DaneSoul 19 января 2017 в 00:00

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

Python, Программирование

Tutorial

Часть 1	Часть 2	Часть 3	Часть 4

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.

1. **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)
```

2. 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]]
```

3. 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]
```

4. 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():

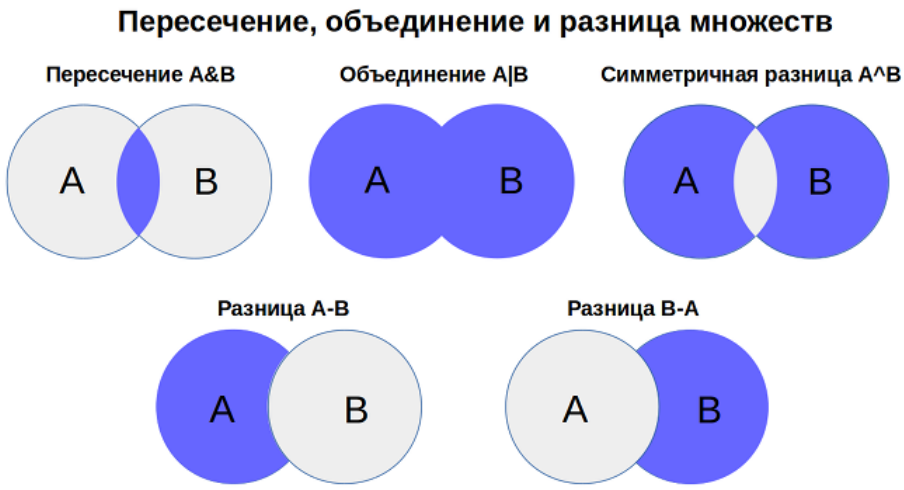
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = dict1.copy()
dict3.update(dict2)
print(dict3)            # {'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

   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
```

1. **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'}
```

2. **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'}
```

3. **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'}
```

4. **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

1. 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]
   ```

2. 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}
   ```

3. 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

1. **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
```

2. **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
```


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, объединение, модификация

Хабы: Python, Программирование