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. Стандартные встроенные коллекционные типы данных в Python: список (list), кортеж (tuple), строку (string), множества (set, frozenset), словарь (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. 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. 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).

Печать элементов коллекции с помощью функции print()

In [1]:

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

```
['a', 'b', 'c', 'd', 'e', 'f']
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
```

In [2]:

```
#Подсчёт количества членов коллекции с помощью функции len()
my_list = ['a', 'b', 'c', 'd', 'e', 'f']
my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
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
```

```
6
6
4
```

In [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
```

```
True
False
False
True
```

Для словаря возможны слежующие варианты:

In [4]:

```
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
```

```
True
True
False
True
```

Проверяем пары:

In [5]:

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

```
True
False
```

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

In [6]:

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

```
True
```

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.

In [8]:

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f']
for elm in my_list:
    print(elm)
```

```
a
b
c
d
e
f
```

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

In [19]:

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

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

```
a
b
c
d
e
f
1
2
3
4
5
6
```

Чаще всего нужны пары ключ(key) — значение (value)

In [21]:

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

```
a 1
b 2
c 3
d 4
e 5
f 6
```

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:

In [36]:

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

```
['a', 'b', 'c', 'd', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

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.

In [37]:

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

```
a
21
```

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

In [38]:

```
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
```

```
4
0
```

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

In [40]:

```
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! Вставьте в код
#соответствующий обраюотчик исключений
```

```
1

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-40-43ae598428f0> in <module>
      1 my_list = [1, 2, 2, 2, 2, 3]
      2 print(my_list.index(2))  # первый элемент равный 2 находится по индексу 1 (индексация с нуля!)
----> 3 print(my_list.index(5))  # ValueError: 5 is not in list - отсутствующий элемент выдаст ошибку! Вставьте в код
      4 #соответствующий обраюотчик исключений

ValueError: 5 is not in list
```

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

In [41]:

```
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
```

```
True
False
```

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

In [42]:

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

```
{1, 2, 3}
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

In [43]:

```
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 - н
адмножество
```

```
True
True
True
```

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

In [44]:

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

```
True
True
```

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).

In [45]:

```
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'})
```

```
['a', 'b', 'a'] {'a', 'b'} frozenset({'a', 'b'})
```

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():

In [46]:

```
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}
```

```
{'a': 1, 'b': 2}
```

Создаем строку из другой коллекции.

In [47]:

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

```
abc
```

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!

In [48]:

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

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-48-fa2d1e84821c> in <module>
      1 my_list = [1, [2, 3], 4]
----> 2 my_set = set(my_list)   # TypeError: unhashable type: 'list'

TypeError: unhashable type: 'list'
```

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). 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).

In [51]:

```
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
```

```
a
e
e
d
```

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

In [52]:

```
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, 2, 3]
1
c
```

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:

In [53]:

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

```
1

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
t)
<ipython-input-53-b4d0623aad98> in <module>
      1 my_tuple = (1, 2, 3, 4, 5)
      2 print(my_tuple[0])        # 1
----> 3 my_tuple[0] = 100         # TypeError: 'tuple' object does not sup
port item assignment

TypeError: 'tuple' object does not support item assignment
```

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

In [54]:

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

```
[10, 2, 3, [40, 5]]
```

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

In [56]:

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call las
t)
<ipython-input-56-35e5d0734f80> in <module>
      1 my_list = [1, 2, 3, 4, 5]
----> 2 my_list[5] = 6      # IndexError: list assignment index out of ran
ge. Обработайте данную ошибку

IndexError: list assignment index out of range
```

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! Выполните произвольный срез в созданной коллекции. 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).

In [58]:

```
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)
```

```
('Alex', 'Smith')
('May', 10, 1980)
```

In [59]:

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

```
[2, 4, 6]
```

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

In [61]:

```
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]
```

```
[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():

In [62]:

```
#срез аналоги
my_list = [1, 2, 3, 4, 5]
my_list[5:] = [6]      # вставляем в конец – лучше использовать .append(6)
print(my_list)         # [1, 2, 3, 4, 5, 6]
my_list[0:0] = [0]     # вставляем в начало – лучше использовать .insert(0, 0)
print(my_list)         # [0, 1, 2, 3, 4, 5, 6]
my_list[3:3] = [25]    # вставляем между элементами – лучше использовать .insert(3, 25)
print(my_list)         # [0, 1, 2, 25, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 25, 3, 4, 5, 6]
```

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

In [63]:

```
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]
```

```
[1, 20, 30, 4, 5]
[1, 0, 4, 5]
[1, 0, 40, 50, 60]
```

Можно просто удалить часть последовательности

In [64]:

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

```
[3, 4, 5]
```

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: (реализуйте обработчик ошибки)

In [66]:

```
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
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
<ipython-input-66-621b38b27be6> in <module>
      1 my_list = [1, 2, 3, 4, 5]
----> 2 print(my_list[-10])       # IndexError: list index out of range
      3 print(my_list[10])        # IndexError: list index out of range

IndexError: list index out of range
```

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

In [67]:

```
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
```

```
[1, 2, 3, 4, 5]
[]
[]
```

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. 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.

In [68]:

```
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]
```

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

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

In [69]:

```
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']
```

```
['pc.png', 'apple.bmp', 'mydog.gif', 'somecat.jpg']
```

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».

In [70]:

```
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
```

```
<list_reverseiterator object at 0x0000000004F6E2B0>
[3, 7, 1, 5, 2]
[3, 7, 1, 5, 2]
```

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.

In [71]:

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

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

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

In [72]:

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

```
None
```

Особенности сортировки словаря

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

In [73]:

```
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]
```

```
['a', 'b', 'c', 'd', 'e', 'f']
[('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5), ('f', 6)]
[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.

In [74]:

```
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', 2425
6800)]
```

```
[('Delhi', 16787941), ('Beijing', 21516000), ('Karachi', 23500000), ('Shan
ghai', 24256800)]
```

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.

In [77]:

```
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)
for det, price in shop:
    print('{:<10} цена: {:>5}р.'.format(det, price))

# каретка    цена:  1200р.
# обод       цена:  2000р.
# педаль     цена:   100р.
# рама       цена: 12000р.
# седло      цена:  2700р.
# седло      цена:  1500р.
# седло      цена:   300р.
# шатун      цена:  1000р.
# шатун      цена:   200р.
```

```
каретка    цена:  1200р.
обод       цена:  2000р.
педаль     цена:   100р.
рама       цена: 12000р.
седло      цена:  2700р.
седло      цена:  1500р.
седло      цена:   300р.
шатун      цена:  1000р.
шатун      цена:   200р.
```

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.

In [83]:

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

shop.sort(key=lambda x: (x[0], -x[1]))
for det, price in shop:
    print('{:<10} цена: {:>5}р.'.format(det, price))
```

```
каретка    цена:  1200р.
обод       цена:  2000р.
педаль     цена:   100р.
рама       цена: 12000р.
седло      цена:  2700р.
седло      цена:  1500р.
седло      цена:   300р.
шатун      цена:  1000р.
шатун      цена:   200р.
```

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 «+»

In [86]:

```
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)
```

```
abcde
(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):

In [87]:

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

```
[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):

In [88]:

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

```
[1, 2, 3] [4, 5] [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():

In [89]:

```
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}
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

В Питоне 3.5 появился новый более изящный способ:

In [90]:

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

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

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). Создайте три своих множества и выполните над ними: Объединение, Пересечение, Разницу и симметричную разницу

In [96]:

```
# 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
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 п
ересечению —
#выбирает элементы из обеих множеств которые не пересекаются, то есть все кроме совпада
ющих:
c = b.symmetric_difference(a)
# c = a.symmetric_difference(b)       # даст такой же результат
c = b ^ a                             # Альтернативная форма записи симметричной разниц
ы
print(c)                              # {'a', 'c'}
```

```
{'a', 'b', 'c'}
{'b'}
{'b'}
{'a'}
{'c'}
{'a', 'c'}
```

Задания: №1 создайте коллекцию на своюодную тему и выполните: №2 подсчет количества членов коллекции с помощью функции len() №3 проверку принадлежности элемента данной коллекции с помощью оператора in №4 выполните поиск подстроки №5 обход коллекции с применением оператора цикла №6 найдите максимальный, минимальный элементы коллекции и сумму элементов №7 найдите количество определленого пользователем элемента коллекции №8 выполните конвертацию типа созданной вами коллекции №9 вытоплните сортировку элементов коллекции №10 *реализовать любые два практических задания из темы словари с применеием коллекции №11* реализовать обюые два задания из практической работы по спискам и кортежам с применеием коллекций

In [ ]:

```

```