Программа Python.
2021-2022 год.

Задача: Пользователь при работе с программой сам вводит числа (данные) в программу. Необходимо вывести на экран список полученных данных.

Как выглядела бы программа создания такого списка данных для нас раньше, до знакомства со списками?

Ответ: Вариант, когда пользователь вводит по одному числу в одной строке.

```
a = int(input())
b = int(input())
c = int(input())


print(a,b,c)
```

Здесь в этом примере мы видим ввод трех значений. А как быть с программой, где необходимо ввести миллион значений?

Писать миллион строк: **... = int(input())**?

Да у нас даже букв в алфавите не хватит. И время написания такой программы займет очень много времени.

Конечно же, на ум приходит знание о циклах. У нас уже в инструментах разработчика есть два цикла. Цикл for и цикл while.

Вспоминаем теорию: Когда мы используем тот или иной цикл (for или while)?

Цикл for мы используем, когда мы знаем заранее сколько необходимо повторять раз наш код внутри цикла.

Цикл while мы используем, когда мы заранее не знаем (или очень трудно и долго высчитывать), сколько будет повторяться цикл, пока условие в цикле не достигнет нужного значения.

Пусть в нашей задаче пользователь вводит только пять чисел (не будем просить его вводить миллион строк).

Тогда, для решения задачи ввода чисел используем цикл for:

1 Способ:

```
a = [] # заводим пустой список (массив)

for i in range(5):
    a.append(int(input())) # 5 раз добавим значение

print(a) # напечатаем список на экран
```

В этой программе мы используем для списка стандартный метод append(), который будет добавлять в конец списка новый элемент. В нашем случае добавляем число, поэтому используем совместно с функцией input() функцию перевода строки в число - int().

2 Способ:

```
a = [0] * 5

for i in range(5):
    a[i] = int(input())

print(a)
```

В этом способе нам пришлось прежде создать список с нужным количеством элементов. И только потом, с помощью строки: **a[i] = int(input())** мы его заполняем новыми значениями. Но элементы в этом примере уже должны быть созданы заранее! Это нужно понимать. Иначе, если в списке ничего нет в начале: a = [], то тогда в списке нет первого элемента a[0]!

Пример:

```
a = []
print(a[0])
```

И будет ошибка в программе - выход за пределы списка (массива), если не создать список - заглушку: **a = [0] * 5**

Потому что:

Пример:

```
a = [] * 5
print(a)
```

```
[]
```

Сколько бы мы не умножали пустой список, он не станет больше. И не появятся в нем элементы, даже не появится элемент с нулевым индексом.

Второй способ не совсем корректно будет работать, если к примеру, нужно заполнить не все ячейки списка **цифрами.**

Нельзя с помощью функции **input()** ничего не передать в функцию **int()**! Будет ошибка.

К тому же, если цифра **0** в нашей программе должна участвовать, а не быть символом того, что ячейка пустая!

Сейчас мы говорим о программе, где список должен наполняться именно **числами**!

Ноль это не пустота, ноль - это конкретная цифра, которая может участвовать в дальнейших вычислениях в программе.

Поэтому, если следовать настоящим правилам Python, нужно вместо **0**, в случае, когда необходимо, чтобы ячейка списка была пустая, использовать для нее значение **None.** Это принятое в Python стандартное обозначение ничего.

None - значит ничего - пустое значение.

Перепишем нашу программу по новым правилам, при этом даже выведем на печать предварительно наш пустой список.

```
a = [None] * 5

print(a)

for i in range(5):
    a[i] = int(input())

print(a)
```

Но все равно остается условие, что мы заполняем обязательно все элементы списка! Потому что нельзя в нашей программе пропустить ввод числа при помощи функции input().

Другое дело, если наполнять список мы будем строками (символами):

```
a = [None] * 5
print(a)

for i in range(5):
    a[i] = input()

print(a)
```

```
[None, None, None, None, None]
1
3


5
```

['1', '3', '', '', '5'] - видим, что пустые строки записались в качестве элемента списка.

Есть еще и третий способ синтаксиса для решения предыдущей задачи:

```
a = [int(input()) for i in range(5)]

print(a)
```

В этом способе можно цикл поместить прямо в квадратные скобки списка.

Такой способ создания списка называют - **генератором списка.**

Можно создать список (сгенерировать список) и без использования ручного ввода чисел (значений), без input().

```
a = [i for i in range(5)]

print(a)
```

Результат:

```
[0, 1, 2, 3, 4]
```

или

```
a = [9 for i in range(5)]
print(a)
```

```
[9, 9, 9, 9, 9]
```

или

```
a = ['www' for i in range(5)]
print(a)
```

```
['www', 'www', 'www', 'www', 'www']
```

Генератор списка - это самый быстрый инструмент создания списков из представленных способов.

Потому, что генератор списков - генерирует только списки с помощью цикла for, другие задачи он не выполняет.

Тогда как цикл for в первых двух способах представляет универсальную конструкцию и при такой конструкции цикл for может не только генерировать списки, но и решать другие задачи в программе.

Правило: Чем более узкая задача, которую решает программа, тем быстрее она работает (можно сказать - тем лучше инструмент, если он решает только одну конкретную задачу).

Но есть еще одно замечание: Мы определились, что третий способ (генератор списка) это самый быстрый способ создать список. А какой способ быстрее или медленнее из первых двух?

..............?

Самый медленный способ создания списка - это способ с использованием функции append(). Почему?

Еще раз посмотрим на программу:

```
a = []
for i in range(5):
    a.append(int(input()))

print(a)
```

Идея заключается в том, что Python должен записать на физическую память компьютера в ячейку памяти наш будущий список. Но дело в том, что в нашей программе мы создаем пустой список и Python, не представляет пока, насколько большим будет список, и может записать его в сектор памяти, которого не хватит для хранения растущего списка, тогда Python будет вынужден искать уже больше места в памяти компьютера, переносить туда список, и продолжать сохранять новые появляющиеся элементы списка. Так же может далее произойти нехватка места. И процедура поиска нового места памяти для переноса списка повторится вновь! Это конечно отнимает время у самой программы, которая генерирует список.

Итак, мы рассмотрели, как вводить числа (формировать список) по одному в одной строке (получается как бы ввод чисел в столбик)

Как можно другим способом вводить числа (данные) сразу в одну строку?

Прежде, чем мы приступим к написанию кода программы, давайте продумаем **алгоритм** работы нашей будущей программы.

Алгоритм?

Своими словами, **алгоритм** – это действия, которые надо выполнять друг за другом или в определенной последовательности, чтобы решить поставленную задачу.

Пользователь программы вводит числа в одну строку, разделяя каждое число пробелом:

12 56 45 98 23 45 21 - и нажимает Enter.

Нам необходимо считать эту строку в программу - для этого мы используем функцию **input()**

В итоге мы получим строку в виде:

'12 56 45 98 23 45 21' - не забываем, что в нашей строке между числами (конечно - символами чисел) находится, по меньшей мере, один пробел.

Теперь мы можем к нашей строке применить встроенный метод **split()** без параметров. Этот метод разрежет нашу строку по пробелам (один или несколько пробелов для метода split() не важно). Метод удалит все пробелы и вернет нам уже список из символов чисел:

['12','56','45','98','23','45','21']

Так как элементы списка на данный момент это не числа, а символы чисел (строки), то задача еще не выполнена. Теперь нам необходимо каждый элемент списка перевести в число. Но функцию **int()**, с помощью которой можно одну строку числовых символов перевести в число, **применить сразу к списку нельзя!**

Доказательство:

```
a = ['12','56','45','98','23','45','21']

print(int(a))
```

Ошибка: **builtins.TypeError: int() argument must be a string, a bytes-like object or a number, not 'list'**

В этой ошибке нам говорится, что аргумент функции int() может быть строкой, двоичным объектом или числом, но никак не списком (не типом данных - list)

Для этого в Python есть встроенная функция **map()**, которая имеет **два параметра. Первый параметр** - это что делать (к примеру, какую функцию применить); **Второй параметр** - с кем (над кем) производить действия.

В нашем случае: **map(int, a)**

Но нужно учитывать, что функция **map() возвращает** не сам измененный список, а **объект класса map!**

Доказательство:

```
a = ['12','56','45','98','23','45','21']

print(map(int, a))
```

Результат:

```
<map object at 0x0000000002D09C40>
```

О различных классах объектов в языке Python мы поговорим немного позднее, а пока нам надо только понять, что мы еще не получили нужный нам список из введенных в одну строку пользователем чисел.

Применяем теперь к объекту класса map функцию **list()**:

```
a = ['12','56','45','98','23','45','21']

print(list(map(int, a)))
```

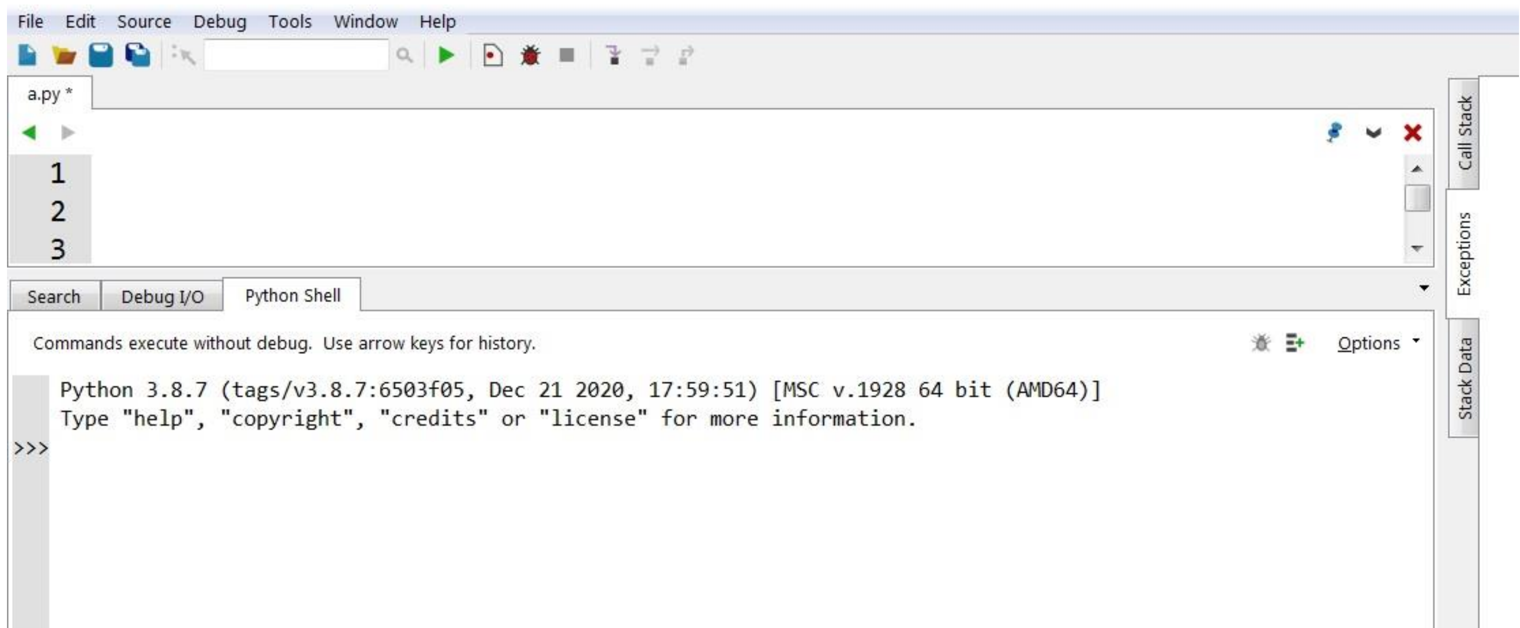
Результат:

```
[12, 56, 45, 98, 23, 45, 21]
```

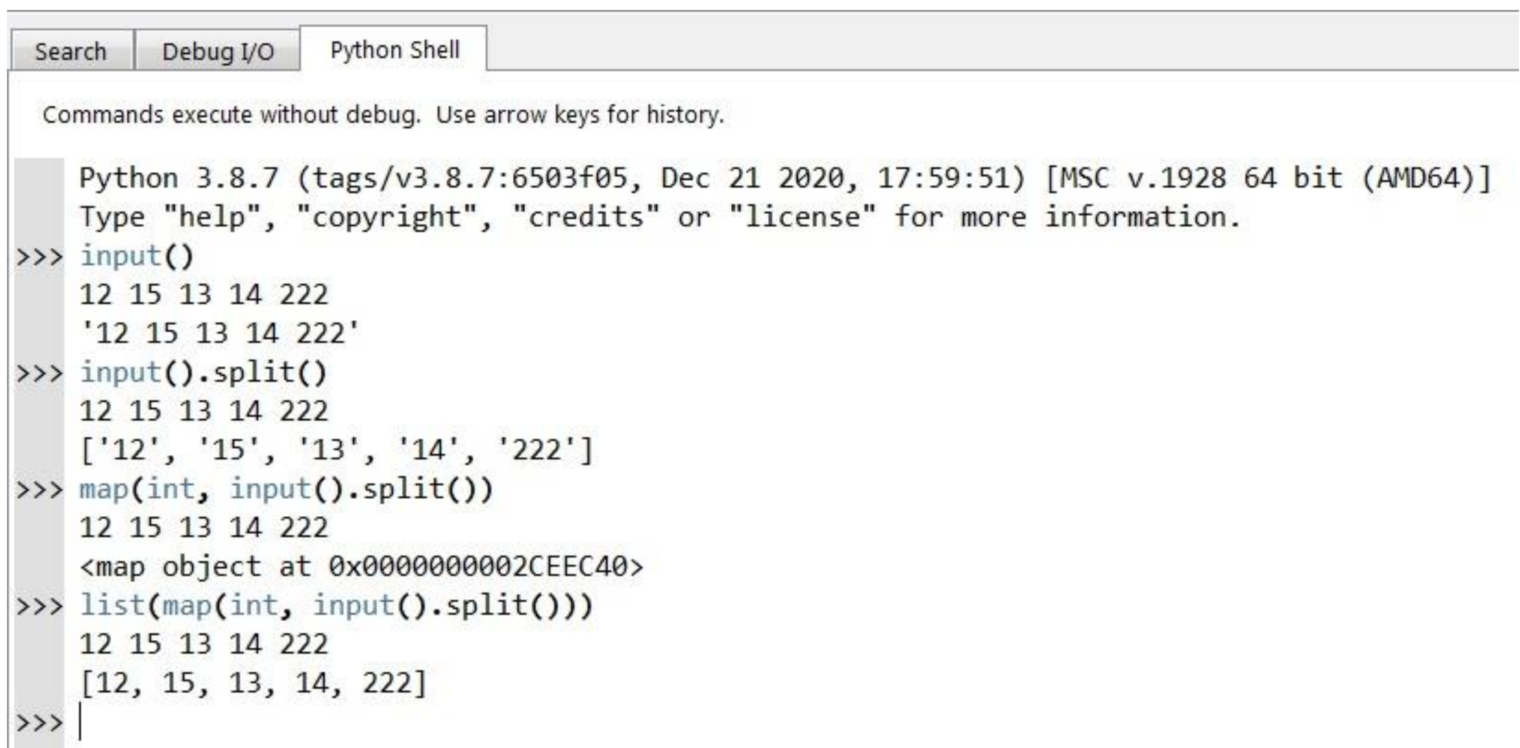
Вот теперь мы получили нужный конечный результат - **задача выполнена.**

Давайте теперь напишем программу на языке Python для решения нашей задачи:

Для наглядности, сначала давайте будем по шагам набирать (выполнять) наш код во вкладке Python Shell (в консоли).



Набираем по шагам нашего алгоритма команды и смотрим результаты трансформации введенных данных:



Удобный способ, чтобы наглядно посмотреть, как будет постепенно решать нашу задачу Python.

В итоге вот так будет выглядеть код нашей программы:

```
a = list(map(int, input().split()))

print(a)
```

Кстати, очень часто придется в дальнейшем получать данные от пользователя таким способом в программе. Поэтому можно

смело записать в своих тетрадях это код, для быстрого его использования в дальнейшем.

```
a = list(map(int, input().split()))
```

Единственное нужно понимать, что данная программа будет работать корректно, только если пользователь будет вводить именно символы чисел! Как только он введет вместо числа буквы, то произойдет ошибка!

Потому, что перевод в число букву или слово с помощью функции int() вызывает ошибку:

Пример:

```
a = 'dfdfd'

print(int(a))
```

Видим ошибку: **builtins.ValueError: invalid literal for int() with base 10: 'dfdfd' - неправильный литерал для функции int().**

В десятичной системе счисления нет числа **dfdfd.**

Можно немного переписать нашу программу на случай получения списка чисел, если пользователь захотел вводить числа через запятую: 1,23,56,89,12,45

```
a = list(map(int, input().split(',')))

print(a)
```

Результат:

```
1,23,56,89,12,45
[1, 23, 56, 89, 12, 45]
```

Здесь мы явно указали в методе split() значение параметра, по которому необходимо делать срез для формирования списка.

Напомню по умолчанию в методе split() срез происходит по пробелам в строке.

А как изменить нашу программу, чтобы она могла корректно формировать список чисел, если пользователь не только ставит между вводимыми числами запятую, но еще дополнительно отделяет вводимые числа пробелом?

1, 23, 56, 89, 12, 45

.................?

Ответ:

```
a = list(map(int, input().split(', ')))

print(a)
```

Результат:

```
1, 23, 89, 45, 56
[1, 23, 89, 45, 56]
```

Но здесь нужно обратить внимание на то, что если пользователь будет вводить числа, и разделять двумя способами, например, через запятую или через запятую с пробелом, то программа выдаст ошибку.

Вводим: 1,2,2, 5, 65

```
a = list(map(int, input().split(', ')))

print(a)
```

Ошибка: **builtins.ValueError: invalid literal for int() with base 10: '1,2,2'**

Опять ошибка - неправильный литерал. Потому, что функция **split(', ')** ищет исключительно разделитель: запятую с пробелом! И первый элемент для списка получается = **'1,2,2'**, потом этот элемент направляется в функцию int(), где и происходит ошибка, потому, что нет такого десятичного числа = **'1,2,2'**.

Метод split(), в данном случае, не интересует одна запятая в качестве разделителя! Будьте внимательны!

Но совсем другое дело с первым нашим кодом:

```
a = list(map(int, input().split(',')))

print(a)
```

Результат:

```
1,2,2, 5,  65
[1, 2, 2, 5, 65]
```

В этом случае все пройдет гладко и программа выполнит успешно задание.

Почему?
......................?

Ответ: Потому, что функция int() умная, и умеет определять и игнорировать пробелы.

```
print(int(' 5'))
print(int('  5'))
print(int('   5'))
```

Результат:

```
5
5
5
```

Видим, что функция int() игнорирует пробелы в строке с числовым символом.

Отлично! Теперь мы умеем получать (формировать) список.

Как теперь вывести на экран не сам список, а только значения его элементов?

К примеру, вот такой код:

```
a = [1,11,22]
```

```
for i in range(len(a)):
    print(a[i])
```

Результат:

```
1
11
22
```

Но правильнее было бы не использовать переменную i в программе. У нас был ранее другой синтаксис для того, чтобы пройтись по списку и вывести его элементы подряд:

```
a = [1,11,22]

for i in a:
    print(i)
```

Результат тот же:

```
1
11
22
```

Эта запись (**for i in a:**) более правильная с точки зрения синтаксиса языка Python.

При этом вспоминаем, что имя переменной i вовсе не обязательно, можно переменную назвать по-другому.

```
a = [1,11,22]

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

Результат тот же:

```
1
11
22
```

Отлично! Мы научились выводить элементы списка в столбик.

Как изменить наш код программы для вывода элементов списка в одну строку (Вспоминаем теорию про end)?

```
a = [1,11,22]

for element in a:
    print(element, end = " ")
```

У функции print() есть необязательный для указания параметр end (по умолчанию end = "\n" символ перевода строки). С помощью него мы можем выводить данные в одну строку, разделяя данные указанным разделителем.

Результат:

1 11 22

Но что будет, если мы захотим вывести данные в строку через запятую?

```
a = [1,11,22]

for element in a:
    print(element, end = ",")
```

Результат:

1,11,22,

На первый взгляд вроде все хорошо, но кажется что в нашей полученной строке последняя запятая лишняя - так никто список элементов не отображает на экране.

На помощь может прийти встроенный служебный метод **join()**. Этот метод может вывести все элементы из списка в одну строку, разделенные тем разделителем, который мы укажем. И самое главное, не будет выведен разделитель в конце последнего элемента в строке, как в случае использования параметра end в функции print().

```
a = [1,11,22]

print(",".join(a))
```

В результате получаем ошибку: **builtins.TypeError: sequence item 0: expected str instance, int found -** Если своими словами, то метод join() в качестве параметра должен получить строку, а в итоге получил число. Поэтому и ошибка.

Так что перепишем немного код нашей программы.

```
a = ['1','11','22']

print(",".join(a))
```

Результат:

1,11,22

Можно и пробел добавить после запятой:

```
a = ['1','11','22']

print(", ".join(a))
```

Результат:

1, 11, 22

Вот теперь полученная строка с числами через запятую с пробелами выглядит правильно (нет запятой после последнего элемента).

Конечно, вместо запятой или пробела мы можем использовать любой разделитель.

```
a = ['1','11','22']

print("любой разделитель ".join(a))
```

Результат:

1любой разделитель 11любой разделитель 22

Возникает вопрос: как нам написать программу с применением метода join(), если нам необходимо вывести элементы списка, которые представляют собой числа? Ведь появлялась ошибка.

Вспомним код программы:

```
a = [1,11,22]

print(",".join(a))
```

Теперь необходимо изменить код программы, чтобы ошибка не появлялась и программа выполнила свою работу корректно.

Для этого нам необходимо представить каждую цифру в списке в виде строки (в виде символа цифры):

И на помощь, как инструмент, приходит уже нам знакомая функция map():

```
a = [1,11,22]

a = map(str, a)

print(",".join(a))
```

Результат:

```
1,11,22
```

Сократим в одну строку нашу программу, установим пробел в качестве разделителя вместо запятой.

Запомним эту строку. Можно записать в тетради для будущего использования. Очень часто нам придется использовать следующий код для печати элементов списка, состоящего из чисел, в одну строку через пробел или запятую.

```
a = [1,11,22]

print(" ".join(map(str, a)))
print(", ".join(map(str, a)))
```

Представленные выше программы, где не присутствуют конструкции циклов, **называются элементами функционального программирования.** Потому, что в коде программ везде присутствуют только функции, и нет циклов.

А как быть, если мы хотим с помощью функционального программирования (без использования цикла for) у списка

чисел вывести на экран каждое число в отдельной строке (столбцом)?

a = [1,23,45]

1
23
45

Вспоминаем символ перевода на другую строку - **"\n"** и применяем его вместо разделителя.

```
a = [1,11,22]

print("\n".join(map(str, a)))
```

Результат:

1
11
22

Есть еще один способ, но он не совсем корректный, потому что мы не только выведем на экран в столбик элементы списка чисел, но и получим не нужный нам список из пустых элементов в конце.
Этот способ использует функцию print():

```
a = [1,11,22]

list(map(print, a))
```

Результат:

1
11
22

Но в данном случае мы видим только результат работы функций print() для каждого элемента списка a.

А вот если мы посмотрим, что возвращает нам в программу функция list() положив результат работы функции list() в переменную b, то увидим еще и созданный (возвращенный) функцией результат в виде списка с элементами None:

```
a = [1,11,22]
b = list(map(print, a))

print(b)
```

Результат:

```
1
11
22
[None, None, None]
```

Получается, что мы конечно выполнили показ в каждой строке элементов списка a, но получили в результате работы программы еще дополнительный список с элементами None, который нас никто не просил получать.

К причинам получения списка **[None, None, None]** в результате выполнения нашей программы мы вернемся немного позднее.

Можно только сказать, что получается следующее:

```
a = [None, None, None]

print(list(a))
```

Так что этот способ мы не будем использовать в данном случае.

Будем использовать для вывода на экран чисел из списка по одному элементу в каждой строке следующий код:

```
a = [1,11,22]

print("\n".join(map(str, a)))
```

Этот способ можно записать, как правило, как шаблон.

Давайте вспомним и соберем все функции и методы, которые мы изучили для работы со списками (с массивами).

Пусть a - это массив

a.append(n) - добавляет в конец списка a элемент n

Пример:

```
a = [1,2,3,4,5,3]
a.append(999)

print(a)
```

Результат:

[1, 2, 3, 4, 5, 3, 999]

Причем можно заменить использование метода **append()** на другую запись:

```
a = [1,2,3,4,5,3]
a += [999]

print(a)
```

Результат:

[1, 2, 3, 4, 5, 3, 999]

Единственное надо запомнить:

Запись: **a += [999]** равна записи: **a = a + [999]**

Но применять запись: **a = a + [999]** мы не будем, так как такая запись будет нагружать память, и расходовать лишние ресурсы компьютера, при этом нет совершенно никакой в этом необходимости.

Чем интереснее может быть такой способ добавления элементов нашему списку **a[]**? Тем, что мы можем таким образом добавлять в конец списка любое количество элементов, а не только один, как в случае с использованием метода **append()**.

```
a = [1,2,3,4,5,3]
a += [999, 111, 555]

print(a)
```

Результат:

[1, 2, 3, 4, 5, 3, 999, 111, 555]

Но если мы попробуем добавить три элемента с помощью метода **append()**:

```
a = [1,2,3,4,5,3]
a.append(999, 111, 555)

print(a)
```

Получим ошибку: **builtins.TypeError: append() takes exactly one argument (3 given) -** ошибка говорит о том, что метод **append()** позволяет передать при его вызове только один аргумент. А мы передали три аргумента и получили ошибку.

Даже, если мы передадим одним аргументом список элементов, то получим список, в котором добавлен все равно один элемент - этот элемент уже является списком элементов, а не числом.

```
a = [1,2,3,4,5,3]
a.append([999, 111, 555])

print(a)
```

Результат:

[1, 2, 3, 4, 5, 3, [999, 111, 555]]

Так что при добавлении элементов в список нужно держать оба синтаксиса в голове, и применять эти способы в зависимости от задач, которые необходимо выполнить в программе.