Массивы

До этого у вас были задания, в которых надо было считать несколько значений с клавиатуры. Для каждого такого значения вы создавали переменные, но что если считать нужно много значений (тысячи)? Создавать для каждого значения отдельную переменную контрпродуктивно. Для хранения большого количества однотипных данных удобно использовать массивы.

Массивы (списки) - структура данных для хранения значений, в которой мы можем обратиться к любому элементу по индексу (позиции).

Обычно массивы хранят значения определенного типа, но Python динамический язык, поэтому в нём в массивах могут одновременно находиться и числа, и строки, и логические значения.

Создание массива и обращение к элементам:

Массив создается с помощью квадратных скобок []. Также квадратные скобки являются **оператором** обращения, то есть с помощью них мы можем получать элементы из массива.

Чтобы создать массив:

1. Можно создать полностью пустой массив

In [9]:
```
A = []
print(A)
```

```
[]
```

В пустом массиве нет никаких элементов, поэтому мы не можем обратиться к ячейке 3, 1 или любой другой. Их просто нет. Но иногда нам надо записывать элементы в конкретные ячейки, поэтому

2. Можно заполнить элементы массива одинаковыми значениями

In [8]:
```
A = [0]*5
print(A)
```

```
[0, 0, 0, 0, 0]
```

Например можно использовать нейтральные нули. В таком массиве мы сразу можем изменять элемент под нужным индексом.

3. Можно сразу указать в квадратных скобках значения, которые должны быть в массиве

In [10]:
```
A = [1,5,3,4,7]
print(A)
```

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

К каждому элементу в массиве можно обратиться:

In [5]:
```
print(A[0])
print(A[1])
print(A[2])
print(A[3])
print(A[4])
```

```
1
5
3
4
7
```

С элементами можно взаимодействовать также как с переменными:

- Мы можем совершать с ними операции

In [6]:
```
print(A[2]+A[4])
```

```
10
```

- Мы можем заменять значения элементов

In [11]:
```
A[2] = A[3]*2 + A[2]
print(A[2])
```

```
11
```

Вывод массива

Мы можем просто передать массив в print, чтобы вывести его

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

```
[1, 5, 11, 4, 7]
```

Но в этом случае он выведет сразу все элементы с квадратными скобочками и запятыми, что не всегда нам нужно.

Для работы с элементами всегда удобно использовать циклы:

Для определения размера массива можно использвать функцию `len()`. В цикле будем обращаться к элементам массива по индексу и выводить их последовательно.

In [15]:
```
N = len(A)
for i in range(N):
    print(A[i],end=" ")
```

```
1 5 11 4 7
```

Но циклы у нас работают нет только с range, а с любыми объектами, которые можно перебирать, а элементы в массиве как раз можно перебирать, поэтому мы можем в цикле перебирать не позиции элементов, а сами элементы:

In [17]:
```
for i in A:
    print(i, end=" ")
```

```
1 5 11 4 7
```

Или можно поставить звездочку перед массивом в print

In [18]:
```
print(*A)
```

```
1 5 11 4 7
```

Обработка массивов

Заполнение случайными числами

Поскольку массивы могут хранить много значений вписывать их все с клавиатуры неудобно и занимает много времени, поэтому для проверки алгоритмов обработки удобно использовать случайные числа.

Для работы со случайными числами есть библиотека random, а в ней метод `randint`, который может генерировать случайные целочисленные значения. `randint` принимает два значения - левая и правая граница диапазона в котором нужно сгенерировать случайное число.

In [23]:
```
from random import randint
A = []
for i in range(5):
    A.append(randint(1,6))
print(A)
```

```
[2, 5, 4, 5, 5]
```

Поскольку мы создали массив A пустым, то для добавления туда элемента мы используем `append()`, который добавлет указанный элемент в конец массива.

Другой способ - работа с массив в котором уже есть ячейки. В этом случае мы не добавляем элемент в конец, а перезаписываем значение, которое находится в массиве на новое, случайное.

In [24]:
```
from random import randint
A = 5*[0]
for i in range(5):
  A[i] = randint(1,6)
print(A)
```

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

Подсчет и сумма элементов

Циклы являются основой для всех алгоритмов обработки элементов в массиве.

Для этого мы также будем часто использовать **внешнюю пременную** - это переменная которая объявляется вне цикла, но используется внутри цикла для хранения значений.
Пример, подсчет элементов, которые >3:

In [25]:
```
A = [1, 2, 3, 4, 5, 6, 7]
k = 0
for i in range(7):
  if A[i] > 3: k += 1
print(k)
```

4

Переменная k - внешняя. Изначально она равна 0.
В цикле проверяется последовательно каждый элемент и, если он >3, то значение k увеличивается на 1.

Аналогично мы можем проверять любые условия, то есть подсчитывать любые элементы, которые соответвуеют условиям:
`if A[i] % 3 == 0: k += 1` - подсчёт элементов, которые делятся на 3 без остатка
`if(A[i] % 10 == 6 and A[i] % 3 == 0): k += 1` - подсчёт элеметов, которые делятся на 3 без остатка и оканчиваются на 6
`if A[i]%7 == 0: k += 1` - подсчёт элементов, которые делятся на 7 без остатка

И аналогично мы можем подсчитывать не количество элементов а сумму:

In [28]:
```
A = [1, 2, 3, 4, 5, 6, 7]
s = 0
for i in range(7):
  if A[i] > 3: s += A[i]
print(s)
```

22

В этом случае мы добавляем к внешеней переменной не 1, а сам элемент.

Поиск минимума и максимума

Еще два основных алгоритма для работы с массивами - поиск минимума и максимума среди элементов. Она также работают с помощью внешней переменной.

Если раньше мы с помощью внешней переменной считали, то теперь мы будем её сравнивать с элементами в массиве.

Максимум

In [30]:
```
A = [1, 21, 3, 46, 53, 117, 6]
m = 0
for i in range(7):
  if A[i] > m: m = A[i]
print(m)
```

```
117
```

Если раньше мы присваивали внешней переменной значение 0 как пустое значение (то есть значащее, что мы пока ничего не посчитатали), то в случае поиска максимума мы устанавливаем в m то значение, которое будет *меньше* всех элементов, которые есть в массиве.

Причём мы не берем это число из головы, мы вибираем его только исходя из условий. В примерах будем считать, что у нас числа в массивах могут быть от 1 до 998 включительно.

В цикле мы будем сравнивать каждый элемент массива с тем значенем, которое хранится в внешней переменной. Если элемент оказался больше, то мы заменяем значение внешней переменной на значение этого элемента. Поскольку элементы проверяются последовательно, то в внешней переменной каждый раз будет находиться самое большое значение которое было найдено, а в конце в ней будет самое больше значение, которое было в массиве, то есть максимум.

Минимум

In [1]:
```
A = [1, 21, 3, 46, 53, 117, 6]
m = 999
for i in range(7):
  if A[i] < m: m = A[i]
print(m)
```

```
1
```

Для поиска минимума алгоритм точно такой же, но заменяетс язнак сравнения с больше на меньше и внешней переменной в начлае присваивается значение, которое точно больше всех элементов, которые есть в массиве.

Но что делать, если диапазон значений массива неизвестен? Если внимательно посмотреть на алгоритмы, то заметим поскольку внешняя переменная m меньше всех других в начале алгоритма, то уже на первой итерации она будет гарантировано заменена значением первого элемента, потому что все элементы массива гарантировано больше, чем то значение, которое было вписано в m, до начала обработки массива. Аналогично с минимумом, где значение переменной тоже гарантированно заменяется уже на первом элементе.

Поэтому можно изначально присвоить, что внешняя перменная равна первому элементу.

In [ ]:
```
A = [... ...]
N = len(A) # длина массива
```

```
m = A[0]
for i in range(N):
  if A[i] < m: m = A[i]
print(m)
```