JSON (JavaScript Object Notation)

— текстовый формат обмена данными, основанный на JavaScript. Но при этом он может использоваться в любом языке программирования. Формат был разработан Дугласом Крокфордом.
JSON используется в REST API. Также в качестве альтернативы можно использовать XML, но разработчики больше предпочитают именно JSON, так как он более читабельный и меньше весит.

JSON основан на JavaScript, но его понимают и другие языки программирования. Данный формат проще читать, он меньше весит, быстрее передается и совместим с разными платформами.
JSON состоит из объектов, ключей и значений. При этом файл сам является основным объектом, внутри которого могут быть другие объекты.
Перед объектом ставятся фигурные скобки {}. Далее идет имя ключа в кавычках “”, двоеточие и значение.

```
Объект{“ключ”: значение}
```

Значения могут содержать числа, строки, массивы. Также бывают пустые и булевые значения.

Пример синтаксиса JSON:

```
{
   "name": "Alice",            // Строка
   "age": 25,                 // Число
   "isStudent": false,          // Булево значение
   "courses": ["Math", "Physics"], // Массив строк
   "address": {               // Объект
     "city": "New York",     "zip": "10001"   },
   "graduationYear": null       // Пустое значение
}
```

В качестве значений в JSON могут быть использоваться:

- числа;
- строки;
- массивы;
- JSON-объекты;
- литералы (логические значения true, false и null).

С простыми значениями не возникнет никаких трудностей. Разберём массивы и JSON-объекты, ведь, по сути, придётся работать именно с ними.

JSON-объект — это неупорядоченное множество пар «ключ:значение», заключённых в фигурные скобки { } и взаимодействие с ним проходит, как со словарем.
Ключ — это название параметра (свойства), который мы передаём серверу. Он служит маркером для принимающей запрос системы, чтобы она поняла, что мы ей отправили.
Ключ — ВСЕГДА строка, и мы в любом случае берём его в кавычки.

Ключи могут быть записаны в любом порядке, ведь, JSON-объект — это неупорядоченное множество пар «ключ:значение».

JSON-массив
Массив заключен в квадратные скобки `[ ]`.

```
["MALE", "FEMALE"]
```

Внутри квадратных скобок идет набор значений, разделённых запятыми. Здесь нет ключей, как в объекте, поэтому обращаться к массиву можно только по номеру элемента. И поэтому в случае массива менять местами данные внутри нельзя. Это упорядоченное множество значений, так что порядок важен.

Well Formed JSON

JSON должен быть `well formed`, то есть синтаксически правильный.
Правила `well formed` JSON:

1. Данные написаны в виде пар «ключ:значение»
2. Данные разделены запятыми
3. Объект находится внутри фигурных скобок `{ }`
4. Массив — внутри квадратных `[ ]`

Чтобы проверить JSON на синтаксис, можно использовать любой JSON Validator. Можно порекомендовать JSON Formatter, он не только проверяет корректность синтаксиса, но и форматирует JSON в читабельный визуальный формат!

JSON Schema: что это и зачем нужно?

JSON Schema — это спецификация для описания структуры JSON-документов. Представь себе это как строгий контракт между клиентом и сервером. Как только ты начинаешь разрабатывать API, в один момент наступает полный бардак, если нет четких правил, что можно, а что нельзя отправлять в запросах или получать в ответах.

JSON Schema позволяет описать:

- Формат данных (числа, строки, объекты, массивы).
- Допустимые значения (минимумы, максимумы).
- Обязательные поля и необязательные.
- И много других классных вещей, которые спасут тебе кучу времени на дебаг.

Простой пример JSON Schema

Сразу к практике. Допустим, нужно описать JSON для пользователя. В JSON будут имя, возраст и email. Начнем с простого:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "integer",
      "minimum": 18
    },
    "email": {
      "type": "string",
      "format": "email"
    }
  },
  "required": ["name", "age", "email"]
}
```

Что тут происходит?

1. **$schema** — это версия спецификации, которой ты следуешь (мы используем draft-07).
2. **type** — указывает на тип данных. В нашем случае это объект.
3. **properties** — это описание полей, где указываются их типы и дополнительные ограничения. Например, возраст (age) должен быть целым числом и минимум 18.
4. **required** — обязательные поля, без которых наш пользователь не имеет права на существование.

Да, в этом примере мы проверяем, чтобы возраст не был меньше 18 лет. Это выглядит довольно тривиально, но под капотом JSON Schema спрятано много возможностей, о которых сейчас и поговорим.

AnyOf, AllOf, OneOf

JSON Schema — это не только про строгие рамки. Часто бывает так, что ты не знаешь точно, какой тип данных ожидается. Например, можно получать объект или строку в зависимости от контекста. Тут помогают такие штуки как anyOf, oneOf и allOf.

AnyOf: как говорить «или то, или другое»

Если нужно, чтобы одно из нескольких условий было выполнено, используй anyOf. Например, хочется, чтобы поле могло быть либо числом, либо строкой:

```
{
  "type": "object",
  "properties": {
    "price": {
      "anyOf": [
        { "type": "number" },
        { "type": "string" }
      ]
    }
  }
}
```

Поле price может быть как числом, так и строкой. Очень полезно, когда есть API, где данные могут поступать в разных форматах, но хочется оставить некоторую свободу.

OneOf: строго одно

С oneOf всё жестче — валидируется только одно из условий:

```
{
  "type": "object",
  "properties": {
    "discount": {
      "oneOf": [
        { "type": "number", "minimum": 0, "maximum": 100 },
        { "type": "boolean" }
      ]
```

```
    }
  }
}
```

Тут поле discount может быть либо процентом скидки (от 0 до 100), либо булевым значением (например, скидка включена или выключена).

AllOf: нужно всё сразу

С allOf идет требование, чтобы выполнялись все условия. Полезно, если есть сложная структура и данные должны соответствовать нескольким критериям:

```
{
  "type": "object",
  "properties": {
    "product": {
      "allOf": [
        { "type": "string" },
        { "minLength": 3 },
        { "pattern": "^[A-Z].*$" }
      ]
    }
  }
}
```

Поле product должно быть строкой, длиной не меньше 3 символов и начинаться с заглавной буквы. Да, вот такие простые требования иногда бывают в реальной жизни.

Библиотеки для работы с JSON

json

- Не нужно устанавливать, встроена по умолчанию.
- Может парсить из строк и файлов.
- Способна конвертировать объекты в JSON-строки.

SimpleJSON

- Может парсить из строк и файлов.
- Способна создавать JSON-строки из объектов.

- Предоставляет более гибкие опции кодирования и декодирования по сравнению с встроенной json.

ujson

- Может парсить из строк и файлов.
- Способна конвертировать объекты в JSON-строки.
- Отличается высокой скоростью работы благодаря реализации на C.

orjson

- Может парсить из строк и файлов.
- Способна создавать JSON-строки из объектов.
- Производительная благодаря реализации на Rust.
- Поддерживает только Python версии 3.6 и выше.

ijson

- Может парсить из строк и файлов.
- Экономит память за счет потоковой обработки данных.

jsonschema

- Специализируется на валидации.
- Поддерживает различные версии спецификации JSON Schema.

cerberus

- Специализируется на валидации.
- Поддерживает валидацию сложных вложенных структур и поддерживает пользовательские правила валидации.

Примеры извлечения JSON данных в разных библиотеках

Предположим, что нам надо спарсить вот этот код и вытащить из него значение "occupation".

```
json_string = """
{
"name": "Alice",
"age": 25,
"city": "London",
"occupation": "Software Engineer"
}
"""
```

json

Если данные уже находятся у нас в виде строки, то мы можем вызвать функцию json.loads() для извлечения:

```
import json
# Преобразуем JSON строку в словарь Python
data = json.loads(json_string)
# Выводим значение ключа "occupation"
print(data["occupation"])  # Вывод: Software Engineer
```

Здесь мы принимаем JSON и записываем его в словарь Python. Далее выводим значение ключа "occupation".

simplejson

Представим, что у нас не строка, а файл json_string.json:

```
import simplejson as json
# Открываем файл json_string.json в режиме чтения ('r')
with open('json_string.json', 'r') as f:
# Используем json.load() из SimpleJSON для чтения и парсинга JSON из
файла
  data = json.load(f)
# Выводим значение ключа "occupation"
print(data["occupation"])  # Вывод: Software Engineer
```

Мы открываем файл json_string.json и называем его f. Далее читаем JSON из файла f с помощью `json.load(f)` и превращаем в словарь data. В конце возвращаем значение, ключа "occupation".

ujson

```
import ujson
# Открываем файл json_string.json для чтения
with open('json_string.json', 'r') as f:
# Используем ujson.load() для чтения и парсинга JSON из файла
     data = ujson.load(f)  # Выводим значение ключа "occupation"
print(data["occupation"])  # Вывод: Software Engineer
```

Здесь происходит то же, что и в примере выше: `ujson.load(f)` читает JSON из файла f и преобразует его в словарь Python, который сохраняется в data.

orjson

Если мы хотим спарсить данные по url, то понадобиться библиотека requests:

```
import requests
import orjson
# Представим, что наш JSON код находится по URL
url = "https://example.com/data.json"
# Получаем данные по URL
response = requests.get(url)
# Проверяем успешность запроса
if response.status_code == 200:
     # Парсим JSON данные с помощью orjson.loads()
     data = orjson.loads(response.content)
      # Выводим значение ключа"occupation"
      print(data["occupation"])
else:
     print("Ошибка при полученииданных")
```

Функция `orjson.loads(response.content)` парсит по url JSON, и преобразует их в словарь data, после чего мы можем обращаться к значениям этого словаря.

Сериализация JSON

Что происходит после того, как компьютер обрабатывает большие объемы информации? Ему нужно принять дамп данных. Соответственно, **модуль json** предоставляет метод **dump**() для записи данных в файлы. Также есть метод **dumps**() для записей в строку Python.

Простые объекты Python переводятся в JSON согласно с весьма интуитивной конверсией.

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

Пример сериализации JSON Python

Представьте, что вы работаете с объектом Python в памяти, который выглядит следующим образом:

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

Сохранить эту информацию на диск — критично, так что ваша задача — записать на файл.

Используя контекстный менеджер Python, вы можете создать файл под названием **data_file.json** и открыть его в режиме write (файлы JSON имеют расширение **.json**).

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

Обратите внимание на то, что **dump**() принимает два позиционных аргумента: (1) объект данных, который сериализуется и (2), файловый объект, в который будут вписаны байты.

Или, если вы склонны продолжать использовать эти сериализированные данные **JSON** в вашей программе, вы можете работать как со строкой.

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

Обратите внимание, что **файловый объект** является пустым, так как вы на самом деле не выполняете запись на диск. Кроме того, **dumps**() аналогичен **dump**().

Десериализация JSON

Отлично, похоже вам удалось поймать экземпляр дикого JSON! Теперь нам нужно предать ему форму. В **модуле json** вы найдете **load**() и **loads**() для превращения кодированных данных JSON в объекты Python.

Как и сериализация, есть простая таблица конверсии для десериализации, так что вы можете иметь представление о том, как все выглядит.

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

Технически, эта конверсия не является идеальной инверсией **таблицы сериализации**. По сути, это значит что если вы кодируете объект сейчас, а затем декодируете его в будущем, вы можете не получить тот же объект назад. Я представляю это как своего рода телепортацию: мои молекулы распадаются в точке А и собираются в точке Б. Буду ли я тем же самым человеком?

В реальности, это как попросить одного друга перевести что-нибудь на японский, а потом попросить другого друга перевести это обратно на русский.

Примеры генерации JSON в разных библиотеках

Библиотеки python используют схожий подход для генерации JSON. Делается это при помощи функции dumps(). Вот несколько примеров:

SimpleJSON

```
import simplejson as json
  # Словарь с данными
 data = {
     "name": "John Doe",
     "age": 30,
     "city": "New  York" }
 # Преобразование словаря в JSON строку
 json_string = json.dumps(data)
# Вывод JSON строки
print(json_string)
```

ujson

```
import ujson
  # Словарь с данными
 data = {
     "name": "John Doe",
     "age": 30,
     "city": "New  York" }
# Преобразование словаря в JSON строку
json_string = ujson.dumps(data)
# Вывод JSON строки
print(json_string)
```

orjson

```
import orjson
    # Словарь с данными
 data = {
     "name": "John Doe",
     "age": 30,
     "city": "New  York" }
 # Преобразование словаря в JSON байты
```

```
 json_bytes = orjson.dumps(data)
  # Декодирование байтов в строку UTF-8
 json_string = json_bytes.decode("utf-8")
  # Вывод JSON строки
print(json_string)
```

Мы создали словарь data с данными и преобразовали их в JSON.