Практическая работа № 3

Коллекции

1. Списки

Изменяемы упорядоченные наборы элементов, могут быть разных типов

In [1]:

```
empty_list = []
empty_list = list()
none_list = [None] * 10
collections = ['list', 'tuple', 'dict', 'set']
user_data = [ ['Elena', 4.4], ['Andrey', 4.2] ]
```

Длина списка

In [2]:

```
len(collections)
```

Out[2]:

4

In [3]:

```
print(collections)
print(collections[0])
print(collections[-1])
```

```
['list', 'tuple', 'dict', 'set']
list
set
```

Присваиванит (изменения элементов):

In [4]:

```
collections[3] = 'frozenset'
print(collections)
```

```
['list', 'tuple', 'dict', 'frozenset']
```

Проверить, существует ли элемент в списке:

In [5]:

```
'tuple' in collections
```

Out[5]:

```
True
```

Срезы в списках работают точно так же, как и в строках. Создадим список из 10 элементов с помощью встроенной функции range и поэкспериментируем на нём со срезам

In [7]:

```
range_list = list(range(10))
print(range_list)

print(range_list[1:3])

print(range_list[::2])

print(range_list[::-1])

print(range_list[5:1:-1])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2]
[0, 2, 4, 6, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[5, 4, 3, 2]
```

Все коллекции (списки в том числе) поддерживают протокол итерации

In [10]:

```
collections = ['list', 'tuple', 'dict', 'set']
for collection in collections:
    print('Изучим {}!'.format(collection))
```

```
Изучим list!
Изучим tuple!
Изучим dict!
Изучим set!
```

Итерация по элементам (а не по индексам). А что, если нужен индекс?

In [11]:

```
for idx, collection in enumerate(collections):
        print('#{} {}'.format(idx, collection))
```

```
#0 list
#1 tuple
#2 dict
#3 set
```

Можно добавлять и уделять элементы

In [15]:

```
collections.append('OrderedDict')
print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'OrderedDict', 'OrderedDict', 'OrderedDic
t', 'OrderedDict']
```

Если нужно расширить список другим списком:

In [17]:

```
collections.extend(['ponyset', 'unicorndict'])
print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'OrderedDict', 'OrderedDict', 'OrderedDic
t', 'OrderedDict', 'ponyset', 'unicorndict', 'ponyset', 'unicorndict']
```

In [20]:

```
collections += [None, None]
print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'OrderedDict', 'OrderedDict', 'OrderedDic
t', 'OrderedDict', 'ponyset', 'unicorndict', 'ponyset', 'unicorndict', None,
None, None]
```

Удаление элемента из списка

In [22]:

```
del collections[4]
print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'OrderedDict', 'OrderedDict', 'ponyset', 'u
nicorndict', 'ponyset', 'unicorndict', None, None, None]
```

Преобразовать список в строку и обратно : *прим. работает только для строк*

In [40]:

```
fac_list = ['RTF', 'FAVT', 'EGF','FIB']
delimeter = ', '
fac_str = delimeter.join(fac_list)
print('строка: ', fac_str)
print(type(fac_str))
faclist_back = fac_str.split(delimeter)
print(faclist_back)
print(type(faclist_back))
```

```
строка:  RTF, FAVT, EGF, FIB
<class 'str'>
['RTF', 'FAVT', 'EGF', 'FIB']
<class 'list'>
[1, 2, 3, 4]
<class 'list'>
```

Сортировка. Сначала создадим список из случайных чисел

In [41]:

```
import random
numbers = []
for _ in range(10): # переменную для итерации называли _, т.к. сама по себе она нам не инте
    numbers.append(random.randint(1, 20))
print(numbers)
```

```
[16, 3, 13, 8, 16, 15, 20, 19, 11, 17]
```

Для сортировки списка в Python есть два способа: стандартная функция sorted, которая возвращает новый список, полученный сортировкой исходного, и метод списка .sort(), который сортирует in-place.

In [42]:

```
print(sorted(numbers))
print(numbers)
```

```
[3, 8, 11, 13, 15, 16, 16, 17, 19, 20]
[16, 3, 13, 8, 16, 15, 20, 19, 11, 17]
```

In [43]:

```
numbers.sort()
print(numbers)
```

```
[3, 8, 11, 13, 15, 16, 16, 17, 19, 20]
```

Если нужно в обратном порядке:

In [44]:

```
numbers.sort(reverse=True)
print(numbers)
```

```
[20, 19, 17, 16, 16, 15, 13, 11, 8, 3]
```

2. Кортежи (tuple)

Кортежи - это неизменяемые списки. Кортежи определяются с помощью круглых скобок или литерала tuple.

In [32]:

```
empty_tuple = ()
empty_tuple = tuple()
```

In [50]:

```
immutables = (int, str, tuple)
```

Попробуем изменить какой-нибудь элемент:

In [47]:

```
immutables[0] = float
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_12640\3783869503.py in <cell line: 1>()
----> 1 immutables[0] = float

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

Кортежи неизменяемые, объекты внутри них могут быть изменяемыми. Парадоксально, правда?

In [53]:

```
fac_tuple = (fac_list,[])
print(fac_tuple)
fac_tuple[0].append('FUES')
print(fac_tuple)
```

```
(['RTF', 'FAVT', 'EGF', 'FIB', 'FUES'], [])
(['RTF', 'FAVT', 'EGF', 'FIB', 'FUES', 'FUES'], [])
```

Давайте чуть подробнее разберемся, чем опасно не учитывать изменяемость и неизменяемость типа данных

Неизменяемые (immutable):

- None
- bool
- int

- float
- str
- tuple

Изменяемые (muttable)

- list
- dict
- set

In [73]:

```
num=1
id(num)
```

Out[73]:

```
2026854875376
```

In [80]:

```
num += 1
id(num)
```

Out[80]:

```
2026854875600
```

прим. Функция id() возвращает уникальный идентификатор для указанного объекта. Все объекты в Python имеют свой уникальный идентификатор. Идентификатор присваивается объекту при его создании. Идентификатор является адресом памяти объекта и будет отличаться при каждом запуске программы.

In [81]:

```
ex_str = 'Кот'
```

In [82]:

```
ex_str[-1] = 'д'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_12640\3313243715.py in <cell line: 1>()
----> 1 ex_str[-1] = 'д'

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

In [84]:

```
ex_str_2 = ex_str
print(id(ex_str))
print(id(ex_str_2))
```

```
2026935441360
2026935441360
```

In [86]:

```
ex_str += 'ан' # На самом деле создали новую переменную с тем же именем
print(ex_str)
print(id(ex_str))
```

```
Котанан
2026935436752
```

Так работает неизменяемость. А что с изменяемостью?

In [91]:

```
print(fac_list)
id(fac_list)
```

```
['RTF', 'FAVT', 'EGF', 'FIB', 'FUES', 'FUES']
```

Out[91]:

```
2026934518912
```

In [92]:

```
fac_list[-1] = 'MECHMAT'
fac_list
```

Out[92]:

```
['RTF', 'FAVT', 'EGF', 'FIB', 'FUES', 'MECHMAT']
```

In [93]:

```
id(fac_list)
```

Out[93]:

```
2026934518912
```

In [110]:

```
dir(list)
```

Out[110]:

```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

id не изменился, т.е. это та же переменная.

Теперь посмотрим, какие неявные ошибки могут возникнуть, если это не учитывать.

In [105]:

```
num = 1
def add(var):
    var += 1
    print(var)
```

In [106]:

```
id(num)
```

Out[106]:

```
2026854875376
```

In [107]:

```
add(num)
```

```
2
```

In [109]:

```
num
```

Out[109]:

```
1
```

In [108]:

```
id(num)
```

Out[108]:

```
2026854875376
```

In [100]:

```
y = [1,2,3]
def change(var):
    var[0] += 1
    print(var)
```

In [101]:

```
id(y)
```

Out[101]:

```
2026935473344
```

In [102]:

```
change(y)
```

```
[2, 2, 3]
```

In [103]:

```
y
```

Out[103]:

```
[2, 2, 3]
```

In [104]:

```
id(y)
```

Out[104]:

```
2026935473344
```

Получается, что произошло неявное изменение глобальной переменной. Как лучше: добавить return var, и переопределить переменную

3. Словари

Позволяют хранить данные в формате ключ-значение. Изменяемые; неупорядоченные

In [111]:

```
empty_dict = {}
empty_dict = dict()
collections_map = {
'mutable': ['list', 'set', 'dict'],
'immutable': ['tuple', 'frozenset']
}
```

In [112]:

```
print(collections_map['immutable'])
```

```
['tuple', 'frozenset']
```

In [113]:

```
print(collections_map['irresistible'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_12640\100508803.py in <cell line: 1>()
----> 1 print(collections_map['irresistible'])

KeyError: 'irresistible'
```

часто бывает полезно попытаться достать значение по ключу из словаря, а в случае отсутствия ключа вернуть какое-то стандартное значение. Для этого есть встроенный метод get

In [114]:

```
print(collections_map.get('irresistible', 'not found'))
```

```
not found
```

Проверка на вхождения ключа в словарь так же осуществляется за константное время и выполняется с помощью ключевого слова in:

In [115]:

```
'mutable' in collections_map
```

Out[115]:

```
True
```

Словари, как и все коллекции, поддерживают протокол итерации. С помощью цикла for можно итерироваться по ключам словаря:

In [121]:

```
print(collections_map)
for key in collections_map:
    print(key)
```

```
{'mutable': ['list', 'set', 'dict'], 'immutable': ['tuple', 'frozenset']}
mutable
immutable
```

Если нам нужно итерироваться не по ключам, а по ключам и значениям сразу, можно использовать метод словаря items, который возвращает ключи и значения.

In [122]:

```
for key, value in collections_map.items():
    print('{} — {}'.format(key, value))
```

```
mutable — ['list', 'set', 'dict']
immutable — ['tuple', 'frozenset']
```

Если нужно итерироваться по значениям, используйте логично метод values, который возвращает именно значения. Также существует симметричный метод keys, который воз вращает итератор ключей.

In [124]:

```
for value in collections_map.values():
    print(value)
```

```
['list', 'set', 'dict']
['tuple', 'frozenset']
```

Упражнение

In [15]:

```
states = {'Россия': 'ru', 'Германия': 'de', 'Узбекистан': 'uz', 'Зимбабве': 'zw','Турция':
states
```

Out[15]:

```
{'Россия': 'ru',
 'Германия': 'de',
 'Узбекистан': 'uz',
 'Зимбабве': 'zw',
 'Турция': 'tr'}
```

In [16]:

```
cities = {'uz':'Ташкент', 'de':'Мюнхен', 'zw':'Harare','tr':'Мармарис'}
cities
```

Out[16]:

```
{'uz': 'Ташкент', 'de': 'Мюнхен', 'zw': 'Harare', 'tr': 'Мармарис'}
```

In [17]:

```
cities['ru'] = 'Таганрог'
cities
```

Out[17]:

```
{'uz': 'Ташкент',
 'de': 'Мюнхен',
 'zw': 'Harare',
 'tr': 'Мармарис',
 'ru': 'Таганрог'}
```

А теперь посмотрим, сколько есть различных способов обращаться к данным в словаре Вывод некоторых городов:

In [18]:

```
print('В стране zw есть город ', cities['zw'])
print('В стране ru есть город ', cities['ru'])
```

```
В стране zw есть город  Harare
В стране ru есть город  Таганрог
```

Вывод некоторых стран:

In [19]:

```
print('Аббревиатура Турции ', states['Турция'])
print('Аббревиатура Германии ', states['Германия'])
```

```
Аббревиатура Турции  tr
Аббревиатура Германии  de
```

Совместное использование двух словарей:

In [20]:

```
print('В России есть город ', cities[states['Россия']])
```

```
В России есть город  Таганрог
```

Вывести аббревиатуры все стран:

In [21]:

```
for state, abbrev in list(states.items()):
    print(f'{state} имеет аббревиатуру {abbrev}')
```

```
Россия имеет аббревиатуру ru
Германия имеет аббревиатуру de
Узбекистан имеет аббревиатуру uz
Зимбабве имеет аббревиатуру zw
Турция имеет аббревиатуру tr
```

Вывод всех городов в странах

In [23]:

```
for abbrev, city in list(cities.items()):
    print(f'В стране {abbrev} есть городу {city}')
```

```
В стране uz есть городу Ташкент
И есть город Ташкент
В стране de есть городу Мюнхен
И есть город Мюнхен
В стране zw есть городу Harare
И есть город Harare
В стране tr есть городу Мармарис
И есть город Мармарис
В стране ru есть городу Таганрог
И есть город Таганрог
```

In [24]:

```
for state, abbrev in list(states.items()):
    print(f'В стране {state} используется аббревиатура {abbrev}')
    print(f'И есть город {cities[abbrev]}')
```

```
В стране Россия используется аббревиатура ru
И есть город Таганрог
В стране Германия используется аббревиатура de
И есть город Мюнхен
В стране Узбекистан используется аббревиатура uz
И есть город Ташкент
В стране Зимбабве используется аббревиатура zw
И есть город Harare
В стране Турция используется аббревиатура tr
И есть город Мармарис
```

Безопасное получение аббревиатуры страны, даже если ее нет в словаре

In [27]:

```
var = 'США'
state = states.get(var)
print(state)
```

```
None
```

In [28]:

```
if not state:
    print(f'Простите, но {var} не существует...')
```

```
Простите, но США не существует...
```

Получение города со значением по умолчанию

In [31]:

```
city = cities.get('US','не существует')
print(f'В стране "US" есть город {city}')
```

```
В стране "US" есть город не существует
```

Словари - неупорядоченный тип данных. Однако есть специальный тип OrderedDict (содержится в модуле collections), который гарантирует вам, что ключи хранятся именно в том порядке, в каком вы их добавили в словарь.

In [125]:

```
from collections import OrderedDict

ordered = OrderedDict()

for number in range(10):
    ordered[number] = str(number)
for key in ordered:
    print(key)
```

```
0
1
2
3
4
5
6
7
8
9
```

4. Множества

Множество в питоне — это неупорядоченный набор уникальных объектов. Множества изменяемы и чаще всего используются для удаления дубликатов и всевозможных проверок на вхождение. Чтобы объявить пустое множество, можно воспользоваться литералом set или использовать фигурные скобки, чтобы объявить множество и одновременно добавить туда какие-то элементы.

In [126]:

```
empty_set = set()
number_set = {1, 2, 3, 3, 4, 5}
print(number_set)
```

```
{1, 2, 3, 4, 5}
```

Чтобы добавить элемент в множество, используется метод add. Также множества в Python поддерживают стандартные операции над множествами --- такие как объедине ние, разность, пересечение и симметрическая разность.

In [127]:

```
odd_set = set()
even_set = set()
for number in range(10):
    if number % 2:
        odd_set.add(number)
    else:
        even_set.add(number)
print(odd_set)
print(even_set)
```

```
{1, 3, 5, 7, 9}
{0, 2, 4, 6, 8}
```

Теперь найдём объединение и пересечение этих множеств:

In [130]:

```
union_set = odd_set | even_set
print(union_set)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

In [132]:

```
general_set = odd_set & even_set
general_set
```

Out[132]:

```
set()
```

In [133]:

```
difference_set = odd_set - even_set
difference_set
```

Out[133]:

```
{1, 3, 5, 7, 9}
```

In [134]:

```
even_set.remove(2)
print(even_set)
```

```
{0, 4, 6, 8}
```

In [135]:

```
dir(set)
```

Out[135]:

```
['__and__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__iand__',
 '__init__',
 '__init_subclass__',
 '__ior__',
 '__isub__',
 '__iter__',
 '__ixor__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__or__',
 '__rand__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__ror__',
 '__rsub__',
 '__rxor__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__xor__',
 'add',
 'clear',
 'copy',
 'difference',
 'difference_update',
 'discard',
 'intersection',
 'intersection_update',
 'isdisjoint',
 'issubset',
 'issuperset',
 'pop',
 'remove',
 'symmetric_difference',
 'symmetric_difference_update',
```

```
 'union',
 'update']
```

In [138]:

```
even_set.add(2)
even_set
```

Out[138]:

```
{0, 2, 4, 6, 8}
```

Также в питоне существует неизменяемый аналог типа set --- тип frozenset.

In [139]:

```
dir(frozenset)
```

Out[139]:

```
['__and__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__or__',
 '__rand__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__ror__',
 '__rsub__',
 '__rxor__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__xor__',
 'copy',
 'difference',
 'intersection',
 'isdisjoint',
 'issubset',
 'issuperset',
 'symmetric_difference',
 'union']
```

Самостоятельно

Дана строка с некоторым текстом.

Требуется создать словарь, который в качестве ключей будет принимать буквы, а в качестве значений – количество этих букв в тексте. Для построения словаря создайте функцию count_it(sequence), принимающую строку с текстом. Функция должна возвратить словарь из 3-х самых часто встречаемых букв.

In [35]:

```
# Решение
def count_it(sequence):
    # При помощи генератора создаем словарь, где ключом выступает уникальный элемент строки
    num_frequency = {item: sequence.count(item) for item in sequence}

    # Сортируем словарь по значениям в порядке возрастания. Для этого методом items() форми
    sorted_num_frequency = sorted(num_frequency.items(), key=lambda element: element[1])

    # Возвращаем последние 3 элемента списка, т. е. кортежи с самыми большими значениями вт
    return dict(sorted_num_frequency[-3:])

# Тесты
print(count_it('фывралвралофравврапыорпаоывпаовпа'))
print(count_it('ывфравдыарыолрафдлваылвпаврпарпаорпаорп'))
print(count_it('dfgjgjhgdjfgjfgfsd'))
```

```
{'р': 5, 'в': 6, 'а': 7}
{'п': 5, 'р': 7, 'а': 8}
{'f': 4, 'j': 4, 'g': 5}
```

Выводы

Справка по рассмотренным темам: https://docs.python.org/3/tutorial/datastructures.html (https://docs.python.org/3/tutorial/datastructures.html)