



Урок 5

Коллекции. Список. Очередь. Словарь

Понятие коллекции. Основные типы коллекций. Стандартные методы работы с коллекциями. Примеры применения коллекций для решения практических задач.

[Введение](#)

[Понятие коллекции](#)

[Основные типы коллекций](#)

[Counter](#)

[Deque](#)

[Defaultdict](#)

[OrderedDict](#)

[Namedtuple](#)

[Стандартные методы работы с коллекциями](#)

[Конвертация одного типа коллекции в другой](#)

[Примеры применения коллекций для решения практических задач](#)

[Пример: Задача 1. Программа сложения и умножения комплексных чисел](#)

[Пример: Задача 2. Определить студентов с баллом выше среднего](#)

[Пример: Задача 3. Изменение данных о товарах](#)

[Пример: Задача 4. Принадлежит ли дата диапазону времени](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

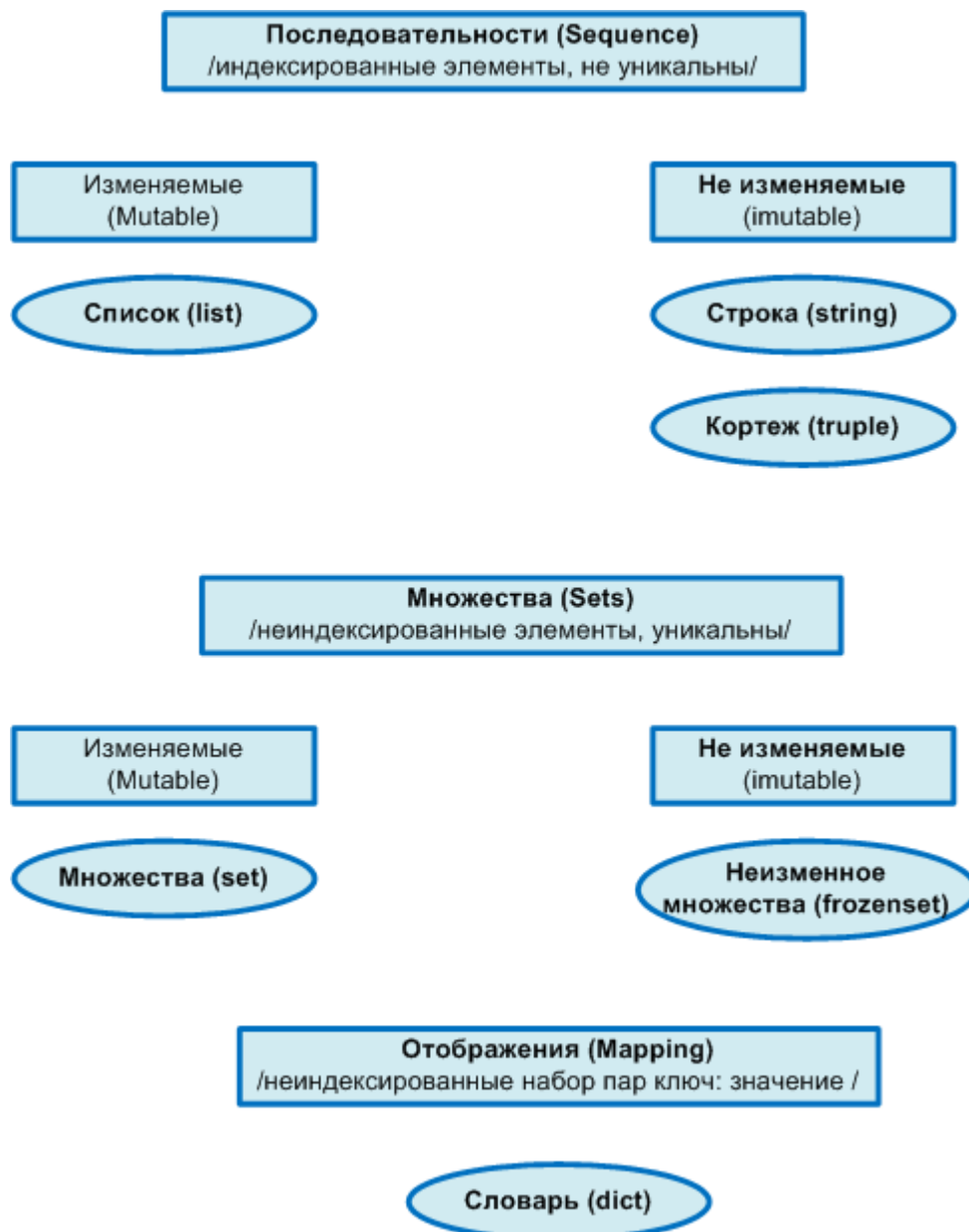
В предыдущем уроке мы изучили основные структуры представления данных и алгоритмы их обработки. Сегодня рассмотрим коллекции в Python.

Понятие коллекции

Коллекция – это обобщенный класс, содержащий набор свойств (полей) одного или разных типов, при этом позволяющий работать с ними и использовать их в специальных функциях и методах в зависимости от ее типа.

При изучении этого обобщенного класса важно понимать, что одну и ту же задачу можно решить с использованием разных коллекций.

Именно поэтому мы рассмотрим несколько стандартных коллекций и разберем общие методы работы с ними. Для начала распределим коллекции по изменяемости, последовательности распределения и ограниченности размера.



В Python специализированным типам коллекций предоставлен стандартный модуль Collections. Все специализированные типы так или иначе основаны на стандартных: словарях, кортежах, множествах, списках.

Основные типы коллекций

Рассмотрим типы коллекций и их особенности в сравнении с обычными видами структур хранения данных.

Counter

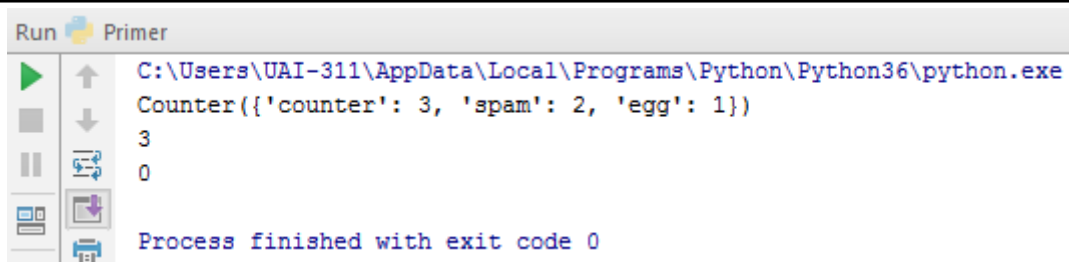
Эта коллекция создана на основе классического словаря – поименованной неупорядоченной последовательности неизменяемых данных, каждый элемент которой имеет уникальный ключ.

Counter позволяет считать количество неизменяемых объектов (в большинстве случаев, [строк](#)).

Пример:

```
import collections

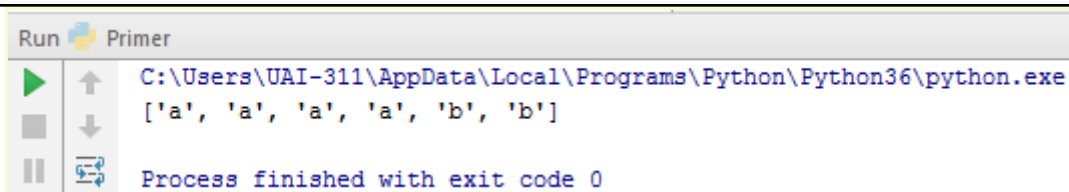
counter = collections.Counter()
for word in ['spam', 'egg', 'spam', 'counter', 'counter', 'counter']:
    counter[word] += 1
print(counter)
print(counter['counter'])
print(counter['collections'])
```



Но возможности Counter на этом не заканчиваются. У нее есть несколько специальных методов:

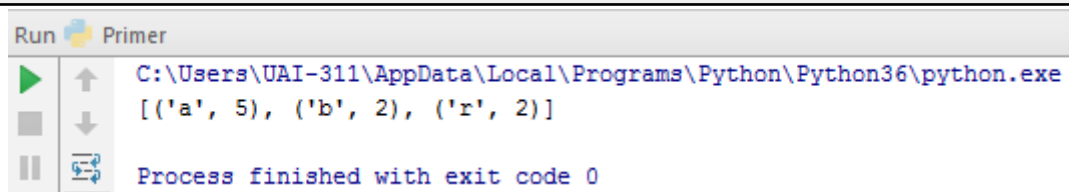
1. **elements()** – возвращает список элементов в порядке добавления ключей. В Python <3.6 возврат происходил в лексикографическом (алфавитном) порядке. При этом всегда учитываются только те ключи, чей вес больше нуля.

```
counter = collections.Counter(a=4, b=2, c=0, d=-2)
print(list(counter.elements()))
```



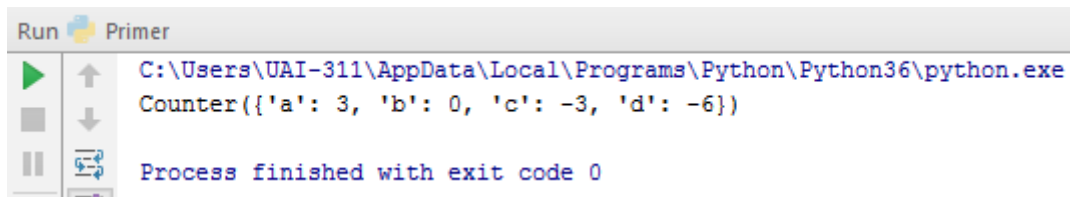
2. **most_common([n])** – возвращает n наиболее часто встречающихся элементов, в порядке убывания частотности. Если n не указано, возвращаются все элементы.

```
print(collections.Counter('abracadabra').most_common(3))
```



3. **subtract([iterable-or-mapping])** – производит вычитание, определяя уменьшаемое и вычитаемое по соотношению ключей элементов словаря.

```
counter = collections.Counter(a=4, b=2, c=0, d=-2)
counter2 = collections.Counter(a=1, b=2, c=3, d=4)
counter.subtract(counter2)
print(counter)
```



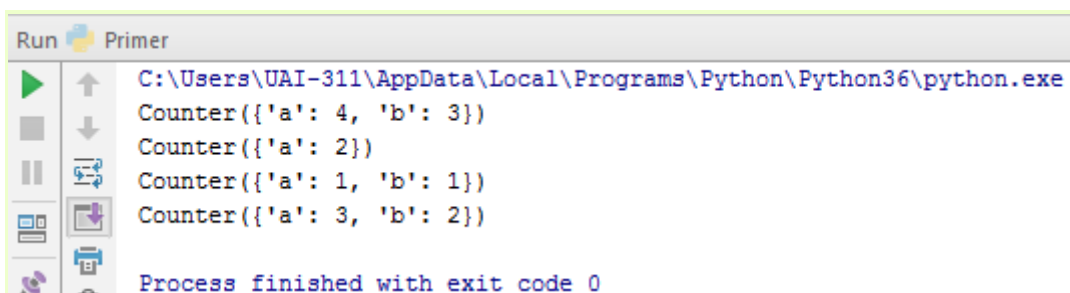
```
Run Primer
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
Process finished with exit code 0
```

Наиболее употребляемые шаблоны для работы с Counter:

- **sum(counter.values())** – показывает общее количество элементов словаря;
- **counter.clear()** – очищает счетчик словаря;
- **list(counter)** – возвращает список уникальных элементов словаря;
- **set(counter)** – преобразовывает словарь в множество;
- **dict(counter)** – преобразовывает в классический тип словаря;
- **counter.most_common()[:n:-1]** – возвращает n наименее часто встречающихся элементов;
- **counter += Counter()** – позволяет удалить элементы, встречающиеся менее одного раза.

Counter также поддерживает сложение, вычитание, пересечение и объединение:

```
counter = collections.Counter(a=3, b=1)
counter2 = collections.Counter(a=1, b=2)
print(counter + counter2)
print(counter - counter2)
print(counter & counter2)
print(counter | counter2)
```



```
Run Primer
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
Counter({'a': 4, 'b': 3})
Counter({'a': 2})
Counter({'a': 1, 'b': 1})
Counter({'a': 3, 'b': 2})
Process finished with exit code 0
```

Deque

В повседневной жизни мы зачастую имеем дело с очередями: в магазине, при записи на прием к врачу, при ожидании поезда метро, движение которого тоже связано с очередностью.

Очередь имеет свойство самоорганизации. Чтобы заплатить за товар, человек подходит к «хвосту» очереди в кассу и интересуется, кто последний. Заняв свое место, он ожидает, когда станет первым и заплатит за покупку. Кроме того, к нему может подойти другой покупатель и встать за ним.

Особенно наглядно самоорганизация прослеживается на примере очереди к врачу. Пациенты располагаются на стульях перед кабинетом, но порядок их «рассадки» чаще всего не совпадает с их местом в очереди. Таким образом, еще одним важным свойством очереди является произвольное расположение элементов, при котором порядок их следования поддерживается за счет связи между ними.

Основное отличие коллекции Deque от классической очереди в том, что можно производить добавление элемента в начало очереди.

Deque(iterable, [maxlen]) – коллекция создает очередь из итерируемого объекта с максимальной длиной **maxlen**. В основе организации лежит список, но добавлять и удалять элементы можно либо справа, либо слева.

Методы, определенные в коллекции deque:

- **append(x)** – добавляет элемент x в конец очереди;
- **appendleft(x)** – добавляет элемент x в начало очереди;
- **clear()** – очищает очередь;
- **count(x)** – возвращает количество элементов очереди, равных x;
- **extend(iterable)** – добавляет в конец очереди все элементы iterable;
- **extendleft(iterable)** – добавляет в начало очереди все элементы iterable (начиная с последнего);
- **pop()** – удаляет и возвращает последний элемент очереди;
- **popleft()** – удаляет и возвращает первый элемент очереди;
- **remove(value)** – удаляет первое вхождение value в очереди;
- **reverse()** – разворачивает очередь;
- **rotate(n)** – последовательно переносит n элементов из начала в конец (если n отрицательно, то с конца в начало).

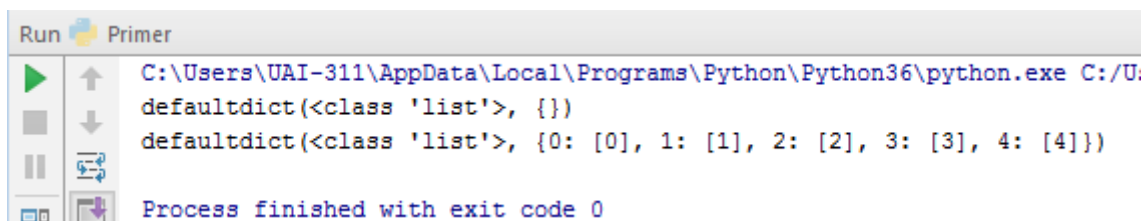
Defaultdict

В основе данной коллекции лежит словарь. Рассмотрим еще одну его специфическую реализацию.

Defaultdict ничем не отличается от обычного словаря за исключением того, что по умолчанию всегда вызывается функция, возвращающая значение:

```
defdict = collections.defaultdict(list)
print(defdict)

for i in range(5):
    defdict[i].append(i)
print(defdict)
```



```
Run Primer
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/U:
defaultdict(<class 'list'>, {})
defaultdict(<class 'list'>, {0: [0], 1: [1], 2: [2], 3: [3], 4: [4]})

Process finished with exit code 0
```

OrderedDict

Еще одна коллекция на базе словаря. Она помнит порядок, в котором были даны ключи.

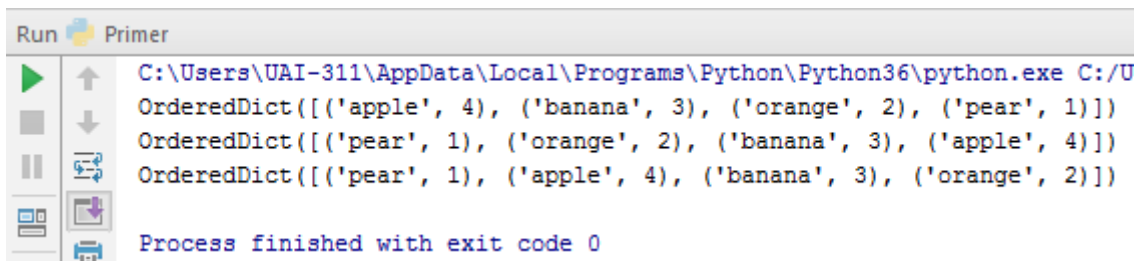
Основные методы работы:

- **popitem(last=True)** – удаляет последний элемент, если last=True, и первый, если last=False;
- **move_to_end(key, last=True)** – добавляет ключ в конец, если last=True, и в начало, если last=False.

```
d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}
print(collections.OrderedDict(sorted(d.items(), key=lambda t: t[0])))

print(collections.OrderedDict(sorted(d.items(), key=lambda t: t[1])))

print(collections.OrderedDict(sorted(d.items(), key=lambda t: len(t[0]))))
```



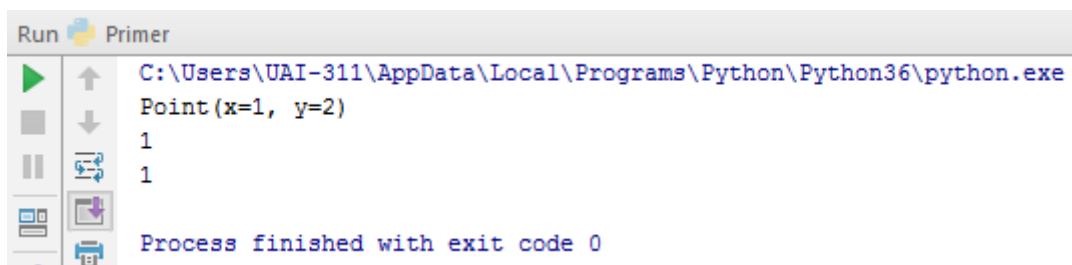
```
Run Primer
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/U
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])
OrderedDict([('pear', 1), ('apple', 4), ('banana', 3), ('orange', 2)])
Process finished with exit code 0
```

Namedtuple

В основе этой коллекции лежит организация кортежа.

Namedtuple позволяет создать тип данных, ведущий себя как кортеж. При этом каждому элементу присваивается имя, по которому можно в дальнейшем получать доступ:

```
Point = collections.namedtuple('Point', ['x', 'y'])
p = Point(x=1, y=2)
print(p)
Point(x=1, y=2)
print(p.x)
print(p[0])
```



```
Run Primer
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
Point(x=1, y=2)
1
1
Process finished with exit code 0
```


Стандартные методы работы с коллекциями

Мы разобрали основные классические виды структур хранения данных и их дополненные реализации с помощью коллекций. Теперь освоим основные методы работы с коллекциями и их организацию.

Ряд методов у коллекционных типов может использоваться в нескольких коллекциях для решения задач одного типа.

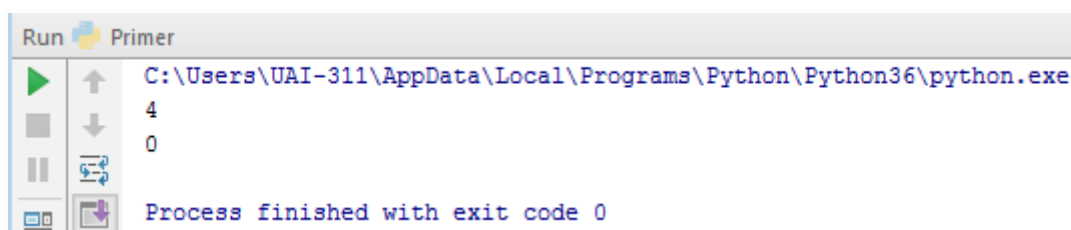
Применимости методов в зависимости от типа коллекции

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)	-	-	+	+
Неизменяемое множество (frozenset)	-	-	+	-
Словарь (dict)	-	-	+	+

1. Первый метод – **count()**:

.count() – метод подсчета определенных элементов для неуникальных коллекций (строка, список, кортеж). Возвращает сведения о том, сколько раз элемент встречается в коллекции.

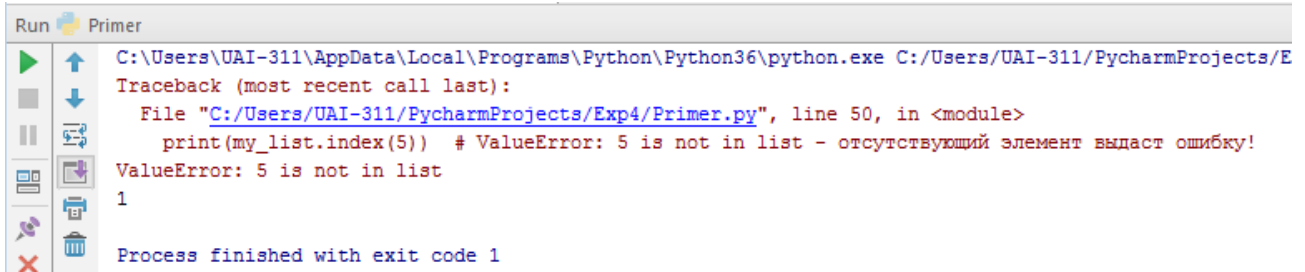
```
my_list = [1, 2, 2, 2, 2, 3]
print(my_list.count(2))      # 4 экземпляра элемента равного 2
print(my_list.count(5))      # 0 - то есть такого элемента в коллекции нет
```



2. Второй метод – **index()**:

.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!
```



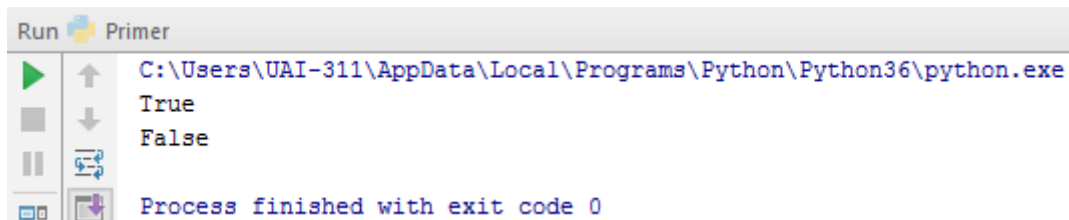
Run Primer

```
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/Users/UAI-311/PycharmProjects/E
Traceback (most recent call last):
  File "C:/Users/UAI-311/PycharmProjects/Exp4/Primer.py", line 50, in <module>
    print(my_list.index(5)) # ValueError: 5 is not in list - отсутствующий элемент выдаст ошибку!
ValueError: 5 is not in list
1
Process finished with exit code 1
```

3. Третий метод – `copy()`:

`.copy()` – метод возвращает неглубокую (нерекурсивную) 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
```



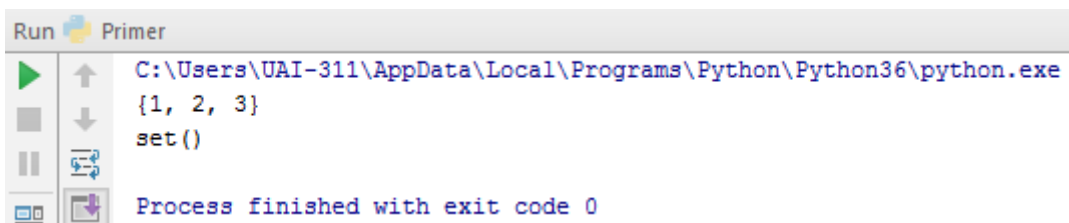
Run Primer

```
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
True
False
Process finished with exit code 0
```

4. Четвертый метод – `clear()`:

`.clear()` – метод изменяемых коллекций (список, словарь, множество), удаляющий из коллекции все элементы (превращающий ее в пустую).

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



Run Primer

```
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
{1, 2, 3}
set()
Process finished with exit code 0
```

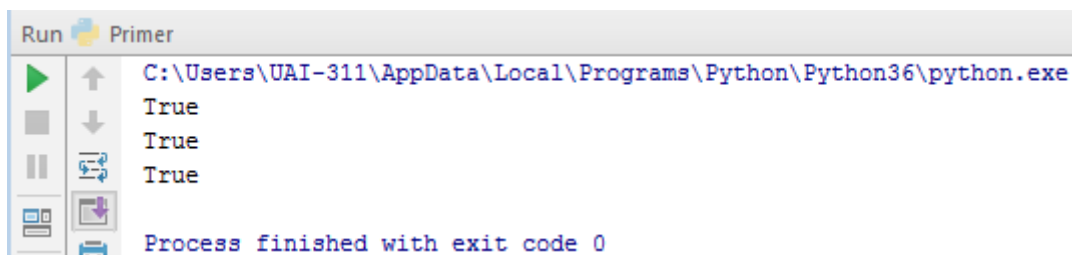
Также существуют особые методы сравнения множеств: **set()** и **frozenset()**:

- **set_a.isdisjoint(set_b)** – возвращает истину (true), если **set_a** и **set_b** не имеют общих элементов. В противном случае – false;
- **set_b.issubset(set_a)** – если все элементы множества **set_b** принадлежат **set_a**, то **set_b** целиком входит в множество **set_a** и является его подмножеством;
- **set_a.issuperset(set_b)** – если условие выше справедливо, то **set_a** – надмножество.

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

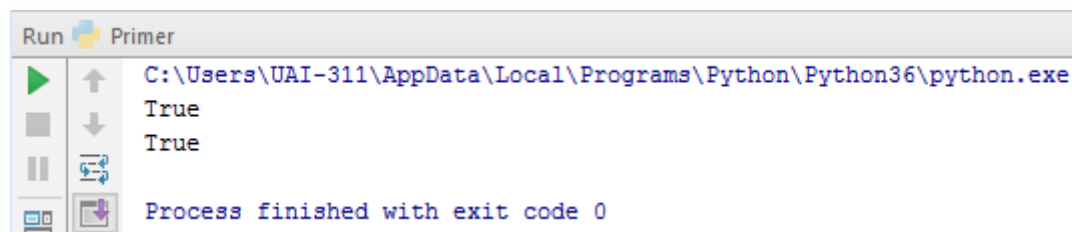
set_a = {1, 2, 3}
set_b = {2, 1}          # порядок элементов не важен!
set_c = {4}
set_d = {1, 2, 3}

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 являются одновременно и подмножеством, и надмножеством друг для друга.

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



Конвертация одного типа коллекции в другой

Для конвертации типа коллекции достаточно передать одну коллекцию в функцию создания другой (они представлены 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'})

```

```

Run Primer
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
['a', 'b', 'a'] {'b', 'a'} frozenset({'b', 'a'})
Process finished with exit code 0

```

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

1. При преобразовании в множество теряются дублирующие элементы, так как множество содержит только уникальные. Собственно множество обычно используется в задачах именно для проверки на уникальность;
2. При конвертации индексированной коллекции в неиндексированную теряется информация о порядке элементов. А ведь в некоторых случаях она может быть критически важной;
3. После конвертации в неизменяемый тип мы больше не сможем менять элементы коллекции: удалять, изменять, добавлять новые. Это может привести к ошибкам в функциях обработки данных, которые были написаны для работы с изменяемыми коллекциями.

Дополнительные детали:

Способом, описанным выше, не получится создать словарь, так как он состоит из пар «ключ-значение».

Это ограничение можно обойти: создать 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}

```

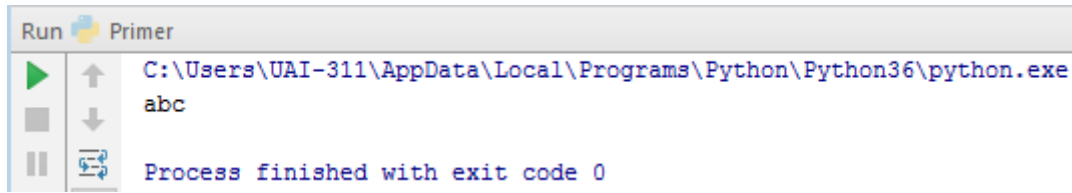
```

Run Primer
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
{'a': 1, 'b': 2}
Process finished with exit code 0

```

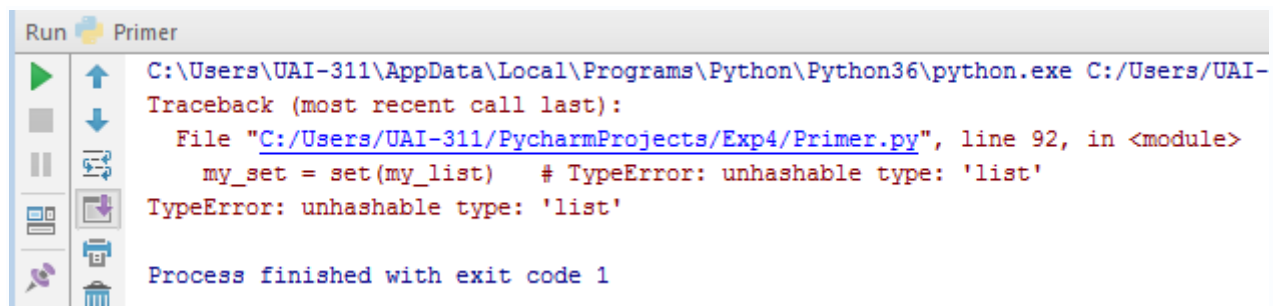
Создаем строку из другой коллекции:

```
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 неизменяемую, элементы которой могут быть только постоянными.

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



Примеры применения коллекций для решения практических задач

Пример: Задача 1. Программа сложения и умножения комплексных чисел

Используя конспект, написать программу сложения и умножения двух комплексных чисел.

Как мы помним из курса математики, комплексные числа состоят из двух частей: действительной и мнимой. Обе эти части являются вещественными.

Например: $4.5 + 0.9i$, где 4.5 – действительная часть, 0.9 – мнимая, а i – мнимая единица. Соответственно, обе части могут быть как положительными, так и отрицательными.

Допустим, наши исходные данные – это два комплексных числа. Для определения этих чисел задействуем переменные a и b , их действительные части обозначим как $a.x$ и $b.x$, а мнимые – $a.y$ и $b.y$.

В соответствии с математическими правилами, сумма и произведение определенных комплексных чисел будет выглядеть так:

$$a + b = (a.x + b.x) + (a.y + b.y)i$$

$$a * b = (a.x * b.x - a.y * b.y) + (a.x * b.y + a.y * b.x)i$$

Для описания комплексных чисел создадим структуру данных. Она будет содержать два поля, описывающих действительную и мнимую части.

Рассмотрим первый вариант реализации с использованием стандартного типа **complex**:

```
# Вариант 1. Использование встроенного типа данных complex:

a = input()
b = input()
a = complex(a)
b = complex(b)
suma = a + b
mult = a * b
print(suma)
print(mult)
```

Рассмотрим второй вариант реализации с использованием собственноручно созданного класса:

```
# Вариант 2. Определение собственного класса и перегрузка операторов:

class Cmplx:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, obj):
        self.sumax = self.x + obj.x
        self.sumay = self.y + obj.y
    def __mul__(self, obj):
        self.multx = self.x * obj.x - self.y * obj.y
        self.multy = self.y * obj.x + self.x * obj.y

x = float(input())
y = float(input())
a = Cmplx(x, y)
x = float(input())
y = float(input())
b = Cmplx(x, y)
a + b
a * b
print('Сумма: %.2f+%.2fj' % (a.sumax, a.sumay))
print('Произв.: %.2f+%.2fj' % (a.multx, a.multy))
```

И рассмотрим третий вариант – с использованием словаря:

```
# Вариант 3. Использование словарей:

a = {'x':0, 'y':0}
b = {'x':0, 'y':0}
a['x'] = float(input())
a['y'] = float(input())
```

```

b['x'] = float(input())
b['y'] = float(input())
suma = {}
mult = {}
suma['x'] = a['x'] + b['x']
suma['y'] = a['y'] + b['y']
mult['x'] = a['x'] * b['x'] - a['y'] * b['y']
mult['y'] = a['y'] * b['x'] + a['x'] * b['y']
print('Сумма: %.2f+%.2fj' % (suma['x'], suma['y']))
print('Произв.: %.2f+%.2fj' % (mult['x'], mult['y']))

```

Результат работы программы:

```

Run Exp4(Complex)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
4
5
(9+0j)
(20+0j)
5
7
1
2
Сумма: 6.00+9.00j
Произв.: -9.00+17.00j
4
5
7
8
Сумма: 11.00+13.00j
Произв.: -12.00+67.00j
Process finished with exit code 0

```

Пример: Задача 2. Определить студентов с баллом выше среднего

Необходимо реализовать программу, которая позволит работать с данными о студентах. Определим, что все данные будет вводить с клавиатуры пользователь.

Описание студента будет состоять из трех параметров: фамилии, имени и среднего балла.

Основная задача программы – определить лучших по успеваемости студентов, у которых балл выше среднего.

Для реализации этой задачи создадим свою собственную структуру Студент и массив объектов.

Алгоритм работы программы будет следующим:

1. Пользователь вводит количество студентов. Определяем размер массива;
2. В цикле заполняем массив студентов;

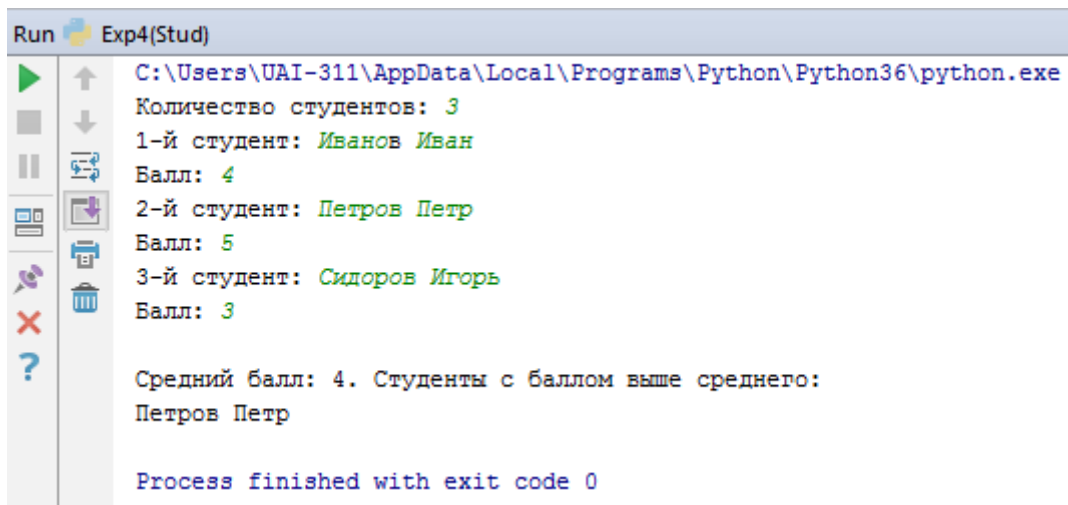
3. Вычисляем средний балл, посчитав сумму всех баллов студентов и разделив на количество учащихся;
4. В цикле перебираем всех студентов и сравниваем балл каждого со средним значением. Если он больше среднего, то выводим этого студента.

Программная реализация:

```
studs = {}
n = int(input("Количество студентов: "))
s = 0
for i in range(n):
    sname = input(str(i+1) + "-й студент: ")
    point = int(input("Балл: "))
    studs[sname] = point
    s += point

avrg = s / n
print("\nСредний балл: %.0f. Студенты с баллом выше среднего:" % avrg)
for i in studs:
    if studs[i] > avrg:
        print(i)
```

Работа программы:



```
Run Exp4(Stud)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
Количество студентов: 3
1-й студент: Иванов Иван
Балл: 4
2-й студент: Петров Петр
Балл: 5
3-й студент: Сидоров Игорь
Балл: 3

Средний балл: 4. Студенты с баллом выше среднего:
Петров Петр

Process finished with exit code 0
```

Пример: Задача 3. Изменение данных о товарах

Необходимо написать программу, которая будет работать с информацией о товарах, хранящихся на складе. Определим, что все данные вводятся с клавиатуры.

Программа должна выводить всю текущую информацию о товарах. Они описываются номером на складе и соответствующим количеством. Пока пользователь не введет 0, программа позволит вносить изменения в сведения о товарах.

После изменений программа также выводит текущую информацию о товарах на складе.

Для описания товаров создадим собственную структуру. Ее полями будут номер товара на складе и его количество. После этого создаем массив объектов – Товар.

Алгоритм работы программы:

1. Исходный массив товаров заполняется внутри программы;
2. Программа в цикле выводит всю информации о каждом объекте, содержащемся в массиве;
3. После создаем «бесконечный» цикл, внутри которого пользователь сможет по введенному номеру товара менять о нем информацию;
4. Как только пользователь решит, что больше никаких изменений о товарах он вносить не хочет, он вводит цифру 0. Срабатывает **break**;
5. В итоге программа выводит всю информацию о товарах.

Программная реализация:

```
goods = {'1': ['Core-i3-4330', 9, 4500],
'2': ['Core i5-4670K', 3, 8500],
'3': ['AMD FX-6300', 6, 3700],
'4': ['Pentium G3220', 8, 2100],
'5': ['Core i5-3450', 5, 6400]}

for i in goods:
    print("%s) %s - %d шт. по %d руб" % (i, goods[i][0], goods[i][1],
goods[i][2]))

while 1:
    n = input('№: ')
    if n != '0':
        qty = int(input('Количество: '))
        goods[n][1] = qty
    else:
        break

for i in goods:
    print("%s) %s - %d шт. по %d руб" % (i, goods[i][0], goods[i][1],
goods[i][2]))
```

Результат работы программы:

```
un Exp4(Goods)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
1) Core-i3-4330 - 9 шт. по 4500 руб
2) Core i5-4670K - 3 шт. по 8500 руб
3) AMD FX-6300 - 6 шт. по 3700 руб
4) Pentium G3220 - 8 шт. по 2100 руб
5) Core i5-3450 - 5 шт. по 6400 руб
  : 1
Количество: 7
  : 3
Количество: 8
  : 0
1) Core-i3-4330 - 7 шт. по 4500 руб
2) Core i5-4670K - 3 шт. по 8500 руб
3) AMD FX-6300 - 8 шт. по 3700 руб
4) Pentium G3220 - 8 шт. по 2100 руб
5) Core i5-3450 - 5 шт. по 6400 руб

Process finished with exit code 0
```

Не забываем, что в словаре нет порядковых номеров, именно поэтому для описания товаров потребовалось ввести поле номер товара на складе. Это поле по сути своей было ключом.

Пример: Задача 4. Принадлежит ли дата диапазону времени

Необходимо реализовать программу, которая позволит сравнивать даты и определять, попадает ли введенная пользователем дата в заданный временной интервал.

Для реализации создадим собственную структуру данных.

Алгоритм работы программы:

1. Исходный диапазон дат для проверки задается внутри программы. Далее пользователь вводит свою дату;
2. Организуем проверку. Сначала сверяем год. Если введенный год попадает в промежуток между двумя исходными, то можем вывести сообщение о том, что пользовательская дата попадает в заданный диапазон. Если год введенной даты равен крайнему году исходного диапазона – значит нам надо проверить месяц. Если месяц введенной даты меньше месяца крайней исходной даты – тоже можно вывести сообщение о попадании даты в интервал. Если же условия не выполняются – выводим сообщение о том, что дата не принадлежит заданному интервалу.

Программная реализация:

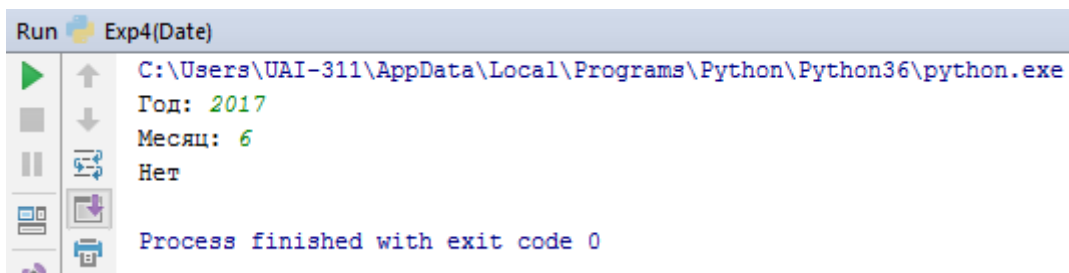
```
d1 = {'year': 2003, 'month': 12}
d2 = {'year': 2014, 'month': 6}
du = {}
du['year'] = int(input('Год: '))
du['month'] = int(input('Месяц: '))
if d1['year'] < du['year'] < d2['year']:
    print('Да')
elif du['year'] == d1['year']:
    if du['month'] >= d1['month']:
```

```

        print('Да')
    else:
        print('Нет')
elif du['year'] == d2['year']:
    if du['month'] <= d2['month']:
        print('Да')
    else:
        print('Нет')
else:
    print('Нет')

```

Результат работы программы:



Подведем итоги урока:

1. Все специализированные коллекции реализованы на стандартных типах данных: массивах, списках, кортежах, словарях;
2. Задачу можно решить с помощью различных коллекций. Важно понимать, какая именно коллекция подходит для выполнения конкретной задачи;
3. Не нужно заикливаться на каком-то одном специализированном типе данных.

Практическое задание

1. Пользователь вводит данные о количестве предприятий, их наименования и прибыль за 4 квартала (т.е. 4 отдельных числа) для каждого предприятия.. Программа должна определить среднюю прибыль (за год для всех предприятий) и вывести наименования предприятий, чья прибыль выше среднего и отдельно вывести наименования предприятий, чья прибыль ниже среднего.
2. Написать программу сложения и умножения двух шестнадцатеричных чисел. При этом каждое число представляется как коллекция, элементы которой это цифры числа. Например, пользователь ввёл A2 и C4F. Сохранить их как ['A', '2'] и ['C', '4', 'F'] соответственно. Сумма чисел из примера: ['C', 'F', '1'], произведение - ['7', 'C', '9', 'F', 'E'].

Примечание: для решения задач попробуйте применить какую-нибудь коллекцию из модуля collections (пусть это и не очевидно с первого раза. Вы же не Голландец ;-).

Дополнительные материалы

1. <http://www.intuit.ru/studies/courses/10/320/info>
2. https://compscicenter.ru/media/slides/python_2014_autumn/2014_10_15_python_2014_autumn.pdf

Используемая литература

1. <https://www.python.org>
2. <http://www.intuit.ru/studies/courses/10/320/info>
3. Марк Лутц. Изучаем Python, 4-е издание.