Интерфейсы

Интерфейсы (перегрузка операторов). Интерфейс итерации (iter(), next()). Метод как атрибут. Статические методы. Расширение встроенных типов.

[Перегрузка операторов](#_2aopvv2ib0d2)

[Интерфейс итерации](#_xtq6wk4o0830)

[Собственный итератор](#_ybfvvxjw7vgy)

[Свойство @property](#_r6k0jjwal8eq)

[Статические методы](#_w8iiovtviitb)

[Расширение встроенных типов](#_huqwgbr8s7qr)

[Домашнее задание](#_q81f2g28ili5)

[Дополнительные материалы](#_ms0pceoqro6)

[Используемая литература](#_uvp6qax5r1ok)

# 

# Перегрузка операторов

Перегрузка операторов позволяет объектам, созданным из классов, перехватывать и участвовать в операциях, которые применяются к встроенным типам: сложение, вычитание, получение среза, вывод и так далее. По большей части это автоматический механизм: при выполнении выражений и других встроенных операций интерпретатор передаёт управление реализации классов.

Мы можем полностью реализовать поведение класса в виде методов, однако перегрузка операторов позволяет объектам действовать так же, как действуют встроенные объекты, потому что она позволяет создавать менее противоречивые и более простые в изучении интерфейсы объектов и обеспечивает возможность обрабатывать объекты, созданные из классов, программным кодом, который предполагает взаимодействие со встроенными типами.

**Имена методов, начинающиеся и заканчивающиеся двумя символами подчеркивания (\_\_X\_\_), имеют специальное назначение.** Перегрузка операторов в языке Python реализуется за счет создания методов со специальными именами для перехвата операций. Язык Python определяет фиксированные и неизменяемые имена методов для каждой из операций. Классы могут переопределять большинство встроенных методов, такие методы вызываются автоматически. Например, если объект экземпляра наследует метод \_\_add\_\_, этот метод будет вызываться всякий раз, когда объект будет появляться в операции сложения (+). Возвращаемое значение метода становится результатом соответствующей операции.

|  |
| --- |
| class Vector:  def \_\_init\_\_(self, pos):  self.x = pos[0]  self.y = pos[1]  # Перегружаем оператор +  def \_\_add\_\_(self, other):  return Vector((self.x + other.x, self.y + other.y))  def as\_point(self):  return self.x, self.y  # Формируем удобное отображение объекта при выводе функции print()  def \_\_str\_\_(self):  return "V(x:{} y:{})".format(self.x, self.y)  # Создаём экземпляры класса (объекты)  v1 = Vector((10, 15))  v2 = Vector((12, 10))  # Наши объекты участвуют в операции сложения (+)  v3 = v1 + v2  # На самом деле это работает так:  # v3 = v1.\_\_add\_\_(v2)  # Благодаря перегрузке мы можем использовать более удобную и привычную запись: v3 = v1+v2 |

Обратите внимание, что метод \_\_add\_\_создаёт и возвращает новый объект экземпляра этого класса.

Аналогичным образом можно перегружать практически все встроенные операций. Полный список magic-методов (именно так они называются в официальной литературе) с описанием можно посмотреть [здесь](http://pythonworld.ru/osnovy/peregruzka-operatorov.html).

Благодаря перегрузке операторов, объекты, реализованные на базе классов, действуют подобно встроенным типам, и тем самым обеспечивают непротиворечивые и совместимые интерфейсы.

Конструктор \_\_init\_\_ так же является примером перегрузки операции и вызывается автоматически при создании экземпляра класса.

Перегрузка операторов является частью механизма полиморфизм, который мы уже рассматривали.

# Интерфейс итерации

С итераторами мы уже знакомы. Разберём, как они устроены внутри и научимся создавать собственные объекты-итераторы.

**Итераторы** — это специальные объекты, представляющие последовательный доступ к данным из контейнера.

При этом немаловажную роль играет то, что память фактически не тратится, так как промежуточные данные выдаются по мере необходимости при запросе, поэтому фактически в памяти останутся только исходные данные и конечный результат, да и их можно читать и записывать, используя файл на диске.

С итераторами работают:

* цикл for in.
* Встроенные функции map(), filter(), zip() и прочие.
* Орерация распаковка \*.

На самом деле, все эти инструменты будут работать с любым объектом, который поддерживает интерфейс итерации. Т.е. предоставляет определенный набор методов, которыми пользуются инструменты, работающие с итераторами.

В python понятие интерфейса несколько отличается от интерфейса в других языках. Если вы можете применить определенный инструмент (функцию, инструкцию, оператор), значит этот объект поддерживает данный интерфейс. Наличие определенного интерфейса определяется наличием нужного методы, да, все так просто!

Вернемся к итераторам.

|  |
| --- |
| test\_list = [1, 2, 3, 4, 'END']  # Когда вы пишете  for el in test\_list:  print(test\_list) |

Python выполняет:

1. Вызывает метод \_\_iter\_\_(): test\_list.\_\_iter\_\_()  
   Метод \_\_iter\_\_() должен вернуть объект у которого есть метод \_\_next\_\_()
2. Цикл for in каждую итерацию вызывает метод \_\_next\_\_()  
   \_\_next\_\_() при каждом вызове возвращает следующий элемент итератора
3. Когда элементы итератора заканчиваются, метод \_\_next\_\_() возбуждает исключение StopIteration  
   for in завершает свою работу, когда перехватывает это исключение

Список сам по себе не итератор, у него есть метод, который возвращает объект-итератор.

Важной особенностью любого объекта-итератора является то, что по нему можно пройти только один раз. Т.е. после вызова next() итератор запоминает своё состояние. Если хотите выполнить итерацию по списку ещё раз, нужно снова получить новый объект-итератор.

## 

## 

## Собственный итератор

Пример создания объекта, поддерживающего интерфейс итерации:

|  |
| --- |
| class IterObj:  """  Объект итератор  """  def \_\_init\_\_(self, start=0):  self.i = start  # Объект считается итератором - если у него есть метод \_\_next\_\_  def \_\_next\_\_(self):  self.i += 1  if self.i <= 5:  return self.i  else:  raise StopIteration  class Iter:  """  Объект поддерживающий интерфейс итерации  """  def \_\_init\_\_(self, start=0):  self.start = start - 1  def \_\_iter\_\_(self):  # Метод \_\_iter\_\_ должен возвращать объект-итератор  return IterObj(self.start)  obj = Iter(start=2)  for el in obj:  print(el)  print("Еще раз ...")  for el in obj:  print(el)  print('sum(obj) -->', sum(obj))  # Функция map() возвращает объект-итератор  map\_iter = map(int, '123')  print('next(map\_iter) --> ', next(map\_iter))  print('next(map\_iter) --> ', next(map\_iter))  # Цикл for in продолжает перебор элементов, т.к. map\_iter является итератором  for el in map\_iter:  print("el in for in -->", el)  class Iter2:  def \_\_init\_\_(self, start=0):  self.i = start  def \_\_iter\_\_(self):  # Метод \_\_iter\_\_ должен возвращать объект-итератор  return self  def \_\_next\_\_(self):  self.i += 1  if self.i <= 5:  return self.i  else:  raise StopIteration  print("Demo Iter2")  obj = Iter2(start=2)  for el in obj:  print(el)  print("Еще раз ...")  for el in obj:  print(el) |

**Обратите внимание!** Экземпляры Iter() - объекты, возвращающие объект итератор. Экземпляры Iter2() - сами по себе итераторы и пройти по ним можно только один раз.

# Свойство @property

@property - это декоратор, но с декораторами мы познакомимся только в следующей части курса по python, пока воспринимайте их как инструкции, которые добавляют дополнительные возможности.

С декоратором @property мы уже встречались на прошлом уроке, когда разбирали инкапсуляцию. Сейчас дам предостережение и объясню, почему этим инструментом не стоит злоупотреблять.

Например, мы использовали @property так:

|  |
| --- |
| class Person(object):  def \_\_init\_\_(self, first\_name, last\_name):  self.first\_name = first\_name  self.last\_name = last\_name  @property  def full\_name(self):  return " ".join([self.first\_name, self.last\_name])  person = Person("Andrey", "Popp")  person.full\_name # "Andrey Popp" |

Это реализация attribute getter/setter шаблона. То есть, обращаюсь к full\_name как к атрибуту, мы неявно вызываем соответствующий метод, и его результат как бы становится значением атрибута. Почему “как бы”? Потому что он будет заново вычисляться при каждом обращении к full\_name.

Операция обращения к атрибуту (а именно так выглядит работа с property) по своей семантике должна быть “дешёвой”. Со свойствами можно сделать следующее – представьте себе, что вы, находясь в консоли Python, обращаетесь к некоему атрибуту myobj.some\_info, в результате чего отправляется запрос в БД, и вычисление свойства происходит в течении n секунд – крайне неприятно. Получается, если обратиться к такому свойству в цикле из 100 итераций, то будет сделано 100 отдельных запросов к БД ,и при этом программист, читающий код, может этого совершенно не заметить.

Отсюда вердикт – если для вычисления свойства нужно произвести побочные эффекты, то лучше сделать это свойство методом.

# Статические методы

На прошлом уроке, когда мы разбирали механизм наследования, использовали класс Учителя

|  |
| --- |
| class Teacher(People):  def \_\_init\_\_(self, name, surname, birth\_date, school, teach\_classes):  People.\_\_init\_\_(self, name, surname, birth\_date, school)  self.teach\_classes = list(map(self.convert\_class, teach\_classes))  # Уникальный метод Учителя  def convert\_class(self, class\_room):  """  '<class\_num> <class\_int>' --> {'class\_num': <class\_num>, 'class\_int': <class\_int>}  """  return {'class\_num': int(class\_room.split()[0]),  'class\_char': class\_room.split()[1]} |

Метод convert\_class() принимает автоматический аргумент self, но его не использует. Подобные методы рекомендуется делать статическими. Для этого используется декоратор @staticmethod.

По сути статический метод не принимает автоматический аргумент self, и тем самым говорит программисту, что использование данного метода никак не связано с обработкой атрибутов/методов текущего экземпляра.

|  |
| --- |
| @staticmethod  def convert\_class(class\_room):  """  '<class\_num> <class\_int>' --> {'class\_num': <class\_num>, 'class\_int': <class\_int>}  """  return {'class\_num': int(class\_room.split()[0]),  'class\_char': class\_room.split()[1]} |

# Расширение встроенных типов

Помимо реализации объектов новых типов, классы иногда используются для расширения функциональных возможностей встроенных типов языка Python, с целью обеспечения поддержки более экзотических структур данных.

|  |
| --- |
| # Расширяем стандартный class dict  class my\_dict(dict):  # Добавляем свой метод  def new\_method(self):  return "I'm new\_method"  # Добавляем дополнительный функционал к существующему методу  def \_\_setitem\_\_(self, key, value):  print('Setting %r to %r' % (key, value))  return super().\_\_setitem\_\_(key, value)  m\_dict = my\_dict({1: 2, 2: 3})  print(m\_dict)  # Данная операция вызывает метод \_\_setitem\_\_  m\_dict["new\_key"] = "new\_value"  print(m\_dict)  # print(m\_dict.keys())  print(m\_dict.new\_method()) |

**Обратите внимание!** При использовании механизма наследования, можно не только добавлять и переопределять методы, но и расширять (модифицировать) существующие. Подробно о функции super() поговорим во второй части курса.

Еще один пример. Предположим, вас не устраивает тот факт, что стандартные списки начинают отсчёт элементов с 0, а не с 1. Это не проблема – вы всегда можете создать свой подкласс, который изменит эту характерную особенность списков.

|  |
| --- |
| class MyList(list):  """  Список - индексы которого начинаются с 1, а не с 0  """  def \_\_getitem\_\_(self, offset):  print('(indexing % s at % s)' % (self, offset))  return list.\_\_getitem\_\_(self, offset - 1)  x = MyList('abc') # \_\_init\_\_ наследуется из списка  print(x) # \_\_repr\_\_ наследуется из списка  print(x[1]) # MyList.\_\_getitem\_\_  print(x[3]) # Изменяет поведение метода суперкласса  x.append('spam')  print(x) # Атрибуты, унаследованные от суперкласса list  x.reverse()  print(x) |

# Домашнее задание

Домашние задания для данного урока вы найдете в папке home\_work в папке с номером текущего урока, скачав проект [здесь](https://github.com/NADiP-Examples/Python_lessons_basic).

Большинство заданий делятся на три категории easy, normal и hard.

* easy - простенькие задачи, на понимание основ.
* normal - если вы делаете эти задачи, то вы хорошо усвоили урок.
* hard - наиболее хитрые задачи, часто с подвохами, для продвинутых слушателей.

Если вы не можете сделать normal задачи - это повод пересмотреть урок, перечитать методичку и обратиться к преподавателю за помощью.

Если не можете сделать hard - не переживайте, со временем научитесь.

Решение большинства задач будем разбирать в начале каждого вебинара.

# Дополнительные материалы

Дополнительные примеры с подробными комментариями находятся [здесь](https://github.com/NADiP-Examples/Python_lessons_basic).

Всё то, о чем сказано здесь, но подробнее:

1. [Перегрузка операторов](http://pythonworld.ru/osnovy/peregruzka-operatorov.html)
2. [Итераторы](https://habrahabr.ru/post/50026/)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Самоучитель по python](http://pythonworld.ru/samouchitel-python)
2. [Книга Лутц М. “Изучаем Python” (4-е издание)](http://www.shashkovs.ru/_prog/Lutc_M._-_Izuchaem_Python_(4-e_izdanie)-_2011.pdf).