Лекция №12

Тестирование программного обеспечения

- Общая информация
- Виды тестирования
- unittest
- coverage
- nose
- nose: coverage
- mock
- mock: @mock.patch
- Системное тестирование
- selenium
- robotframework
- Практика

Общая информация

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом (ISO/IEC TR 19759:2005).

Качество написанной программы определяется, в первую очередь, соответствием реальных результатов работы программы и ожидаемых.

Прежде чем поставляться заказчику, программный продукт должен пройти проверку на соответствие требованиям заказчика. Проверка подразумевает передачу программе (или отдельным ее компонентам) входных данных, считывание результата или определенных статистик выполнения программы и сравнение этих выходных данных с ожидаемым результатом.

Тестовые сценарии определяются в соответствии с требованиями заказчика (для всего продукта), либо с требованиями к компоненту по дизайну проекта (для отдельных компонентов).

Виды тестирования

Существуют различные способы классификации видов тестирования: по объекту тестирования (функциональные тесты, стресс-тесты), по степени изолированности (юнит-тестирование, интеграционное. системное) по степени автоматизации и т.д.

В самом общем случае программное обеспечение должно проходит компонентное (юнит-) и системное тестирование.

Юнит-тестирование подразумевает проверку отдельных модулей, функций и классов. За написание и выполнение соответствующих тестов обычно отвечает сам программист, который пишет тестируемый код. Основная цель при написании тестов – максимально покрыть ими код, т.е. задействовать весь функционал компонента.

Системное тестирование заключается в проверке сразу всей программы на соответствие исходным требованиям. Сама программа обычно рассматривается как черный ящик. За написание и выполнение системных тестов обычно отвечает отдельный специалист – тестировщик.

unittest

Для написания компонентных тестов к коду на языке Python в Python существует удобная библиотека - unittest. unittest обеспечивает автоматизацию тестирования, поддерживает возможности объединения тестов в наборы, задания общего кода для запуска и завершения тестов, независимость самих тестов от фреймворка, генерирующего отчеты. Unittest предоставляет специальные классы, упрощающие управление наборами тестов. В связи с этим unittest использует следующие важные концепции:

- test case – тест как минимальная единица тестирования
- test suite – набор тестов (test cases) или других наборов (test suites)
- test fixture – вспомогательные действия по подготовке к запуску и зачистке после тестов
- test runner – компонент, управляющий запуском тестов и предоставлением отчета

unittest

Каркас тестов в PyCharm можно создавать автоматически, просто выбрав в контекстном меню для функции или класса: Go To -> Test -> Create New Test (или то же самое через меню Navigate).

- В unittest тесты представлены экземплярами класса TestCase
- Для создания своих тестов необходимо порождать подклассы TestCase или FunctionTestCase
- Экземпляр подкласса TestCase может выполнить один метод runTest() с необязательными методами подготовки (setUp) и зачистки (tearDown)

unittest: пример

```
# пример - тестирует методы строк
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('FOO', 'foo'.upper())

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # Проверим, что s.split не работает, если разделитель - не строка
        with self.assertRaises(TypeError):
            s.split(2)


if __name__ == '__main__':
    # обеспечиваем возможность запуска тестового скрипта из консоли
    unittest.main()
```

unittest: пример

```
# пример - добавляем методы setUp и tearDown
import unittest

class TestStringMethods(unittest.TestCase):
    def setUp(self):
        print('start')

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # Проверим, что s.split не работает, если разделитель - не строка
        with self.assertRaises(TypeError):
            s.split(2)

    def tearDown(self):
        print('end')


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

unittest: пример

Запуск тестов в консоли:

```
$ python3 -m unittest test_string_methods.py
start
end
.start
end
.start
end
.
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

unittest

Примеры проверок:

Метод	Проверяет
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises exc
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises exc and the message matches regex r

unittest: пример

Пропуск тестов и ожидаемые падения:

```
import unittest
import sys

class MyTestCase(unittest.TestCase):
    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(sys.version_info.minor != 6,
                     "not supported in this python version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

unittest: пример

Пропуск тестов и ожидаемые падения (опция –v позволяет получить более детальный отчет):

```
$ python3 -m unittest -v test_skip.py
test_format (tests_skip.MyTestCase) ... skipped 'not supported in this
python version'
test_nothing (tests_skip.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (tests_skip.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK (skipped=3)
```

Можно запускать отдельные методы тестового класса:

```
$ python3 -m unittest -v test_skip.MyTestCase.test_format
test_format (tests_skip.MyTestCase) ... skipped 'not supported in this
python version'

----------------------------------------------------------------------
Ran 1 test in 0.000s
```

unittest: соглашения

При создании тестовых скриптов используются следующие принципы наименования и расположения файлов:

- Если тестируется код в файле module.py, скрипт с юнит-тестами к этому файлу должен называться test_module.py.
- При этом скрипт test_module.py можно разместить:
  - В той же папке, что и module.py.
  - В папке ../tests/test_module.py (в отдельной папке для тестов, но на том же уровне, что и файл с кодом).
  - В папке tests/test_module.py (в отдельной папке для тестов, которая расположена в той же папке, что и файл с кодом).

unittest: TestRunner

Для запуска всех тестов в папке необходимо создать TestRunner, либо воспользоваться опцией discover.

```
import unittest

testmodules= ['test_string_methods', 'test_skip' ]


if __name__ == '__main__':
    suite = unittest.TestSuite()
    for tm in testmodules:
        suite.addTest(unittest.defaultTestLoader.loadTestsFromName(tm))
    unittest.TextTestRunner().run(suite)
```

unittest: TestRunner

Результат запуска TestRunner:

```
$ python3 testrunner.py
start
end
.start
end
.start
end
.ss.
----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK (skipped=2)
```

Использование опции discover:

```
$ python3 -m unittest discover . test_*.py
ss.start
end
.start
end
.start
end
.
----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK (skipped=2)
```

coverage

Для оценки покрытия кода тестами используется скрипт coverage:

```
$ pip3 install coverage
$ coverage run test_string_methods.py

$ coverage report
Name                      Stmts   Miss  Cover
-------------------------------------------
test_string_methods.py       18      0   100%
```

Можно получить отчет coverage в более удобном формате html (файл htmlcov/index.html):

```
$ coverage html
```

Coverage report: 44%

Module ↓	statements	missing	excluded	coverage
test_string_methods.py	16	9	0	44%
Total	**16**	**9**	**0**	**44%**

nose

nose является дополнением unittest, еще больше упрощая тестирование. Он автоматически находит тесты для запуска, имеет множество плагинов для запуска отдельных тестов, использования coverage и т.д.

```
from nose.tools import assert_equals, raises

class TestWithNose:
    def setup(self):
        print('Before test case')
    def teardown(self):
        print('After test case')
    @classmethod
    def setup_class(cls):
        print('Before test suite')
    @classmethod
    def teardown_class(cls):
        print('After test suite')
    def test_numbers_5_6(self):
        assert_equals(5 * 6, 30)
    def test_strings_b_2(self):
        assert_equals('b' * 2, 'bb')
    @raises(ZeroDivisionError)
    def test_zero_division(self):
        a = 10 / 0
```

nose

С помощью nose можно управлять запуском тестов, написанных с использованием как nose, так и unittest.

```
$ python3 -m nose -v
test_format (test_skip.MyTestCase) ... SKIP: not supported in this python
version
test_nothing (test_skip.MyTestCase) ... SKIP: demonstrating skipping
test_windows_support (test_skip.MyTestCase) ... ok
test_isupper (test_string_methods.TestStringMethods) ... ok
test_split (test_string_methods.TestStringMethods) ... ok
test_upper (test_string_methods.TestStringMethods) ... ok
test_mylen (test_tasks.TestTasks) ... ok
test_myrange (test_tasks.TestTasks) ... ok
test_reversed (test_tasks.TestTasks) ... ok
test_with_nose.TestWithNose.test_numbers_5_6 ... ok
test_with_nose.TestWithNose.test_strings_b_2 ... ok
test_with_nose.TestWithNose.test_zero_division ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.031s

OK (SKIP=2)
```

nose: coverage, пример

nose также позволяет вызывать coverage для оценки покрытия кода тестами.

```
# Файл tasks.py
def mylen(list_arg):
    res = 0
    for i in list_arg:
        res += 1
    return res

def myrange(start=0, stop=None, step=1):
    if not stop:
        stop, start = start, 0
    if step == 0:
        f = lambda a, b: False
    elif step > 0:
        f = lambda a, b: a < b
    else:
        f = lambda a, b: a > b
    i = start
    l = []
    while f(i, stop):
        l.append(i)
        i += step
    return l
```

nose: coverage, пример

```
# Файл test_tasks.py
import unittest
from tasks import mylen, myrange

class TestTasks(unittest.TestCase):
    def test_mylen(self):
        l = [1, 45, 32, 9]
        self.assertEqual(mylen(l), 4)
    def test_myrange(self):
        self.assertEqual(myrange(5), list(range(5)))
        self.assertEqual(myrange(10, 20), list(range(10, 20)))
        self.assertEqual(myrange(10, 20, 3), list(range(10, 20, 3)))
        self.assertEqual(myrange(20, 10, -2), list(range(20, 10, -2)))
        #self.assertEqual(myrange(20, 10, 0), list(range(20, 10, 0)))


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

nose: coverage, пример

Запуск теста с построением coverage report:

```
$ python3 -m nose test_tasks.py -v --with-coverage --cover-erase --cover-
package=tasks --cover-html
test_mylen (test_tasks.TestTasks) ... ok
test_myrange (test_tasks.TestTasks) ... ok

Name        Stmts   Miss  Cover
-----------------------------
tasks.py       20      1    95%
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Coverage report в формате html (файл cover/index.html):

Coverage report: 95%

Module ↓	*statements*	*missing*	*excluded*	*coverage*
tasks.py	20	1	0	95%
Total	**20**	**1**	**0**	**95%**

nose: coverage, пример

Покрытие кода тестами (файл cover/tasks_py.html):

Coverage for **tasks.py** : 95%

20 statements | 19 run | 1 missing | 0 excluded

```
1  def mylen(list_arg):
2      res = 0
3      for i in list_arg:
4          res += 1
5      return res
6
7  def myrange(start=0, stop=None, step=1):
8       if not stop:
9           stop = start
10          start = 0
11      if step == 0:
12          f = lambda a, b: False
13      elif step > 0:
14          f = lambda a, b: a < b
15      else:
16          f = lambda a, b: a > b
17      i = start
18      l = []
19      while f(i, stop):
20          l.append(i)
21          i += step
22      return l
```

nose: coverage vs unittest: coverage

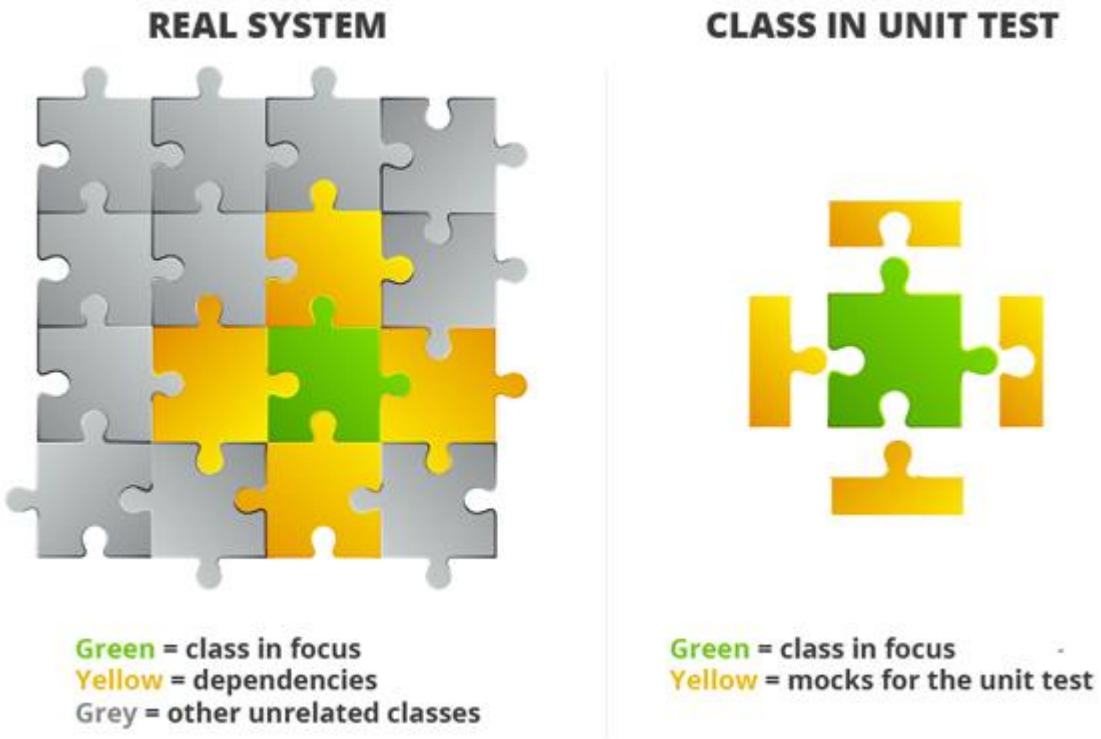
Для получения аналогичного отчета с использованием unittest потребовалось бы выполнить следующий набор команд:

```
$ coverage erase
$ coverage run --source tasks test_tasks.py
$ coverage report -m
$ coverage html
```

mock

Тестируемые компоненты в большинстве случаев имеют зависимости от других компонентов, библиотек, драйверов. Чтобы протестировать такой компонент изолированно надо заменить все его зависимости на mock-объекты - «заглушки», работающие в соответствии с тестовыми сценариями.
Для создания и использования mock-объектов в Python существует библиотека mock. Она позволяет подменять компоненты тестируемой системы mock-объектами и проверять обращения к этим объектам. Например, мы тестируем клиентский модуль, получающий какие-то данные от HTTP сервера. Использовать реальный сервер в данном случае неправильно, т.к. мы тестируем именно код клиента и проверять заодно внешние зависимости было бы излишней работой (это уже относится к системному тестированию). Все, что нам нужно, это получить правильный ответ от сервера, т.е. ответ, предусмотренный тестовым сценарием. В этом случае достаточно использовать mock-объект, чтобы эмулировать ответ, который нам нужен. Таким способом проверить все возможные ответы от сервера, не проверяя и не используя сам сервер.

mock


REAL SYSTEM
Green = class in focus
Yellow = dependencies
Grey = other unrelated classes
CLASS IN UNIT TEST
Green = class in focus
Yellow = mocks for the unit test

mock: пример

```
# Файл cmd_manager.py

class AuditManager:
    def add_call(self, *args, **kwargs):
        pass
    def add_result(self, *args, **kwargs):
        pass

audit_service = AuditManager()


class CmdManager:
    def set_service(self, service):
        self.service = service
        self.service.start()
    def run_command(self, cmd):
        audit_service.add_call(self.service.name, 'run', cmd)
        result = self.service.run(cmd)
        audit_service.add_result(self.service.name, 'run', result)
```

НИИТ

mock: пример

Мы будем тестировать класс CmdManager, объект audit_service нас интересует постольку, поскольку к нему идут обращения от методов CmdManager. Эти обращения можно рассматривать как выходные данные тестируемого класса CmdManager. Поэтому сам объект audit_service мы заменяем mock-объектом.

Метод set_service объекта класса CmdManager принимает параметр service, о котором мы знаем только то, что у него есть атрибут name и метод run, который должен что-то возвращать. В качестве фактического параметра service мы тоже можем использовать mock-объект, предварительно добавив в него требуемые в рамках теста атрибуты и заглушки для методов.

mock: пример

```
# Файл test_cmd_manager.py
import cmd_manager
import mock
import unittest

class TestCmdManager(unittest.TestCase):

    def test_cmd_manager(self):
        cmd_mgr = cmd_manager.CmdManager()

        # Подменяем audit_service mock-объектом
        with mock.patch('cmd_manager.audit_service') as audit_service_mock:
            # Вместо параметра service тоже используем mock-объект
            service = mock.Mock()
            service.name = 'MyService'
            service.run.return_value = 0
            cmd_mgr.set_service(service)
            self.assertTrue(service.start.called)
            cmd_mgr.run_command('status')
            # Используем assert-методы mock-объекта, которым мы подменили audit_service
            audit_service_mock.add_call.assert_called_once_with('MyService', 'run', 'status')
            service.run.assert_called_once_with('status')
            audit_service_mock.add_result.assert_called_once_with('MyService', 'run', 0)

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

mock: пример

Аналогичным, но несколько более изящным решением является использование декоратора @mock.patch.

```
import cmd_manager
import mock
import unittest

class TestCmdManager(unittest.TestCase):

    @mock.patch('cmd_manager.audit_service')
    def test_cmd_manager(self, audit_service_mock):
        cmd_mgr = cmd_manager.CmdManager()
        # Вместо параметра service тоже используем mock-объект
        service = mock.Mock()
        service.name = 'MyService'
        service.run.return_value = 0
        cmd_mgr.set_service(service)
        self.assertTrue(service.start.called)
        cmd_mgr.run_command('status')
        # Используем assert-методы mock-объекта, которым мы подменили audit_service
        audit_service_mock.add_call.assert_called_once_with('MyService', 'run', 'status')
        service.run.assert_called_once_with('status')
        audit_service_mock.add_result.assert_called_once_with('MyService', 'run', 0)


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

mock: принцип работы

Для того, чтоб правильно использовать mock, надо понимать, как он работает. Патчинг некоторого объекта заключается в том, что мы мОчим (заменяем) какой-то подобъект внутри этого объекта. Патчить можно все, что импортируется, а вот мОчить (подменять) - все импортируемое, кроме модулей. Иначе говоря, при патчинге мы заменяем связь между объектом и его подобъектом на связь объекта с фейковым mock-объектом.

Было: объект ==> подобъект

Стало: объект = X => подобъект

||

=> mock-объект

mock: @mock.patch

@mock.patch принимает в качестве первого аргумента любой импортируемый объект, который мы хотим заменить. То, на что мы хотим заменить, можно передать вторым аргументом, а можно указать дальше внутри декорируемой функции.

@mock.patch.dict принимает в качестве первого аргумента словарь, который мы хотим заменить. То, на что мы хотим заменить можно передать вторым аргументом, а можно указать дальше внутри декорируемой функции. Если не выставлен флаг clear=True, то исходный словарь не заменяется, а дополняется.

@mock.patch.object принимает в качестве первого аргумента реальный объект, который мы хотим пропатчить, в качестве второго аргумента - имя атрибута, который хотим заменить. То, на что мы хотим заменить можно передать третьим аргументом, а можно указать дальше внутри декорируемой функции.

mock: side_effect

```
import mock

# Тестируемый класс
class Listener:
    def __init__(self):
        self.running = True
    def listen_forever(self):
        while self.running:
            try:
                msg = msg_broker.wait_message()
                client.process_message(msg)
            except Exception:
                self.running = False
```

mock: side_effect

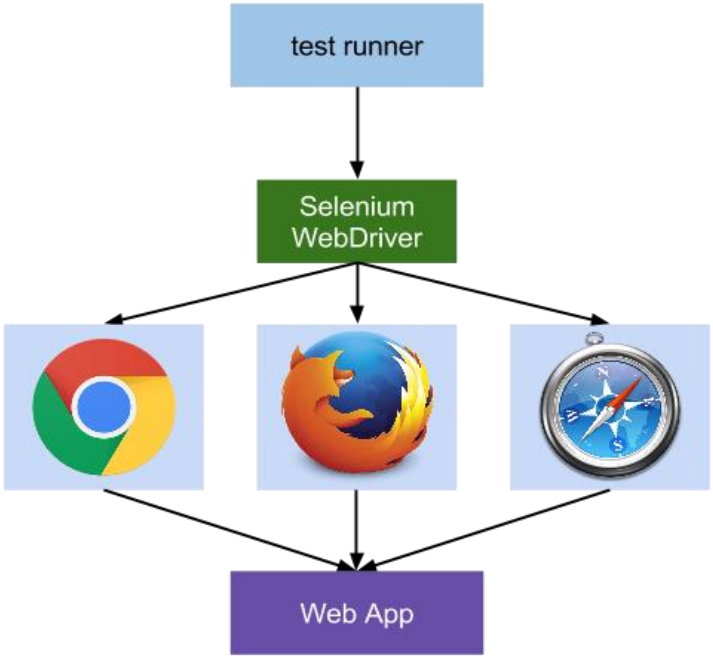
```
# Подготовка к тестированию
# Создаем объект тестируемого класса
listener = Listener()
# Заменяем зависимости на mock-объекты
msg_broker = mock.Mock()
client = mock.Mock()
# Настраиваем поведение mock-объектов
# При инициализации атрибута side_effect итерируемым объектом
# при каждом обращении к wait_message будет возвращаться очередной
# элемент итерируемого объекта (сначала 'message', затем Exception).
msg_broker.wait_message.side_effect = ['message', Exception]

def check_msg(msg):
    assert msg == 'message'
# При инициализации атрибута side_effect функциональным (callable)
# объектом при каждом обращении к process_message будет вызываться
# этот функциональный объект.
client.process_message.side_effect = check_msg


# Тестируем и выполняем проверки
listener.listen_forever()
assert listener.running == False
client.process_message.assert_called_once_with('message')
assert msg_broker.wait_message.call_count == 2
```

Системное тестирование

Тестирование программы может быть ручным, автоматизированным и полуавтоматизированным. Естественно, самым эффективным является автоматизированное тестирование: в этом случае тестовые программы/скрипты, однажды написанные для проверки соответствия программы новым требованиям, могут применяться многократно и быстро, становясь частью набора регрессионных тестов. Это позволяет создавать CI/CD решения, практически полностью автоматизирующие часть цикла разработки программного обеспечения от деливери до релиза и деплоймента.
Python, сам по себе являясь эффективным средством создания тестовых скриптов за счет своей простоты и скорости разработки на языке, поддерживает также различные библиотеки, еще больше упрощающие создание тестов. Примерами таких библиотек являются selenium и robotframework.

selenium

Selenium - это портируемый фреймворк для тестирования программного обеспечения для веб-приложений. Тесты, написанные с его помощью, могут быть запущены в большинстве современных браузеров. Selenium поддерживает Windows, Linux, и Macintosh платформы.



selenium: WebDriver и API

Привязка Selenium к Python предоставляет собой простой API для написания функциональных тестов с использованием веб-драйвера Selenium WebDriver.
Для тестирования необходимо скачать WebDriver для соответствующего браузера. WebDriver – специальное программное обеспечение для автоматического управления браузером. WebDriver можно скачать здесь: https://sites.google.com/a/chromium.org/chromedriver/downloads - для Chrome, https://github.com/mozilla/geckodriver/releases - для Firefox (установка не требуется, в Ubuntu надо прокинуть драйвер в /usr/local/bin). Также необходимо скачать и установить библиотеку selenium с помощью pip.
Selenium API предоставляет множество возможностей. В том числе:

- поиск одного или нескольких элементов;
- поиск по имени/id/ссылке/тегу/классу/css селектору/xpath;
- ожидать изменения/появления/отображения заголовка/элемента;
- выполнять код JavaScript в текущем окне;
- взаимодействовать с элементами UI (клик/ввод текста/прокрутка).

selenium: пример

```
from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Создаем объект для работы с веб-драйвером Chrome
driver = webdriver.Chrome('F:\Python\Soft\chromedriver_win32\chromedriver.exe')
# Идем на домашнюю страницу google
driver.get('http://www.google.com')
# Выводим в консоль заголовок страницы - "Google"
print(driver.title)
# Ищем элемент, чей атрибут name == 'q' ('q' - имя окна поиска qooqle)
inputElement = driver.find_element_by_name('q')
# Вводим текст в окно поиска
inputElement.send_keys('cheese!')
# Выполняем submit (нажатие Enter или кнопки Поиск) для поиска # (хотя google сейчас
выполняет поиск и без submit)
inputElement.submit()
try:
    # ждем обновления страницы (заголовок обычно обновляется последним)
    WebDriverWait(driver, 10).until(EC.title_contains('cheese!'))
    # Выводим в консоль текущий заголовок - "cheese! - Поиск в Google"
    print(driver.title)
except TimeoutException:
    print('Timeout exception!')
finally:
    driver.quit()
```

Robot Framework

Robot Framework - это фреймворк для разработки приемочных автотестов. Тестовые сценарии пишутся с использованием keyword testing методики тестирования и оформляются в виде таблицы. Эти таблицы можно записать в виде простого текста, HTML, разделенных табуляцией значений (TSV) или reStructuredText (reST) в любом текстовом редакторе или с помощью интегрированной среды разработки Robot (Robot Integrated Development Environment, RIDE). PyCharm имеет соответствующий плагин для контроля синтаксиса в .robot файлах.

Robot Framework написан на Python и может напрямую использовать функции из .py файлов в качестве keywords. Это позволяет легко дополнять функционал фреймворка, уже представленный множеством расширений, пользовательскими модулями на языке Python.

Robot Framework использует отступы и многоточия для отделения блоков кода, а также два или более пробелов для отделения keywords от передаваемых им параметров.

robotframework

Библиотека robotframework устанавливается при помощи pip:

```
$ pip3 install robotframework --user
```

Необходимые расширения устанавливаются апгрейдом библиотеки:

```
$ pip3 install --upgrade robotframework-seleniumlibrary
```

Тесты пишутся в файлах .robot. При открытии таких файлов в PyCharm, последний предлагает установить соответствующий плагин, после чего включает контроль синтаксиса и автодополнение.

robotframework: пример

Тестовый файл для поиска в google с использованием selenium будет выглядеть следующим образом:

```
*** Settings ***
Documentation  Test google search
Library        SeleniumLibrary


*** Test Cases ***
Test Google Search
    Open Browser  http://www.google.com  chrome
    Input Text  name=q  Cheese!
    Submit Form
    Wait Until Keyword Succeeds  10s  2s
    ...  Title Should Be  Cheese! - Поиск в Google
```

Для выполнения теста в Windows необходимо скачать WebDriver для Chrome и прописать в PATH полный путь к этому драйверу (например, F:\Python\Soft\chromedriver_win32\chromedriver.exe). В Ubuntu достаточно, чтоб драйвер находился в /usr/local/bin.

robotframework: пример

Запуск теста выполняется из консоли:

```
F:\Python\PyPractice>robot -L TRACE test_robot_sellib.robot
==============================================================================
Test Robot Sellib :: Test google search
==============================================================================
Test Google Search
DevTools listening on ws://127.0.0.1:53713/devtools/browser/61c3f426-7628-450e-
aadc-ddcb470a3646
Test Google Search                                                    | PASS |
------------------------------------------------------------------------------
Test Robot Sellib :: Test google search                               | PASS |
1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
==============================================================================
Output:  F:\Python\PyPractice\output.xml
Log:     F:\Python\PyPractice\log.html
Report:  F:\Python\PyPractice\report.html
```

Опция “–L TRACE” позволяет пошагово вывести детали выполнения теста в html-отчет (log.html).

robotframework: пример

Test Robot Sellib Test Log

Generated
20181202 02:50:23 GMT+03:00
10 minutes 8 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:07	
All Tests	1	1	0	00:00:07	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Test Robot Sellib	1	1	0	00:00:07	

Test Execution Log

SUITE **Test Robot Sellib**

Full Name:	Test Robot Sellib
Documentation:	Test google search
Source:	F:\Python\PyPractice\TrenPro\training3\lec_12\test_robot_sellib.robot
Start / End / Elapsed:	20181202 02:50:16.315 / 20181202 02:50:23.773 / 00:00:07.458
Status:	1 critical test, 1 passed, 0 failed 1 test total, 1 passed, 0 failed

TEST **Test Google Search**

Full Name:	Test Robot Sellib.Test Google Search
Start / End / Elapsed:	20181202 02:50:16.549 / 20181202 02:50:23.772 / 00:00:07.223
Status:	PASS (critical)

KEYWORD SeleniumLibrary . **Open Browser** http://www.google.com, chrome

KEYWORD SeleniumLibrary . **Input Text** name=q, Cheese!

KEYWORD SeleniumLibrary . **Submit Form**

KEYWORD BuiltIn . **Wait Until Keyword Succeeds** 10s, 2s, Title Should Be, Cheese! - Поиск в Google

robotframework: пример

Рассмотрим пример использования robotframework для тестирования скрипта, копирующего файл source.txt в destination.txt

```
# Файл copy_file_task.py
def copyfile(source, destination):
    with open(source, 'r') as src:
        with open(destination, 'x') as dst:
            dst.writelines(src.readlines())


if __name__ == '__main__':
    copyfile('source.txt', 'destination.txt')
```

robotframework: пример

```
# Файл test_copy_file.robot

*** Settings ***
Documentation   Check file actions
Library         OperatingSystem
Test Setup      On Test Setup
Test Teardown   On Test Teardown

*** Variables ***
${copy_script}  python ./copy_file_task.py
${src_file}     ./source.txt
${dst_file}     ./destination.txt
${exp_content}  hello

*** Keywords ***
On Test Setup
    Create File  ${src_file}  ${exp_content}
    File Should Exist  ${src_file}

On Test Teardown
    Remove File  ${src_file}
    Remove File  ${dst_file}
```

robotframework: пример

```
*** Test Cases ***
Test File Copy
    [Documentation]  Test file copy script
    [Tags]  DEBUG
    File Should Not Exist  ${dst_file}
    Run  ${copy_script}
    File Should Exist  ${dst_file}
    ${content}=  Get File  ${dst_file}
    Should Be True  '${content}' == '${exp_content}'
```

Тестовый файл делится на секции Settings, Variables, Keywords и Test Cases.
Тест кейсы нельзя вызывать друг из друга и из кейвордов, кейворды можно вызывать отовсюду.
Переменные обозначаются следующим образом: ${имя_переменной}.
Ключевые слова Test Setup и Test Teardown позволяют задать кейворды, которые буду выполняться до и после каждого тест кейса.
Ключевые слова Suite Setup и Suite Teardown позволяют задать кейворды, которые буду выполняться до и после всего набора тестов из файла.

robotframework: пример

Результат запуска теста:

```
F:\Python\PyPractice>robot -L TRACE test_copy_file.robot
==============================================================================
Test Copy File :: Check file actions
==============================================================================
Test File Copy :: Test file copy script                               | PASS |
------------------------------------------------------------------------------
Test Copy File :: Check file actions                                  | PASS |
1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
==============================================================================
Output:  F:\Python\PyPractice\output.xml
Log:     F:\Python\PyPractice\log.html
Report:  F:\Python\PyPractice\report.html
```

robotframework: пример

Test Copy File Test Log

Generated
20181202 03:19:04 GMT+03:00
7 minutes 51 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:00	
All Tests	1	1	0	00:00:00	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
DEBUG	1	1	0	00:00:00	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Test Copy File	1	1	0	00:00:00	

Test Execution Log

SUITE **Test Copy File**

Full Name:	Test Copy File
Documentation:	Check file actions
Source:	F:\Python\PyPractice\TrenPro\training3\lec_12\test_copy_file.robot
Start / End / Elapsed:	20181202 03:19:04.324 / 20181202 03:19:04.480 / 00:00:00.156
Status:	1 critical test, 1 passed, 0 failed 1 test total, 1 passed, 0 failed

TEST **Test File Copy**

Full Name:	Test Copy File.Test File Copy
Documentation:	Test file copy script
Tags:	DEBUG
Start / End / Elapsed:	20181202 03:19:04.355 / 20181202 03:19:04.480 / 00:00:00.125
Status:	PASS (critical)

- SETUP **On Test Setup**
- KEYWORD Operating System . **File Should Not Exist** ${dst_file}
- KEYWORD Operating System . **Run** ${copy_script}
- KEYWORD BuiltIn . **Wait Until Keyword Succeeds** 5s, 5s, File Should Exist, ${dst_file}
- KEYWORD ${content} = Operating System . **Get File** ${dst_file}
- KEYWORD BuiltIn . **Should Be True** '${content}' == '${exp_content}'
- TEARDOWN **On Test Teardown**

Практика

1. Написать функцию to_roman, которая принимает целое число, а возвращает строку, отображающую это число римскими цифрами. Например, на вход подается 6, возвращается - "VI"; на вход подается 23, возвращается "XXIII". Входные данные должны быть в диапазоне от 1 до 5000, если подается число не из этого диапазона, или не число, то должны выбрасываться ошибка типа NonValidInput. Этот тип ошибки надо создать отдельно. Также необходимо в папке с файлом, содержащим вашу функцию, создать файл tests.py, внутри которой необходимо определить тесты для вашей функции. Тесты должны покрывать все возможное поведение функции, включая порождения ошибки при некорректных входных данных.
2. Написать юнит тесты для класса Money из задания 10 к лекции 7, получить code coverage репорт в html формате.
3. * Написать юнит тесты для TCP-сервера (код см. далее) из лекции 9, использовать mock чтобы эмулировать действия клиента и создание потоков, получить code coverage репорт в html формате.

Практика

```
class TcpServer:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self._socket = None
        self._runnning = False

    def run(self):
        self._socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self._socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self._socket.bind((self.host, self.port))
        self._socket.listen(5)
        self._runnning = True
        print('Server is up')
        while self._runnning:
            conn, addr = self._socket.accept()
            ClientThread(conn, addr).start()

    def stop(self):
        self._runnning = False
        self._socket.close()
        print('Server is down')

if __name__ == '__main__':
    srv = TcpServer(host='127.0.0.1', port=5555)
    try:
        srv.run()
    except KeyboardInterrupt:
        srv.stop()
```