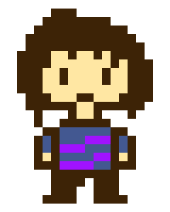
Руководство пользователя

messenger\_app 1.0 RC1



Версия приложения: 1.0 RC1

Версия интерпретатора: Python 3.7

Среда разработки: PyCharm 2019 1.3

Оглавление

**Общие положения3**

**Запуск приложения4**

**Работа с сетью5**

**Работа с сообщениями7**

**Логирование8**

Общие положения:

Данное приложение написано на языке программирования Python и является мессенджером с функцией отправки и приёма сообщений в консольном терминале. Приложение использует встроенные и внешние модули для корректной работы

Протоколом передачи данных в проекте является JIM (JSON Instant Messaging)— протокол для обмена данными между клиентом и сервером, который работает через TCP-сокеты (SOCK\_STREAM) и передачу JSON-объектов. Все сетевые операции проходят в байтовом представлении.

Знаком «решетка» (#) в коде обозначены комментарии, а также проверочные (запасные) функции и конструкции для служебного использования или code review

В описании мы указываем название файла (модуля Python), к которому относятся описанные инструкции, переменные, методы, функции и их аргументы. Подсветкой обозначены основные элементы, которые играют важную роль.

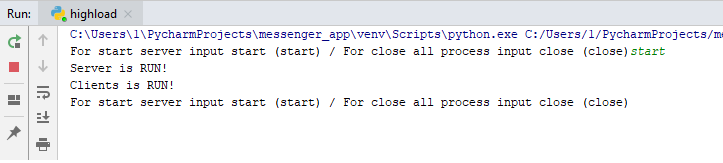
Корректность работы приложения проверялось на ОС Windows- не гарантируется стабильная работа на Linux или macOS

Запуск приложения

highload.py >> Дочерние клиентские процессы

Для того, чтобы запустить клиента, необходимо запустить на выполнение файл **highload.py**. В нем мы запускаем бесконечный цикл while true, где с помощью класса Popen из модуля subprocess создаем серверный и клиентские процессы. Для клиентского процесса не забываем указывать параметры, которые проверяются в файле **client.py**. Флаг CREATE\_NEW\_CONSOLE указывает интерпретатору, что запуск будет выполняться в терминале (консоли). Все процессы хранятся в списке process. Количество запущенных клиентов мы определяем самостоятельно – они указываются в переменной volume\_clients.

Чтобы осуществить запуск клиентских процессов, необходимо при запуске скрипта ввести слово start в консоль. Для закрытия открытых подключений используется all\_process.kill(), при этом мы очищаем список процессов с помощью конструкции process.clear()



Работа с сетью:

сlient.py>> Создание сокета, проверка аргументов

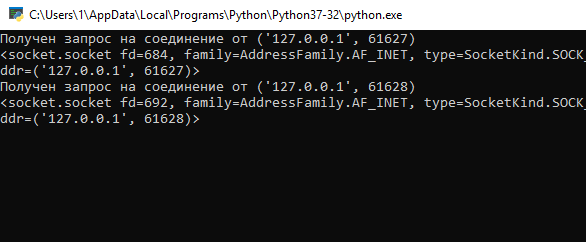
Для инициализации клиентского подключения к серверу мы создаем клиентский сокет и записываем его в переменную clientside, с помощью конструкции socket (AF\_INET, SOCK\_STREAM), где AF\_INET - это коммуникационный домен, который задает сетевую направленность нашему сокету. Тип сокета — SOCK\_STREAM — он определяет сокет как потоковый. С помощью обработки исключений try/except мы можем проверить список аргументов командной строки (sys.argv) – порт, IP-address, режим работы, получателя.

сlient.py >> Установка соединения

После проверки параметров, устанавливаем соединение с помощью метода connect, в котором мы передаем аргументы для подключения (адрес и порт). Далее клиент отправляет серверу специальное служебное сообщение о своем присутствии в сети (presence– см. **Работа с сообщениями**). Ответ сервера на presence-сообщение мы записываем в специальную переменную - response\_from\_server. В неё мы помещаем словарь (dict), в котором ключ response должен иметь значение 200 (ОК). Проверка «рукопожатия» осуществляется с помощью оператора ==

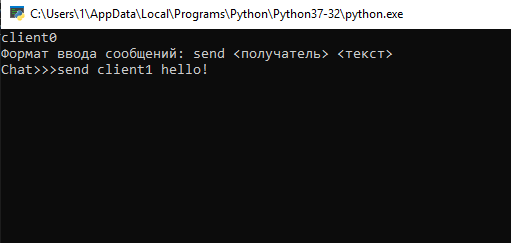
сlient.py>> Создание дочерних процессов

При удачном соединении создаем объект класса multiprocessing.Process . Внутри класса мы передаем два параметра: target – «цель» нашего процесса, т.е. функцию, которую будем запускать внутри, args – позиционные аргументы этой функции. В target мы записываем функцию чтения и вывода сообщений read\_messages (см. **Работа с сообщениями**). Запускаем процесс командой start(). Это позволяет нам выполнить запуск функции в дочернем процессе для каждого подключившегося клиента. Если мы указали в переменной volume\_clients два клиента, то мы увидим два подключения к серверу.



сlient.py>> Работа мессенджера

После того, как мы запустили программу из файла **highload.py** (см. **Запуск приложения**), мы создаем бесконечный цикл while true, в котором мы создаем специальную переменную для создания сообщений: message\_start, где задаем начальное значение (chat>>>). С помощью оператора if мы сигнализируем интерпретатору, что если сообщение начинается со слова send (startswitch), то необходимо проверить параметры ввода: to и text (параметры начинаются с единицы, т.к. «нулевой» параметр сообщения – это send). Если эта проверка не вызывает исключений (try/except), то выполняется блок else. В нём мы формируем сообщение (message) через функцию create\_message c тремя аргументами (to, text, account\_name) и отсылаем его с помощью функции send\_message (см. **Работа с сообщениями**)



Работа с сообщениями:

Для работы с сообщениями реализован специальный модуль (**/functions/message.py**). С его помощью клиент получает весь функционал по приёму, декодированию, чтению, формированию и отправке сообщений. Обмен сообщениями происходит в JSON-формате.

message.py>> Чтение сообщений

Чтение сообщений происходит по вызову функции read\_messages. В качестве аргументов передаются клиентский сокет и имя пользователя. Для начала мы получаем сообщение с помощью функции receive\_message . В терминал мы выводим значение ключа message (message['message']), т.е. сам текст сообщения

message.py>> Создание сообщений

За создание отвечает функция create\_message. Она возвращает словарь с параметрами, которые необходимы для формирования сообщения в формате JSON: ACTION, TIME, TO, FROM, MESSAGE (все ключи указаны в файле **config.py**)

message.py>> Приём сообщения от сервера

Здесь отрабатывает функция receive\_message. Здесь происходит приём данных от сервера c с помощью метода sock.recv, далее проверка типа данных – нам нужны байты. Для этого используем функцию isinstance. Декодируем ответ decode и переводим JSON-формат в юникод json.loads. Опять используем isinstance, чтобы проверить, что на выходе мы имеем тип данных словарь dict. Записываем ответ в переменную message2receive и возвращаем его.

message.py>> Отправка сообщения серверу

Реализована в функции send\_message. Передаем два аргумента: клиентский сокет и словарь, который необходимо конвертировать в объект формата JSON (json.dumps), а затем и в байты (encode) для отправки по сети. Проверку на тип данных, как в случае и с приёмом сообщения, выполняем с помощью инструкции if-else и функции isinstance. Результат записываем в переменную message2send и отправляем с помощью метода sock.send

message.py>> Формирование presence-сообщения

Чтобы сервер распознал наше «приветствие», необходимо сформировать специальное сообщение в формате JSON. Первую пару «ключ-значение» указываем как ACTION: "presence". Так же проверяем, что аргумент account name будет строкой isinstance(account\_name, str) с количеством символов не более 25: len(account\_name) > 25. Для обработки исключений мы используем оператор raise. Он позволяет вручную вызвать исключительную ситуацию raise Exception('Имя должно быть не более 25 символов')

Логирование:

Журналирование будет происходить в файле, отвечающем за функционал мессенджера (**/functions/message.py**). Для логирования клиентских функций используется декоратор @logclient. Для этого необходимо импортировать наш клиентский модуль декорирования (log.decorators) и создать экземпляр класса Log\_Client, в котором мы формируем сообщение для лога, используем метод \_\_call\_\_ для возможности вызова экземпляра как декоратора. Функция-обёртка описана в модуле (**/functions/decorators.py**).

Для записи логов в файл, мы создаем объект класса Logger c помощью функции getLogger(). Настройка логера осуществляется в файле (**/log/client\_log\_config.py)**. Обработчик FileHandler указывает интерпретатору, что мы записываем файл логов в каталог **/log**. Файл с расширением .log можно открыть в стандартном блокноте Windows

