Тыпы коллекций

С.В. Лемешевский (sergey.lemeshevsky@gmail.com)

Институт математики НАН Беларуси

Mar 2, 2020

Содержание

Рассматриваются кортежи и списки, а также новые типы коллекций, включая словари и множества.

1. Последовательности

Последовательности – это один из типов данных, поддерживающих оператор проверки на вхождение (`in`), функцию определения размера (`len()`), оператор извлечения срезов (`[]`) и возможность выполнения итераций. В языке Python имеется пять встроенных типов последовательностей: `bytearray`, `bytes`, `list`, `str` и `tuple`. Ряд дополнительных типов последовательностей реализован в стандартной библиотеке; наиболее примечательным из них является тип `collections.namedtuple`. При выполнении итераций все эти последовательности гарантируют строго определенный порядок следования элементов.

Строки мы уже рассматривали выше, а в этом разделе познакомимся с кортежами, именованными кортежами и списками.

1.1. Кортежи

Кортеж – это упорядоченная последовательность из нуля или более ссылок на объекты. Кортежи поддерживают тот же синтаксис получения срезов, что и строки. Это упрощает извлечение элементов из кортежа. Подобно строкам, кортежи относятся к категории неизменяемых объектов, поэтому мы не можем замещать или удалять какие-либо их элементы. Если нам необходимо иметь возможность изменять упорядоченную последовательность, то вместо кортежей можно просто использовать списки или, если в программе уже используется кортеж, который нежелательно модифицировать, можно преобразовать кортеж в список с помощью функции преобразования `list()` и затем изменять полученный список.

Тип данных `tuple` может вызываться как функция `tuple()` – без аргументов она возвращает пустой кортеж, с аргументом типа tuple возвращает поверхностную копию аргумента; в случае, если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `tuple`. Эта функция принимает не более одного аргумента. Кроме того, кортежи могут создаваться без использования функции `tuple()`. Пустой кортеж создается с помощью пары пустых круглых скобок `()`, а кортеж, состоящий из одного или более элементов, может быть создан с помощью запятых. Иногда кортежи приходится заключать в круглые скобки, чтобы избежать синтаксической неоднозначности. Например, чтобы передать кортеж `1, 2, 3` в функцию, необходимо использовать такую форму записи: `function((1, 2, 3))`.

Рис. 1: Позиции элементов в кортеже

На рис. 1 показан кортеж `t = ”venus”, - 28, ”green”, ”21”, 19.74` и индексы элементов внутри кортежа. Строки индексируются точно так же, но, если в строках каждой позиции соответствует единственный символ, то в кортежах каждой позиции соответствует единственная ссылка на объект.

Кортежи предоставляют всего два метода: `t.count(x)`, который возвращает количество объектов `x` в кортеже `t`, и `t.index(x)`, который возвращает индекс самого первого (слева) вхождения объекта `x` в кортеж `t` или возбуждает исключение `ValueError`, если объект `x` отсутствует в кортеже. (Эти методы имеются также и у списков.)

Кроме того, кортежи могут использоваться с оператором `+` (конкатенации), * (дублирования) и `[]` (получения среза), а операторы `in` и `not in` могут применяться для проверки на вхождение. Можно использовать также комбинированные операторы присваивания `+=` и `*=`. Несмотря на то, что кортежи являются неизменяемыми объектами, при выполнении этих операторов интерпретатор Python создает за кулисами новый кортеж с результатом операции и присваивает ссылку на него объекту, расположенному слева от оператора, то есть используется тот же самый прием, что и со строками. Кортежи могут сравниваться с помощью стандартных операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`), при этом сравнивание

производится поэлементно (и рекурсивно, при наличии вложенных элементов, таких как кортежи в кортежах).

Рассмотрим несколько примеров получения срезов, начав с извлечения единственного элемента и группы элементов:

```
>>> hair = ”black”, ”brown”, ”blonde”, ”red”
>>> hair[2]
’blonde’
```

```
>>> hair[-3:]  # то же, что и hair[1:]
(’brown’, ’blonde’, ’red’)
```

Эта операция выполняется точно так же, как и в случае со строками, списками или любыми другими последовательностями.

```
>>> hair[:2], ”gray”, hair[2:]
((’black’, ’brown’), ’gray’, (’blonde’, ’red’))
```

Здесь мы попытались создать новый кортеж из 5 элементов, но в результате получили кортеж с тремя элементами, содержащий два двухэлементных кортежа. Это произошло потому, что мы применили оператор запятой к трем элементам (кортеж, строка и кортеж). Чтобы получить единый кортеж со всеми этими элементами, необходимо выполнить конкатенацию кортежей:

```
>>> hair[:2] + (”gray”,) + hair[2:]
(’black’, ’brown’, ’gray’, ’blonde’, ’red’)
```

Чтобы создать кортеж из одного элемента, необходимо поставить запятую, но если запятую просто добавить, будет получено исключение `TypeError` (так как интерпретатор будет думать, что выполняется конкатенация строки и кортежа), поэтому необходимо использовать запятую и круглые скобки.

Коллекции допускают возможность вложения с любой глубиной вложенности. Оператор извлечения срезов `[]` может применяться для доступа к вложенным коллекциям столько раз, сколько это будет необходимо. Например:

```
>>> things = (1, -7.5, (”pea”, (5, ”Xyz”), ”queue”))
>>> things[2][1][1][2]
’z’
```

Кортежи могут хранить элементы любых типов, включая другие коллекции, такие как кортежи и списки, так как на самом деле кортежи хранят ссылки на объекты. Использование сложных, вложенных структур данных, таких, как показано ниже, легко может создавать путаницу. Одно из решений этой проблемы состоит в том, чтобы давать значениям индексов осмысленные имена. Например:

```
>>> MANUFACTURER, MODEL, SEATING = (0, 1, 2)
>>> MINIMUM, MAXIMUM = (0, 1)
>>> aircraft = ("Airbus", "A320-200", (100, 220))
>>> aircraft[SEATING][MAXIMUM]
220
```

В первых двух строках вышеприведенного фрагмента мы выполнили присваивание кортежам. Когда справа от оператора присваивания указывается последовательность (в данном случае – это кортежи), а слева указан кортеж, мы говорим, что последовательность справа *распаковывается*. Операция распаковывания последовательностей может использоваться для организации обмена значений между переменными, например:

```
a, b = (b, a)
```

1.2. Именованные кортежы

Именованные кортежи ведут себя точно так же, как и обычные кортежи, и не уступают им в производительности. Отличаются они возможностью ссылаться на элементы кортежа не только по числовому индексу, но и по имени, что в свою очередь позволяет создавать сложные агрегаты из элементов данных.

В модуле `collections` имеется функция `namedtuple()`. Эта функция используется для создания собственных типов кортежей. Например:

```
import collections
Sale = collections.namedtuple("Sale", "productid customerid date quantity price")
```

Первый аргумент функции `collections.namedtuple()` – это имя создаваемого кортежа. Второй аргумент – это строка имен, разделенных пробелами, для каждого элемента, который будет присутствовать в этом кортеже. Первый аргумент и имена во втором аргументе должны быть допустимыми идентификаторами языка Python. Функция возвращает класс (тип данных), который может использоваться для создания именованных кортежей. Так, в примере выше мы можем интерпретировать имя `Sale` как имя любого другого класса (такого как `tuple`) в языке Python и создавать объекты типа `Sale`. Например:

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))
```

В этом примере мы создали список из двух элементов типа `Sale`, то есть из двух именованных кортежей. Мы можем обращаться к элементам таких кортежей по их индексам – например, обратиться к элементу `price` в первом элементе списка `sales` можно с помощью выражения `sales[0][-1]` (вернет значение `7.99`) – или по именам, которые делают программный код более удобочитаемым:

```
total = 0
for sale in sales:
    total += sale.quantity*sale.price
print("Total {0:.2f}".format(total))
```

Очень часто простоту и удобство, которые предоставляют именованные кортежи, можно обратить на пользу делу. Например, ниже приводится версия примера `aircraft` из предыдущего подраздела, имеющая более аккуратный вид:

```
>>> Aircraft = collections.namedtuple("Aircraft", "manufacturer model seating")
>>> Seating = collections.namedtuple("Seating", "minimum maximum")
>>> aircraft = Aircraft("Airbus", "A320-200", Seating(100, 220))
>>> aircraft.seating.maximum
220
```

1.3. Списки

Список – это упорядоченная последовательность из нуля или более ссылок на объекты. Списки поддерживают тот же синтаксис получения срезов, что и строки с кортежами. Это упрощает извлечение элементов из списка. В отличие от строк и кортежей списки относятся к категории изменяемых объектов, поэтому мы можем замещать или удалять любые их элементы. Кроме того, существует возможность вставлять, замещать и удалять целые срезы списков.

Тип данных `list` может вызываться как функция `list()` – без аргументов она возвращает пустой список, с аргументом типа `list` возвращает поверхностную копию аргумента; в случае, если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `list`. Эта функция принимает не более одного аргумента. Кроме того, списки могут создаваться без использования функции `list()`. Пустой список создается с помощью пары пустых квадратных скобок `[]`, а список, состоящий из одного или более элементов, может быть создан с помощью последовательности элементов, разделенных запятыми, заключенной в квадратные скобки. Другой способ создания списков заключается в использовании генераторов списков – эта тема будет рассматриваться ниже в этом подразделе.

Поскольку все элементы списка в действительности являются ссылками на объекты, списки, как и кортежи, могут хранить элементы любых типов данных, включая коллекции, такие как списки и кортежи. Списки могут сравниваться с помощью стандартных операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`), при этом сравнивание производится поэлементно (и рекурсивно, при наличии вложенных элементов, таких как списки или кортежи в списках).

В результате выполнения операции присваивания `L = [ - 17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]` мы получим список, как показано на рис. 2

К спискам, таким как `L`, мы можем применять оператор извлечения среза, повторяя его столько раз, сколько потребуется для доступа к элементам в списке, как показано ниже:

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k'
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] == 'echo'
L[4][2][1] == L[4][2][-3] == L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

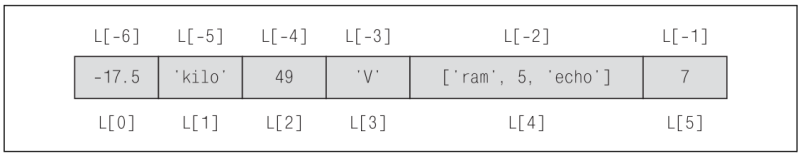

Рис. 2: Позиции элементов в списке

Списки, как и кортежи, могут вкладываться друг в друга; допускают выполнение итераций по их элементам и извлечение срезов. Все примеры с кортежами, которые приводились в предыдущем подразделе, будут работать точно так же, если вместо кортежей в них будут использованы списки. Списки поддерживают операторы проверки на вхождение `in` и `not in`, оператор конкатенации `+`, оператор расширения `+=` (то есть добавляет операнд справа в конец списка) и операторы дублирования `*` и `*=`. Списки могут также использоваться в качестве аргументов функции `len()`.

Кроме того, списки предоставляют методы, перечисленные в табл. 1

Таблица 1: Методы списков

Синтаксис	**Описание**
`L.append(x)`	Добавляет элемент `x` в конец списка `L`
`L.count(x)`	Возвращает число вхождений элемента `x` в список `L`
`L.extend(m)` или `L += m`	Добавляет в конец списка L все элементы итерируемого объекта `m`
`L.index(x, start, end)`	Возвращает индекс самого первого (слева) вхождения элемента `x` в список `L` (или в срез `start:end` списка `L`); в противном случае возбуждает исключение `ValueError`
`L.insert(i, x)`	Вставляет элемент `x` в список `L` в позицию `int i`
`L.pop()`	Удаляет самый последний элемент из списка `L` и возвращает его в качестве результата
`L.pop(i)`	Удаляет из списка `L` элемент с индексом `int i` и возвращает его в качестве результата
`L.remove(x)`	Удаляет самый первый (слева) найденный элемент `x` из списка `L` или возбуждает исключение `ValueError`, если элемент ‘x‘не будет найден ‘
`L.reverse()`	Переставляет в памяти элементы списка в обратном порядке
`L.sort()`	Сортирует список в памяти. Этот метод принимает те же необязательные аргументы `key` и `reverse` что и встроенная функция `sorted()`

Несмотря на то, что для доступа к элементам списка можно использовать оператор извлечения среза, тем не менее в некоторых ситуациях бывает необходимо одновременно извлечь две или более частей списка. Сделать это можно с помощью операции распаковывания последовательности. Любой итерируемый объект (списки, кортежи и другие) может быть распакован с помощью оператора распаковывания «звездочка» (*). Когда слева от оператора присваивания указывается две или более переменных, одна из которых предваряется символом *, каждой переменной присваивается по одному элементу списка, а переменной со звездочкой присваивается оставшаяся часть списка. Ниже приводится несколько примеров выполнения распаковывания списков:

```
>>> first, *rest = [9, 2, -4, 8, 7]
>>> first, rest
(9, [2, -4, 8, 7])
```

```
>>> first, *mid, last = "Charles Philip Arthur George Windsor".split()
>>> first, mid, last
('Charles', ['Philip', 'Arthur', 'George'], 'Windsor')
```

```
>>> *directories, executable = "/usr/local/bin/gvim".split("/")
>>> directories, executable
(['', 'usr', 'local', 'bin'], 'gvim')
```

Когда используется оператор распаковывания последовательности, как в данном примере, выражение `*rest` и подобные ему называются *выражениями со звездочкой*.

В языке Python имеется также похожее понятие *аргументов со звездочкой*. Например, допустим, что имеется следующая функция, принимающая три аргумента:

```
def product(a, b, c):
    return a * b * c
```

тогда мы можем вызывать эту функцию с тремя аргументами или использовать аргументы со звездочкой:

```
>>> product(2, 3, 5)
30
```

```
>>> L = [2, 3, 5]
>>> product(*L)
30
```

```
>>> product(2, *L[1:])
30
```

В первом примере функция вызывается, как обычно, с тремя аргументами. Во втором вызове использован аргумент со звездочкой; в этом случае список из трех элементов распаковывается оператором *, так что функция получает столько аргументов, сколько ей требуется. Того же эффекта можно было бы добиться при использовании кортежа с тремя элементами. В третьем вызове функции первый аргумент передается традиционным способом, а другие два – посредством применения операции распаковывания двухэлементного среза списка `L`.

Имеется возможность выполнять итерации по элементам списка с помощью конструкции `for item in L:`. Если в цикле потребуется изменять элементы списка, то можно использовать следующий прием:

```
for i in range(len(L)):
    L[i] = process(L[i])
```

Замечание.

Встроенная функция `range()` возвращает целочисленный итератор. С одним целочисленным аргументом, `n`, итератор `range()` возвращает последовательность чисел 0, 1, ..., $n-1$.

Этот прием можно использовать для увеличения всех элементов в списке целых чисел. Например:

```
for i in range(len(numbers)):
    numbers[i] += 1
```

Генераторы списков. Небольшие списки часто создаются как литералы, но длинные списки обычно создаются программным способом. Списки целых чисел могут создаваться с помощью выражения `list(range(n))`; когда необходим итератор целых чисел, достаточно функции `range()`; а для создания списков других типов часто используется оператор цикла `for ... in`. Предположим, например, что нам требуется получить список високосных годов в определенном диапазоне. Для начала мы могли бы использовать такой цикл:

```
leaps = []
for year in range(1900, 1940):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        leaps.append(year)
```

Когда функции `range()` передаются два целочисленных аргумента `n` и `m`, итератор возвращает последовательность целых чисел `n`, `n+1`, ..., `m-1`.

Конечно, если диапазон известен заранее, можно было бы использовать литерал списка, например, `leaps = [1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]`.

Генератор списков – это выражение и цикл с дополнительным условием, заключенное в квадратные скобки, в котором цикл используется для создания элементов списка, а условие используется для исключения нежелательных элементов. В простейшем виде генератор списков записывается, как показано ниже:

```
[item for item in iterable]
```

Это выражение вернет список всех элементов объекта `iterable` и семантически ничем не отличается от выражения `list(iterable)`. Интересными генераторы списков делают две особенности – они могут использоваться как выражения и они допускают включение условной инструкции, вследствие чего мы получаем две типичные синтаксические конструкции использования генераторов списков:

```
[expression for item in iterable]
[expression for item in iterable if condition]
```

Вторая форма записи эквивалентна циклу:

```
temp = []
for item in iterable:
    if condition:
       temp.append(expression)
```

Обычно выражение `expression` является либо самим элементом `item`, либо некоторым выражением с его участием. Конечно, генератору списков не требуется временная переменная `temp[]`, которая необходима в версии с циклом `for ... in`.

Теперь можно переписать программный код создания списка високосных годов с использованием генератора списка. Мы сделаем это в три этапа. Сначала создадим список, содержащий все годы в указанном диапазоне:

```
leaps = [y for y in range(1900, 1940)]
```

То же самое можно было бы сделать с помощью выражения `leaps = list(range(1900, 1940))`. Теперь добавим простое условие, которое будет оставлять в списке только каждый четвертый год:

```
leaps = [y for y in range(1900, 1940) if y % 4 == 0]
```

И, наконец, получаем окончательную версию:

```
leaps = [y for y in range(1900, 1940)
         if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

Использование генератора списков в данном случае позволило уменьшить объем программного кода с четырех строк до двух – не так много, но в крупных проектах суммарная экономия может оказаться весьма существенной.

Так как генераторы списков воспроизводят списки, то есть итерируемые объекты, и сами генераторы списков используют итерируемые объекты, имеется возможность вкладывать генераторы списков друг в друга. Это эквивалентно вложению циклов `for ... in`. Например, если бы нам потребовалось сгенерировать список всех возможных кодов одежды для разных полов, разных размеров и расцветок, но исключая одежду для полных женщин, нужды и чаянья которых индустрия моды нередко игнорирует, мы могли бы использовать вложенные циклы `for ... in`, как показано ниже:

```
codes = []
for sex in "MF":
    for size in "SMLX":
        if sex == "F" and size
       continue
        for color in "BGW":
       codes.append(sex + size + color)
```

Этот фрагмент воспроизводит список, содержащий 21 элемент – `['MSB', 'MSG', ..., 'FLW']`. Тот же самый список можно создать парой строк, если воспользоваться генераторами списков:

```
codes = [s + z + c for s in "MF" for z in "SMLX" for c in "BGW"
         if not (s == "F" and z == "X")]
```

Здесь каждый элемент списка воспроизводится выражением `s + z + c`. Кроме того, в генераторе списков несколько иначе построена логика обхода нежелательной комбинации пол/размер – проверка выполняется в самом внутреннем цикле, тогда как в версии с циклами `for ... in` эта проверка выполняется в среднем цикле. Любой генератор списков можно переписать, используя один или более циклов `for ... in`.

2. Множества

Тип `set` – это разновидность коллекций, которая поддерживает оператор проверки на вхождение `in`, функцию `len()` и относится к разряду итерируемых объектов. Кроме того, множества предоставляют метод `set.isdisjoint()` и поддерживают операторы сравнения и битовые операторы (которые в контексте множеств используются для получения объединения, пересечения и т. д.). В языке Python имеется два встроенных типа множеств: изменяемый тип `set` и неизменяемый `frozenset`. При переборе элементов множества элементы могут следовать в произвольном порядке.

В состав множеств могут включаться только *хешируемые* объекты. Хешируемые объекты – это объекты, имеющие специальный метод `__hash__()`, на протяжении всего жизненного цикла объекта всегда возвращающий одно и то же значение, которые могут участвовать в операциях сравнения на равенство посредством специального метода `__eq__()`. (Специальные методы – это методы, имена которых начинаются и оканчиваются двумя символами подчеркивания).

Все встроенные неизменяемые типы данных, такие как `float`, `frozenset`, `int`, `str` и `tuple`, являются хешируемыми объектами и могут добавляться во множества. Встроенные изменяемые типы данных, такие как `dict`, `list` и `set`, не являются хешируемыми объектами, так как значение хеша в каждом конкретном случае зависит от содержащихся в объекте элементов, поэтому они не могут добавляться в множества.

Множества могут сравниваться между собой с использованием стандартных операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`). Обратите внимание: операторы `==` и `!=` имеют обычный смысл, и сравнение выполняется путем поэлементного сравнения (или рекурсивно при наличии таких вложенных элементов, как кортежи и фиксированные множества (`frozenset`)), но остальные операторы сравнения выполняют сравнение подмножеств и надмножеств, как вскоре будет показано.

2.1. Тип `set`

Тип `set` – это неупорядоченная коллекция из нуля или более ссылок на объекты, указывающих на хешируемые объекты. Множества относятся к категории изменяемых типов, поэтому легко можно добавлять и удалять их элементы, но, так как они являются неупорядоченными коллекциями, к ним не применимо понятие индекса и не применима операция

извлечения среза. На рис. 3 иллюстрируется множество, созданное следующим фрагментом программного кода:

```
S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}
```

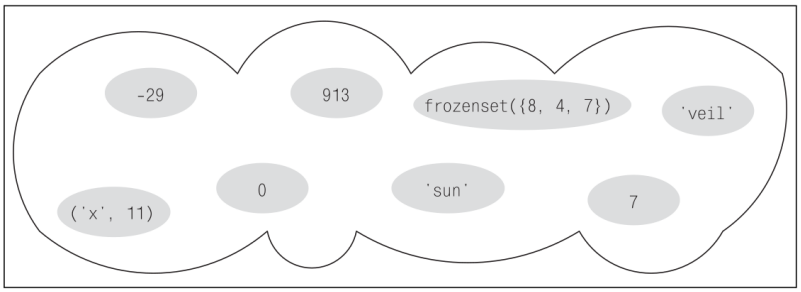

Рис. 3: Множество – это неупорядоченная коллекция уникальных элементов

Тип данных `set` может вызываться как функция `set()` – без аргументов она возвращает пустое множество; с аргументом типа `set` возвращает поверхностную копию аргумента; в случае, если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `set`. Эта функция принимает не более одного аргумента. Кроме того, непустые множества могут создаваться без использования функции `set()`, а пустые множества могут создаваться только с помощью функции `set()` – их нельзя создать с помощью пары пустых скобок. Множество, состоящее из одного или более элементов, может быть создано с помощью последовательности элементов, разделенных запятыми, заключенной в фигурные скобки. Другой способ создания множеств заключается в использовании генераторов множеств. Множества всегда содержат уникальные элементы – добавление повторяющихся элементов возможно, но не имеет смысла. Например, следующие три множества являются эквивалентными: `set("apple")`, `set("aple")` и `{'e', 'p', 'a', 'l'}`. Благодаря этой их особенности множества часто используются для устранения повторяющихся значений. Например, если предположить, что `x` – это список строк, то после выполнения инструкции `x = list(set(x))` в списке останутся только уникальные строки, причем располагаться они могут в произвольном порядке.

Множества поддерживают встроенную функцию `len()` и быструю проверку на вхождение с помощью операторов `in` и `not in`. Они также предоставляют типичный набор операторов, как показано на рис. . Полный перечень методов и операторов, применимых к множествам, приводится в табл. 2. Все методы семейства «update» (`set.update()`, `set.intersection_update()` и т. д.) могут принимать в качестве аргумента любые итерируемые объекты, но эквивалентные им комбинированные операторы присваивания (`|=`, `&=` и т. д.) требуют, чтобы оба операнда были множествами.

Таблица 2: Методы и операторы множеств

Синтаксис	**Описание**

`s.add(x)`	Добавляет элементы `x` во множество `s`, если они отсутствуют в `s`
`s.clear()`	Удаляет все элементы из множества `s`
‘s.difference(t)‘или `s-t`	Возвращает новое множество включающее элементы множества `s`, которые отсутствуют в множестве `t`
`s.difference_update(t)` или `s-=t`	Удаляет из множества `s` все элементы, присутствующие в множестве `t`
`s.discard(x)`	Удаляет элемент `x` из множества `s`, если он присутствует в множестве `s`
`s.intersection(t)` или `s & t`	Возвращает новое множество, включающее элементы, присутствующие одновременно в множествах `s` и `t`
`s.intersection_up` или `s &= t`	Оставляет во множестве `s` пересечение множеств `s` и `t`
`s.isdisjoint(t)`	Возвращает `True`, если множества `s` и `t` не имеют общих элементов
`s.issubset(t)` или `s <= t`	Возвращает `True`, если множество `s` эквивалентно множеству `t` или является его подмножеством; чтобы проверить, является *ли* множество `s` только подмножеством множества `t`, следует использовать проверку `s < t`
`s.issuperset(t)` или `s >= t`	Возвращает `True`, если множество `s` эквивалентно множеству `t` или является его надмножеством; чтобы проверить, является *ли* множество `s` только надмножеством множества `t`, следует использовать проверку `s > t`
`s.pop()`	Возвращает и удаляет случайный элемент множества `s` или возбуждает исключение `KeyError`, если `s` – это пустое множество
`s.remove(x)`	Удаляет элемент `x` из множества `s` или возбуждает исключение `KeyError`, если элемент `x` отсутствует в множестве `s`
`s.symmetric_diffe` или `s ^ t`	Возвращает новое множество, включающее все элементы, присутствующие в множествах `s` и `t`, за исключением элементов, присутствующих в обоих множествах одновременно
`s.symmetric_difference_update(t)` или `s ^= t`	Возвращает во множестве `s` результат строгой дизъюнкции множеств `s` и `t`
`s.union(t)` или `s | t`	Возвращает новое множество, включающее все элементы множества `s` и все элементы множества `t`, отсутствующие в множестве `s`
`s.update(t)` или `s |= t`	Добавляет во множество `s` все элементы множества `t`, отсутствующие в множестве `s`

Генераторы множеств. В дополнение к возможности создавать множества с помощью функции `set()` или *литералов*, существует возможность создавать множества с помощью *ге-*

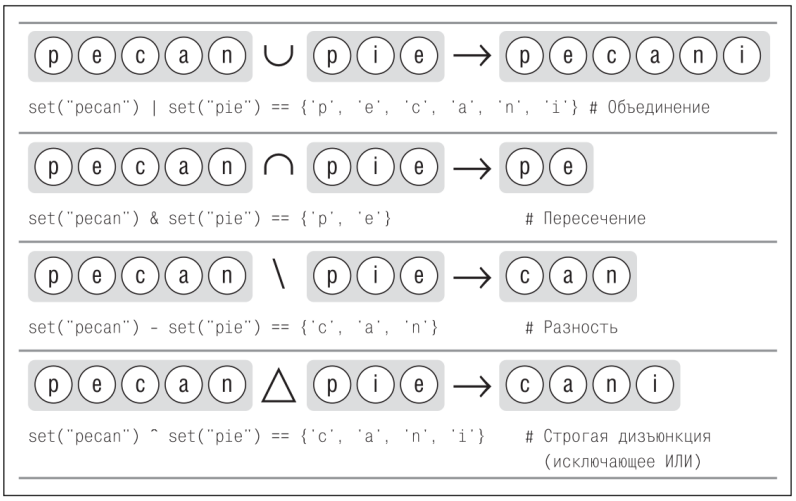

Рис. 4: Стандартные операторы множеств

нераторов множеств. Генератор множества – это выражение и цикл с необязательным условием, заключенные в фигурные скобки. Подобно генераторам списков, генераторы множеств поддерживают две формы записи:

```
{expression for item in iterable}
{expression for item in iterable if condition}
```

Мы могли бы использовать генераторы множеств для фильтрации нежелательных элементов (когда порядок следования элементов не имеет значения), как показано ниже:

```
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

Если предположить, что `files` – это список имен файлов, то данный генератор множества создает множество `html`, в котором хранятся только имена файлов с расширениями `.htm` и `.html`, независимо от регистра символов.

Как и в случае с генераторами списков, в генераторах множеств используются итерируемые объеты, которые в свою очередь могут быть генераторами множеств (или генераторами любого другого типа), что позволяет создавать весьма замысловатые генераторы множеств.

2.2. Тип `frozenset`

Фиксированное множество (`frozenset`) – это множество, которое после создания невозможно изменить. Хотя при этом мы, конечно, можем повторно связать переменную, которая ссылалась на фиксированное множество, с чем-то другим. Фиксированные множества могут создаваться только в результате обращения к имени типа `frozenset` как к функции. При

вызове `frozenset()` без аргументов возвращается пустое фиксированное множество; с аргументом типа `frozenset` возвращается поверхностная копия аргумента; если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `frozenset`. Эта функция принимает не более одного аргумента.

Поскольку фиксированные множества относятся к категории неизменяемых объектов, они поддерживают только те методы и операторы, которые воспроизводят результат, не оказывая воздействия на фиксированное множество или на множества, к которым они применяются.

Поскольку фиксированные множества относятся к категории неизменяемых объектов, они поддерживают только те методы и операторы, которые воспроизводят результат, не оказывая воздействия на фиксированное множество или на множества, к которым они применяются. В табл. 2 перечислены все методы множеств из которых фиксированными множествами поддерживаются: `frozenset.copy()`, `frozenset.difference()` (`-`), `frozenset.intersection()` (`&`), `frozenset.isdis-joint()`, `frozenset.issubset()` (`<=` и `<` для выявления подмножеств), `frozenset.issuperset()` (`>=` и `>` для выявления надмножеств), `frozenset.union()` (`|`) и `frozenset.symmetric_difference()` (`^`).

Если двухместный оператор применяется ко множеству и фиксированному множеству, тип результата будет совпадать с типом операнда, стоящего слева от оператора. То есть если предположить, что `f` – это фиксированное множество, а `s` – это обычное множество, то выражение `f & s` вернет объект типа frozenset, а выражение `s & f` – объект типа `set`. В случае операторов `==` и `!=` порядок операндов не имеет значения, и выражение `f == s` вернет `True`, только если оба множества содержат одни и те же элементы.

Другое следствие неизменности фиксированных множеств заключается в том, что они соответствуют критерию хеширования, предъявляемому к элементам множеств, и потому множества и фиксированные множества могут содержать другие фиксированные множества.

3. Отображения

Отображениями называются типы данных, поддерживающие оператор проверки на вхождение (`in`), функцию `len()` и возможность обхода элементов в цикле. Отображения – это коллекции пар элементов «ключ-значение», которые предоставляют методы доступа к элементам и их ключам и значениям. При выполнении итераций порядок следования элементов отображений может быть произвольным. В языке Python имеется два типа отображений: встроенный тип `dict` и тип `collections.defaultdict`, определяемый в стандартной библиотеке. Мы будем использовать термин словарь для ссылки на любой из этих типов, когда различия между ними не будут иметь никакого значения.

В качестве ключей словарей могут использоваться только хешируемые объекты, поэтому в качестве ключей словаря такие неизменяемые типы, как `float`, `frozenset`, `int`, `str` и `tuple`, использовать допускается, а изменяемые типы, такие как `dict`, `list` и `set`, – нет. С другой стороны, каждому ключу соответствует некоторое значение, которое может быть ссылкой на объект любого типа, включая числа, строки, списки, множества, словари, функции и т. д.

Словари могут сравниваться с помощью стандартных операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`), при этом сравнивание производится поэлементно (и рекурсивно, при наличии вложенных элементов, таких как кортежи или словари в словарях). Пожалуй, единственными

операторами сравнения, применение которых к словарям имеет смысл, являются операторы == и !=.

3.1. Словари

Тип dict – это неупорядоченная коллекция из нуля или более пар «ключ-значение», в которых в качестве ключей могут использоваться ссылки на хешируемые объекты, а в качестве значений – ссылки на объекты любого типа. Словари относятся к категории изменяемых типов, поэтому легко можно добавлять и удалять их элементы, но так как они являются неупорядоченными коллекциями, к ним не применимо понятие индекса и не применима операция извлечения среза.

Тип данных dict может вызываться как функция dict() – без аргументов она возвращает пустой словарь; если в качестве аргумента передается отображение, возвращается словарь, основанный на этом отображении: например, с аргументом типа dict возвращается поверхностная копия словаря. Существует возможность передавать в качестве аргумента последовательности, если каждый элемент последовательности в свою очередь является последовательностью из двух объектов, первый из которых используется в качестве ключа, а второй – в качестве значения. Как вариант, для создания словарей, в которых ключи являются допустимыми идентификаторами языка Python, можно использовать именованные аргументы; тогда имена аргументов будут играть роль ключей, а значения аргументов – роль значений ключей. Кроме того, словари могут создаваться с помощью фигурных скобок – пустые скобки {} создадут пустой словарь. Непустые фигурные скобки должны содержать один или более элементов, разделенных запятыми, каждый из которых состоит из ключа, символа двоеточия и значения. Еще один способ создания словарей заключается в использовании генераторов словарей – эта тема будет рассматриваться ниже, в соответствующем подразделе.

Ниже приводятся несколько способов создания словарей – все они создают один и тот же словарь:

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})
d2 = dict(id=1948, name="Washer", size=3)
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))
d5 = {"id": 1948, "name": "Washer", "size": 3}
```

На рис. 5 демонстрируется словарь, созданный следующим фрагментом программного кода:

```
d = {"root": 18, "blue": [75, "R", 2], 21: "venus", -14: None,
     "mars": "rover", (4, 11): 18, 0: 45}
```

Ключи словарей являются уникальными, поэтому если в словарь добавляется пара «ключ-значение» с ключом, который уже присутствует в словаре, в результате происходит замена значения существующего ключа новым значением. Для доступа к отдельным элементам используются квадратные скобки: например, выражение d["root"] вернет 18, выражение d[21] вернет строку "venus", а выражение d[91] применительно к словарю, изображенному на рис. 5, возбудит исключение KeyError.

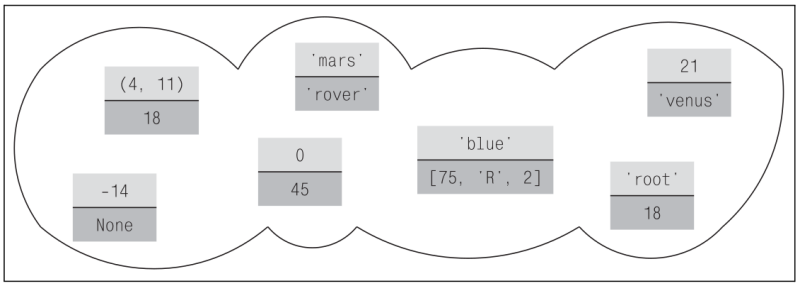

Рис. 5: Словарь – это неупорядоченная коллекция элементов (ключ, значение) с уникальными ключами

Квадратные скобки могут также использоваться для добавления и удаления элементов словаря. Чтобы добавить новый элемент, используется оператор `=`, например, `d["X"] = 59`. Для удаления элементов используется инструкция `del`, например, инструкция `del d["mars"]` удалит из словаря элемент с ключом `"mars"` или возбудит исключение `KeyError`, если элемент с указанным ключом отсутствует в словаре. Кроме того, элементы могут удаляться (и возвращаться вызывающей программе) методом `dict.pop()`.

Словари поддерживают встроенную функцию `len()` и для ключей поддерживают возможность быстрой проверки на вхождение с помощью операторов `in` и `not in`. В табл. 3 перечислены все методы словарей.

Так как словари содержат пары «ключ-значение», у нас может возникнуть потребность обойти в цикле элементы словаря (ключ, значение) по значениям или по ключам. Например, ниже приводятся два эквивалентных способа обхода пар «ключ-значение»:

```
for item in d.items():
    print(item[0], item[1])
```

```
for key, value in d.items():
    print(key, value)
```

Таблица 3: Методы словарей

Синтаксис	**Описание**
`d.clear()`	Удаляет все элементы из словаря `d`
`d.copy()`	Возвращает поверхностную копию словаря `d`
`d.fromkeys(s, v)`	Возвращает словарь типа `dict`, ключами которого являются элементы последовательности `s` значениями либо `None` либо `v`, если аргумент `v` определен
`d.get(k)`	Возвращает значение ключа `k` или `None`, если ключ `k` отсутствует в словаре
`d.get(k, v)`	Возвращает значение ключа `k` или `v`, если ключ `k` отсутствует в словаре

`d.items()`	Возвращает представление всех пар (ключ, значение) в словаре `d`
`d.keys()`	Возвращает представление всех ключей словаря `d`
`d.pop(k)`	Возвращает значение ключа `k` и удаляет из словаря элемент с ключом `k` или возбуждает исключение `KeyError`, если ключ `k` отсутствует в словаре
`d.pop(k, v)`	Возвращает значение ключа `k` и удаляет из словаря элемент с ключом `k` или возвращает значение `v`, если ключ `k` отсутствует в словаре
`d.popitem()`	Возвращает и удаляет произвольную пару (ключ, значение) из словаря `d` или возбуждает исключение `KeyError`, если словарь `d` пуст
`d.setdefault(k, v)`	То же что и `dict.get()` за исключением того, что, если ключ `k` в словаре отсутствует, в словарь вставляется новый элемент с ключом `k` и со значением `None` или `v`, если аргумент `v` задан
`d.update(a)`	Добавляет в словарь `d` пары (ключ, значение) из `a`, которые отсутствуют в словаре `d` а для каждого ключа который уже присутствует в словаре `d` выполняется замена соответствующим значением из `a`; а может быть словарем итерируемым объектом с парами (ключ значение) или именованными аргументами
`d.values()`	Возвращает представление всех значений в словаре `d`

Обход значений в словаре выполняется похожим способом:

```
for value in d.values():
    print(value)
```

Для обхода ключей в словаре можно использовать метод `dict.keys()` или просто интерпретировать словарь как итерируемый объект и вы- полнить итерации по его ключам, как показано в следующих двух фрагментах:

```
for key in d:
    print(key)
```

```
for key in d.keys():
    print(key)
```

Если необходимо изменить значения в словаре, то можно выполнить обход ключей словаря в цикле и изменить значения, используя оператор квадратных скобок. Например, ниже показано, как можно было бы увеличить все значения в словаре `d`, если предполагать, что все значения являются числами:

```
for key in d:
    d[key] += 1
```

Методы `dict.items()`, `dict.keys()` и `dict.values()` возвращают представления словарей. Представление словаря – это в действительности итерируемый объект, доступный только для чтения и хранящий элементы, ключи или значения словаря в зависимости от того, какое представление было запрошено.

Генераторы словарей. *Генератор словарей* – это выражение и цикл с необязательным условием, заключенное в фигурные скобки, очень напоминающее генератор множеств. Подобно генераторам списков и множеств, генераторы словарей поддерживают две формы записи:

```
{keyexpression: valueexpression for key, value in iterable}
{keyexpression: valueexpression for key, value in iterable if condition}
```

Ниже показано, как можно использовать генератор словарей для создания словаря, в котором каждый ключ является именем файла в текущем каталоге, а каждое значение – это размер файла в байтах:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")}
```

Функция `os.listdir()` из модуля os («operating system» – операционная система) возвращает список файлов и каталогов в указанном каталоге, но при этом в список никогда не включаются специальные имена каталогов «.» или «..». Функция os.path.getsize() возвращает размер заданного файла в байтах. Чтобы отфильтровать каталоги и другие элементы списка, не являющиеся файлами, можно добавить дополнительное условие:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")
          if os.path.isfile(name)}
```

Функция `os.path.isfile()` из модуля `os.path` возвращает `True`, если указанный путь соответствует файлу, и `False` – в противном случае, то есть для каталогов, ссылок и тому подобного.

Генераторы словарей могут также использоваться для создания инвер- тированных словарей. Например, пусть имеется словарь `d`, тогда мы можем создать новый словарь, ключами которого будут значения словаря `d`, а значениями – ключи словаря `d`:

```
inverted_d = {v: k for k, v in d.items()}
```

Полученный словарь можно инвертировать обратно и получить первоначальный словарь – при условии, что все значения в первоначальном словаре были уникальными, однако инверсия будет терпеть неудачу, с возбуждением исключения `TypeError`, если какое-либо значение окажется не хешируемым.

3.2. Словари со значениями по умолчанию

Словари со значениями по умолчанию – это обычные словари, они поддерживают те же самые методы и операторы, что и обычные словари. Единственное, что отличает такие словари от обычных словарей, – это способ обработки отсутствующих ключей, но во всех остальных отношениях они ничем не отличаются друг от друга.

При обращении к несуществующему («отсутствующему») ключу слова- ря возбуждается исключение `KeyError`. Это очень удобно, так как нередко для нас бывает желательно знать об отсутствии ключа, который, согласно нашим предположениям, может присутствовать. Но в некоторых случаях бывает необходимо, чтобы в словаре присутствовали все ключи, которые мы используем, даже если это означает, что элемент с заданным ключом добавляется в словарь в момент первого обращения к нему.

Например, допустим, что имеется словарь `d`, который не имеет элемента с ключом m, тогда выражение `x = d[m]` возбудит исключение `KeyError`. Если `d` – это словарь со значениями по умолчанию, созданный соответствующим способом, а элемент с ключом m принадлежит такому словарю, то при обращении к нему будет возвращено соответствующее значение, как и в случае с обычным словарем. Но если в словаре со значениями по умолчанию отсутствует ключ m, то будет создан новый элемент словаря с ключом m и со значением по умолчанию, и будет возвращено значение этого, вновь созданного элемента.

4. Обход в цикле и копирование коллекций

После того как будет создана коллекция элементов данных, вполне естественно возникает желание обойти все элементы, содержащиеся в ней. Еще одна часто выполняемая операция – копирование коллекций. Из-за того, что в языке Python повсеместно используются ссылки на объекты (ради повышения эффективности), существуют некоторые особенности, связанные с копированием.

В этом разделе сначала мы рассмотрим итераторы языка Python, а затем принципы копирования коллекций.

4.1. Итераторы, функции и операторы для работы с итерируемыми объектами

Итерируемый тип данных – это такой тип, который может возвращать свои элементы по одному. Любой объект, имеющий метод `__iter__()`, или любая последовательность (то есть объект, имеющий метод `__getitem__()`, принимающий целочисленный аргумент со значением от `0` и выше), является итерируемым и может предоставлять итератор. Итератор – это объект, имеющий метод `__next__()`, который при каждом вызове возвращает очередной элемент и возбуждает исключение `StopIteration` после исчерпания всех элементов. В табл. 4 перечислены операторы и функции, которые могут применяться к итерируемым объектам.

Таблица 4: Общие функции и операторы для работы с итерируемыми объектами

Синтаксис	Описание

`s + t`	Возвращает конкатенацию последовательностей `s` и `t`
`s * n`	Возвращает конкатенацию из `int n` последовательностей `s`
`x in i`	Возвращает `True`, если элемент `x` присутствует в итерируемом объекте `i`, обратная проверка выполняется с помощью оператора `not in`
`all(i)`	Возвращает `True`, если все элементы итерируемого объекта `i` в логическом контексте оцениваются как значение `True`
`any(i)`	Возвращает `True`, если хотя бы один элемент итерируемого объекта `i` в логическом контексте оценивается как значение `True`
`enumerate (i,start)`	Обычно используется в циклах `for ... in`, чтобы получить последовательность кортежей `(index, item)`, где значения индексов начинают отсчитывать от `0` или от значения `start`
`len(x)`	Возвращает «длину» объекта `x`. Если `x` – коллекция, то возвращаемое число представляет количество элементов. Если `x` – строка, то возвращаемое число представляет количество символов
`max(i, key)`	Возвращает наибольший элемент в итерируемом объекте `i` или элемент с наибольшим значением `key(item)`, если функция `key` определена
`min(i, key)`	Возвращает наименьший элемент в итерируемом объекте `i` или элемент с наименьшим значением `key(item)`, если функция `key` определена
`range(start, stop, step)`	Возвращает целочисленный итератор. С одним аргументом (`stop`) итератор представляет последовательность целых чисел от `0` до `stop-1`, с двумя аргументами (`start`, `stop`) – последовательность целых чисел от `start` до `stop-1`, с тремя аргументами – последовательность целых чисел от `start` до `stop-1` с шагом `step`
`reversed(i)`	Возвращает итератор, который будет возвращать элементы итератора `i` в обратном порядке
`sorted(i, key, reverse)`	Возвращает список элементов итератора `i` в отсортированном порядке. Аргумент `key` используется для выполнения сортировки DSU (Decorate, Sort, Undecorate – декорирование, сортировка, обратное декорирование). Если аргумент `reverse` имеет значение `True`, сортировка выполняется в обратном порядке
`sum(i, start)`	Возвращает сумму элементов итерируемого объекта `i`, плюс ар гумент `start` (значение которого по умолчанию равно `0`). Объект `i` не должен содержать строк
`zip(i1, ..., iN)`	Возвращает итератор кортежей, используя итераторы от `i1` до `iN`

Порядок, в котором возвращаются элементы, зависит от итерируемого объекта. В случае списков и кортежей элементы обычно возвращаются в предопределенном порядке, начиная с первого элемента (находящегося в позиции с индексом `0`), но другие итераторы возвращают элементы в произвольном порядке – например, итераторы словарей и множеств.

Встроенная функция `iter()` используется двумя совершенно различными способами. Применяемая к коллекции или к последовательности, она возвращает итератор для заданного объекта или возбуждает исключение `TypeError`, если объект не является итерируемым. Такой способ часто используется при работе с нестандартными типами коллекций и крайне редко – в других контекстах. Во втором варианте использования функции `iter()` ей передается вызываемый объект (функция или метод) и специальное значение. В этом случае полученная функция или метод вызывается на каждой итерации, а значение этой функции, если оно не равно специальному значению, возвращается вызывающей программе; в противном случае возбуждается исключение `StopIteration`.

Когда в программе используется цикл `for item in iterable`, интерпретатор Python вызывает функцию `iter(iterable)`, чтобы получить итератор. После этого на каждой итерации вызывается метод `__next__()` итератора, чтобы получить очередной элемент, а когда возбуждается исключение `StopIteration`, оно перехватывается и цикл завершается. Другой способ получить очередной элемент итератора состоит в том, чтобы вызвать встроенную функцию `next()`. Ниже приводятся два эквивалентных фрагмента программного кода (оба они вычисляют произведение элементов списка), в одном из них используется цикл `for ... in`, а во втором явно используется итератор:

```
product = 1
for i in [1, 2, 4, 8]:
    product *= i
print(product)
```

```
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= next(i)
    except StopIteration:
        break
print(product)
```

Любой (конечный) итерируемый объект `i` может быть преобразован в кортеж вызовом функции `tuple(i)` или в список – вызовом функции `list(i)`.

4.2. Копирование коллекций

Поскольку в языке Python повсюду используются ссылки на объекты, когда выполняется оператор присваивания (`=`), никакого копирования данных на самом деле не происходит. Если

справа от оператора находится литерал, например, строка или число, в операнд слева записывается ссылка, которая указывает на объект в памяти, хранящий значение литерала. Если справа находится ссылка на объект, в левый операнд записывается ссылка, указывающая на тот же самый объект, на который ссылается правый операнд. Вследствие этого операция присваивания обладает чрезвычайно высокой скоростью выполнения.

Когда выполняется присваивание крупной коллекции, такой как длинный список, экономия времени становится более чем очевидной. Например:

```
>>> songs = [”Because”, ”Boys”, ”Carol”]
>>> beatles = songs
>>> beatles, songs
(['Because', 'Boys', 'Carol'], ['Because', 'Boys', 'Carol'])
```

Здесь была создана новая ссылка на объект (`beatles`), и обе ссылки указывают на один и тот же список – никакого копирования данных не производилось.

Поскольку списки относятся к категории изменяемых объектов, мы можем вносить в них изменения. Например:

```
>>> beatles[2] = ”Cayenne”
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Cayenne'])
```

Изменения были внесены с использованием переменной `beatles`, но это всего лишь ссылка, указывающая на тот же самый объект, что и ссылка songs. Поэтому любые изменения, произведенные с использованием одной ссылки, можно наблюдать с использованием другой ссылки. Часто это именно то, что нам требуется, поскольку копирование крупных коллекций может оказаться дорогостоящей операцией. Кроме того, это также означает, что имеется возможность передавать списки или другие изменяемые коллекции в виде аргументов функций, изменять эти коллекции в функциях и пребывать в уверенности, что изменения будут доступны после того, как функция вернет управление вызывающей программе.

Однако в некоторых ситуациях действительно бывает необходимо создать отдельную копию коллекции (то есть создать другой изменяемый объект). В случае последовательностей, когда выполняется оператор извлечения среза, например, `songs[:2]`, полученный срез – это всегда независимая копия элементов. Поэтому скопировать последовательность целиком можно следующим способом:

```
>>> songs = [”Because”, ”Boys”, ”Carol”]
>>> beatles = songs[:]
>>> beatles[2] = ”Cayenne”
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Carol'])
```

В случае словарей и множеств копирование можно выполнить с помощью методов `dict.copy()` и `set.copy()`. Кроме того, в модуле copy имеется функция copy.copy(), которая возвращает копию заданного объекта. Другой способ копирования встроенных типов коллекций заключается в использовании имени типа как функции, которой в качестве аргумента передается копируемая коллекция. Например:

```
copy_of_dict_d = dict(d)
copy_of_list_L = list(L)
copy_of_set_s = set(s)
```

Обратите внимание, что все эти приемы копирования создают *поверхностные копии*, то есть копируются только ссылки на объекты, но не сами объекты. Для неизменяемых типов данных, таких как числа и строки, это равносильно копированию (за исключением более высокой эффективности), но для изменяемых типов данных, таких как вложенные коллекции, это означает, что ссылки в оригинальной коллекции и в копии будут указывать на одни и те же объекты. Эту особенность иллюстрирует следующий пример:

```
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = x[:] # поверхностное копирование
>>> x, y
([53, 68, ['A', 'B', 'C']], [53, 68, ['A', 'B', 'C']])
```

```
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['Q', 'B', 'C']])
```

Когда выполняется поверхностное копирование списка `x`, копируется ссылка на вложенный список `["A", "B", "C"]`. Это означает, что третий элемент в обоих списках, `x` и `y`, ссылается на один и тот же список, поэтому любые изменения, произведенные во вложенном списке, можно наблюдать с помощью любой из ссылок, `x` или `y`. Если действительно необходимо создать абсолютно независимую копию коллекции с произвольной глубиной вложенности, необходимо выполнить глубокое копирование:

```
>>> import copy
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = copy.deepcopy(x)
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['A', 'B', 'C']])
```

Здесь списки `x` и `y`, а также элементы, которые они содержат, полностью независимы.