Практикум по программированию на языке Python

Занятие 4: Основы ООП, особенности ООП в Python

Мурат Апишев (mel-lain@yandex.ru)

Москва, 2020

Парадигмы проектирования кода

Императивное программирование (язык ассемблера)

```
mov ecx, 7
```

Декларативное программирование (SQL)

```
select * from table where index % 10 == 0
```

Подвид: функциональное программирование (Haskell)

```
filter even [1..10]
```

Объектно-ориентированное программирование (C++)

```
auto car = new Car(); a.fill_up(10);
```

Объектно-ориентированное программирование

- Программа, как и окружающий мир, состоит из сущностей
- Сущности имеют какое-то внутренее состояние
- Также сущности взаимодействуют друг с другом
- ООП нужно для описания программы в виде сущностей и их взаимоотношений
- При этом и на сущности, и на отношения накладываются ограничения
- Это позволяет писать более короткий, простой и переиспользуемый код

Базовые понятия: класс и объект

- **Класс** представляет собой тип данных (как int или str)
- Это способ описания некоторой сущности, её состояния и возможного поведения
- Поведение при этом зависит от состояния и может его изменять
- **Объект** - это конретный представитель класса (как переменная этого типа)
- У объекта своё состояние, изменяемое поведением
- Поведение полностью определяется правилами, описанными в классе

Базовые понятия: интерфейс

- **Интерфейс** - это класс, описывающий только поведение
- У интерфейса нет состояния
- Как следствие, создать объект типа интерфейса невозможно
- Вместо этого описываются классы, которые реализуют этот интерфейс и, в то же время, имеют состояние
- С помощью интерфейсов реализуется полиморфизм (будет далее)
- Программирование на уровне интерфейсов делает код читаемее и проще
- Интерфейсы в некоторых языках (например, Java) решают проблему отсутствия множественного наследования

Интерфейс: пример

```
interface SomeCar {

  fill_up(gas_volume)

  turn_on()

  turn_off()

}
```

Интерфейс

- не содержит информации о состоянии автомобиля
- не содержит информации о том, как выполнять описанные команды
- он только описывает то, какие операции должны быть доступны над объектом, который претендует на то, чтобы быть автомобилем

Реализация интерфейса

```
class ConcreteCar {

  fill_up(gas_volume) { tank += gas_volume }

  turn_on() { is_turned_on = true }

  turn_off() { is_turned_on = false }

  tank = 0

  is_turned_on = false

}
```

- Обычно данные класса называют *полями* (или *атрибутами*), а функции - *методами*
- **Абстрактный класс** - промежуточный вариант между интерфейсом и обычным классом

Принципы ООП

- **Абстракция** - выделение важных свойств объекта и игнорирование прочих
- **Инкапсуляция** - хранение данных и методов работы с ними внутри одного класса с доступом к данным только через методы
- **Наследование** - возможность создания наследников, получающих все свойства родителей с возможностью их переопределения и расширения
- **Полиморфизм** - возможность использования объектов разных типов с общим интерфейсом без информации об их внутреннем устройстве

ООП в Python

- Python - это полностью объектно-ориентированный язык
- В Python абсолютно всё является объектами, включая классы
- Полностью поддерживаются все принципы ООП, кроме инкапсуляции
- Инкапсуляция поддерживается частично: нет ограничения на доступ к полям класса
- Поэтому для инкапсуляции используют договорные соглашения

Так выглядят классы в Python

In [1]:
```
class ConcreteCar:
    def __init__(self):
        self.tank = 0
        self.is_turned_on = False

    def fill_up(self, gas_volume):
        self.tank += gas_volume

    def turn_on(self):
        self.is_turned_on = True

    def turn_off(self):
        self.is_turned_on = False

car = ConcreteCar()
print(type(car), car.__class__)

car.fill_up(10)
print(car.tank)
```

```
<class '__main__.ConcreteCar'> <class '__main__.ConcreteCar'>
10
```

Функция `__init__`

- Главное: `__init__` - не конструктор! Она ничего не создаёт и не возвращает
- Созданием объекта занимается функция `__new__`, переопределять которую без необходимости не надо
- `__init__` получает на вход готовый объект и инициализирует его атрибуты

В отличие от C++, атрибуты можно добавлять/удалять на ходу:

In [28]:
```
class Cls:
    pass

cls = Cls()
cls.field = 'field'
print(cls.field)

del cls.field
print(cls.field)  # AttributeError: 'Cls' object has no attribute 'field'
```

```
field
```

Параметр `self`

- Метод класса отличается от обычной функции только наличием объекта `self` в качестве первого аргумента
- Это то же самое, что происходит в C++/Java (там аналогом `self` является указатель/ссылка `this`)
- Название `self` является общим соглашением, но можно использовать и другое (не надо!)
- Метод класса, не получающий на вход `self` является *статическим*, то есть применяется вне зависимости от существования объектов данного класса
- Статические методы часто используются для специализированного создания объектов класса
- В Python `__new__` является статическим методом

Как быть с инкапсуляцией

- Приватное поле прежде всего должно быть обозначено таковым
- В Python для этого есть соглашения:

In [3]:
```
class Cls:
    def __init__(self):
        self.public_field = 'Ok'
        self._private_field = "You're shouldn't see it"
        self.__very_private_field = "YOU REALLY SHOULDN'T SEE IT!!!"

cls = Cls()
print(cls.public_field)
print(cls._private_field)
print(cls.__very_private_field)
```

```
Ok
You're shouldn't see it

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-6537944786b6> in <module>
      8 print(cls.public_field)
      9 print(cls._private_field)
---> 10 print(cls.__very_private_field)

AttributeError: 'Cls' object has no attribute '__very_private_field'
```

In [4]:
```
print(cls._Cls__very_private_field)
```

```
YOU REALLY SHOULDN'T SEE IT!!!
```

Атрибуты объекта и класса

In [11]:
```
class Cls:
    pass

cls = Cls()
print([e for e in dir(cls) if not e.startswith('__')])

cls.some_obj_attr = '1'
print([e for e in dir(cls) if not e.startswith('__')])
```

```
[]
['some_obj_attr']
```

In [12]:
```
print([e for e in dir(Cls) if not e.startswith('__')])

Cls.some_cls_attr = '1'
print([e for e in dir(Cls) if not e.startswith('__')])
```

```
[]
['some_cls_attr']
```

Переменная `__dict__`

- Для большого числа типов в Python пределена переменная-словарь `__dict__`
- Она содержит атрибуты, специфичные для данного объекта (не его класса и не его родителей)
- Множество элементов `__dict__` является подмножеством элементов, возвращаемых функцией `dir()`

In [91]:
```
class A: pass

print(set(A.__dict__.keys()).issubset(set(dir(A))))

[].__dict__
```

```
True

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-91-5da0a70fb2a1> in <module>
      3 print(set(A.__dict__.keys()).issubset(set(dir(A))))
      4
----> 5 [].__dict__

AttributeError: 'list' object has no attribute '__dict__'
```

Доступ к атрибутам

- Для работы с атрибутами есть функции `getattr`, `setattr` и `delattr`
- Их основное преимущество - оперирование именами атрибутов в виде строк

In [13]:
```
cls = Cls()

setattr(cls, 'some_attr', 'some')

print(getattr(cls, 'some_attr'))

delattr(cls, 'some_attr')

print(getattr(cls, 'some_attr'))
```

```
some

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-7a637e97d7b9> in <module>
      7 delattr(cls, 'some_attr')
      8
----> 9 print(getattr(cls, 'some_attr'))

AttributeError: 'Cls' object has no attribute 'some_attr'
```

Class magic methods

- Магические методы придают объекту класса определённые свойства
- Такие методы получают `self` вызываются интерпретатором неявно
- Например, операторы - это магические методы

Рассмотрим несколько примеров:

In [14]:
```
class Cls:
    def __init__(self):  # initialize object
        self.name = 'Some class'

    def __repr__(self):  # str for printing object
        return 'Class: {}'.format(self.name)

    def __call__(self, counter):  # call == operator() in C++
        return self.name * counter

cls = Cls()
print(cls.__repr__())  # == print(cls)
print(cls(2))
```

```
Class: Some class
Some classSome class
```

Class magic methods

Ещё примеры магических методов:

In [ ]:

```
def __lt__(self, other): pass

def __eq__(self, other): pass

def __add__(self, other): pass

def __mul__(self, value): pass

def __int__(self): pass

def __bool__(self): pass

def __hash__(self): pass

def __getitem__(self, index): pass

def __setitem__(self, index, value): pass
```

Как на самом деле устроен доступ к атрибутам

При работе с атрибутами вызываются магические методы `__getattr__`, `__getattribute__`, `__setattr__` и `__delattr__`:

In [69]:
```
class Cls:
    def __setattr__(self, attr, value):
        print(f'Create attr with name "{attr}" and value "{value}"')
        self.__dict__[attr] = value

    def __getattr__(self, attr):
        print(f'WE WILL ENTER IT ONLY IN CASE OF ERROR!')
        return self.__dict__[attr]

    def __getattribute__(self, attr):
        if not attr.startswith('__'):
            print(f'Get value of attr with name "{attr}"')

        return super().__getattribute__(attr)  # call parent method implementation

    def __delattr__(self, attr):
        print(f'Remove attr "{attr}" is impossible!')
```

Как на самом деле устроен доступ к атрибутам

In [70]:
```
cls = Cls()

cls.some_attr = 'some'
a = cls.some_attr

del cls.some_attr
b = cls.some_attr
cls.non_exists_attr
```

```
Create attr with name "some_attr" and value "some"
Get value of attr with name "some_attr"
Delete of attr "some_attr" is impossible!
Get value of attr with name "some_attr"
Get value of attr with name "non_exists_attr"
WE WILL ENTER IT ONLY IN CASE OF ERROR!

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-70-e2e4cd9a9b98> in <module>
      6 del cls.some_attr
      7 b = cls.some_attr
----> 8 cls.non_exists_attr

<ipython-input-69-7f8a91d6c88c> in __getattr__(self, attr)
      6     def __getattr__(self, attr):
      7         print(f'WE WILL ENTER IT ONLY IN CASE OF ERROR!')
----> 8         return self.__dict__[attr]
      9
     10     def __getattribute__(self, attr):

KeyError: 'non_exists_attr'
```

Магические методы и менеджер контекста

Менеджер контекста (оператор `with`) работает с двумя магическими методами:

- `__enter__` - код, который нужно выполнить над объектом при входе в блок менеджера
- `__exit__` - код, который нужно в любом случае выполнить при выходе из блока

In [35]:
```
class SomeDataBaseDao:
    def __init__(self): self._db = ()

    def append(self, value): self._db.append(value)

    def __enter__(self):
        self._db = list(self._db)
        print('Set DB to read-write mode')
        return self

    def __exit__(self, exception_type, exception_val, trace):
        self._db = tuple(self._db)
        print('Set DB to read-only mode')
        return True

dao = SomeDataBaseDao()
#dao.append(1)  # AttributeError: 'tuple' object has no attribute 'append'
with dao:
    dao.append(1)
print(dao._db)
```

```
Set DB to read-write mode
Set DB to read-only mode
(1,)
```

Наследование в Python

In [1]:
```python
class Parent:
    def __init__(self):
        self._value = 10
        self.__value = 20

    def get_value(self):
        return self.__value

class Child(Parent):
    pass

print(Parent().get_value(), Child().get_value())

print(Child().__dict__)
```

```
20 20
{'_value': 10, '_Parent__value': 20}
```

- `__dict__` содержит информацию об атрибутах объекта, атрибутов класса этого объекта (или родителей этого класса) там нет
- При конструировании объекта класса-наследника создаётся один объект, в котором выставляют атрибуты все вызовы `__init__` в иерархии наследования (если они вызывались снизу вверх с помощью `super()`
- Поэтому нет разницы, добавлены были атрибуты в `__init__` родительского класса или класса-наследника - это всё равно атрибуты этого объекта, они будут содержаться в его `__dict__`

Перегрузка родительских методов

In [72]:
```python
class Parent:
    def __init__(self, value):
        self._value = value

    def get_value(self):
        return self._value

    def __str__(self):
        return f'Value: {self._value}'

class Child(Parent):
    def __init__(self, value):
        Parent.__init__(self, value)  # == super().__init__(value)

    def get_value(self):
        return Parent.get_value(self) * 2  # == super().get_value() * 2

print(Parent(10).get_value())
print(Child(10).get_value())
print(Child(10)._value)
print(Child(10))
```

```
10
20
10
Value: 10
```

Интерфейсы

- На уровне языка интерфейсов нет
- Это некритично в силу наличия множественного наследования
- При этом эмулировать интерфейсы - хорошая практика

In [74]:
```
class Interface:
    def get_value(self):
        raise NotImplementedError

class Cls(Interface):
    def __init__(self, value):
        self.value = value

    def get_value(self):
        return self.value

print(Cls(10).get_value())
print(Interface().get_value())  # NotImplementedError
```

```
10
```

Полезная функция isinstance

In [79]:
```
print('isinstance(1, int) == {}'.format(isinstance(1, int)))
print('isinstance(1.0, int) == {}'.format(isinstance(1.0, int)))
print('isinstance(True, int) == {}'.format(isinstance(True, int)))

class Interface:
    def get_value(self):
        raise NotImplementedError

class Cls1(Interface):
    pass

class Cls2(Interface):
    pass

print('isinstance(Cls1(), Cls1) == {}'.format(isinstance(Cls1(), Cls1)))
print('isinstance(Cls1(), Interface) == {}'.format(isinstance(Cls1(), Interface)))
print('isinstance(Cls1(), object) == {}'.format(isinstance(Cls1(), object)))

print('isinstance(Cls2(), Cls1) == {}'.format(isinstance(Cls2(), Cls1)))
```

```
isinstance(1, int) == True
isinstance(1.0, int) == False
isinstance(True, int) == True
isinstance(Cls1(), Cls1) == True
isinstance(Cls1(), Interface) == True
isinstance(Cls1(), object) == True
isinstance(Cls2(), Cls1) == False
```

Полиморфизм

- Полиморфизм позволяет работать с объектами, основываясь только на их интерфейсе, без знания типа
- В C++ требуется, чтобы объекты полиморфных классов имели общего предка
- В Python это не обязательно, достаточно, чтобы объекты поддерживали один интерфейс
- Такое поведение называется duck-typing
- Общий интерфейс в данной ситуации фиксирует протокол взаимодействия

Полиморфизм: пример

In [80]:
```
class Figure:
    def area(self):
        raise NotImplementedError
```

In [81]:
```
class Square(Figure):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2
```

In [82]:
```
import math

class Circle(Figure):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2
```

In [83]:
```
class Triangle(Figure):
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c

    def area(self):
        s = (self.a + self.b + self.c) / 2.0
        return (s *(s - self.a) * (s - self.b) * (s - self.c)) ** 0.5
```

Полиморфизм: пример

Теперь опишем функцию, которая ожидает объекты, реализующие `Figure`:

In [84]:
```
def compute_areas(figures):
    for figure in figures:
        print(figure.area())
```

Можем запускать, не беспокоясь о том, что именно представляют собой входные объекты:

In [85]:
```
s = Square(10)
c = Circle(5)
t = Triangle(1, 3, 3)

compute_areas([s, c, t])
```

```
100
78.53981633974483
1.479019945774904
```

В Python можно обойтись и без наследования Figure, достаточно наличия метода `area` с нужным поведением

Сохранение объектов: модуль pickle

In [3]:
```
class Cls:
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return self.__value
```

In [111]:
```
import pickle

cls = Cls(Cls(10))

with open('cls.pkl', 'wb') as fout:
    pickle.dump(cls, fout)
```

In [4]:
```
with open('cls.pkl', 'rb') as fin:
    cls_2 = pickle.load(fin)
```

In [113]:
```
cls_2.get_value().get_value()
```

Out[113]:
```
10
```

Исключения

- Исключение - механизм, который был придуман штатной обработки ошибочных ситуаций
- Часто ошибочно относится к ООП, на самом деле это иная концепция
- Python поддерживает исключения, и ими надо пользоваться
- В языке есть большая иерархия классов исключений на все случаи жизни
- Если нужен свой класс, то можно наследовать от какого-то из существующих
- Оптимальный вариант - класс `Exception`

Базовый синтаксис

In [5]:
```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-5-bc757c3fda29> in <module>
----> 1 1 / 0

ZeroDivisionError: division by zero
```

In [6]:
```
try:
    1 / 0
except:
    print('Zero division!')
```

```
Zero division!
```

In [7]:
```
try:
    raise ZeroDivisionError
except:
    print('Zero division!')
```

```
Zero division!
```

Полный синтаксис

In [8]:
```
try:
    # some code
    pass

except ValueError:  # catch value errors
    print('ERROR')  # do something
    raise           # continue rising of this exception (or can skip it)

except RuntimeError as error:  # catch runtume errors and store object
    print(error)               # inspect exception content
    raise error                # continue rising

except:             # try not to use except without class specification
    print('Unknown error')
    pass

else:            # if there's no exception, execute this branch
    print('OK')
finally:         # actions that chould be done in any case
    #some actions (closing files for instance)
    print('finally')
    pass
```

```
OK
finally
```

Спасибо за внимание!