Разработка → Перестаньте писать классы

Проектирование и рефакторинг*, Python*

Признак того, что объект не должен быть классом — если в нём всего 2 метода, и один из них — инициализация, __init__. Каждый раз видя это, подумайте: «наверное, мне нужна просто одна функция».

Каждый раз когда из написанного класса вы создаёте всего один экземпляр, используете только раз и тут же выбрасываете, следует думать: «ой, надо бы это отрефакторить! Можно сделать проще, намного проще!»

Перевод доклада Джэка Дидриха, одного из ключевых разработчиков языка Питон. Доклад прозвучал 9 марта 2012 на конференции PyCon US.

Все из вас читали Дзэн Питона, наверное много раз. Вот несколько пунктов из него:

- Простое лучше сложного
- Плоское лучше вложенного
- Важна читаемость
- Если программу трудно объяснить, она плохая
- Если программу легко объяснить, возможно, она хороша́

Написал этот текст Тим Питерс. Он умнее и вас, и меня. Сколько вы знаете людей, в честь которых назвали алгоритм сортировки? Вот такой человек написал Дзэн Питона. И все пункты гласят: «Не делай сложно. Делай просто.» Именно об этом и пойдёт речь.

Итак, в первую очередь, не делайте сложно, там, где можно сделать проще. Классы очень сложны или могут быть очень сложны. Но мы всё равно делаем сложно, даже стараясь делать проще. Поэтому в этом докладе мы прочитаем немного кода и узнаем, как заметить, что идём неверным путём, и как выбраться обратно.

На своей работе я говорю коллегам: «Я ненавижу код, и хочу чтобы его было как можно меньше в нашем продукте.» Мы продаём функционал, не код. Покупатели у нас не из-за кода, а из-за широкого функционала. Каждый раз, когда код удаляется, это хорошо. Нас четверо, и в последний год мы перестали считать количество строк в продукте, но продолжаем вводить новый функционал.

Классы

Из этого доклада вам в первую очередь нужно запомнить вот этот код. Это крупнейшее злоупотребление классами, встречающееся в природе.

```
class Greeting(object):
    def __init__(self, word='Hello'):
        self.word = word

    def greet(self, name):
        return '%s, %s!' % (self.word, name)

>>> greeting = Greeting('Hola')
>>> greeting.greet('Jorge')
Hola, Jorge!
```

Это не класс, хотя он похож на класс. Имя — существительное, «приветствие». Он принимает аргументы и сохраняет их в __init__. Да, выглядит как класс. У него есть метод, читающий состояние объекта и делающий что-то ещё, как в классах. Внизу написано, как этим классом пользуются: создаём экземпляр Приветствия и затем используем это Приветствие чтобы сделать что-то ещё.

Но это не класс, или он не должен быть классом. Признак этого — в нём всего 2 метода, и один из них — инициализация, __init__. Каждый раз видя это, подумайте: «наверное, мне нужна просто одна функция».

Каждый раз когда из написанного класса вы создаёте всего один экземпляр, используете только раз и тут же выбрасываете, следует думать: «ой, надо бы это отрефакторить! Можно сделать проще, намного проще!»

```
def greet(name):
    ob = Greeting('превед')
    print ob.greet(name)
    return
```

Эта функция состоит из 4 строк кода. А вот как можно сделать то же самое всего за 2 строки:

```
def greet(greeting, name):
    return '%s, %s!' % (greeting, name)

import functools
greet = functools.partial(greet, 'превед')
greet('красавчик')
```

Если вы всё время вызываете функцию с тем же первым аргументом, стандартной библиотеке

есть инструмент! functools.partial. Вот посмотрите в код выше: добавляете аргумент, и результат можно вызывать многократно.

Не знаю, у скольких из вас диплом в ИТ, у меня он есть. Я учил такие понятия как

— разделение полномочий
— уменьшение связанности кода
— инкапсуляция
— изоляция реализации

С тех пор как я закончил вуз, 15 лет я этих терминов не употреблял. Слыша эти слова, знайте, вас дурят. Эти термины сами по себе не требуются. Если их используют, люди имеют в виду совершенно разное, что только мешает разговору.

Пример: брюки превращаются...

Многие из вас пользуются в повседневной работе сторонними библиотеками. Каждый раз когда надо пользоваться чужим кодом, первое, что нужно сделать — прочитать его. Ведь неизвестно, что там, какого качества, есть ли у них тесты и так далее. Нужно проверить код прежде чем включать его. Иногда читать код бывает тяжко.

Сторонняя библиотека API, назовём её ShaurMail, включала 1 пакет, 22 модуля, 20 классов и 660 строк кода. Мне пришлось всё это прочитать прежде чем включить в продукт. Но это был их официальный API, поэтому мы пользовались *им*. Каждый раз когда приходили обновления API, приходилось просматривать диффы, потому что было неизвестно, что они поменяли. Вы посылали патчи — а в обновлении они появились?

660 строк кода, 20 классов — это многовато, если программе нужно только дать список адресов электронной почты, текст письма и узнать, какие письма не доставлены, и кто отписался.

Что такое злоупотребление классами? Часто люди думают, что им понадобится что-то в будущем.... Не понадобится. Напишите всё, когда потребуется. В библиотеке ШаурМаил есть модуль ШаурХэш, в котором 2 строки кода:

```
class ShaurHash(dict):
    pass
```

Кто-то решил, что позже понадобится надстройка над словарём. Она не понадобилась, но везде в коде остались строки, как первая:

```
my_hash = ShaurMail.ShaurHash.ShaurHash(id='cat')
```

```
d = dict(id='cat')
```

```
d = {'id': 'cat'}
```

Вторая и третья строки кода — никому не нужно объяснять их. Но там везде повторялась эта мантра «ШаурМаил-ШаурХэш-ШаурХэш». Троекратное повторение слова «Шаур» — ещё один признак излишества. От повторений всем только вред. Вы раздражаете пользователя, заставляя его писать «Шаур» три раза. (Это не настоящее имя компании, а вымышленное.)

Потом они уволили этого парня и наняли того, кто знал, что делает. Вот вторая версия API:

```
class API:
    def __init__(self, key):
        self.header = dict(apikey=key)

    def call(self, method, params):
        request = urllib2.Request(
            self.url + method[0] + '/' + method[1],
            urllib.urlencode(params),
            self.header
        )
        try:
            response = json.loads(urllib2.urlopen(request).read())
            return response
        except urllib2.HTTPError as error:
            return dict(Error=str(error))
```

В той было 660 строк, в этой — 15. Всё, что делает этот код — пользуется методами стандартной библиотеки. Он читается целиком, легко, за секунды, и можно сразу понять, что он делает. Кстати, в нём был ещё набор тестов из 20 строк. Вот как надо писать. Когда они обновляли API, я мог прочесть изменения буквально за пару секунд.

Но и здесь можно заметить проблему. В классе два метода, и один из них — __init__. Авторы этого не скрывали. Второй метод — call, «вызвать». Вот как этим API пользоваться:

```
ShaurMail.API(key='СЕКРЕТНЫЙ КЛЮЧ').call(('mailing', 'statistics'), {'id': 1})
```

Строка длинная, поэтому мы делаем алиас и вызываем его многократно:

```
ShaurMail.request = ShaurMail.API(key='СЕКРЕТНЫЙ КЛЮЧ').call
ShaurMail.request(('mailing', 'statistics'), {'id': 1})
```

Заметьте, мы пользуемся этим классом как функцией. Ею он и должен быть. Если видите подобное, знайте, класс тут не нужен. Поэтому я послал им третью версию API:

```
ShaurMail_API = url = 'https://api.shaurmail.com/%s/%s'
ShaurMail_API_KEY = 'СЕКРЕТНЫЙ КЛЮЧ'

def request(noun, verb, **params):
    headers = {'apikey': ShaurMail_API_KEY}
    request = urllib2.Request(ShaurMail_API % (noun, verb),
                              urllib.urlencode(params), headers)
    return json.loads(urllib2.urlopen(request).read())
```

Он вообще не создаёт файлов в нашем проекте, потому что я вставил его в тот модуль, где он используется. Он делает всё, что делал 15-строковый API, и всё, что делал 660-строковый API.

Вот с чего мы начали и к чему пришли:

- 1 пакет + 20 модулей => 1 модуль
- 20 классов => 1 класс => 0 классов
- 130 методов => 2 метода => 1 функция
- 660 строк кода => 15 строк => 5 строк

Легче пользоваться, легче писать, никому не надо выяснять, что происходит.

Стандартная библиотека

Кто пришёл из языка Java, возможно, считает, что пространства имён нужны для таксономии. Это неверно. Они нужны чтобы предотвратить совпадения имён. Если у вас глубокие иерархии пространств, это никому ничего не даёт. ShaurMail.ShaurHash.ShaurHash — всего лишь лишние слова, которые людям надо помнить и писать.

В стандартной библиотеке Питона пространство имён очень неглубокое, потому что вы либо помните, как называется модуль, либо надо смотреть в документации. Ничего хорошего если надо выяснять цепочку, в каком пакете искать, в каком пакете в нём, в каком пакете дальше, и как называется модуль в нём. Нужно просто знать имя модуля.

К нашему стыду, вот пример из нашего же кода, и те же грехи видно и здесь:

```
services.crawler.crawlerexceptions.ArticleNotFoundException
```

Пакет, в котором модуль из 2 строк, класс исключения и «pass». Чтобы использовать это исключение, надо дважды написать «crawler», дважды слово «exception». Имя ArticleNotFoundException само себя повторяет. Так не надо. Если вы называете исключения, пусть это будет EmptyBeer, BeerError, BeerNotFound, но BeerNotFoundError — это уже много.

Можно просто пользоваться исключениями из стандартной библиотеки. Они понятны всем. Если только вам не нужно выловить какое-то специфическое состояние, LookupError вполне подойдёт. Если вы получили отлуп по почте, всё равно придётся его читать, поэтому неважно, как называется исключение.

Кроме того, исключения в коде обычно идут после raise и except, и всем сразу понятно, что это исключения. Поэтому не нужно добавлять «Exception» в название.

В стандартной библиотеке Питона есть и ржавые детали, но она — очень хороший пример организации кода:

- 200 000 строк кода
- 200 модулей верхнего уровня
- в среднем по 10 файлов в пакете
- 165 исключений

10 файлов в пакете — это много, но только из-за некоторых сторонних проектов, добавленных в библиотеку, где были пакеты из всего 2 файлов. Если вам вздумается создать новое исключение, подумайте лучше, ведь в стандартной библиотеке обошлись 1 исключением на 1200 строк кода.

Я не против классов в принципе. Классы бывают нужны — когда много меняющихся данных и связанных с ними функций. Однако в каждодневной работе такое бывает нечасто. Регулярно приходится работать со стандартной библиотекой, а там уже есть подходящие классы. За вас их уже написали.

Единственное исключение в библиотеке Питона — модуль heapq. Heap queue, «очередь в куче» — это массив, который всегда отсортирован. В модуле heapq десяток методов, и они все работают с той же «кучей». Первый аргумент всегда остаётся тем же, что значит, здесь действительно напрашивается класс.

```
heapify(data)
pushleft(data, item)
popleft(data)
pushright(data, item)
popright(data)
```

и т.д.

Каждый раз, когда нужно пользоваться heapq, я беру реализацию этого класса из своего инструментария.

```
class Heap(object):
    def __init__(self, data=None, key=lambda x: None):
        self.heap = data or []
        heapq.heapify(self.heap)
        self.key = key

    def pushleft(self, item):
        if self.key:
            item = (self.key(item), item)
        heapq.pushleft(self.heap, item)

    def popleft(self):
        return heapq.popleft(self.heap)[1]
```

Классы разрастаются как сорняки

Состояние OAuth в Питоне — неважное. Опять же, есть сторонние библиотеки, и прежде чем использовать в своём проекте, их нужно прочесть.

Я пытался использовать сокращатель урлов от Гугла: мне нужно было взять урлы и просто сократить их. У Гугла есть проект, в котором 10 000 строк кода. 115 модулей и 207 классов. Я написал отповедь об этом в Гугле+, но мало кто её видел, а Гвидо *(Ван Россум — прим. пер.)* прокомментировал: «Я снимаю с себя ответственность за гугловский код API.» 10 000 строк кода — там же обязательно найдётся какая-нибудь дрянь вроде ШаурМэйла. Вот, например, класс Flow («поток»), от которого наследуют другие.

```
class Flow(object):
    pass


class Storage(object):
    def put(self, data): _abstract()
    def get(self): _abstract()

def _abstract(): raise NotImplementedError
```

Он пустой. Но у него есть свой модуль, и каждый раз читая наследующий его класс, надо сходить, проверить тот файл и снова убедиться, что тот класс пуст. Кто-то глядел в будущее и решил: «Напишу-ка я 3 строчки кода сейчас, чтобы в будущем эти 3 строчки не менять.» И отнял время у всех, кто читает его библиотеку. Есть ещё класс Хранилище, (Storage) который почти ничего не делает. В нём правильно обрабатываются ошибки, используя стандартные исключения, но им делают алиасы, и опять же нужно ходить читать их код, чтобы выяснить, как это работает.

Чтобы внедрить OAuth2 мне понадобилась неделя. Пару дней заняло чтение десяти тысяч строк кода, после чего я стал искать другие библиотеки. Нашёл python-oauth2. Это вторая версия python-oauth, но она на самом деле не умеет работать с OAuth2, что не сразу удалось выяснить. Впрочем, эта библиотека немного лучше гугловской: только 540 строк и 15 классов.

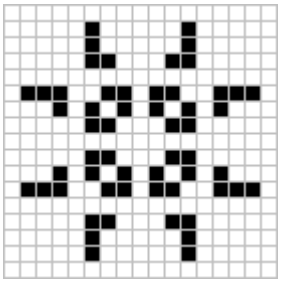
Я переписал её ещё проще и назвал python-foauth2. 135 строк кода и 3 класса, и то всё равно много, я не достаточно её отрефакторил. Вот один из этих трёх классов:

```
class Error(Exception):
    pass
```

Срамота!

Жизнь

Последний пример. Все вы видели игру «Жизнь» Конвэя, даже если не знаете её имени. Есть клетчатое поле, каждый ход вы считаете для каждой клетки соседние, и в зависимости от них она будет либо живой, либо мёртвой. И получаются такие красивые устойчивые узоры, как планер: клетки впереди оживают, а сзади умирают, и планер как будто летит по полю.



Игра «жизнь» очень проста: поле и пара правил. Мы задаём эту задачу на собеседовании, потому что если вы не умеете такого — нам не о чем разговаривать. Многие сразу же говорят «Клетка — существительное. Класс надо.» Какие свойства в этом классе? Место, живая или нет, состояние в следующий ход, всё? Ещё есть соседи. Потом начинают описывать поле. Поле — это множество клеток, поэтому это сетка, у неё метод «подсчитать», который обсчитывает клетки внутри.

```
class Cell(object):
    def __init__(self, x, y, alive=True):
        self.x = x
        self.y = y
        self.alive = alive
        self.next = None

    def neigbors(self):
        for i, j in itertools.product(range(-1, 2), repeat=2):
            if (i, j) != (0, 0):
                yield (self.x + i, self.y + j)


class Board(object):
    def __init__(self):
        self.cells = {} # { (x, y): Cell() }

    def advance(self):
        for (x, y), cell in self.cells.items():
            alive_neighbors = len(cell.neighbors)
            cell.next = (alive_neighbors == 3 or (alive_neighbors == 2 and cell.alive))
```

На этом месте надо сказать «стоп»: у нас есть класс Поле, в котором 2 метода: __init__ и «сделать ход». В нём одно свойство — словарь, значит со словарём и надо работать. Заметьте, что не надо хранить соседей точки, они уже и так есть в словаре. Живая точка или нет — это просто булево значение, поэтому будем хранить координаты только живых клеток. А раз в словаре хранятся только True, нужен не словарь а просто множество (set) координат. Наконец, новое состояние не нужно, можно просто заново создать список живых клеток.

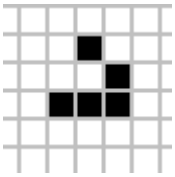
```
def neigbors(point):
    x, y = point
    for i, j in itertools.product(range(-1, 2), repeat=2):
        if any((i, j)):
            yield (x + i, y + j)

def advance(board):
    newstate = set()
    recalc = board | set(itertools.chain(*map(neighbors, board)))
    for point in recalc:
        count = sum((neigh in board)
                for neigh in neighbors(point))
        if count == 3 or (count == 2 and point in board):
            newstate.add(point)
```

```
    return newstate


glider = set([(0, 0), (1, 0), (2, 0), (0, 1), (1, 2)])
for i in range(1000):
    glider = advance(glider)
print glider
```

Получается очень простая, сжатая реализация игры. Двух классов тут не надо. Внизу — координаты планера, их вставляют в поле, и планер летит. Всё.
Это *полная* реализация игры «жизнь».



Резюме

1. Если вы видите класс с двумя методами, включая __init__, это не класс.
2. Не создавайте новых исключений, если они не нужны (а они не нужны).
3. Упрощайте жёстче.

От переводчика: в комментариях я вижу, что многие восприняли доклад как полное отрицание ООП. Это ошибка. Пункт 1 из итогов чётко говорит, что такое не класс. Классы нужны, но суть доклада — в том, что не нужно ими злоупотреблять.

python, ооп, рефакторинг, java, классы, проектирование, игра жизнь