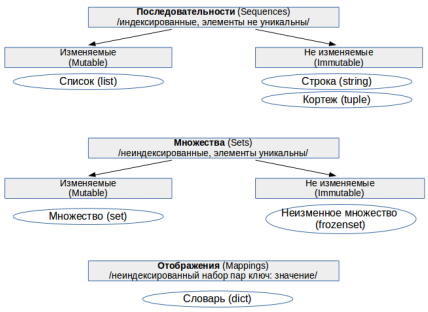
DaneSoul 9 января 2017 в 14:22

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

Python, Программирование

Из песочницы | Tutorial

Часть 1	Часть 2	Часть 3	Часть 4



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.

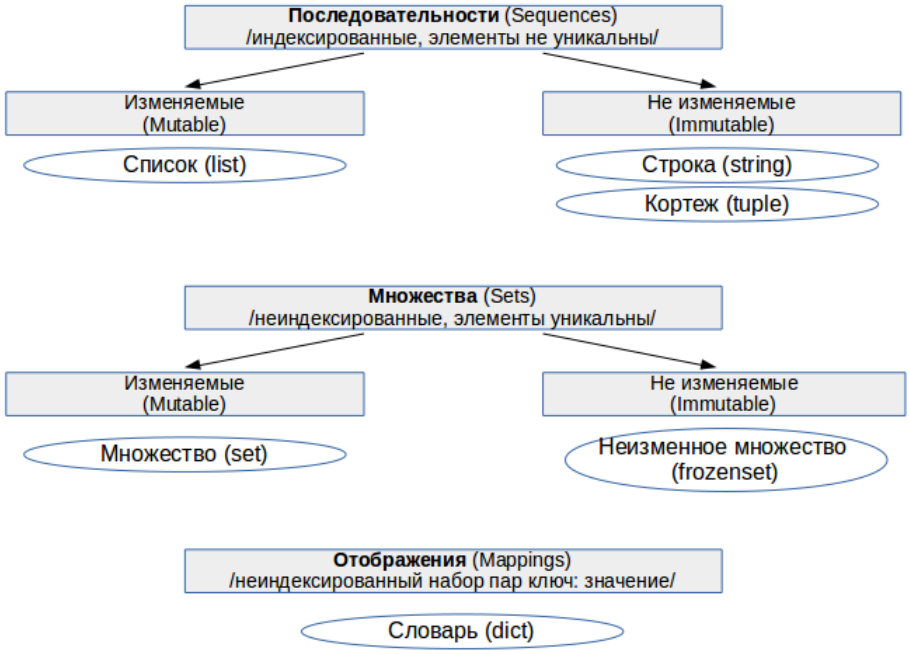
Вот именно эту проблему я хочу попытаться решить в данном цикле статей – рассмотреть ряд подходов к работе со стандартными коллекциями в Python в сравнении между коллекциями разных типов, а не по отдельности, как это обычно показывается в обучающих материалах. Кроме того, постараюсь затронуть некоторые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

- 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)

Часть 1	Часть 2	Часть 3	Часть 4

Приглашаю к обсуждению:

- Если я где-то допустил неточность или не учёл что-то важное — пишите в комментариях, важные комментарии будут позже добавлены в статью с указанием вашего авторства.
- Если какие-то моменты не понятны и требуется уточнение — пишите ваши вопросы в комментариях — или я или другие читатели дадут ответ, а дельные вопросы с ответами будут позже добавлены в статью.

Теги: python, программирование, начинающим, коллекции, структуры данных

Хабы: Python, Программирование