**Работа с файлами, создание, хранение.**

План:

1. Введение;
2. Принципы работы с файлами;
3. Открытие файла;
4. Чтение данных из файла;
5. Запись и добавление данных в файл;
6. Закрытие файла;
7. Запись и чтение файлов CSV;
8. Работа с бинарными файлами.

**1.Введение**

**Файл** — это всего лишь набор данных, сохраненный в виде последовательности битов на компьютере. Информация хранится в куче данных (структура данных) и имеет название «имя файла» (filename).

**В Python существует два типа файлов:**

1. Текстовые
2. Бинарные

**Текстовые файлы**

Это файлы с понятным человеку содержимым. В них хранятся последовательности символов, понятные человеку. Блокнот и другие стандартные редакторы умеют читать и редактировать этот тип файлов.

Текст может храниться в двух форматах: (.txt) — простой текст и (.rtf) — «формат обогащенного текста».

**Бинарные файлы**

В бинарных файлах данные отображаются в закодированной форме (с использованием только нулей (0) и единиц (1) вместо простых символов). В большинстве случаев это просто последовательности битов.

Они хранятся в формате .bin.

Любую операцию с файлом можно разбить на три крупных этапа:

1. Открытие файла
2. Выполнение операции (запись, чтение)
3. Закрытие файла

**2. Принципы работы с файлами**

Принципы работы с файлами включают следующие основные аспекты:

1. **Открытие файла**. Прежде чем начать работу с файлом, его необходимо открыть. В Python для этого используется встроенная функция open(), которая принимает имя файла и режим открытия (чтение, запись, добавление и другие) в качестве аргументов. Файл открывается и создается файловый объект, с помощью которого можно выполнять различные операции над файлом.

2. **Чтение данных из файла**. Считывание данных из файла может происходить различными способами, такими как построчное чтение, чтение определенного количества символов или чтение полного текста. В Python есть разные методы для считывания данных из файла, такие как read(), readline() и readlines().

3. **Запись данных в файл**. Запись данных в файл также может происходить разными способами: добавление текста, замена существующего текста или запись множества строк сразу. В Python для этого используются методы write() и writelines().

4. **Закрытие файла**. После того, как все операции с файлом выполнены, файл следует закрыть, чтобы освободить системные ресурсы и убедиться, что все изменения были сохранены. В Python для закрытия файла используется метод close(). Также можно использовать контекстный менеджер (оператор with), который автоматически закрывает файл после окончания работы.

5. **Обработка исключений**. В процессе работы с файлами может возникнуть множество ошибок – например, при попытке открыть несуществующий файл или при записи в файл без соответствующих прав доступа. Для обработки ошибок ввода-вывода и предотвращения аварийного завершения программы рекомендуется использовать обработчики исключений, такие как try-except.

6. **Работа с различными форматами файлов**. Файлы могут иметь разные форматы, такие как текстовые, бинарные, CSV и JSON. В Python есть функции и модули для работы с различными типами файлов. Они позволяют выполнять сериализацию и десериализацию данных, сохранять и считывать структурированные данные и т.д.

7. **Манипуляция с метаданными файлов**. Основные метаданные файлов включают такие характеристики, как размер, дата создания и дата последней модификации. С помощью модуля os.path можно получить доступ к этим данным и использовать их в программах.

Эти принципы охватывают основные аспекты работы с файлами в повседневной работе. Они обеспечивают гибкость управления файлами и надежность обработки данных.

**Создание функций для работы с файлами.**

Создание функций для работы с файлами предполагает совокупность связанных с ними алгоритмов чтения, записи, проверки читаемости и т.п. Метаданные, полученные в результате работы с файлами, обычно являются возвратным результатом функций. В дальнейшем вы узнаете, как еще эффективнее использовать методы работы с файлами через создание классов ООП (объектно-ориентированного программирования). Классы, наделенные нашими методами, смогут обеспечить управляемый и структурированный способ работы с файлами. Часто создание таких методов способствует повторному использованию кода и расширяет его влияние на программу.

file\_path = "some.txt"

def open\_file(file\_path, mode="r"):  
 try:  
 file = open(file\_path, mode, encoding="utf-8") # encoding="cp1252"  
 except IOError as e:  
 print("Ошибка открытия файла:", e)  
 file = None  
 return file  
  
def read\_file(file):  
 if file is None:  
 print("Файл не открыт")  
 data = None  
 else:  
 data = file.read()  
 return data  
  
def write\_file(file, data):  
 if file is None:  
 print("Файл не открыт.")  
 else:  
 file.write(data)  
  
def close\_file(file):  
 if file is not None:  
 file.close()  
 file = None

В данном примере мы используем функции открытия, закрытия файла, чтения и записи. Мы создали эти функции, чтобы многократно использовать их в программе. Функции – это наши будущие методы класса. Они содержат обработку исключений, когда файл отсутствует или его невозможно открыть.

Теперь для работы с файлом можно использовать эти функции, что обеспечивает более структурированный и удобный способ работы с файлами.

**Функция «open\_file()»**

Функция «open\_file()» отвечает за открытие файла. Это позволяет совместить вызов функции открытия файла с дополнительной логикой и обработкой ошибок, а также обеспечивает удобное управление файловым объектом через параметры функции.

Давайте приведем пример считывания:

Создайте рядом с файлом программы текстовый файл «some.txt»

### file\_path = "some.txt" *def* open\_file(*file\_path*, *mode*="r", code=" utf-8"): *try*: file = open(*file\_path*, *mode*, encoding=code) # encoding="cp1252" для Windows *except* IOError *as* e: print("Ошибка открытия файла:", e) file = *None return* file

### my\_file = open\_file("some0.txt")

Вызов **«open\_file()»** обращает всё, что может прочитать из файла. Функция вместе с собой «тащит» аргументы (file\_path – путь к файлу, т.е. имя файла вместе со строкой расположения этого файла на диске, mode – режим открытия файла [по умолчанию «r»], а значит только для чтения и кодировку текстового файла согласно операционной системы code=" utf-8"). Здесь же обрабатывается возможная ошибка ввода-вывода (IOError) при открытии файла. Если файл будет не найден, то программа продолжит свою работу и сообщит пользователю о невозможности открытия.

Необходимо отметить, что само по себе открытие файла ничего не покажет на экране, поскольку чтение еще не произошло.

Таким образом, благодаря функции **«open\_file()»,** можно обеспечить более удобную и безопасную работу с файлами, а также добавить специфическую логику работы открытия файлов.

**Обработка исключений при открытии файлов внутри класса.**

Обработка исключений при открытии файлов внутри представленных функций предполагает добавление блока `try...except`, который использует внутреннюю функцию `open()` для открытия файла. Это помогает обрабатывать ошибки, возникающие при попытке открыть файл, и предотвращать возникновение исключений, которые могут вызвать непредвиденные проблемы в программе.

В приведенном выше примере метод `open\_file` использует блок `try...except` для обработки исключения `IOError`, которое может возникнуть, если указанный путь к файлу неправильный или недоступный для открытия по другим причинам (например, нет разрешений на доступ к файлу). Если исключение возникает, код внутри блока `except` будет выполнен, и пользователь увидит соответствующее сообщение об ошибке.

Таким образом, обработка исключений при открытии файлов внутри функции позволяет избегать прерывания работы программы и ошибок, связанных с некорректным вводом данных или доступом к файлам. Реализация обработки ошибок внутри функций также позволяет легче поддерживать и развивать код, сохраняя его устойчивость и надежность.

1. **Чтение данных из файла**

Создание функции для чтения содержимого файла включает использование внутренней функции `read()`, которая будет вызываться на файловом объекте. Чтобы реализовать эту функцию необходимо предварительно открыть файл. Другими словами, открытие файла и считывание файла – это парные функции, т.к. невозможно считывать файл без его предварительного открытия.

Пример расширения с ыункцией чтения содержимого файла:

#### file\_path = "some.txt"

#### def open\_file(file\_path, mode="r", code="utf-8"): try: file = open(file\_path, mode, encoding=code) # encoding="cp1252" except IOError as e: print("Ошибка открытия файла:", e) file = None return file def read\_file(file): if file is None: print("Файл не открыт") data = None else: data = file.read() return data

#### # Основной текст программы

#### my\_file = open\_file("some0.txt") data\_read = read\_file(my\_file) print(data\_read)

В этот раз мы добавили вызов функции `read\_file`, который проверяет, был ли файл открыт и существует ли файловый объект, ведь аргументом нашей функции является как раз тот файл, который мы открыли ранее. Если да, то выполняется чтение содержимого файла с помощью внутренней функции `read()`. В результате выполнения функции возвращаются данные `data` (всё, что можно прочитать внутри текстового файла). В противном случае, выводится сообщение об ошибке, и возвращается `None`.

Обычно после считывания закрывают файл, если обращение к файлу больше не актуально.

Таким образом, функции для чтения содержимого файла позволяет организовать удобный и безопасный способ чтения данных из файла.

**Реализация различных способов чтения.**

Для реализации различных способов чтения файла, можно добавить дополнительные параметры и условия в `read\_file`. Возможные варианты чтения включают чтение построчно, определенного количества символов и чтение полного текста файла.

Сначала создадим текстовый файл «some2.txt» из трёх строк:

Первая строка

Вторая строка

Третья строка

Расширим код функций:

### *def* open\_file(*file\_path*, *mode*="r", *code*="utf-8"): *try*: file = open(*file\_path*, *mode*, encoding=*code*) # encoding="cp1252" *except* IOError *as* e: print("Ошибка открытия файла:", e) file = *None return* file *def* read\_file(*file*, *mode\_read*="all", *size*=*None*): data = *None if file is None*: print("Файл не открыт") *else*: *if mode\_read* == "all": data = *file*.read() *elif mode\_read* == "line": data = *file*.readline() *elif mode\_read* == "lines": data = *file*.readlines() *elif mode\_read* == "size" *and size is not None*: data = *file*.read(*size*) *return* data *def* write\_file(*file*, *data*): *if file is None*: print("Файл не открыт.") *else*: *file*.write(*data*) *def* close\_file(*file*): *if file is not None*: *file*.close() file = *None* Теперь функция `read\_file` принимает параметр `mode\_куфв` с указанием режима чтения и дополнительный параметр `size` для указания количества символов при чтении по размеру. Реализованные режимы чтения:

- "all": чтение полного текста файла (по умолчанию).

- "line": чтение одной строки файла.

- "lines": чтение всех строк файла с сохранением в список.

- "size": чтение определенного количества символов, указанных в `size`.

Примеры использования различных режимов чтения файлов:

### # Открываем и читаем всё содержимое файла:

### my\_file = open\_file("some2.txt") data\_read = read\_file(my\_file, mode\_read="all") print(data\_read) close\_file(my\_file) # закрываем файл print()

### # Открываем и читаем содержимое одной строки файла:

### my\_file = open\_file("some2.txt") data\_read = read\_file(my\_file, mode\_read="line") print(data\_read) close\_file(my\_file) # закрываем файл

### # Открываем и читаем содержимое строк файла и помещаем их в список:

### my\_file = open\_file("some2.txt") data\_read = read\_file(my\_file, mode\_read="lines") print(data\_read) close\_file(my\_file) # закрываем файл

### # Открываем и читаем содержимое 10 первых символов:

### my\_file = open\_file("some2.txt") data\_read = read\_file(my\_file, mode\_read="size", size=10) print(data\_read) close\_file(my\_file) # закрываем файл

Таким образом, с помощью параметров и условий в методе `read\_file`, можно реализовать различные способы чтения содержимого файла, предоставляя гибкость в обработке данных в зависимости от требуемой задачи.

**Возврат данных из файла как объектов разных типов.**

Для возврата данных из файла в виде объектов разных типов, можно определить дополнительные параметры для функции чтения файла `read\_file` или определить отдельные специализированные методы для различных форматов данных.

Пример расширения функции с поддержкой чтения данных в виде строк, списков и словарей:

### *import* json *def* open\_file(*file\_path*, *mode*="r", *code*="utf-8"): *try*: file = open(*file\_path*, *mode*, encoding=*code*) # encoding="cp1252" *except* IOError *as* e: print("Ошибка открытия файла:", e) file = *None return* file *def* read\_file(*file*, *mode\_read*="all", *size*=*None*, *return\_type*="string"): data = *None if file is None*: print("Файл не открыт") *else*: *if mode\_read* == "all": data = *file*.read() *elif mode\_read* == "line": data = *file*.readline() *elif mode\_read* == "lines": data = *file*.readlines() *elif mode\_read* == "size" *and size is not None*: data = *file*.read(*size*) *if return\_type* == "list": data = data.splitlines() *if return\_type* = "dict": *try*: data = json.loads(data) *except* (json.JSONDecodeError, TypeError): print("Ошибка преобразования данных в словарь!") *return* data

# ... (остальные методы)

В данном примере `read\_file` получает дополнительный параметр `return\_type`, который указывает, в каком виде должны быть возвращены данные. Включены три возможных значения:

- "string": возвращает данные в виде строки (по умолчанию).

- "list": возвращает данные в виде списка, разбив содержимое файла на строки.

- "dict": возвращает данные в виде словаря, предполагая, что файл содержит JSON-сериализованный объект. Если данные не могут быть преобразованы в словарь, выводится сообщение об ошибке, и метод возвращает `None`.

Примеры использования различных типов данных:

# Чтение данных файла в виде строки

### my\_file = open\_file("some2.txt") data\_read = read\_file(my\_file, mode\_read="all", return\_type="string") print(data\_read) close\_file(my\_file)

# Чтение данных файла в виде списка строк

## my\_file = open\_file("some2.txt") data\_read = read\_file(my\_file, mode\_read="all", return\_type="list") print(data\_read) close\_file(my\_file)

# Чтение данных файла в виде словаря

# my\_file = open\_file("some2.txt") data\_read = read\_file(my\_file, mode\_read="all", return\_type="dict") print(data\_read) close\_file(my\_file)

При запуске последнего примера мы увидим ошибку преобразования данных в словарь! Так как для формирования словаря требуется особый формат данных. Мы сможем повторить этот пример позже, когда создадим такой файл в формате бинарного содержимого json.

Таким образом, с помощью определения дополнительных параметров для метода чтения файла, можно возвращать данные из файла в виде объектов разных типов в зависимости от потребностей. Вы можете дополнительно определить и другие типы данных (например, разбор CSV-файлов) или импортировать специализированные библиотеки и использовать их для преобразования данных.

**5. Запись и добавление данных в файл**

**Создание функции для записи данных в файл.**

Создание функции для записи данных в файл включает функциональность записи (append, overwrite и другие) на основе файлового объекта. Такая реализация позволяет упростить работу с файлами и контролировать процесс записи данных.

Пример записи в файл:

*import* json  
  
*def* open\_file(*file\_path*, *mode*="r", *code*="utf-8"):  
 *try*:  
 file = open(*file\_path*, *mode*, encoding=*code*) # encoding="cp1252"  
 *except* IOError *as* e:  
 print("Ошибка открытия файла:", e)  
 file = *None  
 return* file  
  
*def* read\_file(*file*, *mode\_read*="all", *size*=*None*, *return\_type*="string"):  
 data = *None  
  
 if file is None*:  
 print("Файл не открыт")  
 *else*:  
 *if mode\_read* == "all":  
 data = *file*.read()  
 *elif mode\_read* == "line":  
 data = *file*.readline()  
 *elif mode\_read* == "lines":  
 data = *file*.readlines()  
 *elif mode\_read* == "size" *and size is not None*:  
 data = *file*.read(*size*)  
  
 *if return\_type* == "string":  
 data = str(data)  
 *elif return\_type* == "list":  
 data = data.splitlines()  
 *elif return\_type* == "dict":  
 *try*:  
 data = json.loads(data)  
 *except* (json.JSONDecodeError, TypeError):  
 print("Ошибка преобразования данных в словарь!")  
  
 *return* data  
  
*def* write\_file(*file*, *data*):  
 *if file is None*:  
 print("Файл не открыт.")  
 *else*:  
 *file*.write(*data*)  
  
  
*def* close\_file(*file*):  
 *if file is not None*:  
 *file*.close()  
 file = *None*my\_file = open\_file("new\_some.txt","w")  
write\_file(my\_file, "В Python существует два типа файлов: текстовые и бинарные")  
close\_file(my\_file)

Функция открытия файла open\_file("new\_some.txt","w") содержит параметр названия файла и режима записи «w». Именно от неё зависит поведение внутренней функции «write()». По умолчанию файл открывается для чтения, но нас сейчас интересует сохранение данных, поэтому нужно для записи указать "w". А если указать другой показатель, например "a", тогда произойдет добавление данных в конец существующего файла. Так мы сможем пополнять свой файл новыми данными.

## my\_file = open\_file("new\_some.txt","a") write\_file(my\_file, "\nВ Python существует два типа файлов: текстовые и бинарные") close\_file(my\_file)

В этих примерах программа создаст новый файл «new\_some.txt», если такого в нашем проекте не было, при каждом запуске он будет перезаписываться заново. Внутрь файла прописывается текстовая строка: *«В Python существует два типа файлов: текстовые и бинарные»* (эти данные мы указали в качестве параметра для записи). Стоит лишь поменять второй аргумент функции на "a", как поведение «write()» изменится. Она будет добавлять строки при каждом обращении к функции с показателем "a", а значит нужно учитывать переход данных на следующую строку, если это необходимо. Обычно для этого используют управляющий символ перевода строки “\n” в начале новой строки.

Поэтому можно сделать вывод, что для расширения функционала необходимо модернизировать функцию открытия файла, а не записи для такого случая. Создание таких функций позволяет организовать удобный и безопасный способ работы с файлами, контролировать и изменять процесс записи данных в зависимости от потребностей.

**Реализация различных способов записи: добавление, перезапись, запись множества строк.**

В контексте работы с файлами и данными существует несколько способов записи: добавление, перезапись и запись множества строк. Мы рассмотрим каждый из них, чтобы лучше понять, как они работают:

1. **Добавление**

Добавление информации в файл подразумевает добавление новых данных в конец файла, не затрагивая существующие данные. Например, если у вас есть файл с текстом «Привет, мир!», и вы хотите добавить фразу « Как дела?», результатом будет файл с текстом «Привет, мир! Как дела?».

В Python для добавления данных в файл используется режим "a" при открытии файла. Например:

with open('file.txt', 'a') as f:

f.write(" Как дела?")

2. **Перезапись**.

Перезапись данных в файле означает удаление существующих данных и замена их новыми. Например, если у вас есть файл с текстом «Привет, мир!», и вы хотите перезаписать его с фразой «Как дела?», в файле будет только новый текст «Как дела?».

В Python для перезаписи данных в файле используется режим "w" при открытии файла:

with open('file.txt', 'w') as f:

f.write("Как дела?")

3. **Запись множества строк.**

Запись множества строк в файле – это режим записи, при котором вы можете записать несколько строк в файл за одну операцию. Это может быть полезно, например, при записи списков данных или таблиц.

В Python запись множества строк может быть выполнена с помощью функции writelines():

with open('file.txt', 'w') as f:

lines = ["Привет, мир!\n", "Как дела?\n"]

f.writelines(lines)

Здесь мы записываем список строк `lines` в файл file.txt. В результате файл будет содержать две строки текста:

Привет, мир!

Как дела?

**Запись списка или других форматов данных.**

Запись списка или других форматов данных в файл включает преобразование данных в строковый формат и последующую запись этих строк в файл. Я ниже объясню примеры записи списков и словарей в файлы, используя Python.

1. **Запись списка**.

Предположим, у нас есть список чисел, который мы хотим записать в файл:

numbers = [1, 2, 3, 4, 5]

Чтобы записать этот список в файл, вам нужно будет преобразовать каждое число в строковый формат и разделить их, например, с помощью символов новой строки, чтобы каждое число было на отдельной строке:

with open('numbers.txt', 'w') as f:

for number in numbers:

f.write(str(number) + "\n")

В результате файл `numbers.txt` будет содержать следующие строки:

1

2

3

4

5

2. **Запись словаря**.

Предположим, у нас есть следующий словарь, который мы хотим записать в файл:

data = {"name": "Alice", "age": 30, "city": "New York"}

Для записи словаря в файл, вы можете преобразовать каждую пару ключ-значение в строку и записать их на отдельных строках. Например:

with open('data.txt', 'w') as f:

for key, value in data.items():

f.write(f"{key}: {value}\n")

В результате файл `data.txt` будет содержать следующие строки:

name: Alice

age: 30

city: New York

Обратите внимание, что для записи сложных структур данных, таких как списки и словари, может быть полезно использовать специальные форматы, такие как JSON или CSV, которые предоставляют набор правил для представления структурированных данных в виде текста. В Python для работы с этими форматами существуют встроенные модули, такие как `json` и `csv`.

**6. Закрытие файла**

**Функция close\_file() для работы с файлами.**

Функция `close()`для работы с файлами предназначен для закрытия файла после завершения работы с ним. Когда файл открывается для чтения, записи или других операций, он занимает системные ресурсы (такие как дескриптор файла) и блокирует другие процессы или потоки операций от доступа к этому файлу.

Важно закрывать файлы после использования, чтобы освободить системные ресурсы и избегать потенциальных проблем с доступом к файлу или утечками памяти.

Чтобы использовать метод `close()`, сначала нужно открыть файл с помощью функции `open()`:

file = open("example.txt", "r")

После выполнения всех требуемых операций с файлом, вызывается метод `close()`:

file.close()

Однако, рекомендуется использовать управляемый контекст `with` при работе с файлами, потому что в этом случае файл автоматически закрывается, когда блок кода заканчивается. Таким образом, нет необходимости явно вызывать метод `close()`:

with open("example.txt", "r") as file:

# Здесь выполняются операции с файлом

# После завершения блока кода file автоматически закрывается

Использование `with` гарантирует, что файл будет корректно закрыт даже в случае возникновения исключений внутри этого блока кода.

**7. Чтение и запись файлов CSV**

Файл CSV – это особый вид файла, который позволяет структурировать большие объемы данных.

По сути, он является обычным текстовым файлом, однако каждый новый элемент отделен от предыдущего запятой или другим разделителем. Обычно каждая запись начинается с новой строки. Данные CSV можно легко экспортировать в электронные таблицы или базы данных. Программист может расширять CSV файл, добавляя новые строки.

### Чтение из CSV

#### Вначале создадим файл «some.csv» с данными через разделитель “,”

#### Имя,Профессия,Год рождения Виктор,Преподаватель,1995 Сергей,Сварщик,1983

Мария,Программист,1988

### Напишем код для открытия и чтения без проверки ошибок

### *import* csv *with* open('some.csv', encoding='utf-8') *as* file: data = csv.reader(file, delimiter=",") *for* row *in* data: print(row)

Вместо запятой можно использовать любой другой разделитель, поэтому при чтении CSV файла нужно заранее знать, какой символ используется.

Важно помнить, что CSV – это обычный текстовый файл, который не поддерживает символы в кодировках, отличающихся от ASCII или Unicode.

Расширим функциональные возможности считывания файла:

### *with* open("some.csv", encoding='utf-8') *as* file: # Создаем объект reader, указываем символ-разделитель "," data = csv.reader(file, delimiter = ",") # Счетчик для подсчета количества строк и вывода заголовков столбцов count = 0 # Считывание данных из CSV файла *for* row *in* data: *if* count == 0: # Вывод строки, содержащей заголовки для столбцов print(f'Файл содержит столбцы: {", ".join(row)}') *else*: # Вывод строк print(f' {row[0]} - {row[1]} и он родился в {row[2]} году.') count += 1 print(f'Всего в файле {count} строк.')

### Использование конструкции with…as позволяет программисту быть уверенным, что файл будет закрыт, даже если при выполнении кода произойдет какая-то ошибка.

### Запись в CSV

## Для записи информации в CSV файл необходимо создать объект writer:

## data\_writer = csv.writer(file, delimiter = ",", lineterminator="\r")

## Метод writerow(), позволит создать шапку файла. Он имеет следующий синтаксис: data\_writer.writerow(["Имя", "Класс", "Возраст"])

### *import* csv

### *with* open("classmates.csv", mode="w", encoding='utf-8') *as* file: data\_writer = csv.writer(file, delimiter = ",", lineterminator="\r") data\_writer.writerow(["Имя", "Класс", "Возраст"]) data\_writer.writerow(["Женя", "8", "14"]) data\_writer.writerow(["Саша", "9", "15"]) data\_writer.writerow(["Маша", "11", "18"])

## Обратите внимание, что при записи использовался, lineterminator="\r". Это разделитель между строками таблицы, по умолчанию он "\r\n".

**DictWriter**

Запись в файл также может быть осуществлена с помощью объекта DictWriter. Важно помнить, что он требует явного указания параметра fieldnames. В качестве аргумента метода writerow используется [словарь](https://all-python.ru/osnovy/slovari.html).

### Код программы выглядит так:

### *import* csv *with* open("classmates2.csv", mode="w", encoding='utf-8') *as* file: names = ["Имя", "Возраст"] data\_writer = csv.DictWriter(file, delimiter = ",", lineterminator="\r", fieldnames=names) data\_writer.writeheader() data\_writer.writerow({"Имя": "Глаша", "Возраст": "14"}) data\_writer.writerow({"Имя": "Тома", "Возраст": "15"}) data\_writer.writerow({"Имя": "Вова", "Возраст": "16"})

**Работа с бинарными файлами**

Бинарные файлы, в отличие от текстовых, хранят информацию в виде набора байт. Для работы с ними в Python необходим встроенный модуль «pickle». Этот модуль предоставляет два метода:

* **dump(obj, file)**: записывает объект obj в бинарный файл file
* **load(file)**: считывает данные из бинарного файла в объект

При открытии бинарного файла на чтение или запись также надо учитывать, что нам нужно применять режим "b" в дополнение к режиму записи ("w") или чтения ("r").

Допустим, необходимо сохранить два объекта имя и возраст:

### *import* pickle

### file\_path = "user.dat" name = "Viktor" age = 45 *with* open(file\_path, "wb") *as* file: pickle.dump(name, file) pickle.dump(age, file) *with* open(file\_path, "rb") *as* file: name = pickle.load(file) age = pickle.load(file) print("Имя:", name, "\tВозраст:", age)

### Подобным образом мы можем сохранять и извлекать из файла наборы объектов:

### file\_path = "users\_list.dat" users = [ ["Аня", 28, *True*], ["Глеб", 23, *False*], ["Семён", 34, *False*] ] *with* open(file\_path, "wb") *as* file: pickle.dump(users, file) *with* open(file\_path, "rb") *as* file: users\_from\_file = pickle.load(file) *for* user *in* users\_from\_file: print("Имя:", user[0], "\tВозраст:", user[1], "\tГосслужащий:", user[2])

### В зависимости от того, какой объект мы записывали функцией dump, тот же объект будет возвращен функцией load при считывании файла.

### Таким образом, используя методы считывания и записи данных, у нас всегда есть возможность структурированно хранить информацию не только в памяти компьютера, но и на различных носителях.