

Пример №1

Написать функцию для вычисления среднего от массива чисел

Среднее. Вариант №1

```
1  def mean( numbers ):  
2      res = 0  
3      for num in numbers:  
4          res += num  
5      return nun / len( numbers )
```

Нужно добавить поддержку рациональных чисел

Рациональное число - пара (числитель, знаменатель). Будем передавать рациональные числа в виде кортежа.

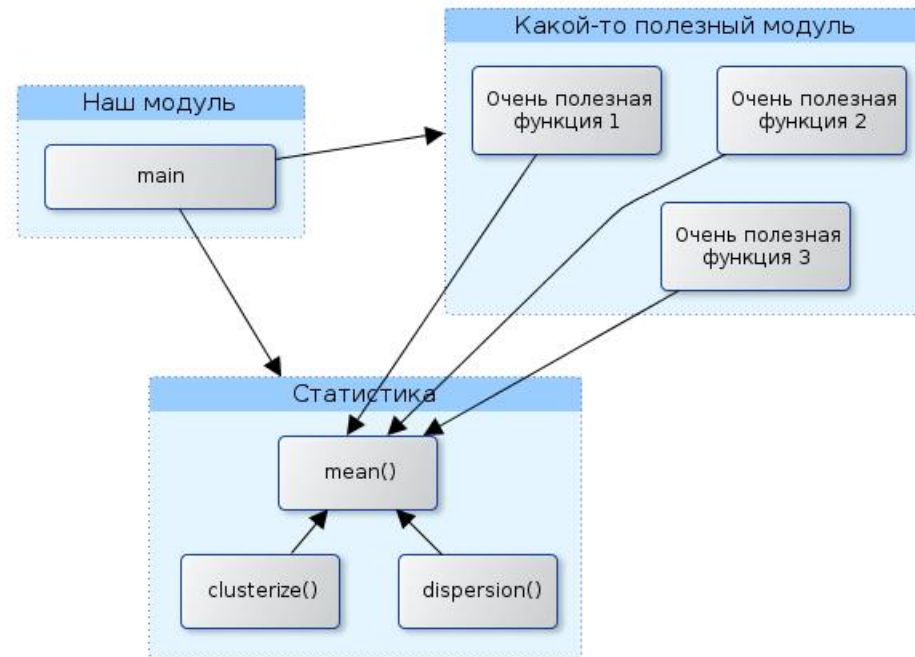
$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd} ; \quad \frac{a}{b} / c = \frac{a}{cd}$$

Среднее. Вариант №2

```
1  def mean( numbers ):
2      res = (0, 1)
3
4      for num in numbers:
5          if isinstance( num, int ):
6              num = (num, 1)
7
8              res = (res[0] * num[1] + res[1] * num[0],
9                    num[1] * res[1])
10
11     return (res[0], res[1] * len( numbers ))
```

Процедурный стиль - анализ

- `if isinstance (num, int):` - ужасно и вызывает массу проблем
- Добавление новых типов требует изменения функции `mean`
- Перегрузка функций решает небольшую часть проблем
- Если `mean` в сторонней библиотеке - ничего не выйдет
- Дописать еще одну функцию и использовать ее - не выход



Процедурный стиль - причины неудачи

- На самом деле теор не нужно знать как устроены данные внутри
- Ей нужно только знать как складывать и делить их
- Эта информация есть в той точке, где мы определяем новый тип

Среднее. Вариант №3

Передавать соответствующие функции снаружи.

```
1     val = (1, 2)
2     div_int = lambda val, div: val / div
3     div_r    = lambda val, div: (val[0], val[1] * div)
4     add_int = lambda val1, val2: val1 + val2
5     def add_r(val1, val2):
6         n1, d1 = val1
7         n2, d2 = val2
8         return (n1 * d2 + d1 * n2, d1 * d2)
9
10    add_ir = lambda val1, val2: add_r((val1, 1), val2)
11    add_ri = lambda val1, val2: add_ir(val2, val1)
12    div = {int: div_int, list: div_rational}
13
14    add = {(int, int): add_int,
15          (list, list): add_rational,
16          (list, int): add_rational_int,
17          (int, list): add_int_rational}
```

Среднее. Вариант №3

```
1  def mean(numbers , add_funcs , div_funcs ):
2      res = numbers[0]
3      for num in numbers[1:]:
4          res = add_funcs[(type(res) , type(num))](res , num)
5      return div_funcs[type(res)](res , len(numbers))
```


Вариант №3 анализ

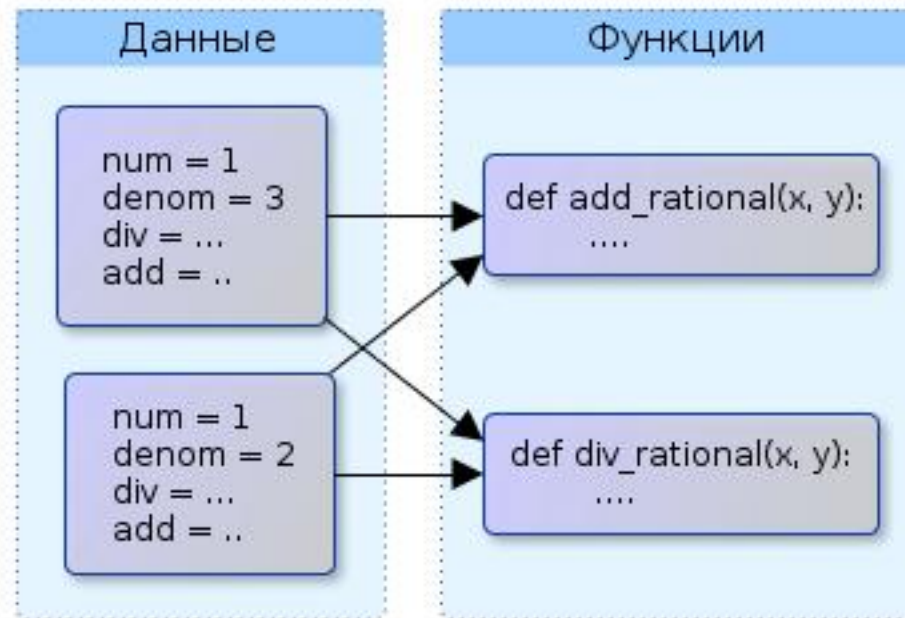
Приходится передавать дополнительно много параметров. Проблема будет только усиливаться для сложных функций. Решение - привязать обработчики к данным. Хранить данные и функции будем в словаре.

Вариант №4

```
1  val = { 'num':1, 'denom':2, 'add':add_rational }
2  def add_rational(x, y):
3      return (x['num'] * y['denom'] + \
4              x['denom'] * y['num'],
5              x['denom'] * y['denom'])
6
7  def div_rational(x, y):
8      return (x['num'], x['denom'] * y)
9
10 def mk_rational(num, denom):
11     return { 'num':num,
12             'denom':denom,
13             'add':add_rational,
14             'div':div_rational }
```

Вариант №4

```
1  def mean(numbers):
2      if isinstance(numbers[0], int):
3          res = numbers[0]
4          for num in numbers[1:]:
5              res += num
6          return res / len(numbers)
7      else:
8          res = numbers[0]
9          for num in numbers[1:]:
10             res = res['add'](res, num)
11         return res['div'](res, len(numbers))
12
13 res = mean([mk_rational(1, 3),
14             mk_rational(1, 2)])
```



Нужно автоматически упрощать после операции и добавить функцию для
вычитания

```
1  def mk_rational_auto_simpl(num, denom):  
2      return dict(num=num,  
3                  denom=denom,  
4                  add=add_rational_auto_simpl,  
5                  div=div_rational_auto_simpl,  
6                  sub=sub_rational_auto_simpl)
```

Немного измененный вариант

```
1  def sub_rational_auto_simpl(x, y):
2      nx = x['neg'](x)
3      return x['add'](x, y)
4
5  def mk_rational_auto_simpl(num, denom):
6      res = mk_rational(num, denom)
7      res['add'] = add_rational_auto_simpl
8      res['div'] = div_rational_auto_simpl
9      return res
```

Не совсем процедурный стиль - анализ

- Кода стало больше
- Его расширение значительно упростилось - функции могут обрабатывать данные, не зная их конкретного типа
- Тип - это операции, которые есть у него (duck typing)

Не совсем процедурный стиль - анализ

- Типовые теги иногда нужны.
- Каждый экземпляр содержит большое количество ссылок на одни и те же функции.
- Решение - вынесение всех методов в отдельный словарь, который все переменные данного типа используют совместно. Одновременно этот словарь становится типовым тегом.

Среднее. Вариант №5

```
1  RN = { 'add': add_rational ,
2         'sub': sub_rational ,
3         'div': div_rational ,
4         'neg': neg_rational ,
5         '__init__': mk_rational }
6
7  ASRN = RN.copy()
8  ASRN['add'] = add_rational_auto_simpl
9  ASRN['div'] = div_rational_auto_simpl
10 ASRN['__init__'] = add_rational_auto_simpl
11
12 x1 = BasicRN['__init__'](1, 2)
13 x2 = ASRN['__init__'](1, 2)
```

Среднее. Вариант №5

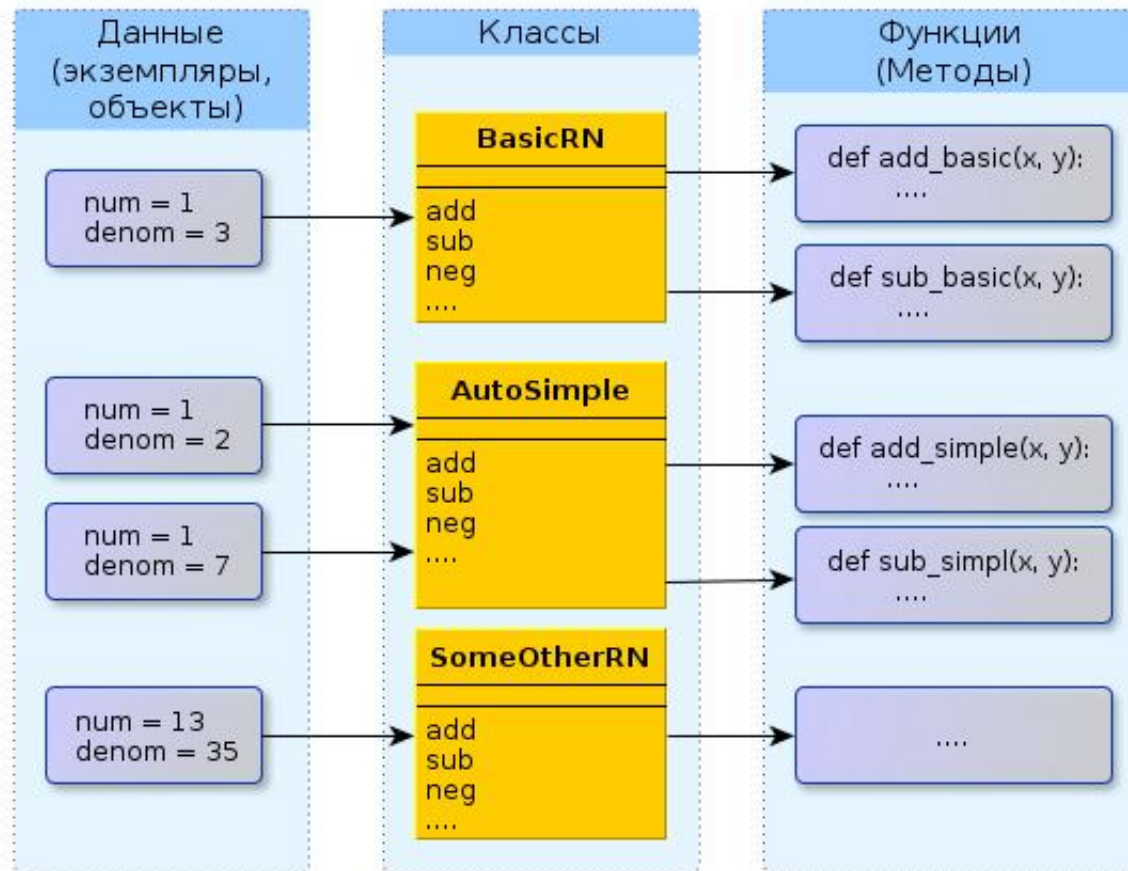
```
1  def mean( numbers ):  
2      if isinstance( numbers[0] , int ):  
3          ...  
4      else:  
5          res = numbers[0]  
6          for num in numbers[1:]:  
7              add_meth = res[ '__class__ '][ 'add ']  
8              res = add_meth( res , num)  
9  
10         div_meth = res[ '__class__ '][ 'div ']  
11     return div_meth( res , len( numbers ) )
```

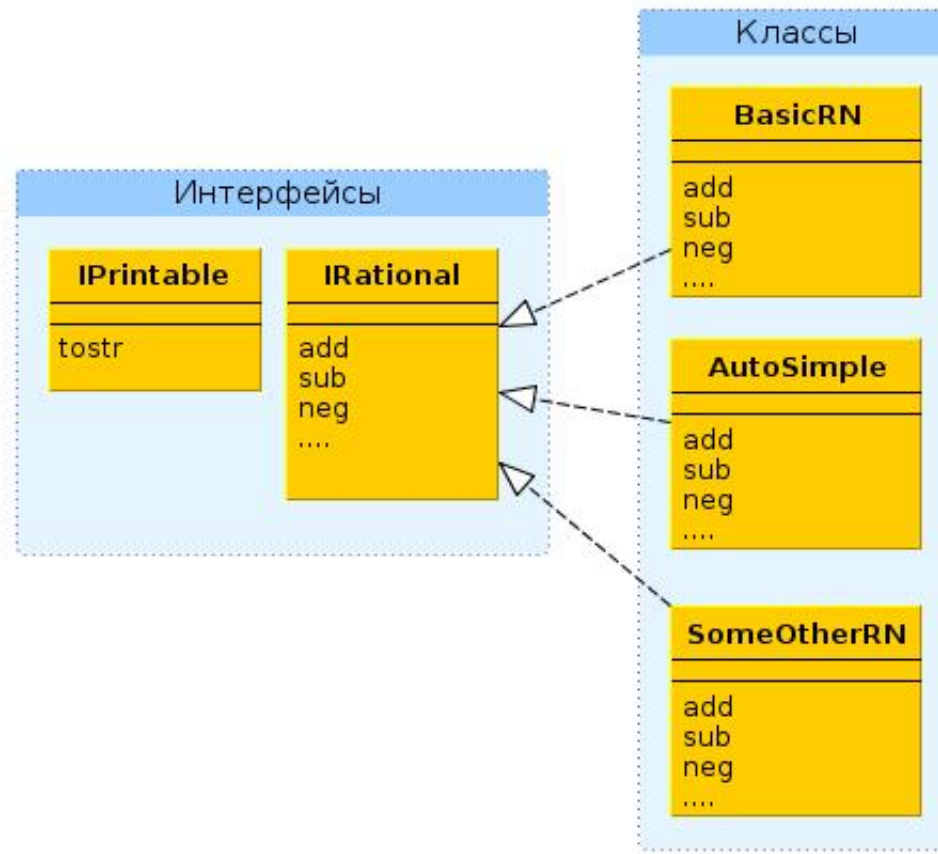
Именно так и устроено ООП

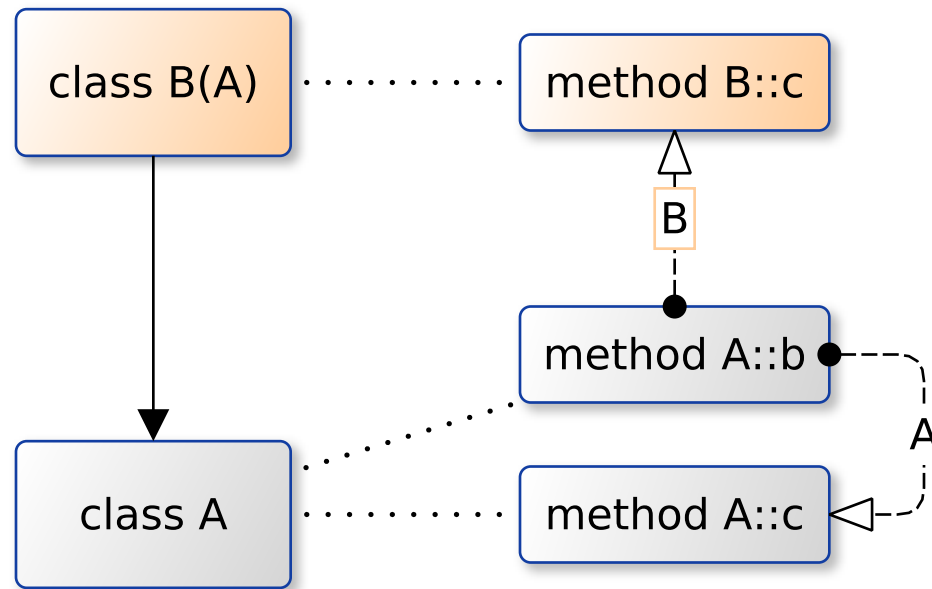
- Шаблон программирования, когда данные хранят ссылку на свой тип (класс)
- Тип хранит ссылки на реализации операций для себя
- А логика программы использует эти операции вместо прямой работы с данными
- Всё - объекты с интерфейсами (iter, число)

Buzzwords

- Класс - тип данных, в котором есть методы
- Методами - функции в классе
- Экземпляры - объект конкретного класса
- Объект - экземпляр какого либо класса*
- Атрибутами - переменные внутри экземпляра
- Интерфейсом - набор методов и атрибутов
- Инкапсуляция - сокрытие внутренней структуры данных за методами
- Наследование - возможность использовать некоторый класс в качестве "базового"
- Полиморфизм - возможность изменить часть методов в базовом классе
- Перегрузка - изменение метода в дочернем классе



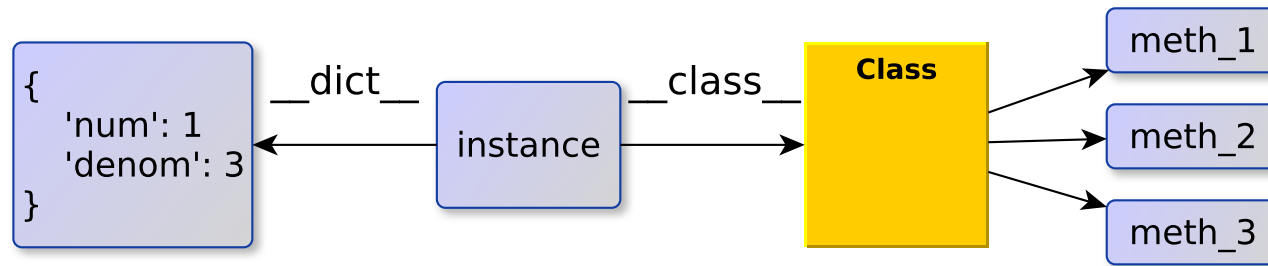




Рациональные числа - классы

```
1  class BasicRational(object):
2      "basic rational number"
3      def __init__(self, num, denom):
4          self.num = num
5          self.denom = denom
6
7      def add(self, y):
8          nd = self.denom * y.denom
9          nn = self.num * y.denom + \
10              y.num * self.denom
11          return BasicRational(nn, nd)
12
13     def neg(self):
14         return BasicRational(-self.num,
15                               self.denom)
16
17     def sub(self, y):
18         return self.add(y.neg())
```


Объект в питоне



`obj.__class__ == type(obj)` - класс объекта

`obj.__dict__` - словарь, содержащий атрибуты объекта

ООП без классов еще раз

```
1 X = Y.copy()
2
3 def mk_X(val):
4     res = mk_Y()
5     res['val'] = val
6     res['__class__'] = X
7     return res
8
9 def tostr_X(x):
10    return "X<val={}>".format(x['val'])
11
12 X = {'tostr': tostr_X,
13      '__init__': mk_X}
14
15 x = mk_X(1)
16
17 def print_list(lst):
18     vals = (obj['__class__']\
19             ['tostr'](obj)
20             for obj in lst)
21     return "[{}]" .format(
22         ", ".join(vals))
```

```
1 class X(Y):
2
3     def __init__(self, val):
4         Y.__init__(self)
5         self.val = val
6
7
8
9     def tostr(self):
10        return "X<val={}>".format(self.val)
11
12
13
14
15 x = X(1)
16
17 def print_list(lst):
18     vals = (obj.tostr()
19             for obj in lst)
20
21     return "[{}]" .format(
22         ", ".join(vals))
```

В BasicRational - ошибка

В BasicRational - ошибка

```
1     def neg(self):
2         return BasicRational(-self.num,
3                               self.denom)
4
5
6     def neg(self):
7         return self.__class__(-self.num,
8                               self.denom)
```

То же и в BasicRational.add

Рациональные числа - классы

```
1    x[ '__class__ '][ 'add '](x, y) == x.add(y)
2    # == BasicRational.add(x, y)
3    x[ 'num ' ] == x.num
4    x[ '__class__ '][ 'add '](x, ...) == x.add
```

Среднее. Вариант №5

```
1  def mean( numbers ):  
2      if isinstance( numbers[0] , int ):  
3          ...  
4      else:  
5          res = numbers[0]  
6          for num in numbers[1:]:  
7              res = res.add(num)  
8  
9      return res.div( len( numbers ) )
```

Среднее. Вариант №6

Можно перегрузить операторы.

```
1      def mean( numbers ) :  
2          res = numbers [ 0 ]  
3          for num in numbers [ 1 : ] :  
4              res += num  
5          return res / len ( numbers )
```

На самом деле эта функция работает для int, float, complex потому что они тоже перегружают операторы типа object.

ООП - в чем причина?

- Дополнительный уровень косвенности
- Написав `add` таким образом мы получили возможность менять ее работу не трогая код
- Код, который создает новую дробь знает подробности того, как с ней работать
- Код, который ее использует - не всегда
- Отделяя основной алгоритм функции от особенностей конкретных типов мы можем сделать ее гораздо более универсальной

Пример №2

Необходимо сравнить файлы одинаковой длины посимвольно и возвращать массив отличающихся позиций

```
1      def compare(fname1, fname2):
2          res = []
3          with open(fname1, "rb") as fd1:
4              with open(fname2, "rb") as fd2:
5                  x1 = fd1.read(1)
6                  x2 = fd2.read(1)
7                  pos = 0
8                  while x1 != '':
9                      if x1 != x2:
10                         res.append(pos)
11                         x1 = fd1.read(1)
12                         x2 = fd2.read(1)
13                         pos += 1
14      return res
```

Пример №2

```
1  from itertools import izip
2  def compare_streams(iter1 , iter2 , cmp, on_diff):
3      it = enumerate(izip(iter1 , iter2))
4      for pos, (ch1, ch2) in it:
5          if cmp(x1, x2) != 0 :
6              on_diff(pos)
7
8  def fileiter(fd):
9      ch = fd.read(1)
10     while " " != ch:
11         yield ch
12         ch = fd.read(1)
13  def compare_files(fname1 , fname2):
14      res = []
15      with open(fname1 , "rb") as fd1:
16          with open(fname2 , "rb") as fd2:
17              compare_streams(fd1 , fd2 , cmp, res.append)
18      return res
```

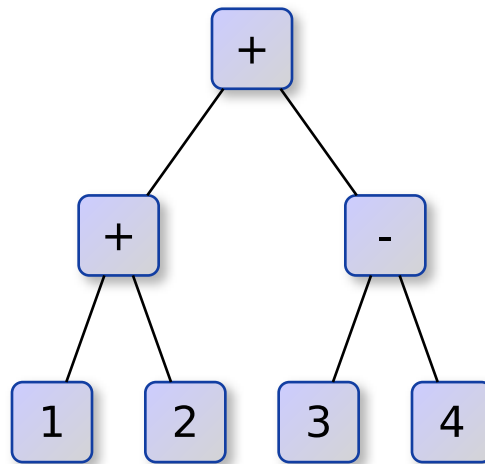
Пример №2

- Теперь `compare_streams` может использоваться для сравнения любых последовательностей
- Как символьных, так и нет
- Можно сделать сравнение нечувствительным к регистру
- И для асинхронных потоков
- Все это очень полезно для создания повторно используемого кода
- Не всегда нужно передавать классы, часто достаточно функции. В python точно. В C++ через шаблоны и `std::function<>`.
- Но только для повторно используемого - **YAGNI**

Пример №3

Написать обработчик выражения, состоящего из -, +, скобок и чисел. В первую очередь нас интересует вычислитель. (Пусть у нас уже есть синтаксический анализатор выражения - функция parse).

```
1      expression = "1 + 2 + (4 - 3)"
2      #pexpr = parse(expression)
3      pexpr = ((1, '+', 2), '+', (4, '-', 3))
```



Решение №1

```
1  def evaluate(val):
2      if isinstance(val, int):
3          res = val
4      else:
5          operator, oper1, oper2 = val
6          assert operator in "+-", \
7              "Unknown operator " + operator
8
9          v1 = evaluate(oper1)
10         v2 = evaluate(oper2)
11
12         if operator == '+':
13             res = v1 + v2
14         elif operator == '-':
15             res = v1 - v2
16
17     return res
```

Добавить в выражение поддержку * и /

Добавить в выражение поддержку * и /

```
1  class Operator(object):
2      def evaluate(self, op1, op2):
3          pass
4
5  class Add(object):
6      def evaluate(self, op1, op2):
7          return op1 + op2
8
9  class Mul(object):
10     def evaluate(self, op1, op2):
11         return op1 * op2
12
13  expr = (Add(), (Add(), 1, 2), (Sub(), 3, 4))
```

Добавить в выражение поддержку * и /

```
1  def evaluate_add(op1 , op2 ):
2      return op1 + op2
3
4  def evaluate_mul(op1 , op2 ):
5      return op1 * op2
6
7  expr = (evaluate_add ,
8          (evaluate_add , 1 , 2) ,
9          (evaluate_mul , 3 , 4))
```


Добавить в выражение поддержку * и /

```
1  def evaluate ( val ) :
2      if isinstance ( val , int ) :
3          res = val
4      else :
5          operator , oper1 , oper2 = val
6
7          v1 = evaluate ( oper1 )
8          v2 = evaluate ( oper2 )
9
10         res = operator ( v1 , v2 )
11         #res = operator.evaluate ( v1 , v2 )
12
13     return res
```

Добавить в выражение поддержку строк

Добавить в выражение поддержку строк

```
1  class Value(object):
2      def add(self, v2):
3          pass
4
5  class Int(Value):
6      def __init__(self, val):
7          self.val = val
8
9      def add(self, v2):
10         if isinstance(v2, int):
11             return v2 + self.val
12         else:
13             return v2.add(self.val)
14
15  def evaluate_add(op1, op2):
16      return op1.add(op2)
```

Проблемы?

Проблемы?

Функция parse. Почему?

parse

- С одной стороны она должны явно знать какие типы создавать
- С другой стороны она не может этого знать
- Нужно передавать parse снаружи.

Когда ООП

- Если есть участок кода, требующий определенного ограниченного набора операций над входными данными
- Одновременно в программе могут быть несколько видов подходящих данных, с различной функциональностью для реализации этого интерфейса
- Причем участок кода в свою очередь может быть одним из методов класса.
- Или группировки функций с общим глобальным состоянием

Классы не предназначен для

- Группировки функций
- Группировки одной функции
- Если вы, ессно, используете нормальный ЯП

ООП vs Процедурный стиль

- (-) Часто больше кода
- (-) Добавление нового метода требует нелокальных изменений
- (-) Замедляет работу
- (-) Усложняет язык
- (-) Логика размазывается
- (-) Разработка ООП дизайна требует больших навыков и времени, чем процедурного
- (-) Работает только если функция выбирается по типу одного параметра

ООП vs Процедурный стиль

- (+) Уменьшает пересечение имен
- (+) Код лучше структурирован
- (+) Избавляет от ручной проверки типов
- (+) Избавляет от знания конкретного типа данных
- (+) Во многих случаях значительно упрощает расширение. Позволяя корректно написанному коду работать с новыми типами данных
- (+) Более высокий уровень абстракции упрощает построение программы путем выделения стандартных шаблонов проектирования
- (+) Многие из идей ООП имеют прямую поддержку в языке

Добавление нового метода требует нелокальных изменений : решение

```
1     def to_xml ( obj ) :  
2         if isinstance ( obj , int ) :  
3             ...  
4         elif isinstance ( obj , str ) :  
5             ...  
6         else :  
7             res = obj . to_xml ( )  
8     return res
```

Проблема

$x.add(y) \neq y.add(x)$. При других типах может быть совсем не верно.
В чем проблема? Как решать?

Проблема

$x.add(y) \neq y.add(x)$. При других типах может быть совсем не верно.

В чем проблема? Как решать?

Нужна перегрузка `add` по обоим параметрам, что-то типа $(x,y).add(x,y)$.

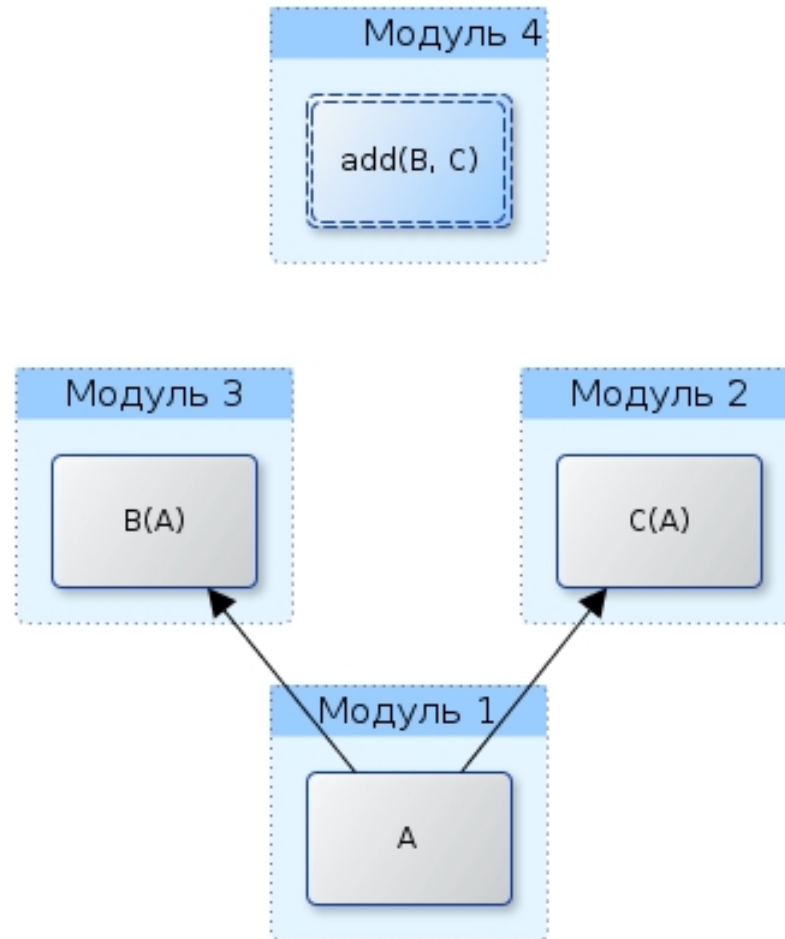
Классическое ООП не предлагает решения.

Почти решение

```
1      class X(Y):
2          def add(self , y, final=False):
3              if isinstance(y, Y):
4                  self.do_add_Y(y)
5              elif isinstance(y, X):
6                  self.do_add_X(y)
7              else:
8                  return y.add(self , True)
```

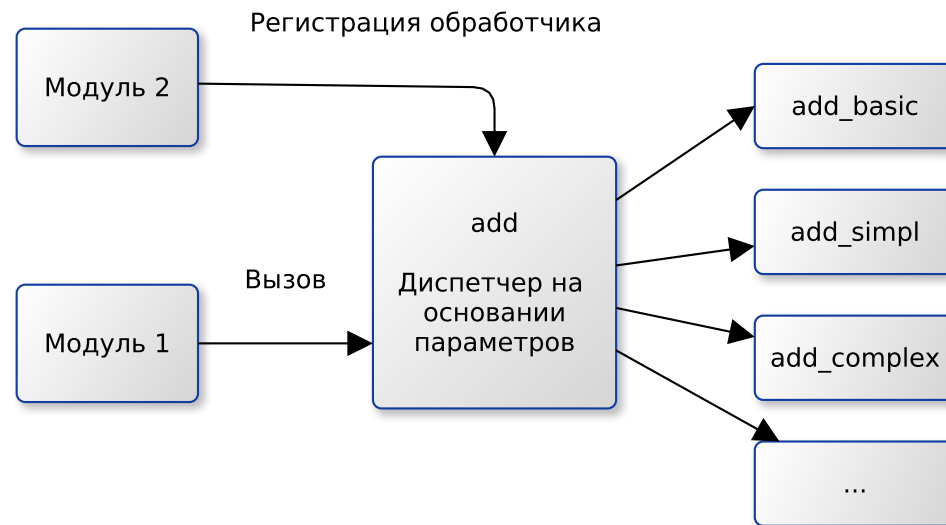
Более-менее работает для линейных иерархий.

Но для не линейных...



Альтернативы

- CLOS - возможность расширять работу функции после ее создания, динамически подключая новые реализации
- Функция превращается в объект-хранилище шаблонов со ссылками на реализации
- Позволяет перегружать поведение не только по одному параметру
- Замедляет работу



Альтернативы

- Аспектное программирование
- <http://pypi.python.org/pypi/PEAK-Rules>