Урок 4

Основы тестирования

Введение в тестирование. Оператор assert. Модульное тестирование и модуль unittest.

© geekbrains.ru

Введение в тестирование

Программы на Python, в отличие от написанных на C или Java, не обрабатываются компилятором. В этих языках компилятор — это первая линия обороны от программных ошибок: он отыскивает вызов функций с недопустимым количеством аргументов или присваивание некорректных значений переменным (проверяет типы). В Python такие проверки выполняются только после запуска программы. Поэтому невозможно сказать, содержит ли она ошибки, пока не будет запущена и протестирована. Если не опробовать программу всеми способами, когда поток управления пройдет все ее возможные ветви, всегда остается вероятность, что в ней скрылась ошибка и ждет своего часа. Такие баги обычно обнаруживаются уже через несколько дней после передачи программы пользователю.

На этом уроке изучим приемы и библиотечные модули для тестирования программного кода на Python. Эти материалы можно использовать для написания собственных тестов при разработке проекта в практических заданиях.

Оператор assert

Это особая конструкция, позволяющая проверять предположения о значениях данных в любом месте программы. Оператор **assert** обеспечивает автоматическое оповещение, если обнаруживаются некорректные данные. Выполнение кода программы аварийно завершается, и выводится сообщение с указанием места обнаружения ошибки.

Если данные программы стали некорректными, ее работа завершится. Оператор **assert** предусмотрен во многих языках программирования, в том числе Python. Благодаря этой конструкции ошибки своевременно локализуются и оперативно исправляются. Пользователь застрахован от некорректной обработки данных.

Когда и где применять

Оператор **assert** может использоваться в любом месте программы даже при проверке очевидных программных инструкций, поскольку при рефакторинге код может потерять очевидность или при выполнении обработать данные с ошибкой. Даже многократное применение **assert** существенно не повлияет на понятность кода и скорость выполнения программы. **Assert**-ы заметны в коде приложения: они несут важную информацию о логике работы программы и могут заменять комментарии.

В различных языках программирования конструкции **assert** отключаются на этапе компиляции или в процессе выполнения приложения, поэтому минимально влияют на параметры производительности программы. Хорошая практика — оставлять **assert**-ы в рабочем состоянии при написании и тестировании приложений, но в продакшн-версиях их, как правило, отключают.

Когда можно отказаться

Сколько использовать операторов **assert** в программе, определяет только сам разработчик. Но в некоторых случаях **assert**-ы дублируют друг друга и становятся избыточными. Их количество в программе можно оптимизировать: например, размещать инструкции проверки входящих аргументов только в процедурах, работающих с данным аргументом. Если процедура **on_clicked()** не выполняет операций с аргументом, а отправляет его в функцию **on_changed()**, от его проверки в **on_clicked()** можно отказаться и выполнять ее только в пределах **on_changed()**. Также не следует размещать **assert** на проверку значений, которые гарантированно ведут к аварийному завершению программы.

Когда нельзя использовать

Основное назначение конструкции **assert** — поиск багов в коде, но для обработки «ожидаемых» ошибок — не из сферы программирования — она не предназначена. Их лучше «отлавливать» с помощью **expect**-ов. Также необходимо следить, чтобы конструкции с участием оператора не изменяли поведение программы, их задача — проверка корректности данных.

Как работать с оператором assert

В общем случае инструкция **assert** имеет следующий вид:

```
assert test [, msg]
```

Здесь **test** — это выражение, которое должно возвращать значение **True** или **False**. Если **False**, инструкция **assert** выдаст исключение **AssertionError** с переданным ему сообщением **msg**. Например:

```
def write_data(file, data):
    assert file, "write_data: файл не определен!"
...
```

Инструкция **assert** не должна содержать программный код, обеспечивающий безошибочную работу программы, потому что он не будет выполняться интерпретатором, работающим в оптимизированном режиме (включается при запуске интерпретатора с ключом **-O**). Ошибочно использовать инструкцию **assert** для проверки ввода пользователя. Обычно с помощью **assert** проверяют условия, которые всегда должны быть истинными; если они нарушаются, это можно рассматривать как баг в программе, а не ошибку пользователя.

Assert указывает, что выражение истинно. Например, существует список и необходимо гарантировать, что в нем будет содержаться хотя бы один элемент, иначе должна генерироваться ошибка. Для этого оператор **assert** оптимален.

```
test_list = ['el_1']
assert len(test_list) >= 1
```

Если запустить данный код на выполнение, он будет успешно выполнен без генерации ошибки и остановки программы.

Если применить к списку метод **pop()**, который возвращает последний элемент с одновременным его удалением, проверяемое выражение примет значение **False** и сгенерируется ошибка: **AssertionError**.

```
test_list = ['el_1']
test_list.pop()
assert len(test_list) >= 1
```

Ошибка:

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AssertionError
```

С помощью **assert** можно писать юнит-тесты. Для этого удобно оформить тест в виде функции:

```
def assert_equal(x, y):
    assert x == y, "{} != {}".format(x, y)
```

Рассмотрим программный код простейшего приложения для расчета зарплаты сотрудников. В основе — количество отработанных часов и ставка за час работы (файл **examples/01_assert_unit_test/01_assert_salary.py**):

```
import datetime
from collections import namedtuple


Salary = namedtuple('Salary', ('surname', 'name', 'worked', 'rate'))

def get_salary(line):
    ''' Вычисление зарплаты работника
    '''
    line = line.split()
    if line:
        data = Salary(*line)
        fio = ' '.join((data.surname, data.name))
        salary = int(data.worked) * int(data.rate)
        res = (fio, salary)
    else:
        res = ()
    return res


def test_get_salary_summ():
    assert get_salary('Лютиков   Руслан     60    1000') ==\
                     ('Лютиков Руслан', 60000), 'Неверная сумма'


def test_get_salary_fio():
    assert get_salary('Лютиков   Руслан     60    1000')[0] ==\
                     'Лютиков Руслан', 'Неверное имя'


def test_get_salary_empty():
    assert get_salary('') == (), 'Непустые данные'

def test_get_salary_wrong_format():
    assert get_salary(' ') == (), 'Непустые данные'


if __name__ == "__main__":
    test_get_salary_fio()
    test_get_salary_summ()
    test_get_salary_empty()
    test_get_salary_wrong_format()
```

Этот скрипт демонстрирует возможности тестов. Здесь определена исходная структура данных — именованный кортеж **Salary** с четырьмя полями: **surname** (фамилия), **name** (имя), **worked** (выработка), **rate** (ставка).

Основной функцией данного сценария, работа которой проверяется с помощью тестов, является **get_salary()**. В ней для конкретного работника фамилия привязывается к имени, он идентифицируется, и выполняется расчет его зарплаты. Следующие четыре функции — это тесты. Каждый проверяет определенное условие. Первый — правильность вычисления зарплаты, второй — корректность указанных имени и фамилии работника, третий и четвертый — наличие данных и их формат.

При старте выполнения сценария (**if__name__ == "__main__"**) запускаются тесты. В каждом из них вызывается основная функция сценария **get_salary()** с передачей в нее аргументов (например, **get_salary('Лютиков Руслан 60 1000')**). Функция выполняется с указанными аргументами, и результат сравнивается с тем, который должен получиться (**'Лютиков Руслан', 60000**). Если итоговый и эталонный результаты различаются, генерируется сообщение об ошибке и программа останавливается.

Плюсы организации тестирования через **assert**:

- тесты легко читать;
- используются стандартные средства Python;
- тесты будут организованы в простые функции.

Недостатки:

- тесты нужно запускать вручную;
- такие тесты сложно отлаживать;
- для каждой проверки нужно написать свою функцию и сообщение об ошибке («равно», «не равно» и т.д.).

Современные библиотеки юнит-тестов (например, **unittest**) предоставляют готовые функции и более широкие возможности для организации тестов.

Модульное тестирование и модуль unittest

Для полноценного тестирования программ можно использовать модуль **unittest**.

При модульном тестировании разработчик пишет набор обособленных тестов для каждого компонента программы (например, для отдельных функций, методов, классов и модулей). Затем эти тесты проверяют корректность поведения основных компонентов крупных программ. По мере роста продуктов модульные тесты могут объединяться в структуры и средства тестирования. Это упрощает проверку корректности поведения, а также определение и исправление проблем.

Введение в юнит-тесты

Миссия юнит-тестов — тестирование с высоким уровнем гранулярности, то есть не всей системы в целом, а отдельных ее компонентов. В этом принципиальное отличие unit-тестов от системных, а

также интеграционных, ориентированных на проверку взаимодействия между составляющими модулями. Благодаря юнит-тестам анализируются маленькие участки кода, что позволяет быстро найти и устранить ошибку.

Модуль **unittest** автоматизирует выполнение тестов, поддерживает использование общего кода для подготовки, завершения и объединения тестов, их интеграции в группы. Для автоматизации тестов в **unittest** реализованы следующие концепции:

1. **Испытательный стенд (test fixture)** — в рамках данной концепции настраиваются тесты и операции, выполняемые по их завершении — например, генерация временных баз данных и старт серверного процесса.

2. **Тестовый случай (test case)** — определяет блок тестирования. Выполняет проверку ответов для различных наборов данных. В модуле **unittest** реализован базовый класс **TestCase**, используемый для подготовки новых тестовых случаев.

3. **Набор тестов (test suite)** — набор тестовых случаев или самих тестов. Объединяет тесты, которые выполняются вместе.

4. **Исполнитель тестов (test runner)** — компонент, контролирующий выполнение тестов и предоставляющий результат пользователю. Может работать через графический или текстовый интерфейс и возвращать специальный объект с сообщением о результате тестирования.

Чтобы понять принцип работы юнит-тестов, рассмотрим несложную функцию, определяющую сумму квадратов двух чисел, передаваемых в функцию в виде параметров **i** и **j** (файл **examples/01_assert_unit_test/02_unittest_sum.py**):

```
def sum_kv_ij(i, j):
    return i*i+j*j
```

Чтобы проверить правильность работы этой несложной команды, необходимо написать функцию-тест, в которой бы вызывалась процедура **sum_kv_ij** и ей передавались два любых аргумента-числа. Их потом нужно сравнить с результатом, который должен получиться от выполнения функции **sum_kv_ij**.

Тест, проверяющий работу данной функции:

```
import unittest
class TestSumKV(unittest.TestCase):
    def testequal(self):
        self.assertEqual(test_sum_kv(2, 3), 23)
if __name__ == '__main__':
    unittest.main()
```

Две последние строки данного фрагмента кода реализуют простейший механизм, запускающий тесты. **Unittest.main()** реализует интерфейс командной строки для этого.

Результат выполнения данного теста — сообщение об ошибке:

```
Ran 1 test in 0.184s
FAILED (failures=1)
Exit code:  True
```

Еще одно определение, раскрывающее сущность юнит-тестов: юнит — небольшой самодостаточный фрагмент кода, реализующий определенную логику и являющийся классом. Другое название юнит-тестирования — модульное. В Python одним из популярных фреймворков для такого тестирования является **unittest**. Он поддерживается стандартной библиотекой Python. Дальнейшая работа в ходе данного урока будет построена именно на его базе.

Характеристики хорошего теста:

- **корректный** — проверяет то, что нужно проверить;
- **понятен** читателю;
- **конкретный** — проверяет что-то одно.

Как используется модуль **unittest**, иллюстрирует следующий фрагмент программного кода (файл **examples/01_assert_unit_test/03_unitest_splitter.py**):

```
def split(line, types=None, delimiter=None):
    """ Разбивает текстовую строку и при необходимости
    выполняет преобразование типов.
    ...
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty, val in zip(types, fields) ]
    return fields
```

Если потребуется написать модульные тесты для проверки различных аспектов применения функции **split()**, можно создать отдельный модуль **testsplitter.py**, например:

```
import unittest

# Модульные тесты
class TestSplitFunction(unittest.TestCase):
    def setUp(self):
        # Выполнить настройку тестов (если необходимо)
        pass
    def tearDown(self):
        # Выполнить завершающие действия (если необходимо)
        pass
    def testsimplestring(self):
        r = splitter.split('GOOG 100 490.50')
        self.assertEqual(r,['GOOG','100','490.50'])
    def testtypeconvert(self):
        r = splitter.split('GOOG 100 490.50', [str, int, float])
        self.assertEqual(r,['GOOG', 100, 490.5])
    def testdelimiter(self):
        r = splitter.split('GOOG,100,490.50',delimiter=',')
        self.assertEqual(r,['GOOG','100','490.50'])

# Запустить тестирование
if __name__ == '__main__':
```

```
    unittest.main()
```

Чтобы начать тестирование, достаточно запустить интерпретатор Python, передав ему файл **testsplitter.py**:

```
% python testsplitter.py
...
----------------------------------------------------------------------
Run 3 tests in 0.01
OK
```

Данное служебное сообщение показывает, что было выполнено 3 теста. Указано время их выполнения и статус «OK» — тесты завершены без ошибок. Программный код функции **split()**, для проверки которого созданы данные тесты, работает корректно.

Модуль **unittest** опирается на объявление класса, производного от **unittest.TestCase**. Отдельные тесты определяются как методы, имена которых начинаются со слова **test**: **testsimplestring**, **testtypeconvert** и так далее. Важно отметить, что имена методов могут выбираться произвольно — главное, чтобы они начинались со слова **test**. Внутри каждого теста выполняются проверки условий.

Экземпляр **t** класса **unittest.TestCase** имеет следующие методы, которые могут использоваться для тестирования и управления этим процессом:

- **t.setUp()** — применяется для настройки перед вызовом любых методов тестирования;
- **t.tearDown()** — вызывается для заключительных действий после выполнения всех тестов;
- **t.assert_(expr [, msg]) / t.failUnless(expr [, msg])** — сообщает об ошибке тестирования, если выражение **expr** оценивается как **False**. **msg** — это строка сообщения, объясняющая причины ошибки (если задана);
- **t.assertEqual(x, y [,msg]) / t.failUnlessEqual(x, y [, msg])** — сообщает об ошибке тестирования, если **x** и **y** не равны;
- **t.assertNotEqual(x, y [, msg]) / t.failIfEqual(x, y, [, msg])** — сообщает об ошибке тестирования, если **x** и **y** равны;
- **t.assertAlmostEqual(x, y [, places [, msg]]) / t.failUnlessAlmostEqual(x, y, [, places [, msg]])** — сообщает об ошибке тестирования, если числа **x** и **y** не совпадают с точностью до знака **places** после десятичной точки. Проверка выполняется за счет вычисления разности между **x** и **y** и округления результата до указанного числа знаков **places** после десятичной точки. Если результат равен нулю, числа **x** и **y** можно считать почти равными;
- **t.assertNotAlmostEqual(x, y, [, places [, msg]]) / t.failIfAlmostEqual(x, y [, places [, msg]])** — сообщает об ошибке тестирования, если числа **x** и **y** совпадают с точностью до знака **places** после десятичной точки;
- **t.assertRaises(exc, callable, ...) / t.failUnlessRaises(exc, callable, ...)** — сообщает об ошибке тестирования, если вызываемый объект **callable** не выдает исключение **exc**. Остальные аргументы методов передаются **callable**. Для тестирования набора исключений в аргументе **exc** передается кортеж с этими исключениями;

- **t.failIf(expr [, msg])** — сообщает об ошибке тестирования, если выражение **expr** оценивается как **True**;
- **t.fail([msg])** — сообщает об ошибке тестирования;
- **t.failureException** — в этом атрибуте сохраняется последнее исключение, перехваченное в тесте. Может использоваться, когда необходимо проверить, что исключение не только вызывается, но и сопровождается требуемым значением (например, сообщение, генерируемое исключением).

У модуля **unittest** множество дополнительных параметров настройки, используемых для группировки тестов, создания их наборов тестов и управления окружением, в котором они выполняются. Эти особенности не имеют прямого отношения к процессу создания тестов (классы обычно пишутся независимо от того, как в действительности выполняются тесты). В документации можно найти дополнительную информацию, как организовать тесты для крупных программ.

Модуль **unittest** предоставляет класс **TestCase**, на основе которого объявляет класс, содержащий отдельные тесты. После этого в зависимости от задачи разработчик может переопределить параметры методов **setUp()** и **tearDown()**, подготовив среду до запуска тестирования и после его завершения. В примере **examples/01_assert_unit_test/02_unitest_splitter.py**, представленном выше, показано, как можно использовать эти методы.

Рассмотрим еще один пример реализации модульного тестирования на базе **unittest**. Этот пример повторяет задачу с расчетом зарплаты сотрудников. Только в этом случае необходимо провести проверку корректности программного кода уже применительно к фреймворку **unittest** (файл **examples/01_assert_unit_test/04_unitest_salary.py**).

```
"""
Фамилия      Имя          Часов   Ставка
Иванов       Иван         45      400
Докукин      Филимон      20      1000
Ромашкин     Сидор        45      500
"""

import datetime
from collections import namedtuple

import unittest

Salary = namedtuple('Salary', ('surname', 'name', 'worked', 'rate'))

def get_salary(line):
    ''' Вычисление зарплаты работника
    '''
    line = line.split()
    if line:
        data = Salary(*line)
        fio = ' '.join((data.surname, data.name))
        salary = int(data.worked) * int(data.rate)
        res = (fio, salary)
    else:
        res = ()
    return res


class TestSalary(unittest.TestCase):

    def test_get_salary_summ(self):
        self.assertEqual(get_salary('Лютиков   Руслан     60    1000'),
                         ('Лютиков Руслан', 60000))

    def test_get_salary_fio(self):
        self.assertEqual(get_salary('Лютиков   Руслан     60    1000')[0],
                         'Лютиков Руслан')

    def test_get_salary_empty(self):
        self.assertEqual(get_salary(''), ('1', '2'))


if __name__ == "__main__":
    unittest.main()
```

Рекомендации по подготовке тестов

Тесты надо писать, придерживаясь следующих рекомендаций:

1. Выполнение одного теста не должно зависеть от результатов других;
2. Тест должен работать только с данными, подготовленными специально для него;
3. Тест не должен требовать ввода данных пользователем;
4. Необходимо избегать перекрытия тестов (не следует писать одинаковые тесты многократно);
5. При обнаружении ошибок программирования следует писать соответствующие тесты;
6. Необходимо поддерживать тесты в рабочем состоянии;
7. Модульные тесты не должны предусматривать проверку производительности сущности (класса или функции);
8. Тест должен предусматривать проверку не только работы сущности на корректных данных, но и ее адекватности при некорректных;
9. Проводить запуск тестов необходимо регулярно.

Особенности организации модульных тестов

Если ведется работа над простым проектом (как в приведенном выше примере, где python-сценарий реализовывал логику расчета зарплаты сотрудников), сам программный код и связанные с ним тесты можно расположить в одном файле. Если же создается серьезное программное приложение, к вопросу структуризации продукта следует подходить более ответственно. Обдумать, следует ли помещать каждый тестовый класс определенного программного пакета в отдельный файл или стоит создать один и сохранить в нем все тесты для данного пакета.

Один из вариантов — полностью отделить подготовленные тесты от написанного программного кода. Для каждого программного пакета следует создать отдельный каталог, в котором будут располагаться его тесты. Модулям пакета должны соответствовать файлы с программным кодом, а каждому из них — файлы с тестовым классом. Если в пакете реализовано четыре Python-модуля: **m_1.py**, **m_2.py**, **m_3.py**, **m_4.py**, — в нем должен быть каталог с четырьмя тестовыми файлами. Название каждого из них должно начинаться с «**test**» и содержать наименование модуля: **test_ m_1.py**, **test_ m_2.py**, **test_ m_3.py**, **test_ m_4.py**.

Так удобнее анализировать программу. Открывая директорию пакета, разработчик видит, что все модули протестированы, и может изучить тестовый класс любого из программных модулей пакета.

Другие возможности применения юнит-тестов

Аналог документации

Юнит-тесты могут выполнять функцию документации для разработанного приложения. Набор грамотно подобранных тестов, которые покрывают все варианты использования программы, определяют ограничения и возможные ошибки, не уступает в информативности специально составленным примерам работы продукта. Тесты можно запустить в любой момент и отследить корректность поведения программы.

Набор тестов может служить полноценной альтернативой привычной документации по программе — например, составленной с использованием генератора документации **sphinx-doc**, о котором мы подробно поговорим в следующих уроках.

Разработка через тестирование

Данный подход к разработке также известен как Test-Driven Development. Согласно TDD, разработка приложения начинается с написания тестов, проверяющих определенные операции с данными. Только после этого создается программный код, который должен корректно проходить эти тесты.

Философия использования TDD заключается в том, что разработчик пишет только тот программный код, который необходим для успешных тестов. Когда они пройдены, проводят рефакторинг кода (переработку для лучшего понимания) и переходят к подготовке очередного блока тестов, а далее — к реализации соответствующей им логики.

Работа по принципу TDD предполагает, что все участки кода должны быть покрыты тестами. По сути, документируется вся логика работы программы.

Итоги

На данном занятии мы познакомились с базовыми возможностями работы с *сокетами* в Python и с подходами к тестированию программного кода — в частности, с модульным тестированием и фреймворком **unittest**.

Гибкие методики разработки рассматривают внесение изменений в код не как гипотетическую и маловероятную возможность, а как обыденную часть работы. Это стимулирует разработчиков к созданию кода, который легче поддается изменениям. Чтобы было проще их вносить, рекомендуется делать архитектуру приложения **слабо связанной**. А чтобы уменьшить скрытые последствия изменений, используют **автоматизированные тесты**. Они позволяют обнаружить отклонение программы от ожидаемого поведения на ранней стадии изменений.

Важно отметить, что **тесты** — такая же часть кода, как и архитектура. Они делают его приспособленным к внесению изменений с минимальными последствиями. Чем стройнее архитектура — тем легче создавать тесты. Чем лучше организованы тесты, тем меньше скрытых последствий будет после изменений. Тем надежнее и качественнее код.

Практическое задание

1. Для всех функций из урока 3 написать тесты с использованием **unittest**. Они должны быть оформлены в отдельных скриптах с префиксом **test_** в имени файла (например, **test_client.py**).

2. * Написать тесты для домашних работ из курса «Python 1».

Дополнительные материалы

1. Тестирование. Начало.
2. Юнит-тесты. Первый шаг к качеству.
3. Зачем нужны юнит-тесты.

4. Тестирование по-пайтоновски. Введение.
5. Test-Driven Development в Python для начинающих. Часть первая.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. Дэвид Бизли. Python. Подробный справочник (каталог «Дополнительные материалы»).
3. Лучано Ромальо. Python. К вершинам мастерства (каталог «Дополнительные материалы»).
4. Кент Бек. Экстремальное программирование: разработка через тестирование (каталог «Дополнительные материалы»).
5. Assert.
6. A byte of Python.
7. Модуль unittest: тестируем свои программы.
8. Python уроки: тестирование с помощью unittest.
9. Как писать профессиональные модульные тесты на Python.
10. Тестирование в Python [unittest]. Часть 1. Введение.