```
fileobj = open(filename, mode)
```

Кратко поясню фрагменты этого вызова:

- fileobj — это объект файла, возвращаемый функцией open();
- filename — это строка, представляющая собой имя файла;
- mode — это строка, указывающая на тип файла и действия, которые вы хотите над ним произвести.

Первая буква строки mode указывает на операцию:

- r означает чтение;
- w означает запись. Если файла не существует, он будет создан. Если файл существует, он будет перезаписан;
- x означает запись, но только если файла еще не существует;
- a означает добавление данных в конец файла, если он существует.

Вторая буква строки mode указывает на тип файла:

- t (или ничего) означает, что файл текстовый;
- b означает, что файл бинарный.

После открытия файла вы вызываете функции для чтения или записи данных, они будут показаны в следующих примерах.

После чего нужно закрыть файл.

Оборудование и материалы.

Персональный компьютер, среда разработки Python.

Указания по технике безопасности:

Соответствуют технике безопасности по работе с компьютерной техникой.

Задания

Создадим файл, содержащий одну строку, в одной программе и считаем его в другой.

Запись в текстовый файл с помощью функции write()

По какой-то причине существует не так уж много лимериков о специальной теории относительности. В качестве источника данных придется использовать всего один:

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
```

Следующий код записывает это стихотворение в файл 'relativity' с помощью всего одного вызова:

```
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
150
>>> fout.close()
```

Функция write() возвращает число записанных байтов. Она не добавляет никаких пробелов или символов новой строки, как это делает функция print(). С помощью функции print() вы также можете записывать данные в текстовый файл:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

Отсюда возникает вопрос: какую функцию использовать — write() или print()?

По умолчанию функция print() добавляет пробел после каждого аргумента и символ новой строки в конце. В предыдущем примере она добавила символ новой строки в файл relativity. Для того чтобы функция print() работала как функция write(), передайте ей два следующих аргумента:

- Ґ sep (разделитель, по умолчанию это пробел, ' ');
- Ґ end (символ конца файла, по умолчанию это символ новой строки, '\n').

Функция print() использует значения по умолчанию, если только вы не передадите ей что-то еще. Мы передадим ей пустые строки, чтобы подавить все лишние детали, обычно добавляемые функцией print():

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

Если исходная строка большая, вы можете записывать в файл ее фрагменты до тех пор, пока не запишете ее всю:

```
>>> fout = open('relativity', 'wt')
>>> size = len(poem)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(poem[offset:offset+chunk])
...     offset += chunk
...
100
50
>>> fout.close()
```

Этот код записал 100 символов за первую попытку и последние 50 символов — за следующую.

Если файл relativity нам очень дорог, проверим, спасет ли режим x от его перезаписывания:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

Вы можете использовать этот код вместе с обработчиком исключений:

```
>>> try:
...     fout = open('relativity', 'xt')]
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close one.')
...
relativity already exists!. That was a close one.
```

Считываем данные из текстового файла с помощью функций read(), readline() и readlines()

Вы можете вызвать функцию read() без аргументов, чтобы проглотить весь файл целиком, как показано в следующем примере. Будьте осторожны, делая это с крупными файлами, файл размером 1 Гбайт потребит 1 Гбайт памяти:

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

Вы можете указать максимальное количество символов, которое функция read() вернет за один вызов. Давайте считывать по 100 символов за раз и присоединять каждый фрагмент к строке poem, чтобы воссоздать оригинал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

После того как вы считали весь файл, дальнейшие вызовы функции read() будут возвращать пустую строку (' '), которая будет оценена как False в проверке if not fragment. Это позволит выйти из цикла while True.

Вы также можете считывать файл по одной строке за раз с помощью функции readline(). В следующем примере мы будем присоединять каждую строку к строке poem, чтобы воссоздать оригинал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
...     line = fin.readline()
...     if not line:
...         break
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

Для текстового файла даже пустая строка имеет длину, равную 1 (символ новой строки), такая строка будет считаться True. Когда весь файл будет считан, функция readline() (как и функция read()) возвратит пустую строку, которая будет считаться False.

Самый простой способ считать текстовый файл — использовать итератор. Он будет возвращать по одной строке за раз. Этот пример похож на предыдущий, но кода в нем меньше:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

Во всех предыдущих примерах в результате получалась одна строка poem. Функция readline() считывает по одной строке за раз и возвращает список этих строк:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
...     print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

Мы указали функции print() не добавлять автоматически символы новой строки, поскольку первые четыре строки сами их имеют. В последней строке этого символа не было, что заставило интерактивное приглашение появиться сразу после последней строки.

Записываем данные в бинарный файл с помощью функции write()

Если вы включите символ 'b' в строку режима, файл будет открыт в бинарном режиме. В этом случае вы вместо чтения и записи строк будете работать с байтами.

У нас под рукой нет бинарного стихотворения, поэтому мы просто сгенерируем 256 байтовых значений от 0 до 255:

```
>>> bdata = bytes(range(0, 256))
>>> len(bdata)
256
```

Откроем файл для записи в бинарном режиме и запишем все данные сразу:

```
>>> fout = open('bfile', 'wb')
>>> fout.write(bdata)
256
>>> fout.close()
```

И вновь функция write() возвращает количество записанных байтов.Как и в случае с текстом, вы можете записывать бинарные данные фрагментами:

```
>>> fout = open('bfile', 'wb')
>>> size = len(bdata)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(bdata[offset:offset+chunk])
...     offset += chunk
...
100
100
56
>>> fout.close()
```

Читаем бинарные файлы с помощью функции read()

Это просто: все, что вам нужно, — открыть файл в режиме 'rb':

```
>>> fin = open('bfile', 'rb')
>>> bdata = fin.read()
>>> len(bdata)
256
>>> fin.close()
```

Закрываем файлы автоматически с помощью ключевого слова with

Если вы забудете закрыть за собой файл, его закроет Python после того, как будет удалена последняя ссылка на него. Это означает, что, если вы откроете файл и не закроете его явно, он будет закрыт автоматически по завершении функции. Но вы можете открыть файл внутри длинной функции или даже основного раздела программы. Файл должен быть закрыт, чтобы все оставшиеся операции записи были завершены.

У Python имеются менеджеры контекста для очистки объектов вроде открытых файлов. Вы можете использовать конструкцию *with выражение as переменная*:

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
...
```

Вот и все. После того как блок кода, расположенный под менеджером контекста (в этом случае одна строка), завершится (или нормально, или путем генерации исключения), файл будет закрыт автоматически.

Меняем позицию с помощью функции seek()

По мере чтения и записи Python отслеживает ваше местоположение в файле. Функция tell() возвращает ваше текущее смещение от начала файла в байтах. Функция seek() позволяет вам перейти к другому смещению в файле. Это значит, что вам не обязательно читать каждый байт файла, чтобы добраться до последнего, — вы можете использовать функцию seek(), чтобы сместиться к последнему байту и считать его.

Для примера воспользуемся 256-байтным бинарным файлом 'bfile', который мы создали ранее:

```
>>> fin = open('bfile', 'rb')
>>> fin.tell()
0
```

Используем функцию seek(), чтобы перейти к предпоследнему байту файла:

```
>>> fin.seek(255)
255
```

Считаем все данные от текущей позиции до конца файла:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

Функция seek() также возвращает текущее смещение.

Вы также можете вызвать функцию seek(), передав ей второй аргумент: seek(offset, origin):

- если значение origin равно 0 (по умолчанию), сместиться на offset байт с начала файла;
- если значение origin равно 1, сместиться на offset байт с текущей позиции;
- если значение origin равно 2, сместиться на offset байт с конца файла.

Эти значения также определены в стандартном модуле os:

```
>>> import os
>>> os.SEEK_SET
0
>>> os.SEEK_CUR
1
>>> os.SEEK_END
2
```

Благодаря этому мы можем считать последний байт разными способами:

```
>>> fin = open('bfile', 'rb')
```

Один байт перед концом файла:

```
>>> fin.seek(-1, 2)
255
>>> fin.tell()
255
```

Считать данные до конца файла:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

Рассмотрим случай, когда мы вызываем функцию seek(), чтобы сместиться с текущей позиции:

```
>>> fin = open('bfile', 'rb')
```

Следующий пример переносит позицию за 2 байта до конца файла:

```
>>> fin.seek(254, 0)
254
>>> fin.tell()
254
```

Теперь перейдем вперед на 1 байт:

```
>>> fin.seek(1, 1)
255
>>> fin.tell()
255
```

Наконец, считаем все данные до конца файла:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

Эти функции наиболее полезны при работе с бинарными файлами. Вы можете использовать их и для работы с текстовыми файлами, но если файл содержит в себе не только символы формата ASCII (занимающие по одному байту в памяти), вам будет трудно определить смещение. Оно будет зависеть от кодировки текста, самая популярная кодировка (UTF-8) использует разное количество байтов для разных символов.

Контрольные вопросы

1. Для чего используется ввод и вывод информации в файл, приведите примеры?
2. Открытие файла, аргументы функции открытия.
3. Запись в текстовый файл с помощью функции write().
4. Считывание данных из текстового файла с помощью функций read(), readline() и readlines().
5. Записываем данные в бинарный файл с помощью функции write().
6. Чтение бинарных файлов с помощью функции read().
7. Закрытие файлов автоматически с помощью ключевого слова with.
8. Изменение позиции с помощью функции seek().

Список литературы, рекомендуемый к использованию по данной теме:

1. Орлов, С. А. Программная инженерия. Технологии разработки программного обеспечения : учебник / С.А. Орлов. - 5-е изд., обновл. и доп. - СПб. : Питер, 2017. - 640 с.
2. Гагарина, Л. Г. Современные проблемы информатики и вычислительной техники : [учеб. пособие] / Л.Г. Гагарина, А.А. Петров. - М. : Форум, 2016. - 368 с.
3. Шкляр, М. Ф. Основы научных исследований : учеб. пособие / М.Ф. Шкляр. - 6-е изд. - М. : Дашков и Ко, 2016. - 208 с.
4. Михеева, Е. В. Информационные технологии в профессиональной деятельности : учеб. пособие / Е.В. Михеева. - 14-е изд., стер. - М. : Академия, 2016. - 384 с.
5. Любанович Билл. Простой Python. Современный стиль программирования. — СПб.: Питер, 2016. — 480 с.
6. Любанович Билл. Простой Python. Современный стиль программирования. — СПб.: Питер, 2016. — 480 с.

Лабораторная работа 14. Моделирование с использованием модуля turtle.

Цель работы:

Изучить Назначение модуля turtle. Метод mainloop(), вывод окна. Команды перемещения пера модуля turtle. Настройка параметров пера. Черчение объектов в модуле turtle. Рисование в окне модуля turtle. Создание графиков функций.

Компетенции:

Код	Формулировка: