Класс

- Класс это тип данных (множество объектов, принадлежащих этому типу и операции над ними)
- Он объединеняет данные и функций, работающие над ними.
- Данные должны быть взаимосвязанны, а функции формировать API, достаточной для полноценной работы над данными ~~без прямого доступа к ним~~
- Язык обеспечивает автоматизацию многих задач по поддержке ООП

Объект, интерфейс и инстанцирование

- Объект - экземпляр класса
- Объект получается путем инстанцирования типа. В питоне инстанцирование это вызов obj = Type (...)
- Одновременно может существовать неограниченное количество экземпляров класса, все они независимы
- Интерфейс - это набор операций, которые можно проделать над типом Например файл - это интерфейс. Из него можно читать, в него можно писать.
- Классы реализуют интерфейсы
- Экзампляры классов предоставляют интерфейсы
- Вообще интерфейсов нет, но ABS

Пример класса

```
1       class Simple(object):
2           "class documentation"
3
4           def method(self, val):
5               "method documentation"
6               self.some_field = val
7               return val ** 0.33333333333333333333333
8
9       simple = Simple()
10      simple.method(8)   # 2.0
```

Поля класса

- Поля это переменные, связанные с экземпляром класса
- Класс явно не определяет какие поля будут у его объектов
- У каждого экзампляра они свои
- Доступ производится с помошью 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 экзампляр, от которого вызван

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

Нет инкапсуляции

- "Принцип открытого кимоно" == "мы не знаем как совместить инкапсуляцию с остальными возможностями языка"
- Есть property, "скрытые поля" (но они предназначенны для другого)
- В отличии от Java всегда можно изменить поле на свойство с сохранением совместимости
- Можно реализовать любой вид инкапсуляции динамической
- Ничего из этого не получило какого-нить 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(1)
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 = B()
10      a.some_method() # A.some_meth
11      b = B()
12      b.some_method() # B.some_meth
```

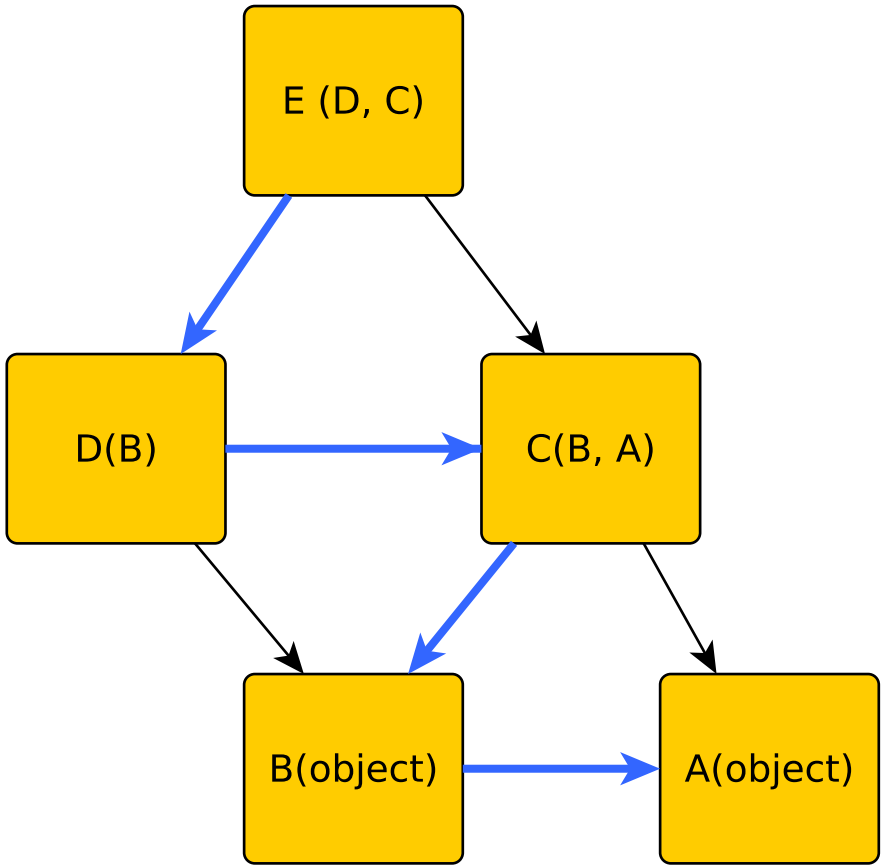
Вызов метода базового класса

```
1       class A(object):
2           def some_method(self, val):
3               print "A inited with value", val
4
5       class B(A):
6           def some_method(self, val):
7               print "B inited with value", val
8               A.some_method(self, val)
9
10      b = B(1)
11      # B inited with value 1
12      # A inited with value 1
```

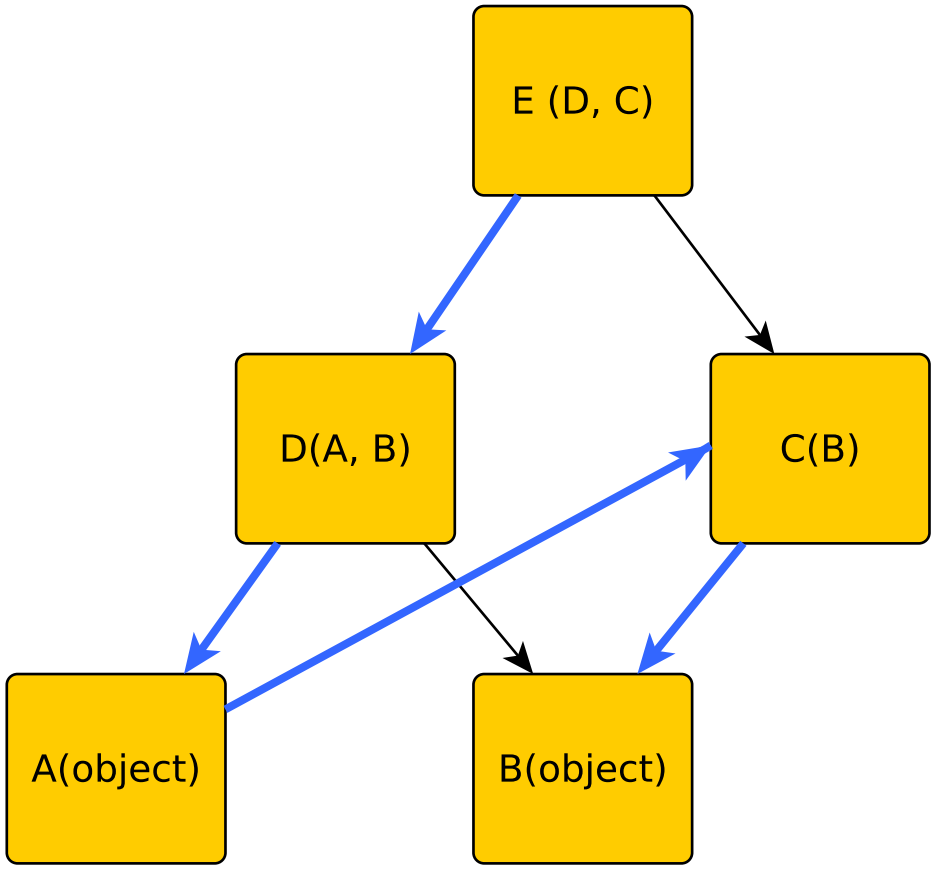
```
1       class A(object):
2           def draw(self, pt):
3               some_action()
4
5       class B(A):
6           def draw(self, pt):
7               #A.draw(self, pt)
8               super(B, self).draw(pt)
```

Подробное описание super.

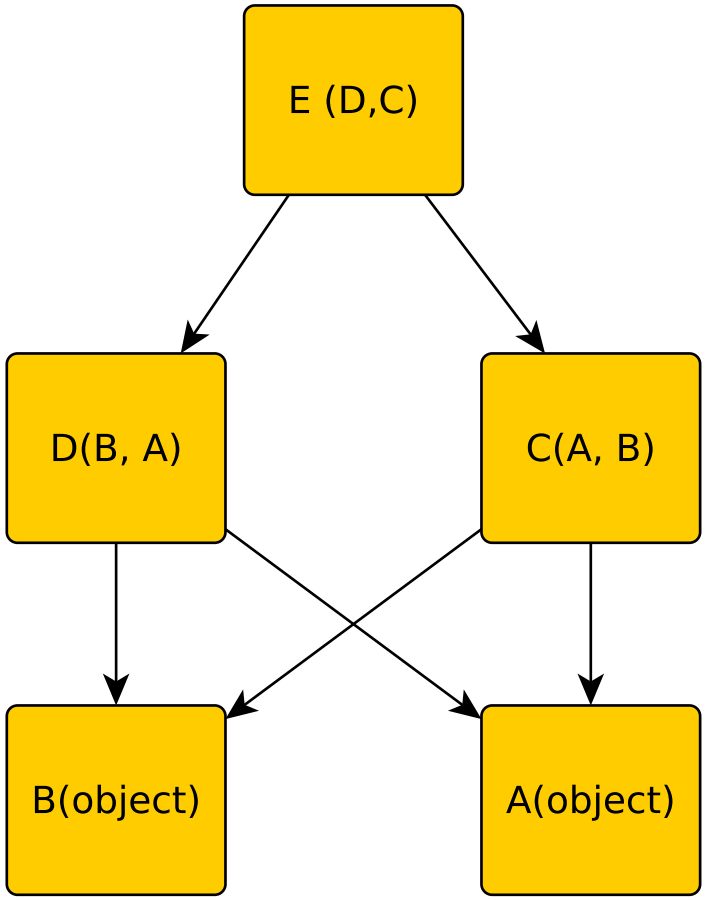
Пример линеаризации (очередность в super)



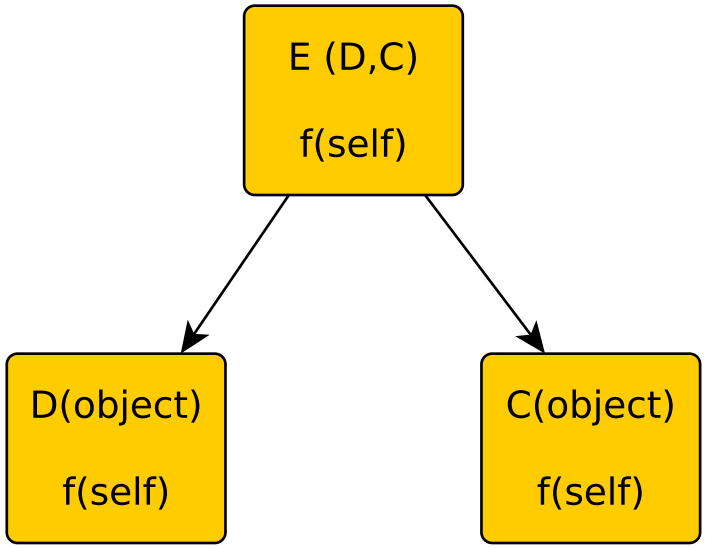
Пример линеаризации 2



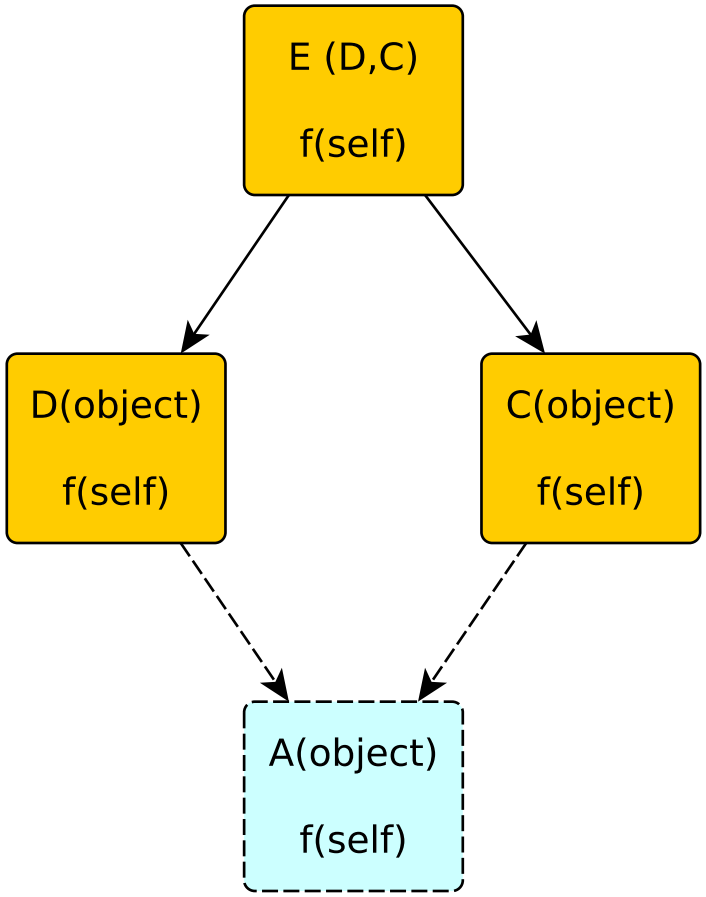
Нелинеаризуемая иерархия классов



Ошибки при построении иерархии



Ошибки при построении иерархии



Конструктор __init__

- __init__ конструктор, вызывается при создании экземпляра класса
-

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

Деструктор __del__

- __del__ Должен вызываться перед удалением объекта (и, обычно, вызывается достаточно предсказуемо)
- Если нет циклических ссылок
- Или объект не попал во фрейм, где произошла ошибка
- 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__.

```
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
9           def __new__(cls, x, y=None):
10              if y is None:
11                  return A(x)
12              else:
```

```
13                  return super(B, cls).__new__(cls, x, y)
14
15      print B(1, 2) #<__main__.B at 0x...>
16      print B(1) # <__main__.A at 0x...>
```

isinstance, issubclass

- 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 расле-дует 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.
- A.b(1) - ошибка, первым параметром должен идти экземпляр класса A.
- (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 превращяет метод в классовый, первым параметром вместо экземпляра такой метод получает класс и может быть вызван как от класса, так и от экземпляра
- staticmethod превращяет 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__

Задача

Написать my_super, которая работает, как встроенный super

присваивание классовым полям и полям экземпляра

property. ДЗ __get__, __set__, __del__

Протоколы/перенругзка операторов

int

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

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