Мы сегодня поговорим про тестирование веб-приложений. Посмотрим, какие библиотеки можно использовать для тестирования и какие дополнительные инструменты можно использовать для тестирования.

Цели тестирования

Первая и главная цель - проверка правильности реализации. Вы знаете, что должна делать ваша система и проверяете, что результат выполнения функции соответствует ожидаемому значению.

Вторая цель - проверка обработки нештатных ситуаций и граничных условий. Здесь проверяется граничные условия (пустая или невалидная строка).

Третья цель - это минимизация последствий. Правильный менеджер знает, что на тестирование тоже должно быть выделено какой-то объём времени.

Виды тестирования (1)

Начнём мы с видом тестирований. Что такое unittest? Отдельные, автономные часть системы, что это может быть?

Дальше идёт функциональное тестирование. Тестирование какой-то функциональности. Эта функциональность может состоять из нескольких функций.

Интеграционное тестирование – это тип тестирования, при котором программные модули объединяются логически и тестируются как группа. Как правило, программный продукт состоит из нескольких программных модулей, написанных разными программистами. Целью нашего тестирования является выявление багов при взаимодействии между этими программными модулями и в первую очередь направлен на проверку обмена данными между этими самими модулями.

Нагрузочное тестирование, тестирование под нагрузкой.

И ещё одна классификация это тестирование клиентской и серверной части. Что касается клиентской части. Есть несколько подходов. Можно писать тесты клиентской части на бекенде. И можем использовать такой инструмент как Selenium. Селениум запускает так называемым веб-браузер, и веб-браузер выполняет ваш скрипт.

Виды тестирования (2)

Test driven development
Техника разработки программного обеспечения. Пишется сначала тест, который покрывается какой-то функционал, а потом пишется сам код.

TDD. Алгоритм

В чём **минусы** такого подхода?

- Существуют задачи, которые невозможно решить только при помощи тестов. В частности, TDD не позволяет механически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами. Безусловно, безопасность основана на коде, в котором не должно быть дефектов, однако она основана также на участии человека в процедурах защиты данных. Тонкие проблемы, возникающие в области взаимодействия между процессами, невозможно с уверенностью воспроизвести, просто запустив некоторый код.
- Разработку через тестирование сложно применять в тех случаях, когда для тестирования необходимо прохождение функциональных тестов. Примерами может быть: разработка интерфейсов пользователя, программ, работающих с базами данных, а также того, что зависит от специфической конфигурации сети. Разработка через тестирование не предполагает большого объёма работы по тестированию такого рода вещей. Она сосредотачивается на тестировании отдельно взятых модулей, используя mock-объекты для представления внешнего мира.
- Требуется больше времени на разработку и поддержку, а одобрение со стороны руководства очень важно. Если у организации нет уверенности в том, что разработка через тестирование улучшит качество продукта, то время, потраченное на написание тестов, может рассматриваться как потраченное впустую.[9]
- Модульные тесты, создаваемые при разработке через тестирование, обычно пишутся теми же, кто пишет тестируемый код. Если разработчик неправильно истолковал требования к приложению, и тест, и тестируемый модуль будут содержать ошибку.
- Большое количество используемых тестов может создать ложное ощущение надежности, приводящее к меньшему количеству действий по контролю качества.
- Тесты сами по себе являются источником накладных расходов. Плохо написанные тесты, например, содержат жёстко вшитые строки с сообщениями об ошибках или подвержены ошибкам, дороги при поддержке. Чтобы упростить поддержку тестов, следует повторно использовать сообщения об ошибках из тестируемого кода.

- Уровень покрытия тестами, получаемый в результате разработки через тестирование, не может быть легко получен впоследствии. Исходные тесты становятся всё более ценными с течением времени. Если неудачные архитектура, дизайн или стратегия тестирования приводят к большому количеству непройденных тестов, важно их все исправить в индивидуальном порядке. Простое удаление, отключение или поспешное изменение их может привести к необнаруживаемым пробелам в покрытии тестами.

Плюсы данного подхода:
Разработка через тестирование предлагает больше, чем просто проверку корректности, она также влияет на дизайн программы. Изначально сфокусировавшись на тестах, проще представить, какая функциональность необходима пользователю. Таким образом, разработчик продумывает детали интерфейса до реализации. Тесты заставляют делать свой код более приспособленным для тестирования.

Несмотря на то, что при разработке через тестирование требуется написать большее количество кода, общее время, затраченное на разработку, обычно оказывается меньше. Тесты защищают от ошибок. Поэтому время, затрачиваемое на отладку, снижается многократно.

Selenium (1)
Selenium (2)
Виды тестирования. Selenium

Степень покрытия тестами (test coverage)
По факту это отношение покрытому тестами код ко всему коду. Обычно проверяется степень покрытия тестами с помощью библиотечки coverage. Есть общий случай, а есть для джанго случай.

С ростом проекта определить, какой код протестирован, а какой нет, становится сложно, хотя подобная потребность возникает регулярно. Обычно это происходит тогда, когда в команде есть разные люди, и не все из них ответственно подходят к написанию тестов. В таком случае может страдать качество проекта.

Покрытие анализируется тестовыми фреймворками, которые считают отношения строчек, задействованных в тестах, ко всем строчкам исходного

кода. Например, если в коде есть условная конструкция, и она не проверяется тестами, это значит, что все строки кода, входящие в неё, не будут покрыты.

Coverage генерирует отчёт. И отчёты можно посмотреть в html-виде.

Так же в зависимости от того, что покрывать тестами, есть такое понятие, как дымовой тест (Smoke testing или smoke test, дымовое тестирование), что означает минимальный набор тестов на явные ошибки. Дымовой тест обычно выполняется программистом; не проходившую этот тест программу не имеет смысла отдавать на более глубокое тестирование.

Первое своё применение этот термин получил у печников, которые, собрав печь, закрывали все заглушки, затапливали её и смотрели, чтобы дым шёл только из положенных мест.

Повторное «рождение» термина произошло в радиоэлектронике. Первое включение нового радиоэлектронного устройства, пришедшего из производства, совершается на очень короткое время (меньше секунды). Затем инженер руками ощупывает все микросхемы на предмет перегрева. Сильно нагревшаяся за эту секунду микросхема может свидетельствовать о грубой ошибке в схеме.

Инструменты для тестирования в Python
Поговорим про инструменты для тестирования. Библиотек множество, но мы поговорим про несколько популярных.

doctest
Это библиотечка, есть в стандартной библиотеке, она позволяет писать тесты в docstring.

Модуль doctest ищет куски текста, которые выглядят как интерактивные сессии Python и затем выполняет эти сессии, чтобы проверить, что они работают точно так же, как показано. Есть несколько стандартных причин, чтобы использовать doctest:

- Для того, чтобы проверить актуальность строк документации, убедившись, что все интерактивные примеры работают именно так, как задокументировано.
- Чтобы организовать регрессионное тестирование, проверяя, что интерактивные примеры из тестового файла или тестового объекта работают как ожидается.

- Чтобы написать руководство для пакета, иллюстрированное примерами ввода-вывода. В зависимости от того, на что обращается внимание - на примеры или на пояснительный текст, это можно назвать либо “литературным тестированием”, либо “исполняемой документацией”.

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

pytest
Это одна из двух популярных библиотек для unit-тестирования. Используется в этой библиотеки assert.

Преимущества Pytest заключаются в следующем —

- pytest может выполнять несколько тестов параллельно, что сокращает время выполнения набора тестов.
- у pytest есть собственный способ автоматического определения тестового файла и тестовых функций, если это не указано явно.
- pytest позволяет нам пропустить подмножество тестов во время выполнения.
- pytest позволяет нам запускать подмножество всего набора тестов.
- pytest является бесплатным и открытым исходным кодом.
- благодаря простому синтаксису, pytest очень прост для запуска.

hypothesis
Переводится как гипотеза. Эту библиотеку не видел никогда в продакшене. Эта библиотечка довольно магическая. Есть такой декоратор given, которому подаётся какой-то текст.

Функция text возвращает то, что Hypothesis называет стратегией поиска. Объект с методами которые описывают, как произвести и упростить некоторые виды значений. Затем декоратор @given берет наш тест функции и превращает его в параметризованный, который при вызове будет выполнять тестовую функцию по широкому диапазону совпадающих данных из этой стратегии.

unittest (1)
В Джанго она идёт по умолчанию, она служит для юнит-тестирования, но в джанго есть некоторая обёртка над unittest. Поэтому мы импортируем классы не из библиотеки unittest, а из джанги.

Структура теста представляет из себя 4 основных концептах: testRunner, вторая это фикстура, некие данные которые подаются на вход, дальше testSuit это набор тестов, объединённых общей идеей, и последняя часть - testCase это собственно один из тестов, объединённых общей идеей.

Но в джанго всё немного перепутано, мы используем класс testcase, именно от него наследуются все наборы этих тестов.

unit (2)
Для каждого приложения тесты в отдельном файлике tests.py. Что будет внутри TestCase? Методы setUp() и tearDown() позволяют определять инструкции, выполняемые перед и после каждого теста, соответственно.

unittest. Расширенный набор проверок assert

unittest и Django
Когда Джанго запускает тесты она создаёт тестовую БД. Выполняются миграции, если они есть и БД будет очищаться или удаляться после каждого testCase.

После каждого теста Джанго откатывает транзакции до последней строки метода setUp.

Тестирование в Django. unittest

mock (1)
Про следующий инструмент поговорим про мок. Это такая библиотека, которая помогает подменять объекты - функции или классы - на заглушки. Из этой библиотеки мы используем декоратор patch, который позволяет подменить реально существующий объект, на какой-то фейковый объект.

Помните, что mock позволяет обеспечить так называемую поддельную реализацию той части вашей программы, которую вы тестируете. Это дает вам больше «гибкости» во время проведения тестов.

mock (2)
Побочные эффекты
Давайте вернемся к нашей функции sum. Что делать, если вместо жесткого кодирования возвращаемого значения мы хотим запустить измененную

функцию sum? Наша измененная функция mock будет долго выполнять time.sleep(), что нежелательно для нас, и останется только с изначальной функциональностью (суммой), которую мы хотим проверить. Мы можем просто определить side_effect в нашем тесте.

```
from unittest import TestCase
from unittest.mock import patch

def mock_sum(a, b):
    # mock sum function without the long running time.sleep
    return a + b

class TestCalculator(TestCase):
    @patch('main.Calculator.sum', side_effect=mock_sum)
    def test_sum(self, sum):
        self.assertEqual(sum(2,3), 5)
        self.assertEqual(sum(7,3), 10)
```

factory_boy

Factory boy возникла от библиотеки factory girl из руби. Эта библиотека, которая нужна для генерации тестовых объектов. Если мы говорили про гипотезы, то она тоже служит для конфигурации, но она не такая гибкая.

Само название factory говорит о том, что используется паттерн фабрика. Мы объявляем какую-то фабрику.

Наследует от factory.Factory, прописываем на какую модель будет смотреть наш фейковый класс. И библиотека сама сгенерирует нам какое-то имя. Там есть набор каких-то имён, каких-то фамилий, адреса.

factory_boy, провайдеры

factory_boy

Запуск тестов. Общий случай

В джанго библиотека unittest интегрирована.

Запуск тестов. Django

Оптимизация. Тестовое окружение (1)

Можно настроить верболизацию для ваших тестов. Они проходят по умолчанию в тихом режиме. Но можно настроить, чтобы писался каждый этап, сколько миграций выполнилось.

Для быстрого обнаружения –fail_fast тогда выполнение тестов остановится при первом провалившемся тесте.

Можно использовать параметр –keepdb чтобы не запускать миграции при каждом запуске и тестовая БД сохраняется между запусками.

И можно распараллелить запуск на несколько потоков, это с помощью parallel.

Оптимизация. Тестовое окружение (2)
Можно так же изменять настройки, если

Фикстуры (1)
Что такое фикстуры? Наборы данных, которые мы обычно представляем в json. Как вы можете видете, здесь указано, на какую модель она ссылается, pk и какие поля. Для чего это нужно?

Это ещё один вариант заполнения данных.

Фикстуры (2)
Фикстуры (3)
Фикстуры (4)
freezegun (1)
freezegun (2)