Седьмое практическое занятие

Композиция и итераторы

Результатом прошлого занятия стал класс `Led`, позволяющий управлять светодиодом:

```
from time import ticks_ms
from machine import Pin, PWM

class Led:
    BRIGHTNESS_SCALE = 100

    def __init__(self, led_pin):
        self.led = PWM(Pin(led_pin, Pin.OUT), freq=1000)
        self.timer = ticks_ms()
        self.state = "OFF"
        self.brightness = 0

    def on(self):
        self.led.duty_u16(65535)
        self.timer = ticks_ms()
        self.state = "ON"

    def off(self):
        self.led.duty_u16(0)
        self.timer = ticks_ms()
        self.state = "OFF"

    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.BRIGHTNESS_SCALE:
            brightness = self.BRIGHTNESS_SCALE
        duty = int(65535 / self.BRIGHTNESS_SCALE * brightness)
        self.led.duty_u16(duty)
        self.timer = ticks_ms()
        self.brightness = brightness

    def smooth_switching_on(self, switching_time=1):
        sleep_time = switching_time / self.BRIGHTNESS_SCALE
        if self.state == "STOP":
            self.state = "UP"
        if self.brightness == self.BRIGHTNESS_SCALE:
            self.state = "STOP"
        if (ticks_ms() - self.timer) / 1000 >= sleep_time and self.state
== "UP":
```

```
            self.brightness += 1
            self.turn_to_brightness(self.brightness)

    def smooth_switching_off(self, switching_time=1):
        sleep_time = switching_time / self.BRIGHTNESS_SCALE
        if self.state == "STOP":
            self.state = "DOWN"
        if self.brightness == 0:
            self.state = "STOP"
        if (ticks_ms() - self.timer) / 1000 >= sleep_time and self.state
== "DOWN":
            self.brightness -= 1
            self.turn_to_brightness(self.brightness)

    def blink(self, blink_time_in_sec):
        if (ticks_ms() - self.timer) / 1000 >= blink_time_in_sec:
            if self.state != "OFF":
                self.off()
            else:
                self.on()
```

Хороший ли это получился класс?

Ответ будет зависеть от того, кто будет отвечать…

Так, разработчик, потративший свое время и силы на создание, развитие и написание кода, будет готов до хрипоты защищать каждый из разработанных и внедренных в продакшн методов. Пользователь, получивший требуемую ему функциональность, тоже не станет сильно возражать - ведь он получил простой и удобный интерфейс для работы со своей лампочкой.

Так в чем проблема? Где, когда, как и почему она себя проявит?

Ответим на эти сократические вопросы!

- Ответ на «где?» прост - он в нашем классе `Led`, ведь больше мы пока ничего не написали! (Хотя более прозорливый читатель справедливо локализует источник проблемы в самом разработчике, в чем он, без сомнения, полностью прав);
- Ответ на «когда?» менее определен и лежит в пределах от «уже, но мы пока в благом неведении» до «в ближайшее время»;
- «Как?» - все начинает ломаться! Довольный пока пользователь всегда найдет, чем нас удивить! Он не пожалеет сил и времени и рано или поздно (скорее рано) подберет такую комбинацию вызовов наших методов и атрибутов, что все перестанет нормально работать, и он в гневе начнет требовать от нас исправления. Это неизбежный процесс. Если быть полностью откровенным, то все наше искусство разработки просто должно обеспечить нам достаточно

времени на завершение работы над проектом + время на поиск и отступление в новый проект…

- «Почему?» односложный ответ - потому. Мы нарушили дзен питона. Создали переусложненный в своем поведении и реализации «божественный» объект, сочетающий в себе как необходимое поведение, так и избыточное. В результате он стал слишком перегружен и сложен!

Ведем в интерпретаторе:

```
import this
```

и еще раз прочитаем: «Простое лучше, чем сложное» и будем в дальнейшем к этому стремиться!

Что же, избавимся без сожалений от лишнего и оставим только то, без чего диод потеряет свое естественное, базовое предназначение: нести свет и, может быть, нести его с контролируемой яркостью.

В итоге получим:

```
from machine import PWM, Pin
from time import ticks_ms


class Led:
    BRIGHTNESS_SCALE = 255

    def __init__(self, led_pin):
        self.led = PWM(Pin(led_pin, Pin.OUT), freq=1000)
        self.brightness = 0

    def on(self):
        self.led.duty_u16(65535)
        self.brightness = self.BRIGHTNESS_SCALE

    def off(self):
        self.led.duty_u16(0)
        self.brightness = 0

    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.BRIGHTNESS_SCALE:
            brightness = self.BRIGHTNESS_SCALE
        duty = int(65535 / self.BRIGHTNESS_SCALE * brightness)
        self.led.duty_u16(duty)
        self.brightness = brightness
```

Пусть мы пока и потеряли на время сложную функциональность, но взамен получили обратно простоту и стабильность. Помни, дорогой друг, не начинай делать новое, пока не уверен до конца в уже сделанном!

Разработка класса RGBLed

Не будем далее удручаться старыми ошибками! У нас не так много времени, чтобы создать новые!

Реализуем класс для RGB светодиода, позволяющий работать с ним в ООП стиле.

Для начала придумаем для создаваемого класса броское, сильное и понятное имя:

```
class RGBLed:
    ...
```

Но чем его наполнить?

Наивный вариант начать писать код по аналогии с ранее разработанным классом `Led`:

```
from machine import PWM, Pin

class RGBLed:

    def __init__(self, red_pin, green_pin, blue_pin):
        self.red = PWM(Pin(red_pin, Pin.OUT), freq=1000)
        self.green = PWM(Pin(green_pin, Pin.OUT), freq=1000)
        self.blue = PWM(Pin(blue_pin, Pin.OUT), freq=1000)

    @staticmethod
    def get_duty_for_color(color_component):
        if color_component <= 0:
            color_component = 0
        elif color_component > 255:
            color_component = 255
        duty = int(65535 / 255 * color_component)
        return duty

    def turn_on_color(self, color_tuple):
        self.red.duty_u16(self.get_duty_for_color(color_tuple[0]))
        self.green.duty_u16(self.get_duty_for_color(color_tuple[1]))
        self.blue.duty_u16(self.get_duty_for_color(color_tuple[2]))

    def turn_on(self):
        self.turn_on_color([255, 255, 255])
```

```
    def turn_off(self):
        self.turn_on_color([0, 0, 0])
```

Можем проверить этот код:

```
from time import sleep

rgb_led_1 = RGBLed(red_pin=15, green_pin=14, blue_pin=13)
rgb_led_2 = RGBLed(red_pin=16, green_pin=17, blue_pin=18)

colors = [[255, 0, 0],
          [255, 255, 0],
          [0, 255, 0],
          [0, 255, 255],
          [0, 0, 255],
          [255, 0, 255]]

while True:
    for i in range(len(colors) - 1):
        rgb_led_1.turn_on_color(colors[i])
        rgb_led_2.turn_on_color(colors[i + 1])
        sleep(1)
```

Но наш опыт, переходящий, в силу второго закона диалектики, неизбежно в качество, заставляет нас подвергнуть сгенерированный текст программы остракизму!

В чем проблема?

- Во-первых, мы неявно нарушили принцип DRY - Don’t repeat yourself. Мы начали повторять уже сделанное.
- Во-вторых, мы вынудили себя опять работать с диодом на уровне сигналов микроконтроллера - реализуя логику работы на уровне ШИМ контроля выводящих пинов. В результате мы опять начинаем переусложнять объект!
- В заключение, мы фактически проигнорировали уже разработанный п протестированный код, обесценив тем самым выполненную ранее работу!

Но как решить эту задачу по-другому?

В ООП существует два основных подхода к реализации нового класса на основе существующего - это наследование и композиция.

> **Наследование** - это перенос всего поведения класса предка в класс потомка.

На первый это может показаться тем, что нужно! Ведь RGB светодиод является светодиодом, для которого так же естественно поведение включаться, выключаться и тд.

Но в подобных задачах целесообразнее отталкиваться от отличий:

- RGB светодиод по-другому подключается;
- Он должен обладать новой функциональностью (изменение цвета), которой нет в обычном светодиоде.

В конце концов, RGB светодиод - это просто комбинация состоящая из красного, зеленого и синего светодиодов, заключенных в один корпус, и, вполне логично перенести такую природу в разрабатываемый код.

Какой там был второй подход кроме наследования? Композиция?

> **Композиция** — это один из фундаментальных принципов объектно-ориентированного программирования (ООП), который позволяет создавать сложные объекты путем объединения более простых объектов.

В отличие от **наследования**, где объекты наследуют свойства и методы от родительских классов, в композиции объекты создаются как совокупность других объектов, каждый из которых отвечает за определенную часть функциональности.

То, что нужно!

Представим, что `RGBLed` просто состоит из трех экземпляров `Led`:

```
from machine import PWM, Pin

class Led:
    BRIGHTNESS_SCALE = 255

    def __init__(self, led_pin):
        self.led = PWM(Pin(led_pin, Pin.OUT), freq=1000)
        self.brightness = 0

    def on(self):
        self.led.duty_u16(65535)
        self.timer = ticks_ms()

    def off(self):
        self.led.duty_u16(0)
        self.timer = ticks_ms()

    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.BRIGHTNESS_SCALE:
            brightness = self.BRIGHTNESS_SCALE
        duty = int(65535 / self.BRIGHTNESS_SCALE * brightness)
        self.led.duty_u16(duty)
```

```
        self.brightness = brightness


class RGBLed:

    def __init__(self, red_pin, green_pin, blue_pin):
        self.red = Led(red_pin)
        self.green = Led(green_pin)
        self.blue = Led(blue_pin)

    def turn_on_color(self, color_tuple):
        self.red.turn_to_brightness(color_tuple[0])
        self.green.turn_to_brightness(color_tuple[1])
        self.blue.turn_to_brightness(color_tuple[2])

    def turn_on(self):
        self.turn_on_color([255, 255, 255])

    def turn_off(self):
        self.turn_on_color([0, 0, 0])
```

Красивое…

Что мы получили в результате?

Мы получили возможность работать с RGB светодиодом как с комбинацией обычных диодов класса `Led`. Тем самым все тонкости работы с микроконтроллером нам перестали быть интересны. Мы просто передаем команды на установку определенной яркости отдельных светодиодов, тонкости работы с которыми инкапсулированы «под капотом» класса `Led`!

Получившийся класс `RGBLed` - хорош своей простотой! Но, что делать если мы захотим реализовать логику цветосветовых эффектов?

Давайте попробуем это сделать!

Наивный способ реализации эффектов в классе `RGBLed`

Не зацикливаясь на мелочах рассмотрим следующий код:

```
from machine import PWM, Pin
from time import ticks_ms


class Led:
    BRIGHTNESS_SCALE = 255

    def __init__(self, led_pin):
```

```
        self.led = PWM(Pin(led_pin, Pin.OUT), freq=1000)
        self.brightness = 0

    def on(self):
        self.led.duty_u16(65535)

    def off(self):
        self.led.duty_u16(0)

    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.BRIGHTNESS_SCALE:
            brightness = self.BRIGHTNESS_SCALE
        duty = int(65535 / self.BRIGHTNESS_SCALE * brightness)
        self.led.duty_u16(duty)
        self.brightness = brightness


class RGBLed:

    def __init__(self, red_pin, green_pin, blue_pin):
        self.red = Led(red_pin)
        self.green = Led(green_pin)
        self.blue = Led(blue_pin)
        self.timer = ticks_ms()
        self.degree = 0

    def turn_on_color(self, color_tuple):
        self.red.turn_to_brightness(color_tuple[0])
        self.green.turn_to_brightness(color_tuple[1])
        self.blue.turn_to_brightness(color_tuple[2])

    @staticmethod
    def rgb_wheel(degree):
        def color_pattern(degree):
            degree %= 360
            if 0 <= degree < 60:
                value = degree / 60
            elif 180 <= degree < 240:
                value = 1 - (degree - 180) / 60
            else:
                value = 1 if 60 <= degree < 180 else 0
            return int(value * 255)

        red = color_pattern(degree - 240)
        green = color_pattern(degree)
        blue = color_pattern(degree - 120)
        return red, green, blue
```

```
    def rainbow(self, circle_time=1):
        self.degree = self.degree % 360
        delta_time = circle_time * 1000 / 360
        current_time = ticks_ms()
        if current_time - self.timer > delta_time:
            self.timer = current_time
            colour_tuple = self.rgb_wheel(self.degree)
            self.degree += 1
            self.turn_on_color(colour_tuple)

    def turn_on(self):
        self.turn_on_color([255, 255, 255])

    def turn_off(self):
        self.turn_on_color([0, 0, 0])
```

Что изменилось?

В атрибуты объекта мы добавили два новых поля:

```
self.timer = ticks_ms()
self.degree = 0
```

Необходимых для контроля времени последнего изменения цвета и текущего «градуса» в цветовом круге `rgb_wheel(degree)`.

В методе:

```
def rainbow(self, circle_time=1):
    self.degree = self.degree % 360
    delta_time = circle_time * 1000 / 360
    current_time = ticks_ms()
    if current_time - self.timer > delta_time:
        self.timer = current_time
        colour_tuple = self.rgb_wheel(self.degree)
        self.degree += 1
        self.turn_on_color(colour_tuple)
```

При каждом вызове метода рассчитывается необходимый период между циклами изменения цвета:

```
delta_time = circle_time * 1000 / 360
```

И в случае превышения требуемого промежутка времени от ранее сохраненного времени `self.timer` выполняется рассвет нового оттенка, «включаемого» на диоде методом `self.turn_on_color(colour_tuple)`.

Проверим работу нашего кода:

```
rgb_led_1 = RGBLed(red_pin=15, green_pin=14, blue_pin=13)
rgb_led_2 = RGBLed(red_pin=16, green_pin=17, blue_pin=18)

while True:
    rgb_led_1.rainbow(circle_time=1)
    rgb_led_2.rainbow(circle_time=3)
```

Все работает!

Но мы опять повторили допущенную нами ранее ошибку - переусложнили объект, добавив в него не свойственные его природе атрибуты и методы!

С чего это вдруг в диоде появились какие-то «радуги» и «цветовые колеса»? Зачем диоду «градусы» и «таймеры» ведь он должен только светить! Все остальное уже лишнее!

А если мы захотим продолжать увеличивать количество эффектов? Делать их сложнее? Вместе с этим мы будем все дальше уходить от природы и смысла программируемого объекта!

Попробуем это исправить!

Реализация эффектов RGB светодиода через итераторы

Перепишем наш класс `RGBLed` следующим образом:

```
from machine import PWM, Pin
from time import ticks_ms


class Led:
    BRIGHTNESS_SCALE = 255

    def __init__(self, led_pin):
        self.led = PWM(Pin(led_pin, Pin.OUT), freq=1000)
        self.brightness = 0

    def on(self):
        self.led.duty_u16(65535)
        self.timer = ticks_ms()

    def off(self):
        self.led.duty_u16(0)
        self.timer = ticks_ms()
```

```
    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.BRIGHTNESS_SCALE:
            brightness = self.BRIGHTNESS_SCALE
        duty = int(65535 / self.BRIGHTNESS_SCALE * brightness)
        self.led.duty_u16(duty)
        self.brightness = brightness


class RGBLed:

    def __init__(self, red_pin, green_pin, blue_pin):
        self.red = Led(red_pin)
        self.green = Led(green_pin)
        self.blue = Led(blue_pin)
        self.timer = ticks_ms()

    def turn_on_color(self, color_tuple):
        self.red.turn_to_brightness(color_tuple[0])
        self.green.turn_to_brightness(color_tuple[1])
        self.blue.turn_to_brightness(color_tuple[2])

    def color_effects(self, color_effect_iter, circle_time=1):
        delta_time = circle_time * 1000 /
color_effect_iter.count_of_iter_cycle
        current_time = ticks_ms()
        if current_time - self.timer > delta_time:
            self.timer = current_time
            colour_tuple = next(color_effect_iter)
            self.turn_on_color(colour_tuple)

    def on(self):
        self.turn_on_color([255, 255, 255])

    def off(self):
        self.turn_on_color([0, 0, 0])
```

Что осталось? А что изменилось?

Остался таймер:

```
self.timer = ticks_ms()
```

В атрибутах объекта, что не очень хорошо, но пока отложим этот недостаток.

А изменилась логика работы с эффектами! Она стала абстрактна!

Вместо конкретной реализации эффекта к нас в классе появился метод:

```
def color_effects(self, color_effect_iter, circle_time=1):
    delta_time = circle_time * 1000 /
color_effect_iter.count_of_iter_cycle
    current_time = ticks_ms()
    if current_time - self.timer > delta_time:
        self.timer = current_time
        colour_tuple = next(color_effect_iter)
        self.turn_on_color(colour_tuple)
```

Структурно похожий на ранее написанный метод:

```
def rainbow(self, circle_time=1):
    self.degree = self.degree % 360
    delta_time = circle_time * 1000 / 360
    current_time = ticks_ms()
    if current_time - self.timer > delta_time:
        self.timer = current_time
        colour_tuple = self.rgb_wheel(self.degree)
        self.degree += 1
        self.turn_on_color(colour_tuple)
```

В сигнатуру которого добавился новый параметр `color_effect_iter`, определяющий тип и поведение эффекта.

Само же получение цвета для диода заменено на:

```
colour_tuple = next(color_effect_iter)
```

В которым через функцию `next()` мы запрашиваем у конкретного итерируемого эффекта `color_effect_iter` следующий цвет.

Чего мы в итоге добились? Мы делегировали логику получения следующего цвета некоторому новому (пока не реализованному) объекту. Что мы о нем пока знаем? Только то, что в нем должна быть реализована функция `next()`, возвращающая новый цвет 8-битного разрешения в формате RGB.

Откуда взялось последнее требование? От логики работы метода `turn_on_color(self, color_tuple)` реализованного в нашем классе ранее.

Итераторы, абстрактные классы и наследование

В Python **итератор** — это объект, который реализует протокол итерации. Этот протокол включает два метода:

- `__iter__()`: Возвращает сам объект итератора.
- `__next__()`: Возвращает следующий элемент из последовательности. Если элементов больше нет, возникает исключение `StopIteration`.

Объекты-итераторы позволяют перебирать элементы коллекций (списков, кортежей, словарей и т.д.) с помощью цикла `for`, оператора `in` и других конструкций.

В контексте решаемой задачи абстрактный вид нашего цветового эффекта может выглядеть следующим образом:

```
class ColorEffectsABC:

    def __init__(self, count_of_iter_cycle):
        self.count_of_iter_cycle = count_of_iter_cycle

    def __iter__(self):
        return self

    def __next__(self):
        raise NotImplementedError("В потомке не реализован метод
'__next__'")
```

Как мы можем видеть, создаваемый эффект должен переопределять в себе работу трех стандартных дандер методов:

В инициализирующем методе:

```
def __init__(self, count_of_iter_cycle):
    self.count_of_iter_cycle = count_of_iter_cycle
```

мы предполагаем наличие во всех эффектах атрибута `count_of_iter_cycle`, определяющего, сколько значений цвета должно быть рассчитано на один полный период эффекта.

Метод `__iter__(self)` - должен вернуть объект итератора. Смотря код мы видим, что он возвращает сам объект.

Почему так происходит? Фактически итератором считается объект, реализующий в себе логику методов `__iter__` и `__next__`.

При этом задача метода `__iter__` состоит только в том, чтобы выдать объект с методом `__next__`.

Из чего видно, что реализация:

```
def __iter__(self):
    return self
```

прекрасно с этим справляется.

Метод `__next__()` определяет логику получения следующего элемента последовательности. В нашем случае следующий цвет эффекта.

Конкретная реализация:

```
def __next__(self):
    raise NotImplementedError("В потомке не реализован метод '__next__'")
```

в классе `ColorEffectsABC` не предусмотрена и просто "выкидывает" исключение `NotImplementedError` с соответствующим комментарием.

Что же получается? Наш класс `ColorEffectsABC` не сможет работать. Когда мы попробуем вызвать метод `next()` в экземпляре этого класса мы получим не цвет, а прекращение работы программы.

Но делает ли это этот класс бесполезным? Отнюдь! Этот класс послужит "макетом" для всех последующих цветовых эффектов, став для них предком!

> ⓘ В полной версии языка `python` классы, определяющие такую функцию называют **абстрактными**, т.е. не предполагающие создание конкретного экземпляра класса. В `micropython` отсутствует встроенная возможность реализации полноценных абстрактных классов, поэтому не будем распылять свое внимание на этот вопрос.

Реализация эффекта "Радуга"

Как и в прошлый раз оттолкнемся от уже законченной реализации:

```
class Rainbow(ColorEffectsABC):

    def __init__(self, start_degree=0, count_of_iter_cycle=360):
        super().__init__(count_of_iter_cycle=count_of_iter_cycle - 1)

        self.degree = start_degree % count_of_iter_cycle

        self.c60 = self.count_of_iter_cycle / 6
        self.c120 = self.count_of_iter_cycle / 3
        self.c180 = self.count_of_iter_cycle / 2
        self.c240 = self.count_of_iter_cycle / 3 * 2
        self.c360 = self.count_of_iter_cycle
```

```
    def rgb_wheel(self):
        def color_pattern(degree):
            degree %= self.count_of_iter_cycle
            if 0 <= degree < self.c60:
                value = degree / self.c60
            elif self.c180 <= degree < self.c240:
                value = 1 - (degree - self.c180) / self.c60
            else:
                value = 1 if self.c60 <= degree < self.c180 else 0
            return int(value * 255)
        red = color_pattern(self.degree - self.c240)
        green = color_pattern(self.degree)
        blue = color_pattern(self.degree - self.c120)
        return red, green, blue

    def __next__(self):
        color_tuple = self.rgb_wheel()
        self.degree += 1
        return color_tuple
```

и разберем ее в сравнении с ранее написанным кодом.

Начнем с того, что наш эффект теперь занимает свой собственный отдельный класс:

```
class Rainbow(ColorEffectsABC):
        ...
```

При этом после названия класса в скобках указана ссылка на суперкласс (класс-предок). После этого все наполнение класса предка полностью переносится на класс потомка.

К чему нас это обязывает?

Во-первых, мы должны передать значение переменной `count_of_iter_cycle`, определяемое при создании объекта `ColorEffectsABC` в его методе `__init__`.

Где и как это проще всего реализовать?

В методе `__init__` нашего класса потомка:

```
def __init__(self, start_degree=0, count_of_iter_cycle=360):
    super().__init__(count_of_iter_cycle=count_of_iter_cycle - 1)
```

В нашем примере мы обращаемся к нашему супер-классу `ColorEffectsABC` через функцию `super()` и явно передаем в метод `__init__` необходимое значение.

Во-вторых, мы должны переопределить с помощью полиморфизма логику работу метода `__next__` супер-класса, так чтобы он передавал требуемый цвет, а не выбрасывал исключение.

```
def __next__(self):
    color_tuple = self.rgb_wheel()
    self.degree += 1
    return color_tuple
```

Все остальное - это тонкости реализации, которые должны быть вполне понятны Вам без дополнительных комментариев.

Проверим работоспособность полученного эффекта запустив следующий код:

```
rgb_led_1 = RGBLed(red_pin=15, green_pin=14, blue_pin=13)
rgb_led_2 = RGBLed(red_pin=16, green_pin=17, blue_pin=18)

rainbow_iter_1 = Rainbow(count_of_iter_cycle=360).__iter__()
rainbow_iter_2 = iter(Rainbow(count_of_iter_cycle=360, start_degree=180))

while True:
    rgb_led_1.color_effects(color_effect_iter=rainbow_iter_1,
circle_time=5)
    rgb_led_2.color_effects(color_effect_iter=rainbow_iter_2,
circle_time=5)
```

> Попробуйте поменять значение в переменной `count_of_iter_cycle` на 7 или другое число!

Отметим только то, что в метод `color_effects` экземпляра класса `RGBLed` необходимо передавать именно итератор класса `Rainbow`, а не сам экземпляр класса.

> ⓘ Если честно, это требование избыточно и нужно скорее в воспитательных целях, для формирования правильной привычки работы с итерируемыми объектами. Это обстоятельство является справедливым в силу того, что наш эффект является бесконечным и не выбрасывает исключение `StopIteration`

Обратное упрощение класса `RGBLed`

Каким бы хорошим нам не казался созданный класс `RGBLed` он все таки содержит в себе поведение `color_effects` и атрибут `timer`, не относящиеся к "чистому" RGB светодиоду.

Исправить это можно, создав новый класс `RGBLedWithEffects`, включающий в себя предыдущую версию класса `RGBLed` через наследование:

```
from machine import PWM, Pin
from time import ticks_ms


class Led:
    BRIGHTNESS_SCALE = 255

    def __init__(self, led_pin):
        self.led = PWM(Pin(led_pin, Pin.OUT), freq=1000)
        self.brightness = 0

    def on(self):
        self.led.duty_u16(65535)
        self.timer = ticks_ms()

    def off(self):
        self.led.duty_u16(0)
        self.timer = ticks_ms()

    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.BRIGHTNESS_SCALE:
            brightness = self.BRIGHTNESS_SCALE
        duty = int(65535 / self.BRIGHTNESS_SCALE * brightness)
        self.led.duty_u16(duty)
        self.brightness = brightness


class RGBLed:

    def __init__(self, red_pin, green_pin, blue_pin):
        self.red = Led(red_pin)
        self.green = Led(green_pin)
        self.blue = Led(blue_pin)

    def turn_on_color(self, color_tuple):
        self.red.turn_to_brightness(color_tuple[0])
        self.green.turn_to_brightness(color_tuple[1])
        self.blue.turn_to_brightness(color_tuple[2])

    def turn_on(self):
        self.turn_on_color([255, 255, 255])

    def turn_off(self):
        self.turn_on_color([0, 0, 0])


class RGBLedWithEffects(RGBLed):
```

```
    def __init__(self, red_pin, green_pin, blue_pin):
        super().__init__(red_pin, green_pin, blue_pin)
        self.timer = ticks_ms()

    def color_effects(self, color_effect_iter, circle_time=1):
        delta_time = circle_time * 1000 /
color_effect_iter.count_of_iter_cycle
        current_time = ticks_ms()
        if current_time - self.timer > delta_time:
            self.timer = current_time
            colour_tuple = next(color_effect_iter)
            self.turn_on_color(colour_tuple)
```

Как видно, классы `Led` и `RGBLed` содержат в себе только базовое поведение диодов, а класс `RGBLedWithEffects` дополняет их поведение необходимой для реализации световых эффектов логикой (что отражено в его названии).

В любом случае, если нам для чего бы то ни было понадобится простой и "чистый" светодиод (не важно монохромный или цветной), у нас есть для этого готовый код!

Но давайте лучше отметим это событие созданием нового кода! Без лишних слов создадим еще несколько эффектов!

Реализация эффекта плавного затухания и разгорания светодиода

Логика реализации этого эффекта была подробно разобрана нами в рамках прошлого задания поэтому просто реализуем следующий класс по аналогии:

```
class Flasher(ColorEffectsABC):

    def __init__(self, initial_color=(255, 0, 0),
count_of_iter_cycle=100):
        super().__init__(count_of_iter_cycle=count_of_iter_cycle)
        self.initial_color = initial_color
        self.brightness = 0
        self.direction = "UP"

    def __next__(self):
        colour = [int(rgb * self.brightness / self.count_of_iter_cycle)
for rgb in self.initial_color]
        if self.direction == "UP":
            self.brightness += 1
            if self.brightness == self.count_of_iter_cycle:
                self.direction = "DOWN"
        if self.direction == "DOWN":
            self.brightness -= 1
```

```
            if self.brightness == 0:
                self.direction = "UP"
        return colour
```

отметим только, что атрибут `initial_color` определяет цвет диода, которым он будет светить.

Проверить работоспособность написанного класса можно, запустив следующий код:

```
rgb_led_1 = RGBLedWithEffects(red_pin=15, green_pin=14, blue_pin=13)
rgb_led_2 = RGBLedWithEffects(red_pin=16, green_pin=17, blue_pin=18)

flasher_iter_1 = iter(Flasher())
flasher_iter_2 = iter(Flasher(initial_color=[0, 255, 0]))

while True:
    rgb_led_1.color_effects(color_effect_iter=flasher_iter_1,
circle_time=1)
    rgb_led_2.color_effects(color_effect_iter=flasher_iter_2,
circle_time=2)
```

Реализация эффекта "моргания" светодиодом

В "моргании" светодиодом нет ничего сложного:

```
class Blinker(ColorEffectsABC):

    def __init__(self, initial_color=(255, 0, 0),
count_of_iter_cycle=100):
        super().__init__(count_of_iter_cycle=count_of_iter_cycle)
        self.initial_color = initial_color
        self.counter = 0
        self.state = "OFF"

    def __next__(self):
        if self.counter % self.count_of_iter_cycle == 0:
            if self.state == "ON":
                self.state = "OFF"
            elif self.state == "OFF":
                self.state = "ON"
            self.counter = 0
        self.counter += 1
        return self.initial_color if self.state == "ON" else [0, 0, 0]
```

отметим только, что переменная `count_of_iter_cycle` выступает в роли "заглушки", определяющей количество циклов вызовов метода `__next__` до следующей смены состояния светодиода.

Проверить реализацию эффекта можно запустив следующий код:

```
rgb_led_1 = RGBLedWithEffects(red_pin=15, green_pin=14, blue_pin=13)
rgb_led_2 = RGBLedWithEffects(red_pin=16, green_pin=17, blue_pin=18)

blinker_iter_1 = iter(Blinker())
blinker_iter_2 = iter(Blinker(initial_color=[0, 255, 0]))

while True:
    rgb_led_1.color_effects(color_effect_iter=blinker_iter_1,
circle_time=1)
    rgb_led_2.color_effects(color_effect_iter=blinker_iter_2,
circle_time=2)
```

Совмещение нескольких эффектов

Разделение логики работы отдельных эффектов на различные классы позволяет добиться высокой гибкости кода.

Так следующий класс совмещает эффект "моргания" с произвольным эффектом, агрегируя их внутри класса, по аналогии с объектами `Led` в классе `RGBLed`:

```
class BlinkerAggregator(ColorEffectsABC):

    def __init__(self, base_color_effects, count_of_iter_cycle=100):
        super().__init__(count_of_iter_cycle=count_of_iter_cycle)
        self.base_iter = iter(base_color_effects)
        self.blink_iter =
iter(Blinker(count_of_iter_cycle=count_of_iter_cycle))

    def __next__(self):
        base_color = next(self.base_iter)
        next(self.blink_iter)
        return base_color if self.blink_iter.state == "ON" else [0, 0, 0]
```

```
rgb_led_1 = RGBLedWithEffects(red_pin=15, green_pin=14, blue_pin=13)
rgb_led_2 = RGBLedWithEffects(red_pin=16, green_pin=17, blue_pin=18)

ba_rainbow_iter = BlinkerAggregator(base_color_effects=Rainbow())
ba_flasher_iter = BlinkerAggregator(base_color_effects=Flasher())

while True:
    rgb_led_1.color_effects(color_effect_iter=ba_rainbow_iter,
circle_time=1)
    rgb_led_2.color_effects(color_effect_iter=ba_flasher_iter,
circle_time=2)
```

Реализация эффекта "огня"

Логика генерации цветов в эффекте не обязательно должна быть заключена в методе `__next__`.

Сам по себе создаваемый класс для эффекта ничем не хуже любого другого класса и может быть реализован с привлечением внутренней сложноподчиненной логики.

Напишем в заключение класс, имитирующий "горение" свечки:

```
class Fire(ColorEffectsABC):
    def __init__(self, initial_color=(255, 30, 0), flicker_range=10,
brightness_range=0.05, count_of_iter_cycle=100):
        """
        Инициализация итератора для имитации свечения свечи.
        :param initial_color: Начальный цвет в формате RGB.
        :param flicker_range: Диапазон изменения цвета.
        :param brightness_range: Диапазон изменения яркости.
                """
                super().__init__(count_of_iter_cycle=count_of_iter_cycle)
        self.initial_color = initial_color
        self.flicker_range = flicker_range
        self.brightness_range = brightness_range
        self.current_color = initial_color
        self.current_brightness = 1.0

    def get_next_random_color(self):
        # Генерация случайных изменений цвета
        r = (self.current_color[0] +
             random.randint(-self.flicker_range, self.flicker_range) *
             self.initial_color[0] / 255)
        g = (self.current_color[1] +
             random.randint(-self.flicker_range, self.flicker_range) *
             self.initial_color[1] / 255)
        b = (self.current_color[2] +
             random.randint(-self.flicker_range, self.flicker_range) *
             self.initial_color[2] / 255)
        # Ограничение значений цвета в диапазоне от 0 до 255
        rgb = [max(0, min(255, rgb_)) for rgb_ in [r, g, b]]
        return rgb

    def get_random_brightness_color(self):
        # Генерация случайных изменений цвета
        self.current_color = self.get_next_random_color()
        # Генерация случайной яркости
        brightness_change = random.uniform(-self.brightness_range,
self.brightness_range)
        self.current_brightness += brightness_change
        self.current_brightness = max(0.0, min(1.0,
```

```
self.current_brightness))
        # Применение яркости к цвету
        rgb = [int(rgb_ * self.current_brightness) for rgb_ in
self.current_color]
        return rgb

    def __next__(self):
        """
        Генерирует следующий цвет и яркость, имитируя свечение свечи.
        :return: Кортеж (R, G, B) с новым цветом.
        """
        # Генерация случайных изменений цвета
        self.current_color = self.get_next_random_color()
        # Генерация случайной яркости
        rgb = self.get_random_brightness_color()
        return rgb
```

Проверим код:

```
rgb_led_1 = RGBLedWithEffects(red_pin=15, green_pin=14, blue_pin=13)
rgb_led_2 = RGBLedWithEffects(red_pin=16, green_pin=17, blue_pin=18)

fire_iter_1 = Fire()
fire_iter_2 = Fire(initial_color=[0, 0, 255])

while True:
    rgb_led_1.color_effects(color_effect_iter=fire_iter_1, circle_time=1)
    rgb_led_2.color_effects(color_effect_iter=fire_iter_2, circle_time=2)
```