Объектно-ориентированное программирование

основные понятия, абстракция

ООП

Объектно-ориентированное программирование (**ООП**) — методология программирования, основанная на представлении программы в виде **совокупности объектов**, каждый из которых является **экземпляром определённого класса**, а классы **образуют иерархию наследования**.

ООП

Основные понятия ООП

- Абстракция
- Инкапсуляция
- Полиморфизм
- Наследование

Абстракция

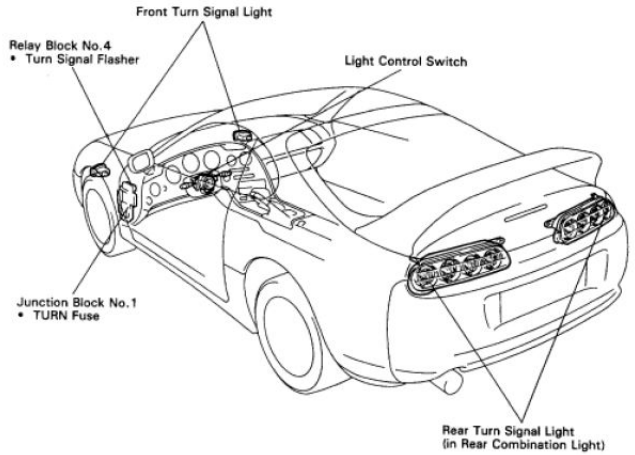
Реальный мир

- Цвет
- Вес
- Марка
- Модель
- Мощность двигателя
- Пробег
- + 100500 характеристик

Программирование на ООП

- Марка
- Модель
- Цвет
- Пробег
- Год выпуска

Только необходимые характеристики



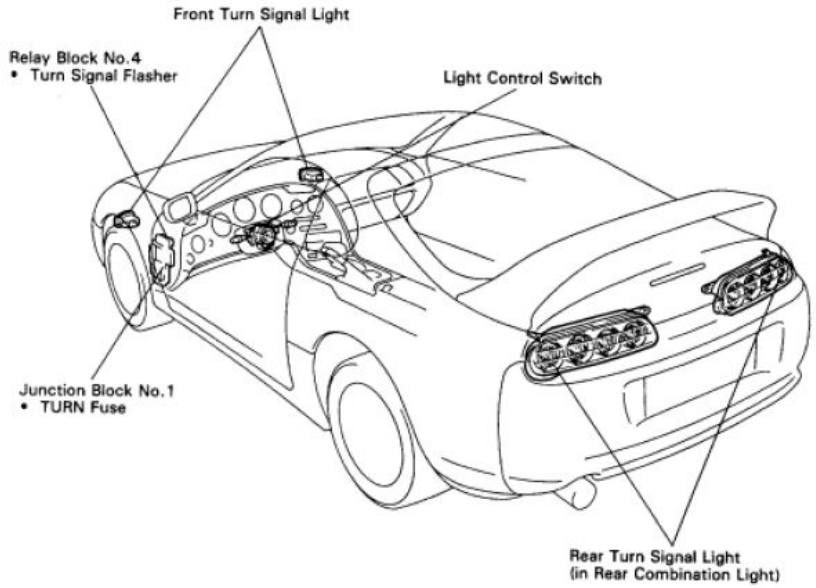
Абстракция в ООП

Если сложно:

- **Абстракция** — это использование только тех характеристик объекта, которые с достаточной точностью представляют его. Нужно представить объект минимальным набором полей и методов и при этом с достаточной точностью для решаемой задачи

Если просто:

- **Абстракция** – это выделение основных, наиболее значимых характеристик объекта и игнорирование второстепенных



Ваши вопросы

что необходимо прояснить в рамках данного раздела

Наследование

создаем иерархию классов, повторно используем код

Наследование

Наследование — это механизм системы, который позволяет, описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью, т.е. наследовать одними классами свойства и поведение других классов для дальнейшего расширения или модификации.

Как называют классы в наследовании

Исходный класс

- Родительский
- Базовый
- Суперкласс
- Предок

Новый класс

- Дочерний
- Подкласс
- Наследник
- Потомок
- Производный

**Данные слова являются синонимами*

Как реализовать в Python

1. Описать родительский класс
2. Описать дочерний:
   - После названия в скобках указывается родительский класс
   - Добавить новый функционал (атрибуты, методы)
   - Изменить родительский функционал (если необходимо)

```python
class Transport:
    brand: str
    model: str
    color: str

    def __init__(self, brand, model, color):
        self.brand = brand
        self.model = model
        self.color = color

    def move(self, distance):
        print(f'{self.brand} {self.model} может двигаться')


class Car(Transport):
    vin: str
    issue_year: int
    mileage: int

    def __init__(self, brand, model, color, vin, issue_year, mileage=0):
        super().__init__(brand, model, color)
        self.vin = vin
        self.issue_year = issue_year
        self.mileage = mileage

    def move(self, distance):
        print(f'{self.model} вжжжжж')
        self.mileage += distance


supra = Car(
    'Toyota',
    'Supra A80',
    'black',
    'JT4RN13P7K0001611',
    1995,
    215000
)
```

Добавление метода

В дочерний класс можно добавить атрибуты и методы, которых нет в родительском для расширения функционала

```python
class Transport:
    brand: str
    model: str
    color: str

    def __init__(self, brand, model, color):
        self.brand = brand
        self.model = model
        self.color = color

    def move(self, distance):
        print(f'{self.brand} {self.model} может двигаться')


class Car(Transport):
    vin: str
    issue_year: int
    mileage: int

    def __init__(self, brand, model, color, vin, issue_year, mileage=0):
        super().__init__(brand, model, color)
        self.vin = vin
        self.issue_year = issue_year
        self.mileage = mileage

    def move(self, distance):
        print(f'{self.model} вжжжжж')
        self.mileage += distance

    def signal(self):
        print(f'{self.supra} бип-бип')

supra = Car(
    'Toyota',
    'Supra A80',
    'black',
    'JT4RN13P7K0001611',
    1995,
    215000
)
```

Плюсы и минусы наследования

- ✔ Новый класс автоматический может использовать весь код старого класса
- ✔ Отличный способ повторного использования кода (привет принцип DRY)

- × Может усложнить понимание и использование кода
- × Замедляет написание кода (если хорошо прорабатывать)

Метод super

Иногда возникает необходимость вызвать метод родительского класса. Для того, чтобы это сделать, нужно вызвать метод super().

- super() получает определение родительского класса
- super() используется, чтобы создать объект, который работает примерно так же, как и объект родительского класса

```
1  class Square(Rectangle):
2      def __init__(self, length):
3          super().__init__(length, length)
4
5      def area(self):
6          return self.length * self.width
7
8  class Cube(Square):
9      def surface_area(self):
10         face_area = super().area()
11         return face_area * 6
12
13     def volume(self):
14         face_area = super().area()
15         return face_area * self.length
```

Наследование vs композиция

Композиция

- Возможность смены в runtime агрегируемого объекта
- Полная замена агрегируемого объекта в классах, производных от класса, включающего агрегируемый объект
- Возможность скрыть определённую часть реализации, а также исходные параметры, необходимые поведению, посредством передачи их через конструктор (при наследовании поведению придётся запрашивать их через методы/свойства собственного интерфейса)

Наследование

- Атрибуты и методы полностью доступны предку для изменения. Перегрузка методов
- Наследник является корректным подтипом предка
- Проще получить интерфейс предка в отличие от композиции
- Наследование имеет статическую природу и устанавливает отношения классов только на этапе интерпретации/компиляции

Вопрос-ответ

? Как мне выбирать, что использовать, композицию или наследование?

✔ Можно воспользоваться шпаргалкой (но не стоит слепо ей следовать, например, при примесях (миксинах)):

- сущность А является сущностью Б? Если да, то скорее всего, тут подойдет наследование
- сущность А является частью сущности Б? Если да, то, наверное, стоит выбрать композицию

```
1  class Building:
2      pass
3
4
5  class Bathroom:
6      pass
7
8
9  class House(Building):
10     bathroom: Bathroom
```

Множественное наследование

Множественное наследование – возможность класса наследоваться от нескольких классов.

Множественное наследование удобно использовать для реализации примесей

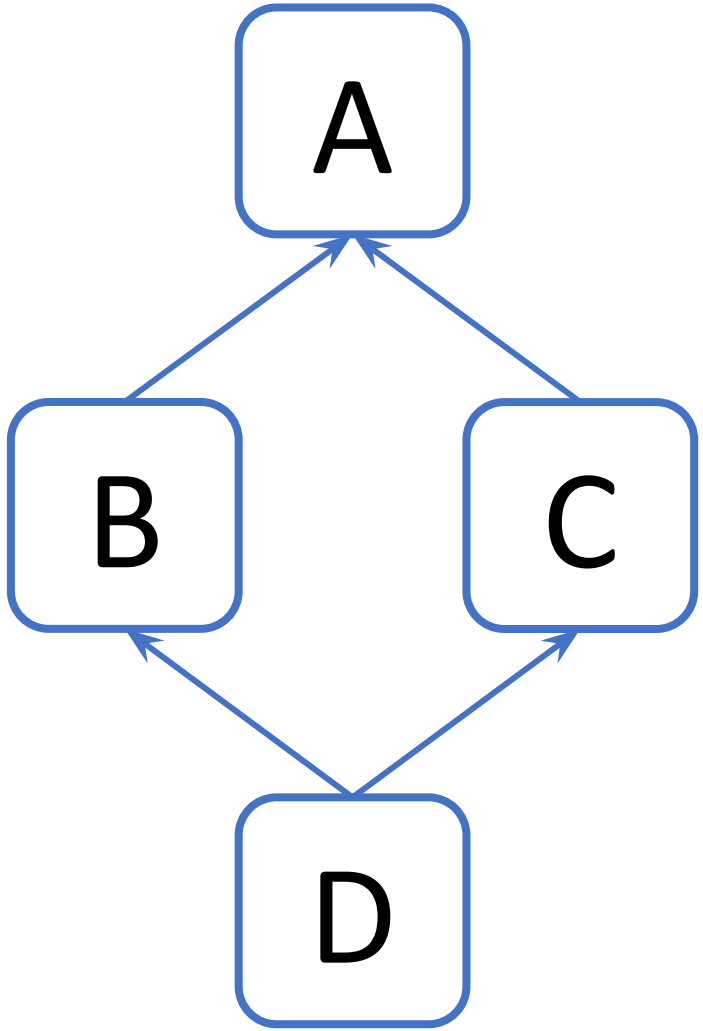
```
1  class Scanner:
2      pass
3
4
5  class Printer:
6      pass
7
8
9  class Copier(Scanner, Copier):
10     pass
```

The Diamond Problem

Ромбовидное наследование – ситуация при множественном наследовании, когда класс наследуется от двух и более, которые является потомками одного суперкласса.

При такой схеме возникают следующие проблемы:

- Неразбериха с конструкторами
- Вызов из D метода, определенного в A и перегруженного в B и C



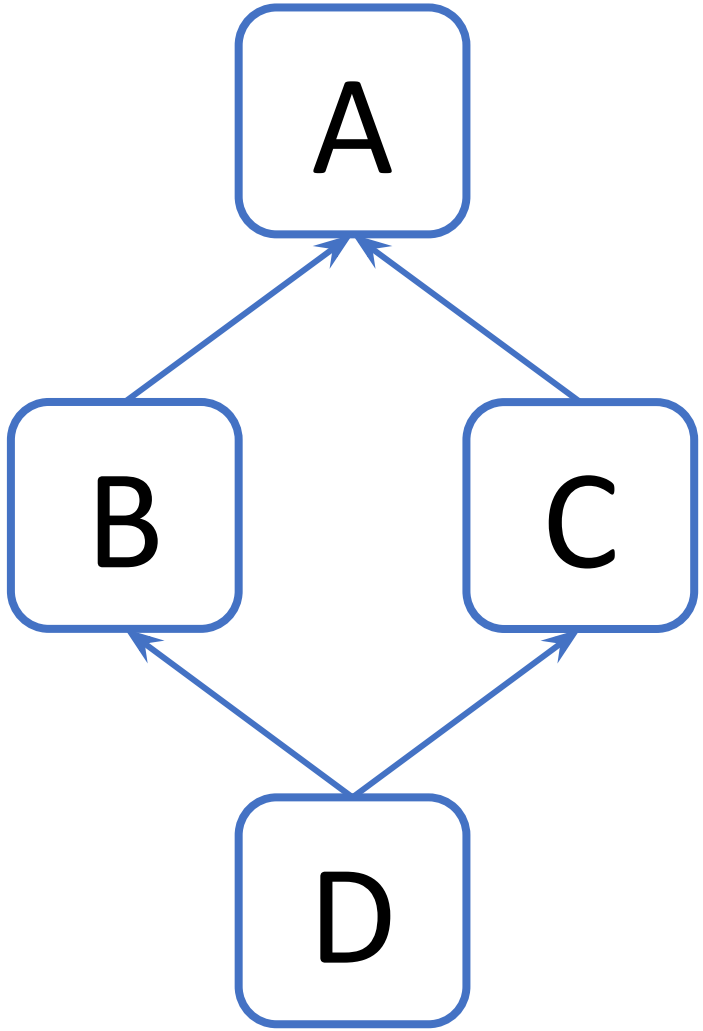
Method Resolution Order / MRO

Порядок разрешения методов (MRO) – что, если у нескольких родителей будут одинаковые методы? Какой метод в таком случае будет использовать наследник?

Diamond problem в Python решается путем установления порядка разрешения методов.

В Python 3 для определения порядка используется алгоритм поиска в ширину (D-B-C-A)

В Python 2 используется алгоритм поиска в глубину (D-B-A-C)



Вопрос-ответ

? Как я могу посмотреть MRO для моей иерархии наследования?

✔ В Python 3 для этого есть метод класса Class.mro()

```python
class A:
    def who_am_i(self):
        print('A')


class B(A):
    def who_am_i(self):
        print('B')


class C(A):
    def who_am_i(self):
        print('C')


class D(B, C):
    def who_am_i(self):
        print('D')


print(D.mro())
# [<class '__main__.D'>,
# <class '__main__.B'>,
# <class '__main__.C'>,
# <class '__main__.A'>,
# <class 'object'>]
```

Ваши вопросы

что необходимо прояснить в рамках данного раздела

Полиморфизм

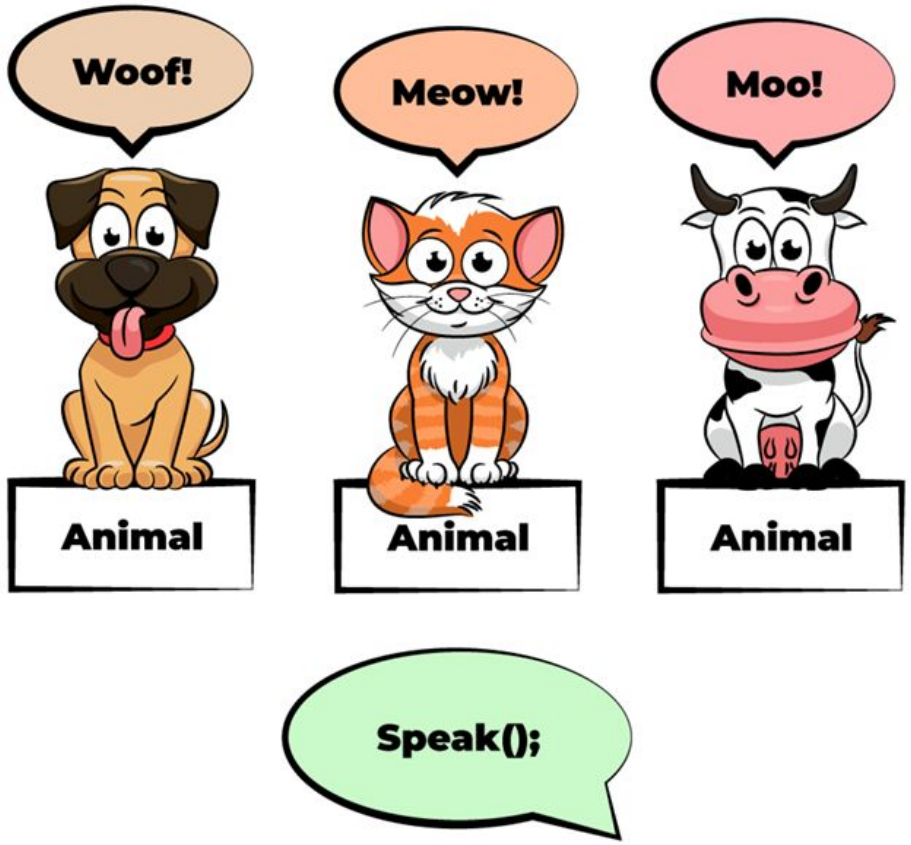
перегрузка, абстрактные класс и утиная типизация

Полиморфизм в Python

Полиморфизм — свойство системы, позволяющее иметь множество реализаций одного интерфейса (объекты разных классов, могут иметь одинаковые интерфейсы).

В Python имеется 2 возможности реализации полиморфизма:

- Классический полиморфизм: перегрузка и разное поведение
- Утиная типизация



В дочернем классе можно

- Добавить атрибуты
- Добавить методы
- Перегрузить (заменить) методы. Нужно для изменения способа работы некоторого метода родительского класса. Перегрузить можно **любые** методы

```python
class Character:
    def attack(self):
        print('Hit')


class Archer(Character):
    arrows: int

    def __init__(self, arrows=30):
        self.arrows = arrows

    def attack(self):
        print('Bow shot')

    def reload_arrows(self):
        self.arrows = 30


class Warrior(Character):
    def attack(self):
        print('Sword strike')
```

Утиная типизация

Python позволяет вызвать одинаковый метод для любых объектов (даже без общего родителя). Такое поведение называется **утиной типизацией**:

- «Если нечто выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка и есть»

Примеры утиной типизации в Python

- Класс Duck реализует метод swim_quack
- Класс RoboticBird реализует метод swim_quack
- Класс Fish реализует метод swim
- При вызове duck_testing с объектами классов Duck и RoboticBird будет выведено сообщение
- При вызове duck_testing с объектом класса Fish будет сгенерировано исключение

```python
class Duck:
    def swim_quack(self):
        print("I'm a duck, and I can swim and quack.")


class RoboticBird:
    def swim_quack(self):
        print("I'm a robotic bird, and I can swim and quack.")


class Fish:
    def swim(self):
        print("I'm a fish, and I can swim, but not quack.")


def duck_testing(animal):
    animal.swim_quack()
```

Примеры утиной типизации в Python

Пример применения функции len() к объектам различных типов

```python
class NamedDuck:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def __len__(self):
        return len(str(self))

ducks = [NamedDuck('Amazing'), 'OK', NamedDuck('Lucky'), 'Cool']
ducks.sort(key=len)

# ['OK', 'Cool', 'Lucky', 'Amazing']
print([str(x) for x in ducks])
```

В Python нет интерфейсов

Интерфейс — это совокупность методов и правил взаимодействия элементов системы. Другими словами, интерфейс определяет как элементы будут взаимодействовать между собой.

- Интерфейс это фактически регламент взаимодействия.
- Класс который реализует интерфейс обязан ре

```java
public interface Swimmable  {

     public void swim();
}

public class Duck implements Swimmable {

    public void swim() {
        System.out.println("Уточка, плыви!");
    }

    public static void main(String[] args) {

        Duck duck = new Duck();
        duck.swim();
    }
}

```

Абстрактные классы

Кроме обычных классов в некоторых языках существуют абстрактные классы (нельзя создать объект такого класса).

- Наряду с обычными методами содержит в себе абстрактные методы без имплементации (с сигнатурой, но без кода), которые обязан имплементировать класс-потомок

```python
from abc import ABC, abstractmethod


class Basic(ABC):
    @abstractmethod
    def hello(self):
        print("Hello from Basic class")


class Advanced(Basic):
    def hello(self):
        super().hello()
        print("Enriched functionality")


a = Advanced()
a.hello()
```

Абстрактные классы

Если в наследнике не будут реализованы все абстрактные методы и атрибуты, то при попытке создать экземпляр класса-наследника мы получим TypeError

```python
from abc import ABCMeta, abstractmethod


class BasePlugin(metaclass=ABCMeta):
    @property
    @abstractmethod
    def supported_formats(self) -> list:
        pass

    @abstractmethod
    def run(self, input_data):
        pass

    def install(self):
        raise NotImplementedError


class VideoPlugin(BasePlugin):
    def run(self, input_data):
        print('Processing video...')


plugin = VideoPlugin()
# TypeError: Can't instantiate abstract class VideoPlugin
# with abstract method supported_formats
```

Вопрос-ответ

? Зачем же нужен абстрактный класс, если его объект нельз создать?

✔ Он нужен для того, чтобы от него могли наследоваться потомки — обычные классы, объекты которых уже можно создавать, а также, чтобы потомки реализовывали те методы, которые определены в абстрактном классе

```python
class Charger(ABC):
    @abstractmethod
    def charge(self):
        pass

class USB(SerialPort, Charger):
    def read(self):
        return ""

    def write(self):
         pass


usb = USB()
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#TypeError: Can't instantiate abstract class USB with abstract methods charge
```

Преимущества использования

В манипулировании наследованием через интерфейс абстрактного класса есть некоторые преимущества:

- Клиентам не нужно знать о конкретных типах объектов, которыми он пользуется, при условии, что все они реализуют ожидаемый интерфейс.
- Клиенту достаточно знать только об абстрактном классе, определяющем интерфейс
- Помогают установить контракт, обязующий имплементировать определенный набор методов

Ваши вопросы

что необходимо прояснить в рамках данного раздела

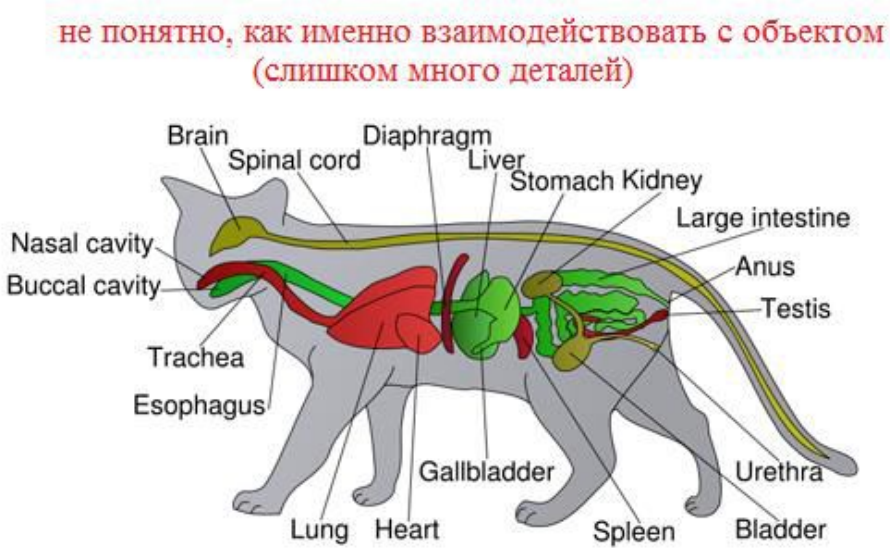
Инкапсуляция

объединение свойств и методов в единое целое, сокрытие данных

Инкапсуляция

Инкапсуляция понимается двояко:

- объединение свойств и поведения в единое целое, т. е. в класс. Инкапсуляция в этом смысле подразумевается самим определением объектно-ориентированного программирования и есть во всех ОО-языках
- сокрытие данных, то есть невозможность напрямую получить доступ к внутренней структуре объекта, так как это небезопасно



Инкапсуляция как контроль доступа

Инкапсуляция — это контроль доступа к полям и методам объекта. Под контролем доступа подразумевается не только можно/не можно, но и различные валидации, подгрузки, вычисления и прочее динамическое поведение.

Квалификаторы доступа

Отдельные объектно-ориентированные языки поддерживают закрытые атрибуты, к которым нельзя получить доступ непосредственно. Программистам зачастую приходится писать геттеры и сеттеры, чтобы прочитать или изменить значение атрибута

- public — к атрибуту может получить доступ любой желающий
- private — к атрибуту могут обращаться только методы данного класса
- protected — то же, что и private, только доступ получают и наследники класса в том числе

```
1  public class Cat
2  {
3      private String name;
4      private int age;
5      private int weight;
6
7      public Cat(String name, int age, int weight)
8      {
9          this.name = name;
10         this.age = age;
11         this.weight = weight;
12     }
13
14     public String getName()
15     {
16         return name;
17     }
18
19     private void setName(String name)
20     {
21         this.name = name;
22     }
23
24     public int getAge()
25     {
26         return age;
27     }
28
29     private void setAge(int age)
30     {
31         this.age = age;
32     }
33 }
```

Искажение имен для безопасности

В Python не существует квалификаторов доступа к полям класса. Отсутствие аналогов связки public/private/protected можно рассматривать как упущение со стороны принципа инкапсуляции.

В Python есть соглашение по именованию private и protected атрибутов

- «_имя» — protected
- «__имя» — protected. Этот метод не делает атрибут закрытым, но имя искажается, чтобы внешний код на него не наткнулся. По секрету: к скрытому методу можно обратиться как obj._ClassName__name

```python
class A:
    _protected = 0
    __private = 1


class B(A):
    def what_i_see(self):
        print(self._protected)


a = A()
print(a._protected)  # OK
print(a.__private)  # Error
print(a._A__private)  # OK
```

Аксессоры (геттеры и сеттеры)

В некоторых языках есть синтаксический сахар, позволяющий аксессоры маскировать под свойства, что делает доступ прозрачным для внешнего кода, который и не подозревает, что работает не с полем, а с методом, у которого под капотом выполняется SQL-запрос или чтение из файла.

В Python геттеры и сеттеры не нужны, так как все атрибуты и методы являются открытыми. Если нам нужно обеспечить сокрытие атрибута, то лучше воспользоваться декоратором property.

- @property размещается перед геттером
- @name.setter размещается перед сеттером

```python
class A:
    __hidden: str

    @property
    def hidden(self):
        return self.__hidden

    @hidden.setter
    def hidden(self, val):
        self.__hidden = val

    def __init__(self, val):
        self.__hidden = val


a = A('value')
print(a.hidden)
a.hidden = 123
print(a.__dict__)
```

Ваши вопросы

что необходимо прояснить в рамках данного раздела

Погружаемся глубже

Еще один взрыв мозга

Декораторы класса

Аналогично декораторам функций, декоратор класса призван модифицировать поведение и содержание класса, не изменяя его исходный код. Похоже на наследование, но есть отличия:

1. Сильные возможности кастомизации: может удалять, добавлять, менять, переименовывать атрибуты и методы класса. Он может возвращать совершенно другой класс
2. Исходный класс «затирается» и не может быть использован как базовый класс при полиморфизме.
3. Декорировать можно любой класс одним и тем же универсальный декоратором, а при наследовании – мы ограничены иерархией классов и должны считаться с интерфейсами базовых классов
4. Презираются все принципы и ограничения ООП (из-за пунктов 1-3)

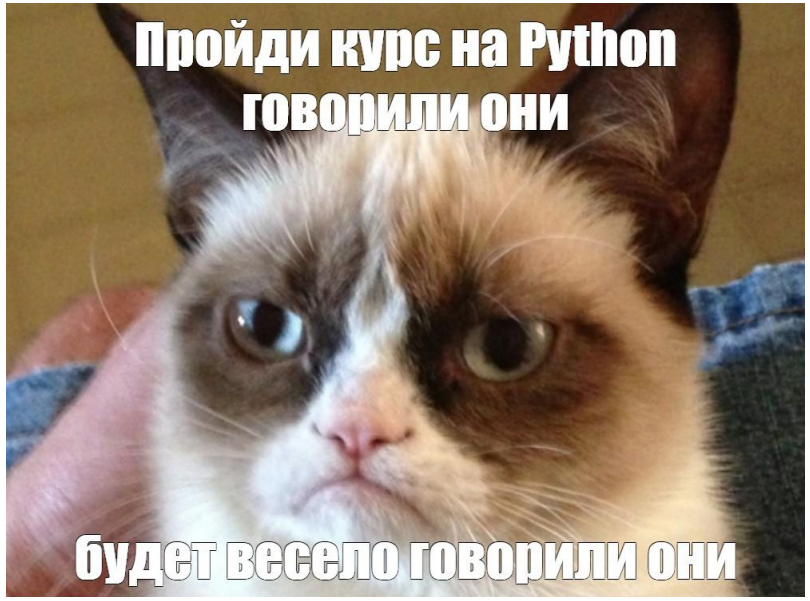
```python
def func_dec(func):
    def wrapper(*args, **kwargs):
        print(f'Run {func.__name__}')
        result = func(*args, **kwargs)
        return result
    return wrapper


def class_decorator(cls):
    call_attr = {k: v for k, v in cls.__dict__.items() if callable(v)}
    for name, val in call_attr.items():
        decorated = func_dec(val)
        setattr(cls, name, decorated)
    return cls


@class_decorator
class SomeClass:
    def __init__(self):
        pass

    def say_hi(self):
        print('Hi')


obj = SomeClass()
obj.say_hi()
```

Немного про классы

В Python все является объектами. Все они являются либо экземплярами классов, либо экземплярами метаклассов. За исключением type. type – сам себе метакласс.

Классы в Python – объекты, а это значит:

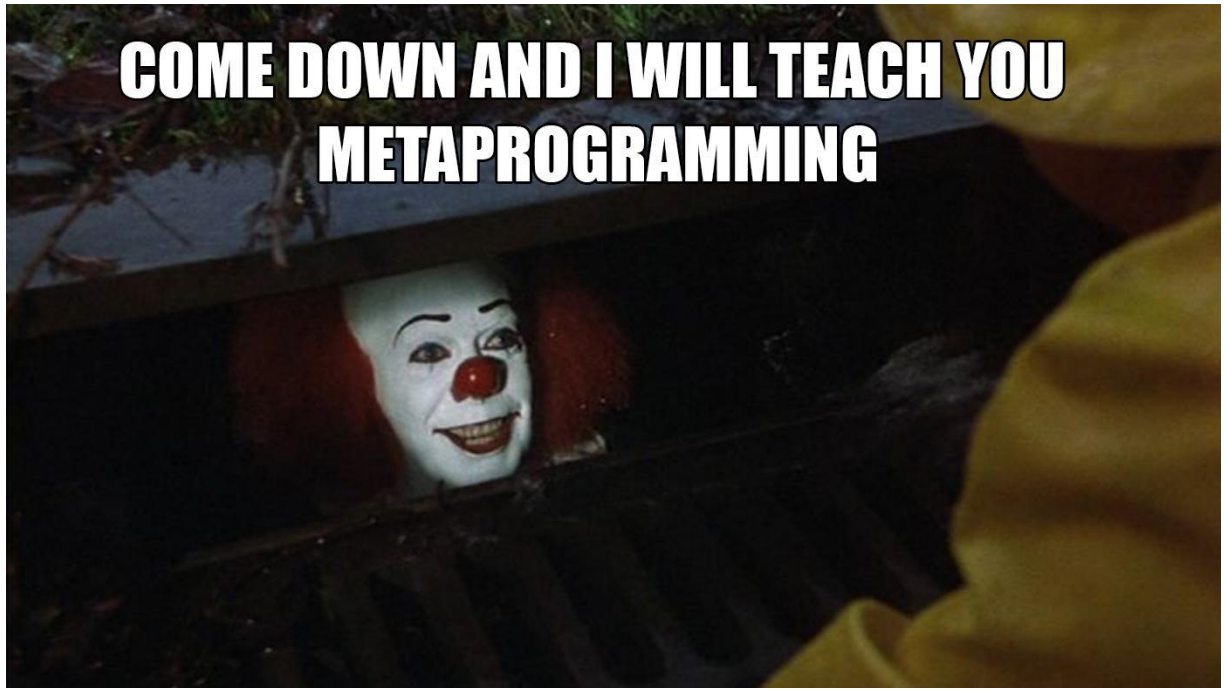
- Можно назначить в качестве переменной
- Можно скопировать
- Можно добавить к нему атрибут
- передается в качестве параметра функции



Метаклассы

"Метаклассы – это магия, о которой 99% пользователей не стоит даже задумываться. Если вам интересно, нужны ли они вам – тогда точно нет. Люди, которым они на самом деле нужны, знают, зачем, и что с ними делать."

© Гуру Python Tim Peters



Что такое метакласс

Класс- шаблон для объектов, метакласс - шаблон для классов.

Метаклассы – это классы, экземпляры которых являются классами.

Можно создать метакласс, унаследованный от метакласса, унаследованного от type.

```
1  MyClass = MetaClass()
2  my_object = MyClass()
```

Для чего используются метаклассы

С помощью метаклассов можно:

- перехватить создание класса
- изменить класс
- вернуть измененный класс

с помощью метаклассов можно относительно проще реализовать сложные паттерны проектирования

```python
class LittleMeta(type):
    def __new__(cls, clsname, superclasses, attributedict):
        print("clsname: ", clsname)
        print("superclasses: ", superclasses)
        print("attributedict: ", attributedict)
        return super().__new__(cls, clsname, superclasses, attributedict)


class S:
    pass


class A(S, metaclass=LittleMeta):
    some_attr = 0
```

Немного о type

- Самым простым метаклассом можно считать type
- Когда type получает на вход один параметр, он возвращает тип объекта, переданного в качестве параметра. В данном случае он не работает как метакласс
- Когда type получает на вход три параметра, он работает как метакласс и создаёт класс на основе переданных параметров. В качестве параметров должны передаваться имя класса, родители (классы, от которых происходит наследование), словарь атрибутов.

```
>>> type('hello')
<class 'str'>
>>> some_class = type('RuntimeClass', (), {'field': 'value'})
>>> some_class
<class '__main__.RuntimeClass'>
>>> some_class.field
'value'
```

Метаклассы это сложно

Причина сложности кода, использующего метаклассы, заключается не в самих метаклассах. Код сложный потому, что обычно метаклассы **используются для сложных задач**, основанных на наследовании, интроспекции (доступ к метаданным классов) и манипуляции такими переменными, как __dict__

```
def upper_attr(future_class_name, future_class_parents, future_class_attr):
    """
    Return a class object, with the list of its attribute turned
    into uppercase.
    """
    # pick up any attribute that doesn't start with '__' and uppercase it
    uppercase_attr = {}
    for name, val in future_class_attr.items():
        if not name.startswith('__'):
            uppercase_attr[name.upper()] = val
        else:
            uppercase_attr[name] = val


class UpperAttrMetaclass(type):
    def __new__(cls, clsname, bases, dct):
        uppercase_attr = {}
        for name, val in dct.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val
        return super(UpperAttrMetaclass, cls).__new__(cls, clsname, bases, uppercase_attr)
```

Метаклассы это сложно

Если вам не нужны сложные изменения класса, метаклассы использовать не стоит. Просто изменить класс можно двумя способами:

- Руками
- Декораторами класса

В 99% случаев лучше использовать эти методы, а в 98% изменения класса вообще не нужны.

```python
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(
                Singleton, cls
            ).__call__(*args, **kwargs)
        return cls._instances[cls]


class SingletonClass(metaclass=Singleton):
    pass


class RegularClass:
    pass


x = SingletonClass()
y = SingletonClass()
print(x == y)

x = RegularClass()
y = RegularClass()
print(x == y)
```

Описание параметров

- mcs – объект метакласса, например <__main__.MetaClass>
- name – строка, имя класса, для которого используется метакласс, например «User»
- bases – кортеж из классов-родителей, например (SomeMixin, AbstractUser)
- attrs – dict-like объект, хранит в себе значения атрибутов и методов класса
- cls – созданный класс, например <__main__.User>
- extra_kwargs – дополнительные keyword-аргументы переданные в сигнатуру класса
- args и kwargs – аргументы переданные в конструктор класса при создании нового экземпляра

```python
class MetaClass(type):
    def __new__(mcs, name, bases, attrs, **extra_kwargs):
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **extra_kwargs):
        super().__init__(cls)

    @classmethod
    def __prepare__(mcs, name, bases, **extra_kwargs):
        return super().__prepare__(mcs, name, bases, extra_kwargs)

    def __call__(cls, *args, **kwargs):
        return super().__call__(*args, **kwargs)


class User(metaclass=MetaClass):
    def __init__(self, name):
        self.name = name
```

Как работает метакласс

Порядок, в котором интерпретатор Python вызывает метаметоды метакласса в момент создания самого класса. При обработке определения класса Python:

1. Python определяет и находит классы-родители для текущего класса (если они есть). Они передаются кортежем в bases
2. Интерпретатор определяет метакласс (MetaClass в нашем случае)
3. Вызывается метод MetaClass.__prepare__. Возвратить dict-like объект, в который будут записаны атрибуты и методы класса. После этого объект будет передан в метод MetaClass.__new__ через аргумент attrs
4. Интерпретатор читает тело класса User и формирует параметры для передачи их в метакласс MetaClass
5. Вызывается MetaClass.__new__. Возвращает созданный объект класса. Если тип аргумента attrs был изменен с помощью __prepare__, то его необходимо конвертировать в dict, прежде чем передать в вызов метода super()
6. Вызывается MetaClass.__init__, с помощью которого можно добавить дополнительные атрибуты и методы в объект класса. На практике используется в случаях, когда метаклассы наследуются от других метаклассов, в остальном все что можно сделать в __init__, лучше сделать в __new__. Например параметр __slots__ можно задать только в методе __new__, записав его в объект attrs.
7. На этом шаге класс считается созданным

```python
class MetaClass(type):
    def __new__(mcs, name, bases, attrs, **extra_kwargs):
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **extra_kwargs):
        super().__init__(cls)

    @classmethod
    def __prepare__(mcs, name, bases, **extra_kwargs):
        return super().__prepare__(mcs, name, bases, extra_kwargs)

    def __call__(cls, *args, **kwargs):
        return super().__call__(*args, **kwargs)


class User(metaclass=MetaClass):
    def __init__(self, name):
        self.name = name
```

Когда создаем объект

1. В момент вызова User(...) интерпретатор вызывает метод MetaClass.__call__(name='Alyosha'), куда передает объект класса и переданные аргументы
2. MetaClass.__call__ вызывает User.__new__(name='Alyosha') – метод-конструктор, который создает и возвращает экземпляр класса User
3. Далее MetaClass.__call__ вызывает User.__init__(name='Alyosha') – метод-инициализатор, который добавляет новые атрибуты к созданному экземпляру
4. MetaClass.__call__ возвращает созданный и проинициализированный экземпляр класса User
5. В этот момент экземпляр класса считается созданным

```python
class MetaClass(type):
    def __new__(mcs, name, bases, attrs, **extra_kwargs):
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **extra_kwargs):
        super().__init__(cls)

    @classmethod
    def __prepare__(mcs, name, bases, **extra_kwargs):
        return super().__prepare__(mcs, name, bases, extra_kwargs)

    def __call__(cls, *args, **kwargs):
        return super().__call__(*args, **kwargs)


class User(metaclass=MetaClass):
    def __init__(self, name):
        self.name = name
```

Абстрактный метакласс

Если мы хотим создать такой метакласс, чтобы класс, который указывает данный метакласс в качестве своего метакласса мог быть абстрактным, то метакласс можно отнаследовать от ABCMeta

```python
from abc import ABCMeta
from typing import Optional

from .field import Field
from .serialize import SerializableModel


class ModelMeta(ABCMeta):
    """
    Metaclass for creation api classes.
    Adds _fields and _aliases attributes.
    """

    def __new__(mcs, name, bases, attr):
        new_class = super().__new__(mcs, name, bases, attr)

        fields = set()
        aliases = {}

        for parent in bases:
            if isinstance(parent, ModelMeta):
                fields.update(getattr(parent, '_fields'))
                aliases.update(getattr(parent, '_aliases'))

        for key, value in attr.items():
            if isinstance(value, Field):
                fields.add(key)
                if value.alias is not None:
                    aliases[key] = value.alias
                if value.self_base:
                    value.base = new_class

        setattr(new_class, '_aliases', aliases)
        setattr(new_class, '_fields', fields)
        return new_class
```