

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

Урок 7

Файлы

Содержание

Файлы	3
Файловая система	3
Типы файлов, текстовые и бинарные.....	13
Работа с файлами	17
Менеджер контекста	32
Практические примеры.....	37
Пример 1. Поиск и замена слов в содержимом текстового файла	37
Пример 2. Подсчет количества слов в содержимом текстового файла, которые не являются числами	39
Пример 3. Вывести слова содержимого файла в обратном порядке.....	41
Пример 4. Удаление заданной по номеру строки из файла	44

Файлы

Файловая система

Файл — это именованная область (совокупность данных), расположенная во внешней памяти (на внешнем устройстве), и обладающая следующими характеристиками:

- имя файла (на определенном диске), которое позволяет программам идентифицировать файл и, при необходимости, работать одновременно с несколькими файлами;
- длина файла может быть ограничена только объемом диска.

Имя файла состоит из его названия и расширения (например, *myData.txt* и *myData.dat* — совершенно разные файлы). Чувствительность к регистру при работе с именами файлов зависит от используемой операционной системы, например, Windows верхний и нижний регистр в именах файлов и папок воспринимает как одно и то же, а вот в Linux используется файловая система, которая чувствительна к регистру.

Очень распространенной операцией является получение необходимых данных из файла или сохранение (запись) результатов, полученных в ходе работы программы, в файл. Полное имя файла представляет собой полный путь к каталогу с файлом, включающий также имя данного файла.

При работе с файлами наши данные хранятся не в оперативной памяти (например, как при использовании массивов), а вне ее (например, на жестком диске или флеш-накопителе), что обеспечивает долговременное хранение данных за пределами программы. В случае отключения питания повреждение данных внутри файла не произойдет. Даже если в этот момент программа не выполняла над ними операции.

Размер файла не является фиксированной, т. е. может уменьшаться и увеличиваться.

Файлы обеспечивают хранение данных произвольного типа (текста, графики, видео и звука, исполняемых программ и т. д.)

Для операционной системы файл является такой же важной сущностью, как и для её пользователя, т.к. все данные обязательно должны находиться внутри какого-то файла. Иначе операционная система, и, как следствие, все ее пользователи не будут иметь доступ к этим данным. Работа со всеми устройствами также осуществляется операционной системой с помощью специальных файлов.

Конечно, файлы, с помощью которых происходит взаимодействие операционной системы и устройств компьютера отличаются от файлов с пользовательскими данными. В специальных файлах устройств находятся данные, которые необходимы операционной системе для взаимодействия с такими устройствами как диски, принтеры и т. д.

Независимо от того, какая операционная система используется на компьютере, файлы бывают текстовые

и бинарные (двоичные). Особенности этих категорий файлов и принципы работы с ними мы рассмотрим немного позже.

Также файлы разделяются на исполняемые (программы, которые могут запускаться пользователями, операционной системой или другими программами) и неисполняемые (файлы, в которых хранятся данные). Содержимое неисполняемых файлов данных может изменяться в процессе работы (выполнения) программ (исполняемых файлов).

Все файлы можно классифицировать таким образом:

- **основные** — наличие которых обязательно для функционирования операционной системы или другого программного обеспечения;
- **служебные** — отвечающие за различные конфигурации основных файлов;
- **рабочие** — с содержимым которых работают основные файлы, изменяя и создавая данные;
- **временные** — иногда создаются в процессе работы некоторых основных файлов для хранения промежуточных результатов их работы.

На любом носителе информации (жестком диске или флеш-накопителе) всегда хранится достаточно большое число файлов. На каждом носителе информации (гибком, жестком или лазерном диске) может храниться большое количество файлов. Порядок хранения и способ организации файлов на диске задается файловой системой.

Файловая система является частью операционной системы, которая предоставляет интерфейс для взаимодействия с различными данными (файлами), хранящихся на дисках.

Можно сказать, что в состав файловой системы некоторого носителя информации (диска) входят:

- все файлы на этом диске;
- структуры данных, определяющие способ организации файлов на диске и порядок работы с ними (например, каталоги);
- специальные таблицы, хранящие информацию о распределении свободного и занятого пространства на диске;
- системное программное обеспечение для управления и работы с файлами (создание, удаление файлов, чтение и запись, поиск и т. д.).

Фактически именно файловая система связывает физический носитель информации и программу, которая с ней взаимодействует.

Когда некоторая программа в процессе своей работы обращается к файлу, то единственная (базовая) информация, которая ей нужна — это путь к файлу и имя файла (иногда его атрибуты, дата создания или размер). Эту информацию ей обеспечивает драйвер файловой системы.

По аналогичному принципу происходит и запись данных: программа передает файловой системе базовую информацию о файле, а процесс записи и сохранения обеспечивается файловой системой, согласно ее правил (алгоритмов работы).

Таким образом, файловая система обеспечивает выполнение следующих задач:

- распределение файлов на физическом носителе информации и их каталогизация;
- реализация процессов создания, удаления, чтения, записи, поиска файлов;
- изменение атрибутов файлов: названия, размера, уровня (прав) доступа и т. д.;
- защита файлов и восстановление информации в них в случаях сбоев работы операционной системы.

Практически всегда файлы, расположенные на каком-то носителе информации, упорядочиваются в каталоги (иногда называемые директориями).

Можно сказать, что диск состоит из двух областей: каталог и область, в которой хранятся файлы.

С точки зрения пользователя каталог — это некоторая группа файлов, объединенная самим пользователем или операционной системой по каким-то принципам (например, каталог «*Мои документы*» предназначен для хранения документов текущего пользователя системы). Однако, фактически каталог представляет собой файл, в котором содержится системная информация: список файлов, входящих в состав этого каталога, информация о размещении файлов (начало каждого файла на диске), данные о соответствии файлов их характеристикам.

Информация, хранимая на любом носителе, размещается в кластерах — специальных ячейках. Если размер файл не превышает размер кластера, то файл будет размещен файловой системой в одном кластере. Иначе — файл будет занимать несколько кластеров.

Давайте представим, что наш накопитель (диск) — это книга, у которой все страницы пронумерованы и есть оглавление. Тогда все содержимое книги (кроме оглавления) — это область хранения файлов, оглавление книги — каталог, а страницы книги — это кластера диска.

Как в оглавлении книги напротив каждой главы представлен номер страницы, на которой начинается эта глава, так в каталоге напротив имени каждого файла хранится номер сектора, с которого начинается его размещение на диске. Обычно на дисках хранится очень большое количество файлов, сотни или даже тысячи. И использовать только один уровень каталога очень неэффективно.

На рисунке ниже показан пример одноуровневого каталога.

```
C:  
  Wiki.txt  
  Tornado.jpg  
  Notepad.exe
```

Рисунок 1

В этом случае все файлы расположены в корневом каталоге, т.е. непосредственно на самом диске. Корневым каталогом называется базовый (начальный) каталог в структуре носителя информации, в котором могут располагаться как файлы, так и другие каталоги. Вложенные каталоги, которые хранятся внутри других часто называют подкаталогами.

Каждый накопитель информации имеет собственное логическое имя (традиционно жесткие диски имеют имена C:, D: и т.д., флеш-накопители E:, F:, для данного

примера). Логическое имя устанавливается диску в момент его форматирования.

Имя корневого каталога всегда совпадает с именем диска. Когда количество файлов у пользователя начинает увеличиваться, то процесс обнаружения нужной информации становится трудным и затратным по времени. В такой ситуации полезно распределить все файлы по каталогам согласно тематике содержимого или по каким-то другим признакам. Поэтому более распространенным и удобным способом организации данных является многоуровневая файловая система, в которой внутри каталогов содержатся не только файлы, но и другие каталоги.

В нашей аналогии с книгой такая иерархическая файловая система похожа на оглавление, содержащее название разделов книги, глав внутри них, которые в свою очередь состоят из рассказов.

Иерархическая файловая система имеет древовидную структуру. Корень (начальная вершина дерева) соответствует корневному каталогу. В других вершинах дерева находятся каталоги, которые содержат другие каталоги и файлы.

Корневые каталоги разных дисков могут образовывать отдельные (независимые) деревья (такая организация существует в операционной системе Windows).

```
C:
  \Program files
    \CDEx
      \CDEx.exe
      \CDEx.hlp
      \mprenc.exe
```

```
D:
  \Music
    \ABBA
      \1974 Waterloo
      \1976 Arrival
        \Money, Money, Money.ogg
      \1977 The Album
```

Рисунок 2

Или объединяться в одно (общее для всех корневых дисков) дерево (UNIX-подобные операционные системы)

```
/
  /usr
    /bin
      /arch
      /Is
      /raw
    /lib
      /libhistory.so.5.2
      /libgpm.so.1
  /home
    /lost+found /host.sh /guest
    /Pictures
    /example.png
  /Video
    /matrix.avi
    /news
    /lost_ship.mpeg
```

Рисунок 3

Примечание: в файловых системах операционных систем Windows и UNIX подобных операционных системах используются разные типы «слешей»: обратный слеш «\» для Windows и обычный слеш «/» для UNIX.

Символы «слешей» используются в качестве разделителей между названиями каталогов, которые составляют путь к файлу.

Рассмотрим на примере файла «CDEx.exe» (см. Рис.1). В записи C:\Program files\CDEx\CDEx.exe, где: C:\Program files\CDEx\ — это путь к файлу «CDEx.exe».

Можно сказать, что каждая программа работает в определённом каталоге файловой системы (текущий или рабочий каталог), который является ее «рабочим местом». И доступ к данным программа получает всегда из своего «рабочего места» (часто программы по умолчанию работают с файлами со своего рабочего каталога).

В иерархических файловых системах путь к файлу может быть полным (абсолютным) или относительным.

Полный (абсолютный) путь к файлу указывает на определенное (всегда одно и то же) место в файловой системе. При этом текущее положение пользователя (или программы) в системе ни на что не влияет. Начало полного пути всегда имя корневого каталога.

Относительный путь — это путь к файлу относительно текущего (рабочего) каталога (программы) пользователя.

Относительный путь формируется так же, как и абсолютный, с использованием и перечислением через «/» всех названий каталогов, которые встречаются при проходе к нужному файлу (каталогу).

Допустим, что мы запускаем некоторую программу из каталога */guest*, которая обращается к файлу «*example.png*».

```
/home
  /lost+found
    /host.sh
  /guest
    /Pictures
      /example.png
    /Video
      /matrix.avi
      /news
        /lostship.mpeg
```

Рисунок 4

В этом случае рабочим каталогом нашей программы является каталог */guest* и мы можем получить доступ к нужному нам файлу такой строкой адреса: *guest/Pictures/example.png*, которая и представляет собой относительный путь к файлу «*example.png*» (относительно местоположения пользователя или программы).

Поэтому и в обращении к файлу можно использовать полное (абсолютное) имя файла (состоящее из полного пути к файлу и самого имени файла) или относительное имя.

А теперь давайте рассмотрим ситуацию, что нам нужно не спускаться, а подниматься по дереву каталогов. Например, наша программа находится в каталоге */lost+found*, а нам нужно обратиться к файлу, расположенному на уровень выше, т.е. в каталоге */home*.

Конечно, мы можем использовать абсолютный путь к нужному файлу. Однако для формирования относительного пути также есть способ.

Для обозначения родительского каталога (внутри которого мы находимся) используется комбинация «точка-точка»: «../», а для обозначения текущего «точка»: «./»

Таким образом, находясь в каталоге */home*, мы можем обратиться к файлу в каталоге */lost+found*, используя относительное имя файла: «../*fileName*».

Теперь рассмотрим аналогичный пример для операционной системы Windows. Допустим, что наша программа — это файл «*CDEx.exe*» и ей нужно «обратиться» к файлу, расположенному в каталоге *Program files*.

```
C:
  \Program files
    \CDEx
      \CDEx.exe
      \CDEx.hip
      \mprenc.exe
```

Рисунок 5

В этом случае мы также будем использовать относительное имя файла: «..\..\ *fileName*».

Типы файлов, текстовые и бинарные

Как мы уже знаем, работа со всеми устройствами компьютера осуществляется операционной системой с помощью специальных файлов. Конечно, эти устройства сильно отличаются друг от друга. Однако файловая

система преобразует их в единое абстрактное логическое устройство, называемое **поток**ом.

Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти — буфер. Буфер накапливает байты, и фактическая передача данных выполняется после заполнения буфера. При вводе это дает возможность исправить ошибки, если данные из буфера еще не отправлены в программу (рис. 6).

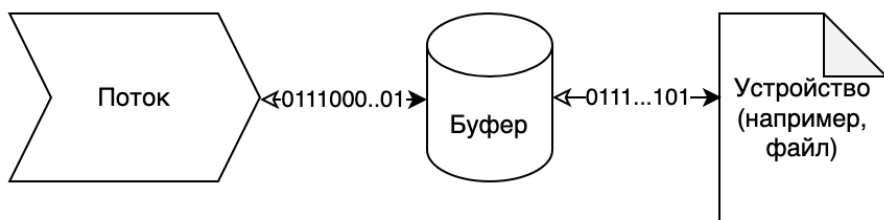


Рисунок 6

Текстовый поток — это последовательность символов. При передаче символов из потока на экран, часть из них не выводится (например, символ возврата каретки, перевода строки).

Двоичный поток — это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве.

Допустим, что наши данные представляют собой строку:

```
s = "Hello\tI likePython\nHi!"
```

При работе с ней, как с текстовый потоком, последовательности «\t» и «\n» будут рассматриваться как escape-последовательности (с особенностями и назначением которых мы уже знакомы). Если же мы обработаем эти данные в бинарном виде, то символ «\» будет воспринят и обработан как обычный символ.

А строка «Hi\xPython» при ее обработке в текстовом виде вообще станет причиной ошибки, т. к. комбинация символов «\x» не является escape-последовательностью, а интерпретатор будет пытаться «понять» это именно так.

Файлы также бывают текстовые и двоичные (бинарные). Однако, вне зависимости от организации данных в файлах, фактически информация в них представлена в двоичном формате, так что это деление условное. В текстовых файлах данные интерпретируются как последовательность символьных кодов. Специальные последовательности используются для указания признака конца строки (как в наших примерах выше).

Двоичные файлы могут использоваться для хранения любых данных (например, изображение в формате JPEG представляет собой двоичный файл, предназначенный для чтения программой по работе с изображениями). Большинство файлов, которые мы видим и с которыми работаем с помощью соответствующего программного обеспечения являются двоичными файлами:

- документы и электронные таблицы: *.pdf*, *.doc*, *.xls* etc.;
- изображения: *.png*, *.jpg*, *.gif*, *.bmp* etc.;
- видео: *.mp4*, *.3gp*, *.mkv*, *.avi* etc.;
- аудио: *.mp3*, *.wav*, *.mka*, *.aac* etc.;

- базы данных: *.mdb*, *.accde*, *.frm*, *.sqlite* etc.;
- архивы: *.zip*, *.rar*, *.iso*, *.7z* etc.;
- исполняемые файлы программ: *.exe*, *.dll*, *.class* etc.

Данные внутри двоичного файла хранятся в виде не-обработанных байтов, которые не могут быть прочитаны человеком.

Конечно, мы можем открыть некоторые двоичные файлы в обычном текстовом редакторе, но мы не можем прочитать содержимое, находящееся внутри файла. Это потому, что все двоичные файлы будут закодированы в двоичном формате, который может «понять» только компьютер (т. е. обработать по каким-то определенным правилам). Если открыть бинарный файл, например, в текстовом редакторе, то человек увидит последовательность непонятных символов, из которых невозможно извлечь смысл. Для работы с такими бинарными файлами нам нужен определенный тип программного обеспечения для их открытия. Например, для открытия файла в формате *.doc* нам нужен Microsoft Word.

Текстовые файлы могут быть открыты и прочитаны человеком в самом обычном текстовом редакторе.

Примеры текстовых файлов:

- web документы, стандарты: *html*, *XML*, *CSS*, *JSON* etc.
- файлы исходных кодов: *c*, *app*, *js*, *py*, *java* etc.
- текстовые документы: *txt*, *tex*, *RTF* etc.
- текстовые представления табличных данных (файлы с разделителями): *csv*, *tsv* etc.
- файлы настроек и конфигураций: *ini*, *cfg*, *reg* etc.

Работа с файлами

Основными операциями при работе с файлами являются следующие четыре:

- открытие файла;
- чтение из файла;
- запись в файл;
- закрытие файла.

В Python для выполнения этих операций есть встроенные функции, находящиеся в модуле *io*.

По умолчанию для работы с файлами используется модуль *io*, т. е. нет необходимости импортировать его перед использованием его функций. Общая схема последовательности работы с файлами представлена на рисунке 7.

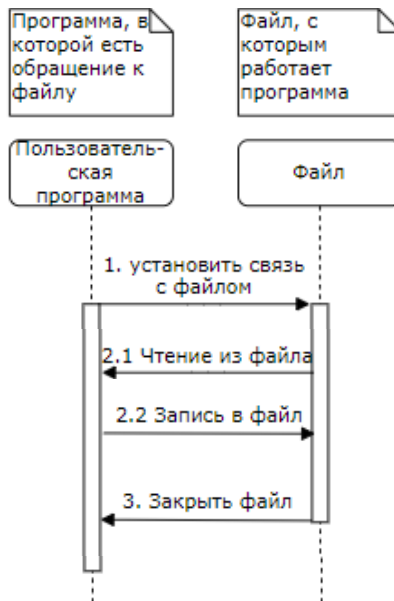


Рисунок 7

Прежде всего файл необходимо открыть, установив тем самым связь пользовательской программы с внешним носителем данных. После этого следует собственно обработка файла — запись в него данных, сформированных в программе, чтение данных из файла и ряд других операций, связанных, например, с поиском в файле нужных данных. И, наконец, завершая работу с файлом, его необходимо закрыть.

В Python есть встроенная функция `open()` для открытия файла.

```
fileObj = open(fileName, mode)
```

Данной функции требуется, как минимум один аргумент — имя файла. Здесь `fileName` — это имя файла (или путь к нему), который вы хотите открыть.

Примечание: *fileName должно включать имя и расширение файла.*

Если мы укажем только имя файла (а не путь к файлу), то это будет означать, что программа будет «искать» такой файл в своем рабочем каталоге.

Второй параметр `mode` позволяет задать режим открытия файла.

Рассмотрим существующие режимы открытия файла (возможные значения параметра `mode`) (таб. 1). По умолчанию значение параметра `mode= 'r'`.

В результате успешного открытия файла мы получим файловый объект (называемый дескриптором, «`handler`», некоторая абстракция, используемая операционной системой), который будет использоваться далее функциями записи в файл, чтения из файла и т. д.

Таблица 1. Режимы открытия файлов

Режим	Описание	Обработка данных начинается с...
r	Открытие текстового файла только для чтения. Если такого файла не существует, то будет сгенерировано исключение	Начала файла
w	Открытие текстового файла только для записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла
a	Открытие текстового файла для добавления. Если такой файл не существует, то он будет создан.	Конца файла
r+	Открытие текстового файла для чтения и записи. Если такого файла не существует, то будет выведена ошибка.	Начала файла
w+	Открытие текстового файла для чтения и записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла
a+	Открытие текстового файла для чтения и добавления. Если такой файл не существует, то он будет создан.	Конца файла
rb	Открытие двоичного файла для чтения. Если такого файла не существует, то будет выведена ошибка.	Начала файла
wb	Открытие двоичного файла для записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла
ab	Открытие двоичного файла для добавления. Если такой файл не существует, то он будет создан. Иначе данные из него будут удалены	Конца файла
wb+	Открытие двоичного файла для чтения и записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла

Режим	Описание	Обработка данных начинается с...
<code>ab+</code>	Открытие двоичного файла для чтения и добавления. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено	Конца файла

Рассмотрим на примере: откроем файл *test.txt*, находящийся в каталоге «[Data](#)», который находится в том же родительском каталоге «[Work](#)», что и каталог «*Python*» — рабочий каталог нашей программы (*example.py*).

```

...
Work
  Data
    test.txt
  Python
    example.py

```

Рисунок 8

В этом случае файл находится не в текущем каталоге нашей программы, поэтому в качестве первого аргумента (имени файла) передадим относительный путь к файлу.

```

fileHandler = open("../Data/test.txt")
if fileHandler:
    print("File is open")

```

В этом примере относительный путь «*../Data/test.txt*» требует от программы *example.py* следующих действий: от текущего положения (в каталоге *Python*) нужно подняться

в родительский каталог *Work*, далее зайти в каталог *Data* (который находится внутри каталога *Work*) и открыть файл *test.txt*.

Так как при вызове функции `open()` мы указали только один аргумент (имя файла), то файл будет открыт в режиме «только для чтения» (будет использовано значение по умолчанию параметра `mode`). В случае, если попытка открыть файл не удалась (например, файл не может быть открыт в указанном режиме) будет сгенерировано исключение `IOError`, иначе (успешное открытие файла) мы получим файловый объект.

В нашем примере мы проверили наличие такого объекта (успешность процедуры открытия файла) и вывели сообщение об этом.

При попытке открыть файл, которого нет по указанному местоположению (например, открытие файла *test.txt* в текущем каталоге, где его нет) получим исключение `FileNotFoundError`:

```
fileHandler = open("test.txt")  
# open file in current directory
```

No such file or directory: 'test.txt'

Рисунок 9

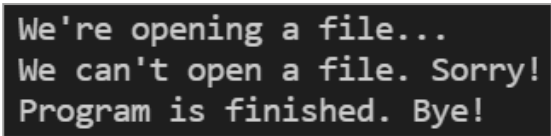
Для того, чтобы это исключение не приводило к внезапному (аварийному) завершению программы, нам нужно его обработать. Для этого есть специальная конструкция обработки исключений «`try..except`».

```
try:
    print("We're opening a file...")
    fileHandler = open("../Data/test.txt")

except:
    print("We can't open a file. Sorry!")

print("Program is finished. Bye!")
```

В этом примере блок `try` содержит фрагмент «опасного» кода, который может привести к генерации исключения (удалим файл *test.txt* из каталога *Data*, чтобы выполнить тестирование этой ситуации).



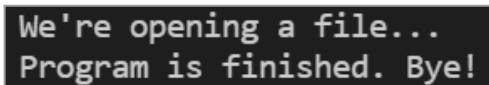
```
We're opening a file...
We can't open a file. Sorry!
Program is finished. Bye!
```

Рисунок 10

Блок `except` содержит команды, которые нужно выполнить в случае возникнувшего исключения.

С таким решением при возникновении исключительной ситуации (нужный файл не был найден) наша программа не станет аварийно завершаться. Для демонстрации этой особенности мы добавили последнюю строку кода функцией `print()`, которая выполнится в любом случае.

В случае успешного открытия файла будет выполнен только блок `try` и функция `print()`.



```
We're opening a file...
Program is finished. Bye!
```

Рисунок 11

Показанная в примере структура кода позволяет обработать исключительную ситуацию, если она возникнет, без аварийного завершения уже работающей программы.

Для того, чтобы прочитать содержимое файла, мы должны открыть файл в режиме чтения.

Выполнять чтение из файла в Python можно разными способами, которые реализованы в виде методов файлового объекта (получаемого в результате успешного открытия файла).

Начнем с метода `read()`:

```
f.read([size]),
```

где

- `f` — имя переменной файлового объекта;
- `size` — необязательный параметр, задающий количество символов, которые будут прочитаны из файла.

Если вам нужно полностью все символы из файла в виде строки, то следует использовать метод `read()` без аргументов.

Давайте прочитаем все содержимое файла *test.txt*, который мы открыли для чтения выше:

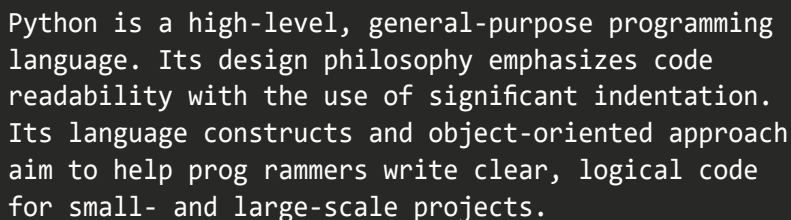


test.txt

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small- and large-scale projects.

Рисунок 12

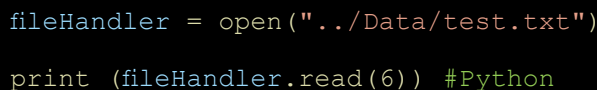
```
fileHandler = open("../Data/test.txt")
print (fileHandler.read())
```



Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small- and large-scale projects.

Рисунок 13

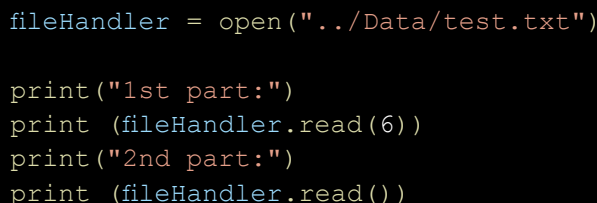
Если нам необходимо прочитать определенное число символов из файла (например, 6 символов, — это первое слово «*Python*» в нашем файле), то нужно задать это число в качестве аргумента метода `read()`.



```
fileHandler = open("../Data/test.txt")
print (fileHandler.read(6)) #Python
```

После чтения указанного количества символов следующей позицией в файле, с которой начнется чтение будет седьмой символ.

Таким образом, если в следующей строке кода мы опять вызовем метод, то чтение начнется уже не с начала файла, а с текущей позиции некоторого указателя, что смещается слева направо (от начала файла до конца) на указанное число позиций (символов).



```
fileHandler = open("../Data/test.txt")

print("1st part:")
print (fileHandler.read(6))
print("2nd part:")
print (fileHandler.read())
```



```
1st part:
Python
2nd part:
is a high-level, general-purpose programming language.
Its design philosophy emphasizes code readability
with the use of significant indentation. Its language
constructs and object-oriented approach aim to help
programmer s write clear, logical code for small-
and large-scale projects.
```

Рисунок 14

Теперь рассмотрим ситуацию, когда наш файл стоит из нескольких строк.

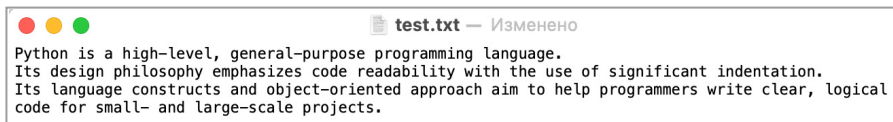


Рисунок 15

Как видно на рисунке 15 при открытии файла в текстовом редакторе символы перехода на новую строку (escape-последовательности «`\n`», с которыми мы уже знакомились при работе со строками) не отображаются, т. е. являются «невидимыми» для человека. Однако мы знаем, что такая комбинация символов присутствует в конце каждой строки.

Метод `read()` считывает символы-разделители строк в виде комбинации символов «`\n`» в соответствующих позициях, чтобы увидеть этот эффект воспользуемся raw-строками:

```
fileHandler = open("../Data/test.txt")
rawStr=repr(fileHandler.read())
print (rawStr)
```

```
'Python is a high-level, general-purpose programming
language, \nits design philosophy emphasizes code
readabil ity with the use of significant indentation,
\nits language constructs and object-oriented
approach
aim to help programmers write clear, logical code for
small- and large-scale projects.'
```

Рисунок 16

А что, если нам нужно считать только одну строку из файла? Для этой задачи более удобным является метод `readline()`. Если мы вызовем этот метод без аргументов, то считаем одну строку в текстовом файле, включая символы новой строки «`\n`» (для демонстрации этой особенности воспользуемся raw-строками, как в предыдущем примере).

```
fileHandler = open("../Data/test.txt")
str1=fileHandler.readline()
print(str1)
rawStr=repr(str1)
print(rawStr)
```

```
Python is a high-level, general-purpose programming
language.
'Python is a high-level, general-purpose programming
language. \n'
```

Рисунок 17

После чтения одной строки (вместе с символом «`\n`») указатель переходит на начало следующей строки файла.

Таким образом, повторное использование метода `readline()` в нашем примере выполнит чтение второй строки файла:

```
fileHandler = open("../Data/test.txt")

str1=fileHandler.readline()
print(str1)

str2=fileHandler.readline()
print(str2)
```

Python is a high-level, general-purpose programming language.

Its design philosophy emphasizes code readability with the use of significant indentation.

Рисунок 18

Для того, чтобы прочитать все строки файла можно воспользоваться методом `readlines()`:

```
fileHandler = open("../Data/test.txt")
lines=fileHandler.readlines()
print(lines)
```

```
['Python is a high-level, general-purpose programming
language. \n , Its design philosophy emphasizes code
rea dability with the use of significant indentation.
\n', 'Its language constructs and object-oriented
approach ai m to help programmers write clear,
logical code for small- and large-scale projects.']
```

Рисунок 19

Как можно заметить результат работы метода `readlines()` — это список строк. Такой формат данных можно обрабатывать, используя широкий набор полезных методов объекта список, с которыми мы уже знакомы.

Мы также можем прочитать весь файл (строка за строкой), перебирая строки в цикле `for`:

```
fileHandler = open("../Data/test.txt")
for line in fileHandler:
    print(line)
```

Python is a high-level, general-purpose programming language.

Its design philosophy emphasizes code readability with the use of significant indentation.

Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small- and large-scale projects.

Рисунок 20

Для того, чтобы выполнить запись в файл, нам нужно открыть его в режиме записи «`w`» или добавления «`a`». Нужно быть осторожными с использованием режима «`w`», так как в этом случае произойдет перезапись существующего файла (т.е. потеря всех предыдущих данных).

Запись строки или последовательности байт выполняется с помощью метода `write()`. Результат работы метода — количество символов (или байт), записанных в указанный файл.

```
fileHandler = open("../Data/newTest.txt", "w+")
n=fileHandler.write("How to Create a Text File in Python?")
print("We wrote {} symbols. Let's check it.".format(n))
print(len("How to Create a Text File in Python?"))
```

We wrote 36 symbols. Let's check it
36

Рисунок 21

В данном пример файла *newTest.txt* по указанному пути не существовало, поэтому он был создан.

В результате работы нашего кода содержание данного файла стало таким:



Рисунок 22

Теперь давайте повторно откроем этот же файл в режиме «W» и с помощью цикла запишем в него пять строк:

```
fileHandler = open("../Data/newTest.txt", "w")
for i in range(5):
    fileHandler.write("This is line %d\n" % (i+1))
```



Рисунок 23

Как мы видим предыдущее содержимое файла было полностью перезаписано новым контентом.

Для записи нескольких строк файл есть отдельный метод `writelines(listOfStr)`. Данный метод записывает содержимое списка (который мы передаем в качестве аргумента) в файл.

Давайте используем его для добавления (не перезаписи) нового контента (двух строк, хранящихся в списке) в наш файл *newTest.txt*. Для этого откроем файл в режиме добавления «a».

```
fileHandler = open("../Data/newTest.txt", "a")
myStrs=["Appended line 1\n", "Appended line 2\n"]
fileHandler.writelines(myStrs)
```

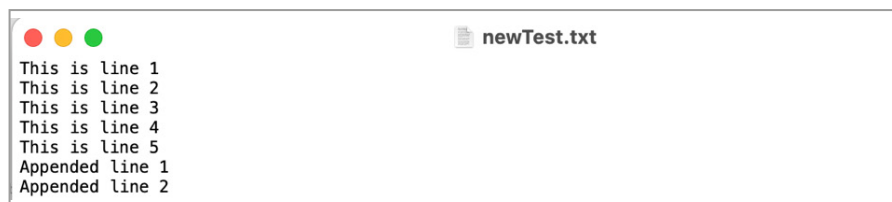


Рисунок 24

Примечание: если мы не вставим символ «`\n`» в строку (в том месте, где подразумевается начать новую), то данные будут последовательно записываться в текстовый файл, например «*Appended line 1Appended line 2*».

После выполнения всех необходимых нам операций с файлом, нам нужно правильно закрыть файл. Закрывание файла освободит ресурсы, которые были связаны с ним. Для решения этой задачи используется метод `close()`:

```
fileHandler = open("../Data/newTest.txt")
#File Handling
fileHandler.close()
```

Данный подход не совсем безопасен. Если возникает исключение, когда мы выполняем какую-либо операцию с файлом, код завершает работу, не закрывая файл. Например, файл был успешно открыт, но потом (пока наша программа все еще работает) файл был случайно удален. В такой ситуации попытка программы выполнить какую-то операцию с файлом (например, чтение очередной порции данных) вызовет генерацию исключения.

Используя «[try-finally](#)», мы гарантируем, что файл будет правильно закрыт, даже если возникнет исключение, которое приведет к остановке выполнения программы:

```
try:
    fileHandler = open("../Data/newTest.txt")
    # perform file operations
finally:
    fileHandler.close()
```

Есть еще один способ работы с файлом — использование оператора [with](#).

```
with open("../Data/newTest.txt") as f:
    # perform file operations
```

Такой подход гарантирует, что файл будет закрыт при выходе из блока внутри оператора [with](#). Нам не нужно явно вызывать метод [close\(\)](#). Это делается внутри блока неявным образом.

Менеджер контекста

В предыдущем разделе мы уже использовали оператор `with` для обеспечения безопасной работы с файлами даже в случаях возникновения исключений. Так что фактически первое знакомство с менеджером контекста уже состоялось.

Во многих задачах нам необходимо реализовывать взаимодействие с внешними ресурсами: файлами или базами данных. И в каждом языке программирования предусмотрены средства для реализации этого взаимодействия. Неконтролируемое использование ресурсов может негативно повлиять на работу приложения.

Например, мы открыли файл *example.txt*, потом изменили его содержимое, добавив новый фрагмент данных (строку «*Hello Again!*»), но не закрыли его после выполнения этих действий. Теперь допустим, что нам нужно прочитать данные из этого же файла. Причем в той же самой программе. Мы опять открываем его и выполняем операцию чтения. Однако, в этом случае мы не увидим внесенных изменений (новых данных), которые мы добавили ранее.

Первоначальное содержимое файла «*example.txt*».



Рисунок 25


```
fileHandler = open("../Data/example.txt", "a")  
fileHandler.write("\nHello Again!")  
  
fileHandler1 = open("../Data/example.txt")  
print(fileHandler1.read())
```

Результат операции чтения содержимого файла

Hello Python!

Рисунок 26

Как видно из полученных результатов работы команды `print()`, мы вывели только исходное содержимое файла. А добавленная строка «*Hello Again!*» не была выведена, т. к. не была прочитана программой. Это отсутствие доступа к обновленным (но пока еще не сохраненным) данным произошло по причине отсутствия операции закрытия файла перед его повторным открытием.

При работе с ресурсами (файлом в нашем случае) одной из основных задач является освобождение ресурсов после их использования. В противном случае произойдет либо утечка информации (как в приведенном примере), либо потеря быстродействия, либо вообще сбой работы приложения.

Менеджеры контекста позволяют выделять и освобождать ресурсы чётко по необходимости. Они автоматизируют этапы установки соединения с ресурсом и отключения от него (освобождение ресурса) всякий раз, когда мы в программе имеем дело с внешними ресурсами. Можно сказать, что менеджеры контекста облегчают правильную обработку ресурсов.

Давайте рассмотрим использование менеджеров контекста в задачах обработки файлов.

Ранее мы уже использовали конструкцию «`try-except-finally`» для гарантии закрытия файла даже в случае возникновения исключения. Однако использование менеджера контекста с ключевым словом `with` является более простым и компактным способом управления ресурсами.

Предположим, что у нас есть две связанные операции, которые мы хотим выполнить в паре (открытие файла и его закрытие) и блок кода, который должен быть выполнен между ними (операции по работе с файлом). Для реализации этого механизма используется такая конструкция на основе оператора `with`:

```
with expression as varName:  
    Operation1 with varName  
    Operation2 with varName  
    .....  
    OperationN with varName
```

Объект менеджера контекста является результатом выполнения выражения `expression` после ключевого слова `with`. И далее в блоке этот объект доступен по имени `varName`.

Например:

```
with open('../Data/example.txt', 'w') as fileHandler:  
    fileHandler.write('Hello!!!')
```

Здесь инструкция `open('some_file', 'w')` это выражение (`expression`), результатом которого является объект, доступный далее по имени `opened_file`.

Этот фрагмент кода открывает файл, записывает в него данные и закрывает файл после этого. При этом даже в случае возникновения исключения (например, ошибка записи в файл) менеджер контекста закроет файл.

Внутри блока `with` могут находиться любые (необходимые для реализации задачи) синтаксические конструкции. Например, выведем каждую строку содержимого файла:

```
with open('../Data/example.txt', 'r') as fileHandler:
    for line in fileHandler:
        print(line)
```

Приведенный выше код откроет файл и будет держать его открытым (пока происходит считывание и вывод строк содержимого) до выхода из блока оператора `with`, после чего файл будет закрыт.

Можно использовать сокращенную форму конструкции без `as`-части. Файловые объекты (дескрипторы, получаемые в результате успешного открытия файла) являются менеджерами контекста. Мы можем получить такой объект заранее (до блока `with`) и продолжить с ним работу таким способом:

```
fileHandler = open('../Data/example.txt')
with fileHandler:
    fileContents = fileHandler.read()
    print(fileContents)
```

После завершения операций по работе с файлом (которые находятся внутри блока `with`) менеджер контекста, как и в предыдущих примерах, закроет файл. В том числе и в случае возникновения исключения.

В Python поддерживается возможность создания нескольких менеджеров контекста в одном операторе `with`. В этом случае они перечисляются после оператора `with`, разделенные запятыми:

```
with A() as a, B() as b:  
    some actions
```

Это может быть полезно, когда нам нужно открыть, например, два файла одновременно, первый для чтения, а второй для записи:

```
path1="../../Data/example.txt"  
path2="../../Data/result.txt"  
  
with open(path1) as inFile, open(path2, "w") as  
outFile:  
    # read the content from example.txt  
    fileContents = inFile.read()  
    # transform the content  
    fileContents=fileContents.lower()  
    # write the transformed content to result.txt  
    outFile.write(fileContents)
```

Практические примеры

Теперь, когда мы уже знакомы со всеми основными операциями по работе с файлами в Python, давайте рассмотрим их практическое применение.

Пример 1. Поиск и замена слов в содержимом текстового файла

Предположим, что у нас есть текстовый файл *Python-About.txt*, содержащий такой текст.

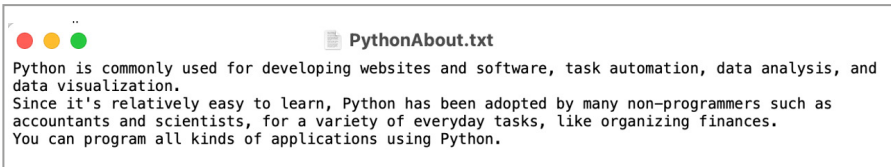


Рисунок 27

И нам необходимо заменить каждое вхождение слова «*Python*» на «*JavaScript*» в содержимом этого файла (три раза в нашем примере).

Для того, чтобы получить доступ к содержимому файла, нам нужно открыть его. Вначале мы откроем наш файл в режим чтения с помощью функции `open()`. Затем мы выполним чтение всего содержимое текстового файла (как одной строки), используя функцию `read()`. Таким образом мы сможем показать (вывести в консоль) содержимое файла до внесения изменений. После выполнения этих действий содержимое файла сохранено в переменной (тип которой строка). Далее применим к этой переменной метод

строки `replace()` для замены одной подстроки («*Python*») на другую («*JavaScript*»).

```
def replaceTextInFile(fileName, originText, newText):
    with open(fileName) as fileHandler:
        data = fileHandler.read()
        data = data.replace(originText, newText)

    with open(fileName, 'w') as fileHandler:
        fileHandler.write(data)

def readFromFile(fileName):
    with open(fileName) as fileHandler:
        data = fileHandler.read()
        print(data)

myFile='../Data/PythonAbout.txt'

print("Original file content:")
readFromFile(myFile)

replaceTextInFile(myFile, 'Python', 'JavaScript')

print("New file content:")
readFromFile(myFile)
```

Результат:

```
Original file content:
Python is commonly used for developing websites and
software, task automation, data analysis, and data
visualiz at ion.
Since it's relatively easy to learn. Python has been
adopted by many non-programmers such as accountants
```

Рисунок 28

and scientists, for a variety of everyday tasks, like organizing finances. You can program all kinds of applications using Python.

New file content:

JavaScript is commonly used for developing websites and software, task automation, data analysis, and data visualization.

Since it's relatively easy to learn, JavaScript has been adopted by many non-programmers such as accountants and scientists, for a variety of everyday tasks, like organizing finances.

You can program all kinds of applications using JavaScript.

Рисунок 28 (продолжение)

Пример 2. Подсчет количества слов в содержимом текстового файла, которые не являются числами

Достаточно часто в тексте содержится информация в виде чисел. Допустим, что нам нужно посчитать количество слов в текстовом файле, которые не являются числами.

Содержимое нашего файла «*Info.txt*»

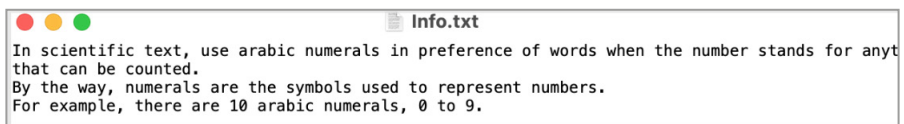


Рисунок 29

Создадим функцию, которая принимает в качестве аргумента имя файла и возвращает количество слов в

нем, не являющимися числами. Внутри функции создадим локальную переменную для хранения этого количества слов и установим ее начальное значение в 0.

Вначале мы откроем наш файл в режиме чтения с помощью функции `open()`. Затем мы выполним чтение всего содержимое текстового файла (как одной строки), используя функцию `read()`. Сохраним содержимое в отдельной переменной — строке, которую далее разобьем отдельные слова с помощью функции `split()`. В результате получим список слов. Далее переберем все слова из списка в цикле, проверяя каждое с помощью функции `isnumeric()`. Если слово не является числом (результат проверки функцией `isnumeric()` — `false`) увеличиваем наш счетчик слов на 1.

```
def readFromFile(fileName):  
    with open(fileName) as fileHandler:  
        data = fileHandler.read()  
        print(data)  
  
def wordCounter(fileName):  
    nWords = 0  
  
    with open(fileName) as fileHandler:  
        data = fileHandler.read()  
        lines = data.split()  
        for word in lines:  
            if not word.isnumeric():  
                nWords+=1  
    return nWords  
  
myFile='../Data/Info.txt'
```



```
print("File content:")
readFromFile(myFile)

print("Number of words:{}".format(wordCounter(myFile)))
```

Результат:

```
File content:
In scientific text, use arable numerals in preference
of words when the nuaber stands for anything that can
be counted.
By the way, numerals are the symbols used to represent
nuabers.
For exaaple, there are 18 arable nt*erals, e to 9

Number of words: 38
```

Рисунок 30

Пример 3. Вывести слова содержимого файла в обратном порядке

Допустим, что наш текстовый файл *simpleText.txt* содержит такой контент:



```
Hello Python!
Python is commonly used for developing websites and software, task automation, data analysis, and
data visualization
```

Рисунок 31

Создадим функцию, которая принимает в качестве аргумента имя файла и выводит слова его содержимого в обратном порядке. Вначале мы откроем наш файл в режим чтения с помощью функции `open()`. Затем мы выполним

чтение всего содержимое текстового файла (как одной строки), используя функцию `read()`. Сохраним содержимое в отдельной переменной — строке, которую далее разобьем отдельные слова с помощью функции `split()`. В результат получим список слов. С помощью в строенной функции `reversed()` выполним реверс списка слов.

Однако в тексте, кроме слов, могут быть знаки препинания, которые в результате работы функции `split()` окажутся последним символом слова (после которого они стояли в тексте). Поэтому прежде, чем выполнять разбиение строки на слова необходимо удалить знаки препинания.

Переберем в цикле все символы считанной из файла строки и, если символ не входит в набор знаков препинания (определим этот перечень в виде отдельной строки), то помещаем его в строку результата. Реализуем для этой задачи отдельную функцию.

```
def removePunctuation(myStr, marks):  
    resultStr=""  
    for symbol in myStr:  
        if symbol not in marks:  
            resultStr+=symbol  
    return resultStr
```

Общий код программы:

```
def removePunctuation(myStr, marks):  
    resultStr=""  
    for symbol in myStr:  
        if symbol not in marks:  
            resultStr+=symbol  
    return resultStr
```

```
def reverseFileWords(fileName):
    with open(fileName) as fileHandler:
        data = fileHandler.read()
        data=removePunctuation(data, punctuationSymbols)
        words=data.split()
        reversedWords=reversed(words)

        for word in reversedWords:
            print(word)

punctuationSymbols='"!()-;?@#%$%:"'\,./*_`'
myFile='../Data/simpleText.txt'

reverseFileWords(myFile)
```

Результат:

```
visualization
data
and
analysis
data
automation
task
software
and
websites
developing
for
used
commonly
is
Python
Python
Hello
```

Рисунок 32

Примечание: если нет необходимости в удалении знаков препинания из содержимого файла, то нужно просто удалить вызов функции `removePunctuation()` (строку кода «`data=removePunctuation(data, punctuation-Symbols)`») из тела функции `reverseFileWords()`.

Пример 4. Удаление заданной по номеру строки из файла

Рассмотрим на примере файла *Info.txt* такого содержания.

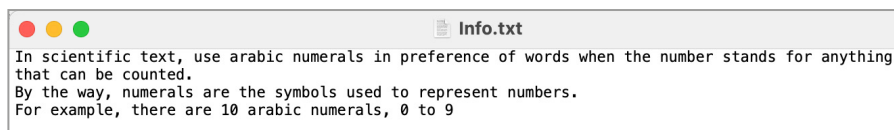


Рисунок 33

И нам необходимо, например, удалить вторую строку в этом контенте, сохранив полученный результат в новый файл.

Мы выполним чтение файла построчно с помощью метода `readlines()`. В результате получим список строк из содержимого файла.

Далее, обрабатывая это список поэлементно, будем проверять сравнивать позицию элемента (строки файла) с заданной. Если текущая строка имеет позицию, равную удаляемой, то она не записывается в текстовый файл с результатами работы программы.

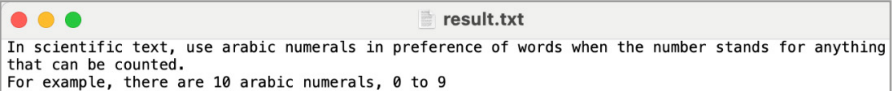
```
def removeLine(fileIn, fileOut, lineNumber):
    with open(fileIn) as fr:
        lines = fr.readlines()
```

```
counter=1 # position pointer
with open(fileOut, 'w') as fw:
    for line in lines:
        if counter != lineNumber:
            fw.write(line)
        counter += 1

myFile='../Data/Info.txt'
resultFile='../Data/result.txt'

removeLine(myFile, resultFile, 2)
```

Результат:



result.txt

In scientific text, use arabic numerals in preference of words when the number stands for anything that can be counted.
For example, there are 10 arabic numerals, 0 to 9

Рисунок 34



Урок 7

Файлы

© STEP IT Academy, www.itstep.org

© Анна Егошина

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии с законодательством о свободном использовании произведения без согласия его автора (или другого лица, имеющего авторское право на данное произведение). Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования. Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника. Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством.