* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок Функции](https://lyceum.yandex.ru/courses/123/groups/568/lessons/661)

**Функции**

**План урока**

1

[Среда разработки PyCharm](https://lyceum.yandex.ru/courses/123/groups/568/lessons/661/materials/1196#1)

2

[Функция как способ группировать команды и именовать участки кода. Функция как инструмент для повторного использования](https://lyceum.yandex.ru/courses/123/groups/568/lessons/661/materials/1196#2)

3

[Определение простейших функций](https://lyceum.yandex.ru/courses/123/groups/568/lessons/661/materials/1196#3)

4

[Начальные знания о локальных переменных](https://lyceum.yandex.ru/courses/123/groups/568/lessons/661/materials/1196#4)

5

[Аргументы функций](https://lyceum.yandex.ru/courses/123/groups/568/lessons/661/materials/1196#5)

**Аннотация**

*Второе полугодие мы начнем с перехода на новую интегрированную среду разработку PyCharm — это среда динамично развивается, постоянно обновляется и предоставляет расширенный функционал, который нам потребуется при решении задач в дальнейшем.*

*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.*

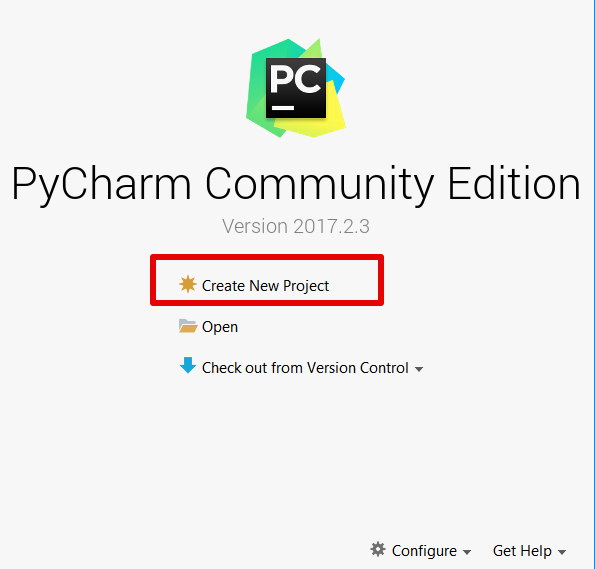
**1. Среда разработки PyCharm**

Для программирования на Python можно использовать различные среды разработки, так в первом полугодии мы работали в Wings, которая очень хороша для старта. В этом полугодии мы познакомимся с другой средой, которая наиболее распространена в профессиональном сообществе — это среда PyCharm, созданная компанией JetBrains. Это среда динамично развивается, постоянно обновляется и доступна для наиболее распространенных операционных систем — 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](https://www.jetbrains.com/pycharm/download/#section=windows) и загрузим установочный файл PyCharm Community. После загрузки выполним его установку.

После завершения установки запустим программу. При первом запуске открывается начальное окно:



**Шаг 1.** Откройте / создайте проект в PyCharm

А зачем вообще нужны проекты? Дело в том, что всё, 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 уже знаете, что в сложные проекты могут делиться на модули реализовываться 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:

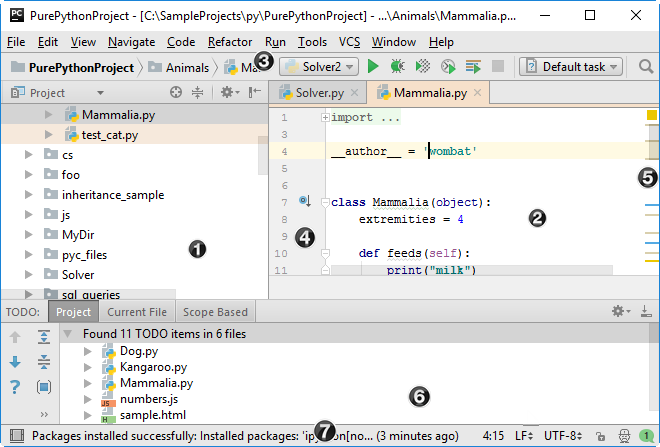
1. Открыть существующий проект

Начните с открыв один из ваших существующих проектов, хранящихся на вашем компьютере. Вы можете 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 и на открывшемся экране введите имя проекта в диалоговом окне и 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 остановки и обеспечивает удобный способ перехода по иерархии кода.
5. **Right gutter.** Правый столбец, справа от редактора. PyCharm постоянно контролирует качество вашего 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 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.

**Функция**

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. Вы уже знаете по материалам прошлого урока, что в Python есть встроенная функция для вычисления суммы. Она называется sum. А 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**.

**PEP 8**

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания — это необходимо, чтобы увеличить читабельность. Стиль mixedCase допускается в тех местах, где уже преобладает такой стиль, для сохранения обратной совместимости.

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 дешифрировать, 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

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.

**3. Определение простейших функций**

Теперь давайте научимся писать простые функции.

**Заголовок и тело функции**

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.

Записывается это так:

**def** <имя функции>([аргументы]):

<тело функции>

**Важно**

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!')

**Вызов фукнции**

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.a

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 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. 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?

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.

**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 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:

**Область видимости функции**

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.

**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 в то числе — они являются **глобальными**.

По возможности избегайте использования глобальных переменных для предотвращения конфликтов. Про локальные переменные и области 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 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**

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.

**Важно**

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

[Справка](https://yandex.ru/support/lyceum-students)

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

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты