Файлы в Python. Типы файлов и работа с ними. Внутреннее устройство файлов

План урока

Аннотация

В уроке даются общие сведения о файлах и их хранении в современных ОС. Затрагиваются наиболее общие аспекты работы с файлами в Python: открытие,

чтение/запись, закрытие текстовых и бинарных файлов. Обзорно рассмотрены вспомогательные функции («перемотка», работа с кодировками, построчное чтение).

1. Общие сведения о файлах

Для работы с блоками логически объединенной информации, их хранения, обработки и передачи широко используются **файлы**.

Дадим определение: **файл** (англ. file) — именованная область данных на носителе информации.

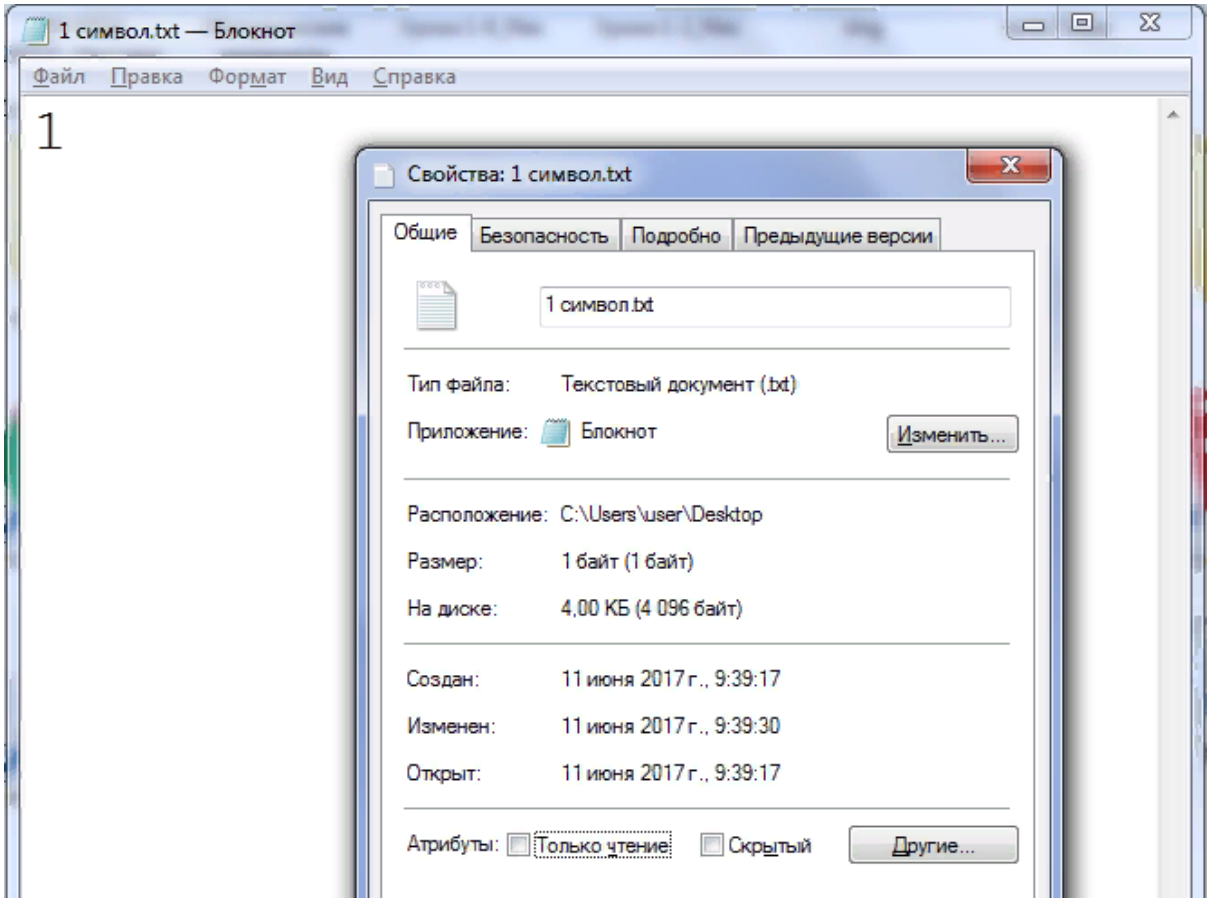
Конкретная физическая организация файлов, их группировка по каталогам (папкам), устройство процедур доступа к информации, механизмы кеширования очень сильно зависят от операционной системы и применяемой в ней файловой системы. Как правило, при работе с файлами прикладные программисты работают на верхнем уровнем **абстракции**.

Это означает, что файл представляет просто собой поток байт или текста заданного размера, а его реальное физическое устройство программиста абсолютно не интересует. Но грамотный программист должен хорошо представлять себе некоторые особенности работы с файлами, чтобы не совершать ошибки, связанные с оптимизацией и «ускорением» работы программы.

О чём надо всегда помнить:

1. Файлы обычно располагаются на носителях медленнее, чем оперативная память. Поэтому работа с ними идёт в буферизированном режиме. Даже если вы запросите один байт из файла, то считается целый блок (до нескольких килобайт). Затем он переместится в буфер оперативной памяти. Дальше файл читается оттуда, поскольку это быстрее (даже если у вас SSD) и экономит ресурсы чтения/записи внешних носителей. Небольшой файл, к которому часто обращаются, можно (прозрачно для прикладных программ) полностью поместить в оперативную память. На самом жёстком диске располагаются буферные зоны с более быстрым доступом — в общем случае мы имеем дело с многоуровневой буферизацией. Поэтому **очень плохо** считывать файл небольшими порциями коротких блоков (до нескольких мегабайт) якобы для уменьшения затрачиваемой памяти.

2. То, как файлы и каталоги (папки) располагаются на жёстком диске, зависит от типа файловой системы. Она может поддерживать сжатие, шифрование, разграничение доступа к данным. Никогда файл (кроме совсем маленьких) не размещается полностью и подряд в определённой области диска. Он разбивается на блоки для рационального расходования места. У блоков есть минимальный размер. Поэтому, даже если вы создадите файл из одного байта, он всё равно займёт целый блок данных (например, 4КБ).



3. Вся работа по буферизации, чтению, записи, открытию, закрытию файла идет через операционную систему. Чаще всего прикладной программист не работает с файловой системой напрямую.

4. Во многих операционных системах понятие файла как некоторой области данных на носителе, поддерживающих операции чтения/записи расширено до областей в оперативной памяти, ресурсов сети, оборудования и т.д. Эта концепция называется **«всё есть файл»**. Например, в Python есть «файл» sys.stdin, который ассоциирован с клавиатурным вводом и не является классическим файлом.

Есть некоторые различия в именовании файлов в Unix-подобных ОС и Windows.

Например, в Linux файлы как правило не имеют расширения. Все устройства и диски добавлены в общее дерево, корень которого обозначается «/».

Для отделения имен папок используется прямой слэш «/», а не обратный, как в Windows.

Приведём примеры специальных «файлов» в ОС Linux:

— /dev/sd{буква} — жёсткий диск (в системах на ядре Linux);

— /dev/sd{буква}{номер} — раздел диска (в системах на ядре Linux);

— /dev/sr{номер} или /dev/scd{номер} — CD-ROM;

— /dev/eth{номер} — сетевые интерфейсы Ethernet;

— /dev/wlan{номер} — сетевые интерфейсы Wireless;

— /dev/lp{номер} — принтеры;

— /dev/video{номер} — устройства изображений, камеры, фотоаппараты и т. д;

— /dev/bus/usb/000/{номер} — устройство номер на шине USB первого контроллера (000) (в системах на ядре Linux);

— /dev/tty{номер} — текстовый терминал;

— /dev/dsp — звуковой вывод;

— /dev/random — случайные данные (псевдоустройство);

— /dev/null — пусто (псевдоустройство);

— /dev/zero — нулевые байты (псевдоустройство).

{номер} — это порядковый номер устройства

2. Перевод строки

Существует два символа **\n** и **\r**, смысл которых взят из эпохи печатных машинок. Посмотрим, какие у них коды:

```
# LINE FEED.
# Перемещает позицию печати на одну строку вниз
# (изначально – без возврата каретки машинки).
print(ord("\n"))
# CARRIAGE RETURN.
# Перемещает позицию печати в крайнее левое положение
# (изначально – без перехода на следующую строку).
print(ord("\r"))

------------------------------------------------------------

    10
    13
```

В ОС Windows для перевода строки принимается последовательность \r\n, в MacOS (до версии X) — \r, а в Linux — \n.

Сейчас всё чаще во всех ОС используется одиночный **\n**.

Поэтому программисту надо быть внимательным и помнить все варианты перевода строки на той ОС, где будет работать его программа.

3. Файловый путь. Относительные и абсолютные пути

Путь файла (или путь к файлу) — последовательное указание имён каталогов, через которые надо пройти, чтобы добраться до объекта. Каталоги в записи пути разделяются **слэшем**.
В зависимости от вида ОС слэши могут быть как прямыми, так и обратными.

На ОС Windows путь выглядит так:

```
f = open('C:\\users\\user\\1.txt')
```

Обратный слэш удваивается как служебный символ. Значит, если он нужен сам по себе, то его нужно «экранировать». Вспомните, что в языке Python есть служебные символы (\n, \t).

Для того, чтобы сделать работу с файлами универсальнее, в путях файлов в Windows в Python-программах рекомендуется ставить прямой слеш. В наших примерах мы так и будем делать.

```
f = open('C:/users/user/1.txt')
```

Если мы укажем относительные пути, например:

```
f = open("user/1.txt")
```

то Python будет искать файл в каталоге, начав отсчёт с папки, в которой находится файл с основной Python-программой. Это важно помнить в проектах, где много файлов и происходит импорт модулей или функций.

4. Кодировки файлов

Сейчас принято использовать одну из самых распространенных кодировок — **utf-8**.
Мы поступим также.

UTF-8 — это сложная кодировка, в которой символ кодируется от одного до шести байт.
Подробнее про эту кодировку можно почитать здесь.

Но помните, что до сих пор существуют и старые однобайтовые кодировки:

— windows-1251;

— cp-866;

— КОИ-8.

Они по умолчанию используются в некоторых ОС, например, в Windows.

5. Типичные операции с файлами

Все файлы на диске — последовательность байт. Операционную систему **не волнует**, какой смысл у содержимого файла: это видео, чертёж, картинка, текстовый документ и т.д. Всё это остается в компетенции прикладной программы.

Единственное **исключение** сделано для текстовых файлов, потому что их состав максимально прост (до внедрения Юникода одному символу соответствовал 1 байт). Поэтому, если программа **знает**, что файл текстовый, то сразу читает из файла символы, а не поток байт.

Таким образом, все файлы искусственно разделены на **текстовые** и **бинарные**. Но не забывайте, что любой текстовый файл является бинарным.

Так как общение с файлами идет не напрямую, а через ОС, общепринятая последовательность операций с файлом следующая:

— Попросить ОС открыть файл в различных режимах для чтения или записи, в бинарном или текстовом режиме;

— Поработать с информацией из файла, используя в том числе операции чтения/записи;

— Закрыть файл.

После успешного завершения Python-программы, все файлы закрываются автоматически. Но важно всё равно закрывать файл, как только он перестаёт быть вам нужным. Это поможет избежать конфликтов совместного доступа или риска получить неконсистентный (испорченный) файл, если программа завершится аварийно.

6. Чтение файлов

Возьмём файл с первым томом «Войны и мира» Льва Толстого.

Для открытия файлов в Python есть функция **open()**.

Выполним команду для получения справки по этой функции:

```
help(open)

----------------------------------------------------------
```

```
Help on built-in function open in module io:

open(file, mode='r', buffering=-1, encoding=None, errors=None,
    newline=None, closefd=True, opener=None)
    Open file and return a stream.  Raise IOError upon failure.

    file is either a text or byte string giving the name (and the path
    if the file isn't in the current working directory) of the file to
    be opened or an integer file descriptor of the file to be
    wrapped. (If a file descriptor is given, it is closed when the
    returned I/O object is closed, unless closefd is set to False.)

...
```

Из всех параметров мы используем следующие:

— file — имя открываемого файла. Оно может быть задано с использованием и относительных и абсолютных путей;

— mode — режим открытия. r (или rt) — чтение в текстовом режиме, rb — чтение в бинарном режиме, w — запись, wb — запись в бинарном режиме. По умолчанию файл открывается в режиме r;

— encoding — если работа идёт в текстовом режиме, Python должен получить имя кодировки, чтобы корректно работать с данными. Независимо от кодировки файла, в результате чтения будет возвращаться стандартная юникод-строка Питона.

Пожалуйста, прочтите полное описание параметров в документации по функции open.

Теперь откроем файл в бинарном режиме и прочитаем первые 20 байт:

```
f = open("files/Толстой.txt", mode="rb")
f.read(20)

------------------------------------------------------------

    b'               -- \xd0\x95'
```

Перед строкой стоит модификатор **b**. Он говорит о том, что перед нами поток байт. Поток байт в языке Python представляется классом **bytes**. Если вы открываете файл для чтения в бинарном режиме, то результат метода read() имеет тип bytes.

```
data = open("files/Толстой.txt", mode="rb").read()
print(type(data))
print(data[19])
```

```
------------------------------------------------------------

 <class 'bytes'>
    149
```

Ещё раз обратите внимание на то, что в путях до файла используются прямые слэши (/). Можно использовать и обратные, но тогда их придётся экранировать, либо применять модификатор строки **r**. Кроме того, в Unix-подобных ОС принято использовать именно прямой слэш.

Чтобы понять, что делает модификатор **r**, рассмотрим пример:

```
print("ab\n12")
print(r"ab\n12")

------------------------------------------------------------

    ab
    12
    ab\n12
```

В первом случае специальный символ \n «отработал» и перевёл строку, а во втором — вывелся на экран **как есть**. Модификатор **r** отключает спец-символы, если он указан перед строкой. То есть каждый символ означает сам себя и ничего более.

Объект, который возвращает нам функция open, ассоциирован (связан) с открытым файлом и содержит следующие поля и методы:

```
from pprint import pprint
pprint(dir(f))

------------------------------------------------------------

 ['__class__',
     '__del__',
     '__delattr__',
     '__dict__',
     '__dir__',
     '__doc__',
     '__enter__',
     '__eq__',
     '__exit__',
     '__format__',
     '__ge__',
     '__getattribute__',
```

```
    '__getstate__',
    '__gt__',
    '__hash__',
    '__init__',
    '__init_subclass__',
    '__iter__',
    '__le__',
    '__lt__',
    '__ne__',
    '__new__',
    '__next__',
    '__reduce__',
    '__reduce_ex__',
    '__repr__',
    '__setattr__',
    '__sizeof__',
    '__str__',
    '__subclasshook__',
    '_checkClosed',
    '_checkReadable',
    '_checkSeekable',
    '_checkWritable',
    '_dealloc_warn',
    '_finalizing',
    'close',
    'closed',
    'detach',
    'fileno',
    'flush',
    'isatty',
    'mode',
    'name',
    'peek',
    'raw',
    'read',
    'read1',
    'readable',
    'readinto',
    'readinto1',
    'readline',
    'readlines',
    'seek',
    'seekable',
    'tell',
    'truncate',
    'writable',
```

```
     'write',
     'writelines']
```

Функция **pprint** (pretty-print, изящный вывод) старается сделать печать больших объектов (списков, словарей) более удобной и читаемой для человека.

Назначение почти всех из них легко определяется из названий.

```
f = open("files/Толстой.txt", mode="rb")
f.read(20)
print(f.name)
print(f.readable())
print(f.tell())
print(f.writable())
print(f.seekable())
print(f.mode)
```

```
----------------------------------------------------------------

    files/Толстой.txt
    True
    20
    False
    True
    rb
```

Теперь откроем файл в текстовом режиме и проверим работу некоторых методов:

```
f = open("files/Толстой.txt", encoding="utf-8")
print(f.read(100))
print(f.tell())
print(f.seek(1245))
print(f.read(100))
print(f.tell())
```

```
----------------------------------------------------------------

                    -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des
              des
    103
    1245
    ла (грипп  был тогда  новое
              слово, употреблявшееся только  редкими).  В записочках, разосл
    1416
```

Что же произошло:

— Если мы открываем файл в текстовом режиме, то чтение происходит посимвольно (символ может занимать физически 1, 2, 4 и даже 6 байт в некоторых случаях в зависимости от кодировки). За одну операцию можно прочитать различное количество символов;

— Если мы используем бинарный режим, то чтение осуществляется побайтно и за одну операцию можно прочитать сразу несколько байт;

— Метод **tell** возвращает позицию в байтах от начала файла, а метод **seek** изменяет её (перематывает) на заданную позицию. Использование **seek** с текстовыми файлами затруднено из-за несоответствия номера байта и номера символа (это видно в предыдущем примере, когда мы считали 100 символов, а **tell** вернул нам смещение 103).

Продолжим экспериментировать:

```
print(f.seek(1246))
print(f.read(1))

------------------------------------------------------------

    1246

    ------------------------------------------------------------

    UnicodeDecodeError                  Traceback (most recent call last)

    <ipython-input-26-bcfce6fd2dac> in <module>()
          1 print(f.seek(1246))
    ----> 2 print(f.read(1))


    ~/anaconda/lib/python3.6/codecs.py in decode(self, input, final)
        319         # decode input (taking the buffer into account)
        320         data = self.buffer + input
    --> 321         (result, consumed) = self._buffer_decode
                                         (data, self.errors, final)
        322         # keep undecoded input until the next call
        323         self.buffer = data[consumed:]


    UnicodeDecodeError: 'utf-8' codec can't decode byte 0xbb
                        in position 0: invalid start byte
```

Исключение появилось потому, что начиная с 1246 байта невозможно прочитать символ в кодировке UTF-8.

Из-за буферизации и оптимизации количества операций чтения-записи иногда получается считать вообще весь файл в оперативную память (для этого не надо указывать параметр в методе read). Если его размер до десятков мегабайт, это можно комфортно делать.
В результате мы получим одну строку со всем содержимым файла.

```
f = open("files/Толстой.txt", encoding="utf8")
data = f.read()
print('Type: %s, length: %d' % (type(data), len(data)))
f.close()

-------------------------------------------------------------

Type: <class 'str'>, length: 887312
```

В текстовом режиме можно читать файл построчно с использованием метода **readline**, при этом маркером конца строки является символ \n:

```
f = open("files/Толстой.txt", encoding="utf8")
for i in range(7):
    print(f.readline(), end="")
f.close()

-------------------------------------------------------------

     -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages,
des поместья, de la famille Buonaparte.  Non, je  vous préviens, que si vous
ne  me dites pas, que nous avons la guerre, si vous vous permettez encore de
pallier  toutes les infamies, toutes les  atrocités  de cet  Antichrist  (ma
parole, j'y  crois) -- je  ne  vous  connais plus, vous n'êtes plus mon ami,
vous n'êtes  plus  мой  верный  раб,  comme  vous  dites.  [1]  Ну,
здравствуйте, здравствуйте.  Je vois  que  je  vous fais  peur, [2]
```

Также можно считать текстовый файл в список строк с помощью метода **readlines**:

```
f = open("files/Толстой.txt", encoding="utf8")
lines = f.readlines()
print('Type: %s, length: %d' % (type(lines), len(lines)))
print(lines[10])
f.close()

-------------------------------------------------------------

    Type: <class 'list'>, length: 12128
              князя  Василия,  первого  приехавшего  на  её вечер. Анна  Павловна
```

Файл может быть построчным итератором (выведем только 10 строк):

```
f = open("files/Толстой.txt", encoding="utf8")
for number, line in enumerate(f):
    print(line)
    if number > 8:
        break
f.close()
```

```
------------------------------------------------------------

    -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages,

des поместья, de la famille Buonaparte.  Non, je  vous préviens, que si vous

ne  me dites pas, que nous avons la guerre, si vous vous permettez encore de

pallier  toutes les infamies, toutes les  atrocités  de cet  Antichrist  (ma

parole, j'y  crois) -- je  ne  vous  connais plus, vous n'êtes plus mon ami,

vous n'êtes  plus  мой  верный  раб,  comme  vous  dites.  [1]  Ну,

здравствуйте, здравствуйте.  Je vois  que  je  vous fais  peur, [2]

садитесь и рассказывайте.

       Так говорила в июле 1805 года известная Анна Павловна Шерер, фрейлина и

приближенная  императрицы  Марии  Феодоровны,  встречая важного и  чиновного
```

7. Запись файлов

Для записи в файл также есть два режима: **w** (если файл существовал, его содержимое будет потеряно) и **a** — запись идёт в конец файла.

После выбора режима можно также ввести и символ «+». Посмотрите в документации по функции open, что означает такая конструкция.

Один из способов записать информацию в файл — это метод **write**. Если мы хотим сделать запись в середину файла, то должны сначала спозиционироваться на место предполагаемой

записи (метод seek), а уже потом уже записывать (метод write).

```
f = open("files/example.txt", 'w')
print(f.write('123\n456'))
print(f.seek(3))
print(f.write('34352'))
f.close()
f = open("files/example.txt", 'r')
print(f.read())
f.close()

---------------------------------------------------------------

    7
    3
    5
    12334352
```

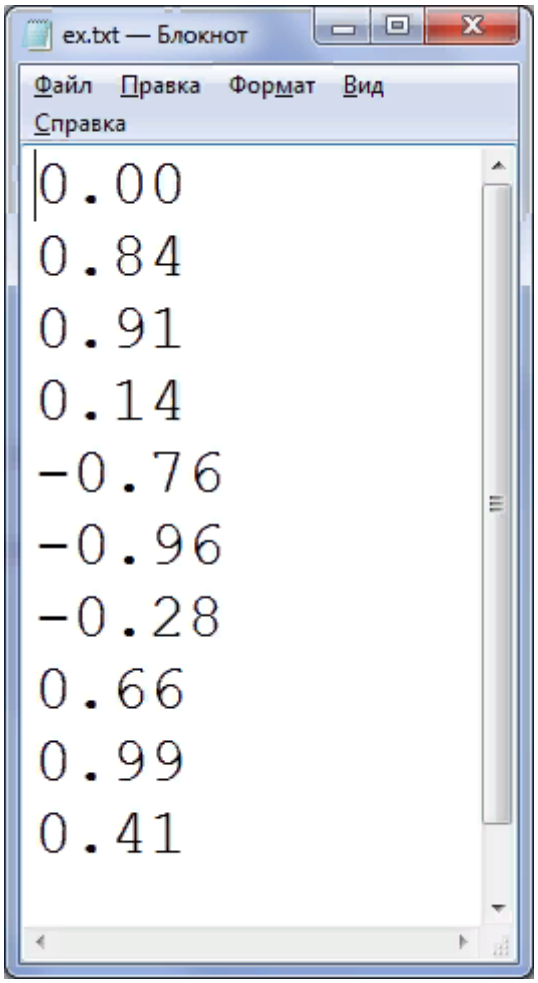
Построчно **пройдитесь** по приведённой программе, чтобы лучше понять, как она работает. Заметьте, что если мы используем seek, то данные при записи все равно будут стёрты. Этот процесс похож на рисование фломастером: рисуя поверх, вы закрашиваете старый рисунок.

Второй способ записи в файл — стандартная функция **print**. Для этого применяется именованный параметр **file**.

Например:

```
from math import sin

f = open("files/example2.txt", 'w')
for i in range(10):
    print("%0.2f" % sin(i), file=f)
f.close()
```



Для записи данных в бинарный файл в функцию write() надо передать переменную типа bytes. Например, так:

```
f = open("files/ex_bin.dt", 'wb')
data = [1, 2, 3, 4, 5]
f.write(bytes(data))
f.close()
```

Обратите внимание, что в примере мы преобразовываем список в поток байт, чтобы потом записать его в файл.

8. Закрытие файлов

Операционная система контролирует доступ к файлам. Если какая-то программа открыла файл для записи, то все попытки любых других программ изменить содержание файла заблокируются для сохранения целостности.

Поэтому после того, как работа с файлом закончена, файл надо **отпустить**, закрыв его методом **close**, что мы делали практически во всех примерах.

```
f.close()
```

Повторим, что после завершения программы все файлы, которая она использовала в своей работе, автоматически закроются. Но **хороший стиль программирования** — это как можно раньше закрыть файл.

Поэтому всегда следуйте принципу: **не нужен файл — отпусти его!**

И самое последнее: для того, чтобы файл закрывался автоматически даже в случае ошибок во время выполнения других операций, в языке Python есть блок **with**. Его назначение шире, но в нашем случае он даёт закрыть файл после выхода из блока.

Его синтаксис такой:

```
with open('files/Толстой.txt', 'rt') as f:
    read_data = f.read()
print(read_data[:100])
print(f.closed)

------------------------------------------------------------

  -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages,
des
True
```

На этом мы заканчиваем изучать базовые возможности Python по работе с файлами.