Шестое практическое занятие

Объектно-ориентированное программирование

ООП (Объектно-Ориентированное Программирование) — это парадигма программирования, где программа строится на основе объектов, которые объединяют данные и методы для работы с этими данными. Основные принципы ООП:

1. **Инкапсуляция** — скрытие внутренних деталей объекта и предоставление доступа только через определенные методы.
2. **Наследование** — создание новых классов на основе уже существующих, наследуя их свойства и методы.
3. **Полиморфизм** — возможность использовать один интерфейс для работы с разными типами данных.
4. **Абстракция** — выделение только важных характеристик объекта и игнорирование несущественных деталей.

ООП позволяет писать более структурированный, понятный и легко поддерживаемый код.

Так, несколько занятий назад, мы познакомились с концепцией обобщения кода с помощью функций. Это позволило нам разделить наш код на независимые между собой смысловые именованные блоки, позволяющие избежать повторов, упростить отладку и тестирование.

Создание объектов в ООП является следующим шагом в построении обобщения кода. Оно позволяет сгруппировать в себе не только различные функции, но и конкретные данные, необходимые для их корректной работы.

По своей сути и форме ООП позволяет представить в результате весь код программы не как последовательность объявления переменных, прямых арифметических действий с ними, ветвлений и циклов, а как взаимодействие программных сущностей, отражающих окружающую нас реальность.

ООП это одновременно очень простая в своей концепции, но и очень объемная и сложная в своем наполнении тема, на понимание которой легко могут уйти месяцы (или годы), но поверьте, результат в итоге окупит все потраченные усилия.

Объективная реальность сегодня такова, что без ООП мы будем не способны написать действительно полезный и объемный код и изучение его это неизбежная необходимость!

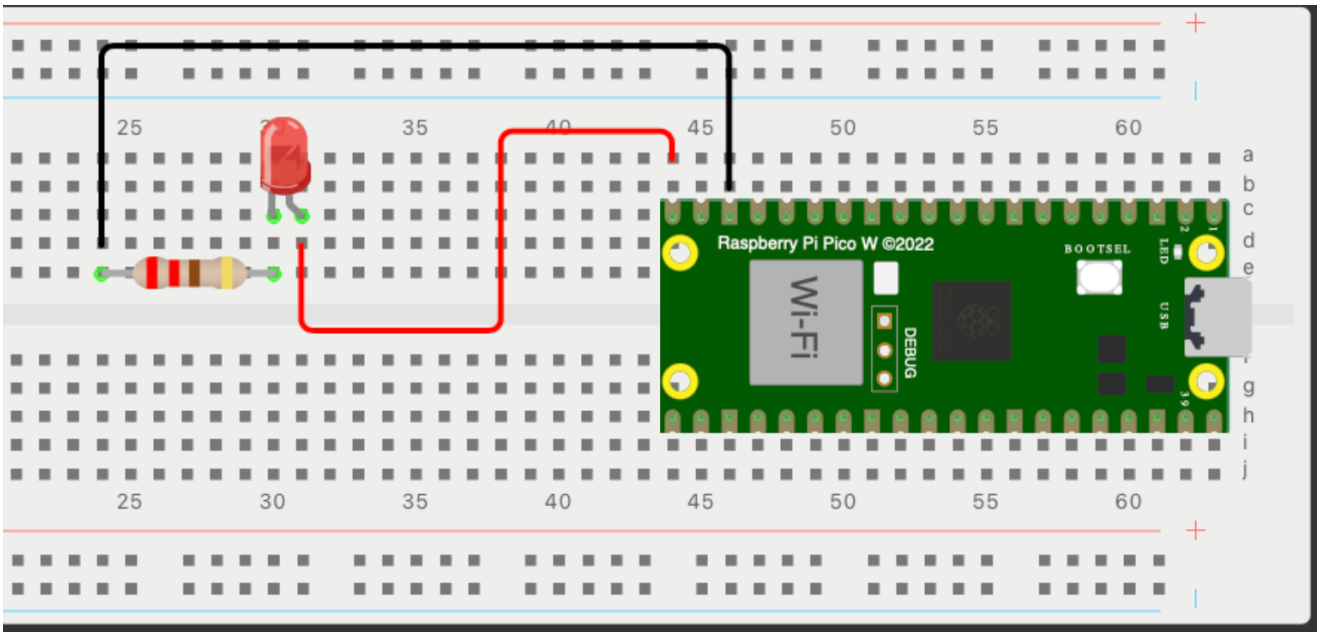
Но не будем удручать себя, пускай и объективными поначалу, сложностями! Как всегда, мы начнем с простого, а дальше все зависит от Вас!

Создание проблемы

Управление светодиодом - что может быть проще? Мы освоили его почти сразу и почти без труда!

Оттолкнемся от этого примера еще раз! Что мы делали?

Сначала подключили светодиод последовательно с резистором к цифровому пину:



Этот этап не относится к непосредственному программированию и, очевидно, не будет меняться, так же как и все условия, описанные в методических указаниях ко второму практическому занятию.

После подключения мы написали код наподобие следующего:

```
from machine import Pin
from time import sleep

led_pin = Pin(15, Pin.OUT)

while True:
    led_pin.on()
    sleep(3)
    led_pin.off()
    sleep(1)
```

После загрузки кода в микроконтроллер подключенный диод стал моргать, загораясь на три секунды и гаснув на одну.

Технически к этому коду сложно придраться - он выполняет без ошибок желаемую нами задачу. Но по форме он очень нестабилен - чтобы работать далее с этой программой пользователь должен полностью понимать контекст решаемой задачи (что мы хотим моргать светодиодом), знать, что такое объект `Pin` (опять же объект! sic!), знать, как его создавать, что нужно настраивать его на выход и как с ним работать далее.

В результате слишком много ответственности и проблем мы оставляем тому, кто будет пользоваться результатами нашего труда... Можно было бы списать это на чужие проблемы, но реальность такова, что, вероятнее всего, этим бедолагой окажемся мы сами, а значит нужно позаботиться о себе!

Предыдущий код можно разделить на три блока:

- импорт библиотек и классов

```
from machine import Pin
from time import sleep
```

- инициализация переменных и объектов

```
led_pin = Pin(15, Pin.OUT)
```

- непосредственный алгоритм выполнения желаемых действий:

```
while True:
    led_pin.on()
    sleep(3)
    led_pin.off()
    sleep(1)
```

В чем проблема, где проблема? Потенциально во всем и везде :(

Вот здесь

```
from machine import Pin
from time import sleep
```

мы импортировали объект `Pin` для подачи напряжения на вывод микроконтроллера, и функцию `sleep` для задержки между подачей и снятием напряжения на него. Проблема не великая, но мы уже смешали код для инициации данных и непосредственной и последующей работы с ними. Пока таких объектов всего два - сложности нет, но что случится когда их станет кратно больше?

Далее:

```
led_pin = Pin(15, Pin.OUT)
```

Мы инициализировали пин нужным нам образом и присвоили сформированный объект переменной `led_pin`.

В чем тут проблема? В том, что это светодиод, можно косвенно понять лишь из названия переменной, а это крайне ненадежная условность.

Замените `led_pin` на `l_p` или просто `l` и ни одна разумная сила во вселенной не сможет гарантировано прозреть всю глубину смыслов Вашей задумки...

Утешим себя тем, что в заключительном блоке особых проблем для нас нет:

```
while True:
    led_pin.on()
    sleep(3)
    led_pin.off()
    sleep(1)
```

Он понятный и рабочий. Но и в нем мы оставили досадную уязвимость!

Пользователь, работающий с нашим кодом, мог бы добиться такого же эффекта написав:

```
while True:
    led_pin.high()
    time.sleep(3)
    led_pin.low()
    time.sleep(1)
```

или

```
while True:
    led_pin.value(1)
    time.sleep(3)
    led_pin.value(0)
    time.sleep(1)
```

А кому потом разбираться с подобным бардаком? Правильно - Вам самим!

Может такая простая задача и не выглядит слишком убедительной, но это всего лишь условность примера.

Перепишем с помощью ООП!

Решение проблемы

ООП определяет смысловой единицей - объект.

Разберемся, что является им в контексте решаемой нами задачи... Правильно светодиод!

Светодиод (LED) - штука довольно понятная и универсальная:

- его нужно подключить и настроить;
- у него есть цвет;
- его можно включать и выключать;
- можно включать его с задаваемой яркостью;
- им можно моргать;
- им можно передавать сигнал с помощью азбуки Морзе;
- и т.д.

Что-то мы размахнулись... Нужно притормозить и подумать, нужно ли нам все это и, что еще важнее, нужно ли нам это все прямо сейчас!

Тут правильный ответ только один - нет!

Первое, с чего необходимо изучать ООП, это с навыка абстрагирования - выделение только существенного и необходимого. А здесь это только инициализация диода и его включение и выключение.

Решим пока это, а дальше будет потом!

Первое, что нужно объекту это понятное, звучное, простое и красивое **имя**!

Обычно жизнь уже придумала его за нас и решаемая нами задача не исключение. Первое, что приходит на ум при выборе простого имени для светодиода это *Led*.

Создадим класс для нашего светодиода:

```
class Led:
    pass
```

Пока кроме имени у нашего класса нет и он ничего не умеет. Отметим пока только то, что:

⚠ Имя класса (по PEP8) должно называться с заглавной буквы. Если имя состоит из нескольких слов, то оно должно быть набрано CamelCase-ом (Слитно каждое новое слово с заглавной буквы).

но мы уже могли бы создать наши объекты:

```
led_1 = Led()
led_2 = Led()
```

при этом `led_1` и `led_2` оба были бы разными *экземплярами* класса `Led`.

Добавим в наш класс блок инициализации:

```
class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)
```

Что мы сделали? Добавили внутрь нашего класса функцию `__init__(self, *args, **kwargs)`

> ⓘ функции, относящиеся к конкретным объектам (реализованные в их классах), принято называть ***методами***

Метод `__init__(self, *args, **kwargs)` является *магическим*, не в смысле, что он чудесный, хотя и это тоже, а в том, что он относится к так называемым "дандер" методам (начинающимся и заканчивающимся двойным нижним подчеркиванием __ - double under).

Такая форма написания является условной для служебных методов, реализованных во всех без исключения объектах, определяющих их стандартное поведение.

Метод `__init__(self, *args, **kwargs)` запускается при один раз при создании объекта и нужен для его инициализации.

ВАЖНО! первым параметром метода `__init__(self, *args, **kwargs)` идет `self` - ссылка на сам создаваемый объект (она будет заполняться объектом неявно самостоятельно), далее определяемые пользователем атрибуты.

> `self` - это пока сложно, воспринимайте его пока просто как то, что вы создаете

В нашем случае:

```
def __init__(self, led_pin):
    self.led = Pin(led_pin, Pin.OUT)
```

мы передали в качестве необходимо для создания объекта `Led` переменную `led_pin` , определяющую номер пина, к которому физически подключен светодиод.

Внутри метода мы создали переменную `led` как атрибут конкретного светодиода (`self.led`) и присвоили в нее созданный объект пина `Pin(led_pin, Pin.OUT)` с соответствующей настройкой (что он работает без ШИМ и на выход).

Отметим, что для работоспособности данного кода, необходимо импортировать класс `Pin` из пакета `machine`.

Сделать это можно несколькими путями:

```
from machine import Pin

class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)
```

или

```
class Led:
    def __init__(self, led_pin):
            from machine import Pin
        self.led = Pin(led_pin, Pin.OUT)
```

В первом случае нужный класс будет импортирован при запуске файла на исполнение, во втором каждый раз при инициализации нового светодиода.

Оба эти варианта имеют свои преимущества и недостатки. Не будем излишне акцентировать на этом свое внимание сейчас.

Сейчас мы настроили логику инициализации наших диодов. Отметим, что после этого наш предыдущий код:

```
led_1 = Led()
led_2 = Led()
```

перестал работать, так как теперь для создания объекта `Led(led_pin)` необходимо указать пин к которому он подключен.

Код:

```
led_1 = Led(15)
led_2 = Led(led_pin=16)
```

уже будет работать, при этом для первый светодиод мы инициализируем на 15 пине, а второй - на 16.

Добавим в наш класс возможность включения и выключения светодиода:

```
from machine import Pin

```

```
class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)

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

    def off(self):
            self.led.off()
```

Добавленные методы `on()` и `off()` работают подобно друг другу: подавая напряжение на пин, инициализированный в атрибуте `led` объекта (внутри `self`).

Отметим, что, так как методы `on()` и `off()` работают с данными конкретного объекта, они так же получают первым параметром сам объект `self`, позволяющий внутри методов оперировать с его атрибутами.

Окончательный результирующий код будет выглядеть:

```
from time import sleep
from machine import Pin

class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)

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

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


led = Led(15)
while True:
    led.on()
    sleep(3)
    led.off()
    sleep(1)
```

Рассмотрим в заключении блок:

```
led = Led(15)
while True:
    led.on()
    sleep(3)
```

```
    led.off()
    sleep(1)
```

именно с ним и будет работать конечный пользователь нашего кода!

Он похож на предыдущий, но теперь пользователь явно создает объект светодиода, а не пина. И теперь у нашего светодиода жестко ограниченный интерфейс - диод можно только включать и выключать (методы `value()`, `high()` и `low()`, которые были у объекта `Pin` теперь нам напрямую не доступны.

Сделаем лучше, сделаем сложнее

Мы добились серьезного, но мало впечатляющего эффекта. Пока это просто рефакторинг кода - мы сделали его лучше и чище, но не внесли в него нового поведения.

Исправим это понемногу!

Для начала заменим код:

```
from machine import Pin

class Led:
    def __init__(self, led_pin):
        self.led = Pin(led_pin, Pin.OUT)

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

    def off(self):
        self.led.off()
```

на:

```
from machine import Pin, PWM

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

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

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

Что мы сделали? Заменили работу нашего светодиода с цифрового сигнала (только включение и выключение) на ШИМ сигнал. Теперь мы потенциально сможем управлять его яркостью!

Отдельно отметим, что блок выполняемого ранее кода:

```
led = Led(15)
while True:
    led.on()
    sleep(3)
    led.off()
    sleep(1)
```

не пострадает от изменения структуры и логики нашего объекта, так как мы сохранили интерфейс создания светодиода (нам также необходим всего один параметр - номер пина `led_pin`) и имена методов `on()` и `off()` вместе с их сигнатурой.

Фактически конечный пользователь и не заметит никаких изменений, хотя весь код внутри нашего класса и изменился.

Раньше нам пришлось бы изменять весь исполняемый код вместе в инициализирующим, а теперь мы их разделили и это серьезно развязало нам руки!

Добавим в наш класс новый функционал:

```
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)

    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)
```

Теперь наш светодиод может зажигаться с заданной яркостью, масштабируемой в пределах от `0` до значения переменной `BRIGHTNESS_SCALE`(ее мы рассмотрим отдельно далее).

Метод `turn_to_brightness(self, brightness)`, также как и методы `on()` и `off()` работает с атрибутами конкретного объекта, а значит ему необходимо получить внутрь себя сам объект `self`, но в отличие от них он должен получить дополнительно еще и значение, определяющее требуемую яркость `brightness`.

```
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)
```

Что мы делаем внутри этого метода? В основном проверяем находится ли переданное значение яркости `brightness` в пределах от `0` до `BRIGHTNESS_SCALE`, далее рассчитывается соответствующее значение ШИМ:

```
duty = int(65535 / self.BRIGHTNESS_SCALE * brightness)
```

которое передается на конкретный пин, определенный при инициализации светодиода в переменную `self.led`:

```
self.led.duty_u16(duty)
```

Зачем столько сложностей и перестраховок? Затем, что мы пишем код не для себя настоящих, а для себя будущих (или вообще для посторонних людей). У нас нет гарантий, что они понимают ту логику, которой мы руководствуемся в данный момент, и поэтому мы должны максимально защитить наш код от ошибок извне.

Но это долгий разговор допускающий дискуссионность вопроса.

Вернемся к атрибуту (переменной) `BRIGHTNESS_SCALE` . В отличие от атрибута `self.led` (являющегося "личным" свойством конкретного светодиода (экземпляра класса `Led`)) атрибут `BRIGHTNESS_SCALE` является общим для всех экземпляров класса `Led`, что позволяет обобщить настройку всех светодиодов между собой.

> ⓘ Сейчас это может показаться хорошей идеей, будем надеяться, что так останется и дальше

> Переменные относящиеся к классу, а не конкретному объекту (как `BRIGHTNESS_SCALE`) называются **статическими переменными**.

Доступ к таким переменным можно получить как через имя переменной конкретного экземпляра класса:

```
led = Led(15)
bs = led.BRIGHTNESS_SCALE
```

через сам класс:

```
bs = Led.BRIGHTNESS_SCALE
```

внутри класса мы обращались к переменной через сам экземпляр объекта (`self`):

```
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)
```

Хотя могли бы и через имя класса:

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

> ⓘ Отметим, что второй вариант менее удачен, так как если мы вдруг захотим изменить имя класса `Led` на что-нибудь другое, нам придется явно указывать его и внутри кода...

Но проверим Наш код! Запустим

```
led = Led(15)
while True:
    led.turn_to_brightness(100)
    sleep(1)
    led.turn_to_brightness(75)
    sleep(1)
```

```
    led.turn_to_brightness(50)
    sleep(1)
    led.turn_to_brightness(25)
    sleep(1)
    led.turn_to_brightness(0)
    sleep(1)
```

и увидим ступенчатое затухание диода!

Усложним наш объект еще!

Ранее мы использовали ШИМ для создания различных эффектов - плавного зажигания и потухания.

Попробуем реализовать это наивным способом:

```
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)

    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)

        def smooth_switching_on(self):
        for duty in range(65535):
            self.led.duty_u16(duty)
            sleep(0.0001)

    def smooth_switching_off(self):
        for duty in range(65535, -1, -1):
            self.led.duty_u16(duty)
            sleep(0.0001)
```

Почему наивным? Сохраним пока эту интригу!

Проверим, работает ли наш код:

```
led = Led(15)
while True:
    led.smooth_switching_on()
    sleep(3)
    led.off()
    sleep(1)
    led.on()
    sleep(3)
    led.smooth_switching_off()
    sleep(1)
```

Какая красота! Где же может нас подстерегать опасность? Наверняка нигде!

Но если все идет так хорошо, и мы так часто моргаем нашим диодом почему не сделать для этого специальный метод? Окееееей....

```
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)

    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)

        def smooth_switching_on(self):
        for duty in range(65535):
            self.led.duty_u16(duty)
            sleep(0.0001)

    def smooth_switching_off(self):
```

```
        for duty in range(65535, -1, -1):
            self.led.duty_u16(duty)
            sleep(0.0001)

        def blink(self, sleep_time_in_sec):
        self.on()
        sleep(sleep_time_in_sec)
        self.off()
        sleep(sleep_time_in_sec)
```

Смотрите, как изящно мы включаем и выключаем светодиод через уже проверенные нами методы `on()` и `off()` нашего класса...

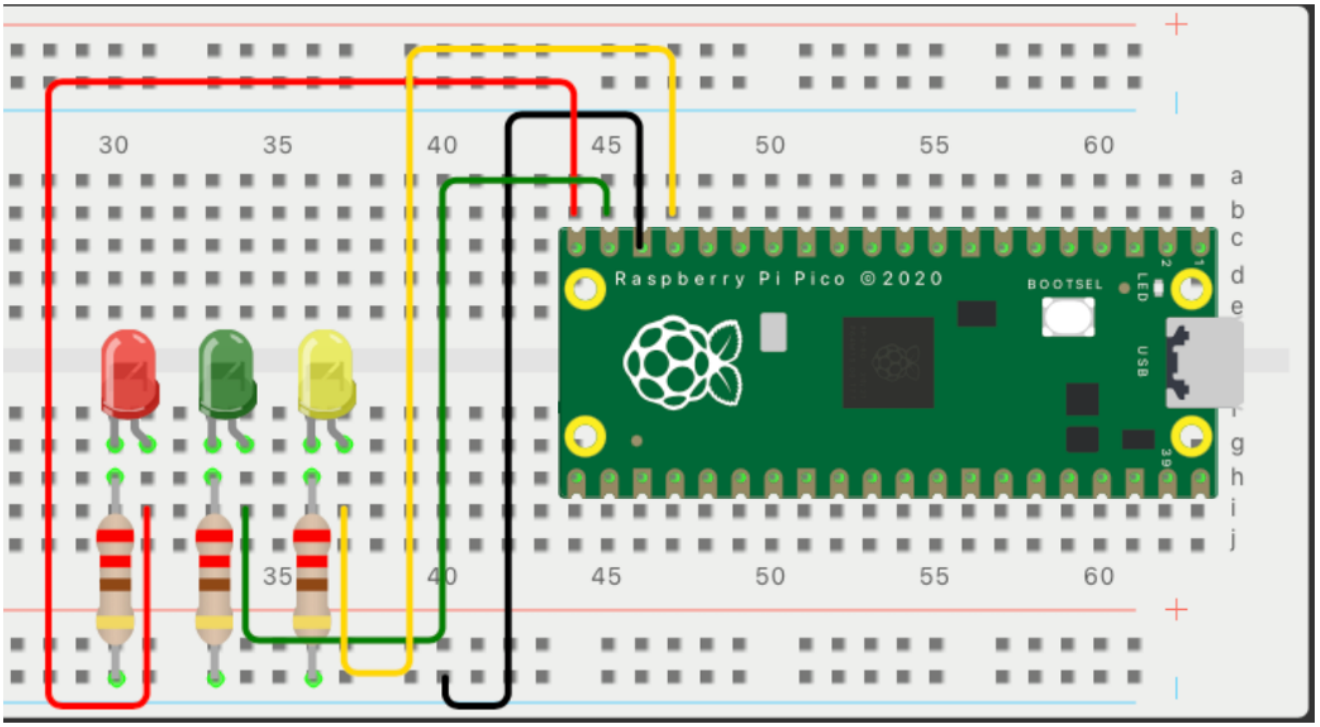
Проверим, будет ли он работать:

```
led = Led(15)
while True:
    led.blink(2)
```

Вроде будет.... Все же хорошо? Правда?!?! Ответ мы увидим, когда попробуем создать для нашего диода пару...

Неожиданная проблема и ее решение в стиле ООП

Попробуем подключить несколько светодиодов одновременно и начать управлять ими через наш класс `Led`:




```
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)

    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)

        def smooth_switching_on(self):
        for duty in range(65535):
            self.led.duty_u16(duty)
            sleep(0.0001)

    def smooth_switching_off(self):
        for duty in range(65535, -1, -1):
            self.led.duty_u16(duty)
            sleep(0.0001)

        def blink(self, sleep_time_in_sec):
        self.on()
        sleep(sleep_time_in_sec)
        self.off()
        sleep(sleep_time_in_sec)

red_led = Led(15)
green_led = Led(14)
yellow_led = Led(13)

while True:
    red_led.blink(1)
    green_led.blink(2)
    yellow_led.blink(0.5)
```

Что мы ожидаем?

Что каждый диод будет моргать с разной частотой! Отчасти это так, но как-то криво...

Чего бы ожидал пользователь с таким кодом?

```
red_led = Led(15)
green_led = Led(14)
yellow_led = Led(13)

while True:
    red_led.blink(1)
    green_led.blink(2)
    yellow_led.blink(0.5)
```

Что красный диод будет моргать с интервалом в 1 секунду, зеленый в две, а желтый в 0.5 секунд.

А что он получает? Очередь - сначала отморгается красный, потом зеленый, в конце - желтый.

Это никуда не годится!

Но в чем проблема?

В коде мы использовали функцию `sleep()`, которая определяет задержку всей работы микроконтроллера.:

```
def blink(self, sleep_time_in_sec):
    self.on()
    sleep(sleep_time_in_sec)
    self.off()
    sleep(sleep_time_in_sec)
```

В результате попав в метод `blink` первого светодиода, остальные вынуждены ждать пока он неторопливо закончит.

Очень нерационально!

Без ООП нам бы пришлось не сладко... С ним правда тоже, но все могло бы быть намного хуже!

Прежде чем идти дальше отметим, что мы использовали задержку `sleep()` в ранее написанных методах `smooth_switching_от()` и `smooth_switching_off()`. Значит, проблема будет и в них... Уничтожим их пока, ведь наше моргание под угрозой и имеет сейчас высший приоритет!

Как нам решить возникшую проблему? Необходимо устранить явную задержку! Но нужно же чем-то ее заменить? Чем-то нужно, но ждать нет времени! Давайте тогда попробуем так....

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

class Led:

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

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

Что мы сделали?

Во-первых, добавили каждому светодиоду по два новых атрибута:

- `self.timer` - определяющий время последнего изменения состояния;
- `self.state` - определяющий текущий статус светодиода.

Исходя их этого понятно, что:

```
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"
```

пришлось изменить, добавив в эти методы изменение статуса светодиода.

ⓘ функция `ticks_ms()` возвращает текущее количество милисекнуд (тысячных долей секунды) прошедших с момента подачи питания на микроконтроллер

Далее в методе:

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

мы проверяем, превысило ли время с последнего изменения состояния светодиода требуемое время задержки между морганием `blink_time_in_sec`, и в случае выполнения условия:

```
if (ticks_ms() - self.timer) / 1000 >= blink_time_in_sec:
```

выполняем смену состояния.

Сложно? Если да, то попробуйте реализовать все это без классов, отдельно для каждого светодиода... А если их много? А если их *ОЧЕНЬ* много?

Так мы реализовали логику для *концепции светодиода* в классе `Led`, который будет общим для всех его реализаций, а наш пользователь даже не поймет, каких трудов нам это стоило! *(Очень благородно с нашей стороны:))*

Но сначала нужно проверить, что все работает. Запустим код:

```
red_led = Led(15)
green_led = Led(14)
yellow_led = Led(13)

while True:
    red_led.blink(1)
    green_led.blink(2)
    yellow_led.blink(0.5)
```

Теперь все стало намного лучше.

Расплата по счетам

Пока нам кажется, что все опять хорошо...

Но пользователь ранее выпущенного нами кода:

```
led = Led(15)
while True:
    led.smooth_switching_on()
    sleep(3)
    led.off()
    sleep(1)
    led.on()
    sleep(3)
    led.smooth_switching_off()
    sleep(1)
```

с нами в корне не согласен!

Раньше он радовался плавному зажиганию светодиода, а теперь он не радуется, так как в приступе паники мы удалили методы `smooth_switching_on()` и `smooth_switching_off()` из реализации нашего класса `Led`.

Простым путем было бы просто вставить ранее удаленный код и вернуть тем самым пропавший функционал, но простые пути не для нас!

К тому же это было бы отложенной в будущее проблемой, так как задержки в методах:

```
def smooth_switching_on(self):
    for duty in range(65535):
        self.led.duty_u16(duty)
        sleep(0.0001)

def smooth_switching_off(self):
    for duty in range(65535, -1, -1):
        self.led.duty_u16(duty)
        sleep(0.0001)
```

будут прерывать всю остальную работу, до тех пор пока цикл не закончит работу.

Только представьте, если кто-то захочет использовать наш крутой класс для сигнальной панели ракеты, летящей к Марсу.

Наверное в космосе такой спешки и не было бы, только до космоса ракета не долетела бы с таким отношением....

Совесть требует все сделать нормально....

```
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()
```

Кстати, тут мы заодно вспомним о том бедолаге, который раньше управлял яркостью светодиода:

```
led = Led(15)
while True:
    led.turn_to_brightness(100)
    sleep(1)
    led.turn_to_brightness(75)
    sleep(1)
    led.turn_to_brightness(50)
    sleep(1)
    led.turn_to_brightness(25)
    sleep(1)
    led.turn_to_brightness(0)
    sleep(1)
```

Но он, наверное, жаловался не так громко, что вспомнили мы о нем только параллельно с восстановлением другой функциональности. Хотя конкретно он пострадал вообще ни за что, ведь метод:

```
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
```

не содержал в себе задержек и ничем не мешал работе.

Но вернемся к нашим изменениям и разберем их!

В наш объект мы добавили новый атрибут, определяющий текущее состояние яркости диода:

```
self.brightness = 0
```

Разберем же методы "разгорания" и потухания светодиода:

```
    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)
```

Главное - нам пришлось избавиться от циклов внутри методов, а прямые задержки заменить проверкой выполнения условия по времени:

```
if (ticks_ms() - self.timer) / 1000 >= sleep_time
```

аналогично морганию диода.

Но, как всегда, есть нюанс. Код:

```
led = Led(15)
while True:
    led.smooth_switching_on()
    sleep(3)
    led.off()
    sleep(1)
    led.on()
    sleep(3)
    led.smooth_switching_off()
    sleep(1)
```

Постоянно повышал бы яркость на один пункт, потом полностью выключал диод, потом включал, потом понижал. Короче работал бы плохо

Отсюда нам пришлось увеличить количество возможных состояний светодиода как:

- "DOWN" - определяющий то, что диод идет в сторону потухания;
- "UP" - в сторону "разгорания";
- "STOP" - и никуда не идет.

По сути весь прочий код внутри этих методов проверяет состояние светодиода и реагирует на то, в какую сторону он должен работать.

Вроде мы все решили, но у пользователя:

```
led = Led(15)
while True:
    led.smooth_switching_on()
    sleep(3)
    led.off()
    sleep(1)
    led.on()
    sleep(3)
    led.smooth_switching_off()
    sleep(1)
```

останется какое-то трудно формулируемое чувство непонятного недовольства.

Ведь раньше время "разгорания" и потухания светодиода к которому он привык было порядка 6.5 секунд:

```
def smooth_switching_on(self):
    for duty in range(65535):
        self.led.duty_u16(duty)
        sleep(0.0001)
```

а теперь мы указали в методах:

```
    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)
```

время переключения в 1 секунду.

> В этом-то и заключается реальная сложность ООП. Это не просто синтаксис атрибутов и методов - это процесс создания и поддержания объектов в рабочем состоянии, их развитие... Допущенные при проектировании объектов ошибки исправить потом будет очень сложно! Но ничего не поделать понимание этого придет только с опытом! У Вас все получится!