

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

Урок 6

Кортежи, множества, словари

Содержание

1. Кортежи	4
Коллекции неизменяемых объектов.....	4
Применение и особенности кортежа.....	5
2. Множества.....	20
Математическое понятие множеств.....	20
Операции над множествами	21
Тип данных set()	24
Операции над множествами.....	26
frozenset()	32
Применение множеств	34
3. Словари.....	37
Ассоциативные массивы.....	37
Хеш-таблицы	38
Создание словаря.....	41

Методы словаря	46
4. Практические примеры использования	61
Пример 1. Поиск в словаре.....	61
Пример 2. Сортировка элементов словаря.....	64
Пример 3: сортировка списка словарей.....	68
Пример 4. Фильтры в словаре	69

1. Кортежи

Коллекции неизменяемых объектов

Кортеж (*tuple*) — это еще один тип Python-коллекций, который представляет собой «неизменяемый список», т. е. является упорядоченным и неизменяемым набором произвольных данных.

Когда есть смысл использовать кортежи и почему нам не для этих задач не подходят списки? Кортежи используются для хранения данных, перечень значений которых не должен изменяться в программе, т. е. с целью обезопасить данные от изменения (явного или случайного) в коде программы.

Например, наш код должен использовать данные о типах пользователя приложения. Если эта информация будет храниться в списке, например, так

```
userTypes = ["admin", "student", "teacher"],
```

то строка кода, подобная этой:

```
userTypes[0]="user"
```

может удалить не только информацию о типе пользователя, но и, возможно, нарушить логику работы тех функций приложения, которые связаны с уровнем доступа «[admin](#)».

Помимо этого, кортежи в Python занимают меньший объем памяти при хранении того же самого набора значений. И само использование кортежей в программе вместо списков также повысит производительностью приложения.

К кортежу можно применять многие функции списков, кроме функций изменения данных.

Применение и особенности кортежа

Начнем с создания кортежей. Можно создать пустой кортеж (без элементов) или кортеж со значениями. Создать пустой кортеж можно одним из способов:

- используя оператор `()`: `myEmptyTuple = ()`;
- или с помощью функции-конструктора `tuple()`: `myEmptyTuple = tuple()`;

```
myEmptyTuple1=()
myEmptyTuple2= tuple()
```

Выведем содержимое и тип переменных `myEmptyTuple1` и `myEmptyTuple2`:

```
print(myEmptyTuple1)
print(type(myEmptyTuple1))

print(myEmptyTuple1)
print(type(myEmptyTuple2))
```

Результат:

```
()
<class 'tuple'>
()
<class 'tuple'>
```

Рисунок 1

Для того, чтобы создать кортеж с нужным содержимым можно перечислить его будущие элемента через запятую,

как при создании списка, но вместо квадратных скобок следует использовать круглые.

Также можно создать кортеж из одного или нескольких элементов просто перечислив их после оператора присваивания через запятую.

Примечание: в случае с одним элементов запятая также обязательна после его значения.

```
myTuple1=('element1', 12, 35.6, False)
myTuple2=('item1')
userTypes='admin','student','teacher'
userName='Jane',
```

```
print("myTuple1:", myTuple1,"type: ", type(myTuple1))
print("myTuple2:", myTuple2,"type: ", type(myTuple2))
print("userTypes:", userTypes,"type: ", type(userTypes))
print("userName:", userName,"type: ", type(userName))
```

Результат:

```
myTuple1: ('element1', 12, 35.6, False) type: <class 'tuple'>
myTuple2: item1 type: <class 'str'>
userTypes: ('admin', 'student', 'teacher') type: <class 'tuple'>
userName: ('Jane',) type: <class 'tuple'>
```

Рисунок 2

Как мы видим кортежи могут содержать данные любых типов. Также возможно дублирование значений внутри кортежа:

```
itemTuple=('item1','item2','item1','item3')
print(itemTuple) #('item1', 'item2', 'item1', 'item3')
```

Если мы будем создавать кортеж с несколькими элементами, используя функцию-конструктор `tuple()`, то нужно использовать двойные круглые скобки:

```
namesTuple = tuple(('Alex', 'Helen'))  
print(namesTuple) #('Alex', 'Helen')
```

Элементы кортежа индексируются также, как и элементы списка, первый элемент имеет индекс `[0]`, второй элемент имеет индекс `[1]` и т. д.

Отрицательное индексирование также возможно.

```
userTypes=('admin','student','teacher','moderator')  
  
print("1st user:", userTypes[0])  
print("last user:", userTypes[len(userTypes)-1])  
print("last user once again:", userTypes[-1])
```

Результат:

```
1st user: admin  
last user: moderator  
last user once again: moderator
```

Рисунок 3

Срезы, с особенностями которых мы уже познакомились, работая со списками также можно применять и для кортежей:

```
print("1st two users:", userTypes[:2])  
print("2nd and 3rd users:", userTypes[1:3])
```

Результат:

```
1st two users: ('admin', 'student')
2nd and 3rd users: ('student', 'teacher')
```

Рисунок 4

После создания кортежа мы не можем изменить его значения, добавить элементы в кортеж или удалить из него.

Подобная попытка:

```
userTypes=('admin', 'student', 'teacher', 'moderator')
userTypes[0]='user'
```

Вызовет такую ошибку:

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

Рисунок 5

Удаление отдельных элементов из кортежа невозможно. Однако мы можем при необходимости удалить весь кортеж, просто используйте оператор `del` для этого:

```
del tupleName
```

В Python нам также разрешено извлекать значения кортежей обратно в переменные. Этот подход называется «распаковка»:

```
userTypes=('admin','student','teacher','moderator')
user1, user2,user3,user4 =userTypes
print(user1) #admin
print(user2) #student
```



```
print(user3) #teacher
print(user4) #moderator
```

При использовании этого подхода количество переменных до знака присваивания должно совпадать с количеством значений в кортеже.

Если количество переменных левее знака присваивания меньше количества значений, то можно добавить символ звездочки `*` к имени переменной (последней), и тогда оставшиеся значения будут собраны в виде списка и присвоены этой переменной:

```
userTypes=('admin', 'student', 'teacher', 'moderator')

user1, *users =userTypes

print(user1) #admin
print(users) #['student', 'teacher', 'moderator']
```

Если звездочка добавлена к имени не последней переменной, то интерпретатор Python будет присваивать ей значения кортежа до тех пор, пока количество оставшихся значений не совпадет с количеством оставшихся после нее переменных.

```
userTypes=('admin', 'student', 'teacher', 'moderator')

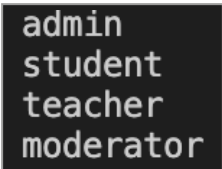
firstUser, *users, lastUser =userTypes

print(firstUser) #admin
print(users) #['student', 'teacher']
print(lastUser) #moderator
```

Мы можем обрабатывать элементы кортежа в цикле, как мы это делали со списками:

```
userTypes=('admin', 'student', 'teacher', 'moderator')
for user in userTypes:
    print(user)
```

Результат:



```
admin
student
teacher
moderator
```

Рисунок 6

Мы также можем перебирать элементы кортежа, обращаясь к их порядковому номеру (индексу), используя функции `range()` и `len()` для организации цикла.

```
userTypes=('admin', 'student', 'teacher', 'moderator')
for i in range(len(userTypes)):
    print(userTypes[i])
```

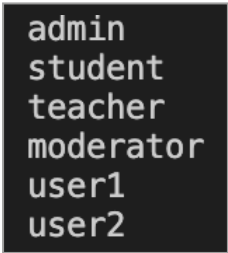
Результат будет аналогичен предыдущему.

Для того, чтобы соединить два или более кортежей, можно использовать оператор `+`:

```
userTypes1=('admin', 'student', 'teacher', 'moderator')
userTypes2=('user1', 'user2')

allUsers=userTypes1+userTypes2
for user in allUsers:
    print(user)
```

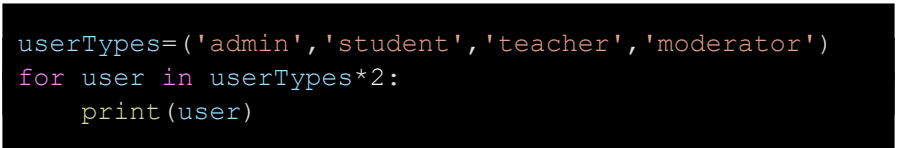
Результат:



```
admin  
student  
teacher  
moderator  
user1  
user2
```

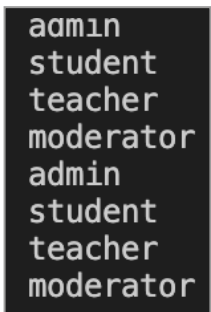
Рисунок 7

Если вы хотите продублировать содержимое кортежа нужное количество раз, используйте оператор *:



```
userTypes=('admin','student','teacher','moderator')  
for user in userTypes*2:  
    print(user)
```

Результат:



```
admin  
student  
teacher  
moderator  
admin  
student  
teacher  
moderator
```

Рисунок 8

В Python есть два встроенных метода, которые вы можете использовать для работы с кортежами.

Метод `count()` возвращает количество раз, когда указанное значение-аргумент появляется в кортеже:

```
userTypes=('admin', 'student', 'teacher', 'moderator',
          'admin')
print(userTypes.count('admin')) #2
```

Метод `index()` находит первое вхождение указанного значения-аргумента и возвращает его позицию:

```
userTypes=('admin', 'student', 'teacher', 'moderator',
          'admin')
print(userTypes.index('admin')) #0
```

Мы можем проверить, существует ли элемент в кортеже или нет, используя оператор `in`:

```
userTypes=('admin', 'student', 'teacher', 'moderator',
          'admin')
if 'student' in userTypes:
    print('student is correct login' )
```

Рассмотрим пример использования кортежей.

Предположим, что нам необходимо хранить и обрабатывать информацию о пользователях. Причем личная информация (фамилия, имя, год рождения и пол) вводится один раз в момент регистрации и дальше не должна изменяться и дополняться (т. е. новые сведения о пользователе, например, адрес в его личную информацию добавлять нельзя).

Также есть и информация о его интересах (перечень «хобби»), которые могут изменяться, перечень языков программирования, которые знает пользователь и список которых также может меняться.

Таким образом, удобно оформить пользовательскую информацию в виде списка, элементами которого будут: кортеж (для хранения неизменяемой личной информации); список интересов и список языков программирования.

Вначале создадим функцию, которая запрашивает у пользователя личную информацию и возвращает ее в кортеж:

```
def askPersonalInfo():
    while True:
        firstName=input("Input your first name:")
        lastName=input("Input your last name:")
        yearBirth=input("Input your year of birth:")
        gender=input("Input your gender (M,F):")
        if firstName=="" or lastName==""
           or yearBirth==""
           or gender==""
           or gender not in ('F','M'):
            print("Wrong data!")
        else:
            return firstName,lastName,yearBirth,gender
```

```
personalInfo=askPersonalInfo()
print(personalInfo)
```

Мы также использовали кортеж ('F', 'M'), чтобы проверить, что пользователь ввел допустимое значение для информации «пол».

Функция возвращает четыре значения, которые основная программа запишет в кортеж `personalInfo`.

Проверим, как наша функция обрабатывает ошибочные ситуации:

```

Your first name:anna
Your last name:smith
Your year of birth:1990
Your gender (M,F):a
Wrong data!
Your first name:
Your last name:a
Your year of birth:1990
Your gender (M,F):F
Wrong data!
Your first name:anna
Your last name:smith
Your year of birth:1990
Your gender (M,F):F
('anna', 'smith', '1990', 'F')

```

Рисунок 9

Идем дальше: создадим функцию для ввода информации об интересах пользователя и о языках программирования, которые он знает.

Процесс запроса хобби будет длиться до тех пор, пока пользователь не введет вместо названия пустую строку (признак окончания ввода). Организуем это логику с помощью бесконечного цикла и его останова оператором **break** по указанному условию.

```

def askHobby():
    hobbyInd = 1
    hobbyList = []
    while True:
        hobbyName = input("Name of the hobby #{:}".format(hobbyInd))
        if hobbyName == "":
            print("No info. Input stopped.")
            break
        else:
            hobbyList.append(hobbyName)

```

```

        hobbyInd+=1
    if len(hobbyList)>0:
        print("You have {} hobbies.".format(hobbyInd-1))
    else:
        print("You have no hobbies at all")

    return hobbyList

```

Проведем ее тестирование.

```

Name of the hobby #1:
No info. Input stopped.
You have no info at all

```

Рисунок 10

```

Name of the hobby #1:cooking
Name of the hobby #2:drawing
Name of the hobby #3:
No info. Input stopped.
You have 2 hobbies.

```

Рисунок 11

Теперь сделаем ее более универсальной, чтобы ее можно было использовать и для формирования списка языков программирования, которые знает пользователь: для этого часть строки запроса («hobby» «programming languages») будем передавать, как параметр нашей функции. Также сделаем имена ее локальных переменных более универсальными.

```

def askAdditionalInfo(queryStr):
    infoInd=1
    infoList=[]

```

```

while True:
    infoName=input("Name of the {} #{}:".
                    format(queryStr,infoInd))
    if infoName=="":
        print("No info. Input stopped.")
        break
    else:
        infoList.append(infoName)
        infoInd+=1

if len(infoList)>0:
    if queryStr=='hobby':
        print("You have {} hobbies.".
              format(infoInd-1))
    elif queryStr=='programming languages':
        print("You know {} programming languages."
              .format(infoInd-1))

else:
    print("You have no info at all")

return infoList

```

Весь код нашей программы:

```

def askPersonalInfo():
    while True:
        firstName=input("Your first name:")
        lastName=input("Your last name:")
        yearBirth=input("Your year of birth:")
        gender=input("Your gender (M,F) :")

        if firstName==" or lastName=="
            or yearBirth=="
            or gender=="
            or gender not in ('F','M'):

```



```

        print("Wrong data!")
    else:
        return firstName, lastName, yearBirth,
            gender

def askAdditionalInfo(queryStr):
    infoInd=1
    infoList=[]

    while True:
        infoName=input("Name of the {} #{}:".
            format(queryStr,infoInd))
        if infoName=="":
            print("No info. Input stopped.")
            break
        else:
            infoList.append(infoName)
            infoInd+=1

    if len(infoList)>0:
        if queryStr=='hobby':
            print("You have {} hobbies.".
                format(infoInd-1))
        elif queryStr=='programming languages':
            print("You know {} programming languages.".
                format(infoInd-1))

    else:
        print("You have no info at all")

    return infoList

userInfo=[]

userInfo.append(askPersonalInfo())

userInfo.append(askAdditionalInfo('hobby'))

```

```

userInfo.append(askAdditionalInfo('programming
                                languages'))

print(userInfo)

```

Результат:

```

Input your first name:Anna
Input your last name:Smith
Input your year of birth:2000
Input your gender (M,F):F
Name of the hobby #1:cooking
Name of the hobby #2:drawing
Name of the hobby #3:
No info. Input stopped.
You have 2 hobbies.
Name of the programming languages #1:JS
Name of the programming languages #2:Python
Name of the programming languages #3:C++
Name of the programming languages #4:
No info. Input stopped.
You know 3 programming languages.
[('Anna', 'Smith', '2000', 'F'),_['cooking', 'drawing'], ['JS', 'Python', 'C++']]

```

Рисунок 12

Поскольку кортежи очень похожи на списки, они оба используются в схожих ситуациях. Однако у кортежа есть определенные преимущества перед списком:

- обычно мы используем кортежи для разнородных (разных) типов данных и списки для однородных (похожих) типов данных;
- поскольку кортежи неизменяемы, итерация по кортежу выполняется быстрее, чем по списку (таким образом, есть небольшой прирост производительности при использовании кортежей);
- кортежи, содержащие неизменяемые элементы, можно использовать в качестве ключа для словаря (с которыми

мы познакомимся немного позже), в вот со списками такое невозможно;

- если у вас есть данные, которые не изменяются (не должны изменяться, например, перечень типов пользователей, с которыми работает приложение), то их реализация в виде кортежа гарантирует, что они останутся защищенными от записи.

2. Множества

Математическое понятие множеств

В повседневной жизни нам постоянно приходится иметь дело с разными наборами (совокупностями) объектов, поэтому понятия числа (количества) и *понятие множества* является очень распространенным. Множество является одним из фундаментальных понятий не только математики, но и в любой предметной области.

Множество — это произвольная совокупность (набор, коллекция) каких-либо объектов. Такие объекты, входящие в состав множества, называются *элементами множества*. Чаще всего они понимаются как единое целое, т.к. имеют общие характеристики, признаки, относятся к одному обстоятельству или критерию, подчиняются одному правилу и т. д.

Понятие множества абстрактно. Если мы рассматриваем некоторый набор как множество, то мы не делаем акцент на связях (отношениями) между ними, а сохраняем их индивидуальные характеристики. Например, множество из трех чашек и множество из трех символов — это абсолютно разные множества. При этом, если мы выставим чашки в ряд или сложим стопкой одну на другую, то это будет одно и тоже множество элементов (чашек).

Если некоторый объект x является элементом множества A , то x **принадлежит** A (пишется $x \in A$).

Обычно множества обозначаются прописными латинскими буквами A, B, C (при необходимости с подстрочными

индексами: A_1, A_2 и т. д.), а элементы множества записываются в фигурных скобках, например:

- $A = \{a, b, c, \dots, z\}$ — множество символов латинского алфавита;
- $S_1 = \{\text{Bob}, \text{Joe}, \dots, \text{Kate}\}$ — множество студентов первой группы.

Множества в наших примерах являются *конечными*, т.е. состоят из конечного числа элементов.

Также существует *пустое множество* — множество, не содержащее ни одного элемента.

Множество B является *подмножеством* множества A , если каждый элемент множества B принадлежит множеству A . Другими словами, множество B содержится во множестве A .

Например, есть множество студентов первой группы $S_1 = \{\text{Bob}, \text{Joe}, \dots, \text{Kate}\}$, тогда множество студентов этой группы, которые имеют средний балл больше $75 - S_{1\text{High}} = \{\text{Bob}, \text{Kate}\}$ — это подмножество множества S_1 . Или есть множество студентов университета $S_u = \{st_1, \dots, st_n\}$. Тогда множество S_1 является подмножеством множества S_u .

Операции над множествами

Если у нас есть несколько множеств, то мы можем выполнять над ними различные операции.

Схематически операции над множествами представляют в виде *кругов Эйлера*.

Круги Эйлера — это общепринятая геометрическая схема для визуализации логических связей между объектами (явлениями, понятиями) предметной области.

Каждому множеству соответствует круг, с помощью которого мы показываем, что все точки внутри круга принадлежат этому множеству, а остальные нет.

Начнем с операции *пересечения множеств* (*intersection*).

Пересечением множеств A и B называется такое множество, каждый элемент которого принадлежит обоим множествам (и множеству A , и множеству B), т.е. — это общие элементы множеств A и B .

Операция пересечения множеств — это логическое **И** (элемент принадлежит одновременно нескольким множествам, одновременное выполнение нескольких условий — это логическое **И**):

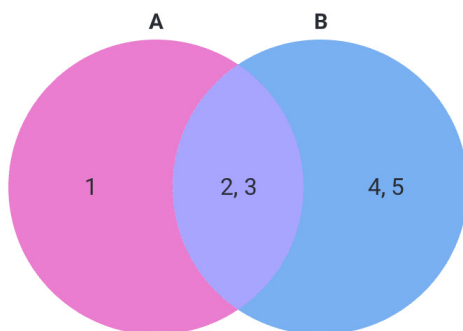


Рисунок 13

Обозначается, как $A \cap B$.

Например, $A = \{1, 2, 3\}$, а $B = \{2, 3, 4, 5\}$.

Тогда $A \cap B = \{2, 3\}$.

Если у множеств нет одинаковых элементов, то их пересечение пусто.

Объединением (*union*) множеств A и B — называется множество, каждый элемент которого принадлежит или множеству A или множеству B .

Операция объединения множеств — это логическое **ИЛИ**.

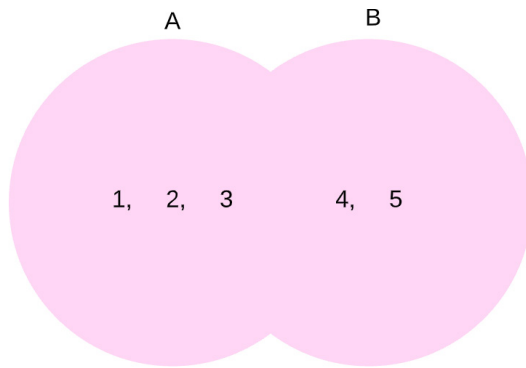


Рисунок 14

Рассмотрим на примере: $A = \{1, 2, 3\}$, а $B = \{3, 4, 5\}$.

$A \cup B = \{1, 2, 3, 4, 5\}$, т. е. это все элементы множества A и B без повторений (т. е. элемент 3 в пересечении будет только один раз).

Разностью (difference) множеств A и B называют такое множество, каждый элемент которого принадлежит множеству A и не принадлежит множеству B :

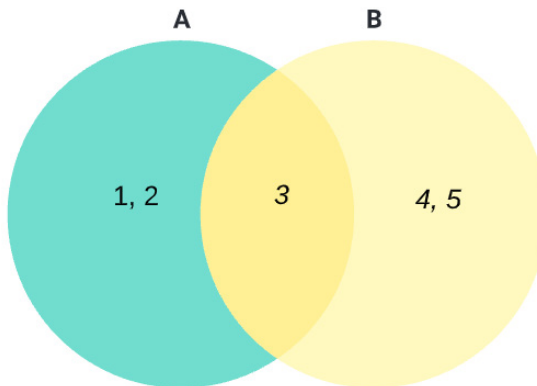


Рисунок 15

Рассмотрим на примере: $A = \{1, 2, 3\}$, а $B = \{3, 4, 5\}$.

$A \setminus B = \{1, 2\}$, т. е. это все элементы множества A , которых нет во множестве B .

Тип данных `set()`

Множество (*set*) в Python — это коллекция неповторяющихся элементов, которая является неупорядоченной (неиндексированной).

В отличие от кортежей во множество (после его определения) можно добавлять (удалять) элементы. Также доступны операции перебора элементов множества и математические операции над множествами: объединения, пересечения, разности. И, конечно, мы можем проверить, принадлежит ли определенный элемент некоторому множеству.

Элементы, входящие в состав множества могут быть любого неизменяемого типа данных (числовые, строки, кортежи, множества). Однако элементы изменяемых типов данных (например, списки) не могут войти во множество.

По аналогии с математикой в Python множества записываются внутри фигурных скобок.

Множество создается путем помещения всех элементов в фигурные скобки {}, разделяя их запятой:

```
mySet1 = {1, 2, 3}
mySet2 = {'Joe', 'Bob', 'Kate'}
mySet3 = {23, 'Bob', False, (45.6, 12)}

print("mySet1 — set of numbers:", mySet1)
print("mySet2 — set of strings:", mySet2)
print("mySet3 — of mixed datatypes:", mySet3)
```


Результат:

```
mySet1 - set of numbers: {1, 2, 3}
mySet2 - set of strings: {'Kate', 'Joe', 'Bob'}
mySet3 - of mixed datatypes: {False, (45.6, 12), 'Bob', 23}
```

Рисунок 16

или с помощью встроенной функции-конструктора `set()`, но тогда нужно использовать двойные круглые скобки:

```
mySet4=set((10,20,30))
print("mySet4:",mySet4) #{10, 20, 30}
```

При создании множества повторяющиеся элементы будут автоматически удалены:

```
uniqueUserName= {'Joe','Bob','Kate','Bob'}
print(uniqueUserName) #{'Kate', 'Joe', 'Bob'}
```

Поэтому множества удобно использовать для удаления дубликатов (например, из списка):

```
allUserName= ['Joe','Bob','Kate','Bob']
uniqueUserName=set(allUserName)
print(uniqueUserName) #{'Kate', 'Joe', 'Bob'}
```

При создании множества с элементом изменяемого типа получим ошибку: `passwordsSet={'111',['222','333']}`.

unhashable type: 'list'

Рисунок 17

Создать пустое множество с помощью пустых фигурных скобок. Пустые фигурные скобки `{}` создают пустой

словарь в Python. Для того, чтобы создать множество без каких-либо элементов, нужно использовать функцию `set()` без аргументов:

```
myEmptySet=set()  
print(myEmptySet) #set()
```

Как мы уже отмечали выше, элементы набора неупорядоченные. Неупорядоченный набор означает, что элементы в нем не имеют определенного порядка.

Проверим эту особенность на примере:

```
mySetA = {1, 2, 3}  
mySetB = {3, 2, 1}  
print(mySetA==mySetB) #True
```

Результат `True` говорит о том, что `mySetA` и `mySetB` — это одно и то же множество. Так и есть: набор элементов один и тот же, а их порядок в наборе не имеет значения.

Операции над множествами

Поскольку множества неупорядоченные, индексация не имеет смысла. Таким образом, мы не можем получить доступ к элементу множества или изменить его с помощью индексации или срезов.

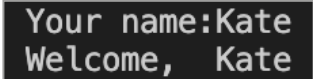
Однако, мы можем пройти по элементам множества, используя цикл `for`:

```
userNames= {'Joe','Bob','Kate'}  
for name in userNames:  
    print(name)
```

или проверить, присутствует ли указанное значение во множестве, используя ключевое слово `in`.

```
userNames= {'Joe', 'Bob', 'Kate'}  
name=input("Your name:")  
if name in userNames:  
    print("Welcome, ", name)
```

Результат:



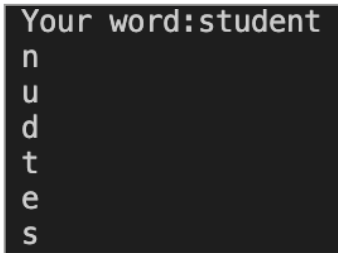
```
Your name:Kate  
Welcome, Kate
```

Рисунок 18

А теперь выведем уникальные символы введенного пользователем слова, используя множества:

```
word=input("Your word:")  
for letter in set(word):  
    print(letter)
```

Результат:



```
Your word:student  
n  
u  
d  
t  
e  
s
```

Рисунок 19

После создания множества вы не можете изменять его элементы, но можете добавлять новые элементы или удалять их из множества.

Мы можем добавить один элемент во множество с помощью метода `add()` и несколько элементов с помощью метода `update()`. Метод `update()` может принимать в качестве аргумента кортежи, списки, строки или другие множества. Во всех случаях дубликаты удаляются.

```
mySet = {1, 2, 3}
print(mySet)

mySet.add(4)
print(mySet)

mySet.update([3, 4, 5])
print(mySet)

mySet.update([5, 6, 7], {8, 9})
print(mySet)
```

Результат:

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

Рисунок 20

Мы также можем удалить конкретный элемент из множества с помощью методов `discard()` и `remove()`. Единственная разница между ними заключается в том, что метод `discard()` оставляет набор без изменений, если удаляемый элемент отсутствует. А вот метод `remove()` вызовет ошибку в таком случае.

```

mySet = {1, 2, 3, 4, 5}
print(mySet)

mySet.discard(4)
print(mySet)

mySet.remove(5)
print(mySet)

mySet.discard(10)
print(mySet)

```

Результат:

```

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

```

Рисунок 21

Как мы видим попытка удалить элемент 10 (которого нет во множестве) с помощью метода `discard()` не изменила наше множество.

А вот если мы попробуем повторить это с помощью метода `remove()`, то получим ошибку:

```
mySet.remove(10)
```

KeyError ✕

Рисунок 22

Мы также можем удалить все элементы из множества, используя метод `clear()`.

Python предоставляет нам набор встроенных методов, которые выполняют рассмотренные нами операции пересечения, объединения и разности множеств. Мы можем выполнять перечисленные операции над множествами с помощью специальных методов или с помощью операторов `&`, `|`, `-`.

Рассмотрим их детальнее.

Пересечение выполняется с помощью оператора `&`. То же самое можно сделать с помощью метода `intersection()`.

Объединение множеств доступно нам с через оператор `|` или метод `union()`.

Разность множеств выполняется с помощью оператора `-`. То же самое можно сделать с помощью метода `different()`.

```
studGroup1={'Hanna','Joe','Kate'}
studGroup2={'Bob','Joe','Jane','Kate','Jack'}

print("studGroup1:",studGroup1)
print("studGroup2:",studGroup2)

print("Intersection of sets:")
print(studGroup1&studGroup2) #
print(studGroup1.intersection(studGroup2))

print("Union of sets:")
print(studGroup1|studGroup2) #
print(studGroup1.union(studGroup2))

print("Difference of two sets:")
print(studGroup2-studGroup1) #
print(studGroup2.difference(studGroup1))
```

Результат:

```
studGroup1: {'Kate', 'Hanna', 'Joe'}
studGroup2: {'Bob', 'Joe', 'Kate', 'Jane', 'Jack'}
Intersection of sets:
{'Kate', 'Joe'}
{'Kate', 'Joe'}
Union of sets:
{'Bob', 'Joe', 'Jack', 'Kate', 'Jane', 'Hanna'}
{'Bob', 'Joe', 'Jack', 'Kate', 'Jane', 'Hanna'}
Difference of two sets:
{'Jack', 'Bob', 'Jane'}
{'Jack', 'Bob', 'Jane'}
```

Рисунок 23

Также при работе со множествами доступны уже известные нам встроенные функции, такие как `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` и т. д., которые полезны при выполнении различных задач.

Примечание: функция `enumerate()` возвращает перечисляемый объект, который содержит индекс и значение для всех элементов набора в виде пары (кортежа).

Рассмотрим их применение со множествами на примерах:

```
studSet={'Bob', 'Joe', 'Jane', 'Kate', 'Jack'}

print("We have {} students in our
group.".format(len(studSet)))

for ind,item in enumerate(studSet):
    print(ind,item)
```

Результат:

```
We have 5 students in our group.
0 Jack
1 Joe
2 Jane
3 Bob
4 Kate
```

Рисунок 24

```
scores={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

print("Min score is", min(scores))
print("Max score is", max(scores))
print("Sum of scores:", sum(scores))
```

Результат:

```
Min score is 1
Max score is 12
Sum of scores: 78
```

Рисунок 25

frozenset()

Frozenset — это дополнительный класс в Python, который имеет все характеристики множества (**set**), но его содержимое нельзя изменять. Также, как мы называем кортежи неизменяемыми списками, мы можем называть **frozenset** неизменяемыми множествами.

Этот тип данных поддерживает рассмотренные подходы для объединения, пересечения и нахождения разницы для множеств.

```
frozenA = frozenset(['Hanna', 'Joe', 'Kate'])
frozenB = frozenset(['Bob', 'Joe', 'Jane', 'Kate', 'Jack'])
```



```

print("frozenA:", frozenA)
print("frozenB:", frozenB)

print("Intersection of frozensets:")
print(frozenA&frozenB)
print(frozenA.intersection(frozenB))

print("Union of frozensets:")
print(frozenA|frozenB)
print(frozenA.union(frozenB))

print("Difference of two frozensets:")
print(frozenB-frozenA)
print(frozenB.difference(frozenA))

```

Результат:

```

frozenA: frozenset({'Joe', 'Kate', 'Hanna'})
frozenB: frozenset({'Jack', 'Bob', 'Kate', 'Joe', 'Jane'})
Intersection of frozensets:
frozenset({'Kate', 'Joe'})
frozenset({'Kate', 'Joe'})
Union of frozensets:
frozenset({'Jack', 'Hanna', 'Bob', 'Kate', 'Joe', 'Jane'})
frozenset({'Jack', 'Hanna', 'Bob', 'Kate', 'Joe', 'Jane'})
Difference of two frozensets:
frozenset({'Bob', 'Jane', 'Jack'})
frozenset({'Bob', 'Jane', 'Jack'})

```

Рисунок 26

Однако в связи с неизменяемостью **frozenset** не поддерживает методы для добавления элементов во множество и удаления из него.

```
frozenA.add('user')
```

'frozenset' object has no attribute 'add'

Рисунок 27

```
frozenB.remove('Bob')
```

```
'frozenset' object has no attribute 'remove'
```

Рисунок 28

Из-за возможности изменения своего содержимого обычные множества нельзя использовать в качестве ключей словарей, с которыми мы познакомимся немного позже. А вот **frozenset** подходят для этих целей.

Применение множеств

Как уже упоминалось выше, использование множеств позволяет легко решить проблему удаления дубликатов (например, из списка). Допустим, что у нас есть общий список категорий пицц (например, из нескольких пиццерий), который содержит повторяющиеся элементы. И нам нужно создать список, содержащий только уникальные названия пицц. С помощью множества эта задача просто решается парой строк кода:

```
allPizzaTypes=['Veggie', 'Pepperoni', 'Meat',
               'Margherita', 'Meat', 'BBQ Chicken',
               'Hawaiian', 'Veggie']

uniquePizzaTypes = list(set(allPizzaTypes))
print(uniquePizzaTypes)
['BBQ Chicken', 'Veggie', 'Meat', 'Margherita',
 'Hawaiian', 'Pepperoni']
```

Теперь рассмотрим следующую ситуацию: у нас есть логины клиентов двух приложений, которые хранятся

отдельно (например, в двух списках). Нам нужно определить:

- какие клиенты пользуются обоими приложениями;
- какие клиенты пользуются только первым (только вторым) приложением;
- общий перечень логинов клиентов.

Первая задача — это пересечение двух множеств, вторая — нахождение разницы между первым и вторым (и наоборот) и третья задача — это объединение множеств.

```
usersApp1=['user134', 'admin56', 'superBob', 'student',
           'spider34']
usersApp2=['fifa56', 'user134', 'studGood', 'admin56',
           'spider34']

print("App1+App2 users:")
print(set(usersApp1)&set(usersApp2))
print("App1 users only:")
print(set(usersApp1)-set(usersApp2))
print("App2 users only:")
print(set(usersApp2)-set(usersApp1))
print("All users:")
print(set(usersApp1)|set(usersApp2))
```

Результат:

```
App1+App2 users:
{'admin56', 'spider34', 'user134'}
App1 users only:
{'student', 'superBob'}
App2 users only:
{'studGood', 'fifa56'}
All users:
{'admin56', 'user134', 'superBob', 'spider34', 'studGood', 'student', 'fifa56'}
```

Рисунок 29

И еще один пример: у нас есть два слова и нужно определить, является ли второе слово анаграммой первого.

Анаграмма — приём, состоящий в перестановке букв определённого слова, что в результате даёт другое слово.

Мы помним, что порядок элементов во множестве не имеет никакого значения. Соответственно, если множества состоят из одного набора символов, которые находятся в разном порядке, то фактически это одно и то же множество.

```
word1=input("1st word:")
word2=input("2nd word:")

if set(word1)==set(word2):
    print("Yes")
else:
    print("No")
```

Результат:

Кейс 1

```
1st word:angel
2nd word:glean
Yes
```

Рисунок 30

Кейс 2

```
1st word:night
2nd word:thing
Yes
```

Рисунок 31

Кейс 3

```
1st word:inch
2nd word:china
No
```

Рисунок 32

3. Словари

Ассоциативные массивы

Большая часть изученных нами структур, которые являются коллекциями, основаны на идее массивов, т. е. упорядоченных наборов элементов.

Упорядоченность — это свойство коллекции (набора), когда каждый ее элемент характеризуется не только своим значением, но и индексом (порядковым номером элемента в коллекции).

Таким образом, при работе со списками или кортежами мы используем индексацию для доступа к определенному элементу. При этом индексом является только целочисленное значение.

Однако, использование числа для идентификации значения (некоторой информации об объекте) не всегда удобно и может противоречить реальной ситуации в предметной области.

Например, для идентификации рейса самолета используется буквенно-цифровой код. Или, если по логину пользователя (которые уникальны) нужно узнать пароль?

Представим, что мы можем использовать только индексы в этих ситуации. Тогда для решения задачи нам нужно уже два списка. Будем хранить в одном списке логины, находить нужный логин и получать его индекс. Далее по этому индексу нужно извлекать пароль из другого списка. Это, конечно, возможно, но очень неудобно и запутано.

Гораздо удобнее было бы использовать связку «логин» — «пароль», где «логин» был бы идентификатором.

Абстрактная структура данных, которая в качестве индекса может использовать произвольный тип данных (а не только целые числа), называется **ассоциативным массивом**. Доступ к элементу в этом случае осуществляется через ключ.

Такая ассоциация является удобной и однозначно отображает многие реальные связи в предметных областях. Например, нам нужно узнать средний балл студента *Joe Smith*, мы не знаем (и не должны) его порядковый номер в группе студентов. Мы получаем нужную информацию (средний балл) через его фамилию. В этом случае фамилия являлась бы ключом ассоциативного массива, а средний балл — значением.

Хеш-таблицы

Для представления ассоциативных массивов в памяти Python использует хеш-таблицы. Термин «хеш-таблица» говорит о реализации: конкретном способе организации ваших данных в памяти.

Хеш-таблицы — это способ организации данных типа «ключ-значение» в памяти в виде специальной структуры, где индекс (позиция элемента в структуре) является результатом работы хеш-функции.

Для получения индекса в хэш-таблице ключи преобразуются как раз с помощью функции хеширования. А сам процесс такого преобразования называется **хешированием**.

Фактически хеш-функция — это функция, которая получает строку (ключ) и возвращает число (индекс).

Абстрактно это процесс можно представить так:

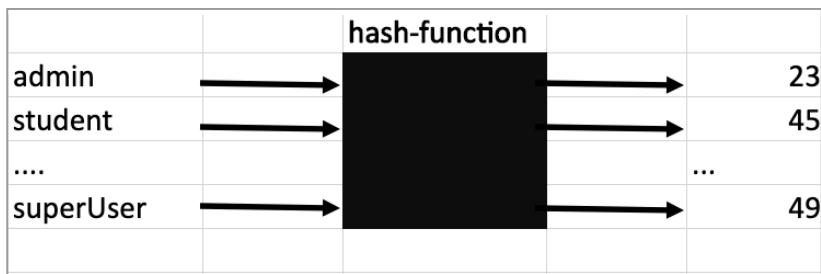


Рисунок 33

К хеш-функции предъявляются следующие требования:

- последовательность: предположим, что мы передали хеш-функции строку «**admin**» и получили **23** и каждый следующий раз, когда мы будем передавать ей строку «**admin**», будем получать только **23**.
- разным ключам должны соответствовать разные индексы (у каждого входного слова должно быть свое число).

Как это работает?

Допустим, что все значения средних баллов студентов будут храниться в массиве.

Мы подаем на вход хеш-функции строку «*Joe Smith*». Хеш-функция выдает значение «**2**». Сохраняем средний балл этого студента в элементе массива с индексом **2**.

При таком подходе к организации данных операции поиска и вставки элемента в набор выполняются быстрее, так как значения ключей преобразуются в индексы массива, в котором хранятся сами данные.

Схематически это можно представить так:

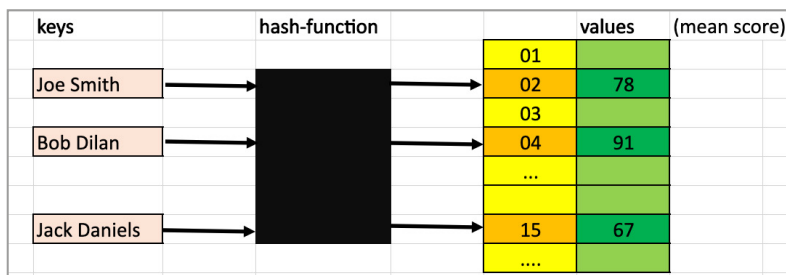


Рисунок 34

А теперь предположим, что нам нужно узнать средний балл студента *Bob Dylan*. Искать в массиве ничего не придется, мы просто передадим ключ «*Bob Dylan*» нашей хеш-функции, получим индекс **4** и по этому индексу извлечем из массива средний балл этого студента — **91**.

Хеш-функция всегда знает размер массива (N) и будет возвращать индексы в его пределах (т. е. от **0** до $N-1$).

Отдельная проблема, с которой мы можем столкнуться при создании хэш-таблицы — это коллизии (столкновения). Эта ситуация может возникнуть, когда функцией хеширования выдаст один и тот же индекс для разных ключей. Для избегания столкновений или уменьшения их количества нужна хорошая функция хеширования.

Хорошей считается хеш-функция, для которой выполняются следующие условия:

- при подаче на вход функции одного и того же ключа мы всегда будем получать одно и тоже хеш-значение (индекс);
- не может существовать двух разных ключей, для которых функция вернёт одно и тоже хеш-значение;
- даже небольшое изменение (отличие) ключа даст настолько отличное хеш-значение, что между предыдущим

и новым хеш-значениями нельзя обнаружить связи (зависимости);

- скорость вычисления хеш-значения приблизительно одинакова для любого значения ключа.

В Python большая часть неизменяемых типов данных (например, кортежи) являются хешируемыми, т.е. имеют хеш-значение. Для выполнения операции хеширования (в том числе и неявного, например, при создании словарей, с которыми мы познакомимся далее) интерпретатором Python используется встроенная функция `hash()`. Данная функция удовлетворяет всем перечисленным выше требованиям, поэтому выполнять подбор и реализацию подходящей хеш-функции Python-разработчику нет необходимости.

Создание словаря

В Python реализация хеш-таблиц представлена в виде типа данных (коллекции) словарь (*Dictionary*).

Словарь — это коллекция элементов, каждый из которых представляет собой пару «ключ-значение».

Изученные нами ранее списки на практике больше подходят для хранения однородных данных. Например, список цен на 10 товаров, которые являются численными значениями (целыми или вещественными) удобно хранить в списке и обрабатывать по номеру товара.

А вот разнородную информацию, например, о характеристиках какого-то объекта предметной области удобнее обрабатывать, ссылаясь на названия этих характеристик.

Допустим, что у нас есть такая информация о клиенте: имя, фамилия, возраст, логин, дата регистрации в системе.

Конечно, можно хранить эти данные в списке:

```
['Joe', 'Smith', 25, 'admin25', '10.01.2022']
```

Однако, при такой организации данных для получения, например, возраста нужно четко знать, что это третий элемент в списке.

Гораздо удобнее обрабатывать эти данные, представив их виде:

```
firstName: 'Joe',  
lastName: 'Smith',  
age: 25,  
login: 'admin25',  
signUpDate: '10.01.2022'
```

Мы можем обращаться к данным по ключу, а не по порядковому номеру в коллекции. Именно такую организацию данных и обеспечивает нам словарь.

При этом ключи в словаре должны быть уникальными, т. к. они будут обрабатываться функцией хеширования для получения уникальных индексов в хеш-таблицах. Поэтому в словарях не может быть двух элементов с одинаковым ключом.

Элементы словаря не упорядочены, т. е. порядок элементов данных в словаре не фиксирован. Поэтому для доступа к элементам используется не их порядковый номер в наборе (индекс), а ключ.

В Python ключом может быть объект любого неизменяемого типа данных: строка, булевская переменная,

число (целое или вещественное), кортеж. Также ключом в словаре может быть элемент типа `frozenset`.

Значение элемента словаря может быть любого типа данных и иметь неограниченный уровень вложенности.

Словари можно изменять: добавлять новые пары «ключ-значение», обновлять существующие значения в парах (получив доступ по ключу), удалять элементы словаря.

Примечание: *изменять значения ключей в уже существующих элементах (парах) словаря нельзя.*

Аналогично рассмотренным ранее коллекциям словари также можно создавать или пустыми, или с некоторым набором элементов.

Для того, чтобы *создать словарь с данными* нужно набор пар в формате «ключ: значение» поместить в фигурные скобки `{}`. Внутри `{}` пары разделяются символом запятой.

Формат словаря:

```
D = {
    <key1>: <value1>,
    <key>2: <value2>,
    .
    .
    .
    <keyM>: <valueM>
}
```

Например:

```
myDict1= {'key1': 1, 'key2': 20.5, 'key3': True}
myDict2= {1: 'student', 2: 'admin'}
```

```
print(myDict1)  #{'key1': 1, 'key2': 20.5, 'key3': True}
print(myDict2)  #{1: 'student', 2: 'admin'}
```

Создадим словарь, содержащий характеристики книги:

```
bookDict = {'author': 'Eric Matthes',
            'title': 'Python Crash Course',
            'price': 14.43,
            'reading age': '12 years and up',
            'language': 'English'
            }
```

Также для создания словаря можно использовать встроенную функцию `dict()`. В качестве аргументов ей можно передать любую последовательность (коллекцию), которая содержит наборы, состоящие из двух элементов (т. е. пары). Тогда первый элемент набора станет ключом в паре, а второй — значением.

Рассмотрим на примерах варианты создания словаря с помощью функцию `dict()`:

- из списка кортежей

```
myDict3 = dict([("a", 111), ("b", 222)])
```

- из списка списков

```
myDict4 = dict([["a", 111], ["b", 222]])
```

- из списка двухсимвольных строк

```
myDict5 = dict(['qw', 'er', 'ty'])
print(myDict5)  #{'q': 'w', 'e': 'r', 't': 'y'}
```

- из двух списков (списка ключей и списка значений) с помощью функции `zip()`.

```
keyList=['a','b']
valueList=[111,222]

myDict6=dict(zip(keyList,valueList))

print(myDict6)    #{'a': 111, 'b': 222}
```

Также можно использовать функцию `dict()` и таким способом:

```
myDict7=dict(firstName='Joe', lastName='Smith')
print(myDict7)    #{'firstName': 'Joe', 'lastName': 'Smith'}
```

Если наши ключи являются простыми строками (стоящими из одного слова), то их можно указать в качестве аргументов функции `dict()`.

Если нам необходимо *создать пустой словарь*, то это также можно сделать двумя способами:

```
myEmptyDict1={}
print(myEmptyDict1)    #{}

myEmptyDict2=dict()
print(myEmptyDict2)    #{} 
```

Для определения количества элементов (пар) в словаре, используйте встроенную функцию `len()`:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
```

```

        'price':14.43,
        'reading age':'12 years and up',
        'language':'English'
    }
    print(len(bookDict))    #5

```

Методы словаря

Как мы знаем, наиболее распространенная операция при работе с коллекциями — это получение значения элемента коллекции. Так же и при работе со словарями: ни одна задача не обходится без получения значения элемента словаря.

Словари являются неупорядоченными наборами, поэтому получение элемента по индексу невозможно. Обращение к элементам словаря происходит с помощью ключа, который указывается в квадратных скобках (как, например, при работе со списками мы указывали индекс):

```
имяСловаря[имяКлюча]
```

```

myDict1= {'key1': 1, 'key2': 20.5, 'key3': True}

print(myDict1['key1'])    #1

bookDict = {'author':'Eric Matthes',
            'title':'Python Crash Course',
            'price':14.43,
            'reading age':'12 years and up',
            'language':'English'
        }
print(bookDict['author']) #Eric Matthes

```

Если при такой операции мы укажем имя ключа, которого нет в словаре, то будет вызвано исключение `KeyError`.

```
print(bookDict['pages'])
```

KeyError ✕

Рисунок 35

Для извлечения элемента по ключу также существует метод `get()`, с которым мы познакомимся немного позже.

Для предотвращения показанной выше ситуации (обращение к несуществующему ключу), перед операцией извлечения элемента по ключу можно проверить, существует ли такой ключ в словаре. Если нам нужно проверить, есть ли определенный (интересующий нас) ключ в словаре, можно воспользоваться оператором `in`.

Если ключ найден в списке ключей словаря, то мы получим значение `True`, иначе — `False`.

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
}
```

```
infoType=input("What info do you want to know
               about the book?")

if infoType in bookDict:
    print(bookDict[infoType])
else:
    print("Sorry!")
```

Результат:

```
What info do you want to know about the book?page
Sorry!
```

Рисунок 36

```
What info do you want to know about the book?title
Python Crash Course
```

Рисунок 37

Так как словари являются изменяемыми типами данных, то мы можем изменять и добавлять элементы по ключу. Для добавления нового элемента (пары «ключ-значение») нужно просто обратиться к элементу по новому ключу и задать ему новое значение с помощью операции присваивания:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
}
```

```
for dictKey, dicVal in bookDict.items():
    print("{}:{}".format(dictKey,dicVal))
bookDict['pagesN']=350
for dictKey, dicVal in bookDict.items():
    print("{}:{}".format(dictKey,dicVal))
```

Результат:

```
author:Eric Matthes
title:Python Crash Course
price:14.43
reading age:12 years and up
language:English
```

Рисунок 38


```
author:Eric Matthes
title:Python Crash Course
price:14.43
reading age:12 years and up
language:English
pagesN:350
```

Рисунок 39

Также можно добавить пару, используя переменные для ключа и значения элемента:

```
newInfo=input("What info about the book do you want
              to add?")
if newInfo!="":
    newInfoValue=input("Input value for the key
                       '{}':".
                       format(newInfo))
    if newInfoValue!="":
        bookDict[newInfo]=newInfoValue
    else:
        print("No value for the key '{}':".
              format(newInfo))
else:
    print("No key!")
```

Если мы проведем описанную операцию для уже существующего в словаре ключа, то его предыдущее значение будет заменено на новое:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
}
```

```

for dictKey, dicVal in bookDict.items():
    print("{}:{}".format(dictKey,dicVal))

bookDict['price']=12

for dictKey, dicVal in bookDict.items():
    print("{}:{}".format(dictKey,dicVal))

```

Результат:

```

author:Eric Matthes
title:Python Crash Course
price:14.43
reading age:12 years and up
language:English

```

Рисунок 40

```

author:Eric Matthes
title:Python Crash Course
price:12
reading age:12 years and up
language:English

```

Рисунок 41

Интерпретатор Python «контролирует» тот момент, что ключ должен быть в словаре уникальным. Поэтому при попытке добавить в словарь новый элемент с существующим ключом происходит обновление значения по ключу (как в нашем примере).

Большинство операций для работы со словарями организованы в виде соответствующих методов словарей. Рассмотрим наиболее популярные и полезные из них.

Метод словаря Python `.get()` — удобный способ получения значения по ключу из словаря без предварительной

проверки на существования ключа (и без возникновения исключения `KeyError`).

Общий синтаксис:

```
dictName.get(<keyname>[, <defaultValue>])
```

Метод ищет в словаре `dictName` ключ `keyname` и возвращает соответствующее значение, если оно найдено, то возвращается значение `defaultValue`. Если параметр `defaultValue` не задан при вызове метода, то возвращается значение `None`:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
        }
```

```
print(bookDict.get('author')) #Eric Matthes
print(bookDict.get('page'))   #None
print(bookDict.get('page',0)) #0
```

Для обновления словаря Python предоставляет метод `update()`:

```
dictName.update(<iterObj>)
```

Метод `update()` обновит словарь элементами, которые представлены в составе аргумента-объекта. Аргумент должен быть словарем или итерируемым объектом с парами «ключ:значение» (например, списком кортежей или списком списков).

Если `iterObj` является словарем, то будет выполнено объединение записей из `iterObj` с записями в `dictName` по следующему алгоритму (Для каждого ключа в `iterObj`):

- если ключ отсутствует в `dictName` к `dictName` добавляется соответствующая пара «ключ-значение» из `iterObj`;
- если ключ уже присутствует в `dictName`, соответствующее значение в `dictName` для этого ключа обновляется до значения из `iterObj`.

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
}

print(bookDict)
```

```
bookDict.update([('pages', 600), ('discount', True)])
print(bookDict)

bookDict.update([('pages', 700), ('online', False)])
print(bookDict)
```

Результат:

```
{'author': 'Eric Matthes', 'title': 'Python Crash Course',
 'price ': 'English'}
{'author': 'Eric Matthes', 'title': 'Python Crash Course',
 'price ': 'English', 'pages': 600, 'discount': True}
{'author': 'Eric Matthes', 'title': 'Python Crash Course',
 'price ': 'English', 'pages': 700, 'discount': True,
 'online': False}
```

Рисунок 42

Данный метод очень удобен, когда нужно скопировать все элементы из одного словаря в другой.

```
studGr1={'Joe':75,'Bob':92}
studGr2={'Kate':62,'Joe':90, 'Jack':84}

print(studGr1) #{'Joe': 75, 'Bob': 92}
studGr1.update(studGr2)
print(studGr1) #{'Joe': 90, 'Bob': 92, 'Kate': 62,
                'Jack': 84}
```

Удалить элемент из словаря можно с помощью встроенной функции `del()`:

```
studGr2={'Kate':62,'Joe':90, 'Jack':84}
del studGr2['Jack']
print(studGr2) #{'Kate': 62, 'Joe': 90}
```

Для удаления всех элементов из словаря нужно использовать метод `clear()`:

```
studGr2.clear()
print(studGr2) #{} 
```

Для того, чтобы получить все ключи словаря (в виде списка) будем использовать метод `keys()`:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
        }
```

```
print(bookDict.keys())
#dict_keys(['author', 'title', 'price', 'reading age',
            'language'])
```

```
keyList=list(bookDict.keys())
print(keyList) #['author', 'title', 'price',
               'reading age', 'language']
```

А для получения всех значений словаря (в виде списка) будем использовать `values()`:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
        }
```

```
valuesList=list(bookDict.values())
print(valuesList)
#['Eric Matthes', 'Python Crash Course', 14.43,
  '12 years and up', 'English']
```

Метод `items()` возвращает список кортежей, содержащих пары «ключ-значение» из словаря. Первый элемент в каждом кортеже — это ключ, а второй — значение ключа:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
        }
```

```
dictItems=list(bookDict.items())
print(dictItems)
```

Результат:

```
[('author', 'Eric Matthes'), ('title', 'Python Crash Course'),
 ('price', 14.43), ('reading age', '12 years and up1') ,
 ('language', 'English')]
```

Рисунок 43

Метод `pop()` используется для удаления ключа из словаря. Результат работы — удаление элемента и возвращение значения по ключу:

```
dictName.pop(<keyname>[, <defaultValue>])
```

Если ключа `keyname` в словаре `dictName` нет, то будет возвращено значение `defaultValue`.

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
        }
print(bookDict)
```

```
delItem=bookDict.pop("price")
print("item {} was deleted".format(delItem))
print(bookDict)

delItem=bookDict.pop("discount","None")
print(delItem)
```

Результат:

```
{'author': 'Eric Matthes', 'title': 'Python Crash Course',
'price': 14.43, 'reading age': '12 years and up',
'language': 'English'} item 14.43 was deleted
{'author': 'Eric Matthes', 'title': 'Python Crash Course',
'reading age': '12 years and up', 'language': 'English'}
None
```

Рисунок 44

Если при вызове метода `pop()` не было задано значение параметра `defaultValue` и указанного ключа тоже нет, то будет вызвано исключение `KeyError`.

```
delItem=bookDict.pop("discount")
```

KeyError ✕

Рисунок 45

Для перебора элементов словаря традиционно используется цикл `for`. При этом на каждой итерации цикла мы получаем ключ словаря.

Выведем все ключи в словаре (каждый с новой строки, в столбик):

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
}
```



```
for dictKey in bookDict:
    print("{}:{}".format(dictKey, bookDict[dictKey]))
```

Результат:

```
author
title
price
reading age
_language
```

Рисунок 46

Имея доступ к ключам, мы можем немного изменить код и вывести все значения элементов словаря:

```
for dictKey in bookDict:
    print(bookDict[dictKey])
```

Результат:

```
Eric Matthes
Python Crash Course
14.43
12 years and up
English
```

Рисунок 47

Или так, для более удобного представления информации:

```
for dictKey in bookDict:
    print("{}:{}".format(dictKey,bookDict[dictKey]))
```

Результат:

```
author:Eric Matthes  
title:Python Crash Course  
price:14.43  
reading age:12 years and up  
language:English
```

Рисунок 48

Как уже упоминалось ранее, ключи словаря являются неизменяемым набором, т.е мы можем добавить или удалить полностью пару «ключ-значение», поменять значение по ключу, но не сам ключ.

Таким образом, если в предыдущий пример внутри цикла добавить строку кода, изменяющую переменную `dictKey`, то будет выполнено только изменение значения этой переменной, а не текущего значения ключа из `bookDict`, который в этой итерации находился в переменной `dictKey`.

```
for dictKey in bookDict:  
    dictKey='name'  
    print(bookDict[dictKey])
```

В данном примере мы вообще получим исключение

KeyError ✕

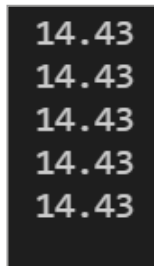
Рисунок 49

т.к. ключа `name` нет в словаре `bookDict`.

Если же мы зададим переменной `dictKey` значение существующего ключа (например, `price`),

```
for dictKey in bookDict:
    dictKey='price'
    print(bookDict[dictKey])
```

то мы продублируем вывод значения по этому ключу столько раз, сколько будет выполняться цикл (а именно столько раз, сколько ключей есть на момент запуска цикла в словаре `bookDict`, т.к. мы используем оператор `in`).



```
14.43
14.43
14.43
14.43
14.43
```

Рисунок 50

Перебор всех ключей и значений словаря возможен в цикле с использованием метода `items()`:

```
bookDict={'author':'Eric Matthes',
          'title':'Python Crash Course',
          'price':14.43,
          'reading age':'12 years and up',
          'language':'English'
        }
```

```
for dictKey, dicVal in bookDict.items():
    print("{}:{}".format(dictKey,dicVal))
```

Результат:

```
author:Eric Matthes  
title:Python Crash Course  
price:14.43  
reading age:12 years and up  
language:English
```

Рисунок 51

Мы не можем сделать копию словаря простым присваиванием (как и в ситуации со списками)

```
myDict2=myDict2
```

Потому что в этом случае `myDict2` будет только ссылкой на `myDict1`, и изменения, сделанные в `myDict1`, будут автоматически сделаны и в `myDict2`.

Для создания копии нужно использовать метод `copy()`.

```
myDict1= {'key1': 1, 'key2': 20.5, 'key3': True}  
myDict2=myDict1.copy()  
  
myDict1['key1']=111  
print(myDict1['key1']) #111  
print(myDict2['key1']) #1
```

4. Практические примеры использования

Теперь, когда мы познакомились с особенностями всех коллекций в Python (списки, кортежи, множества и словари), рассмотрим несколько примеров их использования, наиболее часто встречающихся на практике.

Пример 1. Поиск в словаре

Задача поиска информации встречается в каждом приложении. Поэтому давайте рассмотрим, как искать нужный элемент в словаре.

Мы знаем, что обращение к элементам словаря происходит через ключ. Поэтому необходимо будет пройтись по элементам списка (словаря) проверяя нужную пару «ключ — значение» на соответствие критерию поиска.

Допустим, что у нас есть список пользователей и по каждому из них известна такая информация: имя, возраст, логин. Представим этот набор данных в виде списка словарей, где каждый словарь хранит указанную информацию о пользователе.

```
users = [  
    {'name': 'Hanna', 'age': 10, 'login': 'user56'},  
    {'name': 'Mark', 'age': 15, 'login': 'usER111'},  
    {'name': 'Jane', 'age': 17, 'login': 'superGirl'},  
    {'name': 'Jack', 'age': 7, 'login': 'userJack'}  
]
```

Вначале спросим у пользователя, по какому полю будем проводить поиск и узнаем также критерий поиска (нужное пользователю значение ключа). И далее, как мы уже определились, переберем в цикле все элементы списка (словари), проверяя нужный нам ключ на нужное значение.

Как только встретим нужный элемент, то сразу остановим поиск. Для этого введем переменную `isElementFound`, если после окончания поиска ее значение все еще остается `False`, то, значит, поиск был неуспешен — нужного нам элемента (пользователя с нужными характеристиками) в наборе нет.

```
keyName=input("Input info type:")
keyValue=input("Input info value:")

isElementFound=False

for user in users:
    if user.get(keyName)==keyValue:
        print("Element is found!")
        for key,val in user.items():
            print("{}:{}".format(key,val))
            isElementFound=True
        break
if isElementFound==False:
    print("Element is not found!")
```

Результат:

```
Input info type:login
Input info value:usER111
Element is found!
name:Mark
age:15
login:usER111
```

Рисунок 52

```
Input info type:name
Input info value:Joe
Element is not found!
```

Рисунок 53

Однако, если мы попробуем осуществить такой поиск по полю «age», то результат поиска всегда будет неуспешен:

```
Input info type:age
Input info value:10
Element is not found!
```

Рисунок 54

Причина в том, что поле «age» содержит числовое значение, а результатом выполнения строки кода:

```
keyValue=input("Input info value:")
```

будет строка.

Нам для этого случая (поиска по возрасту) нужно сделать преобразование переменной *keyValue* в число (например, с помощью тернарного оператора) перед ее дальнейшим использованием в процессе поиска:

```
keyValue=keyValue if keyName!='age' else int(keyValue)
```

Результат:

```
Input info type:age
Input info value:15
Element is found!
name:Mark
age:15
login:usER111
```

Рисунок 55

Пример 2. Сортировка элементов словаря

Допустим, что у нас есть словарь, содержащий данные о фильме.

```
filmDict={'originalTitle':'Forever',
          'creator':'Matthew Miller',
          'rate':8.3,
          'description':'A 200-year-old man worksin
            the New York City Morgue trying to find
            a key to unlock the curse of his immortality.',
          'years':[2014,2015]
        }

for key,value in filmDict.items():
    print("{}:{}".format(key,value))
```

Результат:

```
originallitle: Forever
creator:Matthew Miller
rate:8.3
description:A 200-year-old man worksin the New York City Morgue
trying to find a key to unlock the curse of his immorta lity.
years:[2014, 2015]
```

Рисунок 56

Однако нам нужно вывести эту информацию так, чтобы ключи словаря (характеристики фильма) были в алфавитном порядке.

Ранее мы уже работали со встроенной функцией `sorted()`, которая работает с итерируемыми объектами, значит, подойдет и для работы со словарем.

Однако при итерировании словаря мы получаем только ключи. Таким образом, для получения всех элементов (пар «ключ — значение») словаря нужно воспользоваться методом `items()`.

Метод `items()`, как мы уже знаем, выдаст нам список кортежей `[(key1, value1), (key2, value2), ...]`. Опять же элемент списка — это кортеж, комплексный тип данных (содержит несколько значений), поэтому нам нужно указать, по какому значению кортежа будет проводиться сортировка. Здесь нужно вспомнить, что функция `sorted()`, как и метод списков `sort()`, имеет параметр `key`. Этот параметр используется для задания функции, которая будет вызываться для каждого элемента списка перед выполнением сравнений.

Нам нужно, чтобы эта функция просто извлекала первый элемент кортежа (ключ словаря). Поэтому нет смысла создавать отдельную функцию, удобнее применить `lambda`-функцию.

```
sortedTuples = sorted(filmDict.items(), key=lambda x: x[0])
print(sortedTuples)
```

```
for element in sortedTuples:
    print(element)
```

Функция `sorted()` не изменяет переданный ей объект, а возвращает новый. На данном этапе мы получили список кортежей, отсортированных по первому элементу:

```
( 'creator', 'Matthew Miller')
( 'description', 'A 200-year-old man worksin the New York City Morgue
trying to find a key to unlock the curse of his immortality.')
( 'originalTitle', 'Forever')
( 'rate', 8.3)
( 'years', [2014, 2015])
```

Рисунок 57

Теперь соберем их обратно в новый словарь:

```
filmDictSorted=dict(sortedTuples)
for key,value in filmDictSorted.items():
    print("{}:{}".format(key,value))
```

Результат:

```
creator:Matthew Miller
description:A 200-year-old man worksin the New York City Morgue
trying to find a key to unlock the curse of his immorta lity.
originalTitle:Forever rate:8.3
years:[2014, 2015]
```

Рисунок 58

Есть и другой способ решения этой задачи. Мы можем создать отсортированный список ключей. И далее, используя отсортированную последовательность ключей в этом списке, выведем информацию из словаря.

```
keyList=list(filmDict.keys())
print(keyList)
sortedKeys=sorted(keyList)
print(sortedKeys)
```

```
for key in sortedKeys:
    print("{}:{}".format(key, filmDict[key]))
```

Результат:

```
['originalTitle', 'creator1', 'rate1', 'description1', 'years']
['creator', 'description', 'originalTitle', 'rate', 'years']
creator:Matthew Miller
description:A 200-year-old man worksin the New York City Morgue
trying to find a key to unlock the curse of his immorta lity.
originalTitle:Forever
rate:8.3
years:[2014, 2015]
```

Рисунок 59

Используя подход, реализованный в первом способе, мы сможем отсортировать словарь и по значениям.

Рассмотрим на примере словаря, содержащего информацию о книге.

```
bookDict={'author': 'Matthes',
          'title': 'Python',
          'reading age': '12',
          'language': 'English'
        }
```

```
print("Unsorted book info:")
for key,value in bookDict.items():
    print("{}:{}".format(key,value))
```

Единственное, что нам нужно исправить — это изменить в нашей **lambda**-функции индекс с нулевого на первый, т.к. значение элемента словаря — это второй элемент в кортеже. Таким образом мы изменим ключ сортировки.

```
print("Book info sorted by element value:")
bookDictSorted = dict(sorted(bookDict.items(),
                             key = lambda x: x[1]))
for key, value in bookDictSorted.items():
    print("{}:{}".format(key, value))
```

Результат:

```
Unsorted book info:
author:Matthes
title:Python
reading age:12
language:English
Book info sorted by element value:
reading age:12
language:English
author:Matthes
title:Python
```

Рисунок 60

Пример 3: сортировка списка словарей

Рассмотрим более сложную, но и более реальную ситуацию для сортировки: наш список пользовательской информации. Допустим, что данный набор нужно отсортировать по имени пользователя. В этой задаче нам подойдет решение, используемое в Примере 2: применение встроенной функции `sorted()` и задание ключа сортировки с помощью `lambda`-функции.

Единственный нюанс — это то, что обращаться к полю словаря нужно через имя ключа, а не с помощью индекса, как в Примере 2:

```
users = [
    {'name': 'Hanna', 'age': 10, 'login': 'user56'},
```

```
{'name': 'Mark', 'age': 15, 'login': 'usER111'},
{'name': 'Jane', 'age': 17, 'login': 'superGirl'},
{'name': 'Jack', 'age': 7, 'login': 'userJack'}
]

sortedUsersbyName=sorted(users, key=lambda x: x['name'])

print("Users list sorted by name:")
for user in sortedUsersbyName:
    for key,value in user.items():
        print("{}:{}".format(key,value))
```

Результат:

```
Users list sorted by name:
name:Hanna
age:10
login:user56
name:Jack
age:7
login:userJack
name:Jane
age:17
login:superGirl
name:Mark
age:15
login:usER111
```

Рисунок 61

Пример 4. Фильтры в словаре

Задача поиска элементов с нужными нам характеристиками одна из самых распространенных и часто ее называют фильтрацией.

Фильтр — это набор критериев (в простейшем случае мы имеем дело с одним критерием), которым должны соответствовать элементы некоторой коллекции.

Продолжим работу с нашим списком пользователей:

```
users = [
    {'name': 'Hanna', 'age': 10, 'login': 'user56'},
    {'name': 'Mark', 'age': 15, 'login': 'usER111'},
    {'name': 'Jane', 'age': 17, 'login': 'superGirl'},
    {'name': 'Jack', 'age': 7, 'login': 'userJack'}
]
```

При работе со списками мы уже использовали встроенную функцию `filter()`, которая извлекает те элементы из списка, для которых функция, указанная в качестве первого аргумента, возвращает значение `True`. Предположим, что нам нужно вывести тех пользователей, возраст которых больше 12. Очевидно, что фильтр будет: `users['age'] > 12`. Также удобно здесь использовать `lambda`-функцию, которая будет возвращать `True`, если фильтр на элементе «сработал».

Как мы помним, функция `filter()` возвращает объект,

```
<filter object at 0x10b8c5490>
```

Рисунок 62

поэтому нам будет также необходимо выполнить его преобразование в список:

```
users12=list(filter(lambda user: user['age'] >12, users))

for user in users12:
    for key,val in user.items():
        print("{}:{}".format(key,val))
```

Результат:

```
name:Mark  
age:15  
login:usER111  
name:Jane  
age:17  
login:superGirl
```

Рисунок 63



Урок 6

Кортежи, множества, словари

© STEP IT Academy, www.itstep.org

© Анна Егошина

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии с законодательством о свободном использовании произведения без согласия его автора (или другого лица, имеющего авторское право на данное произведение). Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования. Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника. Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством.