Сообщить об ошибке.

Рефа кторинг кода в Python

Рефакторинг на примере одной функции

> ***Википедия гласит:***
> *Рефа кторинг (англ. refactoring), или переработка кода, равносильное преобразование алгоритмов - процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований, которая в последствии может привести к существенной перестройке программы и улучшению её согласованности и чёткости.*

Из определения видно, что рефакторинг кода - это попытка сделать код программы лучше. Улучшение кода может означать разные вещи, в зависимости от контекста:

- код легче поддерживать;
- код легче объяснить новичкам;
- также может означать, что код станет быстрее.

Каждый может извлечь пользу из обучения рефакторингу, так как происходит тренировка ряда навыков, например, способность читать код и действительно понимать его, распознавания образов, критическое мышление и т.д.

> *Одно из главных условий рефакторинга фрагмента кода, это необходимость понимать, что код делает и как он это делает. Если изменить часть кода, не понимая его, то шансы сломать программу возрастают.*

Одна из вещей, на которые необходимо обратить внимание при рефакторинге - это избыточность и повторяемость. Такие вещи очень просто заметить, т.к. в коде будут идентичные строки кода (повторения). Найти структурные сходства между различными частями кода сложнее, чем найти идентичные строки. Но после того как код становится более понятным, сделать это становится намного проще (тренировка навыков распознавания образов).

Читая свой, ранее написанный код, вы несомненно найдете фрагменты, которые находятся, например в не том файле (если это пакет Python). Кусок кода, который находится в неправильной функции или даже кусок кода, который выглядит так, как будто его можно и нужно удалить (критическое мышление).

Рефакторинг также можно охарактеризовать как бесконечный цикл. По мере приобретения опыта, разработчик узнает что-то новое, кроме того, технологии, которые использовались при написании кода год назад, вероятно, претерпели изменения. Это означает, что ранее написанный код естественным образом переходит в состояние необходимости рефакторинга.

Содержание:

- Ненужные вспомогательные переменные;
- Понимание избыточного списка;
- Что еще можно учитывать при рефа кторинге.

Задача на разработку функции Python.

Задача: напишите функцию, которая меняет регистр букв передаваемого ей слова:

- буквы в четных позициях должны стать прописными,
- буквы в нечетных позициях должны стать строчными.

Фрагмент кода, который сразу приходит на ум:

```
def textfunc(a):
    empty=[]
    for i in range(len(a)):
        if i%2==0:
            empty.append(a[i].upper())
        else:
            empty.append(a[i].lower())

    return "".join(empty)
```

Автоматическое форматирование кода.

Самый первый шаг, который можно сделать для написания элегантного кода - это запуск инструмента для проверки кода Python на соответствие некоторым соглашениям о стилях в PEP 8 например pycodestyle (дает рекомендации по исправлению). Также есть инструменты автоматического исправления кода. Например, если использовать сторонний модуль black, то с самого начала можно исправить многие несоответствия и проблемы со стилем.

```
def textfunc(a):
    empty = []
    for i in range(len(a)):
        if i % 2 == 0:
            empty.append(a[i].upper())
        else:
            empty.append(a[i].lower())

    return "".join(empty)
```

Разница этого кода заключается в пробелах в `empty = []` и `if i % 2 == 0:`. Если код следует руководству PEP 8 Python, то он будет более читаемым для себя и для других. Также необходимо знать, что некоторые команды разработчиков частично игнорируют руководство по стилю PEP 8, т. к. у них разработаны собственные стандарты по оформлению кода (в основном это касается правил написания имен функций, классов и их методов, а также количества символов для переноса строки с кодом).

Каткая справка по модулю black.

Модуль black- форматировщик кода Python, который избавляет от мелочей ручного форматирования. Он дает скорость, детерминизм и свободу от pycodestyle, который очень придирчив к форматированию. Модуль black ускоряет проверку кода, производя наименьшие возможные различия.

Модуль black можно установить, запустив `pip install black`. Для запуска требуется Python 3.7+. Если необходимо отформатировать ноутбуки Jupyter, то установите его с помощью `pip install 'black[jupyter]'`.

Чтобы запустить форматирование кода с разумными настройками по умолчанию, нужно выполнить команду:

```
$ black {source_file_or_directory}
```

Если запуск black как скрипта не работает, можно запустить его как модуль Python:

```
$ python3 -m black {source_file_or_directory}
```

Все настройки/параметры командной строки модуля black можно отобразить, выполнив команду терминала `$ black --help`.

Имена переменных, функций, классов и т.д. в Python.

Имена очень важны, и правильное наименование функций имеет решающее значение. Хорошие имена сразу дают понять, что делает код функции класса или метода, а плохие имена заставляют часами анализировать простой код.

Имена должны отражать назначение или очень важное свойство того, к чему они относятся. Это противоположно использованию очень общих имен, таких как textfunc для функции или num для числа.

Заметным исключением является, например, использование i в [циклах for/in](), хотя предпочтительнее использовать немного более подробное имя, например idx.

Глядя на код, который есть в настоящее время, можно выделить три имени, которые можно было бы улучшить.

```
def text_decoration(text):
    letters = []
    for idx in range(len(text)):
        if idx % 2 == 0:
            letters.append(text[idx].upper())
        else:
            letters.append(text[idx].lower())

    return "".join(letters)
```

Изменения, которые сделаны:

- название функции myfunc переименовываем в text_decoration;
- список empty переименовываем в letters
- переменную a в text;
- переменную цикла i в idx (необязательно).

По началу, имя для списка empty = [] кажется довольно неплохим. Но после того, как инициализируется пустой список, программа начинает его заполнять, поэтому имя не отражает свойство объекта, которое сохраняется на протяжении всей программы.

Перемещение данных и индексов.

В Python есть несколько действительно хороших возможностей для работы с [циклами for](), и один из инструментов, который предоставляется - это [встроенная функция enumerate](). Это инструмент, к которому нужно всегда обращаться, когда цикл for работает с индексами и данными одновременно.

В функции text_decoration() как раз нужны индексы и данные, так как нужен индекс для определения выполняемой операции, а затем нужны данные (фактическая буква), чтобы изменить ее регистр.

```
def text_decoration(text):
    letters = []
    for idx, char in enumerate(text):
        if idx % 2 == 0:
            letters.append(char.upper())
        else:
            letters.append(char.lower())

    return "".join(letters)
```

В измененном коде функции убрали [явное индексирование]() (text[idx]), сократив тем самым одну операцию, а также более четко выражен замысел: применение enumerate в цикле всегда означает, что "*в этом цикле нужны как индексы, так и данные*".

Не повторяйся. Вкладываем только то, что необходимо.

В Python отступ указывает на вложенность кода, что указывает на зависимость. Если строка кода вложена в цикл for, то это означает, что она зависит от [цикла for](). Если он дополнительно вложен в [оператор if/else](), это означает, что он применяется только при выполнении определенных условий. Если он дополнительно вложен в [оператор try/except](), то можно ожидать, что он вызовет ошибку и т. д.

Вложенность кода означает, что нужно отслеживать множество контекстов в голове. Чтобы проще было следить за контекстом, нужно стараться как можно меньше писать вложенный код. Необходимо вкладывать только те фрагменты кода, которые абсолютно необходимы для этого. Для циклов for это обычно вещи, которые зависят от переменных итератора между for и in, а для операторов if/else это фрагменты кода, которые уникальны для каждого оператора.

Тело функции text_decoration() имеет такой код:

```
if idx % 2 == 0:
    letters.append(char.upper())
else:
    letters.append(char.lower())
```

Обратите внимание, что код вызывает letter.append() независимо от ветки, в которой находится. Это делает менее очевидным, то, что меняется от одной ветки к другой, так как методы str.upper() и str.lower() занимают одинаковое количество символов, и следовательно выровнены.

Становиться очевидным необходимость вынести операцию добавления символов letter.append() из веток условия if/else. В итоге, результирующий код функции меняется на следующий:

```
def text_decoration(text):
    letters = []
    for idx, char in enumerate(text):
        if idx % 2 == 0:
            capitalised = char.upper()
        else:
            capitalised = char.lower()
        letters.append(capitalised)

    return "".join(letters)
```

Можно конечно много рассуждать на тему, что код стал длиннее, а не короче. Но принимая во внимание один из пунктов философии языка Python ("*Explicit is better than implicit*") иногда более качественный код занимает больше места.

Использование if/else в одну строку.

Вынесение letters.append() за пределы if/else делает совершенно очевидным, что оператор if предназначен только для того, чтобы решить, когда нужно использовать строчные или заглавные буквы. Это открывает дверь для еще одного упрощения, которое придет в виде условного выражения в одну строку. Используя однострочник if/else, перепишем условие как:

```
capitalised = char.upper() if idx % 2 == 0 else char.lower()
```

По большому счету, промежуточная переменная capitalised не нужна, так как тело цикла можно записать как:

```
def text_decoration(text):
    letters = []
    for idx, char in enumerate(text):
        letters.append(char.upper() if idx % 2 == 0 else char.lower())

    return "".join(letters)
```

Использование истинности объекта Python.

Следующий шаг касается упрощения условия оператора if. В Python есть замечательная вещь, которая позволяет интерпретировать многие объекты как булевы значения, даже если они сами не являются логическими значениями. Это часто называют истинность объекта в Python.

В случае с условием if idx % 2 == 0:... важно то, что число 0 рассматривается как False, а любое другое целое число рассматривается как True. Следовательно, условие можно записать как if idx % 2:.... Теперь, если индекс idx - четный, то результат idx % 2 будет РАВЕН 0 (значит False), следовательно букву необходимо делать заглавной. Принимая во внимание это утверждение условие выражение в одну строку примет вид:

```
char.lower() if idx % 2 else char.upper()
```

Перепишем функцию text_decoration() используя истинность объектов в Python:

```
def text_decoration(text):
    letters = []
    for idx, char in enumerate(text):
        letters.append(char.lower() if idx % 2 else char.upper())
    return "".join(letters)
```

Использование списка как выражения.

Еще одну вещь, которую можно научиться замечать - это когда строится список, последовательно вызывая для него list.append(). В этом случае, необходимо искать возможность использовать для этого выражение-генератор списка. При правильном использовании, генератор списка позволяет инициализировать переменную списка letters с правильным содержимым с самого начала, а не инициализировать переменную, чтобы сразу изменить ее.

Используя выражение-генератор списка, можно переписать цикл следующим образом:

```
def text_decoration(text):
    letters = [char.lower() if idx % 2 else char.upper() for idx, char in enumerate(text)]
    return "".join(letters)
```

Длинные строки кода.

Проблема со списком выше заключается в том, что теперь существует очень длинная строка кода. По возможности следует избегать длинных строк, т.к. они затрудняют чтение и работу с кодом, когда он находится рядом, например, с отладчиком и т.д. Горизонтальной прокрутки в коде следует избегать любой ценой.

Есть несколько способов, что бы исправить длинный код выражения-генератора списка.

Первый основывается на том, что имена переменных используемые генератором списка живут только внутри него и играют очень специфическую роль. Из-за этого, если структура происходящего достаточно ясна, то можно использовать более короткие имена переменных внутри генератора списка:

```
def text_decoration(text):
    letters = [c.lower() if i % 2 else c.upper() for i, c in enumerate(text)]
    return "".join(letters)
```

Обратите внимание, что целевая переменная имеет правильное имя (letters), как и переменная, по которой итерируемся (text).

Второй способ основывается на синтаксисе Python: то, что внутри любых скобок - можно переносить. Так что, если предпочтительнее оставить длинные имена, то генератор списка можно записать следующим образом:

```
def text_decoration(text):
    letters = [
        char.lower() if idx % 2 else char.upper()
        for idx, char in enumerate(text)
    ]
    return "".join(letters)
```

Ненужные вспомогательные переменные.

Вспомогательные переменные не всегда нужны. В данном конкретном случае можно просто избавиться от вспомогательной переменной letters и напрямую вызвать метод строки str.join():

```
def text_decoration(text):
    return "".join([c.lower() if i % 2 else c.upper() for i, c in enumerate(text)])
```

или

```
def text_decoration(text):
    return "".join([
        char.lower() if idx % 2 else char.upper()
        for idx, char in enumerate(text)
    ])
```

Понимание избыточного списка.

Есть еще одна последняя вещь, которую можно сделать, и она связана с тем, как можно избавиться от скобок [] в генераторе списка. Можно буквально удалить их, так что в итоге получим следующее:

```
def text_decoration(text):
    return "".join(c.lower() if i % 2 else c.upper() for i, c in enumerate(text))
```

или

```
def text_decoration(text):
    return "".join(
        char.lower() if idx % 2 else char.upper()
        for idx, char in enumerate(text)
    )
```

Что еще можно учесть при рефа кторинге.

1. Использование f-строк вместо форматирования строк str.format() там где это возможно (f-строки не могут использоваться как шаблоны строк).
2. Использовать простую и расширенную распаковку и упаковку переменных.
3. Использование выражение присваивания := в условиях if/else и циклах с условием while.