Язык программирования

Лекция № 5

Владимир Владимирович Руцкий
rutsky.vladimir@gmail.com

План занятия

- Принципы объектно-ориентированного программирования (ООП)
- Классы в Python
- Практика

Объектно ориентированное программирование (ООП)

- ООП — *парадигма* программирования — совокупность идей и понятий, определяющих стиль написания программ
  - Примеры других парадигм программирования: структурное программирование, функциональное программирование.

    См. https://ru.wikipedia.org/wiki/Парадигма_программирования
- Основные понятия ООП: *объект*, *класс*, *абстракция*, *наследование*, *инкапсуляция*, *полиморфизм*

Объект. Состояние. Интерфейс

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

Внешний интерфейс

- Объект «водитель Пётр» взаимодействует с объектом «автомобиль аа030а» посредством **внешнего интерфейса**
  - Пётр нажимает педали, крутит руль, получает информацию о цвете и марке автомобиля

Внутреннее состояние

- Внешний объект «Пётр» **не должен непосредственно взаимодействовать с внутренним состоянием** объекта «автомобиль аа030а»
  - «Пётр» не должен соединять контакты электросистемы объекта «автомобиля», не должен двигать роторы двигателя и т.п.
  - 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 — скрывает своё внутреннее состояние

Преимущества объектов

- Объекты состоят из **внешнего интерфейса** и **внутренней реализации**
- Взаимодействие с объектом — только через внешний интерфейс
  - Обеспечивает гибкость — возможность свободного изменения внутренней реализации без боязни что-то сломать
    - У всех объектов «автомобилей» единый интерфейс управления (с функциями «поверни руль», «включи дальний свет» и т.п.), но разная внутренняя реализация
  - 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», «макс. скорость», «количество мест» «мощность двигателя»
  - Класс «Велосипеды»:
    - поведение: «ехать», «повернуть», «подпрыгнуть»
    - свойства: «марка», «цвет», «макс. скорость», «материал рамы»
  - Класс «Транспортные средства»:
    - поведения: «ехать», «повернуть»
    - свойства: «марка», «цвет», «макс. скорость»

Наследование классов

- Класс «Транспортные средства» содержит в себе классы «Автомобили» и «Велосипеды»
  - каждый экземпляр класса «Автомобили» и класса «Велосипеды» является экземпляром класса «Транспортные средства»
- Классы «Автомобили» и «Велосипеды» **наследуют свойства и поведение** класса «Транспортные средства»
  - Все «транспортные средства» имеют метод «ехать» и свойство «цвет»
- Говорят, что
  - «Автомобили» и «Велосипеды» — **дочерние** (или **производные**) классы для класса «Транспортные средства»
  - класс «Транспортные средства» — **родительский** (или **базовый**) для классов «Автомобили» и «Велосипеды»

Абстракция

- **Абстрагирование** — выделение значимых свойств, опуская незначимые
- Классы — абстракции
  - «Транспортное средство» — абстракция
    - Для «транспортных средств» важны только «цвет», «макс. скорость» и возможность «ехать», «повернуть»
  - «Автомобиль» — тоже абстракция
    - Для «автомобилей» важно, что они имеют двигатель определённой «мощности» (в отличие от «велосипеда»)

Полиморфизм

- 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

Классы в Python

```
>>> # Класс определяется с помощью конструкции `class':
... # class ИмяКласса:
... #     выражение1
... #     выражение2
... #     ...
... class MyClass:
...     def f(self):
...         return 'Hello!'
...
>>> MyClass # Оператор class создал новый класс
<class '__main__.MyClass'>
>>> x = MyClass() # `вызов' класса — создание экземпляра класса
>>> x  # - экземпляр (instance) класса MyClass (конкретный объект)
<__main__.MyClass object at 0x...>
>>> # атрибуты и методы экземпляра доступны через точку
... x.f()   # вызываем метод класса
'Hello!'
>>> y = MyClass() # создадим ещё один экземпляр класса
>>> y.f()
'Hello!'
>>>
```

Методы

```
>>> class MyClass:
...     # Все методы класса принимают первым аргументом экземпляр класса `self'
...     # (можно назвать по-другому, но принято `self')
...     def f(self):
...         return "I'm {0}.".format(id(self))
...     def greet_user(self, name):
...         # Методы — обычные функции с первым аргументом-объектом
...         print("Hello {0}!".format(name))
...
>>> x = MyClass() # создадим экземпляр класса
>>> id(x)
140454803317072
>>> x.f()   # вызываем метод класса
"I'm 140454803317072."
>>> y = MyClass() # создадим ещё один экземпляр класса
>>> id(y)
140454803318096
>>> y.f()
"I'm 140454803318096."
>>> x.greet_user("John")
Hello John!
>>>
```

Атрибуты (1/2)

```
>>> class MyClass:
...     def set_name(self, new_name):
...         # Объектам можно добавлять/удалять/изменять атрибуты
...         self.name = new_name  # установка значения атрибута `name'
...     def greet_user(self):
...         print("Hello, {0}!".format(self.name))
...
>>> x = MyClass()
>>> # атрибуты и методы экземпляра доступны через точку
... x.abc = 'test'  # Добавление (или изменение значения) атрибута `abc'
>>> x.abc  # получение значения атрибута
'test'
>>> x.ttt  # атрибуты должны быть установлены перед использованием
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute 'ttt'
>>> x.set_name("Valery")
>>> x.greet_user()
Hello, Valery!
>>> x.name
'Valery'
>>> x.name = 'John'
>>> x.greet_user()
Hello, John!
>>>
```

Атрибуты (2/2)

```
>>> class MyClass:
...     def set_name(self, new_name):
...         # Объектам можно добавлять/удалять/изменять атрибуты
...         self.name = new_name  # установка значения атрибута `name'
...     def greet_user(self):
...         print("Hello, {0}!".format(self.name))
...
>>> x = MyClass()
>>> y = MyClass()
>>> # У каждого класса свой набор атрибутов
>>> x.set_name('X')
>>> y.set_name('Y')
>>> x.greet_user()
Hello, X!
>>> y.greet_user()
Hello, Y!
>>> x.test = 'test'
>>> y.test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute 'test'
>>> # Атрибуты классов хранятся в специальном атрибуте-словаре __dict__
... x.__dict__
{'test': 'test', 'name': 'X'}
>>> y.__dict__
{'name': 'Y'}
>>>
```

Конструктор

```
>>> class User:
...     """Класс пользователя (это docstring, необязательный)"""
...     def __init__(self, name):
...         # Конструктор класса — этот метод вызывается при инициализации
...         # вновь созданного экземпляра класса.
...         # Первый аргумент `self' — экземпляр класса (кого инициализируем)
...         # Остальные аргументы — те, которые передали при создании экземпляра
...         self.my_name = name # — записываем в атрибут экземпляра класса с
...                             # именем `my_name' значение переменной `name'
...     def hello(self):
...         # С помощью self.my_name получаем значение имени для данного
...         # экземпляра
...         return "Hello, my name is {0}!".format(self.my_name)
...
>>> # Создаём экземпляр класса (в конструктор передаётся name="Peter")
... user_instance = User("Peter")
>>> user_instance.my_name
'Peter'
>>> user_instance.hello()
'Hello, my name is Peter!'
>>>
```

Перегрузка

```
>>> class OverloadsTest:
...     # Перегрузок в Python нет, вторая функция f() заменит первую f()
...     def f(self, a, b, c):
...         print("F1")
...     def f(self, a):
...         print("F2")
>>> o = OverloadsTest()
>>> o.f(1)
F2
>>> o.f(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 2 positional arguments but 4 were given
>>> # Для перегрузок нужно использовать переменное количество аргументов и
... # проверку типов
... class MyRange:
...     def __init__(self, start, stop=None, step=1):
...         self.numbers = []
...         if stop is None:
...             start, stop = 0, start
...         i = start
...         while i < stop:
...             self.numbers.append(i)
...             i += step
...
>>> MyRange(10).numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> MyRange(2, 7).numbers
[2, 3, 4, 5, 6]
>>> MyRange(2, 10, 2).numbers
[2, 4, 6, 8]
>>>
```

Статические переменные

```
>>> class User:
...     # Объявления в классе являются статическими, т.е. общими для всех
...     # экземпляров класса.
...     greeting = "Hello " # статический член
...     def __init__(self, name):
...         self.my_name = name
...     def hello(self):
...         # Также к `greeting' можно обратиться как `User.greeting'
...         return self.greeting + self.my_name
...
>>> peter = User("Peter")
>>> sam = User("Sam")
>>> peter.hello()
'Hello Peter'
>>> sam.hello()
'Hello Sam'
>>> # Изменим статический член User.greeting
... User.greeting = "Hi "
>>> peter.hello()
'Hi Peter'
>>> sam.hello()
'Hi Sam'
>>> def new_hello(self):
...     return "New hello() called with greeting `" + self.greeting + \
...         "' and name `" + self.my_name + "'"
...
>>> # Изменим статический член класса функцию hello():
... User.hello = new_hello
>>> peter.hello()
"New hello() called with greeting `Hi ' and name `Peter'"
>>> sam.hello()
"New hello() called with greeting `Hi ' and name `Sam'"
>>>
```

Приватные атрибуты и методы

```
>>> class User:
...     # Приватные (скрытые, внутренние) атрибуты и методы принято именовать
...     # начиная с одного подчеркивания
...     _greeting = "Hello "
...     def __init__(self, name, surname):
...         self._name = name
...         # Имена внутри классов, начинающиеся с двух подчеркиваний, и
...         # заканчивающиеся не более, чем одним подчеркиванием, прозрачно
...         # "переименовываются" (name mangling): к ним добавляется имя класса
...         self.__surname = surname
...     def _get_full_name(self): # приватный метод
...         return self._name + " " + self.__surname
...     def hello(self):
...         return self._greeting + self._get_full_name()
...
>>> user_instance = User("Peter", "Smith")
>>> user_instance.hello()
'Hello Peter Smith'
>>> user_instance._name  # обращаться к приватным атрибутам плохо!
'Peter'
>>> user_instance.__surname
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'User' object has no attribute '__surname'
>>> user_instance.__dict__
{'_name': 'Peter', '_User__surname': 'Smith'}
>>>
```

Наследование

```
>>> class User:
...     greeting = "Hello "
...     def __init__(self, name):
...         self.my_name = name
...     def hello(self):
...         return "Hello, my name is {0}!".format(self.my_name)
...
>>> # В Python есть множественное наследование классов:
... # class ИмяКласса(ИмяБазовогоКласса1, ИмяБазовогоКласса2, ...):
... #     выражение1
... #     выражение2
... #     ...
... class UserWithSurname(User): # Наследуем свойства класса User
...     def __init__(self, name, surname):
...         # Вызываем конструктор базового класса с необходимыми
...         # аргументами (он присвоит в self.my_name=name)
...         super(UserWithSurname, self).__init__(name)
...         self.my_surname = surname
...     def hello(self): # hello() переопределяется (полиморфизм)
...         return self.greeting + self.my_name + " " + self.my_surname
...     def old_hello(self):
...         # Явно вызываем метод базового класса
...         return super(UserWithSurname, self).hello()
...
>>> peter = UserWithSurname("Peter", "Ivanov")
>>> peter.hello()
'Hello Peter Ivanov'
>>> peter.old_hello()
'Hello, my name is Peter!'
>>> # Замечание: поддерживается «ромбовидное» наследование
```

Функции isinstance(), issubclass()

```
>>> class Base:
...     pass
...
>>> class Derived(Base):
...     pass
...
>>> b = Base()
>>> d = Derived()
>>> # isinstance() проверяет, является ли объект экземпляром класса
... isinstance(b, Base)
True
>>> isinstance(b, Derived)
False
>>> isinstance(d, Base)
True
>>> isinstance(d, Derived)
True
>>> # issubclass() проверяет, является класс подклассом другого класса
... issubclass(Derived, Base)
True
>>> issubclass(Base, Derived)
False
>>> issubclass(Derived, Derived)
True
>>> issubclass(Base, Base)
True
>>>
```

Специальные методы (1/3)

```
>>> class Vector:
...     def __init__(self, x, y):
...         self._x = x
...         self._y = y
...     def __str__(self):
...         # Вызывается при конструировании строки (str) от объекта
...         # (строковой вид объекта произвольного формата)
...         return "({0}, {1})".format(self._x, self._y)
...     def __repr__(self):
...         # Вызывается при выводе объектов на экран в интерпретаторе
...         # (строковой вид объекта произвольного формата, обычно строка
...         # как этот объект можно создать)
...         return "Vector({0}, {1})".format(self._x, self._y)
...     def __call__(self, *args, **kwargs):
...         # Вызывается при "вызове" объекта (obj=Vector(); obj(...))
...         print("Called with {0} {1}".format(str(args), str(kwargs)))
...
>>> v = Vector(2, 3)
>>> str(v)
'(2, 3)'
>>> v
Vector(2, 3)
>>> v(1, 2, test="data")
Called with (1, 2) {'test': 'data'}
>>>
```

Подробно: http://docs.python.org/3/reference/datamodel.html

Специальные методы (2/3)

```
>>> import numbers # в numbers определены классы чисел
>>> class Vector:
...     def __init__(self, x, y):
...         self._x = x
...         self._y = y
...     def __repr__(self):
...         return "Vector({0}, {1})".format(self._x, self._y)
...     def __add__(self, v):
...         # Вызывается при попытке сложить данный объект с чем-то (obj + v)
...         # Возвращаемой значение из функции — результат (obj + v)
...         assert isinstance(v, Vector)  # если сложить не с вектором — ошибка
...         return Vector(self._x + v._x, self._y + v._y)
...     def __sub__(self, v):  # (obj - v)
...         assert isinstance(v, Vector)
...         return Vector(self._x - v._x, self._y - v._y)
...     def __mul__(self, scalar):  # (obj * scalar)
...         assert isinstance(scalar, numbers.Number) # умножаем только на числа
...         return Vector(self._x * scalar, self._y * scalar)
...     def __rmul__(self, scalar): # (scalar * obj)
...         assert isinstance(scalar, numbers.Number)
...         return self * scalar
...     # Можно определить операции для +, -, *, /, //, %, divmod(), pow(), **, <<, >>, &,
...
>>> v1 = Vector(1, 2)
>>> v2 = v1 + Vector(3, -1); v2
Vector(4, 1)
>>> v2 * -2
Vector(-8, -2)
>>> 0.5 * v2
Vector(2.0, 0.5)
>>> v2 += Vector(1, 0); v2 # с помощью __iadd__ можно задать +=, здесь — автоматически
Vector(5, 1)
>>>
```

Специальные методы (3/3)

```
>>> import numbers # в numbers определены классы чисел
>>> class Vector:
...     def __init__(self, x, y):
...         self._x = x
...         self._y = y
...     def __getattr__(self, name):
...         # Вызывается, когда происходит получение атрибута (t = obj.name)
...         # (name — имя запрошенного атрибута) и name не найден в обычных местах
...         if name == 'x':
...             return self._x
...         elif name == 'y':
...             return self._y
...         elif name == 'length':
...             return (self._x ** 2 + self._y ** 2) ** 0.5
...         else:
...             raise AttributeError()
...     def __setattr__(self, name, value):
...         # Вызывается, когда происходит присвоение атрибута (obj.key = t)
...         # (name — имя запрошенного атрибута) и name не найден в обычных местах
...         if name in ['x', 'y']:
...             assert isinstance(value, numbers.Number)
...         super(Vector, self).__setattr__(name, value)
...
>>> v = Vector(3, 4); v.x
3
>>> v.y = 'test'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __setattr__
AssertionError
>>> v.length
5.0
>>>
```

Практика