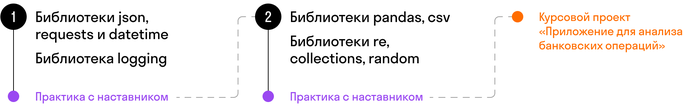
# 1. О курсе

Добро пожаловать на курс «Библиотеки для Python-разработчика»! Вы познакомитесь с инструментами, которыми разработчики пользуются каждый день при написании кода, — библиотеками. Библиотеки расширяют стандартные возможности языка и предоставляют новые инструменты для работы.

**Структура курса**



* Каждую неделю вам будет открываться по два урока с практическими заданиями.
* На еженедельном практическом занятии с наставником вы можете задать вопросы по темам уроков и потренироваться решать задачи в формате live-кодинга (написание кода в реальном времени).
* После двух недель обучения вы сможете начать работу над проектом.

**Домашки курса**

Вы продолжите работать над проектом «Виджет финансовых операций» и наращивать его функциональность.

**Механики уроков**

Каждый урок состоит из определенных разделов, которые помогают разобраться в теме:

* **Контент урока** — посмотрите видео, повторите решение за экспертом, изучите конспект под видео, прорешайте тесты и зафиксируйте для себя ключевые моменты.
* **Вопросы для самопроверки** — проверьте себя в формате прохождения собеседования, устно ответьте на вопросы и сверьтесь с ответами под спойлерами.
* **Итоги** — закрепите ключевые моменты урока и подведите итоги.
* **Подготовка к практике с наставником** — прорешайте задачи, изучите решение наставника и подготовьтесь к прохождению собеседований в формате live-кодинга.
* **Конспект** — перечитайте целиком и закрепите материал урока, зафиксируйте вопросы, которые у вас остались, чтобы обсудить с наставником и одногруппниками.
* **Домашнее задание** — отработайте практический навык в решении задач по теме урока.
* **Практика с наставником (лайв в выходной день)** — задайте наставнику вопросы по теме урока, ответьте на вопросы в формате прохождения интервью и прорешайте задачи в реальном времени.

**Ключевые навыки курса**

На уроках курса вы научитесь:

* Работать с форматом JSON, преобразовывать JSON-строки в Python-объекты и обратно, а также работать с JSON-файлами.
* Получать данные с внешних систем (API) с помощью библиотеки requests, использовать закрытые авторизацией API и передавать параметры в запросы к внешним ресурсам.
* Работать с объектами даты и времени в Python, получать различные атрибуты дат и времени, преобразовывать и форматировать даты.
* Добавлять логирование в проект, использовать тот уровень логирования, который требуется, добавлять логи в файл и изменять их формат.
* Считывать и записывать данные с помощью библиотеки csv.
* Работать с библиотекой pandas для обработки данных: считывать данные из файлов, выводить основную информацию по данным, записывать данные в различные форматы, производить фильтрации, сортировки, очистки, группировки и агрегации данных.
* Работать с регулярными выражениями для поиска и замены текста, использовать классы из библиотеки collections для работы с контейнерами, работать с методами библиотеки random для генерации случайных чисел, выбора случайных элементов из списка, перемешивания списка и генерации случайных строк.

Рекомендуем вам активно практиковаться при решении задач, работе с кодом и разработке проектов, чтобы закрепить полученные знания и стать уверенным разработчиком.

Удачи при изучении курса!

# 2. План урока

**На уроке**

1. Рассмотрите, что такое **JSON**, узнаете о базовых аспектах JSON и его структуре. JSON (JavaScript Object Notation) — это формат обмена данными, используемый в программировании и веб-разработке. Вы узнаете, как использовать **библиотеку**

json

 для преобразования данных между форматом JSON и объектами Python, а также для работы с JSON-файлами, включая чтение, запись и изменение данных.

1. Рассмотрите **методы**

loads

 и

dumps

 для преобразования данных между форматом JSON и объектами Python. Узнаете, как эти методы помогают в сериализации и десериализации данных. Научитесь читать, записывать и изменять JSON-файлы в Python.

1. Познакомитесь с **библиотекой**

requests

. Рассмотрите методы

GET

 и

POST

, обработку ошибок и аутентификацию, а также работу с API и форматом JSON. Вы узнаете, как работать с HTTP-запросами, отправлять данные на сервер и получать ответы, а также как обрабатывать возможные ошибки и аутентифицироваться на сервере.

1. Для тестирования HTTP-запросов и вызовов API через

requests

 часто используются **объекты**

Mock

 и

patch

. Вы рассмотрите, что такое

Mock

 и

patch

 и как их использовать для тестирования кода. Разберете примеры тестирования функций, зависящих от внешних данных.

1. Познакомитесь с **библиотекой**

datetime

, которая используется для работы с датами и временем. Узнаете, как использовать библиотеку

datetime

 для создания объектов

datetime

, форматирования дат и времен, создания объектов

datetime

 из строк, получения текущей даты и времени, а также изменения дат и времен.

**Про домашку**

Вы продолжите работать над проектом. В этой домашке реализуете новый модуль для банковского приложения, который будет читать данные о банковских операциях из JSON-файла, получать данные для конвертации валюты с помощью стороннего API. Напишете тесты для нового функционала.

# 3. Библиотека json

**Формат JSON**

**JSON (JavaScript Object Notation)** — это формат обмена данными, основанный на языке программирования JavaScript. Используется для передачи данных между клиентскими и серверными приложениями в веб-разработке, а также для сохранения и обмена данными в различных программах.

JSON представляет собой формат, который легко читается и создается, а также прост в использовании в различных языках программирования.

Основные правила создания JSON-строки:

* Объекты записываются в виде пар «ключ: значение» и заключаются в фигурные скобки.
* Массивы представляют собой последовательность значений и записываются внутри квадратных скобок.
* Пары «ключ: значение» в объектах и элементы в массивах разделяются запятыми.
* Ключи в объектах должны быть заключены в двойные кавычки

""

. Ключ — это строка, которая обычно является именем свойства или атрибута.

* Значениями могут быть строки, числа, объекты, массивы, логические значения (true или false) или

null

. У значений в двойные кавычки заключаются только строки.

**Обратите внимание**

* В JSON-формате используются только двойные кавычки.
* К сбою JSON-файла может привести одна лишняя или пропущенная запятая или скобка.

Пример, как выглядит JSON-строка:

people\_string = '''

{

"people": [

{

"name": "John Smith",

"phone": "666-555-444",

"emails": ["johnsmith@home.com", "john.smith@work.com"],

"has\_license": false

},

{

"name": "Jane Doe",

"phone": "111-222-333",

"emails": null,

"has\_license": true

}

]

}

'''

В этом примере данные организованы в виде объекта и массива. Объекты заключены в фигурные скобки.

В этом объекте 4 пары ключей и значений, которые разделяются запятыми:

{

"name": "Jane Doe",

"phone": "111-222-333",

"emails": null,

"has\_license": true

}

Если объектов много, они хранятся в массиве. Массив объектов заключается в квадратные скобки:

[

{

"name": "John Smith",

"phone": "666-555-444",

"emails": ["johnsmith@home.com", "john.smith@work.com"],

"has\_license": false

},

{

"name": "Jane Doe",

"phone": "111-222-333",

"emails": null,

"has\_license": true

}

]

**Основные методы библиотеки json**

Для работы с данными в формате JSON в Python есть **библиотека**

json

.

Рассмотрим основные методы библиотеки.

**Преобразование данных**

Методы

loads

 и

dumps

 позволяют преобразовывать данные между форматом JSON и объектами Python. Такие преобразования называются сериализацией и десериализацией данных.

**Сериализация** — это преобразование Python-объектов в JSON-строку.

**Десериализация** — это преобразование JSON-строки в Python-объекты.

**Метод**

loads()

 — используется для преобразования строки JSON в объект Python.

Рассмотрим работу метода на примере:

# Импортируем библиотеку json

import json

# data - строка, тип str

data = '{"name": "John Smith", "age": 30, "city": "New York"}'

# parsed\_data - словарь, тип dict

parsed\_data = json.loads(data)

# Получаем значение словаря по ключу

print(parsed\_data["name"])

>>> John Smith

Еще один пример работы с данными в формате JSON, хранящимися в строке:

import json

people\_string = '''

{

"people": [

{

"name": "John Smith",

"phone": "666-555-444",

"emails": ["johnsmith@home.com", "john.smith@work.com"],

"has\_license": false

},

{

"name": "Jane Doe",

"phone": "111-222-333",

"emails": null,

"has\_license": true

}

]

}

'''

data = json.loads(people\_string)

for person in data["people"]:

print(person["name"])

>>> John Smith

Jane Doe

**Метод**

dumps()

 — используется для преобразования объекта Python в строку JSON.

Пример работы метода:

import json

# data - словарь, тип dict

data = {

"name": "John Smith",

"age": 30,

"city": "New York"

}

# json\_data - строка, тип str

json\_data = json.dumps(data)

print(json\_data)

>>> {"name": "John Smith", "age": 30, "city": "New York"}

**Чтение JSON-файлов**

Для чтения JSON-файла в Python необходимо использовать метод

load

 из библиотеки

json

.

**Метод**

load

 — загружает данные из файла в формате JSON и преобразует их в объект Python.

Предположим, что у нас есть файл

data.json

 с таким содержанием:

{

"name": "John",

"age": 30,

"city": "New York"

}

Для чтения этого файла и получения данных в формате Python необходимо выполнить следующий код:

import json

with open('data.json') as f:

data = json.load(f)

print(data) # data - словарь, тип dict

В результате выполнения этого кода будет выведено содержимое файла

data.json

 в формате Python:

{'name': 'John', 'age': 30, 'city': 'New York'}

**Запись JSON-файлов**

Для записи данных в JSON-файл в Python также используется библиотека

json

. Для этого создайте объект Python, который необходимо сохранить в JSON-файле, а затем используйте метод

dump

.

**Метод**

dump

 — преобразует объект Python в формат JSON и записывает его в файл.

Допустим, мы хотим сохранить следующие данные в файл

output.json

:

data = {

"name": "John",

"age": 30,

"city": "New York"

}

Для записи этих данных в файл

output.json

 выполним код:

import json

with open('output.json', 'w') as f:

json.dump(data, f)

В результате в файл

output.json

 будет записан следующий JSON-объект:

{

"name": "John",

"age": 30,

"city": "New York"

}

**Изменение JSON-файлов**

Чтобы изменить содержимое JSON-файла, необходимо сначала прочитать его в Python, затем произвести изменения в объекте Python и, наконец, записать измененные данные обратно в JSON-файл.

Рассмотрим на примере. Допустим, мы хотим изменить значение ключа

age

 на 32 в файле

data.json

. Для этого нам необходимо выполнить следующий код:

import json

with open('data.json') as f:

data = json.load(f)

data['age'] = 32

with open('data.json', 'w') as f:

json.dump(data, f)

Рассмотрим код подробнее:

1. **Импортируем библиотеку**

json

, которая предоставляет методы для работы с JSON-данными.

1. **Откроем файл и прочитаем данные:**
   * with open('data.json') as f:

 — открывает файл

data.json

 в режиме чтения.

with

 обеспечивает автоматическое закрытие файла после выполнения блока кода.

* + data = json.load(f)

 — читает данные из файла

f

 и преобразует их из формата JSON в объект Python.

1. **Изменяем значение ключа**

age

**в словаре**

data

**на**

32

. Если такого ключа не было, он будет добавлен:

data['age'] = 32

1. **Записываем измененные данные обратно в файл:**
   * with open('data.json', 'w') as f:

 — открывает файл

data.json

 в режиме записи (

'w'

), что позволяет перезаписать его содержимое.

* + json.dump(data, f)

 — преобразует объект

data

 обратно в формат JSON и записывает его в файл

f

.

В результате значение ключа

age

 в файле

data.json

 будет изменено на 32.

**Дополнительные параметры при работе с JSON**

При использовании метода

json.dumps()

 можно указать дополнительные параметры для настройки формата выходной строки JSON.

1. **Параметр**

indent

 — используется для задания отступов при выводе JSON-строки. Это позволяет сделать JSON-строку более читаемой для пользователя.

Пример использования параметра

indent

:

import json

# data - словарь, тип dict

data = {

"name": "John Smith",

"age": 30,

"city": "New York"

}

# json\_data - строка, тип str, с отступами

json\_data = json.dumps(data, indent=4)

print(json\_data)

Этот код выведет отформатированную JSON-строку с отступами в 4 пробела:

{

"name": "John Smith",

"age": 30,

"city": "New York"

}

1. **Параметр**

ensure\_ascii

 — используется, чтобы задать кодировку символов. По умолчанию этот параметр установлен в

True

. Это означает, что все не-ASCII символы (например, кириллица) будут закодированы. Если установить параметр в

False

, то символы будут выводиться в оригинальной кодировке.

Пример использования параметра

ensure\_ascii

:

import json

data = {

"name": "Иван Иванов",

"age": 30,

"city": "Москва"

}

# json\_data - строка с не-ASCII символами

json\_data = json.dumps(data, ensure\_ascii=False)

print(json\_data)

Этот код выведет JSON-строку с не-ASCII символами в оригинальной кодировке:

{"name": "Иван Иванов", "age": 30, "city": "Москва"}

Дополнительные параметры позволяют более гибко настраивать форматирование JSON-строк в зависимости от ваших нужд.

# 4. Ошибки при работе с библиотекой json

Библиотека

json

 используется для работы с форматом данных JSON. Однако при выполнении операций с JSON могут возникать различные ошибки. Рассмотрим некоторые из них.

## JSONDecodeError

Эта ошибка возникает, когда невозможно декодировать (преобразовать) JSON-данные. Например, если JSON-строка имеет неправильный формат, содержит некорректные символы или имеет другие ошибки.

Код ниже вызывает ошибку:

import json

json\_string = "{'name': 'John', 'age': 30, 'city': 'New York'}"

# Попытка декодировать JSON-строку вызовет JSONDecodeError

data = json.loads(json\_string)

В данном примере мы видим одинарные кавычки — их нельзя использовать в JSON.

Рассмотрим еще один пример, когда может возникнуть ошибка JSONDecodeError:

import json

json\_string = '{"name": "Alice", "age": 25'

# Попытка декодировать JSON-строку вызовет JSONDecodeError

data = json.loads(json\_string)

В данном примере неверная JSON-строка — отсутствует закрывающая скобка.

Чтобы обработать ошибку JSONDecodeError в коде, можно воспользоваться следующей конструкцией:

try:

data = json.loads(json\_string)

except json.JSONDecodeError:

print("Invalid JSON data.")

## TypeError

Эта ошибка возникает, когда тип объекта не поддерживается операцией сериализации.

Например, если вы пытаетесь сериализовать объект типа

set

, вы получите ошибку TypeError. Чтобы обработать эту ошибку, нужно использовать следующий код:

import json

data = {'name': 'John', 'age': 30, 'city': 'New York', 'hobbies': {'reading', 'swimming'}}

try:

json\_string = json.dumps(data)

except TypeError:

print("Object of type set is not JSON serializable.")

Ошибка, возникающая при сериализации объекта типа

set

, связана с тем, что формат JSON не поддерживает сериализацию множеств. Другими словами,

set

 не является JSON-совместимым типом данных. Чтобы решить эту проблему, можно преобразовать множество в список, который уже может быть сериализован в JSON.

## KeyError

Эта ошибка возникает, когда ключ не найден в JSON-объекте. Например, если вы пытаетесь получить значение ключа, который не существует.

Чтобы обработать эту ошибку, нужно использовать следующий код:

import json

json\_string = '{"name": "John", "age": 30}'

try:

data = json.loads(json\_string)

city = data['city'] # Вызовет ошибку KeyError

except KeyError:

print("Key 'city' not found in JSON data.")

## ValueError

Эта ошибка возникает, когда значение некорректно. Например, если вы пытаетесь декодировать JSON-данные, которые не являются объектом или массивом. Чтобы обработать эту ошибку, нужно использовать следующий код:

import json

json\_string = 'John'

try:

data = json.loads(json\_string) # Вызовет ошибку ValueError

except ValueError:

print("Invalid JSON data.")

Эти ошибки не охватывают все возможные исключения, но они являются наиболее распространенными. Каждая ошибка должна обрабатываться соответствующим образом, чтобы обеспечить более надежную работу программы.

# 6. Знакомство с API

Чтобы не писать весь функционал самостоятельно, разработчикам нужно эффективно интегрировать свои приложения с внешними сервисами (API) для получения данных, автоматизации задач и расширения функционала разработанного приложения.

**API (Application Programming Interface)** — это набор правил и спецификаций, позволяющих различным программным приложениям общаться друг с другом.

Иными словами, API предоставляет методы и структуры данных, которые разработчики могут использовать, чтобы взаимодействовать с определенным приложением или сервисом.

Предположим, вы разрабатываете мобильное приложение для просмотра погоды. Вы можете использовать API погодного сервиса для получения актуальных данных о погоде.

Рассмотрим, для каких целей используются API:

* Для интеграции с внешними сервисами, например интеграции с сервисами социальных сетей, платежными системами, картографическими сервисами.
* Разработки клиент-серверных приложений — когда сервер предоставляет данные и функционал, а клиент (мобильное приложение, веб-страница) их использует.
* Автоматизации задач, например автоматическое обновление данных, уведомления, предоставление данных для аналитики.

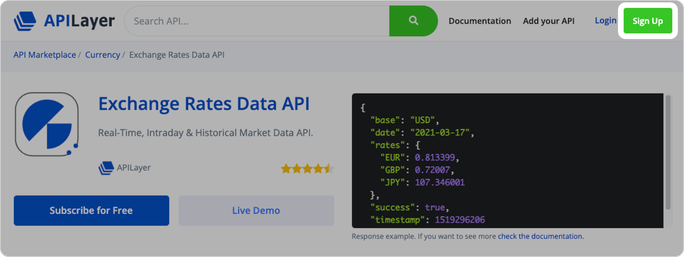
**Регистрация и получение токенов**

Для работы с большинством защищенных API необходима регистрация и получение ключей доступа (токенов).

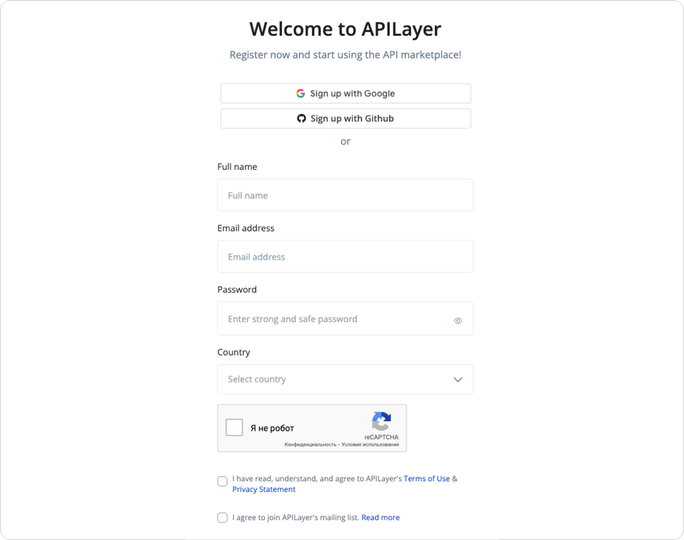
Рассмотрим процесс регистрации и получения токена на примере API конвертации валюты — APILayer.

**APILayer** — это сервис, который является агрегатором различных API. Через него можно получать доступ к API. Все возможные в сервисе API можно найти в поиске.

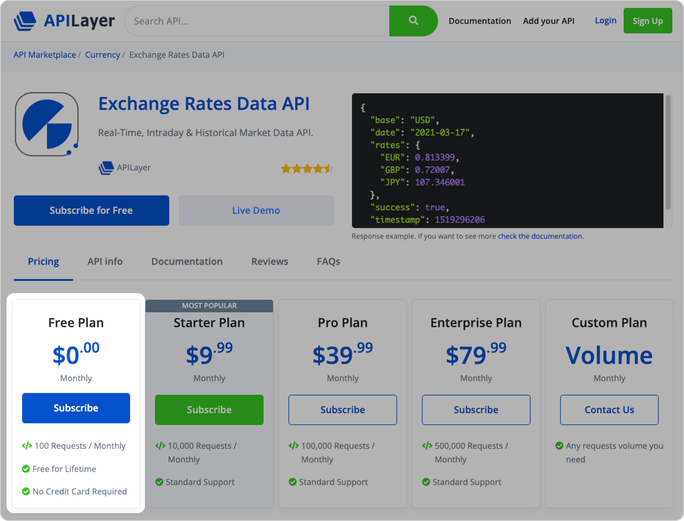
1. Зайдите на сайт APILayer — <https://apilayer.com/marketplace/exchangerates_data-api> — и зарегистрируйтесь на сервисе:



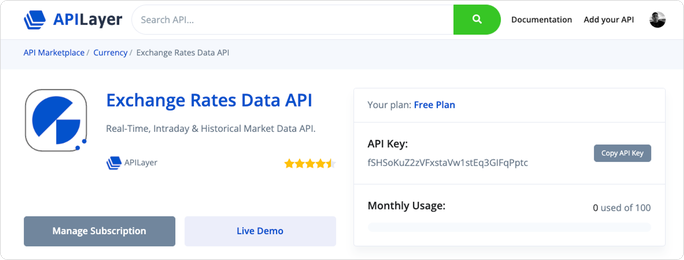
Вы можете выбрать «Зарегистрироваться через Google- или GitHub-аккаунт» или ввести свои данные и пароль в форму ниже:



1. После регистрации система обычно генерирует уникальный API-ключ (токен), который будет использоваться для доступа к API. В сервисе APILayer необходимо выбрать подписку. Для учебных целей нам подойдет бесплатная подписка.



1. После активации подписки будет сгенерирован API-ключ для доступа к API:



1. При отправке запросов к API используйте ваш API-ключ (токен) для идентификации вашего запроса. *Как отправлять запросы к API, мы подробно рассмотрим в одном из следующих разделов.*

**Документация к API**

Чтобы эффективно использовать внешний сервис, необходимо понимать, как он работает. Чтение и понимание документации к API является ключевым навыком для разработчиков, поскольку это основной способ узнать, как работать с внешними сервисами.

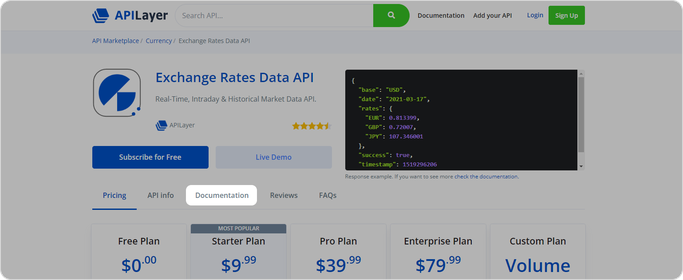
**Документация к API** — это руководство для разработчиков, содержащее описание всех доступных запросов, параметров и примеров использования API. Именно документация помогает разработчикам понять и правильно использовать API.

Документация большинства API написана на английском языке. Воспользуйтесь переводчиком, если читать по-английски окажется сложно.

Рассмотрим, как работать с документацией на примере API для конвертации валюты.

**Поиск документации**

Первым шагом при работе с API является поиск и доступ к документации, которую чаще всего можно найти на официальном сайте сервиса. Для APILayer Exchange Rates Data API это можно сделать, если перейти в раздел **Documentation** по ссылке: <https://apilayer.com/marketplace/exchangerates_data-api>



**Изучение структуры документации**

Документация обычно разделена на несколько основных частей:

* **Общие сведения** — введение в API, общие требования и ограничения.
* **Аутентификация** — информация о том, как регистрировать ваше приложение и получать ключи API.
* **Методы API** — детальное описание доступных методов, включая URL-адреса запросов, параметры запроса, форматы ответа и примеры.
* **Коды ответов** — описание возможных HTTP статус-кодов и их значения.
* **Примеры использования** — реальные примеры запросов и обработки ответов.

**Понимание методов API**

Рассмотрим использование API на примере для конвертирования валюты.

В разделе методов **Endpoints** найдите **GET /convert**. Вы увидите описание, как подготовить запросы для проведения конвертации валют. Основные составляющие:

* **Base URL**:

https://api.apilayer.com/exchangerates\_data/convert

* **API-ключ для авторизации** передается в заголовке запроса по ключу

apikey

.

* **Параметры запроса**:
  + amount

 — обязательный параметр суммы конвертации.

* + from

 — обязательный параметр — трехбуквенный код валюты, из которой происходит конвертация.

* + to

 — обязательный параметр — трехбуквенный код валюты, в которую происходит конвертация.

* **Пример запроса**:

https://api.apilayer.com/exchangerates\_data/convert?to={to}&from={from}&amount={amount}

**Обработка ответов**

Проверьте структуру ответа, как описано в документации. Обратите внимание на различные поля, такие как

info

,

result

,

query

, и убедитесь, что можете их корректно обрабатывать.

Чтение и использование документации API — это навык, который развивается с опытом. Чем больше вы практикуетесь, тем проще будет вам понимать и использовать различные API. Документация — ваш главный ресурс для успешной интеграции с любым сервисом.

# 7. Библиотека requests

Перед тем как начать работать с библиотекой

requests

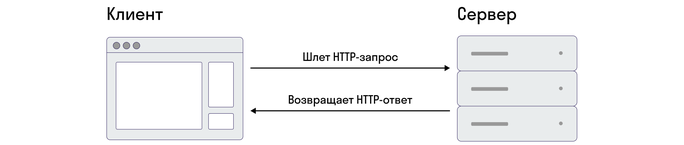
, разберемся, как работает сеть и что такое протокол HTTP.

**HTTP**

**HTTP (HyperText Transfer Protocol)** — это протокол передачи гипертекста, используемый для передачи данных в сети Интернет. Определяет способ передачи данных между клиентом и сервером, а также формат этих данных.

**Клиент** — тот, кто отправляет запросы, а **сервер** — тот, кто принимает запросы и возвращает ответы.

Рассмотрим, из чего состоит HTTP-запрос и HTTP-ответ.



**HTTP-запрос** состоит из заголовков и тела. Заголовки содержат метаинформацию о запросе, такую как тип и версия протокола, а также дополнительную информацию, например, о том, какой тип содержимого передается в теле запроса. Тело запроса содержит данные, которые передаются на сервер.

Пример HTTP-запроса

**HTTP-ответ** также состоит из заголовков и тела. Заголовки содержат метаинформацию об ответе, например, его тип и версию протокола, а также дополнительную информацию, например, о том, какой тип содержимого передается в теле ответа. Тело ответа содержит данные, которые возвращает сервер.

Пример HTTP-ответа

HTTP-запрос может иметь различные методы, например GET или POST.

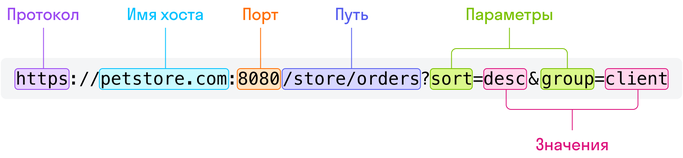
**Метод GET** используется для получения данных с сервера, а метод **POST** — для отправки данных на сервер. Метод GET передает данные в URL-адресе запроса, а метод POST передает данные в теле запроса.

**URL**

HTTP-запросы отправляются на определенный адрес (URL).

**URL (Uniform Resource Locator)** — адрес ресурса в интернете, который определяет, на какой сервер уходит запрос.

URL состоит из частей, которые характерны для каждого адреса в сети.



Давайте рассмотрим подробнее каждый из компонентов URL-адреса:

1. **Протокол** (

https

) — указывает на протокол, который используется для передачи данных. В данном случае это HTTPS (HyperText Transfer Protocol Secure), обеспечивающий защищенное соединение.

1. **Имя хоста** (

petstore.com

), или домен — идентифицирует сервер, к которому происходит обращение. В этом примере это

petstore.com

.

1. **Порт** (

8080

) — это числовой идентификатор, который определяет конкретный процесс на сервере, с которым устанавливается соединение. В данном примере используется порт

8080

. Если порт не указан, по умолчанию для HTTP используется порт 80, а для HTTPS — 443.

1. **Путь** (

/store/orders

) — указывает на конкретный ресурс на сервере. Здесь путь

/store/orders

 указывает на ресурс заказов в магазине.

1. **Параметры запроса** (

sort=desc&group=client

) — передаются после знака вопроса

?

 и состоят из пар «ключ:значение», разделенных знаком

&

. В данном примере:

* + параметр

sort=desc

 указывает, что сортировка должна быть в порядке убывания;

* + параметр

group=client

 указывает, что группировка должна быть по клиентам.

Эта структура позволяет клиентам, например браузерам или другим приложениям, точно указать, какой ресурс они хотят получить от сервера и какие параметры применить к запросу.

**Библиотека requests**

requests

 — библиотека, которая позволяет легко отправлять HTTP-запросы и работать с ответами.

Чтобы установить библиотеку

requests

, выполните команду:

poetry add requests

**Основные функции библиотеки**

**Отправка GET-запросов**

Для отправки GET-запроса необходимо вызвать метод

requests.get()

. Рассмотрим синтаксис его использования:

import requests

r = requests.get(url, headers=headers, params=params)

Где:

* url

 (обязательный параметр) — URL-адрес, к которому выполняется запрос.

* headers

 (необязательный параметр) — словарь, содержащий HTTP-заголовки, которые отправляются с запросом.

* params

 (необязательный параметр) — словарь или байтоподобный объект, который отправляется в строке запроса.

Например, чтобы получить содержимое страницы по адресу

https://xkcd.com/353/

, можно выполнить следующий код:

import requests

r = requests.get('https://xkcd.com/353/')

print(r.text)

В данном случае в консоль будет выведен HTML-текст страницы

https://xkcd.com/353/

.

**Получение содержимого ответа**

Ответ сервера можно получить в виде текста, бинарных данных или JSON-объекта. Для этого используются соответствующие атрибуты ответа от сервера:

* r.text

 — текстовое содержимое ответа;

* r.content

 — бинарное содержимое ответа, то есть информация в двоичном формате — в виде последовательности нулей и единиц;

* r.json()

 — JSON-объект, если возможно преобразование ответа в JSON.

Получив ответ в бинарном виде из ссылки на картинку, мы можем сохранить эту картинку в локальный файл:

r = requests.get('https://imgs.xkcd.com/comics/python.png')

print(r.content)

with open('comic.png', 'wb') as f:

f.write(r.content)

**Отправка POST-запросов**

Для отправки POST-запроса необходимо вызвать метод

requests.post()

. Рассмотрим синтаксис использования метода:

import requests

r = requests.post(url, headers=headers, data=data, json=json)

Где:

* url

 (обязательный параметр) — URL-адрес, к которому выполняется запрос.

* headers

 (необязательный параметр) — словарь, содержащий HTTP-заголовки, которые отправляются с запросом.

* data

 (необязательный параметр) — словарь, список кортежей, байтоподобный объект или файл для отправки в теле запроса.

* json

 (необязательный параметр) — словарь для отправки в теле запроса в формате JSON.

Например, чтобы отправить POST-запрос на адрес

https://httpbin.org/post

 с данными в формате

key=value

, выполните следующий код:

import requests

payload = {'my\_key': 'my\_value'}

r = requests.post('https://httpbin.org/post', data=payload)

print(r.text)

После отправки POST-запроса сервер вернет ответ, который можно получить в свойстве

r.text

. В данном случае мы отправили данные в формате

key=value

 на сервер

https://httpbin.org/post

, который в ответ вернул JSON-объект, содержащий в том числе отправленные данные.

**Статус-коды**

Ответ сервера на запрос начинается с числа — кода состояния HTTP.

**Статус-код** — это трехзначное число, которое указывает на результат запроса.

Статус-коды делятся на пять классов:

* **1xx** — информационные коды, которые означают, что запрос принят и обрабатывается.
* **2xx** — успешные коды, которые означают, что запрос выполнен успешно.
* **3xx** — перенаправляющие коды, которые означают, что запросу нужно следовать за другим адресом.
* **4xx** — ошибочные коды, которые означают, что запрос содержит ошибку или неверные данные.
* **5xx** — серверные коды, которые означают, что на стороне сервера произошла ошибка при обработке запроса.

Перечислим некоторые из наиболее часто встречающихся статус-кодов и их значения:

* **200** — OK. Запрос выполнен успешно и в ответе содержится запрошенная информация.
* **301** — Moved Permanently. Запросу нужно следовать за новым постоянным адресом, указанным в заголовке Location.
* **400** — Bad Request. Запрос содержит синтаксическую ошибку или неверные параметры.
* **401** — Unauthorized. Запрос требует аутентификации пользователя, которая не была предоставлена или не прошла проверку.
* **403** — Forbidden. Запрос отклонен сервером из-за ограничений доступа или безопасности.
* **404** — Not Found. Запросу не соответствует ни один ресурс на сервере.
* **500** — Internal Server Error. На стороне сервера произошла непредвиденная ошибка, которая не позволила выполнить запрос.

При отправке запросов можно ориентироваться на статус-код операции для определения успешности выполнения запросов.

Например:

# Импортируем библиотеку requests

import requests

# Задаем адрес сайта, к которому хотим обратиться

url = "https://example.com"

# Выполняем GET-запрос к сайту и сохраняем ответ в переменную response

response = requests.get(url)

# Получаем статус-код из ответа и выводим его на экран

status\_code = response.status\_code

print(f"Статус код: {status\_code}")

# Проверяем, равен ли статус-код 200, то есть чтобы запрос был успешным

if status\_code == 200:

# Выводим содержимое сайта на экран

content = response.text

print(f"Содержимое сайта:\n{content}")

else:

# Выводим сообщение об ошибке

print(f"Запрос не был успешным. Возможная причина: {response.reason}")

В этом примере:

1. Мы импортируем библиотеку

requests

.

1. Задаем адрес сайта, к которому хотим обратиться.
2. Выполняем GET-запрос к сайту и сохраняем ответ в переменную

response

.

1. Получаем статус-код из ответа и выводим его на экран.
2. Проверяем, равен ли статус-код 200, тем самым определяем успешность выполнения запроса. Если статус-код любой неуспешный (не 200), выводим сообщение об ошибке с причиной, которую получаем с помощью

response.reason

.

**Аутентификация**

Когда пользователь пытается получить доступ к защищенному ресурсу, ему нужно предоставить правильные учетные данные, такие как его API-ключ. Сервер затем проверяет эти учетные данные, чтобы убедиться, что они действительны, и разрешает или запрещает доступ в зависимости от результата проверки.

**Аутентификация** — это процесс проверки подлинности пользователя.

В общих случаях авторизация через API-ключ передается в заголовках (headers) запроса:

import requests

url = "https://api.apilayer.com/exchangerates\_data/convert"

headers = {

"apikey": "fSHSoKuZ2zVFxstaVw1stEq3GIFqPptc"

}

response = requests.get(url, headers=headers)

**JSON и API**

Для получения данных из ответа API в формате JSON с помощью библиотеки

requests

 можно использовать метод

response.json()

.

Код ниже использует библиотеку

requests

 для отправки GET-запроса к API GitHub, чтобы получить список репозиториев пользователя

skypro-008

. Затем он фильтрует репозитории, используя только те, которые написаны на языке Python, и выводит их название и ссылку на страницу репозитория.

import requests

user = "skypro-008"

url = f"https://api.github.com/users/{user}/repos"

response = requests.get(url)

repos = response.json()

for repo in repos:

if repo["language"] == "Python":

print(f"Name: {repo['name']}\nLink: {repo['html\_url']}\n")

**Отправка запросов с параметрами**

Для отправки запроса с параметрами используется параметр

params

 в методе

requests.get

:

import requests

url = "https://api.apilayer.com/exchangerates\_data/convert"

payload = {

"amount": "1200",

"from": "EUR",

"to": "USD"

}

headers = {

"apikey": "fSHSoKuZ2zVFxstaVw1stEq3GIFqPptc"

}

response = requests.get(url, headers=headers, params=payload)

status\_code = response.status\_code

result = response.json()

print(status\_code)

print(result)

>>> 200

{'success': True, 'query': {'from': 'EUR', 'to': 'USD', 'amount': 1200}, 'info': {'timestamp': 1715103603, 'rate': 1.076281}, 'date': '2024-05-07', 'result': 1291.5372}

Отправка GET-запроса с использованием параметров

amount

,

from

 и

to

 с помощью параметра

params

 эквивалентна отправке GET-запроса на URL-адрес, содержащий эти параметры в URL-адресе. Например, URL-адрес

https://api.apilayer.com/exchangerates\_data/convert?to=USD&from=EUR&amount=1200

 в данном случае будет эквивалентен запросу, отправленному с использованием параметра

params

.

# 8. Ошибки при работе с библиотекой requests

Библиотека

requests

 используется для работы с HTTP-запросами. Однако при выполнении запросов могут возникать различные ошибки. Рассмотрим некоторые из них.

## ConnectionError

Эта ошибка возникает, когда запрос не может быть выполнен из-за проблем с сетью. Например, при сбое DNS, отказе соединения и т. д.

Чтобы обработать ошибку

ConnectionError

, нужно использовать следующий код:

import requests

try:

response = requests.get('http://example.com')

except requests.exceptions.ConnectionError:

print("Connection Error. Please check your network connection.")

## HTTPError

Эта ошибка возникает, если полученный ответ от сервера не является корректным HTTP-ответом.

Используйте следующий код, чтобы обработать ошибку

HTTPError

:

import requests

try:

response = requests.get('http://example.com')

response.raise\_for\_status()

except requests.exceptions.HTTPError:

print("HTTP Error. Please check the URL.")

## Timeout

Эта ошибка возникает, если запрос не получил ответа в течение заданного времени.

Чтобы обработать ошибку

Timeout

, используйте следующий код:

import requests

try:

response = requests.get('http://example.com', timeout=5)

except requests.exceptions.Timeout:

print("Request timed out. Please check your internet connection.")

## TooManyRedirects

Эта ошибка возникает, если количество перенаправлений запроса превышает максимально допустимое значение.

Чтобы обработать эту ошибку, нужно использовать следующий код:

import requests

try:

response = requests.get('http://example.com', allow\_redirects=False)

except requests.exceptions.TooManyRedirects:

print("Too many redirects. Please check the URL.")

## RequestException

Это базовый класс для всех исключений, которые может выбросить библиотека

requests

.

Чтобы обработать все возможные исключения одновременно, используйте следующий код:

import requests

try:

response = requests.get('http://example.com')

response.raise\_for\_status()

except requests.exceptions.RequestException:

print("An error occurred. Please try again later.")

Эти ошибки не охватывают все возможные исключения, но они являются наиболее распространенными. Каждая ошибка должна обрабатываться соответствующим образом, чтобы обеспечить более надежную работу программы.

# 9. Переменные окружения: файл .env

Когда вы работаете с API, часто приходится хранить конфиденциальные данные, такие как токены доступа, пароли и ключи API, в коде. Это может быть небезопасно, так как злоумышленник может получить доступ к коду и использовать эти данные.

Решение проблемы — применение

.env

-файлов вместе с библиотекой

python-dotenv

.

**Файлы с расширением**

.env

**(от англ. environment — «среда»)** — используются для хранения переменных окружения, которые содержат конфигурационные параметры, чувствительные и секретные данные, необходимые для работы приложений.

**Переменные окружения** — это именованные переменные, содержащие текстовую информацию, которую могут использовать запускаемые программы.

Переменные окружения удобнее всего получать из

.env

-файлов. Рассмотрим, как использовать

.env

-файлы для хранения конфиденциальных данных и как использовать

python-dotenv

 для чтения этих данных в программе.

**Установка библиотеки python-dotenv**

Библиотеку

python-dotenv

 можно установить с помощью

poetry

:

poetry add python-dotenv

**Создание .env-файла**

Создайте файл с именем

.env

 в корневом каталоге вашего проекта. В этом файле вы можете хранить ваши конфиденциальные данные в формате

KEY=VALUE

.

Например:

API\_KEY=abc123def456

PASSWORD=mysecretpassword

GITHUB\_TOKEN=your\_personal\_access\_token\_here

Файл с расширением

.env

 всегда должен быть в

.gitignore

-файле.

**Использование .env-файла**

Ниже приведен пример того, как вы можете использовать библиотеку

python-dotenv

 вместе с библиотекой

requests

 для получения доступа к API:

import os

from dotenv import load\_dotenv

import requests

# Загрузка переменных из .env-файла

load\_dotenv()

# Получение значения переменной GITHUB\_TOKEN из .env-файла

github\_token = os.getenv('GITHUB\_TOKEN')

# Создание заголовка с токеном доступа API

headers = {

'Authorization': f'token {github\_token}'

}

# Отправка GET-запроса к API

response = requests.get('https://api.github.com/user', headers=headers)

# Обработка ответа

print(response.json())

В этом примере мы используем библиотеку

python-dotenv

, чтобы загрузить переменные из

.env

-файла, а затем используем метод

os.getenv()

 для получения значения переменной

GITHUB\_TOKEN

. Затем мы создаем заголовок с токеном доступа API и отправляем GET-запрос к API, используя библиотеку

requests

.

Использование

.env

-файлов и библиотеки

python-dotenv

 — это простой и безопасный способ хранения конфиденциальных данных при работе с API в вашей программе. Этот метод позволяет хранить конфиденциальные данные отдельно от кода и защищать их от несанкционированного доступа.

**Шаблон файла .env**

Чтобы все разработчики в команде имели представление о том, какие переменные окружения необходимы для работы проекта и как они должны быть организованы, создание шаблона

.env

-файла (часто называемого

.env.example

 или

.env.sample

) является важной частью работы над проектом. Такой файл должен быть включен в отслеживаемые Git файлы для облегчения настройки проекта.

Вот как можно организовать файл

.env.example

:

# Создайте файл .env из копии этого файла и замените значения переменных реальными данными

# Конфигурация базы данных

DATABASE\_HOST=localhost # Хост для базы данных

DATABASE\_USER=username # Имя пользователя базы данных

DATABASE\_PASSWORD=password # Пароль базы данных

DATABASE\_NAME=mydatabase # Имя базы данных

# API-ключи

API\_KEY=your\_api\_key\_here

GITHUB\_TOKEN=your\_github\_token\_here

В этом файле образца:

* Каждая переменная окружения указана с example-значениями или подсказками для заполнения

your\_value\_here

, чтобы указать, что значение должно быть заменено на актуальное.

* Есть комментарии, которые поясняют, для чего каждая переменная используется.

Не забудьте положить файл

.env.example

 в ваш Git-репозиторий.

**Как использовать шаблон .env-файла**

При клонировании репозитория новый разработчик копирует файл

.env.example

, переименовывает его в

.env

 и заменяет значения переменных реальными значениями, необходимыми для работы приложения в соответствующей среде.

Это позволяет новым участникам проекта быстро настроить необходимое окружение, при этом не угадывать требуемые переменные и не искать их у коллег.

Подход с использованием

.env.example

 является лучшей практикой для обеспечения безопасности и упрощения процесса настройки проекта, а также способствует единообразию конфигурационного процесса в команде разработчиков.

# 10. Тестирование с Mock и patch

Когда мы пишем код программы, то часто используем разные функции и объекты, которые не принадлежат нашему коду, а берутся из других мест. Например, мы можем использовать функцию

requests.get

, которая отправляет запрос в интернет и получает ответ от какого-то сайта или сервиса.

Но иногда мы не хотим или не можем использовать реальные функции и объекты, которые работают с API, потому что они могут быть медленными, ненадежными, дорогими или недоступными. Например, мы не хотим отправлять слишком много запросов в интернет, потому что это может занять много времени и трафика, или мы не можем получить ответ от сайта, потому что он не работает или заблокирован. В таких случаях мы можем использовать заглушки.

**Заглушки** — это специальные функции и объекты, которые имитируют реальные, но возвращают заранее заданные значения, которые мы сами выбираем. Заглушки помогают нам тестировать код и не зависеть от реальных условий.

Для создания и использования заглушек в Python есть два специальных инструмента —

Mock

 и

patch

. Они позволяют легко заменять реальные функции и объекты на заглушки и проверять, как код работает с ними.

**Mock**

Mock

 — это специальный объект, который может «притворяться» любым другим объектом.

Вы можете настроить его так, чтобы он возвращал любые значения, которые хотите, когда вызываете его или его методы. Вы также можете проверить, сколько раз он был вызван и с какими аргументами.

Рассмотрим пример. У нас есть функция, которая возвращает случайное число от 0 до 10:

import random

def get\_random\_number():

return random.randint(0, 10)

Мы хотим протестировать эту функцию, но мы не хотим использовать реальную функцию

random.randint

, потому что она каждый раз возвращает разные значения, и мы не можем предсказать, что она вернет. Вместо этого используем

Mock

, чтобы заменить функцию

random.randint

 на заглушку, которая всегда возвращает одно и то же число, например 5. Тогда мы можем проверить, что функция

get\_random\_number

 тоже возвращает 5.

Для этого напишем такой код теста:

from unittest.mock import Mock

def test\_get\_random\_number():

mock\_random = Mock(return\_value=5)

random.randint = mock\_random

assert get\_random\_number() == 5

mock\_random.assert\_called\_once\_with(0, 10)

В этом коде мы делаем следующее:

* Импортируем класс

Mock

 из модуля

unittest.mock

. Это стандартный модуль Python, который предоставляет инструменты для тестирования.

* Создаем объект

Mock

, который будет возвращать 5, когда мы его вызываем. Для этого используем параметр

return\_value

, который указывает, что вернуть при вызове объекта

Mock

.

* Заменяем функцию

random.randint

 на объект

Mock

. Теперь, когда мы вызываем функцию

random.randint

, мы на самом деле вызываем объект

Mock

, который возвращает 5.

* Вызываем функцию

get\_random\_number

 и проверяем, что она возвращает 5. Для этого используем оператор

assert

, который проверяет, что условие истинно. Если условие ложно, то тест не проходит, и мы получаем ошибку.

* Проверяем, что объект

Mock

 был вызван один раз с аргументами 0 и 10. Для этого мы используем метод

assert\_called\_once\_with

, который проверяет, что объект

Mock

 был вызван только один раз и с определенными аргументами. Если это не так, то тест не проходит, и мы получим ошибку.

Таким образом, мы можем использовать Mock, чтобы заменить реальную функцию на заглушку и проверить, как код работает с ней.

**Основные методы Mock**

* assert\_called()

 — проверяет, была ли функция вызвана хотя бы раз.

* assert\_called\_once()

 — проверяет, была ли функция вызвана только один раз.

* assert\_called\_with(args, kwargs)

 — проверяет, была ли функция вызвана с определенными аргументами.

* return\_value

 — устанавливает значение, которое будет возвращаться при вызове функции.

**patch**

patch

 — это специальный декоратор, который позволяет нам заменить реальный объект или функцию на заглушку во время выполнения теста.

Напомним, что декоратор — это функция, которая принимает другую функцию в качестве аргумента и возвращает новую функцию, которая делает что-то дополнительное перед или после вызова исходной функции.

patch

 принимает в качестве аргумента имя объекта или функции, которую мы хотим заменить, и возвращает объект

Mock

, который мы можем настроить так, как нам нужно.

patch

 заменяет объект или функцию только во время выполнения теста, а потом возвращает его в исходное состояние.

Для примера возьмем ту же функцию, которая возвращает случайное число от 0 до 10, и протестируем ее с помощью

patch

:

from unittest.mock import patch

import random

def get\_random\_number():

return random.randint(0, 10)

@patch('random.randint')

def test\_get\_random\_number(mock\_random):

mock\_random.return\_value = 5

assert get\_random\_number() == 5

mock\_random.assert\_called\_once\_with(0, 10)

В этом коде мы делаем следующее:

* Импортируем функцию

patch

 из модуля

unittest.mock

.

* Реализуем функцию

get\_random\_number

, которая возвращает случайное число от 0 до 10.

* Добавляем декоратор

@patch('random.randint')

 перед функцией теста. Это означает, что во время выполнения теста функция

random.randint

 будет заменена на объект

Mock

, который будет передан в нашу функцию теста в качестве аргумента

mock\_random

.

* Настраиваем объект

Mock

, чтобы он возвращал 5, когда мы его вызываем. Для этого используем атрибут

return\_value

, который указывает, что вернуть при вызове объекта

Mock

.

* Вызываем функцию

get\_random\_number

 и проверяем, что она возвращает 5. Для этого используем оператор

assert

, который проверяет, что условие истинно. Если условие ложно, то тест не проходит, и мы получаем ошибку.

* Проверяем, что объект

Mock

 был вызван один раз с аргументами 0 и 10. Для этого используем метод

assert\_called\_once\_with

, который проверяет, что объект

Mock

 был вызван только один раз и с определенными аргументами. Если это не так, то тест не проходит, и мы получаем ошибку.

* После выполнения теста функция

random.randint

 будет возвращена в исходное состояние, и мы снова сможем использовать ее как обычно.

Таким образом, мы можем использовать

patch

, чтобы заменить реальную функцию на заглушку во время выполнения теста и проверить, как код работает с ней.

**Примеры тестирования функций, которые обращаются к API**

Для тестирования функций, которые используют API, мы можем использовать

Mock

 и

patch

 для создания заглушек, которые имитируют ответы от API. Это позволяет тестировать код, не зависеть от реального API и убедиться, что код обрабатывает ответы правильно.

Например, для тестирования функции, которая получает информацию о пользователе GitHub, используем следующий код:

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

@patch('requests.get')

def test\_get\_github\_user\_info(mock\_get):

mock\_get.return\_value.json.return\_value = {'login': 'testuser', 'name': 'Test User'}

assert get\_github\_user\_info('testuser') == {'login': 'testuser', 'name': 'Test User'}

mock\_get.assert\_called\_once\_with('https://api.github.com/users/testuser')

В этом примере мы используем

patch

 для замены функции

requests.get

, которая обычно выполняет запрос к API. Затем мы устанавливаем значение, которое должна вернуть функция

json

, и вызываем функцию

get\_github\_user\_info

. Проверяем, что функция возвращает ожидаемое значение и что функция

requests.get

 была вызвана только один раз с правильным URL.

Разберем подробно шаги, которые выполняются в этом примере:

1. Импортируем функцию

patch

 из модуля

unittest.mock

. Эта функция позволяет нам заменить реальный объект или функцию на заглушку во время выполнения теста.

from unittest.mock import patch

1. Импортируем модуль

requests

, который позволяет нам отправлять запросы в интернет и получать ответы от сайтов и сервисов.

from unittest.mock import patch

import requests

1. Определяем функцию

get\_github\_user\_info

, которая принимает один аргумент —

username

, то есть имя пользователя на GitHub. Эта функция возвращает информацию о пользователе GitHub в виде словаря, который содержит разные данные, такие как логин, имя, количество подписчиков и т. д.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

pass

1. Составляем URL для запроса к API GitHub, используя форматирование строк с помощью f-строки и фигурных скобок. В URL мы подставляем имя пользователя, которое передали в функцию. URL имеет вид

https://api.github.com/users/username

, где

username

 — это имя пользователя на GitHub.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

1. Вызываем метод

json

 у объекта

response

, который возвращает ответ от API в виде словаря. Сохраняем этот словарь в переменную и возвращаем его из функции с помощью ключевого слова

return

.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

1. Добавляем декоратор

@patch('requests.get')

 перед функцией теста. Это означает, что во время выполнения теста функция

requests.get

 будет заменена на объект

Mock

, который будет передан в функцию теста в качестве аргумента

mock\_get

.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

@patch('requests.get')

1. Определяем функцию теста, которая принимает один аргумент —

mock\_get

, который является объектом

Mock

, заменяющим функцию

requests.get

.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

@patch('requests.get')

def test\_get\_github\_user\_info(mock\_get):

pass

1. Настраиваем объект

Mock

, чтобы он возвращал заранее заданный словарь, когда мы вызываем метод

json

 у него. Для этого используем атрибут

return\_value

, который указывает, что вернуть при вызове объекта

Mock

, и атрибут

json

, который указывает, что вернуть при вызове метода

json

 у объекта

Mock

. В нашем случае мы возвращаем словарь, который содержит логин и имя пользователя

testuser

 на GitHub.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

@patch('requests.get')

def test\_get\_github\_user\_info(mock\_get):

mock\_get.return\_value.json.return\_value = {'login': 'testuser', 'name': 'Test User'}

1. Вызываем функцию

get\_github\_user\_info

 с аргументом

testuser

 и проверяем, что она возвращает словарь, который мы задали выше. Для этого используем оператор

assert

, который проверяет, что условие истинно. Если условие ложно, то тест не проходит, и мы получаем ошибку.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

@patch('requests.get')

def test\_get\_github\_user\_info(mock\_get):

mock\_get.return\_value.json.return\_value = {'login': 'testuser', 'name': 'Test User'}

assert get\_github\_user\_info('testuser') == {'login': 'testuser', 'name': 'Test User'}

1. Проверяем, что объект

Mock

 был вызван один раз с правильным URL в качестве аргумента. Для этого используем метод

assert\_called\_once\_with

, который проверяет, что объект Mock был вызван только один раз и с определенным аргументом. Если это не так, то тест не проходит, и мы получаем ошибку.

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

@patch('requests.get')

def test\_get\_github\_user\_info(mock\_get):

mock\_get.return\_value.json.return\_value = {'login': 'testuser', 'name': 'Test User'}

assert get\_github\_user\_info('testuser') == {'login': 'testuser', 'name': 'Test User'}

mock\_get.assert\_called\_once\_with('https://api.github.com/users/testuser')

**Использование patch с контекстным менеджером**

patch

 можно использовать не только как декоратор, но и с контекстным менеджером. Контекстный менеджер позволяет заменить объект или функцию только в определенном блоке кода, а затем вернуться к реальному объекту или функции.

Например, мы можем использовать

patch

 с контекстным менеджером для тестирования функций, которые обращаются к API. Возьмем функцию, которая получает информацию о пользователе GitHub:

from unittest.mock import patch

import requests

def get\_github\_user\_info(username):

response = requests.get(f'https://api.github.com/users/{username}')

return response.json()

def test\_get\_github\_user\_info():

with patch('requests.get') as mock\_get:

mock\_get.return\_value.json.return\_value = {'login': 'testuser', 'name': 'Test User'}

assert get\_github\_user\_info('testuser') == {'login': 'testuser', 'name': 'Test User'}

mock\_get.assert\_called\_once\_with('https://api.github.com/users/testuser')

Мы используем

patch

 с контекстным менеджером, чтобы заменить функцию

requests.get

, которая обычно выполняет запрос к API. Затем мы устанавливаем значение, которое должна вернуть функция

json

, и вызываем нашу функцию

get\_github\_user\_info

. Мы проверяем, что функция возвращает ожидаемое значение и что функция

requests.get

 была вызвана только один раз с правильным URL.

Mock

 и

patch

 — это мощные, но сложные инструменты для тестирования кода в Python. Они позволяют создавать заглушки для объектов и функций, которые помогают тестировать код, зависящий от внешних источников данных или условий.

# 12. Библиотека datetime

datetime

 — это библиотека, которая используется для работы с датами и временем. Эта библиотека предоставляет класс

datetime

, который позволяет работать с датами и временем как с объектами.

**Создание объекта datetime**

Чтобы создать объект

datetime

, используйте следующий синтаксис:

# Импортируем библиотеку datetime

import datetime

# Создаем объект datetime

date\_obj = datetime.datetime(year, month, day, hour, minute, second)

Здесь перечислены:

* year

 — год (целое число от 1 до 9999);

* month

 — месяц (целое число от 1 до 12);

* day

 — день (целое число от 1 до 31);

* hour

 — час (целое число от 0 до 23);

* minute

 — минута (целое число от 0 до 59);

* second

 — секунда (целое число от 0 до 59).

Пример создания объекта

datetime

:

import datetime

# Создание объекта datetime для 8 марта 2022 года, 15:45:00

date\_obj = datetime.datetime(2022, 3, 8, 15, 45, 0)

print(date\_obj)

>>> 2022-03-08 15:45:00

Этот код создает объект

datetime

 для указанной даты и времени и выводит его на экран.

**Атрибуты объекта datetime**

После создания объекта

datetime

 вы можете получить значения его атрибутов, таких как год, месяц, день, час, минута и секунда. Для этого используются соответствующие атрибуты объекта:

year = date\_obj.year

month = date\_obj.month

day = date\_obj.day

hour = date\_obj.hour

minute = date\_obj.minute

second = date\_obj.second

print(year, month, day, hour, minute, second)

>>> 2022 3 8 15 45 0

Этот код выводит на экран значения атрибутов объекта

datetime

, который был создан ранее.

**Форматирование дат и времени**

Иногда необходимо преобразовать объект

datetime

 в строку определенного формата для удобного отображения. Метод

strftime()

 позволяет отформатировать дату и время в строку заданного формата:

date\_string = date\_obj.strftime("%d-%m-%Y %H:%M:%S")

В этом примере используется строка формата

"%d-%m-%Y %H:%M:%S"

, где:

* %d

 — день месяца (01–31)

* %m

 — месяц (01–12)

* %Y

 — год (четырехзначное число)

* %H

 — часы (00–23)

* %M

 — минуты (00–59)

* %S

 — секунды (00–59)

Другие форматы

Пример форматирования даты:

import datetime

date\_obj = datetime.datetime(2022, 3, 8, 15, 45, 0)

date\_string = date\_obj.strftime("%d-%m-%Y %H:%M:%S")

print(date\_string)

print(type(date\_string))

>>> 08-03-2022 15:45:00

<class 'str'>

**Создание объекта datetime из строки**

Для создания объекта

datetime

 из строки используется метод

strptime()

. Он принимает строку и формат, который соответствует этой строке:

date\_string = "08-03-2022 15:45:00"

date\_obj = datetime.datetime.strptime(date\_string, "%d-%m-%Y %H:%M:%S")

print(date\_obj)

print(type(date\_obj))

>>> 2022-03-08 15:45:00

<class 'datetime.datetime'>

Этот код преобразует строку

date\_string

 в объект

datetime

 на основе указанного формата.

**Получение текущей даты и времени**

Чтобы получить текущую дату и время, можно использовать метод

now()

 класса

datetime

:

current\_date\_time = datetime.datetime.now()

print(current\_date\_time)

>>> 2024-05-14 16:35:27.169864

Этот код создает объект

datetime

, содержащий текущую дату и время, и выводит его на экран.

**Изменение даты и времени**

Для изменения даты и времени в объекте

datetime

 используются методы

replace()

 и

timedelta()

. Метод

replace()

 позволяет изменить определенные атрибуты даты и времени, а

timedelta()

 — добавить или вычесть временной интервал.

Пример использования метода

replace()

:

import datetime

date\_obj = datetime.datetime(2022, 3, 8)

new\_date\_obj = date\_obj.replace(year=2023, month=3, day=8)

print(new\_date\_obj)

>>> 2023-03-08 00:00:00

Пример использования метода

timedelta()

:

import datetime

time\_obj = datetime.datetime(2022, 3, 8, 15, 45, 0)

new\_time\_obj = time\_obj + datetime.timedelta(hours=1, minutes=30)

print(new\_time\_obj)

>>> 2022-03-08 17:15:00

**Примеры работы с библиотекой datetime**

1. **Пример работы с датой:**

import datetime

# Создание объекта datetime

date\_obj = datetime.datetime(2022, 3, 8)

print(date\_obj.year) # 2022

print(date\_obj.month) # 3

print(date\_obj.day) # 8

# Форматирование даты в строку

date\_string = date\_obj.strftime("%d-%m-%Y")

print(date\_string) # 08-03-2022

# Изменение даты

new\_date\_obj = date\_obj.replace(year=2023, month=3, day=8)

print(new\_date\_obj.strftime("%d-%m-%Y")) # 08-03-2023

1. **Пример работы со временем:**

import datetime

# Создание объекта datetime со временем

time\_obj = datetime.datetime(2022, 3, 8, 15, 45, 0)

print(time\_obj.hour) # 15

print(time\_obj.minute) # 45

print(time\_obj.second) # 0

# Форматирование времени в строку

time\_string = time\_obj.strftime("%H:%M:%S")

print(time\_string) # 15:45:00

# Изменение времени

new\_time\_obj = time\_obj + datetime.timedelta(hours=1, minutes=30)

print(new\_time\_obj.strftime("%H:%M:%S")) # 17:15:00

С помощью библиотеки

datetime

 можно легко создавать, изменять и форматировать даты и время, что делает эту библиотеку незаменимым инструментом для разработки приложений, работающих с временными данными.