Находясь в корневой папке проекта (lab7\_unit\_testing), выполняем команду:

coverage run -m unittest discover tests/

Краткий отчет в консоли:

coverage report -m

Подробный HTML-отчет:

coverage html

**1 Кто и когда проводит юнит-тестирование?**

Кто: В основном, разработчики программного обеспечения. Иногда специализированные инженеры по автоматизации тестирования (SDET).

Когда: Во время процесса разработки. Юнит-тесты пишутся параллельно с основным кодом (или даже до него, если используется Test-Driven Development - TDD). Они запускаются часто: после написания новой функции, после внесения изменений, перед коммитом кода, в системах непрерывной интеграции (CI).

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

**Преимущества:**

* Обнаружение ошибок на ранней стадии: Позволяет найти баги в отдельных компонентах до их интеграции, что дешевле исправлять.
* Улучшение дизайна кода: Написание тестов заставляет думать о модульности, слабой связанности и тестируемости кода.
* Безопасный рефакторинг: Тесты служат "сетью безопасности", позволяя уверенно изменять и улучшать код, проверяя, что существующая функциональность не сломана.
* Документация: Тесты показывают, как предполагается использовать код модуля/функции.
* Ускорение разработки в долгосрочной перспективе: Хотя написание тестов требует времени, оно экономит время на отладке и исправлении ошибок позже.
* Автоматизация: Тесты можно запускать автоматически и часто.

**Недостатки:**

* Требует времени: Написание и поддержка тестов требует дополнительных усилий и времени разработчиков.
* Не гарантирует отсутствие ошибок: Юнит-тесты проверяют компоненты в изоляции, они не могут выявить ошибки интеграции между модулями или системные проблемы.
* Сложность тестирования некоторого кода: Код с сильными зависимостями (GUI, базы данных, сеть, файловая система) сложнее тестировать в изоляции, требует моков и заглушек.
* Возможность написать "плохие" тесты: Тесты могут быть неполными, проверять не то, что нужно, или быть слишком хрупкими (ломаться от незначительных изменений в коде).
* Поддержка: Тесты нужно обновлять вместе с изменением кода.

**3 Расскажите структуру юнит-теста.**  
 Классическая структура юнит-теста часто описывается паттерном AAA (Arrange-Act-Assert) или Setup-Execute-Assert-Teardown:

* Arrange (Подготовка / Setup): На этом этапе создаются и настраиваются все необходимые объекты, входные данные, моки (mocks) или заглушки (stubs). Подготавливается "сцена" для теста. В unittest это может быть часть самого тестового метода или метод setUp().
* Act (Действие / Execute): Выполняется тестируемый метод или функция с подготовленными входными данными. Обычно это одна строка кода.
* Assert (Проверка / Утверждение): Проверяется результат выполнения действия. Сравнивается фактический результат (полученный на шаге Act) с ожидаемым результатом. Используются специальные функции утверждений (assertEqual, assertTrue, assertRaises и т.д.). Если проверка не проходит, тест считается проваленным.
* (Опционально) Teardown (Очистка): После выполнения теста происходит очистка ресурсов, если они были созданы на шаге Arrange (например, закрытие файлов, удаление временных данных, откат транзакций). В unittest для этого используется метод tearDown().

**4 Для чего проводится изоляция тестируемых функций?**  
Изоляция — ключевой принцип юнит-тестирования. Она проводится для того, чтобы:

* Тестировать только одну единицу (unit): Убедиться, что тест проверяет логику именно тестируемого модуля/функции/класса, а не его зависимостей.
* Избежать побочных эффектов: Зависимости (например, база данных, сеть, файловая система) могут изменять состояние системы, что сделает тесты непредсказуемыми и зависимыми от окружения. Изоляция предотвращает это.
* Ускорить выполнение тестов: Взаимодействие с реальными зависимостями (особенно внешними системами) обычно медленное. Замена их быстрыми имитациями (моками) значительно ускоряет прогон тестов.
* Сделать тесты детерминированными: Результат теста не должен зависеть от внешних факторов (например, доступности сетевого сервиса или содержимого файла). Изоляция помогает этого достичь.
* Упростить настройку теста: Не нужно настраивать сложное окружение (БД, веб-серверы) для запуска простого юнит-теста.
* Локализовать ошибки: Если тест падает, изоляция гарантирует, что проблема, скорее всего, находится внутри тестируемого юнита, а не в его зависимостях.

**5 Какие методы изоляции вы знаете?**  
 Основные методы (часто называемые "Test Doubles" - тестовые двойники):

* Dummy (Пустышка): Объекты, которые передаются в качестве аргументов, но реально не используются. Нужны только для удовлетворения сигнатуры метода.
* Fake (Подделка): Объекты с работающей реализацией, но упрощенной по сравнению с реальным объектом. Они не используют внешние ресурсы (например, Fake-репозиторий, работающий с данными в памяти вместо БД).
* Stub (Заглушка): Предоставляют заранее заготовленные ответы на вызовы во время теста. Используются, когда нужно проверить, как тестируемый объект реагирует на определенные данные от зависимости. Не проверяют само взаимодействие.
* Mock (Имитация/Мок): Объекты, которые регистрируют вызовы, сделанные к ним во время теста. Они позволяют проверить, как тестируемый объект взаимодействует со своими зависимостями (какие методы вызывались, с какими параметрами, сколько раз). Часто включают функциональность заглушек (могут возвращать предопределенные значения). Библиотеки типа unittest.mock в Python предоставляют мощные инструменты для создания моков.
* Spy (Шпион): Похожи на моки, но оборачивают реальный объект, записывая информацию о взаимодействии с ним, но при этом позволяя реальному методу выполниться (если не настроено иное). Реже используются и часто реализуются поверх моков.

**6 Как рассчитать покрытие кода тестами?**  
 Покрытие кода (Code Coverage) — это метрика, показывающая, какая часть исходного кода программы была выполнена во время запуска тестов. Обычно измеряется в процентах.

Формула (упрощенно для Line Coverage):  
Покрытие (%) = (Количество выполненных строк / Общее количество исполняемых строк) \* 100

**Как рассчитать:**

* Инструментация: Используются специальные инструменты (например, coverage.py для Python, JaCoCo для Java, Istanbul для JavaScript), которые модифицируют код (или байт-код) перед выполнением тестов, добавляя счетчики для каждой исполняемой строки (или ветки, инструкции).
* Выполнение тестов: Тесты запускаются с инструментированным кодом. Во время выполнения счетчики фиксируют, какие строки были затронуты.
* Генерация отчета: После завершения тестов инструмент анализирует собранные данные и генерирует отчет, показывающий процент покрытия и конкретные строки/участки кода, которые не были выполнены.

**Типы покрытия**: Существуют разные метрики покрытия:

* Line Coverage (Покрытие строк): Сколько исполняемых строк кода было выполнено. Самая простая и распространенная метрика.
* Branch Coverage (Покрытие ветвей): Сколько ветвей условных операторов (if, while) было выполнено (истинно и ложно). Более строгая метрика, чем Line Coverage.
* Function/Method Coverage: Сколько функций или методов было вызвано.
* Statement Coverage (Покрытие операторов): Похоже на Line Coverage, но считает отдельные операторы (одна строка может содержать несколько операторов).  
  Инструменты обычно позволяют выбрать или комбинировать эти метрики.