Оглавление

Часть II. Python на практике

Приложения

ГЛАВА 7
Кортежи и списки

> Человек отличается от низших приматов страстью
> к составлению списков.
>
> *Гарри Аллен Смит*

В предыдущих главах мы говорили о базовых типах данных Python, таких как булевы значения, целочисленные значения, числа с плавающей точкой и строки. Если представлять их как атомы, то структуры данных, которые мы рассмотрим в этой главе, можно назвать молекулами. Так и есть: мы объединим базовые типы в более сложные структуры, которые вы будете использовать каждый день. Бóльшая часть работы программиста состоит из «разрезания» данных и «склеивания» их в конкретные формы, поэтому сейчас вы узнаете, как пользоваться ножовками и клеевыми пистолетами.

Большинство языков программирования могут представлять последовательность в виде объектов, проиндексированных по их позиции, выраженной целым числом: первый, второй и далее до последнего. Вы уже знакомы со *строками* — последовательностями символов.

В Python есть еще две структуры-последовательности: *кортежи* и *списки*. Они могут содержать ноль и более элементов. В отличие от строк в кортежах и списках допускаются элементы разных типов: по факту каждый элемент может быть *любым* объектом Python. Это позволяет создавать структуры любой сложности и глубины.

Почему же в Python имеются как списки, так и кортежи? Кортежи *неизменяемы*. Когда вы включаете в кортеж элемент (всего один раз), он «запекается» и больше не изменяется. Списки же можно *изменять* — добавлять и удалять элементы в любой удобный момент. Я покажу вам множество примеров использования обоих типов, сделав акцент на списках.

Кортежи

Давайте сразу же рассмотрим один очевидный вопрос. Вы могли слышать два возможных варианта произношения слова *tuple* (кортеж). Какой же из них является правильным? Гвидо ван Россум, создатель языка Python, написал в Twitter

(http://bit.ly/tupletweet): «Я произношу слово tuple как too-pull по понедельникам, средам и пятницам и как tub-pull — по вторникам, четвергам и субботам. В воскресенье я вообще об этом не говорю :)».

Создаем кортежи с помощью запятых и оператора ()

Синтаксис создания кортежей несколько необычен, что вы и увидите в следующих примерах.

Начнем с создания пустого кортежа с помощью оператора `()`:

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

Чтобы создать кортеж, содержащий один элемент и более, после каждого элемента надо ставить запятую. Это вариант для кортежей с одним элементом:

```
>>> one_marx = 'Groucho',
>>> one_marx
('Groucho',)
```

Вы можете поместить элемент в круглые скобки и получить такой же кортеж:

```
>>> one_marx = ('Groucho',)
>>> one_marx
('Groucho',)
```

Однако следует иметь в виду: если в круглые скобки вы поместите один объект и опустите при этом запятую, в результате вы получите не кортеж, а тот же самый объект (в этом примере строку `'Groucho'`):

```
>>> one_marx = ('Groucho')
>>> one_marx
'Groucho'
>>> type(one_marx)
<class 'str'>
```

Если в вашем кортеже более одного элемента, ставьте запятую после каждого из них, кроме последнего:

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

При отображении кортежа Python выводит на экран скобки. Как правило, они не нужны для определения кортежа, но с ними более безопасно, так как они делают кортеж более заметным:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

В тех случаях, когда у запятой могут быть и другие варианты использования, также рекомендуется ставить круглые скобки. В следующем примере вы можете создать кортеж с одним элементом и присвоить ему значение, поставив в конце запятую, но не можете передать эту конструкцию как аргумент функции:

```
>>> one_marx = 'Groucho',
>>> type(one_marx)
<class 'tuple'>
>>> type('Groucho',)
<class 'str'>
>>> type(('Groucho',))
<class 'tuple'>
```

Кортежи позволяют присваивать значение нескольким переменным одновременно:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> a, b, c = marx_tuple
>>> a
'Groucho'
>>> b
'Chico'
>>> c
'Harpo'
```

Иногда это называется *распаковкой кортежа*.

Вы можете использовать кортежи для обмена значениями с помощью одного выражения, не применяя временную переменную:

```
>>> password = 'swordfish'
>>> icecream = 'tuttifrutti'
>>> password, icecream = icecream, password
>>> password
'tuttifrutti'
>>> icecream
'swordfish'
>>>
```

Создаем кортежи с помощью функции tuple()

Функция преобразования `tuple()` создает кортежи из других объектов:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
```

Объединяем кортежи с помощью оператора +

Это похоже на объединение строк:

```
>>> ('Groucho',) + ('Chico', 'Harpo')
('Groucho', 'Chico', 'Harpo')
```

Размножаем элементы с помощью оператора *

Принцип похож на многократное использование оператора +:

```
>>> ('yada',) * 3
('yada', 'yada', 'yada')
```

Сравниваем кортежи

Сравнение кортежей похоже на сравнение списков:

```
>>> a = (7, 2)
>>> b = (7, 2, 9)
>>> a == b
False
>>> a <= b
True
>>> a < b
True
```

Итерируем по кортежам с помощью for и in

Итерирование по кортежам выполняется так же, как и итерирование по другим типам:

```
>>> words = ('fresh','out', 'of', 'ideas')
>>> for word in words:
...     print(word)
...
fresh
out
of
ideas
```

Изменяем кортеж

Этого сделать вы не можете! Как и строки, кортежи неизменяемы. Но ранее вы уже видели на примере строк, что можно *сконкатенировать* (объединить) кортежи и создать таким образом новый кортеж:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> t1 + t2
('Fee', 'Fie', 'Foe', 'Flop')
```

Это означает, что вы можете изменить кортеж следующим образом:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> t1 += t2
>>> t1
('Fee', 'Fie', 'Foe', 'Flop')
```

Это уже не тот же самый кортеж `t1`. Python создал новый кортеж из исходных `t1` и `t2` и присвоил ему имя `t1`. С помощью `id()` вы можете увидеть, когда имя переменной будет указывать на новое значение:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> id(t1)
4365405712
>>> t1 += t2
>>> id(t1)
4364770744
```

Списки

Списки особенно удобны для хранения в них объектов в определенном порядке, особенно если порядок или содержимое нужно будет изменить. В отличие от строк список изменяем: вы можете добавить новые элементы, перезаписать существующие и удалить ненужные. Одно и то же значение может встречаться в списке несколько раз.

Создаем списки с помощью скобок []

Список можно создать из нуля и более элементов, разделенных запятыми и заключенных в квадратные скобки:

```
>>> empty_list = [ ]
>>> weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> big_birds = ['emu', 'ostrich', 'cassowary']
>>> first_names = ['Graham', 'John', 'Terry', 'Terry', 'Michael']
>>> leap_years = [2000, 2004, 2008]
>>> randomness = ['Punxsatawney", {"groundhog": "Phil"}, "Feb. 2"}
```

Список `first_names` показывает, что значения не должны быть уникальными.

Если вы хотите размещать в последовательности только уникальные значения и вам неважен их порядок, множество (set) может оказаться более удобным вариантом, чем список. В предыдущем примере список big_birds вполне мог быть множеством. О множествах вы прочитаете в главе 8.

Создаем список или преобразуем в список с помощью функции list()

Вы также можете создать пустой список с помощью функции `list()`:

```
>>> another_empty_list = list()
>>> another_empty_list
[]
```

Функция `list()` преобразует другие *итерабельные* типы данных (например, кортежи, строки, множества и словари) в списки. В следующем примере строка преобразуется в список, состоящий из односимвольных строк:

```
>>> list('cat')
['c', 'a', 't']
```

В этом примере кортеж преобразуется в список:

```
>>> a_tuple = ('ready', 'fire', 'aim')
>>> list(a_tuple)
['ready', 'fire', 'aim']
```

Создаем список из строки с использованием функции split()

Как я упоминал в разделе «Разделяем строку с помощью функции split()» главы 5, функцию `split()` можно использовать для преобразования строки в список, указав некую строку-разделитель:

```
>>> talk_like_a_pirate_day = '9/19/2019'
>>> talk_like_a_pirate_day.split('/')
['9', '19', '2019']
```

Что, если в оригинальной строке содержится несколько включений строки-разделителя подряд? В этом случае в качестве элемента списка вы получите пустую строку:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

Если бы вы использовали разделитель `//`, состоящий из двух символов, то получили бы следующий результат:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('//')
>>>
['a/b', 'c/d', '/e']
```

Получаем элемент с помощью конструкции [смещение]

Как и в случае со строками, вы можете извлечь одно значение из списка, указав его смещение:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0]
'Groucho'
>>> marxes[1]
'Chico'
>>> marxes[2]
'Harpo'
```

Так же и отрицательные индексы отсчитываются с конца строки:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

Смещение должно быть допустимым для этого списка — позицией, которой вы ранее присвоили значение. Если вы укажете позицию, которая находится перед списком или после него, будет сгенерировано исключение (ошибка). Вот что случится, если мы попробуем получить шестого брата Маркс (Marxes) (смещение равно 5, если считать от нуля) или же пятого перед списком:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[5]
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> marxes[-5]
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Извлекаем элементы с помощью разделения

Можно извлечь из списка подсписок, использовав *разделение* (slice):

```
>>> marxes = ['Groucho', 'Chico,' 'Harpo']
>>> marxes[0:2]
['Groucho', 'Chico']
```

Такой фрагмент списка тоже является списком.

Как и в случае со строками, при разделении можно пропускать некоторые значения. В следующем примере мы извлечем каждый нечетный элемент:

```
>>> marxes[::2]
['Groucho', 'Harpo']
```

Теперь начнем с последнего элемента и будем смещаться влево на 2:

```
>>> marxes[::-2]
['Harpo', 'Groucho']
```

И наконец, рассмотрим прием инверсии списка:

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

Ни одно из этих разделений не затронуло сам список `marxes`, поскольку мы не выполняли присваивание. Чтобы изменить порядок элементов в списке, используйте функцию `list.reverse()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.reverse()
>>> marxes
['Harpo', 'Chico', 'Groucho']
```

Функция reverse() изменяет список, но не возвращает его значения.

Как и в случае со строками, если при разделении указать некорректный индекс, исключение не сгенерируется. Будет использован ближайший корректный индекс или же возвращено пустое значение:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[4:]
[]
>>> marxes[-6:]
['Groucho', 'Chico', 'Harpo']
>>> marxes[-6:-2]
['Groucho']
>>> marxes[-6:-4]
[]
```

Добавляем элемент в конец списка с помощью функции append()

Традиционный способ добавления элементов в список — вызов метода `append()`, который один за одним добавит их в конец списка. В предыдущих примерах мы забыли о `Zeppo`, но ничего страшного не случилось, поскольку список можно изменить. Добавим его прямо сейчас:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

Добавляем элемент на определенное место с помощью функции insert()

Функция `append()` добавляет элементы только в конец списка. Когда вам нужно добавить элемент и поставить его на заданную позицию, используйте функцию `insert()`. Если вы укажете смещение `0`, элемент будет добавлен в начало списка. Если значение смещения выходит за пределы списка, элемент будет добавлен в ко-

нец, как делает и функция append(): таким образом, вам не нужно беспокоиться о том, что Python сгенерирует исключение:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.insert(2, 'Gummo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo']
>>> marxes.insert(10, 'Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
```

Размножаем элементы с помощью оператора *

В главе 5 вы видели, что можно размножить символы строки с помощью оператора *. Точно так же можно сделать и со списками:

```
>>> ["blah"] * 3
['blah', 'blah', 'blah']
```

Объединяем списки с помощью метода extend() или оператора +

Можно объединить один список с другим, используя extend(). Предположим, что некий добрый человек дал нам новый список братьев Маркс, который называется others, и мы хотим добавить его в основной список marxes:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.extend(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Можно также использовать операторы + или +=:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes += others
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Если бы мы использовали append(), список others был бы добавлен как *один* из элементов списка, а не дополнил бы своими элементами список marxes:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.append(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', ['Gummo', 'Karl']]
```

Это еще раз показывает, что список может содержать элементы разных типов. В этом случае — четыре строки и список из двух строк.

Изменяем элемент с помощью конструкции [смещение]

Так же как значение какого-либо элемента из списка можно получить по смещению, его можно и изменить:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[2] = 'Wanda'
>>> marxes
['Groucho', 'Chico', 'Wanda']
```

Опять же смещение должно быть корректным для заданного списка.

Вы не можете таким способом изменить символ в строке, поскольку строки, в отличие от списков, неизменяемы. В списке можно изменить как количество элементов, так и сами элементы.

Изменяем элементы с помощью разделения

В предыдущем разделе вы увидели, как получить подсписок с помощью разделения. Помимо этого, с помощью разделения можно присвоить значения подсписку:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [8, 9]
>>> numbers
[1, 8, 9, 4]
```

То, что находится справа от = и что вы присваиваете списку, может содержать иное количество элементов, нежели список, указанный слева:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [7, 8, 9]
>>> numbers
[1, 7, 8, 9, 4]

>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = []
>>> numbers
[1, 4]
```

На самом деле то, что находится справа от оператора присваивания, может даже не быть списком. Подойдет любой итерабельный объект, элементы которого можно сделать элементами списка:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = (98, 99, 100)
>>> numbers
[1, 98, 99, 100, 4]

>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = 'wat?'
>>> numbers
[1, 'w', 'a', 't', '?', 4]
```