top

КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Основы программирования на языке Python

Урок 8

Обработка файлов. Работа с реальными файлами

Contents

Обработка файлов..... 4
Доступ к файлам в Python 4
Имена файлов 5
Файловые потоки.....10
Файловые дескрипторы 12
Открытие потоков 16
Открытие потоков: режимы 17
Выбор текстового и бинарного режимов 19
Первое открытие потока.....19
Предварительно открытые потоки 20
Закрытие потоков 22
Диагностика проблем потока 23

Работа с реальными файлами..... 27
Обработка текстовых файлов.....27
Обработка текстовых файлов: readline().....31
Работа с текстовыми файлами: write() 34
Что такое bytearray?.....37
Как читать байты из потока.....40
Копирование файлов —
простой и функциональный инструмент.....43

Обработка файлов

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

в файлах), умножаются ли матрицы (которые хранятся в файлах) или рассчитываются зарплата и налоги (путем считывания данных, хранящихся в файлах).

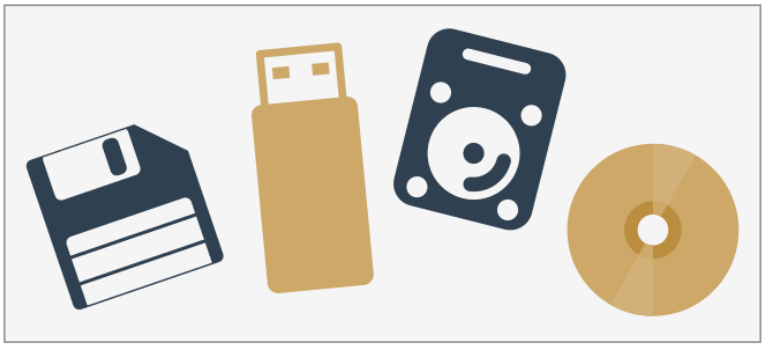

Рисунок 1

Может возникнуть вопрос, почему мы так долго ждали, чтобы поговорить об этих проблемах.

Ответ очень прост — способ доступа и обработки файлов в Python реализован с использованием согласованного набора объектов. И сейчас идеальный момент, чтобы поговорить об этом.

Имена файлов

Разные операционные системы обрабатывают файлы по-разному. Например, в операционных системах Windows и Unix/Linux используются разные соглашения об именах.

Если мы напишем каноническое имя файла (имя, которое однозначно определяет местоположение файла, независимо от его уровня в дереве каталогов), то увидим, что в Windows и в Unix/Linux эти имена выглядят по-разному (рис. 2).

Рисунок 2

Как видите, Unix-подобные системы не используют букву диска (например, «C:») и все каталоги растут из одного корневого каталога с именем «/», а Windows распознает корневой каталог как «\».

Кроме того, имена системных файлов в Unix/Linux чувствительны к регистру. Windows сохраняет регистр букв в имени файла, но сам регистр для него не имеет значения.

То есть эти две строки:

```
ThisIsTheNameOfTheFile
```

и:

```
thisisthenameofthefile
```

описывают два разных файла в Unix/Linux, но Windows воспринимает их как одно и то же имя одного файла.

Главное и самое яркое отличие в том, что приходится использовать *два разных разделителя для имен каталогов:* «\» в Windows и «/» в Unix/Linux.

Эта разница не очень важна для обычного пользователя, но *очень важна при написании программ на Python*.

Чтобы понять, почему это так, попробуйте вспомнить очень специфическую роль, которую играет \ внутри строк Python.

Предположим, вам нужен конкретный файл, расположенный в каталоге *dir*, и этот файл называется «file».

Также предположим, что вы хотите задать строку, содержащую имя файла.

В Unix-подобных системах это может выглядеть следующим образом:

```
name = "/dir/file"
```

Но если попытаться задать строку для Windows следующим образом:

```
name = "\dir\file"
```

то получится неприятный сюрприз: либо Python выдаст ошибку, либо программа выполнится так, будто имя файла как-то искажено.

На самом деле, это вполне очевидный и естественный результат. Python использует символ «\» для экранирования символов (\n).

Это означает, что имена файлов в Windows должны быть записаны следующим образом:

```
name = "\\dir\\file"
```

К счастью, есть еще одно решение. Python умеет преобразовывать косые черты в обратную косую черту каждый раз, когда обнаруживает, что этого требует ОС.

Это означает, что следующие назначения:

```
name = "/dir/file"

name = "c:/dir/file"
```

будут работать и в Windows.

Любая программа, написанная на Python (и не только на Python, поскольку это соглашение применяется практически ко всем языкам программирования), взаимодействует с файлами не напрямую, а через некоторые абстрактные объекты, которые по-разному называются в разных языках или средах. Наиболее часто используемые термины это дескрипторы (handles) или потоки данных (streams) (мы будем использовать их как синонимы).

Если в арсенале програмиста имеется более или менее богатый набор функций/методов, то он может использовать механизмы ядра операционной системы для выполнения определенных операций с потоками данных, которые влияют на реальные файлы.

Таким образом, можно организовать доступ к любому файлу, даже если имя файла неизвестно на момент написания программы (рис. 3).

Операции, выполняемые с абстрактным потоком данных, отражают действия, связанные с физическим файлом.

Чтобы связать поток данных с файлом, необходимо выполнить явную операцию.

Операция связывания потока данных с файлом называется открытием файла, а прерывание этой связи называется закрытием файла.

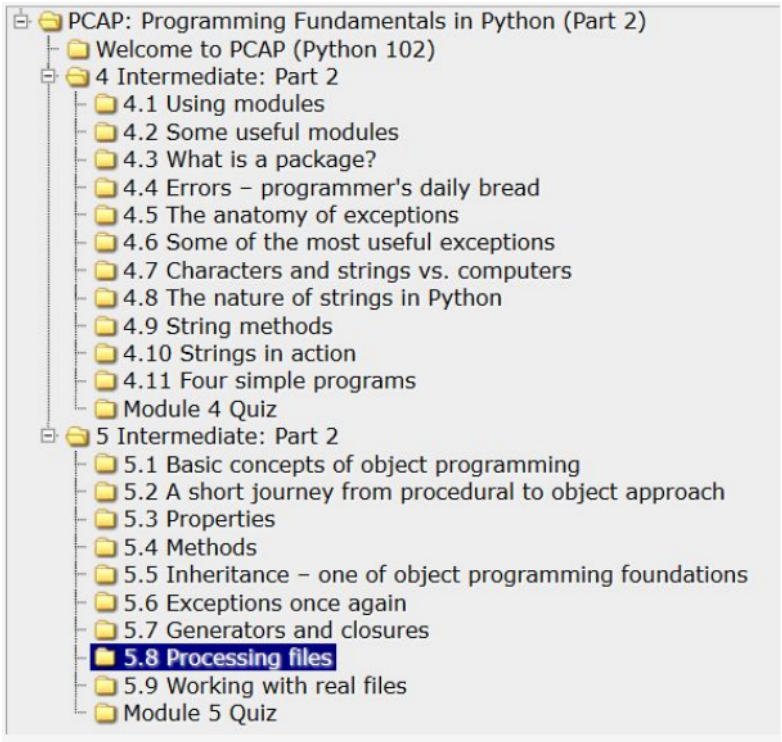

Рисунок 3

Отсюда можно сделать вывод, что самая первая операция, выполняемая в потоке данных, — всегда open, а последняя — close. По сути, программа может свободно управлять потоком данных между этими двумя событиями и обрабатывать связанный файл.

Эта свобода, конечно, ограничена физическими характеристиками файла и способом, которым файл был открыт.

Стоит повторить, что у вас может не получиться открыть поток данных, и это может произойти по нескольким причинам. Наиболее распространенной является отсутствие файла с указанным именем.

А еще бывает так, что физический файл существует, но программе запрещено его открывать. Или, например, программа открыла слишком много потоков данных, а конкретная операционная система не разрешает одновременно открывать более n количества файлов (например, более 200).

Хорошо написанная программа должна обнаруживать эти сбои и реагировать соответствующим образом.

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);

- *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*.

Прежде чем мы обсудим, как управлять потоками, мы должны кое-что объяснить. *Поведение потока очень похоже на принцип работы магнитофона*.

Когда вы что-то читаете из потока, виртуальная головка перемещается по потоку в соответствии с количеством байтов, переданных из потока.

Когда вы записываете что-то в поток, одна и та же головка движется вдоль потока, записывая данные из памяти.

Всякий раз, когда мы говорим о чтении и записи в поток, попытайтесь представить эту аналогию. В книгах по программированию этот механизм называется *текущая позиция файла*, и мы тоже будем использовать этот термин.

Рисунок 4

Теперь нужно показать вам объект, отвечающий за представление потоков в программах.

Файловые дескрипторы

Python предполагает, что *каждый файл спрятан за объектом соответствующего класса*.

Сразу же возникает вопрос, как интерпретировать слово «соответствующий».

Файлы могут быть обработаны разными способами. Некоторые из них зависят от содержимого файла, другие — от целей программиста.

В любом случае, разные файлы могут требовать разных наборов операций и вести себя по-разному.

Объект соответствующего класса *создается при открытии файла и уничтожается при закрытии*.

Между этими двумя событиями объект указывает, какие операции должны выполняться в конкретном потоке. Разрешенные операции зависят от *способа открытия файла*.

В целом, этот объект происходит от одного из классов, показанных ниже:

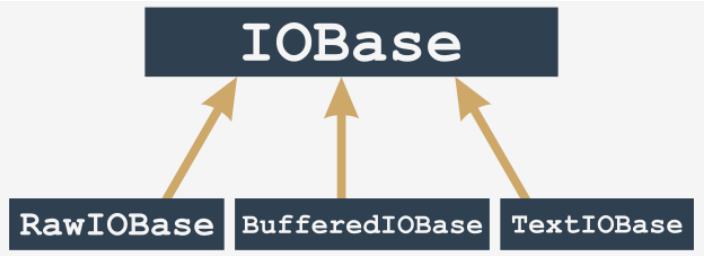

Рисунок 5

> **Примечание:** *конструкторы никогда не используются для вызова этих объектов. Единственный способ их получить — это вызвать функцию open().*

Функция анализирует предоставленные аргументы и автоматически создает требуемый объект.

Чтобы *закрыть объект, нужно вызвать метод* close().

Вызов разорвет соединение с объектом и файлом и удалит объект.

Пока что мы будем заниматься только потоками, которые представлены объектами BufferIOBase и TextIOBase. Скоро вы поймете, почему.

Из-за типа содержимого потока, все потоки делятся на текстовые и двоичные.

Текстовые потоки структурированы в строки; то есть они содержат типографские символы (буквы, цифры, знаки пунктуации и т.д.), расположенные в строках. Это видно невооруженным глазом при просмотре содержимого файла в редакторе.

Чаще всего этот файл записывается (или читается) символ за символом или строка за строкой.

Двоичные потоки содержат не текст, а последовательность байтов любого значения. Эта последовательность может быть, например, исполняемой программой, изображением, аудио- или видео-файлом, файлом базы данных и т.д.

Поскольку эти файлы не содержат строк, операции чтения и записи устанавливают связи с данными любого размера. Следовательно, данные читаются/записыва-

ются байт за байтом или блок за блоком. Размер блока обычно колеблется от одного до произвольно выбранного значения.

Отсюда вытекает неочевидная проблема. В Unix/Linux-системах концы строки отмечены одним символом LF (код ASCII: 10), который обозначается в Python как \n.

Другие операционные системы, особенно наследницы древней операционной системы CP/M (это относится и к системам семейства Windows), используют другое соглашение: конец строки отмечен парой символов: CR и LF (ASCII коды: 13 и 10), которые можно кодировать как \r\n.

Рисунок 6

Эта двусмысленность может вызвать разные неприятные последствия.

Если вы создаете программу обработки текстового файла, которая написана для Windows, то концы строк распознаются по символам \r\n, но та же самая программа будет совершенно бесполезна в среде Unix/Linux, и наоборот — программа, написанная для систем Unix/Linux, может быть бесполезна в Windows.

О программах, которые нельзя использовать в разных средах, говорят, что они «не поддаются портированию (non-portable)».

Особенность программы, которая позволяет ей выполняться в разных средах, называется «портируемостью (portability)». А сама программа, наделенная такой особенностью, называется «переносимой (портативной) программой (portable program)».

Поскольку проблема переносимости всегда стояла остро, было решено организовать все таким образом, чтобы избавить разработчика от необходимости вникать в эти тонкости.

Саму проблему решили на уровне классов, которые отвечают за чтение и запись символов в потоке. Вот как это работает:

- когда поток открыт и есть рекомендация обработать данные в связанном файле как текст (или такой рекомендации нет), то он *переключится в текстовый режим*;
- во время чтения/записи строк в среде Unix ничего особенного не происходит, но когда те же операции выполняются в среде Windows, происходит процесс, который называется *преобразованием символов новой строки*: когда вы читаете строку из файла, каждая пара символов \r\n заменяется одним символом \n

и наоборот. Во время операций записи каждый символ \n заменяется парой символов \r\n;

- этот механизм абсолютно *понятен* программе, поэтому программист может писать ее так, как если бы она предназначалась только для обработки текстовых файлов в Unix/Linux, потому что этот исходный код и в среде Windows будет работать правильно;
- когда поток открыт, и есть соответствующая рекомендация, его содержимое принимается как есть, *без преобразований* — ни один байт не добавляется и не удаляется.

Открытие потоков

Открытие потока выполняет функция, которую можно вызвать следующим образом:

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

Рисунок 7

Давайте проанализируем ее:

- имя функции (open) говорит само за себя; если открытие прошло успешно, функция возвращает объект

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.*

Пора познакомиться с самыми важными и полезными режимами открытия. Готовы?

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.

Выбор текстового и бинарного режимов

Если в конце строки режима стоит буква «b», это означает, что поток должен быть открыт в *двоичном режиме (binary mode)*.

Если в конце строки режима стоит буква «t», поток должен быть открыт в *текстовом режиме (text mode)*.

Текстовый режим — поведение, которое подразумевается по умолчанию, если в строке нет спецификатора двоичного/текстового режима.

Наконец, успешное открытие файла установит текущую позицию файла (виртуальную головку чтения/записи) перед первым байтом файла, *если выбранный режим — не* a, и после последнего байта файла**, е***сли выбранный режим —* a.

Текстовый режим	Двоичный режим	Описание
rt	rb	чтение
wt	wb	запись
at	ab	дозапись
r+t	r+b	чтение и обновление
w+t	w+b	чтение и обновление

Файл также можно открыть для эксклюзивного создания. За это отвечает режим открытия x. Если файл уже существует, функция open() вызовет исключение.

Первое открытие потока

Представьте, что мы хотим разработать программу, которая читает содержимое текстового файла с именем: *C:\Users\User\Desktop\file.txt*.

Как открыть этот файл для чтения? Вот соответствующий фрагмент кода:

```
try:
    stream = open("C:\Users\User\Desktop\file.txt", "rt")
    # здесь происходит обработка
    stream.close()

except Exception as exc:
    print("Cannot open the file:", exc)
```

Что тут происходит?

- мы открываем блок try-except, поскольку хотим мягко обрабатывать ошибки выполнения;
- используем функцию open(), чтобы попытаться открыть указанный файл (обратите внимание на то, как мы указали имя файла).
- режим открытия определяется как текст для чтения (так как *текстовый режим является настройкой по умолчанию*, мы можем не указывать букву «t» в строке режима).
- в случае успеха мы получаем объект от функции open() и присваиваем ее переменной потока;
- если функция open() завершается с ошибкой, мы обрабатываем эту ошибку и выводим полную информацию о ней (всегда полезно знать, что именно произошло).

Предварительно открытые потоки

Ранее мы говорили, что любая потоковая операция должна начинаться с вызова функции open(). Но из этого правила есть три четко определенных исключения.

При запуске программы три потока уже открыты и не требуют дополнительной подготовки. Более того, программа может использовать эти потоки явно, если вы импортируете модуль sys:

```
import sys
```

потому что объявление этих трех потоков происходит именно здесь.

Названия этих потоков: sys.stdin, sys.stdout, и sys.stderr. Давайте проанализируем их:

- **sys.stdin**
  - stdin (*standard input* — стандартный ввод)
  - поток stdin обычно связан с клавиатурой, предварительно открыт для чтения и рассматривается как основной источник данных для запущенных программ;
  - хорошо известная функция input () по умолчанию читает данные из stdin.
- **sys.stdout**
  - stdout (*standard output* — стандартный вывод)
  - поток stdout обычно связан с экраном, предварительно открыт для записи и рассматривается как основной объект для вывода данных, запущенных программой;
  - хорошо известная функция print() выводит данные в поток stdout.
- **sys.stderr**
  - stderr (*standard error output* — стандартный поток ошибок)

- поток stderr обычно связан с экраном, предварительно открыт для записи и рассматривается как основное место, куда запущенная программа должна отправлять информацию об ошибках, возникших при ее работе;
- мы не представили метода для отправки данных в этот поток (но обещаем это сделать в ближайшее время).
- разделение потоков на stdout (полезные результаты, полученные программой) и stderr (сообщения об ошибках, безусловно полезные, но они не выводят результаты) дает возможность перенаправить эти два типа информации разным получателям. Более подробное обсуждение этого вопроса выходит за рамки нашего курса. Руководство по операционной системе даст вам больше информации по этим вопросам.

Закрытие потоков

Последняя операция, выполняемая в потоке (за исключением потоков stdin, stdout, и stderr, которые этого не требуют), — это операция *закрытия потоков*.

За это отвечает метод, которые вызывается из объекта открытого потока: stream.close().

- название функции (close()) прямо указывает на то, что именно она делает.
- функция не ожидает никаких аргументов; поток не нужно открывать.
- функция ничего не возвращает, но генерирует исключение IOError в случае ошибки;
- большинство разработчиков считает, что функция close() всегда выполняется успешно, поэтому нет не-

обходимости проверять, правильно ли она выполнила свою задачу.

Но это не совсем так. Если поток был открыт для записи, а затем была выполнена серия операций записи, то может оказаться, что данные, отправленные в поток, еще не были переданы на физическое устройство (из-за механизма, который называется *кэширование* или *буферизация*). Поскольку закрытие потока требует очистки буферов, может случиться так, что очистка завершится неудачей и функция close() тоже завершится с ошибкой.

Мы уже упоминали о сбоях в функциях, которые работают с потоками, но не сказали ни слова о том, как именно можно определить причину сбоя.

А возможность понять, в чем дело, есть и обеспечивается одним из компонентов потоков исключения, о котором мы как раз и собираемся вам рассказать.

Диагностика проблем потока

У объекта IOError есть свойство errno (то есть error number — номер ошибки), к которому можно получить доступ следующим образом:

```
try:
    # далее — потоковые операции
    except IOError as exc:
    print(exc.errno)
```

Значение атрибута errno можно сравнить с одной из готовых символических констант, установленных в модуле errno.

Давайте посмотрим на избранные *константы, которые помогают обнаруживать ошибки в потоках*:

```
errno.EACCES → Permission denied
```

Эта ошибка возникает при попытке открыть для записи файл с атрибутом read only.

```
errno.EBADF → Bad file number
```

Эта ошибка возникает при попытке, например, работать с неоткрытым потоком.

```
errno.EEXIST → File exists
```

Эта ошибка возникает при попытке, например, переименовать файл и задать его старое имя вместо нового.

```
errno.EFBIG → File too large
```

Эта ошибка возникает при попытке создать файл, размер которого превышает максимально допустимый для этой операционной системы.

```
errno.EISDIR → Is a directory
```

Эта ошибка возникает при попытке обращаться с именем каталога как с именем обычного файла.

```
errno.EMFILE → Too many open files
```

Эта ошибка возникает при попытке одновременно открыть больше потоков, чем допустимо для этой операционной системы.

```
errno.ENOENT → No such file or directory
```

Эта ошибка возникает при попытке получить доступ к несуществующему файлу/каталогу.

```
errno.ENOSPC → No space left on device
```

Эта ошибка возникает, когда на носителе нет свободного места.

Полный список намного длиннее (он также включает некоторые ошибки, не связанные с обработкой потока).

Если вы очень осторожный программист, вам может показаться необходимым использовать последовательность операторов, аналогичную приведенной ниже:

```
import errno

try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # здесь происходит фактическая обработка
    s.close()

except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file doesn't exist.")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files.")
    else:
        printf("The error number is:", exc.errno)
```

К счастью, есть функция, которая может здорово упростить обработку кодов ошибок. Она называется strerror() и является предком модуля os. Она принимает только один аргумент — номер (код) ошибки.

Она работает просто: вы даете номер ошибки и получаете строку, описывающую значение ошибки.

Примечание: *если вы передадите код ошибки, которого не существует, (число, которое не связано с ошибкой), функция вызовет исключение ValueError.*

Теперь мы можем упростить код следующим образом:

```
from os import strerror
try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # здесь происходит фактическая обработка
    s.close()
except Exception as exc:
    print("The file could not be opened:",
          strerror(exc.errno));
```

Хорошо. Теперь пришло время разобраться с текстовыми файлами и ознакомиться с основными приемами обработки таких файлов.

Работа с реальными файлами

Обработка текстовых файлов

На этом уроке мы подготовим простой текстовый файл с простым содержимым.

Наша цель — показать вам основные методы, которые используются для *чтения содержимого файла* с дальнейшей обработкой.

Обработка будет очень простой — надо будет скопировать содержимое файла на консоль и посчитать все символы, которые прочитала программа.

Но помните, что в нашем понимании текстовый файл — это файл, который может содержать только текст, без каких-либо дополнительных украшений (форматирование, разные шрифты и т.д.).

Вот почему не стоит работать в продвинутых текстовых редакторах вроде *MS Word, LibreOffice* и др. Используйте самую простую программу, которую найдете в своей ОС: *Notepad*, *vim*, *gedit* и т.д.

Если в ваших текстовых файлах есть буквы алфавитов, которые не входят стандартную кодировку ASCII, могут потребоваться дополнительные настройки. В вызове функции open() должен быть аргумент, обозначающий конкретную кодировку.

Например, если вы используете ОС Unix/Linux, настроенную на использование UTF-8 в качестве обще-

системного параметра, функция open() может выглядеть следующим образом:

```
stream = open('file.txt', 'rt', encoding='utf-8')
```

где аргумент кодировки должен быть установлен в значение, которое является строкой, представляющей правильную кодировку текста (здесь — UTF-8).

Обратитесь к документации по вашей ОС, чтобы найти имя кодировки, соответствующее вашей среде.

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

Для наших экспериментов с обработкой файлов мы будем использовать предварительно загруженный набор файлов (например, *tzop.txt* или *text.txt*), с которыми вы сможете работать. Если вы будете работать с файлами локально на вашем компьютере, мы настоятельно рекомендуем использовать для этого IDLE.

Чтение содержимого текстового файла может быть выполнено несколькими способами, ни один из них не лучше и не хуже другого. Вам решать, какой из них вы выберете. С одними будет совсем просто, над другими придется покорпеть. Будьте гибкими. Не бойтесь менять свои предпочтения.

Самый базовый метод — это функция read(), которую вы уже видели в действии на предыдущем уроке. Если говорить о текстовом файле, тот тут функция может:

- прочитать нужное количество символов (даже если он один) из файла и вернуть их в виде строки;
- прочитать все содержимое файла и вернуть его в виде строки;

- если читать больше нечего (виртуальная считывающая головка достигает конца файла), функция возвращает пустую строку.

Мы начнем с самого простого варианта и используем файл *text.txt*. Файл имеет следующее содержимое:

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

Теперь посмотрите на код ниже, и давайте проанализируем его.

```
1  from os import strerror
2
3  try:
4      cnt = 0
5      s = open('text.txt', "rt")
6      ch = s.read(1)
7      while ch != '':
8          print(ch, end='')
9          cnt += 1
10         ch = s.read(1)
11     s.close()
12     print("\n\nCharacters in file:", cnt)
13 except IOError as e:
14     print("I/O error occurred: ", strerr(e.errno))
```

Процедура довольно проста:

- используйте механизм try-except и откройте файл с заданным именем (в нашем случае, *text.txt*).
- попробуйте прочитать самый первый символ из файла (ch = s.read(1))

- если у вас получилось (в этом случае проверка условия while даст положительный результат), выведите символ (обратите внимание на аргумент end= — это важно! Нам совершенно не нужен переход на новую строку после каждого символа!);
- обновите счетчик (cnt);
- попытайтесь прочитать следующий символ, и процесс повторяется.

Если вы абсолютно уверены, что длина файла адекватна, и вы можете сразу прочитать весь файл, так и делайте. Воспользуйтесь функцией read(), которая вызывается без аргументов или с аргументом, который равен None.

Помните — чтение терабайтного файла с использованием этого метода может повредить вашу ОС.

Не ждите чудес — память компьютера не резиновая.

Посмотрите на код ниже.

```
1  from os import strerror
2
3  try:
4      cnt = 0
5      s = open('text.txt', "rt")
6      content = s.read()
7      for ch in content:
8          print(ch, end='')
9          cnt += 1
10         ch = s.read(1)
11     s.close()
12     print("\n\nCharacters in file:", cnt)
13 except IOError as e:
14     print("I/O error occurred: ", strerr(e.errno))
```

Что вы о нем думаете?

Давайте проанализируем его:

- открыть файл как раньше;
- читать его содержимое при помощи одного вызова функции read();
- затем обработать текст, прогоняя его через обычный цикл for, и обновить значения счетчика на каждом обороте цикла;

Результат будет точно таким же, как раньше.

Обработка текстовых файлов: readline()

Если вам нужно обработать содержимое файла *как набор строк*, а не кучку символов, то с этим справится метод readline().

Этот метод пытается *прочитать полную строку текста из файла* и в случае успеха возвращает ее в виде строки. В противном случае он возвращает пустую строку.

Это открывает перед вами новые возможности — теперь можно легко считать строки, а не только символы.

Давайте попробуем. Посмотрите на код.

```
1  from os import strerror
2
3  try:
4      ccnt = lcnt = 0
5      s = open('text.txt', 'rt')
6      line = s.readline()
7      while line != '':
8          lcnt += 1
9          for ch in line:
10             print(ch, end='')
11             ccnt += 1
12         line = s.readline()
13     s.close()
```

```
14      print("\n\nCharacters in file:", ccnt)
15      print("Lines in file:      ", lcnt)
16  except IOError as e:
17      print("I/O error occurred:", strerr(e.errno))
```

Как видите, общая идея точно такая же, как в обоих предыдущих примерах.

Другой метод, который обрабатывает текстовый файл как набор строк, а не символов, — это readlines(),

Если метод readlines() вызывается без аргументов, он пытается *прочитать все содержимое файла и возвращает список строк — по одному элементу на строку файла.*

Если вы не уверены, что размер файла достаточно мал и не хотите проверять свою ОС на прочность, можно попытаться убедить метод readlines() прочитать не более указанного количества байтов за раз (возвращаемое значение остается прежним — это список строки).

Не стесняйтесь экспериментировать с примером кода ниже, чтобы понять принцип работы метода readlines().

```
s = open("text.txt")
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
s.close()
```

Максимально допустимый размер входного буфера передается методу в качестве аргумента.

Можно предположить, что readlines() обработает содержимое файла более эффективно, чем readline(), поскольку он потребует меньшего количество вызовов.

Примечание: *если читать из файла нечего, метод возвращает пустой список. Используйте это, чтобы обнаружить конец файла.*

Что касается размера буфера, можно предположить, что его увеличение улучшит производительность ввода, но золотого правила тут нет. Попробуйте найти оптимальные значения самостоятельно.

Посмотрите на этот код.

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readlines(20)
    while len(lines) != 0:
        for line in lines:
            lcnt += 1
            for ch in line:
                print(ch, end='')
                ccnt += 1
        lines = s.readlines(10)
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:     ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Мы изменили его, чтобы показать принцип использования readlines().

Мы решили воспользоваться 15-байтовым буфером. Но это ни в коем случае не рекомендация.

Мы выбрали такое значение, чтобы избежать ситуации, в которой первый вызов метода readlines() прочитает весь файл.

Нам нужно, чтобы методу пришлось поработать дольше и продемонстрировать свои возможности.

В коде есть *два вложенных цикла*: внешний использует результат readlines(), чтобы перебрать его, а внутренний выводит строки символ за символом.

Последний пример демонстрирует очень интересную особенность объекта, который функция open() возвращает в текстовом режиме. Она может вас удивить, ведь этот объект является экземпляром итерируемого класса. Странно? Вовсе нет. Полезно? Еще как!

Протокол итерации, заданный для файлового объекта, очень простой — его метод __next__ просто возвращает следующую строку, прочитанную из файла. Кроме того, этот объект автоматически вызывает функцию close(), когда достигает конца файла.

Посмотрите на код ниже и убедитесь, насколько простым и понятным стал код.

```
1  from os import strerror
2
3  try:
4      ccnt = lcnt = 0
5      for line in open('text.txt', 'rt'):
6          lcnt += 1
7          for ch in line:
8              print(ch, end='')
9              ccnt += 1
10     print("\n\nCharacters in file:", ccnt)
11     print("Lines in file:      ", lcnt)
12 except IOError as e:
13     print("I/O error occurred: ", strerr(e.errno))
```

Работа с текстовыми файлами: write()

Запись в текстовые файлы кажется простой благодаря одному методу, который для этих целей и используется.

Метод называется write(), и он принимает только один аргумент — строку, которая будет передана в открытый файл (не забывайте, что режим открытия должен отражать способ передачи данных — если файл открыт в режиме чтения, то попытка записать в него что-то закончится с ошибкой).

В аргумент функции write() нет символа новой строки , поэтому его нужно добавлять самостоятельно, если файл должен быть заполнен несколькими строками.

Пример ниже показывает очень простой код, который создает файл под названием *newtext.txt*.

> **Примечание:** *режим открытия w гарантирует, что файл будет создан с нуля, даже если он уже существует и содержит данные), а затем помещает в него десять строк.*

```
1  from os import strerror
2
3  try:
4      fo = open('newtext.txt', 'wt') # a new file (newtext.txt) is create
5      for i in range(10):
6          s = "line #" + str(i+1) + "\n"
7          for ch in s:
8              fo.write(ch)
9      fo.close()
10 except IOError as e:
11     print("I/O error occurred: ", strerr(e.errno))
```

Записываемая строка состоит из строчки слов, за которой следует номер строки. Мы решили выводить содержимое строки символ за символом (за это отвечает внутренний цикл for), но вы можете выбрать другой способ.

Мы просто хотим показать вам, что метод write() умеет оперировать отдельными символами.

Код создает файл, который заполнен следующим текстом:

```
line #1
line #2
line #3
line #4
line #5
line #6
line #7
line #8
line #9
line #10
```

Мы рекомендуем вам проверить поведение метода write() локально на вашем компьютере.

Посмотрите на пример.

```
1  from os import strerror
2
3  try:
4      fo = open('newtext.txt', 'wt')
5      for i in range(10):
6          fo.write("line #" + str(i+1) + "\n")
7      fo.close()
8  except IOError as e:
9      print("I/O error occurred: ", strerr(e.errno))
```

Мы изменили предыдущий код, чтобы записывать целые строки в текстовый файл.

Содержимое созданного файла осталось неизменным.

Примечание: *вы можете использовать тот же метод для записи в поток stderr, но не пытайтесь открыть его, потому что он и так всегда неявно открыт.*

Например, если вы хотите отправить строку сообщения в stderrERR, чтобы отличить его от обычного вывода программы, это может выглядеть так:

```
import sys
sys.stderr.write("Error message")
```

Что такое bytearray?

Прежде чем мы начнем говорить о двоичных файлах, стоит разобрать один из *специальных классов, которые Python использует для хранения аморфных данных*.

Аморфные данные — это данные, которые не имеют определенной формы — это просто череда байтов.

Это не означает, что эти байты не могут иметь своего собственного значения или не могут представлять какой-либо полезный объект, например, растровую графику. Самое важное здесь то, что в месте нашего контакта с данными мы не можем или просто не хотим ничего о них знать.

Аморфные данные нельзя сохранить при помощи ранее изученных средств — они не являются ни строками, ни списками. Нужен специальный контейнер, способный обрабатывать такие данные. У Python много таких контейнеров. Один из них — специальный класс bytearray. Как видно из названия, это массив, содержащий (аморфные) байты.

Если вам нужен такой контейнер, например, для того, чтобы прочитать и обработать растровое изображение, то его необходимо создать явно, используя один из доступных конструкторов.

Посмотрите:

```
data = bytearray(100)
```

Такой вызов создает объект bytearray, который способен хранить десять байтов.

Примечание: *такой конструктор заполняет весь массив нулями.*

Bytearray во многом напоминает списки. Например, они изменяемы, являются предметом функции len(), а к любому из их элементов можно получить доступ, используя обычную индексацию.

Есть одно важное ограничение — нельзя устанавливать элементы байтового массива со значением, которое не является целым числом (нарушение этого правила приведет к исключению TypeError) и нельзя присваивать значение, которое не входит в диапазон от 0 до 255 включительно (иначе возникнет исключение ValueError).

Любые элементы байтового массива можно обрабатывать как целочисленные значения, как в примере ниже.

```
1  data = bytearray(10)
2
3  for i in range(len(data)):
4      data[i] = 10 - i
5
6  for b in data:
7      print(hex(b))
```

Примечание: *мы использовали два метода для итерации байтовых массивов и функцию hex(), чтобы видеть элементы, которые выводятся как шестнадцатеричные значения.*

Теперь мы хотим показать вам, как записать байтовый массив в двоичный файл. Двоичный, потому что нам не нужно сохранять его читабельное представление, нам просто нужно побайтово записать точную копию содержимого физической памяти.

Итак, как же нам записать байтовый массив в двоичный файл?

Посмотрите на код.

```
1  from os import strerror
2
3  data = bytearray(10)
4
5  for i in range(len(data)):
6      data[i] = 10 + i
7
8  try:
9      bf = open('file.bin', 'wb')
10     bf.write(data)
11     bf.close()
12 except IOError as e:
13     print("I/O error occurred:", strerr(e.errno))
```

Давайте проанализируем его:

- во-первых, мы инициализируем bytearray с последующими значениями, начиная с 10. Если вы хотите, чтобы содержимое файла легко читалось, замените 10 на что-то вроде ord('a'), чтобы вывести байты, содержащие значения, которые соответствуют алфавитной части кода ASCII (не думайте, что после этого ваш файл станет текстовым. Это все еще двоичный файл, так как он был создан с флажком wb);
- затем мы создаем файл, используя функцию open(). Единственное отличие от предыдущих вариантов со-

стоит в том, что это открытый режим, содержащий флажок b;

- метод write() принимает аргумент (bytearray) и отправляет его (целым) в файл;
- затем поток закрывается обычным способом.

Метод write() возвращает количество успешно записанных байтов.

Если значения отличаются от длины аргументов метода, могут появиться ошибки записи.

В этом случае мы никак не использовали результат, но так бывает не всегда.

Попробуйте запустить код и проанализировать содержимое вновь созданного выходного файла.

Мы используем его на следующем шаге.

Как читать байты из потока

Чтение из двоичного файла требует специального имени метода readinto(), поскольку этот метод не создает новый объект массива байтов, а заполняет ранее созданный объект значениями, взятыми из двоичного файла.

Примечание:

- *метод возвращает количество успешно прочитанных байтов;*
- *метод пытается заполнить все пространство, доступное внутри аргумента. Если в файле больше данных, чем места в аргументе, операция чтения остановится до того, как достигнет конца файла. В противном случае результат метода может указывать на то, что байтовый массив*

был заполнен только фрагментарно (результат покажет это и то, что часть массива, которую прочитанное содержимое не использовало, остается нетронутой).

Посмотрите на полный код ниже:

```
from os import strerror
data = bytearray(10)

try:
    bf = open('file.bin', 'rb')
    bf.readinto(data)
    bf.close()

    for b in data:
        print(hex(b), end=' ')
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Давайте проанализируем его:

- сначала мы открываем файл (тот, который вы создали с помощью предыдущего кода) в режиме rb;
- затем передаем его содержимое в байтовый массив с именем data размером 10 байтов;
- наконец, выводим содержимое байтового массива. Оно отвечает вашим ожиданиям?

Запустите код и проверьте, работает ли он.

Метод read() предлагает альтернативный способ чтения содержимого двоичного файла.

Он вызывается без аргументов и пытается передать все содержимое файла в память, делая его частью созданного объекта байтового класса.

У этого класса есть некоторые сходства с bytearray за исключением одного существенного различия — он неизменяемый.

К счастью, мы всегда можем создать байтовый массив путем получения его начального значения непосредственно из объекта байтов, как здесь:

```
from os import strerror
try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read())
    bf.close()
    for b in data:
        print(hex(b), end=' ')

except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Будьте осторожны — не используйте этот тип чтения, если вы не уверены, что содержимому файла хватит доступной памяти.

Если метод read() вызывается с аргументом, то он указывает максимальное количество байтов для чтения.

Метод пытается прочитать желаемое количество байтов из файла, а длину возвращаемого объекта можно использовать для определения количества фактически прочитанных байтов.

Этот метод можно использовать, например, так:

```
try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read(5))
    bf.close()
```

```
for b in data:
    print(hex(b), end=' ')

except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Примечание: *первые 5 байтов файла были прочитаны, а следующие 5 все еще не обработаны.*

Копирование файлов — простой и функциональный инструмент

Теперь мы объединим все эти свежие знания, добавим к ним несколько новых элементов и напишем реальный код, который будет копировать содержимое файла.

Естественно, мы не собираемся создавать замену командам копирования (copy в *MS Windows*, cp в *Unix/Linux*), наша задача — разработать рабочий инструмент, пусть даже никто не захочет им пользоваться.

Посмотрите на код ниже.

```
1  from os import strerror
2
3  srcname = input("Source file name?: ")
4  try:
5      src = open(srcname, 'rb')
6  except IOError as e:
7      print("Cannot open source file: ", strerror(e.errno))
8      exit(e.errno)
9  dstname = input("Destination file name?: ")
10 try:
11     dst = open(dstname, 'wb')
12 except Exception as e:
13     print("Cannot create destination file: ", strerr(e.errno))
14     src.close()
15     exit(e.errno)
16
17 buffer = bytearray(65536)
```

```
18  total  = 0
19  try:
20      readin = src.readinto(buffer)
21      while readin > 0:
22          written = dst.write(buffer[:readin])
23          total += written
24          readin = src.readinto(buffer)
25  except IOError as e:
26      print("Cannot create destination file: ", strerr(e.errno))
27      exit(e.errno)
28
29  print(total,'byte(s) succesfully written')
30  src.close()
31  dst.close()
```

Давайте проанализируем его:

- *строки с 3 по 8*: запросить у пользователя имя файла для копирования и попробовать открыть его для чтения. Прекратить выполнение программы, если открытие не удалось.

  Примечание: *используйте функцию exit(), чтобы остановить выполнение программы и передать кода завершения в ОС. Любой код завершения, кроме 0, говорит о том, что возникли проблемы. Используйте значение errno, чтобы понять причину возникновения той или иной проблемы;*

- *строки с 9 по 15:* повторить почти то же самое действие, но на этот раз для выходного файла;
- *строка 17:* подготовить фрагмент памяти для передачи данных из исходного файла в целевой. Такую область передачи часто называют буфером, отсюда и название переменной. Размер буфера произвольный, но в этом случае мы решили использовать 64 килобайта. Технически, больший буфер быстрее копирует элементы, так как больший буфер предполагает меньше операций

ввода/вывода, но, на самом деле, всегда есть граница, за которой улучшения уже ни на что не влияют. Проверьте сами, если хотите.

- *строка 18:* посчитать скопированные байты — это счетчик и его начальное значение;
- *строка 20:* попытаться заполнить буфер в самый первый раз;
- *строка 21:* повторять те же действия, пока вы получаете ненулевое число байтов;
- *строка 22:* записать содержимое буфера в выходной файл (примечание: мы использовали фрагмент, чтобы ограничить количество записываемых байтов, так как функция write() всегда пытается заполнить весь буфер);
- *строка 23:* обновить счетчик;
- *строка 24:* прочитать следующий фрагмент файла;
- *строки с 29 по 31:* финальная очистка, и все готово.