GeekBrains

ООП

Узнаем, как создавать собственные классы и их объекты. Разберём 3 основных термина: наследование, инкапсуляция и полиморфизм.
Изучим @staticmethod и @classmethod. Поговорим про свойства @property, сеттеры и геттеры. Изучим множественное наследование и миксины.

Классы и объекты

Собственные классы

Мы с вами уже пользовались классами и создаваемыми объектами для конкретных задач. Но что если стандартных классов недостаточно? В таком случае мы можем создать свои.

Класс — это конструкция, шаблонный код, на основании которого создаются объекты, обладающие разными свойствами, но схожими возможностями

Объект — это переменная, которая является конкретной реализацией определённого одного класса. Объект обладает определёнными свойствами (возможно, изменяемыми) и заранее заданным поведением

```
1 class People:          # Класс People описывающий параметры всех людей
2     age = 0
3     name = None
4     birthday = None
5
6
7 human = People()       # Объект human. Конретный человек от People
8 human2 = People()      # Объект human2. Конретный человек от People
```

Свойства классов

При описании классов мы можем определить, какими параметрами будут обладать все будущие объекты. Для этого нам необходимо просто описать их внутри. Все описанные свойства будут создаваться у каждого объекта. Чтобы прочитать их или изменить, нам нужно обратиться к ним через точку:

```
class Lamp:
    state = False
    model = None
    warranty = 0


lamp = Lamp()

lamp.model = 'Phillips'     # Устанавливаем модель
lamp.warranty = 5           # Устанавливаем гарантию

print(lamp.model, lamp.warranty)    # Читаем параметры
```

Методы классов

Методы — это функции, описывающие действия, которые может совершать объект. Методы также создаются внутри класса и потом вызываются у конкретных объектов:

```
class Lamp:
    state = False
    model = None
    warranty = 0

    def turn_light(self):
        if not self.state:
            self.state = True
            print("Light is on")
        else:
            self.state = False
            print("Light is off")


lamp = Lamp()

lamp.turn_light()
```

Особенности работы со свойствами

При чтении свойства у объекта мы обращаемся к нему через точку. Но если необходимо поработать со свойством внутри, то обращаться к нему обязательно нужно через self:

```
self.state = True
```

Иначе интерпретатор будет воспринимать данную команду как создание новой локальной переменной.

Также различаются места определения свойств. Общие свойства, которые безусловно будут присутствовать у каждого объекта с определённым заданным значением, создаются в самом начале класса. Причём self здесь не используется.

Если мы пользуемся созданием свойств в конструкторе класса (о конструкторе поговорим через минуту), то свойства уже создаются через self и в них можно записать значение, которое будет передано снаружи.

Особенности работы с методами

При создании метода внутри класса обязательно в скобках на первом месте указывать **self**. Это даст понимание, что данная функция (по факту, — функция) относится именно к классу:

```
def turn_light(self)
```

Если нужно, дальше через запятую после self указываем принимаемые методом параметры.
При вызове метода мы должны поставить в конце скобки так же, как и при вызове функции:

```
lamp.turn_light()
```

Причём **self** здесь уже не указывается.

Конструктор

При создании объекта **всегда один раз** вызывается функция внутри класса, именуемая конструктором. Даже если мы не создаём этот метод, он все равно по умолчанию присутствует у каждого класса.

Но ценность конструктора не в этом. А в том, что мы можем уже на этапе создания объекта задать любые свойства, и нам не придется определять их после:

```
1 class Lamp:
2     state = False
3 
4     def __init__(self, model, warranty):  # конструктор с 2 пар-ми
5         self.model = model
6         self.warranty = warranty
7 
8 lamp = Lamp("Phillips", 5)         # обязательно указываем оба пар-ра
```

Деструктор

Ещё один метод, который присутствует у каждого класса и вызывается один раз — при уничтожении объекта — называется деструктором. В него тоже можно заложить определённую логику, если нам необходимо реагировать на уничтожение объекта:

```
class Lamp:
    state = False

    def __init__(self, model, warranty):
        self.model = model
        self.warranty = warranty

    def __del__(self):                  # Деструктор
        print("Lamp recycled")
```

Инкапсуляция

Что такое инкапсуляция

Здесь должен быть умный термин, списанный из первой ссылки Google. Давайте будем проще :)

Инкапсуляция — это механизм, который позволяет сделать невозможным чтение свойств и вызов методов у объекта. Это делают в первую очередь для безопасности.

Представьте, что вы могли бы изменять баланс своей карты, просто меняя свойство объекта «кошелёк».

Чтобы сделать метод или свойство приватным, то есть инкапсулировать его, необходимо в начале поставить символ __:

```
1 class Lamp:
2     __state = False           # Приватное свойство
3
4
5 lamp = Lamp("Phillips", 5)
6 print(lamp.state)              # Ошибка
7 print(lamp.__state)            # И даже так ошибка
```

Наследование

Как работает наследование

Мы можем пойти дальше и создать ещё более сложную структуру классов и их объектов благодаря наследованию.

Наследование — это механизм для создания дочерних классов на основании родительских. Причём дочерние классы будут иметь доступ к методам и свойствам родительского класса. Данные методы и свойства можно либо унаследовать полностью, либо переопределить их поведение, создав новые возможности.

Чтобы создать дочерний класс, необходимо в скобках указать родительский:

```
1  class Lamp:
2      __state = False
3
4  class Led_lamp(Lamp):          # Наследуемся от Lamp
5      voltage = 0
6      dim = True
7
8  class Ebergy_lamp(Lamp):      # Наследуемся от Lamp
9      temp = 2700
10     light_flow = 600
11
```

Как работает наследование

Объектам дочерних классов будут доступны те методы и свойства родительских объектов, которые не инкапсулированы:

```
class Lamp:
    __state = False

    def turn_light(self):
        if not self.__state:
            print("Light is on")
        else:
            print("Light is off")
        self.__state = not self.__state

    def __del__(self):              # Деструктор
        print("Lamp recycled")

class Led_lamp(Lamp):         # Наследуемся от Lamp
    voltage = 0
    dim = True

class Ebergy_lamp(Lamp):     # Наследуемся от Lamp
    temp = 2700
    light_flow = 600


led = Led_lamp()         # создаем новую лампу
enr = Energy_lamp()      # создаем новую лампу

led.turn_light()         # вызываем метод, унаследованный от Lamp
enr.turn_light()         # вызываем метод, унаследованный от Lamp
```

Переопределение методов и super()

Польза наследования ещё и в том, что можно изменить поведение у дочернего класса, сделав его более индивидуальным. Например, для LED-лампочки можно проверить, достаточно ли напряжения, переопределив метод **turn_light()**

```
class Led_lamp(Lamp):
    voltage = 0
    dim = True

    def turn_light(self):
        if self.voltage == 12:
            super().turn_light()
```

В данном примере мы сначала проверяем наличие нужного напряжения и только потом вызываем метод родительского класса. А помогает нам в этом **super()** — указатель на обращение к родительскому классу. Хотя мы могли бы заново написать все строчки из родительского метода **turn_light()**, принцип DRY нас от этого предостерегает.

Переопределение конструктора и super()

С конструктором похожая ситуация. Зачастую необходимо вызывать родительский конструктор, если мы его переопределяем в дочернем классе, и поможет нам в этом тот же super():

```
class Led_lamp(Lamp):
    voltage = 0

    def __init__(self, model, warranty, dim):
        super().__init__(model, warranty)          # конструктор род. класса
        self.dim = dim

    def turn_light(self):
        if self.voltage == 12:
            super().turn_light()
```

Полиморфизм

Что такое полиморфизм

Полиморфизм — это возможность одного и того же метода работать по-разному в разных классах. Самый простой пример полиморфизма — работа оператора сложения, дающая разные результаты для чисел и строк:

```
result = 5 + 5          # 10
result2 = "5" + "5"     # "55"
```

В классах мы просто создаём одинаковые методы, но с разной реализацией:

```
class Led_lamp(Lamp):
    voltage = 0

    def info(self):
        print(f"Led lamp, dim is {self.dim}, current voltage = {self.voltage}")

class Energy_lamp(Lamp):
    temp = 2700
    light_flow = 600

    def info(self):
        print(f"Energy lamp, temp = {self.temp}, light_flow = {self.light_flow}")
```

Статические методы

Декоратор @staticmethod

Мы с вами знаем, что в методы классов автоматически передаются сами объекты, чтобы код имел доступ к их данным.

Но что делать, если необходимо создать метод, которому не нужен доступ к свойству экземпляров класса. Для этого существуют статические методы. Они не принимают экземпляр класса в качестве аргумента. Чтобы преобразовать любой метод в статический, необходимо написать сверху декоратор **@staticmethod.**

```
1 class Animal:
2     def voice(self):              # Обычный метод
3         print('Animal sound')
4
5     @staticmethod                 # Статический метод
6     def eat():
7         print('Eating')
```

Особенности статических методов

Теперь, если создать экземпляр данного класса и выполнить метод, то он будет прекрасно работать:

```
1 pet = Animal()
2 pet.eat()
```

Помимо того, что статический метод не получает доступ к свойствам и методам экземпляров класса и самих классов, есть ещё несколько особенностей:

1. Экономится память и ресурсы процессора на создание объектов функций. Статический метод — это **ОДИН** объект на все экземпляры класса. То есть, они глобальны для всех объектов.
2. Статический метод по факту является функцией класса, а не его методом.
3. По причине п2 мы можем вызывать статические методы без создания объекта класса.
4. Статические методы не связываются ни с классом ни с его экземплярами. Они независимы и используются как вспомогательные методы для различных вычислений.

Область видимости классов

Основные определения

Прежде всего давайте разберёмся с 2-мя основными определениями: область видимости и пространство имён.

Пространство имен — конкретный объект (чаще всего словарь), в котором хранятся имена и их значения. Ссылки на конкретные объекты, созданные и существующие в памяти компьютера.

Нет абсолютно никакой связи с именами, определенных в разных пространствах имен.

Область видимости — это путь, по которому Python ищет определение имени в пространствах имён. По сути — это перечисление словарей через запятую, в которые Python последовательно заглядывает когда ищет имя.

Правила и исключения

При поиске имён используется правило **LEGB**:

- **L**ocal
- **E**nclosed
- **G**lobal
- **B**uiltins

Python идёт последовательно по этим пространствам имен. От локального до уровня Builtins.
Но существует 2 исключения:

1. Определение классов.
2. Порядок разрешения имён для аргументов функции **eval()** и **exec()**.

Пример

Давайте запустим следующий код:

```python
name = 'Murzik'                 # глобальная переменная

class Animal:
    name = 'Snezhok'            # локальный атрибут класса
    def print_name(self):
        print(name)             # пытаемся вывести атрибут класса

pet = Animal()
pet.print_name()
```

Мы увидим переменную, которая находится в глобальной области видимости (Murzik). А если мы закомментируем глобальную переменную **name**, то вовсе получим ошибку:

```
NameError: name 'name' is not defined
```

Пример

Поиск этих имён (свойств и методов) класса выполняется по обычным правилам за исключением того, что несвязанные локальные переменные (unbound local variables) ищутся в глобальном пространстве имён.

Python может создавать переменную только, если ей присваивается значение (оператор =). В случае с классом переменные name находятся в двух разных областях видимости. Переменная **name** класса **Animal** не видна из метода **print_name().** Потому что область видимости методов экземпляров не является вложенной в область видимости самого класса.

Чтобы исправить этот момент, нам необходимо обращаться к свойствам класса через **self**. Тогда согласно правилам разрешения имён, всё будет работать корректно:

```
1 def print_name(self):
2     print(self.name)
```

Подводные камни

При варианте обращения к атрибуту через **self** мы сможем только прочитать его значение. А присвоить новое значение уже не можем, потому что Python просто создаст новое свойство с тем же именем, но уже у самого экземпляра класса.

Протестируем код:

```
pet = Animal()
pet.print_name()
pet.name = 'Vaska'
pet.print_name()
```

Вывод покажется вполне логичным. Был Мурзик, а стал Васька. Но не обманывайтесь на этот счёт. Просто у нашего объекта **pet** появилось свое локальное свойство name. А вот у класса свойство **name** как было в значении Murzik, так и осталось. Добавьте ниже еще одну строку кода:

```
print(Animal.name)
```

Методы классов

Декоратор @classmethod

Если мы хотим изменять значения атрибутов класса, существует декоратор **@classmethod:**

```
1     @classmethod              # Метод класса
2     def print_name(cls):      # Для удобства используют переменную cls
3         print(cls.name)       # Печатаем атрибут класса, а не св-во объекта
```

Получили уже метод класса. А методы класса в качестве первого параметра принимают уже не объект, а сам **класс.** Чтобы это подсветить, используют **cls** вместо **self**.

У методов класса также есть особенности, похожие на статические методы:

1. Они могут быть вызваны как в объекте, так и в классе.
2. Они глобальны для всех экземпляров класса.
3. Они не имеют доступа к пространству имён экземпляров класса.
4. Но имеют полный доступ (чтение/запись) к атрибутам самого класса.

Пример

Методы класса часто применяют для создания различных инициализирующих методов (конструкторов).

```python
class Animal:
    def __init__(self, name):           # стандартный конструктор
        self.name = name

    @classmethod
    def from_file(cls, path):           # создаем объект из файла
        with open(path) as f:
            name = f.read().strip()
        return cls(name=name)

    @classmethod
    def from_obj(cls, obj):             # создаем объект из другого объекта
        if hasattr(obj, 'name'):
            name = getattr(obj, 'name')
            return cls(name=name)
        return cls
```

Геттеры и сеттеры

Ещё раз о свойствах класса

Свойства класса — это способ доступа к переменным класса. Мы обращаемся к ним только с двумя целями. Когда нам нужно прочитать и когда нужно изменить значение.

```
1  class Animal:
2      def __init__(self, name):  # стандартный конструктор
3          self.name = name
4
5  c = cat("Murzik")
6  c.name = "Vaska"               # Не приветствуется
```

Но на самом деле прямое изменение значения атрибутов нигде не приветствуется. Общепринятая модель класса — это создание приватного свойства.

- Чтобы прочитать значение этого свойства, создают метод, который возвращает его значение, который называется геттером (от англ. get).
- Чтобы изменить значение свойства, создают метод, который называется сеттером (от англ. set).

Реализация геттеров и сеттеров

Для реализации геттеров и сеттеров в Python существует специальный встроенный класс property, объектом которого нам нужно сделать наше приватное свойство:

```python
class Animal:
    def __init__(self, name):
        self._name = name       # Приватное свойство

    def get_name(self):         # Геттер
        print('From get_name')
        return self._name

    def set_name(self, value):  # Сеттер
        print('From set_name')
        self._name = value

    name = property(fget=get_name, fset=set_name)   # наше свойство с доступом через сеттер и геттер


c = Animal("Murzik")
print(c.name)               # Вызовется наш геттер

c.name = "Vaska"            # Вызовется наш сеттер
```

Особенности подхода

В качестве аргументов fget и fset класса property можно передавать любые методы. Это поведение не сломает внутренний интерфейс класса.

Геттеры и сеттеры выполняют роль буферных зон. Они позволяют нам выполнять дополнительные действия (например, валидация входного значения или очистка/нормализация значения перед использованием).

Декоратор @property

При создании объекта класса property, мы передаём на вход объект функции (геттер и сеттер), а получаем результат её работы. Получается так потому, что property — это декоратор. Мы можем описать наш геттер следующим образом:

```
    @property
    def name(self):          # Геттер
        print('From get_name')
        return self._name
```

И похожим образом можно сделать функцию сеттера:

```
    @name.setter
    def name(self, value):  # Сеттер
        self._name = value
```

У нас получаются две функции с одинаковым именем и разным поведением. Это не ошибка! Имя здесь имеет значение. Если оно будет другим, то мы получим исключение. Имена функций одинаковые, но у них разные декораторы. И это работает именно так!

Множественное наследование

Множественное наследование

У одного класса может быть несколько родителей, которые мы указываем через запятую. Такой вид наследования называется множественным. В некоторых случаях — это полезный подход, позволяющий быстро спроектировать новый класс, не вмешиваясь в работу родительских. Но у такого подхода могут быть недостатки.

Попробуйте определить, что будет в результате работы следующего кода:

```python
class Animal:
    def voice(self):
        print('Voice')

class Dog(Animal):
    def voice(self):
        print('Woof')

class Cat(Animal):
    def voice(self):
        print('Meow')

class Hybrid(Dog, Cat):     # Множественное наследование от классов Dog и Cat
    pass


h = Hybrid()
h.voice()                   # Чей метод будуте вызыван? И будет ли вызыван вообще?
```

Множественное наследование

Дочерний класс наследует свойства и методы в соответствии с правилом **mro — method resolution order** (Порядок разрешения методов). Суть заключается в том, что наследование происходит в первую очередь от родителя, который указан левее при определении наследования. В нашем случае — это класс Dog.

Порядок указываемых классов при множественном наследовании имеет значение!

У каждого класса есть метод mro() при вызове которого будет возвращен список, в котором перечислен тот порядок, в котором Python будет искать и использовать определение имён:

```
1 h = Hybrid()
2 print(h.__class__.mro())
3
4 >>>
5 [<class '__main__.Hybrid'>, <class '__main__.Dog'>, <class '__main__.Cat'>, <class '__main__.Animal'>, <class 'object'>]
6
```

В связи с наличием множественного наследования, в Python имеется идея миксинов (от mix in — примесь).

Создание миксинов

Миксины - это маленькие классы, которые имеют небольшой набор методов, которые подмешиваются другим классам, за счет того, что они становятся их родителями.

Идея миксинов предполагает их использование только вместе с другими классами для кастомизации и расширения их возможностей без изменения основной функциональности. Создание отдельных экземпляров миксинов не предполагается!

Давайте создадим миксин для нашего кода, который добавит любимую еду для питомцев:

```python
class Animal:
    def voice(self):
        print('Voice')

class Dog(Animal):
    def voice(self):
        print('Woof')

pet = Dog()
```

Создание миксинов

```python
class FoodMixin:                        # Создаем миксин
    food = None

    def get_food(self):
        if self.food is None:
            raise ValueError('Food should be set')
        print(f'I like {self.food}')

class Animal:
    def voice(self):
        print('Voice')

class Dog(FoodMixin, Animal):           # Подмешиваем миксин
    food = 'meat'                       # Переопределяем св-во foodm иначе получим Exception
    def voice(self):
        print('Woof')

pet = Dog()
```

Практическое задание

Создайте класс User и и его наследника класс SuperUser, которые описывают пользователя и супер-пользователя.

В классе User необходимо описать:

- Конструктор, который принимает в качестве параметров значения для атрибутов name, login и password;
- Методы для изменения и получения значений атрибутов;
- Метод show_info, который печатает в произвольном формате значения атрибутов name и login;
- Атрибут класса count для хранения количества созданных экземпляров класса User.

Необходимые условия, которые надо учесть после создания объекта:

- Атрибут name доступен и для чтения, и для изменения;
- Атрибут login доступен только для чтения;
- Атрибут password доступен только для изменения.

В классе SuperUser необходимо описать:

- Конструктор, который принимает в качестве параметров значения для атрибутов name, login, password и role;
- Метод для изменения и получения значения атрибута role;
- Метод show_info, который печатает в произвольном формате значения атрибутов name, login и role;
- Атрибут класса count для хранения количества созданных экземпляров класса SuperUser.

Практическое задание

Как это должно работать:

```
user1 = User('Paul McCartney', 'paul', '1234')
user2 = User('George Harrison', 'george', '5678')
user3 = User('Richard Starkey', 'ringo', '8523')
admin = User('John Lennon', 'john', '0000', 'admin')

user1.show_info() # Например: Name: Paul McCartney, Login: paul
admin.show_info() # Например: Name: John Lennon, Login: john

users = User.count
admins = SuperUser.count

print(f'Всего обычных пользователей: {users}') # Всего обычных пользователей: 3
print(f'Всего супер-пользователей: {admins}') # Всего супер-пользователей: 1

user3.name = 'Ringo Star'
print(user3.name) # Ringo Starr

print(user2.login) # george
user2.login = 'geo' # Должна быть ошибка

user1.password = 'Pa$$w0rd'
print(user2.password) # Должна быть ошибка
```

Вопросы?

Вопросы?

Вопросы?