**Функции и методы модуля math в Python.**

Чтобы выполнять математические операции, существуют специальные функции. Все они находятся в модуле math.

Все модули в Python разделяются на три вида:

Написанные на самом Python. Имеют расширение .py.

Написанные на C и загружаемые динамически. Такие модули могут иметь разные расширения, начиная привычным .dll и заканчивая менее распространенными .pyd, .so, .sl.

Модули, написанные на C, но имеющие связь с Python.

Для того, чтобы получить список последних, необходимо написать следующие строки кода.

import sys

print(sys.builtin\_module\_names)

**Функции представления чисел**

Эти функции выполняют такие действия, как отображение целой части числа, получение факториала, абсолютного значения, остатка от деления и так далее. Рассмотрим функции представления чисел.

**ceil() и floor()** — целая часть числа

С помощью этих функций можно или округлить число до целого или убрать десятичные знаки, оставив часть до запятой неизменной. Обе функции используют в качестве аргумента десятичное число и возвращают целое. Но различается принцип их работы.  С помощью функции ceil() число округляется в большую сторону. А функция floor() по сути также выполняет округление, но в меньшую сторону.

import math

number=8.10

# Используем функцию ceil() для округления в большую сторону. В выводе будет 9.

print(math.ceil(number))

# Используем функцию floor(), чтобы убрать знаки после запятой. В выводе будет 8.

print(math.floor(number))

**fabs()** — абсолютное значение

Эта функция используется для получения модуля числа. То есть, если оно отрицательное, то будет сконвертировано в положительное. Если оно положительное, то так и останется положительным. Функция не округляет число, поэтому если оно окажется дробным, то ничего не изменится.

В качестве аргумента функция принимает то число, которое необходимо конвертировать в положительное значение. Приведем пример кода для примера.

number = -8.10

print(math.fabs(number))

**factorial()** — функция факториала

В качестве аргумента этой функции используется целое положительное число, а возвращаемое значение – его факториал. Это такая функция, которая определяется на множестве неотрицательных целых чисел.

То есть, в данном случае, если перемножить каждое число, входящее в последовательность 1,2,3,4,5, то получится 120.

print(«факториала числа», math.factorial(number))

**fmod()** — остаток от деления

Скорее всего, вы уже знакомы с оператором деления по модулю. Недостаток оператора деления по модулю в том, что он может работать только с целыми числами. Поэтому в ситуациях, когда необходимо получить остаток от деления дробных чисел, используется функция fmod().

В качестве аргументов она принимает те числа, которые необходимо разделить и получить остаток. Сами числа прописываются через запятую.

print(math.fmod(-5.2, 2))

Видим, что здесь мы использовали как натуральные числа, так и отрицательные дробные. В этом плане функция очень гибкая.

**Функции возведения в степень и логарифма**

**exp()**

Но экспонента – это число, которое соответствует числу Эйлера (e) в определенной степени. Используется для представления очень больших чисел, которые невозможно правильно прочитать в своем привычном виде.

Функция exp() принимает целое число или число с плавающей точкой, а возвращает e в соответствующей степени.

**log()** — логарифм числа

Логарифмы используются в целом спектре видов человеческой деятельности. Так, они широко применяются для создания технических устройств и разработки приложений, которые помогают это делать.

**pow()** – степень числа

Операция возведения в степень в Python, как вы могли уже знать, выполняется с помощью оператора \*\*. print(5\*\*6)

Функция pow() работает аналогичным образом. Сначала в качестве первого аргумента в нее записывается основание – то число, которое нужно возводить в степень. А в качестве второго аргумента используется число, обозначающее степень, в которую следует возвести число.

print(pow(2, 1.5))

Заметьте, здесь мы не использовали объект math. Почему? Дело в том, что среди встроенных функций Python такая функция также есть. Работает она абсолютно так же, как и в модуле Math, но есть одно существенное отличие. Если мы рассматриваем функцию из библиотеки, то возвращаться будут числа с плавающей точкой, независимо от того, передаются ей целочисленные значения или нет. Зная эту особенность, можно гибко регулировать

Недостаток функции math.pow() в том, что если значение степени очень большое, будет выдаваться ошибка OverflowError.

**sqrt()** — квадратный корень числа

Math.sqrt() – это не единственная функция с помощью которой можно произвести операцию возведения в степень. Чтобы более глубоко понять принципы ее работы, надо сравнить также с другими способами получения квадратного корня из числа.

Первый вариант – использовать оператор возведения в степень. Если использовать степень 0,5, программа извлекает квадратный корень. Это очень удобный способ, которым можно пользоваться.

С помощью math.sqrt() можно получить квадратный корень из нуля и положительных чисел. Для этого необходимо в качестве аргумента использовать число, для которого нужно получить квадратный корень.

Недостаток этого метода в том, что он не поддерживает извлечение квадратного корня из отрицательных чисел. Чтобы такая возможность появилась, необходимо подключить другую библиотеку – cmath() и использовать метод cmath.sqrt() с числом в качестве аргумента.

import cmath

num = -25

sqrt = cmath.sqrt(num)

print(«Квадратный корень из числа » + str(num) + » это » + str(sqrt))

Особенность этой библиотеки в том, что она проводит математические вычисления с комплексными числами, которое и получается в результате получения квадратного корня из отрицательного значения.

Таким образом, функция math.sqrt() – далеко не самая мощная, и для продвинутого функционала требуется совсем другая библиотека.

**Тригонометрические функции**

Одно из характерных преимуществ модуля math() заключается в том, что он поддерживает тригонометрические функции. Поддерживаются все существующие на данный момент – синус, косинус, тангенс, арксинус, арктангенс и так далее.

**Функция преобразования углов**

библиотека math поддерживает возможность конвертации радиан в градусы и наоборот.

**Итераторы и выражения-генераторы**

На этом занятии мы с вами поговорим об итераторах и выражениях-генераторах. С генераторами списков мы уже сталкивались, например, когда создавали список вот таким образом:

a = [x\*\*2 **for** x **in** range(10)]

Выражения-генераторы очень похожи на генераторы списков и в синтаксисе отличаются только круглыми скобками:

b = (x\*\*2 **for** x **in** range(10))

Если мы теперь отобразим переменную b, то увидим:

<generator object <genexpr> at 0x0000020E8F429C80>

что эта переменная ссылается на объект-генератор.

***Генератор – это итератор, элементы которого можно перебирать (итерировать) только один раз.***

И в этом определении мы сталкиваемся с термином **итератор**:

***Итератор – это объект, который поддерживает функцию next() для перехода к следующему элементу коллекции.***

Наконец, последний термин, что нам пригодится, звучит так:

***Итерируемый объект – это объект, который позволяет поочередно обойти свои элементы и может быть преобразован к итератору.***

Самый распространенный итерируемый объект в Python – это список: но мы не можем его обойти с помощью итератора, используя функцию next: потому что список – это не итератор. Но мы любой итерируемый объект можем легко преобразовать в итератор с помощью функции iter: На выходе образуется объект-итератор для списка. Сохраним его в переменной: Теперь, элементы списка можно обойти с помощью этого итератора:

a = [1,2,3]

next(a)

iter(a)

it = iter(a)

next(it)

При первом ее вызове она возвратит первое значение списка a и изменит позицию итератора it, переместив его на следующий элемент. Поэтому при втором вызове: мы получим уже значение второго элемента и так до конца списка:

Если вызвать функцию next когда мы уже дошли до конца списка, то она возвратит ошибку:

Этот пример показывает как из итерируемого объекта можно получить итератор и с его помощью перебрать соответствующую коллекцию и только один раз. *Дойдя до конца списка, итератор it не может вернуться в начало и перебрать список еще раз.* Этот проход по элементам делается только один раз.

Теперь, возвращаясь к выражению-генератору:

b = (x\*\*2 **for** x **in** range(10))

переменную b можно воспринимать как итератор и перебирать список через функцию next:

Здесь также мы можем выполнить перебор всех значений только один раз, т.е. пройти список от начала до конца единожды: вернуться и повторить операцию здесь невозможно.

Итераторы очень удобно использовать в цикле for:

b = (x\*\*2 **for** x **in** range(10))

**for** i **in** b:

**print**(i, end=" ")

Здесь нам не нужно использовать функцию next для перехода к следующему значению. Это автоматически выполняет оператор in в for. Но использовать его можно только один раз. Если мы выполним цикл с этим же итератором еще раз:

**print**("**\n**new loop")

**for** i **in** b:

**print**(i, end=" ")

То в консоли ничего не отобразится. Здесь всегда следует помнить, что итераторы перебирают коллекцию только один раз.

Некоторые функции, такие как: sum, max, min позволяют работать непосредственно с итераторами. То есть, можно выполнять вот такие операции:

b = (x\*\*2 **for** x **in** range(10))

sum(b) КОНСОЛЬ

Будет вычислена сумма квадратов соответствующих значений.

И здесь возникает, наверное, давно назревший вопрос: зачем вообще нужны эти выражения-генераторы?

У этих объектов есть одно существенное преимущество по сравнению с обычными списками: они не хранят в памяти все значения сразу, а генерируют их по мере необходимости, то есть, при проходе к следующему значению. Например, если возникает необходимость оперировать очень большим списком то у компьютера попросту не хватит памяти и возникнет ошибка.

lst = list(range(1000000000000))

Но генераторы-выражения смогут спокойно перебирать значения такой коллекции например, в цикле for:

lst = (x **for** x **in** range(1000000000))

**for** i **in** lst:

**print**(i, end=" ")

**if** i > 100: **break**

И работать все это будет достаточно быстро, так как lst не хранит в памяти элементы, а вычисляет их налету в цикле for. Правда, из-за этого нельзя определить число элементов в генераторе при помощи функции len или получить доступ к его отдельному элементу по индексу:

a = (x **for** x **in** range(10, 20))

len(a)

a[2]

Если нужно выполнить эти операции, то любое выражение-генератор можно превратить в обычный список. Для этого используется функция list:

a = (x **for** x **in** range(10, 20))

b = list(a)

В результате, переменная b ссылается на список: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

с которым мы уже можем работать как со списком.

**Функции-генераторы**

Предположим, у нас имеется вот такая функция которая вычисляет средние арифметические для каждой из сумм: для одного значения, для двух, трех и так до N.

**def** getAllAverage(N):

    avs = []

    count = 0

    S = 0

**for** i **in** range(1,N+1):

        count += 1

        S += i

        avs.append( S/count )

**return** avs

При вызове этой функции:

**print**( getAllAverage(100) )

получим довольно внушительный список из чисел. В памяти он занимает:

**print**(getAllAverage(100).\_\_sizeof\_\_() )

904 байт. Конечно, это относительно немного. Но представьте, что будет, если N увеличить в десятки раз. Мы получим список, занимающий мегабайты памяти компьютера! Можно ли здесь как-то оптимизировать этот процесс по размеру занимаемой памяти?

Для подобных задач в Python предусмотрены так называемые **функции-генераторы**. Давайте для начала запишем простую функцию-генератор, а потом вернемся к нашей исходной задаче.

Пусть имеется функция, возвращающая обычный список из чисел. Превратим ее в функцию-генератор:

**def** f():

**return** list(range(10))

**print**( f() )

**def** f():

**for** x **in** range(10):

**yield** x

Смотрите, вот этот оператор yield возвращает значение x и замораживает текущее состояние функции. В результате, при ее повторном вызове цикл for начнется не с начала, а с той итерации, на которой был заморожен, то есть, перейдет к следующему значению x и оператор yield возвратит следующее значение.

Для вызова функции-генератора присвоим ее текущее состояние некоторой переменной: Эта переменная будет ссылкой на генератор, то есть, по сути, являться итератором: и мы уже к ней можем применить функцию next для перебора значений, которые будет возвращать функция:

s = f()

**print**( s )

**print**( next(s) )

**print**( next(s) )

Причем, мы здесь сразу получаем все преимущества итераторов, а именно – экономию занимаемой памяти данными.

Теперь вернемся к нашей исходной задаче и превратим нашу первую функцию в генератор: и вызовем ее:

**def** getAllAverage(N):

    count = 0

    S = 0

**for** i **in** range(1,N+1):

        count += 1

        S += i

**yield** S/count

it = getAllAverage(10)

**print**( next(it) )

В результате, мы последовательно будем получать вычисленные значения, не затрачивая дополнительной памяти на их хранение. Вот в этом и заключается преимущество таких функций с оператором yield.

#---------------------------------------------------

Вводится список городов в одну строчку через пробел. Необходимо создать итератор для этого списка и с помощью итератора вывести на экран в столбик первые два значения (названия городов).

**Sample Input:**

Москва Лондон Берлин Пекин

**Sample Output:**

Москва

Лондон

#---------------------------------------------------

# На вход программы поступают два целых числа a и b (a < b), записанные в одну строчку через пробел.  
# На их основе запишите генератор для формирования квадратов чисел в диапазоне [a; b].  
# Преобразуйте этот генератор в список (без использования операторов циклов) и выведите на экран.

#---------------------------------------------------

На вход программы поступают два целых числа a и b (a < b), записанные в одну строчку через пробел. Определите генератор, который бы выдавал модули целых чисел из диапазона [a; b]. В цикле выведите первые пять значений этого генератора. Каждое значение с новой строки. (Гарантируется, что пять значений имеются).

**Sample Input:** -3 3

**Sample Output:**

3

2

1

0

1

Имеется список из названий городов:

cities = ["Москва", "Ульяновск", "Самара", "Уфа", "Омск", "Тула"]

Необходимо записать генератор, который бы используя этот список, выдавал 1 000 000 наименований городов по циклу. То есть, дойдя до конца списка, возвращался в начало и повторял перебор. И так, для выдачи миллиона названий. Вывести на экран первые 20 наименований городов с помощью генератора в одну строчку через пробел.

#---------------------------------------------------

**Упаковка аргументов в Python**

Представим ситуацию. Мы пишем функцию summ(), которая, как нетрудно догадаться из названия, считаем сумму элементов. Сколько входных аргументов ей определить? Можно два, а можно три, в зависимости от требований алгоритма. А если нужно посчитать сначала сумму двух, а потом трех? Две разные функции писать не хочется. Очевидное решение, которое приходит в голову — передать в качестве аргумента массив элементов.

Чем плохо передавать в функцию просто list? А тем, что список — это изменяемая коллекция! Согласитесь, что неправильно функции менять свои аргументы? Это может привести к недопонимаю и ошибкам. Конечно, если речь не идет о разного рода функциях форматирования, цель которых именно изменить входной массив. Хотя даже в этой ситуации правильнее было бы вернуть отформатированный массив, а старый оставить как есть. Но в случае подсчета суммы нам точно нельзя изменять входные данные. Именно поэтому упаковка и распаковка аргументов идеально подходит в таких

Это работает следующим образом. Питон анализирует вызов функции summ и складывает - упаковывает все аргументы в кортеж elements. Кортеж выбран не случайно. Для того, чтобы обеспечить неизменяемость аргументов, о которой я говорил выше. Посмотрим на примере.

Одной из распространенных сфер, где применяются упаковка и распаковка - это параметры функций. Так, в определениях различных функций нередко можно увидеть, что они принимают такие параметры как \*args и \*\*kwargs. Термины args и kwargs — это соглашения по программированию на Python, в реальности вместо них можно использовать любые именования. \*args представляет параметры, которые передаются по позиции. А \*\*kwargs означает параметры, которые передаются по имени.

Оператор \* применяется с любым итерируемым объектом (например, кортежем, списком и строками). Тогда как оператор \*\* можно использовать только со словарями.

**\*args**

Оператор \* позволяет передать в функцию несколько значений, и все они будут упакованы в кортеж:

Здесь функция fun принимает кортеж значений. При вызове мы можем передать ей различное количество значений.

Благодаря такой возможности мы можем передавать в функцию переменное количество значений:

**Оператор \*\***

Оператор \*\* упаковывает аргументы, переданные по имени, в словарь. Имена параметров служат ключами.

**Как пользоваться \*args и \*\*kwargs**

Итак, мы знаем о том, что оператор «звёздочка» в Python способен «вытаскивать» из объектов составляющие их элементы. Знаем мы и о том, что существует два вида параметров функций. Вполне возможно, что вы уже додумались до этого сами, но я, на всякий случай, скажу об этом. А именно, \*args — это сокращение от «arguments» (аргументы), а \*\*kwargs — сокращение от «keyword arguments» (именованные аргументы).

Вот несколько советов, которые помогут вам избежать распространённых проблем, возникающих при работе с функциями, и расширить свои знания:

Используйте общепринятые конструкции \*args и \*\*kwargs для захвата позиционных и именованных аргументов.

Конструкцию \*\*kwargs нельзя располагать до \*args. Если это сделать — будет выдано сообщение об ошибке.

Остерегайтесь конфликтов между именованными параметрами и \*\*kwargs, в случаях, когда значение планируется передать как \*\*kwarg-аргумент, но имя ключа этого значения совпадает с именем именованного параметра.

Оператор \*можно использовать не только в объявлениях функций, но и при их вызове.

## **Оператор \***

Оператор \* используется для упаковки и распаковки итерируемых объектов, таких как списки или кортежи. При использовании перед итерируемым объектом, во время вызова функции, оператор \* распаковывает его. Элементы итерируемого объекта передаются в качестве аргумента функции:

**def** **sum\_of\_values**(a, b, c):

**return** a **+** b **+** c

values **=** [1, 2, 3]

result **=** sum\_of\_values(**\***values)

**print**(result)

*# 6*

В этом примере оператор \* используется, чтобы распаковывать список значений и передавать его элементы в качестве аргументов функции sum\_of\_values.

Также оператор \* можно использовать, чтобы распаковывать итерабельные переменные. Это позволяет присваивать их отдельным переменным:

my\_list **=** [1, 2, 3, 4]

a, b, **\***c **=** my\_list

**print**(a, b, c)

*# 1 2 [3, 4]*

В этом примере оператор \* используется не во время вызова функции, а для распаковки списка my\_list в отдельные переменные a, b и c.

Оператор \* можно использовать для распаковки итерируемого списка в новый список или кортеж:

my\_list **=** [1, 2, 3]

new\_list **=** [**\***my\_list, 4, 5, 6]

**print**(new\_list)

*# [1, 2, 3, 4, 5, 6]*

my\_tuple **=** (1, 2, 3)

new\_tuple **=** (**\***my\_tuple, 4, 5, 6)

**print**(new\_tuple)

*# (1, 2, 3, 4, 5, 6)*

В этих примерах оператор \* используется для распаковки итераций my\_list и my\_tuple в новые списки и кортежи с добавлением дополнительных значений.

## **Оператор \*\***

Оператор \*\* используется для упаковки и распаковки словарей. При использовании перед словарем во время вызова функции оператор \*\* упаковывает пары ключ-значение словаря в аргументы ключевых слов, которые могут быть переданы в функцию:

**def** **print\_details**(name, age):

**print**(**f**"Name: {name}")

**print**(**f**"Age: {age}")

details **=** {"name": "John", "age": 30}

print\_details(**\*\***details)

*# Name: John*

*# Age: 30*

В этом примере оператор \*\* используется для распаковки словаря details и передачи его пар ключ-значение в качестве аргументов ключевых слов в функцию print\_details.

Оператор \*\* также можно использовать для создания словаря из последовательности пар ключ-значение:

dict1 **=** {"a": 1, "b": 2}

dict2 **=** {"c": 3, "d": 4}

combined\_dict **=** {**\*\***dict1, **\*\***dict2}

**print**(combined\_dict)

*# {"a": 1, "b": 2, "c": 3, "d": 4}*

В приведенном выше коде определяются два словаря dict1 и dict2. А оператор \*\* используется для распаковки их пар ключ-значение и объединения в один словарь combined\_dict.

**Замыкания**

Замыкание (closure) представляет функцию, которая запоминает свое лексическое окружение даже в том случае, когда она выполняется вне своей области видимости.

Технически замыкание включает три компонента:

внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные и параметры - лексическое окружение

переменные и параметры (лексическое окружение), которые определены во внешней функции

вложенная функция, которая использует переменные и параметры внешней функции

Для определения замыканий в Python применяются локальные функции: Здесь функция outer определяет локальную переменную n - это и есть лексическое окружение для внутренней функции:

Внутри функции outer определена внутренняя функция - локальная функция inner, которая обращается к своему лексическому окружению - переменной n - увеличивает ее значение на единицу и выводит на консоль:

Эта локальная функция возвращается функцией outer:

В программе вызываем функцию outer и получаем в переменную fn локальную функцию inner:

Переменная fn и представляет собой замыкание, то есть объединяет две вещи: функцию и окружение, в котором функция была создана. И несмотря на то, что мы получили локальную функцию и можем ее вызывать вне ее окружающей функции, в которой она определена, тем не менее она запомнила свое лексическое окружение и может к нему обращаться и изменять, что мы увидим по консольному выводу:

**Применение параметров**

Кроме внешних переменных к лексическому окружению также относятся параметры окружающей функции. Рассмотрим использование параметров:

Здесь внешняя функция - multiply возвращает функцию, которая принимает число и возвращает число.

Вызов функции multiply() возвращает локальную функцию inner:

Эта функция запоминает окружение, в котором она была создана, в частности, значение параметра n. Кроме того, сама принимает параметр и возвращает произведение параметров n и m.

В итоге при вызове функции multiply определяется переменная fn, которая получает локальную функцию inner и ее лексическое окружение - значение параметра n:

В данном случае параметр n равен 5.

Также можно было бы сократить этот код с помощью лямбд:

# Декораторы функций и замыкания

***Декоратор в Python – это функция, которая в качестве аргумента принимает другую функцию и расширяет ее функционал без изменения последней.***

Алгоритм Евклида для поиска НОД двух натуральных чисел a и b. И в случае вычитаний, он выглядел так:

**def** getNOD(a, b):

**while** a != b:

**if** a > b: a-= b

**else**: b -= a

**return** a

Далее, мы хотим создать тест для проверки скорости работы этой функции. Реализуем этот тест в виде декоратора. Это будет выглядеть так:

**import** time

**def** testTime(fn):

**def** wrapper(\*args):

        st = time.time()

        fn(\*args)

        dt = time.time() - st

**print**(f"Время работы: {dt} сек")

**return** wrapper

Смотрите, здесь внутри функции testTime (нашего декоратора) объявлена еще одна функция wrapper (обертка), внутри которой уже и происходит вызов некой функции fn. Далее замеряется время ее работы и информация выводится в консоль. И в конце сам декоратор возвращает ссылку на функцию wrapper.

Почему все реализовано именно так? Здесь вот эта вложенная функция как раз и расширяет функционал для fn, не меняя ее саму. А благодаря вот этому оператору return мы имеем возможность вызывать эту обертку (wrapper) так:

test1 = testTime(getNOD)

test1(100000, 2)

То есть, декоратор testTime возвращает ссылку на wrapper, который в свою очередь будет вызывать getNOD и определять время выполнения именно этой функции при заданных аргументах.

Конечно, мы могли бы записать декоратор и без wrapper: Но это дает меньше гибкости в дальнейшем, в частности, при его вызове нам каждый раз нужно будет указывать функцию:

**def** testTime(fn, \*args):

        st = time.time()

        fn(\*args)

        dt = time.time() - st

**print**(f"Время работы: {dt} сек")

testTime(getNOD, 100000, 2)

Первый вариант выглядит гораздо естественнее. Там мы имеем возможность создавать новое имя для функции и, затем, вызывать ее как новый, независимый объект.

Теперь, когда мы в деталях разобрались с работой декоратора, вызовем его для быстрого алгоритма Евклида. Он реализуется вот таким образом:

**def** getFastNOD(a, b):

**if** a < b: a,b = b,a

**while** b: a,b = b, a%b

**return** a

test1 = testTime(getNOD)

test2 = testTime(getFastNOD)

test1(100000, 2)

test2(100000, 2)

Все выглядит довольно элегантно и красиво. Кстати, функцию wrapper можно записать в еще более универсальном виде, добавив необязательный список возможных именованных параметров:

**def** testTime(fn):

**def** wrapper(\*args, \*\*kwargs):

        st = time.time()

        fn(\*args, \*\*kwargs)

        dt = time.time() - st

**print**(f"Время работы: {dt} сек")

**return** wrapper

Так мы сможем вызывать и тестировать на скорость работы любые функции.

В Python есть один интересный синтаксис использования декораторов. Запишем нашу функцию testTime в самом верху программы:

Далее мы можем применить ее к любой функции, например, getNOD. Для этого перед ней записывается:

@testTime

getNOD(100000,2)

И теперь при ее вызове будет запускаться указанный декоратор. А вот вызов второй функции:

getFastNOD(100000, 2)

никак не связан с декоратором – это просто вычисление НОД для двух чисел.

Вот так элегантно, просто и быстро в Python можно расширять функционал отдельных функций, не меняя их содержимого.

Конечно, если функция fn возвращает какое-либо значение, то это легко предусмотреть в обертке. И, теперь, мы можем получить еще и результат работы функции getNOD:

**def** testTime(fn):

**def** wrapper(\*args, \*\*kwargs):

        st = time.time()

        res = fn(\*args, \*\*kwargs)

        dt = time.time() - st

**print**(f"Время работы: {dt} сек")

**return** res

**return** wrapper

res = getNOD(100000,2)

**print**( res )

Вот что из себя представляют декораторы функций, замыкания и вот так они реализуются в Python.

Объявите функцию с именем get\_sq, которая вычисляет площадь прямоугольника по двум параметрам: width и height - ширина и высота прямоугольника. И возвращает результат (сама ничего на экран не выводит). То есть, функция имеет сигнатуру:

def get\_sq(width, height): ...

Определите декоратор func\_show для этой функции, который отображает результат на экране в виде строки (без кавычек):

**Sample Input:** 8 11

**Sample Output:** Площадь прямоугольника: 88

#---------------------------------------------------

На вход программы поступает строка из целых чисел, записанных через пробел. Напишите функцию get\_list, которая преобразовывает эту строку в список из целых чисел и возвращает его. Определите декоратор для этой функции, который сортирует список чисел по возрастанию. Результат сортировки должен возвращаться при вызове декоратора.

Вызовите декорированную функцию get\_list и отобразите полученный отсортированный список lst

**Sample Input:** 8 11 -5 4 3 10

**Sample Output:** -5 3 4 8 10 11

#---------------------------------------------------

#---------------------------------------------------