# Функции. Возвращение значений из функции

## План урока:

1. Сравнение математических функций и функций в Python. Чистые функции и побочные эффекты.
2. Возвращаемые значения.
3. Разница между возвратом из функции и печатью на экран. Использование результата вычисления в качестве вспомогательного значения. Время жизни возвращаемого значения.
4. Множественные точки возврата из функции. return как выход с любого уровня вложенности.
5. Использование отладчика для трассировки функций. Наблюдение за изменением области видимости.
6. Функции, не возвращающие результат.
7. (дополнительный материал) Условия-стражники.
8. Возврат множественных значений.

## Аннотация

*В этом уроке мы обсудим, почему функции называются функциями, чем они похожи и чем отличаются от функций в математике. И главное, мы разберемся с основной целью вызова функций - возвратом во внешнюю программу результата вычисления.*

## Связь между математическими функциями и функциями в Python

На прошедших занятиях мы разобрали в деталях, как ведут себя переменные и объекты. Теперь, наконец, вернемся от изучения тонкостей работы с переменными — к функциям. Вы уже умеете делать достаточно сложные функции, но эти функции пока плохо взаимодействуют с остальной программой.

Большая часть функций, которые вы писали до сих пор, выводили значение на экран или изменяли внешние переменные. Однако значение, выведенное на экран, полезно только для человека. Сама программа никак не может его использовать. Если бы функции могли выводить результаты своей работы только на экран, то их было бы почти невозможно комбинировать.

Один сложный и небезопасный способ использовать результат вычислений функции вы уже видели: для этого мы применяли либо глобальные переменные, либо изменяемые контейнеры (например, списки). Эти контейнеры передаются в функцию как аргумент, а она изменяет их содержимое.

Но есть способ гораздо проще и правильнее: **возвращаемое значение**. Каждая функция может не только выполнять действия, но и выдавать какой-то результат, который потом можно использовать в программе — например записать в переменную. Вы еще не делали таких функций, но уже не раз пользовались ими. Попробуйте вспомнить несколько примеров.

Если вы посмотрите примеры таких функций, вы увидите среди них много математических... функций.

Функция в математике — это такое преобразование, которое из одного значения или набора значений делает другое значение. Например, функция квадратного корня делает из числа его корень. Функция $f(a,b) = (a+b)^2$ делает из двух чисел квадрат их суммы. Фактически единственное важное свойство математической функции заключается в том, что каждому набору аргументов она сопоставляет значение, и каждый раз вычисление функции на одних и тех же аргументах дает один и тот же результат. Сколько бы раз вы ни вычисляли корень из шестнадцати, каждый раз будете получать четыре.

Заметьте, что математическая функция не обязана работать с числами. Например, в математике можно встретить такую функцию — число перестановок букв в слове. Это функция, которая принимает аргументом строку, а возвращает число. Или функцию пересечения множеств, которая берет в качестве аргументов два множества и возвращает тоже множество.

Чем функции в программировании похожи и чем отличаются от функций в математике?

Функция в языке Python — это некоторый алгоритм, который выполняется каждый раз одинаково. Для большинства функций возвращаемое значение, как и в случае математической функции, зависит только от аргументов. У функций в Python, как вы знаете, также есть список аргументов: иногда одно значение, иногда несколько, а иногда этот список аргументов и вовсе пустой, как у функции `input`. У функций также есть возвращаемое значение, это значение всегда ровно одно. Может показаться, что некоторые функции ничего не возвращают (как функция `print`), но на самом деле они тоже возвращают значение — `None`.

В некоторых других языках программирования существуют функции, которые не возвращают значения, их называют "процедуры".

Отличия заключаются в том, что функция в Python может зависеть не только от аргументов, но и от внешних причин. Что для функции может быть внешними причинами? Например, действия пользователя. Функция `input` помимо пустого списка аргументов получает ввод с клавиатуры пользователя. Некоторые функции читают файлы на жестком диске или в Интернете, а значит, их результат зависит от содержимого файла или веб-страницы. Есть функции, работа которых зависит от текущего времени. Есть функции, зависящие от датчика случайных чисел. Кроме того, как вы уже видели, работа некоторых функций зависит от внешних (глобальных) переменных. Еще одна особенность функций в Python — они могут изменять что-то снаружи функции: менять глобальные переменные, выводить текст на экран, записывать что-то в файлы.

Таким образом функции можно разделить на функции с побочными эффектами и без них.

Функции без побочных эффектов и не использующие внешних источников данных (их еще называют «чистые функции») ведут себя в точности как математические функции. Их результат зависит только от аргументов, а вызов таких функций никак не влияет на ход остальной программы. Большая часть известных вам встроенных функций Python ведет себя так: `math.sqrt`, `math.cos`, `abs`, `int`, `str`, `len`, `min`, `max`.

Функции с побочными эффектами — это, как правило, функции, предназначенные для общения с «внешним миром»: с пользователем, файлами на жестком диске, другими программами или серверами в интернете. Пока что вы знаете две такие функции, встроенные в язык: `input` и `print`.

Побочные эффекты часто используются для изменения аргумента — как в методах `sort` и `reverse`, которые изменяют данные в списке.

Любая функция, которая использует `input()` или `print()`, тоже имеет побочные эффекты. И любая функция, которая изменяет значения внешних переменных, тоже имеет побочные эффекты. Так что писать функции с побочными эффектами вы уже умеете, а вот как делать чистые функции, и как с ними работать, мы начнем говорить сейчас.

## Возвращаемые значения

Для того, чтобы функция вернула значение, используется оператор `return`. Использовать его очень просто. Давайте напишем функцию `doubleIt`, которая удваивает значение:

```
def doubleIt(x):
    return x * 2
```

Эта функция получила число x в качестве аргумента, умножила его на 2 и вернула результат в основную программу. Значением, которое функция вернула, можно воспользоваться. Например, мы можем посчитать длину окружности с использованием этой функции:

```
radius = 3
length = doubleIt(3.14) * radius
```

Когда интерпретатор дойдет до `doubleIt(3.14)`, начнет исполняться код функции. Когда он дойдет до слова `return`, значение, которое указано после `return`, будет подставлено в программе вместо вызова функции. Можно сказать, что сразу после того как функция досчитается, вычисление превратится в такое:

```
length = 6.28 * radius
```

**Заметьте!** Функция `doubleIt` ничего не выводит на экран. Она выполняет вычисления и сообщает их не пользователю, как мы делали раньше, а другой части программы.

Если нам потребуется не просто вернуть удвоенное число, а ещё и вывести его на экран, то лучше не добавлять `print` внутрь функции. Ведь если вы добавите `print` в функцию, то уже никак не сможете вызвать эту функцию в «тихом» режиме, чтобы она ничего не печатала. Вместо этого лучше сначала вернуть результат, а потом уже распечатать его во внешней программе. Вот так:

```
print(doubleIt(3.14))
```

Или, если вам нужно еще как-то использовать вычисленное значение, можно завести специальную переменную, хранящую результат вычисления.

```
double_pi = doubleIt(3.14)
print(double_pi)
length = double_pi * radius
```

Функция удваивания числа, конечно, совершенно бесполезна. А вот функции вычисления длины и площади окружности вполне могут пригодиться.

**Задачи:** Длина и площадь круга, Корни квадратного уравнения

А теперь давайте рассмотрим чуть более сложный пример — вычисление суммы элементов списка:

```
def my_sum(arr):
    result = 0
    for element in arr:
        result += element
    return result
print(my_sum([1, 2, 3, 4]))
```

Здесь мы используем очень распространенный способ написания функций: создаем вспомогательную переменную, а затем возвращаем её значение.

Но мы ведь недавно говорили, что локальные переменные живут только внутри функции. Если переменная `result` исчезнет, то почему результат — число 10 — никуда не пропадает?

Вспомните, что мы говорили в начале урока про отличие между объектами (значениями) и переменными. Объект может существовать даже когда нет переменной, в которой он хранится. Когда мы записываем число в `result`, мы фактически создаем объект числа и даем ему временное имя `result`. Потом, когда мы пишем `return result` мы возвращаем не переменную. Как и в большинстве конструкций языка (кроме, пожалуй, присваивания), вместо переменной подставляется её значение: таким образом, мы возвращаем объект «число 10». У этого объекта нет имени, что не мешает функции `print` использовать его и напечатать 10 на экране.

Однако если ни программист, ни программа не имеют возможности пользоваться объектом, то этот объект становится не нужен. После того как функция `print` отработала, доступ к результату вычисления пропал, ведь мы никуда не сохранили этот результат. На объект «число 10» нет ссылок, поэтому интерпретатор может его «выкинуть». Это называется «сборка мусора»: Python автоматически избавляется от всех объектов, которые невозможно использовать.

**Задача:** Мелочь оставь себе

**Домашние задачи:**

- Морской бой .
- Коровы и быки. Секретный уровень (дополнительная задача).

# Множественные точки возврата из функции

Часто бывают ситуации, когда, в зависимости от входных данных, нужно выполнить различные наборы команд. Например, когда мы считаем модуль, в случае отрицательного числа нужно взять число со знаком минус, а в случае неотрицательного числа (положительное или ноль) мы берем само число. Давайте запишем это в виде функции `my_abs(x)`.

```
def my_abs(x):
    if x >= 0:
        result = x
    else:
        result = -x
    return result
```

Но если вдуматься, то зачем ждать конца функции, когда мы уже вычислили результат и совершили все необходимые действия? Давайте завершим функцию сразу, ведь `return` именно для этого создан: он не только возвращает значение функции, но и возвращает нас из функции. После вызова оператора `return` выполнение кода функции заканчивается. Раз так, давайте немного упростим программу. Сначала мы перенесем `return` к тому месту, где результат получен:

```
def my_abs(x):
    if x >= 0:
        result = x
        return result
    else:
        result = -x
        return result
```

А теперь можно заметить, что переменная `result` лишняя: когда вы подставляете значение `result`, вместо `result` вы легко можете подставить сразу результат.

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

Заметьте, что любую функцию можно написать с одним-единственным оператором `return`, но часто использовать несколько точек выхода из функции просто удобно. Давайте рассмотрим еще один случай, для которого несколько точек выхода позволяют упростить программу.

**Задача:** Среднее значение-2.

**Дополнительная задача:** Уравнения степени не выше второй .

Напишем небольшую программу, в которой есть функция `longCalculation()`. Она выполняет очень долгое вычисление (на современном компьютере — за несколько секунд) и возвращает результат.

Также мы написали функцию `longOnlyOnce()`, которая запускает долгую функцию и выводит её результат на экран. Но эта функция должна работать долго только в первый раз; для этого она записывает результат первого запуска в специальную переменную. И если в этой переменной уже записан ответ, во второй раз она просто не запускает долгое вычисление. Однако в функцию `longOnlyOnce` вкралась ошибка, которую вы должны исправить.

```
def fib(n):
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


def longCalculation():
    return fib(33)


def longOnlyOnce():
    if savedResult:
        print(savedResult)
    else:
        savedResult = longCalculation()
        print(savedResult)


longOnlyOnce()
```

Исправьте ошибку и проверьте правильность выполнения программы. Надо иметь терпение дождаться ответа, данная программа может выполняться достаточно долго.

# Матрицы

В некоторых задачах этого урока вам встретится важный математический объект, который называется "матрица". **Матрица** - это прямоугольная табличка, заполненная какими-то значениями, обычно - числами. В математике вам встретится множество различных применений матриц, поскольку с их помощью многие задачи гораздо проще сформулировать и решить. Мы же сконцентрируемся на том, как хранить матрицу в памяти компьютера.

В первую очередь от матрицы нам нужно уметь получать элемент в i-ой строке и j-ом столбце. Чтобы этого добиться обычно поступают так: заводят список строк матрицы, а каждая строка матрицы сама по себе тоже является списком элементов. То есть мы получили список списков чисел. Теперь, чтобы получить элемент нам достаточно из списка строк матрицы выбрать i-ую и из этой строки взять j-ый элемент.

Давайте заведем простую матрицу `M` размера 2x3 (2 строки и 3 столбца) и получим элемент на позиции (1, 3). Обратите внимание, что сначала нумерация строк и столбцов идёт с единицы, а не с нуля. И по договоренности среди математиков сначала всегда указывается строка, лишь затем - столбец. Элемент на i-ой строке, j-ом столбце матрицы `M` в математике обозначается $M_{i,j}$. Итак:

```
matrix  = [[1, 2, 3],
           [2, 4, 6]]
print(matrix[0][2])  # => 3
```

`matrix` - это вся матрица, `matrix[0]` - это список значений в первой строке, `matrix[0][2]` - элемент в третьем столбце в этой строке.

Чтобы перебрать элементы матрицы приходится использовать двойные циклы. Например, выведем на экран все элементы матрицы, перебирая их по столбцам:

```
for col in range(3):
    for row in range(2):
        print(matrix[row][col])
```

## Возврат из глубины функции

Наша следующая программа будет проверять, есть ли в матрице элемент, отличающийся от искомого не больше, чем на число `eps`. Матрица записывается как список списков. Мы предполагаем, что наша функция будет работать с большими матрицами, поэтому нам не хочется тратить лишнее время на проверку. Мы будем прекращать поиск как только нашли подходящий элемент. Давайте для начала разберемся, как бы мы действовали без множественных операторов `return`.

```
def matrix_has_close_value(matrix, value, eps):
    found = False
    for row in matrix:
        for cell in row:
            if abs(cell - value) <= eps:
                found = True
                break
        if found:
            break
    if found:
        return True
    else:
        return False
```

Как видите, нам приходится прилагать некоторые усилия, чтобы выйти из нескольких уровней вложенности. Каждый уровень вложенности — это дополнительное препятствие на пути к завершению функции. Ему мешают не только циклы, как в этом примере, но и условные операторы.

Перепишем теперь функцию с учетом того, что как только мы нашли элемент, мы уже знаем, что ответ — `True` (т.е. элемент содержится в матрице). А если мы закончили перебор элементов и так и не нашли ни одного элемента, то ответ `False`.

```
def matrix_has_close_value(matrix, value, eps):
    for row in matrix:
        for cell in row:
            if abs(cell - value) <= eps:
                return True
    return False
```

Хотя `return False` и не заключен ни в какое условие, выполняется он только когда элемент не найден. Если элемент найден, мы сразу выходим из функции, и до этой строки просто не доходим. Оператор `return` очень удобен, когда нужно выйти из глубины функции.

## Отладка

Давайте проследим последовательность выполнения команд в отладчике. Запустим следующий код:

```
def matrix_has_close_value(matrix, value, eps):
    for row in matrix:
        for cell in row:
            if abs(cell - value) <= eps:
                return True
    return False

table = [[1,2,3], [4,5,6], [7,8,9]]
result = matrix_has_close_value(table, 3.75, 0.5)
print(result)
```

Будем выполнять инструкции по одной с помощью команды **step into**. Сначала будет определена функция `matrix_has_close_value`. На этом шаге интерпретатор Python в неё ещё не заходит. Затем мы определяем переменную `table`, и она появляется на вкладке **stack data** в разделе глобальных переменных. Затем мы запускаем функцию и попадаем внутрь неё. Обратите внимание, появились локальные переменные `matrix`, `value` и `eps`. Причем `matrix` указывает ровно туда же, куда указывает `table` (у них одинаковый идентификатор `0x<...>`). На прошлых занятиях мы обсуждали, что объект, который был подставлен в аргумент получает новое имя — в нашем случае `matrix`, — но старое имя тоже остается, потому что переменная `table` находится в глобальной области видимости.

Если продолжить выполнение, пройдет несколько итераций цикла, пока мы не дойдем до момента, когда элемент в ячейке (число 4) с точностью до `eps(=0.5)` совпадёт со значением `value(=3.75)`, которое мы ищем. На этом месте мы перейдем на строчку `return True` и следующим шагом выйдем из функции.

После того, как мы вышли из функции, в разделе локальных переменных появляется `<return value>: True`. Следующим шагом возвращённое значение будет записано в переменную `result`, которая появится в списке глобальных. Затем эта переменная будет подставлена в функцию `print` и напечатана.

Что мы видим? Во-первых, у нас есть возможность пронаблюдать, что происходит с объектами и переменными. Во-вторых, слово `return`, действительно, моментально завершает функцию и перемещает нас к строке, в которой мы вызвали функцию. И, конечно, возвращает результат.

Для пошаговой отладки мы пользовались командой **step into**. Она так называется, поскольку заставляет отладчик зайти в функцию (если это возможно) и показывать мельчайшие детали исполнения кода функции.

Бывает, что заходить в функцию незачем: потому что вы в ней уверены и ничего не собираетесь менять. Или потому что в ней слишком много команд, которые не просмотреть. Или потому что решили отложить её отладку на будущее.

На этот случай в отладчике предусмотрена команда **step over statement**. При её применении функция всё равно будет вызвана, но отладчик не покажет вам подробности исполнения. Он просто дождется, пока функция завершит свою работу, вернет результат и переведет вас на следующую строчку после вызова функции.

Также иногда бывает нужно выйти из функции. Например, когда вы зашли в неё отладчиком случайно или когда вы все в ней уже понимаете. В таких случаях вам поможет команда **step out**. Она продолжает выполнение, пока не встретит `return`, выполняет его и оказывается за пределами функции.

**Задание. Поэкспериментируйте с отладчиком:**

- С помощью пошаговой отладки узнайте, что произойдет, если вместо второго аргумента подставить значение, которого нет в матрице?

- Сравните списки локальных и глобальных переменных до того, как мы входим в функцию. Что происходит со списком глобальных переменных, когда мы входим в функцию? Что происходит со списком локальных переменных?

- Узнайте, как будут изменяться переменные при входе в функцию (следите за идентификаторами объектов), если завести еще одну глобальную переменную с именем `matrix`?

- Что произойдет, если, находясь в функции, присвоить что-нибудь переменной `table`?

- А что если, находясь в функции, попробовать изменить какой-нибудь элемент в `table`?

- Разберитесь с отличиями между `step into`, `step out` и `step over`. В каких ситуациях они работают, в каких не работают?

Не бойтесь испортить функцию! Изменяйте её как угодно, лишь бы можно было понять, как ведет себя интерпретатор Python в сложных ситуациях.

## Что можно возвращать из функции

В функциях, которые не возвращают значение, тоже можно использовать `return`. Если написать `return` без аргументов, то функция просто сразу завершит свою работу (без `return` функция завершает работу, когда выполнит последнюю команду).

Как мы уже упоминали, результат возвращает любая функция, даже если в ней нет слова `return`. Результатом такой функции будет `None`. Если в функции использован `return` без аргументов, это фактически эквивалентно `return None`.

**Дополнительная задача:** Секретные материалы.

---

***Дополнительный материал: условия-стражники.***

Такой преждевременный возврат из функции часто используют как условие-стражник (`guard condition`), который «не пропускает интерпретатор» при невыполнении условий. Пример: мы пишем функцию-калькулятор, которая принимает строку с арифметическим выражением и печатает результат на экране. Но если в функцию передана строка `quit`, то мы должны выйти из функции, не пытаясь ничего вычислить. Один способ записать это:

```
def calculate(str):
    if str != 'quit':
        # … сложная логика ...
        print(result)
```

Но бывает удобно убрать один, а иногда и несколько уровней вложенности, написав условие-стражник:

```
def calculate(str):
    if str == 'quit':
        return
    # … сложная логика ...
    print(result)
```

Теперь, если мы ввели `quit`, то до выполнения сложной логики программа просто не дойдет.

***(Конец дополнительного материала)***

---

## Возврат нескольких значений

В задаче про корни квадратного уравнения у нас уже возникала необходимость вернуть несколько значений. Вы видели, что это можно сделать, вернув список значений.

То же самое можно сделать немного проще: если после `return` написать несколько значений через запятую, то Python автоматически поместит эти значения в кортеж и вернет этот кортеж.

```
def getCoordinates():
    return 1, 2
print(getCoordinates())
# => (1, 2)
```

Команда возврата нескольких значений

```
return 1, 2
```

практически идентична команде возврата кортежа с этими значениями

```
return (1, 2)
```

Вы можете пользоваться любой, как вам больше нравится.

Мы разобрались, как возвращать значения из функции. Но как программа их получает, когда значений несколько? Оказывается, есть несколько способов. Один вариант вы знаете, можно записать в переменную весь кортеж:

```
def getCoordinates():
    return 1, 2
result = getCoordinates()
print(result)
# => (1, 2)
```

Можно вместо этого воспользоваться множественным присваиванием, тогда значения автоматически будут распределены по разным переменным:

```
x, y = getCoordinates()
print(x) # => 1
print(y) # => 2
```

Обратите внимание, `getCoordinates` в обоих случаях — это одна и та же функция, которая используется немного по-разному.

**Задача:** «Поиски возвышенного».

Когда в функции используется несколько операторов `return`, позаботьтесь о том, чтобы каждый из операторов возвращал однотипные наборы значений. Ведь вызывающая функция заранее не знает, какой из операторов return выполнится, а значит, мы не сможем использовать множественное присваивание в полную силу.

В следующей функции (это, конечно, бесполезная функция, она нужна только для иллюстрации) мы не последовали этому совету и возвращаем координаты то на плоскости, то в трехмерном пространстве.

```
def getCoordinates(index):
    if index % 2 == 0:
        return 1.5, 2.5
    else:
        return 1.5, 2.5, 0
```

Теперь вызов `x, y = getCoordinates()` будет ломаться на нечётных индексах, а `x, y, z = getCoordinates()` — на чётных индексах. В итоге мы не сможем записать ни один из вариантов так, чтобы он не сломался при каких-нибудь условиях.

На следующем занятии мы немного больше поговорим про возможности множественного присваивания в применении в аргументам и возвращаемым значениям функции.