Файловое хранение данных

Введение в файловое хранение данных. Использование файлов в формате CSV при сохранении данных. Файлы JSON как средство обмена данными. Работа с YAML-файлами при обработке и сохранении данных.

[Введение в файловое хранение данных](#_z0k9oc9q84)

[Использование файлов в формате CSV при сохранении данных](#_1fob9te)

[Чтение данных из файла формата CSV](#_2et92p0)

[Запись данных в файл формата CSV](#_tyjcwt)

[Файлы JSON как средство обмена данными](#_3dy6vkm)

[Чтение JSON-файлов](#_1t3h5sf)

[Запись в JSON-файлы](#_4d34og8)

[Определение дополнительных параметров методов записи](#_2s8eyo1)

[Изменение типа данных](#_sn4y8kj34dm8)

[Ограничения на тип данных](#_3rdcrjn)

[Работа с YAML-файлами при обработке и сохранении данных](#_26in1rg)

[Синтаксис формата YAML](#_lnxbz9)

[Работа со списками](#_35nkun2)

[Работа со словарями](#_1ksv4uv)

[Работа со строками](#_44sinio)

[Комбинация элементов](#_2jxsxqh)

[Использование PyYAML](#_z337ya)

[Считывание данных](#_ow2rm7gi28jl)

[Запись данных](#_1y810tw)

[Практическое задание](#_4i7ojhp)

[Дополнительные материалы](#_2xcytpi)

[Используемая литература](#_1ci93xb)

# 

# Введение в файловое хранение данных

При сетевом взаимодействии клиентская и серверная стороны могут обмениваться данными, которые поступают от клиента на сервер, обрабатываются и возвращаются к клиенту. Пакеты данных при этом сохраняются в определенном формате, чаще всего предусматривающем структурирование. Результатом такого сохранения могут быть файлы различных форматов и базы данных. Как правило, такие файлы сохраняются в форматах CSV, JSON или YAML. Такой процесс называется сериализацией данных.

В Python можно использовать не только форматы CSV, JSON и YAML при сохранении данных, но и применять встроенные средства для записи объектов самого языка. В частности, речь идет о модуле **Pickle**. На этом уроке разберем особенности использования названных выше форматов файлов. В курсе также будет урок, демонстрирующий возможности баз данных.

На практике форматы файлов CSV, JSON и YAML можно применять при сохранении следующего рода данных:

1. Сетевых параметров — например, IP-адресов, которые необходимо сохранять в структурированном (табличном) виде. При этом таблица может быть экспортирована в CSV-формат и обработана средствами Python.
2. Результатов работы программных приложений, которые могут генерироваться в JSON-формате. После преобразования этих выходных данных в Python-объект с ними можно выполнять операции в программе.
3. Формата YAML, который можно применять при описании сетевых параметров — например, IP-адресов, VLAN и подобного.

# Использование файлов в формате CSV при сохранении данных

Аббревиатура CSV расшифровывается как «comma-separated value». Формат CSV реализует представление данных в табличном виде. При этом сохраняемые в соответствующем формате данные могут извлекаться из таблиц или баз. В этом случае отдельная строка файла соответствует строке таблицы. Исходя из названия формата, разделителем колонок является запятая или другие разделители.

Разделители в формате CSV реализованы любые, но предусмотрены отдельные подформаты с собственными — например, TSV (tab separated values).

Пример фрагмента данных файла, сохраненного с расширением **.csv** (файл **examples/01\_csv/kp\_data.csv**):

| hostname,vendor,model,location kp1,Cisco,2960,Moscow kp2,Cisco,2960,Novosibirsk kp3,Cisco,2960,Kazan kp4,Cisco,2960,Tomsk |
| --- |

Чтобы с данным форматом было проще работать, в Python реализован специализированный модуль **csv**.

## Чтение данных из файла формата CSV

Ниже приведен простейший пример кода (файл **examples/01\_csv/csv\_read.py**), обеспечивающего построчный вывод содержимого файла **kp\_data.csv**:

| import csv with open('kp\_data.csv') as f\_n:  f\_n\_reader = csv.reader(f\_n)  for row in f\_n\_reader:  print(row) |
| --- |

Результат выполнения этого кода:

| ['hostname', 'vendor', 'model', 'location'] ['kp1', 'Cisco', '2960', 'Moscow'] ['kp2', 'Cisco', '2960', 'Novosibirsk'] ['kp3', 'Cisco', '2960', 'Kazan'] ['kp4', 'Cisco', '2960', 'Tomsk'] |
| --- |

Получаем набор списков, в первый из которых содержит названия столбцов, а остальные — их значения.

В фрагменте **csv\_read.py** обратим внимание на модуль **csv.reader**, который принимает в качестве параметра ссылку на объект, поддерживающий протокол итератора. Значением параметра может быть любой объект, для которого доступен метод **write()**. При этом в переменной **f\_n\_reader** содержится указатель на сам итератор:

| In[1]: with open('kp\_data.csv') as f\_n:  f\_n\_reader = csv.reader(f\_n)  print(f\_n\_reader) Out[1]: <\_csv.reader object at 0x0000000003116048> |
| --- |

Полученный итератор также можно преобразовать в список (**csv\_read.py**):

| In[2]: with open('kp\_data.csv') as f\_n:  f\_n\_reader = csv.reader(f\_n)  print(list(f\_n\_reader)) Out[2]: [['hostname', 'vendor', 'model', 'location'], ['kp1', 'Cisco', '2960', 'Moscow'], ['kp2', 'Cisco', '2960', 'Novosibirsk'], ['kp3', 'Cisco', '2960', 'Kazan'], ['kp4', 'Cisco', '2960', 'Tomsk']] |
| --- |

На практике может потребоваться отделить строки с заголовками от содержимого таблицы при выводе. Для этого можно применить следующий алгоритм (**csv\_read.py**):

| with open('kp\_data.csv') as f\_n:  f\_n\_reader = csv.reader(f\_n)  f\_n\_headers = next(f\_n\_reader)  print('Headers: ', f\_n\_headers)  for row in f\_n\_reader:  print(row) |
| --- |

Результат:

| Headers: ['hostname', 'vendor', 'model', 'location'] ['kp1', 'Cisco', '2960', 'Moscow'] ['kp2', 'Cisco', '2960', 'Novosibirsk'] ['kp3', 'Cisco', '2960', 'Kazan'] ['kp4', 'Cisco', '2960', 'Tomsk'] |
| --- |

Еще один вариант чтения данных из файла предлагает метод **DictReader** модуля **csv**. Он реализует более удобный и понятный формат вывода, когда каждой строке таблицы соответствует словарь, в котором элементы представляют собой связку «ключ (название столбца): значение (значение столбца)» (**csv\_read.py**):

| with open('kp\_data.csv') as f\_n:  f\_n\_reader = csv.DictReader(f\_n)  for row in f\_n\_reader:  print(row) |
| --- |

Результат:

| {'hostname': 'kp1', 'vendor': 'Cisco', 'model': '2960', 'location': 'Moscow'} {'hostname': 'kp2', 'vendor': 'Cisco', 'model': '2960', 'location': 'Novosibirsk'} {'hostname': 'kp3', 'vendor': 'Cisco', 'model': '2960', 'location': 'Kazan'} {'hostname': 'kp4', 'vendor': 'Cisco', 'model': '2960', 'location': 'Tomsk'} |
| --- |

Можно выводить содержимое отдельных столбцов. При этом необходимо указать их ключи-названия (**csv\_read.py**):

| print(row['hostname'], row['model']) |
| --- |

Результат:

| kp1 2960 kp2 2960 kp3 2960 kp4 2960 |
| --- |

Механизм работы метода **DictReader** в различных подверсиях Python 3 различается. Ранее при использовании интерпретаторов данный метод создавал только стандартные словари. С подверсии 3.6 реализована генерация упорядоченных словарей, благодаря чему последовательность элементов полностью совпадает с порядком столбцов CSV-файла.

## Запись данных в файл формата CSV

В модуле **csv** реализованы и возможности записи данных в файл соответствующего формата. В приведенном ниже примере список строк записывается в файл с расширением **.csv**, а затем содержимое файла считывается в стандартный поток вывода (файл **examples/01\_csv/csv\_write.py**).

| data = [['hostname', 'vendor', 'model', 'location'],  ['kp1', 'Cisco', '2960', 'Moscow, str'],  ['kp2', 'Cisco', '2960', 'Novosibirsk, str'],  ['kp3', 'Cisco', '2960', 'Kazan, str'],  ['kp4', 'Cisco', '2960', 'Tomsk, str']]  with open('kp\_data\_write.csv', 'w') as f\_n:  f\_n\_writer = csv.writer(f\_n)  for row in data:  f\_n\_writer.writerow(row)  with open('kp\_data\_write.csv') as f\_n:  print(f\_n.read()) |
| --- |

Результат:

| hostname,vendor,model,location  kp1,Cisco,2960,"Moscow, str"  kp2,Cisco,2960,"Novosibirsk, str"  kp3,Cisco,2960,"Kazan, str"  kp4,Cisco,2960,"Tomsk, str" |
| --- |

В приведенном выше коде последний столбец содержит значения, которые взяты в кавычки. При этом в потоке вывода данные значения также представлены с кавычками. Это указывает модулю **csv**, что все значение является строкой и запятая не выступает разделителем. Считается хорошей практикой явное указание кавычек для каждого значения, даже если оно не содержит запятых. Но необязательно указывать их явно. В модуле **csv** можно программно определять такую опцию (**csv\_write.py**).

| data = [['hostname', 'vendor', 'model', 'location'],  ['kp1', 'Cisco', '2960', 'Moscow, str'],  ['kp2', 'Cisco', '2960', 'Novosibirsk, str'],  ['kp3', 'Cisco', '2960', 'Kazan, str'],  ['kp4', 'Cisco', '2960', 'Tomsk, str']]  with open('kp\_data\_write\_2.csv', 'w') as f\_n:  f\_n\_writer = csv.writer(f\_n, quoting=csv.QUOTE\_NONNUMERIC)  for row in data:  f\_n\_writer.writerow(row)  with open('kp\_data\_write\_2.csv') as f\_n:  print(f\_n.read()) |
| --- |

Результат:

| "hostname","vendor","model","location"  "kp1","Cisco","2960","Moscow, str"  "kp2","Cisco","2960","Novosibirsk, str"  "kp3","Cisco","2960","Kazan, str"  "kp4","Cisco","2960","Tomsk, str" |
| --- |

Номер модели имеет строковый тип данных, и в данном случае он также преобразуется в формат с кавычками.

В модуле **csv** для итератора реализован полезный метод **writerows**, позволяющий не построчно записывать данные в файл, а передать объект (например, список) с данными в качестве аргумента и выполнить мгновенную запись сразу всех данных (**csv\_write.py**).

| data = [['hostname', 'vendor', 'model', 'location'],  ['kp1', 'Cisco', '2960', 'Moscow, str'],  ['kp2', 'Cisco', '2960', 'Novosibirsk, str'],  ['kp3', 'Cisco', '2960', 'Kazan, str'],  ['kp4', 'Cisco', '2960', 'Tomsk, str']]  with open('kp\_data\_write\_3.csv', 'w') as f\_n:  f\_n\_writer = csv.writer(f\_n, quoting=csv.QUOTE\_NONNUMERIC)  f\_n\_writer.writerows(data)  with open('kp\_data\_write\_3.csv') as f\_n:  print(f\_n.read()) |
| --- |

Метод **DictWriter** позволяет сохранять словари в csv-представлении. Принцип работы этого метода и стандартного **writer** практически совпадают. Но упорядоченность реализована применительно к словарям Python только с версии 3.6, поэтому необходимо явно указывать порядок следования столбцов в файле. За это отвечает параметр **fieldnames**.

В качестве разделителя можно определить любой символ, который устанавливается как значение параметра **delimiter** метода **reader**. Например, если данные в файле разделены с помощью «!», можно указать модулю **csv** использовать именно восклицательный знак при разделении данных (файл **examples/01\_csv/kp\_data\_delimiter.csv**).

| hostname!vendor!model!location kp1!Cisco!2960!Moscow kp2!Cisco!2960!Novosibirsk kp3!Cisco!2960!Kazan kp4!Cisco!2960!Tomsk |
| --- |

Простейший код с использованием разделителя (**csv\_write.py**):

| with open('kp\_data\_delimiter.csv') as f\_n:  f\_n\_reader = csv.reader(f\_n, delimiter='!')  for row in f\_n\_reader:  print(row) |
| --- |

# Файлы JSON как средство обмена данными

Аббревиатура JSON расшифровывается как «JavaScript Object Notation». Это текстовый формат, который используют для операций с данными (хранение, обмен). Синтаксис JSON и Python похожи — оба просты для восприятия. Как и при работе с форматом CSV, в Python реализован специализированный модуль, упрощающий запись и чтение данных в JSON.

## Чтение JSON-файлов

Рассмотрим пример: есть простейший JSON-объект, содержащий сообщение, которое можно отправить в чате от одного пользователя — другому (файл **examples/02\_json/msg\_example\_read.json**).

| {  "action": "msg",  "time": <unix timestamp>,  "to": "account\_name",  "from": "account\_name",  "encoding": "ascii",  "message": "message" } |
| --- |

Для операций с JSON-объектами в Python 3 предназначен модуль **json**, в котором для чтения данных реализовано два метода: **load** и **loads**. Первый считывает файл в JSON-формате и возвращает python-объекты. Второй — отвечает за считывание строки в JSON-формате и тоже возвращает python-объекты (файл **examples/02\_json/json\_read.py**). Пример использования метода **load**:

| import json  with open('mes\_example.json') as f\_n:  objs = json.load(f\_n)  for section, commands in objs.items():  print(section)  print(commands) |
| --- |

Результат выполнения данного кода:

| action msg from account\_name to account\_name encoding ascii message message |
| --- |

Пример использования метода **loads** с аналогичным предыдущему примеру результатом (файл **examples/02\_json/json\_read.py**):

| with open('mes\_example.json') as f\_n:  f\_n\_content = f\_n.read()  objs = json.loads(f\_n\_content)  print(objs)  for section, commands in objs.items():  print(section)  print(commands) |
| --- |

## Запись в JSON-файлы

Для записи в JSON-файлы на Python 3 есть два метода: **dump** и **dumps**. Первый сохраняет python-объект в json-файл. Второй возвращает строку в json-формате. Пример ниже демонстрирует конвертацию python-объекта в формат JSON (файл **examples/02\_json/json\_write.py**). Метод **dumps** можно применять в тех случаях, когда требуется вернуть строку в JSON-формате — например, для последующей ее передачи в API.

| import json  dict\_to\_json = {  "action": "msg",  "to": "account\_name",  "from": "account\_name",  "encoding": "ascii",  "message": "message"  }  with open('mes\_example\_write.json', 'w') as f\_n:  f\_n.write(json.dumps(dict\_to\_json))  with open('mes\_example\_write.json') as f\_n:  print(f\_n.read()) |
| --- |

Чтобы записать информацию в JSON-формате в файл, корректнее применять метод **dump** (файл **json\_write.py**).

| import json  dict\_to\_json = {  "action": "msg",  "to": "account\_name",  "from": "account\_name",  "encoding": "ascii",  "message": "message"  }  with open('mes\_example\_write\_2.json', 'w') as f\_n:  json.dump(dict\_to\_json, f\_n)  with open('mes\_example\_write\_2.json') as f\_n:  print(f\_n.read()) |
| --- |

## Определение дополнительных параметров методов записи

Форматом вывода данных можно управлять, определив для методов записи **dump** и **dumps** дополнительные параметры. По умолчанию эти методы используются без них и обеспечивают запись информации в компактном представлении. Такой подход эффективен, когда данные используются другими приложениями, а визуальное представление — не на первом месте по важности. Если же предполагается, что работать с данными будет человек, а не программа, следует позаботиться о более удобном формате представления (**json\_write.py**).

| dict\_to\_json = {  "action": "msg",  "to": "account\_name",  "from": "account\_name",  "encoding": "ascii",  "message": "message"  }  with open('mes\_example\_write\_3.json', 'w') as f\_n:  json.dump(dict\_to\_json, f\_n, sort\_keys=True, indent=2)  with open('mes\_example\_write\_3.json') as f\_n:  print(f\_n.read()) |
| --- |

В данном случае параметры **sort\_keys** и **indent** позволяют выполнить сортировку данных при записи, а также установить величину отступа. При этом содержимое файла **mes\_example\_write\_3.json** будет выглядеть следующим образом:

| {  "action": "msg",  "encoding": "ascii",  "from": "account\_name",  "message": "message",  "to": "account\_name" } |
| --- |

## 

## Изменение типа данных

Еще один важный момент, связанный с преобразованием данных в JSON-формат: итоговый формат JSON может не совпадать с исходным python-форматом. Например, кортежи при записи в JSON конвертируются в списки (файл **examples/02\_json/data\_type\_change.py**).

| In[3]: import json  In[4]: tuple\_ex = (  "action",  "to",  "from",  "encoding",  "message"  )  In[5]: print(type(tuple\_ex)) Out[5]: <class 'tuple'>  In[6]: with open('tuple\_ex.json', 'w') as f\_n:  json.dump(tuple\_ex, f\_n, sort\_keys=True, indent=2)  In[7]: obj = json.load(open('tuple\_ex.json'))  In[8]: print(type(obj)) Out[8]: <class 'list'> |
| --- |

Эта ситуация возникает из-за различия типов данных JSON и Python, поскольку не для всех из них существуют соответствия. Ниже приведены таблицы, описывающие типы данных при конвертации из Python в JSON и в обратном направлении.

**Python -> JSON**:

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

**JSON -> Python:**

| **JSON** | **Python** |
| --- | --- |
| object | dict |
| array | list |
| string | str |
| number (int) | int |
| number (real) | float |
| true | True |
| false | False |

## Ограничения на тип данных

При использовании формата JSON есть ограничение: в нем нельзя сохранить словарь, где в качестве ключей — кортежи (**data\_type\_change.py**).

| In[9]: dict\_to\_json = {('action', 'to'): 'msg', 'from': 'account\_name'}  In[10]: with open('dict\_to\_json.json', 'w') as f\_n:  json.dump(dict\_to\_json, f\_n) ...  TypeError: key ('action', 'to') is not a string |
| --- |

Использование дополнительного параметра '**skipkeys' = True** позволяет игнорировать такие ключи и избегать ошибок (**data\_type\_change.py**).

| In[11]: with open('dict\_to\_json.json', 'w') as f\_n:  json.dump(dict\_to\_json, f\_n, skipkeys=True)  In[12]: with open('dict\_to\_json.json') as f\_n:  f\_n\_content = f\_n.read()  obj = json.loads(f\_n\_content)  In[13]: print(obj) Out[13]: {'from': 'account\_name'} |
| --- |

Ключами в словарях в JSON-формате могут быть только строковые величины. Если у Python-словаря ключи определены в виде чисел, они будут преобразованы в строковое представление (**data\_type\_change.py**) без ошибок.

| In[14]: d = {5:300, 1:400}  In[15]: d\_to\_json = json.dumps(d)  In[16]: print(d\_to\_json) Out[16]: {"1": 400, "5": 300} |
| --- |

# Работа с YAML-файлами при обработке и сохранении данных

Аббревиатура YAML расшифровывается как «Ain't Markup Language», это еще один формат сохранения данных. Обладает более приятным для восприятия синтаксисом (по сравнению с JSON), поэтому часто используется в описании логики работы скриптов.

## Синтаксис формата YAML

Как и в Python, при определении структуры документа используются отступы, причем их виды ограничены только пробелами — знаки табуляции применять нельзя. Комментарии в YAML тоже начинаются с символа #.

### Работа со списками

Список может иметь стандартное представление в виде строки (файл **examples/03\_yaml/yaml\_ex.yaml**):

| ['action', 'to', 'from', 'encoding', 'message'] |
| --- |

Список может быть структурирован. При этом каждый элемент записывается в своей строке и маркируется символом «- » (обязательно с пробелом после него). Все строки набора должны иметь одинаковую величину отступа (**yaml\_ex.yaml**).

| - action - to - from - encoding - message |
| --- |

### Работа со словарями

Для словарей действуют аналогичные правила. Возможна запись в строковом представлении (**yaml\_ex.yaml**):

| {'action': 'msg', 'to': 'account\_name'} |
| --- |

Или в структурированном виде (в виде блока):

| 'action': 'msg' |
| --- |

### Работа со строками

Формат YAML реализован таким образом, что строки не обязательно оформлять кавычками. Но если они включают специальные символы, следует обязательно взять строку в кавычки, чтобы она была корректно обработана (**yaml\_ex.yaml**):

| command: "action | to" |
| --- |

### Комбинация элементов

Комбинации в YAML объединяют элементы различных типов. Например, в приведенной ниже структуре данных (словаре) каждому ключу соответствует набор элементов (список):

| message: - msg\_1 - msg\_2 - msg\_3 to: - account\_1 - account\_2 - account\_3 |
| --- |

Это пример демонстрирует другой вариант комбинации — список словарей:

| - action: msg\_1  to: account\_1 - action: msg\_2  to: account\_2 |
| --- |

## Использование PyYAML

Для работы с данными из Python в формате YAML разработан модуль **PyYAML**, который не входит в набор стандартных и требует отдельной установки с помощью этой команды:

| pip install pyyaml |
| --- |

Этот модуль используется так же, как **csv** и **json**. **PyYAML** позволяет проводить стандартные операции над данными — считывание и запись.

### 

### Считывание данных

Например, есть файл **examples/03\_yaml/data\_read.yaml:**

| - action: msg\_1  to: account\_1 - action: msg\_2  to: account\_2 |
| --- |

Он обрабатывается представленным ниже программным кодом (файл **examples/03\_yaml/pyyaml\_examples.py**):

| with open('data\_read.yaml') as f\_n:  f\_n\_content = yaml.load(f\_n) print(f\_n\_content) |
| --- |

Результат:

| [{'to': 'account\_1', 'action': 'msg\_1'}, {'to': 'account\_2', 'action': 'msg\_2'}] |
| --- |

Используя формат YAML, повышаем удобство работы с данными, особенно если надо вручную указывать параметры.

### Запись данных

Следующий пример демонстрирует запись Python-объектов (словарей с элементами-списками) в файл формата YAML (файл **examples/03\_yaml/pyyaml\_examples.py**).

| action\_list = ['msg\_1',  'msg\_2',  'msg\_3']  to\_list = ['account\_1',  'account\_2',  'account\_3']  data\_to\_yaml = {'action':action\_list, 'to':to\_list}  with open('data\_write.yaml', 'w') as f\_n:  yaml.dump(data\_to\_yaml, f\_n)  with open('data\_write.yaml') as f\_n:  print(f\_n.read()) |
| --- |

Результат его выполнения (файл **examples/03\_yaml/data\_write.yaml**):

| action: [msg\_1, msg\_2, msg\_3] to: [account\_1, account\_2, account\_3] |
| --- |

Итоговые списки записались в строку. Такой вариант представления определяется по умолчанию. Чтобы его изменить, необходимо установить для параметра **default\_flow\_style** значение **False** (файл **examples/03\_yaml/pyyaml\_examples.py**):

| action\_list = ['msg\_1',  'msg\_2',  'msg\_3']  to\_list = ['account\_1',  'account\_2',  'account\_3']  data\_to\_yaml = {'action':action\_list, 'to':to\_list}  with open('data\_write.yaml', 'w') as f\_n:  yaml.dump(data\_to\_yaml, f\_n, default\_flow\_style=False)  with open('data\_write.yaml') as f\_n:  print(f\_n.read()) |
| --- |

Результат изменения формата:

| action: - msg\_1 - msg\_2 - msg\_3 to: - account\_1 - account\_2 - account\_3 |
| --- |

Так считываемые данные выглядят более понятно.

# Практическое задание

1. Задание на закрепление знаний по модулю **CSV**. Написать скрипт, осуществляющий выборку определенных данных из файлов **info\_1.txt**, **info\_2.txt**, **info\_3.txt** и формирующий новый «отчетный» файл в формате **CSV**. Для этого:
   1. Создать функцию **get\_data()**, в которой в цикле осуществляется перебор файлов с данными, их открытие и считывание данных. В этой функции из считанных данных необходимо с помощью регулярных выражений извлечь значения параметров «Изготовитель системы», «Название ОС», «Код продукта», «Тип системы». Значения каждого параметра поместить в соответствующий список. Должно получиться четыре списка — например, **os\_prod\_list**, **os\_name\_list**, **os\_code\_list**, **os\_type\_list**. В этой же функции создать главный список для хранения данных отчета — например, **main\_data** — и поместить в него названия столбцов отчета в виде списка: «Изготовитель системы», «Название ОС», «Код продукта», «Тип системы». Значения для этих столбцов также оформить в виде списка и поместить в файл **main\_data** (также для каждого файла);
   2. Создать функцию **write\_to\_csv()**, в которую передавать ссылку на CSV-файл. В этой функции реализовать получение данных через вызов функции **get\_data()**, а также сохранение подготовленных данных в соответствующий CSV-файл;
   3. Проверить работу программы через вызов функции **write\_to\_csv()**.
2. Задание на закрепление знаний по модулю **json**. Есть файл **orders** в формате **JSON** с информацией о заказах. Написать скрипт, автоматизирующий его заполнение данными. Для этого:
   1. Создать функцию **write\_order\_to\_json()**, в которую передается 5 параметров — товар (**item**), количество (**quantity**), цена (**price**), покупатель (**buyer**), дата (**date**). Функция должна предусматривать запись данных в виде словаря в файл **orders.json**. При записи данных указать величину отступа в 4 пробельных символа;
   2. Проверить работу программы через вызов функции **write\_order\_to\_json()** с передачей в нее значений каждого параметра.
3. Задание на закрепление знаний по модулю **yaml**. Написать скрипт, автоматизирующий сохранение данных в файле YAML-формата. Для этого:
   1. Подготовить данные для записи в виде словаря, в котором первому ключу соответствует список, второму — целое число, третьему — вложенный словарь, где значение каждого ключа — это целое число с юникод-символом, отсутствующим в кодировке ASCII (например, €);
   2. Реализовать сохранение данных в файл формата YAML — например, в файл **file.yaml**. При этом обеспечить стилизацию файла с помощью параметра **default\_flow\_style**, а также установить возможность работы с юникодом: **allow\_unicode = True**;
   3. Реализовать считывание данных из созданного файла и проверить, совпадают ли они с исходными.

# Дополнительные материалы

1. [Python. Работа с данными в различных форматах](https://ideafix.name/wp-content/uploads/2012/05/Python-7.pdf).
2. [Файлы CSV](https://metanit.com/python/tutorial/4.3.php).
3. [CSV File Reading and Writing](https://docs.python.org/2/library/csv.html).
4. [Работа с JSON в Python](https://ru.stackoverflow.com/questions/540972/%D0%A0%D0%B0%D0%B1%D0%BE%D1%82%D0%B0-%D1%81-json-%D0%B2-python).
5. [Как преобразовать данные JSON в объект Python](http://qaru.site/questions/16673/how-to-convert-json-data-into-a-python-object).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Python 3 для сетевых инженеров](https://natenka.gitbooks.io/pyneng/content/).
2. Билл Любанович. Простой Питон. Современный стиль программирования. (Каталог «Дополнительные материалы».)
3. [Обрабатываем csv-файлы — Модуль CSV](https://python-scripts.com/import-csv-python).
4. [Модуль JSON](https://pythonworld.ru/moduli/modul-json.html).
5. [Знакомимся с YAML](https://www.opennet.ru/base/dev/yaml.txt.html).