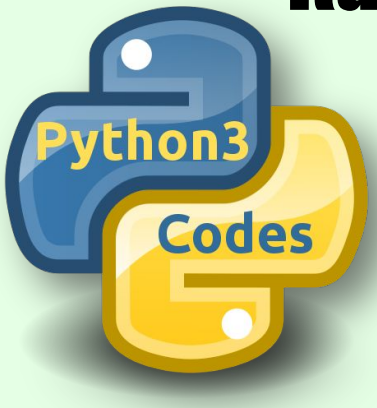
Информационные технологии

Основы программирования на Python 3

Каф. ИКТ РХТУ им. Д.И. Менделеева
Ст. преп. Васецкий А.М.



Москва, 2018

Лекция 5. Встроенные функции и элементы функционального программирования

- Встроенные функции
- Функциональное программирование в Python: ***lambda, zip, filter, map, reduce***

Встроенные функции – 1

- ***bool(x)*** – преобразование к типу bool, использующая стандартную процедуру проверки истинности. Если x является ложным или опущен, возвращает значение False, в противном случае она возвращает True.
- ***bytearray([источник [, кодировка [ошибки]]])*** – преобразование к bytearray. Bytearray – изменяемая последовательность целых чисел в диапазоне 0≤X≤ 255. Вызванная без аргументов, возвращает пустой массив байт.
- **bytes*([источник [, кодировка [ошибки]]])*** – возвращает объект типа bytes, который является неизменяемой последовательностью целых чисел в диапазоне 0≤X≤ 255. Аргументы конструктора интерпретируются как для bytearray().
- ***complex([real[, imag]])*** – преобразование к комплексному числу.
- ***dict([object])*** – преобразование к словарю.
- ***float([X])*** – преобразование к числу с плавающей точкой. Если аргумент не указан, возвращается 0.0.

Встроенные функции – 2

- ***frozenset([последовательность])*** – возвращает неизменяемое множество.
- ***int([object], [основание системы счисления])*** – преобразование к целому числу.
- ***list([object])*** – создает список.
- ***memoryview([object])*** – создает объект memoryview. Объекты *memoryview* позволяют коду Python получать доступ к внутренним данным объекта, который поддерживает буферный протокол без копирования.
- ***object()*** – возвращает безликий объект, являющийся базовым для всех объектов.
- ***range([start=0], stop, [step=1])*** – арифметическая прогрессия от start до stop с шагом step.
- ***set([object])*** – создает множество.
- ***slice([start=0], stop, [step=1])*** – объект среза от start до stop с шагом step.
- ***str([object], [кодировка], [ошибки])*** – строковое представление объекта. Использует метод __str__.
- ***tuple(obj)*** – преобразование к кортежу.

Встроенные функции – 3

- ***abs(x)*** – возвращает модуль числа.
- ***all(последовательность)*** – возвращает ***True***, если все элементы истинные или, если последовательность пуста.
- ***any(последовательность)*** – возвращает ***True***, если хотя бы один элемент – истина. Для пустой последовательности возвращает ***False***.
- ***ascii(object)*** – как и ***repr(),*** возвращает строку, содержащую представление объекта, но заменяет не-ASCII символы на экранированные последовательности.
  (см. также модуль binascii)
- ***bin(x)*** – преобразование целого числа в двоичную строку.
- ***callable(x)*** – возвращает ***True*** для объекта, поддерживающего вызов (как функции).
- ***chr(x)*** – возвращает односимвольную строку, код символа которой равен ***x***.

Встроенные функции – 4

- ***classmethod(x)*** – представляет указанную функцию методом класса См. документацию.
- ***compile(source, filename, mode, flags=0, dont_inherit=False)*** – компиляция в программный код, который впоследствии может выполниться функцией ***eval*** или ***exec***. Строка не должна содержать символов возврата каретки или нулевые байты.
- ***delattr(object, name)*** – Удаляет атрибут с именем *'name'*.
- ***dir([object])*** – Список имен объекта, а если объект не указан, список имен в текущей локальной области видимости.
- ***divmod(a, b)*** – возвращает частное и остаток от деления a на b.
- ***enumerate(iterable, start=0)*** – возвращает итератор, при каждом проходе предоставляющем кортеж из номера и соответствующего члена последовательности.

Встроенные функции – 5

- ***eval(expression, globals=None, locals=None)*** – Выполняет строку программного кода.
- ***exec(object[, globals[, locals]])*** – выполняет программный код на Python.
- ***filter(function, iterable)*** – возвращает итератор из тех элементов, для которых ***function*** возвращает истину.
- *format(value[,format_spec])* – форматирование (обычно форматирование строки).
- ***getattr(object, name ,[default])*** – извлекает атрибут объекта или default.
- ***globals()*** – словарь глобальных имен.
- ***hasattr(object, name)*** – имеет ли объект атрибут с именем 'name'.
- ***hash(x)*** – возвращает хеш указанного объекта (см. документацию).
- ***help([object])*** – вызов встроенной справочной системы.
- ***hex(x)*** – преобразование целого числа в шестнадцатеричную строку.
- ***id(object)*** – возвращает "адрес" объекта – целое число, которое будет однозначно уникальным и постоянным для данного объекта в течение срока его существования.
- ***input([prompt])*** –возвращает введенную пользователем строку. ***prompt*** – подсказка пользователю.

Встроенные функции – 6

- ***isinstance(object, ClassInfo)*** – ***True***, если объект является экземпляром ***ClassInfo*** или его подклассом. Если объект не является объектом данного типа, функция всегда возвращает ***False***.
- ***issubclass(класс, ClassInfo)*** – ***True***, если класс является подклассом ***ClassInfo***. Класс считается подклассом себя.
- ***iter(x)*** – возвращает объект итератора.
- ***len(x)*** – возвращает число элементов в указанном объекте.
- ***locals()*** – словарь локальных имен.
- ***map(function, iterator)*** – итератор, получившийся после применения к каждому элементу последовательности функции function.
- ***max(iter, [args ...] * [, key])*** – максимальный элемент последовательности.
- ***min(iter, [args ...] * [, key])*** – минимальный элемент последовательности.
- ***next(x)*** – возвращает следующий элемент итератора.
- ***oct(x)*** – преобразование целого числа в восьмеричную строку.
- ***open(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True)*** – открывает файл и возвращает соответствующий поток.

Встроенные функции – 7

- ***ord(c)*** – код символа.
- ***pow(x, y[, r])*** – это то же, что и ***(x ** y) % r***.
- ***reversed(object)*** – итератор из развернутого объекта.
- ***repr(obj)*** – представление объекта.
- ***print([object, ...], *, sep=" ", end='\n', file=sys.stdout)*** – печать.
- **property(fget=None, fset=None, fdel=None, doc=None)** – декоратор
- ***round(X [, N])*** – округление до N знаков после запятой.
- ***setattr(объект, имя, значение)*** – устанавливает атрибут объекта.
- ***sorted(iterable[, key][, reverse])*** – отсортированный список.
- ***staticmethod(function)*** – статический метод для функции.
- ***sum(iter, start=0)*** – сумма членов последовательности.
- ***super([тип [, объект или тип]])*** – доступ к родительскому классу.
- ***type(object)*** – возвращает тип объекта.
- ***type(name, bases, dict)*** – возвращает новый экземпляр класса *name*.
- ***vars([object])*** – словарь из атрибутов объекта. По умолчанию – словарь локальных имен.
- ***zip(*iters)*** – итератор, возвращающий кортежи, состоящие из соответствующих элементов аргументов-последовательностей.
- См. также https://docs.python.org/3/library/functions.htm

Примеры встроенных функций

```
E=[True, False, True]
print(all(E))       # False
print(any(E))       # True
a, b = divmod(12, 5)   # 2, 2
a = pow(2, 5)       # 32
a = pow(2, 5, 10)    # 2 (= 32 % 10)
x = 1
y = eval("x+1")      # 2
s = "Хэш меняется при запусках!"
print(hash(s))       # 3850572299821043542
n = input("Введите число ") # введём 5
print(type(n), n)    # <class 'str'> 5
```

Примеры функции *isinstance*

```
b = [1,2,3]
print(type(b) == list)        # True
print(isinstance(b, list))    # True
```

isinstance() по сравнению с ***type()*** позволяет проверить данные на принадлежность хотя бы одному типу из кортежа, переданного в качестве второго аргумента:

```
print(isinstance(b,(list, tuple, dict)))     # True
print(isinstance(b,(tuple, dict)))       # False
```

isinstance поддерживает наследование

Таблицы locals и globals

```
"""Это пример глобальных значений"""
x = 10
def fun():
    y, z = 20, "локальные"
    L = locals()
    G = globals()
    print(L, G)
fun()

# {'z': 'локальные', 'y': 20}
# {'__name__': '__main__', '__doc__': 'Это пример глобальных
значений', '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at
0x00000000021A3F28>, '__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>, '__file__': 'D:/Mou
документы-AMVAS/AMVAS-6_Education/10B-Information
Technologies (Python) 2018_/tests/__main__.py', '__cached__': None, 'x':
10, 'fun': <function fun at 0x0000000002A421E0>}
```

Поиск max, min

```
x, y, z = -1, 2, 5
print(max(x, y, z))   # 5
print(min(x, y, z))   # -1
L = [-1, 2, 5]
print(max(L))      # 5
print(min(L))      # -1
L = [-1, 2, 5, [-5, 5]] # вложенные объекты не поддерживаются
print(max(L)) # ОШИБКА
```

Сортировка

sorted(iterable[, key][, reverse])
iterable – итерируемый объект.

key – ожидается в форме именованного аргумента. Функция, принимающая аргументом элемент, используемая для получения из этого элемента значения для сравнения его с другими.

✔***None*** – сравнить элементы напрямую.

✔***reverse*** – флаг, указывающий следует ли производить сортировку в обратном порядке

В отличие от метода ***sort*** функция ***sorted*** возвращает копию отсортированного объекта, а не производит сортировку внутри него.

Пример сортировки списка

- Прямая и обратная сортировка:

```
L = [1.1, -8, 5.5]
print(sorted(L))                  # [-8, 1.1, 5.5]
print(sorted(L, reverse=True))    # [5.5, 1.1, -8]
```

- Сортировка по квадратам чисел в списке:

```
print(sorted(L, key=lambda x: x*x))
# [1.1, 5.5, -8]
```

- **Случайная сортировка:**

```
from random import random
# Функция random() – часть стандартной
библиотеки, которая выдает числа в случайном
порядке от 0 до 1.
def rnd_key(element):
    return random()
a = sorted(L, key = rnd_key) # [5.5, -8, 1.1]
b = sorted(L, key = rnd_key) # [-8, 5.5, 1.1]
```

Сортировка вложенного списка

```
L = [["Анна",50, 130, 37], ["Женя",12, 135, 19],
     ["Вера", 17, 140, 23], ["Дима", 35, 129, 31]]
i = input("Номер столбца сортировки 0..3")
i = int(i) # input на выходе даёт тип str
print(sorted(L, key=lambda x: x[i]))
```

- Результат сортировки по последнему **3**-му столбцу:

```
[['Женя', 12, 135, 19], ['Вера', 17, 140, 23], ['Дима',
35, 129, 31], ['Анна', 50, 130, 37]]
```

- Аналогично сработает:

```
L.sort(key=lambda x: x[i])
print(L)
[['Женя', 12, 135, 19], ['Вера', 17, 140, 23], ['Дима',
35, 129, 31], ['Анна', 50, 130, 37]]
```

Разница между ними в том, что в первом случае исходный список **L** останется неизменным (вернёт отсортированную копию), – а во втором **L** будет отсортирован.

Пример сортировки кортежа

```
t = (5, 7, 0)
print(sorted(t))   # [0, 5, 7] – СПИСОК!
```

Пример сортировки словаря

При сортировке словаря получаем **список**!

```
d = {"c": 2, "b": 5, "a": 0}
print(sorted(d))    # ['a', 'b', 'c']
```

Упорядочиваем по ключам:

```
a = sorted(d.items(), key=lambda item: item[0])
# [('a', 0), ('b', 5), ('c', 2)]
```

Упорядочиваем по значениям:

```
b = sorted(d.items(), key=lambda item: item[1])
# [('a', 0), ('c', 2), ('b', 5)]
```

По значениям и ключам

```
sorted(d.items(), key=lambda item: (item[1], item[0]))
# [('a', 0), ('c', 2), ('b', 5)]
```

По значениям по убыванию и ключам:

```
e = sorted(d.items(), key=lambda item: (-item[1],
item[0]))
# [('b', 5), ('c', 2), ('a', 0)]
```

Функциональное программирование

- Функциональное программирование является одной из парадигм, поддерживаемых языком программирования Python.
- Основными предпосылками для полноценного функционального программирования в Python являются: функции высших порядков, развитые средства обработки списков, рекурсия, возможность организации ленивых вычислений.
- Элементы функционального программирования в Python могут быть полезны любому программисту, так как позволяют гармонично сочетать выразительную мощность этого подхода с другими подходами.

Элементы функционального программирования

- **Lambda-выражения в Python.**
- **Итераторы :**
  - ✔ ***zip()***
  - ✔ ***map()***
  - ✔ ***filter()***
- **Функция *reduce*.**

Lambda-функции

- **Лямбда-выражение** в программировании – специальный синтаксис для определения функциональных объектов, заимствованный из λ-исчисления. Обычно применяется для объявления анонимных функций по месту их использования, и допускает замыкание на лексический контекст, в котором это выражение использовано. Используя лямбда-выражения, можно объявлять функции в любом месте кода.
- **Лямбда-выражения** в Python, это однострочные функции, которые используются в случаях, когда не нужно повторно использовать функцию в программе. Они идентичны обыкновенным функциям и повторяют их поведение.
- Как правило, **lambda-функции** используются в комбинации с функциями ***filter, map, reduce***.

lambda-выражение

- ***lambda*** **–** это выражение, а не инструкция. Поэтому ключевое слово ***lambda*** может появляться там, где синтаксис языка Python не позволяет использовать определение функции через ***def***, – например, внутри литералов или в вызовах функций. Кроме того, lambda-выражение возвращает значение (новую функцию), которое при желании можно присвоить переменной, в отличие от инструкции *def*, которая всегда связывает функцию с именем в заголовке, а не возвращает ее в виде результата.
- ***def*** используется для сложных функций, вызываемых многократно, в lambda – для однократных вызовов простых функций
- Функции, создаваемые при помощи lambda-выражения, не могут содержать **инструкции**.

Примеры

- Обычное определение функции:

```
def func(x, y, z):
    return x + y + z
print(func(2, 3, 4))      # 9
```

- То же через lambda:

```
f = lambda x, y, z: x + y + z
print(f(2,3,4))          # 9
```

- Можно использовать параметры, заданные по умолчанию:

```
s = (lambda a="az", b="bu", c="ka": a + b + c)
print(s("try"))              # trybuka
print(s("try","my"))         # trymyka
```

lambda для таблиц переходов

- lambda-выражения также часто используются для создания таблиц переходов, которые представляют собой списки или словари действий, выполняемых по требованию.

```
L = [lambda x: x**2,
     lambda x: x**3,
     lambda x: x**4]    # Список из трех функций
for f in L:
    print(f(2))       # Выведет 4, 8, 16
print(L[0](3))          # Выведет 9
print(L[2](2))          # Выведет 16
print(L)              # [<function <lambda> at
0x0000000001D32E18>, <function <lambda> at
0x0000000002A421E0>, <function <lambda> at
0x0000000002A422F0>]
```

Выбор в lambda-функциях

- Логика выбора внутри lambda-функций:
  Возвращает наименьшее из двух значений:
  L = (lambda x, y: x if x < y else y)
  print(L("a", "b")) # "a"
  print(L(1, 2)) # 1

- Неограниченное количество параметров
  fun = lambda *args: args
  print(fun(5,7,9)) # 5,7,9

Простое лучше сложного, явное лучше неявного, а понятные инструкции лучше заумных выражений. Именно поэтому ***lambda*** ограничивается выражениями. Если необходимо реализовать сложную логику, используйте инструкцию ***def***

Функция zip

- ***zip*** – возвращает список кортежей, состоящих из элементов входных списков с одинаковыми индексами. Его можно использовать для одновременного обхода нескольких последовательностей в цикле *for*.
- Формат: ***zip(*iterables)***

iterables – итерируемые объекты, элементы которых следует упаковать в кортежи. Если передана одна последовательность, вернётся итератор по кортежам, состоящим из единственного элемента. Если последовательности не переданы, возвращается пустой итератор.

Итератор останавливается, когда исчерпана кратчайшая из последовательностей.

Реализация zip

```
def zip(*iterables):  # zip("ABCD", "xy") --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result) # кортеж на выходе
```

- ***Примечание:*** если необходимо, чтобы для каждого из элементов более длинного массива в результирующем списке был создан кортеж из одного элемента, то можно воспользоваться ***zip_longest*** из пакета ***itertools***.

Примеры

```
list(zip([1, 2, 3], [4, 5], [6, 7])) # [(1, 4, 6), (2, 5, 7)]

a = [1, 2, 3]
b = "xyz"
c = (None, True)
res = list(zip(a, b, c)) # [(1, 'x', None), (2, 'y', True)]
```

- Часто zip используется для создания пар последовательностей. Например, для словаря:

```
names = ["Аня", "Таня", "Маня"]
ages = [17, 20, 10]
print(dict(zip(names, ages)))
# {'Аня': 17, 'Таня': 20, 'Маня': 10}
```

Операции с zip

- Операция перемножения каждого элемента списка на свой коэффициент:

```
values = [1, 2, 3]
coefficient = [10, 20, 30]
for i, j in zip(values, coefficient):
    print(i*j)          # 10 40 90
```

- Использование ***zip*** с генератором ***range***

```
a = []
b = []
for i, j in zip(range(10, 20), range(1, 10)):
    a.append(i) # 9, а не 10 элементов!
    b.append(j) # 9 элементов
print(a) # [10, 11, 12, 13, 14, 15, 16, 17, 18]
print(b) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Обход нескольких последовательностей

```
a = [1, 2, 3]
b = ["a", "bc", "d", "e"]
for i in zip(a, b):
    print(i , end=" ") # (1, 'a') (2, 'bc') (3, 'd')
for i, j in zip(a, b):
    print(i , j, end=" ") # 1 a 2 bc 3 d
```

zip(*[...])

В сочетании с оператором * функция может быть использована для распаковки списка

```
first, second = zip(*[(1, 4), (2, 5), (3, 6)])
# (1, 2, 3), (4, 5, 6)
```

Гарантируется вычисление слева направо, что делает возможным следующую идиому кластеризации данных по группам n-длины – *zip(*[iter(s)] * n):*

```
seq = [1, 2, 3, 4, 5, 6]
# Для n=2
list(zip(*[iter(seq)] * 2)) # [(1, 2), (3, 4), (5, 6)]
# Для n=3
list(zip(*[iter(seq)] * 3)) # [(1, 2, 3), (4, 5, 6)]
```

Транспонирование списка

```
from pprint import pprint
# модуль pprint используется для удобного вывода на экран
matrix =   [[11, 12, 13, 14, 15],
            [21, 22, 23, 24, 25],
            [31, 32, 33, 34, 35],
            [41, 42, 43, 44, 45]]
matrix_t = list(zip(*matrix)) # непосредственно транспонирование
pprint(matrix_t)
[(11, 21, 31, 41),
 (12, 22, 32, 42),
 (13, 23, 33, 43),
 (14, 24, 34, 44),
 (15, 25, 35, 45)]
```

unzip

- Обратная операция от zip:

```
coord = ["x", "y", "z"]
value = [0, 1, 2, 3, 4]
result = zip(coord, value)
resultList = list(result)
print(resultList)      # [('x', 0), ('y', 1), ('z', 2)]
c, v =  zip(*resultList) # unzip
print("c =", c)        # c = ('x', 'y', 'z')
print("v =", v)        # v = (0, 1, 2)
```

Функция map

- ***map(function, iterable, ...)***
- Возвращает итератор, который применяет *function* к каждому элементу из *iterable*. Если передаются дополнительные аргументы *iterable*, *function* должна принять все множество аргументов и будет применена к элементам из всех параллельно. С несколькими итерируемыми итератор останавливается, когда самая короткая итерация исчерпана. Для случаев, когда входы функции уже расположены в аргументе кортежей см. ***itertools.starmap()***

Примеры использования map

- Пример. Преобразование списка строк к целочисленному списку стандартным путём:

```
oldi = ["1", "2", "3"]
newi = []
for item in oldi:
    newi.append(int(item))    # [1, 2, 3]
```

- ..и с помощью map:

```
newi = list(map(int, oldi))   # [1, 2, 3]
```

- Умножим все элементы списка **d** на 3

```
d = [1, 2, 3]
e = list(map(lambda x:x*3, d)) # [3, 6, 9]
```

- Операции с несколькими параметрами. :

```
e = list(map(lambda a,b,c:a+b*c, [1,2,3,4], [2,3,4,5],
[3,4,5,6])) # [7, 14, 23, 34]
```

Передача нескольких последовательностей в map

- При передаче нескольких последовательностей функция ***map*** предполагает, что ей будет передана функция, принимающая ***N*** аргументов для ***N*** последовательностей.
- Последовательно передаём в функцию ***pow*** два списка: ***(***примечание. ***pow(x, y)*** это ***x**y)***

```
x = list(map(pow, [1, 2, 3], [4, 5, 6])) # [1, 32, 729]
                                         # 1**4, 2**5, 3**6
```

- Современная реализация ***map*** зачастую обладает более высокой производительностью, чем генераторы списков (например, когда отображается встроенная функция), и использовать ее проще.

Примеры map

- Использование пользовательских функций:

```
def metr_to_cm(m):
    return m * 100
meters = [1.0, 5.0, 7.5]
cm = list(map(metr_to_cm, meters))
```

…то же самое через лямбда-функцию:

```
cm = list(map(lambda x: x * 100, meters))
```

- **map** может быть применена для нескольких списков. Тогда функция-аргумент должна принимать количество аргументов, соответствующее количеству списков. Если количество элементов в списках совпадать не будет, то выполнение закончится на минимальном списке:

```
a, b, c = [1, 2, 3], [4, 5, 6], [7, 8]
newi1 = list(map(lambda x, y: x + y, a, b)) # [5, 7, 9]
newi2 = list(map(lambda x, y: x + y, a, c)) # [8, 10]
```

Функция filter

- ***filter(function, iterable)*** – Строит итератор из тех элементов *iterable*, для которых *function* возвращает ***true***.
- *iterable* может быть либо последовательностью, контейнером, который поддерживает итерацию, либо итератором. Если *function* есть ***None***, предполагается идентичная функция, то есть все элементы *iterable*, которые являются ***false***, удаляются.
- ***filter(function, iterable)*** это эквивалент выражению-генератору:
  - ✔(***item for item in iterable if function(item)***), – если функция не ***None*** и
  - ✔(***item for item in iterable if item***), – если функция является ***None***.

Примеры фильтрации

- Отфильтруем все *"ab"* в списке ***mixt***

```
mixt = ["ab", "ac", "ad", "ab", "ab", "ac", "a"]
ft = list(filter(lambda x: x == "ab", mixt))
print(ft)     # ['ab', 'ab', 'ab']
```

- Отфильтруем все нечётные элементы списка:

```
a = list(filter(lambda x:x%2, [1,2,3,4,5,6,7,8]))
# [1, 3, 5, 7]
```

- Отфильтруем все ненулевые элементы списка:

```
a = [-1,0,1,0,0,1,0,-1]
b = list(filter(None, a))       # [-1, 1, 1, -1]
```

- Аналогично со строками:

```
b = ["a","","  ","b","cc"]
d =  list(filter(None, b))      # ['a', ' ', 'b', 'cc']
```

Функция reduce

- ***reduce(func, iterable[, initializer])*** – Применяет указанную функцию к элементам последовательности, сводя её к единственному значению.
- В Python 3.0 вынесена в модуль ***functools***

func : Функция, которую требуется применить к элементам последовательности. Должна принимать два аргумента, где первый аргумент – аккумулированное ранее значение, а второй – следующий элемент последовательности.
iterable : Последовательность, элементы которой требуется свести к единственному значению. Если последовательность пуста и не задан ***initializer***, то выбрасывается ***TypeError***.

initializer=None : Базовое значение, с которого требуется начать отсчёт. Оно же будет возвращено, если последовательность пуста.

Пример использования reduce

```
from functools import reduce
def func(prev, curr):
# prev – предшествующий элемент
# curr – текущий элемент
    return prev + 2 * curr
a = reduce(func, [1, 2, 3, 4, 5])        # 29
# a=1, (1+2*2), (5+2*3), (11+2*4), (19+2*5)
#       =5    =11    =19       =29
```

Python 3 Настоятельно рекомендуется использовать обычный проход по элементам при помощи ***for*** для повышения читаемости кода.

Спасибо за внимание