Основы Python

Урок 6. Работа

с файлами



## 

## На этом уроке

1. Будем читать и сохранять файлы. Вспомним про генераторы.
2. Познакомимся с менеджерами контекста.
3. Поговорим о сериализации

## 

## Оглавление

[Читаем текстовый файл целиком](#_7jog099xiw6d)

[Чтение файлов: тонкости](#_hg4eoglnjz08)

[Запись файлов](#_9h3t6225w4je)

[Изменение файлов: добавление и перезапись контента](#_8g3h3x9ramuc)

[\* Перемещение курсора при работе с файлом: .seek() и .tell()](#_y35u22kjcy5p)

[Сериализация данных](#_6obx2wt98gnf)

[Сериализация при помощи модуля json](#_y4tntcyhguky)

[Работа с бинарными файлами. Модуль pickle](#_xaloh6nobxqc)

[Практическое задание](#_4i7ojhp)

[Дополнительные материалы](#_2xcytpi)

[Используемая литература](#_1ci93xb)

# 

# 

# Читаем текстовый файл целиком

Создадим в папке с уроком текстовый файл:

hello.txt

| Привет всем, добравшимся до 6-го урока. Работаем с файлами. |
| --- |

**Обязательно** сохраняем его в кодировке UTF-8. Можно создать его прямо в PyCharm.

Самый простой способ прочитать содержимое файла в Python:

| file\_1 = open('hello.txt', 'r', encoding='utf-8') content = file\_1.read() print(content) file\_1.close() *# Привет всем, добравшимся до 6-го урока.* *# Работаем с файлами.* |
| --- |

Разберем шаги подробнее. Для доступа к файлам в Python используется функция [open()](https://docs.python.org/3.8/library/functions.html#open). Её первый аргумент — путь к файлу. Он может быть относительным, как в нашем случае, и абсолютным. Второй позиционный аргумент — режим доступа к файлу:

* ‘r’ — режим чтения (значение по умолчанию);
* ‘r+’ — режим редактирования, указатель (курсор) устанавливается на начало файла, данные пишутся поверх существующих. Если файла нет, генерируется исключение [FileNotFoundError](https://docs.python.org/3.8/library/exceptions.html#FileNotFoundError);
* ‘w’ — режим записи, существующий файл стирается (весь его контент исчезнет);
* ‘w+’ — режим записи и чтения, существующий файл стирается (весь его контент исчезнет), после записи данных можно их читать (в режиме ‘w’ можно только писать);
* ‘x’ — режим записи для случая, когда файла нет. Если файл существует, генерируется исключение [FileExistsError](https://docs.python.org/3.8/library/exceptions.html#FileExistsError) и никаких действий с ним не будет;
* ‘a’ — режим дозаписи, указатель (курсор) устанавливается в конец файла, если файла нет — создается новый;
* ‘a+’ — режим дозаписи и чтения, похож на режим ‘a’, но кроме записи можно читать данные;
* ‘t’ — текстовый режим (значение по умолчанию);
* ‘b’ — бинарный режим (работа с исполняемыми файлами, медиафайлами, с дампами).

Следующий аргумент — именованный: кодировка файла. Для бинарных файлов он не имеет смысла. Рекомендуем всегда явно задавать кодировку, чтобы не было проблем с русскими буквами. Если этого не делать, скрипт, который нормально работал в Windows, потом может некорректно работать в Linux. Было бы хорошо вызвать для переменной file\_1 функцию dir(), посмотреть методы и атрибуты [файлового объекта](https://docs.python.org/3.8/glossary.html#term-file-object), который она создаёт. Важно понять, что это не сам файл, а его «представитель», «дилер» в нашей программе. Мы ещё не читали файл в момент создания файлового объекта. Для чтения содержимого есть несколько разных способов. Один из них — вызов метода [.read()](https://docs.python.org/3.8/tutorial/inputoutput.html#tut-files). **Важно**, что при этом читается **всё** содержимое файла. А если размер файла превышает размер оперативной памяти? Этот случай мы рассмотрим позже. На практике он может встречаться достаточно часто: файлы логов бывают очень большими.

Итак, прочитали файл. Можно на этом и завершить скрипт. Но это было бы ошибкой. Причем **очень** серьезной. Дело в том, что файл — это ресурс, к которому может понадобиться доступ не только нашей программе, но и другим процессам, выполняющимся в операционной системе. Во время вызова функции open() мы этот ресурс «заняли» — теперь нужно «освободить»: вызываем метод .close() файлового объекта. Многие начинающие программисты недооценивают важность этой манипуляции, потому что на первых порах «ничего страшного» не происходит. Будьте внимательны.

Вы задумались: почему текст при выводе на экран выглядит точно так же, как в нашем файле? Дело в том, что метод .read() читает «сырой» текст вместе с [управляющими символами](https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%8F%D1%8E%D1%89%D0%B8%D0%B5_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D1%8B) “\n”, ”\r” и другими. Коварство этих символов в том, что они невидимы как символы — просто происходит перенос строки, например. В некоторых алгоритмах мы заменяем эти символы пробелами:

| ... clean\_content = content.replace('\n', ' ').replace('\r', ' ') print(clean\_content) *# Привет всем, добравшимся до 6-го урока. Работаем с файлами.* |
| --- |

В других алгоритмах превращаем текст в список, состоящий из абзацев, при помощи метода [.splitlines()](https://docs.python.org/3/library/stdtypes.html#str.splitlines):

| ... paragraphs = content.splitlines() print(paragraphs) *# ['Привет всем, добравшимся до 6-го урока.', 'Работаем с файлами.']* |
| --- |

Причём можно получить такой список сразу, вызвав метод .readlines() файлового объекта:

| file\_1 = open('hello.txt', 'r', encoding='utf-8') paragraphs = file\_1.readlines() print(paragraphs) file\_1.close() *# ['Привет всем, добравшимся до 6-го урока.\n', 'Работаем с файлами.']* |
| --- |

**Особенность:** управляющие символы при этом тоже будут прочитаны — Python читает «сырые» строки. Можно потом через map() или другим способом удалить эти символы. На наш взгляд, предыдущий пример выглядит более логичным для получения списка абзацев из текста.

# Чтение файлов: тонкости

А что, если в алгоритме можно ограничиться чтением файла по строкам? При этом можем получить серьёзный выигрыш по памяти. НО, ВОЗМОЖНО, И ПРОИГРЫШ ПО ПРОИЗВОДИТЕЛЬНОСТИ скрипта в целом: каждая операция доступа к файлу занимает **очень много** времени по сравнению с чтением информации из оперативной памяти. С появлением SSD-накопителей скорость работы с диском повысилась, но она все равно намного ниже скорости работы с памятью. Поэтому при реализации чтения файла по строкам или другим порциям нужно четко понимать плюсы и минусы такого решения.

При работе с текстовыми файлами можем использовать метод .readline() файлового объекта для чтения одной строки:

| file\_1 = open('hello.txt', 'r', encoding='utf-8') print(file\_1.readline()) print(file\_1.readline()) file\_1.close() *# Привет всем, добравшимся до 6-го урока.* *#* *# Работаем с файлами.* |
| --- |

Вы заметили «лишнюю» строку? Опять имеем дело с чтением «сырой» строки с управляющими символами. А что, если ещё раз вызвать метод .readline()? Получим либо содержимое очередной строки, либо пустую строку, если строк в файле больше нет. Таким образом можно организовать цикл построчного чтения файла:

| file\_1 = open(**'hello.txt'**, **'r'**, encoding=**'utf-8'**)  line = file\_1.readline()  while line:  print(line)  line = file\_1.readline()  file\_1.close() |
| --- |

Мы уже знаем, что в Python подобные манипуляции лучше делать через цикл-итератор, поэтому предпочтительнее следующий код:

| file\_1 = open('hello.txt', 'r', encoding='utf-8') for line in file\_1:  print(line) file\_1.close() *# Привет всем, добравшимся до 6-го урока.* *#* *# Работаем с файлами.* |
| --- |

В этом примере файловый объект «превращается» в генератор (точнее, итератор) и в каждом шаге цикла возвращает очередную «сырую» строку текста (с управляющими символами).

Что можно улучшить в этом коде? Обернуть его в [менеджер контекста](https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers):

| with open('hello.txt', 'r', encoding='utf-8') as file\_1:  for line in file\_1:  print(line) |
| --- |

**Обратите внимание**, что здесь мы **не вызываем** метод .close(). Это не ошибка. Именно в этом и есть смысл менеджера контекста — ответственность за освобождение ресурсов теперь на нём.

В будущем вы сможете писать свои менеджеры контекста. Основная идея следующая: можно задать действия при [входе](https://docs.python.org/3/reference/datamodel.html#object.__enter__) в данный контекст и действия при [выходе](https://docs.python.org/3/reference/datamodel.html#object.__exit__) из контекста.

**Важно**: использование менеджеров контекста ПРИВЕТСТВУЕТСЯ. Также зачастую режим ‘r’ не указывают, он и так будет по умолчанию.

# Запись файлов

От чтения перейдем к записи. Для текстовых файлов есть два метода [файлового объекта](https://docs.python.org/3.8/tutorial/inputoutput.html#tut-files):

* .write() — сохраняет весь текст как одно целое;
* .writelines() — сохраняет список строк.

Попробуем первый:

| txt = '''Пробуем записать в файл текст. Используем метод .write().'''  with open('write\_method.txt', 'w', encoding='utf-8') as f:  f.write(txt) |
| --- |

В результате выполнения кода в папке урока должен появиться файл write\_method.txt. Обратите внимание, что мы переименовали переменную файлового объекта в “f”. Вы часто будете видеть именно такое имя — это традиция. Вторая особенность примера: использовали тройные кавычки: так в Python можно записывать многострочные тексты в оригинальном виде, без символа \.

Теперь пример второго метода для записи данных в текстовый файл:

| txt\_lines = ['Пробуем записать в файл текст.',  'Используем метод .writelines().']  with open('writelines\_method.txt', 'w', encoding='utf-8') as f:  f.writelines(txt\_lines) |
| --- |

Разумеется, в результате выполнения этого кода получим файл writelines\_method.txt. Но если мы его посмотрим, обнаружим, что все сохранилось в одну строку. Вы догадались, почему? На самом деле всё логично: если в Python мы читаем строки в «сыром» виде, то и писать их тоже нужно «сырыми» — без привнесения изменений в виде управляющих символов. Если поправим «исходники», всё станет хорошо:

| txt\_lines = ['Пробуем записать в файл текст.\n',  'Используем метод .writelines().']  with open('writelines\_method\_upd.txt', 'w', encoding='utf-8') as f:  f.writelines(txt\_lines) |
| --- |

Какую ошибку часто допускают начинающие разработчики? Используют метод .writelines() для записи обычного текста. **Помните**: этому методу нужно передавать **список** в качестве аргумента!

**ВАЖНО:** во всех этих примерах всегда создается **НОВЫЙ** файл, если файл существовал — он уничтожается.

Если мы не хотим, чтобы уже существующие файлы перезаписывались, просто меняем режим доступа к файлу: вместо ‘w’ будет ‘x’.

# Изменение файлов: добавление и перезапись контента

В некоторых ситуациях нужно дописать данные в файл. Мы уже знаем про режим доступа ‘a’, но важно понимать ещё кое-что: открыть файл — это полдела. Вы же, когда редактируете текст, постоянно перемещаете **курсор** — Python тоже должен установить курсор (иногда его называют «указатель») в нужную позицию перед добавлением данных. Если вы используете режим ‘a’, указатель будет установлен в конец открытого файла, а если режим ‘r+’ — в начало. Проверим:

| txt = '''Пробуем дозаписать в файл текст. Режим доступа "a"'''  with open('append\_text.txt', 'a', encoding='utf-8') as f:  f.write(txt) |
| --- |

Вот что должно получиться после двукратного запуска скрипта:

append\_text.txt

| Пробуем дозаписать в файл текст. Режим доступа "a"Пробуем дозаписать в файл текст. Режим доступа "a" |
| --- |

Почему такой результат? Мы не перевели вторую строку в исходном тексте — вот и получили «сырую» склейку данных. Как поправить? Можно так:

| txt = '''Пробуем дозаписать в файл текст. Режим доступа "a"  ''' |
| --- |

Теперь второй пример. Нужно создать пустой файл replace\_text\_1.txt. В режиме ‘r+’ файл автоматически не создаётся! Выполним скрипт:

| txt = '''Пробуем дозаписать в файл текст. Режим доступа "r+" '''  with open('replace\_text\_1.txt', 'r+', encoding='utf-8') as f:  f.write(txt) |
| --- |

Что-то меняется после нескольких запусков скрипта? Нет. Почему? Мы же каждый раз начинаем писать в файл с самого начала. При этом все символы перезаписываются новым текстом. Но новый текст совпадает со старым, поэтому ничего не меняется.

Ещё один эксперимент. Создаём пустой файл replace\_text\_2.txt и выполняем один раз скрипт:

| txt = '''Пробуем дозаписать в файл текст. Режим доступа "r+" '''  with open('replace\_text\_2.txt', 'r+', encoding='utf-8') as f:  f.write(txt)  txt = '''Пробуем ДОЗАПИСАТЬ в файл текст! Режим ДОСТУПА '''  with open('replace\_text\_2.txt', 'r+', encoding='utf-8') as f:  f.write(txt) |
| --- |

В результате получаем текстовый файл:

replace\_text\_2.txt

| Пробуем ДОЗАПИСАТЬ в файл текст! Режим ДОСТУПА r+" |
| --- |

Это результат записи второго текста поверх первого. Обратите внимание, что остаток первого текста сохранился — только появился ещё один перенос строки.

На самом деле подобные манипуляции в Python делаются редко, поэтому не нужно тратить много времени на этот раздел. Лучше более глубоко погрузиться в нюансы чтения и записи файлов.

# \* Перемещение курсора при работе с файлом: .seek() и .tell()

Можно ли в Python начать чтение с конкретной позиции курсора в файле? Давайте начнём с интроспекции. Создадим файл hello\_2.txt:

| Учимся читать файлы. Можем установить указатель в нужную позицию. Начало - 0. Можно отсчитывать от конца файла. |
| --- |

И выполним скрипт:

| with open('hello\_2.txt', 'r', encoding='utf-8') as f:  print(f.tell())  line = f.readline()  while line:  print(line.strip(), f.tell(), sep='\n')  line = f.readline() *# 0* *# Учимся читать файлы.* *# 39* *# Можем установить указатель в нужную позицию.* *# 123* *# Начало - 0.* *# 142* *# Можно отсчитывать от конца файла.* *# 203* |
| --- |

Как вы уже догадались, [метод файлового объекта](https://docs.python.org/3.8/tutorial/inputoutput.html#methods-of-file-objects) .tell() возвращает текущую позицию указателя в файле. После чтения очередной строки, разумеется, позиция указателя увеличивается на длину этой строки. Часто возникает вопрос: в каких единицах эти числа? В [документации](https://docs.python.org/3.8/tutorial/inputoutput.html#methods-of-file-objects) для текстового режима сказано opaque number — некоторое условные числа, «попугаи». Почему так? Раз мы имеем дело с разными кодировками, длина кода символа может быть разной, и сделать одну единицу измерения не получится. Попробуйте написать текст на английском языке — получите числа, совпадающие с длиной строки плюс управляющие символы, ведь один символ для английского языка в UTF-8 — один байт. В нашем примере есть русские буквы — они по два байта, поэтому числа получились «труднообъяснимыми». Для файлов, открытых в **бинарном** режиме, метод .tell() даёт точное значение в байтах.

*Примечание:* использовали метод .strip() для очистки строки от управляющих символов в начале и конце.

Теперь попробуем поработать с методом файлового объекта .seek() — он перемещает указатель подобно тому, как мы это делаем курсорными клавишами при редактировании текста. Первым аргументом ему передаем смещение, а вторым — число, задающее начало отсчёта:

* 0 — от начала файла (по умолчанию);
* 1 — от текущей позиции;
* 2 — от конца файла.

Пример:

| with open('hello\_2.txt', 'r', encoding='utf-8') as f:  f.seek(39)  print(f.readline().strip())  f.seek(142)  print(f.readline().strip()) *# Можем установить указатель в нужную позицию.* *# Можно отсчитывать от конца файла.* |
| --- |

Взяли числа из предыдущего примера и прочитали вторую и четвёртую строки файла. В каких ситуациях может понадобиться метод .seek()? Например, мы итеративно обрабатываем очень большой текстовый файл — можем на каждой итерации запоминать положение указателя и при повторном запуске скрипта начинать не с начала, а с последней обработанной позиции. Следует отметить, что в реальном коде такие манипуляции выполняются нечасто.

# Сериализация данных

Предположим, что мы работаем над серьёзным проектом, состоящим из нескольких логических модулей. Например, есть модуль, отвечающий за преобразование исходных данных в нужный формат. И есть другой модуль, где что-то с этими преобразованными данными делают — реализуют некоторую бизнес-логику. Как реализовать обмен данными между удалёнными частями системы? Один из вариантов решения — использовать файлы. Тут возникает вопрос: в каком виде хранить эти данные? Принципиально есть два способа: текстовый и бинарный.

Преимущество текстового формата — универсальность. Текстовые данные можно передавать практически любому адресату и любым способом, хоть в url-адресе — как, впрочем, и происходит в вебе (GET-запросы). Есть и недостаток: избыточность, особенно для чисел — получаем большой объём. Второй недостаток — нужно задавать или описывать способ хранения различных структур данных. Этот способ должны знать обе стороны: и отправитель, и получатель. Третий недостаток — трудности реализации хранения сложных структур, особенно с большой вложенностью. Но именно из-за универсальности и стабильности очень часто выбирают текстовый формат. Вам наверняка знакомы расширения .csv, .xml, .json, .prn. Ещё одним преимуществом является возможность «посмотреть» на данные в обычном текстовом редакторе.

В бинарном формате данные занимают намного меньше места, что позволяет быстрее их загрузить с диска. Так мы храним исполняемые файлы, медиа файлы, дампы данных и многое другое. Самым главным недостатком бинарного формата является меньшая универсальность — данные, которые вы сохранили в Python, не удастся прочитать в другом языке программирования.

Вернёмся к текстовому формату. Как, например, мы можем сохранить список? Если он содержит числа или строки, то вполне подойдёт результат вывода через функцию print(). То есть будем задавать границы через квадратные скобки, а элементы отделять запятой. Строки будем оборачивать в кавычки. То, что мы только что описали, называют [сериализацией](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) — превращением объекта в некоторый заданный формат (например, строковый) для последующей передачи или хранения. Это очень важная задача при разработке [web API](https://en.wikipedia.org/wiki/Web_API). То есть вам надо решить, как вы будете преобразовывать объекты для передачи в виде строки в клиентскую часть: браузер или мобильное приложение. Обычно для этого пишут свои [сериализаторы](https://www.django-rest-framework.org/api-guide/serializers/). Но для большого круга задач достаточно встроенных в Python, например, из модуля [json](https://docs.python.org/3.8/library/json.html).

# Сериализация при помощи модуля json

Формат [JSON](https://en.wikipedia.org/wiki/JSON) можно считать «старожилом». Хоть он и был официально описан в 2006 году в [rfc4627](https://tools.ietf.org/html/rfc4627), но упоминался еще в 2000-х. Изначально он задумывался для передачи объектов JavaScript, но вышел далеко за пределы этой задачи. Параллельно с ним зачастую упоминают формат [XML](https://ru.wikipedia.org/wiki/XML). Он похож на html —тоже используются теги, но более универсальный.

На самом деле для передачи данных при помощи текста (строк) необходима реализация двух процессов: сериализация (encoder) и десериализация (decoder) — обратное преобразование. В Python-модуле json получаем [следующее соответствие](https://docs.python.org/3.8/library/json.html#encoders-and-decoders):

| **JSON** | **Python** |
| --- | --- |
| object | dict |
| array | list |
| string | str |
| number (int) | int |
| number (real) | float |
| true | True |
| false | False |
| null | None |

Попробуем преобразовать список:

| import json import random  nums = [random.randint(0, 100) for \_ in range(10)]  nums\_as\_str = json.dumps(nums) print(nums, type(nums)) print(nums\_as\_str, type(nums\_as\_str)) *# [75, 97, 21, 90, 20, 60, 100, 96, 72, 39] <class 'list'>* *# [75, 97, 21, 90, 20, 60, 100, 96, 72, 39] <class 'str'>* |
| --- |

Использовали функцию [dumps()](https://docs.python.org/3.8/library/json.html#json.dumps), которая выполняет задачу сериализации Python-объектов в соответствии с приведенной выше таблицей. При выводе через функцию print() не видим никаких отличий, потому что типы данных [array](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array) в JavaScript и list в Python похожи по синтаксису. Но есть и принципиальная разница: в первом случае объект имеет тип список, а во втором — строка. При попытке сохранить данные в текстовый файл это сразу проявится:

| ... with open('nums.json', 'w', encoding='utf-8') as f:  f.write(nums) *# ... TypeError: write() argument must be str, not list* |
| --- |

Обычный список сохранить как текст не получилось.

| ... with open('nums.json', 'w', encoding='utf-8') as f:  f.write(nums\_as\_str) *# Process finished with exit code 0* |
| --- |

А сериализованный список — получилось.

Попробуем теперь загрузить и десериализовать список:

| import json  with open('nums.json', 'r', encoding='utf-8') as f:  nums\_as\_str = f.read() nums = json.loads(nums\_as\_str) print(nums, type(nums)) *# [38, 13, 26, 48, 36, 90, 24, 44, 15, 61] <class 'list'>* |
| --- |

Всё получилось. Для десериализации использовали функцию [loads()](https://docs.python.org/3.8/library/json.html#json.loads). Не всегда эти манипуляции проходят [гладко](https://stackoverflow.com/questions/18337407/saving-utf-8-texts-in-json-dumps-as-utf8-not-as-u-escape-sequence), но если вы получили [ошибку](https://stackoverflow.com/questions/3768895/how-to-make-a-class-json-serializable) сериализации, значит, уже добрались до серьёзных задач и сможете найти решение.

Вы уже задумались о том, что можно сделать обёртку, которая будет выполнять сериализацию/десериализацию, а затем сохранение/загрузку файла? Если это так — отлично! Есть хорошие новости: такие обёртки существуют — это функции [dump()](https://docs.python.org/3.8/library/json.html#json.dump) и [load()](https://docs.python.org/3.8/library/json.html#json.load). Коварство заключается в разнице в одну букву, поэтому очень часто начинающие разработчики получают ошибки при работе с модулем json. Перепишем пример заново:

| import json import random  nums = [random.randint(0, 100) for \_ in range(10)]  with open('nums\_again.json', 'w', encoding='utf-8') as f:  json.dump(nums, f)  with open('nums\_again.json', 'r', encoding='utf-8') as f:  nums = json.load(f) |
| --- |

Разумеется, в реальной практике сохранение и загрузка будут выполняться в разные моменты времени и, скорее всего, в разных модулях. Здесь мы только хотели показать, насколько просто можно в Python решать простые задачи.

Рекомендуем внимательно познакомиться с [официальной документацией](https://docs.python.org/3.8/library/json.html#module-json) и понять назначение всех аргументов рассмотренных функций. Попробуйте самостоятельно сериализовать словари.

# Работа с бинарными файлами. Модуль pickle

Как мы уже говорили, хранение данных в бинарном виде позволяет существенно уменьшить объём файлов и повысить скорость их загрузки. Зачастую мы даже не подозреваем, что работаем с бинарными файлами — обработка изображений и видео, документы Microsoft Office, анализ исполняемых файлов ОС, загрузка или сохранение из баз данных.

Рано или поздно вам понадобится сохранить в Python объект некоторого своего класса. Тут модуль json «из коробки» не поможет, нужно будет писать свой сериализатор. Во многих подобных случаях выручает модуль [pickle](https://docs.python.org/3.8/library/pickle.html#module-pickle). В нем реализованы точно такие же функции, что и в модуле json: [dump()](https://docs.python.org/3/library/pickle.html#pickle.dump), [dumps()](https://docs.python.org/3/library/pickle.html#pickle.dumps), [load()](https://docs.python.org/3/library/pickle.html#pickle.load), [loads()](https://docs.python.org/3/library/pickle.html#pickle.loads). Разница только в необходимости дописывать “b” для режима доступа к файлу.

Проведём эксперимент для задачи: «В результате работы скрипта вычисляется список из одного миллиона вещественных чисел, который необходимо сохранить на диске и загрузить при его повторном запуске или для других скриптов в проекте — нужен своего рода файловый кеш». Решим задачу при помощи модулей json и pickle, при этом будем профилировать время:

| import json import pickle import random from time import perf\_counter  nums = [random.random() \* 10 \*\* 6 for \_ in range(10 \*\* 6)]  start = perf\_counter() with open('random\_million.json', 'w', encoding='utf-8') as f:  json.dump(nums, f) print(f'json saved: {perf\_counter() - start}')  start = perf\_counter() with open('random\_million.pickle', 'wb') as f:  pickle.dump(nums, f) print(f'pickle saved: {perf\_counter() - start}') *# json saved: 4.176540619* *# pickle saved: 0.14387575499999983* |
| --- |

Как и предполагали, бинарный формат даёт очень серьёзное преимущество в скорости. Сравним размеры файлов:

* random\_million.pickle — 8,58 МБ;
* random\_million.json — 18,2 МБ.

С бинарным файлом получили более чем двукратное преимущество с точки зрения объёма на жестком диске. Теперь загрузим данные:

| import json import pickle from time import perf\_counter  start = perf\_counter() with open('random\_million.json', 'r', encoding='utf-8') as f:  nums = json.load(f) print(f'json loaded: {perf\_counter() - start}, {type(nums)}, {len(nums)}')  start = perf\_counter() with open('random\_million.pickle', 'rb') as f:  nums = pickle.load(f) print(f'pickle loaded: {perf\_counter() - start}, {type(nums)}, {len(nums)}') *# json loaded: 0.9602890079999999, <class 'list'>, 1000000 # pickle loaded: 0.2095523749999999, <class 'list'>, 1000000* |
| --- |

Здесь бинарный формат снова вне конкуренции. Наверняка у вас возникает вопрос: зачем же тогда используют сериализацию в строку, если у бинарного формата такое существенное преимущество и в скорости, и в объёме? Плюс ещё можно сохранять объекты любых классов, не отвлекаясь на написание своих сериализаторов. Причины могут быть разные:

* **проблема совместимости разных языков программирования** — например, асинхронная передача данных от сервера, работающего на Python (Django, Flask), к фронтенду, работающему на JavaScript;
* **необходимость оперативного просмотра данных** — например, при отладке алгоритмов или анализе фрагментов лога можно json- или текстовые данные посмотреть в любом редакторе;
* **совместимость с будущими версиями приложения** — например, бинарные данные моделей, сохраненные в одной версии Django, могут не читаться в другой версии фреймворка.

\*При работе с бинарными файлами можно читать данные порциями — количество считываемых байт передаётся как аргумент в метод read() файлового объекта:

| import pickle  chunk\_size = 256 with open('random\_million.pickle', 'rb') as f:  binary\_data = bytearray()  while True:  chunk = f.read(chunk\_size)  if not chunk:  break  binary\_data.extend(chunk)  nums = pickle.loads(binary\_data) print(f'{type(nums)}, {len(nums)}') *# <class 'list'>, 1000000* |
| --- |

Здесь читали данные порциями по 256 байт и добавляли в объект класса [bytearray](https://docs.python.org/3.8/library/stdtypes.html#bytearray) — массив байт, точнее, использовали метод .extend() этого объекта. По сути это типизированный список. Опять видим стройность в структуре языка Python. Наверняка у вас возник вопрос: есть ли аналогичный тип, но неизменяемый? Есть — это [bytes](https://docs.python.org/3.8/library/stdtypes.html#bytes): по сути, типизированный кортеж. У класса str есть два метода для преобразования в последовательность байт и наоборот:

| txt = 'Привет,Python!' txt\_binary = txt.encode(encoding='utf-8') txt\_origin = txt\_binary.decode(encoding='utf-8') print(txt\_binary, type(txt\_binary)) print(txt\_origin, type(txt\_origin)) *# b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82,Python!' <class 'bytes'>* *# Привет,Python! <class 'str'>* |
| --- |

Это дополнительный материал, поэтому вам придется самостоятельно изучить документацию по методам [.encode()](https://docs.python.org/3.8/library/stdtypes.html#str.encode) и [.decode()](https://docs.python.org/3.8/library/stdtypes.html#bytes.decode). Они часто используются в реальных проектах.

# Практическое задание

1. Не используя библиотеки для парсинга, [распарсить](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7) (получить определённые данные) файл логов web-сервера nginx\_logs.txt  
   (<https://github.com/elastic/examples/raw/master/Common%20Data%20Formats/nginx_logs/nginx_logs>) — получить список кортежей вида: (<remote\_addr>, <request\_type>, <requested\_resource>). Например:

| [  ...  ('141.138.90.60', 'GET', '/downloads/product\_2'),  ('141.138.90.60', 'GET', '/downloads/product\_2'),  ('173.255.199.22', 'GET', '/downloads/product\_2'),  ... ] |
| --- |

1. \*(вместо 1) Найти IP адрес спамера и количество отправленных им запросов по данным файла логов из предыдущего задания.

Примечание: спамер — это клиент, отправивший больше всех запросов; код должен работать даже с файлами, размер которых превышает объем ОЗУ компьютера.

1. Есть два файла: в одном хранятся ФИО пользователей сайта, а в другом — данные об их хобби. Известно, что при хранении данных используется принцип: одна строка — один пользователь, разделитель между значениями — запятая. Написать код, загружающий данные из обоих файлов и формирующий из них словарь: ключи — ФИО, значения — данные о хобби. Сохранить словарь в файл. Проверить сохранённые данные. Если в файле, хранящем данные о хобби, меньше записей, чем в файле с ФИО, задаём в словаре значение None. Если наоборот — выходим из скрипта с кодом «1». При решении задачи считать, что объём данных в файлах во много раз меньше объема ОЗУ.

Фрагмент файла с данными о пользователях (users.csv):

| Иванов,Иван,Иванович  Петров,Петр,Петрович |
| --- |

Фрагмент файла с данными о хобби (hobby.csv):

| скалолазание,охота  горные лыжи |
| --- |

1. \*(вместо 3) Решить задачу 3 для ситуации, когда объём данных в файлах превышает объём ОЗУ (разумеется, не нужно реально создавать такие большие файлы, это просто задел на будущее проекта). Только теперь не нужно создавать словарь с данными. Вместо этого нужно сохранить объединенные данные в новый файл (users\_hobby.txt). Хобби пишем через двоеточие и пробел после ФИО:

| Иванов,Иван,Иванович: скалолазание,охота  Петров,Петр,Петрович: горные лыжи |
| --- |

1. \*\*(вместо 4) Решить задачу 4 и реализовать интерфейс командной строки, чтобы можно было задать имя обоих исходных файлов и имя выходного файла. Проверить работу скрипта.
2. Реализовать простую систему хранения данных о суммах продаж булочной. Должно быть два скрипта с интерфейсом командной строки: для записи данных и для вывода на экран записанных данных. При записи передавать из командной строки значение суммы продаж. Для чтения данных реализовать в командной строке следующую логику:

* просто запуск скрипта — выводить все записи;
* запуск скрипта с одним параметром-числом — выводить все записи с номера, равного этому числу, до конца;
* запуск скрипта с двумя числами — выводить записи, начиная с номера, равного первому числу, по номер, равный второму числу, включительно.

Подумать, как избежать чтения всего файла при реализации второго и третьего случаев.

Данные хранить в файле bakery.csv в кодировке utf-8. Нумерация записей начинается с 1. Примеры запуска скриптов:

| python add\_sale.py 5978,5 python add\_sale.py 8914,3 python add\_sale.py 7879,1 python add\_sale.py 1573,7 python show\_sales.py 5978,5 8914,3 7879,1 1573,7 python show\_sales.py 3 7879,1 1573,7 python show\_sales.py 1 3 5978,5 8914,3 7879,1 |
| --- |

1. \*(вместо 6) Добавить возможность редактирования данных при помощи отдельного скрипта: передаём ему номер записи и новое значение. При этом файл не должен читаться целиком — обязательное требование. Предусмотреть ситуацию, когда пользователь вводит номер записи, которой не существует.

Задачи со \* предназначены для продвинутых учеников, которым мало сделать обычное задание.

# Дополнительные материалы

1. [Лутц Марк. Изучаем Python](https://www.ozon.ru/context/detail/id/5730448/).
2. [Чтение и запись файлов в Python](https://docs.python.org/3.8/tutorial/inputoutput.html#tut-files).
3. [Модуль marshal для сериализации в Python](https://docs.python.org/3.8/library/marshal.html#module-marshal).
4. [Библиотека для сериализации marshmallow](https://marshmallow.readthedocs.io/en/stable/).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://realpython.com/python-json/>.
2. <https://realpython.com/read-write-files-python/>.
3. <https://docs.python.org/3/>.