Тема 4. Работа с файлами

1 Работа с файлами. Открытие, закрытие, чтение и запись

В процессе работы Python-программа может извлекать необходимую информацию из файла или, наоборот, записывать в него результат обработки информации.

Перед началом работы с файлом необходимо выполнить процедуру его открытия. Для этого применяется встроенная функция **open()**.

```
f = open("my_file.txt", 'r')
```

Пока нам важны только два параметра функции open() — имя файла или путь до него, режим открытия файла. В этом примере мы открываем файл на чтение.

1.1 Чтение данных из файла

Если мы хотим прочитать информацию из файла, необходимо, как написано выше, открыть файл на чтение.

Примеры:

```
f_obj = open("my_file.txt")
f_obj = open(r"C:\Users\User_1\Desktop\proj\text.txt", "r")
```

В первом примере выполняется открытие файла **my_file.txt** только в режиме чтения. Это режим по умолчанию (можно не указывать). При этом указано только имя файла с расширением. Полный путь к файлу не указан. Интерпретатор Python будет искать файл в директории, где расположен файл-модуль. В нём мы выполняем открытие файла **my_file.txt**. Если файл не будет найден, появится сообщение об ошибке **IOError**.

Во втором примере указывается полный путь файла. Стоит обратить внимание на наличие префикса **r** в пути. Префикс **r** указывает на необходимость обработки строки как исходной, не обращая внимание на наличие специальных символов.

Сравним два примера. Первый:

```
print("C:\Users\User_1\Desktop\proj\text.txt")
```

Результат:

```
SyntaxError: (unicode error) 'unicodeescape' codec can't decode
bytes in position 2-3: truncated \UXXXXXXXX escape
```

С помощью префикса **r** строка, содержащая путь, будет обрабатываться корректно, даже если в ней есть зарезервированные конструкции, например:

```
\t, \s, \n, \p
```

Второй:

```
print(r"C:\Users\User_1\Desktop\proj\text.txt")
```

Результат:

```
C:\Users\User_1\Desktop\proj\text.txt
```

Теперь рассмотрим варианты чтения содержимого файла.

Метод read()

Позволяет прочитать файл целиком. Пример:

```
my_f = open("text.txt", "r")
content = my_f.read()
print(content)
my_f.close()
```

Результат:

```
stroka_1
stroka_2
stroka_3
```

После запуска программы выполняется открытие файла и извлечение его содержимого как строки в переменную **content**. Далее осуществляется печать содержимого файла и закрытие дескриптора. Необходимо всегда выполнять закрытие файла, т. к. другой программе может потребоваться получить доступ к нему.

Метод readline()

Позволяет извлечь очередную строку. Пример:

```
my_f = open("text.txt", "r")
content = my_f.readline()
print(content)
my_f.close()
```

Результат:

```
stroka_1
```

В этом случае извлекается и выводится только первая строка файла. Чтобы извлечь и вывести остальные, важно выполнить инструкцию **content = my_f.readline()** столько раз, сколько строк в файле **text.txt**.

Метод readlines()

Позволяет извлечь и вывести полный список строк файла. Пример:

```
my_f = open("text.txt", "r")
content = my_f.readlines()
print(content)
my_f.close()
```

Результат:

```
['stroka_1\n', 'stroka_2\n', 'stroka_3\n']
```

Чтение файла по частям

Для этого можно использовать построчное извлечение информации из файла с помощью цикла. Пример:

```
my_f = open("text.txt", "r")

for line in my_f:
    print(line)

my_f.close()
```

Результат:

```
stroka_1

stroka_2

stroka_3
```

В этом примере выполняется открытие файла в дескрипторе и построчное извлечение содержимого в цикле **for**. Пример:

```
my_f = open("text.txt", "r")

while True:
    content = my_f.read(1024)
```

```
    print(content)

    if not content:
        break
```

Результат:

```
stroka_1
stroka_2
stroka_3
```

Здесь также использован цикл, в котором содержимое файла извлекается не более килобайта информации или 1024 байтов (символов).

Чтение бинарных (двоичных) файлов

В процессе работы над файлами может возникнуть необходимость открытия на чтение файла в двоичном формате. Для этого важно указать специальный режим доступа к файлу. Пример:

```
my_f = open("text.pdf", "rb")
```

В этом примере файл открывается в режиме **rb** (read-binary).

1.2 Запись данных в файл

Механизм записи информации в файл не отличается по сложности от чтения. Пример:

```
out_f = open("out_file.txt", "w")
out_f.write("String string string")
out_f.close()
```

Мы всего лишь изменили режим работы с файлом на **w** и применили метод **write()** к файловому дескриптору для сохранения некоторого текста в файле. Пример:

```
out_f = open("out_file.txt", "w")
str_list = ['stroka_1\n', 'stroka_2\n', 'stroka_3\n']
out_f.writelines(str_list)
out_f.close()
```

Для файлового дескриптора также допустимо применение метода **writelines()**, принимающего список строк.

2 Менеджеры контекста

До этого мы рассматривали традиционный механизм работы с файлами — с открытием и закрытием. Разработчикам также доступен более удобный инструмент, называемый менеджером контекста. Этот инструмент позволяет упростить процедуры чтения и редактирования содержимого файлов. Для использования менеджера контекста применяется оператор **with**. Такой механизм выполняет автоматическое закрытие файла после завершения работы с ним. Пример:

```
with open("text.txt") as f_obj:
    for line in f_obj:
        print(line)
```

Результат:

```
stroka_1

stroka_2

stroka_3
```

Синтаксис менеджера контекста только на первый взгляд кажется сложным. На деле изменения незначительные:

```
# было
f_obj = open("text.txt")

# стало
with open("text.txt") as f_obj:
```

Находясь в пределах блока с **with**, можно выполнять операции ввода и вывода данных при работе с файлами. После выхода за пределы блока файловый дескриптор закрывает этот блок. Но благодаря такому подходу больше нет необходимости явно запускать закрытие файла — **close()**.

3 Выявление ошибок при работе с файлами

Работа с файлами, как и с другими объектами, может сопровождаться возникновением ошибок. Например, операция закрытия файла, притом что некоторый сторонний процесс может работать с этим файлом. Результат — возникновение ошибки **IOError**.

Пример:

```
try:
    f_obj = open("text.txt")
    for line in f_obj:
        print(line)
except IOError:
    print("Произошла ошибка ввода-вывода!")
finally:
    f_obj.close()
```

В представленном примере код работы с файлом и его содержимым помещается в блок **try/except**. При возникновении ошибки появится сообщение на экране. Необходимо также предусмотреть закрытие файла через ветку **finally**. Эту же задачу можно решить немного по-другому. Пример:

```
try:
    with open("text.txt") as f_obj:
        for line in f_obj:
            print(line)
except IOError:
    print("Произошла ошибка ввода-вывода!")
```

Такой подход позволяет отказаться от ветви с **finally.** За закрытие файла отвечает менеджер контекста.

4 Режимы доступа к файлу

Ранее мы рассмотрели только два режима работы с файлами: чтение и запись. Но существуют дополнительные специальные режимы чтения-записи.

Режим	Описание
r	Открыть файл на чтение (режим по умолчанию)
w	Открыть на запись. При этом удалить содержимое файла. Если файла нет, создать новый.
x	Открыть файл на запись, если его нет. Если файл есть, он не будет создан.
a	Открыть файл на дозапись. Добавить информацию в конец файла.
b	Открыть файл в двоичном формате.
t	Открыть файл в текстовом формате (режим по умолчанию)
+	Открыть файл на чтение и запись

Примеры работы со стандартными режимами чтения-записи мы уже рассмотрели. Теперь поработаем с другими режимами.

4.1 Режим x

Режим **x** позволяет открыть файл на запись, если его нет. Если файл есть, он не будет создан. Во втором случае инструкция выполнится успешно только в том случае, если файла с указанным именем нет. Так как в этом примере файл создан ранее, генерируется исключение. Пример:

```
f_1 = open("my_file.txt", 'w')
f_2 = open("my_file.txt", 'x')
```

Результат:

```
f_2 = open("my_file.txt", 'x')
FileExistsError: [Errno 17] File exists: 'my_file.txt'
```

4.2 Режим a

Позволяет открыть файл на дозапись. Добавить информацию в конец файла. Если файл с указанным именем отсутствует, будет создан новый. Иначе файл откроется на запись, и добавится указанная строка. Пример:

```
f_obj = open("new_f.txt", 'a')
f_obj.write("My string")
f_obj.close()
```

4.3 Режим b

Позволяет открыть файл в двоичном формате. В этом примере открывается файл на запись в двоичном режиме. Далее выполняется запись байтового представления некоторого символа и закрывается файл. Пример:

```
f_obj = open("data.bin", "wb")
my_var = "if5s"
f_obj.write(my_var)
f_obj.close()
```

4.4 Режим +

Позволяет открыть файл на чтение и запись. Пример:

```
with open("file.dat", "w+") as f_obj:
```

```
    f_obj.write("another string")
    f_obj.seek(0)
    content = f_obj.read()
    print(content)
```

Результат:

```
another string
```

Режим + используется в комбинации с одним из представленных в таблице режимов. Он позволяет соединить возможности нескольких режимов. В приведённом примере режим **w+** даёт осуществить запись объекта в файл и чтение содержимого файла. Использование только режима чтения даст ошибку:

```
f_obj.write("another string")
io.UnsupportedOperation: not writable
```

5 Параметры файлового объекта

После открытия файла у разработчика появляется возможность получения сведений о соответствующем файловом объекте. Эти сведения приведены в таблице:

Атрибут	Описание
`file.closed`	Возвращает значение True, если файл закрыт
`file.mode`	Возвращает режим доступа, по которому был открыт файл
`file.name`	Возвращает имя файла

Пример:

```
f_obj = open("new_f.txt", "w")
print("Файл. Имя: ", f_obj.name)
print("Файл. Закрыт: ", f_obj.closed)
print("Файл. Режим: ", f_obj.mode)
```

Результат:

```
Файл. Имя:  new_f.txt
Файл. Закрыт:  False
Файл. Режим:  w
```

6 Определение позиции указателя в файле

После вызова метода **read()** для файлового объекта при повторном вызове **read()** появится пустая строка. Пример:

```
f_obj = open("new_f.txt", "r")
content = f_obj.read()
print(content)
content = f_obj.read()
print(content)
f_obj.close()
```

Результат:

```
My string
```

Описанная ситуация происходит из-за того, что после первого прочтения содержимого файла указатель перемещается в конец файла. Для получения информации о позиции указателя можно воспользоваться методом **tell()**. Пример:

```
f_obj = open("new_f.txt")
f_obj.read(10)
print("Текущая позиция:", f_obj.tell())
f_obj.close()
```

Результат:

```
Текущая позиция: 9
```

Метод **tell()** определяет, в скольких байтах от начала файла находится указатель на текущий момент. Существует ещё один метод — **seek()**, позволяющий выполнить переход на нужную позицию. Синтаксис метода:

```
file_obj.seek(offset, [from])
```

Параметр **offset** определяет число байтов, на которое необходимо перейти. Параметр **from** — опциональный. Он соответствует позиции, с которой начинается перемещение: 0 — начало файла, 1 — текущая позиция, 2 — конец файла. Пример:

```
f_obj = open("new_f.txt")
print(f_obj.read(3))
print("Мы находимся на позиции: ", f_obj.tell())
```

```
# Перемещаемся в начало
f_obj.seek(0)
print(f_obj.read(10))
f_obj.close()
```

Результат:

```
My
Мы находимся на позиции:  3
My string
```

7 Print в файл

Мы уже работали с функцией **print()**, отвечающей за вывод объектов на экран (стандартное устройство вывода). У неё есть ещё одна интересная возможность — отправка объектов текстовым потоком в файл. Пример:

```
with open("python.txt", "w") as f_obj:
    print("Необычная работа функции print", file=f_obj)
```

Для указания файла, в который выполняется вывод, используется параметр **file**. Приведённый код работает стандартно — открытие файла на запись, передача объекта в файл и закрытие. Благодаря использованию менеджера контекста, не нужно использовать функцию закрытия файла.

8 Модуль os

Предоставляет широкий спектр функций для работы с файлами. Рассмотрим некоторые из них.

os.remove() Отвечает за удаление указанного файла. Пример:

```
import os

os.remove("my_file.txt")
```

В приведённом коде есть инструкция удаления простого текстового файла из рабочего каталога. Это каталог, в котором расположен скрипт с данным кодом. Если файл не будет найден, вы увидите сообщение об ошибке.

os.rename() Отвечает за переименование файла. Пример:

```
import os

os.rename("test.txt", "pytest.txt")
```

В этом коде файлу с именем **test.txt** присваивается новое имя **pytest.txt**. При выполнении этой операции может возникнуть ошибка, связанная с попыткой переименования несуществующего файла или с отсутствием прав на эту операцию.

os.listdir() Отвечает за получение списка папок и файлов для определённой директории. Пример:

```
import os

content = os.listdir(path=".")
print(content)
```

В этом примере функция **listdir()** отвечает за отображение содержимого текущей директории, из которой мы запускаем скрипт. Для этого случая возможен и другой вариант:

```
import os

content = os.listdir()
print(content)
```

Результаты будут идентичными. Аналогично можно получить содержимое каталога, расположенного на один уровень выше относительно текущего:

```
import os

content = os.listdir(path="..")
print(content)
```

os.path Это подмодуль модуля **os**. Он предоставляет разработчику некоторые полезные функции для выполнения операций с путями.

os.path.basename(). Возвращает название файла пути. В этом примере символ **r** отключает экранирование, то есть специальные символы в имени пути не учитываются. Обратный слэш в этом случае используется только как разделитель. Пример:

```
import os

print(os.path.basename(r"C:\Users\Администратор\settings.py"))
```

Результат:

```
settings.py
```

os.path.dirname(). Возвращает часть каталога пути. Пример:

```
import os

print(os.path.dirname(r"C:\Users\Администратор\settings.py"))
```

Результат:

```
 C:\Users\Администратор
```

os.path.exists(). Проверяет, присутствует ли указанный файл. Пример:

```
import os

print(os.path.exists(r"C:\Users\Администратор\settings.py"))
```

Результат:

```
True
```

os.path.isdir(), **os.path.isfile()**. Проверяет, является ли объект папкой или файлом. Пример:

```
print(os.path.isdir(r"C:\Users\Администратор\settings.py"))
print(os.path.isfile(r"C:\Users\Администратор\settings.py"))
```

Результат:

```
False
True
```

os.path.join(). Позволяет объединить несколько путей. Пример:

```
import os

print(os.path.join(r"C:\Users\Администратор", "settings.py"))
```

Результат:

```
C:\Users\Администратор\settings.py
```

os.path.split(). Разделяет путь на кортеж, содержащий и путь до каталога, и имя файла. Пример:

```
import os

print(os.path.split(r"C:\Users\Администратор\settings.py"))
```

Результат:

```
('C:\\Users\\Администратор', 'settings.py')
```

Полный список функций, предоставляемых модулем **os**, доступен по ссылке. Информацию о других функциях подмодуля **os.path** можно узнать по ссылке.

9 Модуль json

JSON (Java Script Object Notation) — стандарт обмена информацией. Он может, например, применяться при получении данных через API и необходимости их хранения в документной базе данных. Работать с данными в **JSON**-формате можно средствами языка Python. **JSON** — универсальная нотация, и она напоминает Python-словарь.

Пример **JSON**-структуры:

```
{
    "worker": "Jon Smith",
    "skills": ["programming", "design", "engineering"],
    "age": 40,
    "workplaces": [
        {
            "first": "IBM",
            "time": "2010-2014"
        },
        {
            "second": "Apple",
            "time": "2014-2018"
        }
    ]
}
```

9.1 JSON и Python

Для работы с JSON-форматом в Python применяется модуль **json**, который необходимо импортировать:

```
import json
```

Процесс преобразования данных к JSON-формату называется сериализацией. Под этим термином подразумевается трансформация данных в байты для хранения или передачи по сети. Обратный процесс называется десериализацией. Аналогия — запись данных на диски и чтение данных из памяти.

9.2 Сериализация

Применяются методы **dump()** и **dumps()**. Стандартные Python-объекты трансформируются в Python следующим образом:

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

Рассмотрим пример сериализации. Создадим простейший Python-объект (словарь):

```
data = {
    "income": {
        "salary": 50000,
        "bonus": 20000
    }
}
```

Через контекстный мессенджер Python создадим файл **my_file.json** и откроем его в режиме записи:

```
with open("my_file.json", "w") as write_f:
    json.dump(data, write_f)
```

Обратите внимание, что функция **dump()** принимает два позиционных параметра: объект данных для сериализации и файловый объект, в который

необходимо записать соответствующие байты. После запуска представленного кода будет создан указанный JSON-файл со следующим содержимым:

```
{"income": {"salary": 50000, "bonus": 20000}}
```

Если в программе требуется продолжить работу с сериализованными данными, то с ними можно работать, как со строкой:

```
json_str = json.dumps(data)
print(json_str)
print(type(json_str))
```

Результат:

```
{"income": {"salary": 50000, "bonus": 20000}}
<class 'str'>
```

При этом файловый объект остаётся пустым, так как по факту мы не выполняем записи на диск.

9.3 Десериализация

Методы **load()** и **loads()** обеспечивают трансформацию данных JSON-формата в Python-объекты.

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

Таблица десериализации — неабсолютная инверсия таблицы сериализации. Это означает, что при преобразовании объекта в JSON-формат и последующем декодировании получить полностью идентичный объект не получится.

Рассмотрим пример десериализации. Представим, что у нас есть данные на диске, которыми необходимо управлять. Как и в предыдущем примере, воспользуемся контекстными менеджером, но откроем JSON-файл в режиме чтения.

```
with open("my_file.json") as read_f:
    data = json.load(read_f)

print(data)
print(type(data))
```

Результат:

```
{'income': {'salary': 50000, 'bonus': 20000}}
<class 'dict'>
```

В этом примере JSON-объект типа **object** преобразовался в Python-объект **dict** (словарь). Далее в программе с полученным словарём можно осуществлять дальнейшие операции.

Данные в JSON-формате могут быть получены разными способами и быть представлены, например, в виде строки. Строку можно десериализировать с помощью функции **loads()**.

```
json_str = """{"income": {"salary": 50000, "bonus": 20000}}"""
data = json.loads(json_str)

print(data)
print(type(data))
```

Результат:

```
{'income': {'salary': 50000, 'bonus': 20000}}
<class 'dict'>
```

10 Модуль shutil

Предоставляет разработчику возможность работы с функциями высокого уровня для выполнения операций с файлами и папками: копирование, перемещение, удаление.

Копирование содержимого одного файлового объекта (**f_obj_1**) в другой (**f_obj_2**).

```
shutil.copyfileobj(f_obj_1, f_obj_2)
```

Копирование содержимого (но не метаданных) одного файла (**f_1**) в другой (**f_2**).

```
shutil.copyfile(f_1, f_2)
```

Копирование содержимого файла **my_f** в файл или папку **my_target**. Если копирование выполняется в директорию, файл копируется с исходным именем (**my_f**).

```
shutil.copy(my_f, my_target)
```

Рекурсивное копирование дерева директорий **my_tree** в папку **my_target**.

```
shutil.copy(my_tree, my_target)
```

Удаление текущей директории и всех поддиректорий.

```
shutil.rmtree(path)
```

Рекурсивное перемещение файла или директории (**my_obj**) в нужную директорию (**my_target**).

```
shutil.move(my_obj, my_target)
```

11 Модуль sys

Предоставляет доступ к переменным и функциям, взаимодействующим с Python-интерпретатором. Ниже находится описание некоторых возможностей этого модуля.

sys.argv. Параметр **argv** позволяет получить список аргументов, которые связаны со скриптом при его запуске из командной строки. С этим параметром мы уже работали на четвёртом уроке. Первым параметром списка будет имя самого файла-модуля (скрипта). Если скрипт запускается с параметрами, список будет содержать дополнительный набор этих параметров.

sys.executable. Параметр позволяет получить полный путь к Python-интерпретатору. С помощью этого параметра мы можем узнать, где у нас установлен Python. Пример:

```
import sys

print(sys.executable)
```

Результат:

```
C:\Python37\python.exe
```

sys.exit. Представляет собой функцию, обеспечивающую выход из Python-программы. Принимает необязательный параметр (целое число), определяющий статус выхода. Сигнал нормально завершения программы (значение по умолчанию) — это 0. Значения, отличные от 0, интерпретируются в качестве ошибок. Пример:

```
import sys

sys.exit(0)
```

sys.path. Функция **path()** возвращает список строк-путей поиска для модулей. Именно по этим путям (локациям) Python будет осуществлять поиск модулей. Эта функция может пригодиться при отладке программы для поиска причины, по которой не удаётся импортировать модуль. Пример:

```
import sys

print(sys.path)
```

Результат:

```
['C:\\Users\\Администратор\\Desktop', 'C:\\Users',
'C:\\Python37\\python37.zip', 'C:\\Python37\\DLLs',
'C:\\Python37\\lib', 'C:\\Python37', 'C:\\Python37\\lib\\site-
packages']
```

sys.platform. Параметр, соответствующий идентификатору платформы. Может, например, использоваться для запуска различных частей кода в зависимости от платформы. Из приведённого ниже примера видно, что Python работает в Windows. Пример:

```
import sys

print(sys.platform)
```

Результат:

```
win32
```

sys.stdin / stdout / stderr. Аналоги файловых объектов. Соответствуют потокам ввода, вывода и ошибок интерпретатора, соответственно.

stdin – применяется для любого интерактивного ввода. Сюда входят и вызовы **input()**.

stdout – применяется для вывода операторов **print()**, а также **input()**-запросов.

stderr – собственные запросы интерпретатора и его сообщения об ошибках.

Памятка

Режимы чтения-записи.

Режим	Описание
`r`	Открыть файл на чтение (режим по умолчанию)
`w`	Открыть на запись. При этом удалить содержимое файла. Если файла нет, создать новый.
`x`	Открыть файл на запись, если его нет. Если файл есть, он не будет создан.
`a`	Открыть файл на дозапись. Добавить информацию в конец файла.
`b`	Открыть файл в двоичном формате.
`t`	Открыть файл в текстовом формате (режим по умолчанию)
`+`	Открыть файл на чтение и запись

Параметры файлового объекта

Атрибут	Описание
`file.closed`	Возвращает значение True, если файл закрыт
`file.mode`	Возвращает режим доступа, по которому был открыт файл
`file.name`	Возвращает имя файла

Функции для выполнения операций с путями подмодуля **os.path**

Функция	Описание
`os.path.basename()`	Возвращает название файла пути.
`os.path.dirname()`	Возвращает часть каталога пути.
`os.path.exists()`	Проверяет, присутствует ли указанный файл.
`os.path.isdir()`	Проверяет, является ли объект папкой.
`os.path.isfile()`	Проверяет, является ли объект файлом.
`os.path.join().`	Позволяет объединить несколько путей.
`os.path.split()`	Разделяет путь на кортеж, содержащий и путь до каталога, и имя файла.

Описание некоторых возможностей этого модуля **sys**

Функция	Описание
`sys.argv`	Позволяет получить список аргументов, которые связаны со скриптом при его запуске из командной строки
`sys.executable`	Параметр позволяет получить полный путь к Python-интерпретатору.
`sys.exit`	Представляет собой функцию, обеспечивающую выход из Python-программы.
`sys.path`	Возвращает список строк-путей поиска для модулей
`sys.platform`	Параметр, соответствующий идентификатору платформы.
`sys.stdin`	Аналог файловых объектов. Соответствуют потоку ввода
`sys.stdout`	Аналог файловых объектов. Соответствуют потоку вывода
`sys.stderr`	Аналог файловых объектов. Соответствует потоку ошибок интерпретатора