**ЛАБОРАТОРНАЯ РАБОТА №6**

***Тема: Множества. Словари. Генераторы***

***Цель: Рассмотрение способов работы с множествами, словарями и генераторами***

***ОГЛАВЛЕНИЕ***

[***1 Множества***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1)[***1.1 Создание множеств***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.1)[***1.2 Добавление элементов к множеству***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.2)[***1.3 Удаление элементов из множества***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.3)[***1.4 Сравнение и копирование множеств***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.4)[***1.5 Операции над множествами***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.5)[***2 Словари***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2)[***2.1 Создание словарей***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1)[***2.2 Методы словарей***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.2)[***3 Генераторы***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3)[***3.1 Использование генератора для формирования списка***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3.1)[***3.2 Использование генератора для формирования множества***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3.2)[***3.3 Использование генератора для формирования словаря***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3.3)[***4 Копирование объектов***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#4)[***Индивидуальные задания***](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#ind)

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***1 Множества***

Объекты множества в языке Python представляют собой неупорядоченные контейнеры (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.1) лаб. раб. №3) уникальных хэшируемых объектов.   
Объект является хэшируемым, если он имеет значение хэш-функции, которое не меняется в течение времени существования объекта, т.е. поддерживает метод *\_\_hash\_\_()* и может быть сравним с другими объектами с помощью метода *\_\_eq\_\_()* (см. [подраздел 1.3](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.3) лаб. раб. №3). Все хэшируемые объекты являются неизменяемыми объектами и принадлежат ABC-классу Hashable (см. [раздел 1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1) лаб. раб. №3).  
Хэшируемость объектов позволяет использовать их в качестве элементов множества и ключей словарей (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1)), поскольку эти структуры данных используют значения хэш-функции.  
В языке Python все встроенные неизменяемые объекты (такие, как числа, логические значения, строки и кортежи) являются хэшируемыми:  
  
>>> from collections import Hashable  
>>> isinstance (1.25e-1, Hashable)  
True  
>>> isinstance ( False , Hashable)  
True  
>>> isinstance ( 'Web' , Hashable)  
True  
>>> isinstance ((1,2,3),collections.Hashable)  
True  
  
в то время как изменяемые контейнеры, такие как изменяемые множества (см. [подраздел 1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.1)), словари (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1)) и списки (см. [раздел 2](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2) лаб. раб. №3) – нет:  
  
>>> isinstance ([ 'a' , 'b' , 'c' ], Hashable  
False  
  
Объекты, которые являются экземплярами созданных пользователем классов, являются хэшироваными по умолчанию, при сравнении между собой они не равны и значения их хэшей являются значениям их *id()*.  
Объекты множества могут быть использованы при проверке на наличие элементов в последовательности, при удалении дубликатов из последовательности и при выполнении математических операций над множествами, таких как объединение, пересечение и вычитание.   
Имеется два встроенных типа (класса) множеств: *set* и *frozenset* (см. [подраздел 2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1)).   
Тип *set* является изменяемым множеством или просто множеством. Как изменяемый объект, он не может быть использован ни в качестве ключа в словарях, ни в качестве элемента другого множества.  
Тип *frozenset* является неизменяемым множеством. Поэтому может быть ключом в словаре и элементом другого множества.

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***1.1 Создание множеств***

Создавать экземпляры изменяемого множества (тип *set*) можно с помощью литерала, поместив в фигурные скобки элементы множества неизменяемых типов, разделенные запятыми:   
  
>>> a\_set={1,2, 'a' , 'b' }  
>>> a\_set  
{'b', 1, 2, 'a'},  
  
или с помощью генератора элементов множества (см. [подраздел 3.2](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#3.2)).  
Проверим изменяемость (нехэшируемость) множеств:  
  
>>> isinstance (a\_set, Hashable)  
False.  
  
Также для создания множества можно использовать конструктор – встроенную функцию *set([iterable])* (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#set) лаб. раб. №1), аргументом которой является итерабельный объект. Поэтому в объекты множества можно преобразовать такие последовательности, как строки, списки, кортежи и диапазоны. Если строка, список, кортеж или диапазон содержат одинаковые элементы, после преобразования из них останется только один:   
  
>>> set ( 'ABBA' )  
{'B','A'}.  
  
Отметим, что, поскольку множество является неупорядоченным контейнером, то и элементы полученного множества могут указываться интерпретатором в произвольном порядке.  
Раз элементы множества не могут быть изменяемыми типами, то и итерабельный тип, указанный в качестве аргумента функции *set()* не может содержать изменяемые (нехэшируемые) элементы:  
  
>>> set ([1,2,[3,4]])  
TypeError: unhashable type: 'list'  
  
Для создания объектов множества неизменяемого типа *frozenset* необходимо использовать конструктор – встроенную функцию *frozenset()* (см. [подраздел 8.1](https://py-khpi.github.io/!PYTHON/LAB1/lab.htm#frozenset) лаб. раб. №1, аргументом которой является итерабельный объект:  
  
>>> frozenset (( 'a' , 'c' , 'd' , 'c' ))  
frozenset({'d', 'a', 'c'}) .  
  
Экземпляры типов *set* и *frozenset*, являясь контейнерами:  
  
>>> import collections  
>>> isinstance ({1,2},collections.Container)  
True  
>>> isinstance (frozenset({1,2}), collections.Container)  
True  
  
и не являясь последовательностями:  
  
>>> isinstance ({1,2},collections.Sequence)  
False  
>>> isinstance (frozenset({1,2}), collections.Sequence)  
False  
  
поддерживают лишь часть операций, свойственных последовательностям (те из них, которые также поддерживают и контейнеры):

* *x in s – возвращает True, если x содержится в s, False – в противном случае;*
* *x not in s – возвращает True, если x не содержится в s, False – в противном случае;*
* *len(s) – возвращает число элементов s;*
* *max(s) – возвращает максимальный элемент s;*
* *min(s) – возвращает миниимальный элемент s.*

Можно добавить, что объекты изменяемого множества *set* принадлежат ABC-классу *MutableSet*:   
  
>>> s={1,2,3}  
>>> isinstance (s,collections.MutableSet)  
True,

а объекты неизменяемого класса *frozenset* – нет:   
  
>>> fs=frozenset(s)  
>>> isinstance (fs,collections.MutableSet)  
False

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***1.2 Добавление элементов к множеству***

Имеется два способа добавления элементов в существующее множество:

* *метод add() принимает один аргумент, который может быть любого неизменяемого типа и добавляет данное значение к множеству:  
    
  >>> a\_set.add((4,5))  
  >>> a\_set  
  {'a', 1, 2, 'b', (4, 5)}*
* *метод update() может принимать несколько аргументов разных типов и добавлять их к множеству, если аргументами являются последовательности (списки, кортежи или множества), то их элементы должны быть хэшируемыми (неизменяемых типов) и добавляться как новые элементы множества (при этом одинаковые элементы удаляются):  
    
  >>> a\_set.update([2,3],{ 'b' , 'c' })  
  >>> a\_set  
  {1, 2, 3, 'a', 'c', 'b', (4, 5)}*

Отметим, что объекты типа *frozenset*, будучи неизменяемыми множествами, не имеют методов добавления элементов к множеству.

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***1.3 Удаление элементов из множества***

Имеется несколько способов удаления элементов из существующего множества:

* *метод discard() принимает в качестве аргумента одиночное значение и удаляет это значение из множества, если этого значения в множестве нет, то ничего не произойдет:  
    
  >>> a\_set.discard(2)  
  >>> a\_set  
  {1, 3, 4, 'b', 'c', 'a'};*
* *метод remove() также принимает в качестве аргумента одиночное значение, и также удаляет его из множества, но в отличие от метода discard(), если в множестве нет элемента, то будет выработано исключение KeyError:  
    
  >>> a\_set.remove( 'a' )  
  {1, 3, 4, 'c', 'b'},   
  >>> a\_set.remove( 'a' )  
  >>> KeyError: 'a';*
* *метод pop() удаляет из множества первый элемент и возвращает его значение:  
    
  >>> a\_set.pop()  
  1  
  >>> a\_set  
  {3, 4, 'c', 'b'},  
  если множество пусто – обозначается set(), то вырабатывается исключение:  
    
  KeyError: 'pop from an empty set'  
    
  С помощью метода pop() можно просмотреть все элементы множества:  
    
  >>> while a\_set:  
  print (a\_set.pop())   
  3  
  4  
  b  
  c;*
* *метод clear() удаляет все элементы множества, создавая пустое множество:  
    
  >>> a\_set.clear()  
  >>> a\_set  
  set().*

В логическом контексте пустое множество *set()* соответствует False, любое другое – True.   
Отметим, что объекты типа *frozenset*, будучи неизменяемыми множествами, не имеют методов удаления элементов из множества.

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***1.4 Сравнение и копирование множеств***

Python имеет несколько методов сравнения множеств:

* *метод isdisjoint() принимает в качестве аргумента другое множество и возвращает значение True, если множенства не имеют общих элементов:  
    
  >>> a\_set={1,2,5,6}  
  >>> a\_set.isdisjoint({3,7})  
  True,  
  >>> a\_set.isdisjoint({2,7})  
  False;*
* *метод issubset() принимает в качестве аргумента другое множество и возвращает значение True, если каждый элемент данного множества содержится в другом множестве:  
    
  >>> a\_set.issubset({1,2,3,3,5,6})  
  True;*
* *метод issuperset() принимает в качестве аргумента другое множество и возвращает значение True, если каждый элемент другого множества содержится в данном множестве:  
    
  >>> a\_set.issuperset({1,6})  
  True.*

Метод *copy()* используется для "поверхностного" копирования списков (см. [раздел 4)](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#4)):  
  
>>> b\_set=a\_set.copy()  
>>> b\_set  
{1, 2, 5, 6}.

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***1.5 Операции над множествами***

Python имеет несколько методов для выполнения основных операций над множествами:

* *метод union() (объединение) возвращает новое множество, содержащее все элементы каждого из множеств:  
    
  >>> a\_set={1,2,5,6,9}  
  >>> b\_set={2,4,6,8}  
  >>> a\_set.union(b\_set)  
  {1, 2, 4, 5, 6, 8, 9},  
    
  метод симметричен, т.е. a\_set.union(b\_set)=b\_set.union(a\_set):  
    
  >>> b\_set.union(a\_set)  
  {1, 2, 4, 5, 6, 8, 9};*
* *метод intersection() (пересечение) возвращает новое множество, содержащее все элементы, которые есть и в первом множестве, и во втором:  
    
  >>> a\_set.intersection(b\_set)  
  {2, 6},  
    
  метод симметричен;*
* *метод difference() (разность) возвращает новое множество, содержащее все элементы, которые есть в множестве a\_set, но которых нет в множестве b\_set:  
    
  >>> a\_set.difference(b\_set)  
  {9, 5, 1},  
    
  этот метод не симметричен, поскольку a\_set.difference(b\_set) не равно b\_set.difference(a\_set):  
    
  >>> b\_set.difference(a\_set)  
  {8, 4};*
* *метод symmetric\_difference() (симметрическая разность) возвращает новое множество, которое содержит только уникальные элементы обоих множеств:  
    
  >>> a\_set.symmetric\_difference(b\_set)  
  {1, 4, 5, 8, 9},  
    
  метод симметричен.*

 [*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***2 Словари***

Словари в языке Python имеют тип (класс) *dict* и представляют собой множество пар вида *ключ:значение*. Словари в языке Python похожи на ассоциативные массивы языка Jaxascript.

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***2.1 Создание словарей***

Для создания словаря в виде литерала необходимо поместить его элементы, заданные в виде пар *ключ:значение* и разделеные запятыми, в фигурные скобки:  
  
>>> a\_dict={ 'alpha' :1, 'beta' :2, 'gamma' :3}  
>>> a\_dict  
{'alpha': 1, 'beta': 2, 'gamma': 3}  
  
В качестве ключа необходимо использовать только неизменяемые (хэшируемые) типы: числа (целые и с плавающей точкой), строки, логические значения, кортежи, диапазоны и неизменяемые множества. В качестве значения могут быть заданы любые из рассмотренных типов, включая изменяемые множества и словари.  
Словари также можно задавать с помощью конструктора – встроенной функции *dict()* (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#dict) лаб. раб. №1). Если при вызове функция не имеет аргументов, то создается пустой список:  
  
>>> dict ()  
{},  
  
если имеет, то для создания словаря могут быть использованы следующие варианты задания аргументов:

* *именованные аргументы, которые указываются в виде пар "имя=значение" (см.* [*подраздел 1.1.2*](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.1.2) *лаб. раб. №6):  
    
  >>> b\_dict= dict (alpha=1, beta=2, gamma=3);*
* *аргумент, имеющий тип словаря:  
    
  >>> c\_dict= dict ({ 'gamma' :3, 'alpha' :1, 'beta' :2}) ;*
* *аргумент итерабельного типа, например, список, элементами которого являются кортежи, содержащие ровно два элемента: первый из которых задает ключ словаря, а второй – значение:  
    
  >>> d\_dict= dict ([( 'alpha' ,1), ( 'beta' ,2), ( 'gamma' ,3)])*

Выше четырьмя разными способами были созданы словари *a\_dict()*, *b\_dict()*, *c\_dict()* и *d\_dict()*, имеющие одно и то же значение. Проверим это:  
  
>>> a\_dict==b\_dict==c\_dict==d\_dict  
True.  
  
Если ключи и значения словаря заданы последовательностями, то для создания словаря также можно использовать встроенную функцию *zip(\*iterable)* (см. [подраздел 8.1](https://py-khpi.github.io/!nfo/doc/lab/LAB1/lab.htm#zip) лаб. раб. №1), которая возвращает итератор, элементами которого являются кортежи, где i-й кортеж содержит i-е элементы каждого аргумента последовательностей или итерабельных объектов. Если последовательности, явдяющиеся аргументами функции, имеют разную длину, то формируемые кортежи имеют число элементов, равное длине минимальной последовательности. Если указан один элемент – функция возвращает итератор, состоящий из одноэлементных кортежей. Если аргументы не указаны – возвращается пустой итератор.  
Пусть, например, ключи словаря представлены списком *key\_list*, а значения – кортежом *value\_tuple*:  
  
>>> key\_list=[ 'a' , 'b' , 'c' ]  
>>> value\_tuple=(1.32, 'value' , [1,2,3], 25)  
  
Теперь с помощью функции *zip()*, аргументами которой являются последовательности *key\_list* и *value\_tuple*, создаем итератор с именем *zipped*:   
  
>>> zipped= zip (key\_list, value\_tuple),

который имеет тип *zip*:  
  
>>> type (zipped)  
<class 'zip'>  
  
Остается только преобразовать его с помощью конструктора в словарь:  
  
>>> e\_dict= dict (zipped)  
>>> e\_dict   
{'a': 1.32, 'b': 'value', 'c': [1, 2, 3]}  
  
Следует сказать, что словари в настоящее время являются единственными представителями ABC-класса *Mapping*, объекты которого ставят в соответсвие неизменяемые значения произвольным объектам. Являются изменяемыми контейнерами типа *Mapping*:  
  
>>> import collections  
>>> isinstance ({1: 'a' ,2: 'b' ,3: 'c' },collections.Container)  
True  
>>> isinstance ({1: 'a' ,2: 'b' ,3: 'c' },collections.Mapping)  
True  
>>> isinstance ({1: 'a' ,2: 'b' ,3: 'c' },collections.MutableMapping)  
True  
  
Для получения значения словаря по ключу нобходимо воспользоваться выражением:  
  
>>> a\_dict[ 'beta' ]  
2,  
  
если указанный ключ отсутствует – возбуждается исключение KeyError:  
  
>>> a\_dict[ 'delta' ]  
KeyError: 'delta'.  
  
Чтобы получить все значения словаря следует использовать оператор *for in*:  
  
>>> for key in a\_dict:  
print (key, '=' ,a\_dict[key])  
alpha = 1  
beta = 2  
gamma = 3  
  
Можно изменять значение ключа:  
  
>>> a\_dict[ 'alpha' ]=11  
>>> a\_dict  
{'alpha': 11, 'beta': 2, 'gamma': 3}

и добавлять новые пары в словарь:  
  
>>> a\_dict[ 'epsilon' ]=5  
>>> a\_dict  
{'alpha': 22, 'beta': 2, 'gamma': 3, 'epsilon': 5}  
  
Для удаления ключа необходимо указать:  
  
>>> del a\_dict[ 'alpha' ]  
>>> ]  
{'beta': 2, 'gamma': 3, 'epsilon': 5}.  
  
Словари так же, как последовательности и множества поддерживают функцию определения числа элементов *len()* и оператор *in*, позволяющий определить принадлежность элемента контейнеру.  
Пустой словарь – {} в логическом контексте равен False, все остальные словари – True.

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***2.2 Методы словарей***

Словари поддерживают следующие методы:

* *clear() – очищает словарь, возвращая значение {}:  
    
  >>> b\_dict={1:101, 2:102, 3:103}  
  >>> b\_dict.clear()  
  >>> a\_dict  
  {}.  
    
  Метод clear() поддерживается всеми изменяемыми классами данных;*
* *copy() – возвращает ссылку на вновь созданный объект класса dict с теми же значениями (т.е. выполняет "поверхностное" копирование, см.* [*раздел 4*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#4)*):  
    
  >>> b\_dict=a\_dict.copy()  
  >>> b\_dict  
  {'beta': 2, 'gamma': 3, 'epsilon': 5}.  
    
  Метод copy() поддерживается всеми изменяемыми классами данных;*
* *get(key[, default]) – если словарь имеет ключ key, возращает его значение, если нет – возвращает значение, заданное вторым аргументом (значение по умолчанию):  
    
  >>> a\_dict.get( 'gamma' , 'new\_key' )  
  'gamma'  
  >>> a\_dict.get( 'delta' , 'new\_key' )  
  'new\_key';*
* *items() – возвращает элементы словаря в виде пар (key, value):  
    
  >>> items=a\_dict.items()  
  >>> items  
  dict\_items([('epsilon', 5), ('beta', 2), ('gamma', 3)])*
* *keys() – возвращает все ключи словаря:  
    
  >>> a\_dict.keys()  
  dict\_keys(['beta', 'gamma', 'epsilon'])*
* *pop(key[, default) – если указанный ключ содержится в словаре, удаляет его из словаря и возвращает его значение, иначе возвращает значение по умолчанию. Если значение по умолчанию не указано и ключ не содержится в словаре, то возникает исключение KeyError:  
    
  >>> a\_dict.pop( 'gamma' )  
  3  
  >>> a\_dict  
  {'beta': 2, 'delta': 4, 'epsilon': 5} ;*
* *popitem() – удаляет и возвращает произвольную пару вида (key, value) из словаря;*
* *setdefault(key[, default) – похож на метод get(), но если ключа нет в словаре – не только возвращает значение по умолчанию, но и устанавливает его:  
    
  >>> a\_dict.setdefault( 'delta' , 4 )  
  4  
  >>> a\_dict  
  {'beta': 2, 'delta': 4, 'epsilon': 5}*
* *update([dict2]) – возвращает слияние элементов данного словаря и словаря, заданного аргументом (аргумент может быть задан: в виде именованных аргументов, в виде итерабельного типа или в виде словаря (см. описание конструктора dict() в* [*подразделе 2.1*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1)*):  
    
  >>> b\_dict={ 'figure' : 'circle' , 'color' : 'green' }  
  >>> a\_dict.update(b\_dict)  
  >>> a\_dict  
  {'beta': 2, 'figure': 'circle', 'delta': 4, 'color': 'green', 'epsilon': 5}*
* *values() – возвращает все значения словаря:  
    
  >>> a\_dict.values  
  dict\_values([2, 'circle', 4, 'green', 5]).*

 [*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***3 Генераторы***

В литералах при создании списков, множеств и словарей помимо указания перечня их элементов (см. подразделы [2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.1) лаб. раб. №3, [1.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.1) и [2.1](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.1)) можно также указать генераторы элементов этих контейнеров.

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***3.1 Использование генератора для формирования списка***

Например, создание списка кубов целых чилел от 1 до 10 можно осуществить следующим образом:   
  
>>> a\_list=[x\*\*3 for x in range (1, 11)]  
>>> a\_list  
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000].  
  
Выражение *x\*\*3 for x in range(1,11)* является генераторным выражением. Оно имеет тип генератора:  
  
>>> type (x\*\*3 for x in range (1, 11))  
<class 'generator'>  
  
и состоит из выражения *x\*\*3*, за которым следует оператор *for* для итерабельного типа. В таком виде генераторное выражение может использоваться только в качестве единичного аргумента в функциях. В остальных случаях необходимо указывать его в скобках, например при анализе, является ли тип *generator* итерабельным:  
  
>>> import collections  
>>> isinstance ((x\*\*3 for x in range (1, 11)), collections.Iterable)  
True.  
  
В приведенном примере сформированные кубы целых чисел сохранены в памяти в виде списка. Для длинных последовательностей это не всегда удобно. Рассмотрим вариант генерации кубов чисел без сохранения их в памяти:  
  
>>> gen=(x\*\*3 for x in range (1, 11))  
>>> for x in gen:  
print x, end= ' ' )  
1, 8, 27, 64, 125, 216, 343, 512, 729, 1000.  
  
Поскольку значения генератора *gen* не сохранены в памяти повторное его использование ничего не дает:  
  
>>> for x in gen:  
print x, end= ' ' )

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***3.2 Использование генератора для формирования множества***

Создадим с помощью генератора множество букв, из которых состоит предложение "Карл украл у Клары кораллы", представив их в верхнем регистре:  
  
>>> a\_set={x for x in 'Карл украл у Клары кораллы' .upper()}  
>>> a\_set  
{'О', 'Л', 'К', 'А', ' ', 'Ы', 'Р', 'У'} .

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***3.3 Использование генератора для формирования словаря***

В качестве последнего примера использования генераторов рассмотрим создание словаря, ключами которого являются числа, заданные в списке, а значениями – квадратные корни этих чисел:  
  
>>> {x:x\*\*(1/2) for x in [2, 5, 25] }  
{25: 5.0, 2: 1.4142135623730951, 5: 2.23606797749979}.  
  
Формировать элементы последовательностей также можно с использованием встроенной функции *map()* (см. [подраздел 1.5.2](https://py-khpi.github.io/!nfo/doc/lab/LAB6/lab.htm#1.5.2) лаб. раб. №6).

[*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***4 Копирование объектов***

В [подразделе 1.4](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.4) при описании метода *copy()* был использован термин поверхностная копия. Поскольку существует еще и глубокая копия, дадим разъяснение этим терминам.  
Операторы присваивания в языке Python не копируют объекты, они только сооздают ссылки на объект. Для контейнеров, которые являются изменяемыми или содержат изменяемые элементы, возникает иногда необходимость их копирования – чтобы можно было изменить копию, не изменяя исходные значения.  
Модуль *copy* содержит два метода:

* *один для создания поверхностной копии – copy.copy(x);*
* *другой для создания глубокой копии – copy.deepcopy(x).*

Различие между между поверхностным (англ. shallow) и глубоким (англ. deep) копированием имеет смысл только для составных объектов (объектов, которые содержат другие объекты, например, списки, словари или экземпляры классов):

* *поверхностное копирование создает новый составной объект, в который включаются ссылки на объекты, содержащиеся в исходном объекте;*
* *глубокое копирование создает новый составной объект, в который рекурсивно включаются копии объектов, составляющих исходный объект.*

В качестве примера создадим и сравним поверзностную и глубокую копию словаря:  
  
>>> import copy  
>>> a\_dict={1:5,2:{3:25}}  
>>> a\_dict  
{1: 5, 2: {3: 25}}   
>>> b\_dict=copy.copy(a\_dict)  
>>> b\_dict  
{1: 5, 2: {3: 25}}   
>>> b\_dict[2][3]=77  
>>> a\_dict  
{1: 5, 2: {3: 77}}.  
  
Видно, что при использовании поверхностного копирования словаря (функции copy()) изменения в полученной копии привели к изменению оригинала. При использовании глубокого копирования (функции deepcopy()) этого не произойдет:  
  
>>> a\_dict={1:5,2:{3:25}}  
>>> c\_dict=copy.deepcopy(a\_dict)  
>>> c\_dict[2][3]=77  
>>> a\_dict  
{1: 5, 2: {3: 25}}.  
  
Для создания "поверхностной" копии помимо метода *copy()* модуля *copy()* можно также использовать:

* *методы copy() изменяемых объектов: списков, множеств и словарей (см. подразделы* [*2.3*](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#2.3) *лаб. раб. №3,* [*1.4*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#1.4) *и* [*2.2*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#2.2)*);*
* *срезы последовательностей (см.* [*подраздел 1.4.2*](https://py-khpi.github.io/!nfo/doc/lab/LAB3/lab.htm#1.4.2) *лаб. раб. №3);*
* *конструкторы, аргументы которых содержат тот же тип, что и создаваемый объект (см. описания конструкторов всех рассмотренных типов).*

 [*Оглавление*](https://py-khpi.github.io/!nfo/doc/lab/LAB5/lab.htm#beg)

***Индивидуальные задания***

Разработать программу на языке Python, которая выполняет следующее:

* *1 Создает множество a\_set, содержащее не менее 7 элементов любых разрешенных типов, с помощью (см. колонку "Множество"/"Создание" табл. №1):* 
  + *1 – литерала;*
  + *2 – конструктора;*
  + *3 – генератора.*
* *2 Создает итерабельный объект it\_ob, содержащий не менее трех элементов, имеющихся в объекте a\_set, и проверить, все ли элементы it\_ob хэшируемы. Если нет – заменить нехэшируемые элементы хэшируемыми.*
* *3 Преобразует объект it\_ob в множество b\_set и выполняет над множествами a\_set и b\_set операции (см. колонку "Множество"/"Операции" табл. №1):* 
  + *1 – union();*
  + *2 – intersection();*
  + *3 – difference();*
  + *4 – symmetric\_difference().*
* *4 Создает словарь a\_dict с помощью (см. колонку "Словарь"/"Создание" табл. №1):* 
  + *1 – литерала;*
  + *2 – конструктора с именованными аргументами;*
  + *3 – конструктора с аргументом, имеющем тип словаря;*
  + *4 – конструктора с аргументом итерабельного типа;*
  + *5 – генератора.*
* *5 Выполняет следующие методы словаря a\_dict (см. колонку "Словарь"/"Методы" табл. №1):* 
  + *1 – clear();*
  + *2 – get(key[, default]);*
  + *3 – items();*
  + *4 – keys();*
  + *5 – pop(key[, default);*
  + *6 – popitem();*
  + *7 – setdefault(key[, default);*
  + *8 – update([dict2]);*
  + *9 – values().*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 1 – Перечень индивидуальных заданий | | | | |
| **Номер п/п** | **Множество** | | **Словарь** | |
| **Создание** | **Операции** | **Создание** | **Методы** |
| 1 | 3 | 4 | 1 | 2,5,8 |
| 2 | 2 | 3 | 2 | 2,6,9 |
| 3 | 1 | 2 | 3 | 3,5,1 |
| 4 | 3 | 1 | 4 | 3,6,8 |
| 5 | 2 | 4 | 5 | 4,5,9 |
| 6 | 1 | 3 | 1 | 4,6,1 |
| 7 | 3 | 2 | 2 | 7,9,5 |
| 8 | 2 | 1 | 3 | 2,5,8 |
| 9 | 1 | 4 | 4 | 2,6,9 |
| 10 | 3 | 3 | 5 | 3,5,1 |
| 11 | 2 | 2 | 1 | 3,6,8 |
| 12 | 1 | 1 | 2 | 4,5,9 |
| 13 | 3 | 4 | 3 | 4,6,1 |
| 14 | 2 | 3 | 4 | 7,9,5 |
| 15 | 1 | 2 | 5 | 2,5,8 |
| 16 | 3 | 1 | 1 | 2,6,9 |
| 17 | 2 | 4 | 2 | 3,5,1 |
| 18 | 1 | 3 | 3 | 3,6,8 |
| 19 | 3 | 2 | 4 | 4,5,9 |
| 20 | 2 | 1 | 5 | 4,6,1 |