Практика 6. Объектно-ориентированное программирование

материалы преподавателя

Объектно-ориентированное программирование

Объектно-ориентированное программирование — парадигма программирования, которая позволяет объединять данные и алгоритмы для их обработки в единую сущность — объект и обращаться с ним, как с любым другим значением.

Например, рассмотри класс `Student`. Каждый знакомый тебе студент — это **экземпляр** такого класса. То есть студент `Anna` — экземпляр класса `Student`.

Информация о студенте, например, имя `name`, год поступления `year`, номер студенческого `id` — это **атрибуты экземпляра**. Каждый студент имеет эти атрибуты, но их значения для каждого студента индивидуальны. Атрибуты, являющиеся едиными для всех студентов называют **атрибутами класса**. Например, атрибут `university`, будет одинаковым для всех студентов университета.

Все студенты могут делать домашнюю работу, ходить на лекции и так далее. В случае, если функция принадлежит конкретному объекту, её называют **метод**. Указанные действия можно смоделировать методами класса `Student`.

Краткое резюме:

класс
: шаблон для создания объектов;

экземпляр
: отдельный объект, созданный из класса;

атрибут экземпляра
: свойство объекта с разными значениями для разных экземпляров;

атрибут класса
: свойство объекта с общим значением для всех экземпляров;

метод
: действие (функция), доступное всем экземплярам класса.

Вопрос 1

Ниже определены два класса — `Professor` и `Student`. Помни, что в методы класса неявно передается `self` при использовании точечной нотации.

```
class Student:
    students = 0                          # это атрибут класса

    def __init__(self, name, teacher):
        self.name = name                  # это атрибут экземпляра
        self.understanding = 0
        Student.students += 1
        print("Теперь студентов:", Student.students)
        teacher.add_student(self)

    def visit_lab(self, staff):
        staff.assist(self)
        print("Спасибо, " + staff.name)

class Professor:
    def __init__(self, name):
        self.name = name
        self.students = {}

    def add_student(self, student):
        self.students[student.name] = student

    def assist(self, student):
        student.understanding += 1
```

Что получится в результате?

```
>>> snape = Professor("Снегг")
>>> harry = Student("Гарри", snape)
```

Ответ

```
Теперь студентов: 1
```

```
>>> harry.visit_lab(snape)
```

Ответ

```
Спасибо, Снегг
```

```
>>> harry.visit_lab(Professor("Хагрид"))
```

Ответ

```
Спасибо, Хагрид
```

```
>>> harry.understanding
```

Ответ

```
2
```

```
>>> [name for name in snape.students]
```

Ответ

```
['Гарри']
```

```
>>> x = Student("Гермиона", Professor("Макгонагалл")).name
```

Ответ

```
Теперь студентов: 2
```

```
>>> x
```

Ответ

```
'Гермиона'
```

```
>>> [name for name in snape.students]
```

Ответ

```
['Гарри']
```

Вопрос 2

Теперь нужно написать три различных класса `Server`, `Client`, `Email` для эмуляции отправки электронных писем. Заполни определения данные ниже, чтобы всё уже заработало!

рекомендуется сначала дополнить класс `Email`, затем метод `register client` класса `Server`, потом доделать класс `Client`, и, наконец, метод `send` класса `Server`.

```
class Email:
    """Каждый экземпляр email имеет 3 атрибута уровня экземпляра:
    сообщение msg, имя отправителя `sender_name` и имя получателя `recipient_name`.
    """
    def __init__(self, msg, sender_name, recipient_name):
```

Ответ

```
        self.msg = msg
        self.sender_name = sender_name
        self.recipient_name = recipient_name
```

```
class Server:
    """У каждого экземпляра есть атрибут clients — словарь, связывающий имена
    клиентов с объектами.
    """
    def __init__(self):
        self.clients = {}

    def send(self, email):
        """Направляет email во входящие сообщения клиента, которому он адресован.
        """
```

Ответ

```
        client = self.clients[email.recipient_name]
        client.receive(email)
```

```
    def register_client(self, client, client_name):
        """Принимает объект client и имя client_name и добавляет их в атрибут clients.
        """
```

Ответ

```
        self.clients[client_name] = client
```

```
class Client:
    """У каждого клиента есть аттрибуты:
    name (который используется для направления письма),
    server (который используется для отправления писем другим клиентам ) и
    inbox (список всех полученных писем). """
    def __init__(self, server, name):
        self.inbox = []
```

Ответ

```
        self.server = server
        self.name = name
        self.server.register_client(self, self.name)
```

```
    def compose(self, msg, recipient_name):
        """Отправляет email с сообщением msg заданному адресату recipient_name."""
```

Ответ

```
        email = Email(msg, self.name, recipient_name)
        self.server.send(email)
```

```
    def receive(self, email):
        """Принимает email и добавляет его в inbox клиента."""
```

Ответ

```
        self.inbox.append(email)
```

Наследование

Классы Python поддерживают полезную технику абстрагирования — **наследование**.

Для усвоения этой техники рассмотри классы `Cat` и `Dog`.

```
class Dog():
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner

    def eat(self, thing):
        print(self.name + " съел " + str(thing) + "!")

    def talk(self):
        print(self.name + " говорит «гав»!")

class Cat():
    def __init__(self, name, owner, lives=9):
        self.is_alive = True
        self.name = name
        self.owner = owner
        self.lives = lives

    def eat(self, thing):
        print(self.name + " съел " + str(thing) + "!")

    def talk(self):
        print(self.name + " говорит «мяу-у-у-у»!")
```

Видно, что собаки и кошки сильно похожи — очень много кода повторяется! Чтобы избежать повторного задания атрибутов и методов, можно создать **суперкласс** (надкласс), которому будут *наследовать* классы `Cat` и `Dog`. Например, посмотри на определение класса `Pet` и наследующего ему класса `Dog`.

```
class Pet():
    def __init__(self, name, owner):
        self.is_alive = True    # Питомец живой!!!
        self.name = name
        self.owner = owner

    def eat(self, thing):
        print(self.name + " съел " + str(thing) + "!")

    def talk(self):
        print(self.name)

class Dog(Pet):
    def talk(self):
        print(self.name + ' говорит «гав»!')
```

Наследование создаёт иерархическую связь между двумя или более классами, в которой один класс *является* более специфицированной версией другого класса. То есть можно сказать, что класс `Dog` является уточнённым вариантом класса `Pet`. Поскольку класс `Dog` наследует классу `Pet`, нет необходимости переопределять в нём, например, методы __init__ или `eat`. Однако, поскольку требуется, чтобы `Dog` говорил по-собачьи, метод `talk` **переопределяется**.

Вопрос 3

Ниже представлена заготовка для класса `Cat`, наследующая классу `Pet`. Дополни код — переопредели методы __init__ и `talk` и добавь новый метод `lose_life`.

можно вызвать метод __init__ класса `Pet`, чтобы запомнить в атрибутах имя питомца и владельца.

```
class Cat(Pet):
    def __init__(self, name, owner, lives=9):
```

Ответ

```
        Pet.__init__(self, name, owner)
        self.lives = lives
```

```
    def talk(self):
        """ Выводит сообщение котана.
        >>> Cat('Живоглот', 'Гермиона').talk()
        Живоглот говорит «мяу-у-у-у»!
        """
```

Ответ

```
        print(self.name + " говорит «мяу-у-у-у»!")
```

```
    def lose_life(self):
        """Уменьшает количество кошачьих жизней.
        При достижении нуля изменяет 'is_alive' на False.
        """
```

Ответ

```
        if self.lives > 0:
            self.lives -= 1
            if self.lives == 0:
                self.is_alive = False
        else:
            print("У этого котана истрачены все жизни :(")
```

Вопрос 4

Ещё больше котиков! Дополни определение класса `NoisyCat`, который похож на `Cat`, но более говорливый. `NoisyCat` говорит в два раза больше, чем обычный `Cat`.

```
class ______________________________:
    """Кот, повторяющий всё дважды"""

    def __init__(self, name, owner, lives=9):
        # А этот метод нужен? Почему?


        ----------------------------------------

    def talk(self):
        """Говорит всё по два раза.

        >>> NoisyCat('Живоглот', 'Гермиона').talk()
        Живоглот говорит «мяу-у-у-у»!
        Живоглот говорит «мяу-у-у-у»!
        """


        ----------------------------


        ----------------------------
```

Ответ

```
class NoisyCat(Cat):
    """Кот, повторяющий всё дважды"""

    def __init__(self, name, owner, lives=9):
        # А этот метод не нужен. NoisyCat и так наследует метод
        # __init__ класса Cat.

        Cat.__init__(self, name, owner, lives)

    def talk(self):
        """Говорит всё по два раза.

        >>> NoisyCat('Живоглот', 'Гермиона').talk()
        Живоглот говорит «мяу-у-у-у»!
        Живоглот говорит «мяу-у-у-у»!
        """
        Cat.talk(self)
        Cat.talk(self)
```

Дополнительные вопросы

Вопрос 5

Что выведет интерпретатор?

```
class A:

    def f(self):
        return 2

    def g(self, obj, x):
        if x == 0:
            return A.f(obj)
        return obj.f() + self.g(self, x - 1)

class B(A):
    def f(self):
        return 4
```

```
>>> x, y = A(), B()
>>> x.f()
```

Ответ

```
2
```

```
>>> B.f()
```

Ответ

```
Ошибка (отсутствует аргумент self)
```

```
>>> x.g(x, 1)
```

Ответ

```
4
```

```
>>> y.g(x, 2)
```

Ответ

```
8
```

Вопрос 6

Создай класс `Foo`, чтобы он вёл себя как показано ниже.

```
>>> x = Foo(1)
>>> x.g(3)
4
>>> x.g(5)
6
>>> x.bar = 5
>>> x.g(5)
10
```

```
class Foo:
```

Ответ

```
    def __init__(self, bar):
        self.bar = bar
    def g(self, n):
        return self.bar + n
```