# Лекция 5
# Элементы функционального программирования в императивном языке Python

Рекомендуемая литература:

1. Functional Programming with Python, Pramode C.E., Linux Gazette, 2004

2. David Mertz. Functional Programming in Python. — O'Reilly, 2015. — ISBN 978-1-491-92856-1.

3. Steven Lott. Functional Python Programming. — Packt Publishing Ltd, 2015. — 360 p. — ISBN 978-1-78439-761-6.

4. https://www.linuxjournal.com/content/book-excerpt-chapter-6-functions-and-functional-programming

5. Плас Дж.В. Python для сложных задач. Наука о данных и машинное обучение/ - СПб.: Питер, 2018. — 576 с. https://www.twirpx.com/file/2353850/

## Определение и использование функции

Функция в Python может быть определена с помощью оператора `def` или лямбда-выражением. Следующие операторы эквивалентны:

```
def func(x, y):
    return x**2 + y**2

func = lambda x, y: x**2 + y**2
```

В определении функции фигурируют формальные аргументы. Некоторые из них могут иметь значения по умолчанию. Все аргументы со значениями по умолчанию следуют после аргументов без значений по умолчанию.

При вызове функции задаются фактические аргументы. Например:

```
func(2, y=7)
```

В начале идут позиционные аргументы. Они сопоставляются с именами формальных аргументов по порядку. Затем следуют именованные аргументы. Они сопоставляются по именам и могут быть заданы в вызове функции в любом порядке. Разумеется, все аргументы, для которых в описании функции не указаны значения по умолчанию, должны присутствовать в вызове функции. Повторы в именах аргументов недопустимы.

Функция всегда возвращает только одно значение (или `None`, если значение не задано в операторе `return` или этот оператор не встречен по достижении конца определения функции). Однако, это незначительное ограничение, так как возвращаемым значением может быть кортеж.

Определив функцию с помощью лямбда-выражения, можно тут же её использовать:

```
>>> (lambda x: x+2)(5)
7
```

Лямбда-выражения удобны для определения не очень сложных функций, которые передаются затем другим функциям.

Функции в Python являются объектами первого класса, то есть, они могут употребляться в программе наравне с объектами других типов данных.

## Списковые включения

**Списковое включение** — наиболее выразительное из функциональных средств Python. Например, для вычисления списка квадратов положительных целых чисел, меньших 10, можно использовать выражение:

```
l = [x**2 for x in range(1,10)]
```

## Встроенные функции высших порядков

В Python есть функции, одним из аргументов которых являются другие функции: `map()`, `filter()`, `reduce()`, `apply()`.

### map()

Функция `map()` позволяет обрабатывать одну или несколько последовательностей с помощью заданной функции:

```
>>> list1 = [7, 2, 3, 10, 12]
>>> list2 = [-1, 1, -5, 4, 6]
>>> map(lambda x, y: x*y, list1, list2)
[-7, 2, -15, 40, 72]
```

Аналогичного (только при одинаковой длине списков) результата можно добиться с помощью списочных выражений:

```
>>> [x*y for x, y in zip(list1, list2)]
[-7, 2, -15, 40, 72]
```

**filter()**

Функция `filter()` позволяет фильтровать значения последовательности. В результирующем списке только те значения, для которых значение функции для элемента истинно:

```
>>> numbers = [10, 4, 2, -1, 6]
>>> filter(lambda x: x < 5, numbers)     # В результат попадают
только те элементы x, для которых x < 5 истинно
[4, 2, -1]
```

То же самое с помощью списковых выражений:

```
>>> numbers = [10, 4, 2, -1, 6]
>>> [x for x in numbers if x < 5]
[4, 2, -1]
```

**reduce()**

Для организации цепочечных вычислений в списке можно использовать функцию `reduce()`. Например, произведение элементов списка может быть вычислено так (для Python 2):

```
>>>from functools import reduce
>>>numbers = [2, 3, 4, 5, 6]
>>> reduce(lambda res, x: res*x, numbers, 1)
720
```

Вычисления происходят в следующем порядке:

```
(((2*3)*4)*5)*6
```

Цепочка вызовов связывается с помощью промежуточного результата (`res`).

Если список пустой, просто используется третий параметр (в случае произведения нуля множителей это 1):

```
>>> reduce(lambda res, x: res*x, [], 1)
1
```

Разумеется, промежуточный результат необязательно число. Это может быть любой другой тип данных, в том числе и список. Следующий пример показывает *реверс* списка:

```
>>> reduce(lambda res, x: [x]+res, [1, 2, 3, 4], [])
[4, 3, 2, 1]
```

Для наиболее распространенных операций в Python есть встроенные функции:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> sum(numbers)
15
>>> list(reversed(numbers))
[5, 4, 3, 2, 1]
```

В Python 3 встроенной функции `reduce()` нет, но её можно найти в модуле `functools`.

**apply()**

Функция для применения другой функции к позиционным и именованным аргументам, заданным списком и словарем соответственно (для Python 2):

```
>>> def f(x, y, z, a=None, b=None):
...     print x, y, z, a, b
...
>>> apply(f, [1, 2, 3], {'a': 4, 'b': 5})
1 2 3 4 5
```

В Python 3 вместо функции `apply()` следует использовать специальный синтаксис:

```
>>> def f(x, y, z, a=None, b=None):
...     print(x, y, z, a, b)
...
>>> f(*[1, 2, 3], **{'a': 4, 'b': 5})
1 2 3 4 5
```

## Замыкания

Функции, определяемые внутри других функций, представляют собой полноценные замыкания:

```
def multiplier(n):
    "multiplier(n) возвращает функцию, умножающую на n"
    def mul(k):
        return n*k
    return mul
# того же эффекта можно добиться выражением
# multiplier = lambda n: lambda k: n*k
mul2 = multiplier(2) # mul2 - функция, умножающая на 2, например,
mul2(5) == 10
```

## Итераторы

Другие средства функционального программирования доступны из стандартной библиотеки (например, модуль `itertools`) и других библиотек.

Следующий пример иллюстрирует применение перечисляющего и сортирующего итераторов (итератор не может быть напечатан оператором `print`, поэтому оставшиеся в нем значения были помещены в список):

```
>>> it = enumerate(sorted("PYTHON"))  # итератор для перечисленных отсортированных букв слова
>>> it.next()                         # следующее значение
(0, 'H')
>>> print list(it)                    # оставшиеся значения в виде списка
[(1, 'N'), (2, 'O'), (3, 'P'), (4, 'T'), (5, 'Y')]
```

Следующий пример иллюстрирует использование модуля `itertools`:

```
>>> from itertools import chain
>>> print list(chain(iter("ABC"), iter("DEF")))
['A', 'B', 'C', 'D', 'E', 'F']
```

В следующем примере иллюстрируется функция `groupby` (группировать по), с помощью которой порождается список пар значение ключа и соответствующий ключу итератор (в этот итератор собраны все значения исходного списка с одинаковым значением ключа). В примере ключом является True или False в зависимости от положительности значения. (Для целей вывода каждый итератор превращается в список).

```
from math import cos
from itertools import groupby
lst = [cos(x*.4) for x in range(30)]                    # 
косинусоида
[list(y) for k, y in groupby(lst, lambda x: x > 0)]     # группы 
положительных и отрицательных чисел
```

В модуле `itertools` есть и другие функции для работы с итераторами, позволяющие кратко (в функциональном стиле) и с вычислительной точки зрения — эффективно — выразить требуемые процессы обработки списков.

## Модуль functools

В Python 2.5 появился модуль `functools` и в частности возможность *частичного применения функций*:

```
>>> from  functools import partial
>>> def myfun(a, b): return a + b
...
>>> myfun1 = partial(myfun, 1)
>>> print myfun1(2)
3
```

(Частичное применение функций также можно реализовать с помощью замыканий или функторов)

## Ленивые вычисления

**Ленивые вычисления** можно организовать в Python несколькими способами, используя различные механизмы:

- простейшие логические операции or и and не вычисляют второй операнд, если результат определяется первым операндом
- лямбда-выражения
- определенные пользователем классы с ленивой логикой вычислений[3] или функторы
- Генераторы и генераторные выражения
- (Python 2.5) if-выражение имеет «ленивую» семантику (вычисляется только тот операнд, который нужен)

Пример, который иллюстрирует работу if-выражения. С помощью оператора `print` можно проследить, какие функции реально вызывались:

```
>>> def f():
...     print "f"
...     return "f"
...
>>> def g():
...     print "g"
...     return "g"
...
>>> f() if True else g()
f
'f'
>>> f() if False else g()
g
'g'
```

Некоторые примеры из книги рецептов:

- Ленивая сортировка Lazy sorting
- Ленивый обход графа Lazy Traversal of Directed Graphs
- Ленивое вычисление свойств Lazy property evaluation
- Карринг Lazy property evaluation

## Функторы

Функторами называют объекты, синтаксически подобные функциям, то есть поддерживающие операцию вызова. Для определения функтора нужно перегрузить оператор `()` с помощью метода `__call__`. В Python функторы полностью аналогичны функциям, за исключением специальных атрибутов (`func_code` и некоторых других). Например, функторы можно передавать в качестве функций обратного вызова (callback) в C-код. Функторы позволяют заменить некоторые приёмы, связанные с использованием замыкания, статических переменных и т. п.

Ниже представлено замыкание и эквивалентный ему функтор:

```
def addClosure(val1):
    def closure(val2):
        return val1 + val2
    return closure

class AddFunctor(object):
    def __init__(self, val1):
        self.val1 = val1
    def __call__(self, val2):
        return self.val1 + val2

cl = addClosure(2)
fn = AddFunctor(2)

print cl(1), fn(1)  # напечатает "3 3"
```

Следует отметить, что код, использующий замыкание, будет исполняться быстрее, чем код с функтором. Это связанно с необходимостью получения атрибута `val` у переменной `self` (то есть функтор проделывает на одну Python операцию больше). Также функторы нельзя использовать для создания декораторов с параметрами.

С другой стороны, функторам доступны все возможности ООП в Python, что делает их очень полезными для функционального программирования. Например, можно написать функтор, который будет «запоминать» исполняемые над ним операции и затем повторять их. Для этого достаточно соответствующим образом перегрузить специальные методы.

```
class SlowFunctor(object):
        def __init__(self,func):
                self.func = func
        def __add__(self,val):                  # сложение функтора с
чем-то
                if isinstance(val,SlowFunctor): # если это функтор
                        new_func = lambda *dt,**mp : self(*dt,**mp) +
val(*dt,**mp)
                else:                           # если что-то другое
                        new_func = lambda *dt,**mp : self(*dt,**mp) +
val
                return SlowFunctor( new_func )
        def __call__(self,*dt):
                return self.func(*dt)

import math
```

```
def test1(x):
    return x + 1
def test2(x):
    return math.sin(x)

func = SlowFunctor(test1)                # создаем функтор
func = func + SlowFunctor(test2)         # этот функтор можно
складывать с функторами
func = (lambda x : x + 2)(func)          # и числами, передавать в
качестве параметра в функции
                                         # как будто это число

def func2(x):                            # Эквивалентная функция
    return test1(x) + test2(x) + 2

print func(math.pi)                      # печатает 3.14159265359
print func(math.pi) - func2(math.pi)     # печатает 0.0
```

Функторы привносят в Python возможность ленивых вычислений, присущую функциональным языкам: вместо вычисления результата выражения — динамическое определение новых функций комбинированием имеющихся.

Определенный подобным образом функтор создает значительные накладные расходы, так как при каждом вызове проходит по вызовам всех вложенных `lambda`. Можно оптимизировать функтор, применив технику генерирования байткода во время исполнения.

Соответствующий пример и тесты на скорость есть в примерах Python программ - https://ru.wikiversity.org/wiki/Примеры_программ_на_языке_Python .

При использовании этой техники скорость исполнения не будет отличаться от «статического» кода (если не считать времени, требующегося на однократное конструирование результирующей функции). Вместо байтокода Python можно генерировать на выходе, например, код на языке программирования C, других языках программирования или XML-файлы.

Несмотря на накладные расходы, ленивое вычисление может дать заметный выигрыш в скорости в случаях, когда действия, оборачиваемые ленивым функтором, достаточно дороги — например, включают объёмные вычисления или доступ к диску. Предположим некоторый промежуточный результат *X* лениво вычисляется перед условным оператором; для него будет создана цепочка функторов. В той ветке условного оператора, где значение *X* не требуется по ходу вычисления, эта цепочка функторов будет просто отброшена, не приведя к дорогостоящему вычислению. В другой ветке, где *X* требуется для вычисления конечного результата функции, цепочка функторов произведёт его вычисление. При этом программисту не нужно отслеживать, в какой из веток алгоритма значение может не потребоваться: он может рассчитывать, что дорогостоящее вычисление *X* произойдёт только тогда, когда его результат не будет отброшен.

codingground
SIMPLY EASY CODING

# Execute Python Online (Python v2.7.13)

Execute | Share | main.py | STDIN

```
def func(x, y):
    return x**2 + y**2

func = lambda x, y: x**2 + y**2
print func(2, y=7)
print (lambda x: x+2)(5)
list1 = [7, 2, 3, 10, 12]
list2 = [-1, 1, -5, 4, 6]
print map(lambda x, y: x*y, list1, list2)
print [x*y for x, y in zip(list1, list2)]
numbers = [10, 4, 2, -1, 6]
print filter(lambda x: x < 5, numbers)
print [x for x in numbers if x < 5]
from functools import reduce
numbers = [2, 3, 4, 5, 6]
print reduce(lambda res, x: res*x, numbers, 1)
print reduce(lambda res, x: res*x, [], 1)
print reduce(lambda res, x: [x]+res, [1, 2, 3, 4], [])
numbers = [1, 2, 3, 4, 5]
print sum(numbers)
print list(reversed(numbers))
def f(x, y, z, a=None, b=None): print x, y, z, a, b
print apply(f, [1, 2, 3], {'a': 4, 'b': 5})
```

Result

```
$python main.py
53
7
[-7, 2, -15, 40, 72]
[-7, 2, -15, 40, 72]
[4, 2, -1]
[4, 2, -1]
720
1
[4, 3, 2, 1]
15
[5, 4, 3, 2, 1]
1 2 3 4 5
None
```

# Лекция 6

# Правила написания функций на Python

В Python, как и в большинстве современных языков программирования, функция – это основной метод абстрагирования и инкапсуляции. Вы, будучи разработчиком, вероятно, написали уже сотни функций. Но функции функциям – рознь. Причем, если писать «плохие» функции, это немедленно скажется на удобочитаемости и поддержке вашего кода. Итак, что же такое «плохая» функция, а еще важнее – как сделать из нее «хорошую»?

Математика изобилует функциями, правда, припомнить их сложно. Так что давайте вернемся к нашей излюбленной дисциплине: анализу. Вероятно, вам доводилось видеть формулы вроде `f(x) = 2x + 3`. Это функция под названием `f`, принимающая аргумент `x`, а затем «возвращающая» дважды `x + 3`. Хотя, она и не слишком похожа на те функции, к которым мы привыкли в Python, она совершенно аналогична следующему коду:

```
def f(x):
    return 2*x + 3
```

Функции издавна существуют в математике, но в информатике совершенно преображаются. Однако, эта сила не дается даром: приходится миновать различные подводные камни. Давайте же обсудим, какова должна быть «хорошая» функция, и какие «звоночки» характерны для функций, возможно, требующих рефакторинга.

## Секреты хорошей функции

Что отличает «хорошую» функцию Python от посредственной? Вы удивитесь, как много трактовок допускает слово «хорошая». В рамках этой статьи я буду считать функцию Python «хорошей», если она удовлетворяет *большинству* пунктов из следующего списка (выполнить все пункты для конкретной функции порой невозможно):

- Она внятно названа
- Соответствует принципу единственной обязанности
- Содержит докстроку
- Возвращает значение
- Состоит не более чем из 30 строк
- Она идемпотентная и, если это возможно, чистая

Многим из вас эти требования могут показаться чрезмерно суровыми. Однако, обещаю: если ваши функции будут соответствовать этим правилам, то получатся настолько прекрасны, что пробьют на слезу даже единорога. Ниже я посвящу по разделу каждому из элементов вышеприведенного списка, а затем завершу повествование, рассказав, как они гармонируют друг с другом и помогают создавать хорошие функции.

**Именование**

Вот моя любимая цитата на эту тему, часто ошибочно приписываемая Дональду, а на самом деле принадлежащая Филу Карлтону:

> В компьютерных науках есть две сложности: инвалидация кэша и именование.

Как бы глупо это ни звучало, именование – действительно сложная штука. Вот пример «плохого» названия функции:

```
def get_knn_from_df(df):
```

Теперь плохие названия попадаются практически повсюду, но данный пример взят из области Data Science (точнее, машинного обучения), где практикующие специалисты обычно пишут код в блокноте Jupyter, а потом пытаются собрать из этих ячеек удобоваримую программу.

Первая проблема с названием этой функции – в нем используются аббревиатуры. **Лучше использовать полные английские слова, а не аббревиатуры и не малоизвестные сокращения**. Единственная причина, по которой хочется сокращать слова — не тратить сил на набор лишнего текста, но *в любом современном редакторе есть функция автозавершения*, поэтому вам придется набрать полное название функции всего один раз. Аббревиатура – это проблема, поскольку зачастую она специфична для предметной области. В вышеприведенном коде `knn` означает «K-ближайшие соседи», а `df` означает «DataFrame», структуру данных, повсеместно используемую в библиотеке pandas. Если код будет читать программист, не знающий этих сокращений, то он практически ничего не поймет в названии функции.

Еще в названии этой функции есть два более мелких недочета. Во-первых, слово `"get"` избыточно. В большинстве грамотно поименованных функций сразу понятно, что данная функция что-то возвращает, что конкретно – отражено в имени. Элемент `from_df` также не нужен. Либо в докстроке функции, либо (если она находится на периферии) в аннотации типа будет описан тип параметра, если эта информация *и так не очевидна из названия параметра*.

Так как же нам переименовать эту функцию? Просто:

```
def k_nearest_neighbors(dataframe):
```

Теперь даже неспециалисту понятно, что вычисляется в этой функции, а имя параметра (`dataframe`) не оставляет сомнений, какой аргумент ей следует передавать.

## Единственная ответственность

Принцип единственной ответственности касается функций не меньше, чем классов и модулей (о которых изначально и писал господин Мартин). Согласно этому принципу (в нашем случае) у функции должна быть единственная ответственность. То есть, она должна делать одну и только одну вещь. Один из самых веских доводов в пользу этого: если функция делает всего одну вещь, то и переписывать ее придется в единственном случае: если эту самую вещь придется делать по-новому. Также становится ясно, когда функцию можно удалить; если, внеся изменения где-то в другом месте, мы поймем, что единственная обязанность функции более не актуальна, то мы от нее просто избавимся.

Здесь лучше привести пример. Вот функция, делающая более одной «вещи»:

```
def calculate_and print_stats(list_of_numbers):
    sum = sum(list_of_numbers)
    mean = statistics.mean(list_of_numbers)
    median = statistics.median(list_of_numbers)
    mode = statistics.mode(list_of_numbers)

    print('-----------------Stats-----------------')
    print('SUM: {}'.format(sum)
    print('MEAN: {}'.format(mean)
    print('MEDIAN: {}'.format(median)
    print('MODE: {}'.format(mode)
```

А именно две: вычисляет набор статистических данных о списке чисел и выводит их в STDOUT. Функция нарушает правило: должна быть единственная конкретная причина, по которой ее, возможно, потребовалось бы изменить. В данном случае просматриваются две очевидные причины, по которым это понадобится: либо потребуется вычислять новую или иную статистику, либо потребуется изменить формат вывода. Поэтому данную функцию лучше переписать в виде двух отдельных функций: одна будет выполнять вычисления и возвращать их результаты, а другая – принимать эти результаты и выводить их в консоль. *Функцию (вернее, наличие у нее двух обязанностей) с потрохами выдает слово* ***and*** *в ее названии*.

Такое разделение также серьезно упрощает тестирование функции, а еще позволяет не только разбить ее на две функции в рамках одного и того же модуля, но даже разнести две эти функции в совершенно разные модули, если это уместно. Это дополнительно способствует более чистому тестированию и упрощает поддержку кода.

На самом деле, функции, выполняющие ровно две вещи, встречаются редко. Гораздо чаще натыкаешься на функции, делающие намного, намного больше операций. Опять же, из соображений удобочитаемости и тестируемости такие «многостаночные» функции следует дробить на однозадачные, в каждой из которых заключен единственный аспект работы.

## Докстроки

Казалось бы, все в курсе, что есть документ PEP-8 *Style Guide for Python Code*, где даются рекомендации по стилю кода на Python, но гораздо меньше среди нас тех, кто знает PEP-257 *Docstring Conventions*, в котором такие же рекомендации даются по поводу докстрок. Чтобы не пересказывать содержание PEP-257, отсылаю вас самих к этому документу – почитайте в свободное время. Однако, основные его идеи таковы:

- Для каждой функции нужна докстрока
- В ней следует соблюдать грамматику и пунктуацию; писать законченными предложениями
- Докстрока начинается с краткого (в одно предложение) описания того, что делает функция
- Докстрока формулируется в предписывающем, а не в описательном стиле

Все эти пункты легко соблюсти, когда пишешь функции. Просто написание докстрок должно войти в привычку, причем, старайтесь писать их прежде, чем приступать к коду самой функции. Если у вас не получается написать четкую докстроку, характеризующую функцию – это хороший повод задуматься, зачем вы вообще пишете эту функцию.

## Возвращаемые значения

Функции можно (и *следует*) трактовать как маленькие самодостаточные программы. Они принимают некоторый ввод в форме параметров и возвращают результат. Параметры, конечно, опциональны. *А вот возвращаемые значения обязательны с точки зрения внутреннего устройства Python*. Если вы даже попытаетесь написать функцию, которая не возвращает значения – не сможете.

Если функция даже не станет возвращать значения, то интерпретатор Python «принудит» ее возвращать `None`. Не верите? Попробуйте сами:

```
❯ python3
Python 3.7.0 (default, Jul 23 2018, 20:22:55)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> def add(a, b):
...   print(a + b)
...
>>> b = add(1, 2)
3
>>> b
>>> b is None
True
```

Как видите, значение `b` – по сути `None`.

Итак, даже если вы напишете функцию без инструкции return, она все равно будет что-то возвращать. И должна. В конце концов, это ведь маленькая программа, верно? Насколько полезны программы, от которых нет никакого вывода – и поэтому невозможно судить, верно ли выполнилась данная программа? Но самое важное – как вы собираетесь *тестировать* такую программу?

Каждая функция должна возвращать полезное значение, хотя бы ради тестируемости. Код, который я пишу, должен быть протестирован (это не обсуждается). Только представьте, каким корявым может получится тестирование вышеприведенной функции `add` (подсказка: вам придется перенаправлять ввод/вывод, после чего вскоре все пойдет наперекосяк). Кроме того, возвращая значение, мы можем выполнять сцепление методов и, следовательно, писать код вот так:

```
with open('foo.txt', 'r') as input_file:
    for line in input_file:
        if line.strip().lower().endswith('cat'):
            # ... делаем с этими строками что-нибудь полезное
```

Строка `if line.strip().lower().endswith('cat'):` работает, поскольку каждый из строковых методов (`strip()`, `lower()`, `endswith()`) в результате вызова функции возвращает строку.

Вот несколько распространенных доводов, которые вам может привести программист, объясняя, почему написанная им функция не возвращает значения:

> *«Она всего лишь [какая-то операция, связанная с вводом/выводом, например, сохранение значения в базе данных]. Здесь я не могу вернуть ничего полезного.»*

Не соглашусь. Функция может вернуть True, если операция завершилась успешно.

> *«Здесь мы изменяем один из имеющихся параметров, используем его как ссылочный параметр.»*

Здесь – два замечания. Во-первых, всеми силами старайтесь так не делать. Во-вторых, снабжать функцию каким-либо аргументом лишь для того, чтобы узнать, что она изменилась – в лучшем случае удивительно, а в худшем – попросту опасно. Вместо этого, как и при работе со строковыми методами, старайтесь возвращать новый экземпляр параметра, в котором уже отражены примененные к нему изменения. Даже если это не получается делать, поскольку создание копии какого-то параметра сопряжено с чрезмерными издержками, все равно можно откатываться к предложенному выше варианту «Вернуть `True`, если операция завершилась успешно».

> *«Мне нужно возвращать несколько значений. Нет такого единственного значения, которое в данном случае было бы целесообразно возвращать.»*

Этот аргумент немного надуманный, но мне доводилось его слышать. Ответ, разумеется, как раз в том, что автор и хотел сделать – но не знал как: *для возврата нескольких значений используйте кортеж*.

Наконец, самый сильный аргумент в пользу того, что полезное значение лучше возвращать в любом случае – в том, что вызывающая сторона всегда может с полным правом эти значения игнорировать. Короче говоря, возврат значения от функции – практически наверняка здравая идея, и крайне маловероятно, что мы таким образом что-нибудь повредим, даже в сложившихся базах кода.

## Длина функции

**Длина функции прямо сказывается на ее удобочитаемости и, следовательно, на поддержке**. Поэтому старайтесь, чтобы ваши функции оставались короткими.

Если функция соответствует Принципу единственной ответственности, то, вероятно, она будет достаточно краткой. Если она читая или идемпотентная (об этом мы поговорим) ниже – то, наверное, она также получится короткой. Все эти идеи гармонично сочетаются друг с другом и помогают писать хороший, чистый код.

Итак, что же делать, если ваша функция получилась слишком длинной? **РЕФАКТОРИТЬ!** Вероятно, вам приходится заниматься рефакторингом постоянно, даже если вы не знаете этого термина. Рефакторинг – это попросту изменение структуры программы, без изменения ее поведения. Поэтому, извлечение нескольких строк кода из длинной функции и превращение их в самостоятельную функцию – это один из типов рефакторинга. Оказывается, это еще и наиболее распространенный, и самый быстрый способ продуктивного укорачивания длинных функций. Поскольку вы даете этим новым функциям подходящие имена, получающийся у вас код гораздо проще читать. Я написал целую книгу о рефакторинге (на самом деле, я им постоянно занимаюсь), так что здесь вдаваться в детали не буду. Просто знайте, что, если у вас есть слишком длинная функция – то ее следует рефакторить.

## Идемпотентность и функциональная чистота

Заголовок этого раздела может показаться слегка устрашающим, но концептуально раздел прост. Идемпотентная функция при одинаковом наборе аргументов всегда возвращает одно и то же значение, независимо от того, сколько раз ее вызывают. Результат не зависит от нелокальных переменных, изменяемости аргументов или от любых данных, поступающих из потоков ввода/вывода. Следующая функция `add_three(number)` идемпотентна:

```
def add_three(number):
    """вернуть *число* + 3."""
    return number + 3
```

Независимо от того, сколько раз мы вызовем `add_three(7)`, ответ всегда будет равен 10. А вот другой случай – функция, не являющаяся идемпотентной:

```
def add_three():
    """Вернуть 3 + число, введенное пользователем."""
    number = int(input('Enter a number: '))
    return number + 3
```

Эта откровенно надуманная функция не идемпотентна, поскольку возвращаемое значение функции зависит от ввода/вывода, а именно – от числа, введенного пользователем. Разумеется, при разных вызовах `add_three()` возвращаемые значения будут отличаться. Если мы дважды вызовем эту функцию, то пользователь в первом случае может ввести 3, а во втором – 7, и тогда два вызова `add_three()` вернут 6 и 10 соответственно.

Вне программирования также встречаются примеры идемпотентности – например, по такому принципу устроена кнопка «вверх» у лифта. Нажимая ее в первый раз, мы «уведомляем» лифт, что хотим подняться. Поскольку кнопка идемпотентна, то сколько ее потом ни нажимать – ничего страшного не произойдет. Результат будет всегда одинаков.

## Почему идемпотентность так важна

Тестируемость и удобство в поддержке. Идемпотентные функции легко тестировать, поскольку они гарантированно, в любом случае вернут одинаковый результат, если вызвать их с одними и теми же аргументами. Тестирование сводится к проверке того, что при разнообразных вызовах функция всегда возвращает ожидаемое значение. Более того, эти тесты будут быстрыми: скорость тестов – важная проблема, которую часто обходят вниманием при модульном тестировании. А рефакторинг при работе с идемпотентными функциями – вообще легкая прогулка. Не важно, как вы измените код вне функции – результат ее вызова с одними и теми же аргументами всегда будет один и тот же.

## Что такое «чистая» функция?

В функциональном программировании функция считается чистой, если она, *во-первых*, идемпотентна, а *во-вторых* – не вызывает наблюдаемых **побочных эффектов**. Не забывайте: функция идемпотентна, если всегда возвращает один и тот же результат при конкретном наборе аргументов. Однако, это не означает, что функция не может влиять на другие компоненты – например, на нелокальные переменные или потоки ввода/вывода. Например, если бы идемпотентная версия вышеприведенной функции `add_three(number)` выводила результат в консоль, а лишь затем возвращала бы его, она все равно считалась бы идемпотентной, поскольку при ее обращении к потоку ввода/вывода эта операция доступа никак не влияет на значение, возвращаемое от функции. Вызов `print()` – это просто *побочный эффект*: взаимодействие с остальной программой или системой как таковой, происходящее наряду с возвратом значения.

Давайте немного разовьем наш пример с `add_three(number)`. Можно написать следующий код, чтобы определить, сколько раз была вызвана `add_three(number)`:

```
add_three_calls = 0

def add_three(number):
    """Вернуть *число* + 3."""
    global add_three_calls
    print(f'Returning {number + 3}')
    add_three_calls += 1
    return number + 3

def num_calls():
    """Вернуть, сколько раз была вызвана *add_three*."""
    return add_three_calls
```

Теперь мы выполняем вывод в консоль (это побочный эффект) и изменяем нелокальную переменную (другой побочный эффект), но, поскольку ни то, ни другое не влияет на значение, возвращаемое функцией, она все равно идемпотентна.

Чистая функция не оказывает побочных эффектов. Она не только не использует никаких «внешних данных» при расчете значения, но и не взаимодействует с остальной программой/системой, только вычисляет и возвращает указанное значение. Следовательно, хотя наше новое определение `add_three(number)` остается идемпотентным, эта функция уже не чистая.

В чистых функциях нет инструкций логирования или вызовов `print()`. При работе они не обращаются к базе данных и не используют соединений с интернетом. Не обращаются к нелокальным переменным и не изменяют их. **И не вызывают других не-чистых функций**.

Короче говоря, они не оказывают «жуткого дальнодействия». Они не изменяют каким-либо образом остальные части программы или системы. В императивном программировании, такие функции – самые безопасные. Они известны своей тестируемостью и удобством в поддержке; более того, поскольку они идемпотентны, тестирование таких функций гарантированно будет столь же быстрым, как и выполнение. Сами тесты также просты: не приходится подключаться к базе данных либо имитировать какие-либо внешние ресурсы, готовить стартовую конфигурацию кода, а по окончании работы не нужно ничего подчищать.

Честно говоря, идемпотентность и чистота очень желательны, но не обязательны. То есть, нам бы хотелось писать только чистые или идемпотентные функции, учитывая все вышеупомянутые их преимущества, но это не всегда возможно. Суть, однако, в том, чтобы приучиться писать код, естественным образом не допуская побочных эффектов и внешних зависимостей. Таким образом, каждую написанную нами строку кода станет проще тестировать, даже если не удастся обойтись только лишь чистыми или идемпотентными функциями.

## Выводы

Вот и все. Оказывается, секрет создания новых функций – никакой не секрет. Просто нужно придерживаться некоторых выверенных наилучших практик и железных правил.

**Главное правило** везде и во всех случаях писать отличный код. Или, как минимум, прилагать максимум усилий, чтобы не плодить в этом мире «плохой код».