

Лекция 7. Классы ч.1

Объекты и классы. Абстракция.

Методы и атрибуты классов.

Экземпляры классов.

Наследование. Вызов «родительских» методов классов.

Полиморфизм. Переопределение «базовых» методов

Публичные и приватные методы классов





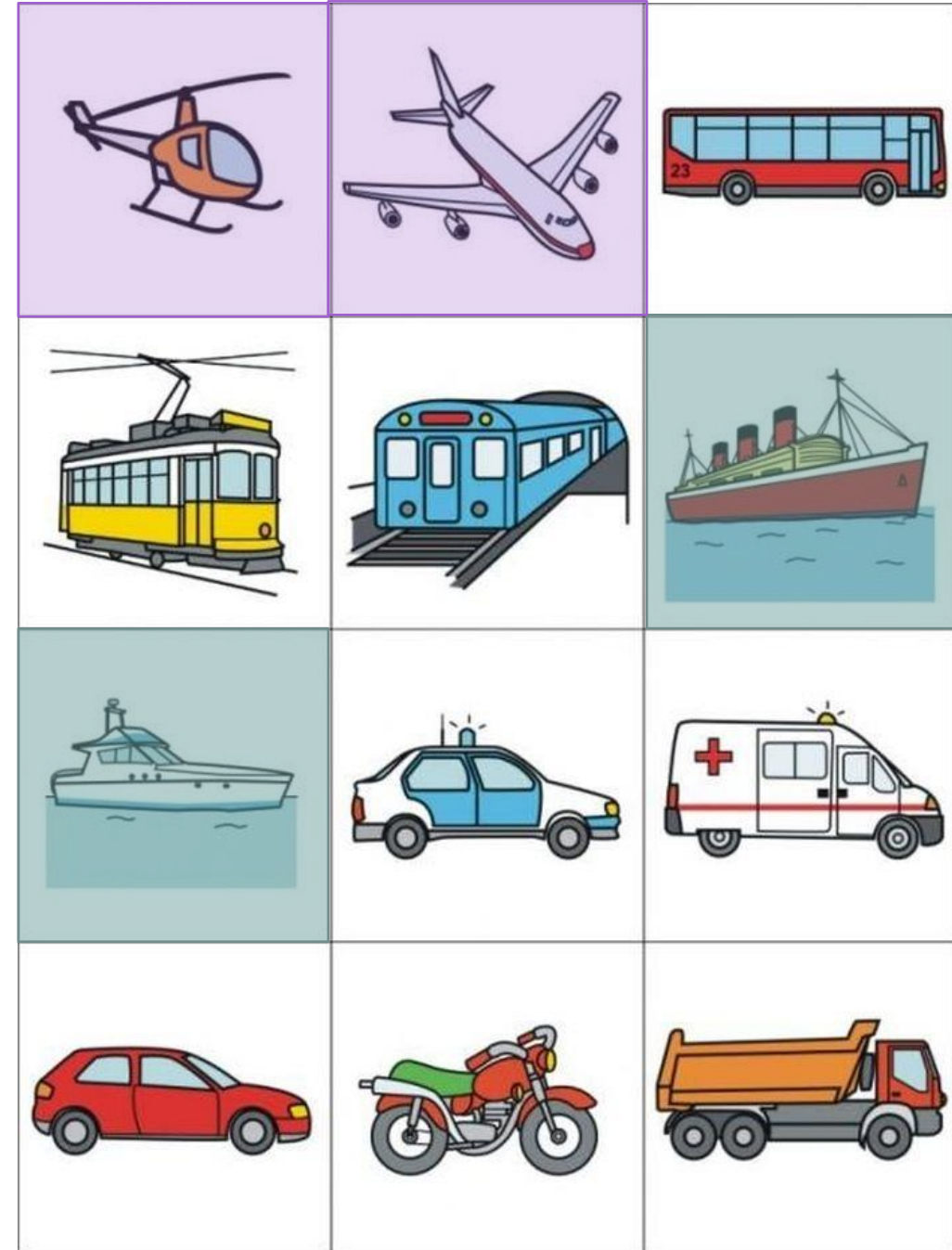
Объекты и классы

Объекты и классы. Абстракция. Методы и атрибуты классов.
Экземпляры классов

Абстракция

Абстракция - выделение значимых характеристик объекта, отличающих его от других объектов.

По этим характеристикам мы можем **объединить** несколько объектов в один **класс**



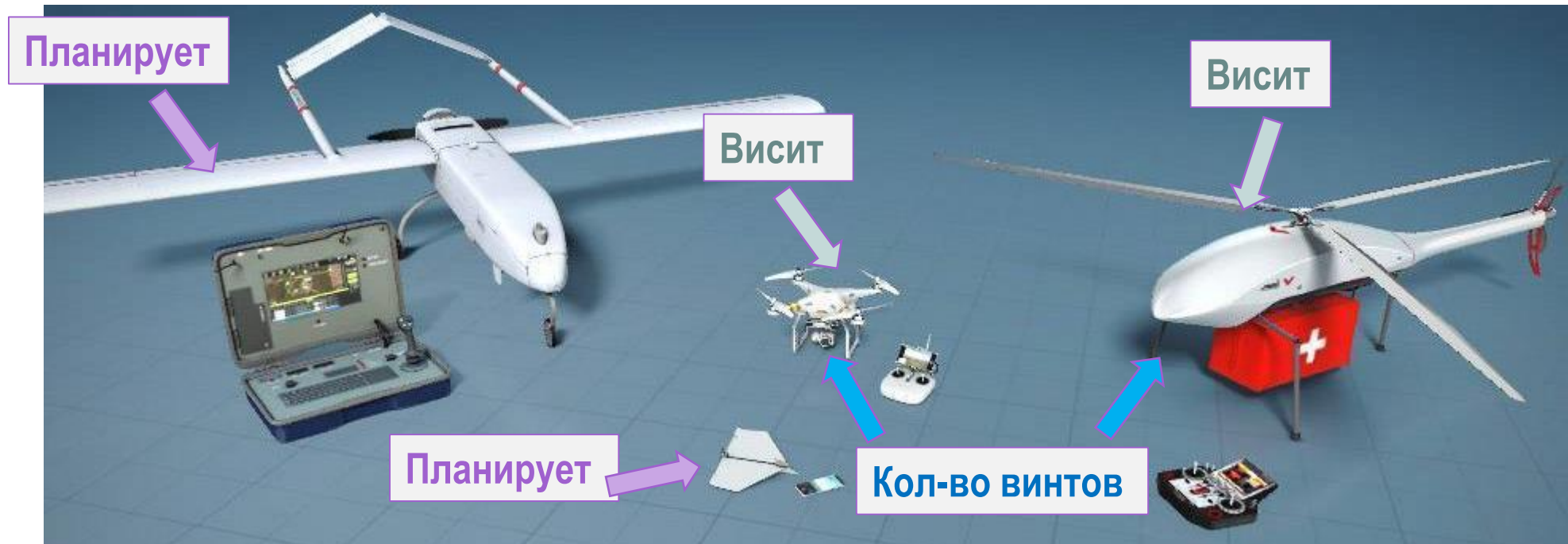
Абстракция (продолжение)

Самолет

Вертолет

Мультиротор

Конвертоплан



Декомпозиция

Декомпозиция – разделение целого на отдельные значимые части по какому-то принципу.

Благодаря этому мы можем выделить в классе его типовые **действия** (методы) и типовые **признаки** (атрибуты)

Взлететь

Лететь

Выбросить парашют

Выполнить посадку



Есть парашют

Умеет планировать

Макс. взлет. масса

Макс. скорость

Модель и
Производитель



Декомпозиция

Мультиротор: Геоскан 401, DJI Mavic 2 Pro, Parrot Anafi, Autel Evo II Pro, ...

- ☐ Может взлетать
- ☐ Может садиться
- ☐ Может висеть (может планировать)

Есть параметры:

- количество двигателей
- ~~■ наличие парашюта~~
- максимальная взлетная масса
- максимальная скорость
- максимальное время полета
- производитель и модель



Декомпозиция

Самолет: Геоскан 201, Zala 421-20, Supercam S350, Птеро-G1, ...

- ☐ Может взлетать
- ☐ Может садиться
- ☐ ~~Может висеть~~ (может планировать)

Есть параметры:

- ☒ ~~количество двигателей~~
- ☒ наличие парашюта
- ☒ максимальная взлетная масса
- ☒ максимальная скорость
- ☒ максимальное время полета
- ☒ производитель и модель



Классы и объекты

Класс – описание объекта, объединяющее его поведение (**методы**), характеристики (**атрибуты**), правила использования и взаимодействия с ним.

Класс: **самолет**, **мультиротор**.

Объект класса – конкретный представитель класса, у которого есть «индивидуальность», отличающая его от других представителей того же класса.

Объект: мультиротор **DJI Mavic 2 Pro**, мультиротор **Autel Evo Nano**



Класс Aircraft - конструкция

Объявление класса

```
class Aircraft:
```

Атрибут

```
    self.weight = ...
```

Метод

```
    def __init__(self, weight):
```

Ссылка на экземпляр

```
        self.weight = weight
        self.flight = False
```

```
aircraft1 = Aircraft(5000)
```



Класс Aircraft - self

```
class Aircraft:
    weight = 0
    def __init__(self, w):
        weight = w

aircraft1 = Aircraft(5000)
aircraft2 = Aircraft(450)
print(aircraft1.weight)
print(aircraft2.weight)
```

Выведет 0 и 0

```
class Aircraft2:
    def __init__(self, w):
        self.weight = w

aircraft1 = Aircraft2(5000)
aircraft2 = Aircraft2(450)
print(aircraft1.weight)
print(aircraft2.weight)
```

Выведет 5000 и 450



Класс Aircraft – методы и атрибуты

Методы

- ❑ Взлететь – `takeoff()`
- ❑ Сесть – `landing()`

Атрибуты

- ❑ Масса - `weight`
- ❑ Модель - `model`
- ❑ Находится в полете (да/нет) - `flight`

```
class Aircraft:
    def __init__(self, model, weight):
        self.weight = weight
        self.model = model

    def takeoff(self):
        self.flight = True

    def landing(self):
        self.flight = False
```





Наследование

Наследование. Вызов "родительских" методов

Наследование

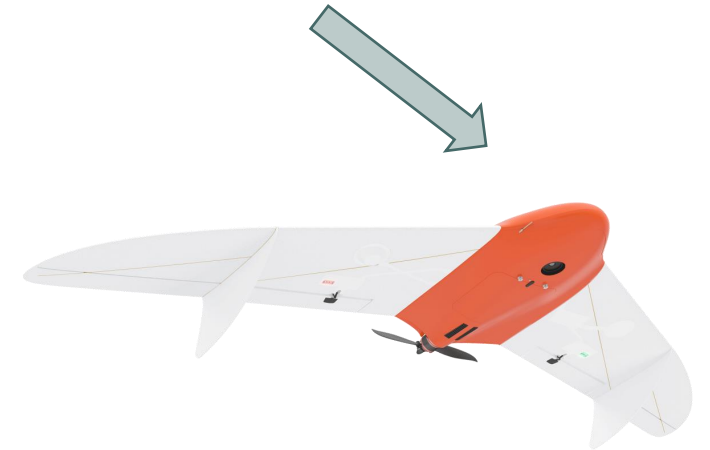
Наследование – создание нового описания объекта на базе существующего объекта и **заимствование** его поведения и свойств (полное или частичное)



Летательный аппарат



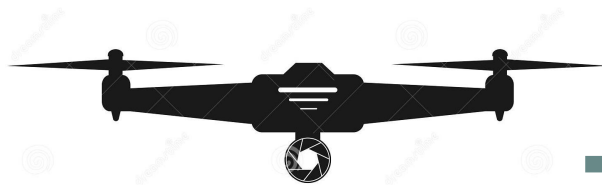
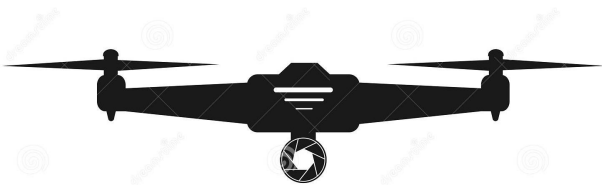
Мультиротор
(квадрокоптер)



Самолет



Наследование (продолжение)



Наследование (продолжение)

```
class Aircraft:
    def __init__(self, model, weight):
        self.weight = weight
        self.model = model

    def takeoff(self):
        self.flight = True

    def landing(self):
        self.flight = False
```

```
class Plane(Aircraft):
    def __init__(self, model, weight):
        self.weight = weight
        self.model = model
        self.parachute = True
        self.can_hover = False
```



+ takeoff() + landing()

```
class Multicopter(Aircraft):
    def __init__(self, model, weight):
        self.weight = weight
        self.model = model
        self.parachute = False
        self.can_hover = True
```



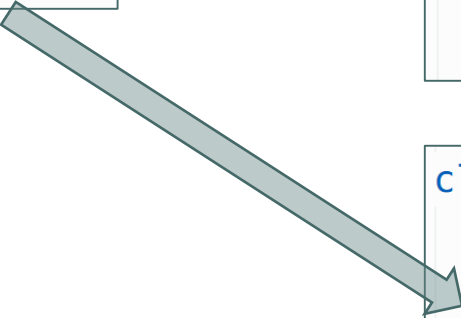
+ takeoff() + landing()



Вызов «родительского» метода

```
class Aircraft:  
    def __init__(self, model, weight):  
        self.weight = weight  
        self.model = model
```

```
class Plane(Aircraft):  
    def __init__(self, model, weight):  
        self.weight = weight  
        self.model = model  
        self.parachute = True  
        self.can_hover = False
```




```
class Plane(Aircraft):  
    def __init__(self, model, weight):  
        Aircraft.__init__(self, model, weight)  
        self.parachute = True  
        self.can_hover = False
```



Вызов «родительского» метода – `super()`

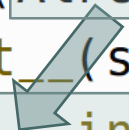
`super()` заменяет собой упоминание родительского класса. При множественном наследовании автоматически «берет» первый по списку класс

```
class Aircraft:
    def __init__(self, model, weight):
        self.weight = weight
        self.model = model
```



```
class Plane(Aircraft):
    def __init__(self, model, weight):
        Aircraft.__init__(self, model, weight)
        self.parachute = True
        self.can_hover = False
```

```
class Plane(Aircraft):
    def __init__(self, model, weight):
        super().__init__(model, weight)
        self.parachute = True
        self.can_hover = False
```



это интересно: <https://pythonist.ru/vvedenie-v-mnozhestvennoe-nasledovanie-i-super/>



Множественное наследование

```
class Aircraft:
    def __init__(self, model, weight):
        self.weight = weight
        self.model = model

    def takeoff(self):
        self.flight = True

    def landing(self):
        self.flight = False
```

```
class UAV:
    def __init__(self):
        self.has_autopilot = True
        self.missions = []
```

```
class PlaneUAV(Aircraft, UAV):
    pass

plane = PlaneUAV("Geoscan 201", 3000)
print(plane.model)
print(plane.weight)
print(plane.has_autopilot)
print(plane.missions)
```

Выведет Geoscan 201
и 3000

Сгенерирует исключение `AttributeError`



Множественное наследование (продолжение)

```
class Aircraft:
    def __init__(self, model, weight):
        self.weight = weight
        self.model = model

    def takeoff(self):
        self.flight = True

    def landing(self):
        self.flight = False
```

```
class UAV:
    def __init__(self):
        self.has_autopilot = True
        self.missions = []
```

```
class PlaneUAV(Aircraft, UAV):
    def __init__(self, model, weight):
        super().__init__(model, weight)
        UAV.__init__(self)

plane = PlaneUAV("Geoscan 201", 3000)
print(plane.model)
print(plane.weight)
print(plane.has_autopilot)
print(plane.missions)
```

Выведет Geoscan 201, 3000,
True, []



MRO (multiple-resolution order)

Это порядок поиска нужного функционала:

текущий класс → первый родитель → остальные родители в порядке их объявления → class object

```
class PlaneUAV(Aircraft, UAV):
```

```
print(PlaneUAV.mro( ))
```

Выведет <class '__main__.PlaneUAV'>, <class '__main__.Aircraft'>, <class '__main__.UAV'>, <class 'object'>





Полиморфизм

Полиморфизм. Переопределение «базовых» методов

Полиморфизм

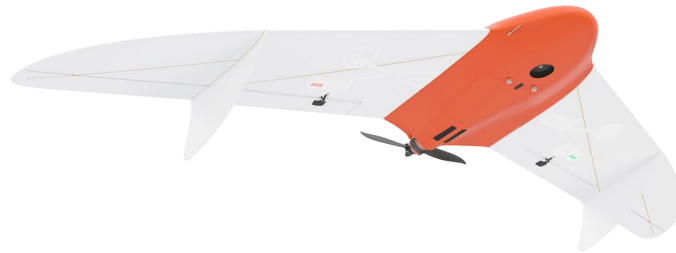
Полиморфизм – один и тот же метод в разных классах делает одно и то же, но по-разному. То есть имеет разную реализацию



Мультиротор
(квадрокоптер)

Посадка:

☐ сесть



Самолет

Посадка:

- ☐ зайти на траекторию
- ☐ выключить двигатель
- ☐ с парашютом? выпустить парашют
- ☐ без парашюта? посадка по-самолетному
- ☐ сесть



Полиморфизм (продолжение)



Мультиротор (квадрокоптер)

```
class MultirotorUAV(Aircraft, UAV):  
    def landing(self):  
        super().landing()
```

или вообще не переопределять
метод landing()



Самолет

```
class PlaneUAV(Aircraft, UAV):  
    def landing(self):  
        if self.parachute:  
            print("Парашют открыт")  
        else:  
            print("Посадка по-самолетному")  
        super().landing()
```



Переопределение «базовых» методов

«Базовые» методы – это методы класса **object**. Например:

`__str__` выводит базовое сообщение класса. Например, в `print()`

`__gt__` («больше») сравнивает два объекта

Любой класс наследуется от **object**, поэтому эти методы легко переопределить



Переопределение `__str__`

`__str__` возвращает строку с информацией о классе. По умолчанию это название класса и адрес экземпляра в памяти.

Примерно так: `<__main__.PlaneUAV object at 0x7f2e74afce80>`.

Переопределение ниже вернет `=== Geoscan 201 ===`

```
class PlaneUAV(Aircraft, UAV):  
    def __str__(self):  
        return f"=== {self.model} ==="  
  
plane = PlaneUAV("Geoscan 201", 3000)  
print(plane)
```



Переопределение `__gt__`

`__gt__` **greater then (больше)** может проверить, какой из объектов больше, но только при условии, что сравниваются подходящие типы данных (числа или все, что можно трактовать на основе чисел, – например, строки).

Поэтому код, приведенный ниже, сгенерирует исключение
`TypeError: '>' not supported between instances of 'PlaneUAV' and 'PlaneUAV'`

```
plane = PlaneUAV("Geoscan 201", 3000)
plane2 = PlaneUAV("Geoscan 201 Геодезия", 3700)
print(plane > plane2)
```



Переопределение `__gt__` (продолжение)

Необходимо выбрать числовое значение, которое можно сравнить.
Например, максимальную взлетную массу.

Код ниже выведет `False`, т.к. первый аппарат был легче (~~$3000 > 3700$~~).

```
class PlaneUAV(Aircraft, UAV):
    def __gt__(self, other):
        if not isinstance(other, Aircraft):
            return
        else:
            return self.weight > other.weight

plane = PlaneUAV("Geoscan 201", 3000)
plane2 = PlaneUAV("Geoscan 201 Геодезия", 3700)
print(plane > plane2)
```



Переопределение `__gt__`: особенности `isinstance()`

`isinstance` проверяет, является ли объект экземпляром **указанного класса** ИЛИ его **потомка** (т.е. класса-наследника).

В коде ниже будет логично написать как **Aircraft**, так и **PlaneUAV** , так как у обоих классов есть атрибут **weight**

```
class PlaneUAV(Aircraft, UAV):  
    def __gt__(self, other):  
        if not isinstance(other, Aircraft):  
            return  
        else:  
            return self.weight > other.weight
```



Переопределение «базовых» методов: что вызывается на самом деле?

«Выводя на печать» объект класса PlaneUAV, мы на самом деле вызываем его встроенный метод `__str__()`. Поэтому обе выделенные ниже строки абсолютно идентичны

```
class PlaneUAV(Aircraft, UAV):  
    def __str__(self):  
        return f"=== {self.model} ==="  
  
plane = PlaneUAV("Geoscan 201", 3000)  
print(plane)  
print(plane.__str__())
```





Модификаторы ВИДИМОСТИ

Модификаторы public, __private, _protected

Модификаторы видимости

- ❑ **public** – видимый по умолчанию. Доступ – откуда угодно.
- ❑ **__private** – видимый только внутри класса. При попытке доступа снаружи генерируется исключение. Чтобы сделать метод или атрибут приватным, перед его названием пишется двойное подчеркивание
- ❑ **_protected** – видимый только внутри класса и наследников этого класса. По факту доступ все равно откуда угодно. Перед названием пишется одинарное подчеркивание



Модификаторы видимости (продолжение)

Модифицируем базовый класс **UAV**: теперь `_has_autopilot` – **protected**, `__missions` – **private**

```
class UAV:
    def __init__(self):
        self._has_autopilot = True
        self.__missions = []
```

```
plane = PlaneUAV("Geoscan 201", 3000)
```

```
print(plane._has_autopilot) выведет True
```

```
print(plane.__missions) вызовет исключение AttributeError: 'PlaneUAV' object has no attribute '__missions'
```



Модификаторы видимости (продолжение)

На самом деле нельзя по-настоящему «закрыть» данные от доступа. Python просто добавляет к имени private атрибута или метода имя класса с одним нижним подчеркиванием: получается `_UAV__missions`.

Однако в списке методов и атрибутов класса он отображаться не будет.

```
plane = PlaneUAV("Geoscan 201", 3000)
print(plane._UAV__missions)
print(dir(plane))
```

```
['_UAV__missions', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_has_autopilot', 'landing', 'model', 'takeoff', 'weight']
```



Модификаторы видимости (продолжение)

Зато это позволяет создавать непересекающиеся методы и атрибуты с одинаковыми именами в базовом и производном классе. В случае ниже получится `_UAV__missions` и `_PlaneUAV__missions` с разными значениями.

```
class PlaneUAV(Aircraft, UAV):  
    def __init__(self, model, weight):  
        super().__init__(model, weight)  
        UAV.__init__(self)  
        self.__missions = [(1,2), (3,4)]
```

```
class UAV:  
    def __init__(self):  
        self._has_autopilot = True  
        self.__missions = []
```

```
plane = PlaneUAV("Geoscan 201", 3000)  
print(plane._UAV__missions)      выведет []  
print(plane._PlaneUAV__missions) выведет [(1, 2), (3, 4)]
```





Добавление и удаление методов

Добавление внешних методов в класс. Удаление «лишних» методов из класса

Добавление метода в класс

```
# объявляем функцию вне класса:
```

```
def need_reg(self):  
    return "Нужно регистрировать" if self.weight > 150 else "Не нужно регистрировать"
```

```
# и присваиваем ее атрибуту внутри класса:
```

```
class PlaneUAV(Aircraft, UAV):  
    puav_need_registration = need_reg
```

```
    def __init__(self, model, weight):  
        super().__init__(model, weight)  
        UAV.__init__(self)  
        self.__missions = [(1,2), (3,4)]
```

```
plane = PlaneUAV("Geoscan 201", 3000)  
print(dir(plane))  
print(plane.puav_need_registration())
```

выведет Нужно регистрировать

```
['_PlaneUAV__missions', '_UAV__missions',  
 '__class__', '__delattr__', '__dict__',  
 '__dir__', '__doc__', '__eq__', '__format__',  
 '__ge__', '__getattr__', '__gt__',  
 '__hash__', '__init__', '__init_subclass__',  
 '__le__', '__lt__', '__module__',  
 '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
 '__repr__', '__setattr__', '__sizeof__', '__str__',  
 '__subclasshook__', '__weakref__', 'has_autopilot', 'landing',  
 'model', 'puav_need_registration', 'takeoff', 'weight']
```



Удаление метода базового класса из класса-наследника

```
class PlaneUAV(Aircraft, UAV):  
    def __init__(self, model, weight):  
        super().__init__(model, weight)  
        UAV.__init__(self)  
        self.parachute = True  
        del Aircraft.landing
```

```
    def landing(self):  
        if self.parachute:  
            print("Парашют открыт")  
        else:  
            print("Посадка по-самолетному")  
            super().landing()
```

```
plane = PlaneUAV("Geoscan 201", 3000)  
print(dir(plane))  
print(dir(Aircraft))  
plane.landing()
```

ВЫЗОВЕТ ИСКЛЮЧЕНИЕ

AttributeError: 'super' object has
no attribute 'landing'

```
['UAV_missions', '__class__', 'delattr', '__dict__', '__dir__',  
 '__doc__', '__eq__', 'format', '__ge__', 'getattribute', '__gt__',  
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',  
 'module', '__ne__', 'new', '__reduce__', '__reduce_ex__', '__repr__',  
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',  
 'ref', '__has_autopilot', 'landing', 'model', 'parachute', 'takeoff',  
 'weight']  
['__class__', 'delattr', '__dict__', '__dir__', '__doc__', '__eq__',  
 'format', '__ge__', 'getattribute', '__gt__', '__hash__', '__init__',  
 '__init_subclass__', '__le__', '__lt__', 'module', '__ne__', 'new',  
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',  
 '__str__', '__subclasshook__', '__weakref__', 'takeoff']
```

```
class Aircraft:  
    def __init__(self, model, weight):  
        self.weight = weight  
        self.model = model  
  
    def takeoff(self):  
        self.flight = True  
  
    def landing(self):  
        self.flight = False
```

