Основы Python

Урок 4. Работа с модулями и пакетами



# На этом уроке

1. Поговорим о виртуальном окружении.
2. Рассмотрим механизм работы с модулями и пакетами в Python.
3. Рассмотрим примеры использования полезных модулей: requests, time, datetime.

## Оглавление

[Батарейки для Python: pypi.org](#_fjuessrof5va)

[Менеджер пакетов pip](#_mwbdkfv9ux7)

[Виртуальное окружение — зачем и как?](#_nrlc4r6nzg1d)

[Создаем виртуальное окружение — venv](#_1atdmy422c8f)

[Работаем с виртуальным окружением](#_p8hmv4uoq1ls)

[Модули и пакеты в Python](#_kd8ryy3t53g5)

[Модули](#_bjzbb3qgrixr)

[Пакеты](#_ygpwlwo08egp)

[Варианты импорта, их особенности](#_w50pwd567y27)

[Библиотека requests](#_hsjynnjtriwd)

[Модуль sys: запуск скрипта с параметрами](#_jzbnuhb7rnle)

[Модуль time: профилируем время выполнения участков кода](#_u012sxvu6ck7)

[Модуль datetime: работа с датой и временем](#_3697kxppqgs9)

[Практическое задание](#_fiukc6deepsd)

[Дополнительные материалы](#_2xcytpi)

[Используемая литература](#_1ci93xb)

# 

# Батарейки для Python: pypi.org

Одно из существенных преимуществ языка Python — огромное количество готовых библиотек, позволяющих мыслить более высокоуровневыми категориями при разработке, не отвлекаясь на мелочи. Многие начинающие программисты [изобретают колесо](https://en.wikipedia.org/wiki/Reinventing_the_wheel) вместо поиска в официальном репозитории <https://pypi.org/>. С точки зрения предстоящих собеседований «изобретение колеса» — полезный навык, но в коммерческой разработке этого не нужно делать. Пример: вам необходимо перевести число в текст. Можно написать свою программу, решающую эту задачу, потратить время на отладку кода. А можно, например, подключить к проекту библиотеку [num2words](https://pypi.org/project/num2words/). Да, придётся потратить время на чтение документации, на понимание логики работы с библиотекой, но вы получите возможность перевода чисел на разные языки, в разные виды валют.

При поиске библиотек нужно обращать внимание на дату последнего релиза. В случае с num2words в конце 2020 года дата видим “Released: May 12, 2019” — 1,5 года обновления не было. Это не очень хорошо. Есть вероятность, что проект заброшен. Это значит, вы получите нехорошую зависимость в своём проекте: например, при выходе новой версии python всё может сломаться. Поэтому вам как разработчику в будущем придётся принимать решение: добавить библиотеку в проект и получить расширение функционала, но и дополнительную зависимость, или «изобрести колесо» и самостоятельно поддерживать и расширять его функциональность.

## 

# Менеджер пакетов pip

Поговорим теперь об инструменте, при помощи которого устанавливаются библиотеки: [pip](https://pypi.org/project/pip/) — менеджер пакетов Python.

Если вы работаете в Windows, он автоматически установился вместе с Python.

Для nix-систем нужно установить его вручную (если вы работаете на Mac OS, рекомендуем установить виртуальную машину с nix-системой, например Ubuntu Server 20.04 LTS, чтобы не отвлекаться на погружение в особенности Mac OS):

| sudo apt install python3-pip |
| --- |

**Если вы увидели ошибку** типа Unable to locate package python3-pip, нужно поправить конфигурацию пакетного менеджера ОС:

| sudo nano /etc/apt/sources.list |
| --- |

Если в пути к репозиторию есть префикс ru., нужно его убрать (обычно такое бывает, когда выбирают русский язык при установке ОС). То есть если путь прописан http://ru.archive.ubuntu.com/ubuntu, меняем на http://archive.ubuntu.com/ubuntu. Далее добавляем universe в первую и вторую строки конфигурации. Для версии Ubuntu 18 должно стать:

| deb http://archive.ubuntu.com/ubuntu bionic main universe deb http://archive.ubuntu.com/ubuntu bionic-security main universe ... |
| --- |

Нажимаем Ctrl+O, подтверждаем имя файла, нажимаем Ctrl+X — выходим из редактора [nano](https://losst.ru/tekstovyj-redaktor-nano-v-linux-dlya-novichkov). Выполняем

| sudo apt update |
| --- |

И снова пробуем установить pip3. Всё должно получиться.

Почему суффикс 3? Потому что мы работаем с python3, а в nix-системах был по умолчанию установлен python 2. В Windows будем писать pip.

Посмотрим, какие пакеты уже установлены в вашей системе:

| pip freeze |
| --- |

Вы должны увидеть список пакетов. Если ничего не увидели — значит, ещё ничего не устанавливали. Установим пакет (библиотеку) [requests](https://requests.readthedocs.io/en/master/):

| pip install requests |
| --- |

Результат — сообщение вида Successfully installed ..., содержащее список установленных вместе с библиотекой зависимостей. Одна зависимость может тянуть за собой другие — это одна из причин **виртуализации окружения** проекта. Если снова выполним команду pip freeze, увидим в списке:

| ... requests==2.25.0 ... |
| --- |

Можно узнать, установлен ли пакет, при помощи команды pip show:

| pip show requests Name: requests Version: 2.25.0 Summary: Python HTTP for Humans. Home-page: https://requests.readthedocs.io Author: Kenneth Reitz Author-email: me@kennethreitz.org License: Apache 2.0 Location: /home/user/.local/lib/python3.6/site-packages Requires: chardet, certifi, urllib3, idna |
| --- |

Если пакет больше не нужен — удаляем его при помощи команды

| pip uninstall <имя пакета> |
| --- |

Когда есть необходимость установить **конкретную** версию пакета:

| pip install django==2.2 |
| --- |

Установили Django именно версии 2.2 в систему. Это бывает особенно важно при работе с фреймворками. Иногда при выходе новых версий ломается старый код.

**\*Внимание:** при установке пакетов могут быть сбои, особенно в системе Windows. Если есть свободное время, попробуйте поставить библиотеку для [распознавания лиц](https://pypi.org/project/face-recognition/) pip install face\_recognition. Может за день всё получиться, а может и недели не хватить. Зато получите много полезного опыта.

# Виртуальное окружение — зачем и как?

Работали ли вы в виртуальной машине, например, [VirtualBox](https://www.virtualbox.org/)? Слышали ли вы о [Docker](https://www.docker.com/get-started)? Может, даже работали в нём? Зачем вообще нужна виртуализация? Начинающему разработчику она не нужна. Но постепенно сложность ваших проектов будет возрастать. Количество используемых библиотек и их версии будут всё разнообразнее. В какой-то момент появится задача деплоя (развёртывания) проекта на сервер. Вы скопируете на него файлы, а дальше нужно будет установить на сервере все необходимые для его запуска библиотеки — мы называем это **зависимостями** (dependencies). Можно сделать это голыми руками, но лучше автоматизировать.

## Создаем виртуальное окружение — venv

Итак, проект растёт. Пакеты добавляются. Попробуем выполнить команду pip freeze — видим список всех библиотек, установленных в системе. Как понять, какие именно нужны для запуска текущего проекта? Никак, только если вы на бумажке записывали их по мере установки в систему. Именно поэтому best practice — создавать для каждого проекта своё **виртуальное окружение**. По сути это отдельный интерпретатор python с его библиотеками. Существуют разные способы работы с виртуальным окружением. Мы рассмотрим самый простой — использование модуля [venv](https://docs.python.org/3.6/library/venv.html#module-venv).

В nix-системах, возможно, понадобится его установить:

| apt install python3.6-venv |
| --- |

**Внимание:** ставим версию venv, соответствующую вашей версии python. Если в системе python 3.7, то ставим python3.7-venv.

Создаём окружение:

| python -m venv virt |
| --- |

Если всё хорошо, должна появиться папка virt. Можно было задать любое имя вместо virt. PyCharm по умолчанию создаёт окружение в директории venv.

Зафиксируйте размер папки. Пока в вашем окружении нет пакетов, он мал.

## Работаем с виртуальным окружением

Для начала виртуальное окружение необходимо активировать.

В Windows:

| "virt/Scripts/activate.bat" |
| --- |

В nix-системах:

| source virt/bin/activate |
| --- |

В результате ваше приглашение в командной строке (терминале) должно начинаться с префикса “(virt)”.

Разумеется, делать это нужно из папки, где создавали виртуальное окружение. Если что-то не получается, убедитесь, что вы находитесь там, где надо (команда dir), что название папки с виртуальным окружением правильное, что не забыли кавычки (в Windows).

Итак, мы активировали виртуальное окружение. Выполним команду pip freeze. Что увидели? В Windows — ничего, в nix-системах либо ничего, либо “pkg-resources==0.0.0” (это [официальный баг](https://stackoverflow.com/questions/39577984/what-is-pkg-resources-0-0-0-in-output-of-pip-freeze-command) некоторых версий Ubuntu). Это значит, что в новом виртуальном окружении пусто, нет установленных библиотек.

**Внимание:** при работе в виртуальном окружении в nix-системах не нужно писать префикс 3: вместо python3 и pip3 пишем python и pip.

Установим пару библиотек — requests и [pillow](https://pillow.readthedocs.io/en/stable/):

| pip install requests pip install pillow |
| --- |

Если теперь выполнить команду pip freeze, должны увидеть следующее:

| certifi==2020.11.8 chardet==3.0.4 idna==2.10 Pillow==8.0.1 pkg-resources==0.0.0 requests==2.25.0 urllib3==1.26.2 |
| --- |

В Windows не будет бага “pkg-resources==0.0.0”.

Если теперь вы будете работать с проектом через виртуальное окружение, то всегда сможете таким способом посмотреть зависимости, связанные именно с этим проектом. Уже намного лучше, чем раньше, когда мы работали в глобальном (не виртуальном) окружении. Остаётся вопрос: как все эти зависимости установить на удаленном сервере? Один из вариантов — создать специальный файл [requirements.txt](https://ru.stackoverflow.com/questions/396133/requirements-txt-%D0%BA%D0%B0%D0%BA-%D0%BF%D0%BE%D0%BB%D1%83%D1%87%D0%B8%D1%82%D1%8C):

| pip freeze > requirements.txt |
| --- |

**Примечание:** если вы работаете в nix-системе, не забудьте удалить из этого файла строку с багом: “pkg-resources==0.0.0”.

Теперь на удалённом сервере можем выполнить команду

| pip3 install -r requirements.txt |
| --- |

При этом должны установиться все зависимости из файла requirements.txt. Создавать или нет виртуальное окружение на удалённом сервере — решать вам. Если планируете развернуть несколько проектов, лучше создать.

Главное — благодаря виртуальному окружению на машине разработчика у вас есть файл с зависимостями конкретного проекта.

Как можно сымитировать ситуацию с удалённым сервером у себя на компьютере? Очень просто: создать ещё одно виртуальное окружение и активировать его — вот и «чистый лист».

Теперь ещё одна манипуляция — **деактивация** виртуального окружения:

| deactivate |
| --- |

Префикс “(virt)” в терминале должен исчезнуть.

# Модули и пакеты в Python

## Модули

Модуль — это файл с расширением .py. То есть всё, что вы до этого писали, — модули. Модуль должен иметь некоторое качество «целостности», у него должно быть какое-то назначение. Возьмём фреймворк [Django](https://www.djangoproject.com/). В каждом приложении есть следующие модули:

* models.py — код, описывающий таблицы в базе данных;
* views.py — код, отвечающий за обработку запросов пользователей;
* urls.py — диспетчеры адресов, отвечающие за роутинг (перенаправление обработчикам) входящих запросов;
* admin.py — настройки админки;
* tests.py — тесты.

Часто в проектах можно встретить модуль exceptions.py — очевидно, что он как-то связан с исключениями. Или модуль utils.py — как правило, содержит полезные функции и классы, которые используются другими модулями в проекте.

Можно сначала писать код в одном файле. Постепенно вы начнёте чувствовать, что размер файла слишком большой и с ним трудно работать. В такой ситуации можно попытаться выполнить [декомпозицию кода](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%B7%D0%B8%D1%86%D0%B8%D1%8F) на отдельные [слабо связанные между собой](https://habr.com/ru/post/449586/) части. В результате вы получите несколько модулей. Как правило, есть модуль, который выполняет основную задачу, он может называться app.py. В этом модуле делаем импорты из остальных и реализуем их взаимодействие.

Первой важной особенностью модулей в Python является их имя — оно ОБЯЗАТЕЛЬНО должно быть написано в стиле snake\_case и только английскими буквами. Иначе модуль невозможно будет импортировать.

Вторая важная особенность — иногда в конце модуля мы пишем следующее условие:

hello\_module.py

| def say\_hello(name):  print(f'Привет, {name}!')   if \_\_name\_\_ == '\_\_main\_\_':  say\_hello('Иван') |
| --- |

Код, написанный внутри этого условия, выполняется ТОЛЬКО если вы запускаете модуль, но он не будет выполняться, если вы его импортируете. Попробуйте запустить этот скрипт в командной строке или в PyCharm. Вы увидели фразу “Привет, Иван!” Теперь создадим в этой же папке файл lesson\_4.py и импортируем наш модуль:

lesson\_4.py

| import hello\_module  hello\_module.say\_hello('Лена') |
| --- |

При запуске этого файла код say\_hello('Иван') не выполнился. Попробуйте переписать файл hello\_module.py:

| def say\_hello(name):  print(f'Привет, {name}!')   say\_hello('Иван') |
| --- |

Что изменилось при запуске lesson\_4.py? Теперь мы видим две фразы, потому что при импорте модуля hello\_module выполняется его код.

Вывод: если вы планируете использовать некоторый скрипт в качестве модуля, обязательно [оборачивайте](https://docs.python.org/3.8/library/__main__.html#module-__main__) вызовы всех функций условием “if \_\_name\_\_ == '\_\_main\_\_':”.

## Пакеты

Пакет в Python — это папка, содержащая модули или другие пакеты и файл “\_\_init\_\_.py”. Пакеты появляются, когда у всех модулей есть какой-то общий смысл или они решают одну общую задачу. Это более высокий уровень интеграции, чем модуль.

Это как фотографии с конкретного мероприятия (модули) и папка (пакет), которая их содержит. То есть вы сначала ищете папку, а потом уже более детально — фотографии в ней.

Давайте посмотрим примеры пакетов. В Django есть пакет contrib (../site-packages/django/contrib) — в нём содержатся все приложения, которые идут «в коробке» вместе с фреймворком. По сути, это пакет пакетов. В нем есть пакет auth — приложение, решающее задачу регистрации и аутентификации пользователей в системе. В этом пакете есть модули, несущие отдельные ответственности, мы о них писали выше.

Таким образом, у разработчика появляется возможность оперировать как большими блоками — пакетами, так и маленькими — модулями и пакетами внутри основного пакета. И его задача — понять, в какой точке нужно вносить изменения, чтобы добавить новую фичу или поправить баг. Если вы понимаете, как это всё работает, то быстро найдёте эту точку.

Построение иерархии системы — решение, из каких пакетов и модулей она будет состоять, какая ответственность будет за этими частями, — это очень сложная задача, требующая большого опыта и знаний. Мы пока можем только изучать готовые решения и понимать, как они работают.

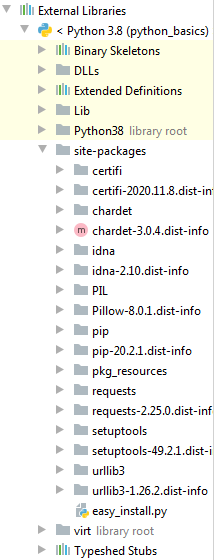
**Важно:** как правило, файл “\_\_init\_\_.py” пустой, но это не всегда так. В будущем вы откроете много новых возможностей, которые он предоставляет.

**Примечание:** иногда пакеты называют библиотеками. Хотя на самом деле библиотека — это [другое измерение](https://stackoverflow.com/questions/148747/what-is-the-difference-between-a-framework-and-a-library) в классификации программных продуктов: по способу использования. Антиподами библиотек являются фреймворки. Таким образом, и модуль, и пакет могут быть библиотекой, а пакет ещё может быть и фреймворком.

## Варианты импорта, их особенности

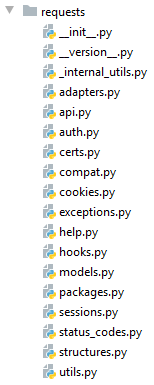
Давайте рассмотрим варианты импорта модулей и пакетов на примере пакета requests, который мы уже установили в виртуальное окружение.

Рекомендуем посмотреть содержимое вкладки External Libraries в PyCharm. Если вы выбрали интерпретатор из созданного виртуального окружения, в этой вкладке в папке site-packages будет папка requests с этим пакетом:



Если вы не работаете в PyCharm, можно найти эту папку в обычном проводнике.

Содержимое пакета:



Видим, что названия модулей в составе пакета «говорящие».

Посмотрим на содержимое файла \_\_init\_\_.py:

| ... from . import utils from . import packages from .models import Request, Response, PreparedRequest from .api import request, get, head, post, patch, put, delete, options from .sessions import session, Session from .status\_codes import codes from .exceptions import (  RequestException, Timeout, URLRequired,  TooManyRedirects, HTTPError, ConnectionError,  FileModeWarning, ConnectTimeout, ReadTimeout ) ... |
| --- |

Это один из вариантов импорта. Читается просто: «из <источник> импорт <что-то>». Точка означает импорт из той же папки, где находится скрипт, её ещё иногда называют корнем (root):

* from . import utils — из корневой папки импортируем модуль utils;
* from .api import request, get, head, ... — из модуля api, находящегося в корневой папке, импортируем функции get(), head() и т.д.

Что дают нам эти импорты в файле \_\_init\_\_.py? Вы можете импортировать всё это прямо из пространства имён пакета, например:

| import requests  response = requests.get('http://geekbrains.ru') print(response) *# <Response [200]>* |
| --- |

вместо

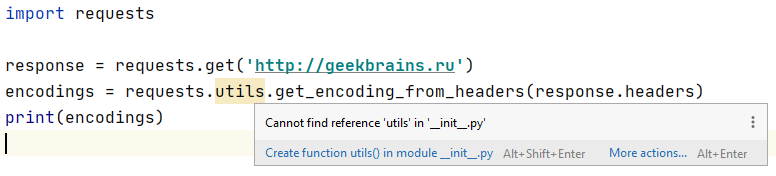
| import requests.api  response = requests.api.get('http://geekbrains.ru') print(response) *# <Response [200]>* |
| --- |

Если пока вы не понимаете разницы — не страшно, но нужно разобраться в этом вопросе в ближайшее время. Как вы думаете, если «похулиганить» и закомментировать строку from .api import request, get, head, ... в файле \_\_init\_\_.py — будет ли работать первый пример? Нет. А второй пример будет.

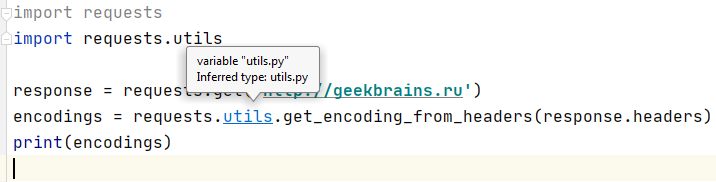
Теперь посмотрим, что дает импорт from . import utils:

| import requests  response = requests.get('http://geekbrains.ru') encodings = requests.utils.get\_encoding\_from\_headers(response.headers) print(encodings) *# utf-8* |
| --- |

Модуль utils доступен нам прямо из пространства имён пакета. Что будет, если закомментировать from . import utils в файле \_\_init\_\_.py? Скрипт по-прежнему запускается, но в PyCharm появилось предупреждение:



И мы не можем открыть этот модуль через сочетание Ctrl + mouse click. Однако мы можем корректно импортировать его, записав путь:



Сейчас модуль utils корректно импортирован, даже несмотря на закомментированную строку “from . import utils” в файле \_\_init\_\_.py. Вы заметили, что первая строка скрипта стала серой? Это подсказка со стороны PyCharm о неиспользуемом импорте. Можете нажать ещё одно полезное сочетание клавиш Ctrl + Alt + O — PyCharm автоматически поправит импорты в скрипте и уберёт лишнее. Какой можем сделать вывод? Импорт модуля из пакета автоматически приводит к импорту самого пакета.

Вернём исходное состояние файла \_\_init\_\_.py и рассмотрим ещё один вариант импорта — [использование псевдонимов](https://www.python.org/dev/peps/pep-0221/#rationale) (alias):

| import requests import requests.utils as utils  response = requests.get('http://geekbrains.ru') encodings = utils.get\_encoding\_from\_headers(response.headers) print(encodings) |
| --- |

Можно прочитать импорт во второй строке следующим образом: «импортируем <источник> как <пространство имён>» . Такой способ используется, когда путь к модулю или имя пакета длинные. С псевдонимами надо быть аккуратнее. Есть некоторые [общепринятые](https://google.github.io/styleguide/pyguide.html#22-imports) в сообществе имена, которые нельзя менять:

| import numpy as np import pandas as pd |
| --- |

Попробуем в нашем скрипте третий вариант импорта — в глобальное пространство имён:

| from requests import get, utils  response = get('http://geekbrains.ru') encodings = utils.get\_encoding\_from\_headers(response.headers) print(encodings) |
| --- |

Этот способ хорош тем, что не нужно писать пространство имён модуля или пакета в коде, но при большом количестве импортируемых функций и классов строка с импортом может получиться длинной:

| from requests import request, get, head, post, patch, put, delete, \  options, utils, session, Session, codes |
| --- |

Многие в такой ситуации используют «коварный» вариант импорта:

| from requests import \* |
| --- |

НАСТОЯТЕЛЬНО не рекомендуем так поступать. В перспективе это может привести к проблемам в проекте. К тому же это «неявно», а значит, противоречит философии Python. Лучше вернуться к варианту импорта в своё пространство имен: import requests.

Ещё один вредный совет:

| import math, requests, numpy |
| --- |

НЕЛЬЗЯ импортировать несколько модулей или пакетов в одной строке. Правильный вариант:

| import math  import numpy import requests |
| --- |

**Внимание:** [сначала](https://www.python.org/dev/peps/pep-0008/#:~:text=Imports%20are%20always%20put%20at,Related%20third%20party%20imports.) должны быть импорты из стандартной библиотеки, потом модули и пакеты третьих лиц, и только потом — ваши. Разумеется, **ВСЕ** импорты должны быть в начале скрипта: ни в коем случае не пишем их где попало.

**\*Замечание:** иногда приходится делать импорты внутри функций — так называемые локальные импорты. Не рекомендуем так делать, пока вы не осознали реальной необходимости: по факту локальные импорты используются редко и зачастую являются «костылём», исправляющим архитектурные проблемы. Но иногда они реально полезны.

# Библиотека requests

Давайте наконец-то посмотрим, что мы можем сделать с результатом функции get() библиотеки requests:

| from requests import get  response = get('http://geekbrains.ru') print(type(response)) *# <class 'requests.models.Response'>* print(dir(response)) *# [... 'apparent\_encoding', 'close', 'connection', 'content', 'cookies',*  *# 'elapsed', 'encoding', 'headers', 'history', 'is\_permanent\_redirect',* *# 'is\_redirect', 'iter\_content', 'iter\_lines', 'json', 'links', 'next', 'ok',* *# 'raise\_for\_status', 'raw', 'reason', 'request', 'status\_code', 'text',*  *# 'url']* |
| --- |

Здесь мы выполнили GET-запрос к серверу. Могли ли выполнить POST, PUT или еще какой-нибудь из доступных для протокола [http](https://ru.wikipedia.org/wiki/HTTP)? Могли. Как это узнать? Читать [документацию](https://requests.readthedocs.io/en/master/) или просто внимательно смотреть исходники.

Результат запроса — это объект класса Response. Где описан этот класс? Видим, что в модуле models пакета requests. Если надо будет, сходим и посмотрим реализацию. Атрибуты и методы у этого объекта:

* атрибут headers — словарь с заголовками ответа сервера;
* атрибут status\_code — число, [код ответа](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP) сервера;
* атрибут content — содержимое ответа;
* метод close() — освобождает (закрывает) соединение.

Смысла описывать здесь все атрибуты и методы нет. Когда у вас будет реальная задача, связанная с выполнением запросов к какому-нибудь [web API](https://developer.mozilla.org/ru/docs/Learn/JavaScript/Client-side_web_APIs/Introduction), тогда будет смысл погрузиться подробнее в эту библиотеку. Главное — понимать её структуру.

Например, попробуем посмотреть ответ сервера:

| from requests import get, utils  response = get('http://geekbrains.ru') encodings = utils.get\_encoding\_from\_headers(response.headers) content = response.content.decode(encoding=encodings) print(content) *# <!DOCTYPE html><html><head>...* |
| --- |

В принципе всё логично: нужен контент — обращаемся к атрибуту .content. Нужно только учесть, что ответ приходит в бинарном виде, поэтому надо его декодировать. А дальше можно делать что угодно. По факту у нас в руках обычный html-код. Можем работать с ним в синтаксисе jQuery через библиотеку [pyquery](https://pypi.org/project/pyquery/), можем использовать методы строк. Есть более серьёзные инструменты для [web-скрейпинга](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%BA%D1%80%D0%B5%D0%B9%D0%BF%D0%B8%D0%BD%D0%B3#:~:text=web%20scraping)%20%E2%80%94%20%D1%8D%D1%82%D0%BE%20%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F%20%D0%BF%D0%BE%D0%BB%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D1%8F,GET%2D%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81%D1%8B%20%D0%BD%D0%B0%20%D1%86%D0%B5%D0%BB%D0%B5%D0%B2%D0%BE%D0%B9%20%D1%81%D0%B0%D0%B9%D1%82.), например, фреймворк [Scrapy](https://scrapy.org/). Есть большая вероятность, что именно такая задача будет в тестовом задании перед собеседованием.

# Модуль sys: запуск скрипта с параметрами

Приходилось ли вам работать с утилитами командной строки? Конечно, да:

| ping geekbrains.ru |
| --- |

На самом деле в будущем вы только в командной строке и будете работать. Очень часто необходимо передать скрипту параметры снаружи. Например, когда мы создаем django-проект — выполняем команду:

| django-admin startproject geekbrains |
| --- |

Здесь вызвали команду startproject с аргументом geekbrains. В результате появится папка с проектом geekbrains. Передадите другое имя — получите другой проект.

Напишем скрипт для сложения чисел в командной строке (терминале):

sum\_numbers.py

| def main(argv):  program, \*args = argv  result = sum(map(int, args))  print(f'результат: {result}')   return 0   if \_\_name\_\_ == '\_\_main\_\_':  import sys   exit(main(sys.argv)) |
| --- |

Запустим его в терминале:

| python sum\_numbers.py 2 3 4 5 6 7 8 результат: 35 |
| --- |

Работает. Как мы получили эти числа внутри скрипта? При помощи переменной [argv](https://docs.python.org/3.8/library/sys.html#sys.argv) из модуля sys. По факту это список, первый элемент которого — имя скрипта, который был запущен, а остальные — аргументы, переданные скрипту через пробел (узнаёте позиционные аргументы у функций в Python?). Мы передали эту переменную функции main() и внутри неё получили аргументы при помощи уже знакомой распаковки. Дальше — преобразовали в целые числа и сложили.

В этом примере мы показали достаточно «взрослый» код: импорт модуля sys происходит только при запуске скрипта, его не будет при импорте. Также мы пробрасываем код возврата из функции main() в функцию exit() — пока это 0 (всё прошло хорошо), но в будущем могут быть и другие числа. Код возврата важен при организации конвейеров из скриптов в будущем.

Упрощённый вариант скрипта:

| import sys  result = sum(map(int, sys.argv[1:])) print(f'результат: {result}') |
| --- |

Подумайте, что еще должно быть в настоящем коде? Конечно, обработка ошибок. Но мы пока этого вопроса касаться не будем.

**Рекомендуем** в будущем изучить модуль [argsparse](https://docs.python.org/3.8/library/argparse.html#module-argparse) — он позволяет писать серьёзные инструменты для командной строки.

**Примечание:** если запустить скрипт не получилось, вам нужно в [командной](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%BE%D0%B9_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8) строке [перейти](https://ru.wikipedia.org/wiki/Cd_(%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0)) в папку, где он находится. Проверить содержимое текущей папки можно при помощи команды dir.

# Модуль time: профилируем время выполнения участков кода

Вы слышали о [профилировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%84%D0%B8%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#:~:text=%D0%9F%D1%80%D0%BE%D1%84%D0%B8%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%20%E2%80%94%20%D1%81%D0%B1%D0%BE%D1%80%20%D1%85%D0%B0%D1%80%D0%B0%D0%BA%D1%82%D0%B5%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%20%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%8B%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B,profiler).)? Это очень важный этап работы над проектом. Задача — поиск узких мест (bottleneck) по времени выполнения или по памяти. Попробуем сравнить скорость работы циклов while и for in при обходе последовательностей:

| import time  nums = [] for num in range(1, 10 \*\* 6 + 1):  nums.append(num)  start = time.perf\_counter() nums\_sum = 0 i = 0 while i < len(nums):  nums\_sum += nums[i]  i += 1 finish = time.perf\_counter() print(nums\_sum, finish - start, 'while')  start = time.perf\_counter() nums\_sum = 0 for num in nums:  nums\_sum += num finish = time.perf\_counter() print(nums\_sum, finish - start, 'for in')  *# 500000500000 0.424889671 while* *# 500000500000 0.127061353 for in* |
| --- |

Пропасть между результатами: вы всё ещё используете while? Тогда мы идём к вам.

При помощи функции perf\_counter() фиксируем моменты времени: старт и финиш. По факту это вещественные числа — секунды, поэтому можем просто их вычесть и получим длину интересующего промежутка времени в секундах. Если нужно точнее, есть функция perf\_counter\_ns().

# Модуль datetime: работа с датой и временем

Поработаем ещё с одним полезным модулем. Представим, что вам понадобилось посчитать, сколько дней было между двумя датами. Можно написать скрипт, который будет решать эту задачу? Можно. Нужно будет учесть количество дней в каждом месяце, причём для февраля придется учесть, високосный год или нет. Теперь решение при помощи модуля datetime:

| from datetime import datetime  date\_1 = datetime(year=2020, month=12, day=5) date\_2 = datetime(year=2019, month=12, day=5) date\_delta = date\_1 - date\_2 print(date\_delta.days) *# 366* |
| --- |

Этим и хорош Python: простое делается по-простому. Создали две даты и вычли их — всё. Правда, часто у начинающих разработчиков возникает путаница: дважды пишем datetime в импорте. Просто так получилось, что в модуле datetime есть класс [datetime](https://docs.python.org/3.8/library/datetime.html#datetime.datetime), который мы импортировали.

А если нужно разницу в секундах? Сделаем:

| import datetime  datetime\_1 = datetime.datetime(year=2020, month=12, day=5,  hour=18, minute=57, second=30) datetime\_2 = datetime.datetime(year=2020, month=1, day=1,  hour=0, minute=0, second=0) datetime\_delta = datetime\_1 - datetime\_2 print(datetime\_delta.seconds) *# 68250* |
| --- |

datetime\_delta.seconds вернул 68250 секунд - разницу между 18:57:30 и 0:00:00 без учёта дней, месяцев и лет.

Вы обратили внимание, что в первом примере обращались к атрибуту .days, а во втором — к атрибуту .seconds? К тому же во втором примере мы показали другой вариант импорта, поэтому пришлось писать более громоздко: datetime.datetime.

А если нужно сделать прибавить к моменту времени несколько дней или часов? Например, если нам нужно определить, валиден ли код активации:

| from datetime import datetime, timedelta  user\_created = datetime(year=2020, month=12, day=4,  hour=15, minute=25, second=32)  activation\_period = timedelta(days=1, hours=12)  datetime\_now = datetime.now() if datetime\_now < user\_created + activation\_period:  print('еще можно активировать аккаунт: {time\_before}'.format(  time\_before=datetime\_now - user\_created  )) else:  print('не хватило {time\_after}'.format(  time\_after=datetime\_now - (user\_created + activation\_period)  ))  *# еще можно активировать аккаунт: 1 day, 4:06:52.026858* |
| --- |

Обратите внимание, как мы получили текущий момент времени — через метод .now() класса datetime. А разницу во времени задали через объект класса timedelta. Также в этом примере мы использовали метод .format() с именованными аргументами для формирования текста сообщения. Почему? Делать вычисления в f-строках не очень хорошая идея. Создавать отдельные переменные — лишние строки кода.

Также в модуле datetime реализована возможность работать с временными зонами и многое другое.

# Практическое задание

1. Проверить, установлен ли пакет pillow в глобальном окружении. Если да — зафиксировать версию. Установить самую свежую версию pillow, если ранее она не была установлена. Сделать подтверждающий скриншот. Создать и активировать виртуальное окружение. Убедиться, что в нем нет пакета pillow. Сделать подтверждающий скриншот. Установить в виртуальное окружение pillow версии 7.1.1 (или другой, отличной от самой свежей). Сделать подтверждающий скриншот. Деактивировать виртуальное окружение. Сделать подтверждающий скриншот. Скрины нумеровать двухразрядными числами, например: «01.jpg», «02.jpg». Если будут проблемы с pillow - можно поработать с другим пакетом: например, requests.
2. Написать функцию currency\_rates(), принимающую в качестве аргумента код валюты (например, USD, EUR, GBP, ...) и возвращающую курс этой валюты по отношению к рублю. Использовать библиотеку requests. В качестве API можно использовать [**http://www.cbr.ru/scripts/XML\_daily.asp**](http://www.cbr.ru/scripts/XML_daily.asp)**.** Рекомендация: выполнить предварительно запрос к API в обычном браузере, посмотреть содержимое ответа. Можно ли, используя только методы класса str, решить поставленную задачу? Функция должна возвращать результат числового типа, например float. Подумайте: есть ли смысл для работы с денежными величинами использовать вместо float тип [Decimal](https://docs.python.org/3.8/library/decimal.html)? Сильно ли усложняется код функции при этом? Если в качестве аргумента передали код валюты, которого нет в ответе, вернуть None. Можно ли сделать работу функции не зависящей от того, в каком регистре был передан аргумент? В качестве примера выведите курсы доллара и евро.
3. \*(вместо 2) Доработать функцию currency\_rates(): теперь она должна возвращать кроме курса дату, которая передаётся в ответе сервера. Дата должна быть в виде объекта date. Подумайте, как извлечь дату из ответа, какой тип данных лучше использовать в ответе функции?
4. Написать свой модуль utils и перенести в него функцию currency\_rates() из предыдущего задания. Создать скрипт, в котором импортировать этот модуль и выполнить несколько вызовов функции currency\_rates(). Убедиться, что ничего лишнего не происходит.
5. \*(вместо 4) Доработать скрипт из предыдущего задания: теперь он должен работать и из консоли. Например:

| > python task\_4\_5.py USD 75.18, 2020-09-05 |
| --- |

Задачи со \* предназначены для продвинутых учеников, которым мало сделать обычное задание.

# Дополнительные материалы

1. [Лутц Марк. Изучаем Python](https://www.ozon.ru/context/detail/id/5730448/).
2. [Модуль datetime](https://docs.python.org/3.8/library/datetime.html#module-datetime).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://realpython.com/command-line-interfaces-python-argparse/>.
2. <https://docs.python.org/3/>.