


```
class Star:
```

```
    def __init__(self, name, galaxy):
```

```
        self.name = name
```

```
        self.galaxy = galaxy
```

```
sun = Star("Солнце", "Млечный путь")
```

```
print(sun)
```

```
<__main__.Star object at 0x0000023A05392188>
```

Адрес объекта в памяти

Когда **Python** нужно вывести класс или объект в виде строки (помещение объекта в качестве аргумента в вызов функции **print()** отвечает этому условию), он пытается вызвать из объекта метод с именем **__str__()** и использовать строку, которую он возвращает.

Метод по умолчанию **__str__()** возвращает строку, которую мы видели на предыдущем слайде, только адрес, обычно, меняется. Но вряд ли пользователю нужно видеть адрес объекта в памяти.

Однако, мы можем в классе переопределить метод `__str__()` для нашего класса.

```
class Star:
```

```
    def __init__(self, name, galaxy):
```

```
        self.name = name
```

```
        self.galaxy = galaxy
```

```
    def __str__(self):
```

```
        return self.name + ' в галактике ' + self.galaxy
```

```
sun = Star("Солнце", "Млечный путь")
```

```
print(sun)
```

```
Солнце в галактике: Млечный путь
```


Наследование - это базовая концепция ООП, позволяющая передавать атрибуты и методы из суперкласса (родительского класса, уже определенного и существующего) в созданный класс, который называется подклассом.

Наследование – способствует структурированию и повторному использованию кода.

Для того, чтобы определить принадлежит ли объект определённому классу или нет, в Python существует специальная функция: **isinstance()**

Эта функция возвращает **True**, если объект является экземпляром класса, в противном случае — **False**.

Напишем следующий код:

```
class Vehicle:
```

```
    pass
```

```
class LandVehicle(Vehicle):
```

```
    pass
```

```
class TrackedVehicle(LandVehicle):
```

```
    pass
```

```
obj = Vehicle()
```

```
print('obj -> Vehicle:', isinstance(obj, Vehicle))
```

```
print('obj -> LandVehicle:', isinstance(obj, LandVehicle))
```

```
print('obj -> TrackedVehicle:', isinstance(obj, TrackedVehicle))
```

```
print()
```


Напишем следующий код:

```
obj = LandVehicle()  
print('obj -> Vehicle:', isinstance(obj, Vehicle))  
print('obj -> LandVehicle:', isinstance(obj, LandVehicle))  
print('obj -> TrackedVehicle:', isinstance(obj, TrackedVehicle))  
print()
```

```
obj = TrackedVehicle()  
print('obj -> Vehicle:', isinstance(obj, Vehicle))  
print('obj -> LandVehicle:', isinstance(obj, LandVehicle))  
print('obj -> TrackedVehicle:', isinstance(obj, TrackedVehicle))  
print()
```

Напомню, что переменные-объекты – содержат ссылку (адрес), на область памяти, где хранятся значения полей объекта.

Таким образом, две разные переменные, могут указывать на один и тот же объект.

Оператор **is** относятся ли две переменные к одному и тому же объекту или нет, если да – то **is** вернёт значение **True**, в противном случае — **False**.

Бывают случаи, когда оператор **is** очень полезен.

```
class Test:  
    def __init__(self, a):  
        self.a = a
```

```
obj1 = Test(1)  
obj2 = obj1  
obj3 = Test(2)
```

```
print('obj1 = obj2 ?', obj1 is obj2)  
print('obj2 = obj3 ?', obj2 is obj3)  
print('obj1 = obj3 ?', obj1 is obj3)
```

```
print ('obj1.a = ', obj1.a)  
print ('obj2.a = ', obj2.a)  
print ('obj3.a = ', obj3.a)
```

```
obj1 = obj2 ? True
```

```
obj2 = obj3 ? False
```

```
obj1 = obj2 ? False
```

```
obj1.a = 1
```

```
obj2.a = 1
```

```
obj3.a = 2
```


Используя функцию **super()**, мы можем в конструкторе подкласса использовать конструктор родительского класса:

```
class Parent:  
    def __init__(self, a):  
        self.a = a
```

```
class SubClass(Parent):  
    def __init__(self, a, b):  
        super().__init__(a)  
        self.b = b
```

```
obj = SubClass(2, 6)  
print('obj.a = ', obj.a)  
print('obj.b = ', obj.b)
```

При попытке получить доступ к объекту какого-нибудь объекта Python попытается сделать следующее (именно в этом порядке):

- найти его внутри самого объекта;
- найти его во всех классах, которые участвуют в наследовании объекта снизу вверх.

Если вышеперечисленные действия не дадут результата, будет сгенерирована ошибка (`AttributeError`).

Наследование становится множественным, когда у класса появляется больше одного суперкласса. Синтаксически такое наследование представлено в виде списка суперклассов через запятую, в круглых скобках, которые следуют за новым именем класса.

```
class SuperA:  
    var_a = 10  
    def func_a(self):  
        return 11
```

```
class SuperB:  
    var_b = 20  
    def func_b(self):  
        return 21
```

```
class Sub(SuperA, SuperB):  
    pass
```

```
obj = Sub()
```

```
print(obj.var_a, obj.func_a())  
print(obj.var_b, obj.func_b())
```

```
class Sub(SuperA, SuperB):  
    pass
```

```
obj = Sub()
```

```
print(obj.var_a, obj.func_a())  
print(obj.var_b, obj.func_b())
```

Мы видим, что объект подкласса унаследовал все атрибуты обоих классов.

Что произойдёт, если несколько родительских класса имеют атрибуты с одинаковым наименованием?

```
class SuperA:  
    var = 10  
    def func(self):  
        return 11
```

```
class SuperB:  
    var = 20  
    def func(self):  
        return 21
```

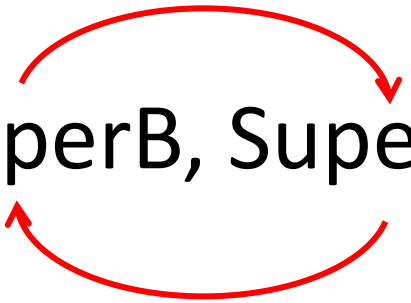
```
class Sub(SuperA, SuperB):  
    pass
```

```
obj = Sub()
```

```
print(obj.var, obj.func())
```



```
class Sub(SuperB, SuperA):  
    pass
```



```
obj = Sub()
```


```
print(obj.var, obj.func())
```


20 21

```
class Sub(SuperB, SuperA):  
    var = 30  
    def func(self):  
        return 33
```

```
obj = Sub()
```

```
print(obj.var, obj.func())
```



1001

Можно заключить, что Python ищет компоненты объекта в следующем порядке:

- внутри самого объекта;
- в его суперклассе, снизу вверх;
- если в конкретной линии наследования более одного класса, Python сканирует их слева направо.

РОМБЫ

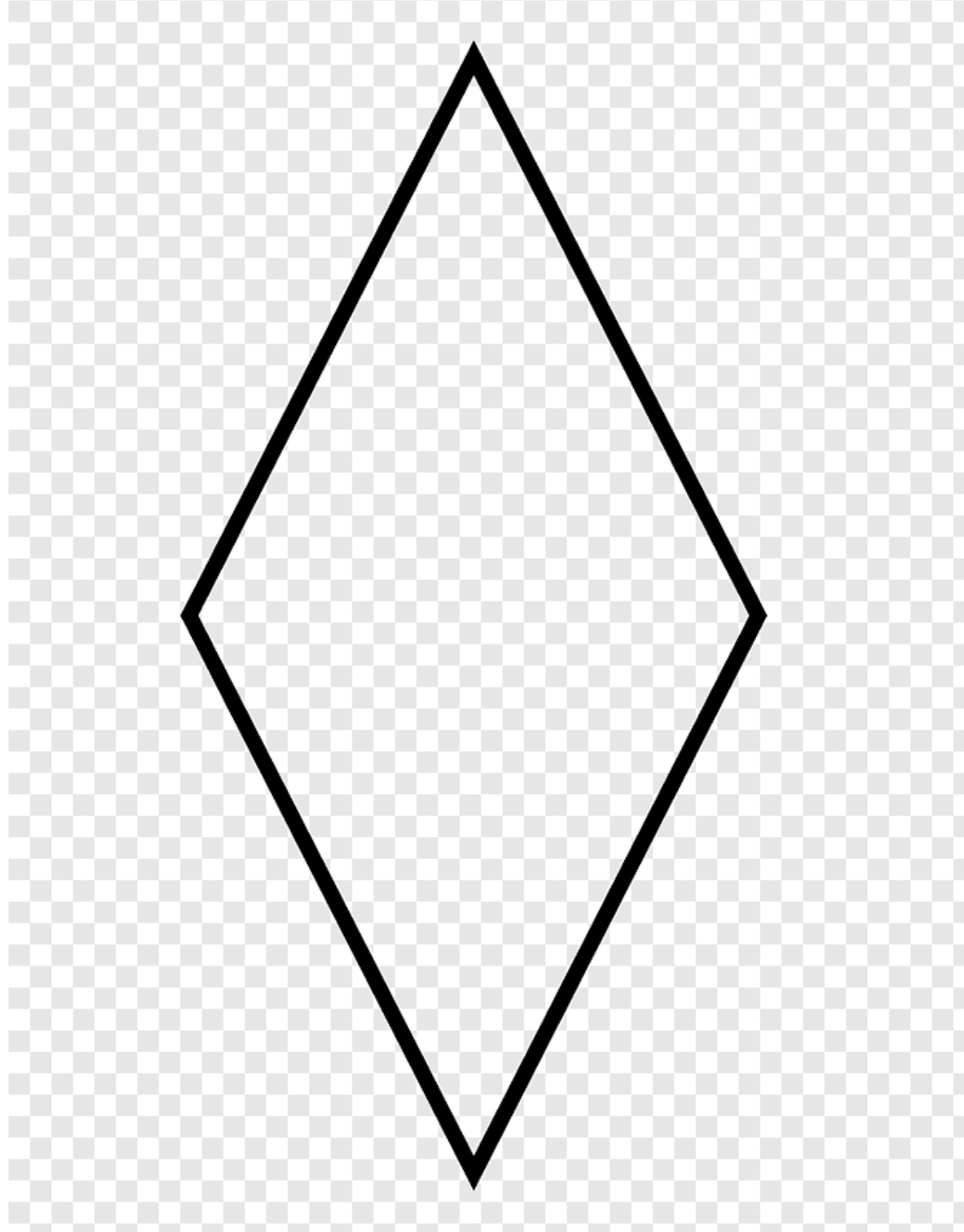
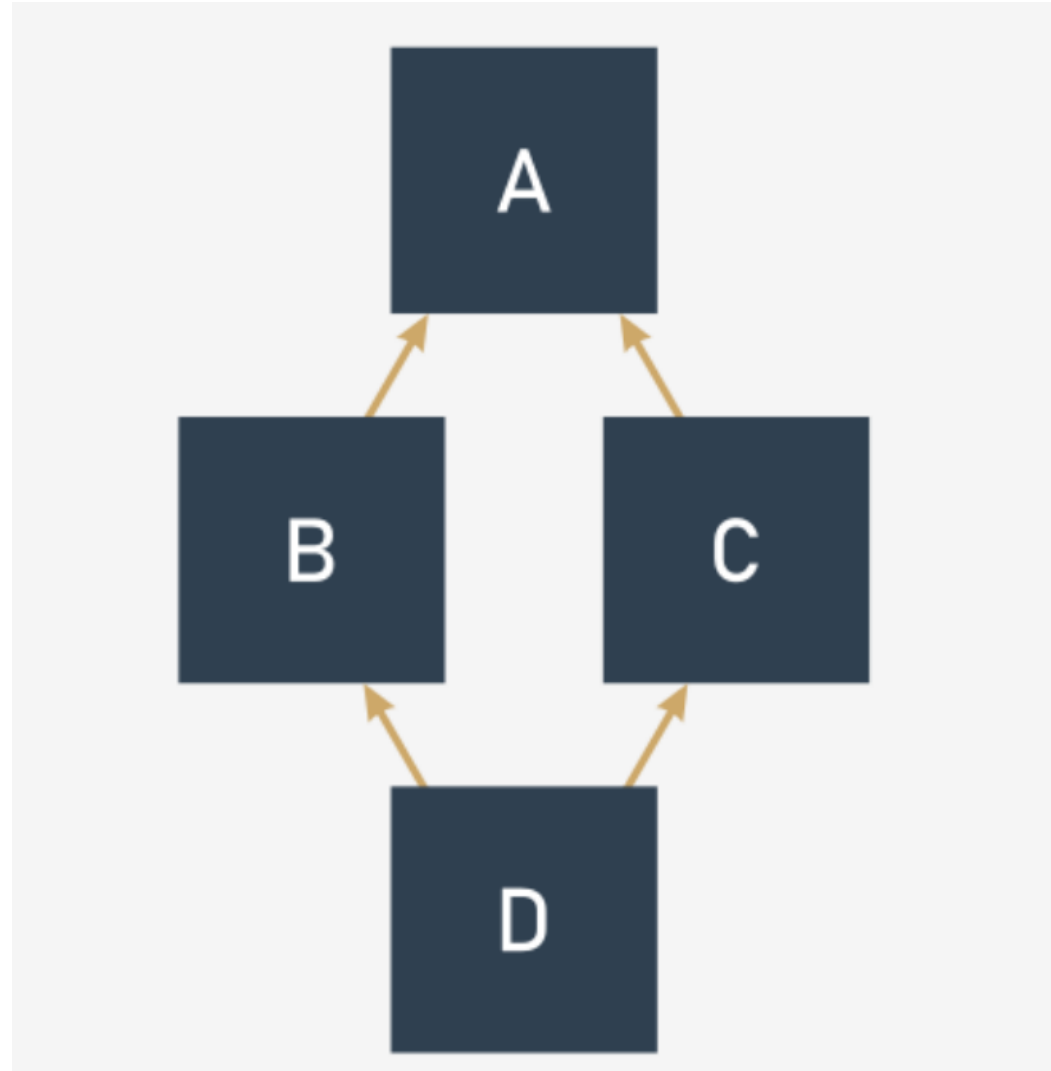


Диаграмма наследования в форме РОМБА.



```
class A:
```

```
    pass
```

```
class B(A):
```

```
    pass
```

```
class C(A):
```

```
    pass
```

```
class D(C, B):
```

```
    pass
```

```
d = D()
```



```
-----  
Traceback (most recent call last):  
  File "C:\Users\БулгунаевБ\Documents\python projects\test.py", line 8, in <module>  
    class D(A, B):  
TypeError: Cannot create a consistent method resolution  
order (MRO) for bases A, B  
>>> |
```

Python не любит ромбы и не позволяет их создавать!