Урок 6. ООП. Введение

Пришло время познакомиться с важнейшей парадигмой программирования — объектно-ориентированным программированием. Оно играет важную роль в Python и позволяет формировать структуру программы из обособленных компонентов. Важные понятия парадигмы — класс, конструктор, атрибут, метод, экземпляр класса. В рамках урока разбираются важнейшие свойства ООП: инкапсуляция, наследование и полиморфизм. Приведено описание механизмов перегрузки и переопределения методов.

## 

Оглавление

[Достоинства и недостатки механизма ООП](#_rgoo3nn84fh7)

[Классы, объекты, атрибуты](#_9d9clggiomnl)

[Понятие класса](#_btv4gedclu4z)

[Понятие объекта](#_1jxgbvl5etwp)

[Понятие объекта](#_cxor8yyz2a7)

[Конструкторы, методы](#_mllkgdufyna2)

[Понятие конструктора](#_4i1i7qcxyhat)

[Понятие метода](#_kdpe4l4hjsq4)

[Локальные переменные](#_ndmihbowxpx8)

[Глобальные переменные](#_954pe980z4m9)

[Модификаторы доступа](#_vsck0qgmiajb)

[Инкапсуляция](#_6q0d43jr6h45)

[Наследование](#_i22jffo3hxq6)

[Множественное наследование](#_qf6n3zxq6t8s)

[Несколько дочерних классов у одного родителя](#_ld2dhyr8hmnt)

[Несколько родителей у одного класса](#_ou0errp62zgz)

[Полиморфизм](#_5ro6kma8fprw)

[Перегрузка методов](#_io14xmjic9lb)

[Переопределение методов](#_amvl9v0wweu)

[Дополнительные материалы](#_vqapgkhnyfqp)

[Используемая литература](#_tnflastqfeho)

# 

**На этом уроке:**

1. Познакомится с преимуществами и недостатками механизма ООП.
2. Узнает, как создаются классы, что такое объекты, атрибуты и методы.
3. Познакомится с локальными и глобальными переменными, модификаторами доступа.
4. Разберёт особенности применения инкапсуляции, наследования и полиморфизма в Python.
5. Научится реализовывать множественное наследование, перегрузку и переопределение методов.

# Достоинства и недостатки механизма ООП

Достоинства:

1. Возможность повторного использования кода. Классы — это шаблоны, описывающие различные объекты (их свойства) и операции, выполняемые со свойствами (атрибутами) этих объектов. Эти шаблоны можно использовать повторно, в других файлах-модулях.
2. Повышение читаемости и гибкости кода. Классы и их код можно хранить в отдельных файлах-модулях и импортировать в другие модули. Модульный принцип организации программ ускоряет изучение кода программы и его модернизации.
3. Ускорение поиска ошибок и их исправления. Опять же, модульность программы предусматривает её разбиение на блоки-классы для решения определённой задачи. Соответственно, для поиска ошибок не нужно просматривать весь код. Необходимо искать ошибку в конкретном классе.
4. Повышение безопасности проекта. Благодаря такому важному свойству ООП, как инкапсуляция, разрабатываемая программа получает дополнительный уровень безопасности.

Недостатки:

1. Для реализации взаимосвязи классов необходимо хорошо разбираться в особенностях предметной области, а также чётко представлять структуру создаваемого приложения.
2. Сложность в разбиении проекта на классы. Новичкам может быть тяжело определить для проекта классы-шаблоны.
3. Сложность в модификации проекта. С добавлением в проект новой функциональности придётся вносить всё больше изменений в структуру классов.

# Классы, объекты, атрибуты

## Понятие класса

Класс в ООП — чертёж объекта. Если проводить аналогию с объектами реального мира, то, например, автомобиль — это объект, а чертёж, описывающий структуру автомобиля, его параметры и функции, — класс. Таким образом, понятие «Машина» будет соответствовать классу. Объектами этого класса будут марки автомобилей с различными характеристиками (атрибутами) и функциональными возможностями (методами), например, Audi, Lexus, Mercedes.

Для определения класса применяется ключевое слово **class**. За ним следует имя класса. Имя класса, в соответствии со стандартом PEP-8, должно начинаться с большой буквы. Далее с новой строки начинается тело класса с отступом в четыре пробельных символа.

Пример:

|  |
| --- |
| class Auto:  *# атрибуты класса*  auto\_name = "Lexus"  auto\_model = "RX 350L"  auto\_year = 2019   *# методы класса*  def on\_auto\_start(self):  print(f"Заводим автомобиль")   def on\_auto\_stop(self):  print("Останавливаем работу двигателя") |

В представленном примере создаётся класс **Auto** с атрибутами **auto\_name**, **auto\_model**, **auto\_year** и методами **on\_auto\_start()** и **on\_auto\_stop()**.

В приведённом выше примере используется служебное слово **self**, которое, в соответствии с соглашением в Python, определяет ссылку на объект (экземпляр) класса. Переменная **self** связывается с объектом класса, к которому применяются методы класса. Через переменную **self** можно получить доступ к атрибутам объекта. Когда методы класса применяются к новому объекту класса, то **self** связывается с новым объектом. Через эту переменную осуществляется доступ к атрибутам нового объекта.

## Понятие объекта

Ранее мы разобрались, что класс — чертёж, на основе которого создаётся некоторый объект. Для создания объекта (экземпляра класса) необходимо в отдельной строке указать имя класса с открывающей и закрывающей круглыми скобками. Эту инструкцию можно связать с некоторой переменной, которая будет содержать ссылку на созданный объект.

Создадим экземпляр для класса, описанного выше.

Пример:

|  |
| --- |
| a = Auto() print(a) print(type(a)) print(a.auto\_name) a.on\_auto\_start() a.on\_auto\_stop() |

Результат:

|  |
| --- |
| <\_\_main\_\_.Auto object at 0x0000001381FD8B38> <class '\_\_main\_\_.Auto'> Lexus Заводим автомобиль  Останавливаем работу двигателя |

В первой строке примера создаётся экземпляр класса **Auto**, ссылка на который связывается с переменной **a**. Содержимое этой переменной выводится во второй строке. В третьей строке проверяется тип переменной **a** — это класс **Auto**. В четвёртой строке осуществляется получение доступа к одному из атрибутов класса, а в пятой и шестой — запуск методов класса.

## Понятие атрибута

Согласно методологии ООП, выделяют атрибуты классов и экземпляров. Атрибуты класса доступны из всех экземпляров класса. Атрибуты экземпляров относятся только к объектам класса. Атрибуты класса объявляются вне любого метода, а атрибуты экземпляра — внутри любого метода. Разберёмся на примере:

Пример:

|  |
| --- |
| class Auto:    *# атрибуты класса*  auto\_count = 0    *# методы класса*  def on\_auto\_start(self, auto\_name, auto\_model, auto\_year):  print("Автомобиль заведен")  self.auto\_name = auto\_name  self.auto\_model = auto\_model  self.auto\_year = auto\_year  Auto.auto\_count += 1 |

В приведённом примере создаётся класс **Auto**, содержащий один атрибут класса **auto\_count** и три атрибута экземпляра класса: **auto\_name**, **auto\_model** и **auto\_year**. В классе реализован один метод **on\_auto\_start()** с указанными атрибутами экземпляра. Их значения передаются в виде параметров методу **on\_auto\_start()**. Внутри этого метода значение атрибута **auto\_count** класса увеличивается на единицу.

Важно отметить, что внутри методов атрибуты экземпляра идентифицируются ключевым словом **self** перед именем атрибута. При этом атрибуты класса идентифицируются названием класса перед именем атрибута.

Пример:

|  |
| --- |
| a = Auto() a.on\_auto\_start("Lexus", "RX 350L", 2019) print(a.auto\_name) print(a.auto\_count) |

Результат:

|  |
| --- |
| Автомобиль заведён Lexus 1 |

В этом примере выводятся значения атрибута экземпляра класса (**auto\_name**) и атрибута класса (**auto\_count**).

Теперь, если создать ещё один экземпляр класса **Auto** и вызвать метод **on\_auto\_start()**, результат будет следующим:

|  |
| --- |
| a\_2 = Auto() a\_2.on\_auto\_start("Mazda", "CX 9", 2018) print(a\_2.auto\_name) print(a\_2.auto\_count) |

Результат:

|  |
| --- |
| Автомобиль заведен Mazda 2 |

Теперь значение атрибута **auto\_count** равняется двум, из-за того, что он — атрибут класса и распространяется на все экземпляры. Значение атрибута **auto\_count** в экземпляре **a** увеличилось до 1, а его значение в экземпляре **a\_2** достигло двух.

# Конструкторы, методы

## Понятие конструктора

Конструктором в ООП называется специальный метод, вызываемый при создании экземпляра класса. Этот метод определяется с помощью конструкции **\_\_init\_\_**.

Пример:

|  |
| --- |
| class Auto:  *# атрибуты класса*  auto\_count = 0   *# методы класса*  def \_\_init\_\_(self):  Auto.auto\_count += 1  print(Auto.auto\_count) |

В примере создаётся класс **Auto** с одним атрибутом **auto\_count** уровня класса. В классе реализован конструктор, увеличивающий значение **auto\_count** на единицу и выводящий на экран итоговое значение.

Теперь при создании экземпляра класса **Auto** вызывается конструктор, значение **auto\_count** увеличивается и отображается на экране. Создадим несколько экземпляров класса:

Пример:

|  |
| --- |
| a\_1 = Auto() a\_2 = Auto() a\_3 = Auto() |

Результат:

|  |
| --- |
| 1 2 3 |

В результат запуска выводятся значения 1, 2, 3, так как для каждого экземпляра значение атрибута **auto\_count** возрастает и выводится на экран. На практике конструкторы используются для инициализации значений атрибутов. Это важно при создании объекта класса.

## Понятие метода

Ранее мы уже познакомились с методами в ООП, то есть, функциями, получающими в качестве обязательного параметра ссылку на объект и выполняющими определённые действия с атрибутами объекта. Мы уже создали методы **on\_auto\_start()** и **on\_auto\_stop()** для класса **Auto**. Вспомним ещё раз, как создаётся метод.

Пример:

|  |
| --- |
| class Auto:    def get\_class\_info(self):  print("Детальная информация о классе")  a = Auto() a.get\_class\_info() |

Результат:

|  |
| --- |
| Детальная информация о классе |

# 

# Локальные переменные

Понятие области видимости переменных используется и в методологии ООП. Локальная переменная в классе доступна только в рамках части кода, где она определена. Например, если определить переменную в пределах метода, не выйдет получить к ней доступ из других частей программы.

Пример:

|  |
| --- |
| class Auto:   def on\_start(self):  info = "Автомобиль заведен"  return info |

В представленном примере создаётся локальная переменная **info** в рамках метода **on\_start()** класса **Auto**. Проверим работу кода, создав экземпляр класса **Auto,** и попытаемся получить доступ к локальной переменной **info**.

Пример:

|  |
| --- |
| a = Auto() print(a.info) |

Результат:

|  |
| --- |
| AttributeError: 'Auto' object has no attribute 'info' |

Ошибка возникает из-за отсутствия возможности получения доступа к локальной переменной вне блока, в котором переменная определена.

# Глобальные переменные

Глобальные переменные, в отличие от локальных, определяются вне различных блоков кода. Доступ к ним возможен из любых точек программы (класса).

Пример:

|  |
| --- |
| class Auto:  info\_1 = "Автомобиль заведён"   def on\_start(self):  info\_2 = "Автомобиль заведён"  return info\_2  a = Auto() print(a.info\_1) |

Результат:

|  |
| --- |
| Автомобиль заведён |

В примере создаётся глобальная переменная **info\_1,** и на экран выводится её значение. При этом ошибка не возникает.

# Модификаторы доступа

Механизмы использования модификаторов позволяют изменять области видимости переменных. В Python ООП доступны три вида модификаторов:

* Public (публичный).
* Protected (защищённый).
* Private (приватный).

Для переменных с модификатором публичного доступа есть возможность изменения значений за пределами класса. Для публичных переменных префиксы (подчеркивания) не применяются.

Защищённая переменная создаётся с помощью добавления одного знака подчеркивания перед именем переменной. При использовании защищённых переменных их значения могут меняться только в пределах одного и того же пакета.

Приватная переменная идентифицируется с помощью двойного подчёркивания перед именем переменной. Значения приватных переменных могут изменяться только в пределах класса.

Пример:

|  |
| --- |
| class Auto:  def \_\_init\_\_(self):  print("Автомобиль заведен")  self.auto\_name = "Mazda"  self.\_auto\_year = 2019  self.\_\_auto\_model = "CX9" |

В примере создаётся класс **Auto** с конструктором и тремя переменными: **auto\_name**, **auto\_model**, **auto\_year**. Переменная **auto\_name** — публичная, а переменные **auto\_year** и **auto\_model** — защищённая и приватная соответственно.

Создадим экземпляр класса **Auto** и проверим доступность переменной **auto\_name**.

Пример:

|  |
| --- |
| a = Auto() print(a.auto\_name) |

Результат:

|  |
| --- |
| Mazda |

Переменная **auto\_name** обладает публичным модификатором. Доступ к ней возможен не из класса. Мы это увидели выше.

Теперь попробуем обратиться к значению переменной **auto\_model**.

Пример:

|  |
| --- |
| print(a.auto\_model) |

Результат:

|  |
| --- |
| AttributeError: 'Auto' object has no attribute 'auto\_model' |

После запуска примера мы получили сообщение об ошибке.

# Инкапсуляция

Пришло время познакомиться с ключевыми принципами ООП: инкапсуляцией, наследованием и полиморфизмом.

Начнём с инкапсуляции, то есть с механизма сокрытия данных. В Python инкапсуляция реализуется только на уровне соглашения, которое определяет общедоступные и внутренние характеристики. Одиночное подчёркивание в начале имени атрибута или метода свидетельствует о том, что атрибут или методы не предназначены для использования вне класса. Они доступны по этому имени.

Пример:

|  |
| --- |
| class MyClass:  \_attr = "значение"  def \_method(self):  print("Это защищенный метод!")  mc = MyClass() mc.\_method() print(mc.\_attr) |

Результат:

|  |
| --- |
| Это защищённый метод! значение |

Использование двойного подчёркивания перед именем атрибута и метода делает их недоступными по этому имени.

Пример:

|  |
| --- |
| class MyClass:  \_\_attr = "значение"  def \_\_method(self):  print("Это приватный метод!")  mc = MyClass() mc.\_\_method() print(mc.\_\_attr) |

Результат:

|  |
| --- |
| AttributeError: 'MyClass' object has no attribute '\_\_method' |

Но и эта мера не обеспечивает абсолютную защиту. Обратиться к атрибуту или методу по-прежнему можно, используя следующий подход: \_ИмяКласса\_\_ИмяАтрибута.

Пример:

|  |
| --- |
| class MyClass:  \_\_attr = "значение"  def \_\_method(self):  print("Это защищённый метод!")  mc = MyClass() mc.\_MyClass\_\_method() print(mc.\_MyClass\_\_attr) |

Результат:

|  |
| --- |
| Это защищенный метод! значение |

# Наследование

Сущность этого понятия соответствует его названию. Речь идёт о наследовании некоторым объектом характеристик другого объекта-родителя. Объект называется дочерним и обладает не только характеристиками родителя, но и собственными свойствами. Благодаря наследованию можно избежать дублирования кода.

Суть принципа наследования заключается в том, что класс может перенимать (наследовать) параметры другого класса. Класс, наследующий характеристики другого класса, называется дочерним, а класс, предоставляющий свои характеристики, — родительским.

Пример:

|  |
| --- |
| *# Класс Transport* class Transport:  def transport\_method(self):  print("Это родительский метод из класса Transport")   *# Класс Auto, наследующий Transport* class Auto(Transport):  def auto\_method(self):  print("Это метод из дочернего класса") |

В представленном примере создаются два класса: **Transport** (родитель), **Auto** (наследник). Для реализации наследования нужно указать имя класса-родителя внутри скобок, следующих за именем класса-наследника. В классе **Transport** реализован метод **transport\_method()**, а в дочернем классе есть метод **auto\_method()**. Класс **Auto** наследует характеристики класса **Transport,** то естьвсе его атрибуты и методы.

Проверим работу механизма наследования:

|  |
| --- |
| a = Auto() a.transport\_method() *# Вызываем метод родительского класса* |

Результат:

|  |
| --- |
| Это родительский метод из класса Transport |

В примере создаётся экземпляр класса **Auto**. Для экземпляра класса вызывается метод **transport\_method()**. Важно, что в классе **Auto** отсутствует метод с названием **transport\_method()**. Так как класс **Auto** унаследовал характеристики класса **Transport**, то экземпляр класса **Auto** работает с методом **transport\_method()** класса **Transport**.

# Множественное наследование

Механизм наследования может быть реализован с использованием нескольких родителей у одного класса. И наоборот, один класс-родитель будет передавать свои характеристики нескольким дочерним классам.

## Несколько дочерних классов у одного родителя

Пример:

|  |
| --- |
| *# класс Transport* class Transport:  def transport\_method(self):  print("Родительский метод класса Transport")   *# класс Auto, наследующий Transport* class Auto(Transport):  def auto\_method(self):  print("Дочерний метод класса Auto")   *# класс Bus, наследующий Transport* class Bus(Transport):  def bus\_method(self):  print("Дочерний метод класса Bus") |

В этом примере у нас есть класс-родитель **Transport**, наследуемый дочерними классами **Auto** и **Bus**. В обоих дочерних классах возможен доступ к методу **transport\_method()** класса-родителя. Для запуска скрипта создадим экземпляры класса.

Пример:

|  |
| --- |
| a = Auto() a.transport\_method() b = Bus() b.transport\_method() |

Результат:

|  |
| --- |
| Родительский метод класса Transport  Родительский метод класса Transport |

Рассмотрим ещё один пример, в котором класс-родитель **Shape** определяет атрибуты. Эти атрибуты могут быть характерны для всех классов-наследников. Например, цвет фигуры, ширина и высота, основание и высота.

Здесь в конструкторах классов-наследников инициализируются параметры. Часть их — собственные атрибуты классов-наследников, а некоторые наследуются от родителей. Чтобы работать с унаследованными атрибутами, нужно их перечислить, например, **super().\_\_init\_\_(color, param\_1, param\_2)**. Тем самым мы показываем, что хотим иметь возможность работы с атрибутами класса-родителя. Если атрибуты не перечислить, то при попытке обращения к ним через экземпляр класса-наследника возникнет ошибка.

Пример:

|  |
| --- |
| class Shape:  def \_\_init\_\_(self, color, param\_1, param\_2):  self.color = color  self.param\_1 = param\_1  self.param\_2 = param\_2   def square(self):  return self.param\_1 \* self.param\_2   class Rectangle(Shape):  def \_\_init\_\_(self, color, param\_1, param\_2, rectangle\_p):  super().\_\_init\_\_(color, param\_1, param\_2)  self.rectangle\_p = rectangle\_p   def get\_r\_p(self):  return self.rectangle\_p   class Triangle(Shape):  def \_\_init\_\_(self, color, param\_1, param\_2, triangle\_p):  super().\_\_init\_\_(color, param\_1, param\_2)  self.triangle\_p = triangle\_p   def get\_t\_p(self):  return self.triangle\_p  r = Rectangle("white", 10, 20, True) print(r.color) print(r.square()) print(r.get\_r\_p()) t = Triangle("red", 30, 40, False) print(t.color) print(t.square()) print(t.get\_t\_p()) |

Результат:

|  |
| --- |
| white 200 True red 1200 False |

## Несколько родителей у одного класса

Пример:

|  |
| --- |
| class Player:  def player\_method(self):  print("Родительский метод класса Player")   class Navigator:  def navigator\_method(self):  print("Родительский метод класса Navigator")   class MobilePhone(Player, Navigator):  def mobile\_phone\_method(self):  print("Дочерний метод класса MobilePhone") |

В этом примере создаются классы: **Player**, **Navigator**, **MobilePhone**. Причём классы **Player** и **Navigator** — родительские для класса **MobilePhone**. Поэтому класс **MobilePhone** имеет доступ к методам классов **Player** и **Navigator**. Проверим это.

Пример:

|  |
| --- |
| m\_p = MobilePhone() m\_p.player\_method() m\_p.navigator\_method() |

Результат:

|  |
| --- |
| Родительский метод класса Player Родительский метод класса Navigator |

Возможна ситуация, когда у классов-родителей совпадают имена атрибутов и методов. В этом случае обращение к такому атрибуту или методу через «наследник» будет адресовано к атрибуту или методу того класса-родителя, который значится первым.

Пример:

|  |
| --- |
| class Shape:  def \_\_init\_\_(self, param\_1, param\_2):  self.param\_1 = param\_1  self.param\_2 = param\_2   def get\_params(self):  return f"Параметры Shape: {self.param\_1}, {self.param\_2}"  class Material:  def \_\_init\_\_(self, param\_1, param\_2):  self.param\_1 = param\_1  self.param\_2 = param\_2   def get\_params(self):  return f"Параметры Material: {self.param\_1}, {self.param\_2}"   class Triangle(Shape, Material):  def \_\_init\_\_(self, param\_1, param\_2):  super().\_\_init\_\_(param\_1, param\_2)  pass  t = Triangle(10, 20) print(t.get\_params()) |

Результат:

|  |
| --- |
| Параметры Shape: 10, 20 |

# Полиморфизм

Дословный перевод этого понятия — «имеющий многие формы». В методологии ООП — это способность объекта иметь различную функциональность. В программировании полиморфизм проявляется в перегрузке или переопределении методов классов.

## Перегрузка методов

Реализуется в возможности метода отражать разную логику выполнения в зависимости от количества и типа передаваемых параметров.

Пример:

|  |
| --- |
| *# класс Auto* class Auto:  def auto\_start(self, param\_1, param\_2=None):  if param\_2 is not None:  print(param\_1 + param\_2)  else:  print(param\_1) |

В этом примере возможны несколько вариантов логики метода **auto\_start()**. Первый вариант — при передаче в метод одного параметра. Второй — при передаче двух параметров. В первом случае будет выведено значение переданного параметра, во втором — сумма параметров.

Пример:

|  |
| --- |
| a = Auto() a.auto\_start(50) a = Auto() a.auto\_start(10, 20) |

Результат:

|  |
| --- |
| 50 30 |

## Переопределение методов

Переопределение методов в полиморфизме выражается в наличии метода с одинаковым названием для родительского и дочернего классов. При этом логика методов различается, но названия идентичны.

Пример:

|  |
| --- |
| *# класс Transport* class Transport:  def show\_info(self):  print("Родительский метод класса Transport")   *# класс Auto, наследующий Transport* class Auto(Transport):  def show\_info(self):  print("Родительский метод класса Auto")   *# класс Bus, наследующий Transport* class Bus(Transport):  def show\_info(self):  print("Родительский метод класса Bus") |

В примере классы **Auto** и **Bus** наследуют характеристики класса **Transport**. В этом классе реализуется метод **show\_info()**, переопределенный классом-потомком. Теперь, если вызвать метод **show\_info()**, результат будет зависеть от объекта, через который осуществляется вызов метода.

Пример:

|  |
| --- |
| t = Transport() t.show\_info()  a = Auto() a.show\_info()  b = Bus() b.show\_info() |

Результат:

|  |
| --- |
| Родительский метод класса Transport Родительский метод класса Auto Родительский метод класса Bus |

В этом примере методы **show\_info()** вызываются с помощью производных классов одного общего базового класса. Но дочерние классы переопределяются через метод класса-родителя, методы обладают разной функциональностью.

# Дополнительные материалы

1. [Объектно-ориентированное Программирование в Python](https://python-scripts.com/object-oriented-programming-in-python).
2. [Объектно-ориентированное программирование. Классы и объекты](https://pythonworld.ru/osnovy/obektno-orientirovannoe-programmirovanie-obshhee-predstavlenie.html).
3. [Обучение ООП](https://younglinux.info/oopython.php).
4. [Python — объектно-ориентированное программирование (ООП)](https://xn--80ahcjeib4ac4d.xn--p1ai/information/python_object_oriented_programming_oop/).

# Используемая литература

1. [Язык программирования Python 3 для начинающих и чайников](https://pythonworld.ru/).
2. [Программирование в Python](https://python-scripts.com/).
3. [Учим Python качественно (habr)](https://habrahabr.ru/post/150302/).
4. [Самоучитель по Python](http://pythonworld.ru/samouchitel-python).
5. [Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание)](http://www.proklondike.com/books/python/lutz_python_2011.html).