Структуры

Представим себе базу данных библиотеки, в которой хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляром и т.д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но информация о книгах разнородна, она содержит целые числа и символьные строки разной длины. Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т.д.), так чтобы **`i`**-ый элемент каждого массива относился к книге с номером **`i`**. Но такой подход оказывается слишком неудобен и ненадежен. Например, при сортировке нужно переставлять элементы всех массивов (отдельно!) и можно легко ошибиться и нарушить связь данных.

Возникает естественная идея – объединить все данные, относящиеся к книге, в единый блок памяти, который в программировании называется структурой.

Структура – это тип данных, который может включать в себя несколько полей – элементов разных типов (в том числе и другие структуры).

Классы

В Python для работы со структурами используют специальные типы данных – *классы*. В программе можно вводить свои классы (новые типы данных). Введём новый класс **`TBook`** – структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только

- фамилию автора (символьная строка);
- название книги (символьная строка);
- имеющееся в библиотеке количество экземпляров (целое число).

Класс можно объявить так:

```
class TBook:
  pass
```

Объявление начинается с ключевого слова **`class`** (англ. класс) и располагаются выше блока объявления переменных. Имя нового типа – **`TBook`** – это удобное сокращение от английских слов ***Type Book*** (*тип книга*), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования.

Слово **`pass`** (англ. пропустить) стоит во второй строке только потому, что оставить строку совсем пустой нельзя – будет ошибка. В данном случае пока мы не определяем какие-то новые характеристики для объектов этого класса, просто говорим, что есть такой класс.

В отличие от других языков программирования (C, C++, Паскаль) при объявлении класса необязательно сразу перечислять все *поля* (данные) структуры, они могут добавляться по ходу выполнения программы. Такой подход имеет свои преимущества и недостатки. С одной стороны, программисту удобнее работать, больше свободы. С другой стороны, повышается вероятность случайных ошибок. Например, при опечатке в названии поля может быть создано новое поле с неверным именем, и сообщения об ошибке не появится.

Теперь уже можно создать объект этого класса:

```
B = TBook()
```

или даже массив (список) из таких объектов:

```
Books = []
for i in range(100):
  Books.append ( TBook() )
```

Обратите внимание, что при создании массива нельзя было написать

```
Books = [TBook()]*100  # ошибочное создание массива
```

Дело в том, что список **Books** содержит указатели (адреса) объектов типа **Book**, и в последнем варианте фактически создаётся один объект, адрес которого записывается во все 100 элементов массива. Поэтому изменение одного элемента массива «синхронно» изменит и все остальные элементы.

Для того, чтобы работать не со всей структурой, а с отдельными полями, используют так называемую *точечную запись*, разделяя точкой имя структуры и имя поля. Например, **B.author** обозначает «поле **author** структуры B», а **Books[5].count** – «поле **count** элемента массива **Books[5]**».

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить их с клавиатуры (или из файла):

```
B.author = input()
B.title = input()
B.count = int( input() )
```

присваивать новые значения:

```
B.author = "Пушкин А.С."
B.title = "Полтава"
B.count = 1
```

использовать при обработке данных:

```
fam = B.author.split()[0]       # только фамилия
print ( fam )
B.count -= 1                    # одну книгу взяли
if B.count == 0:
  print ( "Этих книг больше нет!" )
```

и выводить на экран:

```
print ( B.author, B.title + ".", B.count, "шт." )
```

Работа с файлами

В программах, которые работают с данными на диске, бывает нужно читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля одной структуры в одну строчку и разделяя их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет хранить данные в файлах во внутреннем формате, то есть так, как они представлены в памяти компьютера во время работы программы. Для этого в Python используется стандартный модуль **pickle**, который подключается с помощью команды **import**:

```
import pickle
```

Запись структуры в файл выполняется с помощью процедуры dump:

```
B = TBook()
F = open ( "books.dat", "wb" )
B.author = "Тургенев И.С. "
B.title = "Муму"
B.count = 2
pickle.dump ( B, F );
F.close()
```

Обратите внимание, что файл открывается в режиме «**wb**»; первая буква «**w**», от англ. *write* – писать, нам уже знакома, а вторая – «b» – сокращение от англ. *binary* – двоичный, она говорит о том, что данные записываются в файл в двоичном (внутреннем) формате. С помощью цикла можно записать в файл массив структур:

```
for B in Books:
  pickle.dump ( B, F )
```

а можно даже обойтись без цикла:

```
pickle.dump ( Books, F );
```

При чтении этих данных нужно использовать тот же способ, что и при записи: если структуры записывались по одной, читать их тоже нужно по одной, а если записывался весь массив за один раз, так же его нужно и читать.

Прочитать из файла одну структуру и вывести её поля на экран можно следующим образом:

```
F = open ( "books.dat", "rb" )
B = pickle.load ( F )
print ( B.author, B.title, B.count, sep = ", " )
F.close()
```

Если массив (список) структур записывался в файл за один раз, его легко прочитать, тоже за один вызов функции **load**:

```
Books = pickle.load ( F )
```

Если структуры записывались в файл по одной и их количество известно, при чтении этих данных в массив можно применить цикл с переменной:

```
for i in range(N):
  Books[i] = pickle.load ( F )
```

Если же число структур неизвестно, нужно выполнять чтение до тех пор, пока файл не закончится, то есть при очередном чтении не произойдет ошибка:

```
Books = []
while True:
  try:
    Books.append ( pickle.load ( F ) )
  except:
    break
```

Мы сначала создаём пустой список **Books**, а затем в цикле читаем из файла очередную структуру и добавляем её в конец списка с помощью метода **append**.

Обработка ошибки выполнена с помощью *исключений*. Исключение – это ошибочная (аварийная) ситуация, в данном случае – неудачное чтение структуры из файла. «Опасные» операторы, которые могут вызвать ошибку, записываются в виде блока (со сдвигом) после слова **try**. После этого в блоке, начинаемся с ключевого слова **except**, записывают команды, которые нужно выполнить в случае ошибки (в данном случае – выйти из цикла).

Сортировка

Для сортировки массива структур применяют те же методы, что и для массива простых переменных. Структуры обычно сортируют по возрастанию или убыванию одного из полей, которое называют ключевым полем или ключом, хотя можно, конечно, использовать и сложные условия, зависящие от нескольких полей (составной ключ).

Отсортируем массив **Books** (типа **TBook**) по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле **author**. Предположим, что фамилия состоит из одного слова, а за ней через пробел следуют инициалы. Тогда сортировка методом пузырька выглядит так:

```
N = len(Books)
for i in range(0,N-1):
  for j in range(N-2,i-1,-1):
    if Books[j].author > Books[j+1].author:
      Books[j], Books[j+1] = Books[j+1], Books[j]
```

Как вы знаете из курса 10 класса, при сравнении двух символьных строк они рассматриваются посимвольно до тех пор, пока не будут найдены первые отличающиеся символы. Далее сравниваются коды этих символов по кодовой таблице. Так как код пробела меньше, чем код любой русской (и латинской) буквы, строка с фамилией «Волк» окажется выше в отсортированном списке, чем строка с более длинной фамилией «Волков», даже с учетом того, что после фамилии есть инициалы. Если фамилии одинаковы, сортировка происходит по первой букве инициалов, затем – по второй букве.

Отметим, что при такой сортировке данные, входящие в структуры не перемещаются. Дело в том, что в массиве **Books** хранятся указатели (адреса) отдельных элементов, поэтому при сортировке переставляются именно эти указатели. Это очень важно, если структуры имеют большой размер или их по каким-то причинам нельзя перемещать в памяти.

Теперь покажем сортировку в стиле Python. Попытка просто вызвать метод **sort** для списка **Books** приводит к ошибке, потому что транслятор не знает, как сравнить два объекта класса **TBook**. Здесь нужно ему помочь – указать, какое из полей играет роль ключа. Для этого используем именованный параметр **key** метода **sort**. Например, можно указать в качестве ключа функцию, которая выделяет поле **author** из структуры:

```
def getAuthor ( B ):
  return B.author
Books.sort ( key = getAuthor )
```

Более красивый способ – использовать так называемую «лямбда-функцию», то есть функцию без имени:

```
Books.sort ( key = lambda x: x.author )
```

Здесь в качестве ключа указана функция, которая из переданного ей параметра **x** выделяет поле **author**, то есть делает то же самое, что и показанная выше функция **getAuthor**.

Если не нужно изменять сам список **Books**, можно использовать не метод **sort**, а функцию **sorted**, например, так:

```
for B in sorted ( Books, key = lambda x: x.author ):
  print ( B.author, B.title + ".", B.count, "шт." )
```