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.

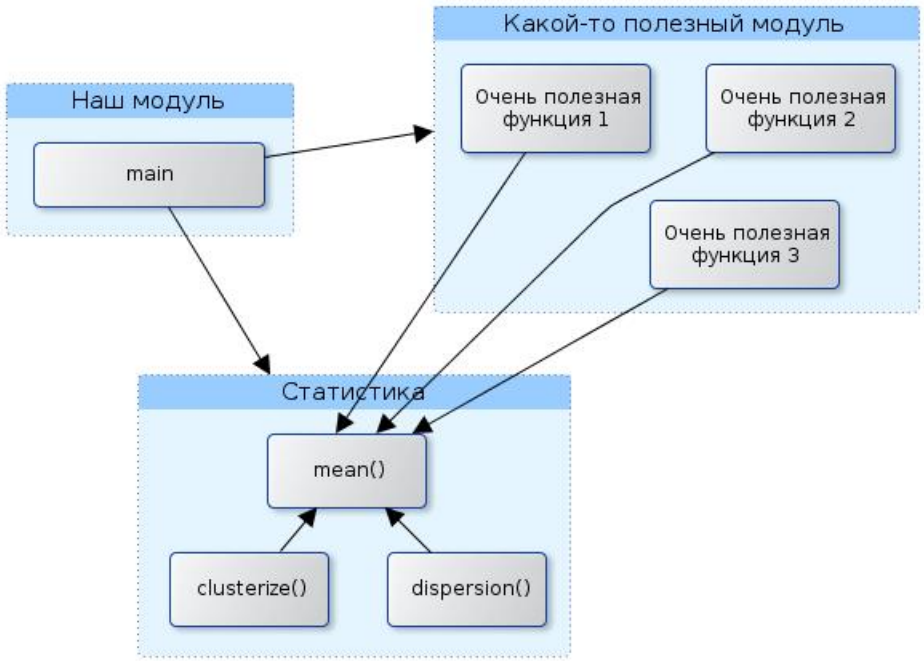
$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd} \; ; \quad \frac{a}{b}/c = \frac{a}{cd}$$

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 mean не нужно знать как устроенны 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.

```
val = (1, 2)
div_int = lambda val, div: val / div
div_r   = lambda val, div: (val[0], val[1] * div)
add_int = lambda val1, val2: val1 + val2
def add_r(val1, val2):
    n1, d1 = val1
    n2, d2 = val2
    return (n1 * d2 + d1 * n2, d1 * d2)

add_ir = lambda val1, val2: add_r((val1, 1), val2)
add_ri = lambda val1, val2: add_ir(val2, val1)
div = {int: div_int, list: div_rational}

add = {(int, int): add_int,
       (list, list): add_rational,
       (list, int): add_rational_int,
       (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

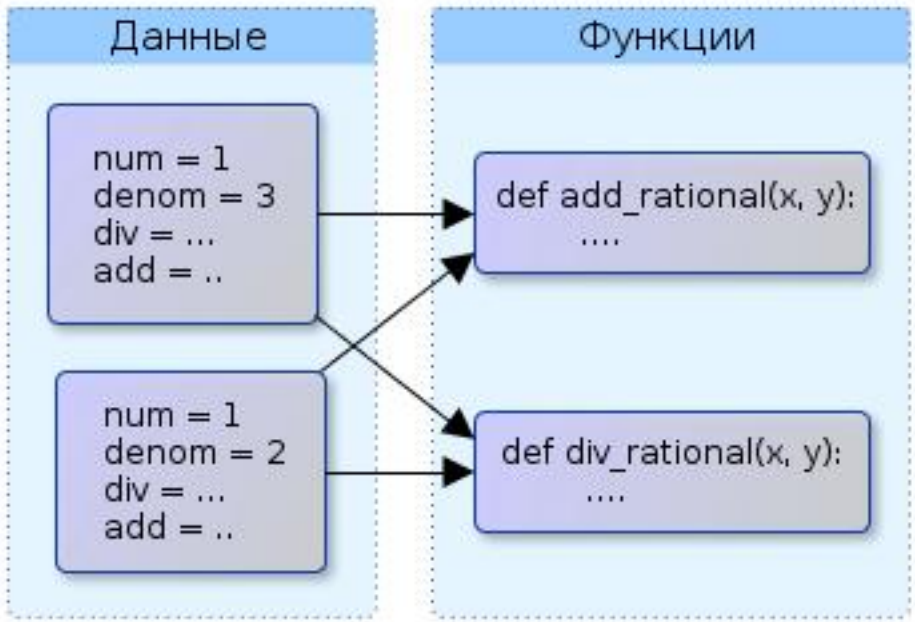
Приходится передавать дополнительно много параметров. Проблема будет только усиливаться для сложных функций. Решение - привязять обработчики к данным. Хранить данные и функции будем в словаре.

Вариант №4

```
val = {'num':1, 'denom':2, 'add':add_rational}
def add_rational(x, y):
    return  (x['num'] * y['denom'] + \
             x['denom'] * y['num'],
             x['denom'] * y['denom'])

def div_rational(x, y):
    return (x['num'], x['denom'] * y)

def mk_rational(num, denom):
    return {'num':num,
            'denom':denom,
            'add':add_rational,
            '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)])
```


Данные
Функции
num = 1
denom = 3
div = ...
add = ..
num = 1
denom = 2
div = ...
add = ..
def add_rational(x, y):
....
def div_rational(x, y):
....

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

```
def sub_rational_auto_simpl(x, y):
    nx = x['neg'](x)
    return x['add'](x, y)

def mk_rational_auto_simpl(num, denom):
    res = mk_rational(num, denom)
    res['add'] = add_rational_auto_simpl
    res['div'] = div_rational_auto_simpl
    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 tuping)

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

```
RN = {'add': add_rational,
      'sub': sub_rational,
      'div': div_rational,
      'neg': neg_rational,
      '__init__': mk_rational}

ASRN = RN.copy()
ASRN['add'] = add_rational_auto_simpl
ASRN['div'] = div_rational_auto_simpl
ASRN['__init__'] = add_rational_auto_simpl

x1 = BasicRN['__init__'](1, 2)
x2 = ASRN['__init__'](1, 2)
```

Среднее. Вариант №5

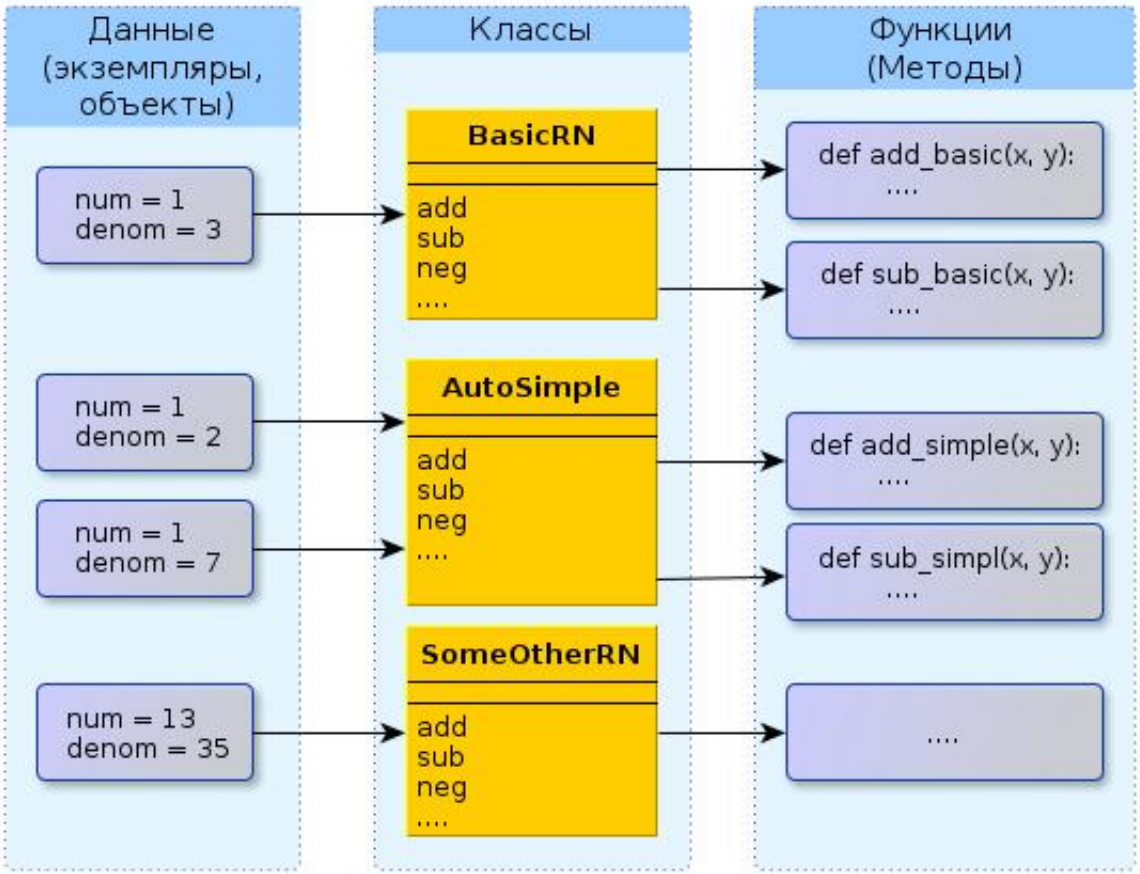
```
def mean(numbers):
    if isinstance(numbers[0], int):
        ...
    else:
        res = numbers[0]
        for num in numbers[1:]:
            add_meth = res['__class__']['add']
            res = add_meth(res, num)

        div_meth = res['__class__']['div']
        return div_meth(res, len(numbers))
```

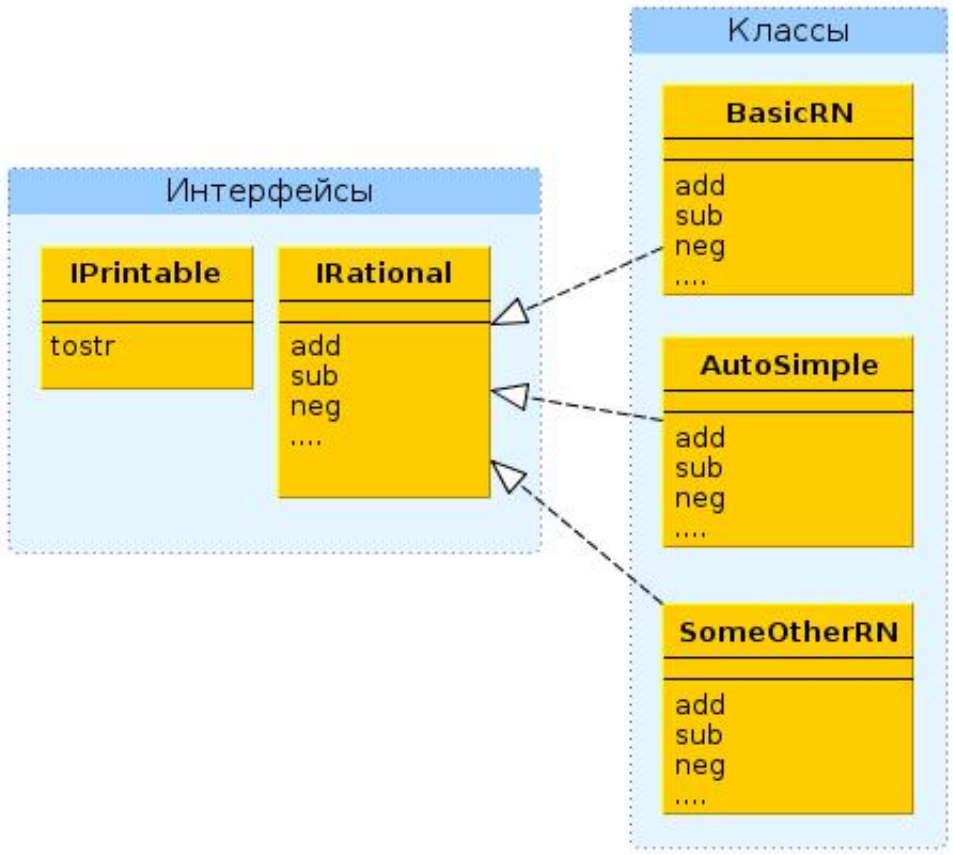
Именно так и устроенно ООП

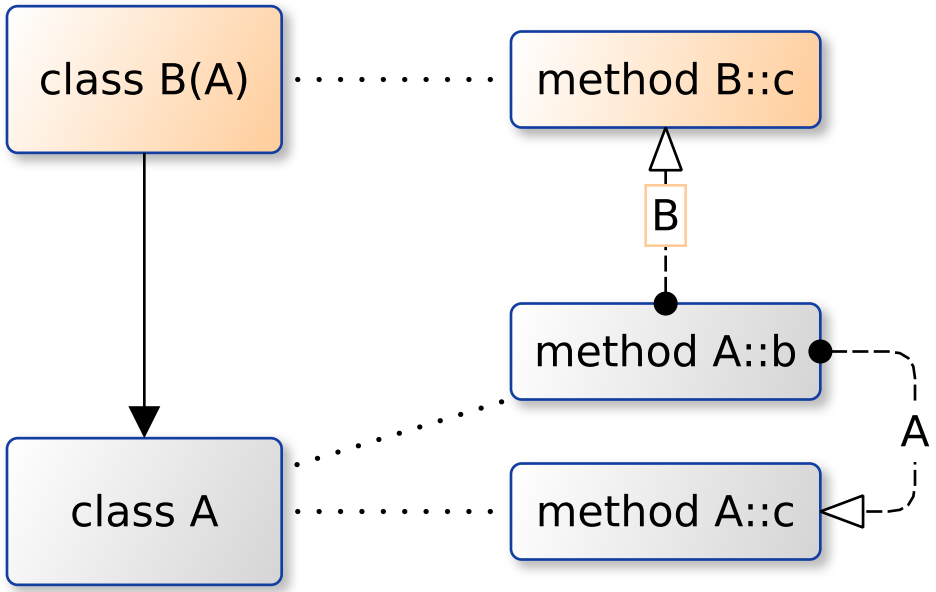
- Шаблон программирования, когда данные хранять ссылку на свой тип (класс)
- Тип хранит ссылки на реализации операций для себя
- А логика программы использует эти операции вместо прямой работы с данными
- Всё - объекты с интерфейсами (iter, число)

Buzzwords

- Класс - тип данных, в котором есть методы
- Методами - функции в классе
- Экземпляры - объект конкретного класса
- Объект - экземплят 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 - позможность изменить часть методов в базовом классе
- Перегрузка - изменение метода в дочернем классе


Данные
(экземпляры,
объекты)
num = 1
denom = 3
num = 1
denom = 2
num = 1
denom = 7
num = 13
denom = 35
Классы
BasicRN
add
sub
neg
....
AutoSimple
add
sub
neg
....
SomeOtherRN
add
sub
neg
....
Функции
(Методы)
def add_basic(x, y):
....
def sub_basic(x, y):
....
def add_simple(x, y):
....
def sub_simpl(x, y):
....
....

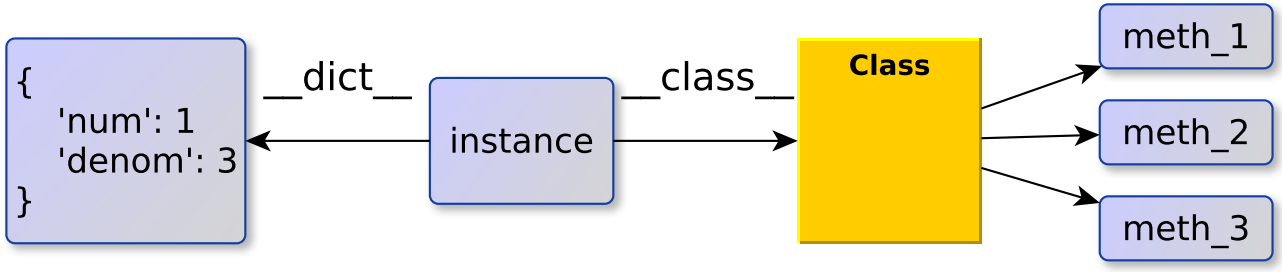

Классы
BasicRN
add
sub
neg
....
Интерфейсы
IPrintable
tostr
IRational
add
sub
neg
....
AutoSimple
add
sub
neg
....
SomeOtherRN
add
sub
neg
....


class B(A)
method B::c
B
method A::b
A
class A
method A::c

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

```
X = Y.copy()

def mk_X(val):
    res = mk_Y()
    res['val'] = val
    res['__class__'] = X
    return res

def tostr_X(x):
    return "X<val={}>".format(x['val'])

X = {'tostr' : tostr_X,
     '__init__': mk_X}

x = mk_X(1)

def print_list(lst):
    vals =  (obj['__class__']\
                    ['tostr'](obj)
                for obj in lst)
    return "[{}]".format(
            ", ".join(vals))
```

```
class X(Y):

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



    def tostr(self):
        return "X<val={}>".format(self.val)



x = X(1)

def print_list(lst):
    vals =  (obj.tostr()
                for obj in lst)

    return "[{}]".format(
        ", ".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

```
x['__class__']['add'](x, y) == x.add(y)
# == BasicRational.add(x, y)
x['num'] == x.num
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 сравненивать файлы одинаковой длинны 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
```

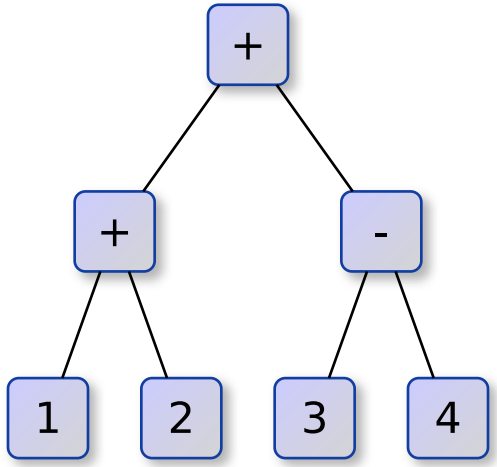
Пример №2

- Теперь compare_streams может использоваться для сравнения любых последовательностей
- Как символьных, так и нет
- Можно сделать сравнение нечуствительным 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

```
class Value(object):
    def add(self, v2):
        pass

class Int(Value):
    def __init__(self, val):
        self.val = val

    def add(self, v2):
        if isinstance(v2, int):
            return v2 + self.val
        else:
            return v2.add(self.val)

def evaluate_add(op1, op2):
    return op1.add(op2)
```

Проблемы?

Проблемы?

Фукция 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) != y.add(x)`. При других типах может быть совсем не верно.
В чем проблема? Как решать?

Проблема

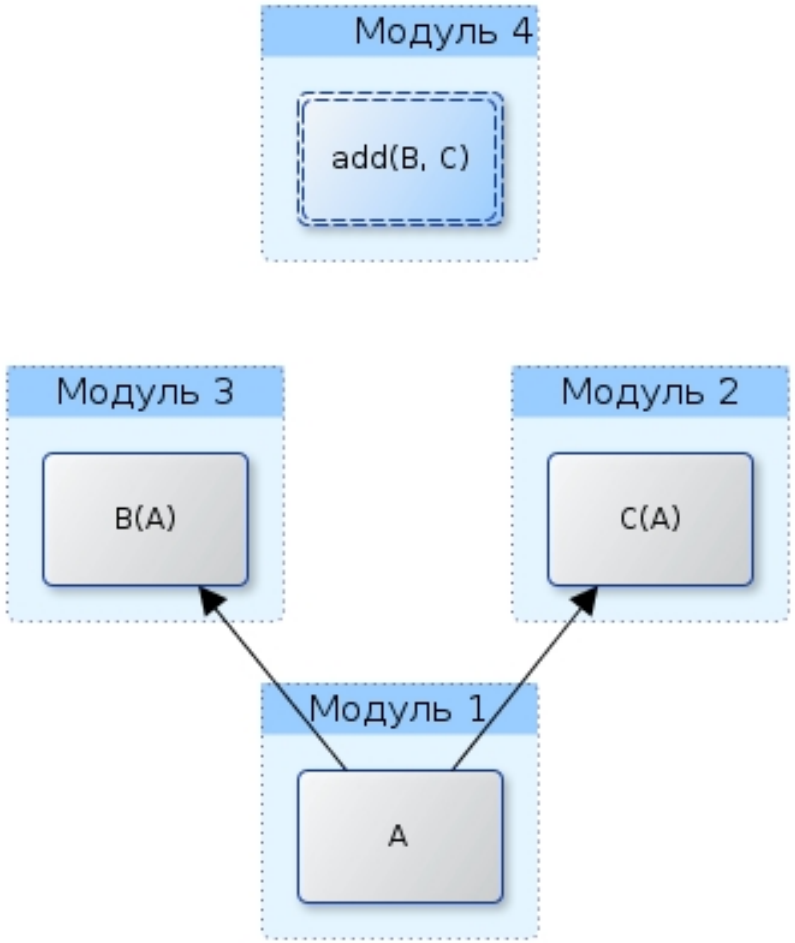
x.add(y) != y.add(x). При других типах может быть совсем не верно.
В чем проблема? Как решать?
Нужна перегрузка add по обеим параметрам, что-то типа (x,y).add(x,y).
Классическое ООП не предлагает решения.

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)
```

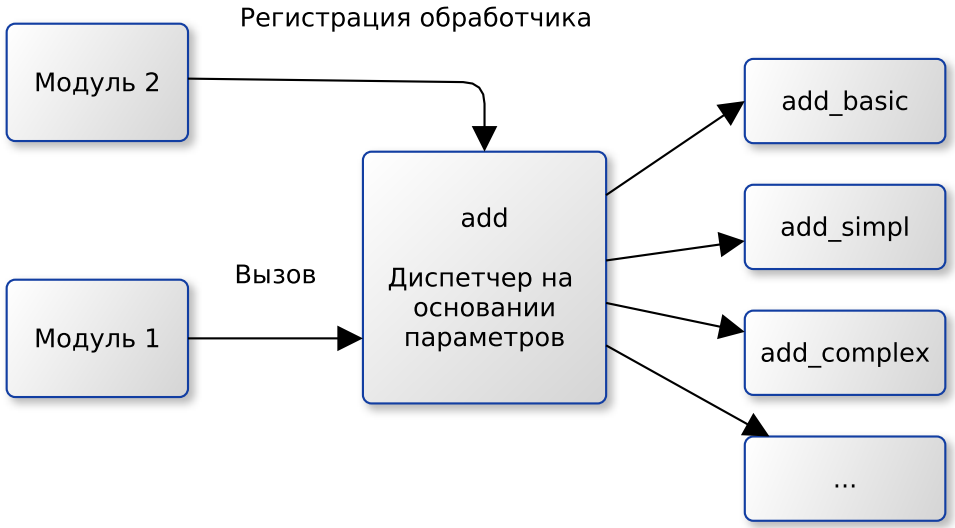
Более-менее работает для линейных иерархий.

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