Объектно-ориентированное программирование. Язык Python

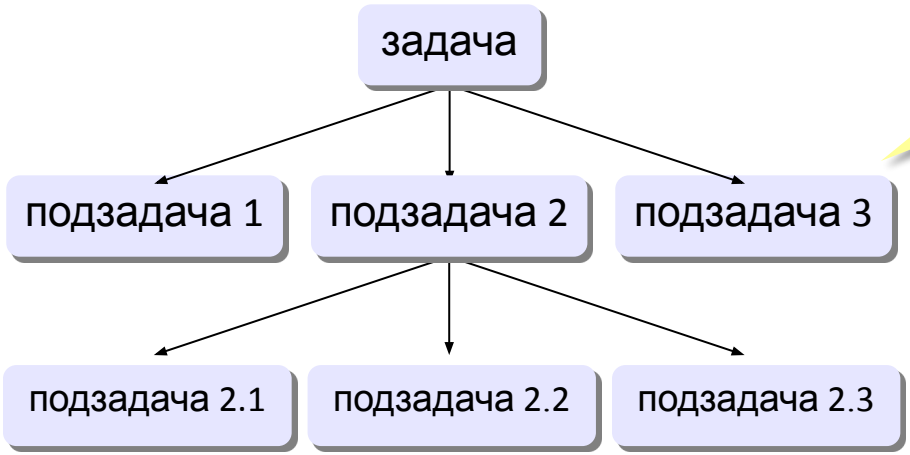
Зачем нужно что-то новое?

Главная проблема – **сложность**!

- программы из миллионов строк
- тысячи переменных и массивов

Э. Дейкстра: «Человечество еще в древности придумало способ управления сложными системами: «**разделяй и властвуй**»».

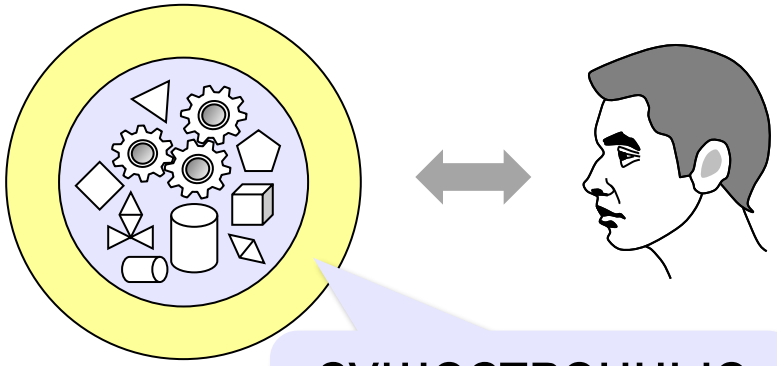
Структурное программирование:



декомпозиция по задачам

человек мыслит иначе, объектами

Как мы воспринимаем объекты?



существенные свойства

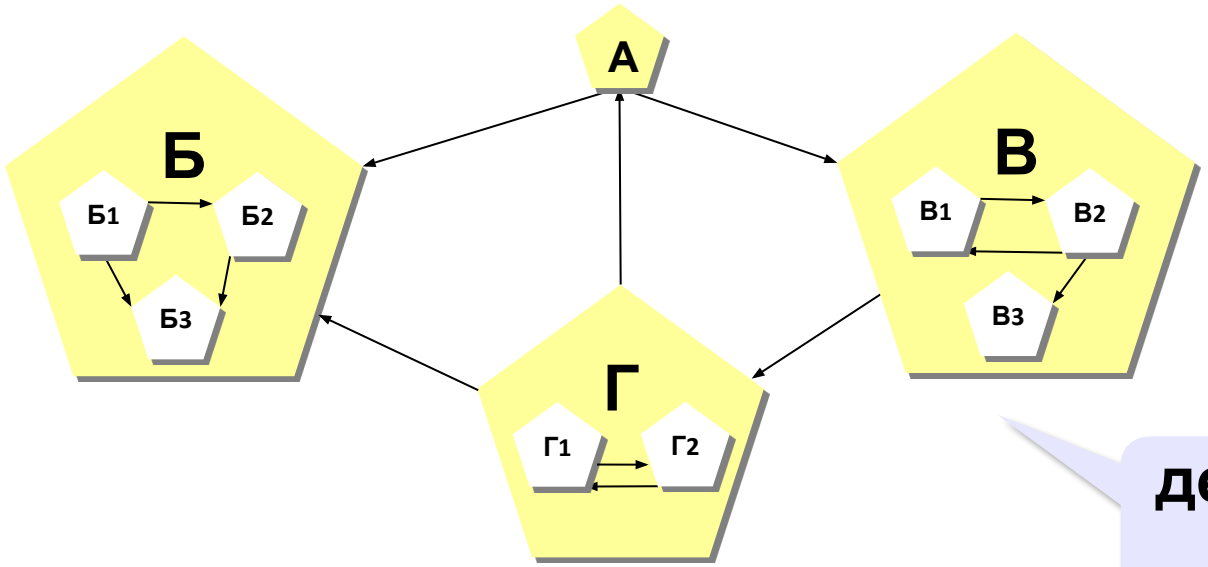
Абстракция – это выделение существенных свойств объекта, отличающих его от других объектов.

! Разные цели – **разные модели**!

Использование объектов

Программа – множество объектов (моделей), каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов.

! Нужно «разделить» задачу на объекты!



декомпозиция по объектам

С чего начать?

Объектно-ориентированный анализ (ООА):

- выделить **объекты**
- определить их существенные **свойства**
- описать **поведение** (команды, которые они могут выполнять)

Что такое объект?

Объектом можно назвать то, что имеет чёткие границы и обладает *состоянием* и *поведением*.

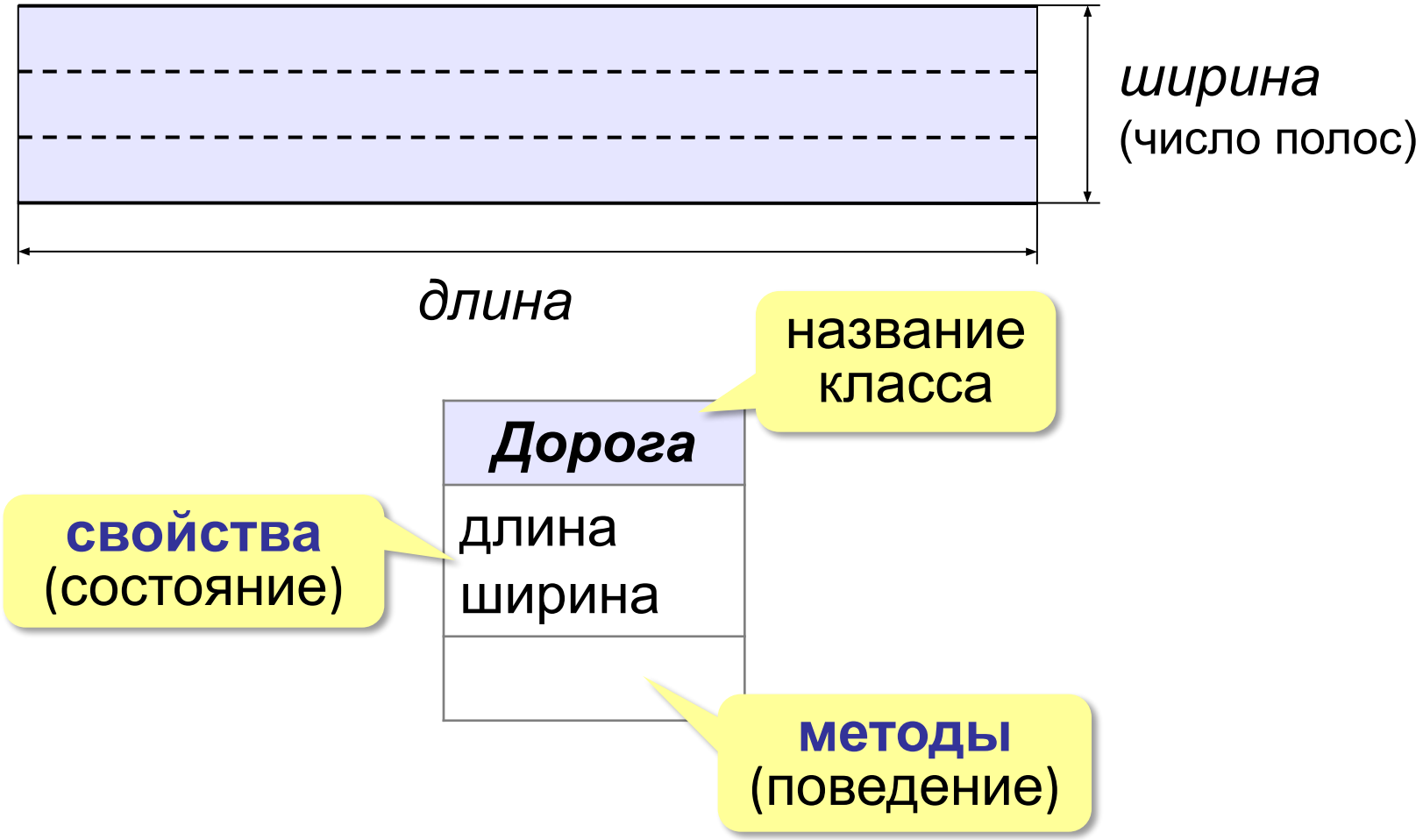
Состояние определяет поведение:

- лежачий человек не прыгнет
- незаряженное ружье не выстрелит

Класс – это множество объектов, имеющих общую структуру и общее поведение.

Модель дороги с автомобилями

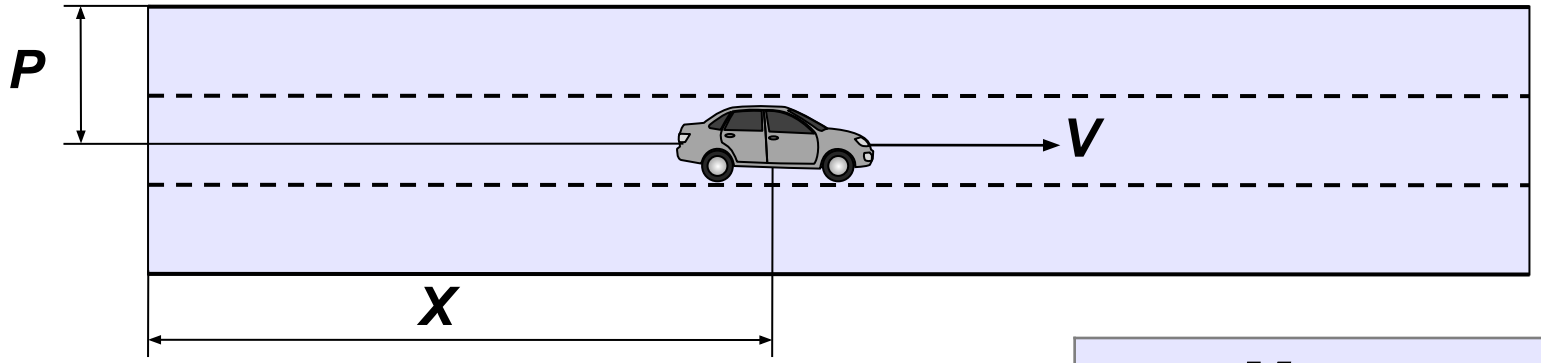
Объект «Дорога»:



Модель дороги с автомобилями

Объект «Машина»:

свойства: координаты и скорость



- все машины одинаковы
- скорость постоянна
- на каждой полосе – одна машина
- если машина выходит за правую границу дороги, вместо нее слева появляется новая машина

Машина
X (координата) P (полоса) V (скорость)
двигаться

Метод – это процедура или функция, принадлежащая классу объектов.

Модель дороги с автомобилями

Взаимодействие объектов:

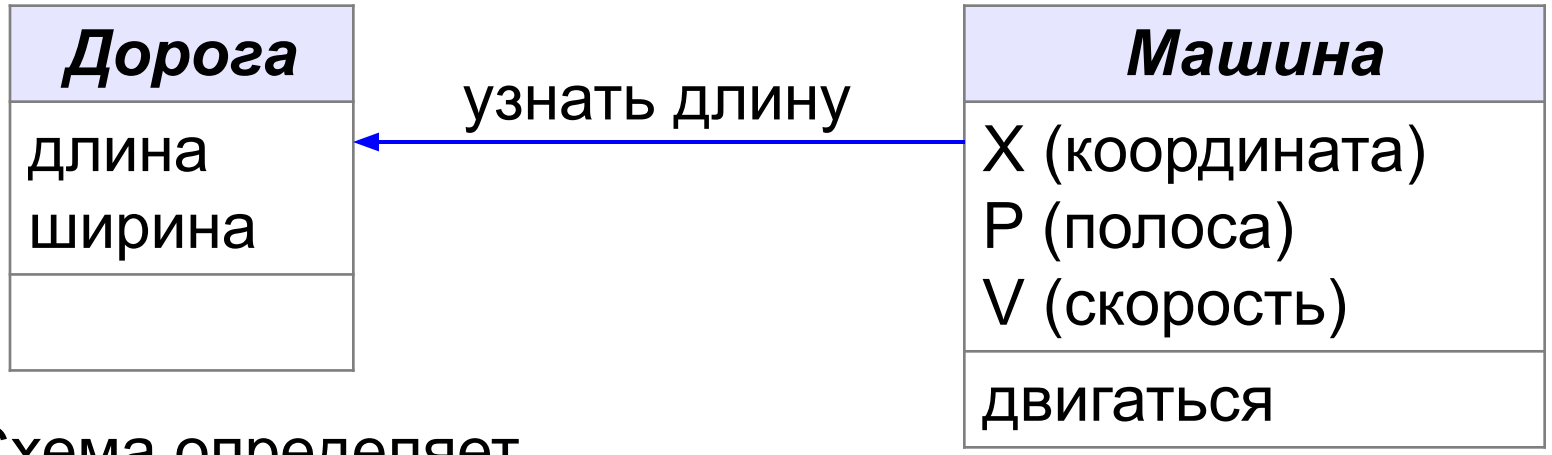


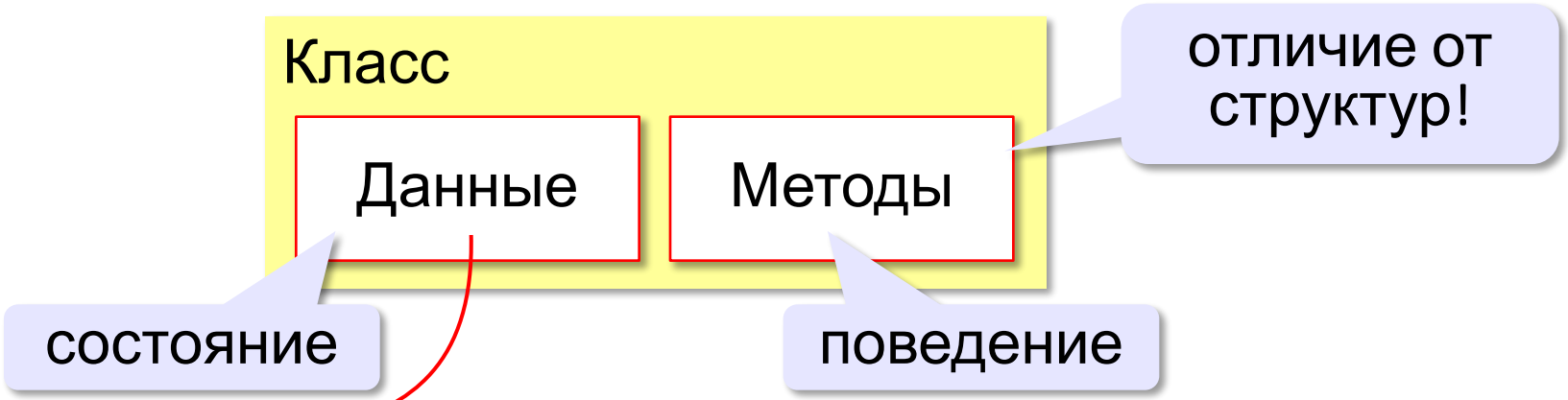
Схема определяет

- **свойства** объектов
- **методы**: операции, которые они могут выполнять
- **связи** (обмен данными) между объектами

Ни слова о внутреннем устройстве объектов!

Классы

- программа – множество взаимодействующих **объектов**
- любой объект – экземпляр какого-то **класса**
- **класс** – описание группы объектов с общей структурой и поведением



Поле – это переменная, принадлежащая объекту.

Класс «Дорога»

Описание класса:

```
class TRoad:
  pass
```

Объекты-экземпляры не создаются!

Создание объекта:

```
road = TRoad()
```

вызов конструктора

Конструктор – это метод класса, который вызывается для создания объекта этого класса.

Конструктор по умолчанию строится автоматически!

Новый конструктор – добавлений полей

initialization – начальные установки

```
class TRoad:
  def __init__( self ):
    self.length = 0
    self.width = 0
```

ссылка для обращения к самому объекту

оба поля обнуляются

точечная запись

! Конструктор задаёт начальные значения полей!

```
road = TRoad()
road.length = 60
road.width = 3
```

изменение значений полей

Конструктор с параметрами

автоматически

```
class TRoad:
  def __init__( self, length0, width0 ):
    self.length = length0
    self.width = width0
```

Вызов:

```
road = TRoad( 60, 3 )
```

Нет защиты от неверных входных данных!

Защита от неверных данных

```
class TRoad:
  def __init__( self, length0, width0 ):
    if length0 > 0:
      self.length = length0
    else:
      self.length = 0
    if width0 > 0:
      self.width = width0
    else:
      self.width = 0
```

```
self.length = length0 if length0 > 0 else 0
self.width = width0 if width0 > 0 else 0
```

Класс «Машина»

дорога, по которой едет

полоса

```
class TCar:
  def __init__( self, road0, p0, v0 ):
    self.road = road0
    self.P = p0
    self.V = v0
    self.X = 0
```

скорость

координата

Класс «Машина» – метод move

```
class TCar:
  def __init__ ( self, road0, p0, v0 ):
    ...
  def move ( self ):
    self.X += self.V
    if self.X > self.road.length:
      self.X = 0
```

перемещение за Δt = 1

если за пределами дороги

Равномерное движение:

$$X = X_0 + V \cdot \Delta t$$

$\Delta t = 1$ интервал дискретизации

перемещение за одну единицу времени

Основная программа

```
road = TRoad( 65, 3 )
car = TCar( road, 1, 10 )

car.move()
print ( "После 1 шага:" )
print ( car.X )

for i in range(10):
  car.move()
  print ( car.X )
```

? Что выведет?

```
10
10
20
30
40
50
60
0
10
20
30
40
```

дошли до конца дороги

```
class TCar:
  ...
  def move ( self ):
    self.X += self.V
    if self.X > self.road.length:
      self.X = 0
```

Массив машин

```
N = 3
cars = []
for i in range(N):
  cars.append( TCar(road, i+1, 2*(i+1)) )

for k in range(100):  # 100 шагов
  for i in range(N):  # для каждой машины
    cars[i].move()

print( "После 100 шагов:" )
for i in range(N):
  print( cars[i].X )
```

Что в этом хорошего и плохого?

ООП – это метод разработки **больших** программ!

- основная программа – простая и понятная
- классы могут разрабатывать разные программисты независимо друг от друга (+интерфейс!)
- повторное использование классов

- неэффективно для небольших задач

Задание

«A»: Построить класс Попугай (**Parrot**), который умеет говорить какую-то фразу, заранее определённую при описании класса.

Пример:

```
p = Parrot()
p.say()                          Привет, друзья!
```

«B»: Изменить класс из задания A так, чтобы фраза задавалась при создании конкретного экземпляра.

Пример:

```
p1 = Parrot( "Гав!" )
p2 = Parrot( "Мяу!" )
p1.say()                 Гав!
p2.say()                 Мяу!
```

Задание

«C»: Изменить класс из задания B так, чтобы фразу можно было изменять во время работы программы.

Пример:

```
p = Parrot( "Гав!" )
p.say()                Гав!
p.newText( "Мяу!" )
p.say()                Мяу!
```

«D»: Изменить класс из задания C так, чтобы при вызове метода **say** можно было задать число повторений.

Пример:

```
p = Parrot( "Гав!" )
p.say()                Гав!
p.newText( "Мяу!" )
p.say( 3 )                Мяу! Мяу! Мяу!
```

Задание

«E»: Изменить класс из задания D так, чтобы можно было добавлять фразы в набор фраз, которые знает попугай. При вызове метода **say** попугай выдаёт случайную фразу из своего набора.

Пример:

```
p = Parrot( "Гав!" )
p.say()              Гав!
p.learn( "Мяу!" )
p.say()              Гав!
p.say(3)             Мяу! Мяу! Мяу!
```

Зачем скрывать внутреннее устройство?

Объектная модель задачи:

- защита внутренних данных
- проверка входных данных на корректность
- изменение устройства с сохранением интерфейса

Инкапсуляция («помещение в капсулу») – скрытие внутреннего устройства объектов.

Также объединение данных и методов в одном объекте!

Защита внутренних данных

состояние

методы

Cat

энергия

настроение

голод

?

Можно изменять вучную?

```
метод есть
  + энергия
  + настроение
  - голод
```

```
метод спать
  + энергия
  + голод
```

```
метод играть
  - энергия
  + настроение
  + голод
```

!

Меняем состояние только через методы!

Пример: класс «перо»

```
class TPen:
  def __init__( self ):
    self.color = "000000"
```

R G B

По умолчанию все члены класса открытые (в других языках – **public**)!

```
class TPen:
  def __init__( self ):
    self.__color = "000000"
```

Как обращаться к полю?

Имена скрытых полей (**private**) начинаются с двух знаков подчёркивания!

Пример: класс «перо»

```
class TPen:
  def __init__( self ):
    self.__color="000000"
  def getColor( self ):
    return self.__color
  def setColor( self, newColor ):
    if len(newColor) !=6:
      self.__color="000000"
    else:
      self.__color=newColor
```

метод чтения

метод записи

если ошибка, чёрный цвет

Защита от неверных данных!

Пример: класс «перо»

Использование:

```
pen = TPen()
pen.setColor ( "FFFF00" )
print ( "цвет пера:", pen.getColor() )
```

установить цвет

прочитать цвет

Не очень удобно!

```
pen.color = "FFFF00"
print ( "цвет пера:", pen.color )
```

Свойство color

Свойство – это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его внутренней переменной.

```
class TPen:
   def __init__ ( self ):
      ...
   def __getColor ( self ):
      ...
   def __setColor ( self, newColor ):
      ...
   color = property ( __getColor,
                      __setColor )
```

метод чтения

свойство

метод записи

```
pen.color = "FFFF00"
print ( "цвет пера:", pen.color )
```

Изменение внутреннего устройства

Удобнее хранить цвет в виде числа:

```
class TPen:
  def __init__ ( self ):
    self.__color = 0
  def __getColor ( self ):
    return "{:06x}".format ( self.__color )
  def __setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = 0
    else:
      self.__color = int ( newColor, 16 )
  color = property (__getColor, __setColor)
```

число

число

число

Интерфейс не изменился!

Преобразование int → hex

Целое – в шестнадцатеричную запись:

16711935 → "FF00FF"

```
x = 16711935
sHex = "{:x}".format(x)
```

в шестнадцатеричной системе

? Что плохо?

255 → "FF" "0000FF"

правильно так!

```
x = 16711935
sHex = "{:06x}".format(x)
```

дополнить нулями слева

занять 6 позиций

Преобразование hex → int

"FF00FF" → 16711935

```
sHex = "FF00FF"
x = int ( sHex, 16 )
```

система счисления

Свойство «только для чтения»

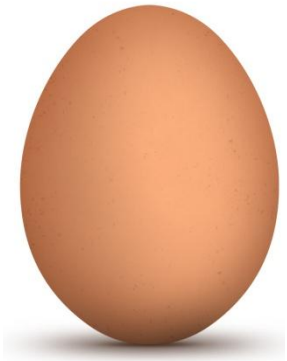
Скорость машины можно только читать:

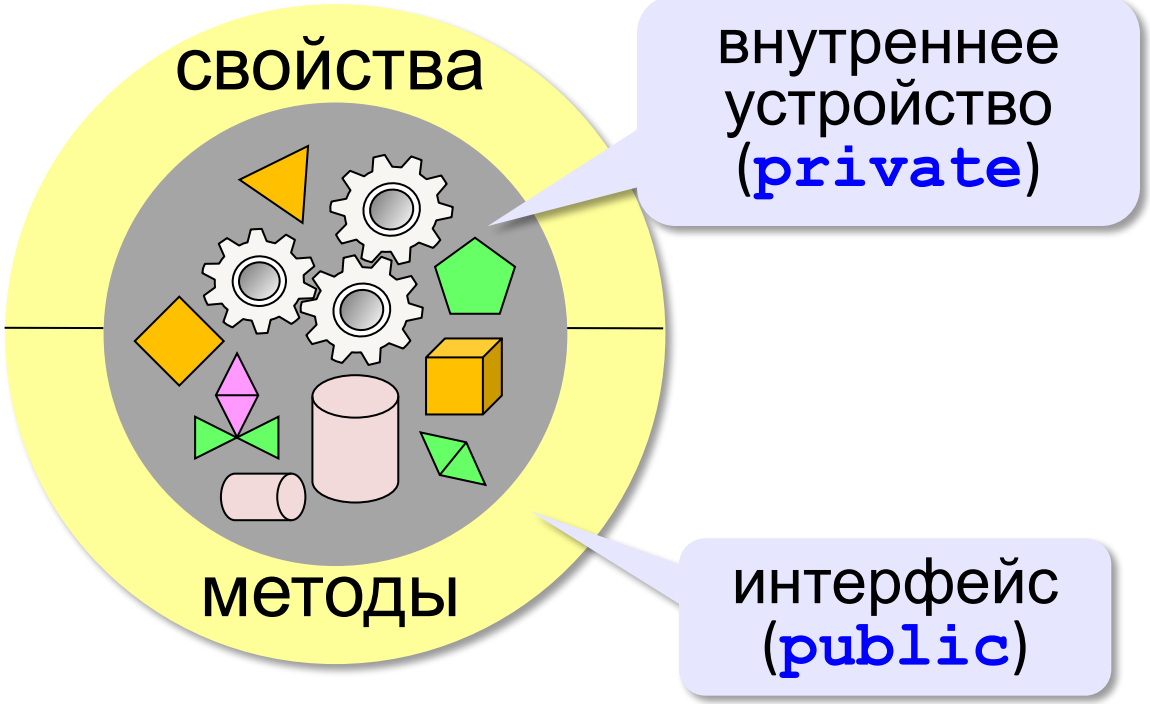
```
class TCar:
  def __init__( self ):
    self.__v = 0
  v = property( lambda x: x.__v )
```

нет метода записи

Скрытие внутреннего устройства

Инкапсуляция («помещение в капсулу»)





Задание

«А»: Построить класс РядЛампочек (**LampRow**), который хранит состояние ряда из 8 лампочек в виде символьной строки. Цифра 0 обозначает выключенную лампочку, цифра 1 – включенную.

Свойство **state** скрывает внутреннюю переменную **__state**, которая хранит состояние лампочек. При записи нового значения проверяется, что длина строки состояния равна 8, иначе записываются все нули.

Метод **show** выводит на экран состояние лампочек, обозначая выключенную лампочку как минус, а включённую – как «*».

Пример:

```
lamps = LampRow()
lamps.show()              --------
lamps.state = "10101010"
print( lamps.state )    10101010
lamps.show()                    *-*-*-*-
```

Задание

«B»: Дополните класс **LampRow** из задания A так, чтобы количество лампочек в цепочке можно было задавать в конструкторе.

Пример:

```
lamps = LampRow( 6 )
lamps.show()                ------
lamps.state = "101010"
print( lamps.state )    101010 lamps.show()
  *-*-*-
lamps.state = "10101010"  # ошибка
print( lamps.state )    000000 lamps.show()
  ------
```

Задание

«C»: Дополните класс **LampRow** из задания B так, чтобы лампочки могли гореть одним из двух цветов – красный цвет имеет код 1 и обозначается при выводе как «*», а зелёный цвет имеет код 2 и обозначается как «o».

Пример:

```
lamps = LampRow( 6 )
lamps.show()          ------
lamps.state = "102102"
print( lamps.state )   102102 lamps.show()
 *-o*-o
lamps.state = "10201010"  # ошибка
print( lamps.state )   000000 lamps.show()
 ------
```

Задание

«D»: Дополните класс **LampRow** из задания C так, чтобы код состояния хранился как целое число. При этом интерфейс (способ чтения и записи свойства **state**) не должен измениться.

Пример:

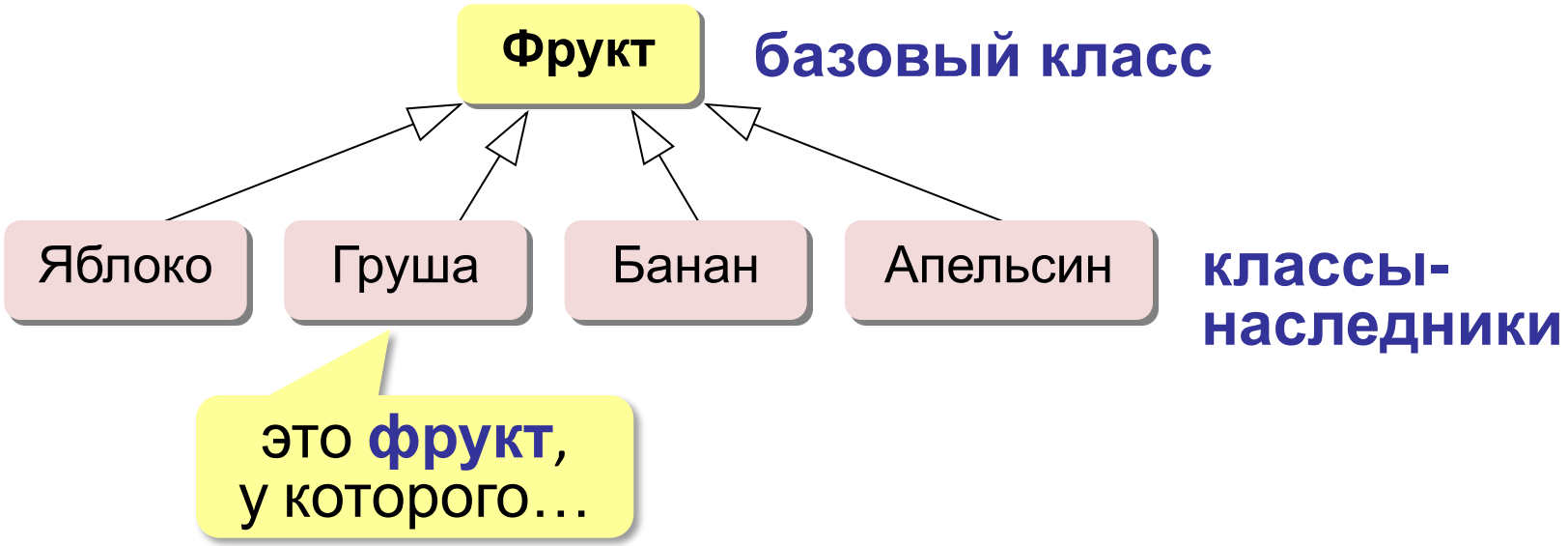
```
lamps = LampRow( 6 )
lamps.show()          ------
lamps.state = "102102"
print( lamps.state )   102102 lamps.show()
 *-o*-o*
lamps.state = "10201010"  # ошибка
print( lamps.state )   000000 lamps.show()
 ------
```

Классификации

? Что такое классификация?

Классификация – разделение изучаемых объектов на группы (классы), объединенные общими признаками.

? Зачем это нужно?



Что такое наследование?

класс *Двудольные*
семейство *Бобовые*
род *Клевер*
горный клевер

наследует свойства (имеет все свойства)

Класс Б является **наследником** класса А, если можно сказать, что Б – **это разновидность** А.

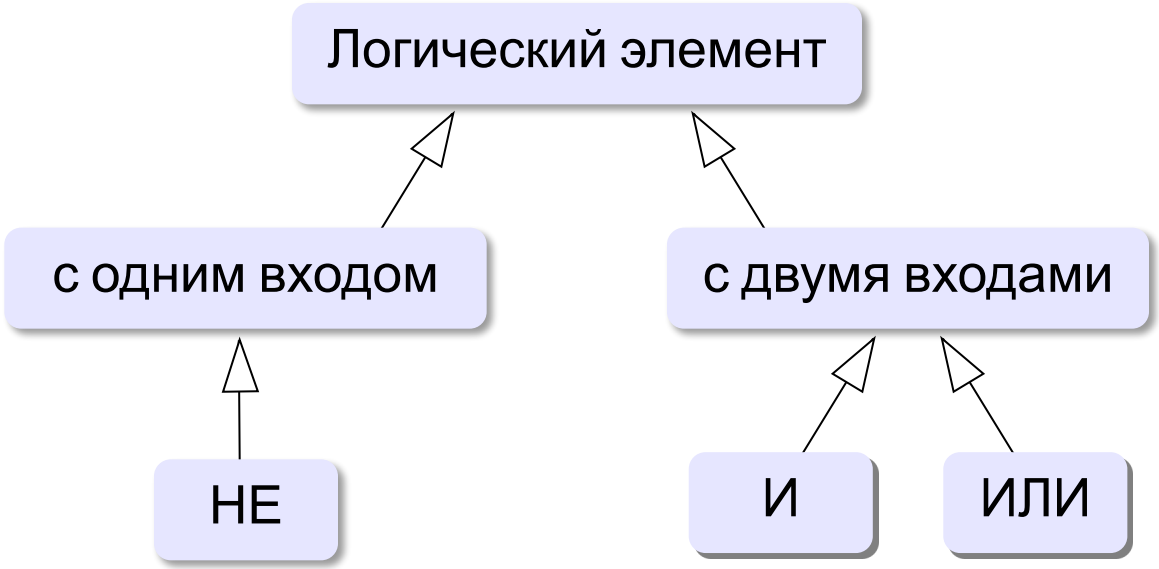
✓ яблоко – фрукт
✓ горный клевер – клевер
✗ машина – двигатель

яблоко – **это** фрукт

горный клевер – **это** растение рода *Клевер*

машина **содержит** двигатель (часть – целое)

Иерархия логических элементов



Объектно-ориентированное программирование – это такой подход к программированию, при котором программа представляет собой множество взаимодействующих **объектов**, каждый из которых является экземпляром определенного **класса**, а классы образуют иерархию **наследования**.

Базовый класс

ЛогЭлемент
In1 (вход 1) In2 (вход 2) Res (результат)
calc

```
class TLogElement:
  def __init__ ( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
```

поле доступно наследникам!

? Зачем хранить результат?

можно моделировать элементы с памятью (триггеры)

Базовый класс

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
  def __setIn1 ( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2 ( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property (lambda x: x.__in1, __setIn1)
  In2 = property (lambda x: x.__in2, __setIn2)
  Res = property (lambda x: x._res )
```

пересчёт выхода

только для чтения

Метод `calc`

Как написать метод **`calc`**?

```
class TLogElement:
    ...
    def calc ( self ):
        pass
```

заглушка

Нужно запретить создавать объекты **`TLogElement`**!

Абстрактный класс

- все логические элементы должны иметь метод **`calc`**
- метод **`calc`** невозможно написать, пока неизвестен тип логического элемента

Абстрактный метод – это метод класса, который объявляется, но не реализуется в классе.

Абстрактный класс – это класс, содержащий хотя бы один абстрактный метод.

нет логического элемента «вообще», как не «фрукта вообще», есть конкретные виды

Нельзя создать объект абстрактного класса!

`TLogElement` – абстрактный класс из-за метода **`calc`**

Абстрактный класс

```
class TLogElement:
  def __init__ ( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False

    if not hasattr ( self, "calc" ):
      raise NotImplementedError(
            "Нельзя создать такой объект!")
```

если у объекта нет атрибута (поля или метода) с именем `calc`...

создать («поднять», «выбросить») исключение

Что такое полиморфизм?

```
class TLogElement:
  def __init__( self ):
    ...
  def __setIn1 ( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
```

для каждого наследника вызывается свой метод **calc**

Полиморфизм – это возможность классов-наследников по-разному реализовать метод с одним и тем же именем.

греч.: $\pi o \lambda \upsilon$ — много, $\mu o \rho \varphi \eta$ — форма

Элемент «НЕ»

наследник от **TLogElement**

вызов конструктора базового класса

```
class TNot ( TLogElement ):
  def __init__ ( self ):
    TLogElement.__init__ ( self )

  def calc ( self ):
    self._res = not self.In1
```

? Почему не **__in1**?

! Это уже не абстрактный класс!

Элемент «НЕ»

Использование:

```
n = TNot()
n.In1 = False
print ( n.Res )
```

создание объекта

установка входа

вывод результата

Элементы с двумя входами

наследник от **TLogElement**

```
class TLog2In ( TLogElement ):
  pass
```

Можно ли создать объект этого класса?

нельзя, он абстрактный

Элементы с двумя входами

Элемент «И»:

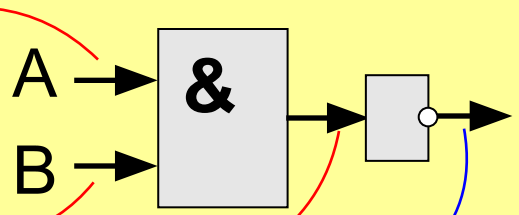
```
class TAnd ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 and self.In2
```

Элемент «ИЛИ»:

```
class TOr ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 or self.In2
```

Пример: элемент «И-НЕ»

```
elNot = TNot()
elAnd = TAnd()
print ( "  A | B | not(A&B) " );
print ( "--------------------" );
for A in range(2):
   elAnd.In1 = bool(A)
   for B in range(2):
      elAnd.In2 = bool(B)
      elNot.In1 = elAnd.Res
      print ( " ", A, "|", B, "|",
                          int(elNot.Res) )
```



Модульность

Идея: выделить классы в отдельный модуль **logelement.py**.

```
class TLogElement:
   ...
class TNot ( TlogElement ):
   ...
class TLog2In ( TLogElement ):
  pass
class TAnd ( TLog2In ):
   ...
class TOr ( TLog2In ):
   ...
```

Модульность

В основную программу:

```
import logelement
elNot = logelement.TNot()
elAnd = logelement.TAnd()
...
```

Сообщения между объектами

Задача – автоматическая передача сигналов по цепочке!

```
class TLogElement:
  def __init__ ( self ):
    ...
    self.__nextEl = None
    self.__nextIn = 0
  ...
  def link ( self, nextEl, nextIn ):
    self.__nextEl = nextEl
    self.__nextIn = nextIn
```

адрес следующего элемента в цепочке

номер входа следующего элемента

установка связи

Сообщения между объектами

После изменения выхода «дергаем» следующий элемент:

```
class TLogElement:
  ...
  def __setIn1 ( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
    if self.__nextEl:
      if self.__nextIn == 1:
        self.__nextEl.In1 = self._res
      elif __nextIn == 2:
        __nextEl.In2 = self._res
```

если следующий элемент установлен…

передать результат на нужный вход

Сообщения между объектами

Изменения в основной программе:

```
elNot = TNot()
elAnd = TAnd()
elAnd.link ( elNot, 1 )
print ( "  A | B | not(A&B) " );
print ( "--------------------" );
for A in range(2):
   elAnd.In1 = bool(A)
   for B in range(2):
      elAnd.In2 = bool(B)
      elNot.In1 = elAnd.Res
      print ( " ", A, "|", B, "|",
                          int(elNot.Res) )
```

установить связь

это уже не нужно!

Задание

«А»: Постройте класс **Pet** (домашнее животное) с двумя скрытыми полями: **__name** (имя) и **__age** (возраст). Они должны быть доступны для чтения через свойства **name** и **age** и недоступны для записи. Метод **gettingOlder** увеличивает возраст на 1 год. Класс **Pet** – абстрактный, он имеет абстрактный метод **say**.

Постройте два класса-наследника – **Cat** (кошка) и **Dog** (собака).Они должны реализовать метод **say**.

Описания классов должны быть в отдельном модуле **animals.py**.

Пример: см. следующий слайд.

Задание

«А»:

Пример:

```
from animals import *
p = Dog("Шарик", 5)
p.gettingOlder()
print( p.name + ":", p.age, "лет")
pets = [ Cat("Мурка", 3), p ]
for p in pets:
  p.say()
```

```
Шарик: 6 лет
Мурка: Мяу!
Шарик: Гав!
```

Задание

«B»: Добавьте класс **Mammal** (млекопитающее) – наследник класса **Pet** и предок для классов **Cat** и **Dog**. Он должен иметь метод **run** (бежать), который выводит сообщение вида «Вася побежал».

Пример:

```
from animals import *
pets = [Cat("Мурзик", 3),
        Dog("Шарик", 5) ]
for p in pets:
  p.say()
  p.run()
```

```
Мурзик: Мяу!
Мурзик побежал...
Шарик: Гав!
Шарик побежал...
```

Задание

«С»: Добавьте класс **Reptilia** (рептилии) – наследник класса **Pet** и предок для новых классов **Turtle** (черепаха) и **Snake** (змея). Он должен иметь метод **crawl** (ползти), который выводит сообщение вида «Вася пополз…».

Пример:

```
from animals import *
pets = [Cat("Мурзик", 3),
        Turtle("Зак", 32),
        Dog("Шарик", 5),
        Snake("Чаки", 2) ]
for p in pets:
  p.say()
  if isinstance(p, Mammal):
    p.run()
  if isinstance(p, Reptilia):
    p.crawl()
```

```
Мурзик: Мяу!
Мурзик побежал...
Зак: ...
Зак пополз...
Шарик: Гав!
Шарик побежал...
Чаки: ш-ш-ш-ш...
Чаки пополз...
```

Задание

«А»: Собрать полную программу и построить таблицу истинности последовательного соединения элементов «ИЛИ» и «НЕ».

Пример:

```
 A | B | not(A+B)
-------------------
 0 | 0 | 1
 0 | 1 | 0
 1 | 0 | 0
 1 | 1 | 0
```

Задание

«B»: Добавить в иерархию классов элементы «И-НЕ» (**TNAnd**) и «ИЛИ-НЕ» (**TNOr**), которые представляют собой последовательные соединения элементов «И» и «ИЛИ» с элементом «НЕ». Построить их таблицы истинности.

Пример:

```
 A | B | A nand B
-------------------
 0 | 0 | 1
 0 | 1 | 1
 1 | 0 | 1
 1 | 1 | 0
 A | B | A nor B
-------------------
 0 | 0 | 1
 0 | 1 | 0
 1 | 0 | 0
 1 | 1 | 0
```

Задание

«С»: Добавить в иерархию классов элемент «исключающее ИЛИ» (**TXor**) и «импликация» (**TImp**). Построить их таблицы истинности.

Пример:

```
 A | B | A xor B
-------------------
 0 | 0 | 0
 0 | 1 | 1
 1 | 0 | 1
 1 | 1 | 0

 A | B | A -> B
-------------------
 0 | 0 | 1
 0 | 1 | 1
 1 | 0 | 0
 1 | 1 | 1
```

Задание

«D»: Добавить в иерархию классов элемент «триггер» (**TTrigger**). Построить его таблицу истинности при начальных значениях выхода Q, равных 0 и 1.

Пример:

```
При Q = 0:
  A | B | Q
-------------------
  0 | 0 | 0
  0 | 1 | 0
  1 | 0 | 1
  1 | 1 | 1
При Q = 1:
  A | B | Q
-------------------
  0 | 0 | 1
  0 | 1 | 0
  1 | 0 | 1
  1 | 1 | 1
```