

## Интерфейсы

- Интерфейс - это логически взаимосвязанный набор операций, которые можно проделать над типом
- Классы реализуют интерфейсы
- Интерфейс должен формировать API, достаточной для полноценной работы над данными без прямого доступа к ним

```
1      x = X()
2      x.pos = (1, 2)
3      print x.pos
4
5      x1 = X1()
6      x1.setPos((1, 2))
7      print x1.getPos()
```

## Интерфейсы

- Вообще в питоне интерфейсы в явном виде используются редко, так как плохо совместимы с утиной типизацией (но см. модуль ABC)

## Класс

- Класс - конкретная реализация интерфейса(ов)

## Объект и инстанцирование

- Объект - экземпляр класса. Хранилище данных.
- Объект получается путем инстанцирования класса. `obj = Class (...)`
- Одновременно может существовать неограниченное количество экземпляров класса, все они независимы

## Пример класса на python

```
1  class Simple(object):
2      "class documentation"
3      def set_power(self, power):
4          self.power = power
5
6      def method(self, val):
7          "method documentation"
8          return val ** self.power
9
10 simple = Simple()
11 simple.set_power(1.0 / 3)
12 simple.method(8) == 2.0 # not the best idea, but works
```

## Поля класса

- Поля это переменные, связанные с экземпляром класса
- Класс явно не определяет какие поля будут у его объектов
- Доступ производится с помощью оператора '.'
- Как и переменные они создаются присваиванием
- = Можно динамически создавать/удалять поля у объекта
- = Разные экземпляры могут иметь разные наборы полей
- Предыдущие два пункта чаще всего - пример плохого поведения

```
1     simple = Simple()  
2     simple.a = 12  
3     print simple.a  
4     simple.a += 4
```

## Методы класса

- Методы это функции, объявленные внутри тела класса
- Должны вызываться только от экземпляра класса
- Первым параметром метод автоматически получает экземпляр, от которого вызван

```
1    Simple.method(12) # ошибка
2    simple = Simple()
3    simple.method(12)
```



## Нет инкапсуляции

- "Принцип открытого кимоно" == "мы не знаем как совместить инкапсуляцию с остальными возможностями языка"
- Есть property, "скрытые поля" (но они предназначены для другого)
- В отличие от Java всегда можно изменить поле на свойство с сохранением совместимости
- Можно реализовать любой вид инкапсуляции динамической
- Ничего из этого не получило какого-либо распространения в питоне
- Документирование API vs чтение заголовков
- Реализуя сокрытие полей объекта вы создаете проблемы многим библиотекам python

## Наследование

- У класса может быть один или несколько родительских классов, класс автоматически получает все методы, которые были у его родителей и может добавить дополнительные
- Поддерживается множественное наследование
- Которое не надо использовать, если на то нет существенных причин

```
1     class A( object ):
2         def some_method( self ):
3             print "A.some_meth "
4
5     class B(A):
6         def some_method2( self ):
7             print "B.some_meth2 "
8
9     b = B()
10    b.some_method2 () # B.some_method2 called
11    b.some_method () # A.some_meth called
```

## Полиморфизм

- Наследник может изменить поведение методов предка
- Все методы - виртуальные

```
1     class A(object):
2         def some_method(self):
3             print "A.some_meth"
4
5     class B(A):
6         def some_method(self):
7             print "B.some_meth"
8
9     a = A()
10    a.some_method() # A.some_meth
11
12    b = B()
13    b.some_method() # B.some_meth
```

## Вызов метода базового класса

```
1  class A(object):
2      def some_method(self, val):
3          print "A.some_method({!r})".format(val)
4
5  class B(A):
6      def some_method(self, val):
7          print "B.some_method({!r})".format(val)
8          A.some_method(self, val)
9
10 b = B()
11 b.some_method(1)
12 # B.some_method(1)
13 # A.some_method(1)
```

## super

- Прямой метод хорошо работает при одиночном наследовании
- `super(CurrentClass, self).method` позволяет корректно вызывать метод у базового класса
- Даже в случае множественного наследования вызывать `super` нужно только один раз
- `super` использует линейаризацию - упорядочивание иерархии базовых классов в последовательность
- [Подробное описание линейаризации.](#)

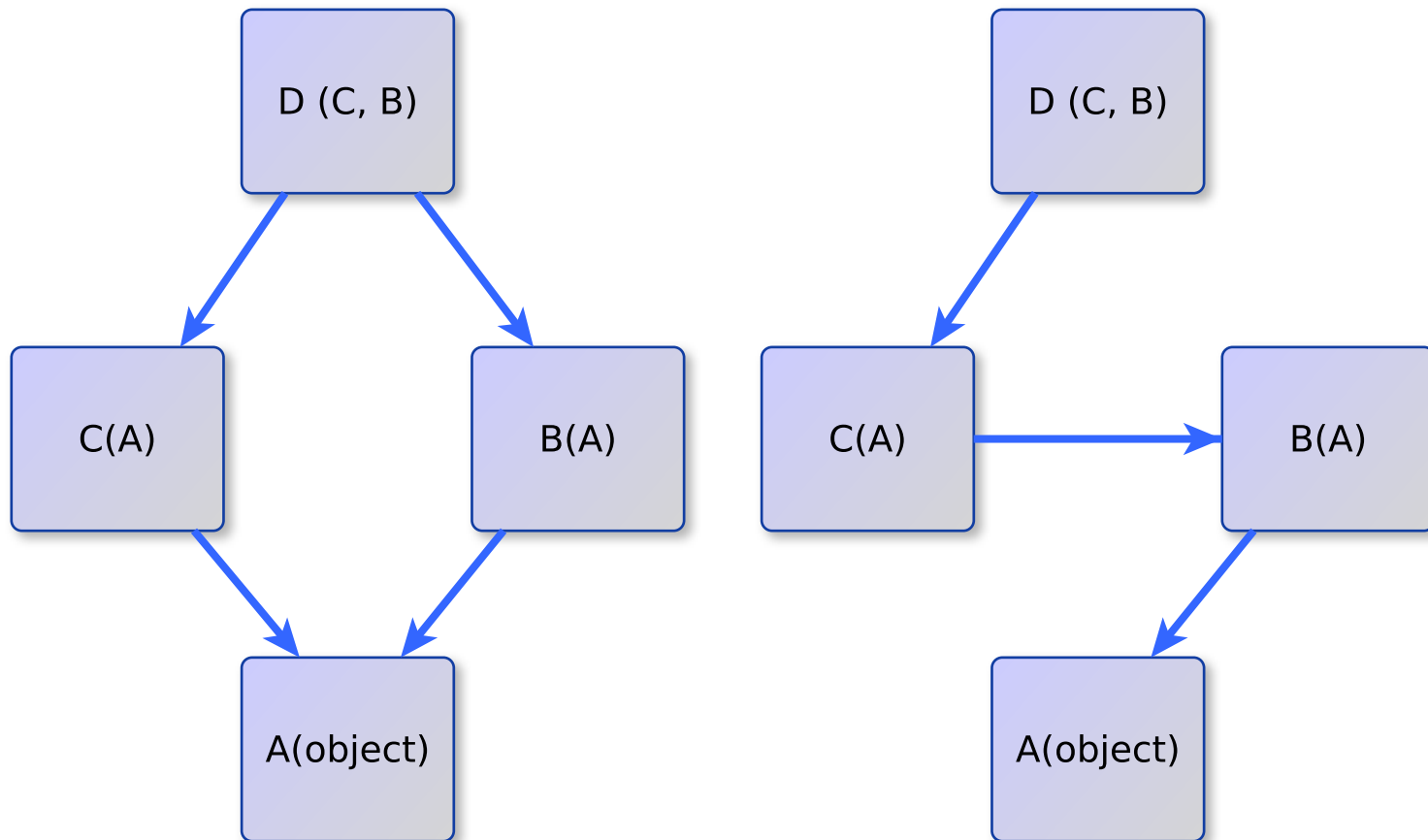
## Прямой вызов при множественном наследовании

```
1  class A( object ):
2      def draw( self , pt ):
3          some_action( pt )
4
5  class B(A):
6      def draw( self , pt ):
7          A.draw( pt )
8
9  class C(A):
10     def draw( self , pt ):
11         A.draw( pt )
12
13  class D(C, B):
14     def draw( self , pt ):
15         C.draw( pt )
16         B.draw( pt )
```

## super при множественном наследовании

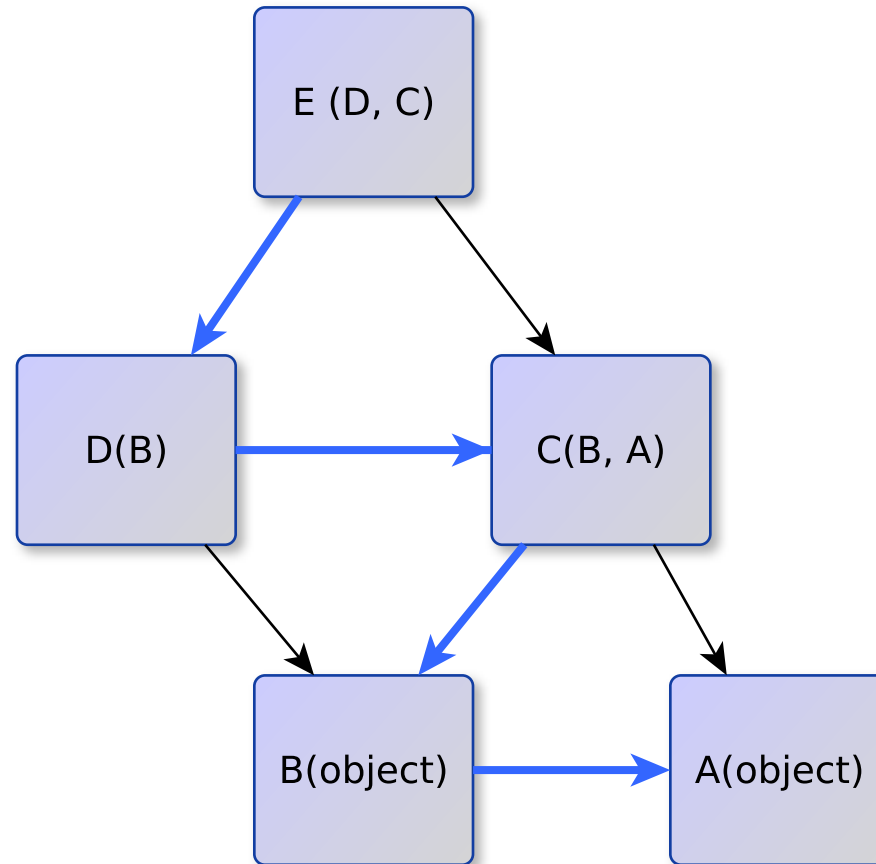
```
1  class A(object):
2      def draw(self, pt):
3          some_action(pt)
4
5  class B(A):
6      def draw(self, pt):
7          super(B, self).draw(pt)
8
9  class C(A):
10     def draw(self, pt):
11         super(C, self).draw(pt)
12
13 class D(C, B):
14     def draw(self, pt):
15         super(D, self).draw(pt)
```

## direct call VS super

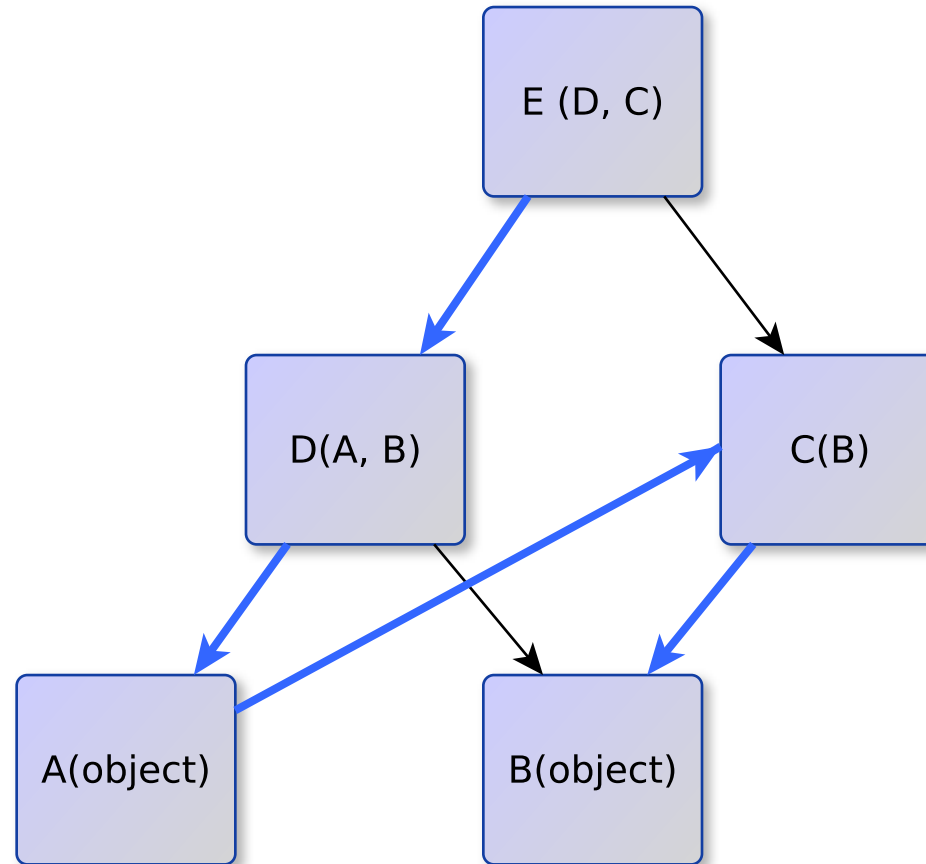




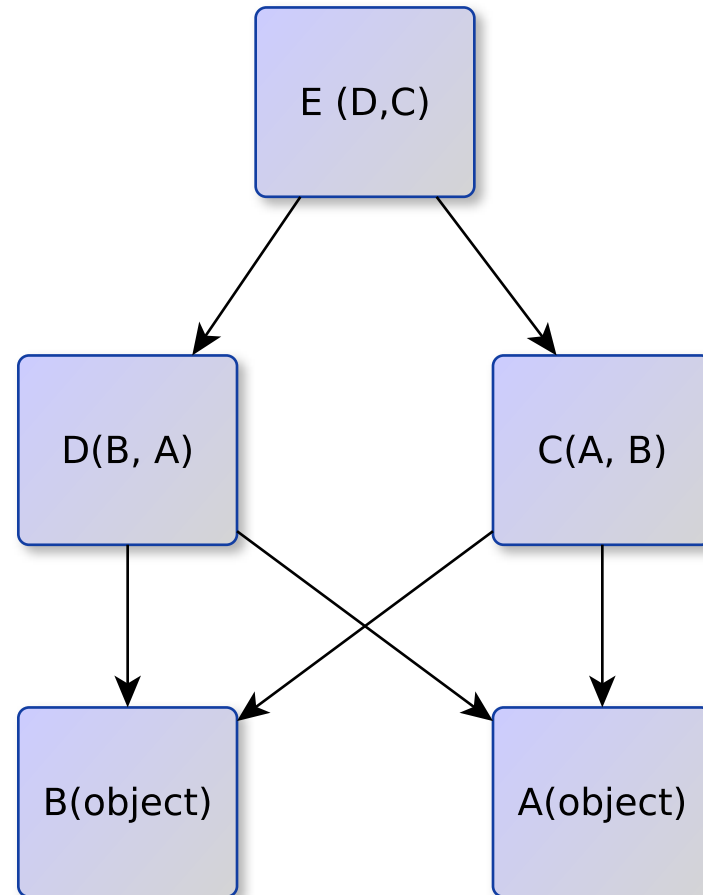
## Пример линеаризации



## Пример линеаризации 2

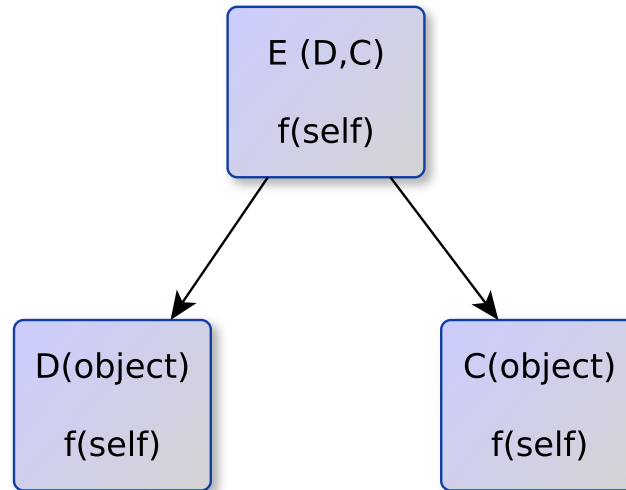


## Нелинеаризуемая иерархия классов



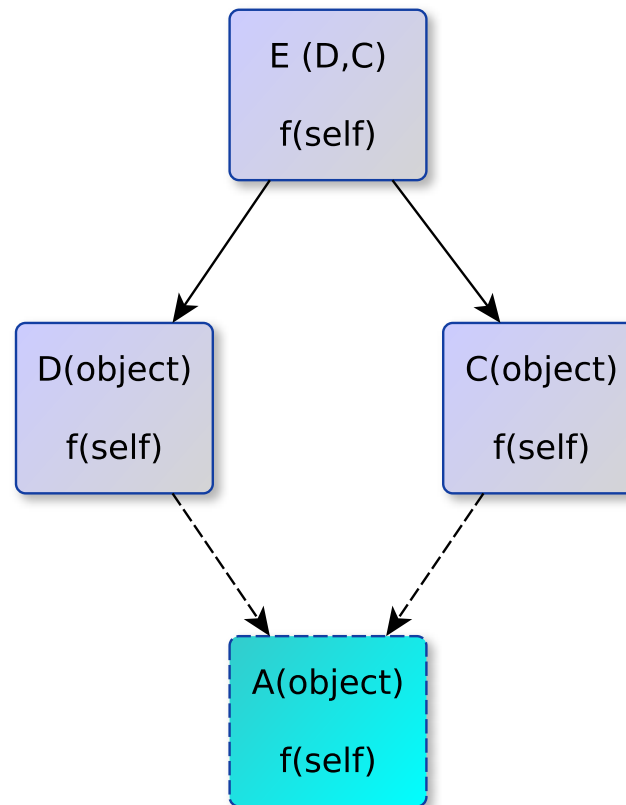
## Ошибки при построении иерархии

Если имена совпали случайно, то нельзя использовать `super` в `E`



## Ошибки при построении иерархии

Если имена совпали не случайно, то нужно добавить базовый класс и super будет работать нормально



## Конструктор `__init__`

- `__init__` (конструктор) - метод, который автоматически вызывается при создании экземпляра класса и должен его проинициализировать - присвоить всем переменным начальные значения.
- К сожалению `__init__` это типичный пример ошибки

```
1     class A(object):
2         def __init__(self, val):
3             self.msg = "val = {}".format(val)
4             print "A inited with value", val
5
6     a = A(1) # A inited with value 1
7     print a.msg # val = 1
```

## Деструктор `__del__`

- `__del__` должен вызываться перед удалением объекта (и, обычно, вызывается).
- Если нет циклических ссылок
- Или объект не попал во фрейм, где произошла ошибка
- Правильно - использовать `with` вместо надежд на `__del__`
- `def __del__( self ):`...

## Аллокатор `__new__`

- `__new__` аналог перегрузки `new` в C++
- Классовый метод(автоматически), вызываемый для создания нового экземпляра объекта, который затем будет проинициализирован с помощью `__init__`. `classmethod` использовать не надо.
- Получает те-же параметры, что и `__init__`.



## Аллокатор \_\_new\_\_

```
1  class X(object):
2      def __new__(cls, val):
3          print "{}.__new__({!r})".format(
4              cls.__name__, val)
5          return object.__new__(cls, val)
6
7      def __init__(self, val):
8          print "{}.__init__({!r})".format(
9              self.__class__.__name__, val)
10
11  X(1)
12  #X.__new__(1)
13  #X.__init__(1)
```

## Аллокатор \_\_new\_\_

- \_\_new\_\_ может вернуть объект другого типа
- Наверное, не самая лучшая идея

```
1     class A(object):
2         def __init__(self, x):
3             pass
4
5     class B(object):
6         def __init__(self, x, y):
7             pass
8         def __new__(cls, x, y=None):
9             if y is None:
10                 return A(x)
11             else:
12                 return super(B, cls).__new__(cls, x, y)
13
14     print B(1, 2) #<__main__.B at 0x...>
15     print B(1)  # <__main__.A at 0x...>
```

## isinstance, isinstance

- `isinstance` (x, Y) - проверяет, что x экземпляр Y или одного из классов, наследованных от Y
- `isinstance` (x, (Y1, Y2, ..., YN))
- `issubclass` (X, Y) - проверяет, что X is Y или X прямо или косвенно наследует Y.
- `issubclass` (X, (Y1, Y2, ..., YN))

```
1     isinstance(1, int) == True
2     isinstance(2.0, int) == False
3     isinstance("as", (str, unicode)) == True
4     issubclass(int, int) == True
```

## Скрытие полей

- \_\_xxxx - "скрытые поля и методы" - переименовываются для избежания пересечения имен
- A.\_\_xxx -> A.\_A\_\_xxx
- Не для инкапсуляции

## Связанные и не связанные методы

- `a = A(); a.b(1) == A.b(a, 1)`
- `A.b` несвязанный метод. Требует получения экземпляра `A` первым параметром.
- `A.b(1)` - ошибка.
- `(A.b)(a, 1)` - ok
- `a.b` - связанный метод, эквивалентен функции.

```
1      a = A()  
2      t = a.b  
3      t(1) # a.b(1)
```

## Классовые и статические методы

- `classmethod` превращает метод в классový, первым параметром вместо экземпляра такой метод получает класс и может быть вызван как от класса, так и от экземпляра
- `staticmethod` превращает метод в статический (обычную функцию). может быть вызван как от класса, так и от экземпляра

## Классовые и статические методы

```
1  class A(object):
2      val = 12
3
4      @classmethod
5      def meth1(cls, x):
6          return x + cls.val
7
8      @staticmethod
9      def meth2(x, y):
10         return x + y
11
12  A.meth1(1) == 13
13  A().meth2(1,2) == 3
14
15  class B(A):
16      val = 13
17
18  B.meth1(1) == 14
```

## Сортировка и сравнение

- По-умолчанию `==` для пользовательских классов использует `is`
- `object_list . sort (cmp=lambda x,y : x.some_attr > y.some_attr)`
- `object_list . sort (key=lambda x : x.some_attr)`



## Классы/объекты внутри

- `__dict__`
- `__class__`
- `__mro__`

```
1      class A(object):
2          pass
3
4      class B(A):
5          pass
6
7      a = A()
8      a.x = 12
9      a.__class__ is A
10     a.__dict__ == {'x': 12}
11     B.__mro__ == (B, A, object)
```

## Задача

Написать `my_super`, которая работает, как встроенный `super`

присваивание классовым полям и полям экземпляра

property. Д3 `__get__`, `__set__`, `__del__`

## Протоколы/перенруззка операторов

int

- `__add__(self, obj) # self + obj`
- `__radd__(self, obj) # obj + self`
- `__iadd__(self, obj) # self += obj`
- `__int__(self) # int(self)`

```
1      a + b == a.__add__(b)
2      a + b == b.__radd__(a)
3      -a == a.__neg__()
```

## Работа некоторых встроенных функций (протоколы встроенных функций)

- `int(x) == x.__int__()`
- `str(x) == x.__str__()`
- `repr(x) == x.__repr__()`
- `len(x) == x.__len__()`
- `iter(x) == x.__iter__()`
- `next(x) == x.next()` O\_o
- `hex`, `oct`, `hash`

## Специальные методы - контейнер

- `x.__getitem__(index) # x[index]`
- `x.__setitem__(index, val) # x[index] = val`
- `x.__delitem__(index) # del x[index]`
- ...

## Специальные методы - доступ к атрибутам

- `x. __getattribute__ (name)`
- `x. __getattr__ (name)`
- `x. __setattr__ (name, val)`
- `x. __delattr__ (name)`
- `getattr (x, name[, val ])`
- `setattr (x, name, val)`
- `delattr (x, name)`



Видео лекции

OOP Programming