Объектно-Ориентированное Программирование

Введение в ООП. Классы. Инкапсуляция, полиморфизм, наследование.

[Словарь как структура данных](#_2aopvv2ib0d2)

[Введение в ООП](#_s4sdipyji4ra)

[Можно ли жить без ООП?](#_9kidtqf1wuqq)

[Классы](#_1m2xxv4ldseq)

[Примеры](#_t5ltfkx975n5)

[Основные свойства ООП](#_bm2ueu3gg1nt)

[Инкапсуляция](#_xrcm4i1nfh3v)

[Наследование](#_u1moefuxx6vj)

[Полиморфизм](#_3vpneg4eyy5h)

[Домашнее задание](#_q81f2g28ili5)

[Дополнительные материалы](#_ms0pceoqro6)

[Используемая литература](#_uvp6qax5r1ok)

# 

# Словарь как структура данных

Когда вам нужно работать с некоей сущностью, у которой можно выделить именованные параметры. Например, со студентами, каждый студент имеет именованные параметры, такие как: имя, фамилия, дата рождения и т.д., то для описания и работы с такой структурой неплохо под ходит словарь

|  |
| --- |
| student = {  "name": "Александр",  "surname": "Иванов",  "birth\_date": '10.11.1998',  "school": "8 гимназия",  "class\_room": "5 А"  }  student2 = {  "name": "Александр",  "surname": "Иванов",  "birth\_date": '10.11.1998',  "school": "8 гимназия",  "class\_room": "5 А"  } |

Потом возникает необходимость изменять и обрабатывать данные каждого студента, для этих задач вы начинаете писать функции.

|  |
| --- |
| # но функция - не очень удобное решение  def next\_class(current\_class):  return str(int(current\_class.split()[0]) + 1) + \  ' ' + current\_class.split()[1]  # т.к. количество функций постоянно увеличивается  def change\_class(current\_class):  pass |

И сложно запомнить, какая функция применяется к какому объекту.

Представьте, что у вас в программе 20 разных структур и для каждой по 10 функций обработки (для реальной программы это совсем немного). Получается 200 (!) перемешанных друг с другом функций, в которых вы рано или поздно запутаетесь.

Гораздо удобнее работать со сгруппированными структурами данных с помощью классов…

# Введение в ООП

Весь окружающий нас мир состоит из объектов: стол, парта, доска - привычные нам объекты. У объектов есть определенные параметры (свойства): цвет, размеры, вес, материал, координаты и прочее. Рассматривая ООП подход в программировании, мы будем опираться на аналогии с объектами реального мира.

Чтобы научиться писать полноценные ООП-программы, нужно понять суть этого подхода и преимущества, которые даёт данный подход, проникнуться философией ООП.

### Можно ли жить без ООП?

Однозначно можно. Есть языки программирования вообще не поддерживающие концепцию ООП(например Pascal). Любую программу, написанную с ООП-подходом, можно написать и без него. Так зачем же тогда тратить столько времени на изучение ООП?

Запомните: ООП это всего лишь инструмент, созданный для того, чтобы оказать вам помощь в написании больших программ.

Python является полностью объектно-ориентированным языкам, в python всё является объектами. Только поняв ООП, вы сможете писать красивые, функциональные и легкорасширяемые программы. Не ждите, что вы поймете всю суть ООП за 2-3 недели, это довольно ёмкая концепция. Читайте лекции, выполняйте задания, пробуйте применять ООП-подход в своих проектах и со временем всё станет на свои места. Как говорится: “Дорогу осилит идущий”.

### Классы

Что такое класс или тип? В реальном мире стол — это объект. Но когда его изготавливают, то руководствуются определенным описанием (знанием), что такое стол? Я могу сказать «стол», не имея в виду какой-то конкретный, но большинство поймут, о чём идет речь, т.к. знают особенности этого предмета (крышка, четыре ножки и т. п.).

Класс — это описание объектов определённого типа. В каком-то смысле это абстракция без материального воплощения, которая позволяет систематизировать объекты той или иной системы.

На основе классов создаются объекты. Может быть множество объектов, принадлежащих одному классу. С другой стороны, может быть класс без объектов, реализованных на его основе.

**Класс** — это пользовательский тип, состоящий из методов и атрибутов.

С точки зрения ООП, класс представляет собой коллекцию данных. Использование классов даёт нам прежде всего преимущества абстрактного подхода в программировании.

Экземпляр класса (объект) создается путём вызова имени класса как функции с параметрами. Объект состоит из атрибутов и методов.

**Атрибут** — это переменная класса, **метод** — это функция. Отличия метода от функции в том, что у него есть первый параметр — **self**, который является ссылкой на тот объект, для которого был вызван метод. Т.е. метод всегда имеет доступ (через self) к атрибутам и другим методам класса, для которого вызван.

Основные свойства ООП — полиморфизм, наследование, инкапсуляция.

### 

### 

### Примеры

|  |
| --- |
| # class - шаблон для создания объектов  # Классы содержат атрибуты - данные, и методы - функции для обработки данных  class Student:  # функция-конструктор - запускается автоматически при создании объекта (экземпляра класса)  def \_\_init\_\_(self, name, surname, birth\_date, school, class\_room):  self.name = name  self.surname = surname  self.birth\_date = birth\_date  self.school = school  self.class\_room = class\_room  # метод  def next\_class(self):  self.class\_room = str(int(self.class\_room.split()[0]) + 1) + ' ' +\  self.class\_room.split()[1]  def get\_full\_name(self):  return self.name + ' ' + self.surname  def set\_name(self, new\_name):  self.name = new\_name |

Используя класс как шаблон, можно создать любое количество объектов (экземпляров класса).

|  |
| --- |
| student1 = Student("Александр", "Иванов", '10.11.1998', "8 гимназия", "5 А")  student2 = Student("Петр", "Сидоров", '10.01.1995', "8 гимназия", "8 Б") |

Аргументы передаются в функцию-конструктор.

Теперь в памяти существуют два независимых объекта типа (класса) Student, но с разным набором атрибутов. Класс - это просто шаблон, он описывает структуру объекта и его поведение.

Для обращения к атрибутам объектов используется точечный синтаксис.

|  |
| --- |
| # Выводим текущий класс первого ученика  print(student1.class\_room)  # Вызываем метод, который переводит ученика в следующий класс  student1.next\_class()  # Проверяем, изменился ли класс  print(student1.class\_room)  # Выводим текущий класс второго ученика. Классы имеют общую структуру и методы, но различные данные  print(student1.class\_room) |

Методы - это обычные функции, но получающие в качестве первого аргумента self ссылку на экземпляр класса.

|  |
| --- |
| # Вызываем метод, для получения полного отображаемого имени студента  print(student1.get\_full\_name())  print(student2.get\_full\_name()) |

По сути, вызов student1.get\_full\_name(), равносилен вызову get\_full\_name(student1), только в случае с методами ссылка на экземпляр передается автоматически.

|  |
| --- |
| # Можно изменить значение любого атрибута, присвоив ему новое значение  student1.name = 'Вася'  print(student1.name) |

Создавая новый класс, мы практически создаем новый тип данных. Между классами и типами есть разница, но пока вы можете считать эти понятия равнозначными.

# Основные свойства ООП

## Инкапсуляция

**Инкапсуляция** - ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.

Цель инкапсуляции - скрыть внутреннюю реализацию класса от пользователей. Для работы с данными, хранящимися в классе, определяются публичные методы.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

|  |
| --- |
| # Исходный класс Студента  class Student:  def \_\_init\_\_(self, name, surname, birth\_date, school, class\_room):  self.name = name  self.surname = surname  self.birth\_date = birth\_date  self.school = school  self.class\_room = class\_room  def next\_class(self):  self.class\_room = str(int(self.class\_room.split()[0]) + 1) + ' ' + \  self.class\_room.split()[1]  def get\_full\_name(self):  return self.name + ' ' + self.surname  def set\_name(self, new\_name):  self.name = new\_name |

Теперь мы пользуемся классом Student, я его импортирую в различные свои программы.

|  |
| --- |
| students = [Student("Александр", "Иванов", '10.11.1998', "8 гимназия", "5 А"),  Student("Петр", "Сидоров", '10.01.1995', "8 гимназия", "8 Б"),  Student("Иван", "Петров", '12.11.1999', "8 гимназия", "4 В"),  ]  # Учебный год закончился  for student in students:  student.next\_class()  for num, student in enumerate(students, start=1):  print("{}) {} класс: {}".format(num, student.get\_full\_name(), student.class\_room))  # ... и ещё много кода с использованием Student |

Но потом вы заметили, что реализация метода .next\_class() смотрится очень плохо. Чтобы написать хороший код, решили хранить информацию о классе по-другому.

|  |
| --- |
| class Student:  def \_\_init\_\_(self, name, surname, birth\_date, school, class\_room):  # ...  self.class\_room = {'class\_num': int(class\_room.split()[0]),  'class\_char': class\_room.split()[1]}  # Класс храним в виде словаря {номер класс , буква класса}  # ...  # Теперь метод выглядит понятно и легкочитаемо  def next\_class(self):  self.class\_room['class\_num'] += 1 |

Чтобы использовать новую реализацию класса, вам придётся переписать много готового кода. Т.к. атрибут class\_room - это словарь, а не строка.

Чтобы этого избежать, делаем следующим образом:

|  |
| --- |
| class Student:  def \_\_init\_\_(self, name, surname, birth\_date, school, class\_room):  self.name = name  self.surname = surname  self.birth\_date = birth\_date  self.school = school  # Символ нижнего подчеркивания говорит пользователям класса, что атрибут для внутреннего использования  self.\_class\_room = {'class\_num': int(class\_room.split()[0]),  'class\_char': class\_room.split()[1]}  # @property - позволяет обращаться к методу как к атрибуту  # .class\_room() --> .class\_room  @property  def class\_room(self):  return "{} {}".format(self.\_class\_room['class\_num'], self.\_class\_room['class\_char'])  def next\_class(self):  self.\_class\_room['class\_num'] += 1  def get\_full\_name(self):  return self.name + ' ' + self.surname  def set\_name(self, new\_name):  self.name = new\_name |

Старый код продолжает работать с новым классом без изменений.

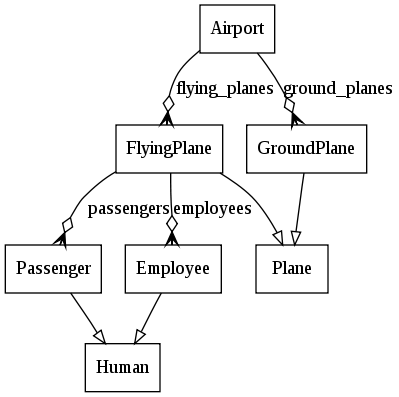
Посмотреть данный пример и его работу, а так же ещё несколько демонстраций инкапсуляции вы можете в демонстрационном проекте, ссылка на который дана в конце лекции.

## Наследование

**Наследование** позволяет создавать специализированные классы на основе базовых. Это позволяет избегать написания повторного кода.

Например, у вас может быть классы Студент и Преподаватель, оба являются людьми. Можно выделить общие атрибуты и методы, присущие людям, и вынести его в отдельный класс, это позволит избежать дублирование кода.

|  |
| --- |
| class Student:  def \_\_init\_\_(self, name, surname, birth\_date, school, class\_room):  self.name = name  self.surname = surname  self.birth\_date = birth\_date  self.school = school  self.\_class\_room = {'class\_num': int(class\_room.split()[0]),  'class\_char': class\_room.split()[1]}  @property  def class\_room(self):  return "{} {}".format(self.\_class\_room['class\_num'], self.\_class\_room['class\_char'])  def next\_class(self):  self.\_class\_room['class\_num'] += 1  def get\_full\_name(self):  return self.name + ' ' + self.surname  def set\_name(self, new\_name):  self.name = new\_name  class Teacher:  def \_\_init\_\_(self, name, surname, birth\_date, school, teach\_classes):  self.name = name  self.surname = surname  self.birth\_date = birth\_date  self.school = school  self.teach\_classes = list(map(self.convert\_class, teach\_classes))  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]}  def get\_full\_name(self):  return self.name + ' ' + self.surname  def set\_name(self, new\_name):  self.name = new\_name  # Эти Классы описывают два различных объекта.  # Но часть информации у них общая(атрибуты, методы)  # Общую информацию выносим в Класс-предок(родитель)  class People:  def \_\_init\_\_(self, name, surname, birth\_date, school):  self.name = name  self.surname = surname  self.birth\_date = birth\_date  self.school = school  def get\_full\_name(self):  return self.name + ' ' + self.surname  def set\_name(self, new\_name):  self.name = new\_name  # Сами классы наследуем  class Student(People):  def \_\_init\_\_(self, name, surname, birth\_date, school, class\_room):  # Явно вызываем конструктор родительского класса  People.\_\_init\_\_(self, name, surname, birth\_date, school)  # Добавляем уникальные атрибуты  self.\_class\_room = {'class\_num': int(class\_room.split()[0]),  'class\_char': class\_room.split()[1]}  # И уникальные методы  @property  def class\_room(self):  return "{} {}".format(self.\_class\_room['class\_num'], self.\_class\_room['class\_char'])  def next\_class(self):  self.\_class\_room['class\_num'] += 1  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]} |

Таким образом, практически любая серьезная программа, с использованием ООП подхода, представляет собой большое дерево классов.

## Полиморфизм

**Полиморфизм** - заключается в том, что в разных объектах одна и та же операция может выполнять различные функции. Слово «полиморфизм» имеет греческую природу и означает «имеющий многие формы». Простым примером полиморфизма может служить функция count(), выполняющая одинаковое действие для различных типов объектов: 'abc'.count('a') и [1, 2, 'a'].count('a'). Оператор плюс полиморфичен при сложении чисел и при сложении строк.  
Полиморфизм позволяет нам работать с различными типами объектов так, что нам не нужно задумываться о том, к какому типу они принадлежат.

|  |
| --- |
| >>> 2 + 4 : 6 >>> '2' + '4' : '24' >>> (2, 4)+(2, 4) :(2, 4, 2, 4) |

## 

Результат операции сложения (+) зависит от объектов, к которым применяется. Т.е. от реализации метода, соответствующего применяемой операции.

Удобство полиморфизма в том, что вы можете не заботиться о том, с какими именно объектами вы работаете, главное знать, что данные объекты поддерживают текущую операцию

Подробнее об этой особенности - на следующем уроке.

# Домашнее задание

Домашние задания для данного урока вы найдёте 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. [Введение в ООП](http://younglinux.info/book/export/html/36)
2. [Основные свойства ООП](http://pythonworld.ru/osnovy/inkapsulyaciya-nasledovanie-polimorfizm.html)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Учим python качественно(habr)](https://habrahabr.ru/post/150302/)
2. [Самоучитель по python](http://pythonworld.ru/samouchitel-python)
3. [Книга Лутц М. “Изучаем Python” (4-е издание)](http://www.shashkovs.ru/_prog/Lutc_M._-_Izuchaem_Python_(4-e_izdanie)-_2011.pdf).