiac_part_dict = get_zodiac_part_dict()
    zodiac_full_dict = create_zodiac_full_dict(zodiac_part_dict)

    yield zodiac_full_dict

    # テストメソッドの後処理がある場合はここに記述します


def test_get_zodiac_sign_name_dict_first_day_of_year(setup):
    """ 年初の山羊座の最終日前についてテスト """
    result = get_zodiac_sign_name_dict(1, 1, setup)
    assert result == '山羊座'


def test_get_zodiac_sign_name_dict_last_day_of_capricorn(setup):
    """ 山羊座の最終日についてテスト """
    result = get_zodiac_sign_name_dict(1, 19, setup)
    assert result == '山羊座'


def test_get_zodiac_sign_name_dict_first_day_of_aquarius(setup):
    """ 水瓶座の開始日についてテスト """
    result = get_zodiac_sign_name_dict(1, 20, setup)
    assert result == '水瓶座'


def test_get_zodiac_sign_name_dict_mid_day_of_aquarius(setup):
    """ 水瓶座の中間日についてテスト"""
    result = get_zodiac_sign_name_dict(1, 25, setup)
    assert result == '水瓶座'


def test_get_zodiac_sign_name_dict_last_day_of_year(setup):
    """ 年末の射手座最終日以降についてテスト """
    result = get_zodiac_sign_name_dict(12, 25, setup)
    assert result == '山羊座'


def test_get_zodiac_sign_name_dict_raise(setup):
    """ 不正な日付で例外が発生することを確認する """
```

```
    with pytest.raises(ValueError):
        get_zodiac_sign_name_dict(13, 31, setup)
```

#### `yield` 文で値を返す

`yield` 文では、呼び出し元たるテスト関数に任意のオブジェクトを渡すこともできます。
この方法は、関数ベースのテストで使います。

> クラスベースのテストでは、インスタンス変数 `self` に属性を追加できるので、このやり方を使う必要はありません。
> すぐあとに `@pytest.fixture(autouse=True)` についての解説のところで具体的なやり方を述べます。

tests/test_pytest/test_deco_func.py

```
import pytest
from zodiac import (
    get_zodiac_sign_name_dict, get_zodiac_part_dict, create_zodiac_full_dict
)


@pytest.fixture
def setup():
    # テストメソッドの前処理
    zodiac_part_dict = get_zodiac_part_dict()
    zodiac_full_dict = create_zodiac_full_dict(zodiac_part_dict)

    yield zodiac_full_dict  # yield でオブジェクト zodiac_dict を渡します

    # テストメソッドの後処理がある場合はここに記述します


def test_get_zodiac_sign_name_dict_first_day_of_year(setup):
    """ 年初の山羊座の最終日前についてテスト """
    result = get_zodiac_sign_name_dict(1, 1, setup)
    assert result == '山羊座'


def test_get_zodiac_sign_name_dict_last_day_of_capricorn(setup):
    """ 山羊座の最終日についてテスト """
    result = get_zodiac_sign_name_dict(1, 19, setup)
    assert result == '山羊座'


def test_get_zodiac_sign_name_dict_first_day_of_aquarius(setup):
    """ 水瓶座の開始日についてテスト """
    result = get_zodiac_sign_name_dict(1, 20, setup)
    assert result == '水瓶座'
```

### @pytest.fixture(autouse=True)

テストクラス内での利用限定ですが、 `@pytest.fixture(autouse=True)` というものもあります。
`@pytest.fixture(autouse=True)` を使うと、テストクラス内のすべてのテストメソッドの前後で実行される処理を定

義できます。

呼び出し元のテストメソッドに値を渡すためには、任意のタイミングでインスタンス変数 `self` に属性を追加すればOKです。
ですので、前述のとおり、関数ベースのテストのように `yield` 文でオブジェクトを渡す必要はありません。

tests/test_pytest/test_deco_class.py

```
import pytest
from zodiac import get_zodiac_part_dict, get_first_month_day_of_zodiac_sign


class TestFixtureClass:
    """ @pytest.fixture(autouse=True) のサンプル
    モジュール test_deco_class のコードにコメントを追加しています"""

    # autouse=True が指定されると、このクラス内の各テストメソッドが実行されるたびに、
    # このフィクスチャが自動的に呼び出されます
    @pytest.fixture(autouse=True)
    def setup(self):
        # テストメソッドの前処理
        print('setup が前処理を開始します')
        self.zodiac_part_dict = get_zodiac_part_dict()

        yield

        # テストメソッドの後処理がある場合はここに記述します
        print('setup が後処理を開始します')

    def test_1(self):
        print('test_1 内部の処理を開始します')
        result = get_first_month_day_of_zodiac_sign(1, self.zodiac_part_dict)
        assert result == {'month': 12, 'day': 22}
        print('test_1 内部の処理が終了しました')

    def test_2(self):
        print('test_2 内部の処理を開始します')
        result = get_first_month_day_of_zodiac_sign(2, self.zodiac_part_dict)
        assert result == {'month': 1, 'day': 20}
        print('test_2 内部の処理が終了しました')

    def test_3(self):
        print('test_3 内部の処理を開始します')
        result = get_first_month_day_of_zodiac_sign(12, self.zodiac_part_dict)
        assert result == {'month': 11, 'day': 22}
        print('test_3 内部の処理が終了しました')

    def test_4(self):
        """ raise ValueError('Invalid month') """
        print('test_4 内部の処理を開始します')
        with pytest.raises(ValueError):
            get_first_month_day_of_zodiac_sign(0, self.zodiac_part_dict)
            print('test_4 内部の処理が終了しました')
```

```
(venv) PS D:\project_dir> pytest tests/pytests/test_deco_class.py::TestFixtureClass -s
================================ test session starts ================================
platform win32 -- Python 3.11.1, pytest-7.3.1, pluggy-1.0.0
rootdir: D:\project_dir\unit_test_samples
collected 4 items

tests\test_pytest\test_deco_class.py setup が前処理を開始します
test_1 内部の処理を開始します
test_1 内部の処理が終了しました
.setup が後処理を開始します

setup が前処理を開始します
test_2 内部の処理を開始します
test_2 内部の処理が終了しました
.setup が後処理を開始します

setup が前処理を開始します
test_3 内部の処理を開始します
test_3 内部の処理が終了しました
.setup が後処理を開始します

setup が前処理を開始します
test_4 内部の処理を開始します
.setup が後処理を開始します



================================= 4 passed in 0.01s =================================
```

## @pytest.mark.parametrize

`@pytest.mark.parametrize` を使うと、複数のテストを1つのテストメソッドで記述できます。
同じ構造のテストを複数実行する場合等に使います。

tests/test_pytest/test_deco_class.py

```
import pytest

from zodiac import (
    get_zodiac_sign_name_dict, get_zodiac_part_dict, create_zodiac_full_dict
)


class TestParametrize:
    """ @pytest.mark.parametrize のサンプル """

    # 同じ構造のテストを複数実行する場合は、以下のような抽象化された書き方もできます。
    # 第一引数では、 ["month", "day", "expected"] のようなリストを渡すことも可能です。
    @pytest.mark.parametrize("month, day, expected", [
        (1, 1, '山羊座'),
        (1, 19, '山羊座'),
        (1, 20, '水瓶座'),
        (1, 25, '水瓶座'),
        (12, 25, '山羊座'),
    ])
    def test_mark_parametrized_str(self, month, day, expected):
        """ 様々な日付について連続的にテスト
        年初の山羊座の最終日前   :  1月 1日
        山羊座の最終日           :  1月19日
        水瓶座の開始日           :  1月20日
        水瓶座の中間日           :  2月 9日
        年末の射手座の最終日以降 : 12月25日
        """
        zodiac_part_dict = get_zodiac_part_dict()
        zodiac_full_dict = create_zodiac_full_dict(zodiac_part_dict)

        result = get_zodiac_sign_name_dict(month, day, zodiac_full_dict)
        assert result == expected


class TestGetZodiacSignNameDictParametrizeIterable:
    """ @pytest.mark.parametrize のサンプル(第一引数に iterable を使用) """

    # 第一引数では、 ["month", "day", "expected"] のような iterable を渡すことも可能です。
    @pytest.mark.parametrize(["month", "day", "expected"], [
        (1, 1, '山羊座'),
        (1, 19, '山羊座'),
        (1, 20, '水瓶座'),
        (1, 25, '水瓶座'),
        (12, 25, '山羊座'),
    ])
    def test_mark_parametrized_iterable(self, month, day, expected):
        zodiac_part_dict = get_zodiac_part_dict()
        zodiac_full_dict = create_zodiac_full_dict(zodiac_part_dict)

        result = get_zodiac_sign_name_dict(month, day, self.zodiac_full_dict)
        assert result == expected
```

以下は、すべてのテストが成功した場合の表示例です。

```
(venv) PS D:\project_dir> pytest tests/pytests/test_deco_class.py::TestParametrize -s
================================= test session starts ==================================
platform win32 -- Python 3.11.1, pytest-7.3.1, pluggy-1.0.0
rootdir: D:\project_dir\unit_test_samples
collected 5 items

tests\test_pytest\test_deco_class.py .....

================================== 5 passed in 0.02s ===================================
```

以下は、失敗したテストがあった場合の表示例です。
2つのテストが失敗しています。

```
(venv) PS D:\project_dir> pytest tests/pytests/test_deco_class.py::TestParametrize -s
================================= test session starts ==================================
platform win32 -- Python 3.11.1, pytest-7.3.1, pluggy-1.0.0
rootdir: D:\project_dir\unit_test_samples
collected 5 items

tests\test_pytest\test_deco_class.py .F.F.

======================================= FAILURES =======================================
_______ TestParametrize.test_mark_parametrized_str[1-19-\u30ae\u30e7\u30fc\u30b6] _______

self = <tests.pytests.test_deco_class.TestParametrize object at 0x000002B605CF0710>
month = 1, day = 19, expected = 'ギョーザ'

    @pytest.mark.parametrize("month, day, expected", [
        (1, 1, '山羊座'),
        (1, 19, 'ギョーザ'),
        (1, 20, '水瓶座'),
        (1, 25, '権力の座'),
        (12, 25, '山羊座'),
    ])
    def test_mark_parametrized_str(self, month, day, expected):
        """ 様々な日付について連続的にテスト
        年初の山羊座の最終日前  :  1月 1日
        山羊座の最終日          :  1月19日
        水瓶座の開始日          :  1月20日
        水瓶座の中間日          :  2月 9日
        年末の射手座の最終日以降 : 12月25日
        """
        zodiac_part_dict = get_zodiac_part_dict()
        zodiac_full_dict = create_zodiac_full_dict(zodiac_part_dict)

        result = get_zodiac_sign_name_dict(month, day, zodiac_full_dict)
>       assert result == expected
E       AssertionError: assert '山羊座' == 'ギョーザ'
E         - ギョーザ
E         + 山羊座

tests\test_pytest\test_deco_class.py:120: AssertionError
_______ TestParametrize.test_mark_parametrized_str[1-25-\u6a29\u529b\u306e\u5ea7] _______

self = <tests.pytests.test_deco_class.TestParametrize object at 0x000002B605CF0AD0>
month = 1, day = 25, expected = '権力の座'

    @pytest.mark.parametrize("month, day, expected", [
        (1, 1, '山羊座'),
        (1, 19, 'ギョーザ'),
        (1, 20, '水瓶座'),
        (1, 25, '権力の座'),
        (12, 25, '山羊座'),
    ])
    def test_mark_parametrized_str(self, month, day, expected):
```

```
        """ 様々な日付について連続的にテスト
        年初の山羊座の最終日前  :  1月 1日
        山羊座の最終日          :  1月19日
        水瓶座の開始日          :  1月20日
        水瓶座の中間日          :  2月 9日
        年末の射手座の最終日以降 : 12月25日
        """
        zodiac_part_dict = get_zodiac_part_dict()
        zodiac_full_dict = create_zodiac_full_dict(zodiac_part_dict)

        result = get_zodiac_sign_name_dict(month, day, zodiac_full_dict)
>       assert result == expected
E       AssertionError: assert '水瓶座' == '権力の座'
E         - 権力の座
E         + 水瓶座

tests\test_pytest\test_deco_class.py:120: AssertionError
============================== short test summary info ==============================
FAILED tests/pytests/test_deco_class.py::TestParametrize::test_mark_parametrized_str[1-19-
\u30ae\u30e7\u30fc\u30b6] - AssertionError: assert '山羊座' == 'ギョーザ'
FAILED tests/pytests/test_deco_class.py::TestParametrize::test_mark_parametrized_str[1-25-
\u6a29\u529b\u306e\u5ea7] - AssertionError: assert '水瓶座' == '権力の座'
============================ 2 failed, 3 passed in 0.06s ============================
```

# unittest

Python 公式ドキュメント unittest --- ユニットテストフレームワーク

## インストール

unittest は、Python に標準でバンドルされています。
なので、 `pip` コマンド等の実行なしですぐに使えます。

## unittest が実行するテストの要件

unittest が実行するテストは、以下の要件を満たさなくてはなりません。

| 項目 | 要件 | 例 |
| --- | --- | --- |
| テストモジュール | `test` で始まるモジュール名にする | `test_example.py` |
| テストクラス | `unittest.TestCase` クラスのサブクラスにする(*1) | `class TestExampleFunc(TestCase)` |
| テストメソッド | `test` で始まるメソッド名にする(*2) | `def test_method_name()` |

> unittest は、関数では実行できません
>
> (*1) 可読性および pytest からも利用可能にするために、 `Test` ではじまる名称にするのが良いでしょう (*2) 可読性および pytest からも利用可能にするために、 `test` ではじまる名称にするのが良いでしょう

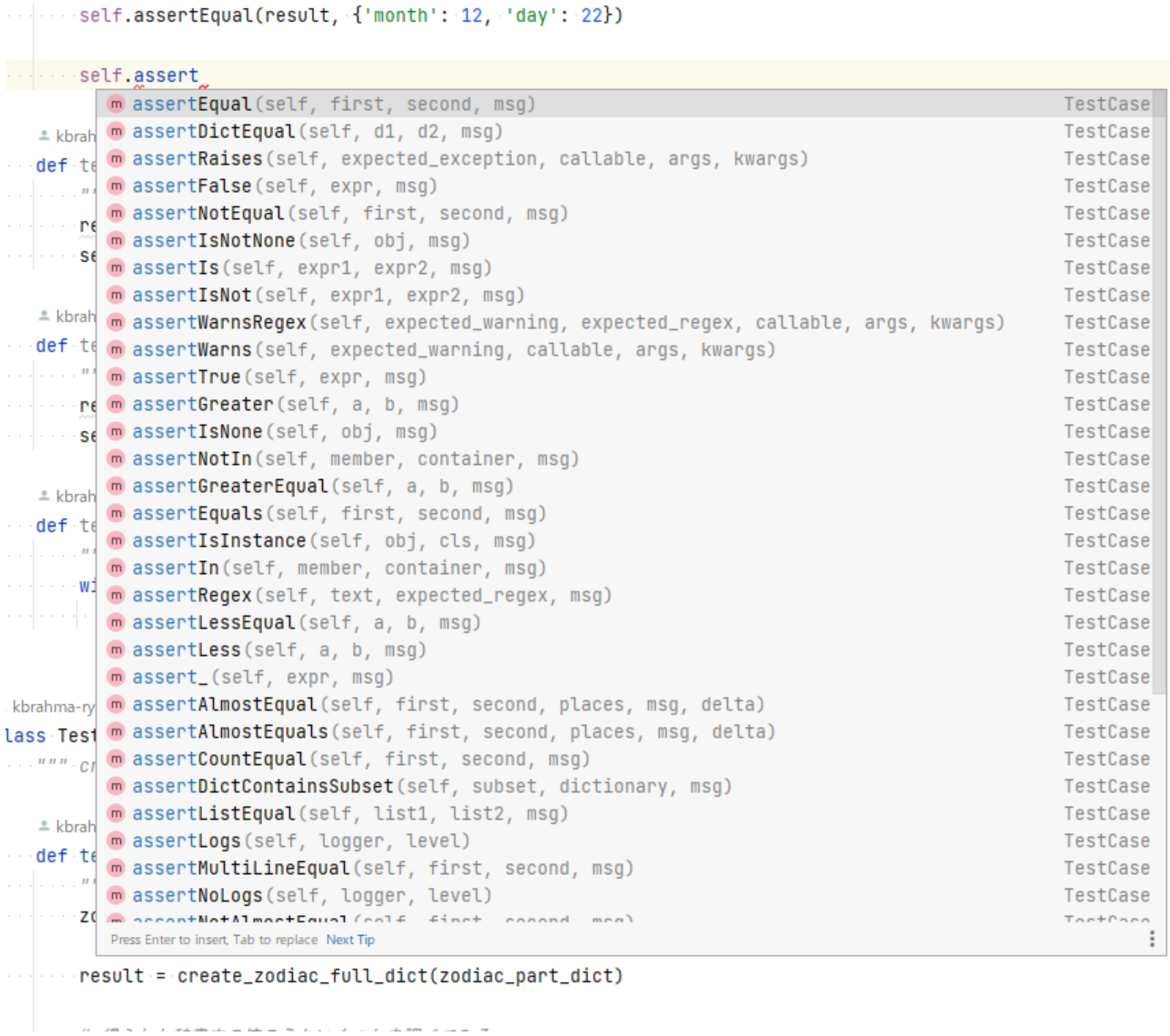
## アサーション

unittest でも、 pytest と同様に、アサーションに `assert` 文を使うことができます。

ですが、 `unittest.TestCase` にはさまざまな assert メソッドが用意されているので、これらを使うのが一般的です。
以下に、その一部を紹介します。

| メソッド名 | 概要 | 使用例 |
| --- | --- | --- |
| assertEqual | 2つの値が等しいことを確認する。 | self.assertEqual(a, b) |
| assertNotEqual | 2つの値が等しいことを確認する。 | self.assertNotEqual(a, b) |
| assertGreater | 第1引数が第2引数より大きいことを確認する。 | self.assertGreater(a, b) |
| assertGreaterEqual | 第1引数が第2引数と同じかより大きいかということを確認する。 | self.assertGreaterEqual(a, b) |
| assertLess | 第1引数が第2引数より小さいことを確認する。 | self.assertLess(a, b) |
| assertLessEqual | 第1引数が第2引数と同じかより小さいということを確認する。 | self.assertLessEqual(a, b) |
| assertTrue | 引数が True であることを確認する。 | self.assertTrue(a) |
| assertFalse | 引数が False であることを確認する。 | self.assertFalse(a) |
| assertIs | 2つの値が同じオブジェクトであることを確認する。 | self.assertIs(a, b) |
| assertIsNot | 2つの値が異なるオブジェクトであることを確認する。 | self.assertIsNot(a, b) |
| assertIsNone | 引数が None であることを確認する。 | self.assertIsNone(a) |
| assertIsNotNone | 引数が None でないことを確認する。 | self.assertIsNotNone(a) |
| assertIn | 第1引数が第2引数に含まれることを確認する。 | self.assertIn(a, b) |
| assertNotIn | 第1引数が第2引数に含まれないことを確認する。 | self.assertNotIn(a, b) |
| assertIsInstance | 第1引数が第2引数のインスタンスであることを確認する。 | self.assertIsInstance(a, b) |
| assertNotIsInstance | 第1引数が第2引数のインスタンスでないことを確認する。 | self.assertNotIsInstance(a, b) |
| assertDictEqual | 2つの辞書が等しいことを確認する。 | self.assertDictEqual(a, b) |
| assertListEqual | 2つのリストが等しいことを確認する。 | self.assertListEqual(a, b) |
| assertTupleEqual | 2つのタプルが等しいことを確認する。 | self.assertTupleEqual(a, b) |
| assertRaises | 所定の Exception が発生することを確認する。 | with self.assertRaises(ValueError): |

IDE のオートコンプリート機能等の支援機能も活用しましょう。




## 例外のテスト記述方法

`assertRaises` を使うと、例外のテストを記述できます。
`assertRaises` は、通常、 `with` 文と組み合わせて使います。

```
class TestGetZodiacSignName(TestCase):
    def test_get_zodiac_sign_name_raise(self):
        """ with を使った 例外テストの書き方 """
        with self.assertRaises(ValueError):
            get_zodiac_sign_name(13, 31)
```

以下に示すように `with` 文を使わない書き方もできます。
しかし、どちらかというと、 `with` 文を使った書き方が好まれます。

```
class TestGetZodiacSignNameNoWith(TestCase):
    def test_get_zodiac_sign_name_raise_now_with(self):
        """  with を使わない 例外テストの書き方 """
        self.assertRaises(ValueError, get_zodiac_sign_name, 13, 31)
```

# コマンドラインからの実行

参考: 公式ドキュメント unittest --- ユニットテストフレームワーク テストディスカバリ
unittest は、以下のコマンドで実行できます。

```
python -m unittest
```

`python -m unittest` コマンドで呼び出されるテストランナーは、主に、以下のような処理を行います。

1. カレントディレクトリからテストファイルを探し、自動的にテストコレクションを行います。
2. テストコレクションでは、unittest の命名規則に従ったテストファイル、テストクラス、テストメソッドが自動的に検出されます。
3. 検出されたテストを実行し、テスト結果を表示します。
4. テスト結果の詳細なレポートを表示します。

テスト対象(テストスイート)を絞りこんで実行することもできます。

`discover` オプションはテスト対象を絞りこむときに使うものです。
もっとも、 `discover` オプションでの複雑な絞りこみの方法を知る必要はありません。
とりあえず、「 `discover` オプションを指定すると、下位のパッケージ、モジュールをすべてテスト対象にする」ということだけ覚えておいてください。

```
# 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
```

# テストレポート

テストの実行結果は、以下の例のように表示されます。

以下は、すべてのテストが成功した場合の表示例です。

```
(venv) PS D:\project_dir> python -m unittest
.......................
----------------------------------------------------------------------
Ran 23 tests in 0.006s

OK
```

以下は、失敗したテストがあった場合の表示例です。

```
(venv) PS D:\project_dir> python -m unittest
.................F.F...
======================================================================
FAIL: test_last_day_of_capricorn
(tests.unittests.unittests.TestGetZodiacSignNameDict.test_last_day_of_capricorn)
山羊座の最終日についてテスト
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\project_dir\tests\test_unittest\test_unittest.py", line 89, in
test_last_day_of_capricorn
    self.assertEqual(result, 'ギョーザ')
AssertionError: '山羊座' != 'ギョーザ'
- 山羊座
+ ギョーザ


======================================================================
FAIL: test_mid_day_of_aquarius
(tests.unittests.unittests.TestGetZodiacSignNameDict.test_mid_day_of_aquarius)
水瓶座の中間日についてテスト
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\project_dir\unit_test_samples\tests\test_unittest\test_unittest.py", line 99,
in test_mid_day_of_aquarius
    self.assertEqual(result, '権力の座')
AssertionError: '水瓶座' != '権力の座'
- 水瓶座
+ 権力の座


----------------------------------------------------------------------
Ran 23 tests in 0.007s

FAILED (failures=2)
```

## unittest.TestCase クラスのそのほかの主要メソッド

以下のメソッドを使って、テストメソッド前後で行う処理を定義できます。
複数のテストメソッドで共通の前処理、後処理を実装したい場合に便利です。

| メソッド名 | メソッドタイプ | 呼び出しのタイミング | 使用目的 |
|---|---|---|---|
| `setUpClass` | クラスメソッド | テストクラス内の<br>最初のテストメソッドが<br>実行される前 | すべてのテストメソッドで<br>使用する変数やファイルの生成に使う |
| `tearDownClass` | クラスメソッド | テストクラス内の<br>全てのテストメソッドが<br>実行された後 | すべてのテストメソッドで<br>生成されたファイルの削除などに使う |
| `setUp` | インスタンス<br>メソッド | 各テストメソッドが<br>実行される前 | 個々のテストメソッドで<br>使用する変数やファイル等の生成に使う |
| `tearDown` | インスタンス<br>メソッド | 各テストメソッドが<br>実行された後 | 個々のテストメソッドで<br>生成されたファイルの削除などに使う |

> 後処理の例としては、以下のようなものが考えられます。
>
> - テストメソッド内で作成したファイルを削除する
> - テストメソッド内でレコード編集したデータベースのロールバックを行う

以下、これらのメソッドの利用にかかる注意点です。

1. `setUpClass` で作ったクラス変数の値をテストメソッド内で変更しない
2. 親クラスで実装されたメソッドを実行する必要がある場合は `super()` で呼び出して実行する
3. `tearDown`, `tearDownClass` は、テストメソッドが異常終了したときも実行される

## 1. `setUpClass` で作ったクラス変数の値をテストメソッド内で変更しない

`setUpClass` で作ったクラス変数の値をテストメソッド内で変更しないようにしましょう。

`setUpClass` は、すべてのテストが実行される前のタイミングで一度だけ実行されるクラスメソッドです。

クラス変数は、その後実行されるすべてのメソッドが参照可能です。
にも関わらずテストメソッド内でこのクラス変数の値を変更してしまうと、他のテストメソッドの実行結果に影響を与えてしまいます。
セットアップ後に値を変更したい変数については、 `setUpClass` ではなく、 `setUp` で作成するようにしましょう。

## 2. 親クラスで実装されたメソッドを実行する必要がある場合は `super()` で呼び出して実行する

上に紹介したいずれのメソッドについても、 `super()` を使って親クラスのメソッドを呼び出すことができます。
`unittest.TestCase` はクラスなので、当然と言えば当然ですね。
`unittest.TestCase` を継承したクラスを使う際はこの点に気をつけてください。

## 3. `tearDown`, `tearDownClass` は、テストメソッドが異常終了したときも実行される

`tearDown`, `tearDownClass` は、テストメソッドが異常終了したときも実行されます。

> ただし、後述のとおり、 `django.test.TestCase` では、テストメソッドが異常終了したときはこれらのメソッドは実行されません。

以下は、これらのメソッドを使ったサンプルコードです。

test_module.py

```
from unittest import TestCase


class TestUnitTestMethodSample(TestCase):
    """ get_zodiac_sign_name_dict のテスト """

    @classmethod
    def setUpClass(cls):
        # super().setUpClass()
        print('\nsetUpClass は一度だけ実行されます')

    def setUp(self):
        print('\nsetUp はテストメソッド毎に実行されます')
        # super().setUp()

    def tearDown(self):
        print('tearDown はテストメソッド毎に実行されます')
        # super().tearDown()

    @classmethod
    def tearDownClass(cls):
        print('\ntearDownClass は一度だけ実行されます')
        # super().tearDownClass()

    def test_1(self):
        print('test_1')

    def test_2(self):
        print('test_2')

    def test_3(self):
        print('test_3')
```

実行結果:

```
(venv) PS D:\project_dir> python -m unittest test_module.TestUnitTestMethodSample

setUpClass は一度だけ実行されます


setUp はテストメソッド毎に実行されます
test_1
tearDown はテストメソッド毎に実行されます
.
setUp はテストメソッド毎に実行されます
test_2
tearDown はテストメソッド毎に実行されます
.
setUp はテストメソッド毎に実行されます
test_3
tearDown はテストメソッド毎に実行されます
.
tearDownClass は一度だけ実行されます

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

以下は、実装例です。

tests/test_unittest/test_unittest.py

```
import unittest
from zodiac import (
    get_zodiac_sign_name_dict, get_zodiac_part_dict, create_zodiac_full_dict)


class TestGetZodiacSignNameDict(unittest.TestCase):
    """ get_zodiac_sign_name_dict のテスト """

    @classmethod
    def setUpClass(cls):
        zodiac_part_data = get_zodiac_part_dict()
        cls.zodiac_full_dict = create_zodiac_full_dict(zodiac_part_data)

    def test_first_day_of_year(self):
        """ 年初の山羊座の最終日前についてテスト """
        result = get_zodiac_sign_name_dict(1, 1, self.zodiac_full_dict)
        self.assertEqual(result, '山羊座')

    def test_last_day_of_capricorn(self):
        """ 山羊座の最終日についてテスト """
        result = get_zodiac_sign_name_dict(1, 19, self.zodiac_full_dict)
        self.assertEqual(result, '山羊座')

    def test_first_day_of_aquarius(self):
        """ 水瓶座の開始日についてテスト """
        result = get_zodiac_sign_name_dict(1, 20, self.zodiac_full_dict)
        self.assertEqual(result, '水瓶座')

    def test_mid_day_of_aquarius(self):
        """ 水瓶座の中間日についてテスト"""
        result = get_zodiac_sign_name_dict(1, 25, self.zodiac_full_dict)
        self.assertEqual(result, '水瓶座')

    def test_last_day_of_year(self):
        """ 年末の射手座最終日以降についてテスト """
        result = get_zodiac_sign_name_dict(12, 25, self.zodiac_full_dict)
        self.assertEqual(result, '山羊座')

    def test_raise(self):
        """ 不正な日付で例外が発生することを確認する """
        with self.assertRaises(ValueError):
            get_zodiac_sign_name_dict(13, 31, self.zodiac_full_dict)

    def test_raise_no_with(self):
        """  with を使わない 例外テストの書き方 """
        self.assertRaises(
            ValueError, get_zodiac_sign_name_dict, 13, 31, self.zodiac_full_dict)
```

以下は、テストメソッドが異常終了したときの流れを示すサンプルです。
テストメソッドが異常終了した場合も、 `tearDown` , `tearDownClass` は実行されます。

tests/unittests/test_raise.py

```
""" unittest が異常終了する場合のサンプル """
import inspect
import unittest


class TestRaiseSample(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        """ すべてのテストが開始する前に一度だけ呼ばれる """
        print(cls.__name__, 'setUpClass')

    def setUp(self):
        """ 各テストが開始する前に呼ばれる """
        print(self.__class__.__name__, 'setUp')

    def tearDown(self):
        """ テストが異常終了しても呼ばれる """
        method_name = inspect.stack()[0][3]
        print(self.__class__.__name__, method_name)

    @classmethod
    def tearDownClass(cls):
        """ テストが異常終了しても呼ばれる """
        print(cls.__name__, 'tearDownClass')

    def test_ok(self):
        """ 成功するテスト """
        method_name = inspect.stack()[0][3]
        print(self.__class__.__name__, method_name)

        self.assertEqual(1, 1)

    def test_raise1(self):
        """ raise して異常終了 """
        method_name = inspect.stack()[0][3]
        print(self.__class__.__name__, method_name)

        a = 3 / 0
        self.assertEqual(a, 0)

    def test_raise2(self):
        """ raise して異常終了 """
        method_name = inspect.stack()[0][3]
        print(self.__class__.__name__, method_name)

        b = "hoge" + 3
        self.assertEqual(b, "hoge3")

    def test_success(self):
        """ 成功するテスト """
        method_name = inspect.stack()[0][3]
        print(self.__class__.__name__, method_name)
```

```
        self.assertEqual(1, 1)
```

```
(venv) PS D:\projectdir> python -m unittest tests.unittests.test_raise
TestRaiseSample setUpClass
TestRaiseSample setUp
TestRaiseSample test_ok
TestRaiseSample tearDown
.TestRaiseSample setUp
TestRaiseSample test_raise1
TestRaiseSample tearDown
ETestRaiseSample setUp
TestRaiseSample test_raise2
TestRaiseSample tearDown
ETestRaiseSample setUp
TestRaiseSample test_success
TestRaiseSample tearDown
.TestRaiseSample tearDownClass

======================================================================
ERROR: test_raise1 (tests.unittests.test_raise.TestRaiseSample)
raise して異常終了
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\projects\lessons\unit_test_samples\tests\unittests\test_raise.py", line 38, in
test_raise1
    a = 3 / 0
ZeroDivisionError: division by zero

======================================================================
ERROR: test_raise2 (tests.unittests.test_raise.TestRaiseSample)
raise して異常終了
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\projects\lessons\unit_test_samples\tests\unittests\test_raise.py", line 46, in
test_raise2
    b = "hoge" + 3
TypeError: can only concatenate str (not "int") to str

----------------------------------------------------------------------
Ran 4 tests in 0.143s

FAILED (errors=2)
```

## やや高度な手法

### unittest.TestCase.subTest

`self.subTest` を使用すると、複数のテストを1つのテストメソッドで記述できます。
同じ構造のテストを複数実行する場合に使います。

pytest の `@pytest.mark.parametrize` デコレータと同様です。

tests/test_unittest/test_advanced.py

```
import unittest

from zodiac import get_zodiac_sign_name


class TestGetZodiacSignName(unittest.TestCase):
    """ get_zodiac_sign_name のテスト """

    def test_values(self):
        """ 様々な日付について連続的にテスト
        年初の山羊座の最終日前   :  1月 1日
        山羊座の最終日           :  1月19日
        水瓶座の開始日           :  1月20日
        水瓶座の中間日           :  2月 9日
        年末の射手座の最終日以降 : 12月25日
        """
        test_cases = [
            (1, 1, '山羊座'),
            (1, 19, '山羊座'),
            (1, 20, '水瓶座'),
            (1, 25, '水瓶座'),
            (12, 25, '山羊座'),
        ]
        for month, day, expected in test_cases:
            with self.subTest(month=month, day=day):
                result = get_zodiac_sign_name(month, day)
                self.assertEqual(result, expected)
```

以下は、失敗したテストがあった場合のテストの実行結果表示例です。
テストメソッドは1つしか記述していませんが、5つのテストが実行され、そのうちの2つについてエラーが出ています。

```
(venv) PS D:\projects_dir> python -m unittest
tests.unittests.test_advanced.TestGetZodiacSignName -v
test_values (tests.unittests.test_advanced.TestGetZodiacSignName.test_values)
様々な日付について連続的にテスト ...
  test_values (tests.unittests.test_advanced.TestGetZodiacSignName.test_values) (month=1,
day=19)
様々な日付について連続的にテスト ... FAIL
  test_values (tests.unittests.test_advanced.TestGetZodiacSignName.test_values) (month=1,
day=25)
様々な日付について連続的にテスト ... FAIL

======================================================================
FAIL: test_values (tests.unittests.test_advanced.TestGetZodiacSignName.test_values)
(month=1, day=19)
様々な日付について連続的にテスト
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\project_dir\unit_test_samples\tests\test_unittest\test_advanced.py", line 91,
in test_values
    self.assertEqual(result, expected)
AssertionError: '山羊座' != 'ギョーザ'
- 山羊座
+ ギョーザ


======================================================================
FAIL: test_values (tests.unittests.test_advanced.TestGetZodiacSignName.test_values)
(month=1, day=25)
様々な日付について連続的にテスト
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\project_dir\unit_test_samples\tests\test_unittest\test_advanced.py", line 91,
in test_values
    self.assertEqual(result, expected)
AssertionError: '水瓶座' != '権力の座'
- 水瓶座
+ 権力の座


----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (failures=2)
```

# doctest

Python 公式ドキュメント doctest --- 対話的な実行例をテストする

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。

テストモジュールやテスト関数等を別途容易しなくて良いので、 pytest, unittest と比べてテストコードの記述が簡単です。
「ドキュメントを書くついでにテストも書いておく」というくらいの感覚で使えます。

## インストール

unittest は、Python に標準でバンドルされています。
なので、 `pip` コマンド等の実行なしですぐに使えます。

## doctest が実行するテストの要件

doctest のテストランナーは、以下の要領でテストを実行します。

- 対話的 Python セッションのように見えるテキストを探し出す
- セッションの内容を実行する
- テキストに書かれている通りに振舞うかを調べる

## doctest の書き方

doctest を含めた docstring の例を示します。

`doctests/doctest_ok.py`

```
def get_last_month(month):
    """ 指定された月の前月を取得する。(簡略番)

    :param month: 月を表す整数
    :return: 月を表す整数

    >>> get_last_month(1)
    12
    >>> get_last_month(2)
    1
    >>> get_last_month(7)
    6
    >>> get_last_month(13)
    Traceback (most recent call last):
        ...
    ValueError: 月は1-12の範囲で指定してください
    """
    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1-12の範囲で指定してください')
    if month > 12:
        raise ValueError('月は1-12の範囲で指定してください')

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


if __name__ == '__main__':
    import doctest

    doctest.testmod()
```

## doctest の実行とレポートの出力

doctest によるテストの実行とテストレポート出力の方法をいくつか紹介します。

1. Python コマンドでファイルを実行する
2. Python シェルで doctest を実行する
3. doctest を unittest から呼び出す

### 1. Python コマンドでファイルを実行する

上記のように `if __name__ == '__main__':` 内で `doctest.testmod` メソッドを呼び出している場合は、 `python` コマンドを実行するだけで doctestを実行できます。

```
python month_funcs/doctest_ok.py
```

テストに成功した場合は何も出力されません。

エラーがあった場合は、以下のようにエラー内容が出力されます。

```
(venv) PS D:\project_dir> python month_funcs/doctest_ng.py
**********************************************************************
File "D:\project_dir\doctests\doctest_ng.py", line 11, in __main__.get_last_month
Failed example:
    get_last_month(7)
Expected:
    1
Got:
    6
**********************************************************************
1 items had failures:
   1 of   4 in __main__.get_last_month
***Test Failed*** 1 failures.
```

もっとも、以下のように `python` コマンドで末尾に `-v` オプションを指定すると、テスト実行過程を詳細に出力できます。
これにより、テストが成功した場合もその過程を把握できますし、少なくとも、「テストが実行された」ということは確認できます。

> `-v` は、英語の verbose (冗長な、詳細な) という単語の頭文字で、コマンドライン引数等でよく登場するオプションです。

```
python month_funcs/doctest_ok.py -v
```

```
(venv) PS D:\project_dir> python month_funcs/doctest_ok.py -v
Trying:
    get_last_month(1)
Expecting:
    12
ok
Trying:
    get_last_month(2)
Expecting:
    1
ok
Trying:
    get_last_month(13)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: 月は1-12の範囲で指定してください
ok
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.get_last_month
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

## 2. Python シェルで doctest を実行する

Python シェルから doctest を実行するには、以下のように `doctest.testmod` を呼び出します。

`doctest.testmod` の受け取る第一引数は、テスト対象のモジュールです。
(引数を省略すると、呼び出し元のモジュール `__main__` がテスト対象になります)

```
import doctest
from month_funcs import doctest_ng

doctest.testmod(doctest_ng)
```

そのほかの引数としては、 `verbose` くらいは覚えておくと良いかもしれません。
`verbose=True` を指定することで、詳細なテスト実行情報が表示されます。

```
import doctest
from month_funcs import doctest_ok

doctest.testmod(doctest_ok, verbose=True)
```

## 3. doctest を unittest から呼び出す

doctest は、unittest から呼び出すこともできます。
これにより、 doctest と unittest を組み合わせてテストを実行することも可能です。

また、複数のモジュールにある doctest を連続的に実行することもできます。

以下で `load_tests` 関数を定義しています。
これは、 unittest がテストモジュールを読みこんでテストスイートを作る際に呼び出される関数です。

この関数の中で、以下の要領で doctest を含むモジュールをテスト対象に追加します。
これにより、モジュール内の doctest も unittest 実行時のテストスイートに追加されます。

Python 公式ドキュメント doctest --- 対話的な実行例をテストする 単体テスト API

month_funcs/test_doctest_by_unittest.py

```
import doctest

from month_funcs import doctest_ok


def load_tests(loader, tests, ignore):
    """ロードされたテストスイートにドックストリングテストを追加する関数

    :param loader: テストローダー
    :param tests: テストスイート
    :param ignore: 無視する要素
    :return: 追加されたテストスイート
    """
    # ドックストリングテスト用のテストスイートを作成する
    test_suites = doctest.DocTestSuite(doctest_ok)

    # ロードされたテストスイートにドックストリングテストを追加する
    tests.addTests(test_suites)

    return tests
```

## doctest のメリットとデメリット

doctest には、以下のような利点があります。

- ドキュメント作成時に手軽にテストコードを書けるので手軽
- ドキュメントとテストコードを一体化できる
- ドキュメントを見ただけで関数やメソッドの使い方が分かる

一方、以下のような欠点もあります。

- ドキュメントが肥大化してしまう
- テスト項目が多い場合は網羅的にテストが書かれているのか分かりにくくなる
- 後述の Coverage のようなツールを利用するには、 unittest との連携が必要になる

doctest を使うか pytest や unittest を使うかは、状況次第です。

# Coverage.py

Coverage.py 公式ドキュメント

## コードカバレッジ（Code Coverage）とは

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、テストの品質と信頼性を測るための指標です。

具体的には、ユニットテストによって、コードのどの行や分岐が実行されたか、または実行されなかったかを調べます。
そして、すべてのコードのうちのどの程度の割合が実行されたかを測定します。この割合の数値は、テストの品質と信頼性についてのある程度の担保となります。

## コードカバレッジ計測ツール Coverage.py

Python プログラムのコードカバレッジを計測するためのツールとして、Coverage.py があります。

この資料では、 Coverage.py の基本的な使用方法と主な機能について説明します。

なお、 Coverage.py のことを、単に Coverage と呼ぶこともあります。
以後、本講座でも、 Coverage という呼び方をします。

## インストール

Coverage のインストールは、pip で行います。

```
pip install coverage
```

## Coverage の使用方法

Coverage は、コマンドラインから実行できます。

以下は基本的な使用方法です。

1. テストを実行しながらカバレッジを計測する

カバレッジを計測するには、以下の例のような書き方でテストランナーを実行します。

```
# pytest を使用する場合の例
coverage run -m pytest

# unitest を使用する場合の例
coverage run -m unittest

# Django のユニットテストを実行する場合の例
coverage run manage.py test
```

すると、 Coverage は、指定されたテストスイートのテストを実行しながらカバレッジデータを収集します。

2. カバレッジレポートを表示する

カバレッジレポートを表示するには、次のコマンドを使用します。

```
coverage report
```

このコマンドは、コンソールにカバレッジレポートを表示します。
各モジュール内のコードのどの程度の割合が実行されたかを測定した結果が表示されます。

以下は、カバレッジレポートの例です。

```
(venv) PS D:\project_dir> coverage report
Name                       Stmts   Miss Branch BrPart  Cover
------------------------------------------------------------
month_funcs\__init__.py        0      0      0      0   100%
month_funcs\compare.py        14      2     10      2    83%
zodiac.py                     43      4     24      2    91%
------------------------------------------------------------
TOTAL                         57      6     34      4    89%
```

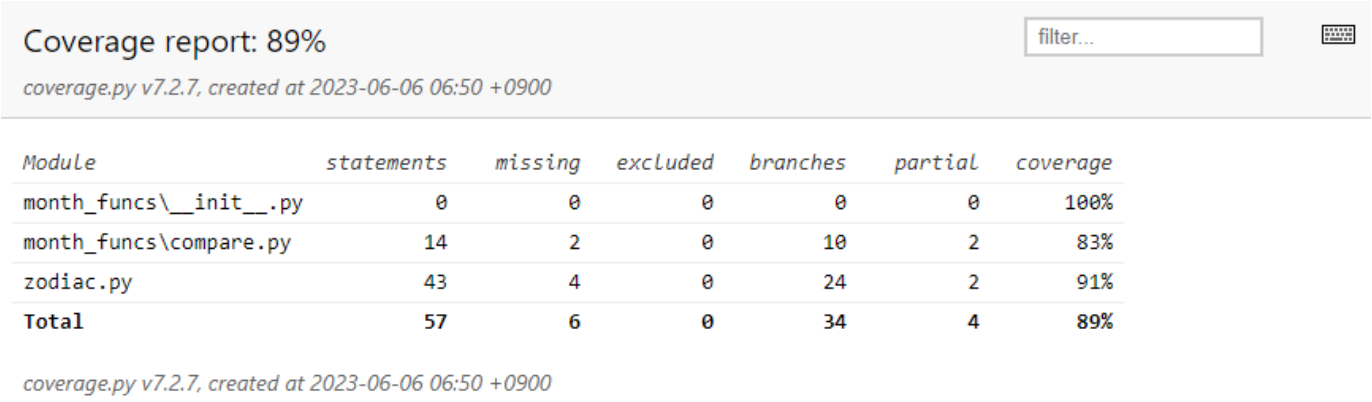
3. HTML形式のカバレッジレポートを生成する

HTML形式のカバレッジレポートを生成するには、次のコマンドを使用します。

```
coverage html
```

このコマンドは、 `htmlcov` ディレクトリにHTMLレポートを生成します。
`index.html` ファイルをウェブブラウザで開いて詳細なカバレッジレポートを表示できます。

以下は、HTMLカバレッジレポートの例です。


Coverage report: 89%

filter...

coverage.py v7.2.7, created at 2023-06-06 06:50 +0900

| Module | statements | missing | excluded | branches | partial | coverage |
|---|---|---|---|---|---|---|
| month_funcs__init__.py | 0 | 0 | 0 | 0 | 0 | 100% |
| month_funcs\compare.py | 14 | 2 | 0 | 10 | 2 | 83% |
| zodiac.py | 43 | 4 | 0 | 24 | 2 | 91% |
| **Total** | **57** | **6** | **0** | **34** | **4** | **89%** |

coverage.py v7.2.7, created at 2023-06-06 06:50 +0900

各モジュールごとの詳細を出力することもできます。
実行されなかった行や、条件分岐のすべてがカバーされなかった行は、以下のように色つきで表示されます。

Coverage for **month_funcs\compare.py**: 83%

14 statements | 12 run | 2 missing | 0 excluded | 2 partial

« prev ^ index » next coverage.py v7.2.7, created at 2023-06-06 06:50 +0900

```
1  def get_last_month(month):
2      """ 指定された月の前月を取得する """
3      if not isinstance(month, int):
4          raise TypeError('月は整数で指定してください')
5      if month < 1:                                          5 →/6
6          raise ValueError('月は1以上で指定してください')
7      if month > 12:
8          raise ValueError('月は12以下で指定してください')
9
10     if month == 1:                                         10 →/11
11         return 12
12     else:
13         return month - 1
14
15
16 def get_last_month_concised(month):
17     """ 指定された月の前月を取得する。(簡略番) """
18     if not 1 <= month <= 12:
19         raise ValueError('月は1-12の範囲で指定してください')
20
21     return 12 if month == 1 else month - 1
```

« prev ^ index » next coverage.py v7.2.7, created at 2023-06-06 06:50 +0900

## .coveragerc ファイル

Coverage の設定は、 `.coveragerc` ファイルによってカスタマイズできます。
`.coveragerc` ファイルでは、特定のファイルやディレクトリの除外、報告されるカバレッジの形式、カバレッジのしきい値などの設定を指定できます。

`.coveragerc` ファイルは、カバレッジの計測を行う対象のプロジェクトのトップレベルディレクトリに置いてください。

以下は、 `.coveragerc` ファイルの例です。

```
[run]
omit = */manage.py
       */migrations/*
       tests/*
       *test_*.py
```

.coveragerc には、以下のような設定項目があるようです。
興味と必要の程度に応じて調べてみてください。

| カテゴリ | キー | 説明 |
|---|---|---|
| [run] | branch | 分岐カバレッジを有効にするかどうかのフラグ |
| | concurrency | 並行処理を使用してテストを実行するかどうかのフラグ |
| | data_file | カバレッジデータの保存先ファイルのパス |
| | parallel | 並行処理を有効にするかどうかのフラグ |
| | source | ソースコードのディレクトリまたはモジュールのパス |
| | include | カバレッジレポートに含めるファイルまたはディレクトリのパターン |
| | omit | カバレッジレポートから除外するファイルまたはディレクトリのパターン |
| | plugins | 使用するカバレッジプラグインの指定 |
| [report] | exclude_lines | レポートから除外する行のパターン |
| | precision | パーセンテージの表示精度 |
| | show_missing | カバレッジレポートで欠損行を表示するかどうかのフラグ |
| [html] | directory | HTMLレポートの出力ディレクトリのパス |
| | title | HTMLレポートのタイトル |
| [xml] | output | XMLレポートの出力ファイルのパス |
| | package | パッケージレベルのカバレッジ情報を含めるパッケージのパス |
| [paths] | source | ソースコードのディレクトリまたはモジュールのパス |
| | data_coverage | カバレッジデータのパス |
| | relative | パスの相対指定を有効にするかどうかのフラグ |

Coverage.py 公式ドキュメント Configuration reference - syntac

## コードカバレッジは、あくまで、「品質のひとつの指標」

コードカバレッジは、あくまで、品質のひとつの指標にしかすぎません。

100%ならば、良いテストコードだとは限りません。
また、100%でなくても、そのことが即座に品質に問題があるということにはなりません。

「ユニットテストとは」の章で紹介した以下のコードについて考えてみましょう。

month_funcs/compare.py

```
    def get_last_month(month):
        """ 指定された月の前月を取得する。

        :param month: 月を表す整数
        :return: 月を表す整数
        """
        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
```

上記のコードでは、分岐は合計5つです。
以下のような5つのテストコードを用意すれば、この関数についてのカバレッジを100%にすることができます。

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
```

ところで、以下ではどうでしょうか。 以下のコードは、上に示した `get_last_month` とほぼ同じ機能を有しています。

month_funcs/compare.py

```
def get_last_month_concised(month):
    """ 指定された月の前月を取得する。(簡略番)

    :param month: 月を表す整数
    :return: 月を表す整数
    """
    if not 1 <= month <= 12:
        raise ValueError('月は1-12の範囲で指定してください')

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

この関数についてのカバレッジを100%にしたければ、以下のような2つのテストコードを用意すれば十分です。
ですが、言うまでもなく、これらのテストだけで十分とは言えないでしょう。

month_funcs/test_last_month_funcs.py

```python
import pytest

from month_funcs.compare import get_last_month_concised


def test_raise():
    """ 例外が発生するケース """
    with pytest.raises(ValueError):
        get_last_month_concised(0)


def test_success():
    """ 戻り値を得られるケース """
    assert get_last_month_concised(2) == 1
```

```
(venv) PS D:\project_dir> coverage report
Name                       Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------
month_funcs\__init__.py        0      0      0      0   100%
month_funcs\compare.py        14      0     10      0   100%
-------------------------------------------------------------
TOTAL                         14      0     10      0   100%
```

コードカバレッジは、あくまで、以下についての調査用のツールでしかありません。

- コードのどの行や分岐が実行されたか、または実行されなかったか
- すべてのコードのうちのどの程度の割合が実行されたか

「テストすべきテストケースのすべてについてテストが実施されたか」ということを判断するには、コードカバレッジだけでは不十分です。

また、コードカバレッジの数字が100%でなくても、そのことが即座に品質に問題があるということにはなりません。
(実際、大規模なプロジェクトでは、100%にすることは困難です)

コードカバレッジは、あくまで、「品質のひとつの指標」にすぎません。
コードの品質担保は、コードカバレッジ以外の方法も含めた総合的な判断で行うべきです。

# django.test.TestCase

Django 公式ドキュメント テストを書いて実行する Django 公式ドキュメント テストツール

Django 公式のユニットテストフレームワークとして、 `django.test.TestCase` があります。

`django.test.TestCase` は、 `unittest.TestCase` を継承しています。
そして、 `unittest.TestCase` に、ウェブフレームワークのテストに便利な機能が追加されています。

本資料は、読者が `unittest.TestCase` を使ったことがあることを前提にしています。

> pytest を使って Django プロジェクトのテストコードを書くこともできますが、そのための準備はやや複雑です。
> pytest を使って Django のテストコードを書くのに必要な手続きを習得するよりも unittest を習得してしまうほうが、学習コストが低く、また応用が効くノウハウになるでしょう。

## テストランナーの機能

### データベースの作成/マイグレーション/削除

Django のテストランナーは、起動時に、データベースの作成とマイグレーションを行います。

Django のテストランナーは、ひとつひとつのテストが終了する都度、テスト用データベースをマイグレーション直後の状態に戻します(ロールバックします)。
これにより、個々のテストは、中身がクリアされた(マイグレーション直後と同様の)状態のデータベースを使えます。
ですので、他のテストの結果に影響を受けない状態でテストを行えます。

そして、Django のテストランナーは、テストが終了すると、データベースを削除します。

> Django のユニットテストでは、データベースのロールバックは自動的に行われます。
> ですので、 `tearDown` メソッドや `tearDownClass` メソッド内でデータベースの初期化作業をする必要はありません。

#### テストデータベースの作成と破棄、所在と権限

テストデータベースは以下のように作成され、そして破棄されます。

SQLite を使っているときは、デフォルトでは、テストにはインメモリのデータベースを使います。
つまり、データベースはメモリ内に作成されるため、ファイルシステムへのアクセスを完全になくすことができます。

PostgreSQL, MySQL 等のデータベースを使用している場合は、テスト用のデータベースがテストの都度作られます。
そして、テスト終了時にこのテスト用のデータベースは破棄されます(*1)

このテストデータベースの名前は、 DATABASES 設定内の `NAME` の値の前に `test_` を付けたものになります。(*2)

> (*1) `python manage.py test` コマンドに `--keepdb` オプションをつけて実行すると、テスト用データベースは破棄されません。
> なので、次回以降のテストをより高速に実行することができます。
>
> (*2) テストデータベースの名前の変更は、 `settings'` の DATABASES で TEST セクションの NAME` キーを書き換えることで可能です。
>
> Django 公式ドキュメント テストを書いて実行する test データベース

Django がデータベースへの接続で利用するユーザがテスト用データベースに対して作成/削除等の権限を有していないとテストが失敗することがあるので注意してください。
この権限の付与手順は、概ね以下のとおりです。

1. テスト用データベースと同名のデータベースをいったん作成する(マイグレーション等は不要)
2. Django が使うデータベースユーザに、 1. で作ったデータベースの適切な権限を付与する

以下は、 PostgreSQL の場合のより詳細な手順例です。

1. Django の設定から、以下を調べる
    - Django が使っているデータベースのデータベース名
    - Django が使っているデータベースユーザ名
2. PostgreSQL に postgres ユーザでログインする
3. Django のデータベース名に test_ という prefix のついた名前のデータベースを作る
4. 3.で作ったデータベースに対する権限を、 Django の設定ファイルに記載のユーザに対して付与する
5. 作成したデータベースを削除する
6. PostgreSQL からログアウトする

#### 1. Django の設定から、Django が使うデータベース名と postgres ユーザのユーザ名を調べる

config.settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'django_mysite_db<',
        'USER': 'django_user',
        'PASSWORD': 'mysiteUserPass123',
        'HOST': 'localhost',
        'PORT': '5432',
        'ATOMIC_REQUESTS': True,
    }
}
```

上記から、以下のことが分かりました。

| 項目 | 値 |
| --- | --- |
| Django が使っているデータベースユーザ名 | `django_user` |
| Django が使っているデータベースのデータベース名 | `django_mysite_db` |
| Django がテストで使うデータベースのデータベース名 | `test_django_mysite_db` |

#### 2. PostgreSQL に postgres ユーザでログインする

```
sudo -u postgres psql
```

3. Django のデータベース名に test_ という prefix のついた名前のデータベースを作る

```
CREATE DATABASE test_django_mysite_db;
```

4. 3.で作ったデータベースに対する権限を、 Django の設定ファイルに記載のユーザに対して付与する

```
GRANT ALL PRIVILEGES ON DATABASE test_django_mysite_db TO django_user;
```

5. 作成したデータベースを削除する

```
DROP DATABASE test_django_mysite_db;
```

6. PostgreSQL からログアウトする

```
\q
```

一度データベースを作成し、適切な権限を付与します。
そうすれば、そのデータベースを削除して以降に Django が同名のデータベースを作ったとしても、もうエラーは発生しません。

# django.test.TestCase.client の機能

## テストクライアント

テストメソッド内では、テスト用の HTTP テストクライアントを利用できます。

このテストクライアントでは、以下の要領で、 `GET` や `POST` などの HTTP メソッドを利用できます。
リクエストの戻り値としてレスポンスを得られます。
また、 View クラスや view 関数がテンプレートに渡した `context` 辞書の値など、様々な関連情報も得られます。

このレスポンスの内容を検証することで、 View の挙動をテストできます。

```
from django.test import TestCase


class TestMethodSample(TestCase):
    def test_get_method_sample(self):
        response = self.client.get('/sample/')
        self.assertEqual(response.status_code, 200)

        messages = list(response.context['messages'])
        self.assertEqual(len(messages), 1)
        self.assertEqual(str(messages[0]), 'ページの表示に成功しました。')

    def test_post_method_sample(self):
        response = self.client.post('/sample/', {'foo': 'bar'})
        self.assertEqual(response.status_code, 302)
```

### Django 公式ドキュメント テストツール テストクライアント

以下のメソッドによって、特定のテストユーザとしてログインすることも可能です。
ログインに成功すれば、ログインユーザとしてウェブページで `GET` や `POST` を行ったときの View の挙動をテストできます。

- `django.test.client.login`
- `django.test.client.force_login`

以下では、未ログイン状態のユーザ、ログイン済のユーザでの挙動の違いをテストする例を示します。

```
from django.contrib.auth import get_user_model
from django.shortcuts import resolve_url
from django.test import TestCase

User = get_user_model()


class TestArticleListViewSample(TestCase):
    """ 未ログインユーザ、ログインユーザでの GET リクエストのテストの例 """

    def test_anonymous(self):
        """ 未ログインユーザとしてGETリクエストを実施 """
        path = resolve_url('log:article_list')
        response = self.client.get(path)

        self.assertEqual(response.status_code, 200)

    def test_authed_user(self):
        """ ログインユーザとしてGETリクエストを実施 """
        user = User.objects.create_user(username='testuser', email='foo@bar.com',
                                        password='testpassword')
        login_result = self.client.login(email=user.email, password='testpassword')
        self.assertTrue(login_result)

        path = resolve_url('log:article_list')
        response = self.client.get(path)

        self.assertEqual(response.status_code, 200)
```

## アサーション

Django 公式ドキュメント テストツール Assertions

`django.test.TestCase` には、 `unittest.TestCase` クラスから継承したもの以外にもウェブフレームワークらしい様々な assert メソッドがあります。
まずは、レスポンス解析用の assert メソッドに慣れていきましょう。

| メソッド名 | 概要 | 例 |
| --- | --- | --- |
| assertContains | レスポンスが指定された要素を含んでいることを確認する | self.assertContains(response, "Hello World") |
| assertNotContains | レスポンスが指定された要素を含んでいないことを確認する | self.assertNotContains(response, "Error") |
| assertHTMLEqual | 2つのHTMLテキストが同じ意味合いになることを確認する | python self.assertHTMLEqual(html1, html2) |
| assertHTMLNotEqual | 2つのHTMLテキストが同じ意味合いにならないことを確認する | python self.assertHTMLNotEqual(html1, html2) |
| assertTemplateUsed | 特定のテンプレートが使用されたことを確認する | self.assertTemplateUsed(response, "template.html") |
| assertTemplateNotUsed | 特定のテンプレートが使用されなかったことを確認する | self.assertTemplateNotUsed(response, "template.html") |
| assertRedirects | レスポンスが指定されたURLにリダイレクトされることを確認する | self.assertRedirects(response, "/redirected/") |

`assertHTMLEqual`, `assertHTMLNotEqual` での `html1`, `html2` は、HTML テキストを文字列で指定します。
Django 公式ドキュメント テストツール assertHTMLEqual

https://github.com/k-brahma/django_photo_diary/blob/main/log/tests/test_views_sample_basic.py

```python
from django.test import TestCase


class TestHTMLEqualSample(TestCase):
    def test_html_equal_1(self):
        # 以下の html1 と html2 は HTMLタグの機能や意味においては等価です。
        html1 = '<p class="my-class" id="my-id">本文</p>'
        html2 = '<P id="my-id" class="my-class" >本文</P>'

        self.assertHTMLEqual(html1, html2)

    def test_html_equal_2(self):
        # 以下の html3 と html4 は HTMLタグの機能や意味においては等価です。
        html1 = '<input type="checkbox" class="my-class" id="my-id" checked>'
        html2 = '<INPUT TYPE="checkbox"  checked="checked" id="my-id" class="my-class">'

        self.assertHTMLEqual(html1, html2)
```

Django の View クラスのテストコードの例を以下に示します。

https://github.com/k-brahma/django_photo_diary/blob/main/log/tests/test_views_sample.py

```python
from django.shortcuts import resolve_url
from django.test import TestCase
from django.contrib.auth import get_user_model

from log.models import Article, Tag

User = get_user_model()


class TestArticleUpdateViewSample(TestCase):
    """
    ArticleUpdateView のテスト

    ページの表示/日記の更新ができるのが投稿者本人または is_staff ユーザのみということを確認する
    """

    @classmethod
    def setUpTestData(cls):
        cls.list_path = resolve_url('log:article_list')

        # テストユーザを作る。この作業は全テストを通じて一度で良いので setUpTestData で行う
        cls.user = User.objects.create_user(username='test', email='foo@bar.com',
                                            password='testpassword')

        # テスト用のタグを作成
        for i in range(1, 5):
            tag = Tag.objects.create(name=f'test_tag{i}', slug=f'test_tag{i}')
            setattr(cls, f'tag{i}', tag)

    def setUp(self):
        # テストの都度改めて article を作成する
        # 投稿編集テストがあるので、 setUpTestData で行うのは不適切
        # (更新された article のままだと他のテストに影響するため)
        self.article = Article.objects.create(title='base_test_title',
                                              body='base_test_body', user=self.user)
        self.article.tags.add(self.tag1, self.tag2)

        # aritcle の pk は生成される都度異なる場合があるので注意(データベース製品による)
        self.path = resolve_url('log:article_update', pk=self.article.pk)

    def result_redirect(self, response):
        """ 権限を持たないユーザが GET/POST でアクセスしたときの処理の検証用メソッド """
        redirect_path = self.list_path
        self.assertRedirects(response, redirect_path)

        messages = list(response.context['messages'])
        self.assertEqual(len(messages), 1)
        self.assertEqual(str(messages[0]), '日記を更新できるのは投稿者と管理者だけです。')

    def result_get_success(self, response):
        """ 権限を持つユーザが GET でアクセスしたときの処理の検証用メソッド """
        self.assertEqual(response.status_code, 200)
```

```
        self.assertTemplateUsed(response, 'log/article_update.html')
        self.assertContains(response, self.article.title)

    def result_post_success(self, response):
        """ 権限を持つユーザが POST で日記の更新を行ったときの処理の検証用メソッド """
        self.assertRedirects(response, self.list_path)

        messages = list(response.context['messages'])
        self.assertEqual(len(messages), 1)
        self.assertEqual(str(messages[0]), '日記を更新しました。')

    def check_article(self, title, body, tags):
        """ post 後の article オブジェクトの状態チェック用メソッド """
        articles = Article.objects.all()
        self.assertEqual(len(articles), 1)
        self.assertEqual(articles[0].title, title)
        self.assertEqual(articles[0].body, body)

        article_tags = articles[0].tags.all()
        self.assertEqual(len(article_tags), 2)
        self.assertIn(tags[0], article_tags)
        self.assertIn(tags[1], article_tags)

    def test_get_anonymous(self):
        """ AnonymousUser は一覧ページにリダイレクトされる """
        response = self.client.get(self.path, follow=True)
        self.result_redirect(response)

    def test_get_another_user(self):
        """ 投稿者本人でなくてスタッフでもない場合は一覧ページにリダイレクトされる """
        another_user = User.objects.create_user(username='another', email='foo2@bar.com',
                                                password='testpassword')
        result = self.client.login(email=another_user.email, password='testpassword')
        self.assertTrue(result)  # ログイン成功しているか確認

        response = self.client.get(self.path, follow=True)
        self.result_redirect(response)

    def test_get_article_user(self):
        """ 投稿者本人の場合は更新ページが表示される """
        result = self.client.login(email=self.user.email, password='testpassword')
        self.assertTrue(result)  # ログイン成功しているか確認

        response = self.client.get(self.path)
        self.result_get_success(response)

    def test_get_is_staff(self):
        """ 投稿者本人でなくてもスタッフの場合は更新ページが表示される """
        another_user = User.objects.create_user(username='another', email='foo2@bar.com',
                                                password='testpassword', is_staff=True)
        result = self.client.login(email=another_user.email, password='testpassword')
        self.assertTrue(result)  # ログイン成功しているか確認
```

```
        response = self.client.get(self.path)
        self.result_get_success(response)

    def test_post_failure_another_user(self):
        """ 投稿者本人でなくてスタッフでもない場合は投稿に失敗し一覧ページにリダイレクトされる """
        another_user = User.objects.create_user(username='another', email='foo2@bar.com',
                                                password='testpassword')
        result = self.client.login(email=another_user.email, password='testpassword')
        self.assertTrue(result)  # ログイン成功しているか確認

        # postメソッドで送信するデータを生成
        data = {
            'title': 'test_title_fail',
            'body': 'test_body_fail',
            'tags': [self.tag3.id, self.tag4.id, ]
        }
        response = self.client.post(self.path, data=data, follow=True)
        self.result_redirect(response)

        # データが更新されていないことを確認
        self.check_article('base_test_title', 'base_test_body', [self.tag1, self.tag2])

    def test_post_success_article_user(self):
        """ 投稿者本人の場合は投稿を更新できる """
        self.client.force_login(self.user)  # ログイン状態にする

        # postメソッドで送信するデータを生成
        data = {
            'title': 'test_title1',
            'body': 'test_body1',
            'tags': [self.tag3.id, self.tag4.id, ]
        }
        response = self.client.post(self.path, data=data, follow=True)

        self.result_post_success(response)

        # データが更新されていることを確認
        self.check_article('test_title1', 'test_body1', [self.tag3, self.tag4])

    def test_post_success_is_staff(self):
        """ 投稿者本人でなくてもスタッフの場合は投稿を更新できる """
        another_user = User.objects.create_user(username='another', email='foo2@bar.com',
                                                password='testpassword', is_staff=True)
        result = self.client.login(email=another_user.email, password='testpassword')
        self.assertTrue(result)  # ログイン成功しているか確認

        # postメソッドで送信するデータを生成
        data = {
            'title': 'test_title2',
            'body': 'test_body2',
            'tags': [self.tag3.id, self.tag4.id, ]
```

```
        }
        response = self.client.post(self.path, data=data, follow=True)

        self.result_post_success(response)

        # データが更新されていることを確認
        self.check_article('test_title2', 'test_body2', [self.tag3, self.tag4])
```

## コマンドラインからの実行

Django 公式ドキュメント テストを書いて実行する テストの実行
Django の unittest を実行するには、以下のコマンドを実行します。

```
python manage.py test
```

テスト対象を絞りこんで実行することもできます。
Django のユニットテストは、 `discover` オプションを指定しないでもパッケージ以下のすべてのテストを再帰的に探索してテストスイートを作ります。その点、 unittest よりも簡単です。

```
# log アプリのテストだけを実行
python manage.py test log

# accounts アプリと log アプリののテストだけを実行
python manage.py test accounts log

# log アプリの tests パッケージ以下にあるテストだけを実行
python manage.py test log.tests

# 1つのテストモジュールだけを実行
python manage.py test log.tests.test_views

# 1つのテストクラスだけを実行
python manage.py test log.tests.test_views.TestArticleUpdateView

# 1つのテストメソッドだけを実行
python manage.py test log.tests.test_views.TestArticleUpdateView.test_get_anonymous
```

## テストレポート

テストの実行結果は、以下の例のように表示されます。

```
(venv) PS D:\django_project> python manage.py test
Found 62 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..............................................................
----------------------------------------------------------------------
Ran 62 tests in 5.327s

OK
Destroying test database for alias 'default'...
```

以下は、失敗したテストがあった場合の表示例です。

```
(venv) PS D:\django_project> python manage.py test
Found 62 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....................................................FF..F..
======================================================================
FAIL: test_get_anonymous
(log.tests.test_views_sample.TestArticleUpdateViewSample.test_get_anonymous)
AnonymousUser は一覧ページにリダイレクトされる
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\django_project\log\tests\test_views_sample.py", line 84, in test_get_anonymous
    self.result_redirect(response)
  File "D:\django_project\log\tests\test_views_sample.py", line 52, in result_redirect
    self.assertEqual(str(messages[0]), '日記を更新できるのは投稿者だけです。')
AssertionError: '日記を更新できるのは投稿者と管理者だけです。' != '日記を更新できるのは投稿者だけ
です。'
- 日記を更新できるのは投稿者と管理者だけです。
?              ----
+ 日記を更新できるのは投稿者だけです。


======================================================================
FAIL: test_get_another_user
(log.tests.test_views_sample.TestArticleUpdateViewSample.test_get_another_user)
投稿者本人でなくてスタッフでもない場合は一覧ページにリダイレクトされる
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\django_project\log\tests\test_views_sample.py", line 94, in
test_get_another_user
    self.result_redirect(response)
  File "D:\django_project\log\tests\test_views_sample.py", line 52, in result_redirect
    self.assertEqual(str(messages[0]), '日記を更新できるのは投稿者だけです。')
AssertionError: '日記を更新できるのは投稿者と管理者だけです。' != '日記を更新できるのは投稿者だけ
です。'
- 日記を更新できるのは投稿者と管理者だけです。
?              ----
+ 日記を更新できるのは投稿者だけです。


======================================================================
FAIL: test_post_failure_another_user
(log.tests.test_views_sample.TestArticleUpdateViewSample.test_post_failure_another_user)
投稿者本人でなくてスタッフでもない場合は投稿に失敗し一覧ページにリダイレクトされる
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\django_project\log\tests\test_views_sample.py", line 128, in
test_post_failure_another_user
    self.result_redirect(response)
  File "D:\django_project\log\tests\test_views_sample.py", line 52, in result_redirect
    self.assertEqual(str(messages[0]), '日記を更新できるのは投稿者だけです。')
AssertionError: '日記を更新できるのは投稿者と管理者だけです。' != '日記を更新できるのは投稿者だけ
```

```
です。'
- 日記を更新できるのは投稿者と管理者だけです。
?                ----
+ 日記を更新できるのは投稿者だけです。


----------------------------------------------------------------------
Ran 62 tests in 5.363s

FAILED (failures=3)
Destroying test database for alias 'default'...
```

## そのほかの注意点

そのほか、Django のユニットテストでは、以下のような注意点があります。

1. リダイレクトのテストで使う2つのメソッドはそれぞれ独自にリダイレクト先へのリクエストを行う
2. `manage.py startapp <アプリ名>` で作成される `tests.py` はすぐに削除する

### 1. リダイレクトの検証方法

リダイレクトのテストには、慣れないと気づきにくいハマりポイントがあります。
なので、このタイミングでポイントをお伝えしておきます。

リダイレクトのテストに関係するのは、 `django.tests.TestCase` クラスの、以下の2つのインスタンス変数とメソッドです。

使い分けも含めてここで説明します。

| 属性 | 引数 | 初期値 | 説明 |
|---|---|---|---|
| `client` | `follow=True` | `False` | テストクライアントでのリクエスト時に使うオブジェクト<br>リダイレクト先で起きるもろもろを調べたいときに使う。 |
| `assertRedirects` | `fetch_redirect_response=True` | `True` | リダイレクトレスポンスのテストに使うメソッド。<br>リダイレクト先のでの `GET` リクエストのステータスコードを調べたいときに使う。 |

まず、 `client` オブジェクトについて説明します。

`client` オブジェクトは、テストクライアントでのリクエスト時に使うオブジェクトです。
`get` や `get` などのリクエストメソッド実行時に、キーワード `follow` を指定できます。

`follow=True` を指定すると、リダイレクト先のページのレスポンスを取得します。
`follow=False` を指定すると、リダイレクト先のページのレスポンスを取得しません。
デフォルト値は `False` です。

https://github.com/k-brahma/django_photo_diary/blob/main/log/tests/test_views_sample_basic.py

```python
from django.contrib.auth import get_user_model
from django.shortcuts import resolve_url
from django.test import TestCase
from log.models import Article


class TestRedirectClient(TestCase):
    """
    client.get でのリダイレクトのテストの例
    follow=True/False での挙動の違いを確認する
    """

    def setUp(self):
        """ すべてのテストメソッドに共通の準備 """
        user = get_user_model().objects.create_user(
            username='test', email='foo@bar.com', password='testpassword')
        article = Article.objects.create(
            title='base_test_title', body='base_test_body', user=user)

        self.redirect_path = resolve_url('log:article_list')
        self.path = resolve_url('log:article_update', pk=article.pk)

    def test_follow_false(self):
        """ follow=False のとき、リダイレクト先のページを取得しない """
        response = self.client.get(self.path, follow=False)

        # リダイレクトコードを受け取るところまでしか処理を進めないので、ステータスコードは 302
        self.assertEqual(response.status_code, 302)

        # リダイレクト先ページのHTMLを取得しない
        html = response.content.decode('utf-8')
        self.assertEqual(html, "")

        # リダイレクト先ページの context を取得しない
        self.assertIsNone(response.context)

    def test_follow_true(self):
        """ follow=True  のとき、リダイレクト先のページを取得する """
        response = self.client.get(self.path, follow=True)

        # リダイレクト先ページを取得するので、ステータスコードは 200
        self.assertEqual(response.status_code, 200)

        # リダイレクト先ページのHTMLを取得するので、空のコンテンツではない
        html = response.content.decode('utf-8')
        self.assertNotEqual(html, "")

        # リダイレクト先ページの context を取得するので、Noneではない
        self.assertIsNotNone(response.context)

    def test_default(self):
        """ follow のデフォルト値は False """
```

```
        response = self.client.get(self.path, follow=False)

        self.assertEqual(response.status_code, 302)

        html = response.content.decode('utf-8')
        self.assertEqual(html, "")

        self.assertIsNone(response.context)
```

次に、 `assertRedirects` メソッドについて説明します。

`assertRedirects` メソッドは、リダイレクトレスポンスのテストに使うメソッドです。
リダイレクト先の情報のうち、以下の2つの項目についてはこのメソッドで検査することができます。
(逆に言うと、以下の2つ以外の項目については検査できません)

- リダイレクト先のパス
- リダイレクト先ページに `GET` リクエストを行ったときのレスポンスのステータスコード

ただし、 `fetch_redirect_response` 引数の値が `False` のときは、リダイレクト先のパスしか検査できません。
リダイレクト先ページのステータスコードも調べるには、 `fetch_redirect_response` 引数の値を `True` にする必要があります。

> もっとも、以下にあるとおり、 `fetch_redirect_response` 引数のデフォルト値は `True` です。> ですので、明示的に `fetch_redirect_response=True` と指定する必要はありません。

**`assertRedirects` メソッドの引数**

| 引数名 | 初期値 | 概要 | 例 |
|---|---|---|---|
| `response` | | テスト中のHTTPリクエストへのレスポンスオブジェクト | `response = self.client.get('/my-url/')` |
| `expected_url` | | 期待されるリダイレクト先のURLを表す文字列 | `'/login/'` |
| `status_code` | `302` | 期待されるリダイレクトのHTTPステータスコード | `status_code=301` |
| `target_status_code` | `200` | リダイレクト先URLへの `GET` リクエストの期待されるHTTPステータスコード | `target_status_code=200` |
| `msg_prefix` | `""` | アサーションエラーメッセージの接頭辞として使用する文字列 | `msg_prefix='Assertion Failed:'` |
| `fetch_redirect_response` | `True` | リダイレクト先のレスポンスを取得するかどうかを制御するフラグ | `fetch_redirect_response=False` |

`assertRedirects` メソッドの使用例:

`status_code` , `target_status_code` , `fetch_redirect_response` 引数の初期値を念頭に入れたうえで、以下のコードを読んでみてください。

```
response = self.client.get('/my-url/')

# 以下はいずれも同じ
self.assertRedirects(response, "/login/")
self.assertRedirects(response, "/login/", status_code=302)
self.assertRedirects(response, "/login/", status_code=302, target_status_code=200)
self.assertRedirects(response, "/login/", status_code=302, target_status_code=200,
fetch_redirect_response=True)
self.assertRedirects(response, "/login/", 302, 200, True)

# 以下はいずれも同じ
self.assertRedirects(response, "/login/", fetch_redirect_response=False)
self.assertRedirects(response, "/login/", status_code=302, fetch_redirect_response=False)
```

`assertRedirects` メソッドは、 `fetch_redirect_response=True` を指定すると、以下の動作をします。

1. 第一引数 `response` の内容を元にしてリダイレクト先のパスを取得する
2. 1.で取得したリダイレクト先に `GET` リクエストを送信する
3. 2.で取得したレスポンスのステータスコードを調べ、 `target_status_code` で指定された値と比較する

`assertRedirects` メソッドは、 `client` がメソッド実行時に `follow=True` を指定しているかどうかに関わりなく、リダイレクト先からのレスポンスを取得できます。
なぜなら、 `assertRedirects` メソッドは、内部で自分でリダイレクト先のパスに対するリクエストを送信しているからです。

ただし、すでに述べたとおり、リダイレクト先の情報のうち、このメソッドで検査することができるのは以下の2つの項目だけです。

- リダイレクト先のパス
- リダイレクト先ページに `GET` リクエストを行ったときのレスポンスのステータスコード

ですので、レスポンスに含まれる HTML やコンテキストの内容等を検査したいというときには、 `client.get(path, follow=True)` とし、レスポンスを `assert` メソッドで検証することになります。

https://github.com/k-brahma/django_photo_diary/blob/main/log/tests/test_views_sample_basic.py

```
    class TestRedirectAssertRedirect(TestCase):
        """
        assertRedirects() の挙動をテスト
        fetch_redirect_response=True/False での挙動の違いを確認する
        """

        def setUp(self):
            """ すべてのテストメソッドに共通の準備 """
            user = get_user_model().objects.create_user(
                username='test', email='foo@bar.com', password='testpassword')
            article = Article.objects.create(
                title='base_test_title', body='base_test_body', user=user)

            self.redirect_path = resolve_url('log:article_list')
            self.path = resolve_url('log:article_update', pk=article.pk)

        def test_fetch_redirect_response_false(self):
            """ fetch_redirect=False のとき、assertRedirects メソッドは、
                リダイレクト先の検証を行わない """
            response = self.client.get(self.path)

            self.assertRedirects(
                response, self.redirect_path,
                target_status_code=500,  # リダイレクト先の検証を行わないのでこの値は無視される
                fetch_redirect_response=False)

        def test_fetch_redirect_response_true(self):
            """ fetch_redirect=True のとき、assertRedirects メソッドは、
                内部でリダイレクト先へのリクエストを発行し、ステータスコードの検証を行う """
            response = self.client.get(self.path)

            # target_status_code は 正しい値なのでOK
            self.assertRedirects(response, self.redirect_path, target_status_code=200,
                                 fetch_redirect_response=True)

            # target_status_code は 正しい値ではないのでこれはNG
            # self.assertRedirects(response, self.redirect_path,
            #                      target_status_code=500,
            #                      fetch_redirect_response=True)

        def test_fetch_redirect_response_default(self):
            """ fetch_redirect のデフォルト値は True """
            response = self.client.get(self.path)

            # target_status_code は 正しい値なのでOK
            self.assertRedirects(response, self.redirect_path, target_status_code=200, )

            # target_status_codeの初期値は 200 なので以下でもOK
            self.assertRedirects(response, self.redirect_path)

            # 以下は、そのほかの初期値もあえて指定したもの
            self.assertRedirects(response, self.redirect_path,
```

```
                        status_code=302,
                        target_status_code=200)
        self.assertRedirects(response, self.redirect_path,
                        status_code=302,
                        target_status_code=200,
                        fetch_redirect_response=True)

        # target_status_code は 正しい値ではないのでこれはNG
        # self.assertRedirects(response, self.redirect_path, target_status_code=500, )
```

リダイレクトの検査についての話を別の角度からまとめます。

| 検査したい項目 | 使うオブジェクト | 引数についての注意 |
| --- | --- | --- |
| リダイレクト先のパス | `assertRedirects` | `expected_url` 引数で検証する<br>(必須) |
| リダイレクトレスポンスの<br>ステータスコード | `assertRedirects` | `status_code` 引数で検証する<br>(デフォルトは 302) |
| リダイレクト先でのレスポンスの<br>ステータスコード | `assertRedirects` | `target_status_code` 引数で検証する<br>(デフォルトは 200)<br><br>`fetch_redirect_response` は `True` が必要<br>(デフォルトは `True` ) |
| 上記以外 | `client` | `follow` は `True` にする |

最後に、様々な検査項目を想定したサンプルコードを紹介します。

```
    class TestRedirectVariousCases(TestCase):
        """ 種々のニーズに対応したリダイレクトのテスト方法まとめ """

        def setUp(self):
            """ すべてのテストメソッドに共通の準備 """
            user = get_user_model().objects.create_user(
                username='test', email='foo@bar.com', password='testpassword')
            article = Article.objects.create(
                title='base_test_title', body='base_test_body', user=user)

            self.redirect_path = resolve_url('log:article_list')
            self.path = resolve_url('log:article_update', pk=article.pk)

        def test_assert_status_code_only(self):
            """ リダイレクト元でのスタータスコードを検査したいだけのとき"""
            response = self.client.get(self.path)
            self.assertEqual(response.status_code, 302)

        def test_assert_redirect_status_code(self):
            """ リダイレクト先のステータスコードを検査したいだけのとき """
            response = self.client.get(self.path, follow=True)
            self.assertEqual(response.status_code, 200)

        def test_assert_redirect_page_content(self):
            """ リダイレクト先のコンテンツを検査したいいとき """
            response = self.client.get(self.path, follow=True)

            html = response.content.decode('utf-8')
            self.assertNotEqual(html, "")

            self.assertContains(response, '記事一覧')

            messages = list(response.context['messages'])
            self.assertEqual(len(messages), 1)
            self.assertEqual(str(messages[0]), '日記を更新できるのは投稿者と管理者だけです。')

        def test_assert_redirect(self):
            """ リダイレクト元とリダイレクト先のスタータスコードの両方を検査したいとき """
            response = self.client.get(self.path, follow=True)
            self.assertRedirects(response, self.redirect_path,
                                 status_code=302, target_status_code=200)

            # status_code=302, target_status_code=200 はともに初期値なので省略可能
            self.assertRedirects(response, self.redirect_path)
```

ところで、 `assertRedirects` メソッドのキーワード引数 `fetch_redirect_response` のデフォルト値は `True` ですが、 `False` は、どのようなときに指定するのでしょうか。
このオプションは、リダイレクト先のパスが他サイトの `URL` になってしまう等、リダイレクト先のコンテンツを取得することができないときに指定します。
(たとえば、 `https://gogole.com/` にリダイレクトするといった場合です)

通常はデフォルトの `True` のままで問題ありません。

## 2. manage.py startapp <アプリ名> で作成される tests.py はすぐに削除する

Django では、 `manage.py startapp <アプリ名>` コマンドでアプリケーションを作成すると、自動的に `tests.py` ファイルが作成されます。

ところで、この `tests.py` を放置したまま同じディレクトリに `tests` パッケージを作ると、どうなるでしょうか。
実は、 `tests.py` と `tests` パッケージが同一ディレクトリにあると、 `python manage.py test` コマンド実行時に `ImportError` エラーが発生してテストそのものが実行できなくなることがあります。

Django テストコードを書くときは、テスト対象のモジュールごとに `tests` パッケージ内に `test_<モジュール名>` といった名称のモジュールを作り、その中にテストコードを書くのが一般的です。
`tests.py` ファイルを残しておいてもトラブルの元になるだけです。
`python manage.py startapp <アプリ名>` コマンドでアプリケーションを作成したら、すぐに削除するよう習慣づけると良いでしょう。

以下は、 log アプリ内に `tests.py` と `tests` パッケージの両方が存在する状態でテストを実行しようとしたときの、 `ImportError` 発生の例です。

```
(venv) PS D:\django_project> python manage.py test
Traceback (most recent call last):
  File "D:\django_project\manage.py", line 22, in <module>
    main()
  File "D:\django_project\manage.py", line 18, in main
    execute_from_command_line(sys.argv)
  File "D:\django_project\venv\Lib\site-packages\django\core\management\__init__.py", line
442, in execute_from_command_line
    utility.execute()
  File "D:\django_project\venv\Lib\site-packages\django\core\management\__init__.py", line
436, in execute
    self.fetch_command(subcommand).run_from_argv(self.argv)
  File "D:\django_project\venv\Lib\site-packages\django\core\management\commands\test.py",
line 24, in run_from_argv
    super().run_from_argv(argv)
  File "D:\django_project\venv\Lib\site-packages\django\core\management\base.py", line
412, in run_from_argv
    self.execute(*args, **cmd_options)
  File "D:\django_project\venv\Lib\site-packages\django\core\management\base.py", line
458, in execute
    output = self.handle(*args, **options)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\django_project\venv\Lib\site-packages\django\core\management\commands\test.py",
line 68, in handle
    failures = test_runner.run_tests(test_labels)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\django_project\venv\Lib\site-packages\django\test\runner.py", line 1048, in
run_tests
    suite = self.build_suite(test_labels, extra_tests)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\django_project\venv\Lib\site-packages\django\test\runner.py", line 898, in
build_suite
    tests = self.load_tests_for_label(label, discover_kwargs)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\django_project\venv\Lib\site-packages\django\test\runner.py", line 872, in
load_tests_for_label
    tests = self.test_loader.discover(start_dir=label, **kwargs)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\kbrah\AppData\Local\Programs\Python\Python311\Lib\unittest\loader.py",
line 322, in discover
    tests = list(self._find_tests(start_dir, pattern))
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\kbrah\AppData\Local\Programs\Python\Python311\Lib\unittest\loader.py",
line 377, in _find_tests
    tests, should_recurse = self._find_test_path(full_path, pattern)
                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\kbrah\AppData\Local\Programs\Python\Python311\Lib\unittest\loader.py",
line 429, in _find_test_path
    raise ImportError(
ImportError: 'tests' module incorrectly imported from 'D:\\django_project\\log\\tests'.
Expected 'D:\\django_project\\log'. Is this module globally installed?
```