Введение в функциональное программирование

Парадигмы и идиомы программирования

Парадигма программирования: подход к программированию основанный на наборе принципов или некоторой теории.

Разные парадигмы программирования, это разные способы мышления.

Идиомы программирования, это шаблоны использования возможностей языка программирования.

Парадигмы программирования:

Императивное программирование

(Как решить задачу?)

- Процедурное программирование
- Объектно-ориентированное

Декларативное программирование

(какую задачу решить?)

- Функциональное программирование
- Логическое программирование

Принцпи универасальности (возможность выполнить любые вычисления).

Императивное программирование

Императивное программирование это последовательность инструкций для компьютера (компьютера с архитектурой фон Неймана)

- содержимое памяти представляет собой состояние компьютера
- выражения изменяют переменные (т.е. переменные являются изменяемыми (mutable))

Для императивного программирования характерны:

- присваивания значений переменным
- управляющие структуры (такие как циклы, условные операторы и пр.)

In [1]:
```
# Пример поиска наибольшего делителя в императивном стиле:

def gcd_imp(x, y):
    r = 0
    while y > 0:
        r = x % y
        print(f'x: {x}, y: {y}, r:{r}')
        x = y
        y = r
    return x
```

In [2]:
```
gcd_imp(9702, 945)
```

```
x: 9702, y: 945, r:252
x: 945, y: 252, r:189
x: 252, y: 189, r:63
x: 189, y: 63, r:0
```

Out[2]:
```
63
```

Ключевые события в мире императивных языков программирования:

- Fortran (1957 г.)
- ALGOL (1960 г.)
- C (1972 г.)
- Ada (1983 г.)
- Java (1995 г.)

Функциональное программирование

Функциона́льное программи́рование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемости этого состояния.

Основано на Лямбда-исчислении (λ-исчисление), базирующемся на двух операциях: абстрации (операции, позволяющей конструировать функции) и аппликации (операции вызова функции).

Определяет программу как вызов фукнции (аппликация).

Принципы функционального программирования:

- вызов функции для одних и тех же значений параметров должен возвращать одинаковый результат (чистые (pure) функции)
- изменения контекста (внешних переменных) при вызове функции определяется как побочный эффект (нежелателен!)
- изменяемость (mutation) переменных нежелательна
- поддержка функций высших порядков (вызов функций для функций)

In [1]:
```
# Пример поиска наибольшего делителя в функциональном стиле:

def gcd_fnc(x, y):
    print(f'x: {x}, y: {y}')
    if y == 0:
        return x
    else:
        return gcd_fnc(y, x % y) # рекурсивный вызов функции
```

In [4]:
```
gcd_fnc(9702, 945)
```

```
x: 9702, y: 945
x: 945, y: 252
x: 252, y: 189
x: 189, y: 63
x: 63, y: 0
```

Out[4]: 63

In [3]:
```
factor = 2
```

In [4]:
```
def not_pure_f(n):
    return n*factor
```

In [5]:
```
not_pure_f(10)
```

Out[5]: 20

In [6]:
```
factor = 5
not_pure_f(10) # при идентичных параметрах функция возвращает другой результат
```

Out[6]: 50

Побочные эффекты

Изменение контекста (внешних переменных) при вызове функции определяется как побочный эффект.

In [9]:
```
# Источником побочных эффектов являются использование глобальных переменных:

factor = 0

def multiples(n):
    global factor
    factor = factor + 1
    return factor * n
```

In [10]:
```
factor
```

Out[10]: 0

In [11]:
```
multiples(2)
```

Out[11]: 2

In [12]:
```
factor
```

Out[12]: 1

In [13]:
```
multiples(2) # Резальтаты вызова функции с одним и тем же параметром различаются!
```

Out[13]: 4

In [55]:
```
multiples(2)
```

Out[55]: 10

In [14]:
```
# Источником побочных переменных так же является использование состояния объекта (или функции):

class Multiplier:
    def __init__(self):
        self.factor = 0

    def multiples(self, n):
        self.factor = self.factor + 1
        return self.factor * n
```

In [15]:
```
m = Multiplier()
```

In [16]:
```
m.multiples(2)
```

Out[16]: 2

In [17]:
```
m.multiples(2)
```

Out[17]: 4

In [18]:
```
# Источником побочных эффектов может быть ввод/вывод:

def read_byte(f):
    return f.read(1)
```

In [19]:
```
# источником побочных эффектов может быть использование датчика случайных чисел:

import random

def get_random(n):
    return random.randrange(1, n + 1)
```

Отрицательные моменты побочных эффектов:

- тяжелее тестировать программу
- тяжелее делать программы для параллельных вычислений (параллельной работы на нескольких ядрах / процессорах / компьютерах)
- тяжелее производить формальную верефикацию (проверку) программы

Но, можем ли мы писать программы без побочных эффектов? Или, как минимум, разделять части программы с побочным эффектом и без?

Функциональные языки программирования

Ключевые события в мире функциональных языков программирования:

- Lisp (1957 г.)
- ML (1973 г.)
- Haskell (1990 г.)

Мультипарадигменные функциональные языки программирования поддерживающие объектно-ориентированный подход:

- Ocaml
- F#
- Scala

Императивные языки с поддержкой возможностей функционального подхода:

- Python
- Ruby
- C#
- Java

Что делает язык функциональным или императивным?

Функции:

- функции "граждане первого класса" (functions as first-class citizens)
- функции высшего порядка (higher-order functions)
- замыкания (closures)
- функции без побочных эффектов (pure functions)
- рекурся, хвостовая рекурсия (recursion, tail recursion)

Данные:

- неизменяемые структуры данных (immutable data structures)
- вычисление результатов функций вместо явного хранения (и изменения) состояния программы (avoid changing-state)

Идиомы:

- итераторы, последовательности, ленивые вычисления, сопоставление с образцом,монады (iterators, sequences, lazy evaluation, pattern matching, monads....

Поддержка функционального программирования в Python

PRO:

- функции "граждане первого класса"
- лямбда-функции (анонимные функции)
- поддержка функциональных идом в стандартной библотеке: map/filter/reduce, itertools, operator
- генераторы могут использоваться для ленивых вычислений (вычислений по требованию)

CONTRA:

- невозможно разделить функции с побочным эффектом и без
- изменяемые переменные
- дорогостоящие операции копирования в памяти
- императивный стиль для циклов
- отсутствие оптимизации для хвостовой рекурсии ()
- нет синтаксиса для проверки шаблонов
- система типов базируется только на классах
- нет механизма перегрузки функций
- в стандартной библиотеке не реализован механизм построения композиции функций
- имеется только императивный механизм обработки ошибок

Функции в Python - граждане первого класса

Это означает, что функции можно динамически создвать и уничтожать, передвать их в другие функции, возвращать их как значения и так далее.

In [20]:
```
def add(a, b):
    return a + b
```

In [21]:
```
add(2, 3)
```

Out[21]: 5

In [22]:
```
f = add
f
```

Out[22]: <function __main__.add>

In [23]:
```
f(2, 3)
```

Out[23]: 5

In [7]:
```
# Применение: ветвление с использованием словарей:

def insert():
    print('Inserting ...')

def delete():
    print('Deleting ...')

def update():
    print('Updating ...')

def test():
    print('Testing ...')
```

In [5]:
```
command = input('Введите комнаду (i/d/u/t)')

if command == 'i':
    insert()
elif command == 'd':
    delete()
elif command == 'u':
    update()
elif command == 't':
    test()
else:
    print('Неправильное имя команды.')
```

```
Введите комнаду (i/d/u/t)ш
Неправильное имя команды.
```

In [8]:
```
com_d = dict(i=insert, d=delete, u=update, t=test)

command = input('Введите комнаду (i/d/u/t)')
com_d[command]()
```

```
Введите комнаду (i/d/u/t)d
Deleting ...
```

In [5]:
```
# лямбда функции:
add2 = lambda a, b: a + b
```

In [6]:
```
add2(2, 3)
```

Out[6]:
```
5
```

In [3]:
```
import collections
```

In [4]:
```
?collections.defaultdict
```

In [9]:
```
other = lambda: print('Неправильное имя команды')

com_dd = collections.defaultdict(lambda: other, com_d.items())

command = input('Введите комнаду (i/d/u/t)')
com_dd[command]()
```

```
Введите комнаду (i/d/u/t)x
Неправильное имя команды
```

Ограничения создания лямбда-функций:

- в теле функции может быть заключено только одно выражение
- значение выражения всега возвращается, как результат работы функции

Глобальные и локальные переменные

Глобальные переменные - это переменные, объявленные в программе вне функции. В Python глобальные персменные видны в любой части модуля, включая функции.

In [20]:
```
def func1(par):
    print('func begin')
    print(f'\tЗначение глобальной переменной glob1:{glob1}')
    glob2 = par
    print(f'\tЗначение ЛОКАЛЬНОЙ переменной glob2:{glob2}')
    print('func end')
```

In [21]:
```
glob1 = 10
glob2 = 5

print(f'Значение глобальной переменной glob1:{glob1}')
print(f'Значение глобальной переменной glob2:{glob2}')
func1(77)
print(f'Значение глобальной переменной glob2:{glob2}')
```

```
Значение глобальной переменной glob1:10
Значение глобальной переменной glob2:5
func begin
        Значение глобальной переменной glob1:10
        Значение ЛОКАЛЬНОЙ переменной glob2:77
func end
Значение глобальной переменной glob2:5
```

Переменной glob2 внутри функции присваивается значение, переданное при вызове функции. По этой причине создается новое имя glob2, которое является локальным. Все изменения этой переменной внутри функции не затронут значение одноименной глобальной переменной.

Локальные переменные - это переменные, которым внутри функции присваивается значение. Если имя локальной переменной совпадает с именем глобальной перемснной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной переменной не изменяется. Локальные персменные видны только внутри тела функции.

In [22]:
```
def func2():
    print('func begin')
    local = 77 # Локальная переменная
    glob = 25 # Локальная переменная
    print(f'\tЗначение ЛОКАЛЬНОЙ переменной local:{local}')
    print(f'\tЗначение ЛОКАЛЬНОЙ переменной glob:{glob}')
    print('func end')
```

In [24]:
```
glob = 10 # Глобальная переменная

print(f'Значение глобальной переменной glob:{glob}')
func2() # Вызываем функцию
print(f'Значение глобальной переменной вне функции glob:{glob}')
```

```
Значение глобальной переменной glob:10
func begin
        Значение ЛОКАЛЬНОЙ переменной local:77
        Значение ЛОКАЛЬНОЙ переменной glob:25
func end
Значение глобальной переменной вне функции glob:10
```

In [25]:
```
print(local) # Ошибка! Попытка обратится к локальной перменной func2 извне ее области видимости
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-fa1a7e4c8267> in <module>()
----> 1 print(local) # Ошибка! Попытка обратится к локальной перменной func2 извне ее области видимости

NameError: name 'local' is not defined
```

Для того чтобы значение глобальной переменной можно было изменить внутри функции, необходимо объявить переменную глобальной с помощью ключевого слова global.

In [28]:
```
def func2():
    print('func begin')
    local = 77 # локальная переменная
    global glob # объявление глобальной переменной внутри func2
    glob = 25 # ИЗМЕНЕНИЕ ГЛОБАЛЬНОЙ переменной внутри функции
    print(f'\tЗначение локальной переменной local:{local}')
    print(f'\tЗначение ГЛОБАЛЬНОЙ переменной glob:{glob}')
    print('func end')
```

In [29]:
```
glob = 10 # глобальная переменная

print(f'Значение глобальной переменной glob:{glob}')
func2() # вызов функции
print(f'Значение глобальной переменной glob:{glob}')
```

```
Значение глобальной переменной glob:10
func begin
        Значение локальной переменной local:77
        Значение ГЛОБАЛЬНОЙ переменной glob:25
func end
Значение глобальной переменной glob:25
```

Поиск идентификатора, используемого внутри функции, будет производиться в следующем порядке:

1. Поиск объявления идентификатора внутри функции (в локальной области видимости).
2. Поиск объявления идентификатора в глобальной области.
3. Поиск во встроенной области видимости (встроенные функции, классы и т. д.).

Побочные эффекты

Изменение контекста (внешних переменных) при вызове функции определяется как побочный эффект.

In [30]:
```
# Источником побочных эффектов являются использование глобальных переменных:

factor = 0

def multiples(n):
    global factor
    factor = factor + 1
    return factor * n
```

In [31]:
```
factor
```

Out[31]:
```
0
```

In [32]:
```
multiples(2)
```

Out[32]:
```
2
```

In [33]:
```
multiples(2) # Резальтаты вызова функции с одним и тем же параметром различаются!
```

Out[33]:
```
4
```

In [34]:
```
multiples(2)
```

Out[34]:
```
6
```

In [35]:
```
# Источником побочных переменных так же является использование состояния объекта (или функции):

class Multiplier:
    def __init__(self):
        self.factor = 0

    def multiples(self, n):
        self.factor = self.factor + 1
        return self.factor * n
```

In [36]:
```
m = Multiplier()
```

In [37]:
```
m.multiples(2)
```

Out[37]: 2

In [38]:
```
m.multiples(2)
```

Out[38]: 4

In [39]:
```
# Источником побочных эффектов может быть ввод/вывод:

def read_byte(f):
    return f.read(1)
```

In [40]:
```
# источником побочных эффектов может быть использование датчика случайных чисел:

import random

def get_random(n):
    return random.randrange(1, n + 1)
```

Отрицательные моменты побочных эффектов:

- тяжелее тестировать программу
- тяжелее делать программы для параллельных вычислений (параллельной работы на нескольких ядрах / процессорах / компьютерах)
- тяжелее производить формальную верефикацию (проверку) программы

Но, можем ли мы писать программы без побочных эффектов? Или, как минимум, разделять части программы с побочным эффектом и без?

Вложенные функции и замыкания

В Python одну функцию можно вложить в другую функцию, причем уровень вложенности неограничен. В этом случае вложенная функция получает свою собственную локальную область видимости и имеет доступ к идентификаторам внутри функции-родителя.

Вложенные функции могут создаваться в целях защиты их от всего что происходит вне функции (инкапсуляции):

In [41]:
```
def outer(num1):
    def inner_add(num2):  # не может быть вызвана вне outer
        return num1 + num2 # используется перменная, объявленная в outer
    num_res = inner_add(2)
    print(num1, num_res)
```

In [42]:
```
outer(10)
```

```
10 12
```

In [28]:
```
inner_add(10) # без специальных действий доступ к внутренней функции извне невозможен
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-b4995115c5ad> in <module>()
----> 1 inner_add(10) # без специальных действий доступ к внутренней функции извне невозможен

NameError: name 'inner_add' is not defined
```

Использование внутренних функций для избавления от повторяющихся фрагментов кода в одной функции (применение принципа DRY).

Don’t repeat yourself, DRY (рус. не повторяйся) — это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода, особенно в системах со множеством слоёв абстрагирования. Принцип DRY формулируется как: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы»

In [44]:
```
def process(file_name):
    def do_stuff(file_process):
        for line in file_process:
            print(line)

    if isinstance(file_name, str):
        with open(file_name, 'r') as f:
            do_stuff(f) # 1е исопльзование вложенной фукнции
    else:
        do_stuff(file_name) #2е использование вложенной функции
```

Вложенная функция получает свою собственную локальную область видимости и имеет доступ к идентификаторам внутри функции-родителя. Таким образом, помимо локальной, глобальной и встроенной областей видимости добавляется вложенная область видимости. При этом поиск идентификаторов вначале производится внутри вложенной функции, затем внутри функции-родителя, далее в функциях более высокого уровня и лишь потом в глобальной и встроенных областях видимости.

In [49]:
```
def f_lvl_1(x):
    print('f_lvl_1 begin')
    y = 2
    def f_lvl_2():
        print('\tf_lvl_2 begin')
        print(f'\tx:{x}')
#         print(f'\ty:{y}')
        y = 3 # создается ЛОКАЛЬНАЯ переменная во вложенной функции
        print(f'\ty:{y}')
        print('\tf_lvl_2 end')

    print(f'x:{x}')
    print(f'y:{y}')
    f_lvl_2()
    print(f'x:{x}')
    print(f'y:{y}')
    print('f_lvl_1 end')
```

In [50]:
```
f_lvl_1(10)
```

```
f_lvl_1 begin
x:10
y:2
        f_lvl_2 begin
        x:10
        y:3
        f_lvl_2 end
x:10
y:2
f_lvl_1 end
```

In [51]:
```
def f_lvl_1(x):
    print('f_lvl_1 begin')
    y = 2
    def f_lvl_2():
        print('\tf_lvl_2 begin')
        print(f'\tx:{x}')
        print(f'\ty:{y}') # ошибка: обращение к локальной перменной до ее объявления
        y = 3 # создается ЛОКАЛЬНАЯ переменная во вложенной функции
        print(f'\ty:{y}')
        print('\tf_lvl_2 end')

    print(f'x:{x}')
    print(f'y:{y}')
    f_lvl_2()
    print(f'x:{x}')
    print(f'y:{y}')
    print('f_lvl_1 end')
```

In [52]:
```
f_lvl_1(10)
```

```
f_lvl_1 begin
x:10
y:2
        f_lvl_2 begin
        x:10

---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-52-9d26c791bd3f> in <module>()
----> 1 f_lvl_1(10)

<ipython-input-51-8d0fc8b364d0> in f_lvl_1(x)
     12     print(f'x:{x}')
     13     print(f'y:{y}')
---> 14     f_lvl_2()
     15     print(f'x:{x}')
     16     print(f'y:{y}')

<ipython-input-51-8d0fc8b364d0> in f_lvl_2()
      5         print('\tf_lvl_2 begin')
      6         print(f'\tx:{x}')
----> 7         print(f'\ty:{y}') # ошибка: обращение к локальной перменной до ее объявления
      8         y = 3 # создается ЛОКАЛЬНАЯ переменная во вложенной функции
      9         print(f'\ty:{y}')

UnboundLocalError: local variable 'y' referenced before assignment
```

In [57]:
```
def f_lvl_1(x):
    print('f_lvl_1 begin')
    y = 2
    def f_lvl_2():
        print('\tf_lvl_2 begin')
        print(f'\tx:{x}')
        nonlocal y # доступ к локальной переменной функции родителя (вложенная область видимости)
        print(f'\ty:{y}') # ошибка: обращение к локальной перменной до ее объявления
        y = 3 # создается ЛОКАЛЬНАЯ переменная во вложенной функции
        print(f'\ty:{y} (после присовения)')
        print('\tf_lvl_2 end')

    print(f'x:{x}')
    print(f'y:{y}')
    f_lvl_2()
    print(f'x:{x}')
    print(f'y:{y}')
    print('f_lvl_1 end')
```

In [58]:
```
f_lvl_1(10)
```

```
f_lvl_1 begin
x:10
y:2
        f_lvl_2 begin
        x:10
        y:2
        y:3 (после присовения)
        f_lvl_2 end
x:10
y:3
f_lvl_1 end
```

Замыкания

In [36]:
```
def calculations(a, b):
    def add():
        return a + b
    return add
```

In [37]:
```
a = calculations(5, 4)
a
```

Out[37]:
```
<function __main__.calculations.<locals>.add>
```

In [38]:
```
a()
```

Out[38]:
```
9
```

Замыкание (англ. closure) в программировании — функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своём контексте.

In [39]:
```
def generate_power_func(n):
    def nth_power(x):
        return x**n
    return nth_power
```

In [40]:
```
n2 = generate_power_func(2)
n2
```

Out[40]:
```
<function __main__.generate_power_func.<locals>.nth_power>
```

In [42]:
```
n2(2), n2(3), n2(4)
```

Out[42]:
```
(4, 9, 16)
```

In [43]:
```
n3 = generate_power_func(3)
n3(2), n3(3), n3(4)
```

Out[43]:
```
(8, 27, 64)
```

Функции высшего порядка

Функции высших порядков — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.

In [44]:
```
# функции обратного вызова (callback functions)
def add(a, b):
    return a + b

def sub(a, b):
    return a - b
```

In [45]:
```
def apply_operation(operation, arg1, arg2):
    'operation - callback function; arg1, arg2 - arguments'
    return operation(arg1, arg2)
```

In [46]:
```
apply_operation(add, 2, 3)
```

Out[46]:
```
5
```

In [47]:
```
apply_operation(sub, 2, 3)
```

Out[47]:
```
-1
```

Создание частично подготовленных функций

In [48]:
```
import functools # модуль с функциями, используемыми в идеомах функционального программирования
```

In [49]:
```
help(enumerate)
```

```
Help on class enumerate in module builtins:

class enumerate(object)
 |  enumerate(iterable[, start]) -> iterator for index, value of iterable
 |
 |  Return an enumerate object.  iterable must be another object that supports
 |  iteration.  The enumerate object yields pairs containing a count (from
 |  start, which defaults to zero) and a value yielded by the iterable argument.
 |  enumerate is useful for obtaining an indexed list:
 |      (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(...)
 |      Return state information for pickling.
```

In [52]:
```
enum1 = functools.partial(enumerate, start=5)
enum1
```

Out[52]:
```
functools.partial(<class 'enumerate'>, start=5)
```

In [53]:
```
lines = list('abcd')
for lino, line in enum1(lines):
    print(lino, line)
```

```
5 a
6 b
7 c
8 d
```

In [23]:
```
reader = functools.partial(open, mode="rt", encoding="utf8")
writer = functools.partial(open, mode="wt", encoding="utf8")
```

Декораторы

Декоратор - это функция, которая принимает функцию или метод в качестве единственного аргумента и возвращает новую функцию или метод, включающую декорированную функцию или метод, с дополнительными функциональными возможностями.

Существуют предопределенные декораторы, например, ©property и @classmethod.

Предположим, что у нас имеется множество функций, выполняющих вычисления, и некоторые из них всегда должны возвращать не отрицательный результат. Мы могли бы добавить в каждую из таких функций инструкцию assert, но использование декораторов проще и понятнее.

In [54]:
```
def positive_result(function): # декоратор
    def wrapper(*args, **kwargs): # обертка
        result = function(*args, **kwargs) #
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

In [55]:
```
@positive_result
def discriminant(a, b, c):
    return (b ** 2) - (4 * a * c)
```

In [56]:
```
discriminant(1, 1, -1)
```

Out[56]: 5

In [57]:
```
discriminant(1, 1, 1)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-57-9a15f5e1ba5a> in <module>()
----> 1 discriminant(1, 1, 1)

<ipython-input-54-7ba388b573cd> in wrapper(*args, **kwargs)
      2     def wrapper(*args, **kwargs): # обертка
      3         result = function(*args, **kwargs) #
----> 4         assert result >= 0, function.__name__ + "() result isn't >= 0"
      5         return result
      6     wrapper.__name__ = function.__name__

AssertionError: discriminant() result isn't >= 0
```

In [58]:
```
@positive_result
def add3(a, b, c):
    return a + b + c
```

In [59]:
```
add3(1,1,-4)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-59-7e38993d9708> in <module>()
----> 1 add3(1,1,-4)

<ipython-input-54-7ba388b573cd> in wrapper(*args, **kwargs)
      2     def wrapper(*args, **kwargs): # обертка
      3         result = function(*args, **kwargs) #
----> 4         assert result >= 0, function.__name__ + "() result isn't >= 0"
      5         return result
      6     wrapper.__name__ = function.__name__

AssertionError: add3() result isn't >= 0
```

In [60]:
```
def positive_result2(function):
# декоратор, который обеспечивает, что wrapper будет носить имя исходной функции и ее строку документирования.
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0 !"
        return result
    return wrapper
```

In [61]:
```
@positive_result2
def discriminant2(a, b, c):
    return (b ** 2) - (4 * a * c)
```

In [62]:
```
discriminant2(1, 1, 1)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-62-bd92bb661561> in <module>()
----> 1 discriminant2(1, 1, 1)

<ipython-input-60-206236d056ec> in wrapper(*args, **kwargs)
      4     def wrapper(*args, **kwargs):
      5         result = function(*args, **kwargs)
----> 6         assert result >= 0, function.__name__ + "() result isn't >= 0 !"
      7         return result
      8     return wrapper

AssertionError: discriminant2() result isn't >= 0 !
```

Создание декоратора с параметрами

In [63]:
```
def bounded(minimum, maximum): # внешний генератор декоратора
    def decorator(function): # вунтренний генератор декоратора
        @functools.wraps(function)
        def wrapper(*args, **kwargs): # декоратор
            result = function(*args, **kwargs) # вызов декарируемой функции
            if result < minimum:
                return minimum
            elif result > maximum:
                return maximum
            return result
        return wrapper
    return decorator
```

In [64]:
```
@bounded(0, 100) # обрезает возвращения по нижней и верхней границе
def percent(amount, total):
    return (amount / total) * 100
```

In [66]:
```
percent(180, 100)
```

Out[66]:
```
100
```

In [67]:
```
percent(15, 100)
```

Out[67]:
```
15.0
```

In [68]:
```
@bounded(-100, 100) # обрезает возвращения по нижней и верхней границе
def div2(a, b):
    return a / b
```

In [72]:
```
div2(-700, 3)
```

Out[72]:
```
-100
```

Материалы для подготовки к следующей лекци

Прохоренок: Глава 11 "Пользовательские функции"

Саммерфильд: Глава 8 "Улучшенные приемы программирования" (разделы: Улучшенные приемы процедурного программирования; Функциональное программирование)

In [ ]:
```

```