Урок 2

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

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:

    a. 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);

b. 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;

c. 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:

   a. 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;

   b. 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:

   a. 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, €);

   b. 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**;

   c. Реализовать считывание данных из созданного файла и проверить, совпадают ли они с исходными.

Дополнительные материалы

1. Python. Работа с данными в различных форматах.
2. Файлы CSV.
3. CSV File Reading and Writing.
4. Работа с JSON в Python.
5. Как преобразовать данные JSON в объект Python.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Python 3 для сетевых инженеров.

2. Билл Любанович. Простой Питон. Современный стиль программирования. (Каталог «Дополнительные материалы».)
3. Обрабатываем csv-файлы — Модуль CSV.
4. Модуль JSON.
5. Знакомимся с YAML.