**Доступ к файлам в Python**

Чаще всего разработчики в своей работе сталкиваются с обработкой данных, которые хранятся в файлах. А файлы, в свою очередь, обычно хранятся на устройствах хранения данных — жестких, оптических, сетевых или твердотельных накопителях.

Легко представить программу, которая сортирует 20 чисел, и столь же легко представить пользователя этой программы, который вводит эти 20 чисел прямо с клавиатуры.

Гораздо сложнее представить ту же самую задачу, когда нужно отсортировать 20 000 чисел. Ну а пользователей, которые могли бы ввести все эти числа без ошибок, просто не существует.

Намного проще представить, что эти цифры хранятся в файле на диске, который считывает программа. Программа сортирует числа и не отправляет их на экран, а создает новый файл и сохраняет там отсортированную последовательность чисел.

Если нужно реализовать простую базу данных, единственный способ сохранить информацию между запусками программы — сохранить ее в файл (или файлы, если база данных более сложная).

В принципе, любая сложная задача в программировании зависит от использования файлов, независимо от того, обрабатываются ли изображения (которые хранятся

**Файловые потоки**

Открытие потока данных не только связано с файлом, но и должно объявлять способ обработки потока. Такое объявление называется режимом открытия файла. Если открытие прошло успешно, программа получит разрешение на выполнение только тех операций, которые соответствуют заявленному режиму открытия. В потоке данных выполняются две основные операции:

■ чтение из потока:части данных извлекаются из файла и помещаются в область памяти, которая управляется программой (например, в переменную);

■ запись в поток: часть данных памяти (например, переменная) передается в файл. Для открытия потока используются три основных режима:

■ режим чтения: поток, открытый в  этом режиме, разрешает только операции чтения; попытка записи в  поток вызовет исключение (которое называется UnsupportedOperation, оно наследует OSError и ValueError и является потомком модуля io); 11 Обработка файлов

■ режим записи: поток, открытый в этом режиме, разрешает только операции записи; попытка прочитать поток вызовет исключение, упомянутое выше;

■ режим обновления: поток, открытый в этом режиме, разрешает и запись, и чтение.

**Открытие потоков**

Открытие потока выполняет функция, которую можно вызвать следующим образом:

stream = open(file, mode = 'r', encoding = None)

Давайте проанализируем ее:

■ имя функции (open) говорит само за себя; если открытие прошло успешно, функция возвращает объект 17 Обработка файлов потока; в противном случае возникает исключение (например, FileNotFoundError если файла, который вы собираетесь прочитать, не существует);

■ первый параметр функции (file) задает имя файла, который будет связан с потоком;

■ второй параметр (mode) задает режим открытия для этого потока; это строка, заполненная последовательностью символов, и каждый из них имеет свое особое значение (скоро разберем подробнее);

■ третий параметр (encoding) указывает тип кодировки (например, UTF-8 при работе с текстовыми файлами).

■ открытие должно быть самой первой операцией, выполняемой в потоке. Примечание: аргументы режима и кодирования могут быть опущены, тогда принимаются значения по умолчанию. Режим открытия по умолчанию — это чтение в текстовом режиме, а кодировка по умолчанию зависит от используемой платформы.

**Открытие потоков: режимы**

■ r: чтение

▷ поток будет открыт в режиме чтения;

▷ файл, связанный с потоком должен существовать

и быть доступным для чтения, в противном случае,

функция open() вызывает исключение.

■ w: запись

▷ поток будет открыт в режиме записи;

▷ файл, связанный с потоком, не обязан существовать. Если его не существует, он будет создан. Если существует, он будет обрезан до нулевой длины (стерт). Если создание невозможно (например, из-за системных разрешений) функция open() вызывает исключение.

■ a: дозапись

▷ поток будет открыт в режиме дозаписи;

▷ файл, связанный с потоком, не обязан существовать. Если его не существует, он будет создан. Если существует, виртуальная записывающая головка будет установлена в конец файла (предыдущее содержимое файла останется без изменений).

■ r+: чтение и обновление

▷ поток будет открыт в режиме чтения и обновления;

▷ файл, связанный с потоком, должен существовать

и быть доступным для записи, в противном случае

функция open() вызывает исключение;

▷ операции чтения и записи разрешены для потока.

■ w+: запись и обновление

▷ поток будет открыт в режиме записи и обновления;

▷ файл, связанный с потоком, не обязан существовать. Если его не существует, он будет создан. Предыдущее содержимое файла остается нетронутым;

▷ операции чтения и записи разрешены для потока.

В Python имеются следующие режимы доступа:

|  |  |
| --- | --- |
| Название | Описание |
| 'r' | открытие на чтение (значение по умолчанию) |
| 'w' | открытие на запись (содержимое файла удаляется, а если его нет, то создается новый) |
| 'x' | открытие файла на запись, если его нет генерирует исключение |
| 'a' | открытие на дозапись (информация добавляется в конец файла) |
| Дополнения | |
| 'b' | открытие в бинарном режиме доступа к информации файла |
| 't' | открытие в текстовом режиме доступа (если явно не указывается, то используется по умолчанию) |
| '+' | открытие на чтение и запись одновременно |

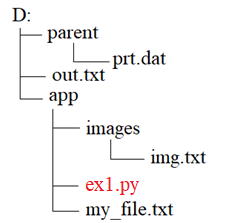
Поэтому сразу перейдем к функции

open(file [, mode=’r’, encoding=None, …])

через которую и осуществляется работа с файлами. Здесь

* file – это путь к файлу вместе с его именем;
* mode – режим доступа к файлу;
* encoding – кодировка файла.

Для начала определимся с понятием «путь к файлу». Представим, что наш файл ex1.py находится в каталоге app:



Тогда, чтобы обратиться к файлу my\_file.txt путь можно записать так:

"my\_file.txt"или "d:\\app\\my\_file.txt" или так: "d:/app/my\_file.txt"

Последние два варианта представляют собой абсолютный путь к файлу, то есть, полный путь, начиная с указания диска. Причем, обычно используют обратный слеш в качестве разделителя: так короче писать и такой путь будет корректно восприниматься как под ОС Windows, так и Linux. Первый же вариант – это относительный путь, относительно рабочего каталога.

В нашем случае мы имеем текстовый файл «myfile.txt», который находится в том же каталоге, поэтому путь можно записать просто указав имя файла:

file = open("myfile.txt")

В результате переменная file будет ссылаться на файловый объект, через который и происходит работа с файлами. Если указать неверный путь, например, так:

file = open("myfile2.txt")

то возникнет ошибка FileNotFoundError. Это стандартное исключение и как их обрабатывать мы с вами говорили на предыдущем занятии. Поэтому, запишем этот критический код в блоке try:

**try**:

    file = open("myfile2.txt")

**except** FileNotFoundError:

**print**("Невозможно открыть файл")

По умолчанию функция open открывает файл в текстовом режиме на чтение. Это режим mode = "r" Если нам нужно поменять режим доступа к файлу, например, открыть его на запись, то это явно указывается вторым параметром функции open:

file = open("out.txt", "w")

Здесь мы имеем три основных режима доступа: на чтение, запись и добавление. И еще три возможных расширения этих режимов, например,

* 'rt' – чтение в текстовом режиме;
* 'wb' – запись в бинарном режиме;
* 'a+' – дозапись или чтение данных из файла.

**Чтение информации из файла**

В чем отличие текстового режима от бинарного мы поговорим позже, а сейчас откроем файл на чтение в текстовом режиме: и прочитаем его содержимое с помощью метода read:

file = open("myfile.txt")

**print**( file.read() )

В результате, получим строку, в которой будет находиться прочитанное содержимое.

Для этого следует воспользоваться именованным параметром encoding и записать метод open вот так:

file = open("myfile.txt", encoding="utf-8" )

Теперь все будет работать корректно.

Далее, в методе read мы можем указать некий числовой аргумент, например,

**print**( file.read(12) )

Тогда из файла будут считаны первые два символа. И смотрите, если мы запишем два таких вызова подряд: то увидим, что при следующем вызове метод read продолжил читать следующие два символа. Почему так произошло?

Дело в том, что у файлового объекта, на который ссылается переменная file, имеется внутренний указатель позиции (file position), который показывает с какого места производить считывание информации.

Когда мы вызываем метод read(12) эта позиция автоматически сдвигается от начала файла на два символа, т.к. мы именно столько считываем. И при повторном вызове read(12) считывание продолжается, т.е. берутся следующие два символа. Соответственно, позиция файла сдвигается дальше. И так, пока не дойдем до конца.

Но мы в Python можем управлять этой файловой позицией с помощью метода

seek(offset[, from\_what])

Например, вот такая запись: будет означать, что мы устанавливаем позицию в начало и тогда такие строчки будут считывать одни и те же первые символы.

**print**( file.read(2) )

file.seek(0)

**print**( file.read(2) )

Если же мы хотим узнать текущую позицию в файле, то следует вызвать метод tell:

pos = file.tell() # считает позицию в байтах

**print**(pos)

Следующий полезный метод – это **readline** позволяет построчно считывать информацию из текстового файла:

s = file.readline()

**print**( s )

Здесь концом строки считается символ переноса ‘\n’, либо конец файла. Причем, этот символ переноса строки будет также присутствовать в строке. Мы в этом можем убедиться, вызвав дважды эту функцию:

Здесь в консоли строчки будут разделены пустой строкой. Это как раз из-за того, что один перенос идет из прочитанной строки, а второй добавляется самой функцией print. Поэтому, если их записать вот так, то вывод будет построчным с одним переносом.

**print**( file.readline(), end="" )

**print**( file.readline(), end="" )

Если нам нужно последовательно прочитать все строчки из файла, то для этого обычно используют цикл for следующим образом:

**for** line **in** file:

**print**( line, end="" )

Этот пример показывает, что объект файл является итерируемым и на каждой итерации возвращает очередную строку.

Или же, все строчки можно прочитать методом и тогда переменная s будет ссылаться на список с этими строками:

s = file.readlines()

**print**( s )

Однако этот метод следует использовать с осторожностью, т.к. для больших файлов может возникнуть ошибка нехватки памяти для хранения полученного списка.

По сути это все методы для считывания информации из файла. И, смотрите, как только мы завершили работу с файлом, его следует закрыть. Для этого используется метод close: file.close()

Конечно, прописывая эту строчку, мы не увидим никакой разницы в работе программы. Но, во-первых, закрывая файл, мы освобождаем память, связанную с этим файлом и, во-вторых, у нас не будет проблем в потере данных при их записи в файл. А, вообще, **лучше просто запомнить:** после завершения работы с файлом, его нужно закрыть. Причем, организовать программу лучше так:

**try**:

    file = open("myfile.txt")

**try**:

        s = file.readlines()

**print**( s )

**finally**:

        file.close()

**except** FileNotFoundError:

**print**("Невозможно открыть файл")

Мы здесь создаем вложенный блок try, в который помещаем критический текст программы при работе с файлом и далее блок finally, который будет выполнен при любом стечении обстоятельств, а значит, файл гарантированно будет закрыт.

**Запись информации в файл**

Теперь давайте посмотрим, как происходит запись информации в файл. Во-первых, нам нужно открыть файл на запись, например, так: и далее вызвать метод write:

file = open("out.txt", "w")

file.write("Hello World!")

В результате у нас будет создан файл out.txt со строкой «Hello World!». Причем, этот файл будет располагаться в том же каталоге, что и файл с текстом программы на Python.

Далее сделаем такую операцию: запишем метод write следующим образом:

file.write("Hello")

И снова выполним эту программу. Смотрите, в нашем файле out.txt прежнее содержимое исчезло и появилось новое – строка «Hello». То есть, когда мы открываем файл на запись в режимах w, wt, wb, то прежнее содержимое файла удаляется. Вот этот момент следует всегда помнить.

Теперь посмотрим, что будет, если вызвать метод write несколько раз подряд:

    file.write("Hello1")

   file.write("Hello2")

    file.write("Hello3")

Смотрите, у нас в файле появились эти строчки друг за другом. То есть, здесь как и со считыванием: объект file записывает информацию, начиная с текущей файловой позиции, и автоматически перемещает ее при выполнении метода write.

Если мы хотим записать эти строчки в файл каждую с новой строки, то в конце каждой пропишем символ переноса строки:

   file.write("Hello1**\n**")

   file.write("Hello2**\n**")

file.write("Hello3**\n**")

Далее, для дозаписи информации в файл, то есть, записи с сохранением предыдущего содержимого, файл следует открыть в режиме ‘a’:

file = open("out.txt", "a")

Тогда, выполняя эту программу, мы в файле увидим уже шесть строчек. И смотрите, в зависимости от режима доступа к файлу, мы должны использовать или методы для записи, или методы для чтения. Например, если вот здесь попытаться прочитать информацию с помощью метода read:

file.read()

то возникнет ошибка доступа. Если же мы хотим и записывать и считывать информацию, то можно воспользоваться режимом a+:

file = open("out.txt", "a+")

Так как здесь файловый указатель стоит на последней позиции, то для считывания информации, поставим его в самое начало:

   file.seek(0)

**print**( file.read() )

А вот запись данных всегда осуществляется в конец файла.

Следующий полезный метод для записи информации – это **writelines**:

file.writelines(["Hello1**\n**", "Hello2**\n**"])

Он записывает несколько строк, указанных в коллекции. Иногда это бывает удобно, если в процессе обработки текста мы имеем список и его требуется целиком поместить в файл.

**Чтение и запись в бинарном режиме доступа**

Что такое бинарный режим доступа? Это когда данные из файла считываются один в один без какой-либо обработки. Обычно это используется для сохранения и считывания объектов. Давайте предположим, что нужно сохранить в файл вот такой список:

books = [

("Евгений Онегин", "Пушкин А.С.", 200),

("Муму", "Тургенев И.С.", 250),

("Мастер и Маргарита", "Булгаков М.А.", 500),

("Мертвые души", "Гоголь Н.В.", 190)

]

Откроем файл на запись в бинарном режиме:

file = open("out.bin", "wb")

Далее, для работы с бинарными данными подключим специальный встроенный модуль pickle:

**import** pickle

И вызовем него метод dump:

pickle.dump(books, file)

Все, мы сохранили этот объект в файл. Теперь прочитаем эти данные. Откроем файл на чтение в бинарном режиме: и далее вызовем метод load модуля pickle: Все, теперь переменная bs ссылается на эквивалентный список:

file = open("out.bin", "rb")

bs = pickle.load(file)

**print**( bs )

Аналогичным образом можно записывать и считывать сразу несколько объектов.

**import** pickle

book1 = ["Евгений Онегин", "Пушкин А.С.", 200]

book2 = ["Муму", "Тургенев И.С.", 250]

book3 = ["Мастер и Маргарита", "Булгаков М.А.", 500]

book4 = ["Мертвые души", "Гоголь Н.В.", 190]

**try**:

    file = open("out.bin", "wb")

**try**:

        pickle.dump(book1, file)

        pickle.dump(book2, file)

        pickle.dump(book3, file)

        pickle.dump(book4, file)

**finally**:

        file.close()

**except** FileNotFoundError:

**print**("Невозможно открыть файл")

А, затем, считывание в том же порядке:

file = open("out.bin", "rb")

    b1 = pickle.load(file)

    b2 = pickle.load(file)

    b3 = pickle.load(file)

    b4 = pickle.load(file)

**print**( b1, b2, b3, b4, sep="**\n**" )

Вот так в Python выполняется запись и считывание данных из файла.

## Задания для самоподготовки

1. Выполните считывание данных из текстового файла через символ и записи прочитанных данных в другой текстовый файл. Прочитывайте так не более 100 символов.

2. Пользователь вводит предложение с клавиатуры. Разбейте это предложение по словам (считать, что слова разделены пробелом) и сохраните их в столбец в файл.

3. Пусть имеется словарь:

d = {"house": "дом", "car": "машина",  
     "tree": "дерево", "road": "дорога",  
     "river": "река"}

Необходимо каждый элемент этого словаря сохранить в бинарном файле как объект. Затем, прочитать этот файл и вывести считанные объекты в консоль.

**Менеджеры контекстов. Оператор with**

Как уже сказано, важно своевременно закрывать файлы с помощью метода close(). Закрытие файлов вручную, а также отдача закрытия на откуп среде исполнения, обладают существенным недостатком: если между открытием файла и его закрытием произойдёт ошибка, в лучшем случае файл окажется открыт слишком долго, а в худшем случае часть данных не сохранится.

Хочется иметь возможность автоматически закрывать файл сразу после окончания работы с ним и осуществлять закрытие даже при возникновении ошибки. Файловые объекты уже умеют работать в таком режиме, но для этого их нужно использовать как **менеджеры контекста**.

Менеджер контекста— объект, реализующий одноименный протокол.

Весь код в теле with-блока работает "в контексте". Чаще всего контекст подразумевает выделение некоего ресурса, например, файла. По выходу из контекста ресурс автоматически освобождается, даже если при выполнении блока возникло исключение.

Как только закончится код, оформленный с отступами в with (аналогичные отступы в циклах или функциях), это будет означать, что контекст закончился, и Python автоматически закроет файл.

Приведенный ниже код:

file = open('languages.txt', 'r', encoding='utf-8')

for line in file:

print(line)

file.close() # ручное закрытие файла

print('Файл закрыт')

можно переписать в виде:

with open('languages.txt', 'r', encoding='utf-8') as file:

for line in file:

print(line)

# автоматическое закрытие файла

print('Файл закрыт')

Обратите внимание: при использовании менеджера контекста не требуется использовать метод close(). При работе с файлами желательно всегда использовать менеджер контекста. Это делает программу надежнее.

Благодаря блоку finally мы гарантированно закрываем файл, даже если в блоке try возникло какое-либо исключение. Но, если воспользоваться файловым менеджером контекста, то программа принимает вид:

**try**:

**with** open("myfile.txt") **as** fp:

**for** t **in** fp:

**print**(t)

**except** Exception **as** e:

**print**(e)

**Вложенные менеджеры контекстов**

При необходимости, менеджеры контекстов можно вкладывать друг в друга. Например, при работе с файлами, можно выполнить такое вложение:

**try**:

**with** open("myfile.txt") **as** fin:

**with** open("out.txt", "w") **as** fout:

**for** line **in** fin:

                fout.write(line)

**except** Exception **as** e:

**print**(e)

Работает все очевидным образом. Сначала завершается (отрабатывает) вложенный менеджер, а затем, внешний (первый). Во всем остальном логика работы такого вложения сохраняется.

**Запись в файл с помощью функции print()**

Для записи данных в файл можно также использовать встроенную функцию print(). Для этого нужно передать ей еще один именованный аргумент file, указывающий на открытый файл. При этом функция print() автоматически добавляет переход на новую строку.

Приведенный ниже код:

with open('philosophers.txt', 'w', encoding='utf-8') as output:

print('Джoн Локк', file=output)

print('Дэвид Хьюм', file=output)

print('Эдмyнд Берк', file=output)

создает файл philosophers.txt с содержимым:

Джoн Локк

Дэвид Хьюм

Эдмyнд Берк

Мы можем использовать всю мощность встроенной функции print() для форматирования выводимого текста.

Приведенный ниже код:

with open('philosophers.txt', 'w', encoding='utf-8') as output:

print('Джoн Локк', 'Дэвид Хьюм', 'Эдмyнд Берк', sep='\*\*\*', file=output)

создает файл philosophers.txt с содержимым:

Джoн Локк\*\*\*Дэвид Хьюм\*\*\*Эдмyнд Берк

Не забывайте, что файловые методы write() и writelines() не добавляют переход на новую строку, поэтому для перехода на новую строку в файле необходимо явно добавить символ '\n'.