Работа с файловой системой и модули

Сегодня рассмотрим:

- Чтение файлов
- Запись файлов
- Работа с пакетами (модулями)

ЧТЕНИЕ ФАЙЛОВ

Рассматривать будем на примере простого файла *.csv, где столбцы данных разделены запятыми. Для чтения файла действуем с помощью следующих шагов:

- Открытие файла (open(файл, опция))
  Опция принимает значения:
  - ‘r’ (read) – только чтение
  - ‘rb’ (read in byte mode) – побайтовое чтение (применяется в особых случаях)
  - ‘w’ (write) – перезапись (старые данные уничтожаются)
  - ‘wb’ (write in byte mode) – байтовая запись (применяется в особых случаях)
  - ‘a’ (append) – добавление в конец
- Чтение построчно (.readline(), next())
- Чтение всех строчек (.readlines())
- Закрытие файла (.close())

Пример:

```
f = open(file.csv, ‘r’)        # Открываем
f.readline()                   # Читаем строку
f.readline()                   # Читаем ещё строку
next(f)                        # Или так
print(f.readline())            # Пример, строчки, из, файла\n
f.close()                      # Закрываем
```

Чтение всех строчек позволяет работать с файлом подобно списку.

Пример:

```
f = open(file.csv, ‘r’)
file = f.readlines()           # Читаем строку
print(file)                    # Съешь, булок\n Выпей, чаю\n
f.close()
```

При этом можно заметить, что в конце строки содержится атрибут переноса строки (\n). Чтобы избавиться от этого, убрать пробелы и разделить столбцы используем комбинацию .strip.split(‘,’):

Пример:

```
f = open(file.csv, ‘r’)
line = f.readline()                  # Пример , строчки ,из, файла\n
print(line.strip().split(‘,’))       # Пример, строчки, из, файла
f.close()
```

После закрытия файла прочитать его не получится.

ЗАПИСЬ В ФАЙЛ

Запись в файл производится аналогично чтению с помощью метода .write(строка). При этом, если кто-то ещё открыл файл, то при закрытии произойдёт перезапись.

Пример:

```
f = open(file.csv, ‘w’)
f.write(‘Карл у Клары украл кораллы’)
g = open(file.csv, ‘w’)
g.write('А Клара украла у Карла кларнет')
f.close()                                    # Запись
g.close()                                    # Перезапись
```

Этот способ является классическим, но не очень удобным. Гораздо чаще используется способ записи через контекстный менеджер. Этот вариант решает проблему множественного доступа тем, что блокирует доступ к файлу, пока он открыт. В этом случае перезапись данных будет невозможна.

Пример:

```
with open(file.csv, ‘w’) as f:
      f.write(‘Карл у Клары украл кораллы’)     # Запись произойдёт
      g = open(file.csv, ‘w’)
      g.write('А Клара украла у Карла кларнет')
      g.close()                                 # Перезапись не произойдёт
```

Но что, если в файле просто строки, а вложенные словари? То есть, файл приблизительно такого вида:

```
{‘user’: ‘user1’, ‘id’: ‘id1’}, {‘user’: ‘user2’, ‘id’: ‘id2’}
```

В этом случае поможет библиотека **json**. А конкретнее – функция .loads(источник).

Пример:

```
import json
line = ‘{‘user’: ‘user1’, ‘id’: ‘id1’}’
json.loads(line)  # {‘user’: ‘user1’, ‘id’: ‘id1’}
```

Метод довольно умный, поскольку в той или иной мере распознаёт содержимое строки. Например, если в строке будет спрятан список, а не словарь, то функция распознает его именно как список. Все соответствующие методы словарей и списков будут доступны.

Таким образом, если необходимо считать подобный файл, нужно обеспечить построчное чтение и работу json.loads(строка). Есть и другие способы (например, через библиотеку eval), но библиотека json лучше отвечает требованиям по безопасности.

Кроме того, если необходимо сделать обратную операцию (например, в веб-запросах), можно воспользоваться методом json.dumps(данные). Однако, при использовании нужно помнить, что кириллические символы могут быть преобразованы к Unicode-кодам, что потребует дальнейшей дополнительной обработки.

Пример:

```
import json
data = {‘user’: ‘user1’, ‘id’: ‘id1’}
json.dumps(data)  # ‘{“user”: “user1”, “id”: “id1”}’
```

Однако, необязательно данные приводить к строчному виду. Модуль **pickle** даёт возможность записи любого объекта сразу в файл: то есть, в виде потока байтов с помощью метода picke.dump(данные, файл).

Пример:

```
import pickle
data = {‘user’: ‘user1’, ‘id’: ‘id1’}
with open(‘bt.pickle’, ‘wb’) as f  # bt.pickle – имя файла. Может быть любым
      pickle.dump(data, f)         # 1a5b 5f0c 6de2 … (пример записи)
```

Обратная операция выполняется методом pickle.load(файл).

Пример:

```
import pickle
with open(‘bt.pickle’, ‘rb’) as f
      data = pickle.dump(f)
      print(data)       # {‘user’: ‘user1’, ‘id’: ‘id1’}
```

В программировании же функцией называют некую часть кода, который можно вызывать для выполнения определённых действий. По сути, это подпрограмма. Кроме того, функция по окончании выполнения возвращает определённое значение. Она позволяет избегать дублирования кода, а также улучшают читаемость и структурированность.

РАБОТА С ПАКЕТАМИ

Большое количество пакетов стоят по умолчанию в Anaconda. Однако, что делать, если какого-то из них всё же нет? Есть 3 способа решить данную проблему:

- Установить через Anaconda Navigator

- Установить через командную строку: pip install имя_пакета
- Скачать нужный пакет с github и установить вручную через pip install .

Во втором и третьем случае используется утилита pip (Python package index), которая производит работает с подключёнными репозиториями Python. Команда install позволяет установить пакет из репозитория, либо из места, которое пользователь укажет. К слову, pip можно вызвать из Jupyter Notebook. Для этого нужно в начале строки указать «!».

Пример:
```
!pip install plotly      # Устанавливаем библиотеку plotly
```

При ручной установке запустится файл setup, в котором описаны действия для установки. Это полезно для каких-то локальных библиотек, которые не опубликованы нигде, но нужно использовать. После этого их можно использовать в файлах, произведя импорт.

Пример:
```
import plotly      # Подключаем библиотеку plotly
```

Собственную библиотеку нужно сделать видимой для Python, добавив в системный реестр:

Пример:
```
import sys
sys.path.append('lib')          # Подключаем библиотеку lib.py
```

Желательно использовать полный путь при работе с удалёнными библиотеками. При работе внутри одной папки допустимо использовать относительный путь. После добавления можно будет производить импорт необходимых элементов:

Пример:
```
from lib import example          # Импортируем функцию example, которая лежит в lib
```

При необходимости можно давать свои имена импортируемым объектам.

Пример:
```
import example as ex             # Теперь обращаться будем через ex
print(ex.procedure())            # Нечто волшебное ☺
```