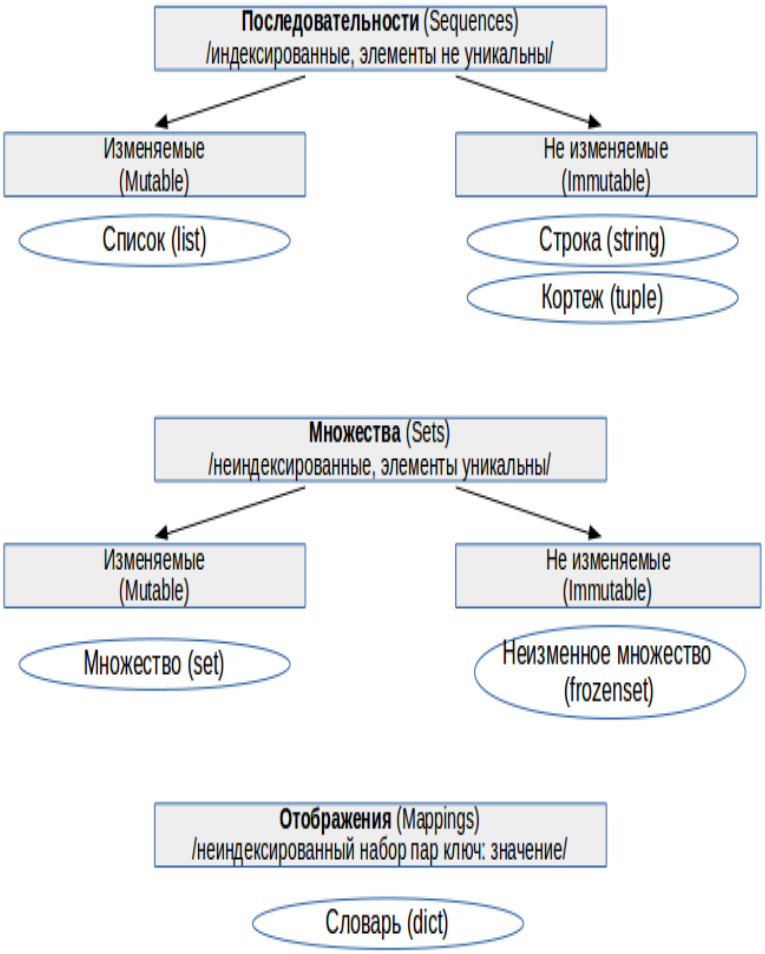
Python: коллекции, часть 1/4: классификация, общие подходы и методы, конвертация

https://habr.com/ru/post/319164/



Коллекция в Python — программный объект (переменная-контейнер), хранящая набор значений одного или различных типов, позволяющий обращаться к этим значениям, а также применять специальные функции и методы, зависящие от типа коллекции.

Частая проблема при изучении коллекций заключается в том, что разобрав каждый тип довольно детально, обычно потом не уделяется достаточного внимания разъяснению картины в целом, не проводятся чёткие сходства и различия между типами, не показывается как одну и туже задачу решать для каждой из коллекций в сравнении.

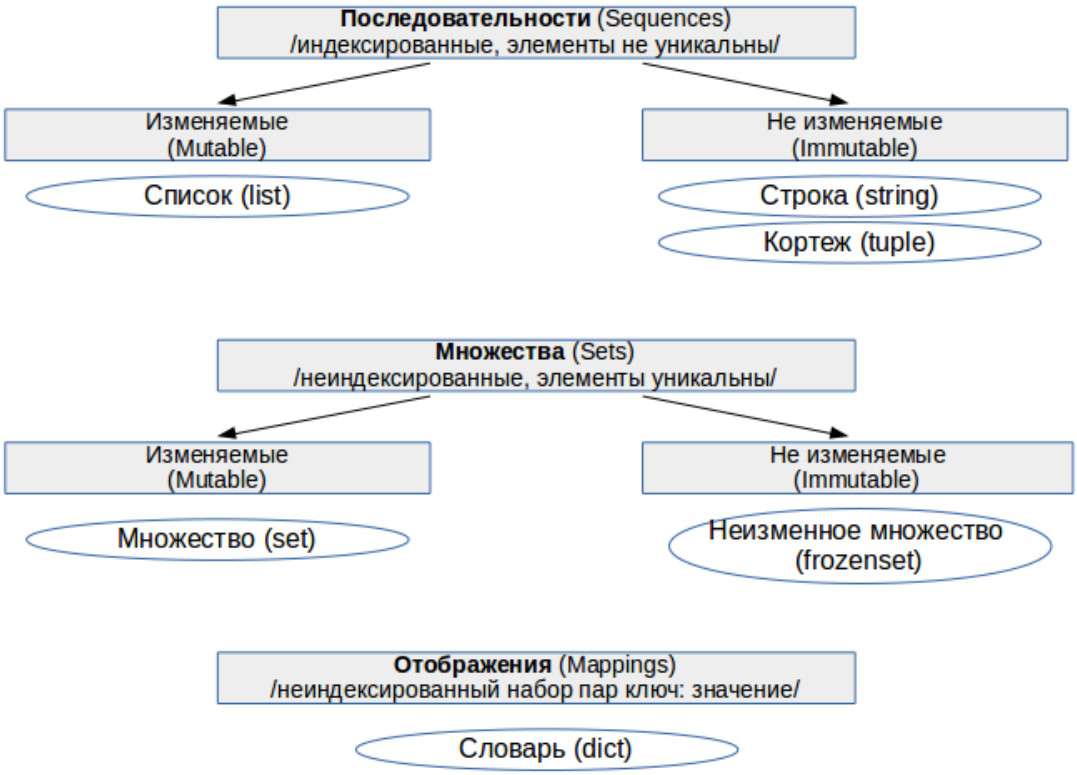
Вот именно эту проблему я хочу попытаться решить в данном цикле статей – рассмотреть ряд подходов к работе со стандартными коллекциями в Python в сравнении между коллекциями разных типов, а не по отдельности, как это обычно показывается в обучающих материалах. Кроме того, постараюсь затронуть некоторые моменты, вызывающие сложности и ошибки у начинающих.

Для кого: для изучающих Python и уже имеющих начальное представление о коллекциях и работе с ними, желающих систематизировать и углубить свои знания, сложить их в целостную картину.

Будем рассматривать стандартные встроенные коллекционные типы данных в Python: список (list), кортеж (tuple), строку (string), множества (set, frozenset), словарь (dict). Коллекции из модуля collections рассматриваться не будут, хотя многое из статьи должно быть применимым и при работе с ними.

ОГЛАВЛЕНИЕ:

1. Классификация коллекций



Обобщение свойств встроенных коллекций в сводной таблице:

Тип коллекции	Изменяемость	Индексированность	Уникальность	Как создаём
Список (list)	+	+	-	*[ ]* *list( )*
Кортеж (tuple)	-	+	-	*( ) ,* *tuple( )*
Строка (string)	-	+	-	*''* *" "*
Множество (set)	+	-	+	*{elm1, elm2}* *set( )*
Неизменное множество (frozenset)	-	-	+	*frozenset( )*
Словарь (dict)	+ элементы - ключи + значения	-	+ элементы + ключи - значения	*{ }* *{key: value,}* *dict()*

Пояснения терминологии:

Индексированность – каждый элемент коллекции имеет свой порядковый номер — индекс. Это позволяет обращаться к элементу по его порядковому индексу, проводить слайсинг («нарезку») — брать часть коллекции выбирая исходя из их индекса. Детально эти вопросы будут рассмотрены в дальнейшем в отдельной статье.

Уникальность – каждый элемент коллекции может встречаться в ней только один раз. Это порождает требование неизменности используемых типов данных для каждого элемента, например, таким элементом не может быть список.

Изменяемость коллекции — позволяет добавлять в коллекцию новых членов или удалять их после создания коллекции.

Примечание для словаря (dict):

сам словарь изменяем — можно добавлять/удалять новые пары ключ: значение;
значения элементов словаря — изменяемые и не уникальные;
а вот ключи — не изменяемые и уникальные, поэтому, например, мы не можем сделать ключом словаря список, но можем кортеж. Из уникальности ключей, так же следует уникальность элементов словаря — пар ключ: значение.

UPD: Важное замечание от sakutylev: Для того, чтобы объект мог быть ключом словаря, он должен быть хешируем. У кортежа, возможен случай, когда его элемент является не хешируемым объектом, и соответственно сам кортеж тогда тоже не является хешируемым и не может выступать ключом словаря.

```
a = (1, [2, 3], 4)
print(type(a))   # <type 'tuple'>
b = {a: 1}       # TypeError: unhashable type: 'list'
```

UPD: Благодарю morff за внимательность — *{}* без значений создают словарь, а со значениями, в зависимости от синтаксиса могут создавать как множество, так и словарь:

```
a = {}
print(type(a))     # <class 'dict'>

b = {1, 2, 3}
print(type(b))     # <class 'set'>

c = {'a': 1, 'b': 2}
print(type(c))     # <class 'dict'>
```

2 Общие подходы к работе с любой коллекцией

Разобравшись в классификацией, рассмотрим что можно делать с любой стандартной коллекцией независимо от её типа (в примерах список и словарь, но это работает и для всех остальных рассматриваемых стандартных типов коллекций):

```
# Зададим исходно список и словарь (скопировать перед примерами ниже):
my_list = ['a', 'b', 'c', 'd', 'e', 'f']
my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
```

2.1 Печать элементов коллекции с помощью функции **print**()

```
print(my_list)   # ['a', 'b', 'c', 'd', 'e', 'f']
print(my_dict)   # {'a': 1, 'c': 3, 'e': 5, 'f': 6, 'b': 2, 'd': 4}
                 # Не забываем, что порядок элементов в неиндексированных коллекциях
не сохраняется.
```

2.2 Подсчёт количества членов коллекции с помощью функции **len**()

```
print(len(my_list)) # 6
print(len(my_dict)) # 6 - для словаря пара ключ-значение считаются одним элементом.
print(len('ab c')) # 4 - для строки элементом является 1 символ
```

2.3 Проверка принадлежности элемента данной коллекции с помощью оператора **in**

x in s — вернет True, если элемент входит в коллекцию s и False — если не входит

Есть и вариант проверки не принадлежности: **x not in s**, где есть по сути, просто добавляется отрицание перед булевым значением предыдущего выражения.

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f']
print('a' in my_list)           # True
print('q' in my_list)           # False
print('a' not in my_list)       # False
print('q' not in my_list)       # True
```

Для **словаря** возможны варианты, понятные из кода ниже:

```
my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
print('a' in my_dict)               # True - без указания метода поиск по ключам
print('a' in my_dict.keys())        # True - аналогично примеру выше
print('a' in my_dict.values())      # False - так как 'a' — ключ, не значение
print(1 in my_dict.values())        # True
```

Можно ли проверять пары? Можно!

```
print(('a',1) in my_dict.items())   # True
print(('a',2) in my_dict.items())   # False
```

Для **строки** можно искать не только один символ, но и подстроку:

```
print('ab' in 'abc')    # True
```

2.4 Обход всех элементов коллекции в цикле **for in**

В данном случае, в цикле будут последовательно перебираться элементы коллекции, пока не будут перебраны все из них.

```
for elm in my_list:
    print(elm)
```

Обратите внимание на следующие моменты:

Порядок обработки элементов для не индексированных коллекций будет не тот, как при их создании
У прохода в цикле по **словарю** есть свои особенности:

```
for elm in my_dict:
      # При таком обходе словаря, перебираются только ключи
      # равносильно for elm in my_dict.keys()
      print(elm)

for elm in my_dict.values():
      # При желании можно пройти только по значениям
```

```
    print(elm)
```

Но чаще всего нужны пары ключ(key) — значение (value).

```
for key, value in my_dict.items():
    # Проход по .items() возвращает кортеж (ключ, значение),
    # который присваивается кортежу переменных key, value
    print(key, value)
```

Возможная ошибка: Не меняйте количество элементов коллекции в теле цикла во время итерации по этой же коллекции! — Это порождает не всегда очевидные на первый взгляд ошибки.

Чтобы этого избежать подобных побочных эффектов, можно, например, итерировать копию коллекции:

```
for elm in list(my_list):
    # Теперь можете удалять и добавлять элементы в исходный список my_list,
    # так как итерация идет по его копии.
```

2.5 Функции **min**(), **max**(), **sum**()

Функции **min**(), **max**() — поиск минимального и максимального элемента соответственно — работают не только для числовых, но и для строковых значений.
sum() — суммирование всех элементов, если они все числовые.

```
print(min(my_list))              # a
print(sum(my_dict.values()))     # 21
```

3 Общие методы для части коллекций

Ряд методов у коллекционных типов используется в более чем одной коллекции для решения задач одного типа.

Применимости методов, в зависимости от типа коллекции

Тип коллекции	.count()	.index()	.copy()	.clear()
Список (list)	+	+	- (Python <3.3) + (Python >=3.3)	- (Python <3.3) + (Python >=3.3)
Кортеж (tuple)	+	+	-	-
Строка (string)	+	+	-	-
Множество (set)	-	-	+	+
Неизменное множество (frozenset)	-	-	+	-
Словарь (dict)	-	-	+	+

UPD: Важные дополнения в третьей статье: Добавление и удаление элементов изменяемых коллекций.

Объяснение работы методов и примеры:

.**count**() — метод подсчета определенных элементов для неуникальных коллекций (строка, список, кортеж), возвращает сколько раз элемент встречается в коллекции.

```
my_list = [1, 2, 2, 2, 2, 3]
print(my_list.count(2))     # 4 экземпляра элемента равного 2
print(my_list.count(5))     # 0 - то есть такого элемента в коллекции нет
```

.**index**() — возвращает минимальный индекс переданного элемента для индексированных коллекций (строка, список, кортеж)

```
my_list = [1, 2, 2, 2, 2, 3]
print(my_list.index(2))  # первый элемент равный 2 находится по индексу 1 (индексация с нуля!)
print(my_list.index(5))  # ValueError: 5 is not in list - отсутствующий элемент выдаст ошибку!
```

.**copy**() — метод возвращает неглубокую (не рекурсивную) копию коллекции (список, словарь, оба типа множества).

```
my_set = {1, 2, 3}
my_set_2 = my_set.copy()
print(my_set_2 == my_set)  # True - коллекции равны - содержат одинаковые значения
```

```
print(my_set_2 is my_set)  # False - коллекции не идентичны - это разные объекты с разными id
```

.**clear**() — метод изменяемых коллекций (список, словарь, множество), удаляющий из коллекции все элементы и превращающий её в пустую коллекцию.

```
my_set = {1, 2, 3}
print(my_set)  # {1, 2, 3}
my_set.clear()
print(my_set)  # set()
```

Особые методы сравнения множеств (set, frozenset)

set_a.**isdisjoint**(set_b) — истина, если set_a и set_b не имеют общих элементов.
set_b.**issubset**(set_a) — если все элементы множества set_b принадлежат множеству set_a, то множество set_b целиком входит в множество set_a и является его подмножеством (set_b — подмножество)
set_a.**issuperset**(set_b) — соответственно, если условие выше справедливо, то set_a — надмножество

```
set_a = {1, 2, 3}
set_b = {2, 1}                  # порядок элементов не важен!
set_c = {4}
set_d = {1, 2, 3}

print(set_a.isdisjoint(set_c))  # True - нет общих элементов
print(set_b.issubset(set_a))    # True  - set_b целиком входит в set_a, значит set_b - подмножество
print(set_a.issuperset(set_b))  # True - set_b целиком входит в set_a, значит set_a - надмножество
```

При равенстве множеств они одновременно и подмножество и надмножество друг для друга

```
print(set_a.issuperset(set_d))  # True
print(set_a.issubset(set_d))    # True
```

4 Конвертация одного типа коллекции в другой

В зависимости от стоящих задач, один тип коллекции можно конвертировать в другой тип коллекции. Для этого, как правило достаточно передать одну коллекцию в функцию создания другой (они есть в таблице выше).

```
my_tuple = ('a', 'b', 'a')
my_list = list(my_tuple)
my_set = set(my_tuple)                   # теряем индексы и дубликаты элементов!
my_frozenset = frozenset(my_tuple)      # теряем индексы и дубликаты элементов!
print(my_list, my_set, my_frozenset)    # ['a', 'b', 'a'] {'a', 'b'} frozenset({'a', 'b'})
```

Обратите внимание, что при преобразовании одной коллекции в другую возможна потеря данных:

При преобразовании в множество теряются дублирующие элементы, так как множество содержит только уникальные элементы! Собственно, проверка на уникальность, обычно и является причиной использовать множество в задачах, где у нас есть в этом потребность.
При конвертации индексированной коллекции в неиндексированную теряется информация о порядке элементов, а в некоторых случаев она может быть критически важной!
После конвертации в не изменяемый тип, мы больше не сможем менять элементы коллекции — удалять, изменять, добавлять новые. Это может привести к ошибкам в наших функциях обработки данных, если они были написаны для работы с изменяемыми коллекциями.

Дополнительные детали:

Способом выше не получится создать **словарь**, так как он состоит из пар ключ: значение.

Это ограничение можно обойти, создав словарь комбинируя ключи со значениями с использованием zip():

```
my_keys = ('a', 'b', 'c')
my_values = [1, 2]      # Если количество элементов разное -
                    # будет отработано пока хватает на пары - лишние отброшены
my_dict = dict(zip(my_keys, my_values))
print(my_dict)         # {'a': 1, 'b': 2}
```

Создаем **строку** из другой коллекции:

```
my_tuple = ('a', 'b', 'c')
my_str = ''.join(my_tuple)
print(my_str)       # abc
```

Возможная ошибка: Если Ваша коллекция содержит изменяемые элементы (например список списков), то ее нельзя конвертировать в не изменяемую коллекцию, так как ее элементы могут быть только не изменяемыми!

```
my_list = [1, [2, 3], 4]
my_set = set(my_list)   # TypeError: unhashable type: 'list'
```

Примечание: Самые мощные и гибкие способы — генераторы коллекций будут рассмотрены отдельно в четвертой части цикла, так как там много нюансов и вариантов использования, на которых редко заостряют внимание, и требуется

детальный разбор.

UPD: ShashkovS в комментариях выложил ссылки на важную и полезную информацию по алгоритмической сложности операций с коллекциями:

TimeComplexity (aka «Big O» or «Big Oh») (на английском)
Complexity of Python Operations (на английском)

==========

Python: коллекции, часть 2/4: индексирование, срезы, сортировка

https://habr.com/ru/post/319200/

Последовательность	a	b	c	d	e	f	g	Результат слайсинга
Индексы	0 (-7)	1 (-6)	2 (-5)	3 (-4)	4 (-3)	5 (-2)	6 (-1)	
[:] (→)	+	+	+	+	+	+	+	abcdefg
[::-1] (←)	+	+	+	+	+	+	+	gfedcba
[::2] (→)	+		+		+		+	aceg
[1::2] (→)		+		+		+		bdf
[:1]	+							a
[-1:]							+	g
[3:4]				+				d
[-3:] (→)					+	+	+	efg
[-3:1:-1] (←)			+	+	+			edc
[2:5] (→)			+	+	+			cde

Данная статья является продолжением моей статьи "Python: коллекции, часть 1: классификация, общие подходы и методы, конвертация".

В данной статье мы продолжим изучать общие принципы работы со стандартными коллекциями (модуль collections в ней не рассматривается) Python.

Для кого: для изучающих Python и уже имеющих начальное представление о коллекциях и работе с ними, желающих систематизировать и углубить свои знания, сложить их в целостную картину.

ОГЛАВЛЕНИЕ:

1. Индексирование

1.1 Индексированные коллекции

Рассмотрим индексированные коллекции (их еще называют последовательности — sequences) — список (list), кортеж (tuple), строку (string).

Под индексированностью имеется ввиду, что элементы коллекции располагаются

в определённом порядке, каждый элемент имеет свой индекс от 0 (то есть первый по счёту элемент имеет индекс не 1, а 0) до индекса на единицу меньшего длины коллекции (т.е. len(mycollection)-1).

1.2 Получение значения по индексу

Для всех индексированных коллекций можно получить значение элемента по его индексу в квадратных скобках. Причем, можно задавать отрицательный индекс, это значит, что будем находить элемент с конца считая обратном порядке.

При задании отрицательного индекса, последний элемент имеет индекс -1, предпоследний -2 и так далее до первого элемента индекс которого равен значению длины коллекции с отрицательным знаком, то есть (-len(mycollection).

элементы	a	b	c	d	e
индексы	0 (-5)	1 (-4)	2 (-3)	3 (-2)	4 (-1)

```
my_str = "abcde"
print(my_str[0])          # a - первый элемент
print(my_str[-1])         # e - последний элемент
print(my_str[len(my_str)-1])     # e - так тоже можно взять последний элемент
print(my_str[-2])         # d - предпоследний элемент
```

Наши коллекции могут иметь несколько уровней вложенности, как список списков в примере ниже. Для перехода на уровень глубже ставится вторая пара квадратных скобок и так далее.

```
my_2lvl_list = [[1, 2, 3], ['a', 'b', 'c']]
print(my_2lvl_list[0])      # [1, 2, 3] - первый элемент — первый вложенный список
print(my_2lvl_list[0][0])   # 1 — первый элемент первого вложенного списка
print(my_2lvl_list[1][-1])  # c — последний элемент второго вложенного списка
```

1.3 Изменение элемента списка по индексу

Поскольку кортежи и строки у нас неизменяемые коллекции, то по индексу мы можем только брать элементы, но не менять их:

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[0])    # 1
my_tuple[0] = 100         # TypeError: 'tuple' object does not support item
assignment
```

А вот для списка, если взятие элемента по индексу располагается в левой части выражения, а далее идёт оператор присваивания =, то мы задаём новое значение элементу с этим индексом.

```
my_list = [1, 2, 3, [4, 5]]
my_list[0] = 10
my_list[-1][0] = 40
print(my_list)            # [10, 2, 3, [40, 5]]
```

UPD: Примечание: Для такого присвоения, элемент уже должен существовать в списке, нельзя таким образом добавить элемент на несуществующий индекс.

```
my_list = [1, 2, 3, 4, 5]
my_list[5] = 6      # IndexError: list assignment index out of range
```

2 Срезы

2.1 Синтаксис среза

Очень часто, надо получить не один какой-то элемент, а некоторый их набор ограниченный определенными простыми правилами — например первые 5 или последние три, или каждый второй элемент — в таких задачах, вместо перебора в цикле намного удобнее использовать так называемый срез (slice, slicing).

Следует помнить, что взяв элемент по индексу или срезом (slice) мы не как не меняем исходную коллекцию, мы просто скопировали ее часть для дальнейшего использования (например добавления в другую коллекцию, вывода на печать, каких-то вычислений). Поскольку сама коллекция не меняется — это применимо как к изменяемым (список) так и к неизменяемым (строка, кортеж) последовательностям.

Синтаксис среза похож на таковой для индексации, но в квадратных скобках вместо одного значения указывается 2-3 через двоеточие:

```
my_collection[start:stop:step]  # старт, стоп и шаг
```

Особенности среза:

Отрицательные значения старта и стопа означают, что считать надо не с начала, а с конца коллекции.
Отрицательное значение шага — перебор ведём в обратном порядке справа налево.
Если не указан старт **[:stop:step]**— начинаем с самого края коллекции, то есть с первого элемента (включая его), если шаг положительный или с последнего (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).
Если не указан стоп **[start:: step]** — идем до самого края коллекции, то есть до последнего элемента (включая его), если шаг положительный или до первого элемента (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).
step = 1, то есть последовательный перебор слева направо указывать не обязательно — это значение шага по умолчанию. В таком случае достаточно указать **[start:stop]**

Можно сделать даже так **[:]** — это значит взять коллекцию целиком
ВАЖНО: ***При срезе, первый индекс входит в выборку, а второй нет!***
То есть от старта включительно, до стопа, где стоп не включается в результат. Математически это можно было бы записать как [start, stop) или пояснить вот таким правилом:

```
[ <первый включаемый> : <первый НЕ включаемый> : <шаг> ]
```

Поэтому, например, mylist[::-1] не идентично mylist[:0:-1], так как в первом случае мы включим все элементы, а во втором дойдем до 0 индекса, но не включим его!

Примеры срезов в виде таблицы:

Последовательность	a	b	c	d	e	f	g	**Результат слайсинга**
Индексы	0 (-7)	1 (-6)	2 (-5)	3 (-4)	4 (-3)	5 (-2)	6 (-1)	
[:] (→)	+	+	+	+	+	+	+	abcdefg
[::-1] (←)	+	+	+	+	+	+	+	gfedcba
[::2] (→)	+		+		+		+	aceg
[1::2] (→)		+		+		+		bdf
[:1]	+							a
[-1:]							+	g
[3:4]				+				d
[-3:] (→)					+	+	+	efg
[-3:1:-1] (←)			+	+	+			edc
[2:5] (→)			+	+	+			cde

Код примеров из таблицы

2.2. Именованные срезы

Чтобы избавится от «магических констант», особенно в случае, когда один и тот же срез надо применять многократно, можно задать константы с именованными срезами с пользованием специальной функции **slice()**()

Примечание: None соответствует опущенному значению по-умолчанию. То есть [:2] становится slice(None, 2), а [1::2] становится slice(1, None, 2).

```
person = ('Alex', 'Smith', "May", 10, 1980)
NAME, BIRTHDAY = slice(None, 2), slice(2, None)
       # задаем константам именованные срезы
         # данные константы в квадратных скобках заменятся соответствующими срезами
print(person[NAME])      # ('Alex', 'Smith')
print(person[BIRTHDAY])  # ('May', 10, 1980)
```

```
my_list = [1, 2, 3, 4, 5, 6, 7]
EVEN = slice(1, None, 2)
print(my_list[EVEN])     # [2, 4, 6]
```

2.3 Изменение списка срезом

Важный момент, на котором не всегда заостряется внимание — с помощью среза можно не только получать копию коллекции, но в случае списка можно также менять значения элементов, удалять и добавлять новые.

Проиллюстрируем это на примерах ниже:

Даже если хотим добавить один элемент, необходимо передавать итерируемый объект, иначе будет ошибка TypeError: can only assign an iterable

```
my_list = [1, 2, 3, 4, 5]
# my_list[1:2] = 20     # TypeError: can only assign an iterable
my_list[1:2] = [20]     # Вот теперь все работает
print(my_list)          # [1, 20, 3, 4, 5]
```

Для вставки одиночных элементов можно использовать срез, код примеров есть ниже, но делать так не рекомендую, так как такой синтаксис хуже читать. Лучше использовать методы списка .append() и .insert():

Срез аналоги .append() и insert()

Можно менять части последовательности — это применение выглядит наиболее интересным, так как решает задачу просто и наглядно.

```
my_list = [1, 2, 3, 4, 5]
my_list[1:3] = [20, 30]
print(my_list)          # [1, 20, 30, 4, 5]
my_list[1:3] = [0]      # нет проблем заменить два элемента на один
print(my_list)          # [1, 0, 4, 5]
my_list[2:] = [40, 50, 60]   # или два элемента на три
print(my_list)               # [1, 0, 40, 50, 60]
```

Можно просто удалить часть последовательности

```
my_list = [1, 2, 3, 4, 5]
my_list[:2] = []    # или del my_list[:2]
print(my_list)      # [3, 4, 5]
```

2.4 Выход за границы индекса

Обращение по индексу по сути является частным случаем среза, когда мы обращаемся только к одному элементу, а не диапазону. Но есть очень важное отличие в обработке ситуации с отсутствующим элементом с искомым индексом.

Обращение к несуществующему индексу коллекции вызывает ошибку:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[-10])       # IndexError: list index out of range
print(my_list[10])        # IndexError: list index out of range
```

А в случае выхода границ среза за границы коллекции никакой ошибки не происходит:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0:10])      # [1, 2, 3, 4, 5] – отработали в пределах коллекции
print(my_list[10:100])      # [] - таких элементов нет – вернули пустую коллекцию
print(my_list[10:11])     # [] - проверяем 1 отсутствующий элемент - пустая
коллекция, без ошибки
```

Примечание: Для тех случаев, когда функционала срезов недостаточно и требуются более сложные выборки, можно воспользоваться синтаксисом выражений-генераторов, рассмотрению которых посвещена 4 статья цикла.

3 Сортировка элементов коллекции

Сортировка элементов коллекции важная и востребованная функция, постоянно встречающаяся в обычных задачах. Тут есть несколько особенностей, на которых не всегда заостряется внимание, но которые очень важны.

3.1 Функция **sorted**()

Мы может использовать функцию sorted() для вывода списка сортированных элементов любой коллекции для последующее обработки или вывода.

функция не меняет исходную коллекцию, а возвращает новый список из ее элементов;
не зависимо от типа исходной коллекции, вернётся список (list) ее элементов;
поскольку она не меняет исходную коллекцию, ее можно применять к неизменяемым коллекциям;
Поскольку при сортировке возвращаемых элементов нам не важно, был ли у элемента некий индекс в исходной коллекции, можно применять к неиндексированным коллекциям;
Имеет дополнительные не обязательные аргументы:
reverse=True — сортировка в обратном порядке
key=funcname (начиная с Python 2.4) — сортировка с помощью специальной функции funcname, она может быть как стандартной функцией Python, так и специально написанной вами для данной задачи функцией и лямбдой.

```
my_list = [2, 5, 1, 7, 3]
```

```
my_list_sorted = sorted(my_list)
print(my_list_sorted)       # [1, 2, 3, 5, 7]

my_set = {2, 5, 1, 7, 3}
my_set_sorted = sorted(my_set, reverse=True)
print(my_set_sorted)        # [7, 5, 3, 2, 1]
```

Пример сортировки списка строк по длине len() каждого элемента:

```
my_files = ['somecat.jpg', 'pc.png', 'apple.bmp', 'mydog.gif']
my_files_sorted = sorted(my_files, key=len)
print(my_files_sorted)      # ['pc.png', 'apple.bmp', 'mydog.gif', 'somecat.jpg']
```

3.2 Функция **reversed**()

Функция reversed() применяется для последовательностей и работает по другому:

возвращает генератор списка, а не сам список;
если нужно получить не генератор, а готовый список, результат можно обернуть в list() или же вместо reversed() воспользоваться срезом [: :-1];
она **не сортирует** элементы, а возвращает их в обратном порядке, то есть читает с конца списка;
из предыдущего пункта понятно, что если у нас коллекция неиндексированная — мы не можем вывести её элементы в обратном порядке и эта функция к таким коллекциям не применима — получим «TypeError: argument to reversed() must be a sequence»;
не позволяет использовать дополнительные аргументы — будет ошибка «TypeError: reversed() does not take keyword arguments».

```
my_list = [2, 5, 1, 7, 3]
my_list_sorted = reversed(my_list)
print(my_list_sorted)           # <listreverseiterator object at 0x7f8982121450>
print(list(my_list_sorted))     # [3, 7, 1, 5, 2]
print(my_list[::-1])            # [3, 7, 1, 5, 2] - тот же результат с помощью среза
```

3.3 Методы списка .**sort**() и .**reverse**()

У списка (и только у него) есть особые методы .sort() и .reverse() которые делают тоже самое, что соответствующие функции sorted() и reversed(), но при этом:

Меняют сам исходный список, а не генерируют новый;
Возвращают None, а не новый список;
поддерживают те же дополнительные аргументы;
в них не надо передавать сам список первым параметром, более того, если это сделать — будет ошибка — не верное количество аргументов.

```
my_list = [2, 5, 1, 7, 3]
```

```
my_list.sort()
print(my_list)          # [1, 2, 3, 5, 7]
```

Обратите внимание: Частая ошибка начинающих, которая не является ошибкой для интерпретатора, но приводит не к тому результату, который хотят получить.

```
my_list = [2, 5, 1, 7, 3]
my_list = my_list.sort()
print(my_list)          # None
```

3.4 Особенности сортировки словаря

В сортировке словаря есть свои особенности, вызванные тем, что элемент словаря — это пара ключ: значение.

UPD: Так же, не забываем, что говоря о сортировке словаря, мы имеем ввиду сортировку полученных из словаря данных для вывода или сохранения в индексированную коллекцию. Сохранить данные сортированными в самом стандартном словаре не получится, они в нем, как и других неиндексированных коллекциях находятся в произвольном порядке.

sorted(my_dict) — когда мы передаем в функцию сортировки словарь без вызова его дополнительных методов — идёт перебор только ключей, сортированный список ключей нам и возвращается;
sorted(my_dict.**keys**()) — тот же результат, что в предыдущем примере, но прописанный более явно;
sorted(my_dict.**items**()) — возвращается сортированный список кортежей (ключ, значение), сортированных по ключу;
sorted(my_dict.**values**()) — возвращается сортированный список значений

```
my_dict = {'a': 1, 'c': 3, 'e': 5, 'f': 6, 'b': 2, 'd': 4}
mysorted = sorted(my_dict)
print(mysorted)           # ['a', 'b', 'c', 'd', 'e', 'f']
mysorted = sorted(my_dict.items())
print(mysorted)           # [('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5), ('f',
6)]
mysorted = sorted(my_dict.values())
print(mysorted)           # [1, 2, 3, 4, 5, 6]
```

Отдельные сложности может вызвать **сортировка словаря** не по ключам, а **по значениям**, если нам не просто нужен список значений, и именно выводить пары в порядке сортировки по значению.

Для решения этой задачи можно в качестве специальной функции сортировки передавать lambda-функцию **lambda x: x[1]** которая из получаемых на каждом этапе кортежей (ключ, значение) будет брать для сортировки второй элемент кортежа.

```
population = {"Shanghai": 24256800, "Karachi": 23500000, "Beijing": 21516000,
"Delhi": 16787941}
# отсортируем по возрастанию населения:
population_sorted = sorted(population.items(), key=lambda x: x[1])
print(population_sorted)
# [('Delhi', 16787941), ('Beijing', 21516000), ('Karachi', 23500000), ('Shanghai',
24256800)]
```

UPD от ShashkovS: 3.5 Дополнительная информация по использованию параметра key при сортировке

Допустим, у нас есть список кортежей названий деталей и их стоимостей.
Нам нужно отсортировать его сначала по названию деталей, а одинаковые детали по убыванию цены.

```
shop = [('каретка', 1200), ('шатун', 1000), ('седло', 300),
        ('педаль', 100), ('седло', 1500), ('рама', 12000),
        ('обод', 2000), ('шатун', 200), ('седло', 2700)]

def prepare_item(item):
    return (item[0], -item[1])

shop.sort(key=prepare_item)
```

Результат сортировки

Перед тем, как сравнивать два элемента списка к ним применялась функция prepare_item, которая меняла знак у стоимости (функция применяется ровно по одному разу к каждому элементу. В результате при одинаковом первом значении сортировка по второму происходила в обратном порядке.

Чтобы не плодить утилитарные функции, вместо использования сторонней функции, того же эффекта можно добиться с использованием лямбда-функции.

```
# Данные скопировать из примера выше
shop.sort(key=lambda x: (x[0], -x[1]))
```

Дополнительные детали и примеры использования параметра key:

wiki.python.org/moin/HowTo/Sorting#Key_Functions (на английском).
habrahabr.ru/post/319200/#comment_10011882 — еще один комментарий с детальным примером ShashkovS к данной статье

UPD от ShashkovS: 3.6 Устойчивость сортировки

Допустим данные нужно отсортировать сначала по столбцу A по возрастанию, затем по столбцу B по убыванию, и наконец по столбцу C снова по возрастанию.

Если данные в столбце B числовые, то при помощи подходящей функции в key

можно поменять знак у элементов B, что приведёт к необходимому результату.
А если все данные текстовые? Тут есть такая возможность.
Дело в том, что сортировка sort в Python устойчивая (начиная с Python 2.2), то есть она не меняет порядок «одинаковых» элементов.

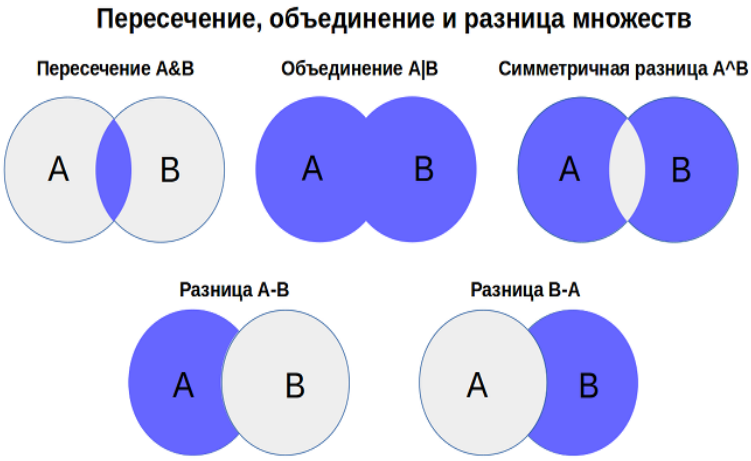
Поэтому можно просто отсортировать три раза по разным ключам:

```
data.sort(key=lambda x: x['C'])
data.sort(key=lambda x: x['B'], reverse=True)
data.sort(key=lambda x: x['A'])
```

Дополнительная информация по устойчивости сортировки и примеры: wiki.python.org/moin/HowTo/Sorting#Sort_Stability_and_Complex_Sorts (на наглийском).
==========

Python: коллекции, часть 3/4: объединение коллекций, добавление и удаление элементов

https://habr.com/ru/post/319876/



Продолжим изучать общие принципы работы со стандартными коллекциями (модуль collections в ней не рассматривается) Python. Будут рассматриваться способы объединения и обновления коллекций с формированием новой или изменением исходной, а также способы добавлять и удалять элементы в изменяемые коллекции.

Для кого: для изучающих Python и уже имеющих начальное представление о коллекциях и работе с ними, желающих систематизировать и углубить свои знания, сложить их в целостную картину.

Оглавление:

1. Объединение строк, кортежей, списков, словарей без изменения исходных

Рассмотрим способы объединения строк, кортежей, списков, словарей без изменения исходных коллекций — когда из нескольких коллекций создаётся новая коллекция того же тип без изменения изначальных.

Объединение строк (string) и кортежей (tuple) возможна с использованием оператора сложения «+»

```
str1 = 'abc'
str2 = 'de'
str3 = str1 + str2
print(str3)         # abcde

tuple1 = (1, 2, 3)
tuple2 = (4, 5)
tuple3 = tuple1 + tuple2
print(tuple3)       # (1, 2, 3, 4, 5)
```

Для **объединения списков** (list) возможны три варианта без изменения исходного списка:

Добавляем все элементы второго списка к элементам первого, (аналог метод .extend() но без изменения исходного списка):

```
a = [1, 2, 3]
b = [4, 5]
c = a + b
print(a, b, c)      # [1, 2, 3]  [4, 5]  [1, 2, 3, 4, 5]
```

Добавляем второй список как один элемент без изменения исходного списка (аналог метода.append() но без изменения исходного списка):

```
a = [1, 2, 3]
b = [4, 5]
c = a + [b]
print(a, b, c)     # [1, 2, 3]  [4, 5]  [1, 2, 3, [4, 5]]
```

UPD: Способ добавленный longclaps в комментариях:

```
a, b = [1, 2, 3], [4, 5]
c = [*a, *b]  # работает на версии питона 3.5 и выше
print(c)      # [1, 2, 3, 4, 5]
```

Со **словарем** (dict) все не совсем просто.

Сложить два словаря чтобы получить третий оператором + Питон не позволяет «TypeError: unsupported operand type(s) for +: 'dict' and 'dict'».

Это можно сделать по-другому комбинируя методы .copy() и .update():

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = dict1.copy()
dict3.update(dict2)
print(dict3)              # {'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

В Питоне 3.5 появился новый более изящный способ:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = {**dict1, **dict2}
print(dict3)              # {'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

2. Объединение множеств без изменения исходных

Для обоих типов множеств (set, frozenset) возможны различные варианты комбинации множеств (исходные множества при этом не меняются — возвращается новое множество).


```
# Зададим исходно два множества (скопировать перед каждым примером ниже)
a = {'a', 'b'}
b = {    'b', 'c'}  # отступ перед b для наглядности
```

Объединение (union):

```
c = a.union(b)     # c = b.union(a) даст такой же результат
# c = a + b        # Обычное объединение оператором + не работает
                 # TypeError: unsupported operand type(s) for +: 'set' and 'set'
c = a | b          # Альтернативная форма записи объединения
print(c)         # {'a', 'c', 'b'}
```

Пересечение (intersection):

```
c = a.intersection(b)    # c = b.intersection(a) даст такой же результат
c = a & b                # Альтернативная форма записи пересечения
print(c)                 # {'b'}
```

Пересечение более 2-х множеств сразу:

```
a = {'a', 'b'}
b = {     'b', 'c'}
c = {    'b', 'd'}
d = a.intersection(b, c)  # Первый вариант записи
d = set.intersection(a, b, c)   # Второй вариант записи (более наглядный)
print(d)                        # {'b'}
```

Разница (difference) — результат зависит от того, какое множество из какого вычитаем:

```
c = a.difference(b)      # c = a - b другой способ записи дающий тот же результат
print(c)                 # {'a'}
c = b.difference(a)      # c = b - a другой способ записи дающий тот же результат
print(c)                 # {'c'}
```

Симметричная разница (symmetric_difference) Это своего рода операция противоположная пересечению — выбирает элементы из обеих множеств которые не пересекаются, то есть все кроме совпадающих:

```
c = b.symmetric_difference(a)
# c = a.symmetric_difference(b)       # даст такой же результат
c = b ^ a                             # Альтернативная форма записи симметричной
разницы
print(c)                        # {'a', 'c'}
```

3. Объединение списка, словаря и изменяемого множества с изменением исходной коллекции

Для **списка**

Добавляем все элементы второго списка к элементам первого с измением первого списка методом .**extend**():

```
a.extend(b)    # a += b эквивалентно a.extend(b)
print(a, b)    # [1, 2, 3, 4, 5]  [4, 5]
```

Добавляем второй список как один элемент с изменением первого списка методом .**append**():

```
 a.append(b)    # a += [b] эквивалентно a.append(b)
 print(a, b)    # [1, 2, 3, [4, 5]]  [4, 5]
```

Для изменения **словаря** с добавления элементов другого словаря используется метод .**update**().

Обратите внимание: для совпадающих ключей словаря при этом обновляются значения:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'a': 100, 'c': 3, 'd': 4}
dict1.update(dict2)
print(dict1)        # {'a': 100, 'c': 3, 'b': 2, 'd': 4}
```

Для **изменяемого множества** (set) кроме операций, описанных в предыдущем разделе, также возможны их аналоги, но уже с изменением исходного множества — эти методы заканчиваются на _update. Результат зависит от того, какое множество каким обновляем.

.**difference_update**()

```
a = {'a', 'b'}
b = {     'b', 'c'}
a.difference_update(b)
print(a, b)         # {'a'} {'b', 'c'}
a = {'a', 'b'}
b = {     'b', 'c'}
b.difference_update(a)
print(a, b)         # {'a', 'b'} {'c'}
```

.**intersection_update**()

```
a = {'a', 'b'}
b = {     'b', 'c'}
a.intersection_update(b)
print(a, b)         # {'b'} {'b', 'c'}

a = {'a', 'b'}
b = {     'b', 'c'}
b.intersection_update(a)
print(a, b)         # {'b', 'a'} {'b'}
```

.**symmetric_difference_update**()

```
a = {'a', 'b'}
b = {     'b', 'c'}
a.symmetric_difference_update(b)
```

```
print(a, b)         # {'c', 'a'} {'c', 'b'}

a = {'a', 'b'}
b = {     'b', 'c'}
b.symmetric_difference_update(a)
print(a, b)         # {'a', 'b'} {'c', 'a'}
```

4 Добавление и удаление элементов изменяемых коллекций

Добавление и удаление элементов в коллекцию возможно только для изменяемых коллекций: списка (list), множества (только set, не frozenset), словаря (dict). Причём для списка, который является индексированной коллекцией, также важно на какую позицию мы добавляем элемент.

Метод / Оператор	Описание	Коллекция		
		list	dict	set
Удаление элемента				
.remove()	Удаляет элемент по значению - для списка удаляется только первый элемент с таким значением, в множестве элемент в любом случае был уникален. В случае отсутствия элемента с таким значением ошибка: *ValueError* (для списка) или *KeyError* (для множества).	+	-	+
.discard()	Аналог .remove(), но работает только на множестве и не даёт ошибки в случае отсутствия элемента.	-	-	+
.pop()	Удаляет и возвращает значение элемента по указанному индексу (для списка), ключу (для словаря) или просто случайный элемент (для множества). В случае отсутствия указанного элемента или если множество пустое - ошибка: *IndexError* (для списка), *KeyError* (для словаря и множества).	+	+	+
.popitem()	Аналог .pop(), но работает только на словаре и возвращает не значение, а случайный кортеж (ключ, значение) или *KeyError*, если словарь пустой.	-	+	-
.clear()	Удаляем из коллекции все элементы и превращаем её в пустую коллекцию.	+	+	+
del	Удаляем элемент по индексу (для списка) или срезу (для списка) или ключу (для словаря), ничего не возвращаем. В случае отсутствия указанного элемента - ошибка: *IndexError* (для списка) или *KeyError* (для словаря).	+	+	-
[i:j:k]	Удаляем часть элементов списка срезом с помощью оператора del или присвоив им пустой список [].	+	-	-
Добавление элемента				
.append()	Добавляем элемент в конец списка.	+	-	-
.extend()	Расширяем список добавлением элементов из переданного итерируемого.	+	-	-
.insert()	Вставляем элемент в список перед переданным индексом.	+	-	-
.add()	Добавляем элемент в множество. Если такой уже существует — ничего не происходит.	-	-	+
[i:j:k]	Добавление элементов списка срезом в указанное место, возможно с заменой существующих.	+	-	-
Обновление коллекции				
.update()	Для словаря для существующих ключей обновляются значения новыми, для новых ключей — добавляется пара ключ: значение. Для множества происходит замена множества его объединением с переданными в метод множествами.	-	+	+

Примечания:

Примеры использования метода .insert(index, element)

Примеры использования оператора del

Удаление и добавление элементов списка срезом рассматривается во второй статье.

Пример работы .append() и .extend() рассматривается в третьей главе этой статьи.

5 Особенности работы с изменяемой и не изменяемой коллекцией

Строка неизменяемая коллекция — если мы ее меняем — мы создаем новый объект!

```
str1 = 'abc'
print(str1, id(str1))       # abc 140234080454000
str1 += 'de'
print(str1, id(str1))       # abcde 140234079974992 - Это НОВЫЙ объект, с другим id!
```

Пример кода с двумя исходно идентичными строками.

```
str1 = 'abc'
str2 = str1
print(str1 is str2)       # True - это две ссылки на один и тот же объект!
str1 += 'de'              # Теперь переменная str1 ссылается на другой объект!
print(str1 is str2)       # False - теперь это два разных объекта!
print(str1, str2)         # abcde abc - разные значения
```

Список изменяем и тут надо быть очень внимательным, чтобы не допустить серьезную ошибку! Сравните данный пример с примером со строками выше:

```
list1 = [1, 2, 3]
list2 = list1
print(list1 is list2)     # True - это две ссылки на один и тот же объект!
# А дальше убеждаемся, насколько это важно:
list1 += [4]
print(list1, list2)       # [1, 2, 3, 4] [1, 2, 3, 4]
# изменилось значение ОБЕИХ переменных, так как обе переменные ссылаются на один
объект!
```

А если нужна **независимая копия**, с которой можно работать отдельно?

```
list1 = [1, 2, 3]
```

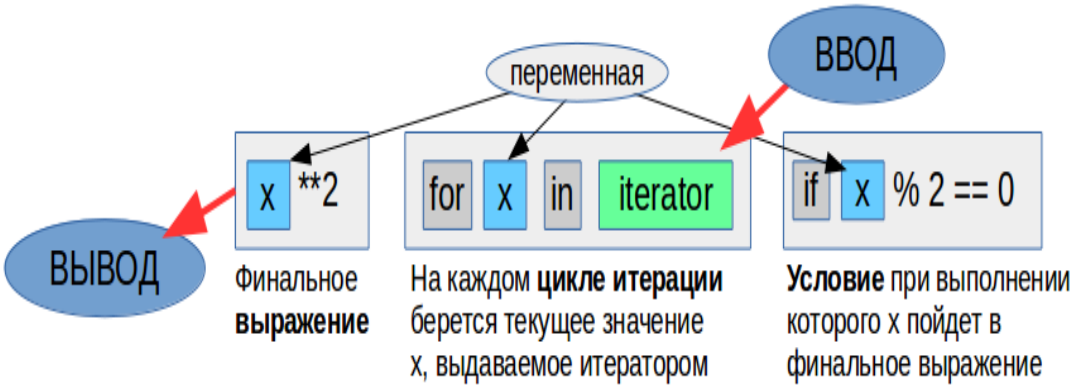
```
list2 = list(list1)       # Первый способ копирования
list3 = list1[:]          # Второй способ копирования
list4 = list1.copy()      # Третий способ копировани - только в Python 3.3+
print(id(list1), id(list2), id(list3), id(list4))
# все 4 id разные, что значит что мы создали 4 разных объекта

list1 += [4]              # меняем исходный список
print(list1, list2, list3, list4)       # [1, 2, 3, 4] [1, 2, 3] [1, 2, 3] [1, 2, 3]
# как мы и хотели - изменив исходный объект, его копии остались не тронутыми
```

==========

Python: коллекции, часть 4/4: Все о выражениях-генераторах, генераторах списков, множеств и словарей

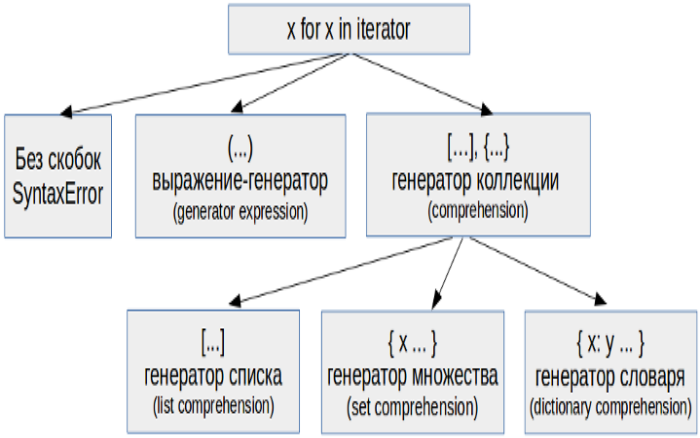
https://habr.com/ru/post/320288/



Заключительная часть моего цикла, посещенного работе с коллекциями. Данная статья самостоятельная, может изучаться и без предварительного изучения предыдущих.

Эта статья глубже и детальней предыдущих и поэтому может быть интересна **не только новичкам, но и достаточно опытным Python-разработчикам**.

Будут рассмотрены: выражения-генераторы, генераторы списка, словаря и множества, вложенные генераторы (5 вариантов), работа с enumerate(), range().
А также: классификация и терминология, синтаксис, аналоги в виде циклов и примеры применения.



Я постарался рассмотреть **тонкости и нюансы**, которые освещаются далеко не во всех книгах и курсах, и, в том числе, отсутствуют в уже опубликованных на Habrahabr статьях на эту тему.

Оглавление:

1. Определения и классификация

1.1 Что и зачем

Генераторы выражений предназначены для компактного и удобного способа генерации коллекций элементов, а также преобразования одного типа коллекций в другой.
В процессе генерации или преобразования возможно применение условий и модификация элементов.
Генераторы выражений являются синтаксическим сахаром и не решают задач, которые нельзя было бы решить без их использования.

1.2 Преимущества использования генераторов выражений

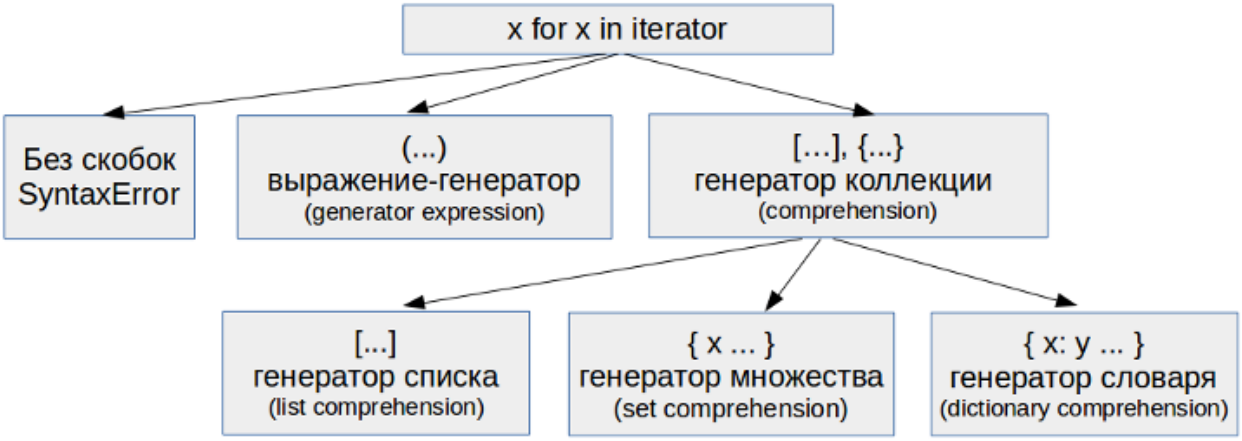
Более короткий и удобный синтаксис, чем генерация в обычном цикле.
Более понятный и читаемый синтаксис чем функциональный аналог сочетающий одновременное применение функций map(), filter() и lambda.
В целом: быстрее набирать, легче читать, особенно когда подобных операций много в коде.

1.3 Классификация и особенности

Сразу скажу, что существует некоторая терминологическая путаница в русских названиях того, о чем мы будем говорить.

В данной статье используются следующие обозначения:

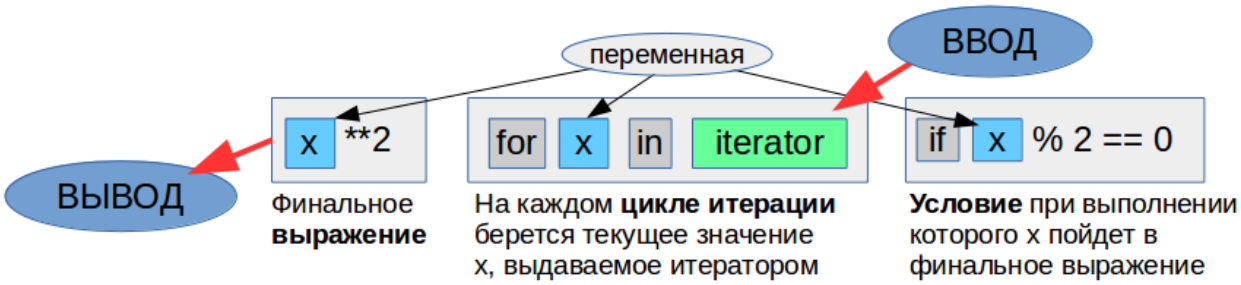
выражение-генератор (generator expression) — выражение в круглых скобках которое выдает создает на каждой итерации новый элемент по правилам.
генератор коллекции — обобщенное название для генератора списка (list comprehension), генератора словаря (dictionary comprehension) и генератора множества (set comprehension).



В отдельных местах, чтобы избежать нагромождения терминов, будет использоваться термин «генератор» без дополнительных уточнений.

2. Синтаксис

Для начала приведем иллюстрацию общего синтаксиса выражения-генератора.
Важно: этот синтаксис одинаков и для выражения-генератора и для всех трех типов генераторов коллекций, разница заключается, в каких скобках он будет заключен (смотрите предыдущую иллюстрацию).



Общие принципы важные для понимания:

Ввод — это итератор — это может быть функция-генератор, выражение-генератор, коллекция — любой объект поддерживающий итерацию по нему.
Условие — это фильтр при выполнении которого элемент пойдет в финальное выражение, если элемент ему не удовлетворяет, он будет пропущен.
Финальное выражение — преобразование каждого выбранного элемента перед его выводом или просто вывод без изменений.

2.1 Базовый синтаксис

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]     # Пусть у нас есть исходный список
list_b = [x for x in list_a]            # Создадим новый список используя генератор
списка
print(list_b)                           # [-2, -1, 0, 1, 2, 3, 4, 5]
print(list_a is list_b)                 # False - это разные объекты!
```

По сути, ничего интересного тут не произошло, мы просто получили копию списка. Делать такие копии или просто перегонять коллекции из типа в тип с помощью генераторов особого смысла нет — это можно сделать значительно проще применив соответствующие методы или функции создания коллекций (рассматривались в первой статье цикла).

Мощь генераторов выражений заключается в том, что мы можем задавать условия для включения элемента в новую коллекцию и можем делать преобразование текущего элемента с помощью выражения или функции перед его выводом (включением в новую коллекцию).

2.2 Добавляем условие для фильтрации

Важно: Условие проверяется на каждой итерации, и только элементы ему удовлетворяющие идут в обработку в выражении.

Добавим в предыдущий пример условие — брать только четные элементы.

```
# if x % 2 == 0 - остаток от деления на 2 равен нулю - число четное
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x for x in list_a if x % 2 == 0]
print(list_b)   # [-2, 0, 2, 4]
```

Мы можем использовать **несколько условий, комбинируя их логическими операторами**:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x for x in list_a if x % 2 == 0 and x > 0]
# берем те x, которые одновременно четные и больше нуля
print(list_b)   # [2, 4]
```

2.3 Добавляем обработку элемента в выражении

Мы можем вставлять не сам текущий элемент, прошедший фильтр, а результат вычисления выражения с ним или результат его обработки функцией.

Важно: Выражение выполняется независимо на каждой итерации, обрабатывая каждый элемент индивидуально.

Например, можем посчитать квадраты значений каждого элемента:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x**2 for x in list_a]
print(list_b)   # [4, 1, 0, 1, 4, 9, 16, 25]
```

Или посчитать длины строк с помощью функции len()

```
list_a = ['a', 'abc', 'abcde']
list_b = [len(x) for x in list_a]
print(list_b)   # [1, 3, 5]
```

2.4 Ветвление выражения

Обратите внимание: Мы можем использовать (начиная с Python 2.5) в выражении конструкцию **if-else для ветвления финального выражения**.

В таком случае:

Условия ветвления пишутся не после, а перед итератором.
В данном случае if-else это не фильтр перед выполнением выражения, а ветвление самого выражения, то есть переменная уже прошла фильтр, но в зависимости от условия может быть обработана по-разному!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x if x < 0 else x**2 for x in list_a]
# Если x-отрицательное - берем x, в остальных случаях - берем квадрат x
print(list_b)   # [-2, -1, 0, 1, 4, 9, 16, 25]
```

Никто не запрещает **комбинировать фильтрацию и ветвление**:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x**3 if x < 0 else x**2 for x in list_a if x % 2 == 0]
# вначале фильтр пропускает в выражение только четные значения
# после этого ветвление в выражении для отрицательных возводит в куб, а для остальных
в квадрат
print(list_b)   # [-8, 0, 4, 16]
```

Этот же пример в виде цикла

2.5 Улучшаем читаемость

Не забываем, что в Python синтаксис позволяет использовать переносы строк внутри скобок. Используя эту возможность, можно сделать синтаксис генераторов выражений более легким для чтения:

```
numbers = range(10)

# Before
squared_evens = [n ** 2 for n in numbers if n % 2 == 0]

# After
squared_evens = [
    n ** 2
    for n in numbers
    if n % 2 == 0
]
```

3. Аналоги в виде цикла for и в виде функций

Как уже говорилось выше, задачи решаемые с помощью генераторов выражений можно решить и без них. Приведем другие подходы, которые могут быть использованы для решения тех же задач.

Для примера возьмем простую задачу — сделаем из списка чисел список квадратов четных чисел и решим ее с помощью трех разных подходов:

3.1 Решение с помощью генератора списка

```
numbers = range(10)
squared_evens = [n ** 2 for n in numbers if n % 2 == 0]
print(squared_evens)   # [0, 4, 16, 36, 64]
```

3.2. Решение с помощью цикла for

Важно: Каждый генератор выражений можно переписать в виде цикла for, но не каждый цикл for можно представить в виде такого выражения.

```
numbers = range(10)
squared_evens = []
for n in numbers:
    if n % 2 == 0:
        squared_evens.append(n ** 2)
print(squared_evens)   # [0, 4, 16, 36, 64]
```

В целом, для очень сложных и комплексных задач, решение в виде цикла может быть понятней и проще в поддержке и доработке. Для более простых задач, синтаксис выражения-генератора будет компактней и легче в чтении.

3.3. Решение с помощью функций.

Для начала, замечу, что выражение генераторы и генераторы коллекций — это тоже функциональный стиль, но более новый и предпочтительный.

Можно применять и более старые функциональные подходы для решения тех же задач, комбинируя map(), lambda и filter().

```
numbers = range(10)
squared_evens = map(lambda n: n ** 2, filter(lambda n: n % 2 == 0, numbers))
print(squared_evens)         # <map object at 0x7f661e5dba20>
print(list(squared_evens))   # [0, 4, 16, 36, 64]
# Примечание: в Python 2 в переменной squared_evens окажется сразу список, а в Python
3 «map object», который мы превращаем в список с помощью list()
```

Несмотря на то, что подобный пример вполне рабочий, читается он тяжело и использование синтаксиса генераторов выражений будет более предпочительным и понятным.

4. Выражения-генераторы

Выражения-генераторы (generator expressions) доступны, начиная с Python 2.4. Основное их отличие от генераторов коллекций в том, что они **выдают элемент по-одному, не загружая в память сразу всю коллекцию**.

UPD: Еще раз обратите внимание на этот момент: если мы создаем большую структуру данных без использования генератора, то она загружается в память целиком, соответственно, это увеличивает **расход памяти** Вашим приложением, а в крайних случаях памяти может просто не хватить и Ваше приложение **«упадет» с MemoryError**. В случае использования выражения-генератора, такого не происходит, так как элементы создаются по-одному, в момент обращения.

Пример выражения-генератора:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a)    # выражение-генератор
print(next(my_gen))     # -2 - получаем очередной элемент генератора
```

```
print(next(my_gen))     # -1 - получаем очередной элемент генератора
```

Особенности выражений-генераторов

Генаратор **нельзя писать без скобок** — это синтаксическая ошибка.

```
# my_gen = i for i in list_a      # SyntaxError: invalid syntax
```

При передаче в функцию дополнительные **скобки** необязательны

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_sum = sum(i for i in list_a)
# my_sum = sum((i for i in list_a))  # так тоже можно
print(my_sum)   # 12
```

Нельзя получить длину функцией **len**()

```
# my_len = len(i for i in list_a)  # TypeError: object of type 'generator' has no
len()
```

Нельзя распечатать элементы функцией **print**()

```
print(my_gen)   # <generator object <genexpr> at 0x7f162db32af0>
```

Обратите внимание, что после прохождения по выражению-генератору оно **остается пустым**!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a)
print(sum(my_gen))  # 12
print(sum(my_gen))  # 0
```

Выражение-генератор может быть **бесконечным**.

```
import itertools
inf_gen = (x for x in itertools.count())  # бесконечный генератор от 0 to
бесконечности!
```

Будьте осторожны в работе с такими генераторами, так как при не правильном использовании «эффект» будет как от бесконечного цикла.

К выражению-генератору **не применимы срезы**!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a)
my_gen_sliced = my_gen[1:3]
# TypeError: 'generator' object is not subscriptable
```

Из генератора легко получать **нужную коллекцию**. Это подробно рассматривается в следующей главе.

5. Генерация стандартных коллекций

5.1 Создание коллекций из выражения-генератора

Создание коллекций из выражения-генератора с помощью функций list(), tuple(), set(), frozenset()

Примечание: Так можно создать и неизменное множество и кортеж, так как неизменными они станет уже после генерации.

Внимание: Для строки такой способ не работает! Синтаксис создания генератора словаря таким образом имеет свои особенности, он рассмотрен в следующем под-разделе.

Передачей готового выражения-генератора присвоенного переменной в функцию создания коллекции.

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a)   # выражение-генератор
my_list = list(my_gen)
print(my_list)          # [-2, -1, 0, 1, 2, 3, 4, 5]
```

Написание выражения-генератора сразу внутри скобок вызываемой функции создания коллекции.

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_list = list(i for i in list_a)
print(my_list)          # [-2, -1, 0, 1, 2, 3, 4, 5]
```

То же самое для кортежа, множества и неизменного множества

5.2 Специальный синтаксис генераторов коллекций

В отличии от выражения-генератора, которое выдает значение по-одному, не загружая всю коллекцию в память, при использовании генераторов коллекций, коллекция генерируется сразу целиком.

Соответственно, вместо особенности выражений-генераторов перечисленных выше, такая коллекция будет обладать всеми стандартными свойствами характерными для коллекции данного типа.

Обратите внимание, что для генерации множества и словаря используются одинаковые скобки, разница в том, что у словаря указывается двойной элемент ключ: значение.

Генератор списка (list comprehension)

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_list = [i for i in list_a]
print(my_list)          # [-2, -1, 0, 1, 2, 3, 4, 5]
```

Не пишите круглые скобки в квадратных!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_list = [(i for i in list_a)]
print(my_list)          # [<generator object <genexpr> at 0x7fb81103bf68>]
```

Генератор множества (set comprehension)

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_set= {i for i in list_a}
print(my_set)       # {0, 1, 2, 3, 4, 5, -1, -2} - порядок случаен
```

Генератор словаря (dictionary comprehension)
переворачивание словаря

```
dict_abc = {'a': 1, 'b': 2, 'c': 3, 'd': 3}
dict_123 = {v: k for k, v in dict_abc.items()}
print(dict_123)  # {1: 'a', 2: 'b', 3: 'd'}
                 # Обратите внимание, мы потеряли "c"! Так как значения были
одинаковы,
                 # то когда они стали ключами, только последнее значение сохранилось.
```

Словарь из списка:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
dict_a = {x: x**2 for x in list_a}
print(dict_a)   # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, -2: 4, -1: 1, 5: 25}
```

Важно! Такой синтаксис создания словаря работает только в фигурных скобках, **выражение-генератор** так создать нельзя, для этого используется немного другой синтаксис (благодарю longclaps за подсказку в комментариях):

```
# dict_gen = (x: x**2 for x in list_a)      # SyntaxError: invalid syntax
dict_gen = ((x, x ** 2) for x in list_a)    # Корректный вариант генератора-выражения
для словаря
# dict_a = dict(x: x**2 for x in list_a)    # SyntaxError: invalid syntax
dict_a = dict((x, x ** 2) for x in list_a)  # Корректный вариант синтаксиса от
@longclaps
```

5.3 Генерация строк

Для создания строки вместо синтаксиса выражений-генераторов используется метод строки .**join**(), которому в качестве аргументов можно передать выражение генератор.
Обратите внимание: элементы коллекции для объединения в строку должны быть строками!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
```

```
# используем генератор прямо в .join() одновременно приводя элементы к строковому типу
my_str = ''.join(str(x) for x in list_a)
print(my_str)  # -2-1012345
```

6. Периодичность и частичный перебор

6.1 Работа с enumerate()

Иногда в условиях задачи в условии-фильтре нужна не проверка значения текущего элемента, а проверка на определенную периодичность, то есть, например, нужно брать каждый третий элемент.

Для подобных задач можно использовать функцию enumerate(), задающую счетчик при обходе итератора в цикле:

```
for i, x in enumerate(iterable)
```

здесь x — текущий элемент i — его порядковый номер, начиная с нуля

Проиллюстрируем работу с индексами:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_d = [(i, x) for i, x in enumerate(list_a)]
print(list_d)   # [(0, -2), (1, -1), (2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5)]
```

Теперь попробуем решить реальную задачу — выберем в генераторе списка каждый третий элемент из исходного списка:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_e = [x for i, x in enumerate(list_a, 1) if i % 3 == 0]
print(list_e)   # [0, 3]
```

Важные особенности работы функции enumerate():

Возможны два варианта вызова функции enumerate():
enumerate(iterator) без второго параметра считает с 0.
enumerate(iterator, start) — начинает считать с значения start. Удобно, например, если нам надо считать с 1, а не 0.

enumerate() возвращает **кортеж** из порядкового номера и значения текущего элемента итератора. Кортеж в выражении-генераторе результате можно получить двумя способами:
(i, j) for i, j in enumerate(iterator) — скобки в первой паре нужны!
pair for pair in enumerate(mylist) — мы работаем сразу с парой

Индексы считаются для **всех обработанных элементов**, без учета прошли они в дальнейшем условие или нет!

```
first_ten_even = [(i, x) for i, x in enumerate(range(10)) if x % 2 == 0]
print(first_ten_even)   # [(0, 0), (2, 2), (4, 4), (6, 6), (8, 8)]
```

Функция enumerate() не обращается к каким-то внутренним атрибутам коллекции, а просто реализует **счетчик обработанных элементов**, поэтому ничего не мешает ее использовать для неупорядоченных коллекций не имеющих индексации.

Если мы ограничиваем количество элементов включенных в результат по enumerate() счетчику (например if i < 10), то итератор будет все равно **обработан целиком**, что в случае огромной коллекции будет очень ресурс-затратно. Решение этой проблемы рассматривается ниже в под-разделе «Перебор части итерируемого».

6.2 Перебор части итерируемого.

Иногда бывает задача из очень большой коллекции или даже бесконечного генератора получить выборку первых нескольких элементов, удовлетворяющих условию.

Если мы используем обычное генераторное выражение с условием ограничением по enumerate() индексу или срез полученной результирующей коллекции, то нам в любом случае придется пройти всю огромную коллекцию и потратить на это уйму компьютерных ресурсов.

Выходом может быть использование функции **islice()** из пакета **itertools**.

```
import itertools
first_ten = (itertools.islice((x for x in range(1000000000) if x % 2 == 0), 10))
print(list(first_ten))  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Для сомневающихся: проверяем время выполнения

7. Вложенные циклы и генераторы

Рассмотрим более комплексные варианты, когда у нас циклы или сами выражения-генераторы являются вложенными. Тут возможны несколько вариантов, со своими особенностями и сферой применения, чтобы не возникало путаницы, рассмотрим их по-отдельности, а после приведем общую схему.

7.1 Вложенные циклы

В результате генерации получаем **одномерную структуру**.

Важно! При работае с вложенными циклами внутри генератора выражений порядок следования инструкций for in будет такой же (слева-направо), как и в аналогичном решении без генератора, только на циклах (сверху-вниз)! Тоже справедливо и при более глубоких уровнях вложенности.

7.1.1 Вложенные циклы for где циклы идут по независимым итераторам

Общий синтаксис: ***[expression for x in iter1 for y in iter2]***
Применение: генерируем одномерную структуру, используя данные из двух итераторов.

Например, создадим словарь, используя кортежи координат как ключи, заполнив для начала его значения нулями.

```
rows = 1, 2, 3
cols = 'a', 'b'
my_dict = {(col, row): 0 for row in rows for col in cols}
print(my_dict)  # {('a', 1): 0, ('b', 2): 0, ('b', 3): 0, ('b', 1): 0, ('a', 3): 0,
('a', 2): 0}
```

Дальше можем задавать новые значения или получать их

Тоже можно сделать и **с дополнительными условиями-фильтрами** в каждом цикле:

```
rows = 1, 2, 3, -4, -5
cols = 'a', 'b', 'abc'
# Для наглядности разнесем на несколько строк
my_dict = {
    (col, row): 0  # каждый элемент состоит из ключа-кортежа и нулевого знаечния
    for row in rows if row > 0   # Только положительные значения
    for col in cols if len(col) == 1  # Только односимвольные
    }
print(my_dict)  # {('a', 1): 0, ('b', 2): 0, ('b', 3): 0, ('b', 1): 0, ('a', 3): 0,
('a', 2): 0}
```

Эта же задача решенная с помощью цикла

7.1.2 Вложенные циклы for где внутренний цикл идет по результату внешнего цикла

Общий синтаксис: ***[expression for x in iterator for y in x]***.

Применение: Стандартный подход, когда нам надо обходить двумерную структуру данных, превращая ее в «плоскую» одномерную. В данном случае, мы во внешнем цикле проходим по строкам, а во внутреннем по элементам каждой строки нашей двумерной структуры.

Допустим у нас есть двумерная матрица — список списков. И мы желаем преобразовать ее в плоский одномерный список.

```
matrix = [[0, 1, 2, 3],
          [10, 11, 12, 13],
          [20, 21, 22, 23]]

# Решение с помощью генератора списка:
flattened = [n for row in matrix for n in row]
print(flattened)    # [0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23]
```

Таже задача, решенная с помощью вложенных циклов

UPD:Изящные решения из комментариев

7.2 Вложенные генераторы

Вложенными могут быть не только циклы for внутри выражения-генератора, но и сами генераторы.
Такой подход применяется когда нам надо **строить двумерную структуру**.

Важно!: В отличии от примеров выше с вложенными циклами, для вложенных генераторов, вначале обрабатывается внешний генератор, потом внутренний, то есть порядок идет справа-налево.

Ниже рассмотрим два варианта подобного использования.

7.2.1 — Вложенный генератор внутри генератора — двумерная из двух одномерных

Общий синтаксис: ***[[expression for y in iter2] for x in iter1]***
Применение: генерируем двумерную структуру, используя данные из двух одномерных итераторов.

Для примера создадим матрицу из 5 столбцов и 3 строк и заполним ее нулями:

```
w, h = 5, 3  # зададим ширину и высотку матрицы
matrix = [[0 for x in range(w)] for y in range(h)]
print(matrix)   # [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Создание этой же матрицы двумя вложенными циклами - обратите внимание на порядок вложения

Примечание: После создания можем работать с матрицей как с обычным двумерным массивом

7.2.2 — Вложенный генератор внутри генератора — двумерная из двумерной

Общий синтаксис: **[[expression for y in x] for x in iterator]**

Применение: Обходим двумерную структуру данных, сохраняя результат в другую двумерную структуру.

Возьмем матрицу:

```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Возведем каждый элемент матрицы в квадрат:

```
squared = [[cell**2 for cell in row] for row in matrix]
print(squared)    # [[1, 4, 9, 16], [25, 36, 49, 64], [81, 100, 121, 144]]
```

Эта же операция в виде вложенных циклов

Обобщим все вышеперечисленные варианты в одной схеме (полный размер по клику):

7.3 — Генератор итерирующийся по генератору

Так как любой генератор может использоваться как итератор в цикле for, это так же можно использовать и для создания генератора по генератору.
При этом синтаксически это может записываться в два выражения или объединяться во вложенный генератор.

Проиллюстрирую и такую возможность.
Допустим у нас есть два таких генератора списков:

```
list_a = [x for x in range(-2, 4)]    # Так сделано для дальнейшего примера синтаксиса,
                                      # конечно в подобной задаче досточно только range(-2, 4)
list_b = [x**2 for x in list_a]
```

Тоже самое можно записать и в одно выражение, подставив вместо list_a его генератор списка:

```
list_c = [x**2 for x in [x for x in range(-2, 4)]]
print(list_c)  # [4, 1, 0, 1, 4, 9]
```

UPD от longclaps: Преимущество от комбинирования генераторов на примере сложной функции f(x) = u(v(x))

```
list_c = [t + t ** 2  for t in (x ** 3 + x ** 4 for x in range(-2, 4))]
```

8. Использование range()

Говоря о способах генерации коллекций, нельзя обойти вниманием простую и очень удобную функцию range(), которая предназначена для создания арифметических последовательностей.

Особенности функции range():

Наиболее часто функция range() **применяется** для запуска цикла for нужное количество раз. Например, смотрите генерацию матрицы в примерах выше.

В Python 3 range() возвращает **генератор**, который при каждом к нему обращении выдает очередной элемент.

Исполльзуемые **параметры** аналогичны таковым в срезах (кроме первого примера с одним параметром):

range(stop) — в данном случае с 0 до stop-1;
range(start, stop) — Аналогично примеру выше, но можно задать начало отличное от нуля, можно и отрицательное;
range(start, stop, step) — Добавляем параметр шага, который может быть отрицательным, тогда перебор в обратном порядке.

В **Python 2** были 2 функции:

range(...) которая аналогична выражению list(range(...)) в Python 3 — то есть она выдавала не итератор, а сразу готовый список. То есть все проблемы возможной **нехватки памяти**, описанные в разделе 4 актуальны, и использовать ее в Python 2 надо очень аккуратно!
xrange(...) — которая работала аналогично range(...) в Python 3 и из 3 версии была исключена.

Примеры использования:

```
print(list(range(5)))           # [0, 1, 2, 3, 4]
print(list(range(-2, 5)))       # [-2, -1, 0, 1, 2, 3, 4]
print(list(range(5, -2, -2)))   # [5, 3, 1, -1]
```

9. Приложение 1. Дополнительные примеры

9.1 Последовательный проход по нескольким спискам

```
import itertools
l1 = [1,2,3]
l2 = [10,20,30]
result = [l*2 for l in itertools.chain(l1, l2)]
print(result)   # [2, 4, 6, 20, 40, 60]
```

9.2 Транспозиция матрицы
(Преобразование матрицы, когда строки меняются местами со столбцами).

Возьмем матрицу.

```
matrix = [[1, 2, 3, 4],
          [5, 6, 7, 8],
          [9, 10, 11, 12]]
```

Сделаем ее транспозицию с помощью генератора выражений:

```
transposed = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
print(transposed)  # [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Эта же транспозиция матрицы в виде цикла

И немного черной магии от @longclaps

9.3 Задача выбора только рабочих дней

```
# Формируем список дней от 1 до 31 с которым будем работать
days = [d for d in range(1, 32)]

# Делим список дней на недели
weeks = [days[i:i+7] for i in range(0, len(days), 7)]
print(weeks)   # [[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14], [15, 16, 17, 18,
19, 20, 21], [22, 23, 24, 25, 26, 27, 28], [29, 30, 31]]

# Выбираем в каждой неделе только первые 5 рабочих дней, отбрасывая остальные
work_weeks = [week[0:5] for week in weeks]
print(work_weeks)   # [[1, 2, 3, 4, 5], [8, 9, 10, 11, 12], [15, 16, 17, 18, 19],
[22, 23, 24, 25, 26], [29, 30, 31]]

# Если нужно одним списком дней - можно объединить
wdays = [item for sublist in work_weeks for item in sublist]
print(wdays)   # [1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 22, 23, 24,
25, 26, 29, 30, 31]
```

Можно убрать выходные еще более изящно, используя только индексы

10. Приложение 2. Ссылки по теме

Хорошая англоязычная статья с детальным объяснением что такое генераторы и итераторы

Иллюстрация из статьи:

Если у Вас есть сложности с пониманием логики работы с генераторными выражениями, посмотрите интересную англоязычную статью, где проводятся аналогии между генераторными выражениями и работой с SQL и таблицами Excel.

Например так:

UPD от fireSparrow: Существует расширение Python — PythonQL, позволяющее работать с базами данных в стиле генераторов коллекций.

Иллюстрированная статья на английском, довольно наглядно показывает синтаксис генераторных выражений.

Если требуются дополнительные примеры по теме вложенных генераторных выражений (статья на английском).