Лекция № 5

Т. Ф. Хирьянов

Работа со списками в Python 3

Кодировки символов

Американская стандартная система кодирования ASCII позволяла сопоставить распространенным печатным и непечатным символам соответствующие числовые коды. Изначально она включала в себя латинский алфавит. С помощью изменения старшего бита на 1 имелась возможность кодировать национальные символы. Из-за этого возникло очень много различных национальных кодировок. Более того, такой подход не позволял писать международные письма, использующие одновременно несколько кодировок.

Это привело к созданию международного стандарта Unicode, который предлагал несколько кодировок, среди которых UTF-16, UTF-32. Первая из них позволяла кодировать 65532 символа, а вторая приблизительно $4 \cdot 10^9$, причем в них на каждый символ отводилось одинаковое количество памяти. Из-за довольно большого размера таких равномерных кодировок была предложена новая UTF-8, которая являлась неравномерной, т.е. в ней разные символы имеют разную длину. Также это позволяет добавлять все новые и новые символы. Язык Python ориентирован на стандарт Unicode, поэтому в строках можно использовать любые символы, если знать их код.

Тип str

Для разных языков программирования существуют разные подходы к реализации символьного типа. Например, в Pascal это тип *char*, который является символьным. В языке программирования Си существует тип с таким же названием, который является числовым. В Python же не существует символьного типа, вместо этого есть тип «строка» (*str*). Чтобы обратиться к конкретному символу в строке необходимо указать его номер в качестве индекса.

```
s = 'Hello'
s[0] == 'H'
type(s[0]) == class<str>
```

Также символ можно понимать как срез строки длиной в один символ. При этом необходимо учитывать, что, например, в Pascal строки изменяемые. В Python это не так – для того чтобы изменить строку, необходимо породить новый объект и затем сделать привязку к старому идентификатору.

Функции строк

Для работы со строками одними из самых важных являются функции поиска подстроки в строке find и rfind, которые имеют похожий синтаксис.

```
s = 'Ehehe...'
s.find('he') # return 1
s.rfind('he') # return 3
```

При этом функция find возвращает позицию первого символа подстроки, найденной от начала строки. Но функция rfind осуществляет поиск с конца строки и, соответственно, возвращает позицию последнего элемента подстроки.

Для того чтобы найти количество вхождений подстроки в строку существует функция count. В случае перекрывания подстрок, как в данном примере,

```
s = 'AAAA'
num_AA = s.count('AA')          # num_AA == 2
```

можно считать, что подстрока вошла два или три раза. В Python 3 принято считать что здесь две подстроки.

Литералы строк

Строка в Python записывается в одинарных или двойных кавычках. Обе записи равнозначны и существуют для того, чтобы избегать двусмысленных выражений. Впрочем, можно было бы использовать обратный слэш для экранирования кавычек (например,
'), чтобы они воспринимались как символы внутри строки.

Если строка слишком длинная, чтобы уместиться на экране, то разумно перенести ее с помощью обратного слэша, который будет экранировать не печатающийся символ переноса каретки. При этом Python будет воспринимать ее как одну целую строку.

Для того чтобы работать с многострочным объектом, его записывают в тройных кавычках (каждая из которых состоит из одной либо двух кавычек).

```
s = 'Ehehe...'
s = "Ehehe..."

s_with_comma = 'I\'m␣a␣student'
s_with_comma = "I'm␣a␣student"
s_with_comma = 'Company␣"Parallels"'

long_s = "...␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣..."

many_s = '''  ...
          ...
          ...  '''
```

Для написания строк бывает удобно пользоваться специальными символами, такими как
«\n» – разрыв строки
«\t» – табуляция

Для того чтобы хранить в строке путь к некоторому файлу в Windows, необходимо экранировать обратный слэш (т.е. сделать его двойным). Можно также написать перед открывающей кавычкой символ *r*. Тогда все, что находится внутри кавычек будет восприниматься так, как написано.

```
path = 'C:\\my\\1.py'
path = r'C:\my\1.py'
```

Наивный поиск подстроки в строке

В качестве примера алгоритма можно рассмотреть поиск позиции первого элемента первого вхождения подстроки «he» в строку «Ehehe...».

Для начала определяются их длины. Затем элементы в строке пробегаются до той позиции, при которой подстрока, начинавшаяся бы с нее, еще не вышла бы за пределы строки. В теле цикла проверяется совпадение подстроки с частью строки, начинающейся с текущей позиции. В случае несовпадения происходит выход из вложенного цикла. Для того чтобы выйти из внешнего цикла сразу после нахождения нужной подстроки используется логическая переменная-флаг found. В случае, если такой подстроки нет, программа возвращает -1.

```
s = 'Ehehe...'
sub = 'he'
len_s = len(s)
len_sub= len(sub)

pos = 0
while pos + len_sub <= len_s:
    found = True
    i = 0
    while i < len_sub:
        if s[pos + i] != sub[i]:
            found = False
        i += 1
    if found:
        break
    pos += 1
else:
    pos = -1
```

Списки в Python

В Python список (тип list) представляет собой массив ссылок на объекты. Он изменяем: каждую ссылку можно связать с другим объектом.

```
A = [1, 2, 3, 4, 5]      # type(A) == class<list>
A[0] = 10                # type(A[0]) == int
                         # A == [10, 2, 3, 4, 5]
```

Можно узнать длину списка с помощью len(A), а также min(A) и max(A), если объекты, с которыми связаны ссылки можно сравнивать друг с другом.

Если присвоить один список другому, то тогда оба идентификатора станут указывать на один и тот же список.

```
A = [1, 2, 3]
B = A
B[0] = 10          # B == [10, 2, 3]
                   # A == [10, 2, 3]
```

Данная особенность связывания справедлива и при вызове функций. Так если в объявлении функции изменить сам объект, то при выходе из нее он останется измененным.

```
def f(B):
    B[0] = 10
f(A)
# B[0] == 10
```

Если же в объявлении функции изменяются только локальные списки, то ссылки на них теряются при выходе из функции.

```
def f(B):
    B = B[::-1]
f(A)
```

В данном примере идентификатор B связывается сначала со списком, с которым уже связан A. По этой ссылке создается новый объект – срез, и с ним связывается идентификатор B. Из-за этого ссылка на прежний объект теряется, и при выходе из функции список A остается неизменным.

Еще один пример показывает, как можно добавить элемент в конец списка и как объединить несколько списков в один, сохраняя тот же порядок элементов, что и в исходных.

```
A = [1, 2, 3]
B = A
B.append(4)       # B == [1, 2, 3, 4]
C = A + B         # C == [1, 2, 3, 4, 1, 2, 3, 4]
```

Копия списка

Создание копии списка можно проводить разными способами. Так, циклически пробежавшись по все элементам списка и добавив их в новый, получится копия прежнего. Однако то же самое можно сделать короче с использованием срезов.

```
B = []
for x in A:
    B.append(x)
# shorter
B = A[:]
```

Если необходимо обезопасить список от изменения после выполнения какой-либо функции, то разумно передавать ей не сам список, а его срез. Т.е. вместо f(A) писать f(A[:]).

Присваивание в срез списка

Присваивание в срез с двумя параметрами никогда не приводит к ошибке. В следующем примере элементы массива, начиная с позиции 1 до элемента с позицией 3, вырезаются, а на их место вставляется другой фрагмент.

```
A = [0, 1, 2, 3, 4]
A[1:3] = [10, 20, 30]       # A == [0, 10, 20, 30, 3, 4]
A[1:4] = []                 # A == [0, 3, 4]
```

Присваивание же в срез с тремя параметрами может привести к ошибке, если размер присваиваемого списка не является подходящим (не укладывается в исходном списке с заданным шагом)

```
A = [0, 1, 2, 3, 4]
A[1:5:2] = [10, 20]             # A == [0, 10, 2, 20, 4]
A[1:5:2] = [10, 20, 30]         # Error!
```

Также имеется возможность вставки фрагмента между соседними элементами исходного списка.

```
A = [1, 2, 3]
A[1:1] = [5, 6]           # A == [1, 5, 6, 2, 3]
```

Следующий пример показывает, как с помощью присваивания в срез можно создавать списки с довольно сложным порядком следования элементов.

```
# [1, 6, 2, 5, 3, 4]
A = [0]*6                    # A == [0, 0, 0, 0, 0, 0]
A[::2] = [1, 2, 3]
A[::-2] = [4, 5, 6]
```

Обращение списка

В некоторых случаях списки бывают достаточно большими и на их обращение с помощью создания среза может не хватить памяти.

```
A = list(map(int, input().split()))
i = 0
while i < len(A) // 2:
    tmp = A[i]
    A[i] = A[-1-i]
    A[-1-i] = tmp
```

Данный алгоритм прост. Для его понимания достаточно заметить, что отрицательный индекс означает перебор элементов списка от последнего (-1) к первому (-n) и что исходный список обходится только до половины, так как иначе новый список был бы обращен в прежний.

Циклический сдвиг

Зачастую в некоторых задачах требуется осуществить циклический сдвиг элементов в списке. Так, при сдвиге влево необходимо сначала сохранить первый элемент, чтобы потом, после работы цикла, поставить его на последнее место.

```
A = [0, 1, 2, 3, 4]
tmp = A[0]
for i in range(0, len(A) - 1):
    A[i] = A[i+1]
A[-1] = tmp
# A == [1, 2, 3, 4, 0]
```

При наличии необходимой памяти, то же самое можно проделать с помощью срезов.

```
A[:] = A[1:] + A[0:1]
```

Кортежи

В отличие от списков *кортежи* являются неизменяемыми. В следующем примере показано, как объявить кортеж и как создать пустой. Важно отметить, что для устранения путаницы с переменными числового типа вводится специальная форма записи кортежа из одного элемента.

```
A = 1, 2, 3, 4, 5        # type(A) == class<tuple>
A = ()                   # empty tuple
A = 1                    # type(A) == int
A = (1, )                # type(A) == class<tuple>
```