# Введение в Python и Eric

Иван Хахаев, 2009

## Строки и последовательности

В главе «Простой ввод и вывод» мы уже имели дело со строками (данными символьного типа). Теперь рассмотрим работу со строками в Python более детально. Строка представляет собой конечный набор символов, у которого можно определить длину (количество символов). К отдельному символу в строке можно обратиться по номеру, причём номера начинаются с нуля. Как мы видели раньше, для строк определена операция «сложения» (конкатенации). Кроме того, в Python для строк определена операция «умножения», т.е. повторения заданное количество раз.

Нужно заметить, что во всех приведённых ниже примерах нужно вводить строки в «английской» раскладке клавиатуры, поскольку со строками, состоящими из символов кириллицы, эти примеры работают некорректно. Эффект связан с необходимостью обработки символов Unicode при работе с кириллицей, что пока не является предметом рассмотрения.

Рассмотрим следующую задачу.

Имеется строка и некоторое число N. Если количество символов в строке меньше N, дополнить строку до длины в N символов «звёздочками» слева.

```
#  -*- coding: utf-8 -*-
# Имеется строка и некоторое число N.
# Если количество символов в строке меньше N,
# дополнить строку до длины в N символов «звёздочками» слева.
#
s1=raw_input("Исходная строка: ")
n=input("Заданный размер: ")
# Определяем длину строки
l=len(s1)
k=n-l
# Сравниваем и решаем задачу
if k > 0:
    s2='*'*k
else:
    s2=''

s=s2+s1
print "Итоговая строка: ",s
```

Ход решения вполне понятен из текста программы. Сначала вводим исходные данные, потом вычисляем длину исходной строки. Затем вычисляем разницу между заданным числом и длиной строки, и если эта разница есть и положительна, создаём строку -заполнитель из символов «*». Если же длина исходной строки не меньше N, строку-заполнитель делаем пустой (" - пустая строка). Результатом является конкатенация строки-заполнителя и исходной строки.

На рис. 1 показаны два примера выполнения этой программы с различными значениями `l` и `n`.

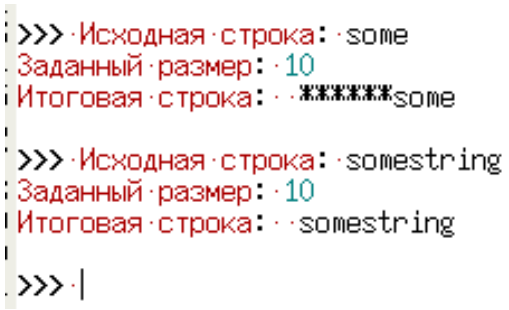


Рисунок 1. Решение задачи о дополнении строки

Как видим, символы и строки, участвующие в операциях, указываются в одиночных (') или двойных кавычках. Операция «умножения» строк обозначается также, как и для чисел, но имеет другое значение.

Отметим, что операция «умножения» (повторения) строк специфична для Python (и некоторых других современных языков), а в «классических» языках программирования (Basic, Pascal) она отсутствует.

Теперь рассмотрим работу со строковыми функциями на примере следующей задачи.

Пусть требуется преобразовать дату в «компьютерном» представлении (системную дату) в «российский» формат, т.е. день/месяц/год (например, 17/05/2009).

Для решения этой задачи нужно прежде всего получить системную дату, посмотреть на её устройство, выделить из неё компоненты (номер дня. номер месяца и номер года, а потом из этих компонентов «сконструировать» требуемый результат.

Текст программы получится примерно следующим (`script-06.py`).

```
#  -*- coding: utf-8 -*-
# Системную дату ПК преобразовать в российский стандарт,
# т.е. день/месяц/год
#
from datetime import date
# Получаем текущую дату
d1=date.today()
# Преобразуем результат в строку
ds=str(d1)
print "Системная дата ",ds
# Используем методы строки
lls=ds.split('-')
yy=lls[0]
mm=lls[1]
dd=lls[2]
```

```
rusdate=dd+"/"+mm+"/"+yy
print "Российский стандарт ",rusdate
```

Результат работы программы показан на рис. 2.

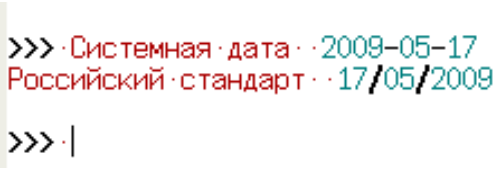


Рисунок 2. Решение задачи о преобразовании дат

В этой простой задаче встречается несколько «новинок». связанных с Python. Во-первых, для работы с датами и временем используется библиотека `datetime`, которую (или нужные модули из неё) необходимо подключать. В данном случае подключена только функция `date`. Далее, для получения текущей даты используется специальный вызов `date.today()`. Как видно на рис. 2, компоненты даты разделены символом «-», поэтому очень соблазнительно найти функцию, выполняющее деление строки по заданному символу. Однако дата относиться к типу данных «дата/время», поэтому перед применением к ней операции деления строки её нужно сначала преобразовать в строку, что и делается с помощью функции `str()`.

К полученной в результате преобразования строке применяется функция (метод) `split()`, которая делит строку на части по заданному символу и записывает результат в специфичную для Python структуру данных — список (набор значений, разделённых запятыми). К элементам списка можно обращаться по номерам, первый элемент имеет номер 0.

В конечном итоге из элементов списка. содержащего компоненты даты, с помощью конкатенации конструируется желаемый результат.

Более «питоновский» вариант решения этой задачи приведён ниже (`script-06-a.py`).

```
#  -*- coding: utf-8 -*-
# Системную дату ПК преобразовать в российский стандарт,
# т.е. день/месяц/год
#
from datetime import date
# Получаем текущую дату
d1=date.today()
# Преобразуем результат в строку
ds=str(d1)
print "Системная дата ",ds
# Используем методы строки и списка
lls=ds.split('-')
```

```
lls.reverse()
#
rusdate="/".join(lls)
print "Российский стандарт ",rusdate
```

Здесь использованы особенности списка — операция `reverse()`, которая «переворачивает» список, и операция `join()`, которая соединяет элементы списка в строку, используя предварительно указанный символ соединения. Предпоследняя строка в этом тексте означает «соединить элементы списка в строку, вставив между ними косую черту («слэш»)».

Строка с точки зрения Python также является вариантом списка. т.е. к её элементам (символам) тоже можно обращаться пор номерам. В качестве примера рассмотрим следующую задачу:

Определить, является ли строка "перевёртышем" типа "abba", "казак" и пр.

Текст программы приведён ниже (`script-07.py`).

```
#  -*- coding: utf-8 -*-
# Определить, является ли строка "перевёртышем"
# типа "abba", "казак" и пр.
s1=raw_input("Исходная строка: ")
# Определяем длину строки
l=len(s1)
key=1
for i in range(0,l//2):
    if s1[i]==s1[l-i-1]:
        k=1
    else:
        k=0
    key=key*k
if key==1:
    print "Является"
else:
    print "Не является"
```

Здесь прежде всего определяется длина строки (функция `len()`), а потом сравниваются символы от концов строки к середине. Причём для строк с нечётным количеством символов «центральный» символ затронут не будет (однако он и так совпадает сам с собой). Для выполнения такого сравнения используются индексы элементов строки в цикле `for`, в котором элементы «перебираются» по номерам от первого (с номером 0) до элемента, соответствующего результату целочисленного деления длины строки на 2, при этом элементы сравниваются с «зеркальными» относительно середины строки элементами. Если символы совпадают, некоторому «флагу» присваивается значение 1, в противном случае — значение 0. «Ключевой» признак того, что строка является «перевёртышем» означает, что она симметрична относительно середины, т.е. все «флаги» должны быть в состоянии 1.

Для каждой пары символов текущий «ключ» умножается на очередное значение «флага», и

если хотя бы один «флаг» окажется в состоянии 0, условие «перевёртыша» не выполнится.

Цикл `for` (составной опенратор `for`) применяется для выполнения повторяющихся действий заранее известное количество раз (или когда это количество повторений можно вычислить, как в рассматриваемом примере). Признаком начала цикла является символ «:», после чего все операторы в цикле должны писаться с отступом. Первая срока без отступа означает окончание цикла (она уже вне цикла).

Аналогично, в операторе `if` также очень важны отступы, поэтому текст программы получился таким «ступенчатым».

IDE Eric, как уже показывалось ранее, расставляет отступы автоматически и показывает начало каждого составного оператора.

Примеры работы программы показаны на рис. 3.

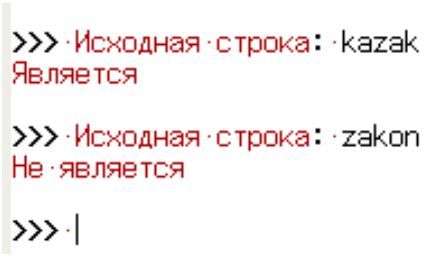


Рисунок 3. Определение строки-»перевёртыша»

Задачу, рассмотренную в начале этой главы, можно также решать и с использованием цикла `for`, как это делается в «классических» языках программирования:

```
#  -*- coding: utf-8 -*-
# Имеется строка и некоторое число N.
# Если количество символов в строке меньше N,
# дополнить строку до длины в N символов «звёздочками» слева.
#
s1=raw_input("Исходная строка: ")
n=input("Заданный размер: ")
# Определяем длину строки
l=len(s1)
k=n-l
s2=''
# Сравниваем и решаем задачу
if k > 0:
    for i in range(0,k):
        s2=s2+"*"
else:
    s2=''
```

```
s=s2+s1
print "Итоговая строка: ",s
```

Здесь по сравнению с первым вариантом решения добавились лишние операции — определение строки `s2` перед началом её формирования и дополнительный составной оператор цикла. Из сравнения двух решений видно что использование особенностей Python даёт более компактное и понятное решение.

Если количество повторений операций заранее неизвестно, но известно условие прекращения выполнения операций, используется цикл (составной оператор) `while`. Покажем его использование на следующем примере (s`cript-08.py`):

```
#  -*- coding: utf-8 -*-
# Последовательно вводятся ненулевые числа.
# Определить сумму положительных и сумму отрицательных чисел.
# Закончить ввод чисел при вводе 0.
#
sp=0 # сумма положительных
sn=0 # сумма отрицательных
#
chislo=input("Следующее число: ")
#
while chislo != 0:
     if chislo > 0:
        sp=sp+chislo
     else:
        sn=sn+chislo
     chislo=input("Следующее число: ")
#
print "Сумма положительных: ",sp
print "Сумма отрицательных: ",sn
```

Здесь в цикле `while` вводимые числа проверяются на знак, соответствующие суммы копятся в переменных `sp` и `sn`. Особенностью является необходимость получить число первый раз, поэтому один ввод числа сделан перед началом цикла.

Работа программы проиллюстрирована на рис. 4.

```
>>> Следующее число: 2
Следующее число: -2
Следующее число: 8
Следующее число: -4
Следующее число: 0
Сумма положительных:  10
Сумма отрицательных:  -6

>>> Следующее число: 0
Сумма положительных:  0
Сумма отрицательных:  0

>>>
```

Рисунок 4. Примеры обработки
числовой последовательности

Последовательности (наборы данных неопределённой длины, с известным условием окончания) могут быть не только числовые, но и символьные, строковые, а также смешанные. В режиме ввода элементов последовательности в Python удобно формировать списки, работа с которыми будет подробнее рассмотрена в следующей главе.