Урок 9.  
Функции. Локальные и глобальные переменные.

короткая линия

# План урока

1. Зачем нужны функции?
2. Создание функций
3. Возвращение значения из функции
4. Аргументы функции
5. Область видимости

# Зачем нужны функции?

Рассмотрим пример ситуаций, которые могут возникнуть при работе над реальными приложениями:

1. Допустим, Вам нужно записывать результаты работы программы в базу данных. Несмотря на то, что в курсе работа с базами данных еще не затрагивалась, в общем виде определим действия, которые необходимо выполнить для успешной записи. Нужно:

* открыть соединение с БД
* если записывать некуда, создать таблицу для записи
* записать данные
* закрыть соединение, чтобы предотвратить утечки данных

Учитывая, что в Вашей программе может быть несколько десятков мест, где необходимо записать данные, Вам не захочется много раз вставлять один и тот же код. К тому же, если Вы вдруг решите немного поменять функциональность (например, записывать данные немного в другом формате), придется менять блок кода во всех местах, куда Вы его вставили.

2) Теперь представьте, что Вам нужно написать программу, составленную из многих компонентов (например, задача “Olympiadnik Starter Pack” из урока 1.7: <https://stepik.org/lesson/195451/step/13?unit=169727>). Не используя функции, Вы написали бы решение, которое последовательно вычисляет сразу все, не разделяя код на подзадачи. Однако, определив функции gcd(a, b) для НОДа и is\_prime(n) для определения простоты числа, можно коротко записать программу так:

|  |
| --- |
| n = int(input()) arr = [int(x) **for** x **in** range(n)] Max = max(arr) Min = min(arr) Gcd = gcd(Min, Max) result = is\_prime(Gcd) **if** result:  print(**"YES"**) **else**:  print(**"NO"**) |

Можно еще короче, в лучших традициях языка Python:

|  |
| --- |
| arr = [int(x) **for** x **in** int(input())] print(**"YES"** **if** is\_prime(gcd(min(arr), max(arr))) **else** **"NO"**) |

И уже после этого можно браться за реализацию самих функций.

Таким образом, функции нужны прежде всего для:

1. Вынесения конкретной функциональности в независимый блок кода, который потом можно использовать где угодно. Изменив этот блок в одном месте, Вы также измените поведение Вашей программы во всех местах, где использована функция.
2. Разложения программы на составляющие, которые Вы реализуете позже. Полезно для планирования разработки, т. е. сначала Вы пишете общую идею, затем реализуете функции, необходимые для ее осуществления.

# Создание функций

Функция в Python – это объект, который, аналогично функциям в математике, принимает набор аргументов и возвращает значение. Создать функцию можно при помощи ключевого слова **def** (от “define” – определить), имени функции и круглых скобок, в которых указываются аргументы. Если аргументов нет, скобки остаются пустыми. Затем идут двоеточие и блок кода, относящийся к функции. Напишем функцию, которая печатает НОД двух чисел:

|  |
| --- |
| **def** gcd(a, b):  **while** b != 0:  a, b = b, a % b  print(a)  gcd(169, 13) gcd(5, 625) gcd(123, 12) gcd(int(input()), int(input())) |

В этой конструкции def – ключевое слово, gcd – имя функции (от “greatest common divisor”), a и b – параметры, т. е. два числа. Ниже объявления функции Вы можете увидеть ее использование для произвольных чисел. Важно: объявление функции обязательно должно быть перед ее использованием, иначе интерпретатор не сможет запустить программу.

Обратите внимание, что порядок аргументов важен. Например, в строчке gcd(169, 13) функция присвоит своей внутренней переменной “a” значение 169, а переменной “b” значение 13, и никак не наоборот. В нашем конкретном случае, при вычислении НОДа, порядок неважен, но об этом следует помнить.

# Возвращение значения

В большинстве случаев нам понадобится функция, которая возвращает какое-то значение, а не просто что-то печатает, чтобы работать с этим значением в будущем. За возврат значения отвечает ключевое слово **return**. Напишем функцию gcd(a, b), но пусть она теперь **возвращает** НОД, а не печатает его:

**def** gcd(a, b):  
 **while** b != 0:  
 a, b = b, a % b  
 **return** a  
  
print(gcd(169, 13))  
print(gcd(5, 625))  
print(gcd(123, 12))  
print(gcd(int(input()), int(input())))

Результат выполнения функции (в нашем случае это локальная переменная “a”) можно присвоить какой-то переменной и использовать дальше в коде без каких-либо ограничений. Если функция не возвращает ничего (т. е. в теле функции отсутствует **return**), то результатом ее выполнения будет **None**. **None** – это ключевое слово в языке Python, которое буквально значит “ничего” и является аналогом слова **null** в таких языках, как Java, JavaScript, C++. Попробуйте убрать строчку с return из функции и посмотрите на результат.

**Задача “Сумма факториалов”**

**Задача “Хитрая сумма”**

**Задача “Перестановки разрешены!”**

**Задача “Называем переменные правильно”**

# Аргументы функции

Как было сказано, при объявлении функции в круглых скобках Вы должны указать принимаемые параметры, которые нужны для работы функции. Попробуйте написать любую функцию и передать ей неверное число параметров. Что на это скажет интерпретатор?

Однако могут возникнуть ситуации, когда Вы не знаете, сколько аргументов можно передать, или когда какой-то параметр нужно менять лишь иногда, а в других случаях он имеет постоянное значение. Из этих ситуаций есть выход.

Напишем функцию вычисления произведения трёх чисел. Однако пусть третье число всегда равно 10, если не указано обратное:

|  |
| --- |
| **def** multiplication(a, b, c=10):  **return** a \* b \* c print(multiplication(5, 6)) print(multiplication(5, 6, 7)) |

Оба вызова функции будут работать корректно: в первом случае переменная “с” останется равна 10, во втором случае примет значение 7.

Напишем ту же функцию, но теперь пусть мы не знаем, сколько чисел хотим перемножить. В этом случае в круглых скобках явно укажем интерпретатору, что мы ждем переменное число аргументов:

|  |
| --- |
| **def** multiplication(\*args):  result = 1  **for** number **in** args:  result = result \* number  **return** result  print(multiplication(100)) print(multiplication(100, 2)) print(multiplication(100, 5, 6)) print(multiplication(100, 9, 12, 51, 32, 9, 0)) |

Любой из вызовов функции корректен. Если вы выведете объект args, вы увидите, что это кортеж из всех переданных аргументов. С args можно работать как с обычным кортежем без всяких ограничений. Строго говоря, называть этот кортеж именно args необязательно, это просто договоренность, главное – это наличие “звездочки”.

**Задача “Бинарный аукцион”**

**Задача “Жадность – корень всех зол”**

# Область видимости

Мы уже видели, что при передаче аргументов функция оперирует своими внутренними переменными, которые не были даже объявлены в программе. Эти переменные существуют только в контексте функции, используются только внутри нее и не “просачиваются” в основную программу. При этом функция может использовать переменные, объявленные вне ее (но не в другой функции). Это поведение иллюстрирует такое понятие, как “область видимости”.

Доступ к глобальной (т. е. объявленной вне функции) переменной можно получить как вне функции, так и внутри нее, однако изменить ее внутри функции нельзя. Доступ к локальной (т. е. объявленной внутри функции) переменной можно получить только в этой функции или в функции, помещенной внутри нее.

Проиллюстрируем на примере:

|  |
| --- |
| **def** func():  x = 101 func() print(x) |

Данная программа завершится с ошибкой, так как в глобальной области видимости нет переменной x, она была объявлена только в функции.

|  |
| --- |
| **def** func():  number = 142 number = 42 func() print(number) |

Программа выведет число 42. Объясним это. В глобальной области видимости есть число number, равное 42. Вызывая функцию, мы создаем локальную область видимости, в которой также есть переменная number с другим значением. Как только функция завершает свою работу, интерпретатор забывает про ту область видимости и оперирует только с глобальной, в итоге выводя number = 42.

Если же Вы все-таки хотите изменить значение переменной из глобальной области видимости внутри функции, это можно сделать с помощью ключевого слова global, после которого через запятую следуют переменные.

**def** func():

global number  
 number = 142  
number = 42  
func()  
print(number)

На этот раз программа выведет число 142, так как func() будет относиться к переменной number как к глобальной.

**Задача “Корректная дата”**

**Задача “Необычная сортировка”**

**Задача “Циклический сдвиг”**

**Задача “Системы счисления. Этап 1”**