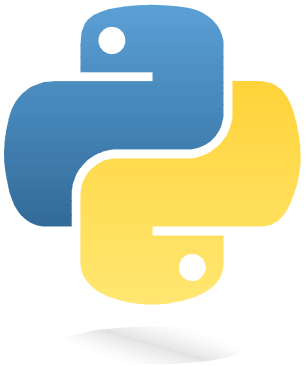
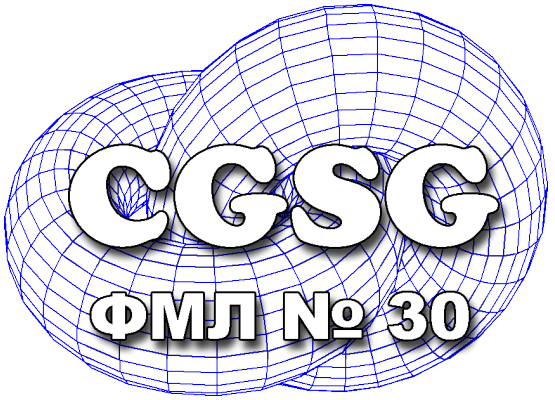
Язык программирования



python™

Лекция 2. Классы, модули, области видимости



Владимир Руцкий
<altsysrq@gmail.com>

План лекции

- Повторение
- Принципы ООП
- Области видимости
- Классы

Ссылки

- Презентации, использованные материалы и примеры можно найти в моём git: https://github.com/rutsky/python-course
- Как скачать:
  - выберите файл
  - нажмите «view raw» или «raw»
  - скопируйте или сохраните предложенный файл
- Презентации:
  - 00_introduction.pdf
  - 01_classes_modules.pdf
- Примеры и материалы в соответствующих директориях:
  - 00_introduction/...
  - 01_classes_modules/...

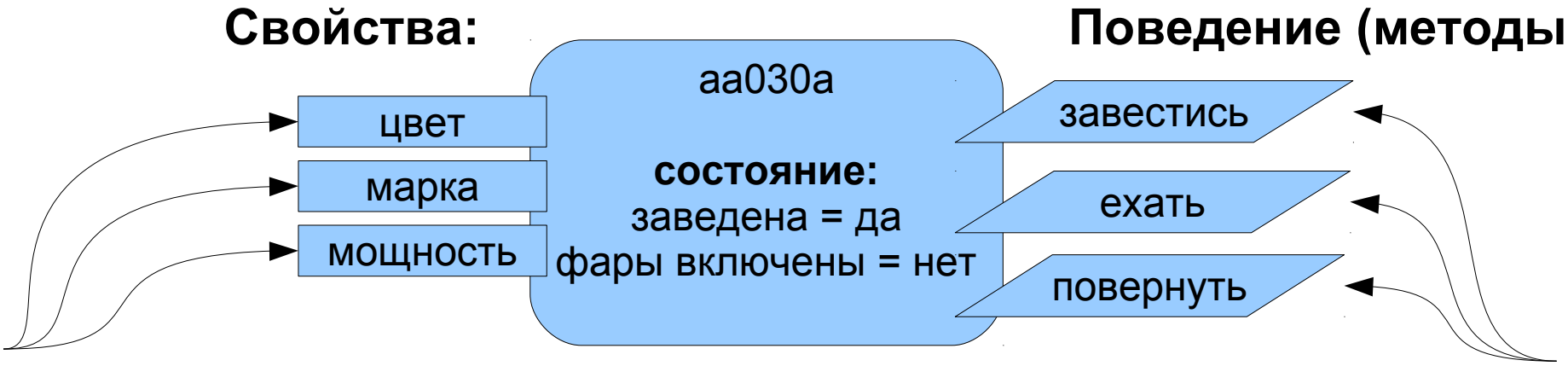
Повторение

Объектно ориентированное программирование (ООП)

- ООП - *парадигма* программирования - совокупность идей и понятий, определяющих стиль написания программ
- Основные понятия: *объект, класс, абстракция, наследование, инкапсуляция, полиморфизм*

Объекты (1/2)

- **Объект** - сущность, обладающая определённым состоянием, поведением и свойствами
  - Объект «автомобиль с номером аа030а» (конкретный)
  - **Внешний интерфейс** (доступен всем пользователям):
    - свойства: «цвет», «марка», «мощность двигателя», «количество мест»
    - поведение (функции): «завестись», «ехать», «повернуть», «включить фары»
  - **Внутреннее состояние** (доступно только объекту):
    - «заведена», «включены фары», «положение роторов», «напряжение на контурах»



Объекты (2/2)

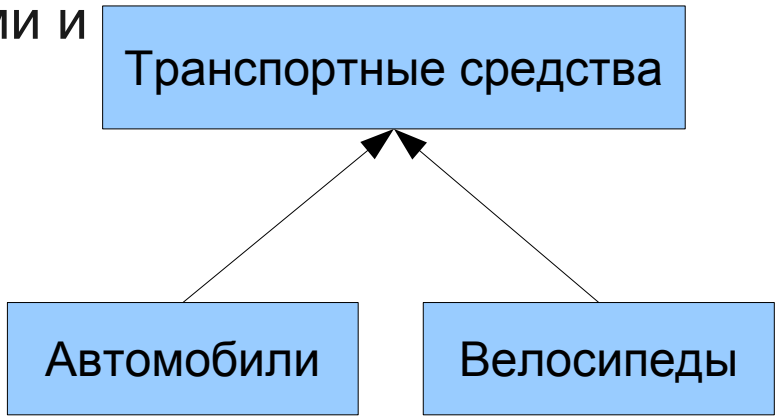
- Объект «водитель Пётр» взаимодействует с объектом «автомобиль аа030а» посредством **внешнего интерфейса**
  - Пётр нажимает педали, крутит руль, получает информацию о цвете и марке автомобиля
- Внешний объект «Пётр» **не может влиять на внутреннее состояние** объекта «автомобиль аа030а»
  - Механизм изменения внутреннего состояния может быть различным у разных объектов-автомобилей
  - Прямое изменение внутреннего состояния внешними объектами скорее всего приведёт к поломке системы (объекта «автомобиль аа030а»)
  - Говорят, что объект **инкапсулирует** свои внутренние свойства

- Объекты состоят из **внешнего интерфейса** и **внутренней реализации**
  - Обеспечивает гибкость - возможность свободного изменения внутренней реализации без боязни что-то сломать
  - Обеспечивает консистентность (согласованность) - объект сам меняет своё внутреннее состояние и обеспечивает его корректность

Классы

- **Класс** - совокупность объектов (**экземпляров класса**), объединённых общими свойствами и поведением
  - Класс «Автомобили» - совокупность объектов, имеющих
    - поведения: «завестись», «ехать», «повернуть»
    - свойства: «марка», «цвет», «количество мест»
  - Класс «Велосипеды»:
    - поведение: «сесть на велосипед», «ехать», «повернуть», «поднять велосипед»
    - свойства: «марка», «цвет», «масса»
  - Класс «Транспортные средства»:
    - поведения: «ехать», «повернуть»
- Класс «Транспортные средства» содержит в себе классы «Автомобили» и «Велосипеды»
- Классы «Автомобили» и «Велосипеды» **наследуют свойства и поведение** класса «Транспортные средства»

родительский (базовый) класс



дочерние (производные) классы (наследники)

Любой экземпляр класса «Автомобиль» является также экземпляром класса «Транспортное средство»

Абстракция и полиморфизм

- **Абстрагирование** - выделение значимых свойств, опуская незначимые
  - «Транспортное средство» - абстракция, такого объекта не существует
- При наследовании реализация метода может быть изменена - **полиморфизм**
  - Рассмотрим класс «Автомобиль Лада Калина»
  - Создадим **производный** от класса «Автомобиль Лада Калина» класс «Автомобиль Лада Калина с двигателем от Ford», в котором изменим **внутреннюю реализацию** методов «завестись» и «поехать» для двигателя от Ford
  - Новые автомобили, экземпляры «Автомобиль Лада Калина с двигателем от Ford», поддерживают интерфейс класса «Автомобиль Лада Калина», но имеют **изменённую (полиморфную)** реализацию

Области видимости Python (1/2)

```
>>> # Все переменные в Python определяются в каком-то модуле.  Создаваемые в интерактивной консоли переменные кладутся в модуль __main__.
... # Для поиска по имени переменной объекта, на который она ссылается, Python использует специальные словари --- области видимости.
... # Переменные, определяемые на уровне модуля, добавляются в глобальную область видимости.  Переменные, определяемые внутри функции или
... # класса сохраняются в локальную область видимости.  В любой строчке кода можно получить ссылки на локальную и глобальную область
... # видимости с помощью функций locals() и globals():
... globals() # вернёт словарь, отображающий имена переменных в объекты, на которые они ссылаются
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, '__package__': None}
>>> __name__ # имя текущего модуля
'__main__'
>>> # При указании в коде имени переменной, Python ищет имя переменной в областях видимости в следующем порядке: в локальной, в глобальной
... # и во встроенной в Python областях видимости.
... # Имя функции `globals' находится во встроенной области видимости, а имя `__name__' - в глобальной
... # Т.о. доступ к __name__ эквивалентен в этом случае:
... globals()['__name__']
'__main__'
>>> # При написания кода на уровне модуля, локальная область видимости совпадает с глобальной:
... locals() is globals()
True
>>> # Определим переменную. При этом Python добавит в текущую локальную область видимости пару с ключом `имя переменной' и значением объектом
... a = "test"
>>> locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, 'a': 'test', '__package__': None}
>>> # Удалим имя переменной (реальный объект удалится при сборке мусора, но об этом потом):
... del a
>>> locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, '__package__': None}
>>>
```

Области видимости Python (2/2)

```
>>> # Оператор `def' создаёт объект-функцию и добавляет её в локальную область видимости:
... def f(arg):
...     v = 30
...     return locals(), globals() # вернём локальную и глобальную о.в. для данной строки
...
>>> # Имя `f' занеслось в локальную о.в.
... locals()
{'f': <function f at 0x24a2848>, '__builtins__': <module '__builtin__' (built-in)>, '__package__': None, '__name__': '__main__', '__doc__': None}
>>> f("aaa")[0] # locals() из внутренности f()
{'arg': 'aaa', 'v': 30}
>>> # globals() из f() совпадает с globals из данного модуля
... globals() is f("bbb")[1]
True
>>> # При каждом вызове функции, для тела функции создаётся новая локальная о.в.
... # Т.к. всё, что делает оператор `def' это создание нового объекта и помещение в локальную
... # область видимости, то можно делать вложенные функции:
... def f1(arg):
...     s1 = "f1() local variable"
...     def f2(arg2):
...         return arg2, s1, some_glb_var # arg2 найдётся в локальной о.в., s1 во внешней локальной,
...                                       # а some_glb_var будет искаться в глобальной о.в.
...     return f2(30)
...
>>> f1("test")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in f1
  File "<stdin>", line 7, in f2
NameError: global name 'some_glb_var' is not defined
>>> some_glb_var = "global variable"
>>> f1("test")
(30, 'f1() local variable', 'global variable')
>>>
```

Классы в Python

```
>>> # Класс определяется с помощью оператора `class':
... # class ИмяКласса:
... #     выражение1
... #     выражение2
... #     ...
... class MyClass:
...     def f(self):
...         return 'Hello!'
...
>>> MyClass # - класс
<class __main__.MyClass at 0x1c47e88>
>>> x = MyClass()
>>> x       # - экземпляр (instance) класса MyClass
<__main__.MyClass instance at 0x1caf7a0>
>>> x.f()   # вызываем метод класса
'Hello!'
>>> class Car:
...     """Класс автомобиля (это docstring)"""
...     def __init__(self, name):
..          # Конструктор класса --- этот метод вызывается при инициализации
...         # вновь созданного экземпляра класса.
...         # Первый аргумент `self' --- экземпляр класса (кого инициализируем)
...         self.car_name = name # --- записываем в атрибут экземпляра класса с
...                              # именем `car_name' значение переменной `name'
...     def hello(self):
...         # Определим метод класса hello()
...         return "Hello, my name is " + self.car_name + "!"
...
>>> # Создаём экземпляр класса (в конструктор передаётся name="aa030a")
... car_instance = Car("aa030a")
>>> car_instance.car_name
'aa030a'
>>> car_instance.hello()
'Hello, my name is aa030a!'
>>>
```

Прочее

- Атрибуты объектов
- Модули