

# **Лекция 7**

## **Объектно-ориентированное программирование**

### **Введение**

# Объектно-ориентированное программирование

- *Объект* это конструкция, сочетающая в себе данные и процедуры, применимые к этим данным.
- *Объектно-ориентированное программирование* это стиль программирования ориентированный на манипулирование объектами
  - Процедурное программирование рассуждает в терминах данных и процедур их обработки
  - Объектно-ориентированное программирование рассуждает в терминах объектов, их свойств и взаимосвязей
- Создание иерархий объектов ведет к явному выделению уровней абстракции, более естественных чем в процедурном программировании
  - В процедурном программировании выделяются *процедуры* которые делятся на более мелкие процедуры, и т.д.
  - В объектно-ориентированном программировании модель реального мира представлена *объектами* которые делятся на более мелкие объекты, и т.д. Такой подход позволяет решать очень большие задачи.

# Свойства объектно-ориентированного программирования

- *Инкапсуляция и абстракция*
  - соединение данных и процедур их обработки в единую конструкцию
  - выделение *интерфейса* для взаимодействия с внешним миром
- *Наследование*
  - создание новых типов данных на основе уже существующих с приобретением свойств "родительских" типов в дополнение к своим собственным
- *Полиморфизм*
  - обработка объектов различных типов единым образом

# Инкапсуляция и абстракция

- Разделение атрибутов объекта на внутренние и внешние
  - Внутренние атрибуты, реализуют устройство объекта
  - Внешние атрибуты, предоставляют средства манипуляции объектом, они образуют *интерфейс* объекта
- Интерфейс объекта является его *абстракцией*, для использования объекта достаточно знать только его интерфейс
- Интерфейс объекта должен быть минимален и, по возможности, содержать только процедуры (методы объекта)
- Пример: интерфейс системы охлаждения, методы класса Cooler

```
power(state) # Параметр state принимает значения 'on' и 'off'  
get_temperature()  
set_alarm_handler(handler)
```

- Интерфейс системы охлаждения оформляется как модуль, содержащий класс Cooler, который может быть импортирован в программу управления экспериментальной установкой

# Свойства инкапсуляции и абстракции

- Внутренняя реализация объекта может меняться при сохранении интерфейса
  - система охлаждения в эксперименте может быть заменен другой без изменений в управляющей системе если интерфейс сопровождающей ее программы остался неизменным
- Абстракция способствует лучшему пониманию программы и упрощает разделение труда в коллективе
  - каждый программист получает набор интерфейсов которые он должен реализовать в своих объектах и набор интерфейсов с которыми он должен взаимодействовать
- В Питоне нет абсолютной защиты внутренних атрибутов, есть лишь способ затруднить доступ к ним извне. Это
  - способствует появлению ошибок при нарушении интерфейса,
  - упрощает отладку программ.

# Наследование

- Наследование это один из способов повторного использования кода
- Наследование позволяет создать новый тип данных, обладающий свойствами всех типов данных являющихся для него родительскими
- Наследование реализует отношение "является" ("is-a") в отличие от *агрегирования*, которое реализует отношение "содержит" ("contains")
- Созданный при наследовании тип данных (потомок) позволяет:
  - добавить новые свойства к унаследованным (расширение)
  - изменить унаследованные свойства (модификация)
- Наследование создает иерархию типов данных
  - тип данных-потомок наследует свойства всех своих предков
  - типы данных становятся все более сложными по мере движения вниз по иерархии

# Наследование, пример

- Геометрическая форма, свойства:
  - цвет
  - прозрачность
  - площадь
  - длина контура
- Круг и многоугольник наследуют геометрической форме
  - цвет и прозрачность наследуются как есть
  - площадь и длина контура модифицируются
  - радиус круга и число углов - новые свойства (расширение)
- Квадрат и треугольник наследуют многоугольнику
  - цвет и прозрачность наследуются как есть
  - площадь, длина контура и число углов могут быть унаследованы или изменены с целью повышения производительности

# Агрегирование, пример

- Геометрическая форма может иметь надпись
  - Геометрическая форма *не является* надписью
  - Геометрическая форма *содержит* надпись как атрибут text
  - Геометрическая форма обладает свойствами надписи, но не непосредственно, а через свойства атрибута text
  - Все наследники геометрической формы наследуют атрибут text вместе с его свойствами
- Выбор между наследованием и агрегированием заключается в ответе на вопрос об отношении создаваемого типа данных к базовому: - "является или содержит" ?
- Пример: блок электроники в экспериментальной установке
  - блок *он что* ? он частный случай *чего* ? (наследование)
  - блок *содержит* что ? он *состоит* из чего ? (агрегирование)



# Полиморфизм

- Полиморфизм это возможность совершать одну и ту же операцию над объектами разных типов
- Полиморфизм реализуется двумя способами
  - *ad hoc полиморфизм*: для каждого типа пишется специализированный код его обработки
  - *параметрический полиморфизм*: все типы обрабатываются одним и тем же кодом, то есть обрабатывающие процедуры сами по себе являются полиморфными
- Задача реализации полиморфизма заключается в вынесении специализированного кода в самый конец цепочки вызовов функций и придании типам нужных свойств
- В Питоне типы имеющие общие свойства не обязательно должны наследовать от общего предка
- Для участия в полиморфной операции типу достаточно иметь только те свойства, которые для нее необходимы

## Полиморфизм, пример

- Функция проверки орфографии применима к геометрической форме, она находит атрибут `text` и совершает с ним необходимые действия
- Создавая класс заметок введем в него атрибут `text`. Теперь объект класса заметок также можно будет использовать с функцией проверки орфографии.
- Геометрическая форма и заметка не имеют общего предка
- Для работы функции проверки орфографии достаточно наличия в объекте атрибута `text`

*Объектная модель языков программирования без строгой типизации обеспечивает высокую гибкость и широкие возможности*

# Классы

- *Класс* это конструкция, описывающая тип данных, его свойства и внутреннюю структуру
- *Объект* это конкретная реализация класса, другими словами, каждый объект имеет определенный тип данных. В Питоне начиная с версии 3 термины *класс* и *тип данных* синонимы.
- Термины *объект* и *экземпляр класса* также синонимы. Термин *экземпляр класса* используется если необходимо явно указать на принадлежность объекта конкретному классу.
- На вершине иерархии классов находится класс *object*. Все классы явно или неявно наследуют от класса *object*.
  - Пример. Свойство "быть напечатанным" реализовано в классе *object*. Если это свойство не переопределено в дочернем классе, при передаче объекта этого класса в функцию `print` будет напечатано:

<имя\_класса object at адрес\_объекта>

# Реализация классов

- В компилируемых языках программирования класс это схема или модель по которой создается объект
  - при создании объекта ему выделяется участок памяти размеченный на части в соответствии с содержанием класса
  - если в классе есть конструктор, он инициализирует принадлежащий объекту участок памяти
- В Питоне класс это *объект (!)* класса *type*
  - при создании объекта в интерпретаторе возникает минимальная конструкция содержащая ссылку на свой класс, затем по этой ссылке вызывается функция `__init__()`, которой в качестве параметра передается создаваемый объект
- Элементарные встроенные типы данных, такие как *int* или *str*, также являются классами

# Объекты

- *Объект* это конструкция, сочетающая в себе данные и функции, применимые к этим данным.
- Терминология:
  - элемент данных это *атрибут* объекта
  - функция это *метод* объекта, также используется термин *функция-атрибут*
  - термин *атрибут* часто используют обобщенно, для обозначения как данных, так и методов объекта
- Доступ к атрибутам и методам обеспечивает точечная нотация (dot notation). Объект похож на словарь, но с иным синтаксисом:

```
v = d['key'] # словарь d, ключ 'key'
v = a.key    # объект a, атрибут key
```
- В отличие от словаря объект не является коллекцией атрибутов
  - список атрибутов возвращает встроенная функция `dir()`
- Точка это *оператор* имеющий один из самых высоких приоритетов

# Объекты, продолжение

- Объект имеет иерархическую структуру, атрибуты объекта также являются объектами
- Объект является воплощением (instance) своего *класса*, при создании он получает атрибуты, реализованные в классе
- Объект создается вызовом функции, совпадающей с именем своего класса

```
i = Decimal(10) # => i = type.__new__(Decimal) ; Decimal.__init__(i, 10)
```

- В ходе работы программы объект может приобретать дополнительные атрибуты.
  - Имя атрибута это *переменная в составе объекта*, она возникает при первом появлении в левой части инструкции присваивания:

```
i.used = True # Объект i приобретает атрибут used
```

- Внутренняя реализация объектно-ориентированных конструкций в Питоне опирается на два механизма: оператор точка и инструкцию присваивания

# Объекты и пространства имен

- Каждый объект имеет собственное пространство имен
- Собственное пространство имен дает объекту контекст исполнения неизменный между обращениями к объекту
- В большинстве случаев функцию с замыканием можно заменить объектом. Использование объектов более традиционный способ создания контекста исполнения.
- Использование оператора точка для обращения к атрибуту объекта:
  - внешнее обращение: `имя_объекта.имя_атрибута`
  - обращение внутри собственного метода объекта требует использования слова *self*: `self.имя_атрибута`
- Метод (функция-атрибут) должен иметь *self* своим первым аргументом: `is_between(self, a, b)`
- *self* это не ключевое слово, а традиционное имя

# Синтаксис определения класса

```
class Empty:  
    pass
```

```
class Sample: # Определение класса, скобки необязательны  
    def fun(self, n):  
        print('Call ' + 'fun(', n, ') of', self.__class__)
```

```
m = Sample() # Создание объекта, скобки обязательны  
m.fun(3)
```

```
class Sample2(Sample, Empty): # Наследует двум классам  
    s2attr = 10                # Атрибут класса Sample2
```

- Имена пользовательских классов принято начинать с прописной буквы
- Определение класса как и определение функции подобно инструкции присваивания: оно создает объект, имеющий класс *type*, с именем как у определяемого класса



# Атрибуты класса

- Определенные внутри класса функции (методы) и атрибуты-данные становятся атрибутами этого класса как объекта
- Класс в свою очередь является атрибутом модуля, в котором он определен
- Атрибуты-данные не имеющие префикса с точкой и созданные внутри класса, но вне его методов, называют *атрибутами класса*
  - Атрибуты класса являются общими для всех объектов, произведенных от этого класса
  - Атрибуты класса подобны глобальным переменным, но их областью видимости является не модуль, а класс
  - Строго говоря, методы класса также являются атрибутами класса общими для всех произведенных от класса объектов. Но так как методы неизменяемы, их общность не создает проблем.

## Метод `__init__(self)`

- Метод `__init__(self)` вызывается для каждого создаваемого объекта и содержит код его инициализации
- Объект создается тогда, когда он в первый раз появляется в левой части инструкции присваивания. Это правило относится и к атрибутам объекта, так как они также являются объектами.
- До начала исполнения метода `__init__` объект не содержит ни одного атрибута
- Метод `__init__` выполняет инструкции присваивания создающие атрибуты объекта

```
class Sample:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def divide(self):
        return self.a / self.b
```

```
m = Sample(6, 2) # Параметры 6 и 2 передаются в метод __init__
m.divide() # => 3.0
```

# Атрибуты объекта

- Созданные таким способом атрибуты называются *атрибутами объекта*
- В отличие от атрибутов класса общих для всех объектов, каждый объект имеет собственный набор атрибутов объекта
- Атрибуты объекта могут создаваться не только в методе `__init__` но и в других методах. Таким образом структура объектов может изменяться по ходу выполнения программы. Объекты имеющие одинаковый тип могут иметь разный набор атрибутов (!)
- Атрибуты объекта могут создаваться и во внешнем по отношению к классу коде

```
m.c = 28 # Объект m приобрел третий атрибут c
```

```
m2 = Sample(5, 6) # Объект m2 имеет только два атрибута a и b
```

- Таким образом возникают объекты с добавленными или измененными свойствами. При этом остальные объекты этого класса остаются неизменными.

## Объекты с дополненными свойствами

- Объект с добавленными или измененными свойствами может стать альтернативой созданию дочерних классов
- Использование такого "усовершенствованного" объекта может быть оправдано если:
  - такой объект нужен только один,
  - требуется много объектов, каждый из которых имеет собственную специфическую модификацию.
  - Во втором случае предпочтительнее создание дочернего класса с атрибутом, хранящим необходимую каждому объекту модификацию.

*Использование объектов с дополненными свойствами не рекомендуется*

# Вызов метода

- Объект связан с классом, элементом которого он является
- Класс может наследовать от многих классов, но сам объект является элементом только одного, конкретного класса
- При вызове метода `m.divide()` производится его поиск:
  - метод ищется среди атрибутов объекта
  - метод ищется среди атрибутов класса объекта
  - метод ищется среди атрибутов классов, которым наследует класс объекта
- Как только метод будет найден, дальнейший поиск прекращается и метод передается на исполнение
- Если найденный атрибут `divide` не является методом, произойдет ошибка `'object is not callable'`
- Если метод найден не будет, произойдет ошибка `'object has no attribute divide'`

# Алгоритм поиска метода

- При множественном наследовании в языках Питон 3 и Perl 6 в качестве алгоритма поиска используется СЗ-линеаризация
- СЗ-линеаризация это алгоритм преобразования графа наследования в последовательность
- На первом шаге алгоритма происходит обход графа методом "сначала в глубину", слева направо. Каждая пройденная вершина записывается в последовательность.
- На втором шаге алгоритма из последовательности удаляются все одинаковые вершины кроме последней
- При отсутствии множественного наследования иерархия типов данных является последовательностью, ее линеаризация не требуется
- В Питоне 3 все классы неявно наследуют от класса object. При поиске метода класс object просматривается последним.
- При чтении атрибута используется тот же алгоритм поиска

## ЯВНЫЙ ВЫЗОВ метода `__init__(self)`

- Методы `__init__` родительских классов не вызываются автоматически, как это происходит с конструкторами в C++
- Традиционной практикой является явный вызов методов `__init__` родительских классов:

```
class Parent1:  
    def __init__(self):  
        print('Parent1', end=',')
```

```
class Parent2:  
    def __init__(self):  
        print('Parent2', end=',')
```

```
class Child(Parent1, Parent2):  
    def __init__(self):  
        Parent1.__init__(self)  
        Parent2.__init__(self)  
        print('Child')
```

```
c = Child() # => Parent1,Parent2,Child
```

## Метод `__del__(self)`

- Метод `__del__(self)` вызывается непосредственно перед уничтожением объекта
- Функции `__del__` родительских классов также не вызываются автоматически, как это происходит с деструкторами в C++
- При необходимости функции `__del__` родительских классов могут быть вызваны явно

*Методы ничем, кроме синтаксиса вызова, не отличаются от обычных функций. Все, что ранее говорилось о функциях, относится и к методам.*



# Атрибуты класса и атрибуты объекта

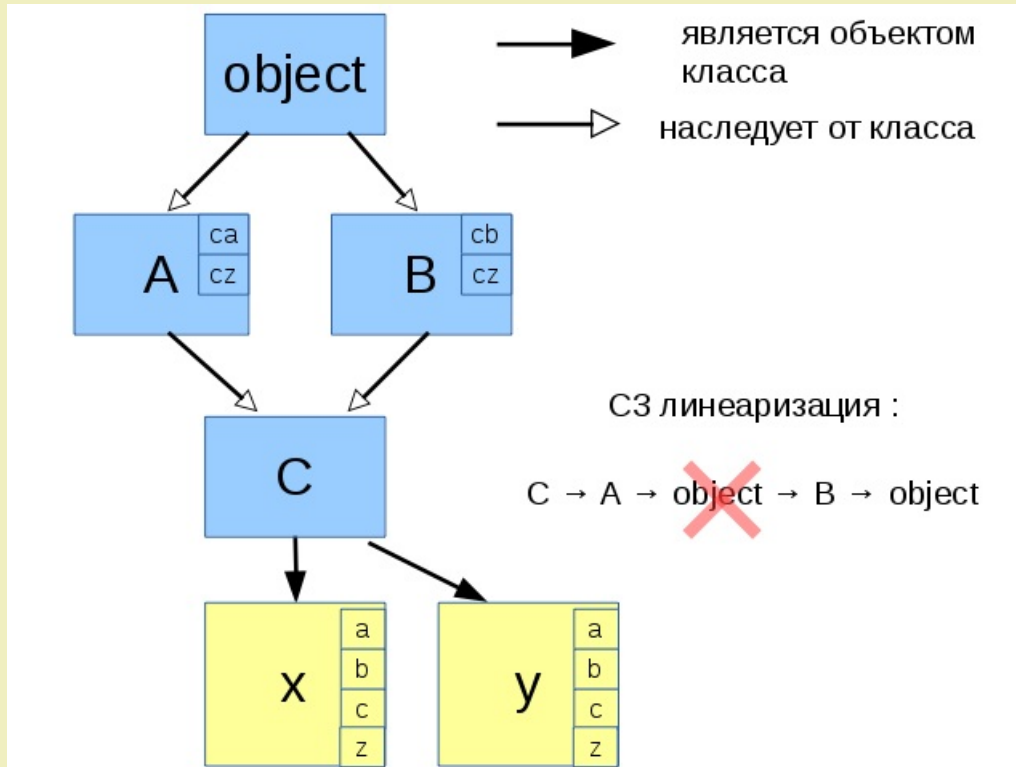
```
class A:
    ca = 'A:ca'           # атрибут класса
    cz = 'A:cz'           # атрибут класса
    def __init__(self):
        self.a = 'A:a'    # атрибут объекта
        self.z = 'A:z'    # атрибут объекта

class B:
    cb = 'B:cb'
    cz = 'B:cz'
    def __init__(self):
        self.b = 'B:b'
        self.z = 'B:z'

class C(A, B):             # При создании объекта x:
    def __init__(self):    # объект x не имеет атрибутов, self это x
        A.__init__(self)  # объект x получил атрибуты a и z
        B.__init__(self)  # объект x получил атрибут b,
                           # атрибуту z присвоено другое значение
        self.c = 'C:c'    # объект x получил атрибут c

x = C() # => C.__init__(x)
y = C() # => C.__init__(y)
```

# Объекты, классы и атрибуты



## Результаты работы примера

```
print(dir(x)) # => [ <пропущено>, 'a', 'b', 'c', 'ca', 'cb', 'cz', 'z']
print('x.a =', x.a)    # => A:a
print('x.b =', x.b)    # => B:b
print('x.c =', x.c)    # => C:c
print('x.ca =', x.ca)  # => A:ca
print('x.cb =', x.cb)  # => B:cb
print('x.cz =', x.cz)  # => A:cz
print('x.z =', x.z)    # => B:z
```

- Атрибут *cz* по правилам поиска выбирается из класса A, так как в списке родительских классов класс A левее класса B

```
class C(A, B):
```

- Атрибут *z* получает значение в методе `__init__` класса B, так как она вызывается после метода `__init__` класса A

```
A.__init__(self)
B.__init__(self)
```

# Классы, документация

- Класс это именованный объект. Если сразу после заголовка *class Name:* поместить текстовую строку, она станет строкой документации класса
- Метод, как и любая другая функция, может иметь строку документации
- Функция `help()` примененная к классу выводит его полную документацию, включающую и общую документацию класса и документацию его методов

```
class Sample:
    """Sample это простой класс используемый
       исключительно как пример
    """
    def fun(self, n):
        """Функция fun() печатает свой аргумент и свой класс
        """
        print('Call ' + 'fun(', n, ') of', self.__class__)

help(Sample)
```

# Классы, рекомендации

- Создавайте все атрибуты объекта в методе `__init__`
  - если атрибут еще не имеет значения, присвойте ему `None`
- Не рекомендуется создавать атрибуты объекта в других методах класса
- Категорически не рекомендуется создавать атрибуты объекта при прямом обращении к объекту
- Категорически не рекомендуется создавать атрибуты класса при прямом обращении к классу
- Будьте особенно осторожны при модификации атрибутов в классе, который не является классом, экземпляром которого объект является
- Если есть выбор между классом и замыканием, выбирайте класс