Разработка программного обеспечения на языке Python

Стили программирования. Code rewiew

Стили программирования

Вопрос внутренней организации кода и стилей кодирования -- довольно многогранный и сложный. В крупных компаниях, как правило применяются разные решения для обеспечения стилевого единства и качества кода. Начиная от гайдлайнов по оформлению и автоматическим проверкам / форматированию перед отправкой в репозиторий и заканчивая процедурами просмотра кода (code review) коллегами и механизмами запросов на добавления в репозиторий (pull requests).

Обычно этой теме уделяется не слишком много внимания в классическом обучающем стеке. Для этого есть и объективные причины. Код, который вы пишите в рамках лабораторных работ, домашних заданий отличается от промышленного.

1. Вы, как правило, работаете над ним в одиночку.
2. Код пишется в один момент, очень близко к дедлайну и не версионируется, то есть сдается первая работающая версия.
3. После сдачи код не поддерживается. Его время жизни не превышает полугода.
4. Вы, как правило, пишете код с нуля, не занимаясь поддержкой уже существующего кода.

Классический пример сферы, где стиль программирования и поддержка кода не важны -- спортивное программирование.

Такие условия кардинально отличаются от условий промышленной разработки. Там:

- Как правило, вы поддерживаете, дорабатываете уже существующий код.
- Время жизни проектов довольно велико и может достигать десятков лет.
- В условиях быстрой разработки (новая версия ПО раз в две недели, месяц) очень большой темп внесения изменений и исправлений, реакций на ошибки.
- Разработка кода ведется коллективом разработчиком, пусть и не одномоментно. То есть нет никакого набора “ваших” файлов, которые только вы можете править.

В таких условиях обеспечение стилевого единства и написание хорошего, пригодного к исправлениями и внесениям дополнений кода, становится очень важным.

Вопрос, что такое хорошо организованный код, спорный, но как и в любой инженерной специальности, выработаны некоторые рецепты. Частично мы обратим внимание на некоторые пункты в этой лекции.

Обратим внимание, что данная тема требует большого количества практики, чтобы начать “видеть” места, которые можно улучшить в коде.

Сначала коротко поясним, вопросы стилевого оформления, которые лежат в сфере договоренностей внутри коллектива или компании. Это своеобразный расширенный PEP8. Как правило, рассматриваются вопросы:

- Наименования объектов, файлов, модулей.
- Использования той или иной библиотеки из многих вариантов.
- Вопросы баланса между объектно-ориентированным и функциональным программированием.
- …

На этом этапе вы можете познакомится с гайдлайнами, которые написаны в стиле “Да/нет”. Приведем пример из PEP8. Рассматривается вопрос пробельных отступов в значениях по умолчанию при вызове функций. Рекомендуется их не ставить.

```
# Да:
def complex(real, imag=0.0):
    return magic(r=real, i=imag)

# Нет:
def complex(real, imag = 0.0):
    return magic(r = real, i =  imag)
```

Более сложные вопросы организации кода связаны с паттернами проектирования и со внутренней структурой кода. Ключевой пункт состоит в том, что код должен быть организован однообразно и предсказуемо, а также быть легко модифицируемым.

Иногда внесение небольших изменений может занимать недели, потому что код плохо организован и изменения “ломают” его в непредсказуемых местах. Появился термин рефакторинг -- улучшение существующего кода без внесения изменений в его функционал. То есть код становится потенциально более гибким в возможным изменениям. Желающих более подробного погружения в тему отошлем к классической книге Мартина Фаулера “Рефакторинг”.

Для того, чтобы вы почувствовали приемы работы с кодом, приведем несколько примеров разного уровня сложности.

Запутанный код

Разработчики Питона пошутили. Модуль this, который печатает основополагающие принципы Питона, в том числе говорящие о том, что нужно писать понятно, они сделали очень запутанным.

Вот сами принципы:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

А вот текст модуля:

```
>>> print(open(this.__file__).read())
s = """Gur Mra bs Clguba, ol Gvz Crgref

Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyrk.
Pbzcyrk vf orggre guna pbzcyvpngrq.
Syng vf orggre guna arfgrq.
Fcnefr vf orggre guna qrafr.
Ernqnovyvgl pbhagf.
Fcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.
Nygubhtu cenpgvpnyvgl orngf chevgl.
Reebef fubhyq arire cnff fvyragyl.
Hayrff rkcyvpvgyl fvyraprq.
Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.
Gurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb vg.
Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.
Abj vf orggre guna arire.
Nygubhtu arire vf bsgra orggre guna *evtug* abj.
Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.
Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.
Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!"""

d = {}
for c in (65, 97):
    for i in range(26):
        d[chr(i+c)]  = chr((i+13) % 26 + c)

print("".join([d.get(c, c) for c in s]))
```

Попробуйте понять, как получаются нормальные предложения. Это будет настоящая детективная история. Именно таким бывает код, когда что-то простое делается слишком сложно.

Используйте стандартную библиотеку

Очень часты анти-паттерн -- написание своего кода там, где можно воспользоваться существующим. По сути, изобретение велосипеда. Например, очень много где встречается работа с цифрами, алфавитом и т.д. и делается что-то типа:

```
dig  = ‘0123456789’
```

Делать этого не нужно, потому что есть модуль `strings` . Оцените, насколько он удобен.

```
>>>  import string
>>>
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
>>> string.octdigits
'01234567'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\)^_`{|}~'
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Императивщина

Вы используете императивный стиль программирования, там где можно воспользоваться готовой функциональность и писать более декларативно.

```
Нет:
>>> s = 0
>>> for i in range(1, 101):
    s += i

>>> s
5050

Да:
>>> sum(range(1, 101))
5050
```

Неверно выбранный интерфейс

Возьмем самый простой случай. У нас есть класс Reporter, который генерирует для нас новости. Как раз тот случай, когда работает в рамках домашних заданий все работает хорошо, а если пытаться встраивать свой код в какую-нибудь систему, то ее надо переписывать.

```
>>> class Reporter:
    def get_news(self):
        print("Вот  это новость!!! Рубль подорожал")


>>> r = Reporter()
>>> r.get_news()
Вот  это новость!!! Рубль подорожал
```

Результат работы такого класса я не смогу залогировать, показать на сайте, поместить в БД. Ведь репортер просто печатает результат.

Мне либо надо переписывать класс. Либо... А давайте сделаем убьем один антипаттерн другим! Используем Monkey patching.

Дальше аккуратней. Мы подменим `stdout`...

```
>>>  import sys
>>>  import io
>>>
>>>  old_stdout = sys.stdout >
>>>  s = io.StringIO("") # Строковый объект, совместимый по интерфейсам с  файловым буфером
>>>  sys.stdout = s # Заменяем стандартный поток вывода этим объектом.
>>>
>>>  r = Reporter()
>>>  r.get_news() # Печать была в наш строковый объект.
>>>
>>>  sys.stdout = old_stdout # Восстанавливаем старый-добрый stdout
>>>  s.seek(0) # Перемотали вначало.
0
>>>  res = s.read() # Прочли все.
>>>  res
'Вот  это новость!!! Рубль подорожал\n'
```

Выделение функций и модулей

Рассмотрим на примере веб-фреймворка Flask и модели MVC. Предположим, у нас есть сайт, на главной странице которого школьник может ввести свои оценки за четверть и узнать, сколько нужно получить пятерок, чтобы в итоге получить “отлично”. Функция index выступает в роли точки входа, собирая параметры из веб-страницы и возвращает результат. По сути, она -- веб-обертка для “обычной функции”. Она является контроллером.

Иногда даже Википедия сообщает полезные сведения:

Начинающие программисты очень часто трактуют архитектурную модель MVC как пассивную модель MVC: модель выступает исключительно совокупностью функций для доступа к данным, а контроллер содержит бизнес-логику. В результате — код моделей по факту является средством получения данных из СУБД, а контроллер — типичным модулем, наполненным бизнес-логикой. В результате такого понимания — MVC-разработчики стали писать код, который Pádraic Brady (известный в кругах сообщества «Zend Framework») охарактеризовал как «ТТУК» («Толстые, тупые, уродливые контроллеры»; Fat Stupid Ugly Controllers)

На самом деле, вопрос о том, делать ли контроллеры толстыми, дискуссионен.

Функции Фласка, к которым применяется роутинг, по факту являются контроллерами.

Их основное предназначение – разобрать параметры, первично проверить их на валидность и решить, что нужно делать, вызвав соответствующие методы у объектов, которые отвечают за модель.

Здесь философия чем-то похожа на правила построения функции main в Си, которая по сути является контроллером программы, принимая параметры вызова и понимающая, что нужно делать. Например, сошлюсь на статью How to write a good C main function.

The purpose of the main() function is to collect the arguments that the user provides, perform minimal input validation, and then pass the collected arguments to functions that will use them.

Если вы некоторое ядро вынесете в отдельные функции и классы, а контроллер будет только их вызывать, вам легко будет менять интерфейс программы, цепляя веб-интерфейс, бота в телеграмме или делая консольную утилиту.

Вызывает вопросы:

```
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        grs =  request.form['MarksInput']
        grs =  [int(i) for i in re.findall('\d', grs)]
        m = mean(grs)
        for i  in count():
            if  mean(grs) >= 4.5:
                break;
            grs.append(5)

        return  render_template('index.html', mean=m, count=i)

    return  render_template('index.html')
```

Не вызывает вопросы:

```
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        grs =  request.form['MarksInput']
        grs =  [int(i) for i in re.findall('\d', grs)]
        cnt =  get_count(grs)
        return  render_template('index.html', mean=m, count=cnt)

    return  render_template('index.html')

def get_count(grs):
    m = mean(grs)
    for i in count():
        if  mean(grs) >= 4.5:
            return  i
        grs.append(5)
```

Совсем-совсем не вызывает вопросы:

```
from grades.utils import get_count

 @app.route('/', methods=['GET', 'POST'])
 def index():
     if request.method == 'POST':
         grs =  request.form['MarksInput']
         grs =  [int(i) for i in re.findall('\d', grs)]
         cnt =  get_count(grs)
         return  render_template('index.html', mean=m, count=cnt)
    return render_template('index.html')
```

Да, в этом примере бизнес-логика состоит из нескольких строчек, и полезность таких манипуляций неочевидна. Однако, если количество кода станет увеличиваться, эта проблема даст о себе знать.

Споткнуться на ровном месте

Иногда даже казалось бы, простая функциональность таит сюрпризы. В Питоне есть функция округления. Но она работает не по “школьным” правилам.

```
>>>  for i in range(1, 10):
     k = i + 0.5
     print(k, round(k))


 1.5  2
 2.5  2
 3.5  4
 4.5  4
 5.5  6
 6.5  6
 7.5  8
 8.5  8
 9.5  10
```

По факту, round из всех типов округления (а их много, все крутится вокруг того, что делать с цифрой , выбирает Round half to even.

If the discarded digits represent greater than half ( ) the value of a one in the next left position then the result coefficient should be incremented by (rounded up). If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored).

Otherwise (they represent exactly half) the result coefficient is unaltered if its rightmost digit is even, or incremented by (rounded up) if its rightmost digit is odd (to make an even digit).

То есть, если цифра сразу после цифры округления , то округляем до ближайшего четного.

Это не Питон такой странный, эта модель округления по умолчанию в IEEE . Еще его называют банковским округлением, оно препятствует накоплению ошибки при работе с несколькими округляемыми величинами.

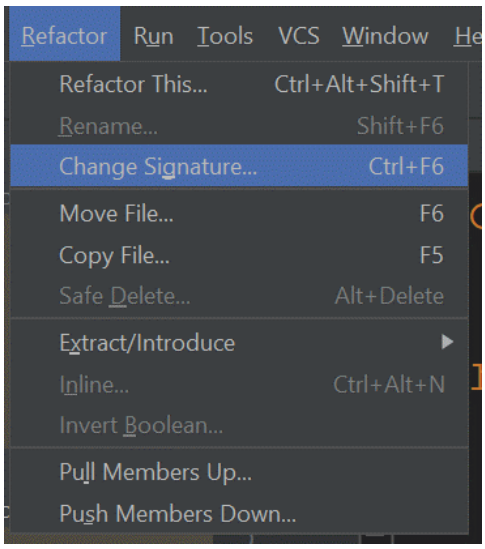
Сделать привычное нам по школе округление (round half up):

```
>>> import decimal
>>>
>>> context = decimal.getcontext()
>>> context.rounding = 'ROUND_HALF_UP'
>>>
>>> for i in range(1, 10):
    k = i + 0.5
    print(k, round(Decimal(k), 0))


1.5 2
2.5 3
3.5 4
4.5 5
5.5 6
6.5 7
7.5 8
8.5 9
9.5 10
```

Пользуйтесь возможностями среды разработки

В современные среды разработки уже включены различные приемы работы с рефакторингом. Попробуйте воспользоваться некоторыми из них.



Как мы уже говорили, темы стилевого оформления кода и рефакторинга -- очень объемная. Иногда даже подверженная влиянию моды, например, включению элементов функционального программирования. Самый лучший способ -- влиться в какой-то проект с большой историей и огромным количеством кода. Благодаря тому, что есть github и огромное количество open source проектов, у вас есть замечательная возможность сделать это.

Рефакторинг кода

ПРЕДЫДУЩИЙ ЭЛЕМЕНТ КУРСА

◄ Задание. Создание собственной библиотеки

Перейти на...

СЛЕДУЮЩИЙ ЭЛЕМЕНТ КУРСА

Вопросы к лекции Стили программирования. Code rewiew ►