

unittest.mock

сделай tutorial по pytest.fixture и unittest.mock и psycopg2?

Конечно! Вот мини-руководство (tutorial) по использованию `pytest.fixture`, `unittest.mock`, и работе с базой данных через `psycopg2`. Это будет полезно для тестирования кода, который взаимодействует с PostgreSQL.

---

## 🔧 Цель

Научиться:

- Создавать фикстуры в `pytest`
  - Использовать `unittest.mock` для подмены зависимостей
  - Подключаться к PostgreSQL через `psycopg2`
  - Тестируировать функции, работающие с базой данных, не трогая реальную БД
- 

## 📦 Установка зависимостей

bash

Copировать

Редактировать

```
pip install pytest psycopg2-binary
```

## 1 Пример кода для тестирования

```
python Копировать Редактировать

# db_module.py
import psycopg2

def fetch_user_by_id(user_id, conn):
    with conn.cursor() as cur:
        cur.execute("SELECT id, username FROM users WHERE id = %s", (user_id,))
        return cur.fetchone()
```

## 2 Использование `pytest.fixture` для моков

```
python Копировать Редактировать

# test_db_module.py
import pytest
from unittest.mock import Mock, MagicMock
from db_module import fetch_user_by_id

@pytest.fixture
def mock_connection():
    conn = Mock()
    cursor = MagicMock()
    cursor.__enter__.return_value = cursor # контекстный менеджер
    cursor.fetchone.return_value = (1, 'john_doe')
    conn.cursor.return_value = cursor
    return conn
```

## 3 Тест с использованием `mock_connection`

python

Копировать

Редактировать

```
def test_fetch_user_by_id(mock_connection):
    result = fetch_user_by_id(1, mock_connection)
    assert result == (1, 'john_doe')
    mock_connection.cursor.return_value.execute.assert_called_once_with(
        "SELECT id, username FROM users WHERE id = %s", (1,))
    )
```

## 4 Использование `psycopg2` в настоящем коде (не в teste)

python

Копировать

Редактировать

```
# main.py
import psycopg2
from db_module import fetch_user_by_id

conn = psycopg2.connect(
    dbname="mydb", user="myuser", password="mypassword", host="localhost"
)

user = fetch_user_by_id(1, conn)
print(user)
```

## 💡 Подсказки

- `pytest.fixture` позволяет создавать повторно используемые объекты для тестов (например, мок соединения)
- `unittest.mock.Mock` и `MagicMock` позволяют точно контролировать поведение объектов в тестах
- Вы не обращаетесь к реальной базе, пока используете моки, что делает тесты быстрыми и безопасными

---

Хочешь, я покажу, как написать интеграционный тест с реальной PostgreSQL через Docker?

```
# test_db_module.py
import pytest
from unittest.mock import Mock, MagicMock
from db_module import fetch_user_by_id

@pytest.fixture
def mock_connection():
    conn = Mock()
    cursor = MagicMock()
    cursor.__enter__.return_value = cursor # контекстный менеджер
    cursor.fetchone.return_value = (1, 'john_doe')
    conn.cursor.return_value = cursor
    return conn

def test_fetch_user_by_id(mock_connection):
    result = fetch_user_by_id(1, mock_connection)
    assert result == (1, 'john_doe')

    mock_connection.cursor.return_value.execute.assert_called_once_with(
        "SELECT id, username FROM users WHERE id = %s", (1,))

```

распиши подробно с разъяснения структуры каждого объекта что  
все это означает?

---

Отлично, давай разберем подробно, что именно происходит в этом коде — шаг за шагом, включая структуру **мок-объекта** и зачем всё это нужно.

## Общая идея

Цель — протестировать функцию `fetch_user_by_id`, которая делает SQL-запрос через `psycopg2`, не подключаясь к реальной базе данных. Для этого мы создаем **мок-объекты** — "поддельные" объекты, которые ведут себя как настоящие, но контролируются в teste.

---

## Файл `test_db_module.py`

python

 Копировать

 Редактировать

```
import pytest
from unittest.mock import Mock, MagicMock
from db_module import fetch_user_by_id
```

Импортируем:

- `pytest` — фреймворк для тестирования
  - `Mock`, `MagicMock` — для создания поддельных объектов
  - `fetch_user_by_id` — функцию, которую тестируем
- 

## Фикстура `mock_connection`

python

 Копировать

 Редактировать

```
@pytest.fixture
def mock_connection():
    conn = Mock() # это будет поддельное соединение с БД
    cursor = MagicMock() # поддельный курсор

    # Настраиваем поведение контекстного менеджера
    cursor.__enter__.return_value = cursor # чтобы можно было писать `with conn.cursor() as cur:`
```

```
# Указываем, что вернется при вызове `cur.fetchone()`
cursor.fetchone.return_value = (1, 'john_doe')

# Настраиваем conn.cursor() -> cursor
conn.cursor.return_value = cursor
return conn
```

## 🔍 Что здесь происходит?

plaintext

Copировать

Редактировать

```
conn = Mock()
```

Создается мок-объект, представляющий собой **соединение с базой данных**.

plaintext

Copировать

Редактировать

```
cursor = MagicMock()
```

Создается более "умный" мок для курсора. Почему `MagicMock`, а не просто `Mock`? Потому что `MagicMock` автоматически поддерживает магические методы вроде `__enter__`, `__exit__`, `__iter__`, и т.п. — это нужно для работы с `with`.

plaintext

Copировать

Редактировать

```
cursor.__enter__.return_value = cursor
```

Так мы делаем так, чтобы этот курсор корректно работал в `with`-блоке:

python

Copировать

Редактировать

```
with conn.cursor() as cur:
```

...

Тут `cur` станет тем самым `cursor`.

plaintext

Copировать

Редактировать

```
cursor.fetchone.return_value = (1, 'john_doe')
```

Настраиваем поведение метода `fetchone`, чтобы вернуть тестовые данные, когда вызывается SQL-запрос.

```
plaintext
```

Копировать

Редактировать

```
conn.cursor.return_value = cursor
```

Когда вызывается `conn.cursor()`, возвращается наш настроенный курсор.

---

### Тест `test_fetch_user_by_id`

```
python
```

Копировать

Редактировать

```
def test_fetch_user_by_id(mock_connection):
    result = fetch_user_by_id(1, mock_connection)
    assert result == (1, 'john_doe')
```

Тест вызывает функцию, передает в неё поддельное соединение, и проверяет, что результат соответствует ожидаемому.

```
python
```

Копировать

Редактировать

```
mock_connection.cursor.return_value.execute.assert_called_once_with(
    "SELECT id, username FROM users WHERE id = %s", (1,))
)
```

Проверка, что `execute(...)` был вызван ОДИН РАЗ с указанными аргументами.

---



### Визуализация структуры моков

```
scss
```

Копировать

Редактировать

```
mock_connection (Mock)
```

```
|  
|   cursor() -> cursor (MagicMock)  
|  
|       |  
|       |_ __enter__() -> cursor ← поддержка `with`  
|       |_ execute(...)           ← проверяем в тесте  
|       |_ fetchone() -> (1, 'john_doe')
```

Хочешь, я сделаю вывод этой структуры в виде дерева или схему через ASCII-графику/блок-схему?  
Или показать как это будет выглядеть в реальном дебагgere (`print(...)`)?

показать как это будет выглядеть в реальном дебагgere (`print(...)`)?

Конечно! Давайте добавим несколько `print`-операторов в код, чтобы увидеть, как будут выглядеть мок-объекты и их методы во время выполнения теста.

## Обновленный код с `print`

python

Копировать

Редактировать

```
import pytest  
from unittest.mock import Mock, MagicMock  
from db_module import fetch_user_by_id  
  
@pytest.fixture  
def mock_connection():  
    # Создаем мок соединения  
    conn = Mock()  
    # Создаем мок курсора  
    cursor = MagicMock()  
  
    # Устанавливаем мок курсора для мок соединения  
    conn.cursor.return_value = cursor  
  
    return conn
```

```

# Печатаем, как выглядит мок курсора
print("Мок курсора:", cursor)

# Настроим поведение контекстного менеджера
cursor.__enter__.return_value = cursor # чтобы можно было использовать с `with`

# Настроим, чтобы fetchone возвращал данные
cursor.fetchone.return_value = (1, 'john_doe')

# Печатаем, как будет выглядеть метод fetchone
print("Мок fetchone:", cursor.fetchone)

# Когда вызываем `cursor()`, возвращаем наш мок курсора
conn.cursor.return_value = cursor
return conn

def test_fetch_user_by_id(mock_connection):
    # Печатаем, как выглядит mock_connection (это соединение с БД)
    print("Мок соединения:", mock_connection)

    # Выполняем тест
    result = fetch_user_by_id(1, mock_connection)

    # Печатаем результат
    print("Результат вызова fetch_user_by_id:", result)

    # Проверяем, что execute был вызван один раз с правильными параметрами
    mock_connection.cursor.return_value.execute.assert_called_once_with(
        "SELECT id, username FROM users WHERE id = %s", (1,))

```

## Объяснение:

1. `print("Мок курсора:", cursor)` — показывает, как выглядит мок-объект курсора.
2. `print("Мок fetchone:", cursor.fetchone)` — выводит информацию о том, как будет вести себя

метод `fetchone`.

3. `print("Мок соединения:", mock_connection)` — выводит информацию о мок-соединении (внешний объект `conn`).
  4. `print("Результат вызова fetch_user_by_id:", result)` — выводит результат выполнения функции `fetch_user_by_id`.
- 

## Ожидаемый вывод в консоли

Когда вы запустите тест, вот как будет выглядеть вывод:

bash

Копировать

Редактировать

```
Мок курсора: <MagicMock id='123456789012345678'>
Мок fetchone: <MagicMock name='mock.fetchone' id='123456789012345678'>
Мок соединения: <Mock id='987654321098765432'>
Результат вызова fetch_user_by_id: (1, 'john_doe')
```

## Объяснение вывода:

1. **Мок курсора:** `<MagicMock id='123456789012345678'>` — это мок-объект с уникальным ID, который имитирует поведение настоящего курсора. ID будет отличаться при каждом запуске, так как это уникальный идентификатор для каждого объекта.
  2. **Мок fetchone:** `<MagicMock name='mock.fetchone' id='123456789012345678'>` — это метод `fetchone` мок-курсора. Мы видим, что он является также объектом `MagicMock` и имеет уникальный ID.
  3. **Мок соединения:** `<Mock id='987654321098765432'>` — это мок-соединение с БД. ID тоже будет уникальным для каждого вызова.
  4. **Результат вызова:** `(1, 'john_doe')` — это то, что возвращает наша функция `fetch_user_by_id`, когда она вызывает `fetchone`.
- 

## Дополнительный вывод для отладки

Если вы хотите ещё больше информации, можно вывести подробности по каждому вызову, например:

```
python
```

Копировать

Редактировать

```
print("Методы мок-курора:", dir(cursor))
```

Это выведет все методы и атрибуты, доступные для мок-объекта `cursor`.

---

Если у тебя возникнут дополнительные вопросы или потребуется дополнительная настройка вывода, дай знать! 😊

теперь примерно такой же учебник, но добавь `patch`?  
и сделай такое же глубокое объяснение

Супер! Давай теперь расширим наш мини-учебник, добавив `unittest.mock.patch`, чтобы подменять объекты или функции "на лету" при выполнении тестов.

---

## Учебник по `pytest.fixture` + `unittest.mock.patch` + `psycopg2`

---

### Цель

Научиться:

- Использовать `patch` для временной подмены зависимостей (например, базы данных)
- Глубоко понять, как работает `patch` в связке с фикстурами

- Смотреть "внутрь" моков через `print` и понимать их структуру
- 

## ✳ Пример: функция, которая сама подключается к базе

Код (`db_module.py`)

```
python  
  
import psycopg2  
  
def fetch_user_by_id(user_id):  
    conn = psycopg2.connect(  
        dbname="mydb", user="myuser", password="mypassword", host="localhost"  
    )  
    with conn.cursor() as cur:  
        cur.execute("SELECT id, username FROM users WHERE id = %s", (user_id,))  
        return cur.fetchone()
```

Теперь функция сама вызывает `psycopg2.connect()`, и мы хотим замокать это в тесте.

---

## ✓ Тест с `patch`

```
python  
  
# test_db_module.py  
import pytest  
from unittest.mock import patch, MagicMock  
from db_module import fetch_user_by_id  
  
@pytest.fixture  
def mock_psycopg_connection():  
    # Мок курсора  
    cursor = MagicMock()
```

```
cursor.__enter__.return_value = cursor
cursor.fetchone.return_value = (1, 'patched_user')

# Мок соединения
conn = MagicMock()
conn.cursor.return_value = cursor

# Печатаем структуру моков
print("cursor.fetchone:", cursor.fetchone)
print("conn.cursor:", conn.cursor)

return conn

def test_fetch_user_by_id_with_patch(mock_psycopg_connection):
    # Patch `psycopg2.connect`, чтобы вернуть наш мок
    with patch('db_module.psycopg2.connect', return_value=mock_psycopg_connection) as mock_connect:
        result = fetch_user_by_id(1)

        # Проверки
        assert result == (1, 'patched_user')
        mock_connect.assert_called_once() # убедимся, что connect вызван
        mock_psycopg_connection.cursor.return_value.execute.assert_called_once_with(
            "SELECT id, username FROM users WHERE id = %s", (1,))
    )

    # Печатаем детали
    print("Результат:", result)
    print("Был ли вызов connect:", mock_connect.called)
```

## 🔍 Разбор: как работает patch

Что делает `patch`?

python

Copировать

Редактировать

```
with patch('db_module.psycopg2.connect', return_value=mock_psycopg_connection):
```

- `patch` говорит: "Когда в `db_module` кто-то вызовет `psycopg2.connect(...)`, вместо настоящей функции верни `mock_psycopg_connection`".
- Это временно — **только внутри блока `with`**.
- После выхода из `with`, всё возвращается на место.

## 🧠 Структура моков (визуально)

scss

Copировать Редактировать

```
fetch_user_by_id(user_id)
|
|__ psycopg2.connect(...) ← замещен через patch
|   __ возвращает: mock_psycopg_connection (MagicMock)
|
|__ conn.cursor()           ← это mock.cursor
|   __ __enter__()          ← вернет cursor (MagicMock)
|     __ .execute(...)      ← проверим в тесте
|     __ .fetchone() -> (1, 'patched_user')
```

## ✍️ Что можно напечатать для отладки?

Добавь эти строки и посмотри вывод:

python

Copировать Редактировать

```
print("Тип connect:", type(mock_connect))
print("mock_connect.called:", mock_connect.called)
print("mock_connect.call_args:", mock_connect.call_args)

cur = mock_psycopg_connection.cursor.return_value
print("execute.called:", cur.execute.called)
```

```
print("fetchone.called:", cur.fetchone.called)
```

💡 Вывод в консоли может быть примерно таким:

```
pgsql
```

Копировать Редактировать

```
cursor.fetch_one: <MagicMock name='mock.cursor().fetchone' id='140379913469968'>
conn.cursor: <MagicMock name='mock.cursor' id='140379913489104'>
Результат: (1, 'patched_user')
Был ли вызов connect: True
Тип connect: <class 'unittest.mock.MagicMock'>
mock_connect.called: True
mock_connect.call_args: call()
execute.called: True
fetchone.called: True
```

owl 🐦 Почему это круто?

С помощью `patch` ты можешь:

- Тестиировать код, который сам вызывает внешние зависимости
- Подменять любую функцию, класс или объект, даже встроенные
- Делать тесты быстрыми, без доступа к настоящей базе, API и т.п.

Если хочешь, я могу показать, как использовать `@patch` как декоратор вместо `with` или добавить поддержку нескольких `patch` одновременно.

```
import psycopg2 as pg2
import psycopg2.extras as extras
import pandas as pd
import numpy as np
from time import sleep
import gc
import json
import wrapt_timeout_decorator as wtd
from typing import Tuple, List, Dict, Union, Optional, Any
```

class Database:

```
    """PostgreSQL Database class for managing connections and
executing queries.
```

This class provides methods to connect to a PostgreSQL database  
and perform various  
operations like selecting, inserting, updating, and deleting data.

Attributes:

```
db (str): Database name.
host (str): Database host address.
username (str): Database username.
password (str): Database password.
port (str): Database port.
tz (str): Timezone setting for the database.
timeout (int): Query timeout in seconds.
```

"""

```
def __init__(self, db: str, host: str, username: str, password: str, port:
str,
tz: str, timeout: int = 0) -> None:
    """Initializes the Database instance with connection parameters.
```

Args:

```
db: Database name.
host: Database host address.
username: Database username.
password: Database password.
port: Database port.
tz: Timezone setting for the database.
```

```
    timeout: Query timeout in seconds (default 1000 if 0).
```

```
    """
    self.db = db
    self.timezone = tz
    self.df_result = pd.DataFrame()
    self.host = host
    self.timeout = timeout if timeout > 0 else 1000
    self.user = username
    self.password = password
    self.query_insert_template = "INSERT INTO %s(%s) VALUES %s"
    self.query_update_set_template = """
        UPDATE {table_target} as target
        SET
            {fields}
        FROM {table_source} as source_table
        WHERE {key_fields};"""
    self.port = port
    self.cur = None
    self.conn = None
    self.logger = None
```

```
def log(self, kind: str = 'postgresql-service', log_level: str = 'info',
       message: str = "") -> None:
    """Logs messages using the configured logger.
```

Args:

```
    kind: Category of the log message.
    log_level: Log level ('info', 'warn', or 'error').
    message: The message to log.
```

```
    """
    if self.logger:
        #-----
```

```
        if log_level == 'info': self.logger.info(message, kind)
        elif log_level == 'warn': self.logger.warning(message, kind)
        elif log_level == 'error': self.logger.error(message, kind)
```

```
def connect(self, first: bool = True) -> None:
```

```
"""Establishes a connection to the PostgreSQL database.
```

Args:

```
    first: If True, logs connection success message (default True).
```

```
"""
```

```
if self.conn is None:
```

```
    try:
```

```
        self.conn = pg2.connect(  
            dbname=self.db,  
            host=self.host,  
            user=self.user,  
            password=self.password,  
            port=self.port  
        )
```

```
    except (Exception, pg2.DatabaseError) as error:
```

```
        self.log('connection to DB', 'error', f"Connection to DB  
{self.db}: {error}")  
        raise error
```

```
    finally:
```

```
        if first:
```

```
            self.log('connection to DB', 'info', f"Connection to DB  
{self.db}: opened successfully!")  
            self.cur = self.conn.cursor()
```

```
        if self.timezone:
```

```
            self.cur.execute(f"SET timezone='{self.timezone}';")  
            self.conn.commit()  
        else:
```

```
            self.cur = self.conn.cursor()
```

```
def select_rows(self, query: str = "") -> Tuple[List[Any], bool]:
```

```
    """Executes a SELECT query and returns the results.
```

Args:

```
    query: The SQL query to execute.
```

Returns:

```
    A tuple containing:
```

- List of rows returned by the query
- Boolean indicating success status

Example:

```
>>> db = Database(...)  
>>> rows, success = db.select_rows("SELECT * FROM users")  
>>> if success:  
...     print(rows)  
....
```

```
@wtd.timeout(self.timeout, use_signals=True)  
def execute_sql_query(query, curr):  
    """  
    decorated procedure of reading data from DB by query  
    """  
  
    curr.execute(query)  
  
    return [row for row in curr.fetchall()]  
  
    self.connect(False)  
    with self.cur as cur:  
        try:  
            records = execute_sql_query(query, cur)  
            self.conn.commit()  
            cur.close()  
            self.close()  
            return records, True  
        except TimeoutError:  
            self.log('query to DB', 'error', f"timeout occurred  
({self.timeout}) during execution query to DB {self.db}")  
            self.close()  
        except Exception as error:  
            self.log('query to DB', 'error', f"selecting query to DB  
{self.db}: {error}")  
            self.close()  
  
    return [], False
```

write pytest for this code?

To effectively test the `Database` class without connecting to an actual PostgreSQL database, we can utilize `pytest` along with `unittest.mock` to mock the `psycopg2` connection and cursor objects. This approach allows us to simulate database interactions and ensure our methods behave as expected.

Here's how you can set up your test suite:

1. **Install Required Packages:** Ensure you have `pytest` and `psycopg2` installed. If not, install them using pip:

```
bash
```

Copy Edit

```
pip install pytest psycopg2
```

2. **Mocking Database Connections:** We'll use `unittest.mock` to replace the `psycopg2.connect` method with a mock that returns a mock connection object. This mock connection will have a mock cursor to simulate executing queries and fetching results.
3. **Testing the `select_rows` Method:** We'll write tests to verify that the `select_rows` method correctly executes queries and handles various scenarios, such as successful data retrieval and exceptions.

Below is the complete test suite for the `Database` class:

```
python
```

Copy Edit

```
import pytest
from unittest.mock import patch, MagicMock
import psycopg2
from psycopg2 import DatabaseError
from your_module import Database # Replace 'your_module' with the actual module name

# Sample database configuration for testing
DB_CONFIG = {
    'db': 'test_db',
    'host': 'localhost',
    'username': 'test_user',
    'password': 'test_password',
    'port': '5432',
    'tz': 'UTC',
    'timeout': 5
}
```

```
@pytest.fixture
def mock_psycopg_connect():
    """Fixture to mock psycopg2.connect."""
    with patch('psycopg2.connect') as mock_connect:
        yield mock_connect

@pytest.fixture
def db_instance():
    """Fixture to create an instance of the Database class."""
    return Database(**DB_CONFIG)

def test_select_rows_success(db_instance, mock_psycopg_connect):
    """Test select_rows method for successful data retrieval."""
    mock_conn = MagicMock()
    mock_cursor = mock_conn.cursor.return_value.__enter__.return_value
    mock_cursor.fetchall.return_value = [(1, 'test_user')]

    mock_psycopg_connect.return_value = mock_conn

    query = "SELECT id, username FROM users;"
    result, success = db_instance.select_rows(query)

    assert success is True
    assert result == [(1, 'test_user')]
    mock_cursor.execute.assert_called_once_with(query)

def test_select_rows_timeout(db_instance, mock_psycopg_connect):
    """Test select_rows method handling a timeout exception."""
    mock_conn = MagicMock()
    mock_cursor = mock_conn.cursor.return_value.__enter__.return_value

    # Simulate a timeout by having execute raise a TimeoutError
    mock_cursor.execute.side_effect = TimeoutError("Query timed out")

    mock_psycopg_connect.return_value = mock_conn

    query = "SELECT id, username FROM users;"
    result, success = db_instance.select_rows(query)

    assert success is False
```

```
assert result == []
mock_cursor.execute.assert_called_once_with(query)

def test_select_rows_database_error(db_instance, mock_psycopg_connect):
    """Test select_rows method handling a database error."""
    mock_conn = MagicMock()
    mock_cursor = mock_conn.cursor.return_value.__enter__.return_value

    # Simulate a database error by having execute raise a DatabaseError
    mock_cursor.execute.side_effect = DatabaseError("Database error occurred")

    mock_psycopg_connect.return_value = mock_conn

    query = "SELECT id, username FROM users;"
    result, success = db_instance.select_rows(query)

    assert success is False
    assert result == []
    mock_cursor.execute.assert_called_once_with(query)

def test_connect_method(db_instance, mock_psycopg_connect):
    """Test the connect method establishes a database connection."""
    mock_conn = MagicMock()
    mock_psycopg_connect.return_value = mock_conn

    db_instance.connect()

    mock_psycopg_connect.assert_called_once_with(
        dbname=DB_CONFIG['db'],
        host=DB_CONFIG['host'],
        user=DB_CONFIG['username'],
        password=DB_CONFIG['password'],
        port=DB_CONFIG['port']
    )
    mock_conn.cursor.assert_called_once()

def test_log_method(db_instance):
    """Test the log method with a mock logger."""
    mock_logger = MagicMock()
    db_instance.logger = mock_logger
```

```
db_instance.log(kind='test', log_level='info', message='Test message')

mock_logger.info.assert_called_once_with('Test message', 'test')
```

#### Explanation:

- **Fixtures:**
  - `mock_psycopg_connect` : Mocks the `psycopg2.connect` function to prevent actual database connections.
  - `db_instance` : Creates an instance of the `Database` class with test configuration.
- **Tests:**
  - `test_select_rows_success` : Verifies that `select_rows` returns expected data when the query executes successfully.
  - `test_select_rows_timeout` : Simulates a `TimeoutError` to ensure `select_rows` handles timeouts gracefully.
  - `test_select_rows_database_error` : Simulates a `DatabaseError` to check error handling in `select_rows`.
  - `test_connect_method` : Ensures that the `connect` method establishes a database connection with the correct parameters.
  - `test_log_method` : Tests the `log` method to confirm it calls the logger with the appropriate arguments.

#### Notes:

- Replace '`your_module`' with the actual name of the module where the `Database` class is defined.
- These tests use `unittest.mock` to mock database connections and interactions, allowing you to test the behavior of your methods without needing a real database.
- The `mock_psycopg_connect` fixture uses `patch` to replace `psycopg2.connect` with a mock that returns a mock connection object.
- Each test function sets up the necessary mocks and asserts that methods are called with expected arguments and that return values are as expected.

By structuring your tests in this way, you can ensure that your `Database` class methods are robust and handle various scenarios appropriately without relying on an actual database connection.

объясни разницу, чем отличается Mock объект от MagicMock подробно?

Отличный вопрос! Разница между `Mock` и `MagicMock` в Python — это нечто вроде "базового против продвинутого мока". Давай разложим это по полочкам.

### ◆ Общая суть: `Mock` и `MagicMock`

Оба класса — из модуля `unittest.mock`, и предназначены для создания **поддельных объектов** (моков), которые имитируют поведение настоящих объектов, чтобы тестировать код в изоляции.

Они позволяют:

- Проверять вызовы методов и их аргументы
- Подменять методы и атрибуты
- Симулировать возвращаемые значения
- Захватывать и анализировать поведение

### ◆ Главное отличие: магические методы

	Mock	MagicMock
одерживает <code>__str__</code> , <code>__len__</code> , <code>__iter__</code> , <code>__getitem__</code> , <code>__enter__</code> , <code>__exit__</code> и т.д.	Нет, не умеет по умолчанию	Да, уже встроены
:пользовать для <code>with ...</code> (контекстный менеджер)	Нужно настраивать вручную	Работает "из коробки"

оведение с операторами (+, -, ==, ...)

Не определено

Можно переопределять, уже есть базовая реализация

значение

Базовая подмена

Подмена с поддержкой протоколов и магии

## ■ Пример: контекстный менеджер

С Mock:

python

Копировать

Редактировать

```
from unittest.mock import Mock

mock_obj = Mock()
mock_obj.__enter__ = lambda self: mock_obj
mock_obj.__exit__ = lambda *args: None

with mock_obj as m:
    print("Inside with")
```

➡ Нужно самому определять \_\_enter\_\_ и \_\_exit\_\_.

С MagicMock:

python

Копировать

Редактировать

```
from unittest.mock import MagicMock

mock_obj = MagicMock()

with mock_obj as m:
    print("Inside with")
```

Работает сразу — `MagicMock` уже содержит все необходимые "магические методы".

## Пример: доступ к атрибутам и магия

python

Копировать

Редактировать

```
mock = Mock()  
magic = MagicMock()  
  
print(len(mock))      # ✗ TypeError: object of type 'Mock' has no Len()  
print(len(magic))     # ✓ 0 (можно изменить: magic.__Len__.return_value = 5)
```

## Использование: когда что применять?

Когда использовать	Mock	<code>MagicMock</code>
Простые объекты	Да	Можно, но не обязательно
Контекстные менеджеры	Неудобно (надо добавлять <code>__enter__</code> )	Удобно — встроено
Ожидаются <code>len(obj)</code> , <code>iter(obj)</code> , <code>obj[0]</code> , и т.д.	Не подходит без донастройки	Идеален для таких случаев
Хотите полный контроль	Да (только то, что явно задали)	Часто может быть "слишком умный"

TL;DR

- `Mock` — проще, но требует ручной настройки для магических методов.
  - `MagicMock` — как `Mock`, но уже умеет `_enter_`, `_exit_`, `_len_`, `_getitem_`, и другие "магии".
  - Используй `MagicMock`, если объект должен вести себя как сложный тип, класс или структура.
- 

Хочешь пример из реальной жизни, где `MagicMock` спасает тест при работе с файлами, базой, или API?

Хочешь пример из реальной жизни, где `MagicMock` спасает тест при работе с файлами, базой, или API -да

Отлично! Давай рассмотрим **реальный пример из практики**, где `MagicMock` буквально спасает тест — там, где без него пришлось бы сильно усложнять код.

---

## 🎯 Сценарий: чтение данных из файла

Представим, что у тебя есть функция, которая читает данные из файла и возвращает первую строку.

### 🔧 Реальный код

python

Копировать  Редактировать

```
def read_first_line(file_path):
    with open(file_path, 'r') as f:
        return f.readline()
```

## 🎯 Цель теста:

Протестировать `read_first_line`, не создавая реальный файл.

---

## ✍ Тест с MagicMock

python

Copировать

Редактировать

```
from unittest.mock import MagicMock, patch
import builtins

def test_read_first_line():
    mock_file = MagicMock()
    mock_file.__enter__.return_value.readline.return_value = "Hello, world!\n"

    with patch.object(builtins, 'open', return_value=mock_file) as mock_open:
        result = read_first_line("fake.txt")

    assert result == "Hello, world!\n"
    mock_open.assert_called_once_with("fake.txt", 'r')
```

## 💥 Почему это сработало?

- `with open(...)` как `f`: требует, чтобы возвращаемый объект поддерживал магические методы `__enter__` и `__exit__`.
- Только `MagicMock` по умолчанию имеет эти методы → `Mock` бы выдал ошибку типа:

makefile

Copировать

Редактировать

```
AttributeError: __enter__
```

## 🔍 Как это выглядит в глубине:

```
python
```

Копировать

Редактировать

```
mock_file = MagicMock()  
mock_file.__enter__.return_value.readline.return_value = "Hello"
```

Это симулирует:

```
python
```

Копировать

Редактировать

```
with open(...) as f:  
    f.readline()
```

## ✓ Результат

- Ты протестировал код, **не создавая файл**
- Всё работает быстро
- Нет зависимости от файловой системы

Хочешь похожий пример с API-запросом (`requests.get(...)`) или с подключением к базе данных через `psycopg2`?

import builtins что это такое и для чего?

Отличный вопрос! Давай разберём, что такое `builtins`, и зачем мы вообще импортируем `import builtins`.

## 🧠 Что такое `builtins`?

Модуль `builtins` — это **встроенный модуль Python**, содержащий **все встроенные функции, классы и исключения**, которые ты используешь каждый день — **без импорта**.

---

## 📦 Примеры объектов из `builtins`

Функция / класс	Что делает
<code>open</code>	Открытие файлов
<code>print</code>	Печать в консоль
<code>len</code> , <code>sum</code> , <code>max</code>	Базовые функции
<code>int</code> , <code>str</code> , <code>list</code>	Стандартные типы данных
<code>Exception</code> , <code>ValueError</code>	Базовые исключения

## 💡 Почему `import builtins` в тестах?

Чтобы можно было **замокать** (подменить) встроенные функции, например `open`, `print`, и т.д.

## 🔍 Пример с подменой `open`

```
python
from unittest.mock import patch, MagicMock
import builtins # 👍 импортируем встроенные объекты

def test_open_mock():
    mock_file = MagicMock()
    mock_file.__enter__.return_value.read.return_value = "Mocked content"
```

```
with patch.object(builtins, 'open', return_value=mock_file) as mock_open:
    with open("fake.txt") as f:
        content = f.read()

    assert content == "Mocked content"
```

### 🛠 Альтернатива: `patch('builtins.open', ...)`

Ты также можешь просто указать строкой:

python

Copировать

Редактировать

```
with patch('builtins.open', return_value=mock_file):
```

Оба способа работают одинаково, разница лишь в стиле:

- `patch('builtins.open', ...)` — строкой
- `patch.object(builtins, 'open', ...)` — с модулем как объектом

### ✓ TL;DR

- `builtins` — это модуль всех встроенных функций Python
- Импорт нужен, чтобы мокать такие функции как `open`, `print`, `len`, и т.д.
- Особенно полезен при тестировании — когда надо подменить стандартное поведение

Хочешь пример с замоканным `print` или с подменой `input()` в teste?

Хочешь похожий пример с API-запросом (`requests.get(...)`) или с подключением к базе данных через `psycopg2`? да и объясни очень подробно все объекты?

Отлично! Давай разберём пример с API-запросом через `requests.get()`, и потом — аналогичный пример с `psycopg2`. Я объясню подробно все объекты, как они работают и что происходит на каждом шаге.

---

## ✓ Пример 1: Тестирование API-функции (`requests.get()`)

📦 Реальный код (функция, которую мы хотим протестировать):

```
python  
  
import requests  
  
def get_user_data(user_id):  
    url = f"https://api.example.com/users/{user_id}"  
    response = requests.get(url)  
    if response.status_code == 200:  
        return response.json()  
    else:  
        return None
```

📝 Тест с использованием `patch` и `MagicMock`

```
python  
  
from unittest.mock import patch, MagicMock
```

```
import pytest
from your_module import get_user_data # Замените your_module на имя файла

def test_get_user_data_success():
    mock_response = MagicMock()
    mock_response.status_code = 200
    mock_response.json.return_value = {'id': 1, 'name': 'Alice'}

    with patch('your_module.requests.get', return_value=mock_response) as mock_get:
        result = get_user_data(1)

    assert result == {'id': 1, 'name': 'Alice'}
    mock_get.assert_called_once_with("https://api.example.com/users/1")
```

## 🔍 Объяснение по шагам:

Компонент	Объяснение
patch('your_module.requests.get', ...)	Заменяет <code>requests.get</code> в <code>your_module</code> на мок
mock_response = MagicMock()	Создаем мок-объект, который будет имитировать ответ от <code>requests.get()</code>
mock_response.status_code = 200	Настраиваем поведение: код ответа 200, как при успешном запросе
mock_response.json.return_value = {...}	Задаем, что вернёт <code>.json()</code> — как будто мы получили JSON из API
assert_called_once_with(...)	Проверяем, что вызов был только один и с нужным URL

## 🎯 Результат:

- Мы не делали реальный HTTP-запрос
  - Мы контролируем поведение API и можем симулировать любую ситуацию
  - Мок-объект играет роль ответа от `requests.get()`
- 

## ✓ Пример 2: Тестирование подключения к базе через `psycopg2`

📦 Код, который будем тестировать:

```
python

import psycopg2

def fetch_user_name(user_id):
    conn = psycopg2.connect(dbname='mydb', user='user', password='pass', host='localhost')
    with conn.cursor() as cur:
        cur.execute("SELECT name FROM users WHERE id = %s", (user_id,))
        result = cur.fetchone()
    conn.close()
    return result[0] if result else None
```

📝 Тест с `patch` и `MagicMock`

```
python

from unittest.mock import patch, MagicMock
import pytest
from your_module import fetch_user_name

def test_fetch_user_name():
    mock_conn = MagicMock()
```

```
mock_cursor = MagicMock()

# Настраиваем cursor как контекстный менеджер
mock_conn.cursor.return_value.__enter__.return_value = mock_cursor
mock_cursor.fetchone.return_value = ("Alice",)

with patch('your_module.psycopg2.connect', return_value=mock_conn) as mock_connect:
    result = fetch_user_name(1)

    assert result == "Alice"
    mock_connect.assert_called_once() # Проверяем, что подключение вызвано
    mock_cursor.execute.assert_called_once_with("SELECT name FROM users WHERE id = %s", (1,))
```

## 🔍 Объяснение по шагам:

Компонент	Объяснение
mock_conn = MagicMock()	Это фейковое соединение к БД
mock_cursor = MagicMock()	Фейковый курсор, через который выполняются запросы
mock_conn.cursor.return_value.__enter__.return_value = mock_cursor	Настраиваем <code>with conn.cursor() as cur:</code> так, чтобы <code>cur</code> стал <code>mock_cursor</code>
mock_cursor.fetchone.return_value = ("Alice",)	Устанавливаем, что вернёт SQL-запрос
patch('your_module.psycopg2.connect', return_value=mock_conn)	Подменяем <code>psycopg2.connect</code> , чтобы не подключаться к реальной базе

## 🧠 Как устроено взаимодействие внутри функции:

scss

Copировать

Редактировать

```
fetch_user_name()
|
|   psycopg2.connect(...)      ← замещен через patch
|   ↘ возвращает: mock_conn
|
|   mock_conn.cursor()        ← возвращает объект с __enter__
|   ↘ __enter__() → mock_cursor
|
|   mock_cursor.execute(...)  ← вызовем SQL
|   mock_cursor.fetchone() → ("Alice",)
|
|   conn.close()
```

## ✓ Итог

Что мы достигли

Как

Безопасное тестирование

Нет настоящей базы/API

Полный контроль

Можно симулировать ошибки, таймауты

Повторяемость

Результаты всегда одинаковые

Удобство

Используем `patch`, `MagicMock` для имитации

Хочешь пример, где мы симулируем ошибку (например, `TimeoutError` или `psycopg2.OperationalError`) для проверки поведения функции?

Хочешь пример, где мы симулируем ошибку (например, `TimeoutError` или `psycopg2.OperationalError`) для проверки поведения функции? -да

Отлично! Сейчас покажу, как **симулировать ошибки** — например, `TimeoutError` или `psycopg2.OperationalError` — в тестах с использованием `patch` и `MagicMock`. Это очень полезно, когда ты хочешь протестировать **обработку ошибок** в своём коде.

---

## ✓ Пример функции: подключение к базе и обработка ошибки

### 📦 Код функции (`db_module.py`):

```
python
```

Копировать Редактировать

```
import psycopg2
from psycopg2 import OperationalError

def connect_to_db():
    try:
        conn = psycopg2.connect(
            dbname='mydb', user='user', password='pass', host='localhost'
        )
        return conn
    except OperationalError as e:
        return f"Database error: {str(e)}"
```

### 🧪 Тест: симуляция `OperationalError`

```
python
```

Копировать Редактировать

```
from unittest.mock import patch
import psycopg2
```

```
import pytest
from db_module import connect_to_db

def test_connect_to_db_operational_error():
    # Подменяем psycopg2.connect, чтобы выбрасывал OperationalError
    with patch('db_module.psycopg2.connect', side_effect=psycopg2.OperationalError("connection failed"))
        result = connect_to_db()

    assert result == "Database error: connection failed"
```

## 🔍 Объяснение:

Компонент	Что делает
patch(..., side_effect=Exception(...))	Вместо обычного вызова, функция выбросит исключение
psycopg2.connect → raises OperationalError	Это имитация сбоя подключения к базе
result == ...	Мы проверяем, что наша функция правильно обработала исключение

## ✓ Пример с TimeoutError

Теперь давай сделаем пример, где функция выполняет SQL-запрос, и мы хотим протестировать, как она реагирует на `TimeoutError`.

## 📦 Код:

```
python
```

Copировать

Редактировать

```
def execute_query(cur, query):
    try:
        cur.execute(query)
        return "Success"
    except TimeoutError:
        return "Timeout!"
```

---

## 💡 Тест:

python

Копировать  Редактировать

```
from unittest.mock import MagicMock
import pytest
from db_module import execute_query

def test_execute_query_timeout():
    mock_cursor = MagicMock()
    mock_cursor.execute.side_effect = TimeoutError("Query took too long")

    result = execute_query(mock_cursor, "SELECT * FROM table")

    assert result == "Timeout!"
    mock_cursor.execute.assert_called_once_with("SELECT * FROM table")
```

---

## 🧠 Что важно понимать:

- `side_effect` — это способ заставить мок выбросить исключение при вызове.
- Это как сказать: "если вызовешь `execute`, я выброшу ошибку!"
- Так можно проверять, как твой код обрабатывает ошибки, не вызывая настоящие сбои.

## Хочешь next level?

Могу показать, как:

- Проверять логирование ошибок
- Проверять повторные попытки (`retry`)
- Проверять коммит и rollback при ошибках в базе

Интересно?

Могу показать, как:

Проверять логирование ошибок

Проверять повторные попытки (`retry`)

Проверять коммит и rollback при ошибках в базе

Интересно, да с подробным объяснением объектов в том числе и  
через дебагер

 Супер! Сейчас покажу продвинутые паттерны тестирования:

1. Проверка логирования ошибок
2. Проверка повторных попыток (`retry`)
3. Проверка `commit` и `rollback` при ошибках

И всё это — с подробным объяснением объектов, плюс пример того, что ты увидишь в дебагере  
(через `print(...)` или `dir(...)`).

---

## 1 Проверка ЛОГИРОВАНИЯ ошибок

## Код с логгером

python

Копировать  Редактировать

```
import logging

logger = logging.getLogger(__name__)

def do_something_risky():
    try:
        raise ValueError("Something went wrong")
    except Exception as e:
        logger.error("Error occurred: %s", e)
```

## Тест с MagicMock

python

Копировать  Редактировать

```
from unittest.mock import MagicMock, patch

def test_log_error():
    with patch('your_module.logger') as mock_logger:
        from your_module import do_something_risky
        do_something_risky()

        mock_logger.error.assert_called_once()
        args, _ = mock_logger.error.call_args
        print("Что было передано в логгер:", args)
```

### Объяснение:

- `patch('your_module.logger')` — подменяем логгер в модуле.
- `mock_logger.error.assert_called_once()` — проверка, что ошибка действительно логировалась.
- `call_args` — покажет, какие аргументы были переданы (`("Error occurred: %s", e)`).

## 2 Проверка ПОВТОРНЫХ попыток (retry)

### 📦 Код с ручным ретрайем

python

Copировать

Редактировать

```
def connect_with_retry(retries=3):
    for attempt in range(retries):
        try:
            return "Connected!"
        except Exception:
            print(f"Retry {attempt + 1}")
    return "Failed"
```

→ Не выбрасывает ошибку — ты сам должен вставить вызов, который может её выбросить.

### 🧪 Тест с side\_effect

python

Copировать

Редактировать

```
from unittest.mock import patch, MagicMock

def test_retry_logic():
    with patch('your_module.connect_with_retry') as mock_func:
        mock_func.side_effect = ["Fail", "Fail", "Connected!"]

        result = [mock_func() for _ in range(3)]
        print("Результаты каждой попытки:", result)
        assert result[-1] == "Connected!"
```

### 🧠 Что ты видишь:

python

Copировать

Редактировать

```
print(dir(mock_func))
print(mock_func.call_count) # = 3
print(mock_func.call_args_list) # список аргументов каждого вызова
```

## 3 Проверка COMMIT и ROLLBACK

📦 Код функции, которая работает с БД:

python

Copировать

Редактировать

```
def save_to_db(conn, data):
    try:
        with conn.cursor() as cur:
            cur.execute("INSERT INTO test VALUES (%s)", (data,))
        conn.commit()
    except Exception:
        conn.rollback()
        raise
```

🧪 Тест с имитацией ошибки

python

Copировать

Редактировать

```
from unittest.mock import MagicMock
import pytest

def test_save_to_db_rollback():
    mock_conn = MagicMock()
    mock_cursor = mock_conn.cursor.return_value.__enter__.return_value
    mock_cursor.execute.side_effect = Exception("DB error")

    with pytest.raises(Exception):
        save_to_db(mock_conn, "test")
```

```
mock_conn.rollback.assert_called_once()
mock_conn.commit.assert_not_called()
print("rollback вызван:", mock_conn.rollback.called)
print("commit вызван:", mock_conn.commit.called)
```

## 🔍 Что увидим через дебаг/печать

python

Copировать

Редактировать

```
print("mock_conn methods:", dir(mock_conn))
print("mock_cursor.execute.side_effect:", mock_cursor.execute.side_effect)
print("commit called:", mock_conn.commit.called)
print("rollback called:", mock_conn.rollback.called)
print("execute вызовы:", mock_cursor.execute.call_args_list)
```

## 🧠 Объяснение объектов:

Объект	Назначение
<code>MagicMock()</code>	Подставной объект, ведет себя как любой другой объект
<code>side_effect</code>	Что произойдет при вызове — исключение, функция или список
<code>call_args_list</code>	История всех вызовов с аргументами
<code>assert_called_once()</code>	Проверка, что был ровно 1 вызов
<code>mock_conn.commit.called</code>	True/False — был ли вызов метода



## Хочешь next level?

Могу показать, как тестировать **автоматический retry** с библиотеками типа `tenacity`, или как подменить SQL-запросы через `mock_cursor.execute.side_effect`.

Скажи, что тебе интереснее?

подменить SQL-запросы через `mock_cursor.execute.side_effect`.

да очень подробно с разъяснениями структура объектов?

Отлично! Сейчас мы подменим SQL-запросы через `mock_cursor.execute.side_effect`. Это позволяет:

- Симулировать разные ответы на разные SQL-запросы
- Эмулировать ошибки на конкретных запросах
- Проверять логику обработки запросов

Я объясню всё подробно — как это работает, какая структура у объектов, и как это выглядит в дебагере.



## Цель

Допустим, у тебя функция, которая выполняет **несколько SQL-запросов**:

### 📦 Реальный код:

python

Копировать

Редактировать

```
def complex_db_workflow(conn):
    with conn.cursor() as cur:
        cur.execute("SELECT 1")
        a = cur.fetchone()[0]
```

```
cur.execute("SELECT 2")
b = cur.fetchone()[0]

return a + b
```

## 💡 Тест: подменяем SQL-запросы через `side_effect`

python

Copировать

Редактировать

```
from unittest.mock import MagicMock
import pytest
from db_module import complex_db_workflow # замените на ваш модуль

def test_complex_db_workflow():
    mock_conn = MagicMock()
    mock_cursor = mock_conn.cursor.return_value.__enter__.return_value

    # Подменим execute: логика будет зависеть от входного запроса
    def execute_side_effect(query, *args, **kwargs):
        print(f"SQL query executed: {query}")
        if query == "SELECT 1":
            mock_cursor.fetchone.return_value = (10,)
        elif query == "SELECT 2":
            mock_cursor.fetchone.return_value = (20,)
        else:
            raise ValueError("Unknown SQL query")

    mock_cursor.execute.side_effect = execute_side_effect

    result = complex_db_workflow(mock_conn)

    assert result == 30
```

## 🔍 Что здесь происходит?

⚙️ `mock_cursor.execute.side_effect = execute_side_effect`

- Каждый раз, когда вызывается `execute(...)`, будет запускаться наша `execute_side_effect()` функция.
  - Внутри неё мы можем настраивать, что вернёт следующий вызов `fetchone()`, в зависимости от SQL.
- 

## 🧠 Структура объектов:

scss

Копировать Редактировать

```
mock_conn (MagicMock)
|
|__ cursor() (MagicMock)
|  |__ __enter__() → mock_cursor
|  |  |__ execute(query) → вызывает execute_side_effect
|  |  |__ fetchone() → возвращает то, что мы задали в side_effect
```

## 📌 Почему так работает?

Пример вызова в коде:

python

Копировать Редактировать

```
cur.execute("SELECT 1")
a = cur.fetchone()[0]
```

Что делает `side_effect`:

```
python
```

Копировать

Редактировать

```
if query == "SELECT 1":  
    mock_cursor.fetchone.return_value = (10,)
```

➡ Это значит, что когда потом вызовется `cur.fetchone()`, он вернёт `(10,)`.

---

## 🔍 Что ты можешь посмотреть в дебагере:

```
python
```

Копировать

Редактировать

```
print("mock_cursor методы:", dir(mock_cursor))  
print("execute вызовы:", mock_cursor.execute.call_args_list)  
print("fetchone вызовы:", mock_cursor.fetchone.call_args_list)  
print("fetchone возвращает:", mock_cursor.fetchone.return_value)
```

## 🔍 Пример вывода:

```
python
```

Копировать

Редактировать

```
SQL query executed: SELECT 1  
SQL query executed: SELECT 2  
mock_cursor.execute.call_args_list:  
[call('SELECT 1'), call('SELECT 2')]
```

## ⚠️ А что если запрос неизвестен?

Мы можем поднять исключение прямо в `side_effect`, например:

```
python
```

Копировать

Редактировать

```
else:  
    raise ValueError("Unknown SQL query")
```

- ➡ Отлично для проверки обработки ошибок в коде, если запрос неправильный.

## ✓ Применение:

Сценарий	Как <code>side_effect</code> помогает
Разные результаты на разные SQL	Смотрим текст запроса — возвращаем нужный результат
Проверка обработки ошибок	Выбрасываем ошибку при определённом SQL
Повторяющиеся запросы	Можно использовать список: <code>side_effect = [res1, res2, ...]</code>

Хочешь, я покажу, как `side_effect` работает со **списком значений**, чтобы эмулировать последовательность результатов (`fetchone()` раз за разом)?

Хочешь, я покажу, как `side_effect` работает со списком значений, чтобы эмулировать последовательность результатов (`fetchone()` раз за разом)? - да

Отлично! Давай посмотрим, как `side_effect` может работать со **списком значений**, чтобы имитировать **последовательные вызовы** `fetchone()` (или любого другого метода).

Это очень удобно, когда:

- Ты вызываешь `fetchone()` или `fetchall()` несколько раз подряд
- Каждый вызов должен возвращать **разные данные**
- Хочешь контролировать поведение каждого шага

---

## 🎯 Сценарий: несколько `fetchone()` вызовов

### 📦 Код функции

python

Copilot Копировать

Редактировать

```
def process_multiple_queries(cur):
    results = []
    for _ in range(3):
        cur.execute("SELECT something")
        results.append(cur.fetchone()[0])
    return sum(results)
```

Функция делает 3 запроса и суммирует результат.

---

## 🧪 Тест с `side_effect` как список

python

Copilot Копировать

Редактировать

```
from unittest.mock import MagicMock

def test_process_multiple_queries():
    mock_cursor = MagicMock()

    # Задаем, что fetchone будет возвращать разные значения при каждом вызове
    mock_cursor.fetchone.side_effect = [(1,), (2,), (3,)]

    result = process_multiple_queries(mock_cursor)

    assert result == 6
    print("Результаты fetchone по порядку:", [call[0] for call in mock_cursor.fetchone.call_args_])
```

---

## 🔍 Как работает `side_effect` со списком?

Если ты установишь:

```
python
```

Копировать    ⚙ Редактировать

```
mock_cursor.fetchone.side_effect = [(1,), (2,), (3,)]
```

То при вызове `fetchone()`:

1. Первый раз вернёт `(1,)`
2. Второй раз — `(2,)`
3. Третий раз — `(3,)`

Потом: `StopIteration` или `IndexError`, если продолжить вызовы.

---

## 🧠 Визуализация

```
python
```

Копировать    ⚙ Редактировать

```
fetchone() calls:
```

1. → returns `(1,)`
2. → returns `(2,)`
3. → returns `(3,)`

➡ Всё это управляется через `side_effect = [val1, val2, val3]`

---

## ⭐ Почему это круто?

Возможность	Пример
Проверка логики с разными данными	Несколько <code>fetchone()</code> с уникальными результатами
Симуляция последовательности	Идеально для пагинации, итераций
Эмуляция поведения курсора	Без настоящей БД

## Что можно посмотреть через дебаг

python

Копировать

Редактировать

```
print("Количество вызовов fetchone:", mock_cursor.fetchone.call_count)
print("Аргументы каждого вызова:", mock_cursor.fetchone.call_args_list)
```

## + Бонус: комбинированный `side_effect`

Ты даже можешь комбинировать значения и исключения:

python

Копировать

Редактировать

```
mock_cursor.fetchone.side_effect = [(1,), Exception("fail"), (3,)]
```

➡ Симулирует:

1. Успех
2. Ошибка
3. Успех

---

Хочешь я покажу, как сделать такой же `side_effect` для `execute()` и `fetchall()` ?

Хочешь я покажу, как сделать такой же `side_effect` для `execute()` и `fetchall()`?  
да но более подробно?

Отлично, сейчас покажу **очень подробно**, как использовать `side_effect` для:

- `cursor.execute(...)` — эмуляция разных SQL-запросов
- `cursor.fetchall()` — разные результаты при каждом вызове

Мы разберём:

- Структуру объектов
- Как работает `side_effect` с функцией и списком
- Пример с `execute()` и `fetchall()` вместе
- Что ты увидишь в дебагере



## Функция: выполнение разных SQL-запросов

python

Копировать

Редактировать

```
def run_batch_queries(cur):
    cur.execute("SELECT * FROM users")
    users = cur.fetchall()

    cur.execute("SELECT * FROM orders")
    orders = cur.fetchall()

    return {
```

```
"users_count": len(users),  
"orders_count": len(orders)  
}
```

## 🎯 Цель теста

- Подменить `execute()` — чтобы **сделать вид**, что выполняются разные запросы
- Подменить `fetchall()` — чтобы вернуть разные результаты для каждого запроса

## 📝 Тест с `side_effect` (подробно)

python

Copировать Редактировать

```
from unittest.mock import MagicMock  
  
def test_run_batch_queries():  
    mock_cursor = MagicMock()  
  
    # -----  
    # 1. Настроим fetchall  
    # -----  
    # Каждый вызов fetchall() будет возвращать следующее:  
    mock_cursor.fetchall.side_effect = [  
        [("user1",), ("user2",)], # ответ на 1-й SELECT  
        [("order1",)]           # ответ на 2-й SELECT  
    ]  
  
    # -----  
    # 2. Настроим execute  
    # -----  
    def execute_side_effect(sql, *args, **kwargs):  
        print("Выполнен запрос:", sql)
```

```
if sql == "SELECT * FROM users":
    pass # никаких действий – результат уже задан в fetchall
elif sql == "SELECT * FROM orders":
    pass
else:
    raise ValueError("Неизвестный запрос")

mock_cursor.execute.side_effect = execute_side_effect

# -----
# 3. Запускаем функцию
# -----
result = run_batch_queries(mock_cursor)

assert result == {"users_count": 2, "orders_count": 1}

# -----
# 4. Проверка и дебаг
# -----
print("call_args_list execute:", mock_cursor.execute.call_args_list)
print("call_args_list fetchall:", mock_cursor.fetchall.call_args_list)
```

## 🔍 Что происходит по шагам?

### 🧠 fetchall.side\_effect = список

python

🔗 Копировать

📝 Редактировать

```
mock_cursor.fetchall.side_effect = [
    [("user1",), ("user2",)], # результатом 1-го запроса
    [("order1",)]           # результатом 2-го запроса
]
```

Каждый `cur.fetchall()` будет возвращать следующий элемент из списка.

## 💡 execute.side\_effect = функция

python

Копировать  Редактировать

```
def execute_side_effect(sql, *args, **kwargs):
    if sql == "SELECT * FROM users":
        pass
    elif sql == "SELECT * FROM orders":
        pass
    else:
        raise ValueError("Неизвестный запрос")
```

### ➡ Полный контроль:

- Можешь логировать
- Можешь проверять SQL
- Можешь эмулировать ошибку

## 🧱 Структура объектов:

scss

Копировать  Редактировать

```
mock_cursor (MagicMock)
|
+-- execute(sql)      → вызывает нашу функцию
+-- fetchall()        → возвращает следующее из side_effect списка
└-- call_args_list   → показывает список всех вызовов (и аргументов)
```

## 🔍 Что покажет дебаг:

python

Копировать

Редактировать

```
print(mock_cursor.execute.call_args_list)
# [call('SELECT * FROM users'), call('SELECT * FROM orders')]

print(mock_cursor.fetchall.call_args_list)
# [call(), call()]
```

## ⚠ Что будет, если `fetchall` вызовется лишний раз?

python

Копировать

Редактировать

```
mock_cursor.fetchall.side_effect = [
    [("user1",)], [("order1",)]
]
cur.fetchall() # ok
cur.fetchall() # ok
cur.fetchall() # ✗ StopIteration
```

Можно избежать этого, используя функцию вместо списка:

python

Копировать

Редактировать

```
def fetchall_side_effect():
    # логика здесь
    return [...]
```

## ✓ Когда использовать список, а когда функцию?

Хочешь...

Используй

Просто последовательные значения

side\_effect = [val1, val2, ...]

Логику в зависимости от запроса/состояния

```
side_effect = функция(...)
```

Возвращать исключения

```
[val1, Exception("error"), val3]
```

---

Хочешь, я покажу как сделать `side_effect` с миксом значений и исключений, или симулировать повторный вызов запроса с разными результатами?

import wrapt\_timeout\_decorator as wtd что это такое и для чего нужно объясни подробно?

---

Отличный вопрос! Давай разберём, что такое `wrapt_timeout_decorator` и зачем он используется.

---



## Что такое `wrapt_timeout_decorator`?

`wrapt_timeout_decorator` — это внешняя Python-библиотека, которая позволяет ограничивать время выполнения функций, т.е. устанавливать тайм-аут. Если функция выполняется дольше заданного времени — она прерывается, и выбрасывается исключение `TimeoutError`.

---

### ❖ Назначение

- Защищает от долгих или зависших операций
- Особенno полезен при работе с:
  - Базами данных (`SELECT`, `INSERT`, ...)

- Веб-запросами
  - Файловыми операциями
  - Любым "долгим" кодом
- 

## Установка

bash

 Копировать

 Редактировать

```
pip install wrapt-timeout-decorator
```

Использует библиотеку `wrapt` под капотом — отсюда и название.

---

## Как работает?

Ты используешь его как декоратор для функции. Пример:

python

 Копировать

 Редактировать

```
import wrapt_timeout_decorator as wtd

@wtd.timeout(2) # Установим таймаут в 2 секунды
def slow_function():
    import time
    time.sleep(3)
    return "Done"
```

### Результат:

- Функция `slow_function()` будет прервана через 2 секунды.
  - Исключение: `wrapt_timeout_decorator.TimeoutError`.
-

## Внутри: что делает `@wtd.timeout`

Этот декоратор:

1. Запускает твою функцию в **отдельном потоке или процессе**
  2. Следит за временем выполнения
  3. Если превышено — вызывает `TimeoutError`
- 



## Аргументы

Аргумент	Описание
<code>seconds</code>	Таймаут в секундах (например, <code>timeout(5)</code> )
<code>use_signals</code>	Если <code>True</code> , использует сигналы (только в Unix)
<code>timeout_exception</code>	Кастомный тип ошибки вместо <code>TimeoutError</code>



## Пример из реального кода

python

Копировать

Редактировать

```
@wtd.timeout(3, use_signals=True)
def run_query(cur, sql):
    cur.execute(sql)
    return cur.fetchall()
```

Если запрос к базе зависнет дольше 3 секунд — он будет прерван и будет выброшена ошибка `TimeoutError`.

## ⚠ Важно:

Особенность	Детали
<code>use_signals=True</code>	Работает только в Unix-системах (Linux, macOS)
Потокобезопасность	Лучше использовать в однопоточном коде
Работает с обычными функциями	Да, не обязательно <code>async</code>

## 📌 Где полезно использовать:

- Защита критических операций в проде
- API, которые могут зависнуть
- Работа с базами, где нет встроенного таймаута
- Бизнес-логику с долгими циклами

## 🎯 Пример обработки таймаута

python

Copировать

Редактировать

```
try:  
    result = run_query(cursor, "SELECT * FROM users")  
except wtd.TimeoutError:
```

```
print("Запрос превысил лимит времени!")
```

---

Хочешь пример, где декоратор применяется в тесте для симуляции таймаута при SQL-запросе (и проверка его обработки)?

Хочешь пример, где декоратор применяется в тесте для симуляции таймаута при SQL-запросе (и проверка его обработки)?  
- да

Отлично, давай создадим пример, где:

- Есть функция с SQL-запросом, обёрнутая в `@wrapt_timeout_decorator.timeout`
- В тесте мы симулируем тайм-аут
- Проверяем, что функция корректно обрабатывает `TimeoutError`



## 1. Реальная функция: SQL с тайм-аутом

python

Копировать

Редактировать

```
import wrapt_timeout_decorator as wtd

@wtd.timeout(2, use_signals=False) # тайм-аут 2 секунды
def run_query_with_timeout(cursor, query):
    cursor.execute(query)
    return cursor.fetchall()
```

Мы оборачиваем `run_query_with_timeout`, чтобы прервать её, если `cursor.execute()` выполняется дольше 2 секунд.

---

## 2 Подготовим функцию, которую мы будем тестировать

python

Копировать

Редактировать

```
def safe_query(cursor, query):
    try:
        return run_query_with_timeout(cursor, query)
    except wtd.TimeoutError:
        return "Timeout!"
```



## 3. Тест с симуляцией долгого запроса

python

Копировать

Редактировать

```
from unittest.mock import MagicMock
import pytest
import time
import wrapt_timeout_decorator as wtd
from db_module import safe_query # заменить на свой модуль

def test_safe_query_timeout():
    mock_cursor = MagicMock()

    # Симулируем "зависший" запрос
    def long_running_execute(query):
        time.sleep(5) # больше таймаута (2 сек)
        return

    mock_cursor.execute.side_effect = long_running_execute

    result = safe_query(mock_cursor, "SELECT * FROM users")
```

```
assert result == "Timeout!"  
print("Тест прошел. Ответ:", result)
```

## 🔍 Что происходит по шагам:

Этап	Что делает
<code>mock_cursor.execute</code>	Подменяется, чтобы "зависнуть" на 5 секунд
<code>run_query_with_timeout</code>	Установлен тайм-аут: 2 секунды
<code>time.sleep(5)</code>	Симулирует долгий SQL-запрос
Тайм-аут срабатывает	<code>wrapt_timeout_decorator.TimeoutError</code> выбрасывается
<code>safe_query()</code>	Ловит исключение и возвращает "Timeout!"

## 🧠 Что можно напечатать для дебага:

```
python
```

Копировать    ⚙ Редактировать

```
print("Был ли вызван execute:", mock_cursor.execute.called)  
print("Сколько раз вызван:", mock_cursor.execute.call_count)  
print("Аргументы вызова:", mock_cursor.execute.call_args_list)
```

## Альтернатива: Тест без sleep (сразу выбрасываем TimeoutError)

Если не хочешь `sleep`, можно так:

python

Копировать

Редактировать

```
mock_cursor.execute.side_effect = wtd.TimeoutError("Запрос завис")
```

➡ И всё равно получишь тот же результат — очень удобно и быстрее.

---

Хочешь, я покажу, как использовать этот декоратор с `pytest.mark.parametrize` для проверки разных тайм-аутов и запросов?



Как бы вы оценили этот GPT на данный момент? X



Спросите что-нибудь...

