Оглавление

[*Декорирование функции с аргументами* **(возможно это немного уже дополнительно)** 12](#_Toc125090512)

[**4.** Последовательности. Операции над последовательностями 1](#_Toc125090513)

[**17.** Генераторы и включения. Генераторные функции. Генераторные выражения 3](#_Toc125090514)

[Разница между атомарными и структурными типами данных 7](#_Toc125090515)

[Числовые типы 8](#_Toc125090516)

[**int (целое число)** 8](#_Toc125090517)

[**float (число с плавающей точкой)** 8](#_Toc125090518)

[**complex (комплексное число)** 9](#_Toc125090519)

[**bool (логический тип данных)** 9](#_Toc125090520)

[Последовательности 10](#_Toc125090521)

[**str (строка)** 10](#_Toc125090522)

[**list (список)** 10](#_Toc125090523)

[**tuple (кортеж)** 11](#_Toc125090524)

[dict (словарь) 11](#_Toc125090525)

[set (множество) 11](#_Toc125090526)

[**Что такое функции map(), filter() и reduce() в Python?** 12](#_Toc125090527)

[**Функция map():** 12](#_Toc125090528)

[**Совместное использование с функциями, определяемыми пользователем и Lambda-функциями:** 12](#_Toc125090529)

[Пользовательские функции совместно с map(): 12](#_Toc125090530)

[Lambda-функции совместно с map(): 13](#_Toc125090531)

[**Функция filter():** 14](#_Toc125090532)

[Использование lambda-функций совместно с filter(): 14](#_Toc125090533)

[**Функция reduce():** 15](#_Toc125090534)

[Исключения в языках программирования 16](#_Toc125090535)

[Ошибки и исключения в Python 16](#_Toc125090536)

[**Синтаксические ошибки в Python** 16](#_Toc125090537)

[**Исключения в Python** 16](#_Toc125090538)

[Иерархия исключений в Python 17](#_Toc125090539)

[Обработка исключений в Python 18](#_Toc125090540)

[Генерация исключений в Python 18](#_Toc125090541)

[Пользовательские исключения (User-defined Exceptions) в Python 19](#_Toc125090542)

[**1. Язык Python - общая характеристика, преимущества и недостатки, краткая история. PEP’ы.** 1](#_Toc125090543)

[Общая характеристика 1](#_Toc125090544)

[Преимущества 1](#_Toc125090545)

[Недостатки 2](#_Toc125090546)

[Краткая история 2](#_Toc125090547)

[PEP’ы 2](#_Toc125090548)

[**15. Декораторы.** 3](#_Toc125090549)

[**30. Диспетчеры контекстов. Протокол управление контекстами.** 4](#_Toc125090550)

[Протокол контекстного менеджера 5](#_Toc125090551)

[**Объекты и классы** 23](#_Toc125090552)

[**9.Функции. Чистота функций. Побочные эффекты. Области видимости. LEGB. global и nonlocal.** 29](#_Toc125090553)

[**Функции** 29](#_Toc125090554)

[**Чистота функций** 33](#_Toc125090555)

[**Побочные эффекты** 35](#_Toc125090556)

[**Области видимости** 37](#_Toc125090557)

[**LEGB** 38](#_Toc125090558)

[**global и nonlocal.** 40](#_Toc125090559)

[**Инструкция global:** 40](#_Toc125090560)

[**Инструкция nonlocal:** 41](#_Toc125090561)

[**Обобщая сказанное:** 41](#_Toc125090562)

[**11. Функции. Документирование и тестирование функций.** 43](#_Toc125090563)

[**Функции** 43](#_Toc125090564)

[**Документирование и тестирование функций** 47](#_Toc125090565)

[**Как распознаются примеры строк документации?¶** 50](#_Toc125090566)

[**22.Оператор class и его работа. Классы и экземпляры.** 51](#_Toc125090567)

[Билет 8. Итерации и включения. Протокол итерации 1](#_Toc125090568)

[⮚ **Создание пользовательских итераторов** 1](#_Toc125090569)

[⮚ **Бесконечные итераторы Python** 1](#_Toc125090570)

[Билет 19. Импортирование. Способы и их работа. 3](#_Toc125090571)

***7.*** Циклы

Выполнение программ, написанных на любом языке программирования, по умолчанию является последовательным. Иногда нам может понадобиться изменить выполнение программы. Выполнение определенного кода может потребоваться повторить несколько раз.

Для этого предусмотрены различные типы циклов, которые способны повторять определенный код несколько раз.

Циклы упрощают сложные задачи до простых. Он позволяет нам изменить поток программы таким образом, что вместо того, чтобы писать один и тот же код снова и снова, мы можем повторять его конечное число раз.

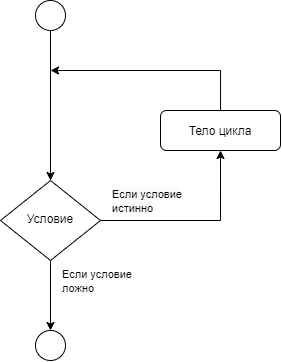
В Python преимущества циклов, как и в других языках программирования, заключаются в следующем:

1. Это обеспечивает возможность повторного использования кода;
2. Используя циклы, нам не нужно писать один и тот же код снова и снова;
3. С помощью циклов мы можем перебирать элементы структур данных (массивов или связанных списков).

В Python существуют следующие операторы циклов:

1. for;
2. while.

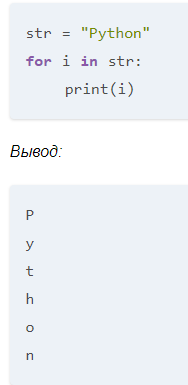
**Цикл for**

Цикл *for* в Python используется для многократного повторения операторов или части программы. Он часто используется для обхода структур данных, таких как список, кортеж или словарь.

Синтаксис цикла:

**for** iterating\_var **in** sequence:

statement(s)

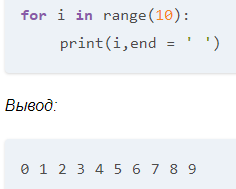
**Цикл For с использованием последовательности**

**Цикл For с использованием функции range()**

Функция *range()* используется для генерации последовательности чисел. Если мы передадим *range(10)*, она сгенерирует числа от 0 до 9. Синтаксис функции:

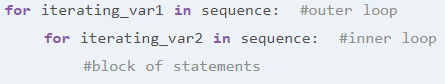
range(start,stop,step size)

* Start означает начало итерации.
* Stop означает, что цикл будет повторяться до *stop-1*. *range(1,5)* будет генерировать числа от *1* до *4* итераций. Это необязательный параметр.
* Размер шага используется для пропуска определенных чисел в итерации. Его использование необязательно. По умолчанию размер шага равен *1*.



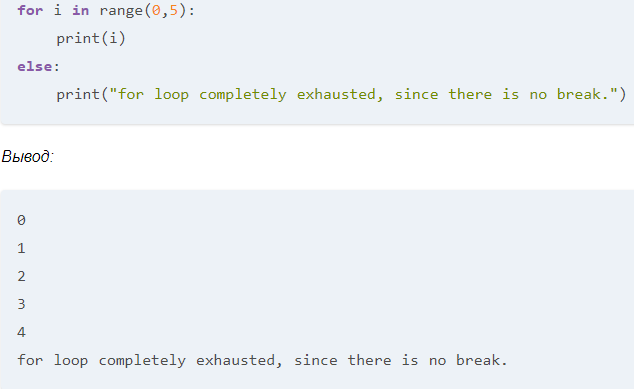
**Вложенный цикл For**

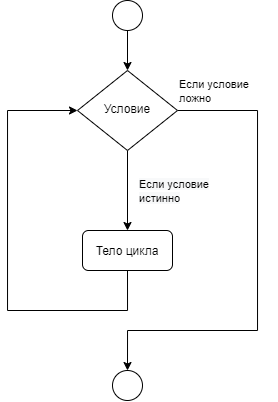
Python позволяет нам вложить любое количество циклов *for* внутрь цикла *for.* Внутренний цикл выполняется *n* раз за каждую итерацию внешнего цикла. Синтаксис:



**Использование оператора else в цикле for**

Python позволяет нам использовать оператор *else* с циклом *for*, который может быть выполнен только тогда, когда все итерации исчерпаны.



**Цикл while**

Цикл *while* позволяет выполнять часть кода до тех пор, пока заданное условие не станет ложным. Он также известен как цикл с предварительной проверкой условия.

Синтаксис цикла:

**while** expression:

statements

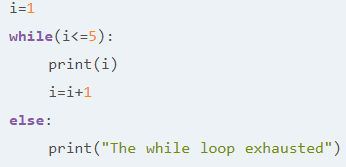
Здесь утверждения могут быть одним утверждением или группой утверждений. Выражение должно быть любым допустимым выражением Python, приводящим к true или false. Истиной является любое *ненулевое* значение, а ложью - 0.

**Операторы управления циклом**

1. Оператор *continue* - Когда встречается оператор *continue*, управление переходит в начало цикла;
2. Оператор *break* - Когда встречается оператор *break*, он выводит управление из цикла;
3. Оператор *pass* - Оператор pass используется для объявления пустого цикла. Он также используется для определения пустого класса, функции и оператора управления.

**Использование оператора else в цикле while**

Python позволяет нам также использовать оператор *else* с циклом *while*. Блок *else* выполняется, когда условие, заданное в операторе *while*, становится ложным.



***8. Функции первого класса. Функции в качестве аргументов. Возвращаемые функции Вложенные функции***

**Функции**

Функция — это группа связанных инструкций, выполняющих определенную задачу.

Функции помогают разбить нашу программу на более мелкие части. По мере того, как наша программа становится все больше и больше, функции делают ее более организованной и управляемой. Кроме того, функцию можно вызвать из различных мест программы, что позволяет избежать повторения программного кода.

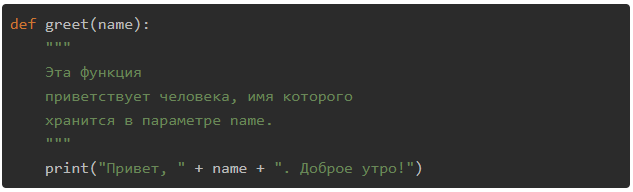
Синтаксис функции:

def имя\_функции(аргументы):

операторы

Компоненты, которые включает в себя определение функции:

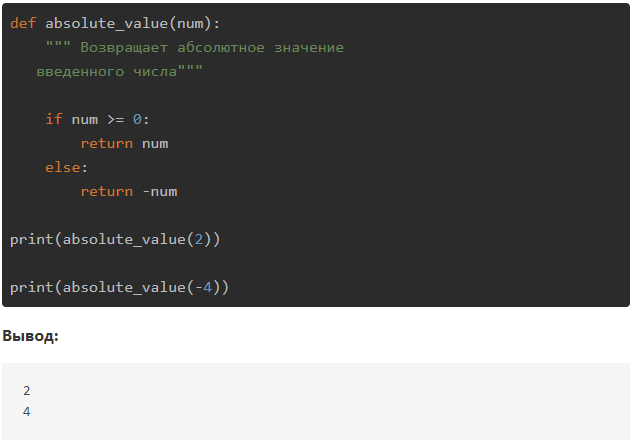
1. Ключевое слово *def* — начало заголовка функции.
2. Имя функции — для однозначной идентификации функции.
3. С помощью параметров (аргументов) мы передаем значения в функцию. Аргументов может и не быть.
4. Двоеточие *:* обозначает конец заголовка функции.
5. Необязательная строка документации нужна для описания того, что делает функция.
6. Один или несколько операторов Python составляют тело функции. Все инструкции должны иметь одинаковый отступ (4 пробела или 1 TAB).
7. Оператор *return* возвращает переданное значение из функции. Он необязателен.

Пример функции:

После того, как мы определили функцию, мы можем вызвать ее в программе или даже из командной строки Python. Чтобы вызвать функцию, мы просто вводим ее имя с соответствующими параметрами.



Оператор *return* используется для выхода из функции и возврата в то место, откуда она была вызвана. Оператор *return* может содержать выражение, которое возвращает значение. Если в операторе нет выражения или самого оператора возврата нет внутри функции, функция вернет объект *None*.

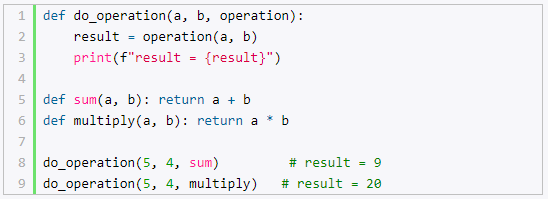
Пример возвращения значений:

**Время жизни переменной**

Время жизни переменной — это период, в течение которого переменная находится в памяти. Время жизни переменной внутри функции длится до тех пор, пока функция выполняется. Переменные «уничтожаются», когда мы выходим из функции. Это значит, что функция не помнит значения переменных из предыдущих вызовов.

**Передача функции в другие функции**

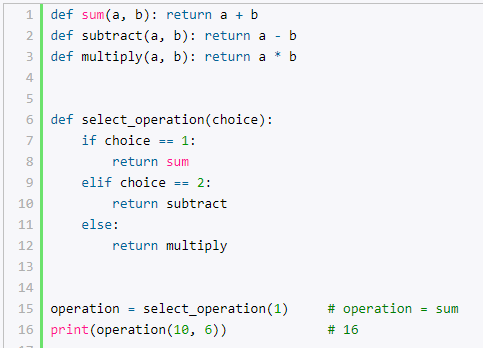
Функции, как и любой другой объект, могут быть переданы в качестве параметра (аргумента) другой функции. Например, определим функцию, которая выводит на консоль результат некоторой операции:



Функция do\_operation имеет три параметра, причем третий параметр будет представлять функцию, которая принимает два параметра и возвращает некоторый результат. Таким образом, получаются более гибкие по функциональности функции, которые через параметры принимают другие функции.

**Возвращаемые функции(Функция как результат другой функции)**

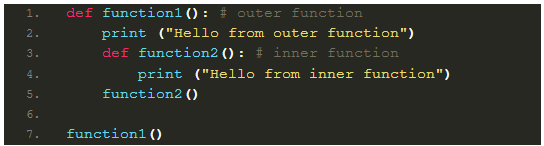
Также одна функция в Python может возвращать другую функцию. Например, можно определить функцию, которая в зависимости от значения параметра возвращает ту или иную операцию:



В данном случае функция select\_operation в зависимости от значения параметра choice возвращает одну из трех функций - sum, subtract и multiply.

**Вложенные (Внутренние) функции**

Чтобы определить внутреннюю функцию в Python, мы просто создаем функцию внутри другой, используя ключевое слово def. Вот пример:



Важно отметить, что внешняя функция должна быть вызвана для выполнения внутренней. Если внешняя функция не вызывается, внутренняя никогда не будет выполняться.

Функцию можно создать как внутреннюю, чтобы защитить ее от всего, что происходит вне функции. В этом случае функция будет скрыта от глобальной области видимости.

В заключении, внутренняя функция – это просто функция, которая определена внутри другой. Она может получить доступ к переменным, которые были определены в рамках внешней функции, но не может их изменить. Существует ряд причин, по которым нам может потребоваться создать внутреннюю функцию. Например, внутренняя функция защищена от того, что происходит вне ее. Внутренние функции также являются хорошим способом создания замыканий в Python.

24. **Создание экземпляров. Конструкторы, деструкторы, инициализаторы**

Процесс создания экземпляра класса запускается всякий раз, когда происходит вызов объекта класса Python. Процесс создания экземпляра состоит из двух отдельных шагов, которые можно описать следующим образом:

* Создание нового экземпляра целевого класса;
* Инициализация нового экземпляра с соответствующим начальным состоянием.

Для выполнения первого шага в классах Python есть специальный метод .\_\_new\_\_(), который отвечает за создание и возврат нового пустого объекта. Затем другой специальный метод .\_\_init\_\_(), принимает результирующий объект вместе с аргументами конструктора класса.

Метод .\_\_init\_\_() принимает новый объект в качестве первого аргумента self. Затем он устанавливает любой требуемый атрибут экземпляра в допустимое состояние, используя аргументы, переданные ему конструктором класса.

**Конструктор, инициализатор, деструктор**

Специальные методы вызываются при создании экземпляра класса (конструктор), при инициализировании экземпляра класса (инициализатор) и при удалении класса (деструктор). В языке Python реализовано автоматическое управление памятью, поэтому конструктор и деструктор требуются достаточно редко, для ресурсов, требующих явного освобождения.

Следующий класс имеет конструктор, инициализатор и деструктор:

**class** **Line**:

**def** \_\_new\_\_(cls): *# Конструктор*

**return** super(Line, cls).\_\_new\_\_(cls)

**def** \_\_init\_\_(self, p1, p2): *# Инициализатор*

self.line = (p1, p2)

**def** \_\_del\_\_(self): *# Деструктор*

print("Удаляется линия **%s** - **%s**" % self.line)

>>> l = Line((0.0, 1.0), (0.0, 2.0))

>>> **del** l

Удаляется линия (0.0, 1.0) - (0.0, 2.0)

>>>

В момент вызова деструктора (например, по завершении программы) среда исполнения может быть уже достаточно «истощенной» [Шаблон:Что](https://ru.wikibooks.org/w/index.php?title=%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD:%D0%A7%D1%82%D0%BE&action=edit&redlink=1), поэтому в деструкторе следует делать только самое необходимое. Кроме того, не обработанные в деструкторе исключения игнорируются.

*Билет 6. Условный оператор. Логические операции и логические сравнения.*

**Условные выражения**

Ряд операций представляют условные выражения. Все эти операции принимают два операнда и возвращают логическое значение, которое в Python представляет тип bool. Существует только два логических значения - True (выражение истинно) и False (выражение ложно).

**Операции сравнения**

Простейшие условные выражения представляют операции сравнения, которые сравнивают два значения. Python поддерживает следующие операции сравнения:

1. == Возвращает True, если оба операнда равны. Иначе возвращает False.
2. != Возвращает True, если оба операнда НЕ равны. Иначе возвращает False.
3. > (больше чем) Возвращает True, если первый операнд больше второго.
4. < (меньше чем) Возвращает True, если первый операнд меньше второго.
5. >= (больше или равно) Возвращает True, если первый операнд больше или равен второму.
6. <= (меньше или равно) Возвращает True, если первый операнд меньше или равен второму.

**Логические операции**

Для создания составных условных выражений применяются логические операции. В Python имеются следующие логические операторы:

* and (логическое умножение) Возвращает True, если оба выражения равны True
* or (логическое сложение) Возвращает True, если хотя бы одно из выражений равно True
* not (логическое отрицание) Возвращает True, если выражение равно False

Если один из операндов оператора and возвращает False, то другой операнд уже не оценивается, так как оператор в любом случае возвратит False. Подобное поведение позволяет немного увеличить производительность, так как не приходится тратить ресурсы на оценку второго операнда.

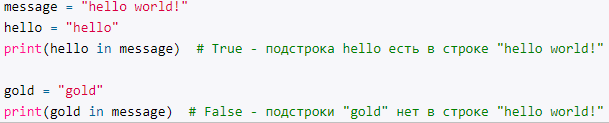
Аналогично если один из операндов оператора or возвращает True, то второй операнд не оценивается, так как оператор в любом случае возвратит True.

**Оператор in**

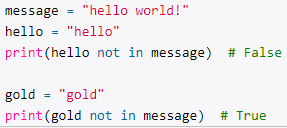
Оператор in возвращает True если в некотором наборе значений есть определенное значение. Он имеет следующую форму:

**значение in набор\_значений**

Например, строка представляет набор символов. И с помощью оператора in мы можем проверить, есть ли в ней какая-нибудь подстрока:

****

Если нам надо наоборот проверить, нет ли в наборе значений какого-либо значения, то мы може использовать модификацию оператора - not in. Она возвращает True, если в наборе значений НЕТ определенного значения:



*Билет 18. Модули и пакеты. Архитектура программы***.**

**Модули и пакеты** являются неотъемлемой частью модульногопрограммирования - организации программы как совокупности небольших независимых блоков, структура и поведение которых подчиняются определенным правилам.

Разработка программы как совокупности модулей позволяет:

* упростить задачи проектирования программы и распределения процесса разработки между группами разработчиков;
* предоставить возможность обновления (замены) модуля, без необходимости изменения остальной системы;
* упростить тестирование программы;
* упростить обнаружение ошибок.

Программный код часто разбивается на несколько файлов, каждый из которых используется отдельно от остальных. Одним из методов написания модульных программ является объектно-ориентированное программирование.

**Модуль** (англ. Module) - специальное средство языка программирования, позволяющее объединить вместе данные и функции и использовать их как одну функционально-законченную единицу (например, математический модуль, содержащий тригонометрические и прочие функции, константы π, ϵ и т.д.).

**Пакеты** (англ. Package) являются еще более крупной единицей и представляют собой набор взаимосвязанных модулей, предназначенных для решения задач определенного класса некоторой предметной области (например, пакет для решения систем уравнений, который может включать математический модуль, модуль со специальными типами данных и т.д.).

**Модуль** - отдельный файл с кодом на Python, содержащий функции и данные:

* имеет расширение \*.py (имя файла является именем модуля);
* может быть импортирован (подключен) (директива import ...);
* может быть многократно использован.

**Пакеты** в Python — это способ структуризации модулей. Пакет представляет собой папку, в которой содержатся модули и другие пакеты, и обязательный файл \_\_init.py\_\_, отвечающий за инициализацию пакета.

Одна из основных целей использования как модулей, так и пакетов - реализация модели пространства имен, позволяющей логически группировать и в то же время изолировать различные идентификаторы. Например, при наличии глобальной переменной author в модуле A и B не произойдет конфликта, т.к. они находятся в разном пространстве имен: A.author и B.author соответственно.

**Все модули/пакеты в Python можно разделить на 4 категории:**

1. **Встроенные** (англ. Built-in).

Модули, встроенные в язык и предоставляющие базовые возможности языка (написаны на языке Си).

К встроенным относятся как модули общего назначения (например, math или random), так и плаиформозависимые модули (например, модуль winreg, предназначенный для работы с реестром ОС Windows, устанавливается только на соответствующей ОС).

1. **Стандартная библиотека** (англ. Standard Library).

Модули и пакеты, написанные на Python, предоставляющие расширенные возможности, например, json или os.

1. **Сторонние** (англ. 3rd Party).

Модули и пакеты, которые не входят в дистрибутив Python, и могут быть установлены из каталога пакетов Python (англ. PyPI - the Python Package Index, более 90.000 пакетов) с помощью утилиты pip. При установке пакета автоматически устанавливаются зависимые пакеты.

1. **Пользовательские** (собственные).

Модули и пакеты, создаваемые разработчиком. В собственной программе рекомендуется выполнять импорт именно в таком порядке: от встроенных до собственных модулей/пакетов.

**Подключение и использование.**

Для использования модуля или пакета в коде необходимо его предварительно подключить (импортировать).

**import**

Импорт модуля или пакета выполняется единожды инструкцией import, располагаемой, как правило, в начале файла. Выполнить подключение модуля можно так: **import module\_1**

Пакет подключается аналогичным образом. Кроме того, имеется возможность импорта отдельных модулей из пакета, если нет необходимости использовать весь пакет: **import package\_1**

*Билет 25. Перегрузка операций***.**

Один и тот же оператор в Python по-разному ведет себя с разными типами. Например, оператор + в зависимости от типа операндов может складывать 2 числа, сливать 2 списка или объединять 2 строки. Когда-нибудь задумывались, почему так происходит? Дело в перегрузке операторов в Python.

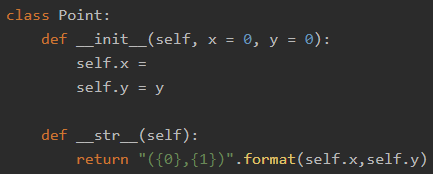
**Перегрузка оператора** — это возможность переопределять различные операторы в классах, то есть менять операции, которые они выполняют, в зависимости от контекста. Имена таких методов включают двойное подчеркивание спереди и сзади.

Под операторами в данном контексте понимаются не только знаки +, -, \*, /, обеспечивающие операции сложения, вычитания и др., но также специфика синтаксиса языка, обеспечивающая операции создания объекта, вызова объекта как функции, обращение к элементу объекта по индексу, вывод объекта и другое.

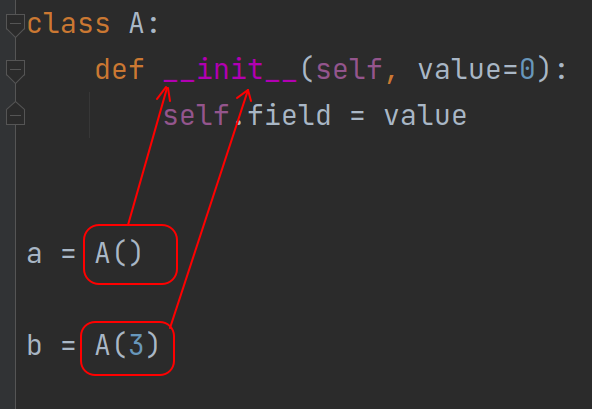
**Специальные функции**

Специальные функции классов начинаются с двойного подчеркивания \_\_. Их еще называют магическими методами. Это не обычные методы, которые мы определяем в классе. В коде выше мы уже использовали специальную функцию \_\_init\_\_(). Она вызывается каждый раз, когда мы создаем новый объект класса Point.

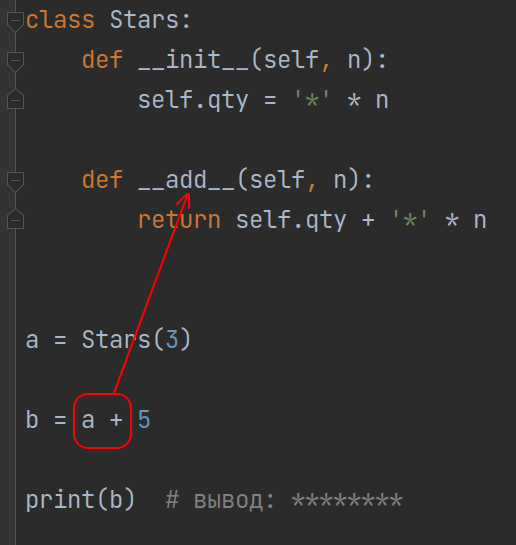
Мы хотим, чтобы функция print() выводила на экран координаты объекта Point. Сейчас print() печатает что-то не то. Мы можем определить метод \_\_str\_\_() в нашем классе, который будет контролировать правильный вывод объекта. Давайте посмотрим, как это сделать.



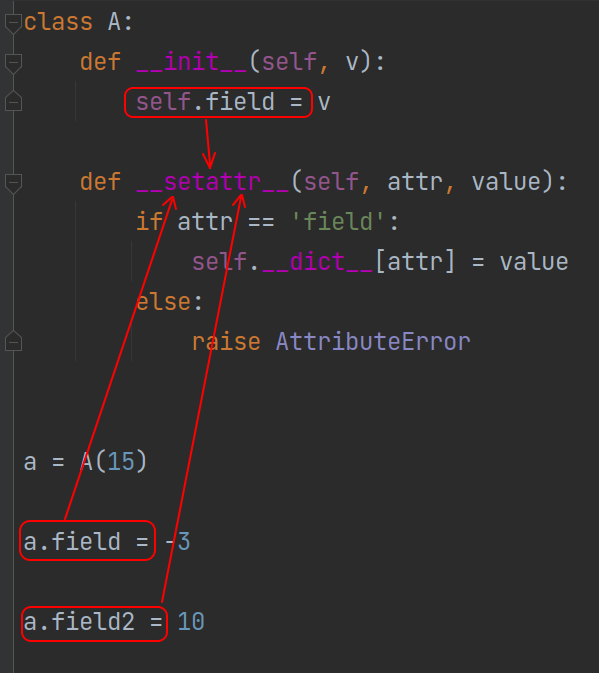
* \_\_init\_\_() – конструктор объектов класса, вызывается при создании объектов



* \_\_del\_\_() – деструктор объектов класса, вызывается при удалении объектов
* \_\_str\_\_() – преобразование объекта к строковому представлению, вызывается, когда объект передается функциям print() и str()
* \_\_add\_\_() – метод перегрузки оператора сложения, вызывается, когда объект участвует в операции сложения будучи операндом с левой стороны



* \_\_setattr\_\_() – вызывается, когда атрибуту объекта выполняется присваивание.



Операция связывания и динамическая типизация.

**Операция связывания**

Операторы присваивания в Python работаю вполне очевидно — значение находящееся справа присваивается переменной, находящейся слева. Существует несколько разновидностей операторов присваивания:

* = — значение правого операнда присвоится левому операнду;

Составные операторы присваивания

* += — сумма левого и правого операнда присвоится левому операнду;
* -= — разность левого и правого операнда присвоится левому операнду;
* \*= — произведение левого и правого операнда присвоится левому операнду;
* /= — разделит левый операнд на правый и результат присвоится левому операнду;
* //= — результат целочисленного деления левого операнда на правый операнд присвоится левому операнду;
* %= — разделит левый операнд на правый по модулю и результат присвоится левому операнду;
* \*\*= — возведет левый операнд в степень правого и результат присвоится левому операнду.
* &= - выполняет побитовое И над операндами и присваивает значение левому операнду
* |= выполняет побитовое ИЛИ над операндами и присваивает значение левому операнду
* ^= выполняет побитовое OR для операндов и присваивает значение левому операнду
* >>= - выполняет побитовый сдвиг вправо для операндов и присваивает значение левому операнду
* <<= - выполняет побитовый сдвиг влево для операндов и присваивает значение левому операнду

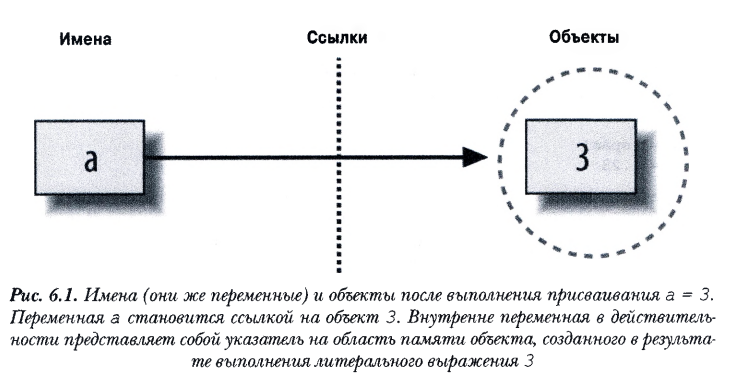
Python поддерживает не только обычное присваивание правого операнда левому, но и множественное присваивание.



Каскадное присваивание a = b = c = 10

Создание переменных в питоне

Переменные и объекты хранятся в разных местах памяти и связываются посредством ссылок (ссылка показана в виде стрелки). Переменные всегда ссылаются на объекты и никогда на другие переменные



>>> a = 3 # Присвоить имени объект

Следующие шаги отражают работу всех присваиваний в языке Python.

* Создание объекта для представления значения 3.
* Создание переменной а, если она еще не существует.
* Связывание переменной a с новым объектом 3.

**Динамическая типизация**

Python является языком с **динамической типизацией**. Это значит, что переменная не привязана жестко с определенному типу. Тип переменной определяется автоматически исходя из значения, которое ей присвоено. Так, при присвоении строки в двойных или одинарных кавычках переменная имеет тип str. При присвоении целого числа. Python автоматически определяет тип переменной как int.

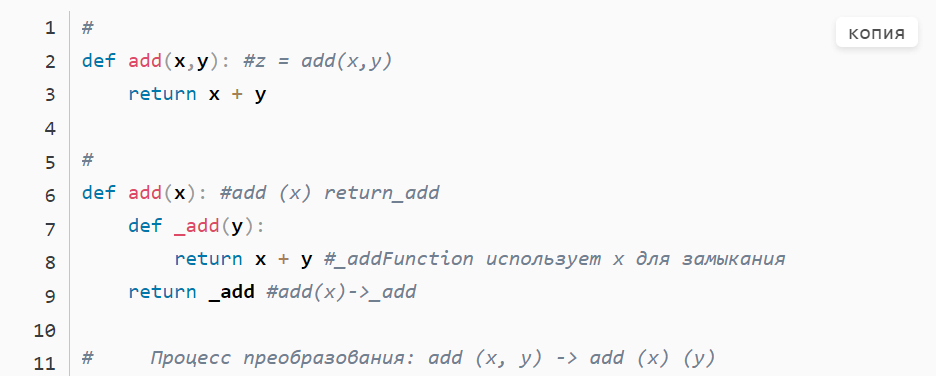
Каррирование и декаррирование

**Каррирование**

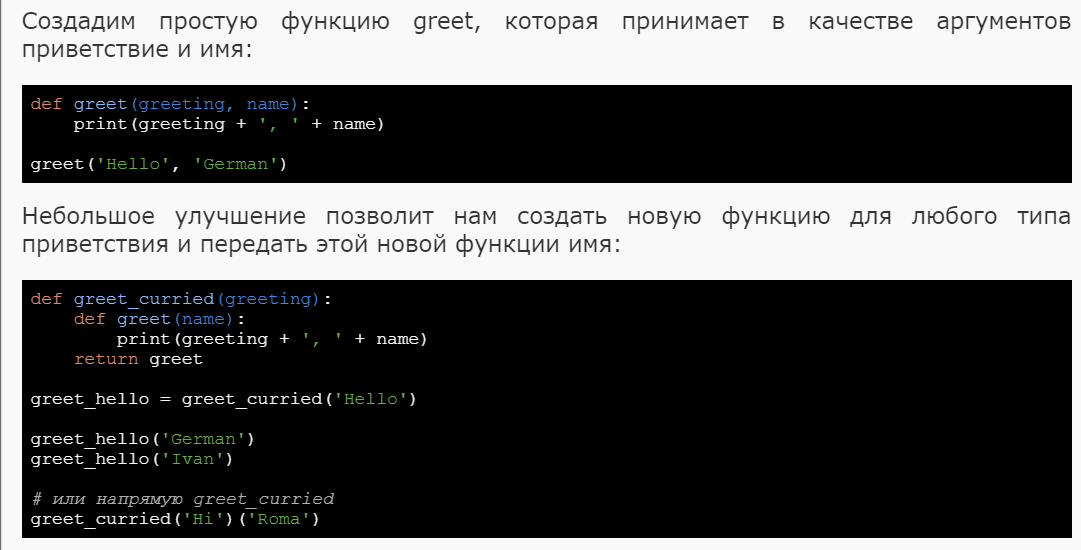
*Каррирование* –это преобразование функции от многих аргументов в набор функций, каждая из которых является функцией от одного аргумента.

*Математическая формула:* перейти от вида *f(x, y, z)* к виду*f(x)(y)(z)*

Пример для представления:



Второй пример:



Вызовы функции со вторым аргументом

Создание переменной, которая вызывает функцию с уже одним заданным аргументом (Hello)

**Декаррирование**

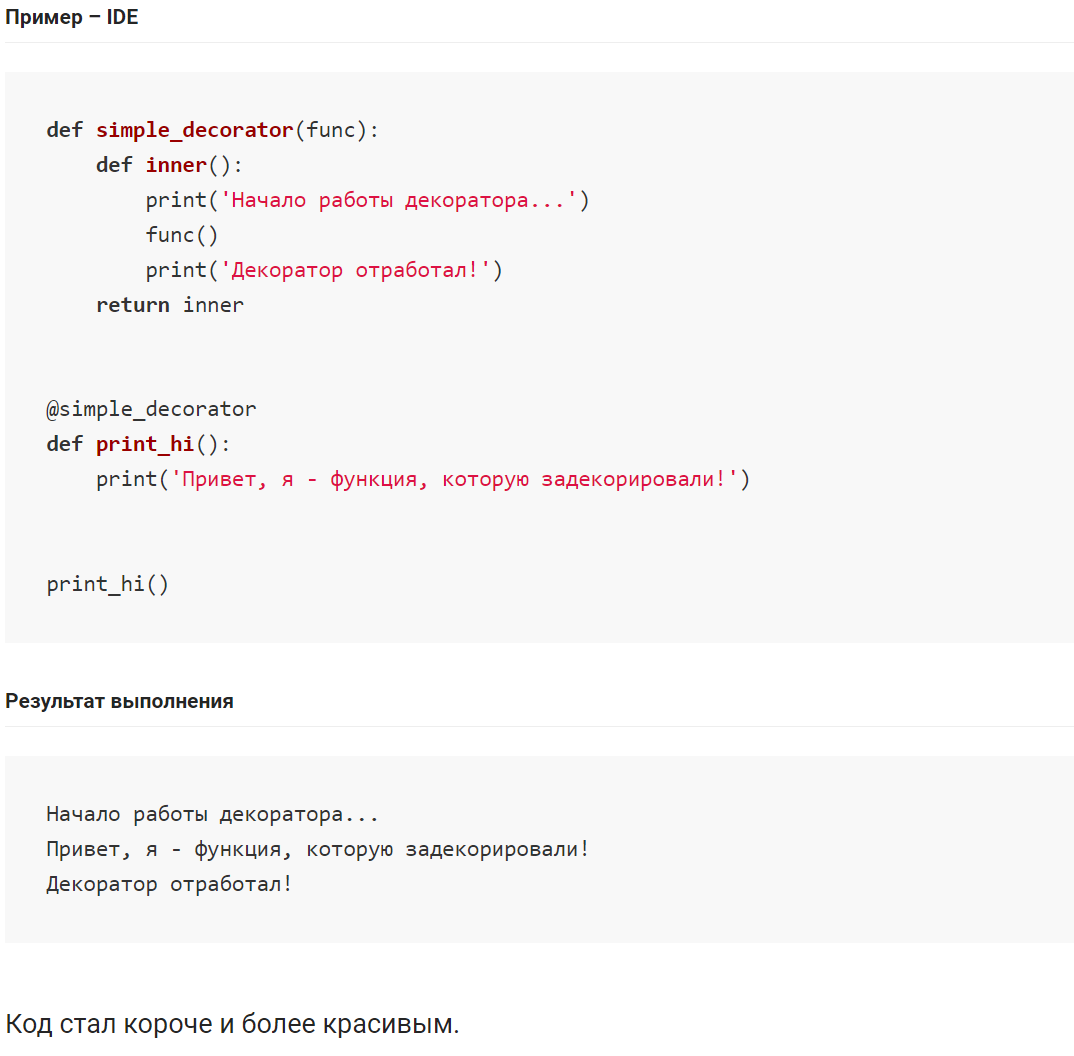
Декоратор функций в общем виде – это функция, которая принимает функцию и возвращает функцию.

Декораторы функций могут производить операции с функцией и возвращают либо саму функцию, либо другую заменяющую её функцию или вызываемый объект.



1. Функция Print\_hi становится аргументом функции simple\_decorator

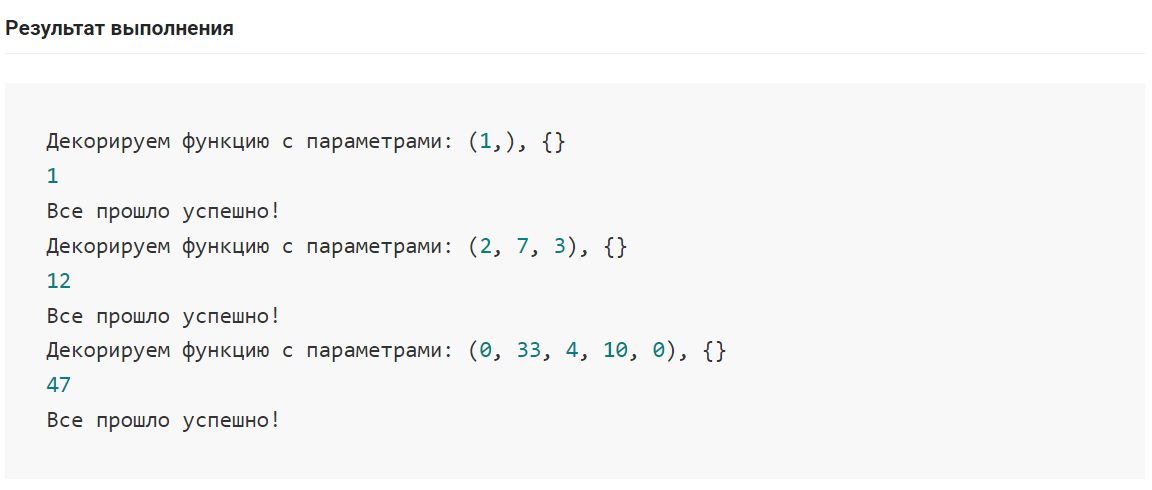
Так, функция **print\_hi()** просто печатает в консоль некое сообщение. Созданный декоратор сообщает о том, что декорируемая функция была подвергнута модификациям (ведь теперь выводится еще 2 дополнительных сообщения).  
  
Чтобы задекорировать объект, нужно передать его внутрь декоратора, а затем вызвать. В нашем случае, мы присвоили задекорированной функции то же имя (**print\_hi**), потому что так принято в Питоне, хотя никто не запрещает давать любое другое имя. Такое применение декорирования не самое удобное, поэтому был придуман синтаксический сахар: **@**.  
  
Осуществим ту же процедуру, что и выше, но уже более питонически.

****

*Декорирование функции с аргументами* **(возможно это немного уже дополнительно)**

Когда мы декорировали функцию**print\_hi()**, то все выглядело очень просто, т.к. у нее не было параметров. А если они будут? Да и у каждой функции есть свой перечень аргументов (как по наименованию, так и по количеству). На помощь приходят **args**, **kwargs**.





**4.** Последовательности. Операции над последовательностями

Последовательность — позиционно упорядоченная коллекция других объектов.

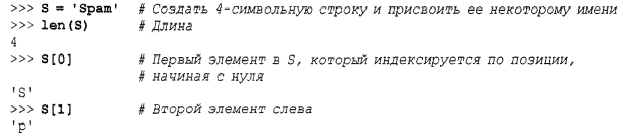
Последовательности могут быть как изменяемыми (список, массив), так и неизменяемыми (кортеж, диапазон, строка).

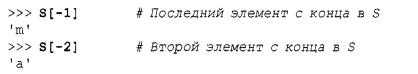
Доступ к значениям последовательностей производится при помощи индексов.

**Операции над последовательностями.**

**Первый пример** – строки. Строки представляют собой последовательности односимвольных строк; другие более универсальные типы последовательностей включают списки и кортежи. Как последовательности, строки поддерживают операции, которые предполагают наличие позиционного порядка среди элементов.

Индексы в Python представляют собой смещения спереди и потому начинаются с 0.

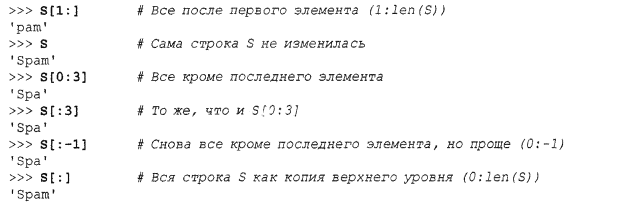




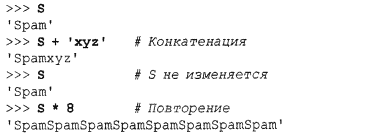
Последовательности также поддерживают нарезку, которая представляет собой способ извлечения целой части (среза) за один шаг.



По умолчанию левая граница среза принимается равной нулю, а правая — длине нарезаемой последовательности.

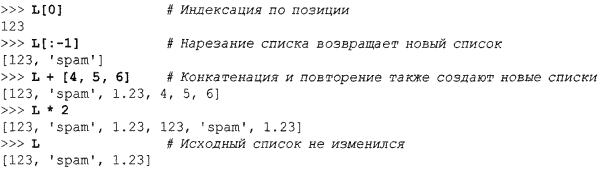


Будучи последовательностями, строки поддерживают также конкатенацию (объединение двух строк в новую строку), обозначаемую знаком “плюс”, и повторение (создание новой строки путем повторения другой).

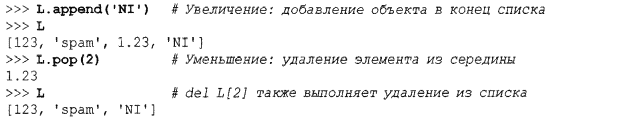


**Второй пример –** списки. Списки представляют собой позиционно упорядоченные коллекции объектов произвольных типов и не имеют фиксированных размеров. Будучи последовательностями, списки поддерживают все операции над последовательностями, которые используются для строк; единственное отличие в том, что результатами обычно будут не строки, а списки.

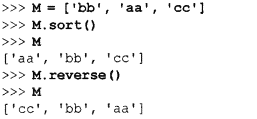




Списки не имеют фиксированных размеров, т.е. они могут увеличиваться и уменьшаться в ответ на операции, специфичные для списков. (append, pop, del, insert, remove, extend)



Поскольку списки являются изменяемыми, большинство списковых методов также модифицируют объект списка на месте, а не создают новый такой объект. (sort, reverse)

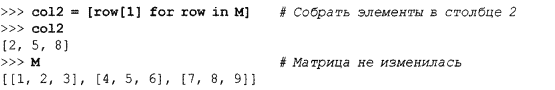


В дополнение к операциям над последовательностями и списковым методам в

Python имеется более сложная операция, известная как выражение спискового включения (list comprehension), способ обработки структур.

Например, необходимо извлечь второй столбец из матрицы





**17.** Генераторы и включения. Генераторные функции. Генераторные выражения

***Генераторы и включения***

Генератор — это функция, в которой содержится хотя бы один оператор yield.

Функция-генератор возвращает объект-генератор.

Объект генератор — это итератор. Поэтому он «исчерпывается», когда не остается ни одного элемента, который можно вернуть и выдает ошибку.

Списковые включения накапливают результаты применения произвольного выражения к итерируемому объекту, содержащему значения, и возвращают их в новом списке. Синтаксически списковые включения помещаются в квадратные скобки, чтобы напоминать о том, что они строят списки.

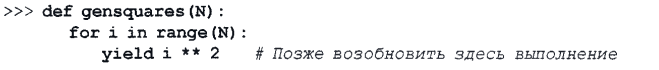


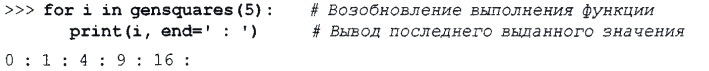
Простейшая форма спискового выражения:



Из-за того, что ни генераторные функции, ни генераторные выражения не создают сразу весь результирующий список, они экономят пространство памяти и позволяют распределить время вычислений по запросам результатов.

**Генераторные функции**

Определение def, содержащее оператор yield, превращается в генераторную функцию. При вызове она возвращает новый объект генератора с автоматическим сохранением локальной области видимости и местоположения в коде, автоматически созданным методом\_\_ iter\_\_, который просто возвращает сам объект, и автоматически созданным методом\_\_ next\_\_, который запускает функцию или возобновляет ее выполнение с места, где она находилась в последний раз, и инициирует исключение StopIteration, когда выпуск результатов завершен. 



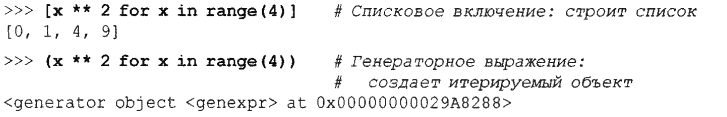
Состояние, которое генераторные функции сохраняют, когда приостанавливаются, содержит местоположение в коде и полную локальную область видимости. Следовательно, их локальные переменные помнят информацию между выпуском результатов и делают ее доступной при возобновлении выполнения функциями.

Основная разница между кодом генераторных и нормальных функций заключается в том, что генератор выдает значение, а не возвращает его.

В крупных программах генераторы могут быть лучше в плане памяти и производительности. Они позволяют функциям избежать выполнения всей работы заранее, что особенно полезно, когда результирующие списки большие или получение каждого значения требует длительных вычислений.

**Генераторные выражения**

Выражение включения, помещенное в круглые скобки, известно как генераторное выражение. Оно возвращает новый объект генератора с таким же автоматически созданным интерфейсом в виде методов и сохранением состояния, как у генераторной функции, т.е. с методом \_\_iter\_\_, просто возвращающим сам объект, и методом \_\_next\_\_, который запускает неявный цикл или возобновляет его выполнение с места, где он находился в последний раз, и инициирует исключение StopIteration, когда выпуск результатов завершен.



В точности как генераторные функции генераторные выражения обеспечивают оптимизацию расхода памяти — они не требуют создания сразу всего результирующего списка

**ВАЖНЫЙ МОМЕНТ!!**

Генераторные функции и генераторные выражения сами представляют собой итераторы и потому поддерживают только одну активную итерацию. Из-за этого итератором генератора является сам генератор.

**27. Обработка исключений. Оператор try**

**Обработка исключений**

Ошибки выполнения (runtime error) появляются в уже скомпилированной программе в процессе ее выполнения. Подобные ошибки еще называются исключениями.

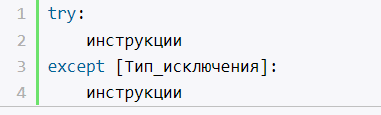
При возникновении исключения работа программы прерывается, и чтобы избежать подобного поведения и обрабатывать исключения в Python есть конструкция try…except.

**Оператор try**

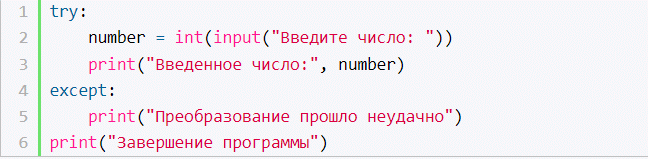
try…except

Перехватывает и производит восстановление после исключений, инициируемых Python или вами.

Конструкция try...except имеет следующее формальное определение:



Весь основной код, в котором потенциально может возникнуть исключение, помещается после ключевого слова try. Если в этом коде генерируется исключение, то работа кода в блоке try прерывается, и выполнение переходит в блок except.

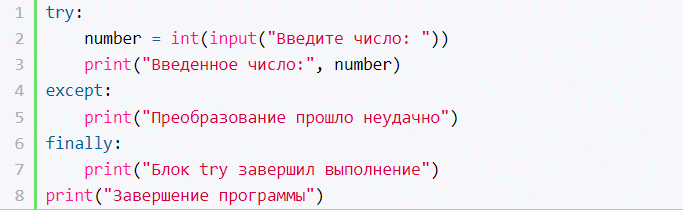


try…finally

При обработке исключений также можно использовать необязательный блок finally.

Как правило, блок finally применяется для освобождения используемых ресурсов, например, для закрытия файлов.

Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение:



3. Основные примитивные типы данных. Реализация операций над ними.

Разница между атомарными и структурными типами данных

По одной из классификаций все типы данных в Python делятся на атомарные и ссылочные.

Атомарные:

* числа;
* строки;

Ссылочные:

* списки;
* кортежи;
* словари;
* функции;
* классы;

Разница между этими двумя группами уходит глубоко в корни языка. Вкратце:

Атомарные объекты, при их присваивании, передаются по значению, а ссылочные — по ссылке

*# пример присваивания атомарного объекта*

atom = 3

btom = atom

atom = 2

print(atom)

> 2

print(btom)

> 3

Из результатов видно, что переменной btom было присвоено именно значение, содержащееся в atom, а не ссылка, указывающая на область памяти.

Посмотрим, как это работает для структурных типов:

*# пример присваивания ссылочного объекта*

link = ['Ipona', 'Master Sword']

alin = link

link[0] = 'Zelda'

print(link)

> ['Zelda', 'Master Sword']

print(alin)

> ['Zelda', 'Master Sword']

Поскольку списки — это ссылочные объекты, то вполне закономерно, что после присваивания переменной link переменной alin передалась именно ссылка на объект list-а и, при печати, на экран были выведены две одинаковые надписи.

Собственно, в этом и вся разница.

Числовые типы

"Все сущее есть Число" — сказал однажды мудрый грек по имени Пифагор. Числа — важнейший и фундаментальнейший из всех типов данных для всех языков программирования. В Python для их представления служит числовой тип данных.

**int (целое число)**

Концепция целых чисел проста и естественна. Это числа без дробной части, которые, говоря математическим языком, являются расширением натурального ряда, дополненного нулём и отрицательными числами.

*# примеры целых чисел*

a = -3000

b = 0

c = 9000

Там, где есть числа, есть и математика. Поэтому резонно, что целые числа используются для исчисления всевозможных математических выражений. Также int применяется в качестве описаний количественных свойств какого-либо объекта.

**float (число с плавающей точкой)**

Действительные или вещественные числа придуманы для измерения непрерывных величин. В отличие от математического контекста, ни один из языков программирования не способен реализовать бесконечные или иррациональные числа, поэтому всегда есть место приближению с определенной точностью, из-за чего возможны такие ситуации:

print(0.3 + 0.3 + 0.3)

> 0.8999999999999999

print(0.3 \* 3 == 0.9)

> False

В плане записи, float ничем не отличаются от int:

*# примеры вещественных чисел*

zero = 0.0

pi = 3.14

e = 2.71

В плане использования — тоже, разве что в любых мало-мальски серьёзных вычислениях без float никуда.

**complex (комплексное число)**

Привет высшей математике! Как вещественный ряд расширяет множество рациональных чисел, так и ряд комплексных чисел расширяет множество вещественных. Показательной особенностью комплексного ряда является возможность извлечения корня из отрицательных чисел.

В Python комплексные числа задаются с помощью функции complex():

*# пример комплексного числа*

z = complex(1, 2)

print(z)

> (1+2j)

*# вещественная часть*

print(z.real)

> 1.0

*# мнимая часть*

print(z.imag)

> 2.0

*# сопряженное комплексное число*

print(z.conjugate())

> (1-2j)

Помните, что операция сравнения для комплексных чисел не определена:

z1 = complex(4, 5)

z2 = complex(100, 200)

print(z1 > z2)

>

Traceback (most recent call last):

print(z1> z2)

TypeError: '>' **not** supported between instances of 'complex' **and** 'complex'

Комплексные числа широко применяются, например, для решения дифференциальных уравнений.

**bool (логический тип данных)**

В каком-то смысле наиболее простой и самый понятный из всех типов данных. У bool есть всего два значения:

* Истина (True);
* Ложь (False).

Однако за этой простотой кроется колоссальный пласт теории в виде булевой алгебры.

*# пример bool*

pravda = True

lozh = False

Переменные логического типа нужны для реализации ветвлений, они применяются для установки флажков, фиксирующих состояния программы, а также используются в качестве возвращаемых значений для функций, названия которых, зачастую, начинаются на "is" (isPrime, isEqual, isDigit). То есть тех, которые, на человеческом языке, отвечали бы на вопрос одним словом "Да" или "Нет".

Последовательности

Ещё одно понятие из математики. Там, последовательность — есть нумерованный набор элементов, в котором возможны их повторения, а порядок имеет значение. Определение Питона схоже с математическим: здесь последовательностью зовётся упорядоченная коллекция объектов.

**str (строка)**

Строки, пожалуй, единственный объект, который может сравниться по степени своей используемости с числовым типом данных. Тавтологическое, но полное определение, справедливое для Python звучит так:

строка — это последовательность односимвольных строк.

s = 'Hello, friend. You are my world'

print(type(s))

> <**class** '**str**'>

Важность строк велика в первую очередь для людей, ведь понятно, что вся письменная речь может рассматриваться, как множество строк. А так как человеку свойственно обмениваться информацией именно в виде набора слов, то можно говорить о практически неограниченном количестве областей применения строкового типа данных. Строки, строки everywhere!

**list (список)**

Список — это ещё один вид последовательностей... Здесь стоит остановиться и отметить, что последовательности в Python бывают изменяемыми и неизменяемыми. Список — изменяемая последовательность, а строки и кортежи — нет. Таким образом, список можно определить, как упорядоченную и изменяемую коллекцию, состоящую из объектов произвольных типов.

*# пример списка*

list\_of\_lists = [['code alpha', 'code beta'], [553, 434]]

list\_of\_lists[0][1] = 'code omega'

print(list\_of\_lists)

> [['code alpha', 'code omega'], [553, 434]]

Само название списков говорит об их предназначении быть объектами для хранения наборов данных. Список покупок, подарков, результатов матчей, ip клиентов или объектов типа Student. Списки в Python — это эдакие массивы из прочих языков "на максималках".

**tuple (кортеж)**

Кортежи в языке Python можно рассматривать, как неизменяемые списки со всеми вытекающими:

*# пример кортежа*

tup = ('i', 'j')

*# мы можем получить первый элемент*

print(tup[0])

> i

*# но изменить его не получится*

tup[0] = 'k'

>

Traceback (most recent call last):

tup[0] = 'k'

TypeError: 'tuple' object does **not** support item assignment

Использование кортежей оправдано, когда разработчику важна скорость работы или неизменяемость элементов последовательности.

dict (словарь)

Словари хоть и являются набором данных, однако не считаются последовательностью, потому как представляют собой неупорядоченный набор пар ключ:значение.

*# пример простого словаря*

dictionary = {'Огонёк': 'уменьш. от слова "Огонь"'}

Применяются они, когда для работы требуется тип данных концептуально схожий с обычным телефонным справочником, где каждая запись — есть пара сопоставленных друг с другом значений, по одному из которых (уникальному ключу) можно получить второе (собственно, значение).

set (множество)

Ещё один "набор, но не последовательность".

*# пример множества*

integer\_num\_set = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}

Мы хотим видеть множество, если нам не существенен порядок элементов, но важна их уникальность.

*# свойство уникальности*

unique\_set = {6, 6, 6, 5}

print(unique\_set)

> {5, 6}

14. Анонимные функции. map, reduce, filter.

**Что такое функции map(), filter() и reduce() в Python?**

Как упоминалось ранее, map(), filter() и reduce() являются встроенными функциями Python. Эти функции обеспечивают функциональный программный аспект Python. В функциональном программировании передаваемые аргументы являются единственными факторами, которые определяют результат. Эти функции могут принимать любую другую функцию в качестве параметра и могут быть переданы другим функциям в качестве параметров.

Также у нас уже выходила статья на тему того, как [функции map(), filter() и reduce() могут улучшить ваш код в Python](https://pythonist.ru/seventipsinpython/).

**Функция map():**

Функция map() является типом высшего порядка. Как упоминалось ранее, эта функция принимает другую функцию в качестве параметра вместе с итерируемой последовательностью и возвращает выходные данные после применения функции на каждый итерируемый элемент из последовательности. Синтаксис выглядит следующим образом:

**map(function, iterables)**

Здесь функция определяет выражение, которое в свою очередь применяется к итерируемым элементам. Функция map может принимать функции, которые определил сам пользователь, а также лямбда-функции в качестве параметра.

Помимо того, мы писали уже про [задачу, как можно сконвертировать список с применением функции map().](https://pythonist.ru/3-varianat-kak-skonvertirovat-spisokl/)

**Совместное использование с функциями, определяемыми пользователем и Lambda-функциями:**

Пользовательские функции совместно с map():

Функция map() может принимать пользовательские функции в качестве параметров. Параметры этих функций устанавливаются исключительно пользователем или программистом.

Пример:

**def newfunc(a):**

**return a\*a**

**x = map(newfunc, (1,2,3,4)) # x - это объект типа map**

**print(x)**

**print(set(x))**

Результат:

**<map object at *0x00000284B9AEA940*>**

**[1, 4, 9, 16]**

Вы также можете передать несколько списков в качестве параметров.

Пример:

**def func(a, b):**

**return a + b**

**a = map(func, [2, 4, 5], [1,2,3])**

**print(a)**

**print(tuple(a))**

Результат:

**<map object at *0x00000284B9BA1E80*>**

**(3, 6, 8)**

Теперь давайте посмотрим, как вы можете использовать lambda-функции внутри map().

Lambda-функции совместно с map():

Lambda-функции — это функции, которые являются анонимными и им не нужно указывать какое-то собственное имя, как это происходит с пользовательскими функциями. Эти функции часто передаются в качестве параметров другим функциям.  
Теперь давайте попробуем применить lambda-функции совместно с функцией map(). Рассмотрим следующий пример:

**tup = (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)**

**newtuple = tuple(map(lambda x: x+3 , tup))**

**print(newtuple)**

Результат:

**(8, 10, 25, 100, 57, 65, 80, 26, 76, 64)**

Приведенный выше вывод является результатом применения lambda-выражения (x + 3) к каждому элементу, присутствующему в кортеже.

**Функция filter():**

Функция filter() используется для создания списка, состоящего из значений, для которых функция возвращает true. Синтаксис этого следующий:

**filter(function, iterables)**

Так же, как и map(), эта функция может использовать в качестве параметра пользовательские функции, а также lambda-функции.

Пример:

**def func(x):**

**if x>=3:**

**return x**

**y = filter(func, (1,2,3,4))**

**print(y)**

**print(list(y))**

Результат:

**<filter object at *0x00000284B9BBCC50*>**

**[3, 4]**

Как видите, y — это объект типа функции filter, а выходной список — это список значений, которые являются истинными для условия (x>=3).

Использование lambda-функций совместно с filter():

Lambda-функция, которая используется в качестве параметра, фактически определяет условие, которое необходимо проверить.

Пример:

**y = filter(lambda x: (x>=3), (1,2,3,4))**

**print(list(y))**

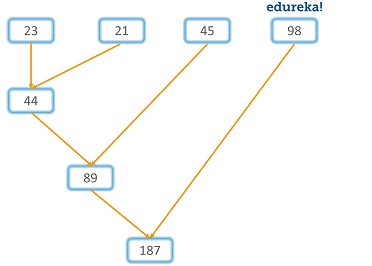
Результат:

**[3, 4]**

Приведенный выше код выдает тот же результат, что и предыдущая функция.

**Функция reduce():**

Функция reduce(), как можно понять из названия, применяет переданную функцию к итерируемому объекту и возвращает одно значение.



Синтаксис:

**reduce(function, iterables)**

Здесь функция определяет, какое выражение необходимо применить к итерируемому объекту. Эту функцию необходимо импортировать из модуля functools.

Пример:

**from functools import reduce**

**reduce(lambda a, b: a + b, [23, 21, 45, 98])**

Результат:

**187**

В приведенном выше примере функция reduce последовательно суммирует каждый элемент из списка и возвращает одно выходное значение.

Функции map(), filter() и reduce() в Python могут использоваться вместе друг с другом.

28. Иерархия исключений. Создание собственных исключений.

Исключения в языках программирования

Исключениями (*exceptions*) в языках программирования называют проблемы, возникающие в ходе выполнения программы, которые допускают возможность дальнейшей ее работы в рамках основного алгоритма. Типичным примером исключения является деление на ноль, невозможность считать данные из файла (устройства), отсутствие доступной памяти, доступ к закрытой области памяти и т.п. Для обработки таких ситуаций в языках программирования, как правило, предусматривается специальный механизм, который называется обработка исключений (*exception handling*).

Исключения разделяют на **синхронные** и **асинхронные**. **Синхронные** исключения могут возникнуть только в определенных местах программы. Например, если у вас есть код, который открывает файл и считывает из него данные, то исключение типа “ошибка чтения данных” может произойти только в указанном куске кода. **Асинхронные** исключения могут возникнуть в любой момент работы программы, они, как правило, связаны с какими-либо аппаратными проблемами, либо приходом данных. В качестве примера можно привести сигнал отключения питания.

В языках программирования чаще всего предусматривается специальный механизм обработки исключений. **Обработка** может быть **с** **возвратом**, когда после обработки исключения выполнение программы продолжается с того места, где оно возникло. И **обработка без возврата**, в этом случае, при возникновении исключения, осуществляется переход в специальный, заранее подготовленный, блок кода.

Различают **структурную** и **неструктурную** обработку исключений. **Неструктурная** обработка предполагает регистрацию функции обработчика для каждого исключения, соответственно данная функция будет вызвана при возникновении конкретного исключения. Для **структурной** обработки язык программирования должен поддерживать специальные синтаксические конструкции, которые позволяют выделить код, который необходимо контролировать и код, который нужно выполнить при возникновении исключительной ситуации.

Ошибки и исключения в Python

В *Python* выделяют два различных вида ошибок: синтаксические ошибки и исключения.

**Синтаксические ошибки в Python**

Синтаксические ошибки возникают в случае если программа написана с нарушениями требований *Python* к синтаксису. Определяются они в процессе парсинга программы. Ниже представлен пример с ошибочным написанием функции *print*.

>>> for i in range(**10**):

prin("hello!")

Traceback (most recent call last):

File "<pyshell#2>", line **2**, in <module>

prin("hello!")

NameError: name 'prin' is not defined

**Исключения в Python**

Второй вид ошибок – это исключения. Они возникают в случае если синтаксически программа корректна, но в процессе выполнения возникает ошибка (деление на ноль и т.п.). Более подробно про понятие исключения написано выше, в разделе “исключения в языках программирования”.

Пример исключения *ZeroDivisionError*, которое возникает при делении на 0.

>>> a = **10**

>>> b = **0**

>>> c = a / b

Traceback (most recent call last):

File "<pyshell#5>", line **1**, in <module>

c = a / b

ZeroDivisionError: division by zero

В *Python* исключения являются определенным типом данных, через который пользователь (программист) получает информацию об ошибке. Если в коде программы исключение не обрабатывается, то приложение останавливается и в консоли печатается подробное описание произошедшей ошибки с указанием места в программе, где она произошла и тип этой ошибки.

Иерархия исключений в Python

Существует довольно большое количество встроенных типов исключений в языке *Python*, все они составляют определенную иерархию, которая выглядит так, как показано ниже.

BaseException  
+– SystemExit  
+– KeyboardInterrupt  
+– GeneratorExit  
+– Exception  
     +– StopIteration  
     +– StopAsyncIteration  
     +– ArithmeticError  
     |    +– FloatingPointError  
     |    +– OverflowError  
     |    +– ZeroDivisionError  
     +– AssertionError  
     +– AttributeError  
     +– BufferError  
     +– EOFError  
     +– ImportError  
          +– ModuleNotFoundError  
     +– LookupError  
     |    +– IndexError  
     |    +– KeyError  
     +– MemoryError  
     +– NameError  
     |    +– UnboundLocalError  
     +– OSError  
     |    +– BlockingIOError  
     |    +– ChildProcessError  
     |    +– ConnectionError  
     |    |    +– BrokenPipeError  
     |    |    +– ConnectionAbortedError  
     |    |    +– ConnectionRefusedError  
     |    |    +– ConnectionResetError  
     |    +– FileExistsError  
     |    +– FileNotFoundError  
     |    +– InterruptedError  
     |    +– IsADirectoryError  
     |    +– NotADirectoryError  
     |    +– PermissionError  
     |    +– ProcessLookupError  
     |    +– TimeoutError  
     +– ReferenceError  
     +– RuntimeError  
     |    +– NotImplementedError  
     |    +– RecursionError  
     +– SyntaxError  
     |    +– IndentationError  
     |         +– TabError  
     +– SystemError  
     +– TypeError  
     +– ValueError  
     |    +– UnicodeError  
     |         +– UnicodeDecodeError  
     |         +– UnicodeEncodeError  
     |         +– UnicodeTranslateError  
     +– Warning  
          +– DeprecationWarning  
          +– PendingDeprecationWarning  
          +– RuntimeWarning  
          +– SyntaxWarning  
          +– UserWarning  
          +– FutureWarning  
          +– ImportWarning  
          +– UnicodeWarning  
          +– BytesWarning  
          +– ResourceWarning

Как видно из приведенной выше схемы, все исключения являются подклассом исключения *BaseException*. Более подробно об иерархии исключений и их описании можете прочитать [здесь](https://docs.python.org/3/library/exceptions.html#bltin-exceptions).

Обработка исключений в Python

Обработка исключений нужна для того, чтобы приложение не завершалось аварийно каждый раз, когда возникает исключение. Для этого блок кода, в котором возможно появление исключительной ситуации необходимо поместить во внутрь синтаксической конструкции *try…except*.

print("start")

try:

val = int(input("input number: "))

tmp = **10** / val

print(tmp)

except Exception as e:

print("Error! " + str(e))

print("stop")

В приведенной выше программе возможных два вида исключений – это *ValueError*, возникающее в случае, если на запрос программы “введите число”, вы введете строку, и *ZeroDivisionError* – если вы введете в качестве числа 0.

Вывод программы при вводе нулевого числа будет таким.

**start input number: 0 Error! stop**

Если бы инструкций *try…except* не было, то при выбросе любого из исключений программа аварийно завершится.

print("start")

val = int(input(“input number: “))

tmp = **10** / val

print(tmp)

print("stop")

Если ввести 0 на запрос приведенной выше программы, произойдет ее остановка с распечаткой сообщения об исключении.

Генерация исключений в Python

Для принудительной генерации исключения используется инструкция *raise*.

Самый простой пример работы с *raise* может выглядеть так.

try:

raise Exception("Some exception")

except Exception as e:

print("Exception exception " + str(e))

Таким образом, можно “вручную” вызывать исключения при необходимости.

Пользовательские исключения (User-defined Exceptions) в Python

В *Python* можно создавать собственные исключения. Такая практика позволяет увеличить гибкость процесса обработки ошибок в рамках той предметной области, для которой написана ваша программа.

Для реализации собственного типа исключения необходимо создать класс, являющийся наследником от одного из классов исключений.

*class* NegValException(Exception):

pass

try:

val = int(input("input positive number: "))

if val < **0**:

raise NegValException("Neg val: " + str(val))

print(val + **10**)

except NegValException as e:

print(e)

**1. Язык Python - общая характеристика, преимущества и недостатки, краткая история. PEP’ы.**

Общая характеристика

Python — высокоуровневый, интерпритируемый язык программирования общего назначения. Он имеет минималистичный синтаксис и направлен на повышение читаемости кода.

Пояснения на случай если вы не знаете, что есть что:

* Высокоуровневые языки — соответственно разрабатываются для удобства использования и скорости написания программы.
* Интерпретируемый - программа выполняется специальной программой — интерпретатором.
* Общего положения - язык программирования для создания программного обеспечения в широком спектре прикладных областей.

Язык поддерживает несколько парадигм программирования: объектно-ориентированное, функциональное, структурное, императивное и аспектно-ориентированное.

* Парадигма — это модель (чего угодно, но чаще всего это слово употребляется в применении к науке или общему мировоззрению)
* В объектно-ориентированных языках основа это классы и экземпляры классов это равносильно типу и объекту этого типа. Выполнение условных задач или же просто работа программы строится на взаимодействии различных классов. (также поддерживает и процедурное программирование - программу можно написать без единого класса)
* Функциональное —процесс вычисления трактуется как вычисление значений функций в математическом понимании последних
* Структурное - повышение ясности, качества и времени разработки компьютерной программы путем широкого использования структурированных конструкций потока управления выбора (if/then/else) и повторения (while и for), блочных структур и подпрограмм.
* Императивное - программист будет указывать последовательность команд для выполнения
* Аспектно – ориентированное - предполагает разбиение логики программы на отдельные части

Основная реализация языка Python — CPython. Написана на C. (также используются Jython, IronPython и др.)

Python является языком с полной динамической типизацией и автоматическим управлением памятью. Динамическая типизация означает, что тип переменной определяется только во время исполнения.

Преимущества

* низкий порог вхождения;
* язык широкого применения;
* минималистичный синтаксис;
* кроссплатформенность;
* открытый исходный код интерпретатора CPython;
* наличие дружелюбного, отзывчивого сообщества;
* поддержка многих IDE;
* огромное количество библиотек;
* входит в поставку большинства дистрибутивов Linux.

(IDE - Integrated Development Environment или «интегрированная среда разработки», например, известные нам: PyCharm, VS Code)

Недостатки

* Низкая скорость.
* Динамическая типизация(отсутствие статической типизации),

Она позволяет писать кратко, не объявляя тип переменной. С одной стороны, это экономит время разработчика, но может привести к ошибкам при попытке выполнить операцию с несоответствующим типом данных. (Чтобы предотвратить это, потребуется дополнительное тестирование кода и добавление проверок на типы.)

Краткая история

Разработку Python начал Гвидо ван Россум в декабре 1989 года. Для ОС Amoeba требовался расширяемый скриптовый язык. На досуге, Гвидо начал писать Python, позаимствовав некоторые наработки из языка ABC.

Python 0.9.0 впервые вышел в свет в 1991 году.

Версия Python 2.0 была выпущена 16 октября 2000 г., а первая обратно-несовместимая версия Python 3.0 — 3 декабря 2008 г. ( последняя версия Python 3.11.1 - Dec. 6, 2022)

PEP’ы

PEP расшифровывается как Python Enhancement Proposal («Предложение по улучшению Python»). Это документы, предлагающие новые особенности языка. Они образуют официальную документацию особенности языка, принятие или отклонение которой обсуждается в сообществе Python.

Существует три вида PEP:

* PEP Standards Track - Дорожка стандартов?? (хз я с англ сайта брала так что перевод только такой )

описывает новую функцию или реализацию для Python. Он также может описывать стандарт взаимодействия, который будет поддерживаться вне стандартной библиотеки для текущих версий Python до того, как последующий PEP добавит поддержку стандартной библиотеки в будущей версии.

* Информационный PEP

описывает проблему проектирования Python или предоставляет общие рекомендации или информацию сообществу Python, но не предлагает новую функцию. (пользователи и разработчики могут свободно игнорировать информационные PEP или следовать их советам)

* PEP процесса

описывает процесс, окружающий Python, или предлагает изменение процесса. PEP процессов аналогичны PEP Standards Track, но применяются не только к самому языку Python, но и к другим областям. Они могут предложить реализацию, но не кодовую базу Python; они часто требуют соглашения сообщества; (в отличие от информационных PEP, они представляют собой больше, чем рекомендации, и пользователи, как правило, не могут их игнорировать)

PEP 0 — Индекс предложений по улучшению Python (PEP)

Этот PEP содержит индекс всех предложений по улучшению Python, известных как PEP. Номера PEP назначаются редакторами PEP, и однажды назначенные номера никогда не меняются.

PEP8 — это официальное руководство по стилю кода Python.

Следование PEP8 не сделает код абсолютно «питоническим», но способствует узнаваемости ко да для многих Python-разработчиков.

В PEP8 решаются вопросы, связанные с символами пробелов.

**15. Декораторы.**

Декоратор - это функцияя, возвращающая функцию-обёртку для функции, переданной в декоратор.

Функция-обёртка - позволяет выполнить некоторые действия до и после вызова декорируемой функции.

Декораторы — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

Но помните, что декораторы вызываются только один раз. Как раз тогда, когда Python импортирует скрипт. Впоследствии вы не сможете динамически задавать аргументы. Когда вы выполняете " import x", функция уже оформлена, поэтому вы ничего не можете изменить.

Легкий пример из лекции препода, лучше ещё видосик на ютубе глянуть, как это все работает.

(<https://github.com/still-coding/se_python_snippets/blob/main/02_functions/f09_decorators.py> тут все примеры от препода)

def my\_decorator(f):  
 def wrapper(\*args, \*\*kwargs):  
 print('something b4 function')  
 res = f(\*args, \*\*kwargs)  
 print('something after function')  
 return res  
 return wrapper  
  
  
@my\_decorator # == my\_decorator(hello)(...) каждый раз вместо hello(...)  
def hello(name):  
 return 'Hello, ' + name  
  
  
print(hello(name='Rustam'))  
print(hello('Ekaterina'))

**30. Диспетчеры контекстов. Протокол управление контекстами.**

Контекстный менеджер – это специальная конструкция управления к ресурсу.( — это объект, определяющий контекст выполнения в операторе with.)

Внешние ресурсы – текстовые файлы, базы данных и др.

Мы должны сначала подключиться к ресурсу и потом от них отключиться, Менеджер контекстов контролирует что нужно делать когда мы получаем доступ к ресурсу и что нужно делать после того, когда доступ нам уже не нужен.

Например, вам может потребоваться открыть файл, вписать в него кучу всего и закрыть. Фактически, Python создает один такой экземпляр автоматически каждый раз, когда вы открываете файл, используя оператор with:

with open(имя\_ресурса.txt, 'w') as имя\_переменной:

имя\_переменной.write(‘добавляемый\_текст’)

В этом примере мы используем функцию open() с оператором with. После блока with Python автоматически закроется.

Область видимости имя\_переменной имеет ту же область видимости, что и оператор with. Это означает, что вы можете обращаться к имя\_переменной как внутри оператора with, так и после него.

<https://www.youtube.com/watch?v=ycVlsU_c4Mg>т– тут подробно все объясняют, обязательно посмотрите хотя бы первые 4 мин, чтобы понять че как там вообще

<https://python-scripts.com/contextlib> тут есть больше примеров (Создание контекстного менеджера с использованием contextlib, ExitStack, Реентерабельные контекстные менеджеры)

Протокол контекстного менеджера

Контекстные менеджеры Python работают на основе протокола контекстного менеджера.

Протокол менеджера контекста включает следующие методы:

* \_\_enter\_\_() — устанавливает контекст и, по желанию, возвращать некоторый объект.
* \_\_exit\_\_() — очищает объект.

Метод \_\_exit\_\_() принимает три аргумента: тип исключения, значение исключения и объект трассировки. Все эти аргументы будут равны None, если исключение не произошло.

Метод \_\_exit\_\_() возвращает логическое значение: True или False.

Если возвращаемое значение равно True, Python заглушит исключение.

Если вы хотите, чтобы класс поддерживал протокол контекстного менеджера, вам необходимо реализовать эти два метода.

Предположим, что у нас есть некий класс ContextManager, поддерживающий протокол контекстного менеджера.

Вот, как можно использовать этот класс:

with ContextManager() as имя\_переменной:

# что-то делаем

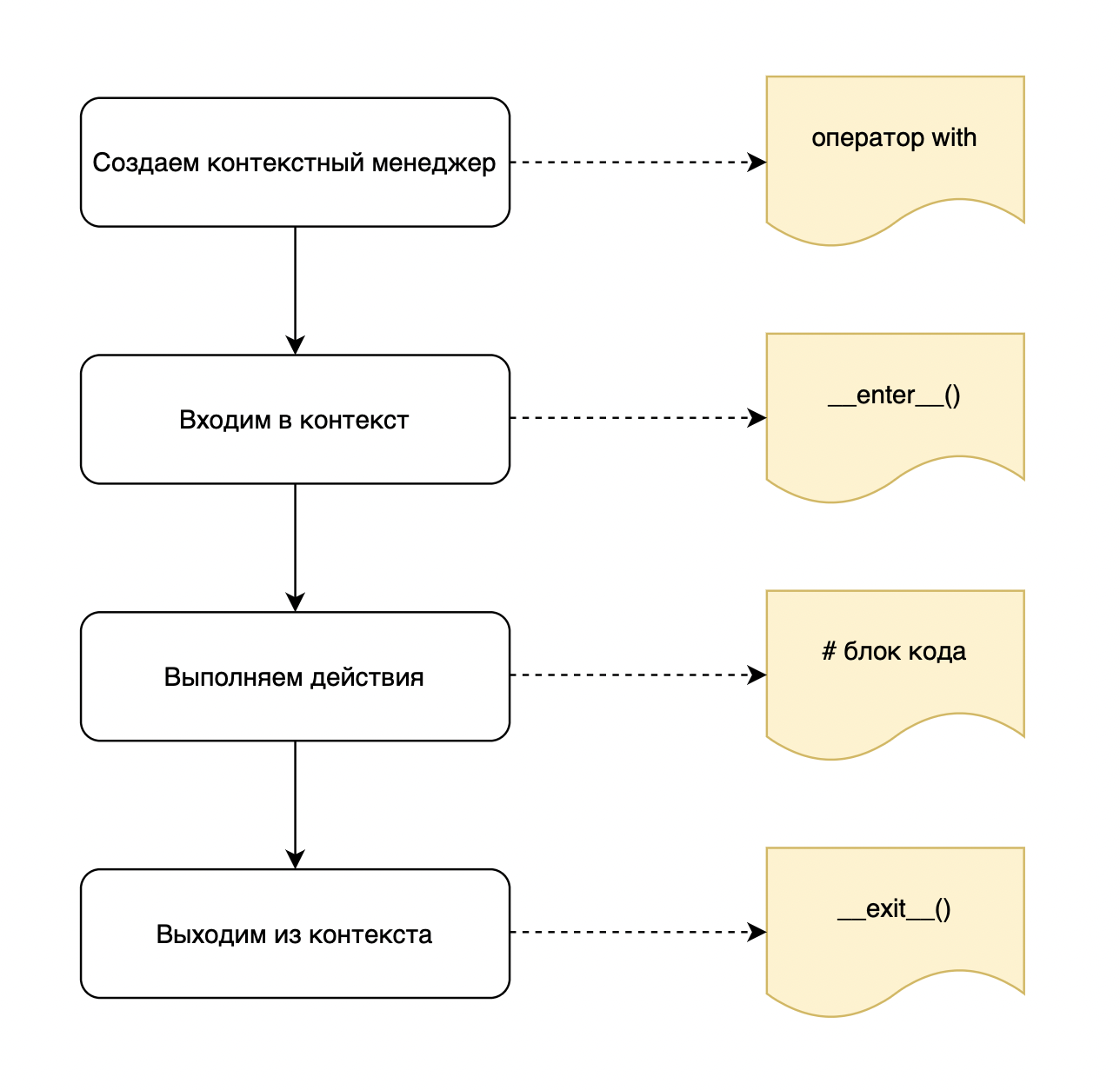
# закончили с контекстом

Когда вы используете класс ContextManager с оператором with, Python неявно создает экземпляр класса — instance — и автоматически вызывает метод \_\_enter\_\_() на этом экземпляре.

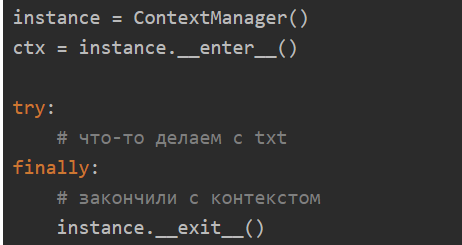
Метод \_\_enter\_\_() может по желанию возвращать объект. Если это так, Python присваивает возвращаемый объект имя\_переменной.

Обратите внимание, что имя\_переменной ссылается на объект, возвращаемый методом \_\_enter\_\_(). Он не ссылается на экземпляр класса ContextManager.

Если внутри блока with или после блока with возникает исключение, Python вызывает метод \_\_exit\_\_() на объекте экземпляра.



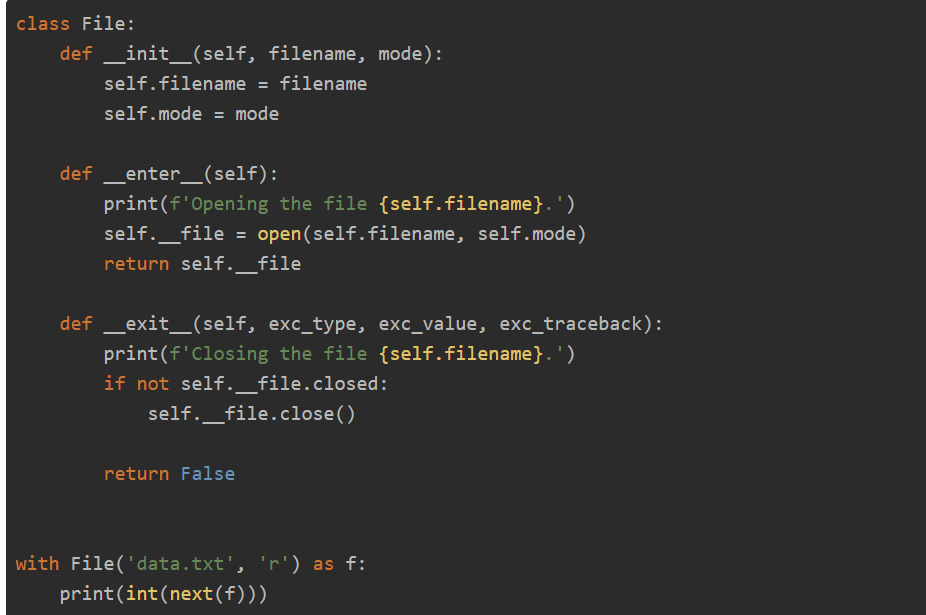
Функционально оператор with эквивалентен конструкции try...finally:



Контекстный менеджер можно использовать для:

* автоматического открытия и закрытия файлов.
* заблокировать — разблокировать.
* Запустить — остановить. Например, можно использовать контекстный менеджер для запуска таймера и его автоматической остановки.
* Изменить — сбросить. Например, вашему приложению необходимо подключиться к нескольким источникам данных. И у него есть соединение по умолчанию.

Пример создания протокола контекстного менеджера:



Как это работает

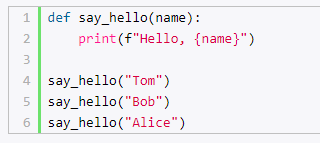
* Инициализируем имя файла и режим в методе \_\_init\_\_().
* Открываем файл в методе \_\_enter\_\_() и возвращаем объект файла.
* Закрываем файл, если он открыт, в методе \_\_exit\_\_().

<https://codechick.io/tutorials/python/python-context-manager> - тут очень хорошо без лишней воды и с объяснением примеров

**№**10. Передача аргументов. Режимы сопоставления аргументов**.**

Функция может принимать параметры. Через параметры в функцию можно передавать данные. Банальный пример - функция print(), которая с помощью параметра принимает значение, выводимое на консоль.

Теперь определим и используем свою функцию с параметрами:



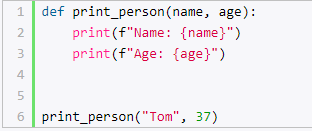
Функция say\_hello имеет параметр name, и при вызове функции мы можем передать этому параметру какой-либо значение. Внутри функции мы можем использовать параметр как обычную переменную, например, вывести значение этого параметра на консоль функцией print. Так, в выражении:



Строка "Tom" будет передаваться параметру name. В итоге при выполнении программы мы получим следующий консольный вывод:



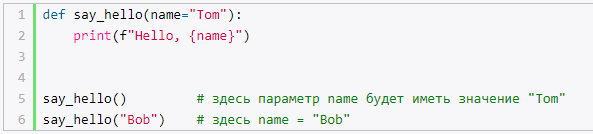
При вызове функции значения передаются параметрам по позиции. Например, определим и вызовем функцию с несколькими параметрами:



Здесь функция print\_person принимает два параметра: name и age. Первое значение - "Tom" передается первому параметру, то есть параметру name. Второе значение - 37 передается второму параметру - age. И внутри функции значения параметров выводятся на консоль: 

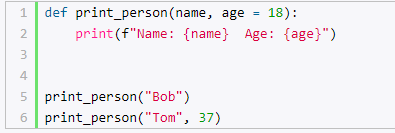
**Значения по умолчанию**

Некоторые параметры функции мы можем сделать необязательными, указав для них значения по умолчанию при определении функции. Например:

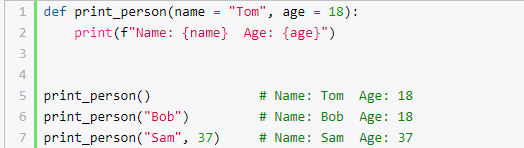


Здесь параметр name является необязательным. И если мы не передаем при вызове функции для него значение, то применяется значение по умолчанию, то есть строка "Tom". Консольный вывод данной программы: 

Если функция имеет несколько параметров, то необязательные параметры должны идти после обязательных. Например:

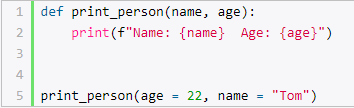


Здесь параметр age является необязательным и по умолчанию имеет значение 18. Перед ним расположен обязательный параметр name. Поэтому при вызове функции мы можем не передавать значение параметру age, но параметру name передать значение необходимо.

При необходимости мы можем сделать все параметры необязательными: 

**Передача значений параметрам по имени. Именованные параметры**

В примерах выше при вызове функции значения передаются параметрами функции по позиции. Но также можно передавать значения параметрам по имени. Для этого при вызове функции указывается имя параметра и ему присваивается значение:



В данном случае значения параметрам age и name передаются по имени. И несмотря на то, что параметр name идет первым в определении функции, мы можем при вызове функции написать print\_person(age = 22, name = "Tom") и таким образом передать число 22 параметру age, а строку "Tom" параметру name.

Символ \* позволяет установить, какие параметры будут именнованными - то есть такие параметры, которым можно передать значения только по имени. Все параметры, которые располагаются справа от символа \*, получают значения только по имени:

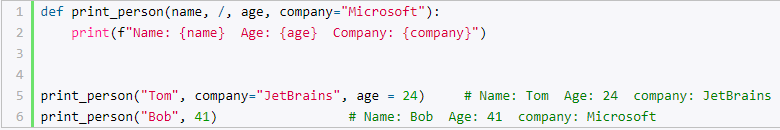


В данном случае параметры age и company являются именнованными.

Можно сделать все параметры именнованными, поставив перед списком параметров символ \*:

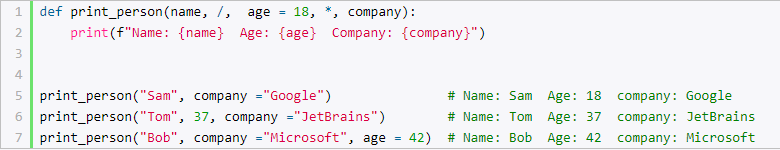


Если наоборот надо определить параметры, которым можно передавать значения только по позиции, то есть позиционные параметры, то можно использовать символ /: все параметры, которые идут до символа / , являются позиционными и могут получать значения только по позиции



В данном случае параметр name является позиционным.

Для одной функции можно определять одновременно позиционные и именнованные параметры.

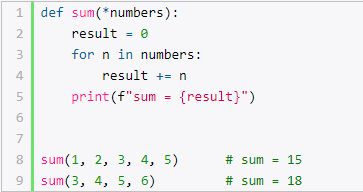


В данном случае параметр name располагается слева от символа /, поэтому является позиционным и обязательным - ему можно можно передать значение только по позиции.

Параметр company является именнованным, так как располагается справа от символа \*. Параметр age может получать значение по имени и по позиции.

**Неопределенное количество параметров**

С помощью символа звездочки можно определить параметр, через который можно передавать неопределенное количество значений. Это может быть полезно, когда мы хотим, чтобы функция получала несколько значений, но мы точно не знаем, сколько именно. Например, определим функцию подсчета суммы чисел:



В данном случае функция sum принимает один параметр - \*numbers, но звездочка перед названием параметра указывает, что фактически на место этого параметра мы можем передать неопределенное количество значений или набор значений. В самой функции с помощью цикла for можно пройтись по этому набору, получить каждое значение из этого набора в переменную n и произвести с ним какие-нибудь действия. Например, в данном случае вычисляется сумма переданных чисел.

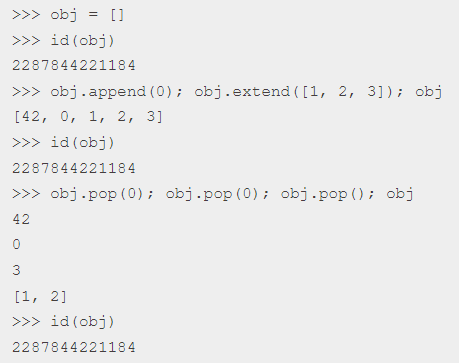
///Доп инфа про объекты, типы. Показалась полезной, оставлю тут

Объекты в Python могут быть изменяемыми (mutable) или неизменяемыми (immutable). Это свойство полностью зависит от типа объекта. Иными словами, не/изменяемость является характеристикой типа, а не конкретных объектов.

Тип является изменяемым, если содержимое объекта может быть изменено без изменений его идентификатора и типа.

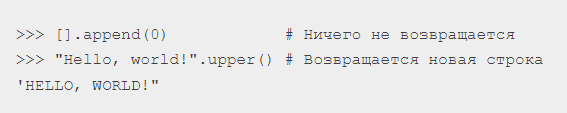
Список (list) является изменяемым типом данных. Почему? Потому что списки являются *контейнерами*: в них можно добавлять данные и из можно удалять данные. Иными словами, их можно спокойно изменять.

Ниже приведен пример того, как содержимое списка меняется, но идентификатор остается тем же самым, чтобы мы не делали.

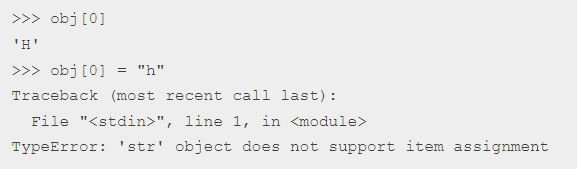


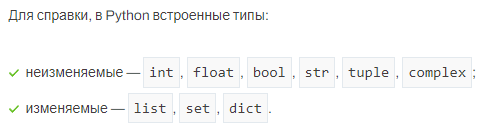
С другой стороны, содержимое неизменяемых объектов нельзя изменить. Они существует так, как их инициализировали в первый раз. Строка str является хорошим примером неизменяемого типа.

Вызывая различные методы, призванные как-то модифицировать состояние, объекты с неизменяемым типом что-то возвращают, в отличие от изменяемых. Например, метод списка append ничего не возвращает (он изменяет существующий), а методы строки возвращают новую строку (новый объект):



Другим сигналом неизменяемости объекта является невозможность изменить отдельные его элементы, например, при присваивании через индексы:

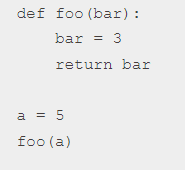




**В Python используется передача параметров через присваивание**

 При вызове функции каждый параметр связывается с соответствующим объектом, указанным в сигнатуре функции.

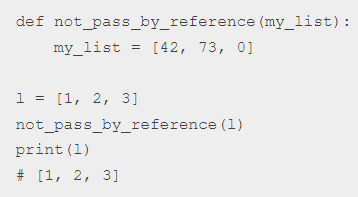
Так, если мы передаем неизменяемые параметр (например, int), то у нас нет возможности хоть как-то его изменить. Каждый раз, когда используется присваивание, то создается новый объект, хоть и имеющий то же самое имя. Взгляните на данный пример:



Вызов функции для неизменяемого типа подразумевает, что используется связывание bar = 5. Сразу после этого в теле функции осуществляется второе связывание bar = 3. Но все эти объекты разные. Поэтому **при использовании неизменяемых объектов в качестве параметров, их передача осуществляется *как будто* по схеме передаче значений**.

С другой стороны, изменяемые объекты, которые передаются в качестве параметров, могут быть изменены, точнее их содержимое. Собственно поэтому функции, которые удаляли или добавляли элементы в переданный в качестве параметра список, изменяли и исходный. А вот когда мы присвоили параметр новому списку (my\_list = [42, 73, 0]), то мы просто создали новый объект и связали его с именем my\_list. (пример ниже) Поэтому стоит говорить, что в Python используется модель ***передачи через присваивание (pass-by-assignment)***.

Пример с my\_list:



**№20.** Пакеты модулей. Абсолютное и относительное импортирование**.**

**Что такое модуль в Python?**

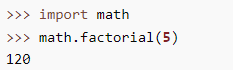
Под модулем в Python понимается файл с расширением .py. Модули предназначены для того, чтобы в них хранить часто используемые функции, классы, константы и т.п. Можно условно разделить модули и программы: программы предназначены для непосредственного запуска, а модули для импортирования их в другие программы. Стоит заметить, что модули могут быть написаны не только на языке Python, но и на других языках (например C).

**Как импортировать модули в Python?**

Самый простой способ импортировать модуль в Python это воспользоваться конструкцией:

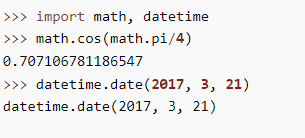
**import имя\_модуля**

Импорт и использование модуля math, который содержит математические функции, будет выглядеть вот так.



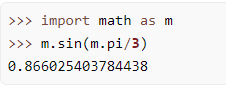
За один раз можно импортировать сразу несколько модулей, для этого их нужно перечислить через запятую после слова import:

**import имя\_модуля1, имя\_модуля2**



Если вы хотите задать псевдоним для модуля в вашей программе, можно воспользоваться вот таким синтаксисом:

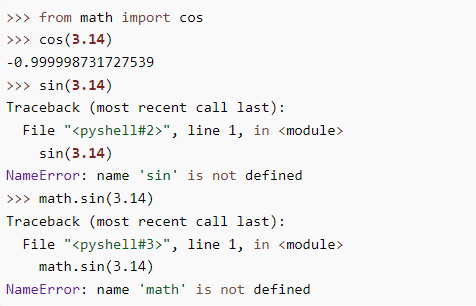
**import имя\_модуля as новое\_имя**



Используя любой из вышеперечисленных подходов, при вызове функции из импортированного модуля, вам всегда придется указывать имя модуля (или псевдоним). Для того, чтобы этого избежать делайте импорт через конструкцию from … import…

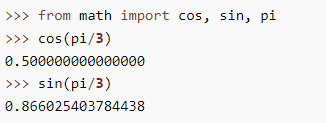
**from имя\_модуля import имя\_объекта**

При этом импортируется только конкретный объект (в нашем примере: функция cos), остальные функции недоступны, даже если при их вызове указать имя модуля.



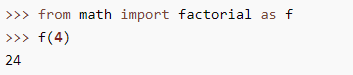
Для имортирования нескольких функций из модуля, можно перечислить их имена через запятую.

**from имя\_модуля import имя\_объекта1, имя\_объекта2**



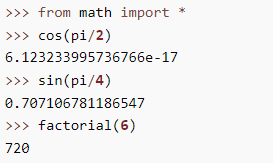
Импортируемому объекту можно задать псевдоним.

**from имя\_модуля import имя\_объекта as псевдоним\_объекта**



Если необходимо импортировать все фукнции, классы и т.п. из модуля, то воспользуйтесь следующей формой оператора from … import …

**from имя\_модуля import \***



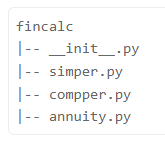
**Что такое пакет в Python?**

Пакет в Python – это каталог, включающий в себя другие каталоги и модули, но при этом дополнительно содержащий файл \_\_init\_\_.py. Пакеты используются для формирования пространства имен, что позволяет работать с модулями через указание уровня вложенности (через точку).

Для импортирования пакетов используется тот же синтаксис, что и для работы с модулями.

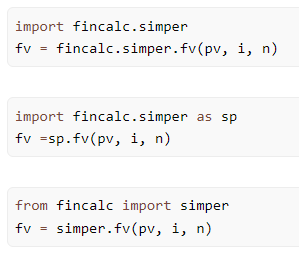
**Использование пакетов в Python**

Рассмотрим следующую структуру пакета:



Пакет fincal содержит в себе модули для работы с простыми процентами (simper.py), сложными процентами (compper.py) и аннуитетами (annuity.py).

Для использования фукнции из модуля работы с простыми процентами, можно использовать один из следующих вариантов:



Файл \_\_init\_\_.py может быть пустым или может содержать переменную \_\_all\_\_, хранящую список модулей, который импортируется при загрузке через конструкцию

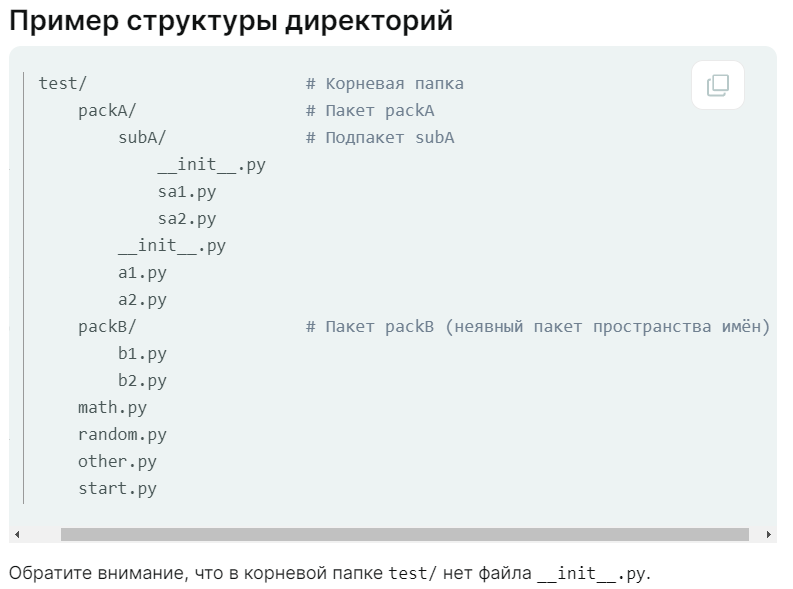
from имя\_пакета import \*

Например для нашего случая содержимое \_\_init\_\_.py может быть вот таким:



Основные определения

* Модуль: любой файл \*.py. Имя модуля — имя этого файла.
* Встроенный модуль: «модуль», который был написан на Си, скомпилирован и встроен в интерпретатор Python, и потому не имеет файла \*.py.
* Пакет: любая папка, которая содержит файл \_\_init\_\_.py. Имя пакета — имя папки.
* С версии Python 3.3 любая папка (даже без \_\_init\_\_.py) считается пакетом.
* Объект: в Python почти всё является объектом — функции, классы, переменные и т. д.



Есть 4 разных вида импортов:

1. import <пакет>
2. import <модуль>
3. from <пакет> import <модуль или подпакет или объект>
4. from <модуль> import <объект>

Пусть X — имя того, что идёт после import:

* Если X — имя модуля или пакета, то для того, чтобы использовать объекты, определённые в X, придётся писать X.объект.
* Если X — имя переменной, то её можно использовать напрямую.
* Если X — имя функции, то её можно вызвать с помощью X().

Опционально после любого выражения import X можно добавить as Y. Это переименует X в Y в пределах скрипта. Учтите, что имя X с этого момента становится недействительным. Частым примером такой конструкции является import numpy as np.

Аргументом для import может быть как одно имя, так и их список. Каждое из имён можно переименовать с помощью as. Например, следующее выражение будет действительно в start.py: import packA as pA, packA.a1, packA.subA.sa1 as sa1.

Пример: нужно в start.py импортировать функцию helloWorld() из sa1.py.

* Решение 1: from packA.subA.sa1 import helloWorld. Мы можем вызвать функцию напрямую по имени: x = helloWorld().
* Решение 2: from packA.subA import sa1 или то же самое import packA.subA.sa1 as sa1. Для использования функции нам нужно добавить перед её именем имя модуля: x = sa1.helloWorld(). Иногда такой подход предпочтительнее первого, так как становится ясно, из какого модуля взялась та или иная функция.
* Решение 3: import packA.subA.sa1. Для использования функции перед её именем нужно добавить полный путь: x = packA.subA.sa1.helloWorld().

Прим. перев. После переименования с помощью as новое имя нельзя использовать в качестве имени пакета или модуля для последующих импортов. Иными словами, команда вроде следующей недействительна: import packA as pA, pA.a1.

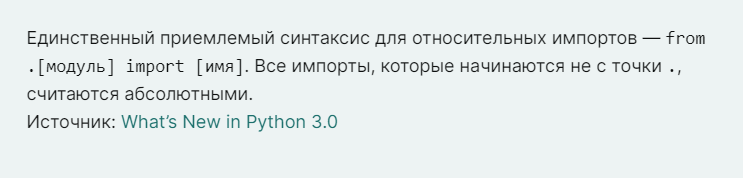
**Абсолютный и относительный импорт**

При абсолютном импорте используется полный путь (от начала корневой папки проекта) к желаемому модулю.

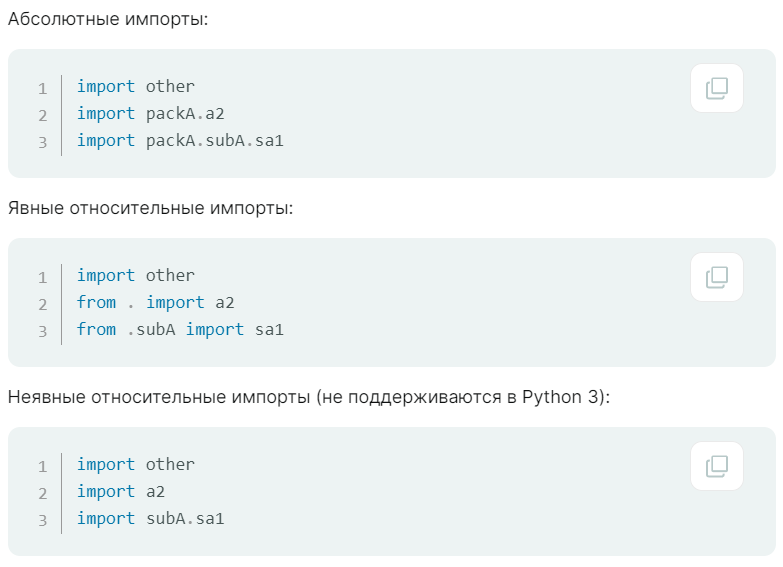
При относительном импорте используется относительный путь (начиная с пути текущего модуля) к желаемому модулю. Есть два типа относительных импортов:

1. При явном импорте используется формат from .<модуль/пакет> import X, где символы точки . показывают, на сколько директорий «вверх» нужно подняться. Одна точка . показывает текущую директорию, две точки .. — на одну директорию выше и т. д.
2. Неявный относительный импорт пишется так, как если бы текущая директория была частью sys.path. Такой тип импортов поддерживается только в Python 2.

В документации Python об относительных импортах в Python 3 написано следующее:

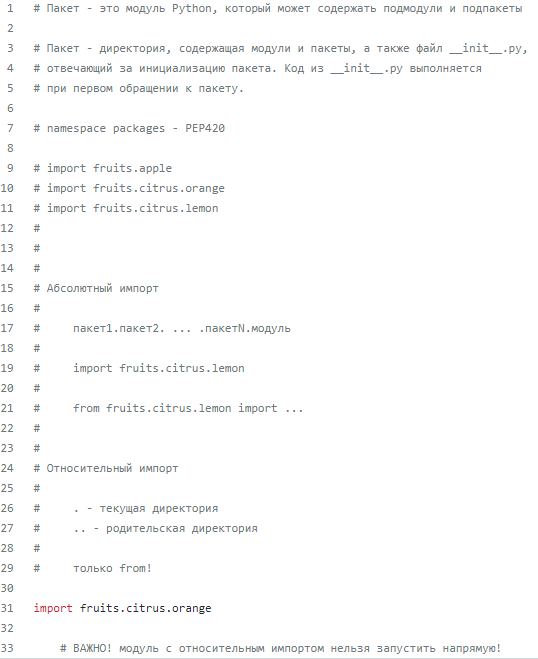


В качестве примера допустим, что мы запускаем start.py, который импортирует a1, который импортирует other, a2 и sa1. Тогда импорты в a1.py будут выглядеть следующим образом:

Учтите, что в относительных импортах с помощью точек . можно дойти только до директории, содержащей запущенный из командной строки скрипт (не включительно). Таким образом, from .. import other не сработает в a1.py. В результате мы получим ошибку ValueError: attempted relative import beyond top-level package.

Как правило, абсолютные импорты предпочтительнее относительных. Они позволяют избежать путаницы между явными и неявными импортами.

Лекция препода по пакетам:



**№21.** Концепции ООП. Классы, объекты, экземпляры, атрибуты, поля, методы**.**

**Объекты и классы**

В Python объекты — это значения, создаваемые на основе шаблона — *класса*. Программист описывает с помощью специального синтаксиса содержимое класса и потом во время исполнения создает объекты — *экземпляры (instances) этого класса*. У класса есть свои данные — атрибуты класса. К ним имеют доступ все экземпляры класса. При этом экземпляры имеют свои атрибуты — атрибуты экземпляра. Эти данные доступны только объекту.

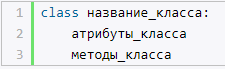
Технически в Python любой объект может получить доступ к содержимому любого другого объекта, если имеет ссылку на него. Но на уровне добровольных соглашений такой доступ можно ограничивать.

Некоторые атрибуты могут быть функциями. В этом случае такие атрибуты называют *методами*.

Python имеет множество встроенных типов, например, int, str и так далее, которые мы можем использовать в программе. Но также Python позволяет определять собственные типы с помощью классов. Класс представляет некоторую сущность. Конкретным воплощением класса является объект.

Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики Человек может выполнять некоторые действия - ходить, бегать, думать и т.д. То есть это представление, которое включает набор характеристик и действий, можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек будет представлять объект этого класса.

Класс определяется с помощью ключевого слова class:



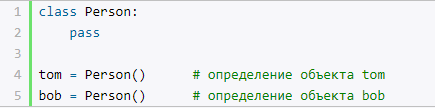
Внутри класса определяются его атрибуты, которые хранят различные характеристики класса, и методы - функции класса.

Создадим простейший класс:



В данном случае определен класс Person, который условно представляет человека. В данном случае в классе не определяется никаких методов или атрибутов. Однако поскольку в нем должно быть что-то определено, то в качестве заменителя функционала класса применяется оператор pass. Этот оператор применяется, когда синтаксически необходимо определить некоторый код, однако мы не хотим его, и вместо конкретного кода вставляем оператор pass.

После создания класса можно определить объекты этого класса. Например:

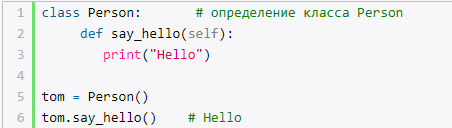


После определения класса Person создаются два объекта класса Person - tom и bob. Для создания объекта применяется специальная функция - конструктор, которая называется по имени класса и которая возвращает объект класса. То есть в данном случае вызов Person() представляет вызов конструктора. Каждый класс по умолчанию имеет конструктор без параметров:



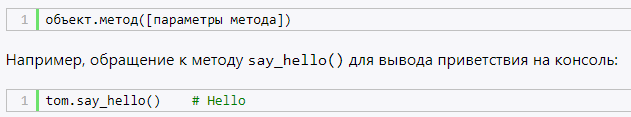
**Методы классов**

Методы класса фактически представляют функции, которые определенны внутри класса и которые определяют его поведение. Например, определим класс Person с одним методом:



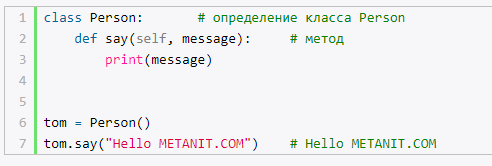
Здесь определен метод say\_hello(), который условно выполняет приветствие - выводит строку на консоль. При определении методов любого класса следует учитывать, что все они должны принимать в качестве первого параметра ссылку на текущий объект, который согласно условностям называется self. Через эту ссылку внутри класса мы можем обратиться к функциональности текущего объекта. Но при самом вызове метода этот параметр не учитывается.

Используя имя объекта, мы можем обратиться к его методам. Для обращения к методам применяется нотация точки - после имени объекта ставится точка и после нее идет вызов метода:



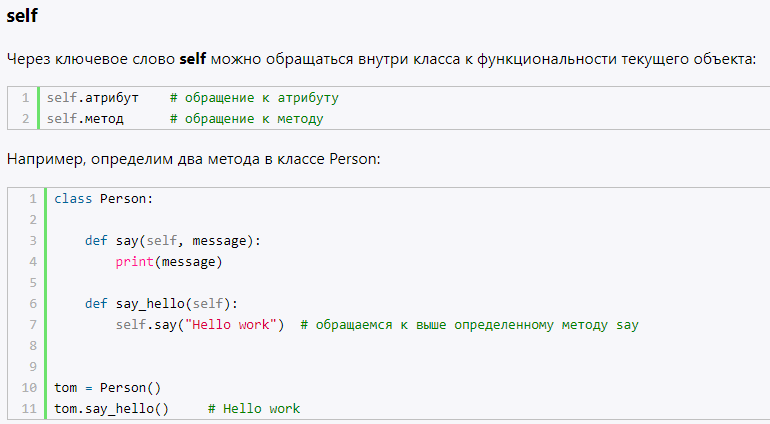
В итоге данная программа выведет на консоль строку "Hello".

Если метод должен принимать другие параметры, то они определяются после параметра self, и при вызове подобного метода для них необходимо передать значения:



Здесь определен метод say(). Он принимает два параметра: self и message. И для второго параметра - message при вызове метода необходимо передать значение

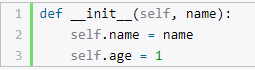
**Атрибуты объекта**



Атрибуты хранят состояние объекта. Для определения и установки атрибутов внутри класса можно применять слово self. Например, определим следующий класс Person: 

Теперь конструктор класса Person принимает еще один параметр - name. Через этот параметр в конструктор будет передаваться имя создаваемого человека.

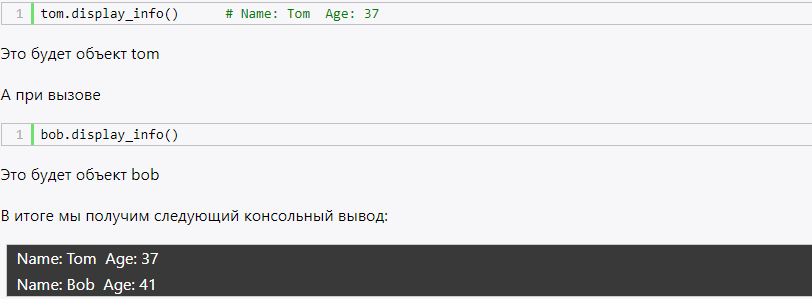
Внутри конструктора устанавливаются два атрибута - name и age (условно имя и возраст человека):



**Создание объектов**

Выше создавался один объект. Но подобным образом можно создавать и другие объекты класса: 

Здесь создаются два объекта класса Person: tom и bob. Они соответствуют определению класса Person, имеют одинаковый набор атрибутов и методов, однако их состояние будет отличаться.

При выполнении программы Python динамически будет определять self - он представляет объект, у которого вызывается метод. Например, в строке: 

Ссылки:

<https://metanit.com/python/tutorial/2.15.php>

<https://tproger.ru/translations/guide-into-python-imports/#1>

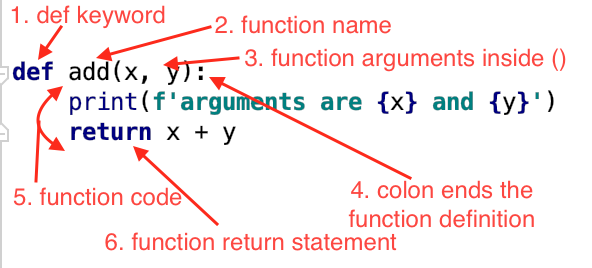
<https://devpractice.ru/python-lesson-13-modules-and-packages/>

<https://metanit.com/python/tutorial/7.1.php>

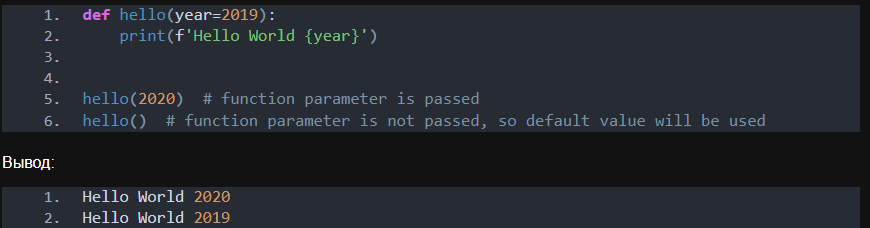
**9.Функции. Чистота функций. Побочные эффекты. Области видимости. LEGB. global и nonlocal.**

**Функции**

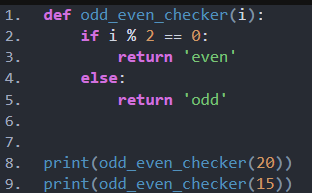
* Функция — это блок кода с именем.
* Мы можем вызвать функцию по ее имени.
* Код внутри функции запускается только при ее вызове.
* Может принимать данные от вызывающей программы, она вызывается как параметры функции.
* Параметры функции заключены в круглые скобки и разделены запятыми. Функция может 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

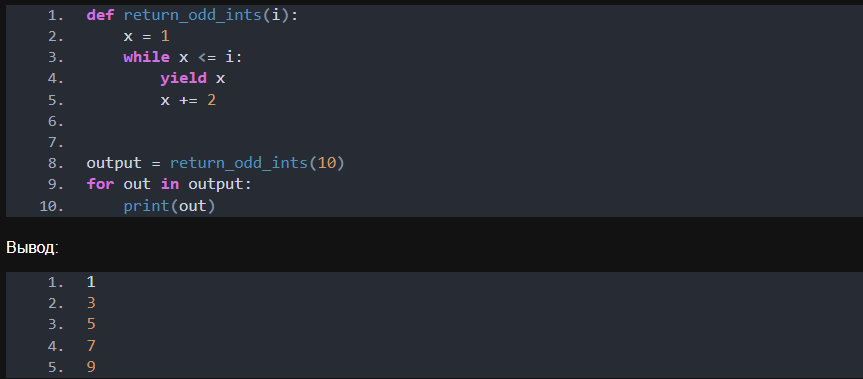


**Можем ли мы иметь несколько операторов возврата внутри функции?**



**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**

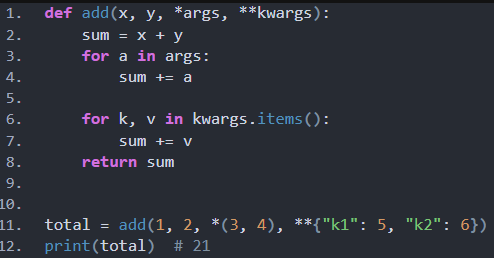
Python допускает три типа параметров в определении функции.

* 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:

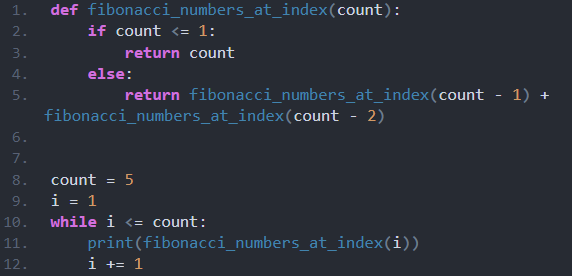
* Порядок аргументов должен быть формальным, \* args и \*\* kwargs.
* Не обязательно использовать имена параметров переменных как args и kwargs. Однако лучше всего использовать их для лучшей читаемости кода.
* Тип args — кортеж. Таким образом, мы можем передать кортеж для отображения с переменной \*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

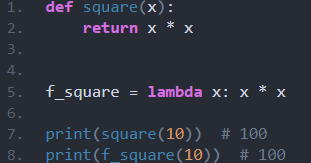
Давайте посмотрим на простой пример функций и методов в Python.

 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

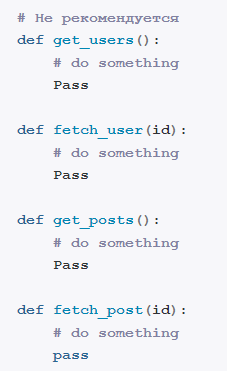
**Анонимная функция**

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

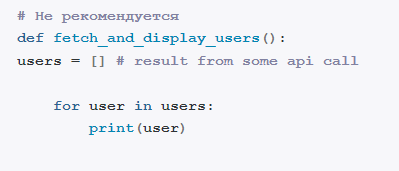
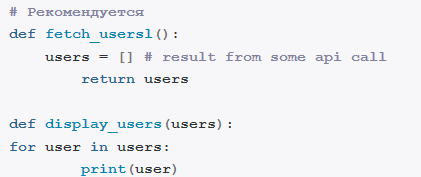
****

**Чистота функций**

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

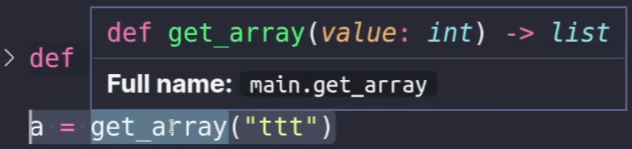


**Каждая функция должна выполнять какое-то одно действие и делать это хорошо.**Пишите короткие и простые функции, выполняющие одну задачу. Полезный прием: если в имени вашей функции появляется союз «и», лучше разделить такую функцию на две разных.

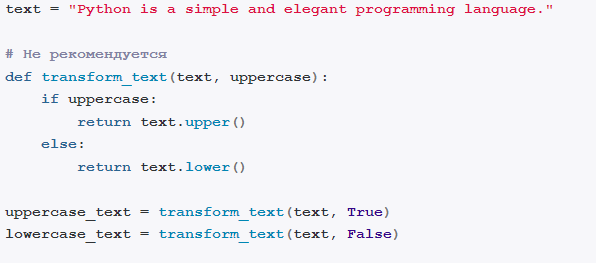
 

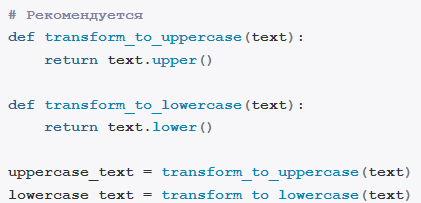
Type **annotations** для функций и внимательно отслеживаем несоответствия по типам данных на уровне проекта. Очень много херни позволяет отловить с минимальной задержкой.



.

**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**

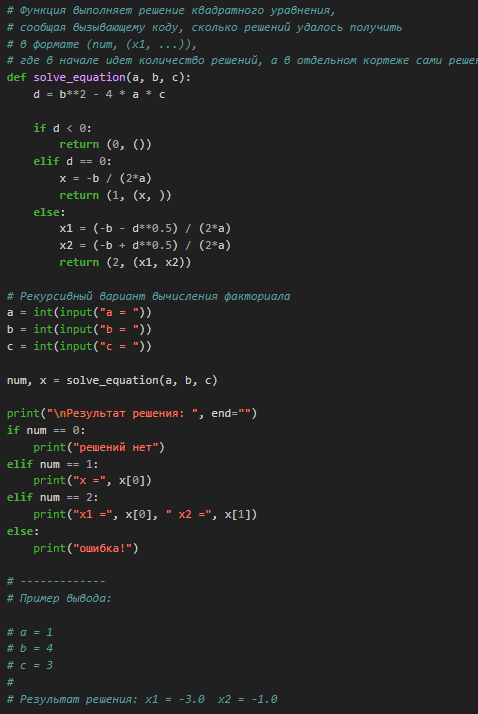
[Побочный эффект](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B1%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (*англ.* Side Effect) - любые действия программы, изменяющие среду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

К побочным эффектам выполнения функции можно отнести:

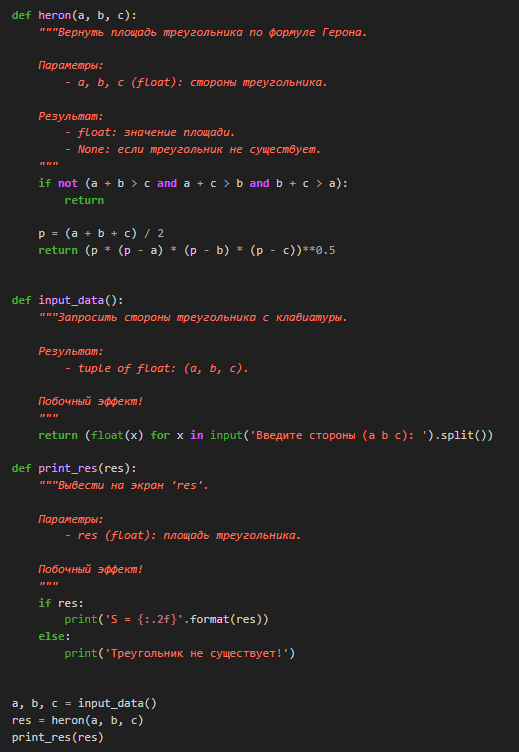
* изменение данных в памяти;
* чтение/запись файла или устройства;
* ввод/вывод значений;
* самостоятельную реакцию на исключительные ситуации;
* и др.

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.

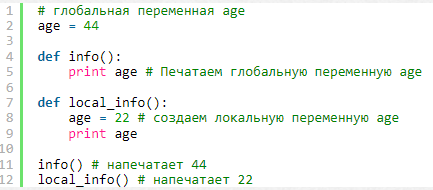
В **Python** две базовых области видимости переменных:

* *Глобальные переменные*
* *Локальные переменные*

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

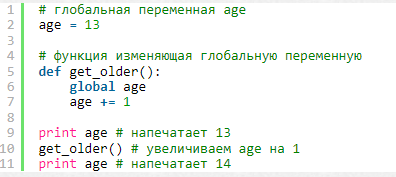
Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

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.

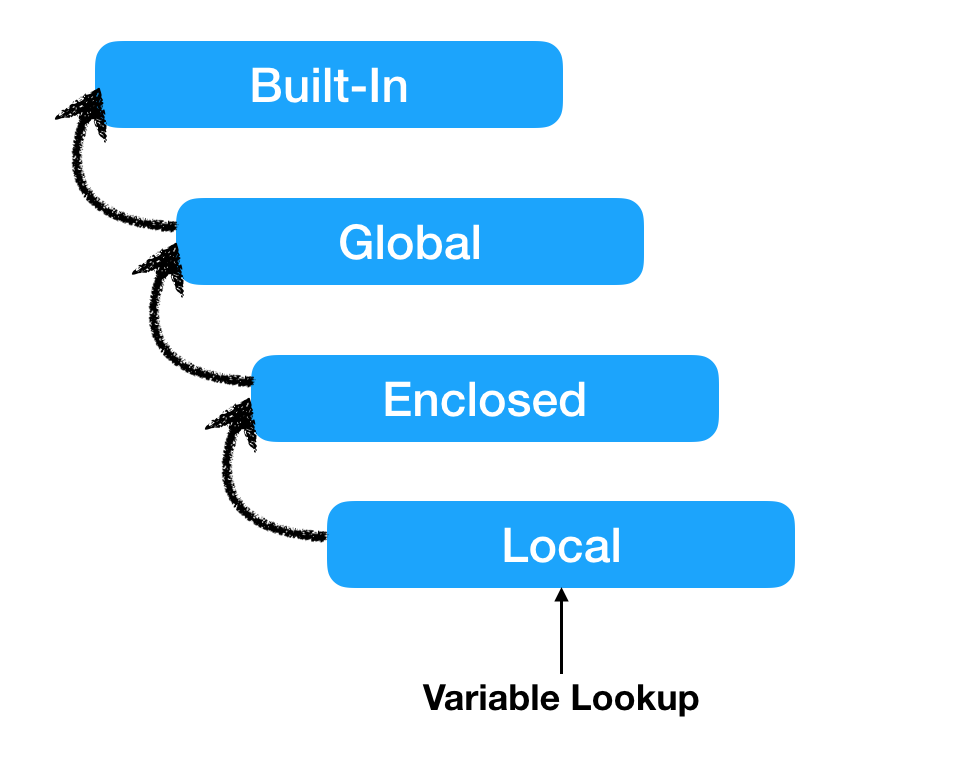
* **local**область видимости — это область действия всех объектов в текущем пространстве имен.
* **охватывающая**область — это область любых объектов в пространстве имен, охватывающих текущее пространство имен, которые могут или не могут существовать. В приведенном ниже примере пространство имен f включает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

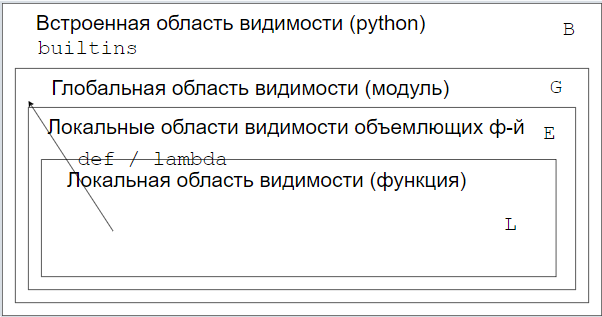
def f (x): def g (y): return yg (x)

* **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/). 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/) и [compile()](https://docs-python.ru/tutorial/vstroennye-funktsii-interpretatora-python/funktsija-compile/).

**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.

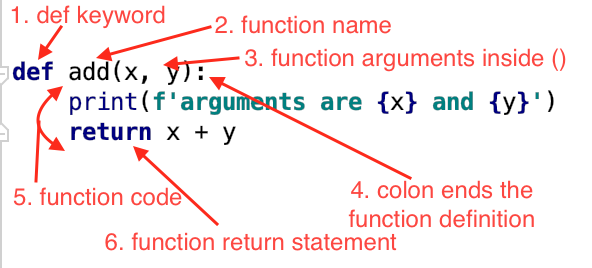
**Обобщая сказанное:**

* 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.

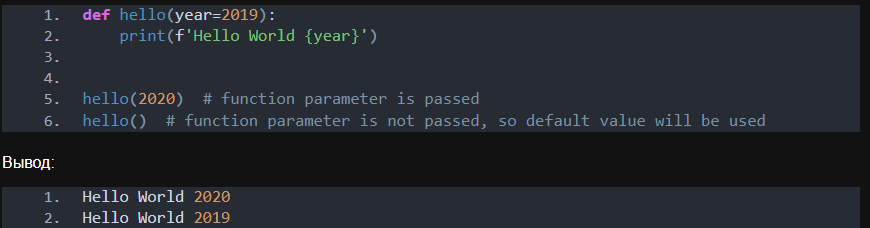
**11. Функции. Документирование и тестирование функций.**

**Функции**

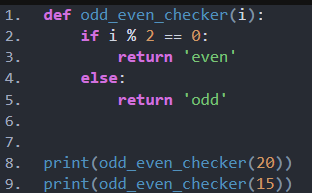
* Функция — это блок кода с именем.
* Мы можем вызвать функцию по ее имени.
* Код внутри функции запускается только при ее вызове.
* Может принимать данные от вызывающей программы, она вызывается как параметры функции.
* Параметры функции заключены в круглые скобки и разделены запятыми. Функция может 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

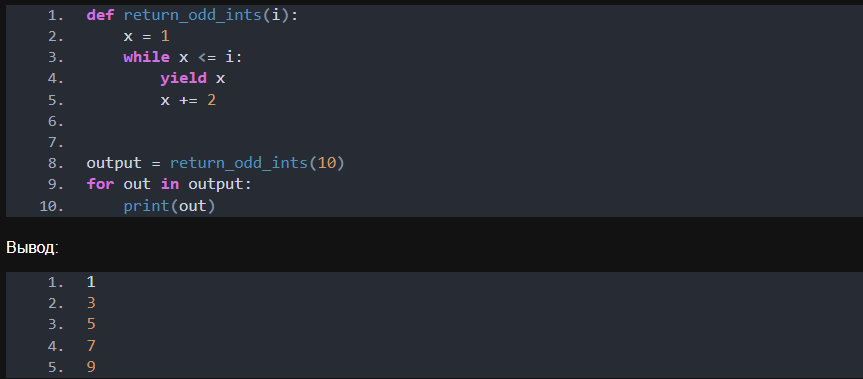


**Можем ли мы иметь несколько операторов возврата внутри функции?**



**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**

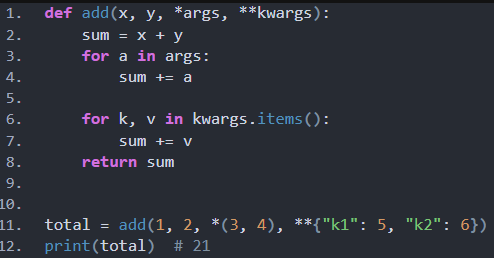
Python допускает три типа параметров в определении функции.

* 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:

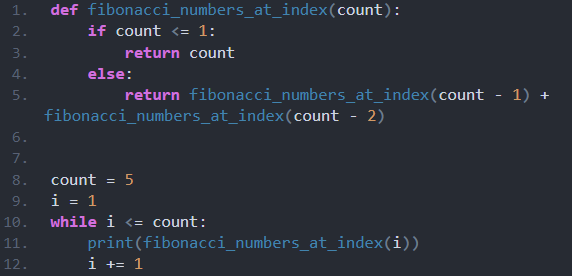
* Порядок аргументов должен быть формальным, \* args и \*\* kwargs.
* Не обязательно использовать имена параметров переменных как args и kwargs. Однако лучше всего использовать их для лучшей читаемости кода.
* Тип args — кортеж. Таким образом, мы можем передать кортеж для отображения с переменной \*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

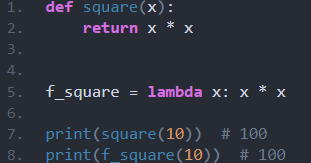
Давайте посмотрим на простой пример функций и методов в Python.

 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.

****

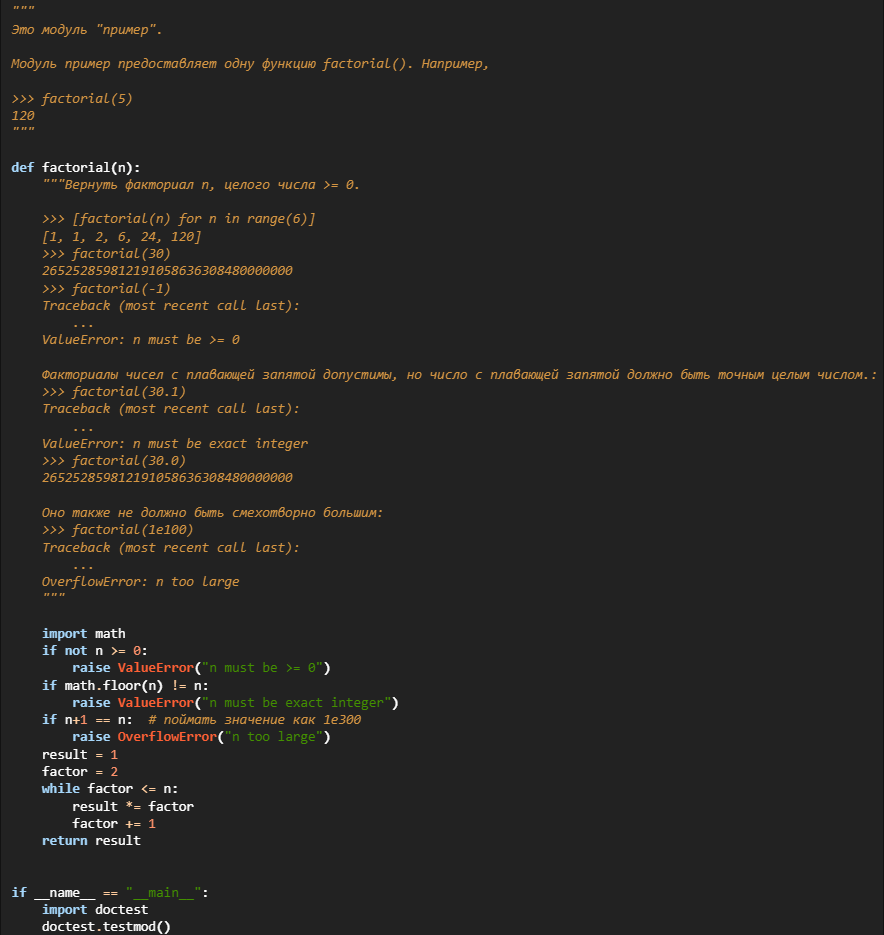
**Документирование и тестирование функций**

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#module-doctest) 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».

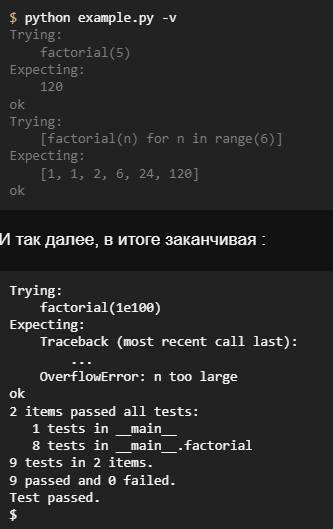
Вот полный, но небольшой пример модуля:



Если вы запустите example.py прямо из командной строки, [doctest](https://digitology.tech/docs/python_3/library/doctest.html#module-doctest) творит чудеса:



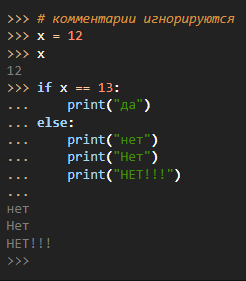
Вывода нет! Это нормально, и это означает, что все примеры рабочие. Передайте **-v**сценарию, и [doctest](https://digitology.tech/docs/python_3/library/doctest.html#module-doctest) напечатает подробный log своих попыток, а в конце напечатает сводку:



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)

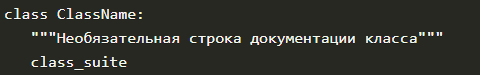
В большинстве случаев копирование и вставка сеанса интерактивной консоли работает нормально, но doctest не пытается выполнить точную эмуляцию какой-либо оболочки Python.



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.

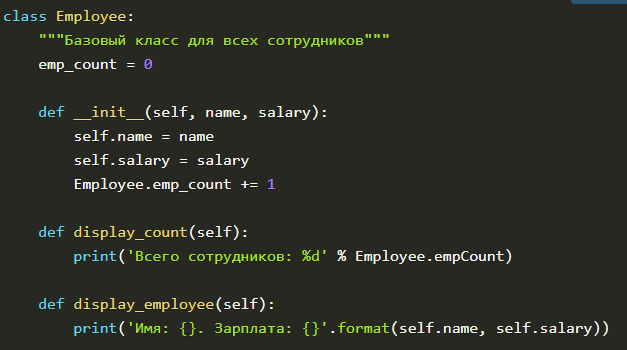
**22.Оператор class и его работа. Классы и экземпляры.**

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.

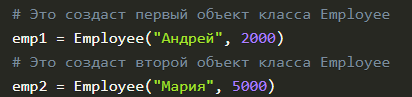
**Пример создания класса на Python**:



* 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.

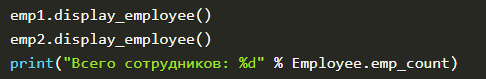
**Создание экземпляров класса**

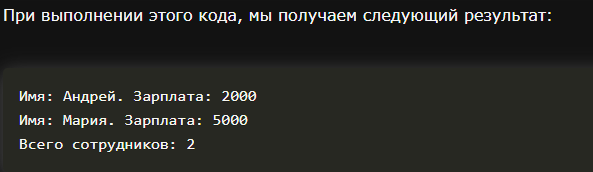
Чтобы создать экземпляры классов, нужно вызвать класс с использованием его имени и передать аргументы, которые принимает метод \_\_init\_\_.

****

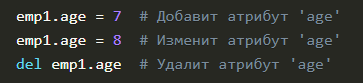
Доступ к **атрибутам**

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



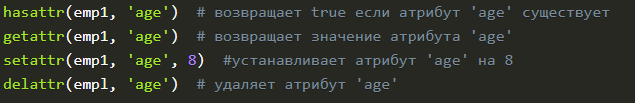


Вы можете добавлять, удалять или изменять атрибуты классов и объектов в любой момент.



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.

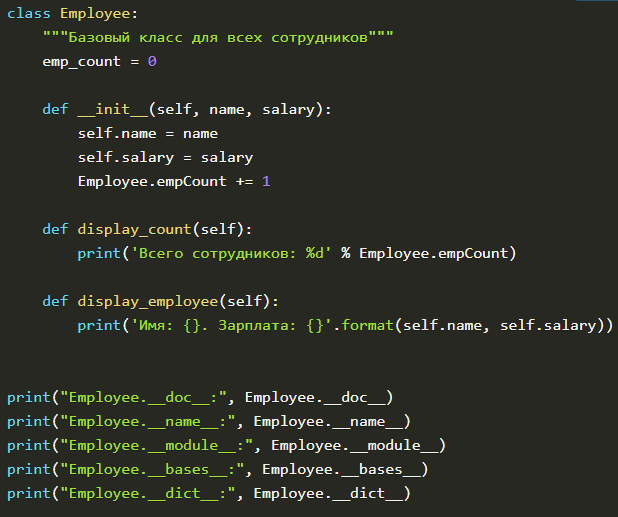


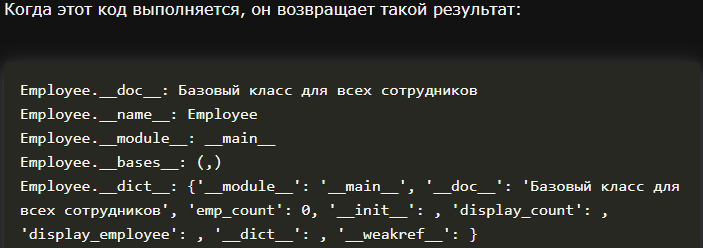
**Встроенные атрибуты класса**

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

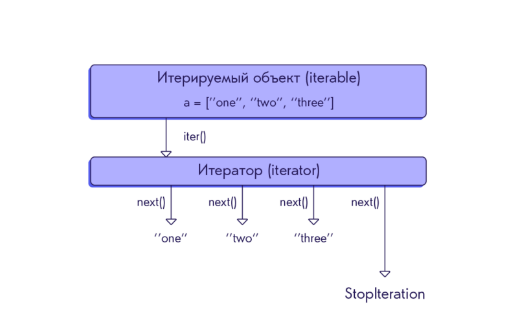
* \_\_dict\_\_ — словарь, содержащий пространство имен класса.
* \_\_doc\_\_ — строка документации класса. None если, документация отсутствует.
* \_\_name\_\_ — имя класса.
* \_\_module\_\_ — имя модуля, в котором определяется класс. Этот атрибут \_\_main\_\_ в интерактивном режиме.
* \_\_bases\_\_ — 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:



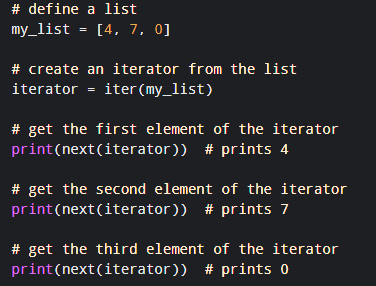
****

Билет 8. Итерации и включения. Протокол итерации

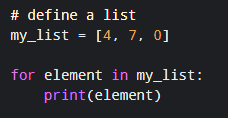
**Итерация** — это повтор какого-либо действия. То есть один шаг цикла.

**Итератор** — это интерфейс, который позволяет получить следующий объект последовательности.

**Итерируемые объекты**— это объекты, которые можно повторять.

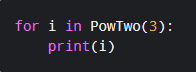
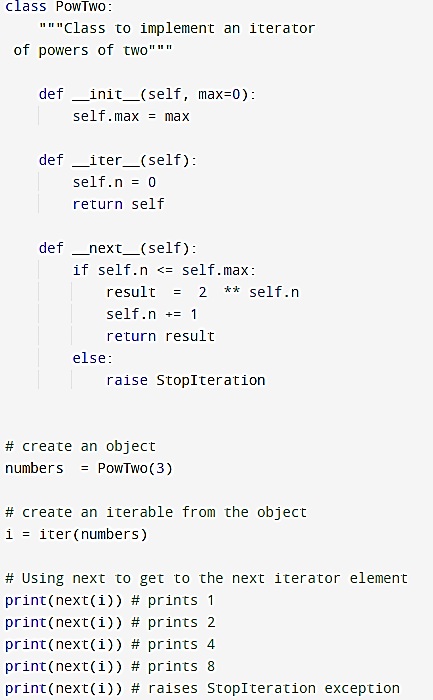
Технически **объект итератора** Python должен реализовывать два специальных метода *\_\_iter\_\_()и\_\_next\_\_(),* которые в совокупности называются **протоколом итератора**. *Метод \_\_iter\_\_ возвращает self — ссылку на экземпляр. С помощью \_\_next\_\_ получают следующий элемент последовательности.*

* **Итерация через итератор**

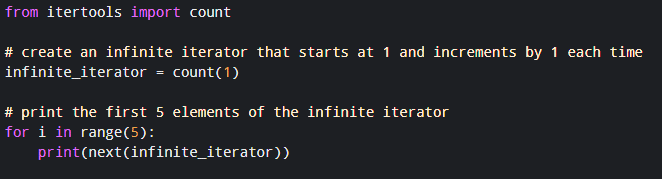
В Python мы можем использовать *next()* функцию для возврата следующего элемента в последовательности. Пример на рис. Здесь сначала мы создали итератор из списка, используя *iter()* метод. А затем использовал *next()* функцию для извлечения элементов итератора в последовательном порядке. Когда мы дойдем до конца и больше не будет возвращаемых данных, мы получим *StopIteration* исключение.

Более элегантный способ автоматической итерации — это использование [цикла *for*](https://www.programiz.com/python-programming/for-loop) .

* **Создание пользовательских итераторов**

Создать итератор с нуля в Python легко. Нам просто нужно реализовать методы *\_\_iter\_\_()*и*\_\_next\_\_().* Пример, который даст нам следующую степень **2** на каждой итерации. Показатель степени начинается от нуля до заданного пользователем числа.   
Мы также можем использовать *for* цикл для перебора нашего класса итераторов. 

* **Бесконечные итераторы Python**

**Бесконечный итератор** — это итератор, который никогда не заканчивается, что означает, что он будет продолжать создавать элементы бесконечно. Вот пример того, как создать бесконечный итератор в Python, используя *count()* функцию из *itertools* модуля

Лучше почитать самим:

1. Понимание итераторов  
   <https://habr.com/ru/post/488112/>
2. Итерации и генераторы  
   https://okpython.net/python/python\_uchebnik/osnovy/iteracii\_i\_generatory\_v\_python.html

При создании коллекций элементов на Python циклы*for* заменяются однострочными выражениями. Python поддерживает четыре типа особенных [*генераторов*](https://ru.wikipedia.org/wiki/Python#%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B), называемых также ***включениями***:

1. **Списковые включения** (*генераторы* *списков, Listcomps*).

**new\_list = [выражение** for **элемент** in **последовательность** if **условие]**

1. **Словарные включения** (*генераторы словарей, Dictcomps*).

**new\_dict = { ключ:значение** for **(ключ,значение)** in **dict.items()** if **условие}**

*Пример:* Давайте возведем в квадрат все числовые значения словаря при помощи словарного включения:

data = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

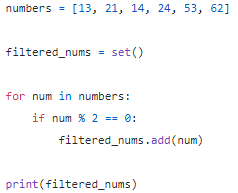
squared = {k:v\*v **for** (k,v) **in** data.items()}print(squared)

*Результат:* *{'a': 1, 'b': 4, 'c': 9, 'd': 16, 'e': 25}*

1. **Множественные включения** (*генераторы множеств, Setcomps*).

**new\_set = { выражение** for **элемент** in **последовательность** if **условие }**

*Пример:* Давайте создадим список произвольных чисел и множество парных чисел на его основе:

при помощи генератора множеств подобная программа пишется в одну строчку:

filtered\_nums = {num **for** num **in** numbers **if** num % 2 == 0}

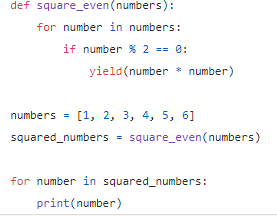
print(filtered\_nums)

*Результат: {24, 62, 14}*

1. **Генераторные выражения** (*GenExp*).

**( выражение** for **элемент** in **последовательность** if **условие )**

*Пример.* Давайте возведем в квадрат все четные числа списка и отбросим все нечетные:



С помощью выражения-генератора можно вообще забыть о потребности в функции *square\_even()* и сделать то же самое с помощью одной строки кода:

squared\_numbers = (num \* num **for** num **in** numbers **if** num % 2 == 0)

**for** number **in** squared\_numbers:

print(number)

*Результат: 4 16 36*

Генераторное выражение можно передавать в качестве параметра любой функции-конструктору стандартных типов данных Python:

**tuple(выражение** for **элемент** in **последовательность** if **условие)**

Разумное применение включений улучшает качество вашего кода, но откажитесь от слепой замены всех циклов на выражения, ведь иногда замена вложенного цикла for на включение снижает понятность кода. **Включения выполняются быстрее**, чем цикл for, однако разница — всего 10%, поэтому вместо производительности **отдавайте предпочтение качеству кода**.

Билет 19. Импортирование. Способы и их работа.

**Как работает импортирование?**

1. *Поиск файла модуля*
   * рабочий каталог
   * каталоги PYTHONPATH
   * каталоги стандартной библиотеки
   * каталог site-packages
   * содержимое файлов .pth
2. *Если модуль найден - он компилируется в байт-код (при необходимости)*
   * если файл с байт-кодом не найден
   * если файл байт-кода старше
   * когда файл байт-кода скомпилирован другой версией интерпретатора
3. *Выполняется код модуля*

**Какие бывают модули?**

1. ***Стандартная библиотека Python*** (os, time, sys, io, re, ...)
   1. *Встроенные модули* - входят в состав интерпретатора, написаны на С (Cpython)
   2. *Модули расширения* - написаны на Python, предназначены для решения частых задач
2. ***Сторонние*** (3rd party) - не поставляются вместе с интерпретатором, но могут быть установлены из репозитория пакетов с помощью пакетных менеджеров (pip) (pandas, numpy, sqlalchemy, ...)
3. **Пользовательские модули** - собственные

**Способы импортирования**

**IMPORT**

Общая форма: ***import модуль / пакет1 [as псевдоним1], модуль / пакет2 [as псевдоним2], ...***

import math, datetime

import math as m *(случай с псевдонимом)*

**FROM ... IMPORT**

Общая форма: ***from модуль / пакет1 import имя / модуль1 [as псевдоним1], имя / модуль2 [as псевдоним2], ...***

Импортируется только конкретный объект (в нашем примере: функция *cos*), остальные функции недоступны, даже если при их вызове указать имя модуля. Для импортирования нескольких функций из модуля, можно перечислить их имена через запятую.

from math import cos

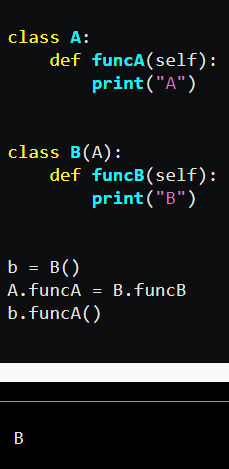
from math import factorial as f *(случай с псевдонимом)*

**FROM МОДУЛЬ / ПАКЕТ IMPORT \***

Импортирование всех функций из модуля

Тут можно почитать (инфы много и что относиться к этому вопросу сложно понять, лучше каждому самому разобраться в этой фигне):

1. <https://ru.hexlet.io/blog/posts/likbez-po-paketam-i-shpargalka-po-modulyam-v-python>
2. https://okpython.net/python/python\_uchebnik/osnovy/moduli\_i\_pakety\_v\_python.html

Билет 23 Наследование. MRO**.**

**Наследование** — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

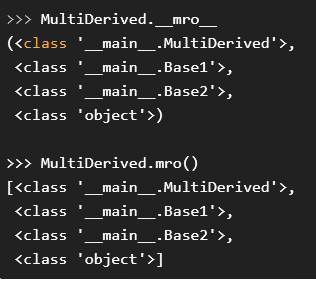
*Чтобы один класс наследовал от другого нужно при объявлении класса передать родительский класс как аргумент.*

Особенность наследования заключается в том, что оно позволяет не просто создать дубликат класса, но и расширить его функционал. Это очень полезно, потому что наследование позволяет повторно использовать уже написанный код.

*Под наследованием понимается возможность создания нового класса на базе существующего. Наследование предполагает наличие отношения “является” между классом наследником и классом родителем. При этом класс потомок будет содержать те же атрибуты и методы, что и базовый класс, но при этом его можно (и нужно) расширять через добавление новых методов и атрибутов.*

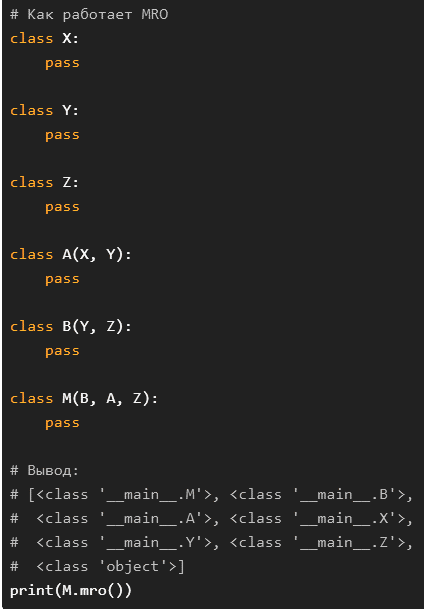
*Примером базового класса, демонстрирующего наследование, можно определить класс “автомобиль”, имеющий атрибуты: масса, мощность двигателя, объем топливного бака и методы: завести и заглушить. У такого класса может быть потомок – “грузовой автомобиль”, он будет содержать те же атрибуты и методы, что и класс “автомобиль”, и дополнительные свойства: количество осей, мощность компрессора и т.п..*

Некоторое видео про наследование:

1. Наследование в ООП.  
   <https://www.youtube.com/watch?v=bmtMUWWFRIk>
2. Множественное наследование.  
   <https://www.youtube.com/watch?v=_475mVLClM8> 

Список правил, по которому мы находим такой порядок, называется **Method Resolution Order** (*порядок разрешения методов*).

MRO должен сохранять локальный порядок старшинства, а также обеспечивать монотонность. Он гарантирует, что класс всегда будет появляться до родителей. В случае нескольких родителей порядок будет таким же, как у кортежей в базовом классе.

MRO класса можно просмотреть в атрибуте *\_\_mro\_\_* или с помощью метода *mro()*. *Вызов атрибута возвращает кортеж, а вызов метода — список.* 

Еще более сложный пример множественного наследования и его визуализация по MRO: