* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок Полиморфизм](https://lyceum.yandex.ru/courses/123/groups/568/lessons/663)

**Введение в ООП. Полиморфизм**

**План урока**

1

[Полиморфизм](https://lyceum.yandex.ru/courses/123/groups/568/lessons/663/materials/1650#1)

2

[Проверка типа объекта](https://lyceum.yandex.ru/courses/123/groups/568/lessons/663/materials/1650#2)

**Аннотация**

*В этом уроке мы рассмотрим возможности предоставление одинаковых средств взаимодействия с объектами разной природы.*

**1. Полиморфизм**

После предыдущего занятия мы уже немного разбираемся в объектно-ориентированном программировании: освоили определение классов и методов, добавление атрибутов в объекты. Понятно, что классы, объекты, методы, атрибуты достаточно удобны и красивы, но в чём их преимущество перед функциями? Ведь некоторые объекты можно было бы передавать в функции и выполнять над ними те же действия, что и с помощью методов. Зачем вводить дополнительный синтаксис и правила? Основное преимущество в том, что объектно-ориентированный подход позволяет писать код, который будет работать с экземплярами различных классов. Иногда код может даже работать с классами, которые ещё не созданы.

**Полиморфизм**

Свойство кода работать с разными типами данных называют **полиморфизмом**.

Мы уже неоднократно пользовались этим свойством многих функций и операторов, не задумываясь о нём. Например, оператор **+** является полиморфным:

**print**(1 + 2) *# 3*

**print**(1.5 + 0.2) *# 1.7*

**print**("abc" + "def") *# abcdef*

Внутренняя реализация оператора **+** существенно отличается для целых чисел, чисел с плавающей точкой и строк. То есть, на самом деле, это три разные операции — интерпретатор Python выбирает одну из них при выполнении в зависимости от операндов. Впрочем, в нашем случае выбор очевиден, потому что операнды — это просто константы.

Усложним задачу:

**def** f(x, y):

**return** x + y

**print**(f(1, 2)) *# 3*

**print**(f(1.5, 0.2)) *# 1.7*

**print**(f("abc", "def")) *# abcdef*

Перехитрить интерпретатор не удалось, ведь Python — язык с динамической типизацией. В таких языках любое значение несёт в себе информацию о типе — она и помогла интерпретатору выбрать правильную реализацию операции **+** (а заодно и правильное строковое представление для функции print). Но мы знаем, что тип данных в Python — это класс объекта, и именно эта информация о классе объекта используется при выборе операции. На следующем занятии мы вернёмся к оператору **+** и рассмотрим, как реализовать его для наших собственных классов.

Давайте теперь вспомним про метод **\_\_init\_\_**. Он выполняется при создании каждого нового экземпляра класса и инициализирует свойства нового экземпляра. Первый аргумент, **self**, он получает от интерпретатора, остальные передаются классу в круглых скобках при создании экземпляра.

**class** Book:

**def** \_\_init\_\_(self, name, author):

self.name = name

self.author = author

**def** get\_name(self):

**return** self.name

**def** get\_author(self):

**return** self.author

book = Book('Война и мир', 'Толстой Л. Н.')

**print**('{}, {}'.format(book.get\_name(), book.get\_author()))

*# Война и мир, Толстой Л. Н.*

При создании экземпляра кодом book = Book('Война и мир', 'Толстой Л. Н.') будет вызван метод **\_\_init\_\_**, который создаст атрибуты **name** и **author**. Читать свойства можно из объекта напрямую (например, book.name) или использовать определённые для этого методы. Второй способ лучше, так как позволяет оградить программистов — пользователей класса от возможных изменений в реализации класса.

Теперь мы готовы определить свои собственные классы, с помощью которых будем разбираться с полиморфизмом.

Посмотрим на реализацию классов «Круг» и «Квадрат» для подсчёта площади и периметра:

**from** math **import** pi

**class** Circle:

**def** \_\_init\_\_(self, radius):

self.radius = radius

**def** area(self):

**return** pi \* self.radius \*\* 2

**def** perimeter(self):

**return** 2 \* pi \* self.radius

**class** Square:

**def** \_\_init\_\_(self, side):

self.side = side

**def** area(self):

**return** self.side \* self.side

**def** perimeter(self):

**return** 4 \* self.side

Мы определили классы **Сircle** и **Square**, экземпляры которых могут считать площадь и периметр окружностей и квадратов. Важно, что у обоих классов одинаковый интерфейс: методы для расчёта площади называются **area**, а для расчёта периметра — **perimeter**. Кроме того, у этих методов одинаковое количество параметров (в данном случае, только **self**), и они оба возвращают в результате работы число, хотя оно и может быть разного типа (целое и вещественное).

Теперь мы можем определить полиморфную функцию **print\_shape\_info**, которая будет печатать данные о фигуре:

**def** print\_shape\_info(shape):

**print**("Area = {}, perimeter = {}.".format(shape.area(),  
 shape.perimeter()))

square = Square(10)

*# Area = 100, perimeter = 40.*

print\_shape\_info(square)

circle = Circle(10)

*# Area = 314.1592653589793, perimeter = 62.83185307179586.*

print\_shape\_info(circle)

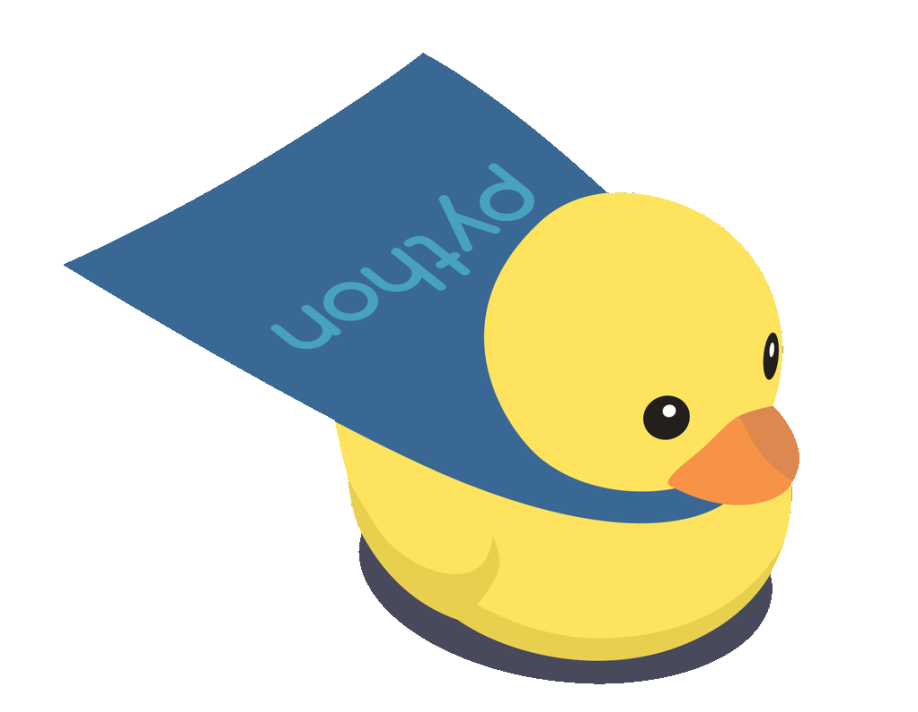
Если аргумент функции **print\_shape\_info** — экземпляр класса **Square**, то выполняются методы, определённые в этом классе, если экземпляр **Circle**, то выполняются методы **Circle**.

**print**(dir(square)) *# Свойства и спецметоды экземпляра.*

**print**(dir(Square)) *# Свойства и спецметоды класса.*

**Утиная типизация**

Данный код использует тот факт, что в Python принята так называемая **«утиная типизация»**. Название происходит от шутливого выражения «если нечто выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка и есть».



В программах на Python это означает, что если какой-то объект поддерживает все требуемые от него операции, то с ним и будут работать с помощью этих операций, не заботясь о том, какого он на самом деле типа. Так и наша функция **print\_shape\_info** будет выводить информацию о любом объекте, у которого есть методы **area** и **perimeter** (и у которых в списке параметров также будет указан один параметр **self**).

В языках без утиной типизации нам бы пришлось добавлять в программу интерфейс как отдельную сущность на уровне описания на языке программирования, а также указывать, что наши классы относятся к этому интерфейсу. В программах на Python этого делать не нужно, однако интерфейсы всё равно существуют.

**Важно**

Чтобы полиморфизм работал, за ними надо следить как **на уровне синтаксиса** (одинаковые имена методов и количество параметров), так и **на уровне смысла** (методы с одинаковыми именами делают похожие операции, параметры методов имеют тот же смысл).

Давайте определим ещё один класс с таким же интерфейсом, как у **Circle** и **Square** — например **Rectangle** (прямоугольник). Если мы всё сделаем правильно, функция **print\_shape\_info** сможет работать с его экземплярами:

**class** Rectangle:

**def** \_\_init\_\_(self, width, height):

self.width = width

self.height = height

**def** area(self):

**return** self.width \* self.height

**def** perimeter(self):

**return** 2 \* (self.width + self.height)

rect = Rectangle(10, 15)

print\_shape\_info(rect) *# Area = 150, perimeter = 50.*

Ещё раз обратите внимание: утиная типизация позволяет заранее написать функцию, которая будет работать со всеми экземплярами любых классов — даже ещё не существующих. Важно лишь, чтобы эти классы поддерживали необходимый функции интерфейс.

**Важно**

И небольшое замечание об инкапсуляции. Дело в том, что с самого начала обычно есть не два класса, как в нашем примере, а один. Пусть это будет **Square**. Если не инкапсулировать внутри него свойство **side** и не определить заранее интерфейс для расчёта площади и периметра, никакого полиморфизма не получится. Важно помнить о том, что инкапсуляция определяет понятие интерфейса класса и создаёт базу для полиморфизма.

**2. Проверка типа объекта**

При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. И с помощью встроенной функции **isinstance()** мы можем проверить тип объекта.

**isinstance**

Эта функция принимает два параметра: **isinstance(object, type)**

Первый параметр представляет объект, а второй — тип, на принадлежность к которому выполняется проверка. Если объект представляет указанный тип, то функция возвращает **True**.

**for** person **in** people:

**if** isinstance(person, Student):

**print**(person.university)

**elif** isinstance(person, Employee):

**print**(person.company)

**else**:

**print**(person.name)

**print**()

[Справка](https://yandex.ru/support/lyceum-students)

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты