

# Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

## Тема 6. Функциональное программирование в Python

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа  
(ауд. 329)

апрель, 2025

Python поддерживает три парадигмы программирования:

- процедурная парадигма (с помощью операторов)
- **функциональная парадигма** (композиция функций)
- объектно-ориентированная парадигма (с помощью пользовательских классов)

Функциональная парадигма — это подход к программированию, когда вычисления выполняются путем *комбинирования/композиции функций*, которые изменяют только переменные из локальной области видимости и не изменяют свои аргументы (no mutability). При этом результатом выполнения функций являются только возвращаемые объекты без побочных эффектов во время выполнения функции (no side effects). Такие функции называют *pure functions*.

Python поддерживает концепцию *функций высшего порядка (higher order functions)*:

- функции могут принимать функции в качестве аргументов,
- функции могут возвращать функции в качестве результатов (например, фабричная функция, декоратор функции),
- функции могут быть элементами структур данных.

Инструменты функционального программирования получают большее распространение в коде на Python 3.x.

### Инструменты функционального программирования:

1. Понятие объекта функции, `lambda`-функции
2. Механизм итераций; включения; *неявные* инструменты организации циклов на итерируемых объектах: функции-конструкторы `map`, `filter`, `enumerate`, `zip`; генераторные объекты
3. Фабричные функции, функции-замыкания
4. Декораторы функций

## 6.1 Функция-конструктор `map`

`map(func, *iterables) -> object of type map`

Для отображения значений используется функция `map`. Функция-конструктор `map` возвращает генераторный объект, который выдает результат вызова функции `func` со значениями аргументов, последовательно задаваемыми элементами итерируемых объектов из `iterables`.

Первый аргумент `func` функции `map` является *объектом функции*. Объект функции может соответствовать встроенной функции, `lambda`-функции или пользовательской функции на основе оператора `def`

```
In [1]: map_object = map(ord, 'abcd'); print(map_object, list(map_object))
```

```
<map object at 0x00000266A9BCCBB0> [97, 98, 99, 100]
```

```
In [2]: map_object = map(lambda x: x+10, range(10)); tuple(map_object)
```

```
Out[2]: (10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

```
In [3]: def f(x): return chr(x)
map_object = map(f, range(90,100)); set(map_object)
```

```
Out[3]: {'Z', '[', '\\', ']', '^', '_', '`', 'a', 'b', 'c'}
```

При выполнении функции `map` реализуется протокол итераций для *одновременного* прохода по итерируемым объектам `iterables`. Функция `map` заканчивает выполнение, когда закончен проход по элементам для наименьшего по длине итерируемого объекта из `iterables`

```
In [4]: list(map(pow, [1,2,3],(2,3)))
```

```
Out[4]: [1, 8]
```

```
In [5]: iterables = [1,2,3],(2,3)
list(map((lambda x,y: pow(x,y)), *iterables))
```

```
Out[5]: [1, 8]
```

Первый аргумент `func` функции `map` должен поддерживать вызов с  $n$  аргументами при указании  $n$  итерируемых объектов `iterables`

```
In [6]: print(iterables)
list(map((lambda x,y: x+y), *iterables))
```

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

```
Out[6]: [3, 5]
```

```
In [7]: list(map(abs, *iterables))
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 list(map(abs, *iterables))

TypeError: abs() takes exactly one argument (2 given)
```

Функция `map` выполняется со скоростью кода на языке C внутри интерпретатора, которая выше, чем скорость выполнения байт-кода циклов `for` внутри PVM

```
In [8]: %timeit list(map(abs, range(-10**6, 0))) # функциональный стиль
80.9 ms ± 6.24 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [9]: %timeit [abs(x) for x in range(-10**6, 0)] # питоновский стиль

res = []
%timeit for x in range(-10**6, 0): res.append(abs(x)) # процедурный стиль

82.2 ms ± 4.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
90.8 ms ± 6.76 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Выражения вызова функции `map` должны быть синтаксически простыми. Для более сложных задач применяйте операторы циклов `for` и `while`.

## 6.2 Функция-конструктор `filter`

`filter(function or None, iterable) -> object of type filter`  
Для *фильтрации* значений используется функция `filter`. Функция-конструктор `filter` возвращает генераторный объект, который выдает только те элементы `elem` итерируемого объекта `iterable`, для которых вызов функции-предиката от элемента `func(elem)` возвращает значение `True`. Другими словами, функция `filter` *фильтрует* элементы итерируемого объекта `iterable` на основе функции-предиката `function`.

Первый аргумент `func` функции `filter` является объектом функции. Объект функции может соответствовать встроенной функции, `lambda`-функции или пользовательской функции на основе оператора `def`.

```
In [10]: gen_object = filter(str.isalpha, 'ab12'); print(gen_object, list(gen_object))
<filter object at 0x00000266AC426260> ['a', 'b']

In [11]: gen_object = filter((lambda x: x%5==1), range(100)); tuple(gen_object)
Out[11]: (1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71, 76, 81, 86, 91, 96)

In [12]: def f(x): return bool(x)
gen_object = filter(f, range(-5, 5)); set(gen_object)
Out[12]: {-5, -4, -3, -2, -1, 1, 2, 3, 4}
```

Первый аргумент функции `filter` может быть объектом `None`. В этом случае функция `filter` возвращает генераторный объект, который выдает только те

элементы итерируемого объекта `iterable`, булевское значение которых равно `True`

```
In [13]: list(filter(None, (None, 0, [], {}, 5, '')))
```

```
Out[13]: [5]
```

Альтернативный синтаксис с аналогичным результатом

```
In [14]: list(filter(bool, (None, 0, [], {}, 5, '')))
```

```
Out[14]: [5]
```

При выполнении функции `filter` реализуется протокол итераций для прохода по итерируемому объекту `iterable`.

Вместо функции `filter` можно использовать включения с условием

```
In [15]: %timeit filter(str.isalpha, 'ab12') # функциональный стиль
%timeit (x for x in 'ab12' if str.isalpha(x)) # питоновский стиль
```

```
114 ns ± 16 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
224 ns ± 26.9 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

```
In [16]: tuple(filter((lambda x: x>5), range(10))) # функциональный стиль
tuple(x for x in range(10) if x>5) # питоновский стиль
```

```
Out[16]: (6, 7, 8, 9)
```

```
In [17]: set(filter(None, range(-5,5))) # функциональный стиль
{x for x in range(-5,5) if bool(x)} # питоновский стиль
```

```
Out[17]: {-5, -4, -3, -2, -1, 1, 2, 3, 4}
```

## 6.3 Функция-конструктор `enumerate`

`enumerate(iterable, start=0)` -> object of type `enumerate`

Функция-конструктор `enumerate` возвращает генераторный объект, который выдает кортежи из двух элементов. Первый элемент кортежа содержит индекс элемента итерируемого объекта `iterable` при индексации, начиная со значения аргумента `start`. Второй элемент кортежа содержит соответствующий элемент итерируемого объекта `iterable`. Аргумент `start` функции `enumerate` имеет стандартное значение `start=0`.

При выполнении функции `enumerate` реализуется протокол итераций для прохода по итерируемому объекту `iterable`.

```
In [18]: list(enumerate("String"))
```

```
Out[18]: [(0, 'S'), (1, 't'), (2, 'r'), (3, 'i'), (4, 'n'), (5, 'g')]
```

```
In [19]: list(enumerate(range(-3,3), start=1))
```

```
Out[19]: [(1, -3), (2, -2), (3, -1), (4, 0), (5, 1), (6, 2)]
```

Пример фрагмента кода для выполнения Задания 1.2 (Момент времени падения тела) из Лб1

```
In [20]: s_y = range(-10,10)
for (k,el) in enumerate(s_y):
    if el*s_y[k+1]<=0:
        break
k, s_y[k], s_y[k+1]
```

```
Out[20]: (9, -1, 0)
```

Альтернативная реализация функции `enumerate` с помощью пользовательской генераторной функции

```
In [21]: def my_enumerate(sequence, start=0):
        i = start
        for elem in sequence:
            yield i, elem
            i += 1

gen_object = my_enumerate(range(-3,3))
```

```
In [22]: tuple(gen_object)
```

```
Out[22]: ((0, -3), (1, -2), (2, -1), (3, 0), (4, 1), (5, 2))
```

## 6.4 Функция-конструктор `zip`

`zip(*iterables) -> object of type zip`

Функция-конструктор `zip` возвращает генераторный объект, который генерирует *кортежи* из  $n$  элементов, где  $n$  -- количество итерируемых объектов `iterables`. Каждый элемент кортежа состоит из  $n$  соответствующих элементов итерируемых объектов `iterables`.

```
In [23]: gen_object = zip("String", range(10)); print(gen_object, list(gen_object))
```

```
<zip object at 0x000002673B119600> [('S', 0), ('t', 1), ('r', 2), ('i', 3), ('n', 4), ('g', 5)]
```

При выполнении функции `zip` реализуется протокол итераций для прохода по итерируемым объектам `iterables`. Функция `zip` заканчивает выполнение, когда закончен проход элементов для наименьшего по длине итерируемого объекта из `iterables`.

### Использование в конструкторе `dict`

Создание словаря из итерируемого объекта ключей и итерируемого объекта значений

```
In [24]: D = dict(zip('abc',range(3)))
D
```

```
Out[24]: {'a': 0, 'b': 1, 'c': 2}
```

Копирование словаря

```
In [25]: D_copy = dict(zip(D.keys(),D.values()))
D_copy
```

```
Out[25]: {'a': 0, 'b': 1, 'c': 2}
```

## Использование в цикле for и включениях

Функция `zip` позволяет организовывать циклы `for` и включения с параллельным обходом нескольких итерируемых объектов

```
In [26]: for (x,y,z) in zip(range(3), range(3), range(3)):
          print(x+y+z)

#List(map(lambda x,y,z: print(x+y+z), range(3), range(3), range(3)))

0
3
6
```

```
In [27]: [f'a{x}{y}{z}' for (x,y,z) in zip(range(3), range(3), range(3))]
```

```
Out[27]: ['a000', 'a111', 'a222']
```

Пример использования функции `zip` с распаковкой

```
In [28]: a = tuple(range(5)); b = tuple(range(5))
z = zip(a, b)
A, B = zip(*z)
A == a, B == b
```

```
Out[28]: (True, True)
```

```
In [29]: (a,b) == tuple(zip(*zip(a,b)))
```

```
Out[29]: True
```

Пример использования функции `zip` для транспонирования матрицы:

```
In [30]: matrix = [[1,2],[3,4]]
matrixT = list(map(list,zip(*matrix)))
print(matrix, matrixT)
```

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

## 6.5 Функция reduce из модуля functools

```
In [31]: import functools
```

```
In [32]: print(functools.__doc__)
```

functools.py - Tools for working with functions and callable objects

Модуль `functools` предоставляет инструменты для работы с функциями.

### Функция `reduce`

`reduce(func, iterable, initial=0) -> any object`

Для накопления или *свертки* значений используется функция `reduce`. Функция `reduce` вызывает функцию *двух аргументов* `func` последовательно для каждого элемента итерируемого объекта `iterable`. При этом результат текущего вызова функции `func` используется в качестве *первого аргумента* для последующего вызова `func`. Аргумент `initial` является необязательным. Значение аргумента `initial` задает первый аргумент при первом вызове функции `func`.

```
In [33]: func = lambda x, y: x+y
         iterable = range(5)
         functools.reduce(func, iterable)
         # (((0+0)+1)+2)+3)+4
```

Out[33]: 10

`reduce(func, iterable, initial)`

Значение аргумента `initial` задает *первый аргумент* при первом вызове функции `func`.

Пример вычисления факториала с помощью функции `reduce`

```
In [34]: functools.reduce(lambda x, y: x*y, range(1,5), 1)
         # ((1*1)*2)*3)*4
```

Out[34]: 24

Пример вычисления минимального элемента коллекции:

```
In [35]: data = [5, 6, 3, 10]
         functools.reduce(lambda x, y: x if x<y else y, data)
```

Out[35]: 3

Альтернативная реализация функции `reduce` с помощью пользовательской функции

```
In [36]: def my_reduce(func, iterable, initial=None):
         it = iter(iterable)

         value = next(it) if initial is None else initial

         for x in it:
```

```

        value = func(value, x)

    return value

my_reduce(lambda x, y: x*y, range(1,5), 1)

```

Out[36]: 24

Иногда результат выполнения функции `reduce` можно получить другими средствами

```

In [37]: %timeit functools.reduce(lambda x,y: x + y**2, range(1_000_000))
functools.reduce(lambda x,y: x + y**2, range(1_000_000)) # функциональный стиль

```

129 ms ± 4.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Out[37]: 333332833333500000

```

In [38]: %timeit sum(y**2 for y in range(1_000_000))
sum(y**2 for y in range(1_000_000)) # питоновский стиль

```

112 ms ± 3.68 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Out[38]: 333332833333500000

```

In [39]: import numpy as np
%timeit np.sum(np.arange(1_000_000)**2)
np.sum(np.arange(1_000_000)**2) # векторизация вычислений в numpy
# !!!результат отличается от предыдущих вычислений

```

2.55 ms ± 77.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

Out[39]: 584144992

## 6.6 Фабричная функция, функция-замыкание

Если функция `funcOut` содержит внутри себя определение другой функции `funcIn` (оператор `def` или `lambda`-выражение), то функцию `funcOut` называют **объемлющей функцией**, а функцию `funcIn` – **вложенной функцией**. Уровней вложения функций может быть конечное число.

**Фабричной функцией** (function builder) будем называть объемлющую функцию, которая возвращает объект встроеной функции.

Результат вызова фабричной функции будем называть **функцией-замыканием** (closure), если вложенная функция использует локальные переменные объемлющей функции.

Пример определения фабричной функции `maker`

```

In [40]: def maker(exponent):
        return lambda x: x**exponent

```

Создадим две функции-замыкания `closure2` и `closure3` из фабричной функции `maker`

```
In [41]: closure2, closure3 = maker(2), maker(3)
closure2, type(closure2)
```

```
Out[41]: (<function __main__.maker.<locals>.<lambda>(x)>, function)
```

Вызовем функции-замыкания

```
In [42]: closure2(3), closure3(3)
```

```
Out[42]: (9, 27)
```

Пример фабричной функции одного аргумента `sin_diff(dx)`, которая для заданного значения приращения  $dx$  переменной  $x$  определяет функцию-замыкание для приближенного вычисления производной от функции  $\sin(x)$  по формуле:

$$f'(x) \approx \frac{\sin(x + dx) - \sin(x)}{dx}.$$

Определение фабричной функции

```
In [43]: import numpy as np
def sin_diff(dx):
    assert dx != 0, 'dx не должно быть нулем'
    return lambda x: (np.sin(x+dx)-np.sin(x))/dx
```

Определение функций-замыканий для различных значений  $dx$

```
In [44]: closures = [sin_diff(dx) for dx in (0.1, 0.3, 0.5)]
```

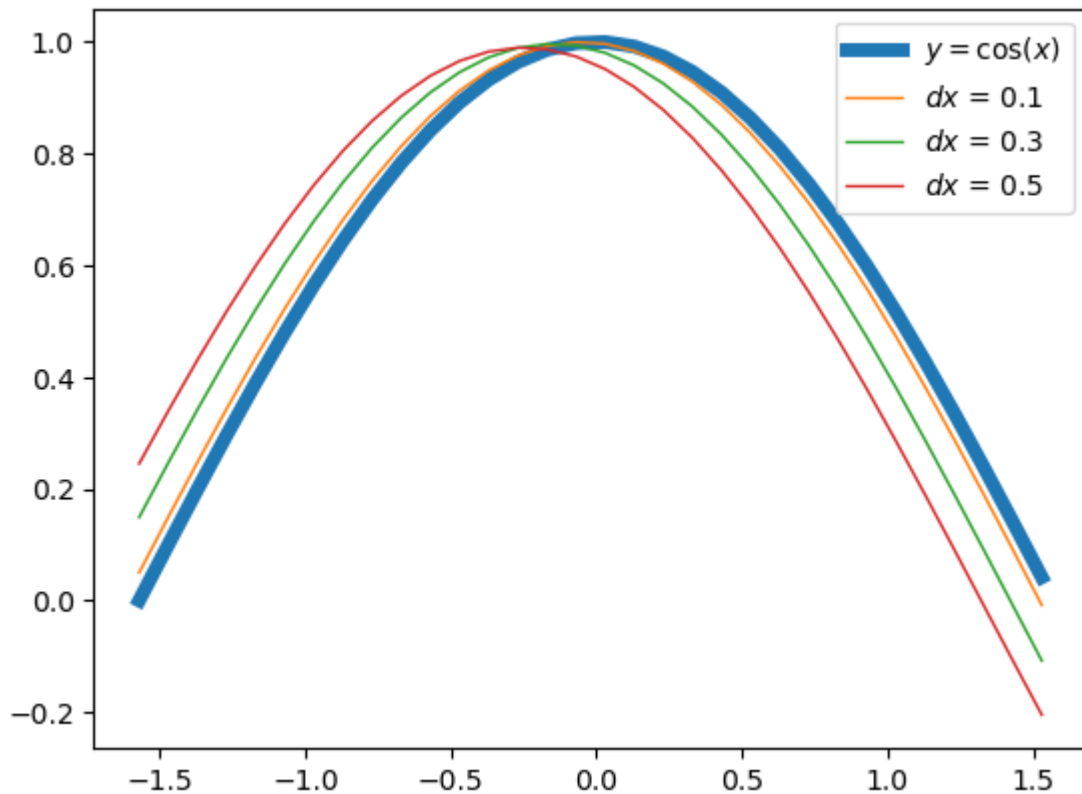
Изобразим в одной графической области графики функций приближений для производной от функции  $\sin(x)$ , а также функцию производной  $\cos(x)$

```
In [45]: import matplotlib.pyplot as plt

x = np.arange(-np.pi/2, np.pi/2, 0.1)
y = np.transpose(np.array([closure(x) for closure in closures]))
```

```
In [56]: plt.plot(x, np.cos(x), lw=5)
plt.plot(x, y, lw=1)
plt.legend([r'$y = \cos(x)$', '$dx$ = 0.1', '$dx$ = 0.3', '$dx$ = 0.5'])
```

```
Out[56]: <matplotlib.legend.Legend at 0x266ac729760>
```



Объект функции-замыкания может использовать локальные переменные из пространства имен фабричной функции после окончания выполнения фабричной функции!

Пример определения функции-замыкания со счетчиком вызовов функции. Переменная счетчика `n` создается в пространстве имен фабричной функции `maker` и *изменяется* при вызове функции-замыкания

```
In [47]: def maker(start):
          n = start
          def funcIn(x):
              nonlocal n # !!!
              n += 1
              return n, x
          return funcIn
```

```
In [48]: def maker(start):
          n = start
          def funcIn(x):
              nonlocal n # !!!
              n += 1
              return n, x
          return funcIn
```

Объявление `nonlocal` в теле вложенной функции для переменных объемлющих функций позволяет *изменять* значения этих переменных в теле вложенной функции.

Вызов фабричной функции `maker` возвращает функцию-замыкание

```
In [49]: closure = maker(0)
closure
```

```
Out[49]: <function __main__.maker.<locals>.funcIn(x)>
```

```
In [50]: closure('first'), closure('second')
```

```
Out[50]: ((1, 'first'), (2, 'second'))
```

Новый вызов фабричной функции создаст новое пространство имен фабричной функции и, как следствие, переопределение счетчика

```
In [51]: closure_new = maker(0); closure_new('new')
```

```
Out[51]: (1, 'new')
```

## 6.7 Декоратор функции

**Декоратор функции** -- это функция высшего порядка, создающая обертку (wrapper) другой функции для расширения поведения заданной функции с помощью дополнительного кода без изменения кода заданной функции.

Декоратор функции в Python -- это инструмент, который позволяет реализовать с минимальным и читабельным синтаксисом следующую ситуацию: определение функции, которая принимает другую функцию в качестве аргумента, расширяет функциональность переданной функции дополнительным кодом до и после ее вызова без изменения кода переданной функции и возвращает измененную функцию (обертку) в качестве результата.

Пример определения функции декоратора `funcOut` для декорируемой функции `my_func`

```
In [52]: def funcOut(func):
          def wrapper(*args, **kwargs):
              print(f'Calling {func.__name__}()')
              func(*args, **kwargs)
              print(f'Done {func.__name__}()')
          return wrapper

          def my_func():
              print('function is working ...')

          my_func = funcOut(my_func) # создание декорированной версии my_func
          my_func()
```

```
Calling my_func()
function is working ...
Done my_func()
```

Альтернативная реализация с использованием специального синтаксиса: `@<имя функции декоратора>` перед определением декорируемой функции

```
In [53]: @funcOut
def my_func():
    print('function is working ...')

# my_func = funcOut(my_func) # этот код выполняется автоматически
my_func()
```

Calling my\_func()  
function is working ...  
Done my\_func()

После использования декоратора переменная `my_func` ссылается не на  
изначальную функцию, а на декорированную.

```
In [54]: help(my_func)
```

Help on function wrapper in module \_\_main\_\_:  
  
wrapper(\*args, \*\*kwargs)

Пример функции декоратора для измерения времени выполнения функции

```
In [55]: from time import time

def executiontime(func):
    def wrapper(*args, **kargs): # обертка
        start = time()
        func(*args, **kargs)
        end = time()
        print(f'Функция {func.__name__} выполнялась: {end - start:.2f} сек')
    return wrapper

@executiontime
def create_tuple():
    return tuple(range(10**7))

create_tuple()
```

Функция create\_tuple выполнялась: 0.24 сек

Декораторы функций позволяют сократить повторяющийся код до и после вызова  
функций.