Information technologies

Особенности работы с изменяемыми типами данных

Для начала я хотел бы обсудить некоторые тонкости при работе со списками, словарями и другими изменяемыми типами данных. Рассмотрим простой пример вызова функции.

In [ ]:

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

Эта функция увеличивает свой аргумент на единицу и возвращает то, что получилось.

In [ ]:

```
y = 5
print("Function returns", my_func(y))
print("y =", y)
```

Ничего неожиданного. Функция не изменила переменную `y` и не должна была этого делать. Давайте теперь попробуем сделать что-то аналогичное со списком.

In [ ]:

```
def other_func(my_list):
    my_list.append(1)
    return my_list
```

Эта функция получает в качестве аргумента список, добавляет к нему в конец элемент `1` и возвращает то, что получилось. Попробуем её вызвать:

In [ ]:

```
this_list = [6, 9, 33]
print("this_list before function:",this_list)
print("function returns:",other_func(this_list))
print("this_list after function", this_list)
```

Ой. Произошло что-то странное. Функция `other_func` модифицировала список, который ей был передан, хотя создан он был вне этой функций и внутри функции не был определён как `global` (да и вообще, внутри функции работа велась с другой переменной).

Почему это произошло? Чтобы разобрать в этом вопросе, проще всего посмотреть на визуализацию.

In [ ]:

```
%load_ext tutormagic
```

In [ ]:

```
%%tutor --lang python3
def other_func(my_list):
    my_list.append(1)
    return my_list

this_list = [6, 9, 33]
other_func(this_list)
```

Две сортировки

Давайте рассмотрим ещё два примера, которые показывают различие между этими двумя сценариями — в них снова будут участвовать списки. Выполните две визуализации и найдите, в чём разница. После этого читайте дальше.

In [ ]:

```
%%tutor --lang python3
def return_sorted(my_list):
    my_list = sorted(my_list)
    return my_list

this_list = [33, 1, 55]
print("this_list before function:", this_list)
print("function returned:", return_sorted(this_list))
print("this_list after function:", this_list)
```

In [ ]:

```
%%tutor --lang python3
def sort_and_return(my_list):
    my_list.sort()
    return my_list

this_list = [33, 1, 55]
print("this_list before function:", this_list)
print("function returned:", sort_and_return(this_list))
print("this_list after function:", this_list)
```

Итак, в чём разница?

Функции `return_sorted()` и `sort_and_return()` выполняют примерно одну и ту же работу: принимают на вход список, сортируют его и возвращают. В тот момент, когда происходит вход в функцию (Step 5), ситуация в обоих фрагментах идентична: вне функции есть переменная `this_list`, внутри функции есть переменная `my_list`, они обе указывают на один и тот же список. Ключевое различие происходит на следующем шаге. Функция `return_sorted()` использует функцию `sorted()`, создающую новый список. Затем она присваивает (`=`) результат выполнения `sorted()` переменной `my_list`. Это приводит к тому, что переменная с этим именем начинает ссылаться на новый объект (Step 7), а старый (на который по-прежнему ссылается `this_list`) остаётся в неприкосновенности. Совсем иначе действует `sort_and_return()` — она использует метод `sort()`, сортирующий список *in place*, внутри самого себя. При этом новый объект не создаётся, операция присваивания не используется и переменная `my_list` продолжает ссылаться на тот же список, что и раньше. Просто он оказывается отсортированным.

Обратите внимание на схожесть `return_sorted()` и `my_func()` из примера выше: в обоих случаях используется оператор присваивания. Разница состоит в том, что с числовой переменной никак иначе поступить было нельзя, поскольку числа неизменяемы, а в случае со списком у разработчика есть выбор: можно создать новый объект и присвоить его старой переменной, а можно изменить уже существующий объект, не создавая новый.

Создание копии

Предположим теперь, что мы хотим написать функцию, которая принимает на вход список, возвращает такой же список, но с одним добавленным элементом, а сам исходный список не меняет. Это можно сделать, например, вот так:

In [ ]:

```
def return_append(L, a):
    new_L = L.copy()
    new_L.append(a)
    return new_L
```

In [ ]:

```
outer_list = [7, 8, 9]
print("outer_list before funciton", outer_list)
print("function returned", return_append(outer_list, 55))
print("outer_list after function", outer_list)
```

Здесь главная хитрость в использовании `L.copy()` — напомним, что этот метод создаёт копию существующего списка. Дальше мы снова выполняем операцию присваивания (то есть теперь `new_L` является именем для копии списка `L`, а не для самого списка `L`) и мы можем делать с этим новым списком `new_L` что угодно. Старый список не изменится.

Не только вызовы функций

Проблемы, аналогичные рассмотренным выше, возникают не только при вызове функций. Начнём с простого примера с циклом.

In [ ]:

```
some_list = [7, 9, 11]
for x in some_list:
    x = x + 1
print(some_list)
```

Список `some_list` не поменялся, и в этом нет ничего удивительного. Но давайте теперь рассмотрим чуть более сложную ситуацию со списком списков.

In [ ]:

```
table = [[1, 5], [7, 9]]
for row in table:
    row.append(77)
print(table)
```

И снова «ой». Что случилось? Посмотрим на визуализаторе.

In [ ]:

```
%%tutor --lang python3
table = [[1, 5], [7, 9]]
for row in table:
    row.append(77)
print(table)
```

При выполнении первого шага цикла (Step 3) в переменную `row` записывается первый элемент списока `table`. Однако, этот элемент сам является списком — а точнее ссылкой на список. На следующем шаге (Step 4) к этому списку добавляется элемент `77`. Потом `row` становится ссылкой на второй элемент списка `table`. К ней тоже добавляется элемент `77`.

Обратите внимание на параллель с предыдущим сюжетом: здесь тоже участвует вызов метода списка, который меняет этот список *in place*.

Головоломка

Как вы думаете, что будет, если выполнить следующий код? Попробуйте его выполнить, посмотрите на результат и попытайтесь объяснить.

```
A = [[]]*5
A[0].append(1)
print(A)
```

Изменение итерируемого объекта в цикле

В примере выше мы меняли содержимое «внутренних» списков, но сам список `table` оставался неизменным: в нём не менялось число элементов и элементы оставались ссылками на те же самые списки-строки, что и раньше. А можно ли менять сам список во время итераций? Оказывается, можно. Хотя в большинстве случаев лучше этого не делать. Прежде, чем рассматривать пример, напомним о том, как работает метод `pop()` у списка:

In [ ]:

```
L = [6, 9, 44, 8]
print(L.pop())
print(L)
```

Он удаляет последний элемент из списка и возвращает его же. Применим теперь его следующим образом:

In [ ]:

```
L = [7, 8, 9, 10]
for x in L:
    print("Pop element", L.pop())
    print(x)
```

Цикл исполняется два раза: к тому моменту, как цикл закончит обрабатывать элемент 8, элементы 9 и 10 из списка будут уже удалены, необработанных элементов в списке не останется и цикл прекратится.

Догадаться о том, что произойдёт, можно лишь очень внимательно изучив код. Это значит, что код не очень хорош: глядя на хороший код, вы можете понять, что он будет делать.

Со словарями ситуация иная.

In [ ]:

```
d = {1:2, 3:4}
for k, v in d.items():
    del d[3]
    print(k, v)
```

Здесь команда `del d[3]` удаляет элемент с ключом `3` из словаря. Поскольку порядок итерирования элементов словаря не определён, как корректно продолжить итерации после того, как размер словаря был изменён, никто не знает. Поэтому такая операция запрещена.

Однако, это не означает, что запрещено менять значение словаря при выполнении цикла. Например, мы хотим прибавить ко всем значениям число 1. Следующий наивный метод ожидаемо не сработает:

In [ ]:

```
d={1:2, 3:4}
for k, v in d.items():
    v = v + 1
print(d)
```

На самом деле, эту задачу следует решать так:

In [ ]:

```
d = {1:2, 3:4}
for k in d:
    d[k] = d[k] + 1
print(d)
```

Множества

Ещё один базовый тип данных в Python — это множество. Оно соответствует математическому понятию множества — то есть набора каких-то элементов. Каждый элемент может или входить в множество, или не входить.

> Если вы в данный момент являетесь слушателем нашего курса по Python, то вы принадлежите множеству слушателей. Нельзя быть «дважды слушателем курса»: каждый элемент может входить в множество только один раз.

In [ ]:

```
my_set = {6, 9, 11, 11, 9, 'hello'}
```

In [ ]:

```
my_set
```

Как видно из этого простого примера, элементы множества также не упорядочены.

In [ ]:

```
{6, 9, 11, 11, 9, 'hello'} == {9, 'hello', 11, 6}
```

Вот так можно проверить, лежит ли элемент в множестве.

In [ ]:

```
9 in my_set
```

In [ ]:

```
10 in my_set
```

Конечно, оператор `in` работает не только для множеств. Например, `4 in [2, 4, 8, 10]` вернёт `True`. Однако для списков эта операция является медленной — вернее, *массовой*: чем больше список, тем больше операций сравнения нужно произвести, чтобы понять, лежит ли в нём какой-то конкретный элемент. В случае с множествами время на проверки практически не растёт с ростом числа элементов множества.

С множествами можно делать разные операции — с ними мы знакомы по математическим курсам. Например, объединение и пересечение двух множеств даёт новое множество:

In [ ]:

```
{6, 8, 9} | {6, 11, 7}
```

In [ ]:

```
{6, 8, 9} & {6, 11, 7}
```

> Отметим ещё раз: порядок элементов во множестве не определён. Если вам нужно вывести элементы множества в каком-то заранее заданном порядке, то можно превратить его в сортированный список с помощью функции `sorted()`.

In [ ]:

```
s = {"Hello", "World", "Test", "Guest", "Aaaaa", "Zzzzz","Zz","Q"}
print(s)
print(sorted(s)) # здесь выводится уже не множество, а список: обратите внимание на ква
дратные скобки
```

Пример использования множеств

допустим, мы просим пользователя ввести команду, но хотим дать ему возможность ввести одну и ту же команду разными способами. Например, чтобы остановить программу, пользователь может ввести слово `stop` или `STOP` или `Stop` или просто букву `s` или `S`. Можно обработать этот случай с помощью нескольких условий, соединённых `or`:

In [ ]:

```
s = 'stop'
if s == 'stop' or s == 'Stop' or s == 'STOP' or s == 'S' or s == 's':
    print("Okay, stopping")
```

А можно создать множество для всех возможных вариаций команды `stop` и проверять, входит ли наша команда в это множество:

In [ ]:

```
s = 'stop'
STOPS = {'stop', 'Stop', 'STOP', 'S', 's'}
if s in STOPS:
    print("Okay, stopping")
```

Впрочем, в этом месте, вероятно, вместо множества можно было бы использовать и просто список.

Ещё немного про строки

Я давно собирался рассказать про методы для работы со строками. Вообще этих методов существует много и про все не расскажу, но некоторые из них мы сейчас обсудим.

In [ ]:

```
s = "hello world, hello"
new_s = s.replace("hello", "Hi")
print(new_s)
print(s)
```

Вот так, например, можно заменить подстроку в строке. Обратите внимание: строка является неизменяемым типом данных, поэтому, в отличие от методов списков типа `append()` методы строк никогда не меняют саму строку (это вообще невозможно), а вместо этого создают новую строку и возвращают результат.

Если вы хотели заменить только несколько первых вхождений (например, только первое слово `hello`, но не второе), можно было добавить третий аргумент метода `replace` — он показывает, сколько раз нужно произвести замену.

In [ ]:

```
"hello world, hello".replace("hello", "Hi", 1)
```

Вот так можно найти подстроку в строке:

In [ ]:

```
s.index("world")
```

In [ ]:

```
s.find("world")
```

Оба метода возвращают индекс первого символа подстроки. Разница состоит в том, что если `index()` не сможет найти подстроку вообще, он выдаст ошибку (exception), а если с аналогичной проблемой столкнётся `find()`, то он вернёт число `-1` в качестве индекса.

Кстати, проверить, входит ли подстрока в строку, можно ещё вот так:

In [ ]:

```
"world" in s
```

А вот так можно посчитать, сколько подстрок в строке встречается:

In [ ]:

```
s.count("o")
```

Подробности можно найти в официциальной справке (https://docs.python.org/3/library/string.html).

Файловый ввод-вывод

Мы начинаем работать с файлами. Сейчас будем обсуждать только чтение и запись. О том, как запускать файлы на исполнение, отдельная история — для этого существует метод `subprocess`, мы до него когда-нибудь дойдём. (Может быть.) Также для начала речь пойдёт о текстовых файлах или похожих на текстовые (например, код на Python или CSV-файл будет текстовым). Бывают также бинарные файлы, которые «глазками» читать бесполезно — о некоторых из них будет отдельный рассказ.

Допустим, мы хотим прочитать файл.

In [ ]:

```
f = open("func.txt")
s = f.read()
f.close()
print(s)
```

Что здесь произошло? Во-первых, мы открыли для чтения файл `func.txt`, лежащий в нашем текущем рабочем каталоге. Узнать, какой каталог рабочий, можно следующим образом:

In [ ]:

```
import os
os.getcwd()
```

Функция `open()` вернула объект типа `file` — переменную, которую можно использовать, чтобы работать с файлом. Затем мы считали содержимое файла в строчку `s`, после чего закрыли файл. Закрывать файлы очень полезно: если вы забудете закрыть файл, другое приложение не сможет его открыть (например, чтобы в него что-нибудь записать).

Функция `read()` считает весь файл в одну большую строковую переменную. Это не всегда бывает удобно (учитывая, что строки в Python неизменяемые и из-за этого работа с ними не всегда бывает эффективной), поэтому есть разные другие сценарии работы с файлами. Например, можно считать содержимое файла в список, разбив по строкам.

In [ ]:

```
f = open("func.txt")
lines = f.readlines()
f.close()
```

In [ ]:

```
print(lines)
```

Заметим, что каждая из строк заканивается символом перевода строки `\n` — они присутствовали в файле и мы их честно из него считали. Вот так можно вывести файл по строкам, пронумеровав их:

In [ ]:

```
for i, line in enumerate(lines, 1):
    print(i, line, end="")
```

Другой способ это сделать — не создавать отдельный список, а итерировать прямо сразу файловый объект.

In [ ]:

```
f = open("func.txt")
for i, line in enumerate(f, 1):
    print(i, line, end="")
f.close()
```

Этот метод является более предпочтительным, если файл большой. В этом случае считать его в память целиком может быть невозможно, а обработать по одной строчке — вполне возможно.

Здесь есть, правда, некоторые хитрости. Рассмотрим, например, такой код:

In [ ]:

```
f = open("func.txt")
for line in f:
    print(line, end="")
print("----The next one----")
for line in f:
    print(line, end="")
f.close()
```

Что здесь произошло? Почему второй цикл вообще не выполнился (ничего не выведено после строчки `----The next one----`)? Очень просто: переменная `f` , хоть и прикидывается списком строк, когда мы её итерируем, на самом деле таковым не является. В действительности при открытии файла мы запоминаем позицию, на которой мы этот файл читаем. Изначально она указывает на самое начало файла, но с каждой итерацией сдвигается. Когда мы прочитаем файл целиком, дальнейшие попытки из него что-то прочитать ни к чему не приведут: указатель текущей позиции сдвинулся до самого конца и файл закончился.

Впрочем, есть возможность вернуться в начало: для этого нужно использовать метод `seek()` .

In [ ]:

```
f = open("func.txt")
for line in f:
    print(line, end="")
print("----The next one----")
f.seek(0)
for line in f:
    print(line, end="")
f.close()
```

Запись в файлы

Чтобы содать файл и записать в него что-то, нужно открыть его *на запись*. Это делается путём передачи функции `open` второго аргумента — здесь надо написать строчку `"w"` (от *write*).

> **Внимание!** Если файл, который вы пытаетесь открыть на запись, уже существует, он будет **удалён без какого-либо предупреждения**.

Записывать информацию в файл, открытый на запись, можно, например, с помощью метода `write()`.

In [ ]:

```
f = open("other.txt", "w")
f.write("Hello\n")
f.close()
```

Проверим, что получилось:

In [ ]:

```
open('other.txt').read()
```

Мы видим, что действительно записали в файл `other.txt` строчку `Hello\n`. Заметим, что здесь мы открыли файл на запись, но не стали присваивать файловый объект какой-либо переменной, а сразу вызвали от него метод `read()`. В этом случае файл будет закрыт автоматически через некоторое время после выполнения этой команды. (Система выдаёт предупреждение о том, что мы не закрыли файл явно — в некоторых случаях это может приводить к каким-то проблемам.)