**1. Что такое @patch?**

python

КопироватьРедактировать

@patch('task\_manager.manager.datetime.date')

* **@patch** — это декоратор из библиотеки **unittest.mock**.
* Декоратор @patch используется для замены объектов в коде на **мок-объекты** во время тестирования. В данном случае он используется для того, чтобы заменить datetime.date на мок-объект, который позволит контролировать текущую дату в тестах.

Вместо того чтобы использовать реальную текущую дату (которую можно получить через datetime.date.today()), мы будем подменять её на фиксированное значение. Это нужно, чтобы тесты были предсказуемыми и не зависели от настоящей текущей даты.

**2. def test\_get\_overdue\_tasks\_with\_mocked\_today(self, mock\_date):**

Здесь начинается метод теста, и его имя указывает, что мы проверяем получение просроченных задач с подмененной датой.

* **self** — ссылка на текущий экземпляр класса TestTaskManager, в котором будет выполняться тест.
* **mock\_date** — это параметр, который будет передан в тестовую функцию, и это **мок-объект**, замещающий datetime.date. Он создается автоматически благодаря декоратору @patch.

**3. mock\_date.today.return\_value = date(2024, 1, 10)**

python

КопироватьРедактировать

mock\_date.today.return\_value = date(2024, 1, 10)

* Мы подменяем метод today() из datetime.date на фиксированное значение: **date(2024, 1, 10)**.
* Это значит, что в тесте вызов mock\_date.today() всегда будет возвращать дату 10 января 2024 года, независимо от реальной текущей даты.

**4. manager = TaskManager()**

python

КопироватьРедактировать

manager = TaskManager()

* Создается новый экземпляр класса **TaskManager**. Этот менеджер задач будет использоваться для добавления задач и получения просроченных задач.

**5. manager.add\_task("Old task", date(2024, 1, 5))**

python

КопироватьРедактировать

manager.add\_task("Old task", date(2024, 1, 5))

* Мы добавляем задачу с названием "Old task" и датой дедлайна **5 января 2024 года**.
* Это значит, что эта задача должна считаться просроченной, если сегодняшняя дата — 10 января 2024 года.

**6. overdue = manager.get\_overdue\_tasks(mock\_date.today())**

python

КопироватьРедактировать

overdue = manager.get\_overdue\_tasks(mock\_date.today())

* Мы вызываем метод **get\_overdue\_tasks**, который должен вернуть все задачи с просроченным дедлайном. Для этого мы передаем в него **мок-дату** (которую мы установили на 10 января 2024 года).
* Метод **get\_overdue\_tasks** будет проверять, есть ли задачи, у которых дедлайн меньше сегодняшней даты и которые ещё не выполнены.
* Поскольку мы добавили задачу с дедлайном 5 января, она должна быть признана просроченной.

**7. self.assertEqual(len(overdue), 1)**

python

КопироватьРедактировать

self.assertEqual(len(overdue), 1)

* Мы проверяем, что список **overdue** содержит ровно одну задачу. То есть ожидается, что задача с дедлайном 5 января будет просрочена и попадет в список просроченных задач.

python -m task\_manager.main

python -m unittest discover -s task\_manager/tests:

1. **python -m**: Запускает Python-модуль как скрипт.
2. **unittest**: Модуль для тестирования в Python.
3. **discover**: Команда для автоматического поиска тестов.
4. **-s task\_manager/tests**: Параметр -s указывает на начальную директорию для поиска тестов (в данном случае, папка task\_manager/tests).
5. **Кто и когда проводит юнит-тестирование?**

Модульное тестирование (Unit Testing) – это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения. Цель модульного тестирования – проверить, что каждая единица программного кода работает должным образом.

• Модульное тестирование это метод тестирования WhiteBox, который обычно выполняется разработчиками на этапе кодирования приложения.

1. **Перечислите преимущества и недостатки юнит-тестирования.**

Преимущества юнит-тестирования:

1. Раннее обнаружение ошибок: Тесты помогают выявить ошибки на ранних этапах разработки, что снижает затраты на их исправление.
2. Упрощение рефакторинга: Наличие тестов помогает безопасно изменять код, проверяя, что функциональность не нарушена.
3. Документация: Юнит-тесты служат документацией для кода, демонстрируя, как должны работать отдельные компоненты.
4. Повышение надежности: Систематическое тестирование компонентов помогает повысить стабильность программы.
5. Автоматизация: Тесты могут быть автоматизированы, что ускоряет процесс разработки и интеграции.

Недостатки юнит-тестирования:

1. Затраты времени: Написание и поддержка тестов требует времени, особенно для сложных систем.
2. Не охватывает весь функционал: Юнит-тесты проверяют только отдельные компоненты, но не гарантируют правильность работы всей системы.
3. Не исключает всех ошибок: Тесты могут не выявить ошибки, связанные с интеграцией, производительностью или пользовательским интерфейсом.
4. Избыточность: В некоторых случаях можно писать тесты, которые не добавляют ценности или не тестируют реально важные части системы.
5. Проблемы с поддержкой: Если код изменяется часто, тесты могут потребовать постоянных изменений и обновлений.

**3. Расскажите структуру юнит-теста.**

🔧 1. Тестовые фикстуры – подготовка среды

Это всё, что нужно настроить перед тестом:

* подключение к БД,
* создание тестовых данных (например, пост блога),
* временные файлы,
* конфигурации,
* тестовые пользователи и изолированная "песочница",  
  чтобы тест не затрагивал реальную систему.

✅ 2. Тест-кейс – сам тест

Это код, который проверяет работу конкретной функции или метода.  
Внутри обязательно есть assert-ы – они сравнивают фактический результат с ожидаемым.

▶️ 3. Test Runner – запуск тестов

Это инструмент, который находит, запускает и показывает результаты тестов.  
Он может сортировать, группировать тесты, запускать только важные и управлять средой.

📊 4. Тестовые данные – входные значения

Нужно проверять на:

* нормальные случаи – обычный ввод,
* граничные случаи – минимумы/максимумы,
* ошибки – некорректные данные,
* редкие случаи – особые или экстремальные значения.

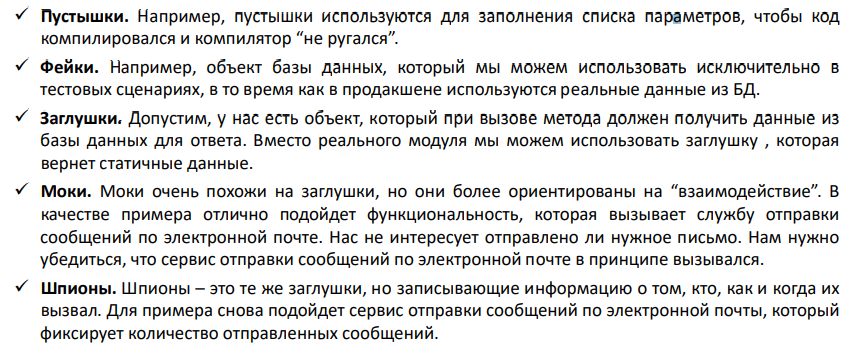
🧪 5. Моки – подмены зависимостей

Если код зависит от других компонентов (например, отправка email),  
вместо настоящих объектов используем моки – "подделки",  
которые имитируют поведение без выполнения реционных действий.

**4. Для чего проводится изоляция тестируемых функций?**

Разработчик может изолировать единицу кода для более качественного тестирования. Эта практика подразумевает копирование кода в собственную среду тестирования. Изоляция кода помогает выявить ненужные зависимости между тестируемым кодом и другими модулями или пространствами данных в продукте. Если класс имеет зависимость от другого класса или нескольких классов, все такие зависимости должны быть заменены на тестовые заглушки (test doubles). Это позволяет сосредоточиться исключительно на тестируемом классе, изолировав его поведение от внешнего влияния. Тестовая заглушка (test double) — объект, который выглядит и ведет себя как его рабочий аналог, но в действительности представляет собой упрощенную версию, более удобную для тестирования

**5. Какие методы изоляции вы знаете?**

****

**6. Как рассчитать покрытие кода тестами?**

