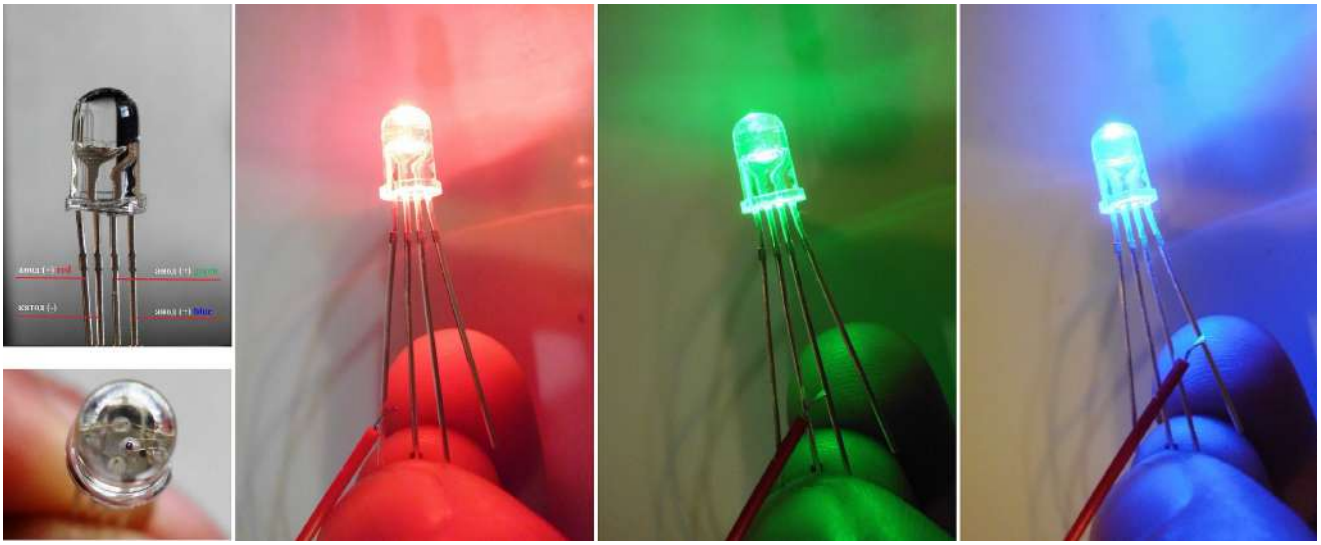
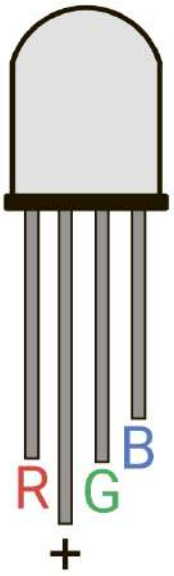
Четвертое практическое занятие

Функции. Рефакторинг. Работа с RGB диодом

RGB светодиод

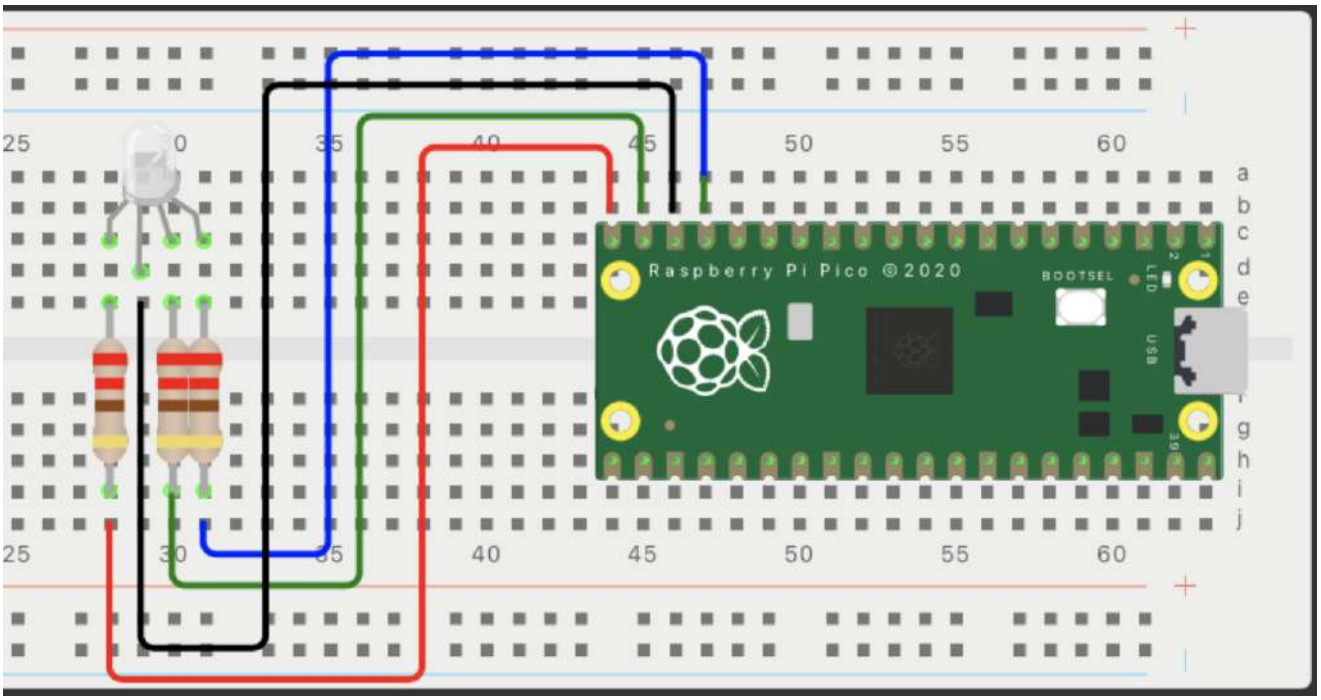


RGB LED модули имеют характерную конструкцию — три цветных диода, установленные на одной матрице и покрытые единой оптической линзой. В качестве базы используется гибкая лента или жесткая матрица с трехслойной структурой. Каждый кристалл имеет отдельное подключение к источнику питания. Соответственно, RGB-светодиод имеет 4 контакта — общий и по одному на полупроводник.

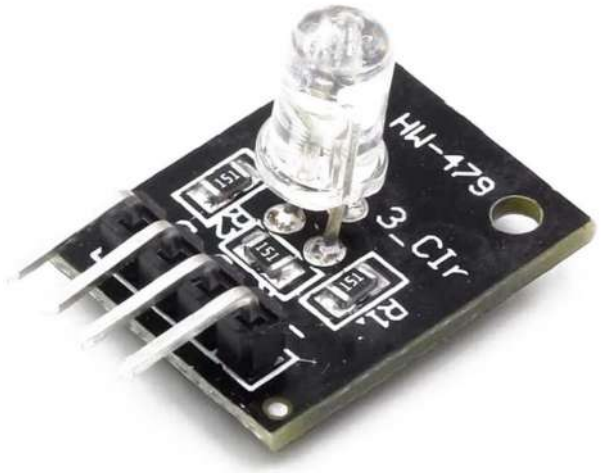


При одновременном включении 2-3 элементов на определенной мощности появляется вторичное свечение люминофора с формированием различных оттенков. Так, при парном включении красного и зеленого кристалла, RGB-светодиод даст желтый свет. Одновременная активация синего и зеленого позволит получить бирюзовый оттенок.

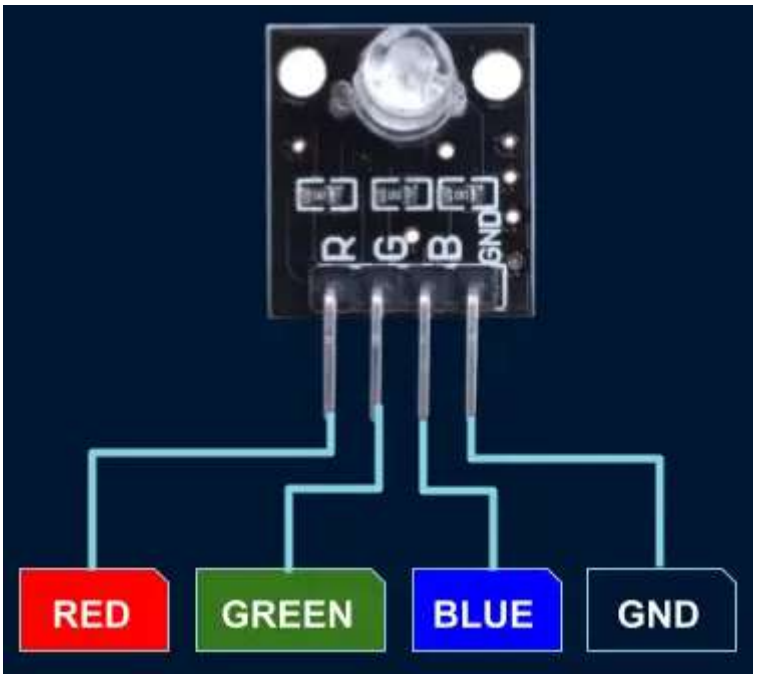
При работе с "чистым" RGB светодиодом необходимо подключать внешние резисторы к каждой его ножке отвечающей за конкретный цвет:



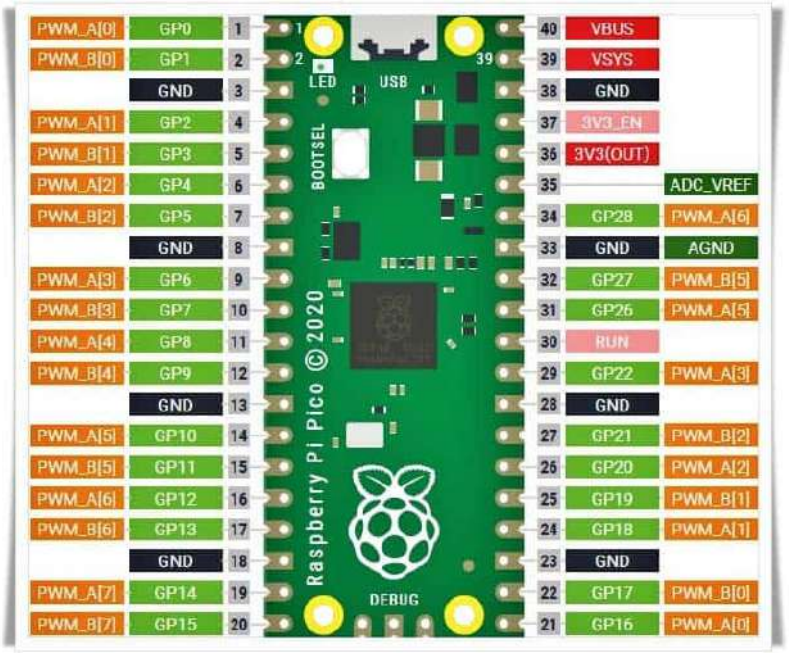
Так как это требует большого количества дополнительных компонентов в процессе обучения эффективнее использовать уже готовые модули, включающие в себя помимо самого светодиода необходимые сопротивления:



Следует отметить, что распиновка RGB модуля отлична от распиновки светодиода, поэтому при его подключении необходимо следовать соответствующим обозначениям на модуле:

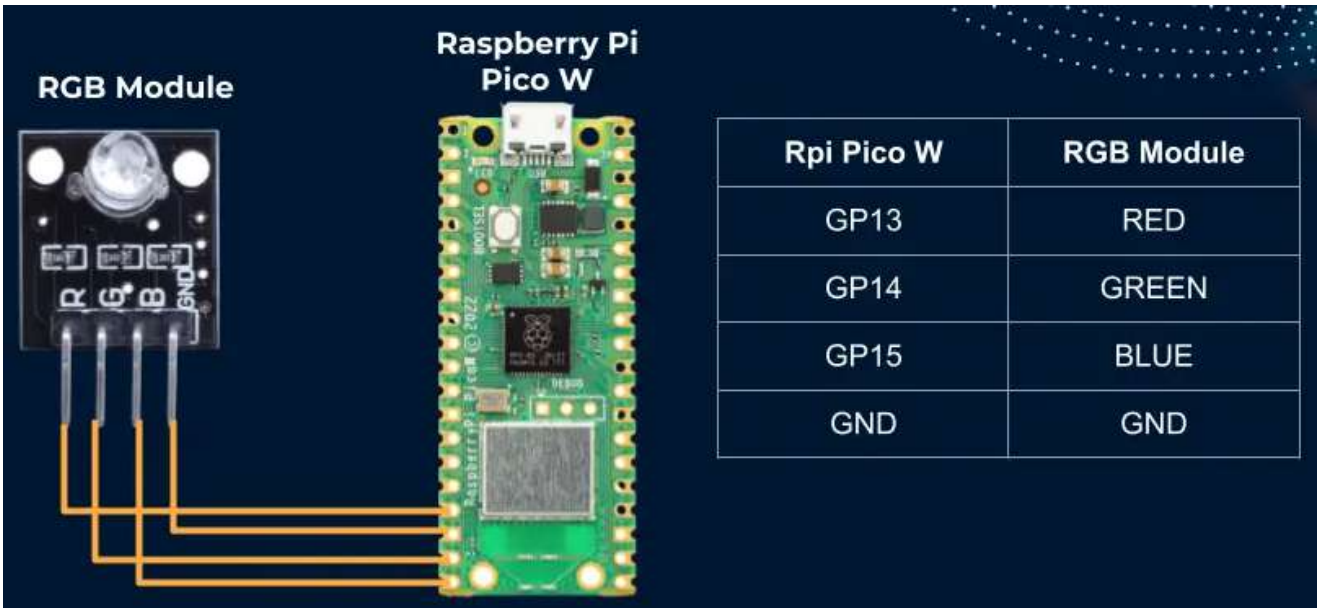


Так как все цифровые пины Raspberry Pi Pico поддерживают ШИМ, то выводы R, G и B модуля могут быть подключены к любому из них при условии отсутствия повторов в "срезах" ШИМ:



Например, нельзя подключить одновременно два вывода модуля к пинам GP21 и GP5 так как они дублируют один и тот же срез PWM_B[2].

Пример возможной схемы подключения представлен на рисунке:



Rpi Pico W	RGB Module
GP13	RED
GP14	GREEN
GP15	BLUE
GND	GND

Управляя RGB модулем с помощью цифрового сигнала мы можем полностью включать и выключать по отдельности красный, зеленый и синий цвет, формируя, при этом, 8 возможных комбинаций:

Цвет	R	G	B
"Черный"	-	-	-
Красный	+	-	-
Желтый	+	+	-
Зеленый	-	+	-
Бирюзовый	-	+	+
Синий	-	-	+
Пурпурный	+	-	+
Белый	+	+	+

```
from machine import Pin
from time import sleep

RED_PIN = 13
GREEN_PIN = 14
BLUE_PIN = 15

red_led = Pin(RED_PIN, Pin.OUT)
green_led = Pin(GREEN_PIN, Pin.OUT)
blue_led = Pin(BLUE_PIN, Pin.OUT)

while True:
        # "Черный"
    red_led.off()
    green_led.off()
    blue_led.off()
```

```
    sleep(1)

        # Красный
    red_led.on()
    green_led.off()
    blue_led.off()
    sleep(1)

        # Желтый
    red_led.on()
    green_led.on()
    blue_led.off()
    sleep(1)

        # Зеленый
    red_led.off()
    green_led.on()
    blue_led.off()
    sleep(1)

        # Бирюзовый
    red_led.off()
    green_led.on()
    blue_led.on()
    sleep(1)

        # Синий
    red_led.off()
    green_led.off()
    blue_led.on()
    sleep(1)

        # Пурпурный
    red_led.on()
    green_led.off()
    blue_led.on()
    sleep(1)

        # Белый
    red_led.on()
    green_led.on()
    blue_led.on()
    sleep(1)
```

Для получения более сложных оттенков необходимо "смешивать" отдельные цвета с помощью ШИМ. Код ниже позволяет получить чередующиеся красный, зеленый и синий цвета:

```
from machine import PWM
from time import sleep

RED_PIN = 13
GREEN_PIN = 14
BLUE_PIN = 15

red_led = PWM(RED_PIN, freq=1000)
green_led= PWM(GREEN_PIN, freq=1000)
blue_led = PWM(BLUE_PIN, freq=1000)

while True:
    red_led.duty_u16(65535)
    green_led.duty_u16(0)
    blue_led.duty_u16(0)
    sleep(1)
    red_led.duty_u16(0)
    green_led.duty_u16(65535)
    blue_led.duty_u16(0)
    sleep(1)
    red_led.duty_u16(0)
    green_led.duty_u16(0)
    blue_led.duty_u16(65535)
    sleep(1)
```

Вводя значения в пределах от 0 до 65535, можно "смешивать" и управлять яркостью (интенсивностью) базовых цветов в разных пропорциях, получая множество различных оттенков.

Рефакторинг кода

Стройный, хорошо структурированный код легко читается и быстро дорабатывается. Но редко удаётся сразу сделать его таким. Разработчики спешат, в процессе могут меняться требования к задаче, тестировщики находят баги, которые нужно быстро исправить, или возникают срочные доработки, и их приходится делать второпях.

В результате даже изначально хорошо структурированный исходник становится беспорядочным и непонятным. Программисты знают, как легко завязнуть в этом хаосе. Причём неважно, чужой это код или собственный.

Чтобы решить все эти проблемы, делается рефакторинг программы. В новом проекте он нужен, чтобы:

- сохранить архитектуру проекта, не допустить потери структурированности;
- упростить будущую жизнь разработчиков, сделать код понятным и прозрачным для всех членов команды;

- ускорить разработку и поиск ошибок.

Но любое приложение со временем устаревает: язык программирования совершенствуется, появляются новые функции, библиотеки, операторы, делающие код проще и понятнее. То, что год назад требовало пятидесяти строк, сегодня может решаться всего одной.

Поэтому даже идеальная когда-то программа со временем требует нового рефакторинга, обновляющего устаревшие участки кода.

> Программный код предназначен не только для компьютера, но и для человека, который будет его дорабатывать. Плохо, если ему придётся неделю разбираться в исходниках, чтобы изменить в программе несколько строк. И не исключено, что этим человеком окажетесь вы сами.

Рассмотрев предыдущий блок кода видно, что в нем есть повторяющиеся смысловые конструкции:

```
    red_led.duty_u16(65535)
    green_led.duty_u16(0)
    blue_led.duty_u16(0)
    sleep(1)
    red_led.duty_u16(0)
    green_led.duty_u16(65535)
    blue_led.duty_u16(0)
    sleep(1)
    red_led.duty_u16(0)
    green_led.duty_u16(0)
    blue_led.duty_u16(65535)
    sleep(1)
```

А понимание того какой цвет непосредственно будет включен находится в области допущения, что читающий код пользователь знаком с непосредственной логикой реализации ШИМ на конкретном микроконтроллере, и сможет сопоставить передаваемые в него значения с конкретными названиями переменных.

Все это в перспективе приведет к тому, что рано или поздно (скорее всего очень рано) код программы превратится в неподдерживаемую, непонятную и забагованную конструкцию, поэтому необходимо начать работы по его рефакторингу.

> **Рефакторинг** — это переработка исходного кода программы, чтобы он стал более простым и понятным.

> Рефакторинг не меняет поведение программы, не исправляет ошибки
> и не добавляет новую функциональность. Он делает код более понятным

> и удобочитаемым.

Для начала создадим используемую в программе палитру цветов присвоив кортежи с определяемым цветом переменным:

```
red = (65535, 0, 0)
green = (0, 65535, 0)
blue = (0, 0, 65535)

magenta = (65535, 0, 65535)
yellow = (65535, 65535, 0)
cian = (0, 65535, 65535)
orange = (65535, 32768, 0)
white = (65535, 65535, 65535)
```

Сделав это становится возможным итерировать необходимый набор цветов через цикл *for* упростив тем самым блок кода *while*:

```
while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        red_led.duty_u16(color[0])
        green_led.duty_u16(color[1])
        blue_led.duty_u16(color[2])
        sleep(1)
```

Остается только собрать это все обратно в целую программу.

Результирующий код первого этапа рефакторинга

```
from machine import PWM
from time import sleep

RED_PIN = 13
GREEN_PIN = 14
BLUE_PIN = 15

red_led = PWM(RED_PIN, freq=1000)
green_led= PWM(GREEN_PIN, freq=1000)
blue_led = PWM(BLUE_PIN, freq=1000)

red = (65535, 0, 0)
green = (0, 65535, 0)
blue = (0, 0, 65535)

magenta = (65535, 0, 65535)
yellow = (65535, 65535, 0)
```

```
cian = (0, 65535, 65535)
orange = (65535, 32768, 0)
white = (65535, 65535, 65535)

while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        red_led.duty_u16(color[0])
        green_led.duty_u16(color[1])
        blue_led.duty_u16(color[2])
        sleep(1)
```

Золотое правило рефакторинга:

> Не сломай то, что работает!

И оно пока выполнено, что хорошо, но, плохо, что код программы все еще имеет ряд стилистических и логических недочетов.

Так переменные цветов заданы в нестандартной системе (16-битный цвет):

```
red = (65535, 0, 0)
green = (0, 65535, 0)
blue = (0, 0, 65535)
```

Более привычным является 8-битное представление:

```
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
```

Сделав такую замену и перезапустив предыдущий код, мы можем увидеть, что он работает, но не очень хорошо - диод начал гореть очень тускло (золотое правило рефакторинга нарушено).

Чтобы исправить это, необходимо заново "масштабировать" значения цветов обратно до 16-битного значений:

```
while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        red_led.duty_u16(int(color[0] * 65535 / 255))
        green_led.duty_u16(int(color[1] * 65535 / 255))
        blue_led.duty_u16(int(color[2] * 65535 / 255))
        sleep(1)
```

С одной стороны программа вернула исходную работоспособность, но вместе с ней преумножила перегруженность приведенного выше блока *while*.

В идеале этот блок не должен содержать в себе сложной логики, а прямое обращение к переменным и реализация сложных алгоритмов должны быть сведены к минимуму.

Решить эту проблему возможно, определив по логически обособленным блокам кода функции.

Для начала сгруппируем переменные *red_led*, *green_led*, *blue_led* в одну коллекцию, определяющую по смыслу RGB светодиод.

Приняв во внимание ранее замеченный эффект, что умножив значения цветов на константу от 0 до 1 мы можем понизить тем самым яркость диода, можно инкапсулировать все выполняемые алгоритмические операции в функции - *switch_on_a_colour_of_the_specified_brightness*:

```
rgb_led = (red_led, green_led, blue_led)

def switch_on_a_colour_of_the_specified_brightness(rgb_led, color_tuple,
brightness=1):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(int(color * 65535 / 255 * brightness))

while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        switch_on_a_colour_of_the_specified_brightness(rgb_led, color)
        sleep(1)
```

Сделав это, мы вынесли из блока *while* все, что требует явных вычислений и конкретных знаний по работе с микроконтроллером, оставив в нем практически человекочитаемый текст:

- бесконечно повторяя ...
- перебирай по одному из перечисленных цветов ...
- зажги этот цвет на светодиоде *rgb_led*;
- ничего не делай 1 секунду.

Собрав все вместе получим...

Результирующий код второго этапа рефакторинга

```
from machine import PWM
from time import sleep

RED_PIN = 15
GREEN_PIN = 14
BLUE_PIN = 13
```

```
red_led = PWM(RED_PIN, freq=1000)
green_led= PWM(GREEN_PIN, freq=1000)
blue_led = PWM(BLUE_PIN, freq=1000)

rgb_led = (red_led, green_led, blue_led)

red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

magenta = (255, 0, 255)
yellow = (255, 255, 0)
cian = (0, 255, 255)
orange = (255, 128, 0)
white = (255, 255, 255)


def switch_on_a_colour_of_the_specified_brightness(rgb_led, color_tuple,
brightness=1):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(int(color * 65535 / 255 * brightness))


while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        switch_on_a_colour_of_the_specified_brightness(rgb_led, color)
        sleep(1)
```

На прошлом этапе рефакторинга мы достигли большого успеха - избавили блок *while* от всякой алгоритмической и микроконтроллерной шелухи! Но какой ценой?.. Мы ввели в наш код новую сущность - функцию "*switch_on_a_colour_of_the_specified_brightness*", но хороша ли она? Это не риторический вопрос и на него есть однозначный, честный и конкретный ответ - эта функция божественна! Поверхностному пользователю или программисту прочитав эти слова можно успокоиться и остановиться, но мы пытливо постараемся проникнуть вглубь этого понятия!

Божественная функция является частным случаем антипаттерна (плохой модели поведения) божественный объект. В чем суть этого явления? Она делает слишком много....

> Правильно реализованная функция должна реализовывать в себе только одно действие!

Сколько же делает наша? Целых три!

- Пересчитывает значения цвета из 8-битного значения в 16-битное;
- Учитывает в полученном цвете требуемую яркость;
- Передает полученное значение на светодиод.

И что же делать?

Нужно создать по отдельной функции для каждого из этих действий. Разделим ранее написанный код:

```
def switch_on_a_colour_of_the_specified_brightness(rgb_led, color_tuple,
brightness=1):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(int(color * 65535 / 255 * brightness))
```

на три отдельные функции отвечающие за каждое из названых действий:

```
def get_colour_data_for_pwm(color_8bit_tuple):
    color_16bit_tuple = tuple([int(rgb * 65535 / 255) for rgb in
color_8bit_tuple])
    return color_16bit_tuple


def set_the_colour_brightness(color_tuple, brightness):
    colour_with_a_specified_brightness = tuple([int(rgb * brightness /
100) for rgb in color_tuple])
    return colour_with_a_specified_brightness


def switch_on_a_color(rgb_led, color_tuple):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(color)
```

Отлично! Теперь каждая из функций отвечает за конкретное действие:

- *get_colour_data_for_pwm* - переводит цвет к требуемому для ШИМ значению;
- *set_the_colour_brightness* - нормирует его по требуемой яркости;
- *switch_on_a_color* - включает цвет на светодиоде.

Решили ли мы проблему "божественной" функции "*switch_on_a_colour_of_the_specified_brightness*"? Однозначно! Теперь ее вообще нет в нашем коде! И что же получается? Блок кода:

```
while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        switch_on_a_colour_of_the_specified_brightness(rgb_led, color)
        sleep(1)
```

утратил консистентность - он перестал работать по причине отсутствия функции "*switch_on_a_colour_of_the_specified_brightness*" в нашем коде. Другими словами мы опять нарушили золотое правило рефакторинга и сломали нашу программу.

Как решить эту проблему? Очевидно! Нужно создать эту функцию, но не перегружая ее дополнительными обязанностями.

Исходя из наших требований и своего названия она должна:

- включить нужный цвет;
- установить нужную яркость;

и хотя пользователю функции это не очевидно (да и ему на самом деле все равно) должна подготовить данные о цвете для микроконтроллера.

Фактически эта функция должна выполнить все три предыдущие, или на языке кода:

```
def switch_on_a_colour_of_the_specified_brightness(rgb_led, color_tuple,
brightness=1):
    color_tuple = get_colour_data_for_pwm(color_tuple)
    color_tuple = set_the_colour_brightness(color_tuple, brightness)
    switch_on_a_color(rgb_led, color_tuple)
```

В очередной раз собрав все вместе можем получить...

Результирующий код третьего этапа рефакторинга

```
from machine import PWM
from time import sleep

RED_PIN = 15
GREEN_PIN = 14
BLUE_PIN = 13

red_led = PWM(RED_PIN, freq=1000)
green_led = PWM(GREEN_PIN, freq=1000)
blue_led = PWM(BLUE_PIN, freq=1000)

rgb_led = (red_led, green_led, blue_led)

red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

magenta = (255, 0, 255)
yellow = (255, 255, 0)
```

```
cian = (0, 255, 255)
orange = (255, 128, 0)
white = (255, 255, 255)


def get_colour_data_for_pwm(color_8bit_tuple):
    color_16bit_tuple = tuple([int(rgb * 65535 / 255) for rgb in
color_8bit_tuple])
    return color_16bit_tuple


def set_the_colour_brightness(color_tuple, brightness):
    colour_with_a_specified_brightness = tuple([int(rgb * brightness /
100) for rgb in color_tuple])
    return colour_with_a_specified_brightness


def switch_on_a_color(rgb_led, color_tuple):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(color)


def switch_on_a_colour_of_the_specified_brightness(rgb_led, color_tuple,
brightness=1):
    color_tuple = get_colour_data_for_pwm(color_tuple)
    color_tuple = set_the_colour_brightness(color_tuple, brightness)
    switch_on_a_color(rgb_led, color_tuple)

while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        switch_on_a_colour_of_the_specified_brightness(rgb_led, color)
        sleep(1)
```

Теперь все снова работает!

Внимательный скептик может заметить, что часть нашего запланированного поведения не используется в нашей программе. В блоке:

```
while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        switch_on_a_colour_of_the_specified_brightness(rgb_led, color)
        sleep(1)
```

мы не передаем (и не меняем) значение яркости и принимаем ее значение по умолчанию:

```
brightness=1
```

исправить это можно изменив блок кода while:

```
while True:
    for color in red, orange, yellow, green, cian, blue, magenta:
        for brightness in range(100):
            switch_on_a_colour_of_the_specified_brightness(rgb_led, color,
brightness)
            sleep(0.01)
        for brightness in range(100, 0, -1):
            switch_on_a_colour_of_the_specified_brightness(rgb_led, color,
brightness)
            sleep(0.01)
```

теперь каждый цвет будет разгораться и потухать перед своей сменой.

Что еще можно сделать?

Ну у нас есть целый набор цветов:

```
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

magenta = (255, 0, 255)
yellow = (255, 255, 0)
cian = (0, 255, 255)
orange = (255, 128, 0)
white = (255, 255, 255)
```

и каждому их них соответствует отдельная переменная...

Сколько таких цветов может быть?

$$255^3 = 16581375$$

Весьма немало... но что будет, если присваивать каждому из них свою переменную? Скорее всего ничего хорошего и имеет смысл обобщить их в одну коллекцию...

Сделаем это!

```
colors = {"red": (255, 0, 0),
          "green": (0, 255, 0),
          "blue": (0, 0, 255),
          "magenta": (255, 0, 255),
          "yellow": (255, 255, 0),
          "cian": (0, 255, 255),
```

```
          "orange": (255, 128, 0),
          "white": (255, 255, 255),
          "black": (0, 0, 0),
          }
```

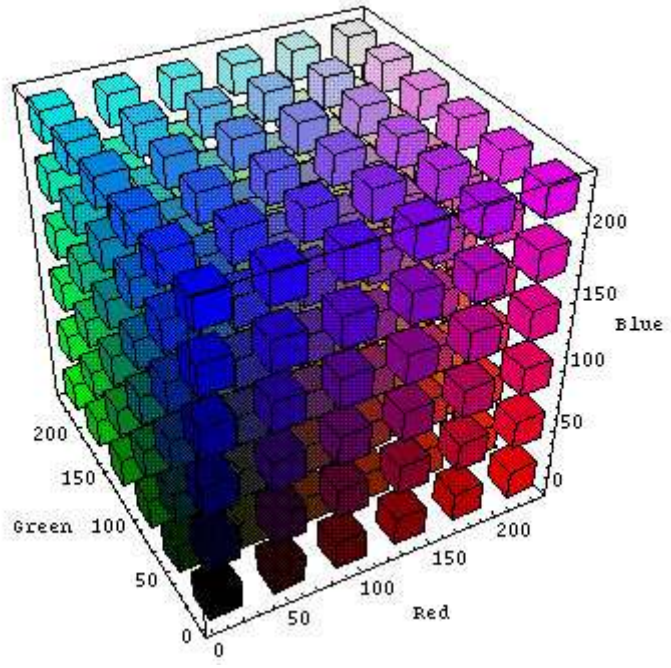
а весь код будет выглядеть как:

```
from machine import PWM
from time import sleep

RED_PIN = 15
GREEN_PIN = 14
BLUE_PIN = 13

red_led = PWM(RED_PIN, freq=1000)
green_led = PWM(GREEN_PIN, freq=1000)
blue_led = PWM(BLUE_PIN, freq=1000)

rgb_led = (red_led, green_led, blue_led)

colors = {"red": (255, 0, 0),
          "green": (0, 255, 0),
          "blue": (0, 0, 255),
          "magenta": (255, 0, 255),
          "yellow": (255, 255, 0),
          "cian": (0, 255, 255),
          "orange": (255, 128, 0),
          "white": (255, 255, 255),
          "black": (0, 0, 0),
          }

def get_colour_data_for_pwm(color_8bit_tuple):
    color_16bit_tuple = tuple([int(rgb * 65535 / 255) for rgb in
color_8bit_tuple])
    return color_16bit_tuple

def switch_on_a_color(rgb_led, color_tuple):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(color)

print(*colors.keys())
while True:
    for color, color_tuple in colors.items():
        switch_on_a_color(rgb_led, get_colour_data_for_pwm(color_tuple))
        print(color + "        ", end="\r")
        sleep(1)
```

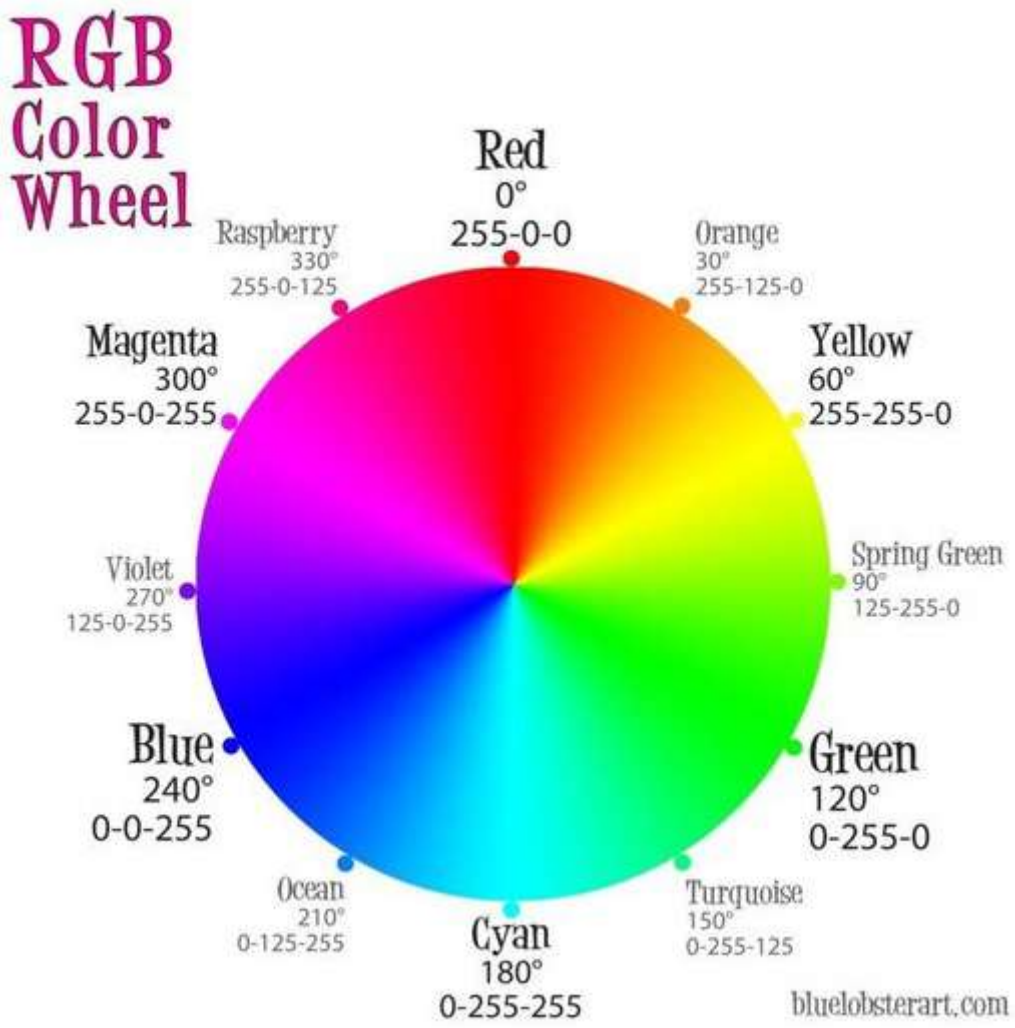
Для компактности мы не меняем яркость, а значит можем не использовать все созданные функции. Но стоит обратить внимание на один важный момент: перебирая элементы словаря *colors*, порядок цветов не будет совпадать с порядком перечисления при его создании. Это объясняется тем, что хэш таблица (словарь) является не упорядоченной коллекцией, в которой последовательность перечисления элементов не гарантируется!

Круг цвета

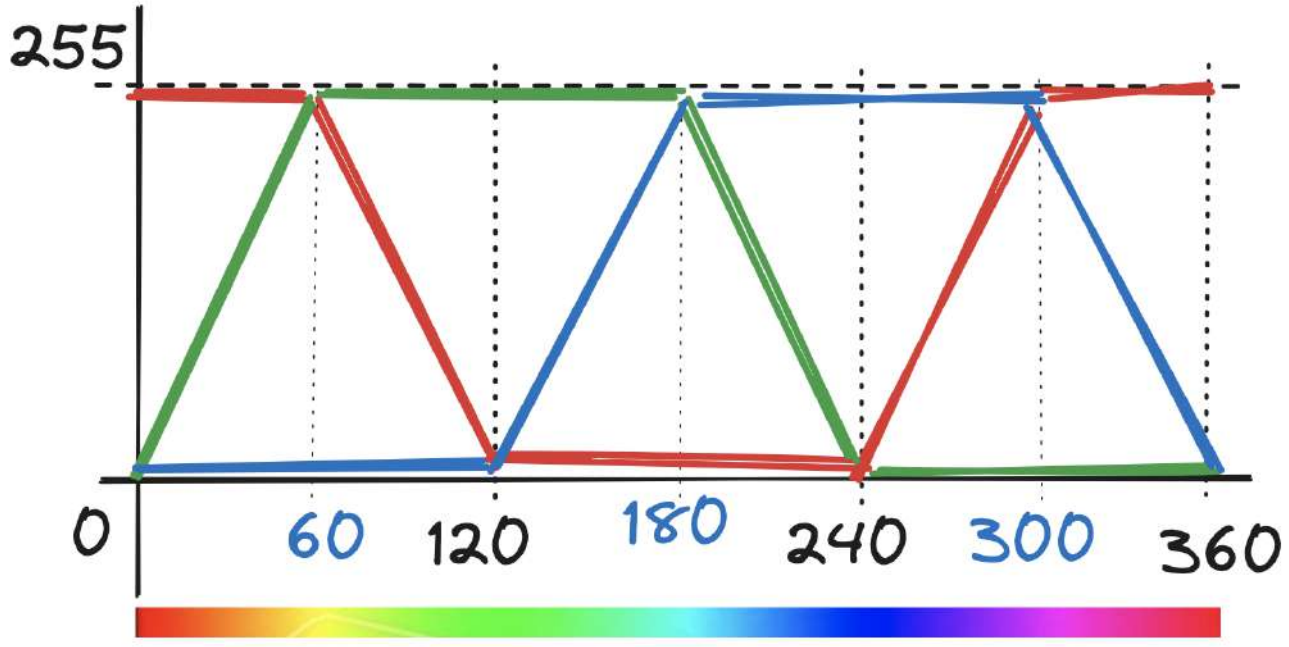
Проработав предыдущий материал, мы видим, что меняя компоненты красного, зеленого и синего цветов, мы можем получать различные оттенки. Форма представления переменной соответствующей конкретному цвету может быть различной, в частности мы использовали кортежи из трех целочисленных значений, находящихся в пределах от 0 до 255. Визуализировать такую структуру описания цвета можно следующим графиком:



Но можно ли описать цвет по другому? Можно ли свести вызов конкретного цвета к одному числа? Для ответа на этот вопрос взглянем на известную диаграмму "цветового круга":



Из представленного рисунка видно, что переход из одного оттенка в другой происходит по некоторому паттерну (закономерности). Попробуем визуализировать его следующим графиком:



Напишем код реализующий представленную закономерность:

```
def rgb_wheel(degree):
    degree %= 360
    if 0 <= degree < 60:
        red = 1
        green = degree / 60
        blue = 0
    elif 60 <= degree < 120:
        red = 1 - (degree - 60) / 60
        green = 1
        blue = 0
    elif 120 <= degree < 180:
        red = 0
        green = 1
        blue = (degree - 120) / 60
    elif 180 <= degree < 240:
        red = 0
        green = 1 - (degree - 180) / 60
        blue = 1
    elif 240 <= degree < 300:
        red = (degree - 240) / 60
        green = 0
        blue = 1
    else:           # 300 <= degree <= 360
        red = 1
        green = 0
        blue = 1 - (degree - 300) / 60
    rgb_tuple = (int(red * 255), int(green * 255), int(blue * 255))
    return rgb_tuple
```

Может он и выглядит неказисто, но должен работать! Подставим его в ранее написанный код и заставим диод плавно изменять цвет:

```
from machine import PWM
from time import sleep

RED_PIN = 15
GREEN_PIN = 14
BLUE_PIN = 13

red_led = PWM(RED_PIN, freq=1000)
green_led = PWM(GREEN_PIN, freq=1000)
blue_led = PWM(BLUE_PIN, freq=1000)

rgb_led = (red_led, green_led, blue_led)

def get_colour_data_for_pwm(color_8bit_tuple):
    color_16bit_tuple = tuple([int(rgb * 65535 / 255) for rgb in
color_8bit_tuple])
```

```
    return color_16bit_tuple


def set_the_colour_brightness(color_tuple, brightness):
    colour_with_a_specified_brightness = tuple([int(rgb * brightness /
100) for rgb in color_tuple])
    return colour_with_a_specified_brightness


def switch_on_a_color(rgb_led, color_tuple):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(color)


def switch_on_a_colour_of_the_specified_brightness(rgb_led, color_tuple,
brightness=1):
    color_tuple = get_colour_data_for_pwm(color_tuple)
    color_tuple = set_the_colour_brightness(color_tuple, brightness)
    switch_on_a_color(rgb_led, color_tuple)

def rgb_wheel(degree):
    degree %= 360
    if 0 <= degree < 60:
        red = 1
        green = degree / 60
        blue = 0
    elif 60 <= degree < 120:
        red = 1 - (degree - 60) / 60
        green = 1
        blue = 0
    elif 120 <= degree < 180:
        red = 0
        green = 1
        blue = (degree - 120) / 60
    elif 180 <= degree < 240:
        red = 0
        green = 1 - (degree - 180) / 60
        blue = 1
    elif 240 <= degree < 300:
        red = (degree - 240) / 60
        green = 0
        blue = 1
    else:           # 300 <= degree <= 360
        red = 1
        green = 0
        blue = 1 - (degree - 300) / 60
    rgb_tuple = (int(red * 255), int(green * 255), int(blue * 255))
    return rgb_tuple

while True:
```

```
    for degree in range(0, 361):
        color_tuple = get_colour_data_for_pwm(rgb_wheel(degree))
        print(rgb_wheel(degree))
        switch_on_a_color(rgb_led, color_tuple)
        sleep(0.05)
```

Мы в очередной раз достигли успеха! Все работает!

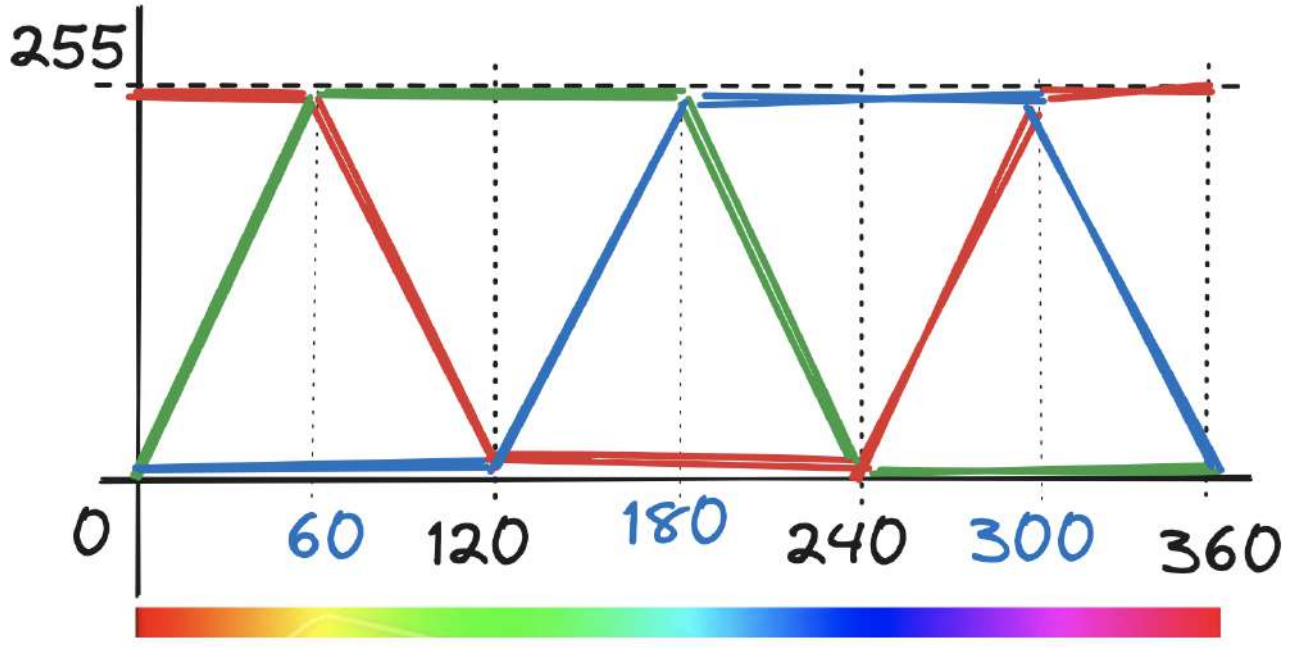
Но можем мы улучшить наш код? В большинстве случаев ответ на подобное размышление всегда - "да", и этот случай не исключение.

В функции *rgb_wheel* мы последовательно прописали логику для 6 сценариев одновременного изменения трех цветов. Пускай и не так явно, но код внутри нашей функции содержит в себе многочисленные повторы, что хотя и не нарушает ход его работы, но делает его туповатым и некрасивым.

Избавимся от этого с помощью рефакторинга!

Рефакторинг функции *rgb_wheel*

Для этого взглянем еще раз на ранее полученный график:



Из графика легко видно, что характер изменения значений для каждого из базовых цветов цикличен и одинаков, а разница состоит только в "фазе". Так синий цвет "смещен" относительно зеленого на 120°, а красный на 240°.

Исходя из этого, логичным представляется выделить паттерн изменения одного цвета в отдельную функцию:

```
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)
```

По своему наполнению полученная функция описывает изменения зеленого цвета, но этот выбор продиктован субъективным личным выбором из-за того, что просто изменение цвета в нем начинается с 0, в отличие от прочих цветов. В любом случае, для нас сейчас это только кусочек решаемой задачи и эту функцию следует воспринимать просто как абстрактное изменение значения цвета в интервале от 0° до 360°.

Перепишем нашу функцию *rgb_wheel* опираясь на разработанную функцию *color_pattern*:

```
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)

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

Как минимум, мы добились серьезного упрощения функции *rgb_wheel*, но есть и недостаток - мы ввели служебную, вспомогательную функцию color_pattern в нашу программу наравне с остальными полезными функциями (*get_colour_data_for_pwm*, *switch_on_a_color*, *set_the_colour_brightness* и т.д.)

И прежде, чем сделать это, нужно ответить себе на вопрос: имеет ли она смысл за пределами функции *rgb_wheel*?

Если ответ - Нет! То значит, что этот блок логики и кода и нужно в нем оставить, оформив его как *вложенную функцию*:

```
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
```

В такой записи доступ к функции *color_pattern* будет возможен только внутри кода функции *rgb_wheel*, и мы убережем пользователя нашего кода от соблазна (и возможности) использовать эту функцию и получить тем самым последующие ошибки и проблемы.

В итоге результирующий код будет выглядеть как:

```
from machine import PWM
from time import sleep

RED_PIN = 15
GREEN_PIN = 14
BLUE_PIN = 13

red_led = PWM(RED_PIN, freq=1000)
green_led = PWM(GREEN_PIN, freq=1000)
blue_led = PWM(BLUE_PIN, freq=1000)

rgb_led = (red_led, green_led, blue_led)

def get_colour_data_for_pwm(color_8bit_tuple):
    color_16bit_tuple = tuple([int(rgb * 65535 / 255) for rgb in
color_8bit_tuple])
    return color_16bit_tuple


def set_the_colour_brightness(color_tuple, brightness):
    colour_with_a_specified_brightness = tuple([int(rgb * brightness /
100) for rgb in color_tuple])
    return colour_with_a_specified_brightness
```

```
def switch_on_a_color(rgb_led, color_tuple):
    for idx, color in enumerate(color_tuple):
        rgb_led[idx].duty_u16(color)


def switch_on_a_colour_of_the_specified_brightness(rgb_led, color_tuple,
brightness=1):
    color_tuple = get_colour_data_for_pwm(color_tuple)
    color_tuple = set_the_colour_brightness(color_tuple, brightness)
    switch_on_a_color(rgb_led, color_tuple)

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

while True:
    for degree in range(0, 361):
        color_tuple = get_colour_data_for_pwm(rgb_wheel(degree))
        print(rgb_wheel(degree))
        switch_on_a_color(rgb_led, color_tuple)
        sleep(0.05)
```