Продвинутый курс Python

Урок 1

# Работа с сетью, сокеты. Тестирование кода

PEP-8. Основы разработки сетевых приложений, сокеты. Модульное тестирование. Инструкция assert. Модули doctest, unittest. Библиотека PyTest, фикстуры, параметризация тестов.

# Введение

В курсе “Python-1” слушатели изучили основы программирования и особенности синтаксиса языка Python, научились решать небольшие задачи на практике.

Данный курс является продолжением курса “Python-1”. Слушателям предлагается продолжить знакомство с возможностями и особенностями языка программирования Python. В рамках данного курса будет продолжено знакомство с ООП в Python, будет изучаться взаимодействие с базами данных, создание графического интерфейса пользователя, создание многопоточных приложений, тестирование кода, также планируется рассмотреть вопросы обеспечения информационной безопасности в процессе разработки приложений. Слушателям стоит быть готовым к тому, что многие базовые темы не будут повторно рассказаны в данном курсе.

В курсе “Python-2” другой характер будут носить и домашние задания - слушателям будет предложено реализовать полноценную клиент-серверную систему обмена сообщениями на Python, с каждой новой темой добавляя новый функционал в приложение.

Реализация клиент-серверного взаимодействия обязывает в первую очередь приступить к изучению основ сетевого взаимодействия и реализации их на языке Python. В ходе курса тема сетевого взаимодействия будет дополняться постепенно в каждом уроке.

Но перед этим обратим внимание на рекомендации PEP-8.

# PEP-8. Внешний вид кода

В Python существует руководство по написанию легкочитаемого кода **PEP-8**. PEP расшифровывается как Python Enhancement Proposals (предложение по развитию Python). Документ под номером 8 - правила написания кода. Рассмотрим их кратко (см. также шпаргалку pep8_cheatsheet.pdf в материалах урока и полную версию PEP-8 ).

## Отступы

Для отступа используйте 4 пробела или табуляцию. Нельзя смешивать символы табуляции и пробелы. Лучше использовать пробелы.

## Максимальная длина строки

Ограничьте максимальную длину строки 79 символами.

## Пустые строки

Отделяйте функции (верхнего уровня, не функции внутри функций) и определения классов двумя пустыми строчками. Определения методов внутри класса отделяйте одной пустой строкой. Дополнительные отступы строками могут быть изредка использованы для выделения группы логически связанных функций. Пустые строки могут быть пропущены, между несколькими выражениями, записанными в одну строку, например, “заглушки” функций.

## Кодировки (PEP 263)

Начиная с версии Python 3.0 предпочтительной является кодировка UTF-8.

## import-секции

Импортирование разных модулей должно быть на разных строчках.

Правильно:

```
import os
import sys
```

Неправильно:

```
import os,sys
```

В то же время, можно писать вот так:

```
from subprocess import Popen, PIPE
```

Импортирование всегда нужно делать сразу после комментариев к модулю и строк документации, перед объявлением глобальных переменных и констант. Группируйте импорты в следующем порядке:

- импорты стандартной библиотеки;
- импорты сторонних библиотек;
- импорты модулей текущего проекта.

Вставляйте пустую строку между каждой группой импортов. Относительные импорты крайне не рекомендуются — всегда указывайте абсолютный путь к модулю для всех импортирований.

## Пробелы в выражениях и инструкциях

Избегайте использования пробелов в следующих ситуациях:

1. Сразу после или перед скобками (обычными, фигурными и квадратными).

Можно:

```
spam(ham[1], {eggs: 2})
```

Нельзя:

```
spam( ham[ 1 ], { eggs: 2 })
```

2. Сразу перед запятой, точкой с запятой, двоеточием.

Можно:

```
if x == 4: print x, y; x, y = y, x
```

Нельзя:

```
if x == 4 : print x , y ; x , y = y , x
```

3. Сразу перед открывающей скобкой, после которой начинается список аргументов при вызове функции.

Можно:

```
spam(1)
```

Нельзя:

```
spam (1)
```

4. Сразу перед открывающей скобкой, после которой следует индекс или срез.

Можно:

```
dict['key'] = list[index]
```

Нельзя:

```
dict ['key'] = list [index]
```

5. Использование более одного пробела вокруг оператора присваивания (или любого другого) для того, чтобы выровнять его с другим таким же оператором на соседней строке.

Можно:

```
x = 1
y = 2
long_variable = 3
```

Нельзя:

```
x             = 1
y             = 2
long_variable = 3
```

### Комментарии

Комментарии, которые противоречат коду, хуже, чем отсутствие комментариев. Всегда исправляйте комментарии, если меняете код. Комментарии должны являться законченными предложениями. Если комментарий — фраза или предложение, первое слово должно быть написано с большой буквы, если

только это не имя переменной, которая начинается с маленькой буквы (кстати, никогда не отступайте от этого правила для имен переменных). Если комментарий короткий, можно опустить точку в конце предложения. Блок комментариев обычно состоит из одного или более абзацев, составленных из полноценных предложений, поэтому каждое предложение должно оканчиваться точкой.

### Блок комментариев

Блок комментариев обычно объясняет код (весь, или только некоторую часть), идущий после блока, и должен иметь тот же отступ, что и сам код. Каждая строчка такого блока должна начинаться с символа # и одного пробела после него (если только сам текст комментария не имеет отступа). Абзацы внутри блока комментариев лучше отделять строкой, состоящей из одного символа #.

### Комментарии в строке с кодом

Старайтесь реже использовать подобные комментарии.

### Соглашению по именованию

Также важно правильно именовать переменные, функции и классы.

## Стили имен

Никогда не используйте символы l (маленькая латинская буква «эль»), O (заглавная латинская буква «о») или I (заглавная латинская буква «ай») как однобуквенные идентификаторы. В некоторых шрифтах эти символы неотличимы от цифры: один и нуль. Если очень нужно использовать l имена, пишите вместо неё заглавную L.

### Имена модулей и пакетов

Модули должны иметь короткие имена, состоящие из маленьких букв. Можно использовать и символы подчеркивания, если это улучшает читабельность. То же, за исключением символов подчеркивания, относится и к именам пакетов.

### Имена классов

Все имена классов должны следовать соглашению CapWords почти без исключений. Классы внутреннего использования могут начинаться с символа подчеркивания.

### Имена исключений (exceptions)

Так как исключения являются классами, к исключениям применяется стиль именования классов. Однако вы можете добавить Error в конце имени (если конечно исключение действительно является ошибкой).

### Имена функций

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания - это необходимо, чтобы увеличить читабельность.

### Аргументы функций и методов

Всегда используйте self в качестве первого аргумента метода экземпляра объекта. Всегда используйте cls в качестве первого аргумента метода класса.

Если имя аргумента конфликтует с зарезервированным ключевым словом python, обычно лучше добавить в конец имени символ подчеркивания, чем исказить написание слова или использовать аббревиатуру. Таким образом, print_ лучше, чем prnt. (Возможно, хорошим вариантом будет подобрать синоним).

**Имена методов и переменные экземпляров классов**

Используйте тот же стиль, что и для имен функций: имена должны состоять из маленьких букв, а слова разделяться символами подчеркивания. Чтобы избежать конфликта имен с подклассами, добавьте два символа подчеркивания, чтобы включить механизм изменения имен. Если класс Foo имеет атрибут с именем __foo, к нему нельзя обратиться, написав Foo.__a. (Настойчивый пользователь всё равно может получить доступ, написав Foo._Foo__a). Вообще, двойное подчеркивание в именах должно использоваться, чтобы избежать конфликта имен с атрибутами классов, спроектированных так, чтобы от них наследовали подклассы.

**Константы**

Константы обычно объявляются на уровне модуля и записываются только заглавными буквами, а слова разделяются символами подчеркивания. Например: MAX_OVERFLOW, TOTAL.

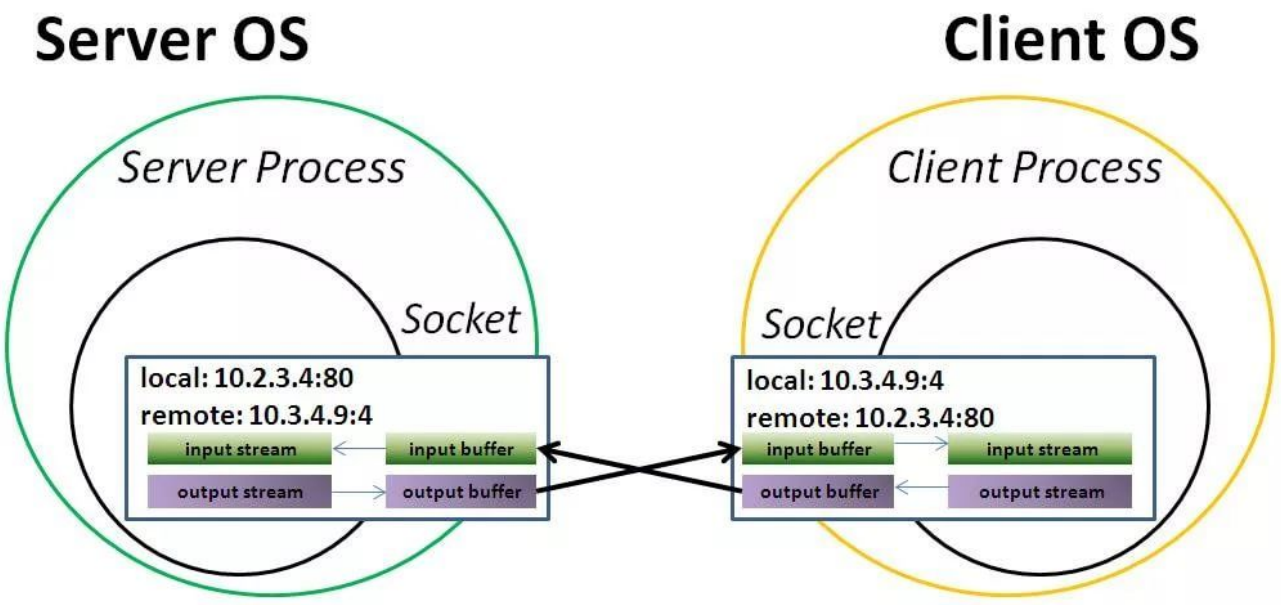
# Работа с сетью и сокеты

Стандартная библиотека языка Python обеспечивает широкую поддержку сетевых операций, начиная от операций с сокетами и заканчивая функциями для работы с прикладными протоколами высокого уровня, такими как HTTP. Для начала будет дано краткое введение в разработку сетевых приложений. За дополнительной информацией читателям рекомендуется обращаться к специализированным книгам, таким как “UNIX. Разработка сетевых приложений” У. Ричарда Стивенса (W. Richard Stevens).

## Основы разработки сетевых приложений

Модули, входящие в стандартную библиотеку Python и предназначенные для разработки сетевых приложений, главным образом поддерживают два протокола Интернета: TCP и UDP. Протокол TCP – это надежный протокол с созданием логического соединения, используемый для создания между компьютерами двустороннего канала обмена данными. Протокол UDP – это низкоуровневый протокол, обеспечивающий возможность обмена пакетами, с помощью которого компьютеры могут отправлять и получать информацию в виде отдельных пакетов, без создания логического соединения. В отличие от TCP, взаимодействия по протоколу UDP не отличаются надежностью, что усложняет управление ими в приложениях, в которых необходимо гарантировать надежность обмена информацией. По этой причине большинство интернет-приложений используют протокол TCP.

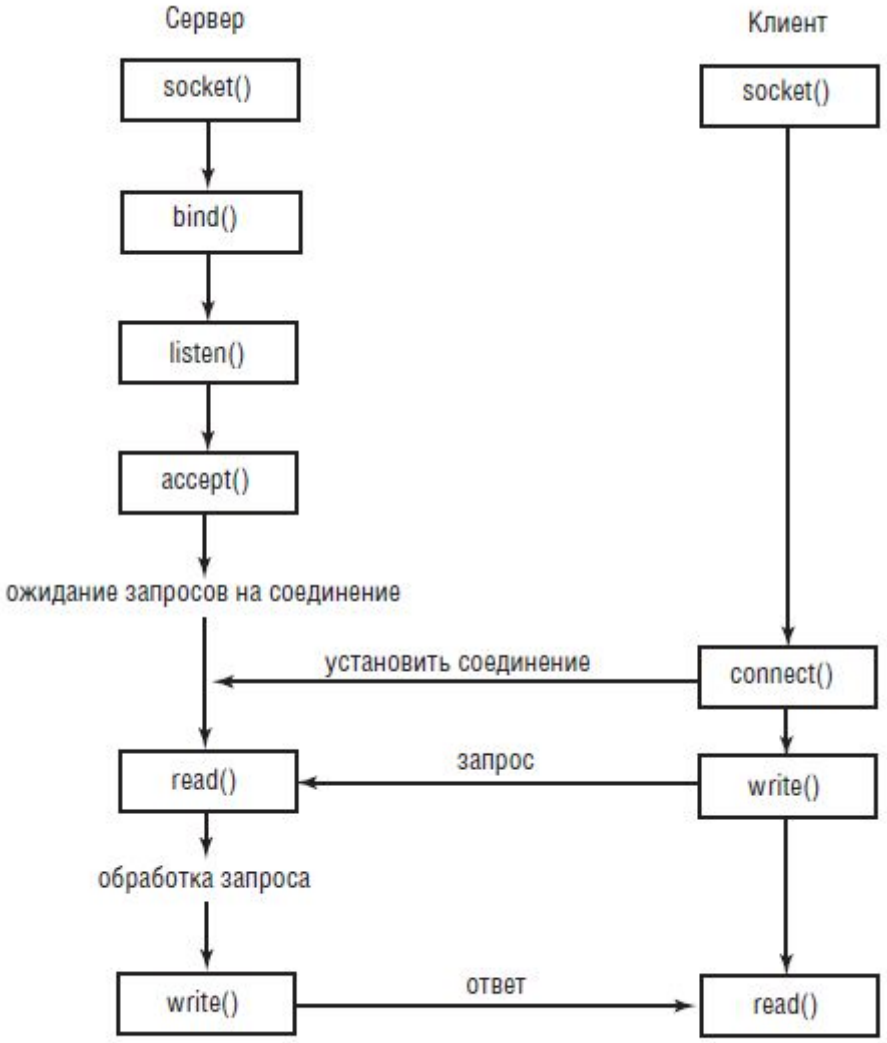
Работа с обоими протоколами осуществляется с помощью программной абстракции, известной как *сокет*. **Сокет** – это объект, напоминающий файл, позволяющий программе принимать входящие соединения, устанавливать исходящие соединения, а также отправлять и принимать данные. Прежде чем два компьютера смогут обмениваться информацией, на каждом из них должен быть создан объект сокета.




Компьютер, принимающий соединение, (сервер) должен присвоить своему объекту сокета определенный номер порта. **Порт** – это 16-битное число в диапазоне 0 – 65535, которое используется клиентами для уникальной идентификации серверов. Порты с номерами 0 – 1023 зарезервированы для нужд системы и используются наиболее распространенными сетевыми протоколами. Ниже перечислены некоторые распространенные протоколы с присвоенными им номерами портов (более полный список можно найти по адресу: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml):

| Служба | Номер порта |
|---|---|
| FTP-Data | 20 |
| FTP-Control | 21 |
| SSH | 22 |
| Telnet | 23 |
| SMTP (электронная почта) | 25 |
| HTTP (WWW) | 80 |
| IMAP | 143 |
| HTTPS (безопасный WWW) | 443 |

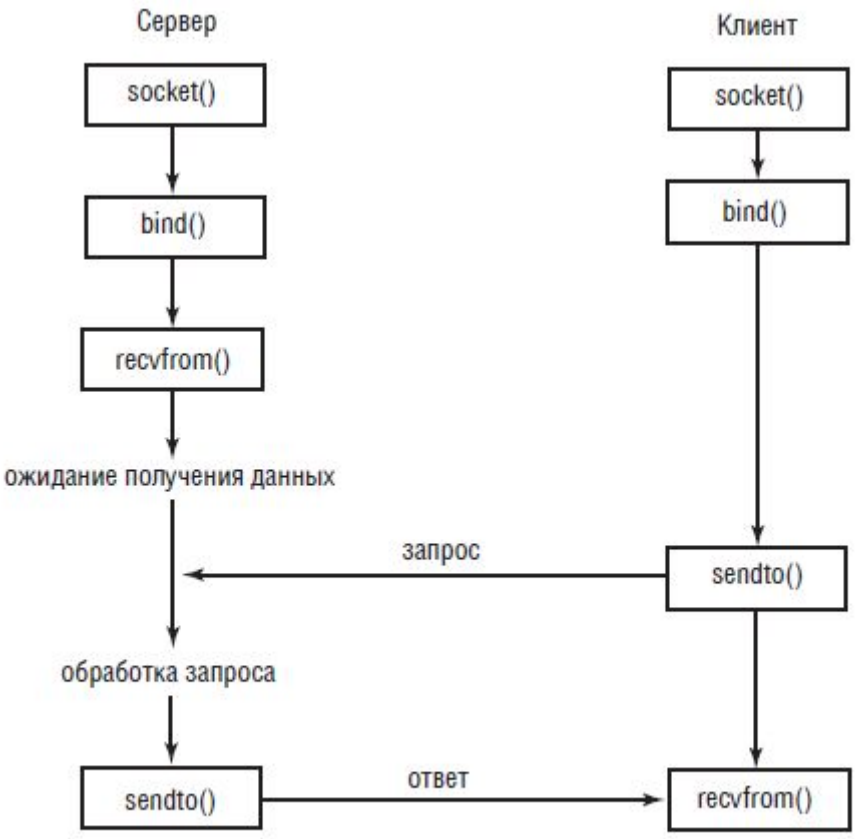
Процедура установки TCP-соединения между клиентом и сервером определяется точной последовательностью операций, как показано на рисунке ниже.




*Процедура установки TCP-соединения*

На стороне сервера, работающего по протоколу TCP, объект сокета, используемый для приема запросов на соединение, – это **не тот же самый сокет**, что в дальнейшем используется для обмена данными с клиентом. В частности, системный вызов accept() возвращает новый объект сокета, который фактически будет использоваться для обслуживания соединения. Это позволяет серверу одновременно обслуживать соединения с большим количеством клиентов.

Взаимодействия по протоколу UDP выполняются похожим способом, за исключением того, что клиенты и серверы не устанавливают логическое соединение друг с другом, как это показано на рисунке ниже:




*Обмен данными по протоколу UDP*

Следующий пример иллюстрирует применение протокола TCP-клиентом и сервером, использующими модуль socket. В этом примере сервер просто возвращает клиенту текущее время в виде строки:

```
# Программа сервера времени
from socket import *
import time

s = socket(AF_INET, SOCK_STREAM)   # Создает сокет TCP
s.bind(('', 8888))                 # Присваивает порт 8888
s.listen(5)                        # Переходит в режим ожидания запросов;
                                   # одновременно обслуживает не более
                                   # 5 запросов.
while True:
    client, addr = s.accept()      # Принять запрос на соединение
    print("Получен запрос на соединение от %s" % str(addr))
    timestr = time.ctime(time.time()) + "\n"
    client.send(timestr.encode('ascii'))
    client.close()
```

Ниже приводится клиентская программа:

```
# Программа клиента, запрашивающего текущее время
from socket import *

s = socket(AF_INET,SOCK_STREAM)   # Создать сокет TCP
s.connect(('localhost', 8888))    # Соединиться с сервером
tm = s.recv(1024)                 # Принять не более 1024 байтов данных
s.close()
print("Текущее время: %s" % tm.decode('ascii'))
```

В сетевых протоколах обмен данными часто выполняется в текстовой форме. Поэтому особое внимание необходимо уделять кодировке текста. В Python 3 все строки состоят из символов Юникода, что влечет за собой необходимость кодировать строки, передаваемые через сеть. Именно по этой причине в программе сервера к отправляемым данным применяется метод encode('ascii'). Точно так же, когда клиент принимает данные из сети, эти данные поступают в виде простой последовательности кодированных байтов. Если вывести эту последовательность на экран или попробовать интерпретировать ее как текст, результат, скорее всего, получится совсем не тот, какого вы ожидали. Поэтому, прежде чем работать с данными, их необходимо декодировать. Для этого в программе клиента применяется метод decode('ascii') к принимаемым данным (также обратите внимание на файл примером кода для кодирования строк и байтов - examples/01_sockets/str_bytes_bytearray.py).

В последующих занятиях будет продолжено знакомство с работой с сетью в Python.

# Тестирование кода

## Введение

В отличие от программ, написанных на таких языках, как C или Java, программы на языке Python не обрабатываются компилятором, который создает выполняемую программу. В этих языках программирования компилятор является первой линией обороны от программных ошибок - он отыскивает такие ошибки, как вызов функций с недопустимым количеством аргументов или присваивание недопустимых значений переменным (то есть выполняет проверку типов). В языке Python такие проверки выполняются только после запуска программы. Поэтому невозможно сказать, содержит программа ошибки или нет, пока она не будет запущена и протестирована. Но и это еще не все, - если не опробовать программу всеми возможными способами, когда поток управления пройдет все возможные ветви программы, всегда остается вероятность, что в программе скрыта какая-нибудь ошибка, которая ждет своего часа (к счастью, такие ошибки обычно обнаруживаются уже через несколько дней после передачи программы пользователю).

Данный урок посвящен изучению приемом и библиотечных модулей, используемых для тестирования программного кода на языке Python. Данная тема намеренно помещена в начало курса Python-2, чтобы в ходе работы с домашними заданиями слушатели дополнительно писали тесты для своего кода.

## Строки документирования и модуль doctest

Если первая строка функции, класса или модуля является строкой, эта строка называется строкой документирования. Включение строк документирования считается хорошим тоном, потому что эти строки используются в качестве источников информации различными инструментами разработки. Например, строки документирования можно просматривать с помощью команды help(), а также

средствами интегрированной среды разработки на языке Python. Программисты обычно просматривают содержимое строк документирования при проведении экспериментов в интерактивной оболочке, поэтому в эти строки принято включать короткие примеры сеансов работы с интерактивной оболочкой. Например:

```
# splitter.py
def split(line, types=None, delimiter=None):
    """ Разбивает текстовую строку и при необходимости
        выполняет преобразование типов.
        Например:
        >>> split('GOOG 100 490.50')
        ['GOOG', '100', '490.50']
        >>> split('GOOG 100 490.50',[str, int, float])
        ['GOOG', 100, 490.5]
        >>>
        По умолчанию разбиение производится по пробельным символам,
        но имеется возможность указать другой символ-разделитель, в виде
именованного аргумента:

        >>> split('GOOG,100,490.50',delimiter=',')
        ['GOOG', '100', '490.50']
        >>>
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

Типичная проблема, связанная с документацией, состоит в том, чтобы обеспечить ее соответствие актуальной реализации функции. Например, программист может изменить функцию и забыть обновить документацию. Помощь в решении этой проблемы может оказать модуль doctest. Модуль doctest собирает строки документирования, просматривает их на наличие примеров интерактивных сеансов и выполняет эти примеры, как последовательность тестов. Чтобы воспользоваться модулем doctest, обычно требуется создать отдельный модуль, который будет выполнять тестирование. Например, если предположить, что реализация функции из предыдущего примера находится в файле splitter.py, можно было бы создать файл testsplitter.py со следующим содержимым:

```
# testsplitter.py
import splitter
import doctest

nfail, ntests = doctest.testmod(splitter)
```

В этом фрагменте вызов doctest.testmod(module) запускает процесс тестирования указанного модуля module и возвращает количество ошибок и общее количество выполненных тестов. Если все тесты прошли успешно, ничего не выводится. В противном случае на экране появится отчет об ошибках, в котором будут показаны различия между ожидаемыми и фактическими результатами. Если потребуется получить более подробный отчет о тестировании, можно вызвать функцию в виде testmod(module, verbose=True).

В противоположность созданию отдельного файла, выполнение тестов можно реализовать непосредственно в библиотечных модулях, для чего достаточно включить в конец файла модуля следующий программный код:

```
...
if __name__ == '__main__':
    # тестирование самого себя
    import doctest
    doctest.testmod()
```

После этого тестирование, основанное на содержимом строк документирования, будет выполняться, если запустить модуль как самостоятельную программу. В противном случае, при импортировании файла, тесты будут игнорироваться.

При тестировании функций модуль doctest ожидает вывод, **в точности совпадающий** с тем, что будет получен в интерактивной оболочке. В результате такой вид тестирования очень чувствителен к лишним или отсутствующим пробельным символам и к точности представления чисел. В качестве примера рассмотрим следующую функцию:

```
def third(x):
    """ Возвращает треть от x. Например:

        >>> third(6.9)
        2.3
        >>>
    """
    return x / 3
```

Если провести тестирование этой функции с помощью модуля doctest, будет получен следующий отчет об ошибках:

```
**********************************************************************
File "third.py", in __main__.third
Failed example:
    third(6.9)
Expected:
    2.3
Got:
    2.3000000000000003
**********************************************************************
```

Чтобы исправить эту проблему, необходимо либо привести строку документирования в точное соответствие с получаемыми результатами, либо привести в документации более удачный пример.

Модуль doctest чрезвычайно прост в использовании, поэтому не может быть никаких оправданий, чтобы не использовать его в своих программах. Однако имейте в виду, что модуль doctest – не тот инструмент, который можно использовать для полного тестирования программы. Применение этого модуля для полного тестирования может привести к чрезмерному раздуванию и усложнению строк документирования, что снижает полезность документации (пользователь наверняка будет недоволен,

если он обратится за справочной информацией, а в ответ ему будет представлен список из 50 примеров, охватывающих все хитрости использования функции). Для такого тестирования предпочтительнее использовать модуль unittest.

Наконец, модуль doctest имеет огромное количество параметров настройки различных аспектов, определяющих, как выполнять тестирование и как отображать результаты. Поскольку в большинстве типичных случаев использования модуля эти параметры не требуют изменения, они здесь не рассматриваются. Дополнительную информацию по этой теме можно найти по адресу: http://docs.python.org/library/doctest.html.

## Оператор assert

Инструкция assert позволяет добавлять в программу отладочный код. В общем случае инструкция assert имеет следующий вид:

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

где test – выражение, которое должно возвращать значение True или False. Если выражение test возвратит значение False, инструкция assert возбудит 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 (этот режим включается при запуске интерпретатора с ключом -O). В частности, будет ошибкой использовать инструкцию assert для проверки ввода пользователя. Обычно инструкции assert используются для проверки условий, которые всегда должны быть истинными; если такое условие нарушается, это можно рассматривать, как ошибку в программе, а не как ошибку пользователя.

Оператор assert можно использовать для написания юнит-тестов, для этого удобно будет оформить тест в виде функции. Например:

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

**Плюсы** организации тестирования через assert:

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

**Недостатки**:

- тесты нужно запускать вручную;
- такие тесты сложно отлаживать;

- для каждой проверки нужно написать свою функцию и своё сообщение об ошибке (“равно”, “не равно” и т.д.).

Современные библиотеки юнит-тестов (unittest, pytest, nose) предоставляют уже готовые функции и более широкие возможности для организации тестов.

## Модульное тестирование и модуль unittest

Для более 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 используются для проверки корректности поведения основных компонентов, составляющих крупные программы. По мере роста программы в размерах модульные тесты для различных компонентов могут объединяться в крупные структуры и средства тестирования. Это может существенно упростить задачу проверки корректности поведения, а также определения и исправления проблем по мере их появления.

**Характеристики** хорошего теста:

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

Использование модуля unittest иллюстрирует следующий фрагмент программного кода, взятый из предыдущего раздела:

```
# 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:

```
# testsplitter.py
import splitter
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 запустить тестирование, достаточно просто запустить интерпретатор Python, передав ему файл testsplitter.py. Например:

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

В своей работе модуль unittest опирается на объявление класса, производного от класса unittest.TestCase. Отдельные тесты определяются как методы, имена которых начинаются со слова ‘test’, например ‘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 различных условий.

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

- t.setUp() - Вызывается для выполнения настроек, перед вызовом любых методов тестирования.
- t.tearDown() - Вызывается для выполнения 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. msg – это строка сообщения, объясняющая причины ошибки (если задана).
- t.assertNotEqual(x, y [, msg]) / t.failIfEqual(x, y, [, msg]) - Сообщает об ошибке тестирования, если x и y равны. msg – это строка сообщения, объясняющая причины ошибки (если задана).
- 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. msg – это строка сообщения, объясняющая причины ошибки (если задана).
- 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. msg – это строка сообщения, объясняющая причины ошибки (если задана).
- t.assertRaises(exc, callable, ...) / t.failUnlessRaises(exc, callable, ...) - Сообщает об ошибке тестирования, если вызываемый объект callable не возбуждает исключение exc. Остальные аргументы методов передаются вызываемому объекту callable, как аргументы. 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. msg – это строка сообщения, объясняющая причины ошибки (если задана).
- t.fail([msg]) - Сообщает об ошибке тестирования. msg – это строка сообщения, объясняющая причины ошибки (если задана).
- t.failureException - В этом атрибуте сохраняется последнее исключение, перехваченное в тесте. Может использоваться, когда необходимо не только проверить, что исключение возбуждается, но что при этом оно сопровождается требуемым значением, – например, когда необходимо проверить сообщение, генерируемое исключением.

Следует отметить, что модуль unittest имеет огромное количество дополнительных параметров настройки, используемых для группировки тестов, создания наборов тестов и управления окружением, в котором выполняются тесты. Эти особенности не имеют прямого отношения к процессу создания тестов (классы обычно пишутся независимо от того, как в действительности выполняются тесты). В документации, по адресу http://docs.python.org/library/unittest.html, можно найти дополнительную информацию о том, как организовать тесты для крупных программ.

# py.test

"Everybody is using py.test anyway..."
Guido van Rossum

## Введение

Наряду с входящими в стандартную библиотеку Python средствами тестирования, существуют и альтернативные инструменты, например, pytest .

**Установка** выполняется стандартно:

```
pip install pytest
```

**Запуск тестов** выполняется командой в консоли:

```
python -m pytest test_module01.py
```

При этом возможны различные **способы запуска**:

- Найти тесты в текущей и во всех вложенных директориях:: python -m pytest;
- Найти и запустить тесты в указанном файле:: python -m pytest test_module01.py;
- Запустить 1 тест в файле по имени:: python -m pytest test_module01.py::test_fio.

**Тесты** могут быть оформлены разными способами:

- функции с префиксом test_;
- метод с префиксом test_ в классе с префиксом Test, или в классе, унаследованном от unittest.TestCase;
- доктест, при запуске с параметром –doctest-modules.

Сам механизм тестирования не предполагает использования специальных функций (как в unittest) - для тестирования используется оператор языка assert. Например, код ниже будет полноценным тестом для pytest:

```
def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()

def test_failed_upper():
    assert 'foo'.upper() == 'FOo'
```

При этом библиотека pytest предоставляет более широкий отчет о тестировании (в сравнении с doctest, unittest):

```
> python -m pytest pytest_begin.py
========================== test session starts ==========================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0

pytest_begin.py ..F

================================ FAILURES ================================
_____________________________ test_failed_upper _____________________________

    def test_failed_upper():
>       assert 'foo'.upper() == 'FOo'
E       AssertionError: assert 'FOO' == 'FOo'
E         - FOO
E         + FOo

pytest_begin.py:8: AssertionError
=================== 1 failed, 2 passed in 0.18 seconds ===================
```

## Фикстуры

**Фикстуры** (fixtures) - это функции или методы, которые запускаются для создания соответствующего окружения для теста. PyTest, как и unittest, имеет названия для фикстур всех уровней:

```
import pytest

def setup():
    print ("01. setup Установка окружения по умолчанию в модуле")

def teardown():
    print ("02. teardown Сброс окружения по умолчанию в модуле")

def setup_module(module):
    print ("03. setup Установка окружения на уровне модуля")

def teardown_module(module):
    print ("04. teardown Сброс окружения на уровне модуля")

def setup_function(function):
    print ("05. setup Установка окружения на уровне функции")

def teardown_function(function):
    print ("06. teardown Сброс окружения на уровне функции")

def test_numbers_3_4():
    print("07. >> test 3*4")
    assert 3*4 == 12

def test_strings_a_3():
    print("08. >> test a*3")
    assert 'a'*3 == 'aaa'
```

```
class TestUM:
    def setup(self):
        print ("09. setup Установка окружения по умолчанию в классе")

    def teardown(self):
        print ("10. teardown Сброс окружения по умолчанию в классе")

    @classmethod
    def setup_class(cls):
        print ("11. setup Установка окружения на уровне класса")

    @classmethod
    def teardown_class(cls):
        print ("12. teardown Сброс окружения на уровне класса")

    def setup_method(self, method):
        print ("13. setup Установка окружения на уровне метода")

    def teardown_method(self, method):
        print ("14. teardown Сброс окружения на уровне метода")

    def test_numbers_5_6(self):
        print("15. >> test 5*6")
        assert 5*6 == 30

    def test_strings_b_2(self):
        print("16. >> test b*2")
        assert 'b'*2 == 'bb'
```

Чтобы увидеть все сообщения, выдаваемые print(), необходимо запускать тест с флагом -s:

```
> python -m pytest -s pytest_base_fixtures.py

============================= test session starts ============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
collected 4 items

pytest_base_fixtures.py
03. setup Установка окружения на уровне модуля
05. setup Установка окружения на уровне функции
01. setup Установка окружения по умолчанию в модуле
07. >> test 3*4
.
02. teardown Сброс окружения по умолчанию в модуле
06. teardown Сброс окружения на уровне функции
05. setup Установка окружения на уровне функции
01. setup Установка окружения в модуле по умолчанию
08. >> test a*3
.
02. teardown Базовый сброс окружения для модуля
06. teardown Сброс окружения на уровне функции
11. setup Установка окружения на уровне класса
13. setup Установка окружения на уровне метода
09. setup Установка окружения по умолчанию в классе
15. >> test 5*6
.
10. teardown Сброс окружения по умолчанию в классе
14. teardown Сброс окружения на уровне метода
13. setup Установка окружения на уровне метода
09. setup Установка окружения по умолчанию в классе
16. >> test b*2
.
10. teardown Сброс окружения по умолчанию в классе
14. teardown Сброс окружения на уровне метода
12. teardown Сброс окружения на уровне класса
04. teardown Сброс окружения на уровне модуля

========================== 4 passed in 0.06 seconds ==========================
```

Данный пример достаточно полно показывает иерархию и повторяемость каждого уровня фикстур (например, setup_function вызывается перед каждым вызовом функции, а setup_module – только один раз для всего модуля). Также можно увидеть, что уровень фикстуры по умолчанию — функция/метод (фикстуры setup и teardown).

## Расширенные фикстуры

В случае, когда для части тестов нужно определенное окружение, а для других другое, на помощь приходят **расширенные фикстуры** PyTest.

Для создания расширенной фикстуры в PyTest необходимо:

1. импортировать модуль pytest;

2. использовать *декоратор* @pytest.fixture(), чтобы обозначить что данная функция является фикстурой;
3. задать уровень фикстуры (scope). Возможные значения: “function”, “cls”, “module”, “session”. Значение по умолчанию - “function”;
4. если необходим вызов teardown для этой фикстуры, то надо добавить в него финализатор (через метод addfinalizer объекта request передаваемого в фикстуру или же через использование конструкции yield);
5. добавить имя данной фикстуры в список параметров функции.

Рассмотрим пример создания фикстуры:

```
import pytest

@pytest.fixture()
def resource_setup(request):
    print("Подготовка ресурсов")

    def resource_teardown():                    # <- Функция сброса окружения
        print("Освобождение ресурсов")
    request.addfinalizer(resource_teardown)     # <- Добавление финализатора

def test_1_that_needs_resource(resource_setup):
    print("test_1 - требуются ресурсы")

def test_2_that_does_not():
    print("test_2 - не требуются ресурсы")

def test_3_that_does_again(resource_setup):
    print("test_3 - требуются ресурсы")
```

Запуск теста:

```
> python -m pytest -s pytest_user_fixtures.py

============================= test session starts ============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
collected 3 items

pytest_user_fixtures.py Подготовка ресурсов
test_1 - требуются ресурсы
.Освобождение ресурсов
test_2 - не требуются ресурсы
.Подготовка ресурсов
test_3 - требуются ресурсы
.Освобождение ресурсов

========================== 3 passed in 0.05 seconds ==========================
```

Расширенные фикстуры можно вызывать еще двумя способами:

1. декорирование теста декоратором @pytest.mark.usefixtures();
2. использование флага autouse для фикстуры. Следует использовать данную возможность с осторожностью, так как можно получить неожиданное поведение тестов.

Изменённый пример будет выглядеть следующим образом:

```
@pytest.fixture(scope="function", autouse=True)
def another_resource_setup_with_autouse(request):
    print("another_resource_setup_with_autouse")
    def resource_teardown():
        print("another_resource_teardown_with_autouse")
    request.addfinalizer(resource_teardown)

def test_1_that_needs_resource(resource_setup):
    print("test_1_that_needs_resource")

def test_2_that_does_not():
    print("test_2_that_does_not")

@pytest.mark.usefixtures("resource_setup")
def test_3_that_does_again():
    print("test_3_that_does_again")
```

Запуск:

```
> python -m pytest -s pytest_user_fixtures.py
========================== test session starts ==========================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
collected 3 items

pytest_user_fixtures.py another_resource_setup_with_autouse
Подготовка ресурсов
test_1 - требуются ресурсы
.Освобождение ресурсов
another_resource_teardown_with_autouse
another_resource_setup_with_autouse
test_2 - не требуются ресурсы
.another_resource_teardown_with_autouse
another_resource_setup_with_autouse
Подготовка ресурсов
test_3_that_does_again
.Освобождение ресурсов
another_resource_teardown_with_autouse

======================== 3 passed in 0.05 seconds ========================
```

## teardown расширенной фикстуры

Вызов teardown для определенной расширенной фикстуры можно реализовать двумя способами:

1. добавив в фикстуру финализатор (через метод addfinalizer объекта request передаваемого в фикстуру);
2. через использование конструкции yield (начиная с PyTest версии 2.4).

Продемонстрируем функционал teardown с использованием yield. Следует заметить, что для использования yield при декорировании функции, как фикстуры, необходимо использовать декоратор @pytest.yield_fixture(), а не @pytest.fixture():

```
import pytest

@pytest.yield_fixture()
def resource_setup_with_yield():
    print("resource_setup_with_yield")
    yield
    print("resource_teardown_with_yield")

def test_4_needs_resources(resource_setup_with_yield):
    print("test_4_needs_resources")
```

Вывод в консоль:

```
resource_setup_with_yield
test_4_needs_resources
.resource_teardown_with_yield
another_resource_teardown_with_autouse
```

## Возвращаемое фикстурой значение

Фикстура в PyTest может **возвращать** что-то в тест через return. Это может быть какое-то состояние или объект/ресурс (например, файл). Например, пусть фикстура возвращает “объект базы данных”:

```
import pytest

# Фикстура может возвращать что-то в тест (объект/ресурс)
@pytest.fixture(scope="module")
def resource_setup(request):
    print("\n >> 'Подключение' к БД")
    db = {"Red":1, "Blue":2, "Green":3}
    def resource_teardown():
        print("\n >> 'Отключение' от БД")
    request.addfinalizer(resource_teardown)
    return db

def test_db(resource_setup):
    for k in resource_setup.keys():
        print "color {0} has id {1}".format(k, resource_setup[k])

def test_red(resource_setup):
    assert resource_setup["Red"] == 1

def test_blue(resource_setup):
    assert resource_setup["Blue"] != 1
```

Результат запуска:

```
> python -m pytest -v -s pytest_user_fixtures.py
============================= test session starts ========================
 >> 'Подключение' к БД

pytest_user_fixture.py::test_db
Color Red has id 1
Color Blue has id 2
Color Green has id 3
PASSED

pytest_user_fixture.py::test_red
PASSED

pytest_user_fixture.py::test_blue
PASSED

 >> 'Отключение' от БД
```

## Уровень фикстуры (scope)

Уровень фикстуры может принимать следующие возможные значения “function”, “cls”, “module”, “session”. Значение по умолчанию - “function”:

- “function” – фикстура запускается для каждого теста;
- “cls” – фикстура запускается для каждого класса;
- “module” – фикстура запускается для каждого модуля;
- “session” – фикстура запускается для каждой сессии (фактически один раз).

Например, в предыдущем примере можно поменять scope на function и вызывать подключение к базе данных и отключение для каждого теста:

```
import pytest

@pytest.fixture(scope="function")
def resource_setup(request):
    print("\n >> 'Подключение' к БД")
    db = {"Red":1, "Blue":2, "Green":3}
    def resource_teardown():
        print("\n >> 'Отключение' от БД")
    request.addfinalizer(resource_teardown)
    return db
```

### Объект request

В примере создания расширенной фикстуры в нее передавался параметр request. Это было делалось для того, чтобы через его метод addfinalizer добавить финализатор. Объект request имеет также

достаточно много атрибутов и других методов (полный список в официальной документации), часть из них можно увидеть в примере:

```
@pytest.fixture(scope="function")
def resource_setup(request):
    print()
    print ('01', request.fixturename)
    print ('02', request.scope)
    print ('03', request.function.__name__)
    print ('04', request.cls)
    print ('05', request.module.__name__)
    print ('06', request.fspath)

def test_1(resource_setup):
    assert True

class TestClass():
    def test_2(self, resource_setup):
        assert True
```

Результат:

```
> python -m pytest -v -s pytest_request.py
============================= test session starts ============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
collected 2 items

05_pytest_request.py::test_1
01 resource_setup
02 function
03 test_1
04 None
05 05_pytest_request
06 \testing\05_pytest_request.py
PASSED
05_pytest_request.py::TestClass::test_2
01 resource_setup
02 function
03 test_2
04 <class '05_pytest_request.TestClass'>
05 05_pytest_request
06 \testing\05_pytest_request.py
PASSED

======================== 2 passed in 0.05 seconds ========================
```

# Параметризация тестов

**Параметризация** — это способ запустить один и тот же тест с разным набором входных параметров. Например, у нас есть функция, которая добавляет знак вопроса к строке, если она длиннее 5 символов, восклицательный знак — менее 5 символов и точку, если в строке ровно 5 символов. Соответственно вместо того, чтобы писать три теста, мы можем написать один, но вызываемый с разными параметрами.

Задать параметры для теста можно двумя способами:

1. Через значение параметра params фикстуры, в который нужно передать массив значений. То есть фикстура будет “оберткой”, передающей параметры. В сам тест они передаются через атрибут param объекта request.
2. Через декоратор (метку) @pytest.mark.parametrize, в который передается список названий переменных и массив их значений.

Рассмотрим пример первого варианта параметризации:

```
import pytest

def strange_string_func(s):
    ''' 'Подопытная' функция для тестов
    '''
    if len(s) > 5:
        return s + '?'
    elif len(s) < 5:
        return s + '!'
    else:
        return s + '.'

@pytest.fixture(scope="function", params=[
    ("abcdefg", "abcdefg?"),
    ("abc", "abc!"),
    ("abcde", "abcde.")
])
def param_test(request):
    return request.param

def test_strange_string_func(param_test):
    in_str, expected_output = param_test
    result = strange_string_func(in_str)
    print("\ninput: {0}, output: {1}, expected: {2}".format(in_str, result,
expected_output))
    assert result == expected_output
```

Результат:

```
> python -m pytest -v -s pytest_parametrize.py
========================== test session starts ==========================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
collected 3 items

pytest_parametrize.py::test_strange_string_func[param_test0]
input: abcdefg, output: abcdefg?, expected: abcdefg?
PASSED
pytest_parametrize.py::test_strange_string_func[param_test1]
input: abc, output: abc!, expected: abc!
PASSED
pytest_parametrize.py::test_strange_string_func[param_test2]
input: abcde, output: abcde., expected: abcde.
PASSED

======================== 3 passed in 0.05 seconds ========================
```

Второй способ параметризации имеет одно преимущество: если указать несколько меток с разными параметрами, то в итоге тест будет запущен со всеми возможными наборами параметров (то есть декартово произведение параметров):

```
@pytest.mark.parametrize("x", [1, 2])
@pytest.mark.parametrize("y", [10, 11])
def test_cross_params(x, y):
    print("x: {0}, y: {1}".format(x, y))
    assert True
```

Результат:

```
pytest_parametrize.py::test_cross_params[10-1] x: 1, y: 10
PASSED
pytest_parametrize.py::test_cross_params[10-2] x: 2, y: 10
PASSED
pytest_parametrize.py::test_cross_params[11-1] x: 1, y: 11
PASSED
pytest_parametrize.py::test_cross_params[11-2] x: 2, y: 11
PASSED
```

## Метки

PyTest поддерживает класс декораторов @pytest.mark называемый **метками** (marks). Базовый список включает в себя следующие метки:

1. @pytest.mark.parametrize — для параметризации тестов (было рассмотрено выше);
2. @pytest.mark.xfail – помечает, что тест должен не проходить и PyTest будет воспринимать это, как ожидаемое поведение (полезно, как временная метка для тестов на разрабатываемые

функции). Также эта метка может принимать условие, при котором тест будет помечаться данной меткой;

3. @pytest.mark.skipif – позволяет задать условие, при выполнении которого тест будет пропущен;
4. @pytest.mark.usefixtures – позволяет перечислить все фикстуры, вызываемые для теста.

Список меток шире и его можно получить выполнив команду pytest –markers.

Пример использования меток:

```
import pytest
import sys

@pytest.mark.xfail()
def test_failed():
    assert False

@pytest.mark.xfail(sys.platform != "win64", reason="requires windows 64bit")
def test_failed_for_not_win32_systems():
    assert False

@pytest.mark.skipif(sys.platform != "win64", reason="requires windows 64bit")
def test_skipped_for_not_win64_systems():
    assert False
```

Результат:

```
> python -m pytest -v -s pytest_marks.py
============================= test session starts ============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0 cov-2.4.0
collected 3 items

pytest_marks.py::test_failed xfail
pytest_marks.py::test_failed_for_not_win32_systems xfail
pytest_marks.py::test_skipped_for_not_win64_systems SKIPPED

================== 1 skipped, 2 failed in 0.24 seconds ===================
```

# Резюме

**Положительные особенности** pytest:

- никакого специфического API, тесты - это обычные функции;
- проверки посредством assert. Это обеспечивает потенциальную возможность запустить тест даже без установленного pytest, например, на продакшен-сервере. Тесты можно оформлять как классами (в стиле unittest), так и просто функциями вида test_*;
- удобный вывод результата тестирования;
- удобно параметризовать тесты;
- переиспользуемые фикстуры.

При использовании py.test можно отметить пару **недостатков**:

- py.test не входит в стандартную библиотеку Python;

- много скрытой “магии”.

# 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 библиотекой PyTest.

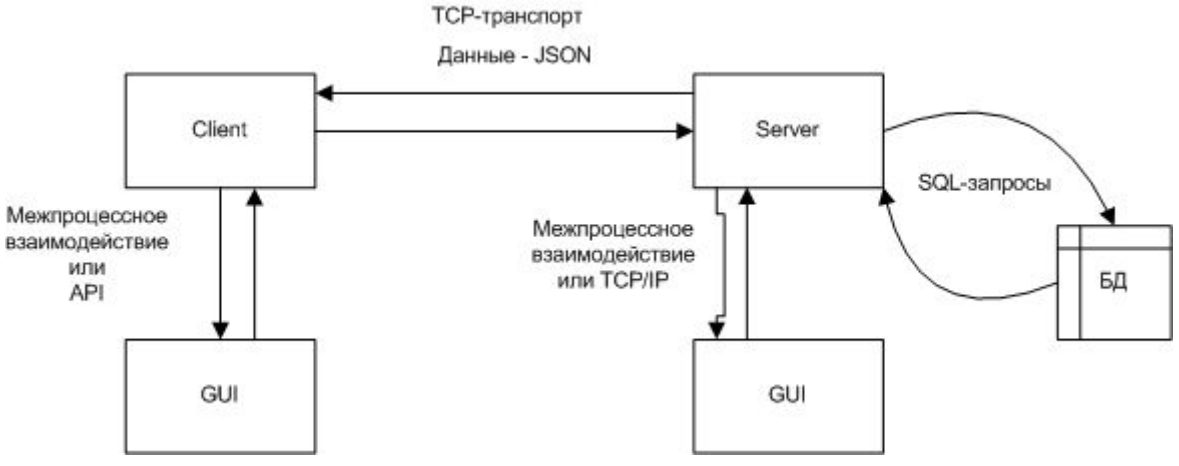
Современные гибкие методики разработки рассматривают внесение изменений в код не как некую гипотетическую и маловероятную возможность, а как совершенно обыденную часть работы. Это стимулирует разработчиков к созданию кода, который легче поддается изменениям. Для упрощения внесения изменений в код рекомендуется делать архитектуру приложения **слабосвязанной**. А для уменьшения скрытых последствий внесения изменений предназначены **автоматизированные тесты**. Они позволяют обнаружить отклонение программы от ожидаемого поведения на самой ранней стадии внесения изменений.

Важно отметить, что **тесты** – такая же часть кода, как и его архитектура. Тесты – это та часть кода, которая делает его приспособленным к внесению изменений с минимальными последствиями. Чем лучше устроена архитектура – тем легче создавать тесты. Чем лучше организованы тесты, тем меньше скрытых последствий будет после внесения изменений. И тем надежнее и качественнее будет полученный код.

# Протоколы обмена в курсовом проекте

Как уже говорилось в начале занятия, домашнее задание будет носить характер одного цельного проекта. Предлагаем Вам реализовать проект под названием **“Мессенджер”** - упрощенный аналог существующих мессенджеров. Необходимо реализовать как серверную, так и клиентскую часть.

Обобщенная схема взаимодействия элементов системы представлена на рисунке:




## Протокол обмена

За основу протокола обмена между клиентом и сервером взят проект JIM (JSON IM Protocol)

Протокол JIM базируется на передаче JSON-объектов через TCP-сокеты.

Все сетевые взаимодействия осуществляются в байтах.

### Спецификация объектов

JSON-данные пересылаемые между клиентом и сервером обязательно должны содержать поля “action” и “time”.

Поле “action” задаёт тип сообщения между клиентом и сервером. Поле “time” - временная метка отправки JSON-объекта, UNIX-время (число секунд от 1 января 1970 года).

Например, для аутентификации должен быть сформирован JSON-объект:

```
{
 "action": "authenticate",
 "time": <unix timestamp>,
        "user": {
                "account_name": "C0deMaver1ck",
                "password":     "CorrectHorseBatterStaple"
        }
}
```

Ответы сервера должны содержать поле “response”, а также могут содержать поле “alert”/“error” с текстом ошибки.

Все объекты имеют ограничения длины (количество символов):

- **“action”**-поле: 15 символов (сейчас самое длинное название - “authenticate” (11 символов); вряд ли должно понадобиться что-то длиннее);
- **“response”**-поле с кодом ответа сервера: 3 цифры;
- **имя пользователя** / **название чата** (name): 25 символов;
- **сообщение**: максимум 500 символов ( ” ” ).

Итоговое ограничение для JSON-объекта - 640 символов (что оставляет возможность добавить дополнительные поля или изменить имеющиеся).

Исходные ограничения на длину сообщений позволяют упростить реализации клиента и сервера.

### Подключение, отключение, авторизация

JIM-протокол не подразумевает обязательной авторизации при подключении к серверу. Это позволяет реализовать функционал для гостевых пользователей.

В случае, если какое-то действие требует авторизации, сервер должен ответить соответствующим кодом ошибки 401.

После подключения при необходимости авторизации клиент должен отправить сообщение авторизации с логином/паролем, например:

```
{
        "action": "authenticate",
        "time": <unix timestamp>,
        "user": {
                "account_name":  "C0deMaver1ck",
                "password":      "CorrectHorseBatteryStaple"
        }
}
```

В ответ сервер может прислать один из кодов:

```
{
    "response": 200,
    "alert":"Необязательное сообщение/уведомление"
}

{
    "response": 402,
    "error": "This could be "wrong password" or "no account with that name""
}

{
    "response": 409,
    "error": "Someone is already connected with the given user name"
}
```

Отключение от сервера должно сопровождаться сообщением “quit”:

```
{
    "action": "quit"
}
```

### Присутствие

Каждый пользователь при подключении к серверу отсылает сервисное сообщение о присутствии presence с необязательным полем type:

```
{
        "action": "presence",
        "time": <unix timestamp>,
        "type": "status",
        "user": {
                "account_name":  "C0deMaver1ck",
                "status":      "Yep, I am here!"
        }
}
```

Для проверки доступности пользователя online сервер выполняет probe-запрос:

```
{
        "action": "probe",
        "time": <unix timestamp>,
}
```

Probe-запрос может отправлять только сервер для проверки доступности клиентов контакт-листа. На probe-запрос клиент должен ответить простым presence-сообщением.

### Коды ответов сервера

JIM-протокол использует коды ошибок HTTP (они уже многим знакомы).

Поддерживаемые коды ошибок:

- **1xx** - информационные сообщения:
    - **100** - базовое уведомление;
    - **101** - важное уведомление.
- **2xx** - успешное завершение:
    - **200** - OK;
    - **201** (created) - объект создан;
    - **202** (accepted)- подтверждение.
- **4xx** - ошибка на стороне клиента:
    - **400** - неправильный запрос/JSON-объект;
    - **401** - не авторизован;
    - **402** - неправильный логин/пароль;
    - **403** (forbidden) - пользователь заблокирован;
    - **404** (not found) - пользователь/чат отсутствует на сервере;
    - **409** (conflict) - уже имеется подключение с указанным логином;
    - **410** (gone) - адресат существует, но недоступен (offline).
- **5xx** - ошибка на стороне сервера:
    - **500** - ошибка сервера.

Коды ошибок могут быть дополнены новыми кодами.

Сообщения-ответы имеют следующий формат (в зависимости от кода ответа):

```
{
    "response": 1xx / 2xx,
    "time": <unix timestamp>,
    "alert": "message (optional for 2xx codes)"
}
```

или

```
{
    "response": 4xx / 5xx,
    "time": <unix timestamp>,
    "error": "error message (optional)"
}
```

### Сообщение “Пользователь-Пользователь”

Простое сообщение имеет формат:

```
{
    "action": "msg",
    "time": <unix timestamp>,
    "to": "account_name",
    "from": "account_name",
    "encoding": "ascii",
    "message": "message"
}
```

Когда сервер видит действие “msg”, ему не нужно читать/парсить всё сообщение. Следует только проверить адресата и передать ему сообщение.

Если поле “to” (адресат) имеет префикс #, то это сообщение для группы. Обрабатывается как приватное сообщение, но по шаблону “Пользователь-Чат”.

В ответ на такое событие клиенту возвращается код ошибки.

Поле “encoding” указывает кодировку сообщения. Если не указывается, то считается “ascii”.

### Сообщение “Пользователь-Чат”

Обрабатывается также, как и “Пользователь-Пользователь”, но с дополнением:

- Имя чата имеет префикс ‘#’ (то есть сервер должен проверять поле “to” для всех сообщений и переправлять сообщение всем online-пользователям данного чата).

Сообщение:

```
{
    "action": "msg",
```

```
    "time": <unix timestamp>,
    "to": "#room_name",
    "from": "account_name",
    "message": "Hello World"
}
```

Присоединиться к чату:

```
{
    "action": "join",
    "time": <unix timestamp>,
    "room": "#room_name"
}
```

Покинуть чат:

```
{
    "action": "leave",
    "time": <unix timestamp>,
    "room": "#room_name"
}
```

**Методы протокола (actions)**

- “action”: “presence” - присутствие. Сервисное сообщение для извещения сервера о присутствии клиента online;
- “action”: “probe” - проверка присутствия. Сервисное сообщение от сервера для проверки присутствии клиента online;
- “action”: “msg” - простое сообщение пользователю или в чат;
- “action”: “quit” - отключение от сервера;
- “action”: “authenticate” - авторизация на сервере;
- “action”: “join” - присоединиться к чату;
- “action”: “leave” - покинуть чат.

Протокол может быть расширен новыми методами.

# Домашнее задание

1. Функционал.

Первая часть домашнего задания будет заключаться в реализации простого клиент-серверного взаимодействия по протоколу JIM (JSON instant messaging):

- клиент отправляет запрос серверу;
- сервер отвечает соответствующим кодом результата.

Клиент и сервер должны быть реализованы в виде отдельных скриптов, содержащих соответствующие функции.

**Функции клиента**:

- сформировать presence-сообщение;
- отправить сообщение серверу;
- получить ответ сервера;
- разобрать сообщение сервера;
- параметры командной строки скрипта client.py <addr> [<port>]:
    - addr - ip-адрес сервера;
    - port - tcp-порт на сервере, по умолчанию 7777.

**Функции сервера**:

- принимает сообщение клиента;
- формирует ответ клиенту;
- отправляет ответ клиенту;
- имеет параметры командной строки:
    - -p <port> - TCP-порт для работы (по умолчанию использует порт 7777);
    - -a <addr> - IP-адрес для прослушивания (по умолчанию слушает все доступные адреса).

2. Тесты.

Для всех функций необходимо написать тесты с использованием doctest (небольшие тесты в документации функций), unittest или py.test (в дальнейшем упор будет делаться на библиотеку py.test). Тесты должны быть оформлены в отдельных скриптах с префиксом test_ в имени файла (например, test_client.py).

3. Дополнительно.

В качестве практики написания тестов напишите тесты для домашних работ курса Python-1.

# Дополнительные материалы

1. Памятка по написанию тестов при помощи PyTest
2. Как в Яндексе используют PyTest и другие фреймворки для функционального тестирования
3. Хабрахабр. PyTest
4. Юнит тесты. Первый шаг к качеству
5. Тестирование. Начало
6. Блог Евгения Курочкина. PyTest
7. [Видео] Продвинутое использование py test, Андрей Светлов

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:


1. David Beazley, Brian K. Jones “Python Cookbook”, Third Edition.
2. Дэвид Бизли. “Python. Подробный справочник”
3. Лучано Ромальо. “Python. К вершинам мастерства”.

4. Кент Бек. “Экстремальное программирование: разработка через тестирование”
5. habrparser.blogspot.com/2015/10/pytest.html
6. grabduck.com/s/l0f0h98S
7. habrahabr.ru/post/269759/
8. pythondigest.ru/view/7436/
9. dml.compkaluga.ru/forum/index.php?showtopic=107344
10. fliphtml5.com/oizl/upcx/basic
11. toster.ru/q/390692