

Урок Функции

Функции

- 1 Среда разработки PyCharm
- 2 Функция как способ группировать команды и именовать участки кода
- 3 Определение простейших функций
- 4 Начальные знания о локальных переменных
- 5 Аргументы функций

Аннотация

Второе полугодие мы начнем с перехода на новую интегрированную среду разработки PyCharm. Эта среда динамично развивается, постоянно обновляется и предоставляет расширенную функциональность, которая в дальнейшем потребуется нам при решении задач.

Сегодня 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, коснемся вопроса, что представляют собой локальные переменные.

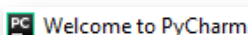
1. Среда разработки PyCharm

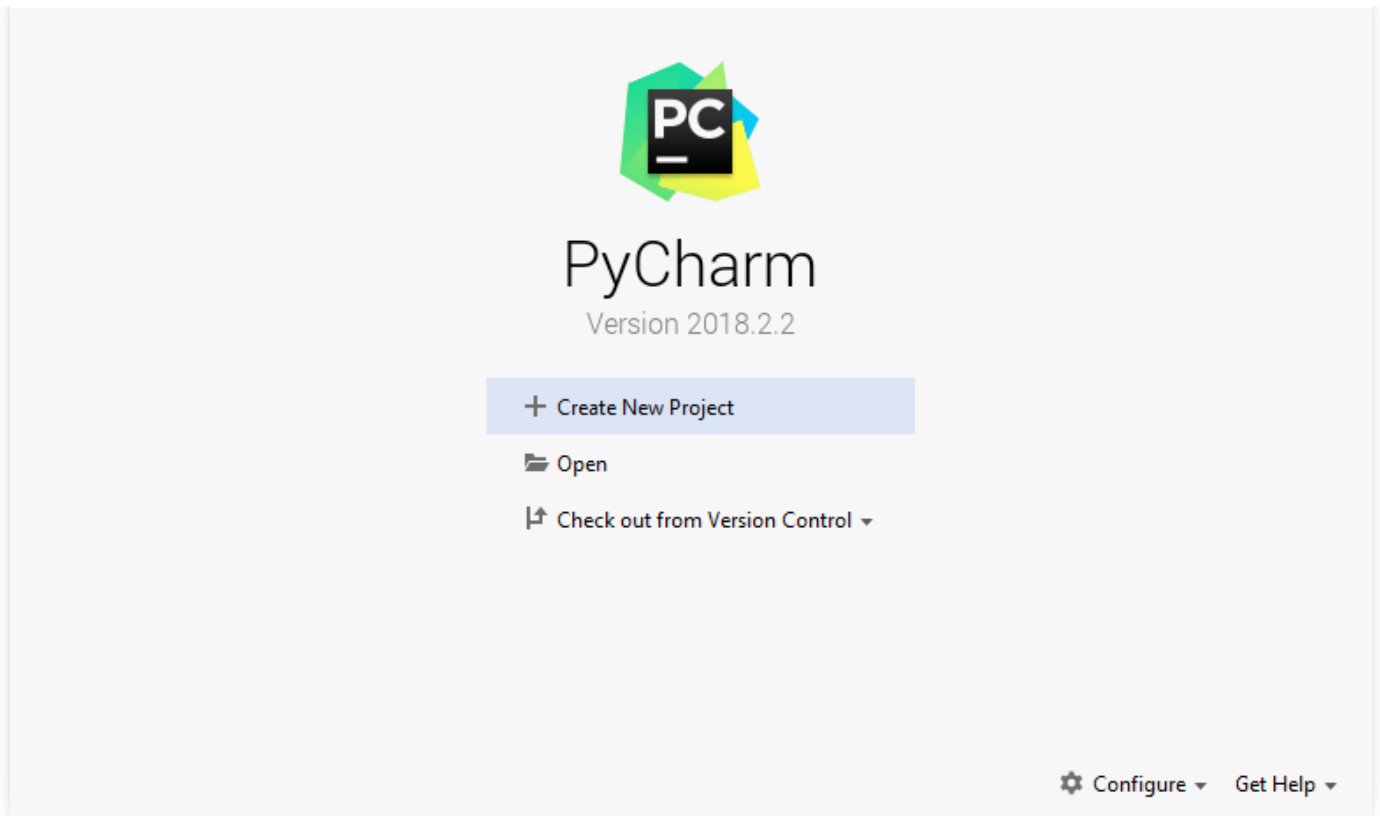
Для программирования на Python можно использовать различные среды разработки: так, в первом полугодии мы работали в Wing IDE, которая очень хороша для старта. В этом полугодии мы познакомимся с другой средой, которая наиболее распространена в профессиональном сообществе, — PyCharm, созданной компанией JetBrains. Ее отличает значительный прогресс в развитии, наличие постоянных обновлений и широкое распространение на таких операционных системах, как Windows, MacOS, Linux.

PyCharm доступна в двух основных вариантах: платный выпуск Professional и бесплатный Community. Мы будем использовать бесплатную версию, в которой доступны все необходимые в рамках нашего курса возможности.

Для этого перейдем на [страницу загрузки](#) и загрузим установочный файл PyCharm Community. После загрузки выполним его установку.

Далее запустим программу. При первом запуске открывается начальное окно:

The image shows the 'Welcome to PyCharm' dialog box. It has a dark header bar with the PyCharm logo on the left and the text 'Welcome to PyCharm' on the right. Below the header, there are several buttons and links for getting started, but they are not clearly legible in this low-resolution image.



Шаг 1. Откройте/создайте проект в PyCharm

А зачем вообще нужны проекты? Дело в том, что все, что вы делаете в PyCharm, выполняется в контексте проекта. Он служит основой для поддержки кодирования, переработки кода, согласованности стиля кодирования и т. д. Кроме того, вы, наверное, догадываетесь, что сложные проекты могут делиться на части (в терминологии Python — модули, но об этом позднее), которые реализовываются разными разработчиками. Концепция организации программного кода в проекты позволяет сделать такое взаимодействие более удобным.

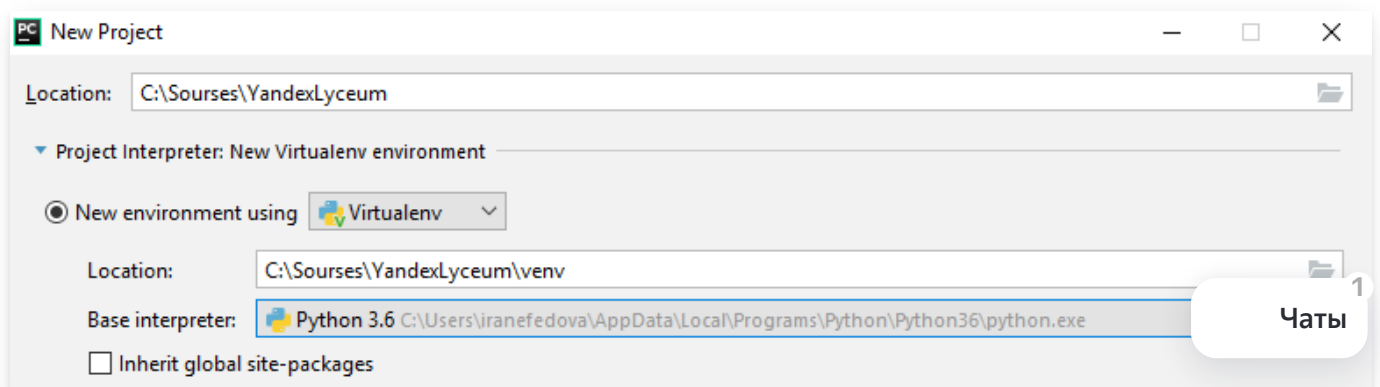
У вас есть два варианта начать работу над проектом внутри среды IDE:

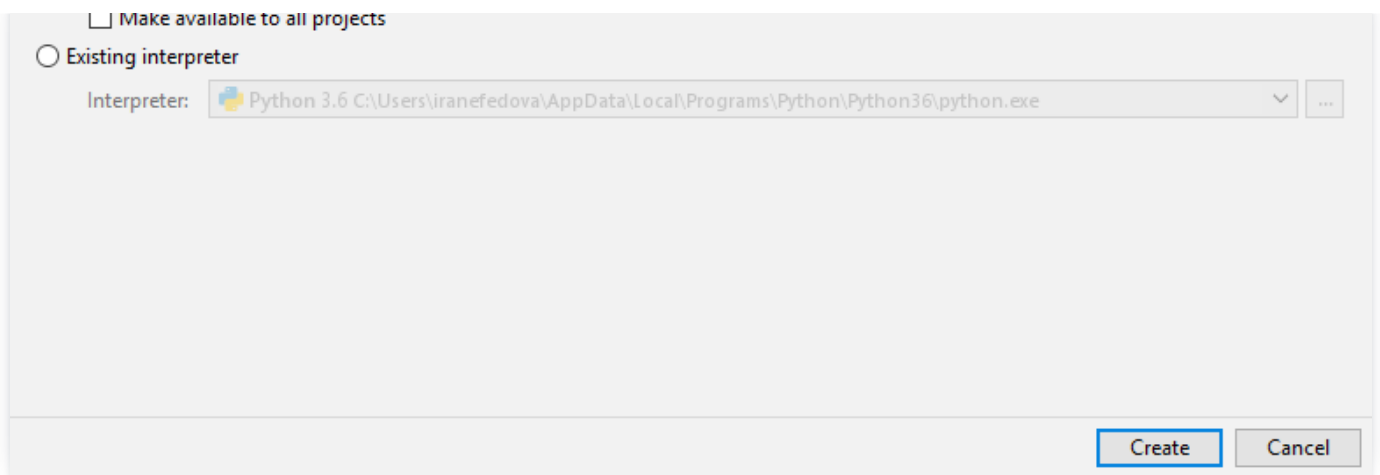
1. Открыть существующий проект

Начните, открыв один из своих проектов, хранящихся на вашем компьютере. Папку с вашими решениями можно рассматривать как проект. Вы можете это сделать, нажав **Открыть проект (Open)** на экране приветствия (или **File** → **Open**).

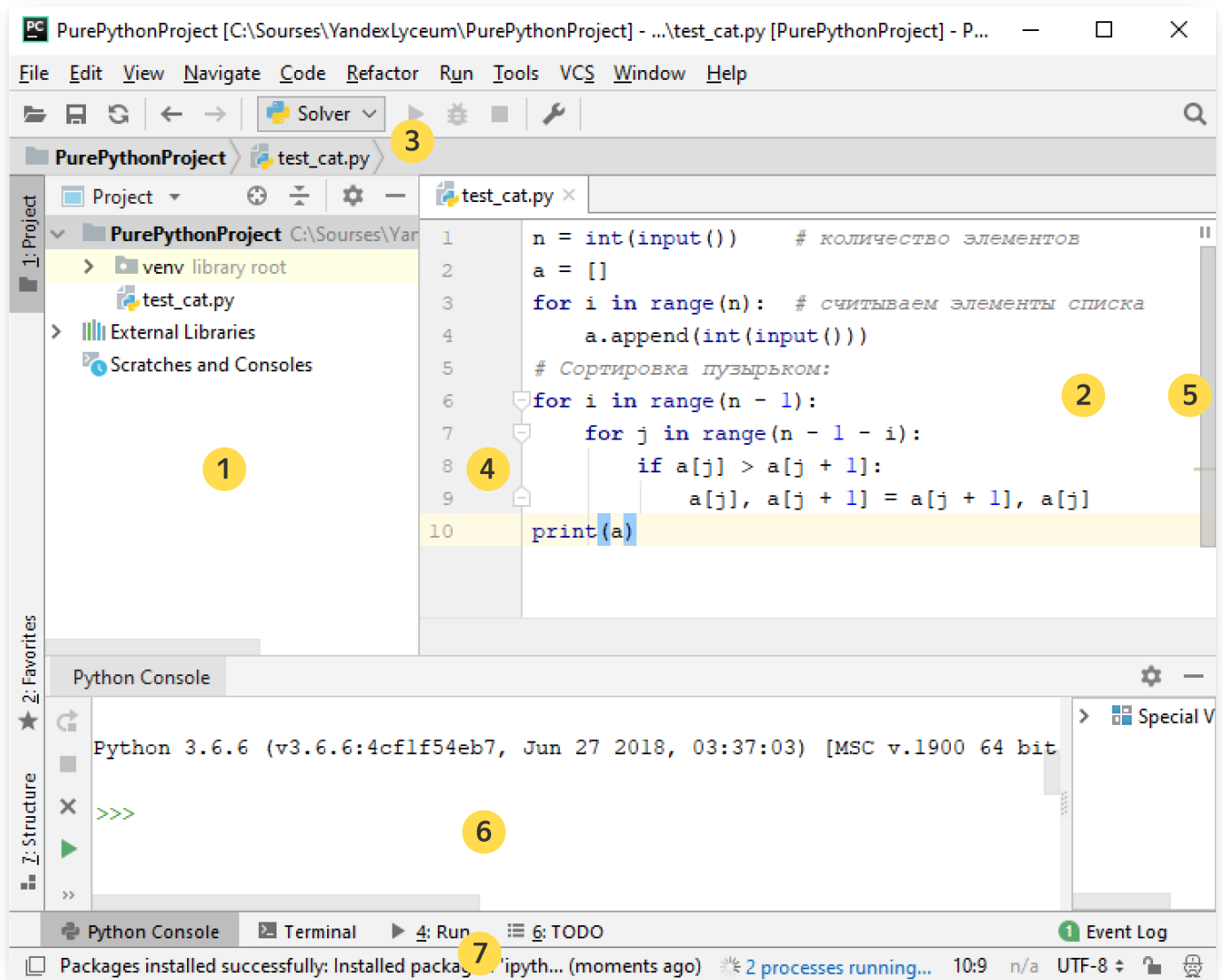
2. Создать проект с нуля

Если вы предпочитаете начинать с нуля, нажмите **New Project**, на открывшемся экране введите местоположение проекта или выберите его в диалоговом окне (название конечной папки принято считать именем проекта). Нажмите кнопку **Create**, после чего будет создан проект Python.





Шаг 2. Знакомство с интерфейсом



1. **Project Tool Window.** Панель инструментов проекта. В этом окне отображаются файлы вашего проекта.
2. **PyCharm Editor.** Редактор PyCharm. Находится с правой стороны, где вы пишете свой код. В нем есть вкладки для удобной навигации между открытыми файлами.
3. **Navigation Bar.** Панель навигации. Находится над редактором, позволяет быстро запускать и отлаживать ваше приложение.
4. **Left gutter.** Левый столбец, вертикальная полоса рядом с редактором, показывает точки и обеспечивает удобный способ перехода по иерархии кода.

1
Чаты

5. **Right gutter.** Правый столбец, справа от редактора. PyCharm постоянно контролирует качество кода и постоянно показывает результаты проверки в правом столбце: ошибки, предупреждения и т. д.

Тут же вы можете отслеживать соответствие вашего кода стандарту PEP 8. Индикатор в правом верхнем углу показывает общий статус проверки кода для всего файла.

6. **PyCharm Tool Windows.** Панель инструментов PyCharm. Это специальные окна, прикрепленные к низу и сторонам рабочей области, которые обеспечивают доступ к типичным задачам, таким как управление проектами, поиск и навигация по исходному коду, интеграция с системами контроля версий и т. д.

7. **Status Bar.** Строка состояния. Указывает состояние вашего проекта и показывает различные предупреждения и информационные сообщения.

У PyCharm есть гибкая система настройки интерфейса и даже возможность задания собственного стиля кода. Но сейчас мы не будем останавливаться на этом, а перейдем непосредственно к написанию программ.

2. 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 нулем
- 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 все 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, даже простейшие программы — вроде наших учебных заданий — создавались бы несколько дней. При этом даже опытному программисту было бы очень легко допустить ошибку.

В качестве мотивирующего примера рассмотрим программу, в которой последовательно запрашивается имя, а затем выводится приветствие по имени для трех живых существ:

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

Чаты ¹

```
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 уже видно много проблем:

- Во-первых, три раза приходится повторять фактически одно и то же
- Во-вторых, приходится вводить разные имена переменных, чтобы ничего не перепутать и не поприветствовать кого-нибудь неправильным именем
- В-третьих, если вы захотите исправить приветствие на более официальное — например, «Здравствуйте», вам придется внести одинаковые исправления сразу в трех разных местах

Даже в такой маленькой программе можно при исправлении допустить опечатку. А представьте, что фраза используется десять раз в разных местах большой программы — тогда придется искать каждое приветствие и исправлять его.

Было бы здорово иметь возможность устранить дублирование кода: в каком-то одном месте сообщить интерпретатору, что именно мы понимаем под словом «поприветствовать», а затем попросить интерпретатор использовать определение термина «поприветствовать» там, где мы его попросим об этом.

Итак, сформулируем, чего мы хотим добиться:

- **Один раз** определить, что значит «поприветствовать», т. е. сгруппировать и поименовать повторяющийся кусок кода
- **Многократно** в дальнейшем «ссылаться» на это определение везде, где нам только потребуется

Замечательно, что язык Python действительно обладает такими выразительными возможностями — **функциями**.

Функция

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

Чаты¹

Для того чтобы использовать какую-нибудь собственную функцию, вначале необходимо ее **объявить**, т. е. рассказать, что именно она будет делать. В нашем примере мы объявим функцию `greet` с помощью

ключевого слова `def`, после которого идет название нашей функции, скобки, двоеточие, а затем на двух последующих строчках — описание того, что она собой представляет:

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

Обратите внимание: это описание расположено в блоке кода с отступом.

Далее мы можем использовать нашу функцию `greet` всякий раз, когда в нашем коде возникает необходимость кого-нибудь поприветствовать:

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

Вызов функции

Обращение к ранее объявленной функции с целью выполнения ее команд называется вызовом.

В нашем примере функция `greet` один раз объявляется, а затем три раза вызывается.

Что мы получили в результате:

1. Код сократился и стал понятнее. Теперь нам не нужно выискивать, где какая переменная заводится, где и для чего она используется. Функция сама говорит, что она делает: `greet` — «поприветствовать».
2. Нам не приходится заводить несколько разных переменных.
3. Чтобы поменять приветствие во всей программе, достаточно изменить одну строчку.

Итак, функции нужны, чтобы группировать команды, а заодно — чтобы не писать один и тот же код несколько раз.

Например, достаточно один раз написать функцию `greet` и потом пользоваться ею постоянно. Польза от этого особенно очевидна, когда функция действительно сложная и используется много раз в разных местах программы. Например, загрузку данных из Интернета или отрисовку персонажа компьютерной игры удобно оформлять в виде отдельных функций.

Еще одна важная вещь состоит в том, что функции имеют **имена**.

PEP 8

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания, это необходимо, чтобы увеличить читабельность.

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, сложно угадать, что в ней хранится. Если участок кода не сгруппирован в функцию, иногда приходится буквально дешифровать, для чего он нужен в программе. А если он оформлен в виде функции, название функции само подскажет, что делает этот код.

Проиллюстрируем сказанное на примере. 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)
```

После некоторых нетривиальных усилий по дешифровке можно понять, что этот код вычисляет среднее, минимальное и максимальное значение списка:

```
9.75
-10
27
```

А вот тот же самый код, но переработанный с помощью встроенных функций и хороших названий переменных:

```
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
```

Какой вариант вам нравится больше?

Чаты

На самом деле, когда программист думает о том, что должна делать программа, он обычно представляет

ее 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.

3. Определение простейших функций

Давайте подытожим то, что мы знаем о функциях.

Заголовок и тело функции

У каждой функции есть заголовок (его обычно называют сигнатурой) и тело.

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

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

Записывается это так:

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

Важно!

Тело функции, как и в операторе `if` или в операторе цикла, обязательно идет 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 все равно принято писать, чтобы упростить чтение программы.

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

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

Вызов функции

Теперь, чтобы поприветствовать пользователя, вам достаточно в основной программе написать: `simple_greetings()`. Это называется «вызвать функцию».

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

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

```
simple_greetings2()
```

```
def simple_greetings2():
```



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

```
NameError                                Traceback (most recent call last)  
<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. Поэтому в виде функции стоит оформлять только логически законченный блок кода, особенно если он необходим в нескольких местах программы.

PEP 8

После функции до кода, который находится вне функции, необходимо делать отступ в две пустые строки для повышения читаемости кода. Если у вас есть несколько функций в одном файле, между кодом одной и сигнатурой другой функции тоже надо оставлять две пустые строки.

А теперь то, что заставит вас полюбить новую среду разработки еще сильнее: PyCharm умеет в автоматическом режиме исправлять часть недочетов кода, связанных в основном с отступами, пропущенными или лишними пробелами. Для автоформатирования кода по PEP 8 можно нажать комбинацию клавиш **Ctrl + Alt + I**.

Обратите внимание на некоторые особенности, которые существуют при сдаче в LMS задач на функции:

1. В LMS необходимо отправить файл только с необходимой функцией. Если в файле есть код вызова данной функции, закомментируйте его. В PyCharm можно выделить код, который необходимо закомментировать, и нажать **Ctrl + /**. Для раскомментирования надо повторить то же самое.
2. При проверке функции тестирующая система запускает программу на Python, которая осуществляет вызов вашей функции (с передачей в нее параметров), поэтому необходимо точно соблюсти формат сигнатуры функции, который дается в условии.

4. 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 еще 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? Нет, мы не используем одну и ту же переменную. При каждом вызове функции эта переменная заново, а в конце работы функции — прекращает свое существование. Это очень важный момент.

Чаты

Область видимости переменной

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 **живет**, называется областью видимости переменной (по-английски — *scope*).

Благодаря ограничению области видимости переменной программисту не нужно беспокоиться, не «всплывет» ли эта переменная в другом месте программы. Изменяя переменную внутри функции, программист понимает, что он может что-то испортить **только внутри** функции, но не ломает работу остальной программы. Можно сказать, что вся работа с переменной локализована, т. е. сосредоточена внутри функции.

Локальные и глобальные переменные

Переменные, создаваемые внутри функций, недоступны извне и существуют только внутри функции. Они называются локальными.

Создаваемые вне функции переменные могут быть доступны из функций. Они являются глобальными.

По возможности избегайте использования глобальных переменных для предотвращения конфликтов. О локальных переменных и областях видимости мы поговорим намного подробнее на следующих уроках.

5. Аргументы функций

Мы рассмотрели функции, которые выполняют всякий раз одни и те же действия. Это бывает полезно, но все 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 функций

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 строке. 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 ее 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.

```
Hello  
world
```

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())
```

Аргументы в функции `print_hello` никак не используются, но сейчас это неважно. Рассмотрим, в каком порядке выполняются функции.

Важно!

В момент вызова функции ей необходимо передать вычисленные аргументы. Если аргументы не вычислены, 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
```

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»