Списки (list)

In [6]:
```
a=5
id(a)
```

Out[6]:
```
3129067792816
```

In [7]:
```
id(a+2)
```

Out[7]:
```
3129067792880
```

In [9]:
```
a='str1'
```

In [10]:
```
a
```

Out[10]:
```
'str1'
```

In [11]:
```
type(a)
```

Out[11]:
```
str
```

In [23]:
```
type(True)
```

Out[23]:
```
bool
```

In [21]:
```
a=[1,2,3,4],['s','f','sf', True]
```

In [22]:
```
a
```

Out[22]:
```
([1, 2, 3, 4], ['s', 'f', 'sf', True])
```

Давайте представим, что при написании программы нам нужно работать, например, с большой базой данных студентов университета.

Если студентов несколько сотен, нет смысла создавать для каждого отдельную переменную - нам нужно научиться сохранять их всех в одной переменной. Начиная с этого занятия, мы будем изучать типы данных, которые позволяют это делать.

Начнем мы со **списков**. Если вы изучали другие языки программирования, то наверняка знакомы с аналогичным типом данных - массивами. Как и строки, списки - это последовательности, упорядоченные данные.

Давайте для начала попробуем создать список из 3 студентов.

In [24]:
```
students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']
print(students)
print(type(students))
```

```
['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']
<class 'list'>
```

Пустой список можно создать двумя способами - оператором [] и функцией list().

In [2]:
```
print([])
print(list())
```

```
[]
[]
```

In [3]:
```
a = list()
a
```

Out[3]:
```
[]
```

Список может содержать любые данные - например, числа.
Давайте создадим список оценок студента.

In [4]:
```
notes = [6, 5, 7, 5, 8]
print(notes)
```

```
[6, 5, 7, 5, 8]
```

Список может быть даже смешанным. Например, давайте сохраним в одном списке имя студента, его год рождения, средний балл, и логическую переменную, которая будет равна True, если студент учится на бюджете.

In [28]:
```
student1 = ['Ivan Ivanov', 1987, 7.5, True]
print(student1)
```

```
['Ivan Ivanov', 1987, 7.5, True]
```

Список может даже содержать другие списки.
Давайте создадим еще одного студента по аналогии со student1 и положим этих двух студентов в еще один список.

In [29]:
```
student2 = ['Maria Smirnova', 1991, 7.9, False]
students = [student2, student1]
print(students)
```

```
[['Ivan Ivanov', 1987, 7.5, True], ['Maria Smirnova', 1991, 7.9, False], 3]
```

Элементы списков нумеруются, начиная с 0. Мы можем получить доступ к элементу списка по его индексу.

In [34]:
```
students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']
print(students[0]) # первый элемент
print(students[1]) # второй элемент
print(students[-3]) # последний элемент
```

```
Ivan Ivanov
Tatiana Sidorova
Ivan Ivanov
```

Ксати, индексация работает и в строках. Там отдельными элементами являются символы.

In [35]:

```
x = 'слово'
print(x[0])
print(x[-1])
```

```
c
o
```

Мы можем узнать длину списка с помощью функции len() (работает и для строк).

In [36]:
```
print(len(students)) # количество элементов в списке students
print(len(x)) # количество символов в строке x
```

```
3
5
```

Но, в отличие от строк, список можно изменить.

In [41]:
```
students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']
students
id(students)
```

Out[41]:
```
3129149118848
```

In [42]:
```
students[0] = 'Ian Pile'
id(students)
```

Out[42]:
```
3129149118848
```

In [40]:
```
string = 'ololololol'
string[1] = 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\DARKSI~1\AppData\Local\Temp/ipykernel_8972/2655226930.py in <module>
      1 string = 'ololololol'
----> 2 string[1] = 'a'

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

Строки - неизменяемый тип данных, поэтому присвоение символа по индексу не сработает.

Еще одна операция, которая работает со всеми последовательностями - проверка на наличие элемента in и not in. Возвращает True или False.

In [43]:
```
### students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']
# print('Ivan Ivanov' in students)

n = [1,2,3,4]
x = 500
x not in n
# print('Petr' not in students[2])
# print(2 in students)
```

Out[43]:
```
True
```

In [45]:
```
a =c
```

```
//
go to:
    a and b
    b or c
//
```

Out[45]:
```
type
```

Способ расширить список - метод .append(), который добавляет аргумент в список в качестве последнего элемента.

In [44]:
```
lst = [0, 1, 2, 3, 4, 'cat', 'dog','cat']

lst.append(5)
print(lst)


lst += [6] # эквивалентное append выражение
print(lst)
```

```
[0, 1, 2, 3, 4, 'cat', 'dog', 'cat', 5]
[0, 1, 2, 3, 4, 'cat', 'dog', 'cat', 5, 6]
```

In [47]:
```
a= a + 1
a+=1
```

In [48]:
```
tmp a = a
tmp b = b
a=tmp b
b=tmp a
```

Out[48]:
```
2
```

In [49]:
```
a=1
b=2
a,b=b,a
```

Удалить элемент из списка можно с помощью метода .remove() (без возвращения удаленного элемента) или .pop (с возвращением удаленного элемента).

In [51]:
```
lst.remove('cat') # аргумент - объект, который ходим удалить
print(lst)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
C:\Users\DARKSI~1\AppData\Local\Temp/ipykernel_8972/1039161537.py in <module>
----> 1 lst.remove('cat') # аргумент - объект, который ходим удалить
      2 print(lst)

ValueError: list.remove(x): x not in list
```

In [55]:
```
x = lst.pop(0) # аргумент - индекс объекта. Результат операции можно сохранить в пер
print(lst)
print(x)
```

```
[3, 4, 'dog', 5, 6]
```

```
1
```

Поиском в списках занимается метод .index(), который вернет индекс объекта, переданного в качестве аргумента.

In [56]:
```
lst.append('dog')
```

In [57]:
```
lst
```

Out[57]:
```
[3, 4, 'dog', 5, 6, 'dog']
```

In [58]:
```
print(lst.index('dog'))
```

```
2
```

In [59]:
```
print(lst.count('dog'))
```

```
2
```

In [60]:
```
 # находим индекс объекта 'dog'
print(lst[lst.index('dog')]) # используем метод, возвращающий индекс, для обращения
```

```
dog
```

Если говорить еще о полезных методах, то это .count(), который подсчитывает количество элементов и .reverse(), который разворачивает список. Ниже еще отдельно поговорим о сортировке.

In [62]:
```
b = [1,2,3,4,5]
b.reverse()
b
```

Out[62]:
```
[5, 4, 3, 2, 1]
```

In [63]:
```
 # считаем количество 'dog' в списке
lst.reverse() # разворачиваем список. Осторожно - метод меняет список!
print(lst)
```

```
['dog', 6, 5, 'dog', 4, 3]
```

Все методы списков здесь

Сортировки

Отдельно следует рассказать про метод **sort()**. Метод производит сортировку списка. Задачи сортировки - очень распространены в программировании. В общем случае, они сводятся к выстроению элементов списка в заданном порядке. В Python есть встроенные методы для сортировки объектов для того, чтобы программист мог не усложнять себе задачу написанием алгоритма сортировки. Метод **list.sort()** - как раз, один из таких случаев.

In [31]:
```
test_list = [5, 8, 1, 4, 3, 7, 2]
print(test_list)  # Элементы списка расположены в хаотичном порядке
```

```
test_list.sort()
print(test_list)  # Теперь элементы списка теперь расположены по возрастанию
```

```
[5, 8, 1, 4, 3, 7, 2]
[1, 2, 3, 4, 5, 7, 8]
```

In [33]:
```
test_list.sort(reverse = True)
```

In [34]:
```
test_list
```

Out[34]:
```
[8, 7, 5, 4, 3, 2, 1]
```

Таким образом, метод **list.sort()** упорядочил элементы списка test_list Если нужно отсортировать элементы в обратном порядке, то можно использовать именнованный параметр reverse.

In [19]:
```
test_list.sort(reverse=True)  # параметр reverse указывает на то, что нужно отсортир
print(test_list)
```

```
[8, 7, 5, 4, 3, 2, 1]
```

Следует обратить внимание, что метод **list.sort()** изменяет сам список, на котором его вызвали. Таким образом, при каждом вызове метода "**sort()**", наш список "test_list" изменяется. Это может быть удобно, если нам не нужно держать в памяти исходный список. Однако, в противном случае, или же - в случае неизменяемого типа данных (например, кортежа или строки) - этот метод не сработает. В таком случае, на помощь приходит встроенная в питон функция **sorted()**

In [35]:
```
print(sorted(test_list))  # Сам список при сортировке не изменяется
```

```
[1, 2, 3, 4, 5, 7, 8]
```

In [36]:
```
test_list
```

Out[36]:
```
[8, 7, 5, 4, 3, 2, 1]
```

Так как sorted() функция, а не метод, то будет работать и с другими типами данных.

In [37]:
```
print(sorted('something'))
```

```
['e', 'g', 'h', 'i', 'm', 'n', 'o', 's', 't']
```

In [38]:
```
# print(sorted('something')) отсортирует буквы в строке, но выведет список
print(''.join(sorted('something'))) # с помощью метода join можно собрать отсортиров
```

```
eghimnost
```

У функции sorted(), как и у метода list.sort() есть параметр key, с помощью которого можно указать функцию, которая будет применена к каждому элементу последовательности при сортировке.

In [39]:
```
test_string = 'A string With upper AND lower cases'
```

In [40]:
```
print(sorted(test_string.split())) # заглавные буквы получили приоритет над строчным
```

```
['A', 'AND', 'With', 'cases', 'lower', 'string', 'upper']
```

In [20]:
```
print(sorted(test_string.split(), key=str.upper)) # все буквы были приведены к верхн
```

```
['A', 'AND', 'With', 'cases', 'lower', 'string', 'upper']
['A', 'AND', 'cases', 'lower', 'string', 'upper', 'With']
```

Кортежи (tuple)

Кортежи очень похожи на списки.

In [41]:
```
student = ('Ivan Ivanov', 2001, 7.5, True)
print(student)
print(type(student))
```

```
('Ivan Ivanov', 2001, 7.5, True)
<class 'tuple'>
```

Пустой кортеж можно создать с помощью оператора () либо функции tuple.

In [12]:
```
print(())
print(tuple())
```

```
()
()
```

In [43]:
```
x = tuple()
x
```

Out[43]:
```
()
```

In [44]:
```
student
```

Out[44]:
```
('Ivan Ivanov', 2001, 7.5, True)
```

Основное отличие кортежей от списков состоит в том, что кортежи нельзя изменять (да-да, прямо как строки).

In [45]:
```
student[1] = 2002
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-cf841071c98c> in <module>
----> 1 student[1] = 2002

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

Списки и кортежи могут быть вложены друг в друга.
Например, пусть в информации о студенте у нас будет храниться не его средний балл, а список всех его оценок.

In [46]:
```
student = ('Ivan Ivanov', 2001, [8, 7, 7, 9, 6], True)
print(student)
```

```
('Ivan Ivanov', 2001, [8, 7, 7, 9, 6], True)
```

In [47]:
```
student = ('Ivan Ivanov', 2001, (8, 7, 7, 9, 6), True)
print(student)
```

```
('Ivan Ivanov', 2001, (8, 7, 7, 9, 6), True)
```

Мы можем обратиться к элементу вложенного списка или кортежа с помощью двойной индексации.

In [ ]:
```

```

In [49]:
```
print(student[2][1]) # получили вторую оценку
```

```
7
```

Иногда бывает полезно обезопасить себя от изменений массивов данных и использовать кортежи, но чаще всего мы все-таки будем работать со списками.

Опасность работы с изменяемыми типами данных

В работе со списками есть важный момент, на который нужно обращать внимание.
Давайте рассмотрим такой код:

In [54]:
```
a = [1, 2, 3]
```

In [68]:
```
b = a
```

In [56]:
```
b[0] = 4
```

In [57]:
```
print(a)
print(b)
```

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

Почему так происходит?
Дело в том, что переменная a ссылается на место в памяти, где хранится список [1, 2, 3]. И когда мы пишем, что b = a, b начинает указывать на то же самое место. То есть образуется два имени для одного и того же кусочка данных. И после изменения этого кусочка через переменную b, значение переменной a тоже меняется!

Как это исправить? Нужно создать копию списка a! В этом нам поможет метод .copy()

In [58]:
```
a = [1, 2, 3]
b = a.copy() # теперь переменная b указывает на другой список, который хранится в др
b[0] = 4
print(a, b)
```

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

Копию можно создавать и с помощью пустого среза

In [62]:
```
a = [1, 2, 3]
b = a[:] # по умолчанию берется срез от первого элемента до последнего, то есть копи
b[0] = 4
print(a, b)
```

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

In [61]:
```
a = [1, 2, 3]
```

Out[61]:
```
[1, 2, 3]
```

Но не путайте изменение с присваиванием!

In [69]:
```
a = [1, 2, 3]
```

In [64]:
```
b = a
```

In [65]:
```
a = [4,5,6]
```

In [66]:
```
print(a)
```

```
[4, 5, 6]
```

In [67]:
```
print(b)
```

```
[1, 2, 3]
```

В примере выше переменная a была не изменена, а перезаписана, она начала указывать на другой список, хранящийся в другом месте памяти, поэтому переменная b осталась нетронута.

Конкатенация списков и кортежей

На списках и кортежах определен оператор +. По аналогии со строками, он будет склеивать две части выражения.

Но складывать можно только данные одного типа, список с кортежом склеить нельзя.

In [20]:
```
print([1, 2] + [3, 4])
print((1, 2) + (3, 4))
```

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

In [71]:
```
[1,2] *2
```

Out[71]:
```
[1, 2, 1, 2]
```

In [21]:
```
print((1, 2) + [3, 4]) # а так нельзя
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-dbea205bf98e> in <module>()
----> 1 print((1, 2) + [3, 4]) # а так нельзя

TypeError: can only concatenate tuple (not "list") to tuple
```

Но можно превратить список в кортеж, а потом сложить (или наоборот).

In [22]:
```
print(list((1, 2)) + [3, 4])
print((1, 2) + tuple([3, 4]))
```

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

In [74]:
```
x = tuple()
x
```

Out[74]:
```
()
```

Методы .split(), .join(), функция map(), вывод и ввод списков

При работе со списками довольно часто нам придется вводить их и выводить в отформатированном виде. Сейчас мы рассмотрим несколько методов, которые помогут сделать это в одну строку.

Метод .split()

Метод строки .split() получает на вход строку-разделитель и возвращает список строк, разбитый по этому разделителю.

По умолчанию метод разбивает строку по пробелу

In [78]:
```
print('Hello darkness my old friend     !#$@%$^#         bhfdsajklfgdsajk'.split())
```

```
['Hello', 'darkness', 'my', 'old', 'friend', '!#$@%$^#', 'bhfdsajklfgdsajk']
```

In [ ]:
```

```

In [79]:
```
print('Ночь. Улица. Фонарь. Аптека'.split('. '))
```

```
['Ночь', 'Улица', 'Фонарь', 'Аптека']
```

Метод .join()

Метод .join() ведет себя с точностью до наоборот - он склеивает массив в строку, вставляя между элементами строку-разделитель.

In [80]:
```
print('-'.join(['8', '800', '555', '35', '35']))
```

```
8-800-555-35-35
```

Функция map()

Функция map() берет функцию и последовательность и применяет эту функцию ко всем ее элементам (map() всегда будет ожидать от вас два аргумента).

Обратите внимание, чтобы увидеть результат работы этой функции надо дополнительно вручную преобразовать в список (или в кортеж, в зависимости от ваших целей).

In [83]:
```
print(map(bool, [9, 0, 8, -288, 998, 0])) # не совсем то, что надо
print(list(map(bool, [9, None, 8, -288, 998, 0]))) # а теперь работает, каждое число
```

```
<map object at 0x1090f5290>
[True, False, True, True, True, False]
```

Ввод и вывод списков

Теперь мы можем быстро вводить и выводить списки, содержащие разные данные через разные разделители.

In [85]:
```
print('-'.join(list(map(int,input().split()))))
```

```
1 2 3 4
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-14681131a48f> in <module>
----> 1 print('-'.join(list(map(int,input().split()))))

TypeError: sequence item 0: expected str instance, int found
```

In [24]:
```
phone = list(map(int, input().split())) # получаем строку, разбиваем по пробелу, сра
print(phone)
print('-'.join(list(map(str, phone)))) # преобразуем массив чисел в массив строк и с
```

```
[8, 913, 899, 99, 99]
8-913-899-99-99
```

Также, если нет задачи сохранить результат в переменную в виде строки, вместо join() можно использовать распаковку. Мы ставим оператор * перед списком, и, например, функция print() будет воспринимать его не как список, а как последовательность объектов.

In [51]:
```
print(phone, sep='-') # не работает, получили список
print(*phone, sep='-') # теперь сработало. По сути, распакованный список питон видит
print(phone[0], phone[1], phone[2], phone[3], phone[4], sep='-')
```

```
[8, 913, 899, 99, 99]
8-913-899-99-99
8-913-899-99-99
```