Оглавление

[1. 9.Функции. Чистота функций. Побочные эффекты. Области видимости. LEGB. global и nonlocal. 2](#_Toc124407519)

[Функции 2](#_Toc124407520)

[Чистота функций 6](#_Toc124407521)

[Побочные эффекты 8](#_Toc124407522)

[Области видимости 10](#_Toc124407523)

[LEGB 11](#_Toc124407524)

[global и nonlocal. 13](#_Toc124407525)

[**Инструкция global:** 13](#_Toc124407526)

[**Инструкция nonlocal:** 14](#_Toc124407527)

[**Обобщая сказанное:** 14](#_Toc124407528)

[1. 11. Функции. Документирование и тестирование функций. 16](#_Toc124407529)

[Функции 16](#_Toc124407530)

[Документирование и тестирование функций 20](#_Toc124407531)

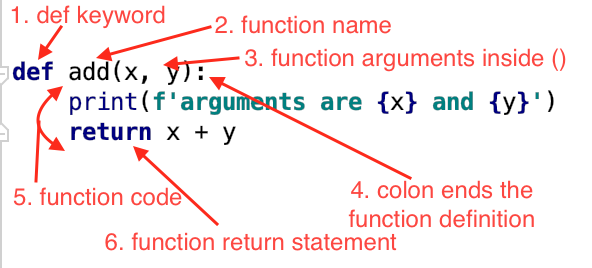
[**Как распознаются примеры строк документации?¶** 23](#_Toc124407532)

[2. 22.Оператор class и его работа. Классы и экземпляры. 24](#_Toc124407533)

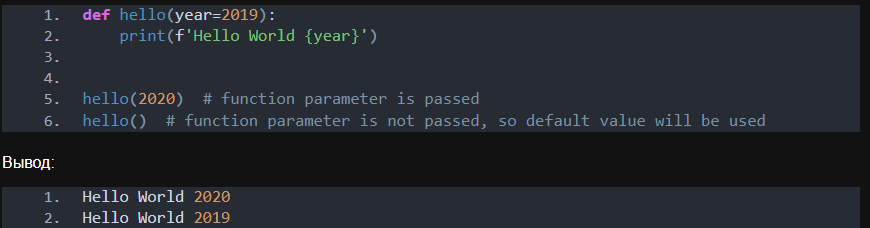
# 9.Функции. Чистота функций. Побочные эффекты. Области видимости. LEGB. global и nonlocal.

## Функции

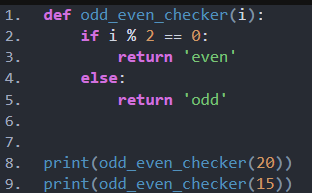
* Функция — это блок кода с именем.
* Мы можем вызвать функцию по ее имени.
* Код внутри функции запускается только при ее вызове.
* Может принимать данные от вызывающей программы, она вызывается как параметры функции.
* Параметры функции заключены в круглые скобки и разделены запятыми. Функция может принимать любое количество аргументов.
* Функция может возвращать данные вызывающей программе. В отличие от других популярных языков программирования, определение функций Python не указывает тип возвращаемого значения.
* Мы не можем использовать зарезервированные ключевые слова в качестве имени функции. Имя должно соответствовать правилам определения идентификаторов.



Python допускает значения по умолчанию для параметров функции. Если вызывающий абонент не передает параметр, используется значение по умолчанию.  
Источник: https://pythononline.ru/osnovy/funktsii-python

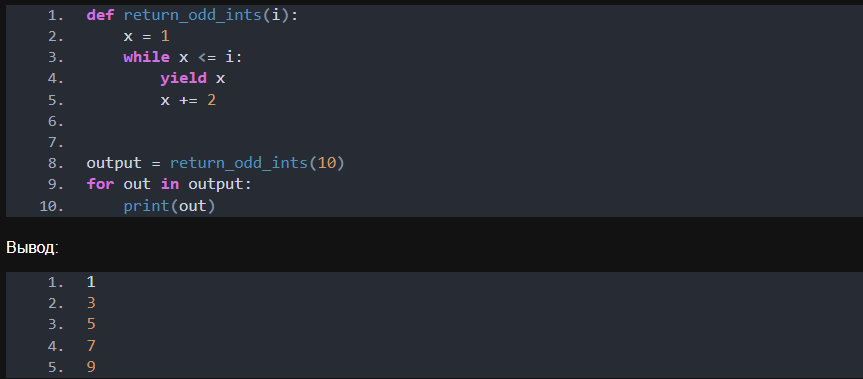


**Можем ли мы иметь несколько операторов возврата внутри функции?**



**Может ли функция Python возвращать несколько значений одно за другим?**

Функция Python может возвращать несколько значений одно за другим. Это реализовано с использованием ключевого слова **yield**. Это полезно, когда вы хотите, чтобы функция возвращала большое количество значений и обрабатывала их. Мы можем разделить возвращаемые значения на несколько частей, используя оператор **yield**. Этот тип функции также называется функцией генератора.

****

**Аргументы**

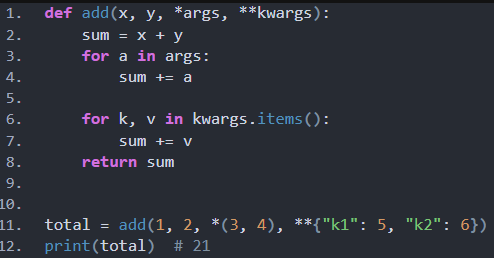
Python допускает три типа параметров в определении функции.

* Формальные аргументы: те, которые мы видели в примерах до сих пор.
* Переменное количество аргументов без ключевых слов: например, def add(\*args)
* Переменное количество аргументов ключевых слов или именованных аргументов: например, def add(\*\*kwargs)

Некоторые важные моменты относительно переменных аргументов в Python:

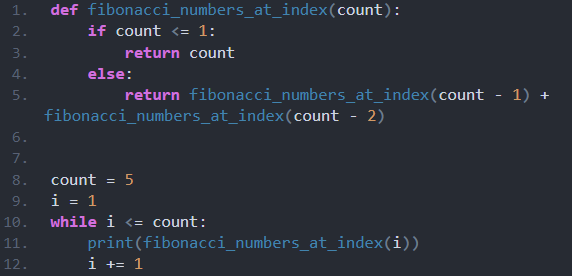
* Порядок аргументов должен быть формальным, \* args и \*\* kwargs.
* Не обязательно использовать имена параметров переменных как args и kwargs. Однако лучше всего использовать их для лучшей читаемости кода.
* Тип args — кортеж. Таким образом, мы можем передать кортеж для отображения с переменной \* args.
* Тип кваргов — дикт. Таким образом, мы можем передать словарь для сопоставления с переменной \*\* kwargs.

Вот простой пример.



Рекурсивная функция Python

Когда функция вызывает сама себя, она называется рекурсивной функцией. В программировании этот сценарий называется рекурсией. Вы должны быть очень осторожны при использовании рекурсии, потому что есть вероятность, что функция никогда не завершится и перейдет в бесконечный цикл. Вот простой пример печати ряда Фибоначчи с использованием рекурсии.



* **Сравнение функции с методом**
* Функция Python является частью файла сценария Python, в котором она определена, тогда как методы определены внутри определения класса.
* Мы можем вызвать функцию напрямую, если она находится в том же модуле. Если функция определена в другом модуле, мы можем импортировать модуль, а затем вызвать функцию напрямую. Нам нужен класс или объект класса для вызова методов.
* Функция Python может обращаться ко всем глобальным переменным, тогда как методы класса Python могут обращаться к глобальным переменным, а также к атрибутам и функциям класса.
* Тип данных функций Python — это «функция», а тип данных методов Python — «метод»

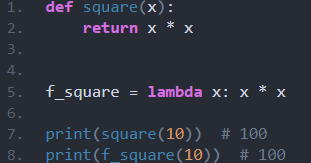
Давайте посмотрим на простой пример функций и методов в Python.

 Преимущества

* Возможность повторного использования кода, потому что мы можем вызывать одну и ту же функцию несколько раз
* Модульный код, поскольку мы можем определять разные функции для разных задач
* Улучшает ремонтопригодность кода
* Абстракция, поскольку вызывающему абоненту не нужно знать реализацию функции

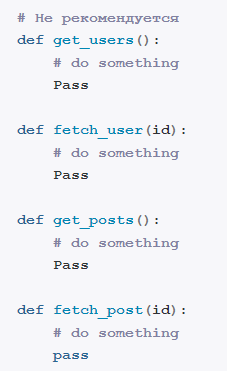
**Анонимная функция**

Анонимные функции не имеют имени. Мы можем определить анонимную функцию в Python, используя ключевое слово lambda.

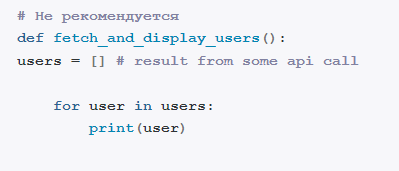
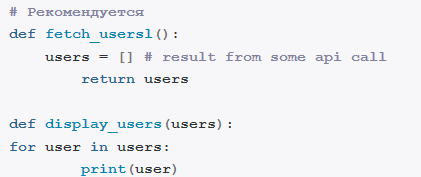
****

## Чистота функций

**При именовании функций строго придерживайтесь соглашения об именах.**Использование разных подходов будет путать читателя.

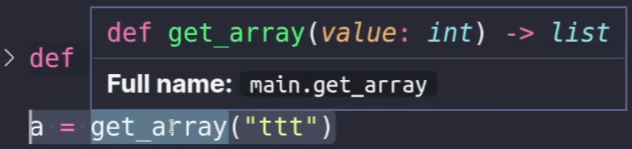


**Каждая функция должна выполнять какое-то одно действие и делать это хорошо.**Пишите короткие и простые функции, выполняющие одну задачу. Полезный прием: если в имени вашей функции появляется союз «и», лучше разделить такую функцию на две разных.

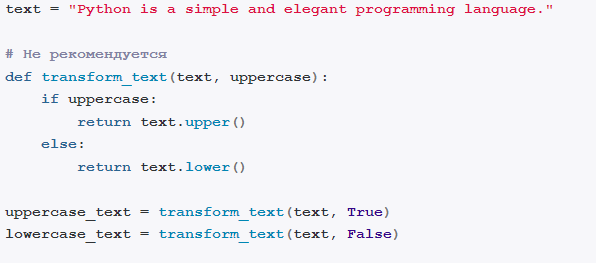
 

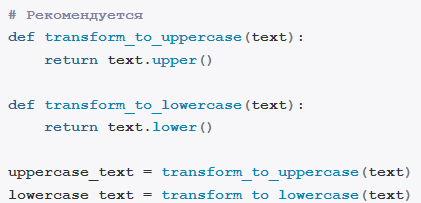
Type **annotations** для функций и внимательно отслеживаем несоответствия по типам данных на уровне проекта. Очень много херни позволяет отловить с минимальной задержкой.



.

**Не используйте Boolean flags.** Это переменные, содержащие булевы значения — True или False. Они передаются в функцию для определения ее поведения.





**Докстринги**, в которых описываемых всё, что функция принципиально делает. Комментарии - только там, где надо пояснить какое-то неочевидное решение, из-за, например, изврата бизнес-логики.

## Побочные эффекты

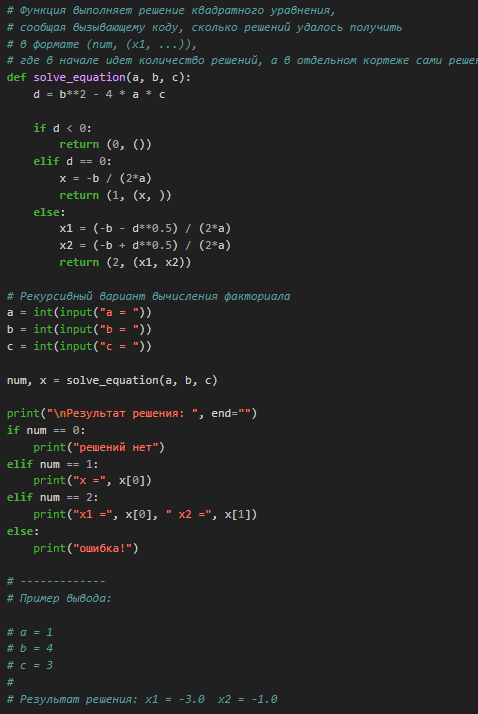
[Побочный эффект](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B1%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (*англ.* Side Effect) - любые действия программы, изменяющие среду выполнения (*англ.* Execution Environment).

К побочным эффектам выполнения функции можно отнести:

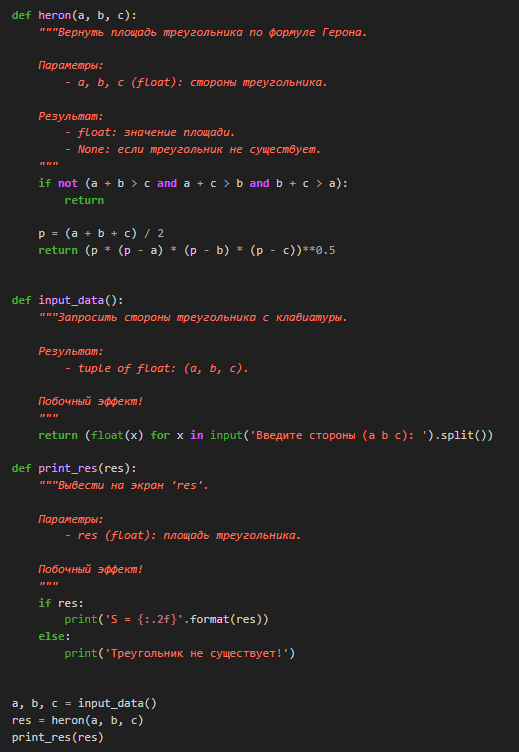
* изменение данных в памяти;
* чтение/запись файла или устройства;
* ввод/вывод значений;
* самостоятельную реакцию на исключительные ситуации;
* и др.

Часто функции с побочным эффектом при вызове несколько раз с одними и теми же аргументами в качестве результата возвращают разные значения. Пример - стандартная функция генерации случайного числа.

Создавая функцию, необходимо избегать побочных эффектов - такие функции легче тестируются и не содержат скрытых действий. Один из примеров такой функции - функция **solve\_equation**() в Листинге 5.1.12.



Естественно, что полностью избежать побочных эффектов невозможно. В таких случаях необходимо локализовать участки кода с побочным эффектом в отдельные функции (Листинге 5.1.16).



## Области видимости

Некоторые переменные скрипта могут быть недоступны некоторым областям программы. Все зависит от того, где вы объявили эти переменные.

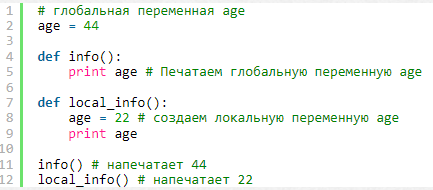
В **Python** две базовых области видимости переменных:

* *Глобальные переменные*
* *Локальные переменные*

Переменные объявленные внутри тела функции имеют локальную область видимости, те что объявлены вне какой-либо функции имеют глобальную область видимости.

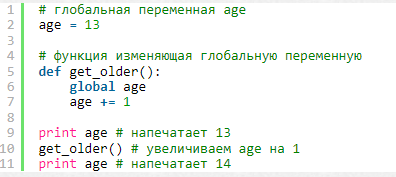
Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

Например:



Важно помнить, что для того чтобы получить доступ к глобальной переменной, достаточно лишь указать ее имя. Однако, если перед нами стоит задача изменить глобальную переменную внутри функции - необходимо использовать ключевое слово **global**.

Например :



## LEGB

КРАТКО--🡪<https://github.com/still-coding/se_python_snippets/blob/main/02_functions/f00_basics_legb.py>

LEGB расшифровывается как «Локальный», «Замкнутый», «Глобальный», «Встроенный». Это возможные области действия переменной в Python. Мы также используем эти слова для описания **пространств имен**, в которых хранятся имена и значения переменных, а также то, как Python определяет область видимости.

Порядок **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in — это порядок приоритета для разрешения области. Это означает, что Python по очереди ищет значение имени в каждом из этих пространств имен.

**Правило LEGB**

Теперь вернемся к правилу LEGB. Каждая буква обозначает разную область видимости.

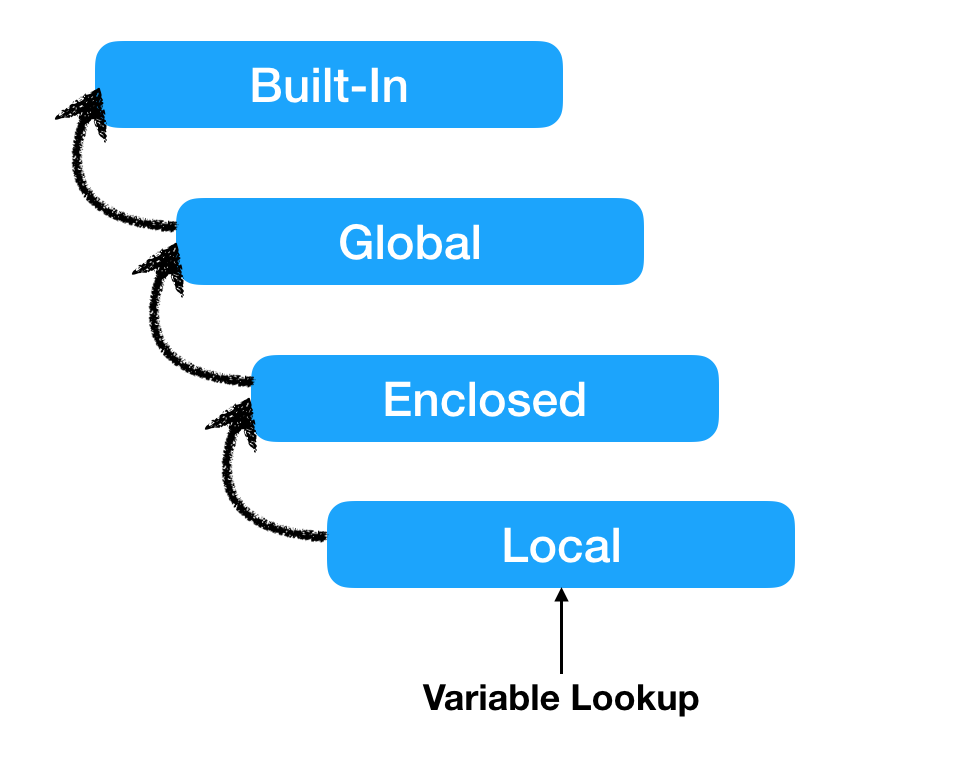
* **local**область видимости — это область действия всех объектов в текущем пространстве имен.
* **охватывающая**область — это область любых объектов в пространстве имен, охватывающих текущее пространство имен, которые могут или не могут существовать. В приведенном ниже примере пространство имен f включает пространство имен g , поэтому имя x находится во включающем область действия в строке 3.

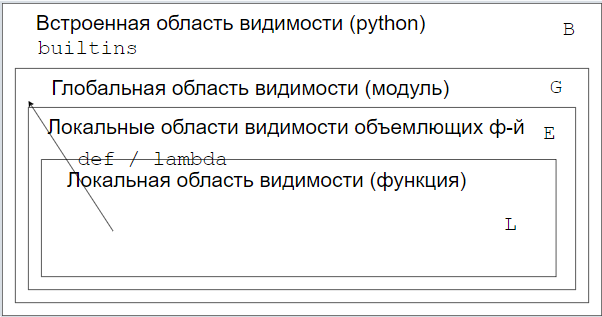
def f (x): def g (y): return yg (x)

* **global**область видимости — это область объектов в пространстве имен уровня модуля. Создание переменной вне функции или класса поместит ее в глобальное пространство имен.
* **Встроенная**область видимости — это объем всех объектов, встроенных в язык Python. В этом пространстве имен находятся зарезервированные имена, такие как dict.

При поиске объекта по имени Python сначала выполняет поиск в локальном пространстве имен. . Если имя отсутствует в локальном пространстве имен, Python выполняет поиск во всех включающих пространствах имен, начиная с самого внутреннего включающего пространства имен. Когда больше нет закрывающих пространств имен для поиска, Python выполняет поиск в глобальном пространстве имен. Если имя объекта не определено ни в одном из предыдущих пространств имен, Python выполняет поиск во встроенном пространстве имен. Наконец, если Python не может найти переменную с запрошенным именем в любом пространстве имен, он вызовет исключение NameError .

Итак, Python выполняет поиск изнутри наружу.. Вывод состоит в том, что пространства имен **изолированы**. Создание, удаление или изменение значения объекта в одном пространстве имен не повлияет на объекты с таким же именем в других пространствах имен.





Тут примеры, если нужно--🡪 <https://nanomode.ru/coding2/pravilo-legb-v-python/>

## global и nonlocal.

### **Инструкция global:**

[Инструкция global](https://docs-python.ru/tutorial/opredelenie-funktsij-python/operatory-global-nonlocal/) это объявление, которое выполняется для всего текущего блока кода. Это означает, что перечисленные в инструкции переменные должны интерпретироваться как глобальные. [Свободные переменные](https://docs-python.ru/tutorial/opredelenie-funktsij-python/oblast-vidimosti-peremennyh/) могут ссылаться на глобальные переменные с помощью global, не будучи объявленными в глобальной области.

Имена, перечисленные в инструкции global, не должны использоваться в блоке кода, текстуально предшествующем global.

Имена, перечисленные в инструкции global, не должны быть определены как [аргументы функции](https://docs-python.ru/tutorial/opredelenie-funktsij-python/parametry-argumenty-funktsii/) или использоваться в целевом элементе item управления циклом [for item in ...](https://docs-python.ru/tutorial/tsikly-upravlenie-vetvleniem-python/tsikl-for-in/" \o "Цикл for ... in ...: в Python.). Не должны быть [связаны с именем](https://docs-python.ru/tutorial/struktura-programmy-python/operatsija-svjazyvanija-imen-obektami/) в [определении класса](https://docs-python.ru/tutorial/klassy-jazyke-python/opredelenie-klassov/), [определении функции](https://docs-python.ru/tutorial/opredelenie-funktsij-python/), [оператора import](https://docs-python.ru/tutorial/sistema-importa-python/spetsifikatsija-instruktsii-import/) или аннотации переменной.

Инструкция global, содержащийся в строке или объекте кода, переданном во [встроенную функцию exec()](https://docs-python.ru/tutorial/vstroennye-funktsii-interpretatora-python/funktsija-exec/), не влияет на блок кода, содержащий вызов функции, и на код, содержащийся в такой строке, не влияют операторы global в коде, содержащем вызов функции. То же самое относится к функциям [eval()](https://docs-python.ru/tutorial/vstroennye-funktsii-interpretatora-python/funktsija-eval/" \o "Функция eval() в Python, выполняет строку-выражение с кодом.) и [compile()](https://docs-python.ru/tutorial/vstroennye-funktsii-interpretatora-python/funktsija-compile/" \o "Функция compile() компилирует блок кода Python.).

### **Инструкция nonlocal:**

[Инструкция nonlocal](https://docs-python.ru/tutorial/opredelenie-funktsij-python/operatory-global-nonlocal/) вызывает перечисленные имена для обозначения ранее определенных переменных в ближайшей области видимости, исключая глобальную. Это важно потому, что интерпретатор сначала выполняет поиск в локальном пространстве имен. Оператор позволяет инкапсулированному коду [связывать переменные](https://docs-python.ru/tutorial/struktura-programmy-python/operatsija-svjazyvanija-imen-obektami/) вне локальной области, кроме глобальной (модульной) области.

Имена, перечисленные в инструкции nonlocal, в отличие от тех, что перечислены в инструкции global, должны ссылаться на ранее существовавшие переменные в охватывающей области.

Имена, перечисленные в выражении nonlocal, не должны вступать в противоречие с ранее существующими переменными, определенными локальной области видимости.

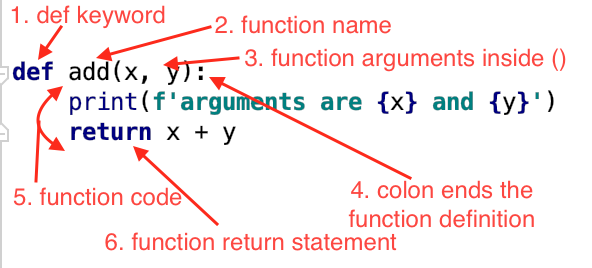
### **Обобщая сказанное:**

* операторы global и nonlocal используются в области видимости той [функции](https://docs-python.ru/tutorial/opredelenie-funktsij-python/), где будет требуется изменение переменной;
* оператор global можно использовать как в простых функциях, так и во вложенных;
* оператор nonlocal используются только во вложенных функциях;
* с помощью оператора global можно определить глобальную переменную из области видимости функции, которая ранее отсутствовала;
* при использовании оператора nonlocal, переменные родительской функции должны существовать.

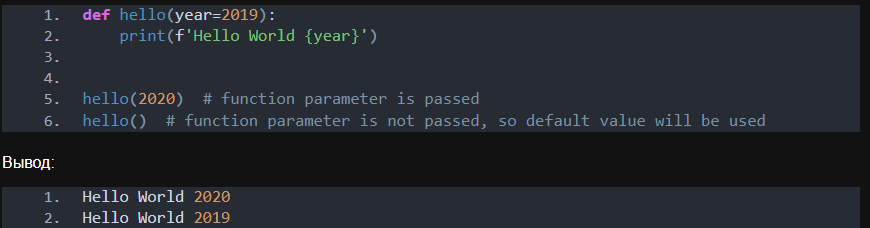
# 11. Функции. Документирование и тестирование функций.

## Функции

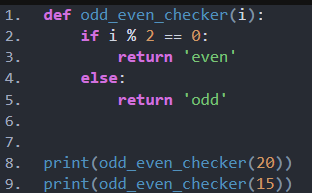
* Функция — это блок кода с именем.
* Мы можем вызвать функцию по ее имени.
* Код внутри функции запускается только при ее вызове.
* Может принимать данные от вызывающей программы, она вызывается как параметры функции.
* Параметры функции заключены в круглые скобки и разделены запятыми. Функция может принимать любое количество аргументов.
* Функция может возвращать данные вызывающей программе. В отличие от других популярных языков программирования, определение функций Python не указывает тип возвращаемого значения.
* Мы не можем использовать зарезервированные ключевые слова в качестве имени функции. Имя должно соответствовать правилам определения идентификаторов.



Python допускает значения по умолчанию для параметров функции. Если вызывающий абонент не передает параметр, используется значение по умолчанию.  
Источник: https://pythononline.ru/osnovy/funktsii-python

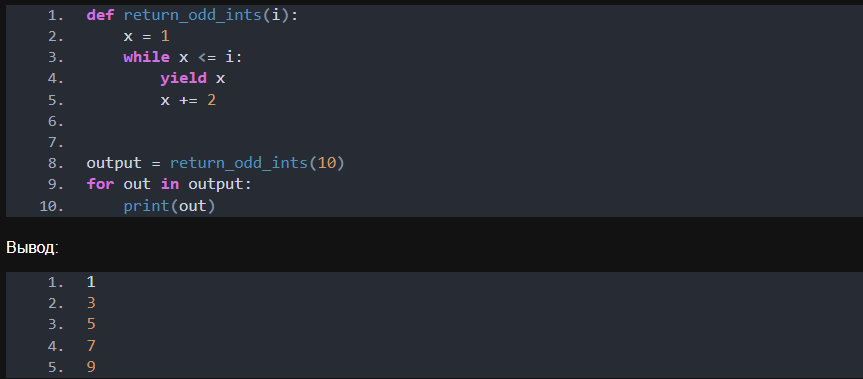


**Можем ли мы иметь несколько операторов возврата внутри функции?**



**Может ли функция Python возвращать несколько значений одно за другим?**

Функция Python может возвращать несколько значений одно за другим. Это реализовано с использованием ключевого слова **yield**. Это полезно, когда вы хотите, чтобы функция возвращала большое количество значений и обрабатывала их. Мы можем разделить возвращаемые значения на несколько частей, используя оператор **yield**. Этот тип функции также называется функцией генератора.

****

**Аргументы**

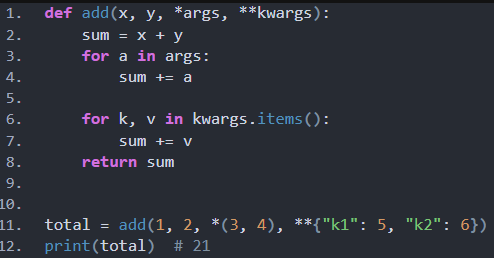
Python допускает три типа параметров в определении функции.

* Формальные аргументы: те, которые мы видели в примерах до сих пор.
* Переменное количество аргументов без ключевых слов: например, def add(\*args)
* Переменное количество аргументов ключевых слов или именованных аргументов: например, def add(\*\*kwargs)

Некоторые важные моменты относительно переменных аргументов в Python:

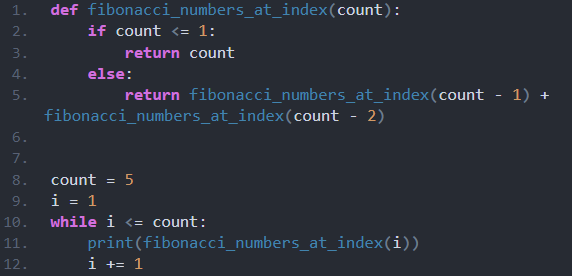
* Порядок аргументов должен быть формальным, \* args и \*\* kwargs.
* Не обязательно использовать имена параметров переменных как args и kwargs. Однако лучше всего использовать их для лучшей читаемости кода.
* Тип args — кортеж. Таким образом, мы можем передать кортеж для отображения с переменной \* args.
* Тип кваргов — дикт. Таким образом, мы можем передать словарь для сопоставления с переменной \*\* kwargs.

Вот простой пример.



Рекурсивная функция Python

Когда функция вызывает сама себя, она называется рекурсивной функцией. В программировании этот сценарий называется рекурсией. Вы должны быть очень осторожны при использовании рекурсии, потому что есть вероятность, что функция никогда не завершится и перейдет в бесконечный цикл. Вот простой пример печати ряда Фибоначчи с использованием рекурсии.



* **Сравнение функции с методом**
* Функция Python является частью файла сценария Python, в котором она определена, тогда как методы определены внутри определения класса.
* Мы можем вызвать функцию напрямую, если она находится в том же модуле. Если функция определена в другом модуле, мы можем импортировать модуль, а затем вызвать функцию напрямую. Нам нужен класс или объект класса для вызова методов.
* Функция Python может обращаться ко всем глобальным переменным, тогда как методы класса Python могут обращаться к глобальным переменным, а также к атрибутам и функциям класса.
* Тип данных функций Python — это «функция», а тип данных методов Python — «метод»

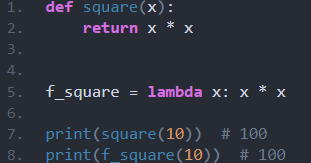
Давайте посмотрим на простой пример функций и методов в Python.

 Преимущества

* Возможность повторного использования кода, потому что мы можем вызывать одну и ту же функцию несколько раз
* Модульный код, поскольку мы можем определять разные функции для разных задач
* Улучшает ремонтопригодность кода
* Абстракция, поскольку вызывающему абоненту не нужно знать реализацию функции

**Анонимная функция**

Анонимные функции не имеют имени. Мы можем определить анонимную функцию в Python, используя ключевое слово lambda.

****

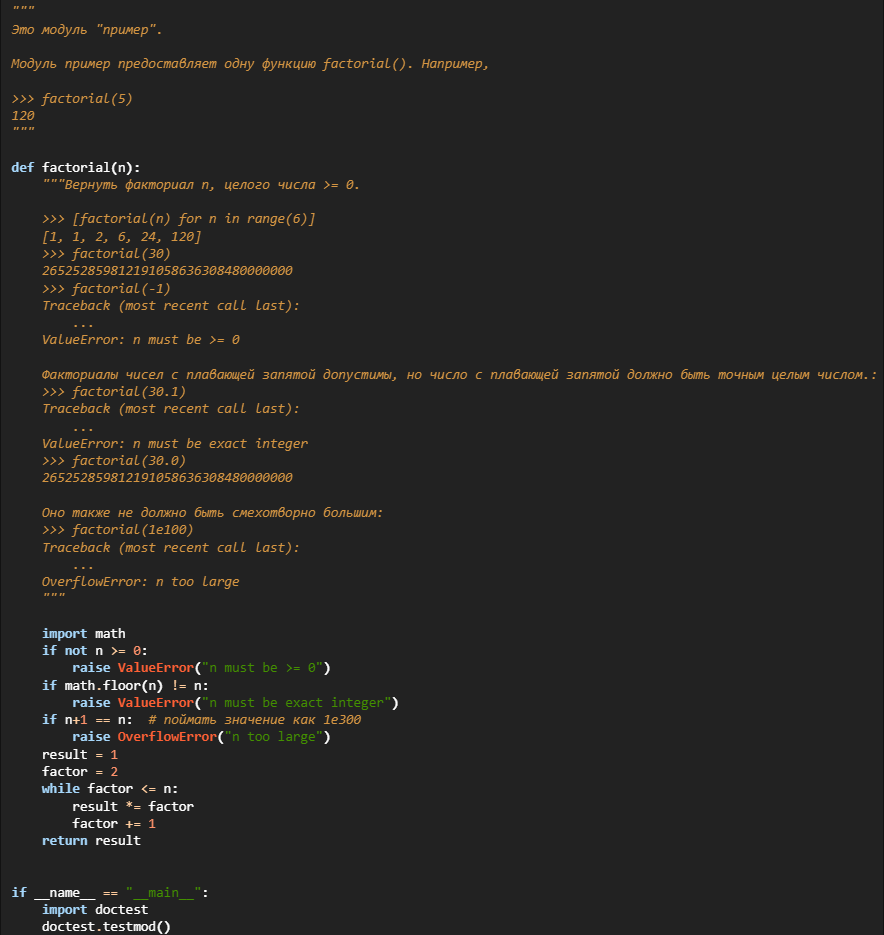
## Документирование и тестирование функций

ТУТ ПОБРОБНЕЙ И С ПРИМЕРАМИ, НО ТЕКСТА ДО ПИЗДЫ---🡪<https://digitology.tech/docs/python_3/library/doctest.html>

Модуль **[doctest](https://digitology.tech/docs/python_3/library/doctest.html" \l "module-doctest" \o "doctest: Тестовые фрагменты кода в строках документации.)** ищет фрагменты текста, которые выглядят как интерактивные сеансы Python, а затем выполняет данные сеансы, чтобы убедиться, что они работают именно так, как показано. Существует несколько распространенных способов использования доктест:

* Проверка актуальности строк документации модуля путём проверки того, что все интерактивные примеры по-прежнему работают в соответствии с документацией.
* Чтобы выполнить регрессионное тестирование, убедившись, что интерактивные примеры из тестового файла или тестового объекта работают должным образом.
* Написать учебную документацию для пакета, обильно иллюстрированную примерами ввода-вывода. В зависимости от того, выделены ли примеры или пояснительный текст, это может быть что-то вроде «грамотного тестирования» или «исполняемой документации».

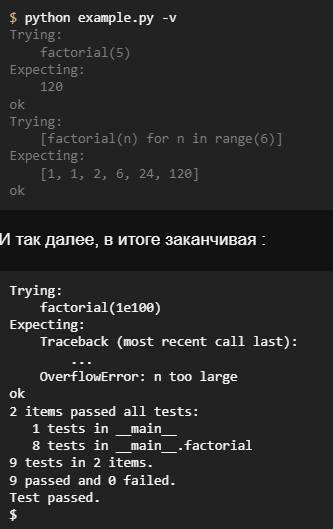
Вот полный, но небольшой пример модуля:



Если вы запустите example.py прямо из командной строки, [doctest](https://digitology.tech/docs/python_3/library/doctest.html" \l "module-doctest" \o "doctest: Тестовые фрагменты кода в строках документации.) творит чудеса:



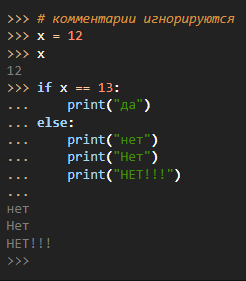
Вывода нет! Это нормально, и это означает, что все примеры рабочие. Передайте **-v**сценарию, и [doctest](https://digitology.tech/docs/python_3/library/doctest.html" \l "module-doctest" \o "doctest: Тестовые фрагменты кода в строках документации.) напечатает подробный log своих попыток, а в конце напечатает сводку:



Это всё, что вам нужно знать, чтобы начать продуктивно использовать doctest! Приступайте к делу. В следующих разделах приведены подробные сведения. Обратите внимание, что в стандартном наборе тестов и библиотеках Python есть много примеров доктестов. Особенно полезные примеры можно найти в стандартном тестовом файле Lib/test/test\_doctest.py.

### **Как распознаются примеры строк документации?**[**¶**](https://digitology.tech/docs/python_3/library/doctest.html#doctest-finding-examples)

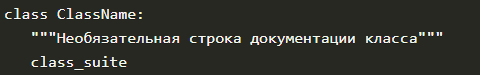
В большинстве случаев копирование и вставка сеанса интерактивной консоли работает нормально, но doctest не пытается выполнить точную эмуляцию какой-либо оболочки Python.



Любой ожидаемый вывод должен следовать сразу за последней строкой '>>> ' или '... ', содержащей код, а ожидаемый вывод (если есть) распространяется на следующую строку '>>> ' или строку, состоящую только из пробелов.

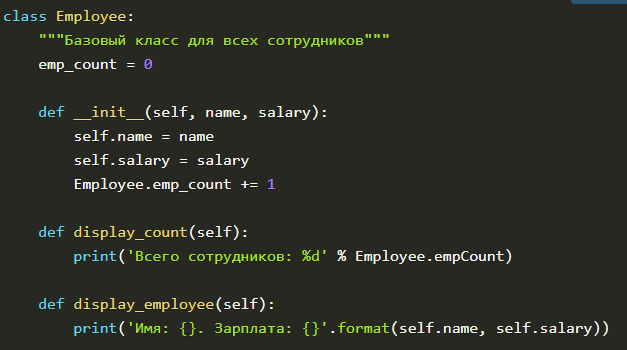
# 22.Оператор class и его работа. Классы и экземпляры.

Оператор class создает новое определение класса. Имя класса сразу следует за ключевым словом class, после которого ставиться двоеточие:



* У класса есть строка документации, к которой можно получить доступ через ClassName.\_\_doc\_\_.
* class\_suite состоит из частей класса, атрибутов данных и функции.

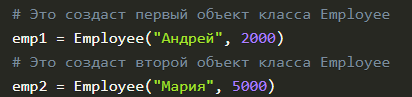
**Пример создания класса на Python**:



* Переменная emp\_count — переменная класса, значение которой разделяется между экземплярами этого класса. Получить доступ к этой переменной можно через Employee.emp\_count из класса или за его пределами.
* Первый метод \_\_init\_\_() — специальный метод, который называют конструктором класса или методом инициализации. Его вызывает Python при создании нового экземпляра этого класса.
* Объявляйте другие методы класса, как обычные функции, за исключением того, что первый аргумент для каждого метода self. Python добавляет аргумент self в список для вас; и тогда вам не нужно включать его при вызове этих методов.

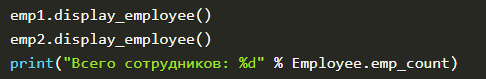
**Создание экземпляров класса**

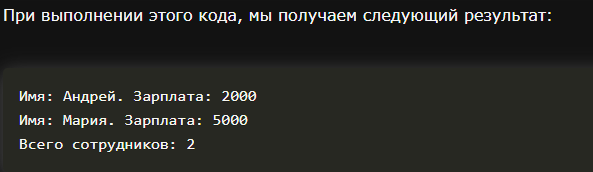
Чтобы создать экземпляры классов, нужно вызвать класс с использованием его имени и передать аргументы, которые принимает метод \_\_init\_\_.

****

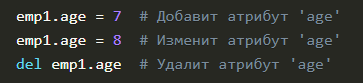
Доступ к **атрибутам**

Получите доступ к атрибутам класса, используя оператор . после объекта класса. Доступ к классу можно получить используя имя переменой класса:



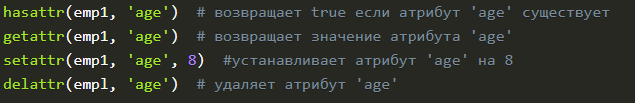


Вы можете добавлять, удалять или изменять атрибуты классов и объектов в любой момент.



Вместо использования привычных операторов для доступа к атрибутам вы можете использовать эти функции:

* getattr(obj, name [, default]) — для доступа к атрибуту объекта.
* hasattr(obj, name) — проверить, есть ли в obj атрибут name.
* setattr(obj, name, value) — задать атрибут. Если атрибут не существует, он будет создан.
* delattr(obj, name) — удалить атрибут.

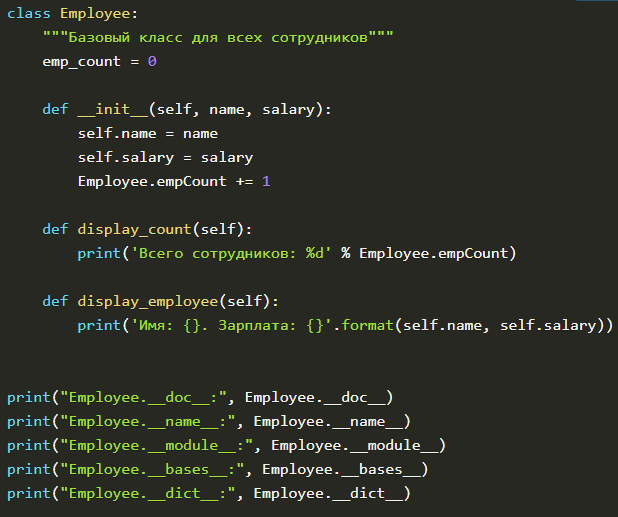


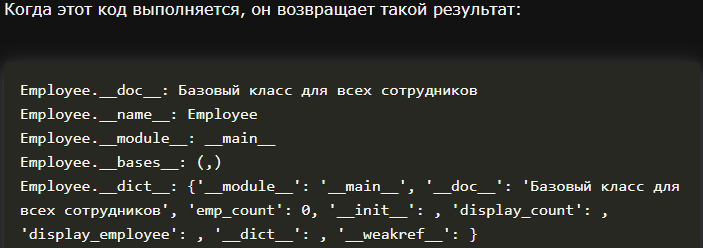
**Встроенные атрибуты класса**

Каждый класс Python хранит встроенные атрибуты, и предоставляет к ним доступ через оператор ., как и любой другой атрибут:

* \_\_dict\_\_ — словарь, содержащий пространство имен класса.
* \_\_doc\_\_ — строка документации класса. None если, документация отсутствует.
* \_\_name\_\_ — имя класса.
* \_\_module\_\_ — имя модуля, в котором определяется класс. Этот атрибут \_\_main\_\_ в интерактивном режиме.
* \_\_bases\_\_ — могут быть пустые tuple, содержащие базовые классы, в порядке их появления в списке базового класса.

Для вышеуказанного класса давайте попробуем получить доступ ко всем этим атрибутам:



****