Урок Функции: return

Функции. Возвращение значений из функций

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.

1. Связь между математическими функциями и функциями в Python

Функции, которые мы писали до сих пор, выводили значение на экран. 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, их было бы почти невозможно комбинировать.

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 — 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 о… 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

Чаты 1

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: 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

Функция в языке Python — некий 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, тоже есть список аргументов: иногда одно значение, иногда несколько, а иногда он и вовсе пустой, как у функции `input`. У функций также есть возвращаемое значение — значение всегда ровно одно.

Важно!

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 могут не возвращать значения, их называют «процедуры».

Их отличие от математических функций 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, работа некоторых функций зависит от внешних (глобальных) переменных (об этом мы будем говорить на следующем занятии). Это 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

Функции без побочных эффектов и использования внешних источников данных (их еще на… «чистые функции») ведут себя в точности как математические функции. Их результат зави… от аргументов, а вызов таких функций никак не влияет на ход остальной программы.

Чаты 1

Большая часть известных вам встроенных функций Python ведет себя так: `math.sqrt`, `math.cos`, `abs`, `int`, `str`, `len`, `min`, `max`.

Функции с побочными эффектами

Функции с побочными эффектами — как правило, функции, предназначенные для общения с «внешним миром»: 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.
На следующем занятии мы отдельно поговорим о том, как функции с побочными эффектами могут влиять на глобальные переменные, а как делать чистые функции и работать с ними, мы начнем говорить сейчас.

2. Возвращаемые значения

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

```
def double_it(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 = double_it(3.14) * radius
```

Когда интерпретатор дойдет до `double_it(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
```

Важно!

Заметьте: функция `double_it` ничего не выводит на экран. Она выполняет вычисления и со
их не пользователю, как мы делали раньше, а другой части программы.

Чаты 1

Если нам потребуется не просто вернуть удвоенное число, а еще и вывести его на экран, лучше не добавлять `print` внутрь функции. Ведь если вы добавите `print` в функцию, 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(double_it(3.14))
```

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

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

Функция удваивания числа, конечно, совершенно бесполезна.

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 ее значение.

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

Объект (значение) 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 ее 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 (подробнее об этом мы поговорим, когда будем рассматривать, как передаются параметры в функции).

Однако, если ни программист, ни программа не имеют возможности пользоваться объектом, 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».

1
Чаты

Сборка мусора

Сборка мусора

Это называется «сборка мусора»: Python автоматически избавляется от всех объектов, которые невозможно использовать.

3. 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: 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 в основную программу. 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
```

Можно обратить внимание, что в последней записи `else`-часть становится нам вообще не нужна, и тогда сократить функцию до вот такой:

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

1 Чаты

```
    return -x
```

Заметьте, что любую функцию можно написать с одним-единственным оператором `return`, но часто использовать несколько точек выхода из функции просто удобно.

4. Возврат из глубины функции

Множественные точки возврата из функции позволяют нам упростить обработку и более сложных структур, например, вложенных списков. 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 — 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, 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` не заключен ни в какое условие, выполняется он только тогда, когда эле не найден. Если элемент найден, мы сразу выходим из функции и до этой строки просто не Оператор `return` очень удобен, когда нужно выйти из глубины функции.

1 Чаты

5. Отладка

В PyCharm процесс отладки аналогичен тому, как он происходил в Wing IDE. Для создания/удаления точки останова, надо кликнуть на левое поле рядом со строчкой кода, где мы хотим остановиться в режиме отладки. Запустить программу в режиме отладки можно несколькими способами:

1. Из пункта меню Run → Debug <имя файла с программой>.
2. Из меню быстрых действий — кнопка с жуком над кодом в правом верхнем углу.
3. Кликнув правой кнопкой мышки в любом месте нашего файла и выбрав в выпадающем меню пункт Debug <имя файла с программой>.
4. Используя сочетание клавиш Shift + F9.

После того как программа запустится в режиме отладки, в нижней части программы откроется уже привычная консоль, но только перед выводом самой программы в нее выведется надпись вроде

```
pydev debugger: process 20924 is connecting
```

а затем

```
#Connected to pydev debugger (build 191.7479.30)#
```

Эти надписи говорят, что отладчик сначала попытался «прицепиться» к нашей программе, а потом — что он преуспел в этом и готов работать.

После остановки на брейкпоинте и в процессе дальнейшей отладки вам, как и в Wing IDE, будет доступен просмотр текущих значений переменных во вкладке Debugger в нижней части окна PyCharm. Приятным отличием от Wing является то, что текущие значения переменных пишутся еще и в коде в том месте, где они были объявлены.

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 (F7). Сначала будет определена функция `matrix_has_close_value`. На этом шаге интерпретатор Python в нее еще не заходит. Затем мы определяем переменную `table`, и она появляется на вкладке Stack data в разделе глоба переменных. Затем мы запускаем функцию и попадаем внутрь нее.

1 Чаты

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)` совпадет 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, возвращенное значение True 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.

Step Into

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: вы в ней уверены и ничего не собираетесь менять. Или в ней слишком много команд, которые не просмотреть. Или решили отложить ее отладку на будущее.

Step Over Statement

На этот случай в отладчике предусмотрена команда Step Over Statement (F8). При ее применении функция все 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.

Step Out

Иногда бывает нужно также выйти из функции. Например, когда вы зашли в нее отладчиком случайно или когда вы все в ней уже понимаете. В таких случаях вам поможет команда Step Out (Shift + F8). Она продолжает выполнение, пока не встретит return, выполняет его и оказывается за пределами функции.

Resume Program

Если вы посмотрели то, что хотели, в режиме отладки и хотите, чтобы выполнение программы продолжилось далее в обычном режиме (или до следующего брейкпоинта), удобно воспользоваться командой Resume Program (F9).

Чаты 1

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, а в каких нет?

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

6. Что можно возвращать из функции

В функциях, которые не возвращают значение, тоже можно использовать `return`. Если написать `return` без аргументов, 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`.

7. 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, Python автоматически поместит эти значения в кортеж и вернет этот кортеж.

```
def get_coordinates():
    return 1, 2


print(get_coordinates())
# => (1, 2)
```

Чаты 1

Важно!

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 get_coordinates():
    return 1, 2


result = get_coordinates()
print(result)
# => (1, 2)
```

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

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

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

Важно!

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

Чаты 1

```
    else:
        return 1.5, 2.5, 0
```

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

Вы уже видели, как локальные переменные помогают интерпретатору Python не запутаться в именах переменных. Но не всегда бывает просто понять, что за переменная используется в функции: собственная локальная переменная или чужая — из внешней программы. Чтобы разобраться в этих тонкостях, на следующем занятии мы очень подробно обсудим, что такое область видимости переменных.

8. Аннотация типов

Python — язык с динамической типизацией. Это означает, что типы связаны со значением переменной, а не с ней самой. Переменные могут принимать любое значение в любой момент и проверяются только перед выполнением операций над ними.

Однако, при написании кода, мы так или иначе предполагаем переменные каких типов будут использоваться (это может быть вызвано ограничением алгоритма или логики). Для корректной работы программы нам важно как можно раньше найти ошибки, связанные с передачей данных неверного типа.

Аннотации типов — это новая возможность, описанная в PEP484, которая позволяет добавлять подсказки о типах переменных. Они используются, чтобы информировать читателя кода, каким должен быть тип переменной.

Аннотации типов не влияют на время выполнения программы. Эти подсказки игнорируются интерпретатором и используются исключительно для повышения удобочитаемости для других программистов и вас самих. Аннотации типов поддерживаются многими IDE для Python, которые выделяют некорректный код или выдают подсказки в процессе набора текста.

Для создания аннотированных переменных можно использовать один из способов:

```
count = 5 # type: int
count: int; count = 5
count: int = 5
```

В первой строке тип переменной указываем в комментариях. В двух других строках тип переменной указываем в коде, через двоеточие от имени переменной и дальше двумя разными способами задаем значение переменной.

Для аннотирования списоков, кортежей, словарей и множеств используют имена классов.

```
my_list: list
my_tuple: tuple
my_dic: dict
my_set: set
```

Чаты 1

Такое аннотирование не слишком информативно, поскольку не показывает, какого типа элементы

содержатся внутри.

Что бы показать тип элементов его указывают внутри квадратных скобок:

```
my_list: list[int]
my_tuple: tuple[str, float, float]
my_dic: dict[str, int]
my_set: set[str]
```

Удобнее всего использовать аннотации типов при описании функции: можно указать типы аргументов и тип ожидаемого результата или что-то одно. Аргумент аннотируется через двоеточие после имени, а возвращаемое значение через -> после имени и аргументов функции.

```
def double_it(x: int) -> int:
    return x * 2


def get_coordinates() -> tuple:
    return 1, 2


def greet_all(names: list[str]) -> None:
    for name in names:
        print("Hello", name)
```

Справка
Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Лицей Академии Яндекса», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».
Пользовательское соглашение.

© 2018 – 2022 ООО «Яндекс»

1
Чаты