

Практическая работа 12

Получение данных пользователя

Поскольку использовать Github API v3 довольно просто, вы можете сделать простой запрос GET на определенный URL-адрес и получить результаты:

```
import requests
from pprint import pprint
# Имя пользователя github
username = "kubernetes"
# url для запроса
url = f"https://api.github.com/users/{username}"
# делаем запрос и возвращаем json
user_data = requests.get(url).json()
# довольно распечатать данные JSON
pprint(user_data)
```

Использовал самую популярную учетную запись, вот часть возвращенного JSON :

```
{ 'avatar_url': 'https://avatars.githubusercontent.com/u/13629408?v=4',
  'bio': 'Kubernetes',
  'blog': 'https://kubernetes.io',
  'company': None,
  'created_at': '2015-08-03T17:55:43Z',
  'email': None,
  'events_url': 'https://api.github.com/users/kubernetes/events{/privacy}',
  'followers': 0,
  'followers_url': 'https://api.github.com/users/kubernetes/followers',
  'following': 0,
  'following_url': 'https://api.github.com/users/kubernetes/following{/other_user}',
  'gists_url': 'https://api.github.com/users/kubernetes/gists{/gist_id}',
  'gravatar_id': '',
  'hireable': None,
  'html_url': 'https://github.com/kubernetes',
  'id': 13629408,
  'location': None,
  'login': 'kubernetes',
  'name': 'Kubernetes',
  'node_id': 'MDEyOk9yZ2FuaXphdGlvbjEzNjI5NDA4',
  'organizations_url': 'https://api.github.com/users/kubernetes/orgs',
  'public_gists': 0,
  'public_repos': 75,
  'received_events_url': 'https://api.github.com/users/kubernetes/received_events',
  'repos_url': 'https://api.github.com/users/kubernetes/repos',
  'site_admin': False,
  'starred_url': 'https://api.github.com/users/kubernetes/starred{/owner}{/repo}',
  'subscriptions_url': 'https://api.github.com/users/kubernetes/subscriptions',
  'twitter_username': 'kubernetesio',
  'type': 'Organization',
  'updated_at': '2021-11-09T15:39:50Z',
  'url': 'https://api.github.com/users/kubernetes'}
```

Python поддерживает JSON

Изначально Python поставляется со стандартным (встроенным) модулем `json` для кодирования и декодирования данных в формате JSON. Для этого просто вставьте в начале вашего файла с кодом программы следующие инструкции:

```
import json
```

Основные термины

Процесс кодирования JSON называется **сериализацией** (serialization). Этот термин обозначает преобразование данных в линейную последовательность байтов для хранения на диске или передачи по сети. Интересуясь материалами по этой тематике, вы также могли слышать термин «маршалинг» (marshaling).

Соответственно, **десериализация** (deserialization) является обратным процессом, а технически декодированием данных из формата JSON в структуру данных в памяти.

На самом деле проще думать об этих двух взаимообратимых процессах как об обычном чтении и записи данных: кодирование предназначено для записи данных на диск (или передачи по сети), а декодирование — для чтения данных в память и последующей обработки.

Сериализация JSON

Модуль `json` предоставляет удобный метод `dump()` для записи данных в файл. Существует также метод `dumps()` для записи данных в обычную строку. Типы данных Python кодируются в формат JSON в соответствии с интуитивно понятными правилами преобразования, представленными в виде таблице ниже.

Python	JSON
<code>dict</code>	<code>object</code>
<code>list, tuple</code>	<code>array</code>
<code>str</code>	<code>string</code>
<code>int, long, float</code>	<code>number</code>
<code>True</code>	<code>true</code>
<code>False</code>	<code>false</code>
<code>None</code>	<code>null</code>

Пример простой процедуры сериализации данных

Теперь представим, что мы работаем в памяти с объектом следующего вида:

```
data = {  
    "president": {  
        "name": "Zaphod Beeblebrox",  
        "species": "Betelgeusian"  
    }  
}
```

Нам необходимо сохранить эту информацию на диске, то есть записать ее в файл. Используя диспетчер контекстов Python сначала создадим файл, например, с именем `data_file.json`, а затем откроем его в режиме записи:

```
with open("data_file.json", "w") as write_file:  
    json.dump(data, write_file)
```

Обратите внимание, на то что метод `dump()` принимает два аргумента: объект данных, подлежащий сериализации и файлоподобный объект, в который они затем будут записаны после кодирования.

Если вы захотите далее использовать сериализованные данные в своем приложении, вы можете записать их в обычную строку типа `str`. Для этого используйте следующий код:

```
json_string = json.dumps(data)
```

Обратите внимание, второй аргумент который содержит ссылку на файлоподобный объект для записи в коде выше отсутствует, так данные не записываются на диск, а сохраняются в переменной `json_string`. Кроме этой особенности, во всем остальном метод `dumps()` аналогичен `dump()`.

Некоторые полезные именованные аргументы

Напомним JSON должен быть легко читаем для людей. Но что если наши данные будут упакованы в одну строку без отступов и разделения по отдельным строкам. Кроме всего этого, у вас вероятно имеется свой стиль форматирования (`styleguide`) или же вам проще читать код отформатированный по вашим правилам.

ПРИМЕЧАНИЕ. Оба метода `dump()` и `dumps()` используют одни и те же именованные аргументы.

Первая опция, которую большинство людей хочет изменить — это количество пробельных символов в отступе. Вы можете использовать именованный аргумент `indent`, для того чтобы указать размер отступа во вложенных структурах. Используя данные, которые мы передали переменной `data`, выполните следующие команды в консоли, а затем сравните результат выполнения обеих инструкций:

```
>>> json.dumps(data)
>>> json.dumps(data, indent=4)
```

Другая популярная опция для изменения стиля форматирования — использование именованного аргумента `separators`. По умолчанию в качестве разделителей в файлах JSON используется строка, состоящая из двух символов: «, » или «: » (символ + символ *пробел*). Альтернативным способом придания файлу более компактного вида является использование разделителей в виде строк вида: «,» и «:» (без *пробела* в конце). Выполнив в качестве примера в консоли команды, приведенные выше, и задав новое значение аргумента `separators`, можно заметить, как вид разделителей изменит форматирование ваших данных.

Десериализация JSON

В модуле `json` определены методы `load()` и `loads()`, предназначенные для преобразования кодированных в формате JSON данных в объекты Python. Подобно операции *серIALIZАЦИИ*, также существует таблица преобразования типов, определяющая правила для обратного декодирования данных. Хотя вероятно вы уже наверное догадались, как она будет выглядеть:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Технически это преобразование не является в точности обратным к таблице для *серIALIZАЦИИ* данных, рассмотренной нами выше. Это означает, что если вы кодируете объект в формат JSON, а затем декодируете его обратно, то вы можете получить уже не тот объект, каким он был изначально. Простым иллюстрирующим этот факт примером будет кодирование данных с типа кортеж `tuple` и получение после декодирования данных с типа список `list`:

```
>>> blackjack_hand = (8, "Q")
>>> encoded_hand = json.dumps(blackjack_hand)
>>> decoded_hand = json.loads(encoded_hand)
>>> blackjack_hand == decoded_hand
False
>>> type(blackjack_hand)
<class 'tuple'>
>>> type(decoded_hand)
<class 'list'>
>>> blackjack_hand == tuple(decoded_hand)
True
```

Простой пример десериализации данных

Представим теперь, что у вас есть данные, хранящиеся на диске в виде файла, которые вы хотели бы обрабатывать в памяти. Как в задаче выше вы также можете использовать диспетчер контекста, но на этот раз для того, чтобы открыть существующий файл `data_file.json` в режиме чтения:

```
with open("data_file.json", "r") as read_file:
    data = json.load(read_file)
```

Здесь все довольно просто, но имейте в виду, что результат выполнения этого кода будет возвращать результат декодирования, в соответствии с нашей таблицей преобразования типов данных. Об этом важно помнить если вы загружаете из файла данные, состав которых вам заранее неизвестен.

В большинстве случаев корневой объект будет представлять собой объект типа словарь `dict` или список `list`. Допустим, что вы получаете данные в формате JSON из другой программы или ваш код Python должен обработать строку данных типа `str` в формате JSON. В этом случае вы можете легко десериализовать их с помощью метода `loads()`. В приведенном ниже коде, данные просто загружаются из строки и затем декодируются:

```
json_string = """
{
    "researcher": {
        "name": "Ford Prefect",
        "species": "Betelgeusian",
        "relatives": [
            {
                "name": "Zaphod Beeblebrox",
                "species": "Betelgeusian"
            }
        ]
    }
}"""
```

....

```
data = json.loads(json_string)
```

Пример (как бы) из реальной жизни

Для демонстрации нашего «реального» примера мы будем использовать online-сервис JSONPlaceholder. Он представляет собой удаленный источник данных в формате JSON, получаемых по сети по запросу, и могут использоваться для отладки приложений. Вначале создадим файл сценария с именем scratch.py или под любым другим именем. Нам необходимо будет сформировать запрос request к служебному API JSONPlaceholder, для этого мы будем использовать модуль requests. Просто добавьте инструкции импорта в начало файла:

```
import json
import requests
```

Запросим у JSONPlaceholder список задач TODO, обращаясь через интерфейс его API, относительно входной точки /todos. Если вы не знакомы с модулем requests, вы можете использовать другой удобный метод json(), который выполнит эту же задачу. В нашем же примере мы будем использовать модуль json для десериализации атрибута text объекта ответа response, полученного с помощью модуля requests. Код нашего примера будет выглядеть следующим образом:

```
response = requests.get("https://jsonplaceholder.typicode.com/todos")
todos = json.loads(response.text)
```

Запустите файл в интерактивном режиме с помощью командной строки. Сделав это, проверьте тип объекта todos, а также содержимое элементов списка значений.

```
>>> todos == response.json()
True
>>> type(todos)
<class 'list'>
>>> todos[:10]
```

JSONPlaceholder генерирует набор данных содержащий: список пользователей, каждый из которых имеет уникальный идентификатор userId, а также поле completed (статус задачи) с типом Boolean. Как определить какие пользователи выполнили наибольшее количество задач? Представленный ниже код поможет определить это:

```
# таблица userId пользователей полностью выполнивших все задачи из TODO
todos_by_user = {}
# Подсчет количества задач из списка TODO выполненных каждым пользователем
```

```

for todo in todos:
    if todo["completed"]:
        try:
            # Суммируем количество выполненных пользователем задач.
            todos_by_user[todo["userId"]] += 1
        except KeyError:
            # Этот пользователь ничего не сделал. Зададим количество выполненных задач равным 1.
            todos_by_user[todo["userId"]] = 1

        # Создадим отсортированный список пар значений (userId, num_complete)
        top_users = sorted(todos_by_user.items(),
                           key=lambda x: x[1], reverse=True)
        # Зададим максимальное количество выполненных задач TODO в списке
        max_complete = top_users[0][1]

        # Создадим список всех пользователей, которые имеют максимальное количество выполненных задач из списка
        TODO
        users = []
        for user, num_complete in top_users:
            if num_complete < max_complete:
                break
            users.append(str(user))
        max_users = " and ".join(users)

```

Теперь мы можем манипулировать данными прочитанными из файла в формате JSON и декодированными как с обычным объектом Python. Если мы запустим следующие инструкции в консоли, то получим:

```

>>> s = "s" if len(users) > 1 else ""
>>> print(f"user{s} {max_users} completed {max_complete} TODOs")
users 5 and 10 completed 12 TODOs
# пользователи 5 и 10 выполнили 12 задач из TODO

```

Далее создадим файл JSON, который будет содержать заполненные списки задач TODO для каждого из пользователей, которые завершили максимальное количество задач из списка. Все, что теперь нужно сделать – отфильтровать задачи todos и записать полученный список в файл. Назовём файл с результатами обработки данных filter_data_file.json. Существует несколько способов, которыми можно это сделать. Ниже приведен код **одного из них**:

```

# Определим функцию для фильтрации списка пользователей,
# выполнивших максимальное количество заданий из TODO
def keep(todo):
    is_complete = todo["completed"]
    has_max_count = todo["userId"] in users
    return is_complete and has_max_count

# Запишем отфильтрованные данные в файл
with open("filtered_data_file.json", "w") as data_file:

```

```
filtered.todos = list(filter(keep, todos))
json.dump(filtered.todos, data_file, indent=2)
```

Отлично, мы сохранили нужные нам данные в файл, отфильтровав все лишнее. Запустите сценарий еще раз и проверьте файл filter_data_file.json, чтобы убедиться, что все работает так как нужно. Он будет создан в том же каталоге, что и файл scratch.py.

Далее изучаем : <https://pythonist.ru/format-dannyh-json-v-python/>

Задание: даны самые популярные репозитории на github

<https://habr.com/ru/post/453444/>, по последней цифре зачетки получить JSON для вашего варианта .

Программа с графическим интерфейсом вводим в поле имя репозитория и по нажатию кнопки получаем результат.

Необходимо получить в новый файл следующую информацию:

```
'company': None,
'created_at': '2015-08-03T17:55:43Z',
'email': None,
'id': 13629408,
'name': 'Kubernetes',
'url': 'https://api.github.com/users/kubernetes'}
```

Все прикрепить одним архивом.