Урок 13. Функции. Начало.

План урока

1. Функция как способ группировать команды и именовать участки кода. Функция как инструмент для повторного использования.
2. Определение простейших функций.
3. Начальные знания о локальных переменных.
4. Аргументы функций.
5. Порядок выполнения кода при передаче вычисляемого значения в качестве аргумента функции.

Аннотация

В этом уроке мы поговорим о том, как группировать команды в функции — участки кода, которые можно использовать многократно. Обсудим, как можно сделать так, чтобы код функции работал по-разному в зависимости от параметров. Наконец, коснёмся вопроса о том, что из себя представляют локальные переменные.

Функция как способ группировать команды и именовать участки кода

Работа любой компьютерной программы — это выполнение процессором большого набора элементарных инструкций. В машинном коде, с которым работает процессор, все команды очень простые:

- считать из оперативной памяти одно целое число в специальную ячейку,
- прибавить к одному числу значение другой ячейки,
- сравнить ячейку с нулём,
- вернуться на пару команд назад и пр.

Команды машинного кода не могут вывести окошко программы или проиграть аудиофайл, не могут посчитать среднюю оценку в классе или загрузить страничку из интернета. Машинный код не умеет полноценно работать даже с обычными строками или списками и не может выполнять сложные математические расчеты. Однако программа целиком всё это делает, потому что состоит из множества команд, которые в комбинации дают нужный эффект.

Многие из команд Python, которые вы уже знаете, требуют от процессора выполнения десятков команд. Если бы программист писал их вручную, то даже простейшие программы — вроде наших учебных заданий — создавались бы несколько дней. При этом даже опытному программисту было бы очень легко допустить ошибку.

Чтобы упростить разработку программ, наборы команд принято группировать в **функции** (их иногда называют **подпрограммами**).

Функция — это особым образом сгруппированный набор команд, которые выполняются последовательно, но воспринимаются как единое целое. При этом функция может возвращать (или не возвращать) свой результат.

Например, чтобы получить сумму элементов списка, мы обычно выполняем такой набор операций:

1. Записать в ячейку результата ноль.
2. Пройти в цикле по всем элементам списка и прибавить к результату каждый из этих элементов.

Чтобы не писать такой цикл каждый раз, когда вам нужно получить сумму элементов, можно завести специальную функцию. Она принимает на вход список и возвращает подсчитанный результат:

```
def my_sum(arr):
    result = 0
    for element in arr:
        result += element
    return result
```

Теперь, чтобы вычислить сумму списка чисел, нам достаточно написать в программе что-то вроде `my_sum([5,10,7,8])`.

Мы написали функцию `my_sum` исключительно для демонстрации. На самом деле в `Python` уже есть встроенная функция для вычисления суммы. Она называется `sum`. Мы еще подробно разберём, что означает такое определение, и что означает вызов функции.

А пока давайте подумаем, какие еще встроенные функции вы уже знаете?

На самом деле, в любой современный язык программирования встроены сотни функций. Много, но всё равно недостаточно, поэтому в любой серьезной программе приходится писать множество собственных функций.

Итак, мы уже выяснили, что функции нужны, чтобы группировать команды, а заодно, чтобы не писать один и тот же код несколько раз. Например, достаточно один раз написать функцию суммы и потом пользоваться ей постоянно. Польза от этого особенно очевидна, когда функция действительно сложная и используется много раз в разных местах программы. Например, загрузить данные из интернета или нарисовать персонажа компьютерной игры.

Ещё одна важная вещь — функции имеют **имена**.

Благодаря им программу можно сделать понятной не только компьютеру, но и человеку. Тут все так же, как с именами переменных: если переменная имеет ничего не говорящее название, то сложно угадать, что в ней хранится. Если участок кода не сгруппирован в функцию, то приходится буквально дешифровывать, для чего он нужен в программе. А если он оформлен в виде функции, то название функции само подскажет, что делает этот код.

Попробуйте угадать, что делает такой код:

```
t = [-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3]
s = 0
mm = 1000
mx = -1000
for e in t:
    s += e
    if e < mm:
        mm = e
    if e > mx:
        mx = e
print(s / len(t))
print(mm)
print(mx)
```

```
9.75
-10
27
```

```
temperatures = [-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3]
average_temperature = sum(temperatures) / len(temperatures)
print(average_temperature)
print(min(temperatures))
print(max(temperatures))
```

```
9.75
-10
27
```

На самом деле, когда программист думает о том, что должна делать программа, он обычно представляет её как раз в форме функций. Мы обычно не говорим, какие действия должен выполнить алгоритм, а описываем, что мы хотим получить. Например, мы хотим посчитать среднегодовую температуру — значит, нам нужна функция вычисления среднего значения из набора чисел.

Определение простейших функций.

Теперь давайте научимся писать простые функции.

Самый просто пример - это функция, которая просто повторяет одни и те же действия, независимо ни от чего. У каждой функции есть **заголовок** (его обычно называют сигнатурой) и **тело функции**.

Сигнатура описывает, как функцию вызывать, а тело описывает, что эта функция делает.

Сигнатура содержит имя функции, а также аргументы (то есть параметры), которые передаются в функцию.

Аргументов, в принципе, может и не быть. Записывается это так:

```
def <имя функции>([аргументы]):
    <тело функции>
```

Тело функции, как и в операторе `if` или в операторе цикла, обязательно идёт **с отступом**. Это нужно, чтобы интерпретатор `Python` знал, где заканчивается код функции. Заодно это здорово помогает структурировать программу. Даже в языках, где отступы не требуются, их всё равно принято писать, чтобы упростить чтение программы.

Давайте теперь напишем совсем простую функцию из одной единственной команды, которая просто выводит на экран приветствие.

```
def simple_greetings():
    print('Привет, username!')
```

Теперь, чтобы поприветствовать пользователя, вам достаточно в основной программе написать:

```
simple_greetings()
```

Это называется вызвать функцию.

Обратите внимание, что у этой функции нет аргументов ни в определении, ни при вызове. Однако пустые скобочки после названия функции писать всё равно нужно.

Функцию, как и переменную, необходимо сначала объявить, а только потом использовать. Поэтому следующая программа выдаст вам ошибку `name 'simple_greetings2' is not defined`.

```
simple_greetings2()


def simple_greetings2():
    print('Привет, username!')
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-12-4f64c3fab903> in <module>()
----> 1 simple_greetings2()
      2
      3
      4 def simple_greetings2():
      5     print('Привет, username!')

NameError: name 'simple_greetings2' is not defined
```

Впрочем, от такой функции проку не очень много: она не сокращает количество кода и не сильно упрощает понимание происходящего в программе. Давайте теперь немного усложним программу. Пусть она спрашивает имя и здоровается по имени с тремя людьми. Давайте для начала напишем такую программу без собственных функций:

```
print('Как тебя зовут?')
name_1 = input()
print('Привет', name_1)
print('А тебя?')
name_2 = input()
print('Привет', name_2)
print('А твоего пса?')
name_3 = input()
print('Привет', name_3)
```

```
Как тебя зовут?
Вася
Привет Вася
А тебя?
Коля
Привет Коля
А твоего пса?
Шарик
Привет Шарик
```

Программа небольшая, но в ней уже видно много проблем.

Во-первых, три раза приходится повторять фактически одно и то же.

Во-вторых, приходится вводить разные имена переменных, чтобы не записывать в одну переменную три разных имени. При этом нельзя перепутать имена переменных, иначе вы поприветствуете кого-нибудь неправильным именем.

В-третьих, если вы захотите исправить приветствие на более официальное **Здравствуйте**, вам придется внести одинаковые исправления сразу в трёх разных местах. Даже в такой маленькой программе можно при исправлении допустить опечатку. А представьте, что приветствие используется десять раз в разных местах большой программы — тогда придется искать каждое приветствие и исправлять его.

Чтобы избежать всех этих проблем, мы сгруппируем одинаковые действия в функцию.

```
def greet():
    name = input()
    print('Привет,', name)


print('Как тебя зовут?')
greet()
print('А тебя?')
greet()
print('А твоего пса?')
greet()
```

```
Как тебя зовут?
Иван
Привет, Иван
А тебя?
Мария
Привет, Мария
А твоего пса?
Бобик
Привет, Бобик
```

Посмотрите: код программы сократился и стал еще понятнее. Теперь вам не нужно выискивать, где какая переменная заводится, где и для чего она используется. Программа сама говорит вам, что она делает: `greet` — **здороваться**. Не приходится заводить несколько разных переменных. Чтобы поменять приветствие во всей программе, достаточно изменить одну строчку. Словом, одни плюсы.

Задача: Формальное приветствие (https://lms.yandexlyceum.ru/task/view/1290)

Начальные знания о локальных переменных

В тот момент, когда вы вызываете функцию `greet`, начинают выполняться команды, написанные в теле функции. Когда работа функции доходит до конца, исполнение программы продолжается со строки, которая вызывала функцию. Мы ещё посмотрим на этот процесс подробнее с помощью отладчика.

Обратите внимание, теперь в программе используется только одна переменная: `name`. Как же так? Ведь мы договорились, что не будем использовать одну и ту же переменную для разных имен? На самом деле мы не используем одну и ту же переменную. При каждом вызове функции эта переменная создается заново, а в конце работы функции — прекращает своё существование. Это очень важный момент: снаружи функции `greet` переменная `name` вообще не существует. Таким образом, функция очерчивает тот участок программы, где переменная нужна и используется. Этот участок, в котором переменная **живёт**, называется **областью видимости переменной** (по-английски — `scope`).

Благодаря ограничению области видимости переменной, программисту не нужно беспокоиться, не «всплывёт» ли эта переменная в другом месте программы. Изменяя переменную внутри функции, программист понимает, что он может что-то испортить **только внутри** функции, но не поломает работу остальной программы.

Можно сказать, что вся работа с переменной локализована, т.е. сосредоточена внутри функции. Такие переменные, которые существуют только внутри функции, называются **локальными**.

Про локальные переменные и области видимости мы поговорим намного подробнее в следующем уроке.

Задачи в классе:

- **Таблица умножения (https://lms.yandexlyceum.ru/task/view/1291)**
- **Привет, как тебя там (https://lms.yandexlyceum.ru/task/view/1292)**

Домашние задачи:

- **Улыбайтесь, господа (https://lms.yandexlyceum.ru/task/view/1295)**
- **Скажи пароль и проходи (https://lms.yandexlyceum.ru/task/view/1296)**

Дополнительные задачи:

- **Треугольник Паскаля (https://lms.yandexlyceum.ru/task/view/1339)**

Аргументы функций

Мы рассмотрели функции, которые выполняют всякий раз одни и те же действия. Это бывает полезно, но всё же большая часть программ требует выполнения немного разных действий. Например, функция `print` (а это именно функция) должна каждый раз выводить на экран разные сообщения — в зависимости от переданных аргументов. **Аргументы** (параметры) могут изменять поведение функции. Например, функция `len` принимает строки или списки (и другие коллекции). В зависимости от конкретного аргумента она возвращает разный результат, а значит, выполняет внутри немного различные действия.

Рассмотрим функцию, которая должна выводить на экран содержимое списка, печатая каждый элемент на своей строчке. Вряд ли нам захочется заводить функцию, которая раз за разом выводит содержимое одного и того же списка. Скорее нужна функция, которая может распечатать любой список. Конкретный список мы передаем функции в качестве параметра при её вызове. Функция же работает с тем, что ей передали. Выглядит это так:

```
def print_array(array):
    for element in array:
        print(element)


print_array(['Hello', 'world'])
print()
print_array([123, 456, 789])
```

```
Hello
world

123
456
789
```

При первом вызове функции `print_array` переменная `array` будет равна `['Hello', 'world']`. При втором вызове переменная `array` будет равна `[123, 456, 789]` .

Разберемся, в каком порядке выполняется код при вызове функций. В примере:

```
print_array(['Hello'] + ['world'])
```

ничего удивительного не происходит: списки складываются, а затем передаются в функцию.

Давайте рассмотрим более сложный пример:

```
def print_hello(arg_1, arg_2):
    print('hello')


def print_comrade():
    print('comrade')


def print_petrov():
    print('Petrov')


print_hello(print_comrade(), print_petrov())
```

```
comrade
Petrov
hello
```

Аргументы в функции `print_hello` никак не используются, но сейчас это не важно. Рассмотрим, в каком порядке выполняются функции. В момент вызова функции ей необходимо передать вычисленные аргументы. Если аргументы не вычислены, то они вычисляются слева направо. В данном случае функция `print_hello` принимает аргумент `arg_1`, который является значением функции `print_comrade` (по-умолчанию — `None`), и аргумент `arg_2`, который является значением функции `print_petrov`. Таким образом, сначала выполнится функция `print_comrade`, затем `print_petrov` и лишь в самом конце `print_hello`. Результатом работы программы будет напечатанный текст

```
comrade
Petrov
hello.
```

Задачи:

- "Аргументированная" таблица умножения (https://lms.yandexlyceum.ru/task/view/1293)
- Среднее значение (https://lms.yandexlyceum.ru/task/view/1294)

Дополнительная задача:

- Статистики (https://lms.yandexlyceum.ru/task/view/1340)

Задачи на дом:

- Золотое сечение (https://lms.yandexlyceum.ru/task/view/1297)
- Длинношеее (https://lms.yandexlyceum.ru/task/view/1341)

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.