

# Мимикрия, паттерны

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Среда, 5 октября 2016 года

# План занятия

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

## Зачем перегружать?

- Задание требуемой семантики у своих объектов:

```
class Foo:
    def __init__(self, value):
        self.value = value
Foo("hello") == Foo("hello")  # False
```

- Упрощение кода с математическими объектами:

```
# До
res = a.multiply(x).add(b.multiply(y)) \
    .add(c).multiply(5).add(2)
middle = vector1.add(vector2).multiply(0.5)
```

```
# После
res = (a * x + b * y + c) * 5 + 2
middle = (vector1 + vector2) / 2
```

# Почему не перегружать?

## Почему не перегружать?

- Разное поведение у похожих типов. Например: определим / как целочисленное деление. Лучше определить только //.

## Почему не перегружать?

- Разное поведение у похожих типов. Например: определим / как целочисленное деление. Лучше определить только //.
- ```
line a = /* ... */, b = /* ... */;  
if (a || b) { /* ... */ }
```

## Почему не перегружать?

- Разное поведение у похожих типов. Например: определим / как целочисленное деление. Лучше определить только //.
- ```
line a = /* ... */ , b = /* ... */ ;  
if (a || b) { /* ... */ }
```

Можно сказать, что || возвращает, параллельны ли прямые. Полностью изменяется семантика оператора.



## Почему не перегружать?

- Разное поведение у похожих типов. Например: определим / как целочисленное деление. Лучше определить только //.

```
line a = /* ... */, b = /* ... */;  
if (a || b) { /* ... */ }
```

Можно сказать, что || возвращает, параллельны ли прямые. Полностью изменяется семантика оператора.

- Скрывает сложные операции при чтении кода. Может мешать при отладке и поиске медленных мест — нет явного вызова функции:

```
a = 10 ** 10000
```

```
b = 10 ** 10000
```

```
# ...
```

```
result = a * b # Почему же тормозит?
```

- Неочевидное поведение: `vec1 * vec2`.

## Почему не перегружать?

- Разное поведение у похожих типов. Например: определим / как целочисленное деление. Лучше определить только //.

```
line a = /* ... */, b = /* ... */;  
if (a || b) { /* ... */ }
```

Можно сказать, что || возвращает, параллельны ли прямые. Полностью изменяется семантика оператора.

- Скрывает сложные операции при чтении кода. Может мешать при отладке и поиске медленных мест — нет явного вызова функции:

```
a = 10 ** 10000
```

```
b = 10 ** 10000
```

```
# ...
```

```
result = a * b # Почему же тормозит?
```

- Неочевидное поведение: `vec1 * vec2`. Векторное или скалярное произведение?

# Магические методы

- *Магическим* зовётся метод, название которого начинается и заканчивается на `__`. Например, `__init__` или `__str__`.
- Ничего магического, кроме предназначения, в них нет.
- Напрямую их вызывать не стоит!
- Перечислены в документации по группам. Объекты могут прикидываться:
  - 1 Числами.
  - 2 Чем-то, что можно сравнивать.
  - 3 Функциями.
  - 4 Коллекциями (массив, словарь, множество...).
  - 5 Итераторами (обслуживают цикл `for`).
  - 6 Чем-что, что можно автоматически закрывать (файл, сетевое соединение).
  - 7 И ещё много чем.
- Какая-то информация гуглится и [на русском](#).

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

## Как перегружать

```
class Natural:
    def __init__(self, value):
        assert value >= 1
        self.value = value
    def __add__(self, other):
        return Natural(self.value + other.value)
    def __sub__(self, other):
        return Natural(self.value - other.value)
    def __repr__(self): # Почему __str__
        return "Natural({})".format(self.value)
print(Natural(4) + Natural(3)) # Natural(7)
print(Natural(4) - Natural(3)) # Natural(1)
print(Natural(4).__sub__(Natural(3))) # Не надо так!
print(Natural(4) - Natural(4)) # AssertionError
```

## Разные типы-1

```
print(Natural(4) + 3)           # AttributeError
def better_add(self, other):
    if isinstance(other, Natural):
        return Natural(self.value + other.value)
    elif isinstance(other, int):
        return Natural(self.value + other)
    else:
        return NotImplemented
Natural.__add__ = better_add

print(Natural(4) + Natural(3)) # Natural(7)
print(Natural(4) + 3)         # Natural(7)
print(Natural(4) + "3")      # TypeError
print(3 + Natural(4))        # TypeError?
```

## Разные типы-2

```
print(3 + Natural(4))  # У int нет метода __add__ для Natural
int.__add__ = None     # И задать часто нельзя. И не надо.
Natural.__radd__ = Natural.__add__
```

```
print(Natural(4) + 3)      # Natural(7)
print(3 + Natural(4))     # Natural(7)
```

При вычислении выражения  $a + b$ :

- 1 Вызывается `a.__add__(b)`.
- 2 Если метод найден и не вернули `NotImplemented` — успех.
- 3 Иначе, если `a` и `b` разных типов, вызывается `b.__radd__(a)`<sup>1</sup>.
- 4 Если не помогло — неуспех.

Также есть методы `__rsub__`, `__rmul__` и другие.

---

<sup>1</sup>reverse add

## Разные типы-3

```
class Foo:
    def __add__(self, other):
        print("add")
        return NotImplemented
    def __radd__(self, other):
        print("radd")
        return self
```

```
Natural(3) + Foo() # radd
```

```
Foo() + Natural(3) # add, TypeError
```



- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - **Синтаксис сравнений**
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

# Сравнения-1

```
class Natural:
    def __init__(self, value):
        self.value = value
    def __lt__(self, other): # Less than
        return self.value < other.value
    def __le__(self, other): # Less or equal
        return self.value <= other.value
    def __eq__(self, other):
        return self.value == other.value

ONE, TWO = map(Natural, [1,2])
print(ONE < TWO, ONE > TWO)      # True False
print(ONE <= TWO, ONE >= TWO)   # True False
print(ONE == TWO, ONE != TWO)   # False True
print(ONE == ONE, ONE != ONE)   # True False
```

## Сравнения-2

- Операторы `__lt__` и `__gt__` считаются отражениями друг друга. Почти как `__add__` и `__radd__`.
- Оператор `__ne__` по умолчанию берёт отрицание от `__eq__`.
- Оператор `__lt__` автоматически из `__le__` и `__eq__` не выводится.

Можно использовать декоратор `total_ordering`:

```
from functools import total_ordering
@total_ordering # Магия.
class Natural:
    # Надо задать метод __eq__ и один из четырёх сравнивающих.
    #
    # Остальное сгенерируется.
```

# Хэш-таблицы-1

```
a = { Natural(1): 1 } # TypeError: unhashable type
```

```
class Natural:
    def __init__(self, value):
        self.value = value
    def __eq__(self, other):
        return self.value == other.value
    def __hash__(self):
        return hash(self.value)
    def __repr__(self):
        return "Natural({})".format(self.value)
```

```
a = {Natural(x): x for x in range(5)}
print(a)
```

## Хэш-таблицы-2

- `__hash__` вызывается функцией `hash`, когда элемент кладут в хэш-таблицу. Должна вернуть `int`.
- Требование: если `a == b`, то `hash(a) == hash(b)` (в обратную сторону необязательно).
- Если определён `__hash__`, то обязательно определить `__eq__`.
- Для помещения в хэш-таблицу методы `__lt__` необязательны.
- В языке Java идеология похожа: методы `equals()` и `hashCode()`.
- Если объект может измениться (*мутабельный*), то `__hash__` определять не стоит. Почему?

## Хэш-таблицы-2

- `__hash__` вызывается функцией `hash`, когда элемент кладут в хэш-таблицу. Должна вернуть `int`.
- Требование: если `a == b`, то `hash(a) == hash(b)` (в обратную сторону необязательно).
- Если определён `__hash__`, то обязательно определить `__eq__`.
- Для помещения в хэш-таблицу методы `__lt__` необязательны.
- В языке Java идеология похожа: методы `equals()` и `hashCode()`.
- Если объект может измениться (*мутабельный*), то `__hash__` определять не стоит. Почему? Потому что если он изменится, пока лежит в хэш-таблице, она об этом не узнает и сломается.

## Хэш-таблицы-3

Чем плоха реализация строки ниже?

```
class Str:
    def __init__(self, value): self.value = value
    def __eq__(self, other): return self.value == other.value
    def __hash__(self):
        result = 0
        for c in self.value:
            result = result * 239017 + ord(c)
        return result
```

## Хэш-таблицы-3

Чем плоха реализация строки ниже?

```
class Str:
    def __init__(self, value): self.value = value
    def __eq__(self, other): return self.value == other.value
    def __hash__(self):
        result = 0
        for c in self.value:
            result = result * 239017 + ord(c)
        return result

print(hash("a" * 1000))
print(hash(Str("a" * 1000)))
print(hash("a" * 100000))
print(hash(Str("a" * 100000)))
```



## Хэш-таблицы-3

Чем плоха реализация строки ниже?

```
class Str:
    def __init__(self, value): self.value = value
    def __eq__(self, other): return self.value == other.value
    def __hash__(self):
        result = 0
        for c in self.value:
            result = result * 239017 + ord(c)
        return result

print(hash("a" * 1000))
print(hash(Str("a" * 1000)))
print(hash("a" * 100000))
print(hash(Str("a" * 100000)))
```

Тормозит, потому что в Python int автоматически преобразуется в длинную арифметику, а не переполняется.

## Упражнение

Реализуйте класс `Natural` для хранения натуральных чисел с поддержкой сложения, умножения (в том числе с `int`), хэширования, вывода на экран и сравнений:

```
class Natural:
    # ... ваш код здесь ...

# Тесты:
l = list(map(Natural, [5, 2, 3, 4, 1, 4]))
print(Natural(2) + Natural(3))
print(20 * Natural(100) + 5)
print(sum(l))
print(sorted(l))
d = {Natural(1): 10, Natural(2): 20}
print(list(d.keys())) # [Natural(1), Natural(2)]
```

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - **Наследование**
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

# Беда?

```
class Point:
    def __init__(self, x, y): self.x, self.y = x, y
    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)
class PointWithId(Point):
    def __init__(self, x, y, id):
        super(PointWithId, self).__init__(x, y)
        self.id = id
    def __eq__(self, other): # Принцип Лисков?
        return ((self.x, self.y, self.id) ==
                (other.x, other.y, other.id))
base, child = Point(1, 2), PointWithId(1, 2, 3)
print(base.__eq__(child))           # True
print(child.__eq__(base))           # AttributeError
print(base == child, child == base) # ???
```

# Беда!

Неясно, как сравнивать объекты разных типов на равенство в общем случае:

- 1 Можно запретить сравнение объектов разных типов, но нарушится ???
- 2 Можно сравнивать только по общим полям, но тогда нарушится ???

# Беда!

Неясно, как сравнивать объекты разных типов на равенство в общем случае:

- 1 Можно запретить сравнение объектов разных типов, но нарушится принцип подстановки.
- 2 Можно сравнивать только по общим полям, но тогда нарушится ???

# Беда!

Неясно, как сравнивать объекты разных типов на равенство в общем случае:

- 1 Можно запретить сравнение объектов разных типов, но нарушится принцип подстановки.
- 2 Можно сравнивать только по общим полям, но тогда нарушится транзитивность:

$$(child_1 = base) \wedge (base = child_2) \not\Rightarrow child_1 = child_2$$

# Беда!

Неясно, как сравнивать объекты разных типов на равенство в общем случае:

- 1 Можно запретить сравнение объектов разных типов, но нарушится принцип подстановки.
- 2 Можно сравнивать только по общим полям, но тогда нарушится транзитивность:

$$(child_1 = base) \wedge (base = child_2) \not\Rightarrow child_1 = child_2$$

В любом случае, суперкласс ничего про детей не знает, поэтому вызывать его метод скорее бессмысленно. Python всегда вызовет метод подкласса (будь то `__eq__`, `__le__` или `__ge__`).

В других языках может быть по-другому!

Надёжнее всего считать объекты разных типов разными.



## Вообще беда

С сортировкой ещё хуже. Надо определить какой-то линейный порядок на всех объектах: не только транзитивность, но ещё и согласованность с равенством, например:

$$a \neq b \iff (a < b) \vee (a > b)$$

Как разрулить в общем случае?

## Вообще беда

С сортировкой ещё хуже. Надо определить какой-то линейный порядок на всех объектах: не только транзитивность, но ещё и согласованность с равенством, например:

$$a \neq b \iff (a < b) \vee (a > b)$$

Как разрулить в общем случае? Запретить сравнивать объекты разных типов!

Как тогда не нарушить принцип подстановки?

## Вообще беда

С сортировкой ещё хуже. Надо определить какой-то линейный порядок на всех объектах: не только транзитивность, но ещё и согласованность с равенством, например:

$$a \neq b \iff (a < b) \vee (a > b)$$

Как разрулить в общем случае? Запретить сравнивать объекты разных типов!

Как тогда не нарушить принцип подстановки? Запретить сравнивать объекты типа «суперкласс», если только не известно заведомо, что они одного типа.

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - **Притворяемся функцией**
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

## Притворяемся функцией

```
class Summer: # Сумма, а не лето :(
    def __init__(self, k):
        self.k = k
    def __call__(self, *args):
        return self.k * sum(args)

s = Summer(3)
print(s())      # 0
print(s(1))    # 3
print(s(1, 10)) # 33
```

Всё, что имеет метод `__call__`, может быть вызвано. И наоборот:

```
def foo(): print("foo")
print(foo.__call__)
foo.__call__()
print(foo.__call__.__call__)
```

# Но зачем?

- Объект каком-то смысле представляет собой функцию. Например: «преобразование плоскости», «логгер» или «выражение от одной переменной».
- В некоторых других языках это единственный способ сделать функцию с некоторым внутренним состоянием (кроме глобальных переменных).
- Все проблемы с перегрузкой операторов остаются. Не злоупотребляйте!

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 **Притворяемся коллекцией**
  - **Словарь**
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

# Словарь

```
class KeyToPrependedKey:
    def __init__(self, prefix):
        self.prefix = prefix
    def __getitem__(self, name):
        return self.prefix + name
a = KeyToPrependedKey("foo_")
print(a["bar"]) # foo_bar
```

Ещё бывают методы `__setitem__`, `__delitem__`, `__len__`, `__contains__`, `__reversed__`, `__missing__`, `__iter__` (см. дальше).



## Массив со срезами

Срезы передаются просто как объект типа slice:

```
class RangeMeasurer:
    def __getitem__(self, s):
        if isinstance(s, slice):
            return s.stop - s.start
        else:
            return 1

print(RangeMeasurer()[2:4])      # 2
print(RangeMeasurer()[4:2])     # -2
print(RangeMeasurer()[2:4:2])   # 2
print(RangeMeasurer()[ :2])     # ???
print(RangeMeasurer()[2: ])     # ???
```

## Массив со срезами

Срезы передаются просто как объект типа `slice`:

```
class RangeMeasurer:
    def __getitem__(self, s):
        if isinstance(s, slice):
            return s.stop - s.start
        else:
            return 1

print(RangeMeasurer()[2:4])      # 2
print(RangeMeasurer()[4:2])     # -2
print(RangeMeasurer()[2:4:2])   # 2
print(RangeMeasurer()[ :2])     # ???
print(RangeMeasurer()[2: ])     # ???
```

Все случаи надо либо разбирать руками, либо использовать функцию `slice.range()`.

# Присваивание-1

Оператор `=` в Python в общем случае перегрузить нельзя: `a = b` всегда изменит значение `a` на `b` копированием ссылки.

Но можно перегрузить в частных случаях, например, когда мы пишем `a.foo = b` или `a["foo"] = b`.

В C++ наоборот: перегрузить можно только оператор `=` в общем случае, поэтому возникают прокси-объекты (тут не рассматриваем).

## Присваивание-2

```
class KeyPrepender:
    def __init__(self, backend, prefix):
        self.backend = backend
        self.prefix = prefix
    def __getitem__(self, name):
        return self.backend[self.prefix + name]
    def __setitem__(self, name, val):
        self.backend[self.prefix + name] = val

d = {}
cache = KeyPrepender(d, "my_")
cache["foo"] = 10
print(cache["foo"])    # 10
print(d)                # {'my_foo': 10}
```

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - **Итераторы**
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

## Пример итератора

```
class CountdownIterator:
    def __init__(self, start): self.value = start
    def __iter__(self):
        return self # Так надо.
    def __next__(self):
        if self.value < 1:
            raise StopIteration
        self.value -= 1
        return self.value + 1

v = CountdownIterator(5)
print(v)
print(next(v)) # 5
print(next(v)) # 4
```

## Коллекции и итераторы

Всё, у чего есть метод `__iter__`, можно запихнуть в цикл `for` и другие интересные места:

```
a = [1, 2, 3]
print([ x for x in a ]) # [1, 2, 3]
it = iter(a) # Вызываем __iter__.
print(next(it)) # 1
print(next(it)) # 2
print(next(it)) # 3
print(next(it)) # StopIteration
```

```
a = CountdownIterator(5)
print(list(a)) # 5 4 3 2 1
print(list(a)) # ???
```

# Что произошло

- Если коллекция закончилась, итератор должен вызвать `raise StopIteration`.
- Итератор — штука одноразовая, переиспользовать нельзя.
- Метод `__iter__` у коллекций возвращает **новый** итератор, указывающий на начало коллекции.
- Метод `__iter__` есть у каждого итератора, чтобы их можно было использовать там же, где и коллекции.
- В других языках интерфейсы коллекции и итератора разнесены более явно.



# StopIteration

StopIteration — это так называемое *исключение*.

- Исключения — один из механизмов обработки ошибок (исключительных ситуаций):
  1. Произошла ошибка.
  2. Создали объект класса *исключение* (или подкласса: `TypeError`, `KeyError`, ...).
  3. Кинули его (сленг) командой `raise SomeException()`.
  4. Исключение пошло вверх по стеку вызовов до ближайшего обработчика, соответствующего типа.
  5. Обработчик решает, что делать с исключением.
- Можно воспринимать как такой «return из всех функций сразу до ближайшего обработчика».
- Более подробно пока не рассматриваем.

# Как ловить StopIteration

```
def print_next(it):
    print(next(it))
def print_two_next(it):
    print_next(it)
    print_next(it)
    print("Printed two items")
try: # Начало блока, где может вылететь исключение.
    print_two_next(iter([1]))
    print("Finished")
except StopIteration: # Обработчик исключения StopIteration.
    print("Stopped")
```

## Упражнение

Напишите «свою» реализацию цикла for:

```
def foreach(items, callback):
    # Перепишите эту функцию без цикла for.
    for item in items:
        callback(item)

# Примеры.
foreach([1, 2, 3], print) # 1 2 3
foreach([1, 2, 3], lambda x: print(x, end=";")) # 1;2;3;
print()
foreach(CountDownIterator(3), print) # 3 2 1

foreach({"a": False, "b": True}.items(), print)
# ('a', False) ('b', True) в любом порядке.
```

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

## Кто такие

```
with open("file.txt", "r") as f:
    f.read()
# Почти то же самое, что:
f = open("file.txt", "r")
f.__enter__()
f.read()
f.__exit__()
```

Тут `f` — *менеджер контекста*, потому что он реализует методы `__enter__` и `__exit__`. Жизненный цикл:

- 1 Создали менеджер, вызвался `__init__`.
- 2 Вошли в блок `with`, вызвался `__enter__`.
- 3 Вышли из блока (в том числе при помощи `return`), вызвался `__exit__`.

# Зачем

## Примеры:

- Файлы: открываем и закрываем.
- Блокировки: их можно создавать, а дальше захватывать для эксклюзивного доступа (и потом отпустить).
- Папка с временными файлами: создать, потом почистить.
- Смена текущей папки на время работы функции.

## Применение:

- Когда ваш объект использует какие-то ресурсы, которые надо закрывать: файлы, сетевые соединения.
- Менеджер может предполагать, что он создаётся и сразу используется в блоке `with`. Тогда разница между `__init__` и `__enter__` тонкая. Пример: файл.
- Менеджер может предполагать, что за время жизни его могут использовать в разных блоках `with`. Тогда надо различать `__init__` и `__enter__`. Пример: блокировка.

## Упражнение

Напишите менеджер контекста для смены текущей папки (функция `os.chdir()`):

```
class ChangeDir:
    # ... ваш код здесь ...

print(os.getcwd())           # /home/foo/bar
with ChangeDir("../"):      # as можно опускать
    print(os.getcwd())      # /home/foo
print(os.getcwd())          # /home/foo/bar
```

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor



## Что такое паттерн

- *Паттерн (или шаблон проектирования)* — это какая-то стандартная конструкция для решения каких-то архитектурных задач.
- Другими словами: какие абстракции и интерфейсы полезно использовать в каких ситуациях.
- Некоторые паттерны идут «от капитана» (можно назвать «очевидным здравым смыслом»), некоторые более хитры.
- Часто может казаться, что без паттерна легко обойтись, потому что в программе нужен очень частный случай. Зато если требования поменяются — можно огрести.
- Легко перегнуть палку: «абстрактные фабрики абстрактных фабрик» и прочие радости.
- Мы пройдемся очень поверхностно. Вообще есть большие книжки (от «банды четырёх») и курсы, где про это рассказывают.
- Часто называются по-английски.

# Поведенческие шаблоны

Рассказывают, как объекты могут между собой взаимодействовать.

Примеры:

- *Итератор (Iterator)*: уже познакомились. Позволяет итерироваться по произвольным коллекциям.
- *Наблюдатель (Observer)*: если некоторые объекты могут рассылать события, а некоторые должны на них реагировать, то можно создать интерфейс «наблюдатель за событием» и вспомогательные классы для рассылки событий. Тогда наблюдателю надо лишь добавиться в нужный список, а инициатору события — вызвать метод «оповести наблюдателей из списка».
- *Посетитель (Visitor)*: будет в домашнем задании, разберём позже.

# Структурные шаблоны

Рассказывают, как компоновать между собой классы и объекты.

Примеры:

- *Адаптер (Adapter)*: если у нас есть класс с интерфейсом *A*, а нам нужно передать куда-то класс с другим интерфейсом *B* (другие название), то можно создать класс, который просто будет конвертировать вызовы интерфейса *B* в вызовы *A*.
- *Компоновщик (Composite)*: если нам часто нужно совершать одинаковые операции над разными объектами (например, отрисовать элементы окна на экране), то их можно объединить в коллекцию, которая предоставляет общий интерфейс для этих объектов.

# Порождающие шаблоны

Рассказывают, как создавать и компоновать объекты в коде. Примеры:

- *Строитель (Builder)*: если есть объект с очень сложным конструктором, то можно создать промежуточный объект, который будет «накапливать» в себе параметры конструктора, а потом создаст объект:

```
def create_button():
    builder = ButtonBuilder()
    builder.set_text("Кнопка")
    if some_complex_condition(): builder.set_disabled()
    return builder.build()
```

- *Абстрактная фабрика (Abstract factory)*: если объекты постоянно требуют какого-то включения в систему, то можно не вызывать конструкторы напрямую, а выделить кусок системы, который будет правильным образом конструировать объекты.

- 1 Перегрузка операторов
  - Мотивация
  - Немного магии
  - Синтаксис арифметики
  - Синтаксис сравнений
  - Наследование
  - Притворяемся функцией
- 2 Притворяемся коллекцией
  - Словарь
  - Итераторы
- 3 Менеджеры контекста
- 4 Паттерны
  - Классификация
  - Паттерн Visitor

## Постановка задачи

Пусть есть разношёрстные классы, например, элементы синтаксического дерева языка ЯТЪ: `Scope`, `Function`, `Read`, `Write`. И ещё есть операции над этими объектами: вывести красиво отформатированный код, оптимизировать кусочек дерева (например, заменить `2+2` на `4`), скомпилировать программу.

Вопрос: где описывать эти операции?

## Постановка задачи

Пусть есть разношёрстные классы, например, элементы синтаксического дерева языка ЯТЪ: `Scope`, `Function`, `Read`, `Write`. И ещё есть операции над этими объектами: вывести красиво отформатированный код, оптимизировать кусочек дерева (например, заменить `2+2` на `4`), скомпилировать программу.

Вопрос: где описывать эти операции? Несколько вариантов:

- 1 Добавить методы `pretty_print`, `optimize`, `compile` всем классам.
- 2 Сделать одну функцию на операцию, которая может вызывать сама себя рекурсивно и руками проверяет тип объекта.
- 3 Паттерн *Visitor*: сделать класс, соответствующий операции, с методами «обработай `Scope`», «обработай `Function`», и так далее.

## Пример внутри классов

```
import random
class Cat:
    def pat_head(self): print("Purr!")
    def rub_belly(self): print("Don't you dare!")
    def happiness(self): return random.randint(1, 5)
    def pet(self): self.pat_head()
    def is_safe(self): return self.happiness() >= 3
class Dog:
    def pat_head(self): print("I'm happy!")
    def rub_belly(self): print("I'm very happy!")
    def tail_wagging(self): return True
    def pet(self): self.rub_belly()
    def is_safe(self): return self.tail_wagging()
```



## Пример в функциях

```
import random

class Cat:
    def pat_head(self): print("Purr!")
    def rub_belly(self): print("Don't you dare!")
    def happiness(self): return random.randint(1, 5)

class Dog:
    def pat_head(self): print("I'm happy!")
    def rub_belly(self): print("I'm very happy!")
    def tail_wagging(self): return True

def pet(a):
    if isinstance(a, Cat): a.pat_head()
    elif isinstance(a, Dog): a.rub_belly()

def is_safe(a):
    if isinstance(a, Cat): return a.happiness() >= 3
    elif isinstance(a, Dog): return a.tail_wagging()
```

## Возможные проблемы

Если добавляем методы в классы, то:

## Возможные проблемы

Если добавляем методы в классы, то:

- Логика операции оказывается разнесённой по разным кускам кода.
- Добавили операцию — надо проверить, что ни в одном из классов не забыли (Python-то не проверяет соответствие интерфейсам).
- При добавлении операций изменяются интерфейсы классов, придётся всё перекомпилировать.

## Возможные проблемы

Если добавляем методы в классы, то:

- Логика операции оказывается разнесённой по разным кускам кода.
- Добавили операцию — надо проверить, что ни в одном из классов не забыли (Python-то не проверяет соответствие интерфейсам).
- При добавлении операций изменяются интерфейсы классов, придётся всё перекомпилировать.

Если делаем одну функцию, то:

## Возможные проблемы

Если добавляем методы в классы, то:

- Логика операции оказывается разнесённой по разным кускам кода.
- Добавили операцию — надо проверить, что ни в одном из классов не забыли (Python-то не проверяет соответствие интерфейсам).
- При добавлении операций изменяются интерфейсы классов, придётся всё перекомпилировать.

Если делаем одну функцию, то:

- Куча неприятного кода для определения типа, захочется разделить определение типа и содержательную обработку.
- В некоторых языках проверить тип переданного объекта в run time очень сложно (например, C — надо что-то руками делать).
- В статически типизированных языках надо ещё и изменять тип переменной.

# Волшебный единорог

Паттерн Visitor (*посетитель*):

- 1 Создаём интерфейс Visitor с функциями `visit_scope`, `visit_function`, `visit_read...`<sup>2</sup>.
- 2 Требуем, чтобы каждый класс имел функцию `accept(visitor)`, которая бы вызывала у параметра нужный метод.
- 3 Для определения операции достаточно создать новый класс, реализующий интерфейс Visitor.
- 4 Для вызова операции на элементе достаточно вызывать `item.accept(visitor)`.

---

<sup>2</sup>в некоторых языках (C++, Java) все методы назовут `visit`, а отличия будут лишь в типе аргумента

## Пример-1

```
import random
class Cat:
    def pat_head(self): print("Purr!")
    def rub_belly(self): print("Don't you dare!")
    def happiness(self): return random.randint(1, 5)
    def visit(self, v): return v.visit_cat(self)
class Dog:
    def pat_head(self): print("I'm happy!")
    def rub_belly(self): print("I'm very happy!")
    def tail_wagging(self): return True
    def visit(self, v): return v.visit_dog(self)
```

## Пример-2

```
class PetVisitor:
    def visit_cat(self, a): a.pat_head()
    def visit_dog(self, a): a.rub_belly()
class IsSafeVisitor:
    def visit_cat(self, a): return a.happiness() >= 3
    def visit_dog(self, a): return a.tail_wagging()

Cat().visit(PetVisitor())    # Purr!
Dog().visit(PetVisitor())    # I'm very happy!
print(Cat().visit(IsSafeVisitor()))    # random
print(Dog().visit(IsSafeVisitor()))    # True
```



## Замечание про возвращаемые значения

- 1 В статически типизированных языках нельзя вернуть произвольное значение из методов, надо указывать какой-то тип.
- 2 При этом указывать тип «всё что угодно» нехорошо, так как теряется безопасность. Фиксировать конкретный тип для всех посетителей тоже нехорошо.
- 3 Поэтому часто считают, что `visit/ассерт` вообще ничего не возвращают. А в Python легко забыть `return` и не заметить :)
- 4 Если посетитель что-то должен вычислять, то он хранит результат вычисления внутри себя:

```
class IsSafeVisitor:
    def visit_cat(self, a):
        self.result = a.happiness() >= 3
    def visit_dog(self, a):
        self.result = a.tail_wagging()
```