Программирование на языке Python для сбора и анализа данных

Сортировка. Форматирование строк

**kwargs

Расскажем об одном способе передачи аргументов функции, использующем словари. Напомним два способа передачи аргументов:

In [ ]:

```
def myfunc(x=0, y=1):
    print("x =",x)
    print("y =",y)
```

In [ ]:

```
myfunc(12,19)
```

In [ ]:

```
myfunc(y=2, x=5)
```

Как показывает второй пример, аргументы можно передавать, указывая их имена. Допустим, мы хотим написать функцию, которая принимает неопределенное количество именованных аргументов (мы даже не знаем заранее, каких). Это можно сделать следующим образом:

In [ ]:

```
def new_func(**kwargs):
    print(kwargs)
```

In [ ]:

```
new_func(x=1, y=5, z=8, s = "Some string")
```

Две звёздочки в определении функции `new_func()` говорят следующее: «все именованные параметры, переданные этой функции, следует поместить в словарь `kwargs`». Как видим, именно так это и работает: например, параметр `x=1` превратился в запись `'x': 1` в словаре `kwargs`. Однако, это происходит только с «бесхозными» параметрами: если бы у функции был отдельный параметр `x`, то он бы не попал в `kwargs`. Например:

In [ ]:

```
def other_func(x, **kwargs):
    print(kwargs)
other_func(x=10,y=5)
```

Сортировка

Сортировка — то есть расположение элементов списка в каком-то определённом порядке — распространённая программистская задача. Для сортировки списков в Python есть два главных инструмента. Первый — метод `sort()`, выполняющий сортировку *in place*, то есть внутри самого списка. Например:

In [ ]:

```
my_list = [6, 9, 2, 7, 12, 8]
```

In [ ]:

```
my_list.sort()
```

In [ ]:

```
my_list
```

Метод `sort()` *меняет* исходный список (и поэтому, кстати, умеет работать только со списками — у кортежей такого метода нет). Если вместо этого вы хотите создать новый список, то следует использовать функцию `sorted()`.

In [ ]:

```
my_list = [6, 9, 2, 7, 12, 8]
```

In [ ]:

```
sorted_list = sorted(my_list)
sorted_list
```

Так мы создали новый список. Старый при этом остался без изменений.

In [ ]:

```
my_list
```

Функцию `sorted()` можно применять не только к спискам, но и к неизменяемым последовательностям — например, к кортежам. На выходе всегда получается список.

In [ ]:

```
my_tuple = (7, 1, 2, 6)
print (sorted(my_tuple))
```

Сортировка строк

Сортировать можно списки, состоящие не только из чисел, но и из более сложных объектов — лишь бы умели сравнивать их между собой. Например, строки можно сравнивать между собой — они упорядочены в *лексикографическом порядке*, то есть «по алфавиту» и так, как они шли бы в словаре (имеется в виду обычный бумажный словарь, а не тип данных Python).

In [ ]:

```
"abcd" < "b"
```

In [ ]:

```
"abcd" < "addd"
```

In [ ]:

```
"a" < "aa"
```

Вот так выглядит сортировка списка из строк:

In [ ]:

```
str_list = ["Bob", "Alice", "Bill", "Weigu"]
str_list.sort()
str_list
```

Сортировка и циклы

Можно использовать функцию `sorted()` совместно с оператором `for` чтобы обрабатывать элементы списка в каком-то определённом порядке. Например, у нас есть словарь и мы хотим вывести его элементы в порядке возрастания значения ключа. Тогда об этом надо явно попросить Python, как показывает следующий пример:

In [ ]:

```
gradebook = {'Bob': 3,  'Alice': 5, 'Weigu': 4, 'Bill': 2}
for k in gradebook:
    print(k, gradebook[k])
```

Как видите, элементы не упорядочены. Вот так их можно упорядочить при выводе:

In [ ]:

```
for k in sorted(gradebook):
    print(k, gradebook[k])
```

Более сложные примеры сортировки

Можно сортировать список в обратном порядке (по убыванию). Для этого используется параметр `reverse`.

In [ ]:

```
sorted([4, 8, 1, 7], reverse=True)
```

Можно сортировать не только числа и строки, но и более сложные объекты. Нарпимер, рассмотрим такую табличку (реализованную как список кортежей), в которой записаны имена студентов и их оценки по нескольким работам.

In [ ]:

```
names = [("Bob", 8, 4, 9),
         ("Alice", 7, 8, 9),
         ("Weigu", 7, 5, 3),
         ("Dan", 6, 4, 3)]
```

In [ ]:

```
names.sort()
```

In [ ]:

```
names
```

Судя по результату, логично предположить, что сортировка была выполнена по первому элементу — имени студента. Действительно, кортежи сравниваются примерно так же, как строки, лексикографически. Сначала сравниваются первые элементы:

In [ ]:

```
('a', 8) < ('b', 7)
```

Если первый элемент совпадает, то сравниваются вторые элементы и т.д.

In [ ]:

```
('a', 8) < ('a', 7)
```

А что делать, если бы нам захотелось отсортировать кортежи в списке `names` не по первому элементу, а по второму или ещё какому-нибудь? Для этого необходимо использовать параметр `key`, задающий ключ сортировки. Прежде, чем мы это сделаем, нужно сказать несколько слов о том, как можно одной функции передать другую функцию в качестве параметра.

Отступление: функции как аргументы функций

Рассмотрим такую функцию:

In [ ]:

```
def superfunc(f):
    return f(2)
```

В качестве аргумента она принимает какую-то функцию `f`, вызывает эту функцию передаёт ей в качестве аргумента число 2 и возвращает тот результат, который вернула `f`.

Например:

In [ ]:

```
from math import sqrt
superfunc(sqrt)
```

Мы импортировали функцию `sqrt()` из модуля `math`, после чего передали функции `superfunc()` функцию `sqrt` в качестве параметра. Обратите внимание: при передаче после функции `sqrt` нет скобок: это потому, что мы её не *вызываем*, а *передаём* другой функции. Функция `superfunc` взяла нашу функцию `sqrt` и вызвала её, передав ей число 2 в качестве параметра. То есть вычислила корень из двух.

Можете представить себе, что `sqrt` — это рецепт, записанный на бумажке. Мы передаём его в виде такой бумажки функции `superfunc` и она его как-то использует. Передадим другую бумажку — она использует её. Например:

In [ ]:

```
def plusodin(x):
    return x + 1
```

In [ ]:

```
superfunc(plusodin)
```

Если мы попробуем передать функции `superfunc` что-то другое — например, строчку или число — ничего не получится (она ожидает именно функцию).

In [ ]:

```
superfunc("sqrt")
```

Ключи сортировки

Вернёмся к задаче о сортировке таблицы, представленной в виде списка кортежей. Чтобы отсортировать такой список по второму элементу, необходимо сначала создать функцию, которая будет возвращать второй элемент переданного ей кортежа (или списка).

In [ ]:

```
def get_second_element(x):
    return x[1]
```

Посмотрим, как она работает:

In [ ]:

```
get_second_element([7, 8, 4, 2])
```

Теперь мы передаем эту функцию в качестве параметра `key` методу `sort()` (функция `sorted()` тоже сработает):

In [ ]:

```
names.sort(key=get_second_element)
```

In [ ]:

```
names
```

Видно, что теперь строки оказались упорядочены по второму столбцу (первой оценке): у `Dan` она самая низкая (`6`), у `Bob` самая высокая (`8`), а у `Alice` и `Weigu` одинаковые (`7`).

Возникает естественный вопрос: а как упорядочены в этом случае строчки, соответствующие `Alice` и `Weigu`? Ответ: в том порядке, в котором они шли в исходном списке. Это бывает удобно, если вы хотите упорядочить сначала по одному параметру, а потом по другому: просто выполните сортировку последовательно, сначала по *второму* параметру, а потом по первому.

Чтобы не определять каждый раз функцию типа `get_second_element` можно использовать готовую: для этого надо импортировать специальную функцию `itemgetter`:

In [ ]:

```
from operator import itemgetter
```

In [ ]:

```
sorted(names, key=itemgetter(2))
# упорядочили по третьему столбцу
```

In [ ]:

```
sorted(names, key=itemgetter(3))
# упорядочили по четвертому столбцу
```

Допустим, мы хотим упорядочить по третьему столбцу, а если третий столбец даёт одну и ту же оценку, то по алфавиту. Это можно сделать так: *сначала* упорядочим по алфавиту, а *потом* — по третьему столбцу.

In [ ]:

```
print(names)
names.sort(key=itemgetter(0))
print(names)
names.sort(key=itemgetter(2))
print(names)
```

Больше подробностей о сортировке можно найти в официальном tutorial (https://docs.python.org/3/howto/sorting.html), а мы перейдём к следующей теме.

Форматирование строк

Зачастую требуется вставить значение каких-то переменных в какую-то строчку. Пример, который нам уже встречался.

In [ ]:

```
name = "Alice"
grade = 5
print("Student", name,"has grade", grade)
```

С помощью `print()` можно вывести такую строку на печать, но если бы мы хотели передать её какой-то другой функции, то надо было бы придумывать что-то другое. И это другое уже придумано!

Есть два распространённых способа подставлять значение переменных в строку (это часто называется *интерполяцией*, хотя не имеет никакого отношения к одноименной математической операции). Первый способ более классический.

In [ ]:

```
new_str = "Student %s has grade %i" % (name, grade)
print(new_str)
```

Здесь используется оператор %, который для строк делает следующую операцию: берёт строку слева от него, находит там все «поля для подстановки» (*placeholders*) — в данном случае это `%s` и `%i`, после чего берёт переменные, перечисленные справа от него (это может быть одна переменная или кортеж из нескольких переменных, как в данном случае) и подставляет их последовательно — первую переменную на место первого placeholder, вторую на место второго и т.д.

Буквы в обозначениях placeholders означают тип переменной: в данном случае `%s` — это строка, а `%i` — целое число. Вот ещё несколько примеров:

In [ ]:

```
print("The number is %i" % 2.3)
print("The number is %f" % 2) # f значит float
print("The number is %.2f" % 2.1393) # два знака после точки
print("The number is %04i" % 3) # дополнить до четырёх знаков нулями
```

При использовании оператор % нужно быть осторожным: он имеет приоритет по сравнению с арифметическими операциями, поэтому вы можете получить неожиданный результат, если не поставите скобки:

In [ ]:

```
print("a = %i" % 3*3)
```

Здесь произошло следующее: сначала выполнился код `"a = %i" % 3`, а потом результат умножился на 3 (что для строчек эквивалентно трёхкратному повторению). Если вы хотели подставить результат выполнения `3*3`, то нужно было сделать вот так:

In [ ]:

```
print("a = %i" % (3*3))
```

Второй способ форматирования («новый») заключается в использовании метода `format()`. Он действует примерно так:

In [ ]:

```
"hello, {0}, this is {1}, again {0}, {var}".format(7, 9, var="test")
```

Здесь не приходится явно указывать типы данных (подставляется строковое представление переменной). Одно и то же значение может использоваться несколько раз (к ним можно обращаться по номерам и именам. Впрочем, можно и не указывать явно номера — тогда переменые будут подставляться по очереди:

In [ ]:

```
"Fist var: {}, the second one: {}".format(8, 1)
```

Форматирование может быть довольно сложным и никто не в силах запомнить все тонкости. Неплохая документация на эту тему (как по оператору %, так и по методу `format()`) собрана здесь (https://pyformat.info/).

Хитрости с вещественными числами

Кстати:

In [ ]:

```
print("%f" % (0.1+0.2))
```

Вроде бы, ничего неожиданно, но давайте увеличим точность…

In [ ]:

```
print("%.18f" % (0.1+0.2))
```

Когда мы попросили вывести результат с точностью 18 знаков после запятой, откуда-то взялись непонятные значащие цифры в конце. Это связано с тем фактом, что компьютеры используют двоичную систему счисления, а в ней числа типа `0.1` записываются в виде *бесконечной* периодической дроби и не могут быть представлены в виде конечной дроби. При арифметических операциях возникают ошибки округления, которые и приводят к таким эффектам.

Иногда эти эффекты становятся опасными. Вы думаете, что `0.1 + 0.2` это `0.3`? У вашего компьютера другое мнение на этот счёт:

In [ ]:

```
0.1 + 0.2 == 0.3
```

Однако, не стоит отчаиваться: вы можете использовать обыкновенные дроби или специальный модуль `decimal` для работы с десятичными дробями.

In [ ]:

```
from fractions import Fraction
```

In [ ]:

```
Fraction(1, 10) + Fraction(2, 10)
```

In [ ]:

```
Fraction(1, 3) + Fraction(1,2)
```

In [ ]:

```
from decimal import Decimal
```

In [ ]:

```
Decimal("0.1")+Decimal("0.2")
```

In [ ]:

```
Decimal("0.1") + Decimal("0.2") == Decimal("0.3")
```

Подробнее о десятичных и бинарных дробях можно прочитать в официальной документации (https://docs.python.org/3/tutorial/floatingpoint.html#tut-fp-issues).