Files

Файловый ввод и вывод

Файловый ввод и вывод

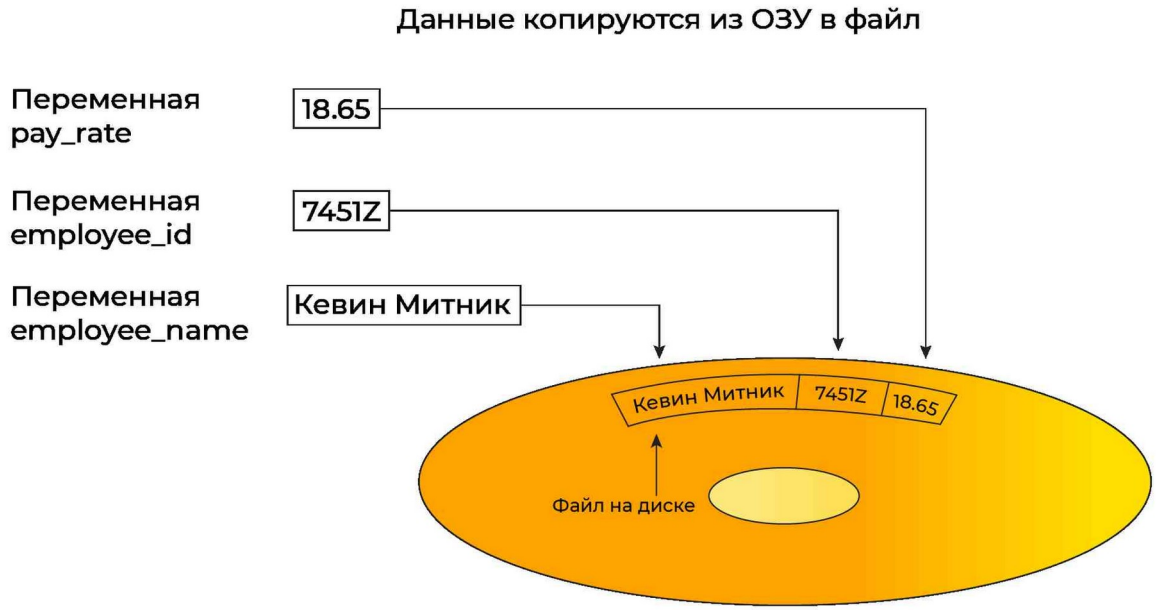
Программы, которые мы писали до сих пор, требовали повторного ввода данных при каждом запуске, потому что, как только программа заканчивает свою работу, данные для переменных исчезают из оперативной памяти. Если их нужно сберечь между выполнениями программы, требуется запись. Данные записываются в файл, обычно сохраняющийся на диске компьютера.

Файл (англ. *file*) — именованная область данных на носителе информации.

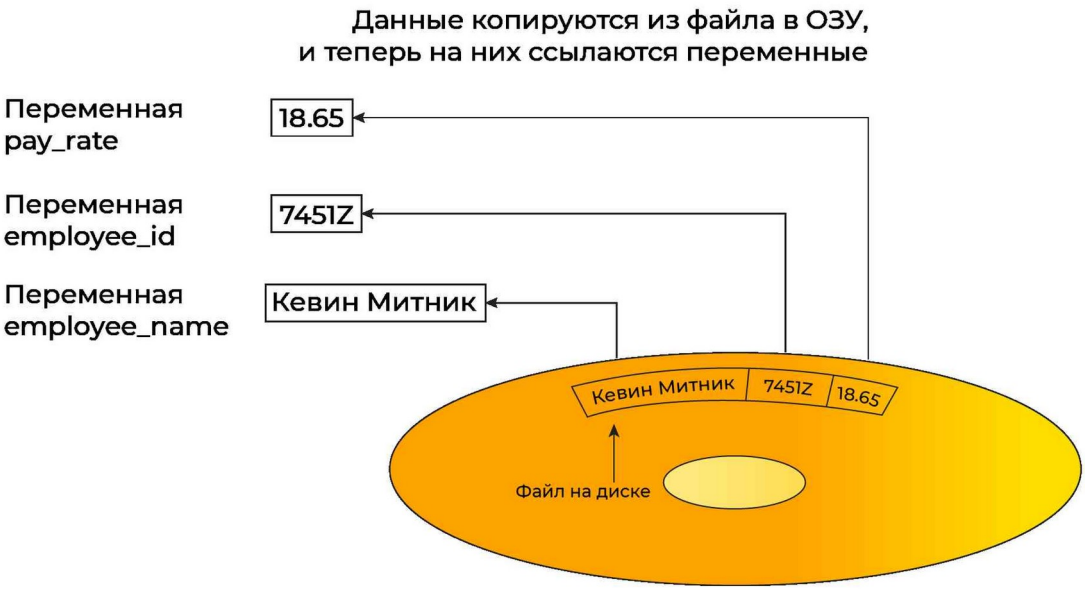
Сохраненные в файле данные обычно остаются в нем после завершения работы программы, их можно позже извлечь и использовать.

Когда программе нужно сохранить данные для дальнейшего использования, она пишет их в файл. Из файла записанные данные можно считать.

Программисты называют такой процесс сохранения данных — **запись данных** в файл. Когда часть данных пишется в файл, она копируется из переменной, находящейся в оперативной памяти. Файл, куда сохраняются данные, называется **файл вывода**, потому что программа помещает в него выходные данные.



Процесс извлечения данных из файла называется **чтением данных** из файла. Данные считываются из **файла ввода**. Программа извлекает входные данные из этого файла. Когда порция данных считывается из файла, она копируется в оперативную память, где на нее ссылается переменная.



Работа с файлами

Когда в программе используется файл, как правило требуется выполнить три шага:

1. **Открыть файл**. В процессе открытия файла создается связь между файлом и программой. Открытие файла вывода обычно создает файл на диске и позволяет программе записать в него данные. Открытие файла ввода позволяет программе прочитать данные из файла.
2. **Обработать файл.** На этом шаге данные либо записываются в файл (если это файл вывода), либо считываются из файла (если это файл ввода).
3. **Закрыть файл.** После использования файла программой его нужно закрыть, тем самым освободить ресурс и разорвать связь файла с программой.

Типы файлов

Существует два типа файлов: **текстовые** и **двоичные (бинарные)**. Текстовый файл содержит данные, которые были закодированы в виде текста при помощи такой схемы кодирования, как ASCII или Юникод. Даже если файл содержит числа, эти числа в файле хранятся как набор символов. В результате файл можно открыть и просмотреть в текстовом редакторе, таком как Блокнот.

Рекомендуем использовать текстовый редактор Notepad++.

Двоичный файл содержит данные, которые не были преобразованы в текст. Данные, которые помещены в двоичный файл, предназначены только для чтения программой, и такой файл невозможно просмотреть в текстовом редакторе.

Python позволяет работать и с текстовыми, и с двоичными файлами, но в рамках этого курса мы будем работать с текстовыми файлами, чтобы вы могли использовать текстовый редактор для исследования файлов, создаваемых вашими программами.

Разделение файлов на текстовые и бинарные искусственное, так как любой текстовый файл бинарен.Деление на текстовые и бинарные удобно концептуально и имеет практическое

значение при работе с файлами и их обработке. Но важно понимать, что это действительно весьма условное разделение.

Методы доступа к файлам

Большинство языков программирования обеспечивает два способа получения доступа к данным в файле:

- **последовательный,**
- **прямой** или **произвольный**.

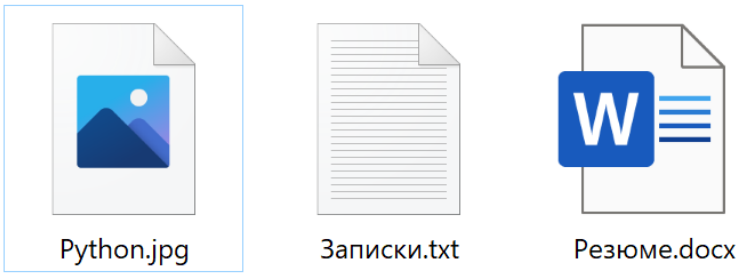
Последовательный, как при проигрывании кассет с записью на пленке, выдает порции информации одну за другой. При работе с таким файлом не получится перескочить сразу к нужной части данных, сначала придется прочитать все предыдущие.

При работе с файлом с прямым или произвольным доступом можно перескочить непосредственно к любой порции данных, не читая предыдущие. Как проигрыватель компакт-дисков или MP3-плеер перескакивает сразу к любой песне.

Имена файлов

Большинство пользователей компьютеров привыкли, что файлы определяются по имени. Когда создаете документ с помощью текстового редактора и сохраняете его в файле, указываете имя файла. Если исследуете содержимое диска с помощью **проводника Windows**, видите список имен файлов.

На рисунке показано, как в Windows могут выглядеть значки файлов с именами `Python.jpg`, `Записки.txt` и `Резюме.docx`.



У каждой операционной системы собственные правила именования файлов. Многие системы поддерживают использование расширений файлов, т.е. коротких последовательностей символов, которые расположены в конце имени файла и предваряются точкой. Файлы, изображенные на рисунке, имеют расширения `jpg`, `txt` и `docx`. Расширение обычно говорит о типе данных в файле. Например, расширение `jpg` сообщает, что файл содержит графическое изображение, сжатое согласно стандарту JPEG. Расширение `txt` свидетельствует, что в файле текст. Расширение `docx` информирует о наличии в файле документа Microsoft Word.

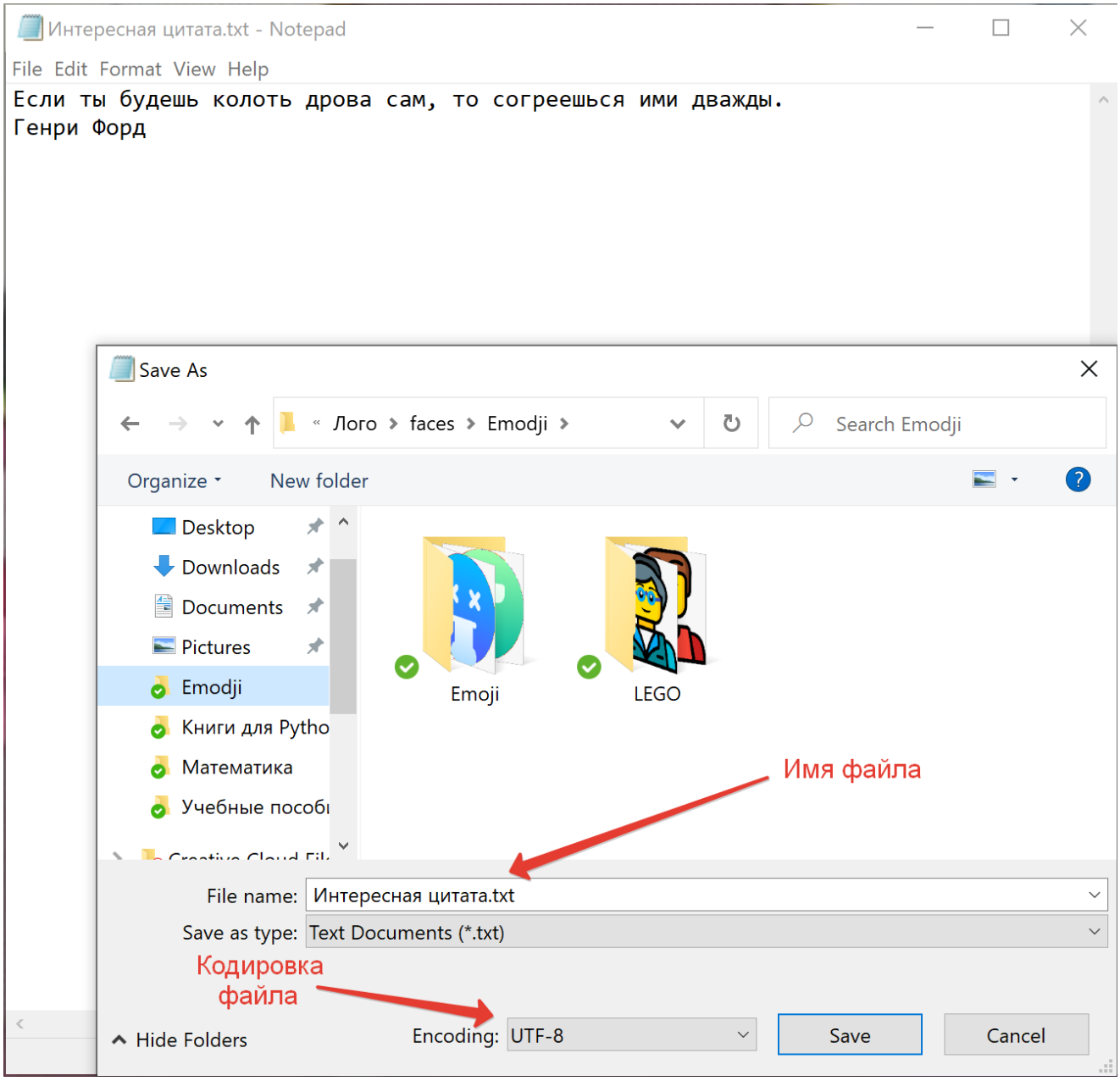
Кодировка файлов

В прошлом курсе мы говорили о том, как строки текста хранятся в памяти компьютера, таблицах символов ASCII и Юникод, а также о кодировке UTF-8.

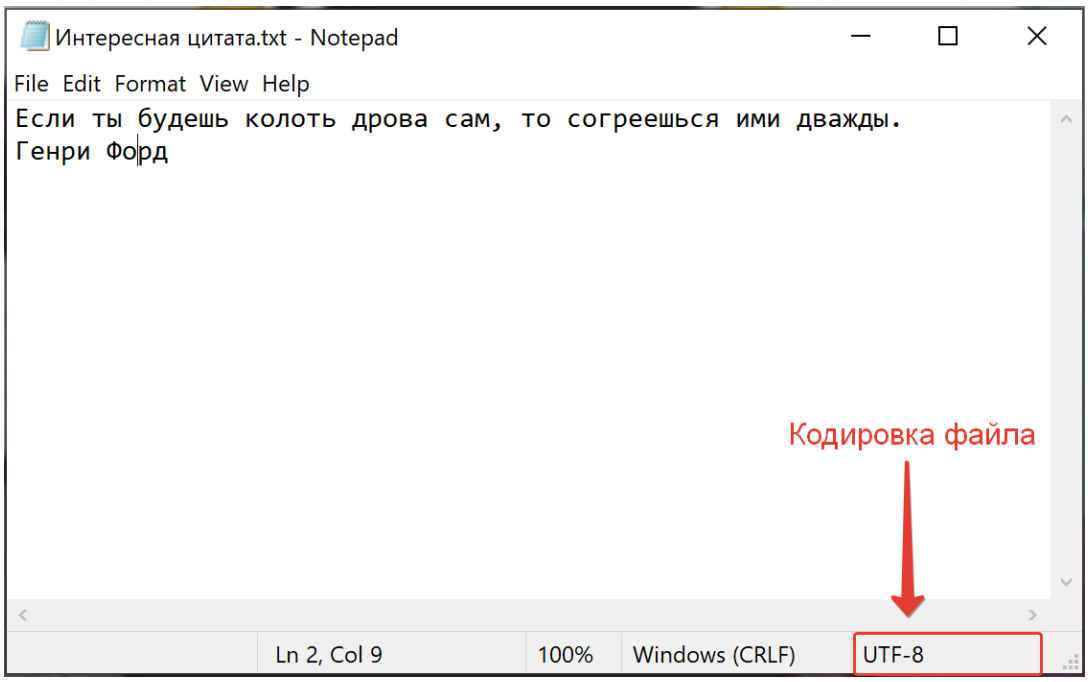
Кодировка UTF-8 самая распространенная, рекомендуем использовать именно ее в качестве кодировки по умолчанию для текстовых файлов.

UTF-8 — сложная кодировка, на обозначение одного символа в ней может использоваться от одного до шести байт. Подробнее про эту кодировку можно почитать в википедии.

В операционной системе Windows до сих пор используется однобайтовая кодировка Windows-1251 ( ). Чтобы избежать проблем при работе с текстовыми файлами в Windows нужно явно указывать кодировку. В редакторе Notepad («Блокнот») можно указывать кодировку при сохранении файла.



Если после такого сохранения открыть файл `Интересная цитата.txt`, то мы увидим нужную кодировку.



При работе с ОС Linux и MacOS таких проблем не возникает вовсе, поскольку в них кодировка UTF-8 применяется по умолчанию.

Относительные и абсолютные пути

Путь файла (или путь к файлу) — последовательное указание имен папок, через которые надо пройти, чтобы добраться до объекта.

Пути к файлу бывают двух типов:

- абсолютные;
- относительные.

Абсолютный путь – полный путь к файлу, показывающий **точное место** его расположения. Он всегда один и тот же, пока файл не перемещен.

Примеры абсолютного пути:

- `D:\Data\MyFiles\picture.png`;
- `C:\MyPrograms\Python\script.py`;
- `C:\Users\timur\YandexDisk\BeeGeek\Python\book.pdf`.

Указывая абсолютный путь на компьютере, обязательно нужно указывать диск, а также использовать `\` (для Windows) в качестве разделителя имен папок.

В unix-подобных ОС, например, в Linux и Mac OS для отделения имен папок используется прямой слеш `/`, а не обратный, как в Windows.

Относительный путь – привязан к какой-либо "отправной точке" и указан по отношению к ней.

Например, у нас есть картинка `picture.png`, которая хранится на диске `D`. Абсолютный путь к ней будет `D:\Data\MyFiles\picture.png`, а относительно папки `Data` можно указывать `MyFiles\picture.png`.

Абсолютный путь показывает точное местонахождение файла, а относительный показывает путь к файлу от какой-либо "отправной точки".

Примечания

Примечание 1. Конкретная физическая организация файлов, их группировка по папкам, устройство процедур доступа к информации, механизмы кеширования очень сильно зависят от операционной системы и применяемой в ней файловой системы. Как правило, при работе с файлами прикладные программисты работают на верхнем уровне **абстракции**.

Примечание 2. Файлы обычно располагаются на носителях, работающих медленнее, чем оперативная память. Поэтому работа с ними идет в буферизированном режиме. Даже если вы запросите один байт из файла, считается целый блок (до нескольких килобайт). Он переместится в буфер оперативной памяти. Дальше файл читается оттуда, поскольку это быстрее и экономит ресурсы чтения/записи внешних носителей.

Примечание 3. Примеры однобайтовых кодировок:

- Windows-1251;
- cp-866;
- КОИ-8.

Примечание 4. Прочитать про относительные и абсолютные пути можно тут и тут.

Работа с текстовыми файлами. Часть 1

Работа с файлами в Python

Высокоуровневый язык программирования Python предоставляет своим пользователям массу полезных средств для взаимодействия с файлами. Встроенные функции и методы позволяют создавать файлы, читать из них данные, а также всячески манипулировать их содержимым.

Открытие файла

Для открытия файлов в Python существует функция `open()`. Она создает файловый объект и связывает его с файлом на диске. Общий формат применения функции `open()`:

файловая_переменная = open(имя_файла, режим_доступа)

Здесь

- `файловая переменная` – имя переменной, которая ссылается на файловый объект;
- `имя_файла` – строковый литерал, задающий имя файла;
- `режим_доступа` – строковый литерал, задающий режим доступа (чтение, запись, и т.д.), в котором файл будет открыт.

Ниже представлены строковые литералы (символы), используемые для задания режима доступа.

Стр. литерал	Режим	Описание
'r'	Read (чтение)	Открыть файл только для чтения. Такой файл не может быть изменен.
'w'	Write (запись)	Открыть файл для записи. Если файл уже существует, стереть его содержимое. Если файл не существует, он будет создан.
'a'	Append (добавление)	Открыть файл для записи. Данные будут добавлены в конец файла. Если файл не существует, он будет создан.
'r+'	Read + Write	Открыть файл для чтения и записи. В этом режиме происходит частичная перезапись содержимого файла.
'x'	Create (создание)	Создать новый файл. Если файл существует, произойдет ошибка.

Предположим, файл `students.txt` содержит данные о студентах, и мы хотим открыть его для чтения.

Это можно сделать с помощью строки кода:

```
student_file = open('students.txt', 'r')
```

По умолчанию режим доступа (второй аргумент функции `open()`) определен для чтения (литерал `'r'`), поэтому файл `students.txt` можно открыть для чтения так:

```
student_file = open('students.txt')   # по умолчанию режим доступа для чтения ('r')
```

В результате исполнения этой инструкции будет открыт файл `students.txt` и переменная `student_file` будет ссылаться на файловый объект, который можно использовать для чтения данных из файла.

Обратите внимание, что в переменную `student_file` в примере выше не попадает содержимое файла `students.txt`. Фактически это ссылка на файл, ее еще называют **дескриптор файла**.

Предположим, надо создать файл с именем `sales.txt` и записать в него данные о продажах. Это можно сделать с помощью строки кода:

```
sales_file = open('sales.txt', 'w')
```

После исполнения этого кода будет создан файл `sales.txt` и переменная `sales_file` будет ссылаться на файловый объект, который можно использовать для записи в него данных.

Указание места расположения файла

Когда в функцию `open()` передается имя файла без указания пути, интерпретатор Python исходит из предположения, что место расположения файла то же, что у исполняемой программы. Например, программа расположена в папке `C:\Users\Documents\Python`. Если программа выполняется и исполняет инструкцию:

```
customer_file = open('customers.txt', 'r')
```

то файл `customers.txt` программа станет искать в папке `C:\Users\Documents\Python`.

Аналогично, если программа выполняется, и она исполняет инструкцию:

```
sales_file = open('sales.txt', 'w')
```

то файл `sales.txt` создается в той же папке.

Если имя файла не содержит путь, то используется относительный путь, относительно папки, где находится исполняемая программа.

Если требуется открыть файл, расположенный в другом месте, нужно указать путь и имя файла в аргументе, передаваемом в функцию `open()`.

Приведенный ниже код создает файл `test.txt` в папке `C:\Users\temp`:

```
test_file = open('C:\\Users\\temp\\test.txt', 'w')
```

Обратите внимание: символ `\` является специальным символом в Python и его нужно экранировать (`\\`), чтобы интерпретатор Python рассматривал обратную косую черту как обычный символ.

Вместо экранирования символов можно использовать сырые строки (raw strings). Для этого следует снабдить строковый литерал префиксом в виде буквы `r`.

```
test_file = open(r'C:\Users\temp\test.txt', 'w')
```

Приведенный выше код создает файл `test.txt` в папке `C:\Users\temp`. Префикс `r` указывает на то, что строковый литерал является сырым (неформатированным).

Механизм сырых строк очень удобен не только при работе с файлами.

Приведенный ниже код:

```
path = 'C:\new\text.txt'
print(path)
```

выводит:

```
C:
ew      ext.txt
```

поскольку символы `\n` и `\t` интерпретируются как перенос строки и табуляция.

Приведенный ниже код:

```
path = r'C:\new\text.txt'
print(path)
```

выводит содержимое строки `path`:

```
C:\new\text.txt
```

Чтобы сделать работу с файлами универсальнее, в путях файлов в Windows в Python-программах рекомендуется ставить прямой слеш (`/`). В наших примерах мы так и будем делать:

```
file1 = open(r'C:/Users/temp/test.txt')    # используем прямой слеш / (абсолютный путь)
file2 = open(r'temp/data.txt')             # используем прямой слеш / (относительный путь)
```

Кодировка

Открыть файл, содержащий только латиницу и цифры, можно так:

```
file = open('info.txt', 'r')
```

При работе с текстом на русском языке нужно указать кодировку, для этого служит параметр `encoding`:

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

Указание кодировки при открытии файла – хороший тон. Рекомендуем придерживаться этого правила.

Чтобы получить кодировку открытого файла, используют файловое свойство `encoding`.

Приведенный ниже код:

```
file1 = open('students.txt', 'w')
file2 = open('customers.txt', 'w', encoding='utf-8')

print(file1.encoding)
print(file2.encoding)
```

```
file1.close()
file2.close()
```

выводит на компьютере с операционной системой Windows:

```
cp1252
utf-8
```

Закрытие файлов

После окончания работы с файлом его необходимо закрыть. Для этого есть несколько причин:

- если файл изменялся, это позволит корректно его сохранить;
- если открытый файл потребуется другим программам, ваша программа может его блокировать;
- не стоит держать в памяти лишние, уже не нужные, данные;
- удалить открытый кем-то файл проблематично.

Для закрытия файла используется файловый метод `close()`:

```
file = open('info.txt', 'r')    # открываем файл с именем info.txt для чтения

                        # работаем с содержимым файла info.txt

file.close()                # закрываем файл после окончания работы
```

Чтобы проверить открыт файл или закрыт можно использовать файловое свойство (атрибут) `closed`.

Приведенный ниже код:

```
file1 = open('students.txt', 'w')
file2 = open('customers.txt', 'w')

file1.close()

print(file1.closed)
print(file2.closed)

file2.close()
```

выводит:

```
True
False
```

Обратите внимание на то, что при вызове метода мы используем скобки: `close()`, а при вызове свойства (атрибута) скобок нет `closed`. Методы совершают действия, а свойства возвращают информацию об объекте (подробнее об этом узнаете в курсе по ООП).

Чтение содержимого файла

Как уже сказано, при открытии файла с помощью функции `open()` в файловую переменную попадает не содержимое файла, а ссылка на файл (дескриптор файла).

Приведенный ниже код:

```
file = open('info.txt', 'w', encoding='utf-8')    # открываем файл для записи

print(file)
```

выводит:

```
<_io.TextIOWrapper name='info.txt' mode='w' encoding='utf-8'>
```

Для чтения содержимого открытого для чтения файла используются три файловых метода:

- **`read()` – читает все содержимое файла;**
- **`readline()` – читает одну строку из файла;**
- **`readlines()` – читает все содержимое файла и возвращает список строк.**

Предположим, в папке с исполняемой программой есть текстовый файл `languages.txt` с содержимым:

```
Python
Java
Javascript
C#
C
C++
PHP
R
Objective-C
```

Метод read()

Файловый метод `read()` считывает все содержимое из файла и возвращает строку, которая может содержать символы перехода на новую строку `'\n'`.

Приведенный ниже код:

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

content = file.read()

file.close()
```

считывает содержимое файла `languages.txt` в переменную `content`. В переменной `content` будет содержаться строка `'Python\nJava\nJavascript\nC#\nC\nC++\nPHP\nR\nObjective-C'`.

Таким образом, метод `read()` считывает все содержимое файла, **включая переносы строк**:

Если методу `read()` передать целочисленный параметр, то будет считано не более заданного количества символов. Например, считывать файл посимвольно можно при помощи метода `read(1)`.

Метод readline()

Файловый метод `readline()` считывает одну строку из файла (до символа конца строки `'\n'`), при этом возвращается считанная строка вместе с символом `'\n'`. Если считать строку не удалось – достигнут конец файла и больше строк в нем нет, возвращается пустая строка.

Приведенный ниже код:

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

language = file.readline()

file.close()
```

считывает содержимое первой строки файла `languages.txt` в переменную `language`. В переменной `language` будет содержаться строка `'Python\n'`.

Для удаления символа `'\n'` из конца считанной строки удобно использовать строковый метод `rstrip()`.

Приведенный ниже код:

```
line = 'Python\n'
line = line.rstrip()
```

удаляет символ `\n` из содержимого строки `line`, в результате чего в переменной `line` содержится строка `'Python'`.

Если вдруг вы забыли о строковых методах, освежить знания можно тут, тут и тут .

Прочитать содержимое всего файла построчно можно двумя способами.

С помощью цикла `while`:

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

line = file.readline()        # считываем первую строку

while line != '':             # пока не конец файла
    print(line.strip())       # обрабатываем считанную строку
    line = file.readline()    # читаем новую строку

file.close()
```

С помощью цикла `for` (предпочтительный вариант):

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

for line in file:
    print(line.strip())
```

```
file.close()
```

Метод `readline()` довольно удобен, когда мы хотим управлять процессом чтения из файла, особенно если файл очень большой и его полное считывание может привести к нехватке памяти.

Метод readlines()

Файловый метод `readlines()` считывает все строки из файла и возвращает список из всех считанных строк (одна строка — один элемент списка). При этом, каждая строка в списке заканчивается символом переноса строки `'\n'` .

Приведенный ниже код:

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

languages = file.readlines()

file.close()
```

считывает содержимое файла `languages.txt` в переменную `languages`. В переменной `languages` будет содержаться список:

```
['Python\n', 'Java\n', 'Javascript\n', 'C#\n', 'C\n', 'C++\n', 'PHP\n', 'R\n', 'Objective-C']
```

Чтобы удалить символ `'\n'` можно использовать списочное выражение:

```
languages = [line.strip() for line in file.readlines()]
```

либо функцию `map()` :

```
languages = list(map(str.strip, file.readlines()))
```

либо анонимную функцию:

```
languages = list(map(lambda line: line.strip(), file.readlines()))
```

Если передать в функцию `list()` ссылку на файловый объект `list(file)` , получим тот же результат, что при вызове метода `file.readlines()`.

Примечания

Примечание 1. Язык Python позволяет работать не только с текстовыми, но и с бинарными файлами. Ниже представлены строковые литералы, которые можно использовать для задания режима обработки файла.

Символ	Режим	Описание
't'	Текстовый режим (значение по умолчанию)	Работа с текстовым файлом
'b'	Бинарный режим	Работа с бинарными файлами (картинки, звук и т.д.)

Режим обработки файла указывается **после** режима доступа к файлу.

Приведенный ниже код открывает файл `file.dat` в режиме чтения как бинарный файл.

```
file = open('file.dat', 'rb')
```

По умолчанию функция `open()` использует литерал `'rt'`, то есть файл открывается для чтения в текстовом режиме.

Таким образом, открыть текстовый файл для чтения можно так `open('info.txt')`, или так `open('info.txt', 'r')`, или так `open('info.txt', 'rt')`.

Примечание 2. Так как Python — язык с автоматическим управлением памятью, все файлы закрываются автоматически после успешного завершения программы или когда удаляется последняя ссылка на файловый объект. Но важно все равно закрывать файл, как только он перестает быть нужным. Это помогает избегать конфликтов совместного доступа и риска получить испорченный файл, если программа завершится аварийно.

Файл должен быть закрыт сразу после того, как все нужное из него прочитано или в него записано.

Примечание 3. Еще раз обратите внимание на то, что в путях до файла используются прямые слеши `/`. Можно использовать и обратные, но тогда их придется экранировать либо применять модификатор сырой строки `r`. Кроме того, в unix-подобных операционных системах принято использовать именно прямой слеш.

Примечание 4. Существуют специальные символы:

- `\n` – перемещает позицию печати на одну строку вниз;
- `\r` – перемещает позицию печати в крайнее левое положение строки.

Приведенный ниже код:

```
print('aaaaaa\nbb')
```

выводит:

```
aaaaaa
bb
```

Приведенный ниже код:

```
print('aaaaaa\rbb')
```

выводит:

```
bbaaaa
```

Приведенный ниже код:

```
print(ord('\n'))
print(ord('\r'))
```

выводит:

```
10
```

Примечание 5. После того как файл (`file`) открыт, можно получить различную относящуюся к нему информацию. Три полезных атрибута (свойства):

Атрибут (свойство)	Описание
file.closed	возвращает истину (True), если файл закрыт, иначе возвращает ложь (False)
file.mode	возвращает режим доступа, с помощью которого был открыт файл
file.name	возвращает имя файла

Работа с текстовыми файлами. Часть 2

Позиция в файле

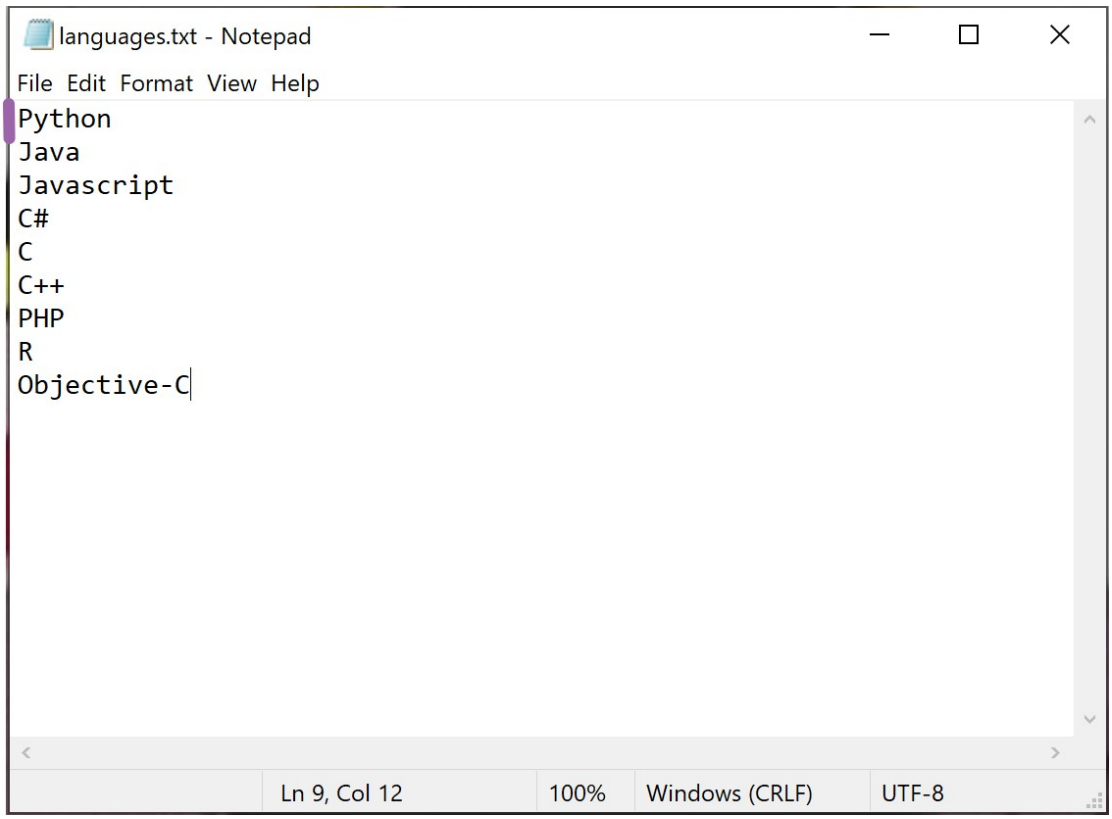
Когда мы читаем текст из файла с помощью методов `read()` или `readlines()` происходит перемещение **текущей позиции** в конец файла. При использовании метода `readline()` текущая позиция перемещается на следующую строку файла.

При открытии файла текущая позиция всегда равна нулю – указывает на первый символ текста. При прочтении файла до конца с помощью вызова методов `read()`, `readlines()` позиция перемещается в конец файла и последующие чтения ничего не дают.

Вызов методов `read()`, `readlines()`, `readline()` перемещает текущую позицию туда, где завершилось чтение. Для методов `read()` и `readlines()` это конец файла, для метода `readline()` – следующая строка после прочитанной.

Текущую позицию обычно называют "**курсор**".

Предположим, у нас есть файл `languages.txt`. Когда мы его открываем, курсор находится в начале файла, в нулевой позиции, это выглядит примерно так:

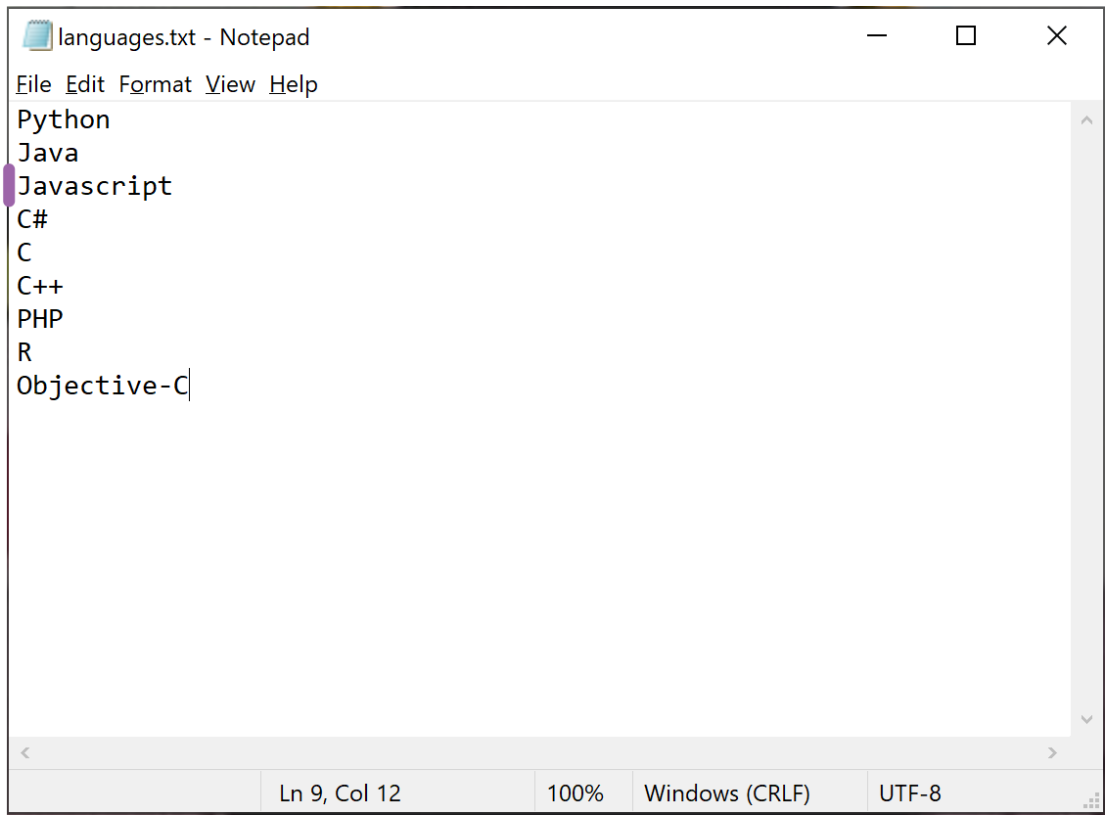


Если мы считаем две строки с помощью метода `readline()`:

```
file = open('languages.txt', 'r', encoding='utf-8')
line1 = file.readline()
line2 = file.readline()

file.close()
```

курсор переместится в начало третьей строки:

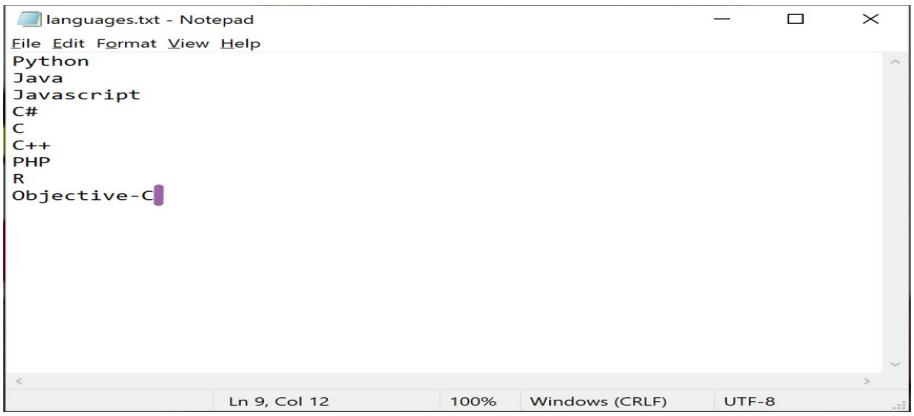


Чтение всегда происходит слева направо от курсора. Таким образом, если после двух вызовов метода `readline()` вызвать метод `read()`, он считает не весь файл, а только оставшиеся строки:

```
file = open('languages.txt', 'r', encoding='utf-8')
line1 = file.readline()
line2 = file.readline()
remaining_lines = file.read()    # считывание начинается с 3 строки до конца файла

file.close()
```

После того, как мы считали все строки файла, курсор находится в конце.



После завершения чтения мы больше не можем считать ни одного символа из файла. Все последующие вызовы методов `read()` или `readline()` будут приводить к считыванию пустой строки.

Для повторного чтения данных из файла, можно:

- переоткрыть файл, тогда курсор снова попадёт в начало;
- переместить курсор с помощью файлового метода `seek()`.

Файловый метод seek()

Файловый метод `seek()` задаёт позицию курсора в байтах от начала файла. Чтобы перевести курсор в самое начало файла необходимо вызвать метод `seek()`, передав ему в качестве аргумента значение 0.

Приведенный ниже код:

```
file = open('languages.txt', 'r', encoding='utf-8')
line1 = file.readline()
file.seek(0)              # переводим курсор в самое начало
line2 = file.readline()

print(line1, line2)

file.close()
```

выводит:

```
Python
Python
```

Метод `seek()` **не очень** полезен при работе с текстовыми файлами, так как не учитывает разделение текста на строки. А вот при работе с файлами в двоичном режиме умение работать с позицией и смещениями очень важно!

Будьте аккуратны с символами, использующими более 1 байта (кириллица в кодировке `utf-8`), обращение к "промежуточному" байту может вызвать ошибку.

Если метод `seek()` устанавливает курсор (текущую позицию), то метод `tell()` получает ее.

Приведенный ниже код:

```
file = open('languages.txt', 'r', encoding='utf-8')
print(file.tell())
line1 = file.readline()
print(file.tell())

file.close()
```

выводит:

```
0
8
```

В самом начале курсор (текущая позиция) равен нулю, после считывания первой строки, курсор смещается на 8 байт (по байту на каждый из символов `'P', 'y', 't', 'h', 'o', 'n'` и два байта на символ перевода строки `'\n'`).

Менеджер контекста

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

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

Менеджер контекста — объект, реализующий одноименный протокол. Объекты, реализующие этот протокол, позволяют использовать следующий специальный синтаксис:

```
with object as name:
    # Здесь нам доступен ресурс name.
    # Это тело with-блока.
# А здесь ресурс name уже освобождён, даже если в теле with-блока произошла ошибка.
```

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

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

Приведенный ниже код:

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

for line in file:
    print(line)

file.close()            # ручное закрытие файла

print('Файл закрыт')
```

можно переписать в виде:

```
with open('languages.txt', 'r', encoding='utf-8') as file:
    for line in file:
        print(line)
                    # автоматическое закрытие файла
print('Файл закрыт')
```

Обратите внимание: при использовании менеджера контекста не требуется использовать метод `close()`.

При работе с файлами желательно всегда использовать менеджер контекста. Это делает программу надежнее.

Примечания

Примечание 1. Подробнее о файловом методе `seek()` можно почитать в документации.

Примечание 2. В современных операционных системах файловый ввод-вывод устроен достаточно сложно. Для обеспечения максимального быстродействия чтения и записи в файлы, а также контроля безопасности этого процесса, большинство операционных систем не позволяют программам напрямую работать с диском. Операционная система предоставляет программам специальные объекты — файловые дескрипторы (функция `open()` возвращает как раз файловый дескриптор). Имея файловый дескриптор, можно записывать и читать данные, не задумываясь о файловой системе. Файловые дескрипторы удобны, но на создание каждого расходуется достаточно большое количество ресурсов. Поэтому у операционной системы есть общий лимит на количество одновременно использующихся файловых дескрипторов. И при этом каждая программа имеет свой собственный лимит. Как только программа исчерпает доступное ей количество дескрипторов, следующая попытка открыть очередной файл закончится с ошибкой. Программисту важно следить за тем, сколько файлов программа открывает в каждый момент и закрывает ли она их своевременно. Используйте менеджер контекста `with` и жизнь станет проще .

Примечание 3. С помощью менеджера контекста можно работать с несколькими файлами.

```
with open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    # обработка файлов
```

Работа с файлами. Часть 3.

Запись данных в файлы

Файлы могут быть открыты для чтения, а могут для записи данных. Ниже представлены строковые литералы для задания режима доступа.

Стр. литерал	Режим	Описание
'r'	Read (чтение)	Открыть файл только для чтения. Такой файл не может быть изменен.
'w'	Write (запись)	Открыть файл для записи. Если файл уже существует, то стереть его содержимое. Если файл не существует, он будет создан.
'a'	Append (добавление)	Открыть файл для записи. Все записываемые в файл данные будут добавлены в его конец. Если файл не существует, то он будет создан.
'r+'	Read + Write	Открыть файл для чтения и записи. В этом режиме происходит частичная перезапись содержимого файла с самого начала.
'x'	Create (создание)	Создать новый файл. Если файл уже существует, произойдет ошибка.

Для записи используются два файловых метода:

- write() – записывает переданную строку в файл;
- writelines() – записывает переданный список строк в файл.

Метод write()

Общий формат применения файлового метода write():

файловая_переменная.write(строковое_значение)

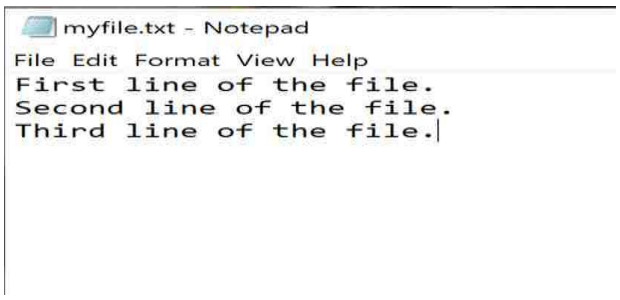
Здесь

- файловая переменная – это имя переменной, которая ссылается на файловый объект;
- строковое значение – это символьная последовательность, которая будет записана в файл.

Для записи данных в файл он должен быть открыт для записи (режимы 'w', 'a', 'r+'), иначе произойдет ошибка.

Рассмотрим текстовый файл myfile.txt, содержащий следующие строки:

First line of the file.
Second line of the file.
Third line of the file.



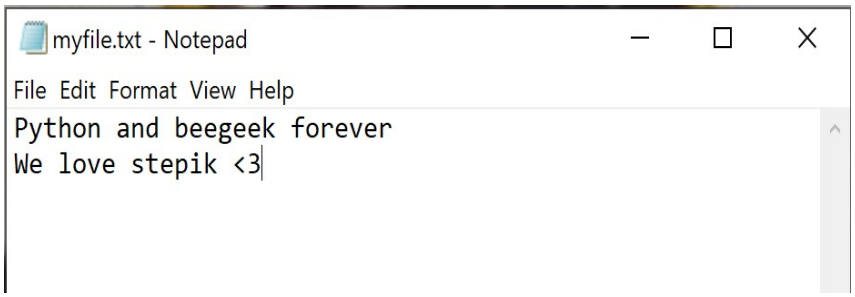
Если файл открыт в режиме 'w', то его содержимое сначала полностью стирается, а уже затем в него добавляются данные.

После выполнения следующего кода:

```
with open('myfile.txt', 'w', encoding='utf-8') as file:
    file.write('Python and beegeek forever\n')
    file.write('We love stepik <3')
```

файл myfile.txt будет содержать:

```
Python and beegeek forever
We love stepik <3
```



Если файл открыт в режиме 'a', то запись происходит в самый конец файла.

После выполнения следующего кода:

```
with open('myfile.txt', 'a', encoding='utf-8') as file:
    file.write('Python and beegeek forever\n')
    file.write('We love stepik <3')
```

файл myfile.txt будет содержать:

```
First line of the file.
Second line of the file.
Third line of the file.Python and beegeek forever
We love stepik <3
```

Если файл открыт в режиме 'r+', то происходит частичная перезапись его содержимого.

После выполнения следующего кода:

```
with open('myfile.txt', 'r+', encoding='utf-8') as file:
    file.write('Python and beegeek forever\n')
```

```
    file.write('We love stepik.')
```

файл myfile.txt будет содержать:

```
Python and beegeek forever
We love stepik. file.
Third line of the file.
```

Метод writelines()

Последовательные вызовы метода write() дописывают текст в конец файла.

Приведенный ниже код создает файл philosophers.txt и записывает в него три строки текста:

```
with open('philosophers.txt', 'w', encoding='utf-8') as file:
    file.write('Джон Локк\n')
    file.write('Дэвид Хьюм\n')
    file.write('Эдмунд Берк\n')
```

На практике часто приходится записывать в файл содержимое целого списка. Это можно сделать с помощью цикла или метода writelines(), что удобнее. Метод writelines() принимает в качестве аргумента список строк и записывает его в файл.

Приведенный ниже код создает файл philosophers.txt и записывает в него содержимое списка philosophers:

```
philosophers = ['Джон Локк\n', 'Дэвид Хьюм\n', 'Эдмунд Берк\n']

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

Обратите внимание, что все записанные в файл строковые значения оканчиваются символом \n', экранированной последовательностью новой строки. Символ \n' не только отделяет находящиеся в файле значения друг от друга, но и обеспечивает появление каждого из них на отдельной строке во время просмотра данных в текстовом редакторе.

Такой вариант записи предпочтителен, когда нужно записать большой объем текста, который вы получаете и обрабатываете строчка за строчкой. Можно предварительно накопить весь текст в одну большую строку, однако для этого может потребоваться большой объём памяти. Гораздо лучше записывать строчки по мере готовности и writelines для этого подходит идеально!

Запись в файл с помощью функции print()

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

Приведенный ниже код:

```
with open('philosophers.txt', 'w', encoding='utf-8') as output:
    print('Джон Локк', file=output)
    print('Дэвид Хьюм', file=output)
    print('Эдмунд Берк', file=output)
```

создает файл philosophers.txt с содержимым:

```
Джон Локк
Дэвид Хьюм
Эдмунд Берк
```

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

Приведенный ниже код:

```
with open('philosophers.txt', 'w', encoding='utf-8') as output:
    print('Джон Локк', 'Дэвид Хьюм', 'Эдмунд Берк', sep='***', file=output)
```

создает файл philosophers.txt с содержимым:

```
Джон Локк***Дэвид Хьюм***Эдмунд Берк
```

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

Примечания

Примечание 1. В некоторых операционных системах невыполнение операции закрытия файла может привести к потере данных. Данные сначала пишутся в буфер – небольшую область временного хранения в оперативной памяти. Когда буфер заполняется, система записывает его содержимое в файл. Это увеличивает производительность системы, потому что запись данных в оперативную память быстрее записи на диск. Процесс закрытия файла записывает любые несохраненные данные из буфера в файл. Чтобы принудительно записать содержимое буфера в файл, используется файловый метод flush().

Примечание 2. Используйте конструкцию with для чтения и записи файлов. Закрывать файлы — полезная привычка, и если вы используете команду with при работе с файлами, вам не придется беспокоиться об их закрытии. Команда with автоматически закрывает файл за вас.