**Тема занятия №10: Работа с датой и временем. Файлы и каталоги**

**1. Введение**

Python содержит несколько инструментов, которые разработчик может использовать для работы с датой и временем. Мы изучим принципы их работы и способы их применения. Начнем с модуля datetime.

**Модуль datetime**

Мы изучим следующие классы модуля:

datetime.date

datetime.timedelta

datetime.datetime

Благодаря этим классам мы в будущем сможем работать с большей частью случаев, в которых нам понадобятся объекты date и datetime. Также существует класс tzinfo, который применяется для работы с часовыми поясами. Для дополнительной информации об этом классе вы можете обратиться к документации Python.

**2. Модуль datetime**

**datetime.date**

Python может представлять даты различными способами. Для начала, рассмотрим формат datetime.date, так как это один из самых простых объектов date.

**print( datetime.date(2012, 13, 14) )**

**Traceback (most recent call last):**

**File "<string>", line 1, in <fragment>**

**builtins.ValueError: month must be in 1..12**

**print(datetime.date(2012, 12, 14)) # datetime.date(2012, 12, 14)**

В данном коде показано, как создать простой объект даты. Класс date принимает три аргумента: год, месяц и день. Если вы укажите неправильное значение, вы увидите ошибку ValueError, как в случае выше. В противном случае вы увидите, что объект datetime.date вернулся. Давайте взглянем на еще один пример:

**import datetime**

**d = datetime.date(2012, 12, 14)**

**print(d.year) # 2012**

**print(d.day) # 14**

**print(d.month) # 12**

Здесь мы присваиваем объекту date переменную d. Теперь мы можем получить доступ к различным компонентам даты по названиям, таким как d.year или d.month. Давайте посмотрим, какой сейчас день:

**import datetime**

**print(datetime.date.today()) # datetime.date(2017, 4, 5)**

Это может быть полезно, когда вам нужно записать, какой сейчас день. Или, если вам нужно выполнить основанный на сегодняшней дате расчет. Мелочь, а приятно.

**datetime.datetime**

Объект datetime.datetime содержит всю информацию объектов datetime.date плюс datetime.time. Давайте приведем несколько примеров, для лучшего понимания разницы между этим объектом, и объектом datetime.date.

**import datetime**

**a = datetime.datetime(2017, 3, 5)**

**print(a) # datetime.datetime(2017, 3, 5, 0, 0)**

**b = datetime.datetime(2017, 3, 5, 12, 30, 10)**

**print(b) # datetime.datetime(2017, 3, 5, 12, 30, 10)**

**d = datetime.datetime(2017, 3, 5, 12, 30, 10)**

**print(d.year) # 2017**

**print(d.second) # 10**

**print(d.hour) # 12**

Мы видим, что datetime.datetime принимает несколько дополнительных аргументов: год, месяц, день, час, минута и секунда. Это также позволяет вам указывать информацию о микросекундах и часовом поясе. При работе с базами данных, данные типы объектов будут использоваться достаточно часто. Большую часть вашей работы, вам нужно будет конвертировать форматы date или datetime Python в форматы SQL datetime или timestamp. Обратите внимание на то, что today совместно с datetime.datetime использует два разных метода:

**import datetime**

**a = datetime.datetime.today()**

**print(a) # datetime.datetime(2017, 4, 5, 0, 16, 54, 989663)**

**b = datetime.datetime.now()**

**print(b) # datetime.datetime(2017, 4, 5, 0, 17, 8, 24239)**

Модуль datetime содержит другой метод, под названием strftime. Этот метод позволяет разработчику создавать строку, отображающую время в более понятной для человека форме. Давайте взглянем на несколько примеров, показывающих всю полезность данного метода:

**import datetime**

**a = datetime.datetime.today().strftime("%Y%m%d")**

**print(a) # '20170405'**

**today = datetime.datetime.today()**

**print( today.strftime("%m/%d/%Y") ) # '04/05/2017'**

**print( today.strftime("%Y-%m-%d-%H.%M.%S") ) # 2017-04-05-00.18.00**

Первый пример – это скорее хитрость. В нем показано, как конвертировать сегодняшний объект datetime в строку, следующую за форматом YYYYMMDD (ГГГГММДД). Второй пример более наглядный.

В нем мы присваиваем объект datetime переменной под названием today и применяем два разных параметра форматирования строки. Первый параметр добавляет косые черточки между элементами datetime, а также перегруппировывает datetime, теперь он делится на месяц, день и год. В последнем примере мы создаем временную отметку, которая следует типичному формату: YYYY-MM-DD.HH.MM.SS. Если вам нужно указать год как двухзначный (“YY”), вы можете заменить %Y на %y.

**datetime.timedelta**

Объект datetime.timedelta отображает длительность времени. Другими словами, это разница между двумя датами или временными отметками. Давайте взглянем на наглядный пример:

**import datetime**

**# Значение: datetime.datetime(2017, 4, 5, 0, 18, 51, 980187)**

**now = datetime.datetime.now()**

**then = datetime.datetime(2017, 2, 26)**

**# Кол-во времени между датами.**

**delta = now - then**

**print(delta.days) # 38**

**print(delta.seconds) # 1131**

Мы создали два объекта datetime. Один указывает на сегодняшний день, второй – на прошедшую неделю. После этого, мы берем разницу между ними. Это возвращает объект timedelta, который мы можем далее использовать, чтобы выяснить, сколько прошло дней или секунд, между этими двумя датами. Если вам нужно узнать количество часов или минут между двумя датами, вам понадобится немножко математики, чтобы выяснить это. Давайте взглянем на проверенный способ:

**seconds = delta.total\_seconds()**

**hours = seconds // 3600**

**print(hours) # 186.0**

**minutes = (seconds % 3600) // 60**

**print(minutes) # 13.0**

Отсюда мы узнаем, что в неделе 186 часов и 13 минут. Обратите внимание на то, что мы используем двойную косую черту в качестве нашего оператора деления, также известного как floor division. Теперь мы готовы к тому, чтобы узнать больше о модуле time Python.

Модуль time открывает разработчику Python доступ к нескольким связанным со временем функциям. Модуль основан на «эпохе», точке, с которой начинается время. Для систем Unix, эпоха началась в 1970 году. Чтобы узнать, какая эпоха в вашей системе, попробуйте запустить следующий код:

**import time**

**print(time.gmtime(0))**

**Результат**

**time.struct\_time(tm\_year=1970, tm\_mon=1, tm\_mday=1, tm\_hour=0, tm\_min=0, tm\_sec=\**

**0, tm\_wday=3, tm\_yday=1, tm\_isdst=0)**

Запустил его на Windows 7, которая также уверена в том, что начало времен датируется 1970м годом. В любом случае, в данном разделе мы ознакомимся со следующими функциями:

**time.ctime**

**time.sleep**

**time.strftime**

**time.time**

**Приступим!**

**time.ctime**

Функция time.ctime конвертирует время в секунды, начиная с эпохи, в строку, показывающую местное время. Если вы ничего не передадите данной функции, то настоящее время вернется обратно. Давайте взглянем на несколько примеров:

**import time**

**print(time.ctime()) # 'Wed Apr 5 00:02:49 2017'**

**print(time.ctime(1384112639)) # 'Sun Nov 10 13:43:59 2013'**

В данном коде мы показали результаты вызова функции time.ctime, как со случайным набором секунд, начиная с эпохи, так и с пустым значением. Это можно применить, к примеру, в том случае, кода кому-нибудь нужно сохранить дату в секундах, начиная с эпохи, после чего конвертировать это в более удабриваемый формат. Немного проще, сохранить большое целое число (или длинное) в базу данных, чем страдать над ним, форматируя объект datetime в какой-либо объект даты, который принимает база данных. Конечно, это также имеет свой недостаток: вам, возможно, нужно будет преобразовать целое число или значение с плавающей запятой обратно в строку.

**time.sleep**

Функция time.sleep дает разработчику возможность приостановить выполнение его скрипта на определенное количество секунд. Это можно сравнить с добавлением функции «Пауза» в вашу программу. Я нашел этот класс особенно полезным, когда мне нужно было подождать несколько секунд, пока закроется файл, или база данных закончит выполнять свою задачу. Давайте взглянем на пример. Откройте новое окно в IDLE и сохраните следующий код:

**import time**

**for x in range(5):**

**time.sleep(2)**

**print("Slept for 2 seconds")**

Теперь запустите этот код в IDLE. Сделав это, вы увидите фразу «Slept for 2 seconds» пять раз, с интервалом в 2 секунды между каждым сообщением. Это очень легко в использовании!

**time.strftime**

Модуль time имеет функцию strftime, которая работает по схожему с версией datetime принципу. Разница заключается, главным образом, в том, что именно она принимает для ввода: кортеж, или объект struct\_time, вроде тех, которые возвращаются после вызова time.gmtime() или time.localtime(). Вот небольшой пример:

**import time**

**a = time.strftime("%Y-%m-%d-%H.%M.%S", time.localtime())**

**print(a) # '2017-04-05-00.11.20'**

Этот код очень похож на код временной отметки, который мы создавали в разделе datetime. На мой взгляд, метод datetime более интуитивен при создании объекта datetime.datetime а затем при вызове его метода strftime в том формате, который вам более удобен. С модулем time, вам нужно передать формат и кортеж времени. На самом деле выбор за вами, каким именно способом вам будет удобнее решить вашу задачу.

**time.time**

Функция time.time отображает время в секундах, начиная с эпохи, как число с плавающей запятой. Давайте взглянем:

**import time**

**x = time.time()**

**print(x) # 1491340367.478385**

Весьма просто. Вы можете использовать данную функцию, когда вам нужно сохранить настоящее время в базу данных, но вам не нужно конвертировать его в метод datetime вашей базы данных. Вы можете вспомнить о том, что метод ctime принимает время в секундах, так что мы можем использовать time.time для получения количества секунд для передачи их ctime, вот так:

**import time**

**a = time.ctime(time.time())**

**print(a) # Wed Apr 5 00:13:47 2017**

**3. Получение текущей даты и времени**

Для вывода даты и времени нужно выполнить следующее:

**import datetime**

**day\_time = datetime.datetime.now()**

**print(day\_time)**

**Получение текущей даты и времени в Python datetime**

Как видно мы получаем для создания объекта даты и времени используется класс с тем же именем что и модуль datetime.

Когда нужно получить только дату используется класс date:

**import datetime**

**day = datetime.date.today()**

**print(day)**

Получение только времени выполняется через метод time():

**import datetime**

**date\_time = datetime.datetime.now().time()**

**print(date\_time, type(date\_time))**

Каждый из описываемых классов можно импортировать следующим способом:

**from datetime import date**

**from datetime import datetime**

**print(datetime.now())**

**print(date.today())**

**4. Форматирование и перевод в строку**

Для получения части даты или времени можно использовать следующие атрибуты:

year

month

day

weekday

hour

minute

second

**import datetime**

**# Получение дня от даты**

**date = datetime.date.today()**

**print(date.day)**

**date\_time = datetime.datetime.now()**

**# Получение часа суток от времени**

**print(date\_time.hour)**

Так же есть метод strftime, который форматирует даты в нужном формате в строку.

Например, так мы получим дату в формате, который используется у нас:

**import datetime**

**date = datetime.datetime.today()**

**print(date.strftime('%d-%m-%Y %H:%M'))**

Где:

%d - день месяца с 1 по 31;

%m - месяц с 1 по 12;

%Y - год;

%H - час в формате 0-24;

%M - минуты;

%S - секунды.

Таким же способом можно получить время и дату:

%c - время и дата;

%x - дата;

%X - время.

date = datetime.datetime.today()

print(date.strftime('%x'))

Обратите внимание, что таким способом мы преобразуем объект класса datetime в строку и мы больше не сможем использовать методы по работе с датой (например сравнение):

**import datetime**

**date = datetime.datetime.today()**

**local\_date = date.strftime('%d-%m-%Y %H:%M')**

**print('Объект класса datetime: ', type(date))**

**print('Дата преобразованная в строку: ', type(local\_date))**

**print(local\_date.day)**

Мы получим ошибку так как уже работаем со строкой:

**AttributeError: 'str' object has no attribute 'day'**

Выше описаны основные возможности форматирования используя метод strftime(), но их, конечно, больше.

**5. Получения дня недели и название месяца**

Можно получить название дня недели или название. Численный вариант эквивалентен следующим значениям:

0 - Monday (Понедельник);

1 - Tuesday (Вторник);

2 - Wednesday (Среда);

3 - Thursday (Четверг);

4 - Friday (Пятница);

5 - Saturday (Суббота);

6 - Sunday (Воскресенье).

Следующий пример вернет день недели в виде числа:

**date = datetime.datetime.today()**

**print(date.weekday())**

Или получить название:

**date = datetime.datetime.today()**

**print(date.strftime('%A'))**

**print(date.strftime('%a'))**

**Получение имени дня недели в Python datetime**

Где:

%A - полное название дня недели;

%a - сокращенное название дня недели;

%s - представление в виде числа.

Такой же принцип по работе с месяцами, где:

%B - полное название месяца;

%b - сокращенное название месяца;

%m - месяц в виде числа.

**Создание объекта даты и времени**

Для создания даты используется класс с аргументами date(год, месяц, число):

**date = datetime.date(2019, 1, 13)**

**print(date, type(date))**

Можно создать дату и время. В обоих случаях мы можем использовать именованные аргументы если так удобнее:

**date\_time = datetime.datetime(year = 2019,**

**month = 2,**

**day = 4,**

**hour = 7,**

**minute = 9,**

**second = 13 )**

**print(date\_time, type(date\_time))**

Каждый аргумент времени, по умолчанию, имеет значение 0. Мы так же можем использовать подход выше для получения, например, только года или времени:

**date\_time = datetime.datetime(year = 2019,**

**month = 2,**

**day = 4,**

**second = 13 )**

**# Получаем только время**

**time = date\_time.strftime('%X')**

**print(time, type(time))**

**date = datetime.date(**

**year = 2018,**

**month = 5,**

**day = 3**

**)**

**# Получаем только год**

**year = date.year**

**print(year, type(year))**

Если вы не укажете год, месяц или день, то получите ошибку т.к. они по умолчанию равны None:

**TypeError: Required argument 'year' (pos 1) not found**

**Есть отдельный класс для создания времени time:**

**time = datetime.time(12, 33, second=33, microsecond=122)**

**print(time, type(time))**

**print(time.hour)**

Используя strptime() можно создавать объект datetime из строки:

**date\_string = '21 September. 1999'**

**date\_object = datetime.strptime(date\_string, '%d %B. %Y')**

**print(date\_object, type(date\_object))**

timedelta - это класс с помощью которого можно установить не дату, как в примерах выше, а продолжительность. Так мы создадим объект с 12 днями и 33 секундами:

**duration\_time = datetime.timedelta(days = 12, seconds = 33)**

**print(duration\_time, type(duration\_time))**

**Продолжительность с timedelta в Python datetime**

Все атрибуты, которые мы можем указывать для этого класса:

days

seconds

microseconds

milliseconds

minutes

hours

Кроме этого мы можем преобразовывать эти объекты в секунды:

**duration\_time = datetime.timedelta(minutes = 1, seconds = 10)**

**# Преобразуем в секунды**

**seconds = duration\_time.total\_seconds()**

**print(seconds, type(seconds))**

**6. Разница между датами**

Мы можем искать разницу между датами получая объект timedelta:

**date\_now = datetime.datetime.now()**

**date\_tomorrow = datetime.datetime(2019, 12, 29)**

**# Вычисляем разницу между датами**

**difference = date\_tomorrow - date\_now**

**print(difference, type(difference))**

**print('Разница только в днях: ', difference.days)**

Каждый их объектов выше можно изменить. Так мы изменим объект timedelta прибавив часы к минутам:

**duration\_hours = datetime.timedelta(hours = 1)**

**duration\_minutes = datetime.timedelta(minutes = 10)**

**result = duration\_hours + duration\_minutes**

**print(result, type(result))**

С помощью timedelta изменяется и дата. Пример ниже изменяет текущую дату прибавляя к ней 1 день и 1 час:

**date\_now = datetime.datetime.now()**

**duration\_minutes = datetime.timedelta(days=1, minutes=10)**

**result = date\_now + duration\_minutes**

**print(result, type(result))**

**7. Сравнение и условия**

Каких-либо хитростей в сравнении объектов нет. В следующем примере идет сравнение двух дат с разницей в день:

**date\_one = datetime.datetime(2017,12,1)**

**date\_two = datetime.datetime(2017,12,2)**

**if date\_two > date\_one:**

**print('{0} больше чем {1}'.format(date\_two, date\_one))**

**Сравнение объектов datetime в Python**

При этом стоит проверять, что объекты относятся к классу datetime, а не являются строками. В следующем примере мы сравниваем года, но они уже относятся к численным:

**date\_one = datetime.datetime(2017,12,1)**

**date\_two = datetime.datetime(2017,12,2)**

**if date\_two.year == date\_one.year:**

**print('Год одинаковый. Тип данных: ', type(date\_two.year))**

Объект timedelta тоже можно сравнивать:

**time\_one = datetime.timedelta(seconds=2)**

**time\_two = datetime.timedelta(minutes=1)**

**if time\_one < time\_two:**

**print('Pass')**

**8. Работа с метками (штампами) timestamp**

При работе с API или в Unix системах мы можем увидеть отображение времени в таком формате:

1578238360

Данные числа обозначают количество секунд с 1 января 1970 года. Мы можем конвертировать данное число в понятный формат используя datetime:

**timestamp = date.fromtimestamp(1578238360)**

**print(timestamp)**

Так же сработает если мы хотим получить и время:

**timestamp = datetime.fromtimestamp(1578238360)**

**print(timestamp)**

Для конвертирования в timestamp используется метод с аналогичным названием:

**from datetime import datetime**

**date\_now = datetime.now()**

**print(date\_now.timestamp)**

**Работа с часовыми поясами**

Библиотека datetime не хранит часовые пояса, данные о переводах часов (летнее и зимнее время) и високосных секундах. К тому же, некоторые страны, могут изменить время опираясь на локальные ситуации. Эти ситуации опасны, когда идет запись в базу данных. Для вывода в GUI, можно использовать datetime.datetime.now() или высчитывать часовой пояс из базы.

Для записи в базу данных мы можем использовать время UTC и отдельно считать часовой пояс:

**utc = datetime.datetime.utcnow()**

**print(utc)**

**Время UTC в Python Datetime**

Следующий пример вычислит разницу времени между UTC и локальным.

**gmt = datetime.today() - datetime.utcnow()**

**print(gmt)**

Для вычисления других часовых поясов можно использовать стороннюю библиотеку pytz, которая их хранит:

pip install pytz

Вывести все часовые пояса мы можем так:

**import pytz**

**for tz in pytz.all\_timezones:**

**print(tz)**

На примере ниже я перевожу локальное время в часовой пояс Киева:

**import pytz**

**from datetime import datetime**

**tz\_kiev = pytz.timezone('Europe/Kiev')**

**time\_kiev = datetime.now(tz\_kiev)**

**print(time\_kiev)**

**1. Доступ к файлам в Python**

Чаще всего разработчики в своей работе сталкиваются с обработкой данных, которые хранятся в файлах. А файлы, в свою очередь, обычно хранятся на устройствах хранения данных — жестких, оптических, сетевых или твердотельных накопителях.

Легко представить программу, которая сортирует 20 чисел, и столь же легко представить пользователя этой программы, который вводит эти 20 чисел прямо с клавиатуры.

Гораздо сложнее представить ту же самую задачу, когда нужно отсортировать 20 000 чисел. Ну а пользователей, которые могли бы ввести все эти числа без ошибок, просто не существует.

Намного проще представить, что эти цифры хранятся в файле на диске, который считывает программа. Программа сортирует числа и не отправляет их на экран, а создает новый файл и сохраняет там отсортированную последовательность чисел.

Если нужно реализовать простую базу данных, единственный способ сохранить информацию между запусками программы — сохранить ее в файл (или файлы, если база данных более сложная).

В принципе, любая сложная задача в программировании зависит от использования файлов, независимо от того, обрабатываются ли изображения (которые хранятся в файлах), умножаются ли матрицы (которые хранятся в файлах) или рассчитываются зарплата и налоги (путем считывания данных, хранящихся в файлах).

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

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

**2. Имена файлов**

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

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



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

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

То есть эти две строки:

ThisIsTheNameOfTheFile

и:

thisisthenameofthefile

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

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

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

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

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

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

В Unix-подобных системах это может выглядеть следующим образом:

name = "/dir/file"

Но если попытаться задать строку для Windows следующим образом:

name = "\dir\file"

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

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

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

name = "\\dir\\file"

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

Это означает, что следующие назначения:

name = "/dir/file"

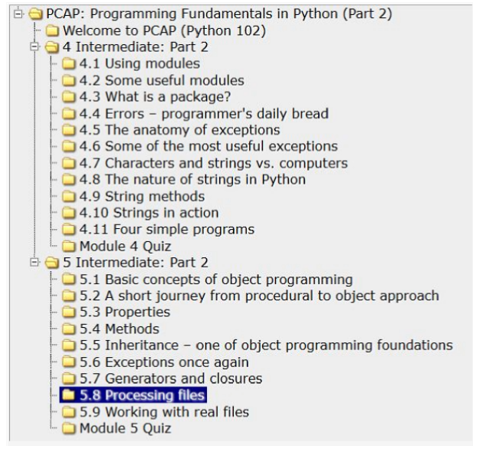
name = "c:/dir/file"

будут работать и в Windows.

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

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

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



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

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

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

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

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

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

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

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

**Файловые потоки**

Открытие потока данных не только связано с файлом, но и должно объявлять способ обработки потока. Такое объявление называется режимом открытия файла.

Если открытие прошло успешно, программа получит разрешение на выполнение только тех операций, которые соответствуют заявленному режиму открытия.

В потоке данных выполняются две основные операции:

■ чтение из потока: части данных извлекаются из файла и помещаются в область памяти, которая управляется программой (например, в переменную);

■ запись в поток: часть данных памяти (например, переменная) передается в файл.

Для открытия потока используются три основных режима:

■ режим чтения: поток, открытый в этом режиме, разрешает только операции чтения; попытка записи в поток вызовет исключение (которое называется UnsupportedOperation, оно наследует OSError и ValueError и является потомком модуля io);

■ режим записи: поток, открытый в этом режиме, разрешает только операции записи; попытка прочитать поток вызовет исключение, упомянутое выше;

■ режим обновления: поток, открытый в этом режиме, разрешает и запись, и чтение.

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

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

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

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

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

**3. Файловые дескрипторы**

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

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

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

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

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

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

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

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

Функция анализирует предоставленные аргументы и автоматически создает требуемый объект.

Чтобы закрыть объект, нужно вызвать метод close().

Вызов разорвет соединение с объектом и файлом и удалит объект.

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

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

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

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

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

Поскольку эти файлы не содержат строк, операции чтения и записи устанавливают связи с данными любого размера. Следовательно, данные читаются/записываются байт за байтом или блок за блоком. Размер блока обычно колеблется от одного до произвольно выбранного значения.

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

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

Эта двусмысленность может вызвать разные неприятные последствия.

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

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

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

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

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

■ когда поток открыт и есть рекомендация обработать данные в связанном файле как текст (или такой рекомендации нет), то он переключится в текстовый режим;

■ во время чтения/записи строк в среде Unix ничего особенного не происходит, но когда те же операции выполняются в среде Windows, происходит процесс, который называется преобразованием символов новой строки: когда вы читаете строку из файла, каждая пара символов \r\n заменяется одним символом \n и наоборот. Во время операций записи каждый символ \n заменяется парой символов \r\n;

■ этот механизм абсолютно понятен программе, поэтому программист может писать ее так, как если бы она предназначалась только для обработки текстовых файлов в Unix/Linux, потому что этот исходный код и в среде Windows будет работать правильно;

■ когда поток открыт, и есть соответствующая рекомендация, его содержимое принимается как есть, без преобразований — ни один байт не добавляется и не удаляется.

**4. Открытие потоков**

Открытие потока выполняет функция, которую можно вызвать следующим образом:

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

Давайте проанализируем ее:

■ имя функции (open) говорит само за себя; если открытие прошло успешно, функция возвращает объект потока; в противном случае возникает исключение (например, FileNotFoundError если файла, который вы собираетесь прочитать, не существует);

■ первый параметр функции (file) задает имя файла, который будет связан с потоком;

■ второй параметр (mode) задает режим открытия для этого потока; это строка, заполненная последовательностью символов, и каждый из них имеет свое особое значение (скоро разберем подробнее);

■ третий параметр (encoding) указывает тип кодировки (например, UTF-8 при работе с текстовыми файлами).

■ открытие должно быть самой первой операцией, выполняемой в потоке.

Примечание: аргументы режима и кодирования могут быть опущены, тогда принимаются значения по умолчанию. Режим открытия по умолчанию — это чтение в текстовом режиме, а кодировка по умолчанию зависит от используемой платформы.

Пора познакомиться с самыми важными и полезными режимами открытия. Готовы? Открытие потоков: режимы

■ r: чтение

▷ поток будет открыт в режиме чтения;

▷ файл, связанный с потоком должен существовать и быть доступным для чтения, в противном случае, функция open() вызывает исключение.

■ w: запись

▷ поток будет открыт в режиме записи;

▷ файл, связанный с потоком, не обязан существовать. Если его не существует, он будет создан. Если существует, он будет обрезан до нулевой длины (стерт). Если создание невозможно (например, из-за системных разрешений) функция open() вызывает исключение.

■ a: дозапись

▷ поток будет открыт в режиме дозаписи;

▷ файл, связанный с потоком, не обязан существовать. Если его не существует, он будет создан. Если существует, виртуальная записывающая головка будет установлена в конец файла (предыдущее содержимое файла останется без изменений).

■ r+: чтение и обновление

▷ поток будет открыт в режиме чтения и обновления;

▷ файл, связанный с потоком, должен существовать и быть доступным для записи, в противном случае функция open() вызывает исключение;

▷ операции чтения и записи разрешены для потока.

■ w+: запись и обновление

▷ поток будет открыт в режиме записи и обновления;

▷ файл, связанный с потоком, не обязан существовать. Если его не существует, он будет создан. Предыдущее содержимое файла остается нетронутым;

▷ операции чтения и записи разрешены для потока.

**Выбор текстового и бинарного режимов**

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

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

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

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

Текстовый режим Двоичный режим Описание

rt rb чтение

wt wb запись

at ab дозапись

r+t r+b чтение и обновление

w+t w+b чтение и обновление

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

**5. Первое открытие потока**

Представьте, что мы хотим разработать программу, которая читает содержимое текстового файла с именем:

C:\Users\User\Desktop\file.txt.

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

**try:**

**stream = open("C:\Users\User\Desktop\file.txt", "rt")**

**# здесь происходит обработка**

**stream.close()**

**except Exception as exc:**

**print("Cannot open the file:", exc)**

Что тут происходит?

■ мы открываем блок try-except, поскольку хотим мягко обрабатывать ошибки выполнения;

■ используем функцию open(), чтобы попытаться открыть указанный файл (обратите внимание на то, как мы указали имя файла).

■ режим открытия определяется как текст для чтения (так как текстовый режим является настройкой по умолчанию, мы можем не указывать букву «t» в строке режима).

■ в случае успеха мы получаем объект от функции open() и присваиваем ее переменной потока;

■ если функция open() завершается с ошибкой, мы обрабатываем эту ошибку и выводим полную информацию о ней (всегда полезно знать, что именно произошло).

**Предварительно открытые потоки**

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

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

import sys

потому что объявление этих трех потоков происходит именно здесь.

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

■ sys.stdin

▷ stdin (standard input — стандартный ввод)

▷ поток stdin обычно связан с клавиатурой, предварительно открыт для чтения и рассматривается как основной источник данных для запущенных программ;

▷ хорошо известная функция input () по умолчанию читает данные из stdin.

■ sys.stdout

▷ stdout (standard output — стандартный вывод)

▷ поток stdout обычно связан с экраном, предварительно открыт для записи и рассматривается как основной объект для вывода данных, запущенных программой;

▷ хорошо известная функция print() выводит данные в поток stdout.

■ sys.stderr

▷ stderr (standard error output — стандартный поток ошибок)

▷ поток stderr обычно связан с экраном, предварительно открыт для записи и рассматривается как основное место, куда запущенная программа должна отправлять информацию об ошибках, возникших при ее работе; ▷ мы не представили метода для отправки данных в этот поток (но обещаем это сделать в ближайшее время).

▷ разделение потоков на stdout (полезные результаты, полученные программой) и stderr (сообщения об ошибках, безусловно полезные, но они не выводят результаты) дает возможность перенаправить эти два типа информации разным получателям. Более подробное обсуждение этого вопроса выходит за рамки нашего курса. Руководство по операционной системе даст вам больше информации по этим вопросам.

**6. Закрытие потоков**

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

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

■ название функции (close()) прямо указывает на то, что именно она делает.

■ функция не ожидает никаких аргументов; поток не нужно открывать.

■ функция ничего не возвращает, но генерирует исключение IOError в случае ошибки;

■ большинство разработчиков считает, что функция close() всегда выполняется успешно, поэтому нет необходимости проверять, правильно ли она выполнила свою задачу.

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

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

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

**Диагностика проблем потока**

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

try:

# далее — потоковые операции

except IOError as exc:

print(exc.errno)

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

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

errno.EACCES → Permission denied

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

errno.EBADF → Bad file number

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

errno.EEXIST → File exists

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

errno.EFBIG → File too large

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

errno.EISDIR → Is a directory

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

errno.EMFILE → Too many open files

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

errno.ENOENT → No such file or directory

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

errno.ENOSPC → No space left on device

Эта ошибка возникает, когда на носителе нет свободного места.

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

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

**import errno**

**try:**

**s = open("c:/users/user/Desktop/file.txt", "rt")**

**# здесь происходит фактическая обработка**

**s.close()**

**except Exception as exc:**

**if exc.errno == errno.ENOENT:**

**print("The file doesn’t exist.")**

**elif exc.errno == errno.EMFILE:**

**print("You’ve opened too many files.")**

**else:**

**printf("The error number is:", exc.errno)**

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

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

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

Теперь мы можем упростить код следующим образом:

**from os import strerror**

**try:**

**s = open("c:/users/user/Desktop/file.txt", "rt")**

**# здесь происходит фактическая обработка**

**s.close()**

**except Exception as exc:**

**print("The file could not be opened:",**

**strerror(exc.errno))**

**Работа с файлами и каталогами**

**1. Введение**

Обработка файлов в Python с помощью модуля os включает создание, переименование, перемещение, удаление файлов и папок, а также получение списка всех файлов и каталогов и многое другое.

В индустрии программного обеспечения большинство программ тем или иным образом обрабатывают файлы: создают их, переименовывают, перемещают и так далее. Любой программист должен обладать таким навыком.

Важно знать, что модуль os используется не только для работы с файлами. Он включает массу методов и инструментов для других операций: обработки переменных среды, управления системными процессами, а также аргументы командной строки и даже расширенные атрибуты файлов, которые есть только в Linux.

Модуль встроенный, поэтому для работы с ним не нужно ничего устанавливать.

**2. Вывод текущей директории. Модуль OS**

Для получения текущего рабочего каталога используется os.getcwd():

**import os**

**# вывести текущую директорию**

**print("Текущая директория:", os.getcwd())**

os.getcwd() возвращает строку в Юникоде, представляющую текущий рабочий каталог. Вот пример вывода:

Текущая директория: C:\python3\bin

Создание папки

Для создания папки/каталога в любой операционной системе нужна следующая команда:

**# создать пустой каталог (папку)**

**os.mkdir("folder")**

После ее выполнения в текущем рабочем каталоге тут же появится новая папка с названием «folder».

Если запустить ее еще раз, будет вызвана ошибка FileExistsError, потому что такая папка уже есть. Для решения проблемы нужно запускать команду только в том случае, если каталога с таким же именем нет. Этого можно добиться следующим образом:

**# повторный запуск mkdir с тем же именем вызывает FileExistsError,**

**# вместо этого запустите:**

**if not os.path.isdir("folder"):**

**os.mkdir("folder")**

Функция os.path.isdir() вернет True, если переданное имя ссылается на существующий каталог.

**Изменение директории**

Менять директории довольно просто. Проделаем это с только что созданным:

**# изменение текущего каталога на 'folder'**

**os.chdir("folder")**

Еще раз выведем рабочий каталог:

**# вывод текущей папки**

**print("Текущая директория изменилась на folder:", os.getcwd())**

**Вывод:**

**Текущая директория изменилась на folder: C:\python3\bin\folder**

Создание вложенных папок

Предположим, вы хотите создать не только одну папку, но и несколько вложенных:

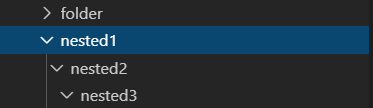
**# вернуться в предыдущую директорию**

**os.chdir("..")**

**# сделать несколько вложенных папок**

**os.makedirs("nested1/nested2/nested3")**

Это создаст три папки рекурсивно, как показано на следующем изображении:



**Создание файлов**

Для создания файлов в Python модули не нужны. Можно использовать встроенную функцию open(). Она принимает название файла, который необходимо создать в качестве первого параметра и желаемый режим открытия — как второй:

**# создать новый текстовый файл**

**text\_file = open("text.txt", "w")**

**# запить текста в этот файл**

**text\_file.write("Это текстовый файл")**

w значит write (запись), a — это appending (добавление данных к уже существующему файлу), а r — reading (чтение). Больше о режимах открытия можно почитать здесь.

Переименование файлов

С помощью модуля os достаточно просто переименовать файл. Поменяем название созданного в прошлом шаге.

**# переименовать text.txt на renamed-text.txt**

**os.rename("text.txt", "renamed-text.txt")**

Функция os.rename() принимает 2 аргумента: имя файла или папки, которые нужно переименовать и новое имя.

**Перемещение файлов**

Функцию os.replace() можно использовать для перемещения файлов или каталогов:

**# заменить (переместить) этот файл в другой каталог**

**os.replace("renamed-text.txt", "folder/renamed-text.txt")**

Стоит обратить внимание, что это перезапишет путь, поэтому если в папке folder уже есть файл с таким же именем (renamed-text.txt), он будет перезаписан.

**Список файлов и директорий**

**# распечатать все файлы и папки в текущем каталоге**

**print("Все папки и файлы:", os.listdir())**

Функция os.listdir() возвращает список, который содержит имена файлов в папке. Если в качестве аргумента не указывать ничего, вернется список файлов и папок текущего рабочего каталога:

Все папки и файлы: ['folder', 'handling-files', 'nested1', 'text.txt']

А что если нужно узнать состав и этих папок тоже? Для этого нужно использовать функцию os.walk():

**# распечатать все файлы и папки рекурсивно**

**for dirpath, dirnames, filenames in os.walk("."):**

**# перебрать каталоги**

**for dirname in dirnames:**

**print("Каталог:", os.path.join(dirpath, dirname))**

**# перебрать файлы**

**for filename in filenames:**

**print("Файл:", os.path.join(dirpath, filename))**

**os.walk()** — это генератор дерева каталогов. Он будет перебирать все переданные составляющие. Здесь в качестве аргумента передано значение «.», которое обозначает верхушку дерева:

Каталог: .\folder

Каталог: .\handling-files

Каталог: .\nested1

Файл: .\text.txt

Файл: .\handling-files\listing\_files.py

Файл: .\handling-files\README.md

Каталог: .\nested1\nested2

Каталог: .\nested1\nested2\nested3

Метод os.path.join() был использован для объединения текущего пути с именем файла/папки.

**Удаление файлов**

Удалим созданный файл:

**# удалить этот файл**

**os.remove("folder/renamed-text.txt")**

**os.remove() удалит файл с указанным именем (не каталог).**

**Удаление директорий**

С помощью функции os.rmdir() можно удалить указанную папку:

**# удалить папку**

**os.rmdir("folder")**

Для удаления каталогов рекурсивно необходимо использовать os.removedirs():

# удалить вложенные папки

os.removedirs("nested1/nested2/nested3")

Это удалит только пустые каталоги.

**Получение информации о файлах**

Для получения информации о файле в ОС используется функция os.stat(), которая выполняет системный вызов stat() по выбранному пути:

**open("text.txt", "w").write("Это текстовый файл")**

**# вывести некоторые данные о файле**

**print(os.stat("text.txt"))**

**Вывод:**

**os.stat\_result(st\_mode=33206, st\_ino=14355223812608232, st\_dev=1558443184, st\_nlink=1, st\_uid=0, st\_gid=0, st\_size=19, st\_atime=1575967618, st\_mtime=1575967618, st\_ctime=1575966941)**

Это вернет кортеж с отдельными метриками. В их числе есть следующие:

st\_size — размер файла в байтах

st\_atime — время последнего доступа в секундах (временная метка)

st\_mtime — время последнего изменения

st\_ctime — в Windows это время создания файла, а в Linux — последнего изменения метаданных

Для получения конкретного атрибута нужно писать следующим образом:

**# например, получить размер файла**

**print("Размер файла:", os.stat("text.txt").st\_size)**

**Вывод:**

**Размер файла: 19**

**Извлечение имени файла из пути**

Функция os.path.split совершает обратное действие — отрезает имя файла или ниже лежащей папки от пути:

**>>> import os**

**>>> path = './work/project/version8/final.txt'**

**>>> dirpath, filename = os.path.split(path)**

**>>> print(dirpath)**

**./work/project/version8**

**>>> print(filename)**

**final.txt**

**>>> project\_dir, version\_dir = os.path.split(dirpath)**

**>>> print(project\_dir)**

**./work/project**

**>>> print(version\_dir)**

**version8**

**Извлечение расширения**

Кроме того, может пригодиться функция os.path.splitext, котоая отрезает расширение файла:

**>>> import os**

**>>> path = './work/project/version12/final.txt'**

**>>> base, ext = os.path.splitext(path)**

**>>> print(base, ext, sep='\n')**

**./work/project/version12/final**

**.txt**

**Проверка типа файла**

Кроме прочего, модуль os.path содержит функции для проверки существования файла и для определения его типа:

**>>> import os**

**>>> path = './kursach/text'**

**>>> if os.path.exists(path):**

**... print(path, 'существует')**

**... if os.path.isfile(path):**

**... print(path, '— это файл')**

**... elif os.path.isdir(path):**

**... print(path, '— это папка')**

**... else:**

**... print(path, '— это ни файл и ни папка')**

**... else:**

**... print(path, 'не существует')**

**3. CSV в Python**

**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.

Пример CSV файла, где в качестве разделителя используется запятая:

**Имя,Профессия,Год рождения**

**Виктор,Токарь, 1995**

**Сергей,Сварщик,1983**

Рассмотрим работу модуля на примере:

**import csv**

**FILENAME = "users.csv"**

**users = [**

**["Tom", 28],**

**["Alice", 23],**

**["Bob", 34]**

**]**

**with open(FILENAME, "w", newline="") as file:**

**writer = csv.writer(file)**

**writer.writerows(users)**

**with open(FILENAME, "a", newline="") as file:**

**user = ["Sam", 31]**

**writer = csv.writer(file)**

**writer.writerow(user)**

В файл записывается двухмерный список - фактически таблица, где каждая строка представляет одного пользователя. А каждый пользователь содержит два поля - имя и возраст. То есть фактически таблица из трех строк и двух столбцов.

При открытии файла на запись в качестве третьего параметра указывается значение newline="" - пустая строка позволяет корректно считывать строки из файла вне зависимости от операционной системы.

Для записи нам надо получить объект writer, который возвращается функцией csv.writer(file). В эту функцию передается открытый файл. А собственно запись производится с помощью метода writer.writerows(users) Этот метод принимает набор строк. В нашем случае это двухмерный список.

Если необходимо добавить одну запись, которая представляет собой одномерный список, например, ["Sam", 31], то в этом случае можно вызвать метод writer.writerow(user)

В итоге после выполнения скрипта в той же папке окажется файл users.csv, который будет иметь следующее содержимое:

**Tom,28**

**Alice,23**

**Bob,34**

**Sam,31**

Для чтения из файла нам наоборот нужно создать объект reader:

**import csv**

**FILENAME = "users.csv"**

**with open(FILENAME, "r", newline="") as file:**

**reader = csv.reader(file)**

**for row in reader:**

**print(row[0], " - ", row[1])**

При получении объекта reader мы можем в цикле перебрать все его строки:

**Tom - 28**

**Alice - 23**

**Bob - 34**

**Sam – 31**

**Работа со словарями**

В примере выше каждая запись или строка представляла собой отдельный список, например, ["Sam", 31]. Но кроме того, модуль csv имеет специальные дополнительные возможности для работы со словарями. В частности, функция csv.DictWriter() возвращает объект writer, который позволяет записывать в файл. А функция csv.DictReader() возвращает объект reader для чтения из файла. Например:

**import csv**

**FILENAME = "users.csv"**

**users = [**

**{"name": "Tom", "age": 28},**

**{"name": "Alice", "age": 23},**

**{"name": "Bob", "age": 34}**

**]**

**with open(FILENAME, "w", newline="") as file:**

**columns = ["name", "age"]**

**writer = csv.DictWriter(file, fieldnames=columns)**

**writer.writeheader()**

**# запись нескольких строк**

**writer.writerows(users)**

**user = {"name" : "Sam", "age": 41}**

**# запись одной строки**

**writer.writerow(user)**

**with open(FILENAME, "r", newline="") as file:**

**reader = csv.DictReader(file)**

**for row in reader:**

**print(row["name"], "-", row["age"])**

Запись строк также производится с помощью методов writerow() и writerows(). Но теперь каждая строка представляет собой отдельный словарь, и кроме того, производится запись и заголовков столбцов с помощью метода writeheader(), а в метод csv.DictWriter в качестве второго параметра передается набор столбцов.

При чтении строк, используя названия столбцов, мы можем обратиться к отдельным значениям внутри строки: row["name"].

Пример чтения CSV файла:

**import csv**

**with open('some.csv', newline='') as f:**

**reader = csv.reader(f)**

**for row in reader:**

**print(row)**

Чтение файла формата passwd:

**import csv**

**with open('passwd', newline='') as f:**

**reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE\_NONE)**

**for row in reader:**

**print(row)**

**Бинарные файлы**

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

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

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

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

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

Посмотрите:

**data = bytearray(100)**

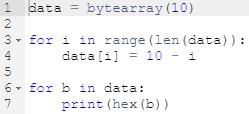
Такой вызов создает объект bytearray, который способен хранить десять байтов.

Примечание: такой конструктор заполняет весь массив нулями.

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

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

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



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

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

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

Давайте проанализируем его:

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

затем мы создаем файл, используя функцию open().

Единственное отличие от предыдущих вариантов состоит в том, что это открытый режим, содержащий флажок b;

метод write() принимает аргумент (bytearray) и отправляет его (целым) в файл;

затем поток закрывается обычным способом.

Метод write() возвращает количество успешно записанных байтов.

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

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

Попробуйте запустить код и проанализировать содержимое вновь созданного выходного файла.

Мы используем его на следующем шаге.

**Как читать байты из потока**

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

метод возвращает количество успешно прочитанных байтов;

метод пытается заполнить все пространство, доступное внутри аргумента. Если в файле больше данных, чем места в аргументе, операция чтения остановится до того, как достигнет конца файла. В противном случае результат метода может указывать на то, что байтовый массив был заполнен только фрагментарно (результат покажет это и то, что часть массива, которую прочитанное содержимое не использовало, остается нетронутой).

Посмотрите на полный код ниже:

**from os import strerror**

**data = bytearray(10)**

**try:**

**bf = open('file.bin', 'rb')**

**bf.readinto(data)**

**bf.close()**

**for b in data:**

**print(hex(b), end=' ')**

**except IOError as e:**

**print("I/O error occurred:", strerr(e.errno))**

Давайте проанализируем его:

сначала мы открываем файл (тот, который вы создали с помощью предыдущего кода) в режиме rb;

затем передаем его содержимое в байтовый массив с именем data размером 10 байтов;

наконец, выводим содержимое байтового массива. Оно отвечает вашим ожиданиям?

Запустите код и проверьте, работает ли он.

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

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

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

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

**from os import strerror**

**try:**

**bf = open('file.bin', 'rb')**

**data = bytearray(bf.read())**

**bf.close()**

**for b in data:**

**print(hex(b), end=' ')**

**except IOError as e:**

**print("I/O error occurred:", strerr(e.errno))**

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

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

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

Этот метод можно использовать, например, так:

**try:**

**bf = open('file.bin', 'rb')**

**data = bytearray(bf.read(5))**

**bf.close()**

**for b in data:**

**print(hex(b), end=' ')**

**except IOError as e:**

**print("I/O error occurred:", strerr(e.errno))**

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

**Диалекты**

Чтобы каждый раз не указывать формат входных и выходных данных, определенные параметры форматирования сгруппированы в диалекты (dialect). При создании объекта reader или writer программист может указать нужный ему диалект, кроме того, некоторые параметры диалекта можно переопределить вручную, также указав их при создании объекта.

Для создания диалекта используется команда:

register\_dialect("имя", delimiter = "\t", ...)

Класс Dialect позволяет определить следующие атрибуты форматирования:

Атрибут Значение

delimiter Устанавливает символ, с помощью которого разделяются элементы в файле. По умолчанию используется запятая.

doublequote Если True, то символ quotechar удваивается, если False, то к символу qutechar добавляется ecsapechar в качестве префикса.

escapechar Строка из одного символа, которая используется для экранирования символа-разделителя.

lineterminator Определяет разделитель для строк, по умолчанию используется «\r\n»

quotechar Определяет символ, который используется для окружения символа-разделителя. По умолчанию используются двойные кавычки, то есть quotechar = ‘ » ‘.

quoting Определяет символ, который используется для экранирования символа разделителя (если не используются кавычки).

skipinitialspace Если установить значение этого параметра в True, то все пробелы после символа-разделителя будут игнорироваться.

strict Если установить в True, то при неправильном вводе CSV будет возбуждаться исключение Error.

Пример использования:

**import csv**

**csv.register\_dialect('my\_dialect', delimiter=':', lineterminator="\r")**

**with open("classmates.csv", mode="w", encoding='utf-8') as w\_file:**

**file\_writer = csv.writer(w\_file, 'my\_dialect')**

**file\_writer.writerow(["Имя", "Класс", "Возраст"])**

**file\_writer.writerow(["Женя", "3", "10"])**

**file\_writer.writerow(["Саша", "5", "12"])**

**file\_writer.writerow(["Маша", "11", "18"])**

В результате получим:

**Имя:Класс:Возраст**

**Женя:3:10**

**Саша:5:12**

**Маша:11:18**