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 ============================
```