

Математические функции в Python

Информатика. Язык программирования Python в курсе информатики. Все классы.

Для работы с математическими функциями нужно импортировать библиотеку **math:**

import math

После этого к функциям из этой библиотеки можно обращаться следующим образом:

math.имя\_функции(…)

ФУНКЦИИ В БИБЛИОТЕКЕ MATH

|  |  |
| --- | --- |
| ceil(x) | Возвращает округленное x как ближайшее целое значение типа int, большее или равное x (округление "вверх"). |
| fabs(x) | Возвращает абсолютное значение (модуль) числа x. В Python есть встроенная функция abs, но она возвращает модуль числа с тем же типом, что число, здесь же всегда float abs (fabs). |
| factorial(x) | Возвращает факториал целого числа x, если x не целое возбуждается исключение ValueError. |
| floor(x) | В противоположность ceil(x) возвращает округленное x как ближайшее целое значение типа int, меньшее или равное x (округление "вниз"). |
| frexp(x) | Представляет число в экспоненциальной записи x=m∗2n и возвращает мантиссу m (действительное число, модуль которого лежит в интервале от 0.5 включительно до 1 не включительно) и порядок n (целое число) как пару чисел (m, n). Если x=0, то возвращает (0.0, 0) |
| fsum(iterable) | Возвращает float сумму от числовых элементов итерируемого объекта. |
| isinf(x) | Проверяет, является ли float объект x плюс или минус бесконечностью, результат соответственно True или False. |
| isnan(x) | Проверяет, является ли float объект x объектом NaN (not a number). |
| ldexp(x, i) | Возвращает значение x∗2i, то есть осуществляет действие, обратное функции frexp(x). |
| modf(x) | Возвращает дробную и целую часть float числа. Оба результата сохраняют знак исходного числа x и представлены типом float. |
| trunc(x) | Возвращает целую часть числа x в виде int объекта. |

СТЕПЕННЫЕ И ЛОГАРИФМИЧЕСКИЕ ФУНКЦИИ

|  |  |
| --- | --- |
| exp(x) | Возвращает ex. |
| log(x[, base]) | При передаче функции одного аргумента x, возвращает натуральный логарифм x (логарифм по основанию e = 2.7182…). При передаче двух аргументов, второй берется как основание логарифма. |
| log10(x) | Возвращает десятичный логарифм x. |
| pow(x, y) | Возвращает x в степени y. В отличие от операции \*\* приводит оба аргумента к типу float. |
| sqrt(x) | Квадратный корень (square root) из x. |

ТРИГОНОМЕТРИЧЕСКИЕ ФУНКЦИИ

|  |  |
| --- | --- |
| acos(x) | Возвращает арккосинус x, в радианах. |
| asin(x) | Возвращает арксинус x, в радианах. |
| atan(x) | Возвращает арктангенс x, в радианах. |
| atan2(y, x) | Возвращает atan(y/x), в радианах. Результат лежит в интервале [-&pi;, &pi;]. Вектор, конец, которого задается точкой (x, y) образует угол с положительным направлением оси x. Поэтому эта функция имеет более общее назначение, чем предыдущая. Например и atan(1), и atan2(1, 1) дадут в результате pi/4, но atan2(-1, -1) это уже -3\*pi/4. |
| cos(x) | Возвращает косинус x, где x выражен в радианах. |
| hyp(x, y) | Возвращает sqrt(x\*\*2+y\*\*2). Удобно для вычисления гипотенузы (hyp) и длины вектора. |
| sin(x) | Возвращает синус x, где x выражен в радианах. |
| tan(x) | Возвращает тангенс x, где x выражен в радианах. |

РАДИАНЫ В ГРАДУСЫ И НАОБОРОТ

|  |  |
| --- | --- |
| degrees(x) | Конвертирует значение угла x из радиан в градусы. |
| radians(x) | Конвертирует значение угла x из градусов в радианы. |

ПРИМЕР ПРОГРАММЫ С МАТЕМАТИЧЕСКИМИ ФУНКЦИЯМИ

from math import \*   # Импортируем библиотеку math

def my\_function(x):
    x = fabs(x) # Наша функция будет четной
    y = sqrt(x) # Извлекаем корень квадратный
    y = exp(sin(y) + 1) # Берем синус, прибавляем 1, а затем это выражение сразу в показатель экспоненты
    return y

print(my\_function(2))

А можно написать эту функцию так (в функциональном стиле):

from math import \*

def my\_function(x):
    return exp(sin(sqrt(fabs(x))) + 1)

print(my\_function(2))

**Функции** в Python создаются с помощью инструкции def. Это действие создает объект функции и присваивает ему имя, которое становится ссылкой на объект-функцию.

Пример определения функции:

def MyFirstFunction(arg1, arg2, arg3):
    return arg1 + arg2 + arg3

Здесь MyFirstFunction — это **имя функции**, используемое как для ее определения, так и для ее вызова.

После заголовка с новой строки и с отступом следуют выражения **тела функции**.

В теле функции присутствует (их может быть несколько) инструкция **return** (может и не быть), которая прерывает выполнение функции возвращает значение в основную ветку программы.

ВЫЗОВ ФУНКЦИИ И ВОЗВРАТ ЗНАЧЕНИЯ

Пример вызова функции:

x = MyFirstFunction(10, 2, 5)
В данном случае, если бы в функции не было инструкции **return**, то в основную программу ничего бы не возвращалось, и переменной x числовое значение не присваивалось бы. Если функция не возвращает значение явно с помощью команды return, то автоматически возвращается значение None.

ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИЮ

При описании функции после имени в скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми, но они обязательно должны быть. Далее идет двоеточие, обозначающее окончание заголовка функции.

При вызове функции в скобках указывается нужное количество переменных или выражений, значения которых будут переданы функции в качестве параметров. Если у функции нет параметров, скобки (пустые) при вызове все равно нужны.

ПРИМЕРЫ

Допустим, необходимо вычислить число сочетаний из n элементов по k, равное n!/(k!(n−k)!), для чего необходимо вычисление факториалов трех величин: n, k и n-k. Для этого можно сделать три цикла, что приводит к увеличению размера программы за счет трехкратного повторения похожего кода. Вместо этого лучше сделать одну функцию, вычисляющую факториал любого данного числа n и трижды использовать эту функцию в своей программе. Соответствующая функция может выглядеть так:

def factorial(n):
   result = 1
   for i in range(2, n + 1):
       result \*= i
   return result

Этот код должен идти до того места, где мы захотим воспользоваться функцией **factorial**.

Первая строчка этого примера является описанием нашей функции. factorial — идентификатор, то есть имя нашей функции. После идентификатора в круглых скобках идет список параметров, которые получает наша функция. Список состоит из перечисленных через запятую идентификаторов параметров. В нашем случае список состоит из одной величины n. В конце строки ставится двоеточие.

Далее идет тело функции, оформленное в виде блока, то есть с отступом. Внутри функции вычисляется значение факториала числа n и оно сохраняется в переменной result. Функция завершается инструкцией
return result
которая завершает работу функции и возвращает значение переменной result. Инструкция return может встречаться в произвольном месте функции, ее исполнение завершает работу функции и возвращает указанное значение в место вызова. Если функция не возвращает значения, то инструкция return используется без возвращаемого значения, также в функциях, не возвращающих значения, инструкция return может отсутствовать.

Теперь мы можем использовать нашу функцию несколько раз. В этом примере мы трижды вызываем функцию factorial для вычисления трех факториалов: factorial(n), factorial(k), factorial(n-k).

n = int(input())
k = int(input())
print(factorial(n) // (factorial(k) \* factorial(n - k)))

Мы также можем, например, объявить функцию **binomial**, которая принимает два целочисленных параметра n и k и вычисляет число сочетаний из n по k:

def binomial(n, k):
   return factorial(n) // (factorial(k) \* factorial(n - k))

Тогда в нашей основной программе мы можем вызвать функцию binomial для нахождения числа сочетаний:

print(binomial(n, k))

Вернемся к задаче нахождения наибольшего из двух или трех чисел. Функцию нахождения максимума из двух чисел можно написать так (впрочем, в Python есть и встроенная функция max):

def max(a, b):
    if a > b:
        return a
   else:
       return b

Теперь мы можем реализовать функцию **max3**, находящую максимум трех чисел:

def max3(a, b, c):
   return max(max(a, b), c)

Функция **max3** дважды вызывает функцию max для двух чисел: сначала, чтобы найти максимум из a и b, потом чтобы найти максимум из этой величины и c.



Полиморфизм функций в Python

Информатика. Язык программирования Python в курсе информатики. Все классы.

Python – язык с динамической типизацией, полиморфизм в нем проявляется повсюду.

Фактически все операции в языке Python являются полиморфическими: вывод, извлечение элемента, оператор \* и многие другие. В языке Python именно объекты определяют синтаксический смысл операции. Например, оператор \* – это всего лишь указание для обрабатываемых объектов. Смысл операции зависит от типов обрабатываемых объектов.

**Единственная функция может автоматически применяться к целой категории типов объектов.** Пока объекты поддерживают ожидаемый интерфейс (или протокол), функция сможет обрабатывать их. **Если объект, фактически передаваемый функции, поддерживает ожидаемые методы и операторы выражений, он будет совместим с логикой функции.**

Если функции будут переданы объекты, которые не поддерживают ожидаемый интерфейс, интерпретатор обнаружит ошибку при выполнении и автоматически возбудит исключение. Отсутствие проверки на наличие ошибок в программном коде можно заменить последующим тестированием программы. Добавив такую проверку, мы ограничим область применения нашей функции, так как она сможет работать только с теми типами объектов, которые мы заранее предусмотрели.

Это важнейшее отличие философии языка Python от языков программирования со статической типизацией, таких как C++ и Java: программный код на языке Python не делает предположений о конкретных типах данных. В противном случае он сможет работать только с теми типами данных, которые ожидались на момент его написания, и он не будет поддерживать объекты других совместимых типов, которые могут быть созданы в будущем. Проверку типа объекта можно выполнить с помощью таких средств, как встроенная функция type, но в этом случае программный код потеряет свою гибкость.

Конечно, такая модель полиморфизма предполагает необходимость тестирования программного кода на наличие ошибок, так как из-за отсутствия объявлений типов нет возможности с помощью компилятора выявить некоторые виды ошибок на ранней стадии. Однако в обмен на незначительное увеличение объема отладки мы получаем существенное уменьшение объема программного кода, который требуется написать, и существенное увеличение его гибкости.

В языке Питон отсутствует полиморфизм функций в модели С++. Разные функции должны иметь разные имена.

Адекватной заменой является полиморфизм по типу объекта, для которого вызывается функция, что требует перенесения функции в объект (будет изучаться в уроках по ООП).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Будет ли ошибочным следующий код:

def f(x):
    return x \* 2

f(1) \* f("2")

f(3.) \* f(f(4))

РЕШЕНИЕ

Нет, не будет.

Благодаря полиморфизму функций по типам параметров с каждым из аргументов будет приключаться своё: числа будут умножаться на два, а строка — повторяться дважды.

2. Будет ли ошибочным следующий код:

def f(x):
    return x

def f(x):
    return 2 \* x

def f(x):
    return 3 \* x

print(f(5))

РЕШЕНИЕ

Нет, не будет. При этом первые две функции, поскольку их имена более недоступны в пространстве имен, фактически перестают существовать.

При вызове print(f(5)) будет вызвана последняя функция, результат вычислений будет равен 15

.



Области видимости переменных в Python

Информатика. Язык программирования Python в курсе информатики. Все классы.

Всякий раз, когда в программе используется некоторое имя, интерпретатор создает, изменяет или отыскивает это имя в **пространстве имен**.

Под термином **область видимости** подразумевается пространство имен для конкретного идентификатора (имени).

В языке Python **область видимости тесно связана с операциями присваивания**.

Имена появляются в тот момент, когда им впервые присваиваются некоторые значения, и прежде чем имена смогут быть использованы, им необходимо присвоить значения. Поскольку имена не объявляются заранее, интерпретатор **Python по местоположению операции присваивания связывает имя с конкретным пространством имен**. Место, где выполняется присваивание, определяет пространство имен, в котором будет находиться имя, а следовательно, и область его видимости.

По умолчанию все имена, значения которым присваиваются внутри функции, ассоциируются с пространством имен этой функции. Это означает, что:

* Имена, определяемые внутри инструкции def, видны только программному коду внутри инструкции def. К этим именам нельзя обратиться за пределами функции.
* Имена, определяемые внутри инструкции def, не вступают в конфликт с именами, находящимися за пределами инструкции def, даже если и там и там присутствуют одинаковые имена. Имя X, которому присвоено значение за пределами данной инструкции def (например, в другой инструкции def или на верхнем уровне модуля), полностью отлично от имени X, которому присвоено значение внутри инструкции def.

В любом случае **область видимости переменной (где она может использоваться) всегда определяется местом, где ей было присвоено значение**, и никакого отношения не имеет к месту, откуда была вызвана функция или осуществлена операция изменения объекта.

Значения переменным могут быть присвоены в трех разных местах, соответствующих трем разным областям видимости:

* Если присваивание переменной выполняется внутри инструкции def, переменная является *локальной* для этой функции.
* Если присваивание производится в пределах объемлющей инструкции def, переменная является *нелокальной* для этой функции.
* Если присваивание производится за пределами всех инструкций def, она является *глобальной* для всего файла.

Правила видимости имен

Программный код, находящийся вне функций, находится **на верхнем уровне модуля**. Функции же образуют вложенные пространства имен (области видимости), которые ограничивают доступ к используемым в них именам, благодаря чему имена внутри функций не вступают в конфликт с именами за их пределами (внутри модуля или внутри других функций).

Функции образуют локальную область видимости, а модули – глобальную. Эти две области взаимосвязаны между собой следующим образом:

* **Каждый вызов функции создает новую локальную область видимости**.
* **Операция присваивания создает локальные имена, если они не были объявлены глобальными или нелокальными.** По умолчанию все имена, которым присваиваются значения внутри функции, помещаются в локальную область видимости (пространство имен, ассоциированное с вызовом функции).

Порядок сопоставления имен:

1. локальные
2. нелокальные (внутри объемлющей инструкции def)
3. глобальные (в пространстве имен модуля)
4. встроенные (предопределенные имена в модуле builtins)

Схема разрешения имен в языке Python иногда называется правилом LEGB, название которого состоит из первых букв названий областей видимости.

ПРИМЕР ПЕРЕКРЫТИЯ ОБЛАСТЕЙ ВИДИМОСТИ

Здесь инструкция присваивания X = 99 создает глобальную переменную с именем X (она видима из любого места в файле), а инструкция X = 88 создает локальную переменную X (она видима только внутри инструкции def).

Даже при том, что обе переменные имеют имя X, области видимости делают их различными. Таким образом, области видимости функций позволяют избежать конфликтов имен в программах и превращают функции в самостоятельные элементы программ.

Доступ на присваивание к нелокальным именам

Любые операции присваивания, выполняемые внутри функции, классифицируют имена как локальные.

**Если какая-либо из разновидностей операции присваивания выполняется в пределах инструкции def, имя становится локальным по отношению к этой функции.**Если необходимо присвоить значение имени верхнего уровня в модуле, который вмещает функцию, это имя необходимо объявить внутри функции глобальным с помощью инструкции global. Если необходимо присвоить значение имени, которое находится в объемлющей инструкции def, это имя необходимо объявить внутри функции с помощью инструкции nonlocal.

Следует также заметить, что операции непосредственного изменения объектов не рассматривают имена как локальные – это свойственно только операциям присваивания. Например, если имени L присвоен список, определенный на верхнем уровне в модуле, то такая инструкция, как L.append(X), внутри функции не будет классифицировать имя L как локальное, тогда как инструкция L = X — будет. В первом случае происходит изменение объекта списка, на который указывает L, а не самого имени L, – список L будет найден в глобальной области видимости, как обычно, и Python изменит этот список, без необходимости объявления имени global (или nonlocal). Этот пример должен помочь явственнее ощутить различия между именами и объектами: операция, изменяющая объект, совсем не то, что операция присваивания объекта имени.

Инструкция **global** сообщает интерпретатору, что функция будет изменять одно или более глобальных имен. global объявляет глобальные переменные без присваивания им значений.

*Например:*

X = 88 # Глобальная переменная X
def func():

    global X

    X = 99 # Глобальная переменная X: за пределами инструкции def

func()

print(X) # Выведет 99

Итак, инструкция global обеспечивает возможность изменения переменных в модуле из функций. Существует также родственная ей инструкция nonlocal, которая обеспечивает возможность изменения переменных в объемлющих функциях.



Локальные и глобальные переменные в Python

Информатика. Язык программирования Python в курсе информатики. Все классы.

Внутри функции можно использовать переменные, объявленные вне этой функции

def f():
    print(a)

a = 1
f()

Здесь переменной a присваивается значение 1, и функция f печатает это значение, несмотря на то, что выше функции f эта переменная не инициализируется. Но в момент вызова функции f переменной a уже присвоено значение, поэтому функция f может вывести его на экран.

Такие переменные (объявленные вне функции, но доступные внутри функции) называются **глобальными**.
Но если инициализировать какую-то переменную внутри функции, использовать эту переменную вне функции не удастся.

Например:

def f():
    a = 1

f()
print(a)

Получим **NameError: name ‘a’ is not defined**. Такие переменные, объявленные внутри функции, называются локальными. Эти переменные становятся недоступными после выхода из функции.

Интересным получится результат, если попробовать изменить значение глобальной переменной внутри функции:

def f():
    a = 1
    print(a)
a = 0
f()
print(a)

Будут выведены числа 1 и 0. То есть несмотря на то, что значение переменной a изменилось внутри функции, то вне функции оно осталось прежним! Это сделано в целях “защиты” глобальных переменных от случайного изменения из функции (например, если функция будет вызвана из цикла по переменной i, а в этой функции будет использована переменная i также для организации цикла, то эти переменные должны быть различными). То есть если внутри функции модифицируется значение некоторой переменной, то переменная с таким именем становится локальной переменной, и ее модификация не приведет к изменению глобальной переменной с таким же именем.

Более формально: интерпретатор Питон считает переменную локальной, если внутри нее есть хотя бы одна инструкция, модифицирующая значение переменной (это может быть оператор =, += и т.д., или использование этой переменной в качестве параметра цикла for,) то эта переменная считается локальной и не может быть использована до инициализации. При этом даже если инструкция, модицифицирующая переменную никогда не будет выполнена: интерпретатор это проверить не может, и переменная все равно считается локальной. Пример:

def f():
    print(a)
    if False:
        a = 0
a = 1
f()

Возникает ошибка: **UnboundLocalError: local variable ‘a’ referenced before assignment**. А именно, в функции f идентификатор a становится локальной переменной, т.к. в функции есть команда, модифицирующая переменную a, пусть даже никогда и не выполняющийся (но интерпретатор не может это отследить). Поэтому вывод переменной a приводит к обращению к неинициализированной локальной переменной.

Чтобы функция могла изменить значение глобальной переменной, необходимо объявить эту переменную внутри функции, как глобальную, при помощи ключевого слова **global**:

def f():
    global a
    a = 1
    print(a)
a = 0
f()
print(a)

В этом примере на экран будет выведено 1 1, так как переменная a объявлена, как глобальная, и ее изменение внутри функции приводит к тому, что и вне функции переменная будет доступна.

Тем не менее, лучше не изменять значения глобальных переменных внутри функции. Если функция должна поменять какую-то переменную, то как правило это лучше сделать, как значение, возвращаемое функцией.

Если нужно, чтобы функция вернула не одно значение, а два или более, то для этого функция может вернуть кортеж из двух или нескольких значений:

return (a, b)

Тогда результат вызова функции тоже нужно присваивать кортежу:

(n, m) = f(a, b)



Граф вызовов функций

Информатика. Язык программирования Python в курсе информатики. Все классы.

Граф вызовов функции отображает связь функций друг с другом по логике их вызовов.

Заметим, что он может быть составлен не только для функций, но и для процедур, то есть для любых подпрограмм. В языках Си и Python процедуры синтаксически не отличаются от функций и даже не выделяются в отдельное понятие. Т.е. все подпрограммы — функции, только некоторые могут ничего не возвращать.



По данному графу видно, что функция A непосредственно вызывает функцию B, которая, в свою очередь, вызывает функцию C(), а та пользуется функцией D().

После вызова функцией А функции B, она сама погружается в ожидание завершения подпрограммы.

Таким образом, функции A() важен результат B(), и она не может без него продолжить свое выполнение.

Если мы отобразим на графе обратный ход, как возвращается исполнение к функциям вместе с результатами вычислений с более глубоких уровней вызова, то получится вот так:



ПРИМЕР НА ЯЗЫКЕ СИ

//заголовки функций:
int A();
int B();
int C();
int D();

//тела функций
int A()
{
    printf("A() started\n");
    int result = B() + 1;
    printf("A() finished\n");
    return result;
}

int B()
{
    printf("B() started\n");
    int result = C() + 1;
    printf("B() finished\n");
    return result;
}

int C()
{
    printf("C() started\n");
    int result = D() + 1;
    printf("C() finished\n");
    return result;
}

int D()
{
    printf("D() started\n");
    int result = 1; //никого не вызывает - это последняя по глубине вызовов функция
    printf("D() finished\n");
    return result;
}

ПРИМЕР НА ЯЗЫКЕ PYTHON

def A():
    print('A() started')
    result = B() + 1
    print('A() finished')
    return result

def B():
    print('B() started')
    result = C() + 1
    print('B() finished')
    return result

def C():
    print('C() started')
    result = D() + 1
    print('C() finished')
    return result

def D():
    print('D() started')
    result = 1 # никого не вызывает - это последняя по глубине вызовов функция
    print('D() finished')
    return result

ЧТО ВЕРНЕТ ФУНКЦИЯ A() В МЕСТО СВОЕГО ВЫЗОВА?

4

**Это значение будет зависеть от значений, возвращенных остальными функциями.**

В КАКОМ ПОРЯДКЕ БУДУТ НАПЕЧАТАНЫ X() STARTED И X() FINISHED ДЛЯ A, B, C, D?

A() started
B() started
C() started
D() started
D() finished
C() finished
B() finished
A() finished

Причина обратного порядка завершения в том, что возврат из функций происходит именно в обратном порядке. Пока не завершится D, функция C простаивает и ожидает от нее результат. Так же и другие функции.



Стек вызовов

Информатика. Язык программирования Python в курсе информатики. Все классы.



**Стек вызовов** (call stack) — стек, хранящий информацию для возврата управления из подпрограмм (функций) в программу или подпрограмму (при вложенных или рекурсивных вызовах).

При вызове подпрограммы в стек заносится **адрес возврата** — адрес в памяти следующей инструкции приостанавливаемой программы, а управление передается подпрограмме. При последующем вложенном или рекурсивном вызове в стек заносится очередной адрес возврата и так далее.

При возврате из подпрограммы адрес возврата снимается со стека, и управление передается на следующую инструкцию приостановленной программы (или подпрограмм