Программирование на языке Python для сбора и анализа данных

Словари

Рассмотрим такую задачу: у нас есть информация об оценках студентов по некоторому предмету и мы хотим иметь возможность с этой информацией работать — например, по имени студента определить, какую оценку он получил. Мы могли бы пытаться решить эту задачу, создав два списка — один с именами студентов, а другой с оценками:

In [ ]:

```
students = ["Вася", "Коля", "Петя", "Аня"]
grades = [5, 4, 2, 3]
# Вася получил 5, Коля 4 и т.д.
```

Было бы хорошо, если бы у нас была возможность иметь тип данных, в котором элементы нумеруются не натуральными числами, а произвольными объектами. Такой тип данных существует: в Python он называется *словарём* (*dictionary*).

Вот так можно создать словарь в Python:

In [ ]:

```
gradebook = {"Вася": 5, "Коля": 4, "Петя":2, "Аня": 3}
```

Это похоже на создание списка, но есть ряд отличий. Во-первых, мы использовали фигурные скобки вместо квадратных, чтобы показать, что создаём именно словарь. Во-вторых, словарь состоит из *записей*, каждая запись состоит из двух частей: *ключа* (*key*) и *значения* (*value*). Ключ и значение разделяются двоеточием. Например, у нас есть запись `"Аня": 3` с ключом `"Аня"` и значением `3`. Всего наш словарь `gradebook` сейчас содержит четыре записи, ключами которых являются имена студентов, а значениями — их оценки.

In [ ]:

```
gradebook
```

Заметим, что при печати Python переупорядочил записи в словаре. На самом деле, порядок вывода записей в словаре является произвольным: внутри словаря записи не имеют никакого порядка. Поэтому нельзя обратиться, например, к «первой записи», но зато можно обратиться к записи с данным ключом:

In [ ]:

```
gradebook['Аня']
```

In [ ]:

```
gradebook['Вася']
```

Можно изменить значение записи, точно так же, как изменить элемент списка.

In [ ]:

```
gradebook['Аня'] = 5
```

In [ ]:

```
gradebook
```

Можно добавить новую запись.

In [ ]:

```
gradebook['Иннокентий'] = 4
```

In [ ]:

```
gradebook
```

При попытке обратиться к записи, которой нет, мы получим сообщение об ошибке:

In [ ]:

```
gradebook['Alice']
```

Часто нам хочется иметь возможность запросить запись, а в случае, если её нет, получить какое-нибудь «значение по умолчанию», а не ошибку. Для этого нужно использовать метод `get()` вместо квадратных скобок.

In [ ]:

```
gradebook.get('Alice')
```

Здесь вернулось `None`:

In [ ]:

```
print(gradebook.get('Alice'))
```

In [ ]:

```
gradebook.get('Вася')
```

Можно было бы передать `get()` второй аргумент, и тогда в случае, если такого ключа в словаре нет, то будет возвращен он.

In [ ]:

```
gradebook.get('Alice', 'No such student')
```

In [ ]:

```
gradebook.get('Вася',  'No such student')
```

Можно получить список всех ключей словаря:

In [ ]:

```
gradebook.keys()
```

На самом деле это не совсем список, но эта штука ведёт себя почти как список и из неё можно сделать список. Аналогично со списком всех значений словаря.

In [ ]:

```
gradebook.values()
```

Ключами словарей могут быть не только строчки. Допустим, мы хотим создать словарь, в котором ключами будут числа. Нет ничего проще:

In [ ]:

```
squares={1:1, 2:4, 3:9}
```

In [ ]:

```
squares
```

In [ ]:

```
squares[1]
```

In [ ]:

```
squares[2]
```

В предыдущих двух строчках `squares` ведёт себя примерно как список, но если внимательно приглядеться, то видно, что это не список, а всё-таки словарь.

In [ ]:

```
squares
```

Например, у любого непустого списка есть элемент с индексом 0, а у `squares` такого нет:

In [ ]:

```
squares[0]
```

Перебор записей в словаре

Как обрабатывать информацию в словаре? Для перебора всех элементов списка можно было использовать цикл `for`. А что будет, если ему скормить словарь вместо списка? Попробуем:

In [ ]:

```
for k in gradebook:
    print(k)
```

Понятно! Цикл `for` в этом случае перебирает все *ключи* нашего словаря. А зная ключ, можно получить и значение:

In [ ]:

```
for k in gradebook:
    print("Студент", k, "имеет оценку", gradebook[k])
```

Однако, есть более изящный способ получить сразу ключ и значение очередной записи: использовать `items()`.

In [ ]:

```
for k, v in gradebook.items():
    print("Студент",k,"имеет оценку", v)
```

Как работает этот код? Здесь используется метод `items()`, возвращающий список (точнее, итератор), состоящий из кортежей вида `(ключ, значение)`.

In [ ]:

```
list(gradebook.items())
```

Оператор `for` в этом случае понимает, что нужно при каждом проходе цикла выбрать очередной кортеж и присвоить его первый элемент (то есть ключ) переменной `k`, а второй элемент (то есть значение) переменной `v` (конечно, эти переменные могли бы называться иначе). С аналогичным поведением мы уже встречались, когда обсуждали конструкцию `enumerate`.

Вот так можно найти все записи с заданным значением — например, всех студентов, получивших оценку 4:

In [ ]:

```
for k, v in gradebook.items():
    if v==4:
        print(k)
```

Заметим, что такой «поиск по значению» требует перебора всех записей в словаре и если словарь большой, то он будет занимать много времени — хотя «поиск по ключу» будет по-прежнему выполняться быстро. Кстати, можно быстро проверить, существует ли в словаре запись с данным ключом:

In [ ]:

```
"Коля" in gradebook
```

In [ ]:

```
"Alice" in gradebook
```

Если бы мы хотели искать среди значений, то нужно было бы явно это указать с помощью метода `values()`:

In [ ]:

```
1 in gradebook.values()
```

In [ ]:

```
4 in gradebook.values()
```

Вообще оператор `in` не ограничивается только использованием со словарями: он может использоваться, например, со списками:

In [ ]:

```
5 in [1,2,3,5,8]
```

In [ ]:

```
6 in range(1,5)
```

Создание словарей и функция `zip()`

Есть разные способы создавать словари. Например, можно создать пустой словарь и постепенно заполнять его элементами:

In [ ]:

```
my_dict = {}
```

In [ ]:

```
my_dict[1] = 1
my_dict['hello'] = 'world'
```

In [ ]:

```
my_dict
```

Заметим, что в одном и том же словаре прекрасно уживаются элементы разных типов (в данном случае — строки и целые числа).

Можно создать словарь иначе, передав функции `dict()` список, состоящий из пар ключ-значение (в некоторо смысл, это обратная операция методу `items()`):

In [ ]:

```
dict([('hello','world'), ('one', 'two')])
```

Допустим, у нас есть два списка, в одном находятся имена студентов, а в другом их оценки. Как можно из этих списков создать словарь, для которого имена были бы ключами, а оценки значениями?

А вот так:

In [ ]:

```
students = ["Вася", "Коля", "Петя", "Аня"]
grades = [5, 4, 2, 3]
new_gradebook = list(zip(students,grades))
new_gradebook
```

Здесь используется удобная функция `zip()`, применение которой не ограничивается созданием словарей. Подобно застёжки-молнии, она «состёгивает» (отсюда и название) несколько списков. Например, `zip()` делает из пары списков список пар:

In [ ]:

```
list(zip([1,2,3],['a','b','c']))
```

Эту конструкцию можно использовать, когда нам нужно перебрать элементы двух связанных между собой списков. Например, вот так можно вывести информацию о том, какой студент какую оценку имеет, не используя словари:

In [ ]:

```
for student, grade in zip(students, grades):
    print(student, "has grade", grade)
```

Функцию `zip()` можно использовать и более чем с двумя списками:

In [ ]:

```
list(zip([1,2,3,4], [5,6,7,8], ['a','b','c','d']))
# списка три, поэтому на выходе получится список из троек
```

Если какой-то из списков окажется короче, то `zip()` «обрежет» остальные списки:

In [ ]:

```
list(zip([1,2,3], ['a','b']))
```

Какие объекты могут быть ключами словарей

До сих пор мы рассматривали словари, ключами которых являются строки и числа. На самом деле, ключами могут быть и более сложно устроенные объекты. Например, представим себе такую реализацию фрагмента таблицы сложения в виде словаря:

In [ ]:

```
sums = {(2,3): 5, (4, 1): 5, (5, 7): 12}
```

In [ ]:

```
sums
```

Здесь ключами являются кортежи, состоящие из двух чисел, а значениями — суммы этих чисел.

In [ ]:

```
sums[(2,3)]
```

In [ ]:

```
sums[(4,1)]
```

In [ ]:

```
sums[(5,7)]
```

В этом месте проявляется важное отличие кортежей от списков: последние не могут быть ключами словарей, поскольку могут изменяться.

In [ ]:

```
sums = { [1,2]: 3}
```

На этом мы закончим краткое введение в словари и перейдём к следующей теме.

Списковые включения (list comprehensions)

Мы ранее частенько сталкивались с такой задачей: дан список, в котором записаны числа, но в виде строчек. Создать новый список, в котором числа были бы числами. Мы могли решить эту задачу с помощью цикла:

In [ ]:

```
str_list = ["1", "5", "12", "7"]

int_list = []
for s in str_list:
    int_list.append(int(s))

print(int_list)
```

За создание нового списка отвечают три строчки. Писать их каждый раз довольно тоскливо, и создатели Python придумали (точнее, заимствовали из функциональных языков программирования, а те его заимствовали у математиков) гораздо более изящный синтаксис. Он устроен вот так:

In [ ]:

```
int_list = [int(s) for s in str_list]
```

Кадратные скобки вокруг выражения должна подсказать, что мы создаём список (потому что когда нужно создать список, мы обычно заключаем его элементы в квадратные скобки). Выражение внутри скобок нужно читать буквально:

список, состоящий из элементов `int(s)` для (`for`) элементов `s` из (`in`) списка `str_list`

In [ ]:

```
int_list
```

Видите? Кавычки исчезли — перед нами список, состоящий из чисел. Магия! Исходный список `str_list` при этом не изменился:

In [ ]:

```
str_list
```

Аналогично можно применять любую операцию к элементам списка. Например, возведём все элементы из `int_list` в квадрат:

In [ ]:

```
[x**2 for x in int_list]
```

или удвоим все элементы списка:

In [ ]:

```
double_list = [x*2 for x in int_list]
```

In [ ]:

```
double_list
```

или прибавим к ним 1:

In [ ]:

```
[x+1 for x in int_list]
```

или превратим их в числа с плавающей точкой:

In [ ]:

```
[float(x) for x in int_list]
```

Как видите, с элементами списков можно делать что угодно! Однако, это ещё не все. В синтаксисе списочных включений можно производить фильтрацию. Например, нам нужны только те элементы, которые больше 6. Мы можем их выбрать таким образом:

In [ ]:

```
[x for x in int_list if x > 6]
```

Когда мы пишем здесь `x for x` мы имеем в виду, что нужно просто подставить в новый список элементы старого, ничего с ними не делая (только выбирая нужные). Но можно и как-то их модифицировать:

In [ ]:

```
[x**2 for x in int_list if x > 6]
```

Решим теперь такую задачу: есть два списка с числами, а мы хотим найти их поэлементную сумму.

In [ ]:

```
X = [2, 5, 8]
Y = [1, 3, 100]
```

Её можно решить таким образом (для перебора элементов двух списков одновременно используем конструкцию `zip()`, обсуждавшуюся выше):

In [ ]:

```
Z = []
for x, y in zip(X, Y):
    Z.append(x + y)
print(Z)
```

Но со списочными включениями тот же код выглядит гораздо симпатичнее:

In [ ]:

```
[x + y for x, y in zip(X, Y)]
```

Кстати, можно использовать синтаксис, похожий на списочное включение, чтобы создавать словари:

In [ ]:

```
squared = {i: i**2 for i in range(10)}
squared
```

Функция `map()`

У списочных включений есть аналог, который сейчас считается не слишком удобным, но иногда встречается: функция `map()`.

In [ ]:

```
str_list
```

Вот так она решается с помощью `map()`:

In [ ]:

```
int_list = list(map(int, str_list))
```

In [ ]:

```
list(int_list)
```

Функция `map()` принимает два аргумента. Первым аргументом она принимает функцию, после этого она применяет эту функцию к каждому из элементов списка. В общем, записи вида `list(map(int, str_list))` и `[int(x) for x in str_list]` почти эквивалентны.

Когда действие, которое нужно применить, уже существует в виде функции (как в случае с `int`), то конструкция с `map()` выглядит даже более лаконичной, чем списочное включение. Но если нам нужно сделать что-то менее тривиальное, списочные включения явно проще:

In [ ]:

```
[int(x)+1 for x in str_list]
```

Чтобы реализовать это с помощью `map()`, нужно объявить новую функцию, которая будет возвращать значение выражения `int(x)+1` и передать её `map()`.

In [ ]:

```
def my_func(x):
    return int(x)+1
```

In [ ]:

```
list(map(my_func,str_list))
```

Для краткости можно использовать `lambda`-функции, но такой подход гораздо менее прозрачен, чем списочные включения, и сейчас использовать его не рекомендуется.

Два слова об эффективности

Использовать списочные включения не только приятно, но и полезно: они работают эффективнее, чем код с циклом.

In [ ]:

```
from random import random
from math import sqrt
N = 10000
mylist = [random() for _ in range(N)]
```

In [ ]:

```
%%timeit

newlist = []
for x in mylist:
    newlist.append(sqrt(x))
```

In [ ]:

```
%%timeit
newlist = [sqrt(x) for x in mylist]
```

In [ ]:

```
%%timeit
newlist = list(map(sqrt, mylist))
```

Как видно из этих данных (магическое слово `%timeit` позволяет измерить, сколько времени уходит на какую-то операцию), списочные включения быстрее обычного цикла. `map()` работает примерно с такой же скоростью, как и списочные включения (иногда чуть медленнее, иногда чуть быстрее).

Сложные структуры данных

Списки позволяет сохранить некоторый ряд значений, но зачастую нужно уметь работать с более сложные структурами — например, с таблицами. В некоторых языках программирования есть *двумерные массивы*. Аналогом двумерного массива в Python является «список списков», то есть такой список, элементами которого являются другие списки. С чем-то подобным мы уже встречались.

Рассмотрим пример: таблица, в которой записаны результаты по нескольким домашним работам у нескольких студентов. (Допустим, мы присвоили студентам некоторые номера и поэтому нам не нужно знать, кого как зовут.) Её можно записать в виде списка списков, например, по строчкам:

In [ ]:

```
table = [["HW1", "HW2", "HW3", "HW4"], [4, 3, 4, 4], [3, 4, 3, 4], [4, 5, 5, 4]]
```

Здесь каждый элемент списка `table` — это строчка нашей таблицы, то есть тоже список. Например, вот так можно узнать, что записано в третьей строке и четвертом столбце нашей таблицы:

In [ ]:

```
table[2][3]
```

Что здесь произошло? Мы сначала вызвали третью строку таблицы с помощью

In [ ]:

```
table[2]
```

А потом из этой третьей строки выбрали четвертый элемент с помощью `[3]` . Можно было бы записать это более подробно:

In [ ]:

```
row = table[2]
print(row[3])
# row[3] это то же самое, что table[2][3]
```

Вот так можно напечатать все элементы таблицы по строчкам:

In [ ]:

```
for row in table:
    print(*row)
```

Допустим теперь, что нам всё же хочется знать, какой студент какую оценку получил. Тогда мы могли бы вместо списка списков использовать словарь, у которого списки были бы значениями:

In [ ]:

```
gradebook = {'Bill': [4, 3, 2], 'Alice': [3, 4, 5], 'Bob': [5, 5, 4]}
```

Вот так можно посмотреть, какую оценку получил Боб по второй домашке:

In [ ]:

```
gradebook['Bob'][1]
```

На сегодня всё! :)