ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

(Python Beginning)

«Раз, два, три, четыре, пять» – произнес Винни Пух, отсчитывая патроны, заряжаемые в обойму дробовика…

Урок 5. (Lesson 5) Основы программирования

КОНТЕЙНЕРЫ ПОСЛЕДОВАТЕЛЬНОГО ДОСТУПА

В данном разделе предстоит познакомиться с последовательностями – ***наследниками*** контейнеров последовательного доступа из языка *C++*. Ведь, как ни крути, а *Python* реализован средствами *C/C++* и явился миру как альтернатива «***великого и могучего***», но очень требовательного к квалификации программиста, языка *C++*.

Списки

Поначалу вам будет не хватать привычных массивов, но это лишь на первых порах. Вам предстоит привыкнуть к высокоуровневым средствам языка *Python*, отчасти заимствованным из стандартной библиотеки *C++*, и, лишь затем, снова познакомиться со специальными классами, используемыми для представления массивов в *Python*. Эти специальные классы обычно используются для решения задач линейной алгебры и матричных вычислений. Но об этом чуть позже…

Список – это изменяемый тип, используемый для хранения ссылок на нумерованные наборы объектов. Обращение к элементам списка производится по индексу. Правила нумерации элементов списка совпадают с правилами нумерации элементов массивов в *C/C++* и *Java*. Но данные правила расширены возможность задавать ***отрицательное значение индекса*** элемента. В этом случае нумерация ведется с последнего элемента списка, имеющего номер равный -1 (минус один).

```
>>> people=["Johannes Kepler","Giordano Bruno","Galileo Galilei"]
>>> people[0]
'Johannes Kepler'
>>> people[1]
'Giordano Bruno'
>>> people[2]
'Galileo Galilei'
>>> people[-1]
'Galileo Galilei'
>>> people[-2]
'Giordano Bruno'
```

```
>>> people[-3]
'Johannes Kepler'
>>> people[4]
Traceback (most recent call last):
  File "<pyshell#83>", line 1, in <module>
    people[4]
IndexError: list index out of range
>>>
>>> people[-4]
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    people[-4]
IndexError: list index out of range
>>>
```

Как видно из выше приведенного примера, обращение к элементу списка представляет собой обыкновенное ***индексное выражение***.

Из предыдущих уроков вы уже знакомы с таким неизменяемым типом, как строка. Любые операции над строкой приводят к созданию нового экземпляра класса строка. Списки – изменяемые типы данных. Поэтому содержимое списков можно изменять. Как-никак, а это одна из стратегий экономии памяти в *Python*.

Благодаря тому, что в *Python* вместо обычных переменных используются ссылки, списки могут хранить ссылки на объекты совершенно разных типов. Списки отчасти похожи на шаблон *std::vector* из *C++*, они тоже реализуют безопасный в плане проверки значение индекса доступ к элементам, но есть и существенные отличия.

Конструктор по умолчанию *создает пустой список, не содержащий в себе никаких ссылок. Списки, как и обычные программные переменные, содержат в себе исключительно ссылки на объекты.*

Создать список очень просто, для этого нужно определить выражение, где будет фигурировать объект-список, либо вызвать конструктор класса список – *list*().

```
>>> my_arr_1 = list()  # вызвать конструктор и создать пустой список
>>> len(my_arr_1)  # получить значение длины списка
0  # список содержит ноль ссылок на объекты, он пуст
>>> my_arr_2 = []  # неявный вызов конструктора для создания списка
>>> len(my_arr_2)
0
>>>
```

Можно создать список, указав его содержимое в квадратных скобках. По сравнению с C/C++ появилась возможность создавать списки, содержащие ссылки на объекты разных классов, эта возможность обеспечивается отказом от

использования традиционных программных переменных и использование исключительно переменных-ссылок.

К слову говоря, в C/C++ тоже можно создать гетерогенный список, то есть список, содержащий элементы разного типа. Для этого можно воспользоваться либо механизмом полиморфизма, то есть использовать указатели на базовый класс, либо создать список, содержащий указатели на пустой тип «*void**».

```
>>> my_arr = [None]  # создать не пустой список, содержащий 1 элемент
>>> my_arr = [0,1,2,3,4,5,6,7,8,9]
>>> my_arr = [1, 1.2, 'A', None] # список ссылок на разные объекты
>>> my_arr_1 = [1,2,3,"This is a list",[1,2],['A','b']]
>>> my_arr_2 = list("ABCDEFG")  # преобразование строки в список
>>> my_arr_2
['A', 'B', 'C', 'D', 'E', 'F', 'G']
>>> len(my_arr_2)  # длина списка – количество ссылок в списке
7
>>>
```

За счет того, что ссылка может ссылаться на объект любого типа и любой сложности, на структуру списка практически не налагается ограничений по степени вложенности и сложности…

Напоминаю, значение *None* класса *NoneType* можно рассматривать в качестве эквивалента «***пустого указателя***» «*void**»/*null* из *C/C++*. В *Python* он используется для манипуляций с памятью и программирования методов.

```
>>> [1,2,3,4]  # одномерный список
[1, 2, 3, 4]
>>> [1,[1, 2]]; [[1],[[2],[3]], 4, None]  # и этот список одномерный
[1, [1, 2]]
[[1], [[2], [3]], 4, None]
>>>
```

Обратите внимание на размерность создаваемых объектов: при вызове метода *len()* возвращает результат, соответствующий ***верхнему уровню*** созданной списочной структуре – количество ссылок на ***верхнем уровне***.

```
# создать список, содержащий ссылку на один список, содержащий ...
>>> [[[[[1],[2,3]]]]]
[[[[[1], [2, 3]]]]]
>>> len([[[[[1],[2,3]]]]])  # список содержит одну ссылку
1
>>> len([[1,2,3,4],[5,6,7,8,[1,2]]])
2
>>>
```

Основные положения по организации списков в Python:

- *По механизму обращения к элементам, списки в Python **внешне похожи** на массивы в ANSI C/C++, но, на самом деле, по своей организации они соответсвуют контейнеру `std::list` STL C++. Те, кто утверждают, что списки в Python используются вместо массивом, лукавят…*
- *Язык программирования `Python` поддерживает только одномерные списки, элементами которого могут быть ссылки на любые типы данных, в том числе и на другие списки. Примером абстракции «многомерого списка» может быть **дерево** – **граф, имеющий корневой узел**, но класс `list` не является однозначным примером реализации данной абстракции, хотя с помощью списка можно организовать и структуру дерева.*
- *Исходя из механизма работы ссылок в `Python`, можно смело утверждать, что даже многомерный на наш взгляд список на самом деле является одномерным, как и массивы в языке `ANSI C`.*
- *Как и в C/C++, для работы со списком Python использует структуру дескриптора списка, содержащую как минимум указатель на первый узел списка. Детали реализации данной структуры сокрыты от «**обычного**» программиста, но **системный программист** должен учитывать этот факт при разработке гибридных программных систем…*

```
>>> my_list = [1,[2,3],[[4,5],[6,7,8]]] # якобы многомерный список...
>>> # my_list - это ссылка на объект - дескриптор списка
>>> for i in my_list:  # перебор элементов списка в пошаговом цикле
     print(i)          # вывод содержимого элементов списка

1
[2, 3]
[[4, 5], [6, 7, 8]]
>>> # Список содержит три элемента, два из которых также являются
>>> # списками. Элементы списка выведены построчно.
```

Для списка реализованы три вида конструкторов: ***конструктор по умолчанию***, ***конструктор копирования*** и ***конструктор преобразования***. Конструктор преобразования позволяет создать список из объекта другого класса, который поддерживает ***итерационность*** (`iterables`), то есть для него определены объекты-итераторы – вспоминаем итераторы для последовательных контейнеров в `STL C++`. Поддержка ***итерационности*** обеспечивается за счет наличия в классе двух методов: `__iter__()` и `__next__()`. Более подробно это будет рассмотрено в разделе, посвященному ООП.

```
>>> arr = (12, 23)  # создать список (кортеж)
>>> arr1 = list(arr)  # вызвать конструктор преобразования
>>> arr1
[12, 23]  # содержимое списка совпадает с содержимым кортежа
>>> arr2 = list(arr1)  # вызов конструктора копирования
```

```
>>> arr2
[12, 23]
list(((1,2,3),(4,5,6), [7,8,9]))  # параметр – смешанный список
>>> len(list(((1,2,3),(4,5,6), [7,8,9])))  # его длина – 3 элемента
3
>>>
```

Небольшое замечание, использованная в примерах конструкция – список параметров, заключенных в круглые скобки, определяет для интерпретатора создание экземпляра класса ***кортеж*** (`tuple`), который тоже относится к категории последовательностей. Но, в отличие от списков, кортежи относятся к категории неизменяемых типов данных.

Операции над списками

- **Встроенные функции и методы для работы со списками**

`in` – проверка на наличие элемента в списке. Метод возвращает значение `True` (истина) в случае, если в списке имеется элемент, содержащий заданное значение, в противном случае возвращает значение `False` (ложь).

```
>>> arr = [0,1,2,3,4,5,6,7,8,9]
>>> 5 in arr
True
>>> x = 12
>>> x in arr
False
>>>
>>> arr = [1, [1], [1,2]]
>>> x = [1]
>>> x in arr
True
>>>
>>> "ll" in "Hello"  # строка рассматривается как список символов
True
>>>
```

`not in` – проверка на отсутствие элемента, содержащее указанное значение. Метод возвращает значение `True` (истина) в случае, если в списке имеется элемент, содержащий заданное значение, в противном случае возвращает значение `False` (ложь).

```
>>> "ll" not in "Hello"  # строка рассматривается как список символов
False
>>>
```

Операция на проверку вхождения и отсутствия указанного значения позволяют автоматизировать процесс анализа больших массивов данных.

Конечно, в языке *C++* имеются аналогичные функторы и методы для работы с контейнерами, но радует тот факт, что такие высокоуровневые средства работы с последовательностями вводятся с самого начал обучения языку *Python*.

```
>>> arr = [12, 34, 56, 78, 90]
>>> x = [12, 34]
>>> x not in arr
True  # список arr не содержит элемент - список, содержащий...
>>>
```

+ – сложение списков, метод «плюс». Фактически, это перегруженная операция, позволяющая выполнять манипуляции над списками, она позволяет вставить в конец существующего списка содержимое другого списка.

```
>>> arr1 = [1,2,3]; arr2 = [4,5,6,7,8,9]
>>> arr1 = arr1 + arr2
>>> arr1
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> arr2 in arr1
False  # все содержимое arr2 стало частью arr1
>>>
>>> arr1.append(arr2)  # введем arr2 как новый элемент в arr1
>>> arr2 in arr1       # теперь он есть в arr1
True
>>>
```

* – «умножение» содержимого списка. Данный метод позволяет увеличить содержимое последовательности на указанное число раз. Значения элементов последовательности при этом повторяются, то есть, вставка производится блоками.

```
>>> Arr = [1,2]; Arr = Arr * 10; print(Arr)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>>
>>> A = [1,2]; B = [3,4,5]; C = ['A','B','C','D']
>>> D = A * 4 + B * 2 + C; print(D)
[1, 2, 1, 2, 1, 2, 1, 2, 3, 4, 5, 3, 4, 5, 'A', 'B', 'C', 'D']
>>>
>>> Arr = [1]; Arr *= 20
>>> Arr
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
>>> Arr = []  # создать пустой список
>>> Arr *= 10 # умножить содержимое пустого списка на 10
>>> Arr  # количество элементов осталось равным нулю!
[]
>>>
>>> Arr = [None]  # не пустой список, содержащий ссылку на ничто
```

```
>>> Arr *= 10; print(Arr)
[None, None, None, None, None, None, None, None, None, None]
>>>
```

Перегруженная операция – метод «умножение» упрощает процедуру создания последовательности, содержащей нужное количество элементов, значения которых не имеет значения, не важны. Можно предположить, что она была введена по образу перегруженного конструктора *std::vector(size_type count)*, позволяющего указывать количество элементов контейнера.

append(obj) – добавить элемент в конец списка. Фактически, является аналогом метода *std::vector.puch_back()*.

```
>>> Arr = []  # создать пустой список
>>> Arr.append(1)      # Вставить в конец списка новый элемент
>>> Arr.append(2)      # Вставить в конец списка новый элемент
>>> # Вставить в конец списка новый элемент-последовательность
>>> Arr.append([1,2])
>>> Arr                # Отобразить содержимое последовательности
[1, 2, [1, 2]]
>>> Arr[0]
1
>>> Arr[1]
2
>>> Arr[2]
[1, 2]
>>> for i in Arr: print(i)

1
2
[1, 2]
>>>
```

clear() – очищает содержимое списка.

```
>>> arr = [0,1,2,3,4,5,6,7,8,9]; arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # в списке было 10 элементов
>>> arr.clear(); arr
[]  # в результате выполнения операции clear() получен пустой список
>>>
```

copy() – создание поверхностной копии списка.

```
>>> arr1 = [1,2,3,4,5]
>>> arr2 = arr1.copy()
>>> arr1 is arr2  # проверка - arr1 и arr2 один объект?
False             # Ложь. Это разные объекты в памяти
>>>
```

count(obj) – возвращает значение – объект класса *int*, равный значению количества вхождений искомого объекта в списке.

```
>>> arr = list("Hello")
>>> arr.count('H'), arr.count('l'), arr.count('R')
(1, 2, 0)
>>>
```

```
#!/usr/bin/python3
# filename: test_list_01.py
# BEGIN ---------------------------
arr = ['q','w','e','r','t','y']
# count element 'y'
count = arr.count('y')
# print count
print("The count of 'y' is:", count)
# count element 'q'
count = arr.count('q')
# print count
print("The count of 'q' is:", count)
# END -----------------------------
```

```
#!/usr/bin/python3
# filename: test_list_02.py
# BEGIN ---------------------------
arr = ['q','w',('q','w'),('q','w'),[1,2]]
# count element 'y'
count = arr.count('y')
# print count
print("The count of 'y' is:", count)
# count element ('q','w')
count = arr.count(('q','w'))
# print count
print("The count of ('q','w') is:", count)
# count element [1,2]
count = arr.count([1,2])
# print count
print("The count of [1,2] is:", count)
# END -----------------------------
```

extend(iter_obj) – добавляет объект-последовательность в конец списка. Отличается от метода *append()* тем, что вставляет не просто сам объект, а его элементы в конец списка.

```
>>> arr1 = [0,1,2,3,4,5,6]; arr2 = [7,8,9]
>>> arr = arr1; arr.append(arr2); arr
[0, 1, 2, 3, 4, 5, 6, [7, 8, 9]]
>>> # маленький глюк, связанный с экономией памяти
>>> # если две ссылки указывают на одинаковое значения, то они
>>> # связываются с одним объектом
```

```
>>> arr = arr1; arr.extend(arr2); arr
[0, 1, 2, 3, 4, 5, 6, [7, 8, 9], 7, 8, 9]  # некорректный результат
>>> arr1 = [0,1,2,3,4,5,6]; arr2 = [7,8,9]
>>> arr = arr1; arr.extend(arr2); arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  # теперь всё правильно
>>>
>>> arr = list("Hello, "); arr; arr.extend("Dummy"); arr
['H', 'e', 'l', 'l', 'o', ',', ' ']
['H', 'e', 'l', 'l', 'o', ',', ' ', 'D', 'u', 'm', 'm', 'y']
>>>
```

Немного опережая события, рассмотрим примеры добавления в конец списка, то есть его расширение, элементов таких контейнеров, как кортежи и множества.

```
>>> language = ["Franch", "English", "German"]
>>> language_tuple = ("Spanish", "Portuguese")
>>> language_set = {"Chines","Japanese"}
>>> language_list = ["Russian"]
>>> language.extend(language_tuple); language
['Franch', 'English', 'German', 'Spanish', 'Portuguese']
>>> language.extend(language_list); language
['Franch', 'English', 'German', 'Spanish', 'Portuguese', 'Russian']
>>> language.extend(language_set); language
['Franch', 'English', 'German', 'Spanish', 'Portuguese', 'Russian',
'Japanese', 'Chines']
>>>
```

Несколько неожиданной может показаться результат расширения множества объектом, который не являющимся контейнером последовательного доступа, но поддерживает ***итерационность***, словарём – *dict*. Метод *extend()* извлекает из словаря только ***значения*** элементов словаря, ***ключи*** словаря игнорируются.

```
>>> language = ["Spanish"]
>>> language_dict = {'fr': "Franch", 'en': "English", 'ge': "German"}
>>> language.extend(language_dict); language
['Spanish', 'fr', 'ge', 'en']
>>>
```

В списке можно хранить указатели на функции, но расширять список просто ссылкой на функцию нельзя: её предварительно нужно поместить в список.

```
>>> arr = [1,2,3]    # создали непустой список
>>> def method(X):   # объявили функцию
     print(X)         # определили операторы тела функции


>>> arr.extend(method)  # попытались сделать неправильно и получили
Traceback (most recent call last):
```

```
  File "<pyshell#36>", line 1, in <module>
    arr = [1,2,3]; arr.extend(method)
TypeError: 'function' object is not iterable
>>>
>>> func = [method]  # создали список, содержащий ссылку на функцию
>>> arr.extend(func) # расширили список arr содержимым списка func
>>> arr  # вывели содержимое списка arr
[1, 2, 3, <function method at 0x02469348>]
>>>
```

index(obj[,start_pos, end_pos]) - возвращает значение – объект класса *int*, равный значению индекса позиции искомого элемента в списке. Если в списке нет элементов, содержащих указанное значение (ссылающихся на него), то данный метод выбрасывает исключение *ValueError*.

```
>>> names = ["Bob", "Sue", "Tom", "Andry", "John"]
>>> name_position = names.index("Sue")
>>> name_position
1
>>> names.index("Vova")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    names.index("Vova")
ValueError: 'Vova' is not in list
>>>
```

Рассмотрим задачу перевода целочисленного 16-ричного значения в 10-ричный формат. Если опустить действия, необходимые для сепарации префикса системы счисления, то получится такой код.

```
>>> # создадим список, содержащий символы для 16-ричной записи
>>> arr1 = [0,1,2,3,4,5,6,7,8,9,'A','B','C','D','E','F']
>>> numb = 'FF'      # обрабатываемая 16-ричная запись числа
>>> numb_int = 0     # переменная для представления результата
>>> mult = 1         # множитель в пизиционной системе счисления
>>> for i in numb:  # цикл перебора символов 16-ричной записи числа
     digit = arr1.index(i)      # вычичляем значение разряда
     numb_int += digit * mult   # формируем результат
     mult *= 16                 # корректируем значение множителя


>>> numb_int  # результат перевода из 16-ричного формата в 10-чный
255
>>>
```

insert(index, obj) – вставляет в список объект по указанной позиции. Благодаря тому, что список реализован по принципу ***самоссылочной структуры***, данная операция не связана с большими затратами по перемещению данных в памяти программы. Мы еще не рассматривали операцию «***среза***» для списка, но,

забегая вперед, нам необходимо рассмотреть следующие эквивалентные записи: `s.insetr(i,x)` и `s[i:i] = [x]`. Кроме того, список представляет очень интересный механизм проверки правильности значения индекса для вставки нового элемента. Сами сделайте вывод о том, как производится пересчет значения исходя из следующих примеров.

```
>>> arr = [1,2,3,4,5]
>>> arr.insert(10, 6); arr
[1, 2, 3, 4, 5, 6]
>>> arr.insert(8,7); arr
[1, 2, 3, 4, 5, 6, 7]
>>> arr.insert(10,8); arr
[1, 2, 3, 4, 5, 6, 7, 8]
>>>
```

Примеры, в которых используются допустимые значения индексов:

```
>>> arr = [0,1,2,3,4,5,6,7,8,9]; arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr.insert(1,'A'); arr
[0, 'A', 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr.insert(-1,'B'); arr
[0, 'A', 1, 2, 3, 4, 5, 6, 7, 8, 'B', 9]
>>> arr.insert(-0,'C'); arr
['C', 0, 'A', 1, 2, 3, 4, 5, 6, 7, 8, 'B', 9]
>>> arr.insert(4,'D'); arr
['C', 0, 'A', 1, 'D', 2, 3, 4, 5, 6, 7, 8, 'B', 9]
>>>
```

Конечно, можно вставлять элементы и в пустой список. Формат значения позиции вставки при этом может быть не только 10-чным, важно, что он должен быть целым числом – внутренняя проверка и приведение типа в методе не производятся.

```
>>> arr = []
>>> arr.insert(1000000, 1); arr
[1]
>>> arr.insert(0x1,2); arr
[1, 2]
>>> arr.insert(2.5, 3)  # ошибка: значение индекса типа float
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    arr.insert(2.5, 3)
TypeError: integer argument expected, got float
>>>
```

*Обратите внимание на факт активного использования в Python **механизмов исключения**, как стратегии обработки программных ошибок. Это*

требует от программиста аккуратной и грамотной организации кода проекта! Оставил исключение не обработанным – «подложил свинью» в свой проект…

pop([index]) – возвращает объект – элемент списка по указанной позиции. Объект при этом удаляется/извлекается из списка. В случае попытки извлечения данных элемента из пустого списка, генерируется исключения *IndexError*. Аналогичным образом обрабатывается ошибка передачи неверного значения индекса извлекаемого элемента.

```
>>> arr = []  # создать пустой список
>>> arr.pop() # попытаться извлечь элемент из пустого списка
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    arr.pop()
IndexError: pop from empty list
>>> arr = [1,2,3]
>>> arr.pop()  # по умолчанию извлекается последний элемент списка
3
>>> arr.pop(3) # максимальное значение индекса равно 1
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    arr.pop(3)
IndexError: pop index out of range
>>>
>>> arr = [0,1,2,3,4,5,6,7,8,9]; arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr.pop(1); arr
1
[0, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr.pop(-1); arr
9
[0, 2, 3, 4, 5, 6, 7, 8]
>>>
```

Одномерная структура списков, прошу не путать их за ***внешнюю похожесть*** с массивами, хорошо проявляется именно на примере операции извлечения элемента – метод *pop()* принимает ***только одно значение индекса*** извлекаемого элемента.

```
>>> arr = [[1,2,3],[4,5,6],[7,8,9]]; arr
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> arr.pop()  # извлечь из списка последний элемент
[7, 8, 9]      # извлеченный элемент является списком
>>> arr        # отобразить содержимое списка
[[1, 2, 3], [4, 5, 6]]  # в списке остались два элемента - списка
>>>
```

remove(obj) – удаляет из списка первый встреченный экземпляр объекта, переданного в качестве аргумента.

```
>>> arr = [i for i in range(10)]  # использован генератор списка
>>> arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr.remove(2)
>>> arr
[0, 1, 3, 4, 5, 6, 7, 8, 9]
>>> arr = [1,2]; arr *= 10; arr
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> arr.remove(1); arr
[2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> arr.remove(1); arr
[2, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 2]
>>>
```

В случае, если значения, подлежащего удалению из списка, в обрабатываемом списке нет возбуждается исключение *ValueError*.

```
>>> arr = list("Hello World"); arr
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
>>> arr.remove('Y')  # попытка удалить несуществующий элемент
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    arr.remove('Y')
ValueError: list.remove(x): x not in list
>>>
```

reverse() – изменяет порядок следования элементов в списке на противоположный.

```
>>> arr = [i for i in range(10)]; arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr.reverse(); arr
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>>
```

sort([key=None, reverse=False]) – сортирует список в порядке возрастания. Сортировка возможна лишь в том случае, если для элементов списка определена операция сравнения **"<"** (больше). Дополнительным ограничением для использования данного метода является «однородность» списка – его элемента должны быть экземплярами одного класса. Необязательный параметр *key* определяет ссылку на функцию, которая выполняет сортировку, а параметр *reverse* – порядок сортировки. Фактически, *sort()* – это шаблонная функция с параметрами по умолчанию.

```
#!/usr/bin/python3
# -*- coding: cp1251 -*-
```

```
# filename: test_sort_01.py
# BEGIN ---------------------------------------------------------
arr = ['q','ww','eee','rrrr','ttttt','yyyyyy']
# функция для сравнения в алфавитном порядке
def sortByAlphabet(inputStr):
    return inputStr[0] # сравниваются только первые символы
# функция для сравнения длин строк
def sortByLenght(inputStr):
    return len(inputStr) # возвращается длина строки
# выводим содержимое списка
print("Исходный список:",arr)
# сортировка списка в алфавитном порядке.
# анализируются исключительно значения только первых символов
arr.sort(key=sortByAlphabet)
# выводим содержимое списка
print("Отсортированный в алфавитном порядке список:\n",arr)
# сортировка списка в порядке длины элементов-строк.
arr.sort(key=sortByLenght)
# выводим содержимое списка
print("Список,   отсортированный   в   порядке   длин   элементов-
строк:\n",arr)
# сортировка списка в обратном порядке длины элементов-строк.
arr.sort(key=sortByLenght, reverse=True)
# выводим содержимое списка
print("Список,  отсортированный  в  обратном  порядке  длин  элементов-
строк:\n",arr)
# использование метода sort() со значениями параметров по умолчанию
# использование метода sort() со значениями параметров по умолчанию
arr.sort()
# выводим содержимое списка
print("Список, отсортированный методом sort() с пустым"
      "списком параметров:\n",arr)
# END -----------------------------------------------------------
```

И еще один пример использования метода *sort()* для сортировки списков. В качестве критерия сравнения используется метод *len()*, возвращающий количество элементов в списке.

```
>>> [arr1,arr2] = [[10,9,81,43,12,111],[5,6,7]]
>>> sorted([arr1,arr2], key=lambda x: len(x), reverse=True)
[[10, 9, 81, 43, 12, 111], [5, 6, 7]]
>>> sorted([arr1,arr2], key=lambda x: len(x), reverse=False)
[[5, 6, 7], [10, 9, 81, 43, 12, 111]]
>>>
```

Надеюсь, что вы ещё не забыли, что методы *list.sort()* и *sorted()* фактически равноценны.

```
>>> arr = [[10,11,12,13,14],[5,6,7,8,9]]
>>> arr.sort(key=lambda x: my_sum(x), reverse=False)
```

```
>>> arr
[[5, 6, 7, 8, 9], [10, 11, 12, 13, 14]]
>>> arr.sort(key=lambda x: my_sum(x), reverse=True)
>>> arr
[[10, 11, 12, 13, 14], [5, 6, 7, 8, 9]]
>>> arr.sort(key=my_sum, reverse=False); arr
[[5, 6, 7, 8, 9], [10, 11, 12, 13, 14]]
>>> arr.sort(key=my_sum, reverse=True); arr
[[10, 11, 12, 13, 14], [5, 6, 7, 8, 9]]
>>>
```

Использованные в примере *lambda*-вызовы используются в *Python* для реализации «***отложенных вызовов***» и формирования ссылок на функции.

- **Создание копии списка**

Ввиду наличия в Python особой стратегии работы с памятью, связанной с указателями на «***одинаковые***» объекты, операция создания копии списка выделяется в отдельную группу операций над списками.

Рассмотрим пример реализации группового присваивания.

```
>>> list_1 = list_2 = [1,2,3,4,5]
>>> print(list_1, ':', list_2)
[1, 2, 3, 4, 5] : [1, 2, 3, 4, 5]
>>> list_1 is list_2  # Проверка: list_1 это list_2?
True  # Да, ссылки list_1 и list_2 указывают на один объект в памяти
>>> list_1.append(6)  # Изменим объект в памяти по ссылке list_1
>>> # Произведено изменение объекта в памяти
>>> print(list_1, ':', list_2)
[1, 2, 3, 4, 5, 6] : [1, 2, 3, 4, 5, 6]
>>> # видно, что ссылки указывают на один объект в памяти
```

При реализации списочного присваивания такой коллизии не наблюдается.

```
>>> list_1, list_2 = [1,2,3,4,5],[1,2,3,4,5]
>>> list_1 is list_2 #  Проверка: list_1 это list_2?
False  # Нет, ссылки list_1 и list_2 указывают на разные объекты
>>> list_1.append(6); list_2.insert(0,0)  # изменим списки
>>> print(list_1, ':', list_2)
[1, 2, 3, 4, 5, 6] : [0, 1, 2, 3, 4, 5]
>>>
```

Надеюсь, вы еще не забыли примеры выражения списочного присваивания из первого занятия? Они могут выполняться не только над ссылками на списки, но и на любые программные переменные.

```
>>> a,b,c,d = 1, 2.3, 'A', [15, 7]
>>> print(a,b,c,d, sep=',')
1,2.3,A,[15, 7]
>>> # процедура обмена значениями двух переменных
```

```
>>> A = 2; B = 3
>>> A, B = B, A  # выполняется через списочное присваивание
>>> print(A,B, sep=',')
3,2
>>>
```

Мы только что рассмотрели метод *copy()*, позволяющий создать поверхностную копию списка. Данные метод является методом класса *list*. Но в языке *Python* существует ещё и модуль *copy*, в котором определена функция *deepcopy()*, позволяющая создавать поверхностные копии объектов.

```
>>> import copy
>>> arr = [i for i in range(1,21,2)]
>>> copy_arr = copy.deepcopy(arr)  # создание поверхностной копии
>>> # Проверка: arr1 и copy_arr указывают на один объект в памяти
>>> arr is copy_arr
False  # Нет, ссылки указывают на разные объекты в памяти
>>> # Хотя содержимое объектов при этом совпадает
>>> print(arr, copy_arr, sep='\n')
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>>
```

Поэтому постарайтесь для себя четко определить, когда создается полная копия объекта, а когда – поверхностная.

И это, кстати, ещё не все «***цветочки***», связанные со списочным присваиванием в языке Python. Рассмотрим следующие примеры.

```
>>> a1, a2, a3, *a4 = [1,2,3,4]; a1, a2, a3, a4
(1, 2, 3, [4])
>>> a1, a2, a3, *a4 = [1,2,3]; a1, a2, a3, a4
(1, 2, 3, [])
>>> a1, a2, *a3 = [1,2,3,4,5]; a1, a2, a3
(1, 2, [3, 4, 5])
>>> a1, *a2, a3 = [1,2,3,4,5]; a1, a2, a3
(1, [2, 3, 4], 5)
>>> *a1, = [1,2,3,4,5]; a1
[1, 2, 3, 4, 5]
>>> *a1 = [1,2,3,4,5]
SyntaxError: starred assignment target must be in a list or tuple
>>>
```

Дело в том, что при списочном присваивании создается вспомогательный объект – ***кортеж*** (*tuple*), а предшествующий элементу символ «звездочка» - '*' является признаком того, что данный элемент является списком. Именно поэтому последнее выражение привело к выбросу объекта исключения.

- **Удаление списка**

Удаление списка выполняется также как и удаление обычных программных объектов – используется операция *del*.

```
>>> Arr = [i**3 for i in range(20) if i%2==0]; Arr
[0, 8, 64, 216, 512, 1000, 1728, 2744, 4096, 5832]
>>> len(Arr)
10
>>> del Arr  # удалить и объект, и ссылку на него
>>> Arr  # попытка обращения к несуществующей ссылке
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    Arr
NameError: name 'Arr' is not defined
>>>
```

- **Извлечение среза списка (*slices*)**

Если рассматривать список как множество элементов, то операция ***извлечения среза*** позволяет выделить из него подмножество на основании правил, налагаемых на значения индексов элементов исходного списка-множества. В результате выполнения данной операции может быть получен как один элемент, так и совокупность элементов – новый список.

Срезы позволяют выполнить следующие действия:

1. Получить копию списка.
2. Получит новый список, содержащий первые/последние N элементов списка.
3. Получить N элементов с позиции M.
4. Получить новый список, содержащий каждый N элемент.
5. Получить новый список, содержащий элементы исходного списка но в обратном порядке.

Синтаксис операции извлечения среза:

`[<Начало=0>:<Конец=len(список)>:<Шаг=1>].`

Параметры *<Начало>* и *<Конец>* соответствуют не номерам элементов, а номерам позиций между элементами. При этом в список вводятся два «несуществующих» элемента, для того, чтобы номера позиций тоже начинались с нуля. Таким образом, промежуток перед первым элементом имеет номер 0, а перед вторым – 1. Рассмотрим следующие варианты использования операции ***извлечения среза*** списка:

1. *Имя_списка*`[`*start*`:`*end*`]` – срез из (*end-start*) элементов: *Имя_списка*`[`*start*`]`, *Имя_списка*`[`*start*`+1],… ,` *Имя_списка*`[end-1]`
2. *Имя_списка*`[`*start*`:`*end*`:-1]` – срез из (*end - start*) элементов, полученный в обратном порядке уменьшения значения индексов: *Имя_списка*`[`*start*`],` *Имя_списка*`[`*start*`-1],`*Имя_списка*`[start+2·step],…,` *Имя_списка*`[end+1]`

3. *Имя_списка*[*start*:*end*:*step*] – срез из элементов, формируемый с шагом равным значению *step*: *Имя_списка*[*start*], *Имя_списка*[*start*+step], *Имя_списка*[*start* + 2·*step*],…

Примеры извлечения среза:

```
>>> arr = [value for value in range(10)]; arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr[:]  # создать – извлечь копию списка
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr[len(arr):0:-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> arr[-3:]
[7, 8, 9]
>>> arr[1:-1]  # отбрасываем первый и последний элементы списка
[1, 2, 3, 4, 5, 6, 7, 8]
>>> arr[::2]  # извлечь парные элементы, начиная с первого
[0, 2, 4, 6, 8]
>>> arr[1::2]  # извлечь парные элементы, начиная со второго
[1, 3, 5, 7, 9]
>>>
```

Операция извлечения среза используется совместно с операцией *del* для удаления элементов из списка (список относится к категории изменяемых типов).

```
>>> arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del arr[2:-2]  # удалить часть списка, определенную срезом
>>> arr
[0, 1, 8, 9]
>>> arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del arr[::2]  # удалить каждый второй элемент
>>> arr
[1, 3, 5, 7, 9]
>>> arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del arr[:]  # удаление содержимого списка
>>> arr
[]  # получили пустой список
>>>
>>> arr = [[0,1,2,3,4],[5,6,7,8,9]]
>>> arr[::2]
[[0, 1, 2, 3, 4]]
>>>
>>> arr = [[0,1,2,3,4],[5,6,7,8,9]]
>>> arr[2:-2]
[]  # получили пустой список
>>> arr[1:-1]
[]  # срез – пустой список
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
>>> arr = [[1,2,3],[4,5,6],[7,8,9]]; arr
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
>>> arr[1:-1]
[[4, 5, 6]]
>>>
```

Результат последних выражений объясняется тем фактом, что список для самого *Python* имеет именно одномерную структуру.

Кроме того, операция извлечения среза позволяет изменить содержимое элементов списка.

```
>>> arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr[::2]=['A','B','C','D','A']; arr
['A', 1, 'B', 3, 'C', 5, 'D', 7, 'A', 9]
>>>
```

В результате выполнения операции извлечения среза вы получаете доступ к последовательности – части списка. При выполнении операции присваивания операнд, стоящий в правой части присваивания, должен ***совпадать по размерности*** с извлекаемым срезом, в противном случае будет выброшен объект исключения *ValueError*. Данный параметр должен быть объектом – последовательностью (списком, кортежем, строкой, множеством).

```
>>> arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr[::2]=['A','B','C','D']
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    arr[::2]=['A','B','C','D']; arr
ValueError: attempt to assign sequence of size 4 to extended slice of
size 5
>>>
>>> arr[::2] = list("ABCDE"); arr
['A', 1, 'B', 3, 'C', 5, 'D', 7, 'E', 9]
>>> arr[::] = [(i+1)**3 for i in range(10)]; arr
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
>>>
>>> arr[::] = "qwertyuiop"
>>> arr
['q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p']
>>>
>>> arr[::] = {0,1,2,3,4,5,6,7,8,9}; arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

С размерностью объекта, стоящего в правой части выражения присваивания срезу есть некоторые нюансы…

```
>>> arr = [1,2,3]; arr
[1, 2, 3]
>>> arr[::] = ["Hello my friend"]; arr
['Hello my friend']
```

```
>>> len(arr)
1  # длина списка была равна 3, а стала – 1
>>> arr[::] = list("Hello my friend"); arr
['H', 'e', 'l', 'l', 'o', ' ', 'm', 'y', ' ', 'f', 'r', 'i', 'e',
'n', 'd']
>>> len(arr)
15  # теперь длина списка стала равна 15
>>>
>>> arr = []  # пустой список
>>> arr[:] = "Hello"
>>> arr
['H', 'e', 'l', 'l', 'o']  # в списке 5 элементов
>>>
```

- **Использование цикла *for* для обработки элементов списков**

Чуть ранее мы уже использовали цикл *for* для обработки элементов простых, не вложенных списков. Сейчас же рассмотрим возможности использования данного оператора для обработки вложенных списков.

```
>>> arr = [(0,1,2,3,4),(5,6,7,8,9)]
>>> for a1 in arr:
     for a2 in a1:
          print(a2, end = ' ')


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

Сразу замечу, что при разработке пособия использовался *Python* версии 3.5, поэтому ***устаревшие примеры*** из учебника [3] приводят к выбросу объекта исключения.

```
>>> arr = [[0,1,2,3,4],[5,6,7,8,9]]
>>> for a1, a2 in arr:
     print(a1, a2)


Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    for a1, a2 in arr:
ValueError: too many values to unpack (expected 2)
>>> arr = [(0,1,2,3,4),(5,6,7,8,9)]
>>> for a1, a2 in arr:
     print(a1, a2)


Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
```

```
    for a1, a2 in arr:
ValueError: too many values to unpack (expected 2)
```

Такая организация вложенных циклов требует от программиста досконального знания конструкции обрабатываемых структур. И это еще раз доказывает, что ***списки – это не массивы****! Массивы в* `Python` *реализованы в модуле* `numpy`*!*

Ошибка, содержащаяся в вышеприведенных примерах, основана на том , что на каждой итерации цикла из списка извлекаются «составные объекты – последовательности», которые можно рассматривать и как «последовательность», и как «группу» скалярных объектов, но доступ к ним возможен только как к «последовательности».

```
>>> arr = [[0,1,2,3],[4,5,6,7]]
>>> for a1 in arr:
     print(a1, type(a1))


[0, 1, 2, 3] <class 'list'>
[4, 5, 6, 7] <class 'list'>
>>>
>>> for ai in arr:
     for aij in ai:
          print(aij, type(aij), end = ' ')  # вывод в одной строке
     print()  # переход на новую строку


0 <class 'int'> 1 <class 'int'> 2 <class 'int'> 3 <class 'int'>
4 <class 'int'> 5 <class 'int'> 6 <class 'int'> 7 <class 'int'>
>>>
```

- **Генераторы списков**

В языке Python введены специальные выражение – генераторы списков. Они являются частью интерфейса, определенного для последовательностей (`Sequence`). Подробную информацию о генераторах вы можете почерпнуть из ряда источников [4], для полного пересказа которых в настоящем руководстве нет ни возможности, ни острой необходимости.

```
>>> arr = [i for i in range(10)]; arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr = [[] for i in range(10)]; arr
[[], [], [], [], [], [], [], [], [], []]
>>> arr = [i**3 for i in range(10)]; arr
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> arr = [i**4 for i in range(10) if i%2 == 0]; arr
[0, 16, 256, 1296, 4096]
```

```
>>>
```

Как видно из примера, выражения генераторы строятся на базе операторов цикла *for* и могут иметь сложную структуру.

```
>>> arr = [i for i in range(0,10,2)]; arr
[0, 2, 4, 6, 8]
>>> arr = [i**2 for i in arr]; arr
[0, 4, 16, 36, 64]
>>> arr = [i+10 for i in arr if i%10==6]; arr
[26, 46]
>>>
>>> arr1 = [1,2,3,4,5]; arr2 = [16,17,18,19,20,21,22]
>>> print([(x,y) for x in arr1 for y in arr2 \
       if x * y > 25 and x + y < 25])
[(2, 16), (2, 17), (2, 18), (2, 19), (2, 20), (2, 21), (2, 22), (3,
16), (3, 17), (3, 18), (3, 19), (3, 20), (3, 21), (4, 16), (4, 17),
(4, 18), (4, 19), (4, 20), (5, 16), (5, 17), (5, 18), (5, 19)]
>>>  # создан список кортежей
>>> print([[x,y] for x in arr1 for y in arr2 \
       if x * y > 25 and x + y < 25])
[[2, 16], [2, 17], [2, 18], [2, 19], [2, 20], [2, 21], [2, 22], [3,
16], [3, 17], [3, 18], [3, 19], [3, 20], [3, 21], [4, 16], [4, 17],
[4, 18], [4, 19], [4, 20], [5, 16], [5, 17], [5, 18], [5, 19]]
>>>  # создан список списков
```

- **Функции *map()*, *zip()*, *filter()* и *reduce()***

Встроенная функция *map()* позволяет применить указанную функцию к каждому элементу последовательности, причем этот объект может быть не только списком. По своей сути, данная функция является объектом-функцией, то есть функтором. Синтаксис использования функции имеет следующий вид:

map(<*Функция*>, <*Последовательность*_1>, [, … , <*Последовательность*_N>])

Функция *map()* возвращает объект типа *map*, поддерживающий итерацию и если его необходимо привести к типу список, то опять же придется воспользоваться конструктором преобразования – *list()*.

Как понимаете, в качестве параметра <*Функция*> должна фигурировать ссылка на функцию, которой в качестве параметра будет передаваться элементы последовательности. Данная функция ***должна возвращать новое значение*** для обрабатываемой последовательности.

```
>>> def func(elem):
     temp_list = [elem]
     return temp_list

>>> Arr = [i**3 for i in range(1,11)]; Arr
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

```
>>> type(map(func,Arr))
<class 'map'>  # метод map() вернул объект класса map
>>> Arr  # исходный объект остался без изменений
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
>>> newArr = list(map(func, Arr)); newArr
[[1], [8], [27], [64], [125], [216], [343], [512], [729], [1000]]
>>>
```

Функция *map()* может обрабатывать сразу несколько последовательностей, причем эти последовательности могут иметь различное количество элементов – разную длину. Функция *map()* организует цикл (итерационный перебор элементов контейнер) вызова функции, определенной параметром <*Функция*>, по количеству элементов последовательности, имеющей наименьшую длину.

```
>>> def func(el_1,el_2,el_3):
     temp = [el_1, el_2, el_3]
     return temp

>>> Arr1 = [i for i in range(10)]
>>> Arr2 = [i**2 for i in range(11)]
>>> Arr3 = [i**3 for i in range(12)]
>>> Arr = list(map(func, Arr1, Arr2, Arr3)); Arr
[[0, 0, 0], [1, 1, 1], [2, 4, 8], [3, 9, 27], [4, 16, 64], [5, 25,
125], [6, 36, 216], [7, 49, 343], [8, 64, 512], [9, 81, 729]]
>>>
```

Не пытайтесь использовать функцию *map()* для таких сложных задач, как умножение матриц: ***списки – это не массивы***. Для представления массивов в *Python* имеется модуль *numpy*, содержащий класс для представления числовых массивов и все необходимые методы, определяющие операции над матрицами.

Функция *zip()* предназначена для группировки нескольких списков, она реализована как объект-функция (***функтор***), которая в качестве параметра принимает ссылку на список последовательностей. На каждой итерации обращения к последовательности она возвращает кортеж (*tuple*), содержащий элементы последовательностей, переданные ей в качестве параметров, расположенные на одинаковой позиции. Результатом работы функции является объект класса *zip*. Данный класс также поддерживает итерации. Если нужно преобразовать его в список, то необходимо использовать конструктор преобразования.

Синтаксис вызова функции *zip()*:

```
zip(<Последовательность_1>, [, … , <Последовательность_N>])
```

```
>>> arr = [[0,1,2],[3,4,5],[6,7,8],[9,10,11]]
>>> zip(arr)
<zip object at 0x00421D78>
>>> arr1 = [0,1,2,3]; arr2 = [4,5,6,7]; arr3 = [8,9,10,11]
>>> zip(arr1, arr2, arr3)
```

```
<zip object at 0x02603990>
>>> list(zip(arr1, arr2, arr3))
[(0, 4, 8), (1, 5, 9), (2, 6, 10), (3, 7, 11)]
>>>
>>> arr1=[0,1,2,3]; arr2=[4,5,6,7,8]; arr3=[9,10,11,12,13,14]
>>> zip(arr1, arr2, arr3)
<zip object at 0x02603990>
>>> list(zip(arr1, arr2, arr3))
[(0, 4, 9), (1, 5, 10), (2, 6, 11), (3, 7, 12)]
>>>
```

Обратите внимание на последний пример, размеры параметров вызова функции `zip()` различаются. Функция `zip()` организует итерации по самому «***короткому***» параметру, в котором меньше всего элементов.

Еще один пример использования функции zip(), заимствованный из учебника Прохоренок Н.А. [3] – функция `zip()` используется вместо функции `map()` для обработки элементов списков.

```
>>> arr1=[1,2,3,4,5]
>>> arr2=[10,20,30,40,50]
>>> arr3=[100,200,300,400,500]
>>> arr = [a1 + a2 + a3 for (a1,a2,a3) in zip(arr1,arr2,arr3)]
>>> arr
[111, 222, 333, 444, 555]
>>>
```

А вот, кстати, и пример для нахождения суммы элементов последовательностей с использованием функции `map()`[3](пример был несколько изменен ☺☺☺).

```
>>> def func(a1, a2, a3):
    """Суммирование элементов трех разных списков"""
    return a1 + a2 + a3

>>> arr1 = [i for i in range(1,6)]
>>> arr2 = [i for i in range(10, 51,10)]
>>> arr3 = [i for i in range(100, 501, 100)]
>>> print(arr1, arr2,arr3, sep='\n')
[1, 2, 3, 4, 5]
[10, 20, 30, 40, 50]
[100, 200, 300, 400, 500]
>>> print(list(map(func, arr1, arr2, arr3)))
[111, 222, 333, 444, 555]
>>> # Создадим вспомогательный контейнер-список, содержащий
>>> # несколько списков
>>> arr = [arr1, arr2, arr3]
>>> print(list(map(func, *arr)))
[111, 222, 333, 444, 555]
>>> # Теперь создадим вспомогательный контейнер-кортеж,
```

```
>>> # также содержащий несколько списков
>>> arr = (arr1, arr2, arr3)
>>> print(list(map(func, *arr)))
[111, 222, 333, 444, 555]
>>> # получен одинаковый результат
```

Для функции *zip()* работает стандартное ***правило передачи параметра, являющегося контейнером***, поддерживающим ***итерационность*** – ему должна предшествовать операция '*' «звёздочка».

```
>>> arr1 = [1,2,3]; arr2 = [4,5,6]; arr3 = [7,8,9]
>>> arr = [arr1, arr2, arr3]
>>> print(zip(arr))
<zip object at 0x023F5918>
>>> print(list(zip(arr)))
[([1, 2, 3],), ([4, 5, 6],), ([7, 8, 9],)]
>>>
>>> print(list(zip(*arr)))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>>
```

Результаты вызовов `zip(*arr)` и `zip(arr)` существенно различаются – получены совершенно разные объекты. Последний пример можно упростить.

```
>>> print(list(zip(*[arr1, arr2, arr3])))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>>
```

Нам удалось сэкономить на создании временного вспомогательного объекта – оптимизация по памяти, и на вызове конструктора – оптимизация производительности. Даже в *Python*`е требуется грамотная и аккуратная организация кода, позволяющая не писать «***индусский код***» ☺ ☺ ☺.

С функцией *zip()* нам ещё придется не раз встречаться, так как она активно используется при генерации таких объектов, как словари, а словари мы будем использовать при работе с базами данных и *shelve*-хранилищами. Вот небольшой пример с прицелом на будущее.

```
>>> tags = ['name', 'age', 'pay']    # список ключей
>>> bob = ['Bob Smith', 42, 50000]  # информационный список
>>> dict(zip(tags, bob))  # создаем контейнер ассоциативного доступа
{'name': 'Bob Smith', 'age': 42, 'pay': 50000}
>>>
>>> list(zip(tags, bob))  # просто группируем два списка
[('name', 'Bob Smith'), ('age', 42), ('pay', 50000)]
>>>
```

Функция *filter()*, при помощи переданной в качестве первого параметра функции, выполняет фильтрацию элементов перееденного объекта. Формат вызова функции *filter()*:

filter(<Функция>, <Последовательность>)

Функция *filter()* возвращает список элементов, для которых заданная функция возвращает True.

```
>>> arr = [17, 16, 178, 23, 40, 13, 11]
>>> def any_func(x):
     if x % 2 == 0: return True
     else: return False


>>> new_arr = filter(any_func, arr); new_arr
<filter object at 0x023EFCB0>
>>> # Результат функции filter() – объект класса filter, а не list
>>> new_arr = list(new_arr)  # используем конструктор преобразования
>>> new_arr
[16, 178, 40]
>>>
>>> numbers = [i for i in range(-9,9,2)]
>>> positive_numbers = list(filter(lambda x: x > 0, numbers))
>>> print(positive_numbers)
[1, 3, 5, 7]
>>>
```

Как видно из синтаксиса вызова, функция *filter()* применяется в отношении не только списков, но и всех контейнеров, поддерживающих итерационность, то есть является обобщенной шаблонной функцией (вспоминаем *STL C++* – «***матерь***» языка Python).

Если в качестве первого параметра вызова функции *filter()* указано значение *None*, то элементы последовательности будут проверять на соответствие логическому значению *True*. То есть, при фильтрации будут отсеяны элементы, значение которых при приведении к логическому типу будет равно *False*.

```
>>> str1 = str()   # создали пустую строку
>>> bool(str1)     # преобразовали пустую строку к логическому типу
False
>>>
```

Помимо пустых строк ещё ряд значений имеют логический эквивалент, равный *False*. Рассмотрим следующие примеры использования функции *filter()*:

```
>>> arr = [1, 2, 0, 0.12, None, '', [],[1,2]]
>>> list(filter(None, arr))
[1, 2, 0.12, [1, 2]]
```

```
>>>
>>> Arr = [2, 4, 5, 6, 7, 8, 12, 13, 17]
>>> subset_of_Arr = [12, 13, 17]
>>> result = list(filter(lambda x: x not in subset_of_Arr, Arr))
>>> print(result)
[2, 4, 5, 6, 7, 8]
>>>
```

Для последнего примера есть альтернативная реализация, основанная на использовании класса множеств – *set*, коллекции уникальных объектов, уникальных в плане того, что в коллекцию они входят в единственном числе, то есть, не повторяются.

```
>>> set_Arr = set(Arr); set_subset_of_Arr = set(subset_of_Arr)
>>> set_result = list(set_Arr.difference(set_subset_of_Arr))
>>> set_result  # результат - список
[2, 4, 5, 6, 7, 8]
>>>
>>> set_result = tuple(set_Arr.difference(set_subset_of_Arr))
>>> set_result  # результат - кортеж
(2, 4, 5, 6, 7, 8)
>>>
```

Функция *reduce()*, определённая в модуле *functools*, предназначена для выполнения вычислений на списке. Формат вызова этой функции:

reduce(<*Функция*>, <*Последовательность*> [, <*Начальное_значение*>])

Данная функция «сворачивает» список, применяя в качестве аргумента функцию по очереди к последовательности пар элементов исходного списка. Для того чтобы это стало более понятно, рассмотрим задачу нахождения произведения всех элементов целочисленного списка.

```
>>> numbers = [i for i in range(1,21) if i % 2 == 0 or i % 3 == 0]
>>> mult_numbers = 1
>>> for numb in numbers: mult_numbers *= numb

>>> print(mult_numbers)
1504935936000
>>>
```

А теперь решение этой же задачи с использованием функции *reduse()*.

```
>>> from functools import reduce
>>> numbers = [i for i in range(1,21) if i % 2 == 0 or i % 3 == 0]
>>> # 1-й вариант с использованием lambda-выражения
>>> mult_numbers = reduce((lambda x, y: x * y), numbers, 1);
mult_numbers
1504935936000
>>> # 2-й вариант
```

```
>>> def func_mult(x, y):  # объявили функцию
     return x * y

>>> mult_numbers = reduce(func_mult, numbers, 1); mult_numbers
1504935936000
>>>
```

Задачи

Для закрепления материала решите несколько простых задач. Рассмотрите варианты решения, основанные на использовании встроенных методов и основанные на использовании процедуры перебора элементов списков. Программа должна обеспечивать процедуру ручного ввода данных списка непосредственно пользователем программы. По возможности, напишите вариант программы с графическим интерфейсом ☺☺☺. Программный интерфейс должен содержать одно поле для ввода значения или последовательности значений

1. Создайте программу для нахождения суммы элементов списка.
2. Найти сумму элементов введенного набора, которые удовлетворяют некоторому условию. Например, лежат в заданном интервале или наоборот, не входят в указанный интервал.
3. Подсчет количества элементов списка, обладающих некоторым свойством. Например, для списка сотрудников необходимо определить количество оных, чей непрерывный стаж работы в кампании превышает 10 лет…
4. Определить порядковый номер некоторого значения в списке.
5. Определить максимальное значение в списке.
6. Определить количество элементов, содержащих максимальное значение в списке.
7. Определить второе по величине значение в списке.
8. Определить номер второго по величине значения в списке.
9. Определить, является ли набор значений, помещенных в список, пилообразным.
10. Определить количество локальных максимумов в списке.
11. Заполнить список из n элементов случайными целыми числами из интервала [a;b].
12. Заполнить список степенями числа 2 (от 2^1 до 2^N). Функцию возведения в степень и операцию умножения не использовать…

Литература и источники в Интернет

1. Numeric and Mathematical Modules [электронный ресурс]: https://docs.python.org/3/library/numeric.html (дата обращения – 07.08.2017).
2. Дистанционная подготовка по информатике [электронный ресурс]: https://informatics.mccme.ru/ (дата обращения – 17.08.2017).
3. Прохоренок Н.А. Python 3 и PyQt 5. Разработка приложений [Текст] / Н.А. Прохоренок, В.А. Дронов. – СПб.: БХВ-Петербург, 2016. – 832.: ил.
4. Python: коллекции, часть 4/4: Все о выражениях-генераторах, генераторах списков, множеств и словарей [электронный ресурс]: https://habrahabr.ru/post/320288/ (дата обращения – 2.09.2017).
5. Doug Hellmann. The Python 3 Standard Library by Example [Текст / Электронный ресурс] Addison-Wesley ISBN-13: 978-0-13-429105-5, ISBN-10:0-13-429105-0 https://www.amazon.com/Python-Standard-Library-Example-Developers/dp/0134291050 [Дата обращения - 2.09.2017].
6. Doug Hellmann. Python 3 Module of the Week [электронный ресурс]: https://pymotw.com/3/ (дата обращения – 2.09.2017)