**Тестирование кода**

Тестирование кода очень важно.

Привыкать к написанию тестового кода и параллельному выполнению этого кода теперь считается хорошей привычкой. Использовать с умом, этот метод позволяет более точно определить намерения вашего кода и более изолированным архитектуры.

Некоторые общие правила тестирования:

* Блок тестирования должен сосредоточиться на одной крошечной части функциональности и доказать ее правильность.
* Каждый испытательный блок должен быть полностью независимым. Каждый тест должен быть в состоянии работать в одиночку, а также в наборе тестов, независимо от порядка их вызова. Следствием этого правила является то, что каждый тест должен быть загружен с новым набором данных и, возможно, придется сделать некоторые очистки после этого. Обычно этим занимаются **setUp()** и **tearDown()** методы.
* Постарайтесь сделать тесты, которые выполняются быстро. Если для выполнения одного теста требуется более нескольких миллисекунд, разработка будет замедляться или тесты будут выполняться не так часто, как это желательно. В некоторых случаях тесты не могут быть быстрыми, поскольку для их работы необходима сложная структура данных, и эта структура данных должна загружаться при каждом запуске теста. Храните эти более тяжелые тесты в отдельном наборе тестов, который выполняется какой-либо запланированной задачей, и выполняйте все остальные тесты так часто, как это необходимо.
* Узнать свои инструменты и узнать, как выполнить один тест или тестовые примеры. Затем, при разработке функции внутри модуля, часто запускайте тесты этой функции, в идеале автоматически при сохранении кода.
* Всегда запускайте полный набор тестов перед сеансом кодирования и запускайте его снова после. Это даст вам больше уверенности в том, что вы ничего не сломали в остальной части кода.
* Рекомендуется реализовать обработчик, который запускает все тесты перед передачей кода в общий репозиторий.
* Если вы находитесь в середине сеанса разработки и должны прервать вашу работу, это хорошая идея, чтобы написать сломанный модульный тест о том, что вы хотите разработать дальше. Возвращаясь к работе, вы будете иметь указатель на то, где вы были и вернуться на путь быстрее.
* Первым шагом при отладке кода является написание нового теста, указывающего на ошибку. Хотя это не всегда возможно сделать, эти тесты на улавливание ошибок являются одними из самых ценных элементов кода в проекте.
* Используйте длинные и описательные имена для тестирования функций. Руководство по стилю здесь немного отличается от руководства по выполнению кода, где часто предпочтение отдается коротким именам. Причина в том, что функции тестирования никогда не вызываются явно. Square() или даже sqr() нормально при выполнении кода, но при тестировании кода у вас будут такие имена, как test\_square\_of\_number\_2(),test\_square\_negative\_number().Эти имена функций отображаются при сбое теста и должны быть как можно более описательными.
* Когда что-то пойдет не так или должно быть изменено, и если ваш код имеет хороший набор тестов, вы или другие сопровождающие будут полагаться в основном на набор тестирования для устранения проблемы или изменения данного поведения. Поэтому тестовый код будет считываться столько или даже больше, чем запущенный код. Единый тест, цель которого неясна, в этом случае не очень полезен.
* Еще одно применение тестового кода - это введение в работу с новыми разработчиками. Когда кому-то придется работать на базе кода, то запуск и чтение соответствующего тестового кода часто является лучшим, что они могут сделать, чтобы начать. Они будут или должны выявить горячие точки, где большинство трудностей возникает, и угловые случаи. Если они должны добавить некоторые функциональные возможности, первым шагом должно быть добавление теста, чтобы убедиться, что новая функциональность еще не рабочий путь, который не был подключен к интерфейсу.

**Основы**

**Unittest**

[unittest](http://docs.python.org/library/unittest.html#module-unittest)- это батарейки – включают в себя тестовый модуль в стандартной библиотеке Python. Его API знаком каждому, кто использовал любой из JUnit/nUnit/CppUnit серии инструментов.

Создание тестовых примеров выполняется путем подкласса: unittest.TestCase.

**import** unittest

**def** fun**(**x**):**

**return** x + 1

**class** MyTest**(**unittest.TestCase**):**

**def** test**(**self**):**

self.assertEqual**(**fun**(**3**),** 4**)**

Начиная с Python 2.7, unittest также включает свои собственные механизмы обнаружения тестов.

**Doctest**

Модуль doctest ищет фрагменты текста, которые выглядят как интерактивные сеансы Python в docstrings, а затем выполняет эти сеансы, чтобы убедиться, что они работают точно так, как показано на рисунке.

У Doctests есть другой вариант использования, чем правильные модульные тесты: он, как правило, менее детализирован и не ловит специальные случаи или неясные ошибки регрессии. Они полезны в качестве выразительной документации основных случаев использования модуля и его компонентов. Тем не менее, доктрины должны запускаться автоматически каждый раз, когда запускается полный набор тестов.

Простой doctest в функции:

**def** square**(**x**):**

*"""Return the square of x.*

*>>> square(2)*

*4*

*>>> square(-2)*

*4*

*"""*

**return** x \* x

**if** \_\_name\_\_ == '\_\_main\_\_'**:**

**import** doctest

doctest.testmod**()**

При запуске этого модуля из командной строки, как в python module.py, doctests будет работать и жаловаться, если что-либо не ведет себя, как описано в docstrings.

**Инструменты**

### py.test

### py.test – это не шаблонный альтернативный стандартный модуль Python unittest.

$ pip install pytest

Несмотря на то, что он является полнофункциональным и расширяемым инструментом тестирования, он может похвастаться простым синтаксисом. Создание набора тестов так же просто, как написание модуля с несколькими функциями:

# content of test\_sample.py

**def** func**(**x**):**

**return** x + 1

**def** test\_answer**():**

**assert** func**(**3**)** == 5

и затем запись команды py.test :

$ py.test

=========================== test session starts ============================

platform darwin -- Python 2.7.1 -- pytest-2.2.1

collecting ... collected 1 items

test\_sample.py F

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_answer():

> assert func(3) == 5

E assert 4 == 5

E + where 4 = func(3)

test\_sample.py:5: AssertionError

========================= 1 failed in 0.02 seconds =========================

На это уходит гораздо меньше усилий, чем потребовалось бы для аналогичной функциональности модуля unittest!

### Nose

Nose расширяет unittest для того, чтобы сделать тестирование более легким.

$ pip install nose

Nose обеспечивает автоматическое обнаружение испытания для того чтобы избавить вас от необходимости вручную создавать тестовые наборы. Он также предоставляет многочисленные плагины для таких функций, как xUnit-совместимый тестовый вывод, отчеты о покрытии и выбор теста.

### tox

tox - это инструмент для автоматизации управления тестовой среды и тестирование множества конфигураций интерпретатора.

$ pip install tox

tox позволяет настроить сложный многопараметрический тест матриц через простой ini-файл конфигурации.

### Unittest2

unittest2-это бэкпорт модуля unittest Python 2.7, который имеет улучшенный API и лучшие утверждения по сравнению с тем, который доступен в предыдущих версиях Python.

Если вы используете Python 2.6 или ниже, вы можете установить его с помощью pip

$ pip install unittest2

Вы можете импортировать модуль под именем unittest, чтобы упростить перенос кода в будущем на новые версии модуля

**import** unittest2 **as** unittest

**class** MyTest**(**unittest.TestCase**):**

...

Таким образом, если вы когда-либо перейдёте на новую версию Python и больше не будете нуждаться в модуле unittest2, вы сможете просто изменить импорт в тестовом модуле без необходимости изменять любой другой код.

### mock

unittest.mock - библиотека для тестирования на Python. Начиная с Python 3.3, он доступен в стандартной библиотеке.

Для более старых версий Python:

$ pip install mock

Он позволяет заменять тестируемые части системы на макеты объектов и делать утверждения о том, как они были использованы.

Например, вы можете применить метод monkey-patch:

**from** mock **import** MagicMock

thing = ProductionClass**()**

thing.method = MagicMock**(**return\_value=3**)**

thing.method**(**3**,** 4**,** 5**,** key='value'**)**

thing.method.assert\_called\_with**(**3**,** 4**,** 5**,** key='value'**)**

Чтобы дразнить классы или объекты в тестируемом модуле, используется декоратор исправлений. В приведенном ниже примере внешняя поисковая система заменяется макетом, который всегда возвращает один и тот же результат (но только на время теста).

**def** mock\_search**(**self**):**

**class** MockSearchQuerySet**(**SearchQuerySet**):**

**def** \_\_iter\_\_**(**self**):**

**return** iter**([**"foo"**,** "bar"**,** "baz"**])**

**return** MockSearchQuerySet**()**

*# SearchForm here refers to the imported class reference in myapp,*

*# not where the SearchForm class itself is imported from*

@mock.patch**(**'myapp.SearchForm.search'**,** mock\_search**)**

**def** test\_new\_watchlist\_activities**(**self**):**

*# get\_search\_results runs a search and iterates over the result*

self.assertEqual**(**len**(**myapp.get\_search\_results**(**q="fish"**)),** 3**)**

Mock имеет много других способов, которыми вы можете настроить его и контролировать его поведение.