

Практикум по программированию на языке Python

Занятие 5: Продвинутое использование ООП, проектирование кода

Мурат Апишев (mel-lain@yandex.ru)

Москва, 2020

Напоминание: 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

Напоминание: интерфейс и абстрактный класс

- **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
- **Абстрактный класс** - это промежуточное состояние между интерфейсом и обычным классом

Виды отношений между классами

- **Наследование** - класс наследует класс
- **Реализация** - класс реализует интерфейс
- **Ассоциация** - горизонтальная связь между объектами двух классов (может быть "один ко многим")
- **Композиция** - вложенность объекта одного класса в другой (главный управляет жизненным циклом зависимого)
- **Агрегация** - вложенность объекта одного класса в другой (объекты остаются независимыми)

Объектно-ориентированное проектирование

- Проектирование - определение наборов интерфейсов, классов, функций, их свойств и взаимных отношений
- Система для решения одной и той же задачи может спроектирована многими способами
- Задача в том, чтобы спроектировать систему, которая будет
 - понятной, поддерживаемой
 - избыточной
 - несложно модифицируемой и расширяемой
 - эффективной
- Для этого нужны собственный опыт, знания, основанные на опыте предшественников и владение возможностями языка

Набор правил SOLID

- **Принцип единственной ответственности** - объект класса отвечает за одну и только одну задачу
- **Принцип открытости/закрытости** - класс должен быть открыт для расширения функциональности и закрыт для изменения
- **Принцип подстановки Барбары Лисков** - наследник класса должен только дополнять родительский класс, а не менять в нём что-либо
- **Принцип разделения интерфейса** - интерфейсы не должны покрывать много задач (== класс не должен реализовывать методы, которые ему не нужны из-за слишком большого базового интерфейса)
- **Принцип инверсии зависимостей** - нижестоящее в иерархии зависит от вышестоящего, а не наоборот

Принцип подстановки Барбары Лисков (LSP)

- Принцип не означает, что не нужно использовать перегрузку виртуальных методов
- Требуется, чтобы код наследника:
 - не изменял состояния родительского класса
 - не расширял предусловия
 - не сужал постусловия

```
In [1]: class Base:
        def method(self, value):
            if not isinstance(value, int):
                raise Exception
            return abs(value)

        class Deriv(Base):
            def method(self, value):
                if not isinstance(value, int) and value < 0:
                    raise Exception # 1st error (more pre-conditions)
                return value # 2st error (less post-conditions)
```

Принцип инверсии зависимостей

- Принцип состоит из двух пунктов:
 - Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций
 - Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций
- Главное, что стоит вынести отсюда:
 - нужно программировать на уровне интерфейсов
 - нужно избавляться от вложенных зависимостей
- Все принципы связаны между собой и нарушение одного часто приводит к нарушению сразу нескольких
- Рассмотрим ниже пример (источник: <https://blog.byndyu.ru/2009/12/blog-post.html> (<https://blog.byndyu.ru/2009/12/blog-post.html>))

Пример: отправитель отчётов

```
In [2]: class Reporter:
        def send_reports():
            report_builder = ReportBuilder()
            reports = report_builder.create_reports()

            if len(reports) == 0:
                raise Exception

            report_sender = EmailReportSender()
            for report in reports:
                report_sender.send(report)
```

Какие есть проблемы?

Пример: отправитель отчётов

Проблем в таком коде полно:

- сложность тестирования: как проверить поведение `Reporter`, которое зависит от других классов?
- высокая связность, `Reporter`
 - требует, чтобы отчёты выдавал именно `ReportBuilder`
 - требует, чтобы отправление производил именно `EmailReportSender`
 - создаёт объекты обоих классов внутри себя
- Нарушаются сразу три принципа:
 - принцип единственной ответственности (класс должен просто отправлять отчёты)
 - принцип открытости/закрытости (для отправки отчётов не по e-mail нам потребуется исправлять `Reporter`)
 - принцип инверсии зависимостей (классы жёстко зависят друг от друга)

Пример: отправитель отчётов

Изменим код так, чтобы исправить эти проблемы:

```
In [ ]: class ReportBuilderInterface:
        def create_reports(self):
            raise NotImplementedError

        class ReportSenderInterface:
            def send(self, report):
                raise NotImplementedError

        class Reporter:
            def __init__(self, report_builder, report_sender):
                self.report_builder = report_builder
                self.report_sender = report_sender

            def send_reports():
                reports = report_builder.create_reports()

                if len(reports) == 0:
                    raise Exception

                for report in reports:
                    report_sender.send(report)
```

Полезно для проектирования: множественное наследование

- Даёт возможность классу получить методы и свойства сразу нескольких предков
- Позволяет строить сложные иерархии зависимостей
- Использовать без необходимости не нужно, поскольку архитектура кода сильно усложняется
- В Python поддерживается без ограничений
- У класса может быть один и более предков (object есть всегда)

Множественное наследование в Python

- Методы и атрибуты ищутся в следующем порядке:
 1. имя ищется в объекте (т.е. в его `__dict__`)
 2. дальше в классе объекта
 3. дальше в предках класса
- Для поиска по предкам используется MRO (Method resolution order)
- У каждого класса в момент создания вычисляется этот порядок и сохраняется в атрибуте `__mro__`

```
In [12]: class A():
          def method(self): return 'A'

          class B():
              def method(self): return 'B'

          class C(A, B): pass

          print(C.__mro__)
          print(C().method())
```

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
A
```

Вызов метода класса-родителя

- В Python есть два способа обращения к родительским методам
 - через функцию `super` (использует порядок mro)
 - напрямую по имени

```
In [37]: class A():
          def method(self): return 'A'

          class B():
              def method(self): return 'B'

          class C(A, B):
              def method(self):
                  return (A.method(self), B.method(self), super().method())

          print(C.__mro__)
          print(C().method())
```

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
('A', 'B', 'A')
```

Полезное для проектирования: статические методы

- Обычные методы класса принимают на вход первым параметром вызывающий объект и работают с ним
- Статические методы являются методами класса и к объекту не привязаны
- Как следствие, их можно вызывать без объекта

```
In [26]: class A:
          def method(self):
              print('Regular method')

          @staticmethod # example of decorator
          def static_method():
              print('Static method')

# A.method() # method() missing 1 required positional argument: 'self'

A().method()
A.static_method()
A().static_method() # object also has his type methods
A.method(A())
```

```
Regular method
Static method
Static method
Regular method
```

Шаблоны 00-проектирования

- Набор практик и рекомендаций по организации кода, полученных опытным путём
- Оптимизируют решения типовых задач, упрощают разработку и поддержание кода
- Существуют десятки шаблонов, кратко рассмотрим несколько основных:
 - Singleton
 - Mixin
 - Фасад
 - Адаптер, DAO
 - Простая фабрика
 - Фабричный метод
 - Абстрактная фабрика
 - Декоратор

Шаблон Singleton

- Иногда возникает необходимость создать класс, который может иметь не более одного экземпляра (например, глобальный конфиг)
- Есть несколько способов реализации такого поведения

Один из вариантов:

```
In [40]: class Singleton:
          def __new__(cls): # is called before init to create object
              if not hasattr(cls, 'instance'):
                  cls.instance = super().__new__(cls)
              return cls.instance
```

```
In [44]: s = Singleton()
          s.some_attr = 'some_attr'

          s_1 = Singleton()
          s_1.some_attr
```

```
Out[44]: 'some_attr'
```

Singleton также удобно реализовывать с помощью статического метода

Шаблон Mixin

- Mixin - шаблон, основанный на множественном наследовании
- Используется в одном из двух основных случаев:
 - хочется иметь в одном классе много опциональных фичей
 - хочется использовать одну и ту же фичу во многих классах
- Реализуется в виде класса, содержащего необходимую функциональность
- Далее этот класс наследуется любым нужным классом, умеющим эту функциональность использовать

Пример использования Mixin

```
In [45]: class VersionMixin:
          def set_version(self, version):
              self._version = version

          @property # One more example of decorator
          def version(self):
              return self._version if hasattr(self, '_version') else 'Unk. version'

          class SomeBaseClass:
              def __init__(self, value):
                  self.value = value

          class SomeClass(SomeBaseClass, VersionMixin):
              def __init__(self, value):
                  super().__init__(value)

          sc = SomeClass(10)
          print(sc.value)
          sc.set_version(1.0)
          print(sc.version)
```

10

1.0

Шаблон Фасад

Предназначен для сокрытия сложного поведения за простым внешним интерфейсом

```
In [48]: class ATMImpl:
          def check_pin(self, pin): pass
          def make_db_request(self, user): pass
          def check_money(self, user, amount): pass
          def send_money(self, amount): pass
          ...
```

```
In [ ]: class ATM:
          def __init__(self, atm_impl):
              self.atm_impl = atm_impl

          def get_money(self, pin, user, amount):
              pass # use self.atm_impl methods to proceed operation
```

Шаблон Адаптер

- Предназначен для соединения двух систем, которые могут работать вместе, но имеют несовместимые интерфейсы
- Особенно полезен при работе с внешними API

```
In [54]: class ValueProvider:
    def __init__(self, value):
        if not isinstance(value, int):
            raise ValueError
        self.__value = value

    def get_value(self):
        return self.__value

class ValueUser:
    def use_value(self, value_with_name):
        if not isinstance(value_with_name, tuple):
            raise ValueError

        print(value_with_name[0])
```

```
In [55]: class ValueAdapter:
    def transform_value(self, value):
        return (value, '')

vp = ValueProvider(10)
va = ValueAdapter()
vu = ValueUser()

vu.use_value(va.transform_value(vp.get_value()))
```

Шаблон Data Access Object

- Предоставление единого интерфейса доступа к источнику данных (как правило, к БД)
- DAO скрывает детали утавления соединений, отсылки и обработки результатов запросов и т.п.
- По сути, это адаптер между клиентским приложением и базой данных

```
In [ ]: class DbDao:
        def request(self, query): raise NotImplementedError

class DbOneDao(DbDao):
    def __init__(self, db_one_config):
        self.__db_one_impl = DbOne(db_one_config)

    def request(self, query):
        return self.__db_one_impl.request(query)

class DbTwoDao(DbDao):
    def __init__(self, db_two_config):
        self.__db_two_impl = DbTwo(db_two_config)

    def request(self, query):
        return self.__db_two_impl.proceed_request(query)
```

Фабрики

- Фабрика - это общая идея создания объекта с помощью какой-то другой сущности
- Создавать объекты могут
 - обычные функции
 - методы этого же класса (в т.ч. и статические)
 - методы других классов
- При этом в зависимости от ситуации могут использоваться как примитивные решения, так и сложные фабрики
- Рассмотрим ниже три основных фабричных шаблона:
 - простая фабрика
 - фабричный метод
 - абстрактная фабрика

Шаблон Простая фабрика

- Есть объект класса `VendingMachine`, описывающий торговый автомат
- Объект имеет метод `get_item`, который по имени товара возвращает объект этого товара

```
In [1]: class VendingMachine:
        def get_item(self, item):
            if item == 'cola':
                return Cola() # skip ammount checks
            elif item == 'chocolate':
                return Chocolate()
            else:
                raise ValueError('Unknown item')
```

- Создаваемые продукты могут быть никак не связаны друг с другом
- Класс `VendingMachine` является простой фабрикой

Шаблон Фабричный метод

- Есть объект класса `CottageBuilder`, который производит объекты-коттеджи `Cottage`
- Весь наш программный комплекс заточен под обработку производства коттеджей
- В некоторый момент было принято решение начать производство многоэтажных домов (`MultiStoreyBuilding`)
- Неправильная архитектура приведёт к необходимости переписывания кода во многих местах
- Фабричный метод позволяет избежать этого

Шаблон Фабричный метод: общая схема

- Имеется класс-производитель ConcreteFactory и класс-продукт ConcreteProduct
- Вводим интерфейс производителя FactoryInterface и интерфейс продукта ProductInterface
- Каждый конкретный производитель получает возможность создавать конкретный продукт с помощью одного и того же метода

Для нашего примера код выглядит так:

```
In [5]: class BuilderInterface:
        def build(self):
            raise NotImplementedError

        class BuildingInterface:
            def open_door(self, door_number): raise NotImplementedError
            def open_window(self, window_number): raise NotImplementedError
            ...
```

```
In [8]: class CottageBuilder(BuilderInterface):
        def build(self):
            return Cottage()

        class MultiStoreyBuilder(BuilderInterface):
            def build(self):
                return MultiStoreyBuilding()
```

Шаблон Фабричный метод: общая схема

Осталось описать код классов продуктов (то есть зданий):

```
In [6]: class Cottage(BuildingInterface):  
        def open_door(self, door_number): self.__open_door(door_number)  
        def open_window(self, window_number): self.__open_window(window_number)  
        ...
```

```
In [7]: class MultiStoreyBuilder(BuildingInterface):  
        def open_door(self, door_number): self.__open_door(door_number)  
        def open_window(self, window_number): self.__open_window(window_number)  
        ...
```

- теперь в коде нет разницы, с каким зданием мы работаем, интерфейс у них всех общий
- тип здания, который нам надо построить, определяется только внутри конкретного строителя

Шаблон Абстрактная фабрика

- Шаблон предназначен для создания систем взаимосвязанных объектов без указания их конкретных классов
- Проще всего разобрать на примере (источник: <https://refactoring.guru/ru/design-patterns/abstract-factory> (<https://refactoring.guru/ru/design-patterns/abstract-factory>))
- Задача:
 - есть набор одних и тех же элементов графического интерфейса (CheckBox, Button, TextField и прочие)
 - в каждой операционной системе должны отображаться все эти элементы
 - при этом в каждой OS собственный стиль отображения
- Опишем использование абстрактной фабрики для такого случая

Интерфейсы фабрик и продуктов

```
In [9]: class AbstractGui: # more declarative than necessary, remember about duck-typing
        def create_check_box(self): raise NotImplementedError
        def create_button(self): raise NotImplementedError
        def create_text_field(self): raise NotImplementedError
        ...
```

```
In [10]: class AbstractCheckBox:
        def set_state(self): raise NotImplementedError
        ...
```

```
In [11]: class AbstractButton:
        def on_press(self): raise NotImplementedError
        ...
```

```
In [12]: class AbstractTextField:
        def is_empty(self): raise NotImplementedError
        ...
```

Конкретные классы фабрики и продуктов

```
In [13]: class WindowsGui(AbstractGui):  
        def create_check_box(self): return WindowsCheckBox()  
        def create_button(self): return WindowsButton()  
        def create_text_field(self): return WindowsTextField()  
        ...
```

```
In [14]: class WindowsCheckBox(CheckBox):  
        def set_state(self): self.__get_state_impl()  
        ...
```

```
In [15]: class WindowsAbstractButton(AbstractButton):  
        def on_press(self): self.__on_press_impl()  
        ...
```

```
In [16]: class WindowsAbstractTextField(AbstractTextField):  
        def is_empty(self): self.__is_empty_impl()  
        ...
```

Аналогично определим классы для Mac OS и Ubuntu (Unity)

Использование абстрактной фабрики

```
In [ ]: class GuiApplication:
    def __init__(self):
        import sys

        self.gui = None
        if sys.platform == 'win32':
            self.gui = WindowsGui()
        elif sys.platform == 'linux':
            self.gui = LinuxGui()
        elif sys.platform == 'darwin':
            self.gui = MacOSGui()
        else:
            raise SystemError(f'Unsupported OS type {sys.platform}')

    def draw_window(self):
        # write cross-platform code, based on interface, not implementations
        self.gui.create_button()
        self.gui.create_check_box()
        self.gui.create_text_field()
        ...
```

Шаблон Декоратор

- Шаблон проектирования, позволяющий динамически наделить объект дополнительными свойствами
- Каноническая (но не единственная) реализация включает в себя интерфейс, реализующие его объекты-компоненты и объекты-декораторы
- Задача из жизни: сбор набора метрик для отправки в сервис мониторинга
 - собирается несколько сотен различных метрик
 - метрики могут агрегироваться несколькими способами (mean/median/mode/max/min по интервалу)
 - метрики могут преобразовываться несколькими способами (положительная срезка, фильтрация значений)
- Модификаторы могут применяться в произвольных сочетаниях, количествах и порядках
- Как можно реализовать подобную систему без необходимости расписывать кучу условий?

Декоратор идеален для такой задачи

Опишем интерфейс метрики и реализующие его классы:

```
In [24]: class MetricInterface(): # again for Python it's unnecessary declaration
         def calculate(self): raise NotImplementedError
```

```
In [25]: class MetricA(MetricInterface):
         def calculate(self):
             self.values = self.__calculate_a() # returns list
```

```
In [26]: class MetricB(MetricInterface):
         def calculate(self):
             self.values = self.__calculate_b() # returns list
```

Собственно декораторы

Теперь опишем классы операций над метриками, которые тоже реализуют интерфейс метрики:

```
In [27]: class CalculateMeanDecorator(MetricInterface):  
    def __init__(self, metric):  
        self.metric = metric  
  
    def calculate(self):  
        self.values = [self.__calculate_mean(self.metric.calculate())]  #[scalar] == list
```

```
In [28]: class FilterZerosDecorator(MetricInterface):  
    def __init__(self, metric):  
        self.metric = metric  
  
    def calculate(self):  
        self.values = list(filter(lambda x: x != 0.0, self.metric.calculate()))
```

Использование декоратора

Теперь можно легко создавать метрики и тут же снабжать их нужными свойствами:

```
In [ ]: metric_1 = MetricA()
metric_2 = MetricB()
metric_3 = FilterZerosDecorator(MetricA())
metric_4 = CalculateMeanDecorator(FilterZerosDecorator(MetricB()))

for metric in [metric_1, metric_2, metric_3, metric_4]:
    metric.calculate()
    MetricSender.send(metric.values)
```

Декораторы в общем случае

- В силу популярности и полезности декораторы поддерживаются рядом языков на уровне синтаксиса (Python, Java)
- На самом деле декоратором является любой объект, который "оборачивается" вокруг других объектов, меняя их свойства и/или поведение
- В Python это особенно просто за счёт duck-typing
- На уровне языка поддерживаются декораторы функций (методов) и декораторы классов
- Этот тот случай, когда нужно вспомнить о замыканиях (функциях, возвращающих функции)

Пример декоратора функции

Опишем декоратор, замеряющий время работы функции:

```
In [35]: def timed(callable_obj):
import time

def __timed(*args, **kw):
    time_start = time.time()
    result = callable_obj(*args, **kw)
    time_end = time.time()

    print('{} {:.3f} ms'.format(callable_obj.__name__,
                                (time_end - time_start) * 1000))

    return result

return __timed
```

```
In [36]: @timed
def func():
    for i in range(10000000):
        pass

func()
```

func 33.965 ms

Пример декоратора функции

Посмотрим, что это за функция на самом деле:

```
In [37]: import inspect
lines = inspect.getsource(func)
print(lines)

def __timed(*args, **kw):
    time_start = time.time()
    result = callable_obj(*args, **kw)
    time_end = time.time()

    print('{} {:.3f} ms'.format(callable_obj.__name__,
                               (time_end - time_start) * 1000))
    return result
```

Пример декоратора класса

Опишем декоратор класса, который получает на вход декоратор функции и оборачивает его вокруг каждого публичного метода класса:

```
In [57]: def decorate_class(decorator):  
        def __decorate(cls):  
            for f in cls.__dict__:  
                if callable(getattr(cls, f)) and not f.startswith("_"):  
                    setattr(cls, f, decorator(getattr(cls, f)))  
            return cls  
        return __decorate
```

```
In [60]: @decorate_class(timed)  
class Cls:  
    a = 10  
    def method(self): pass  
    def _method_2(self): pass  
  
Cls().method()  
Cls().a # not callable  
Cls()._method_2() # not public
```

method 0.001 ms

При определении класса он производится вызов `decorate_class.__decorate`, которая возвращает обновлённый класс.

Мета-классы в Python

- Вспомним, что в Python всё, включая классы, является объектами
- Это позволяет создавать классы и менять их свойства динамически
- Описание класса определяет свойства объекта
- Описание мета-класса определяет свойства класса
- В Python есть один класс, не являющийся объектом - это мета-класс `type`
- Мета-программирование - ещё один способ динамического изменения свойств классов (помимо декораторов)

Мета-классы в Python

`type` можно использовать напрямую для создания классов:

```
In [62]: Class = type('MyClass', (object, ), {'field': lambda self: 'value'})
          c = Class()

          print(type(c))
          print(c.field())
```

```
<class '__main__.MyClass'>
value
```

Мета-классы в Python

- А можно на основе `type` описывать мета-классы
- Напишем простой мета-класс, который добавляет метод `hello` в создаваемый класс
- Источник примера: <https://gitjournal.tech/metaklassy-i-metaprogrammirovanie-v-python/>
(<https://gitjournal.tech/metaklassy-i-metaprogrammirovanie-v-python/>)

```
In [60]: class HelloMeta(type): # always inherit type or it's childs
    def hello(cls):
        print("greetings from %s, a HelloMeta type class" % (type(cls())))

    # call meta-class
    def __call__(self, *args, **kwargs):
        cls = type.__call__(self, *args, **kwargs) # create class as usual

        setattr(cls, "hello", self.hello) # add 'hello' attribute

        return cls

class TryHello(object, metaclass=HelloMeta):
    def greet(self):
        self.hello()

greeter = TryHello()
greeter.greet()
```

```
greetings from <class '__main__.TryHello'>, a HelloMeta type class
```

Зачем нужно работать с мета-классами?

- На самом деле, это не нужно почти никогда, лучше использовать декораторы
- Бывает полезно в тех случаях, когда нужно штамповать классы с заданными свойствами или при отсутствии информации о деталях класса до момента выполнения кода
- **Пример:**
 - вы определили класс данных с методами обработки, зависящими от формата
 - класс имеет метакласс, который определяется аргументами программы
 - в аргументах передаются различные форматы и методы обработки
 - мета-класс в зависимости от них переопределяет методы вашего класса
- **Пример:** генерация API, в текущем контексте может требоваться, чтобы все методы ваших классов были в верхнем регистре

Спасибо за внимание!