

Пример №1 - Рациональные числа

- Сделать набор функций для работы с простыми дробями.
- Дробь хранится в виде тройки `r_type`, `numer`, `denom`
- `r_type` - тип дроби: "basic" или "auto_simpl"
- При операциях с "auto_simpl" в отличии от "basic" нужно сокращать числитель и знаменатель на НОД (наибольший общий делитель)
- Сделать поддержку функций `add`, `sub`, `mul`, `tostr`

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

$$\text{Алгоритм евклида : } a \geq b, \text{НОД}(a, b) = \begin{cases} b, a \div b = 0 \\ \text{НОД}(b, a \div b) \end{cases}$$

Рациональные числа - API

```
1  r1 = ( 'basic ' , 1 , 3)
2  r2 = ( 'basic ' , 1 , 2)
3
4  print tostr(r1) # 1/3
5
6  r3 = sub(r2 , r1)
7  print tostr(r3) # 1/6
8
9  r4 = add(r1 , r1)
10 print tostr(r4) # 2/6
11
12 r5 = ( 'auto_simpl ' , r1[1] , r1[2])
13 r6 = add(r5 , r5)
14 print tostr(r6) # 1/3
```

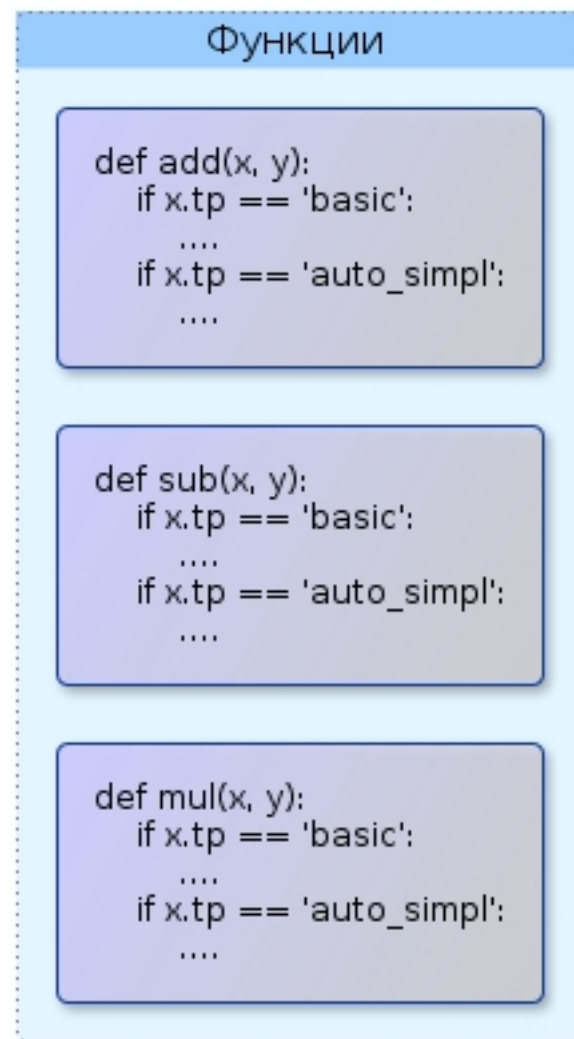
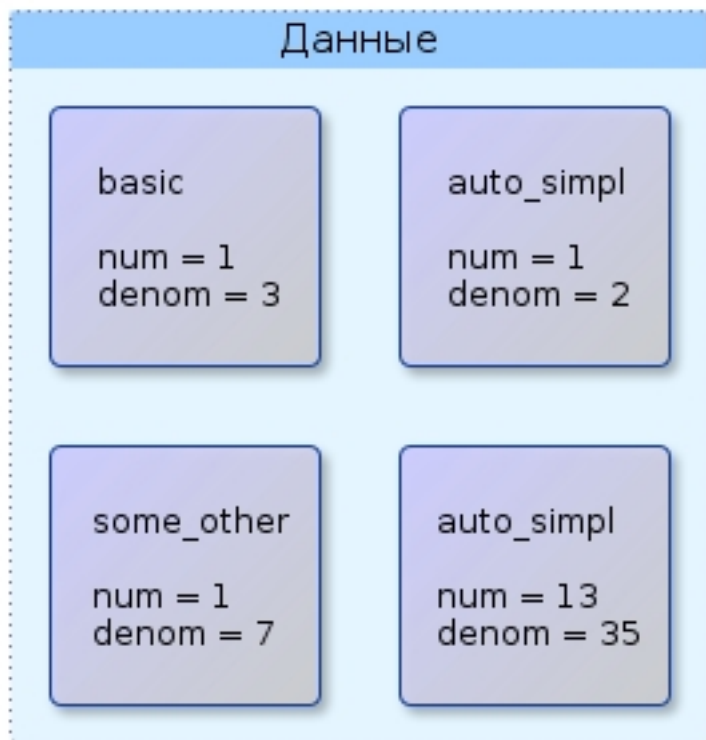
НОД

```
1  def gcd(x, y):
2      x = abs(x)
3      y = abs(y)
4      return _gcd(max(x, y), min(x, y))
5
6  def _gcd(x, y):
7      if y == 0:
8          return x
9      return gcd(y, x % y)
```

Рациональные числа - процедурный стиль

```
1  def add(x, y):
2      xtp, xnum, xdenom = x
3      ytp, ynum, ydenom = y
4      ndenom = xdenom * ydenom
5      nnum = xnum * ydenom + xdenom * ynum
6      if xtp == 'basic':
7          return ('basic', nnum, ndenom)
8      elif xtp == 'auto_simpl':
9          cur_gcd = gcd(nnum, ndenom)
10         return ('auto_simpl',
11                 nnum / cur_gcd,
12                 ndenom / cur_gcd)
13     assert False, "Unsupported_{}_rational_type" + \
14                   "_for_add" + xtp
```

```
1  def tostr(x):
2      return "{0[1]}/{0[2]}".format(x)
3
4  def sub(x, y):
5      ytp, ynum, ydenom = y
6      return add(x, (ytp, -ynum, ydenom))
```



Процедурный стиль - анализ

- `if xtp == 'basic':` - ужасно
- Декомпозиция логики затруднена
- Добавление новых типов требует изменения функции `add`
- Перегрузка функций решает часть проблем, но только часть

Перегрузка функции

```
1  BasicRN add( BasicRN v1 , BasicRN v2 )
2  {
3      . . . .
4  }
5
6  SimplifiedRN add( SimplifiedRN v1 , SimplifiedRN v2 )
7  {
8      . . . .
9  }
```


- (+) Одно имя для всех функций
- (+) Перегрузка по двум и более параметрам параметрам и хитрым правилам
- (+) Статическая перегрузка - никаких накладных расходов
- (-) Нужна перекомпиляция
- (-) Нужно явно знать типы во время компиляции
- (-) Все прототипы должны быть видны в точке компиляции

Еще решение

```
1  def add_2(x, y):
2      xtp, xnum, xdenom = x
3      if xtp == 'polynom':
4          # add polynoms
5      else:
6          return add(x, y)
```

Проблема

Весь код должен вызывать `add_2`.

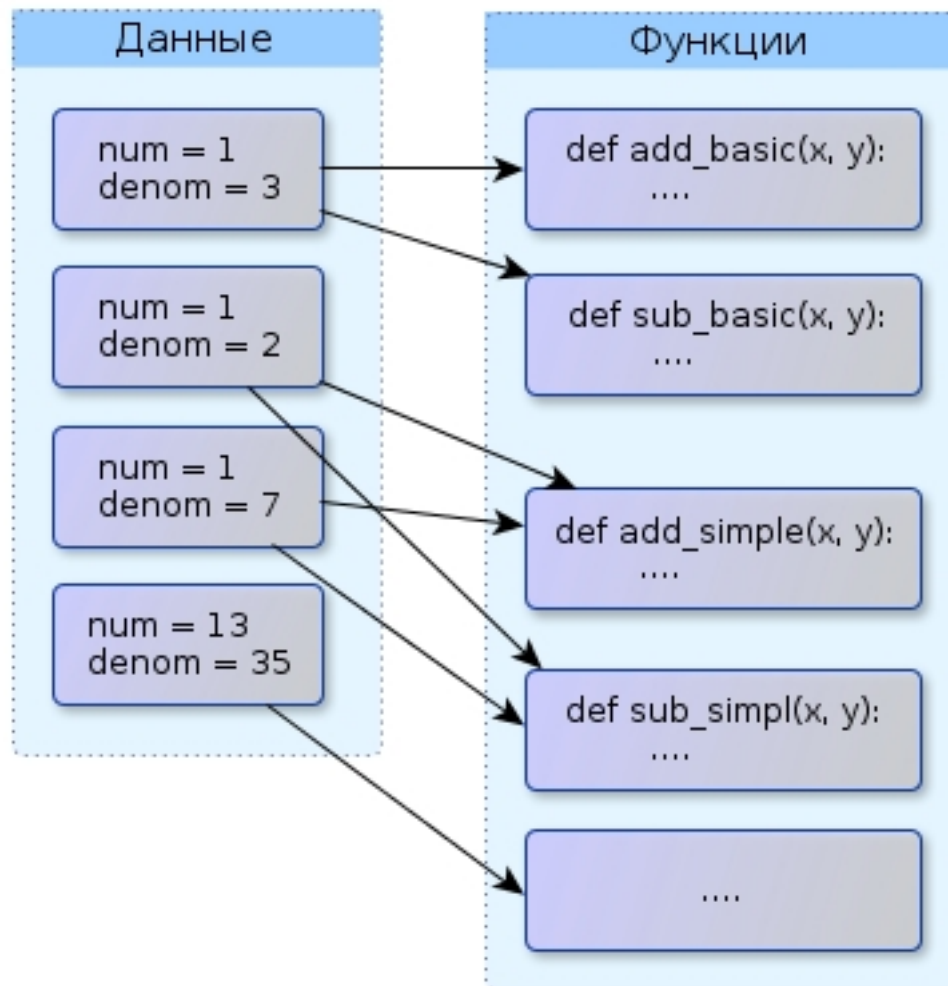
Рациональные числа - не совсем процедурный стиль

Один из вариантов решения - привязать функции к данным.

```
1      x = { 'num': 1 ,
2            'denom': 3 ,
3            'add': add_simplified ,
4            'sub': sub }
5
6      def add(x, y):
7          return x[ 'add' ](x, y)
```

Рациональные числа - не совсем процедурный стиль

```
1  def mk_basic(num, denom):
2      return { 'num':num, 'denom':denom, 'add':add_basic, 'sub'
3
4  def mk_simpl(num, denom):
5      return { 'num':num, 'denom':denom, 'add':add_simplified,
6
7  r1 = mk_basic(1, 3)
8  r2 = mk_basic(1, 2)
9  r3 = sub(r2, r1)
10
11 r5 = mk_simpl(1, 3)
12 r6 = add(r5, r5)
```



Не совсем процедурный стиль - анализ

- Кода стало больше
- Его расширение упростилось - не нужно модифицировать функцию `add`, при добавлении нового типа
- Типовые теги стали менее нужны - тип это операции, которые есть у него
- Вместо `func(x)` теперь `x['func'](x)`. Для упрощения вызова старая процедурная семантика оставлена, но внутри нее перенаправление на новый вызов

Не совсем процедурный стиль - анализ

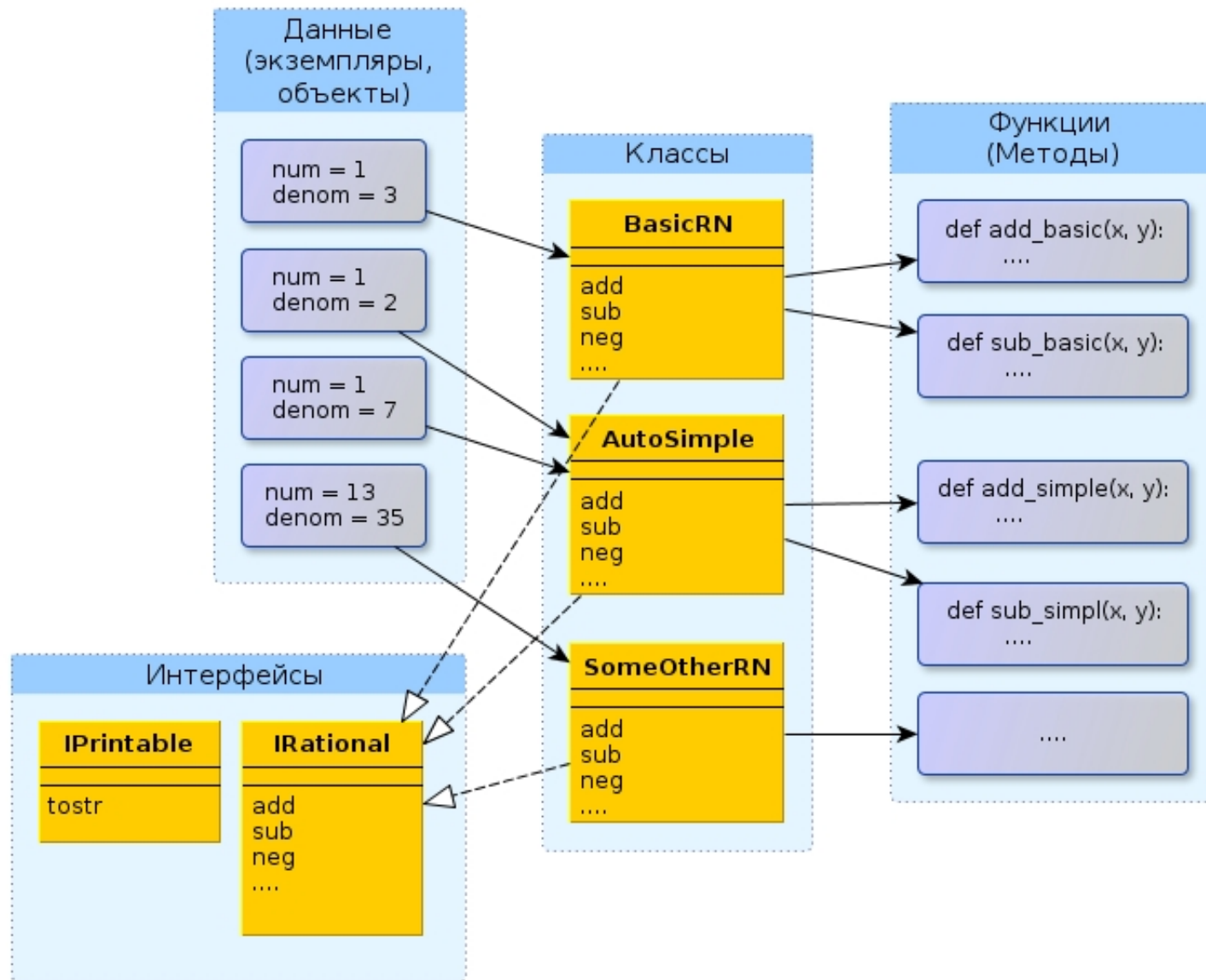
- Типовые теги иногда нужны.
- Каждый экземпляр содержит большое количество ссылок на одни и те же функции.
- Решение - вынесение всех методов в отдельный словарь, который все переменные данного типа используют совместно. Одновременно этот словарь становится типовым тегом.

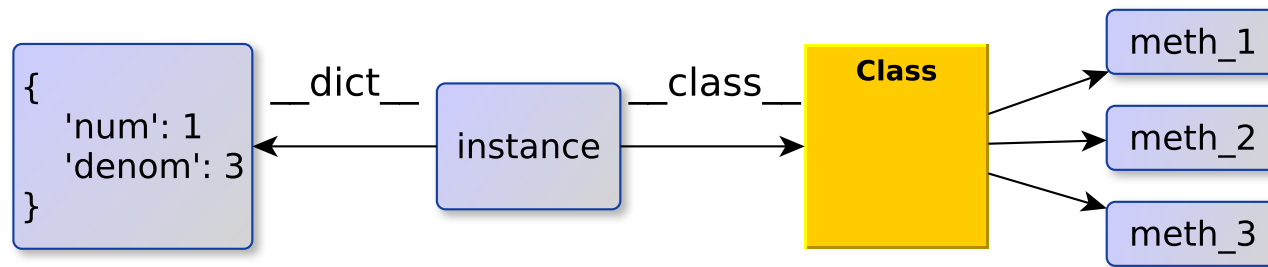
Рациональные числа - совсем не процедурный стиль

```
1 BasicRN = { 'add': add_basic ,
2             'sub': sub_basic ,
3             '__init__': mk_basic }
4
5 ASRN = { 'add': add_simplified ,
6          'sub': sub_simplified ,
7          '__init__': mk_simple }
8
9 def new( tp ):
10     def closure( *args , **kwargs ):
11         return tp[ '__init__' ]( *args , **kwargs )
12     return closure
13
14 x1 = new( BasicRN )( 1 , 2 )
15 x2 = new( ASRN )( 1 , 2 )
16
17 def add( x , y ):
18     return x[ '__class__' ][ 'add' ]( x , y )
```

Рациональные числа - совсем не процедурный стиль

- Шаблон, использованный в функции `add` часто используется в python и позволяет имитировать перегрузку функций
- Почти так устроено ООП в питоне внутри





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

`obj.__dict__` - словарь, содержащий атрибуты объекта

Рациональные числа - классы

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

```
19     def tostr(self):  
20         return "{0.num}/{0.denom}".format(self)
```

Рациональные числа - классы

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

Рациональные числа - классы

```
1  class AutoSimpl( BasicRational ):
2      "Auto_simplified_rational_number"
3
4      def add( self , y ):
5          res = BasicRational.add( self , y )
6          cur_gcd = gcd( res.num, res.denom )
7          res.num /= cur_gcd
8          res.denom /= cur_gcd
9      return res
```


Рациональные числа - классы

```
1  class AutoSimpl( BasicRational ):
2      "Auto_simplified_rational_number"
3
4      def __init__( self , num, denom ):
5          cur_gcd = gcd( num, denom )
6          self.num = num / cur_gcd
7          self.denom = denom / cur_gcd
```

ООП

- Шаблон проектирования, в котором данные имеют ссылки на функции их обработки
- Позволяет писать код, который сохраняет работоспособность при изменении данных в некоторых границах

ООП - в чем причина?

- Везде есть закон сохранения

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.

Проблема

$x.add(y) = y.add(x)!$, а может и вообще не сработать
В чем проблема? Как решать?

Проблема

`x.add(y) = y.add(x)!`, а может и вообще не сработать

В чем проблема? Как решать?

Нужна перегрузка `add` по обоим параметрам, что-то типа `(x,y).add(x,y)`.
Классическое ООП не предлагает решения.

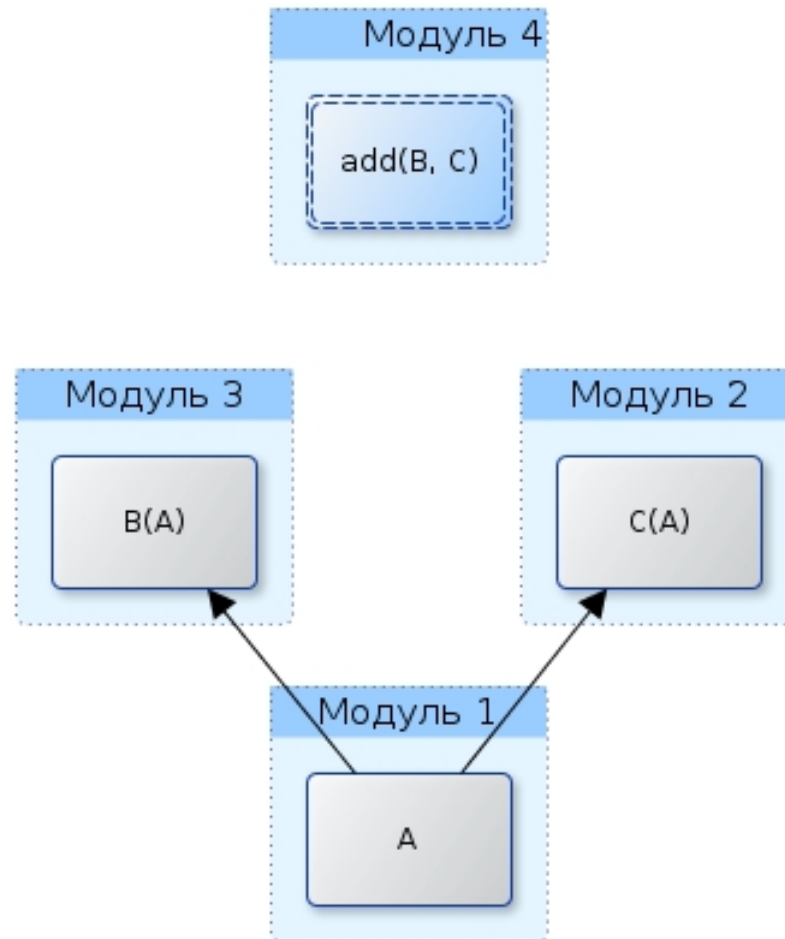
Вариант 1

```
1  class X(Y):
2      def add(self, y):
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)
```

Вариант 2

```
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)
```

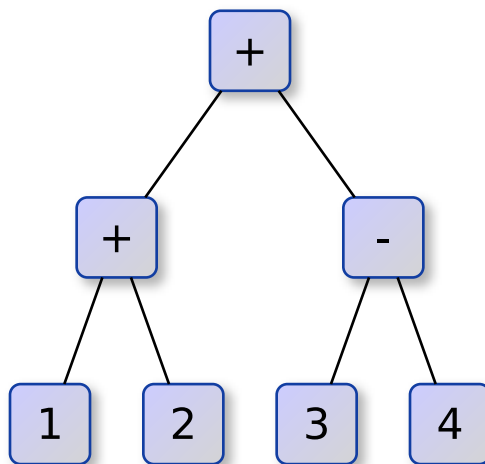

Опять проблема



Пример №2

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    print pexpr # ('+', ('+', 1, 2), ('-', 3, 4))
```



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
```

Задача 2

Добавить в выражение поддержку * и /

Решение 2

```
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))
```

Решение 2

```
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))
```

Решение 2

```
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
```

Задача 3

Добавить в выражение поддержку строк

Решение 2

```
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)
```

Решение 2

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

Проблемы?

Проблемы?

Функция parse. Почему?

- Ооп решает проблему передавая функции в код снаружи, привязанными к данным.
- Это не может решить проблему конструктора
- Нужно передвать parse снаружи

ООП vs Процедурный стиль

- (-) Часто больше кода
- (-) Усложняет язык
- (-) Разработка ООП дизайна требует больших навыков, чем процедурного
- (-) Добавление нового метода требует нелокальных изменений
- (-) Работает только если функция выбирается по типу одного параметра
- (-) Замедляет работу
- (+) Уменьшает пересечение имен
- (+) Код лучше структурирован
- (+) Избавляет от ручной проверки типов
- (+) Избавляет от знания конкретного типа данных
- (+) Во многих случаях значительно упрощает расширение
- (+) Позволяет 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

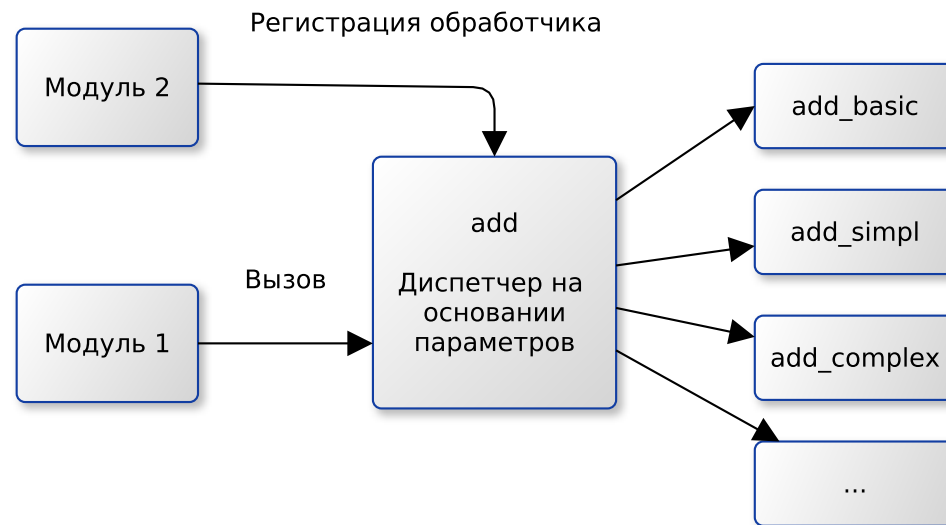
```
1  def to_xml ( obj ) :  
2      if isinstance ( obj , int ) :  
3          ...  
4      elif isinstance ( obj , str ) :  
5          ...  
6      else :  
7          res = obj . to_xml ( )  
8      return res
```


Альтернативы

- ?

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
- PEAK-Rules

Python ООП vs Процедурный стиль

- Возможность перегрузки функций
- Возможность перегрузки операторов
- $x.y \Rightarrow x.__dict__[y]$
- $x.func(\dots) \Rightarrow x.__class__.func(x, \dots)$
- $x + y \Rightarrow x.__add__(y)$
- $-x \Rightarrow x.__neg__()$

Добавление метода ко всем классам сразу требует нелокальных изменений

Один из вариантов решения - совместить стили.

```
1     def some_new_method( obj ):
2         if isinstance( obj , ClS1 ):
3             # code for CLS1
4         elif isinstance( obj , ClS2 ):
5             # code for CLS2
6         else :
7             return obj . some_new_method ( )
```

Рациональные числа - python way

```
1  class BasicRational(object):
2      "basic_rational_number"
3
4      def __init__(self, num, denom):
5          self.num = num
6          self.denom = denom
7
8      def __add__(self, y):
9          nd = x.denom * y.denom
10         nn = x.num * y.denom + x.denom * y.num
11         return self.__class__(nn, nd)
12
13     def __neg__(self):
14         return self.__class__(-nn, nd)
15
16     def __sub__(self, y):
17         return self.add(y.neg())
18
```

```
19         def __str__( self ):
20             return " { 0.num } / { 0.denom } ".format( self )
21
22         def __repr__( self ):
23             return str( self )
24
25     b1 = AutoSimpl(1 , 2)
26     b2 = AutoSimpl(1 , 3)
27     b3 = b2 - b1 - b1
28     print b1 , b2 , b3
```