Как писать профессиональные модульные тесты на Python

Тестирование, это основа серьёзной разработки программного обеспечения. Существует много видов тестирования, но наиболее важный вид, это модульное тестирование. Модульное тестирование даёт уверенность в том, что вы сможете использовать хорошо протестированные блоки в качестве базовых элементов, полагаться на них и использовать при создании программы. Они увеличивают ваш инструментарий из проверенного кода за пределами ваших конструкций и стандартной библиотеки. Кроме того Python предоставляет отличную поддержку для написания модульных тестов.

### Основные идеи

Написание хороших модульных тестов это тяжелый труд. Написание модульных тестов занимает время. Когда вы меняете код, необходимо изменять тесты. Иногда в вашем тесте будут ошибки. Это означает, что вы должны быть по-настоящему идейным. Польза огромна, даже для небольших проектов, но это не бесплатно.

### Будьте дисциплинированы

Вы должны быть дисциплинированным. Будьте последовательным. Убедитесь, что все тесты выполнены. Не отказывайтесь от тестов, только потому что вы «знаете», что код в порядке.

### Автоматизируйте

Чтобы помочь вам быть дисциплинированным, необходимо автоматизировать модульные тесты. Тесты должны запускаться автоматически на значимых этапах, таких как проектирование или развертывание. В идеале ваша система управления версиями должна отклонять код, который не прошел все тесты.

### Непротестированный код плохой по определению

Если вы не проверили его, вы не сможете сказать, что он работает. Это значит, что вы должны рассматривать его как плохой. Если это критический код, не разворачивайте его в производство.

## Пояснения

### Что такое модуль?

Модуль в смысле тестирования это файл, содержащий набор определённых функций или класс. Если у вас есть файл с несколькими классами, вы должны написать модульный тест для каждого из них.

### Делать TDD или не делать TDD

[Тест драйв разработка](https://en.wikipedia.org/wiki/Test-driven_development), это практика, где вы пишете тесты, до того как вы пишете код. Есть несколько преимуществ этого подхода, но я рекомендую отказаться от него, если у вас есть возможность написать тесты позже.

Причина заключается в том, что я проектирую код. Я пишу код, смотрю на него, переписываю, еще раз смотрю и быстро переписываю ещё раз. Написание тестов сначала ограничивает и замедляет меня.

После того, как я сделаю первоначальный дизайн, я сразу напишу тесты, до интеграции со всей системой. Тем не менее, это отличный способ найти себя в создании модульных тестов, и это гарантирует, что весь ваш код будет проверен.

### Запуск тестов

OK. Вы написали модульные тесты. Для большой системы у вас будет десятки / сотни / тысячи модулей и классов, возможно,размещенных в разных папках. Как вы будете запускаете все тесты?

Модуль unittest дает различные возможности для проведения групповых тестов и их программирования. Проверка [Загрузки и Выполнения тестов /Loading and Running Tests.](https://docs.python.org/2/library/unittest.html" \l "loading-and-running-tests)" \t "_self) Но самый простой способ - открытие теста. Данный параметр был добавлен только в Python 2.7. В Pre-2.7 вы могли использовать [nose](http://nose.readthedocs.org/en/latest/" \t "_self), чтобы найти и запустить тесты. У nose есть несколько других преимуществ, таких как запуск тестовых функций без необходимости создания класса для ваших тестовых случаев. Но для целей в этой статьи, давайте придерживаться unittest.

Чтобы найти и запустить тесты на основе unittest, просто введите в командной строке:

python -m unittest discover

Unittest будет проверять все файлы и подкаталоги, запускать все найденные тесты и обеспечит хороший отчет, а также покажет время выполнения. Если вы хотите увидеть, какие тесты выполняются, вы можете добавить флаг -v:

python -m unittest discover -v

Существует несколько флагов, которые управляют операцией:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | python -m unittest -h    Usage: python -m unittest [options] [tests]        Options:      -h, --help       Show this message      -v, --verbose    Verbose output      -q, --quiet      Minimal output      -f, --failfast   Stop on first failure      -c, --catch      Catch control-C and display results      -b, --buffer     Buffer stdout and stderr during test runs        Examples:      python -m unittest test\_module               - run tests from test\_module      python -m unittest module.TestClass          - run tests from module.TestClass      python -m unittest module.Class.test\_method  - run specified test method        [tests] can be a list of any number of test modules, classes and test    methods.        Alternative Usage: python -m unittest discover [options]        Options:      -v, --verbose    Verbose output      -f, --failfast   Stop on first failure      -c, --catch      Catch control-C and display results      -b, --buffer     Buffer stdout and stderr during test runs      -s directory     Directory to start discovery ('.' default)      -p pattern       Pattern to match test files ('test\*.py' default)      -t directory     Top level directory of project (default to                       start directory)        For test discovery all test modules must be importable from the top    level directory of the project. |

### Определение степени покрытия кода

Определение степени покрытия кода часто игнорируют. Само понятие означает, сколько кода действительно проверено тестами. Например, если у вас есть функция с инструкцией if-else, и вы проверяете только ветвьif, то вы не знаете, работает ли ветка else или нет.  В следующем примере кода функция add() проверяет тип своих аргументов. Если оба являются целыми числами, они просто добавляют их.

Если оба являются строками, то он пытается преобразовать их в целые числа и добавить. В противном случае он вызывает исключение. Функция test\_add() проверяет функцию add() с аргументами, которые являются как целыми числами, так и аргументами, которые являются плавающим значением и проверяют правильное поведение в каждом случае. Но степень покрытия теста является незавершенной. В случае, если строковые аргументы были не протестированы. В результате тест проходит успешно, но ошибка в ветке, где аргументы были строками, не были обнаружены (см. «intg»?).

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | import unittest        def add(a, b):        """This function adds two numbers a, b and returns their sum            a and b may integers        """        if isinstance(a, int) and isinstance(b, int):            return a + b        elseif isinstance(a, str) and isinstance(b, str):            return int(a) + intg(b)        else:            raise Exception('Invalid arguments')        class Test(unittest.TestCase):        def test\_add(self):            self.assertEqual(5, add(2, 3))            self.assertEqual(15, add(-6, 21))            self.assertRaises(Exception, add, 4.0, 5.0)        unittest.main() |

Вот вывод:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | ----------------------------------------------------------------------    Ran 1 test in 0.000s        OK        Process finished with exit code 0 |

## Практические Unit Tests

Написание тестов промышленной мощности нелегкая и непростая задача. Есть несколько вещей, которые нужно учитывать и компромиссы, на которые следует пойти.

### Разработка тестирования

Если ваш код - это мешанина или набор нелепых строк, где разные уровни абстракции смешиваются друг с другом, и каждый фрагмент кода зависит от другого кода, вам будет трудно его протестировать.  Кроме того, всякий раз, когда вы что-то будете менять, вам также придется обновить кучу тестов.

Хорошей новостью является то, что разработка программного обеспечения общего назначения - это именно то, что вам нужно для проверки.  В частности, хорошо продуманный модульный код, в котором каждый компонент несет четкую ответственность и взаимодействует с другими компонентами через четко определенные интерфейсы, создаст удобные модульные тесты.

Например, наш класс SelfDrivingCar отвечает за высокоуровневую работу автомобиля: ехать, останавливаться, перемещаться.  Он имеет метод calculate\_distance\_to\_object\_in\_front(), который еще не реализован. Эта функциональность, вероятно, должна быть реализована полностью отдельной подсистемой.  Он может включать в себя считывание данных с различных датчиков, взаимодействие с другими автомобилями с помощью самостоятельного управления, целый стек "машинного зрения" для анализа изображений с нескольких камер.

Давайте посмотрим, как это работает на практике.  SelfDrivingCarпримет аргумент, называемый object\_detector, который имеет метод calculate\_distance\_to\_object\_in\_front(), и он делегирует эту функцию объекту. Теперь нет необходимости в проверке, потому что object\_detector отвечает (и должен быть протестирован) за него. Вам по-прежнему нужно, чтобы модуль тестировал то, что вы правильно используете object\_detector.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15 | class SelfDrivingCar(object):        def \_\_init\_\_(self, object\_detector):            self.object\_detector            self.speed = 0            self.destination = None            def \_calculate\_distance\_to\_object\_in\_front(self):            return self.object\_detector.calculate\_distance\_to\_object\_in\_front() |

### Затраты и выгоды

Количество усилий, которые вы вкладываете в тестирование, должно быть сопоставимо с затратами на неудачу, насколько стабильным является код и насколько легко его можно исправить, если проблемы обнаружены в строке.

Например, наш класс автопилота автомобилей критически важен. Если метод stop() не работает должным образом, наш автомобиль может убить людей, уничтожить имущество и пустить под откос другие автомобили. Если вы создаете автопилот для автомобиль, я подозреваю, предполагаю, что ваши юнит-тесты для метода stop()будут более тщательными, чем мои.

С другой стороны, если одна кнопка в вашем веб-приложении на странице, которая расположена тремя уровнями ниже главной страницы, немного мерцает, когда кто-то ее щелкает, вы можете это исправить, но, вероятно, вы не будете добавлять отдельный модульный тест для этого случая. Данный метод не оправдывает себя экономически.

### Мышление тестирования

Мышление тестирования очень важно. Один из принципов, который я использую, состоит в том, что каждый кусок кода имеет как минимум двух пользователей: код, который использует его, и пользователь, который его тестирует. Это простое правило помогает с разработкой и зависимостей. Если вы помните, что вам нужно написать тест для своего кода, вы не добавите много зависимостей, которые трудно восстановить во время тестирования.

Например, предположим, что ваш код должен что-то вычислять. Для этого ему необходимо загрузить данные из базы, прочитать файл конфигурации и динамически обратиться к некоторыми REST API для получения актуальной информации. Все это может потребоваться по разным причинам, но при этом, ели вы разместите всё это в одной функции, это очень затруднит тестирование. Это возможно звучит смешно, но гораздо лучше изначально структурировать ваш код.

### Чистые функции

Самым простым кодом для тестирования являются чистые функции. Чистые функции - это функции, которые имеют доступ только к значениям их параметров, не имеют побочных эффектов и возвращают один и тот же результат при вызове с теми же аргументами. Они не изменяют состояние вашей программы, не получают доступа к файловой системе или сети. Их преимуществ слишком много, чтобы их сейчас перечислять.

Почему их легко проверить? Потому что нет необходимости устанавливать специальную среду для их тестирования. Вы просто передаете аргументы и проверяете результат. Вы также знаете, что до тех пор, пока тестируемый код не изменится, ваш тест тоже не должен изменяться.

Сравните его с функцией, которая считывает XML-файл конфигурации. Ваш тест должен будет создать XML-файл и передать его имя файла в тестируемый код. Не трудная задача. Но предположим, что кто-то решил, что XML просто ужасен, и все файлы конфигурации должны находиться в JSON. Они занимаются своим делом и конвертируют все файлы конфигурации в JSON. Они проводят все тесты, включая ваши, и все они успешно завершаются!

Почему? Потому что код не изменился. Он все еще ожидает файл конфигурации XML, и ваш тест по-прежнему создает XML-файл для него.  Но при выполнении ваш код получит файл JSON, который он не сможет проанализировать.

#### Обработка ошибок тестирования

Обработка ошибок - это еще одна вещь, которая важна для тестирования. Она также является частью разработки. Кто несет ответственность за правильность ввода? Каждая функция и метод должны быть понятны. Если это ответственность функции, то она должна проверять вводные данные, но если это ответственность клиента, то функция выполняется, предполагая, что вводные данные правильные. Общая точность системы будет обеспечена путем тестирования для клиента, для того, чтобы убедиться, что он передает правильные данные в вашу функцию.

Как правило, вы хотите проверить ввод в общедоступном интерфейсе, потому что вам не обязательно знаеть, кто будет обращаться к вашему коду. Давайте посмотрим на метод drive() автопилота автомобиля. Этот метод ожидает параметр "destination". Параметр «destination» будет использоваться позже в навигации, но метод "drive" ничего не делает, чтобы мы смогли убедиться в его корректности.

Предположим, что цель должна быть параметрами широты и долготы. Существуют всевозможные тесты, которые можно выполнить, чтобы убедиться, что они действуют(например, определить пункт назначения в середине моря). Для наших целей давайте просто убедимся, что это параметры в диапазоне от 0,0 до 90,0 по широте и от -180,0 до 180,0 по долготе.

Вот обновленный класс SelfDrivingCar. Я просто выполнил некоторые из нереализованных методов, потому что метод drive() вызывает некоторые из этих методов прямо или косвенно.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | class SelfDrivingCar(object):        def \_\_init\_\_(self, object\_detector):            self.object\_detector = object\_detector            self.speed = 0            self.destination = None            def \_accelerate(self):            self.speed += 1            def \_decelerate(self):            if self.speed > 0:                self.speed -= 1            def \_advance\_to\_destination(self):            distance = self.\_calculate\_distance\_to\_object\_in\_front()            if distance < 10:                self.stop()                elif distance < self.speed / 2:                self.\_decelerate()            elif self.speed < self.\_get\_speed\_limit():                self.\_accelerate()            def \_has\_arrived(self):            return True            def \_calculate\_distance\_to\_object\_in\_front(self):            return self.object\_detector.calculate\_distance\_to\_object\_in\_front()            def \_get\_speed\_limit(self):            return 65            def stop(self):            self.speed = 0            def drive(self, destination):            self.destination = destination            while not self.\_has\_arrived():                self.\_advance\_to\_destination()            self.stop() |

Чтобы проверить, как обработались ошибки в тесте, я передаю неверные аргументы и думаю, что они должным образом будут отвергнуты. Вы можете сделать это, используя self.assertRaises()метод unittest.TestCase. Этот метод очень удачный, если код под тестированием действительно вызывает исключение.

Давайте посмотрим на это в действии. Метод test\_drive() пропускает широту и долготу вне допустимого диапазона и ждет, что метод drive() вызовет исключение.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | from unittest import TestCase    from self\_driving\_car import SelfDrivingCar            class MockObjectDetector(object):        def calculate\_distance\_to\_object\_in\_front(self):            return 20            class SelfDrivingCarTest(TestCase):        def setUp(self):            self.car = SelfDrivingCar(MockObjectDetector())            def test\_stop(self):            self.car.speed = 5            self.car.stop()            # Verify the speed is 0 after stopping            self.assertEqual(0, self.car.speed)                # Verify it is Ok to stop again if the car is already stopped            self.car.stop()            self.assertEqual(0, self.car.speed)            def test\_drive(self):            # Valid destination            self.car.drive((55.0, 66.0))                # Invalid destination wrong range            self.assertRaises(Exception, self.car.drive, (-55.0, 200.0)) |

Тест не выполняется, поскольку метод drive () не проверяет его аргументы на достоверность и не вызывает исключения. Вы получите хороший отчет с полной информацией о том, что не удалось, где и почему.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | python -m unittest discover -v    test\_drive (untitled.test\_self\_driving\_car.SelfDrivingCarTest) ... FAIL    test\_stop (untitled.test\_self\_driving\_car.SelfDrivingCarTest) ... ok        ======================================================================    FAIL: test\_drive (untitled.test\_self\_driving\_car.SelfDrivingCarTest)    ----------------------------------------------------------------------    Traceback (most recent call last):      File "/Users/gigi/PycharmProjects/untitled/test\_self\_driving\_car.py", line 29, in test\_drive        self.assertRaises(Exception, self.car.drive, (-55.0, 200.0))    AssertionError: Exception not raised        ----------------------------------------------------------------------    Ran 2 tests in 0.000s        FAILED (failures=1) |

Чтобы исправить это, давайте обновим метод drive(), чтобы фактически проверить диапазон его аргументов:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21 | def drive(self, destination):            lat, lon = destination            if not (0.0 <= lat <= 90.0):                raise Exception('Latitude out of range')            if not (-180.0 <= lon <= 180.0):                raise Exception('Latitude out of range')                self.destination = destination            while not self.\_has\_arrived():                self.\_advance\_to\_destination()            self.stop()<br> |

Теперь все тесты проходят.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15 | python -m unittest discover -v    test\_drive (untitled.test\_self\_driving\_car.SelfDrivingCarTest) ... ok    test\_stop (untitled.test\_self\_driving\_car.SelfDrivingCarTest) ... ok        ----------------------------------------------------------------------    Ran 2 tests in 0.000s        OK |

### Тестирование частных методов

Должны ли вы проверять каждую функцию и метод? В частности, следует ли тестировать частные методы, называемые только вашим кодом?  Обычный неудовлетворительный ответ: «В зависимости от ситуации».

Я постараюсь помочь вам и рассказать, от чего это зависит. Вы точно знаете, кто называет ваш частный метод - это ваш собственный код. В том случае, если ваши тесты служат для общедоступных методов, которые вызывают ваш частный метод и являются комплексными, которые вы уже полностью протестировали. Но если частный метод очень сложный, вы можете протестировать его отдельно. Решайте сами.

### Как организовать ваши модульные тесты

В большой системе не всегда ясно, как провести тесты. Должен ли быть один большой файл со всеми тестами для пакета или отдельный тестовый файл для каждого класса? Должны ли тесты быть в том же файле, что и тестируемый код, или в том же каталоге?

Вот система, которую я использую. Тесты должны быть полностью отделены от тестируемого кода (следовательно, я не использую doctest). В идеале ваш код должен быть в пакете. Тесты для каждого пакета должны находиться в каталоге вашего дочернего узла. В каталоге тестов должен быть один файл для каждого модуля вашего пакета с именем test\_<module name>.

Например, если в вашем пакете есть три модуля: module\_1.py, module\_2.py и module\_3.py, вы должны иметь три тестовых файла: test\_module\_1.py, test\_module\_2.py и test\_module\_3.py в каталоге тестов.

Этот подход имеет ряд преимуществ. Это дает определённость уже при просмотре каталогов, сразу видно, что вы не забыли протестировать.  Это также помогает сохранять тесты в приемлимых размерах. Предполагая, что ваши модули имеют приемлимый размер, тестовый код для каждого модуля будет в собственном файле, который может быть немного больше, чем тестируемый модуль, но все же удобно размещён в одном файле.

## Заключение

Юнит- тесты являются основой качествнного кода. В этом уроке я изучил некоторые принципы и рекомендации для работы с модульным тестированием и объяснил нескольких лучших практик. Чем больше система, которую вы строите, тем более важными становятся модульные тесты.  Но одних модульных тестов недостаточно. Другие типы тестов также необходимы для крупномасштабных систем: интеграционные тесты, тесты производительности, тесты нагрузки, тестирование на уязвимость, тесты на прием данных и другие.