introduccion.py clase07.py

```
Programming ‘Language’ = {
    Introducción = [Python, MicroPython]

}
```

introduccion.py clase07.py

```
Clase 07 = {
    Presentación = [Les damos la bienvenida al curso]
}
```

Clases = {

- **Clase 05** — Conceptos Básicos de POO. Clases y objetos. Métodos y atributos.
- **Clase 06** — Instalación de MicroPython en la placa ESP32. Introducción a la herramientas de desarrollo Thonny. Conexión y configuración de la placa.
- 👉 **Clase 07** — **Control de Hardware Básico. Manejo de pines GPIO. Lectura de sensores y actuadores.**
- **Clase 08** — Comunicación Serial. UART, I2C, SPI. Comunicación entre dispositivos.

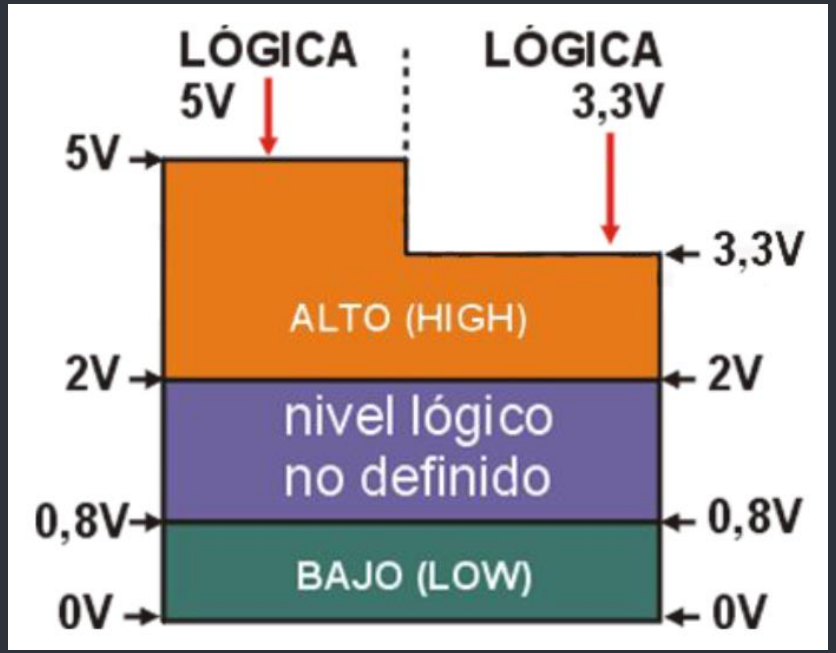
Pines digitales {

```
    # Los pines digitales en el ESP32 son pines de entrada/salida (GPIO, por
    sus siglas en inglés: General Purpose Input/Output) que pueden ser
    configurados para leer o escribir señales digitales. Estos pines son
    fundamentales para interactuar con otros componentes electrónicos, como
    sensores, actuadores, LEDs, botones, y más.

    # Estos pines pueden ser configurados independientemente como entrada o
    como salida. Como entradas van a leer información de un botón, un sensor,
    etc. Como salida van a entregar información para configurar un
    dispositivo, controlar un actuador, etc. También van a poder tener su
    resistencia de Pull-Up o Pull-Down independientemente.

    # Los niveles de tensión de los pines digitales pueden ser 0V (nivel bajo)
    o 3.3V (nivel alto).

    # Algunos pines tienen funciones especiales, como ADC (convertidor
    analógico a digital), DAC (convertidor digital a analógico), PWM
    (modulación por ancho de pulso), I2C, SPI, UART, etc.
}
```

Pines digitales {

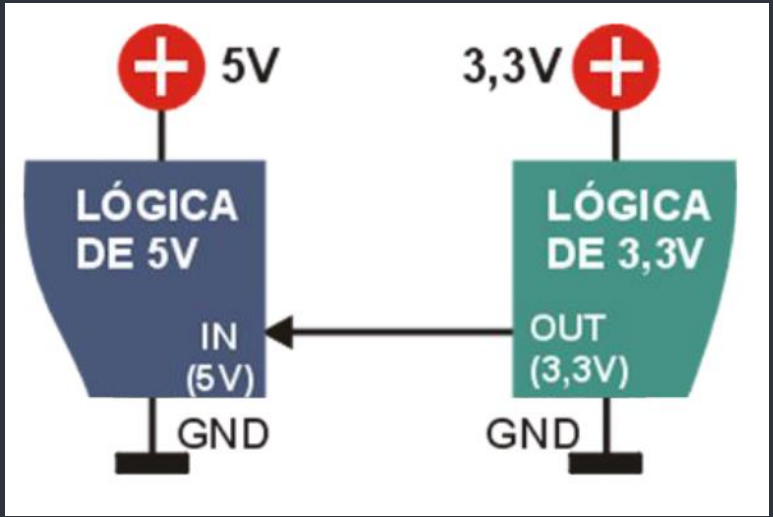
```
    # Los pines digitales van a poder imponer o leer estados. Estos estados pueden ser altos o bajos.

    # Los estados altos (alto, True, 1, on, high) van a ser representados por tensiones de 3.3v

    # Los estados bajos (bajo, False, 0, off, low) van a ser representados por tensiones de 0v
```

}



Adaptación {

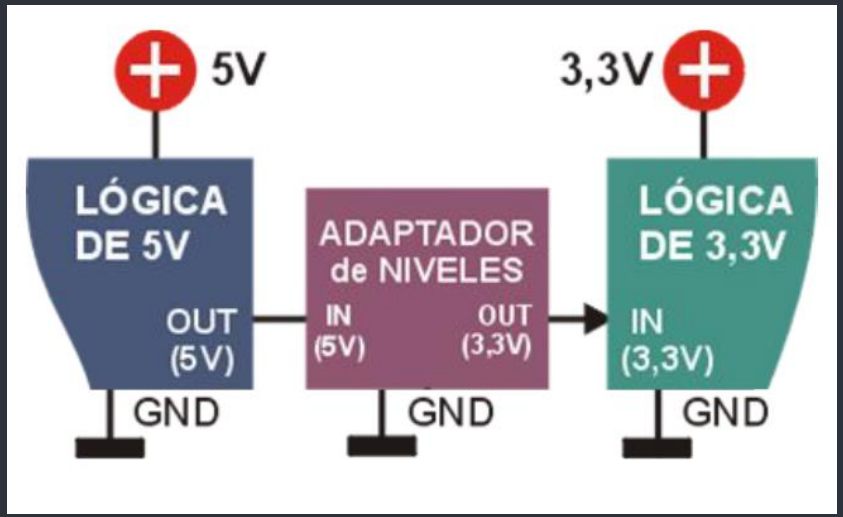
```
# Si vamos a interactuar con un dispositivo que tiene que ser alimentado
con 5v, podemos hacerlo directamente sin adaptar, si en nuestro ESP32 los
pines son de salida.
```



}

Adaptación {

```
# Si vamos a interactuar con un dispositivo que tiene que ser alimentado
con 5v, tenemos que adaptar los niveles, si en nuestro ESP32 los pines son
de entrada.
```



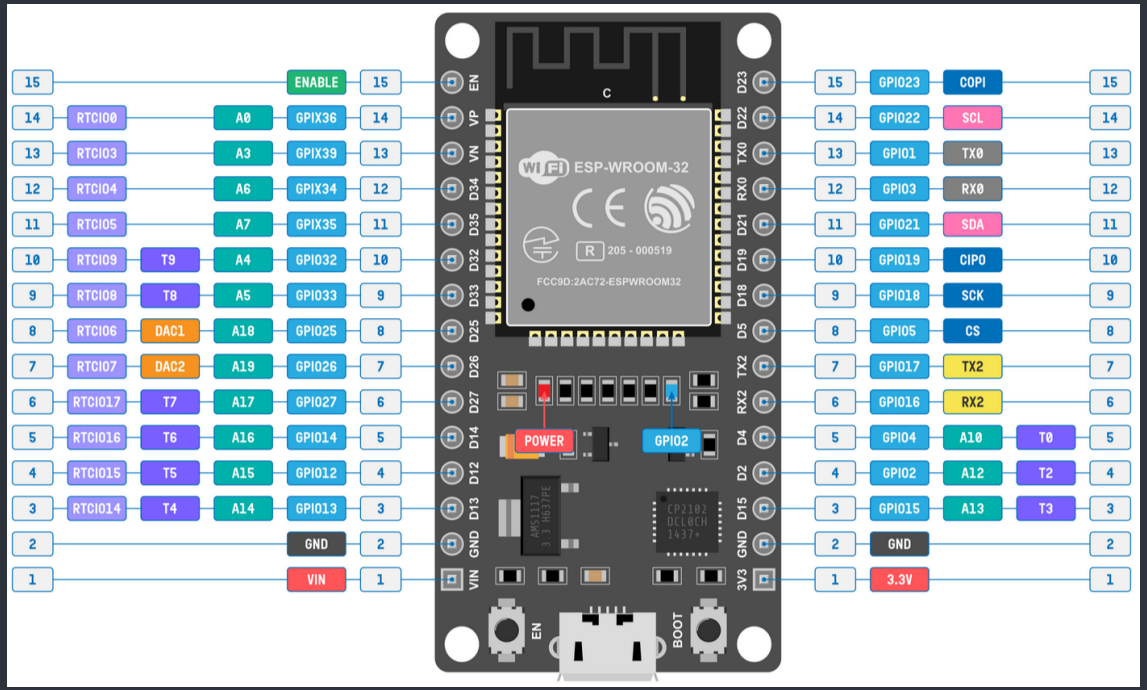
}

Biblioteca Machine {

```
    # Para acceder a utilizar los pines GPIO, es importante hacerlo a través de
    la biblioteca Machine.

    # Machine en MicroPython es una biblioteca fundamental que proporciona
    acceso a las funcionalidades de hardware del microcontrolador. Esta
    biblioteca permite interactuar con los pines GPIO, ADC, DAC, PWM, I2C, SPI,
    UART, y otros periféricos del microcontrolador. Es esencial para
    desarrollar aplicaciones embebidas que interactúan con el hardware.

    # Si la utilizamos erróneamente, podemos configurar mal los pines o sus
    funciones, provocar mal funcionamiento del circuito, bloquear el
    funcionamiento de la placa o en algunos casos dañar la placa (por ejemplo,
    interconectar dos pines configurados uno como entrada y otro como salida,
    sin tener configuradas alguna resistencia interna de pull-up o pull-down).
```

}

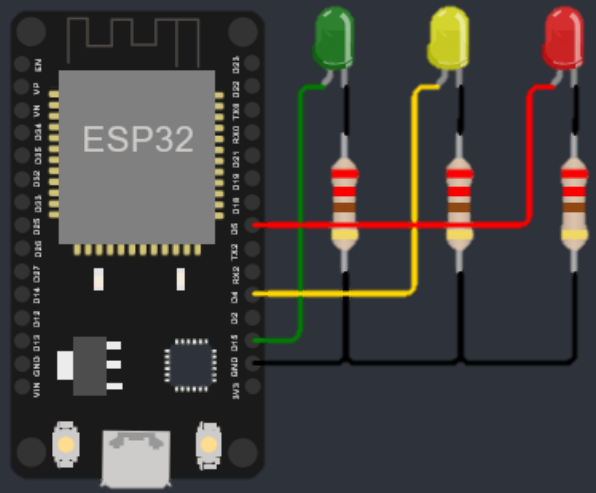
ESP32 DevKit V1 Pinout {



}

Ejemplo 1 – Salidas digitales {

```
    # -- Led Verde -- #
    verde.on()
    sleep(0.5)
    verde.off()
    sleep(0.5)

    # -- Led Amarillo -- #
    amarillo.value(1)
    sleep(0.5)
    amarillo.value(0)
    sleep(0.5)

    # -- Led Rojo -- #
    rojo.value(True)
    sleep(0.5)
    rojo.value(False)
    sleep(0.5)
}
```



Entradas digitales {

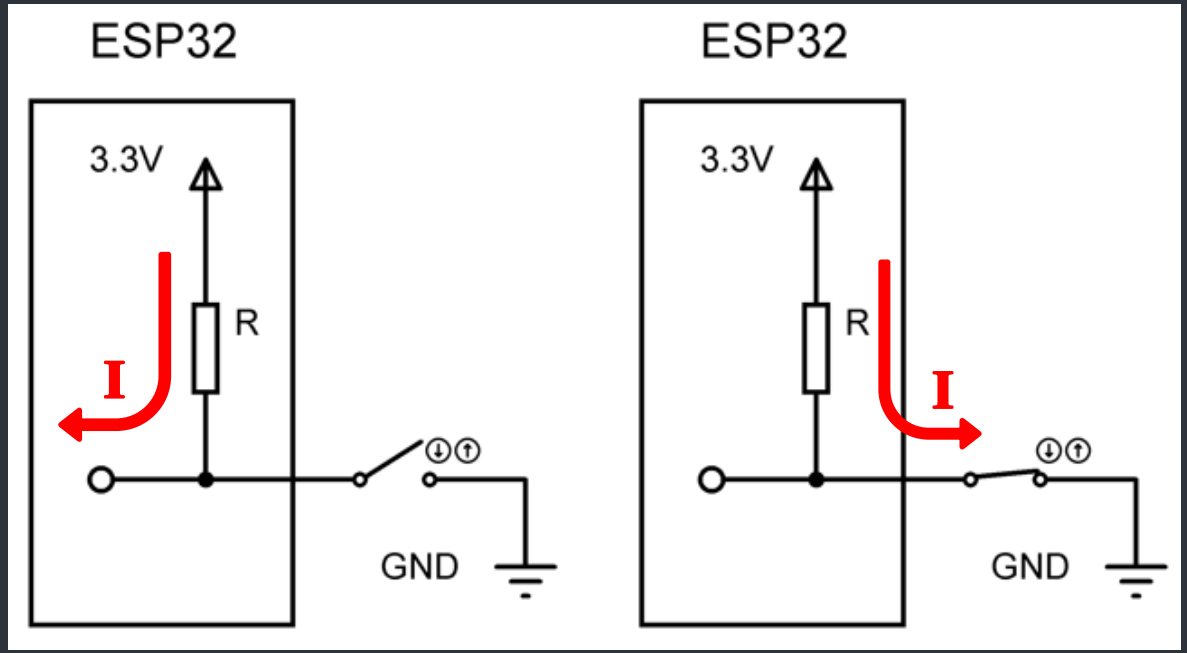
```
    # Las entradas digitales en un ESP32 son pines GPIO (General Purpose
    Input/Output) configurados para leer señales digitales. Estos pines pueden
    detectar dos estados: alto (HIGH) y bajo (LOW), que corresponden a niveles
    de voltaje específicos. En el caso del ESP32, un nivel alto generalmente
    corresponde a un valor entre 3V 3.3V y un nivel bajo a 0V a 0.8V

    # Los pines de entrada pueden ser configurados con resistencias pull-up o
    pull-down internas para evitar estados flotantes (indeterminados).


            # Configurar el pin GPIO 0 como entrada (Botón)

            boton = Pin(0,Pin.IN) #instancia entrada digital
            boton = Pin(0,Pin.IN,Pin.PULL_UP) #pull-up interno
            boton = Pin(0,Pin.IN,Pin.PULL_DOWN) #pull-down interno
}
```

Resistencia de Pull-Up {

```
    # Una resistencia de Pull-Up es una resistencia conectada entre un pin de
    entrada digital y el voltaje de alimentación (Vcc). Su propósito es
    asegurar que el pin tenga un estado predefinido (alto) cuando no hay
    ninguna señal activa conectada al pin. Esto evita que el pin quede en un
    estado flotante, lo que podría causar lecturas erráticas o indeterminadas.

    # El uso de resistencia de Pull-Up también aumenta la compatibilidad con
    dispositivos o periféricos externos, ya que muchos de ellos necesitan
    obligatoriamente tener conectada una.

    # El ESP32 tiene resistencias de pull-up internas que se pueden habilitar
    mediante software. Esto elimina la necesidad de agregar resistencias
    externas en muchos casos. Las resistencias de pull-up internas se pueden
    habilitar al configurar un pin de entrada digital.
```

}

Resistencia de Pull-Up {



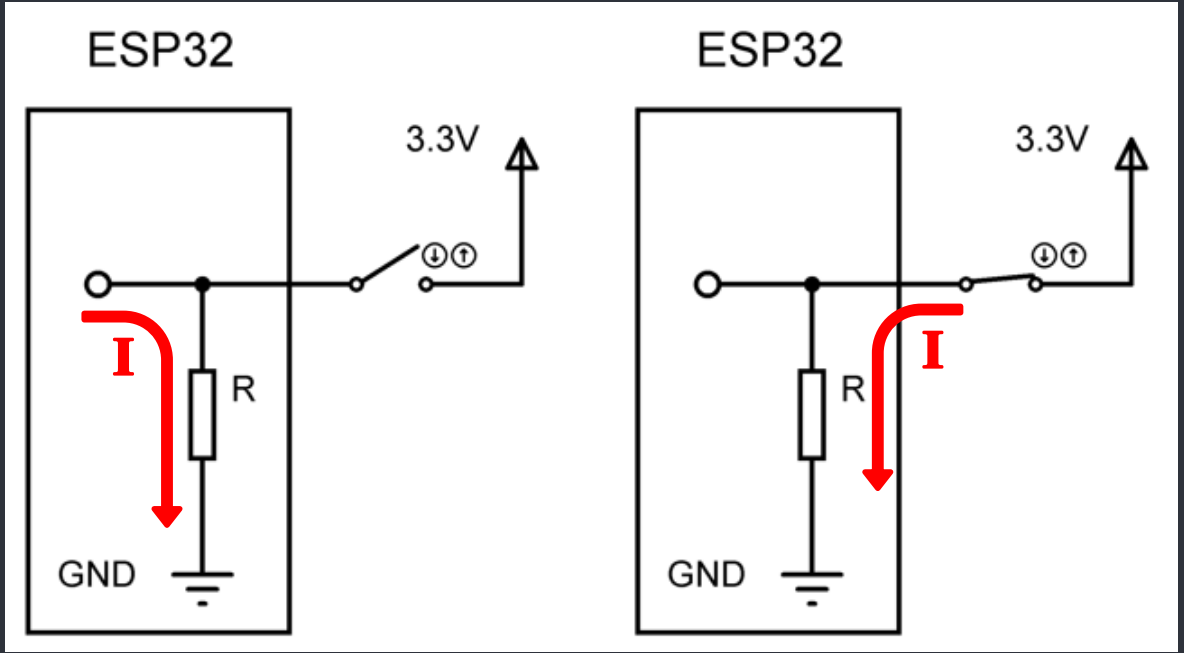
}

Resistencia de Pull-Down {

```
    # Una resistencia de Pull-Down es una resistencia conectada entre un pin de
    entrada digital y tierra (GND). Su propósito es asegurar que el pin tenga
    un estado predefinido (bajo) cuando no hay ninguna señal activa conectada
    al pin. Esto evita que el pin quede en un estado flotante, lo que podría
    causar lecturas erráticas o indeterminadas.

    # El uso de resistencia de Pull-Down también aumenta la compatibilidad con
    dispositivos o periféricos externos, ya que muchos de ellos necesitan
    obligatoriamente tener conectada una.

    # El ESP32 tiene resistencias de Pull-Down internas que se pueden habilitar
    mediante software. Esto elimina la necesidad de agregar resistencias
    externas en muchos casos. Las resistencias de Pull-Down internas se pueden
    habilitar al configurar un pin de entrada digital.
```

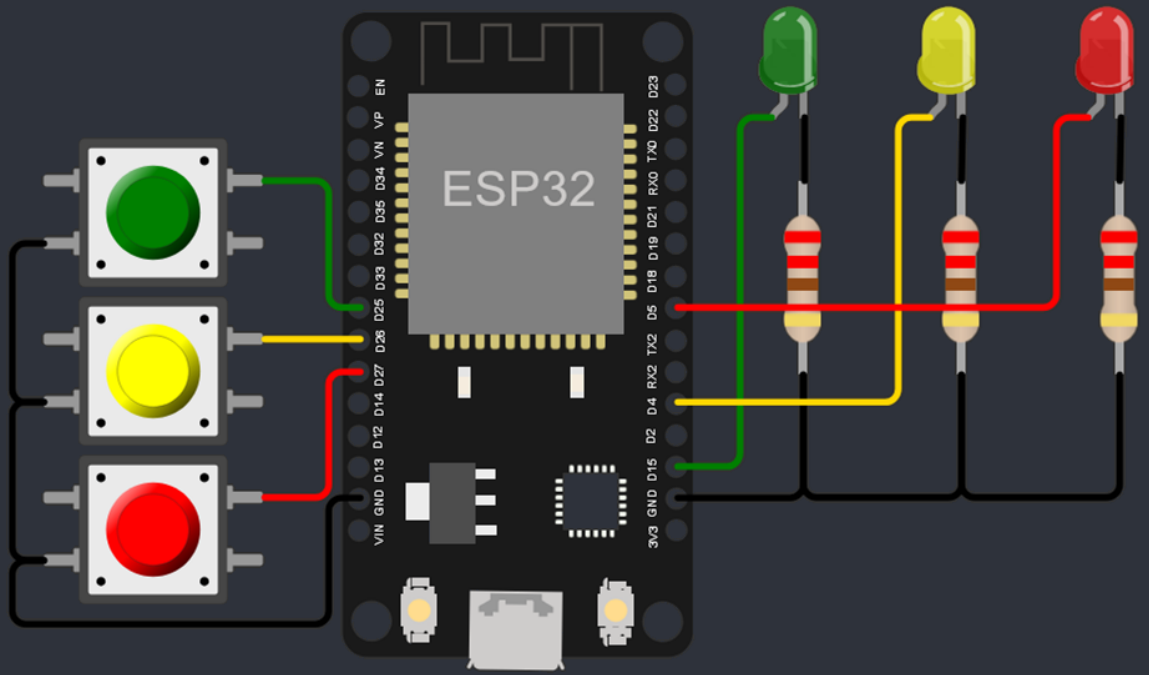
}

Resistencia de Pull-Down {



}

Ejemplo 2 – Entradas digitales {



}

Ejemplo 2 – Entradas digitales {

```
    # Configurar los pines de los leds como entradas con resistencias de pull-up
    boton_verde = Pin(25, Pin.IN, Pin.PULL_UP)
    boton_amarillo = Pin(26, Pin.IN, Pin.PULL_UP)
    boton_rojo = Pin(27, Pin.IN, Pin.PULL_UP)

    while True:
        if boton_verde.value() == 0:  # Si el botón está presionado
            led_verde.on()
            sleep(0.1)
            led_verde.off()
        if boton_amarillo.value() == 0:  # Si el botón está presionado
            led_amarillo.on()
            sleep(0.1)
            led_amarillo.off()
        if boton_rojo.value() == 0:  # Si el botón está presionado
            led_rojo.on()
            sleep(0.1)
            led_rojo.off()
```

}

Entradas analógicas {

```
# Las entradas analógicas en un ESP32 son pines especiales que permiten
medir señales de voltaje continuo (analógicas), es decir, voltajes que
pueden tener cualquier valor dentro de un rango, en lugar de solo ser 0 o 1
como en las señales digitales. Estas señales analógicas se convierten a un
valor digital a través de un convertidor Analógico a Digital o ADC (Analog
to Digital Converter) integrado en el microcontrolador.

# El ESP32 convierte este voltaje en un número digital utilizando un ADC
para que puedas trabajar con él en tu código. Esta conversión es necesaria
porque el microcontrolador, internamente, solo puede trabajar con valores
digitales (números discretos).

# En la mayoría de las versiones del ESP32, las entradas analógicas están
etiquetadas como ADC1 y ADC2.
Los pines específicos pueden variar según la placa, pero generalmente están
en el rango de GPIO32 a GPIO39 para ADC1 y GPIO0 a GPIO27 para ADC2.
```

}

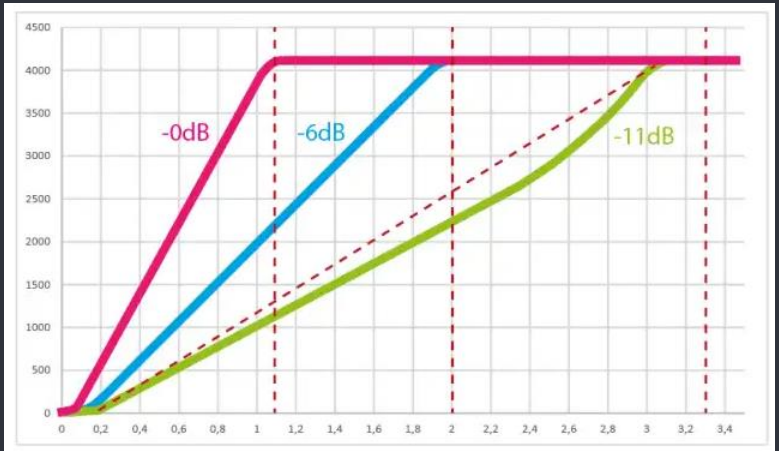
Entradas analógicas {

```
# Las entradas analógicas del ESP32 están generalmente diseñadas para leer voltajes entre 0V y 3.3V (máximo de 3.6V).

# Si el voltaje en una entrada analógica supera los 3.3V, el adc no lo va a registrar y existe el riesgo de dañar la entrada del microcontrolador, por lo que es importante asegurarse de que la señal esté dentro del rango permitido.

# El ESP32 utiliza un ADC de 12 bits (en la mayoría de los casos), lo que significa que puede convertir un voltaje analógico a un valor digital en un rango de 0 a 4095. Aquí se detalla:

                    0 representa 0V
                4095 representa el máximo de 3.3V

# Por lo tanto, la resolución de la medida sería aproximadamente 3.3V / 4096 ≈ 0.0008V por cada unidad del ADC.
```

}

Entradas analógicas {

```
# Muy importante, el ADC2 es usado por el módulo de Wi-Fi, por lo que no
podremos usar los pines del ADC2 pines cuando el Wi-Fi está habilitado. Si
su proyecto requiere Wi-Fi, utiliza únicamente los pines del ADC1.

# Comportamiento del ADC según la atenuación
```



}

Entradas analógicas {

```
    # Lo cierto es que el conversor analógico digital del ESP32 no es de los
    mejores, pero nos sirve para tener una idea. Podemos calibrar el ADC
    utilizando valores de referencia para conocer la desviación especifica del
    nuestro. Hay que relativizar la importancia de la falta de precisión del
    ESP32, en función de la que realmente necesites en tu proyecto. Si necesita
    muy buena precisión, no deberías usar ADC internos sino uno dedicado.

    # Esto sucede porque el ESP32 compara con un valor Vref interno de 1.1V.
    Sin embargo, las placas salen con distintas variaciones que hacen que ese
    1.1V sea realmente 1.0 a 1.2V. A partir de 2019/2020, los ESP32 salen
    precalibrados de fábrica y el código se cambió para añadir una corrección
    por software. Por lo que el comportamiento es bastante mejor.

    # En el caso del ESP32-S3 la cosa es diferente. Este modelo incluye un chip
    de calibrado por hardware interno, por lo que la respuesta del ESP32-S3 es
    muy buena.
}
```

Ejemplo 4 – Potenciometro {

```
    adc_pin = ADC(Pin(13))
    adc_pin.width(ADC.WIDTH_12BIT)
    adc_pin.atten(ADC.ATTN_11DB)

    resolucion_adc = 4095
    tension_referencia = 3.3

    while True:

        adc_valor = adc_pin.read()

        tension = (adc_valor / resolucion_adc) * tension_referencia

        print('Valor ADC:', adc_valor, '- Tension:', tension, 'V')
}
```

PWM {

```
    # PWM (Pulse Width Modulation) es una técnica utilizada para simular una
    señal analógica utilizando una señal digital. En el contexto de un
    microcontrolador como el ESP32, PWM se utiliza para controlar dispositivos
    como motores, LEDs, y otros actuadores variando el ciclo de trabajo (duty
    cycle) de la señal digital.

    # El PWM funciona modulando la duración del pulso (tiempo en que la señal
    está en estado alto) dentro de un período fijo. Esto se conoce como el
    ciclo de trabajo o duty cycle, que se mide en porcentaje:

    # Un duty cycle del 0% significa que la señal está siempre en estado bajo
    (apagado).
    # Un duty cycle del 50% significa que la señal está en estado alto la mitad
    del tiempo y en estado bajo la otra mitad.
    # Un duty cycle del 100% significa que la señal está siempre en estado alto
    (encendido).
```

}

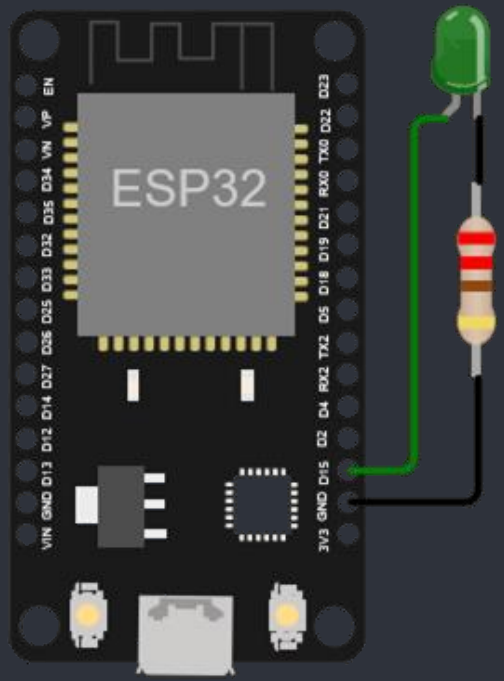
PWM {

```
    # El PWM se caracteriza por dos parámetros principales:

    # Frecuencia: Número de ciclos completos (de encendido y apagado) que ocurren en un segundo. Se mide en Hertz (Hz). Por ejemplo, 500 Hz significa 500 ciclos por segundo.

    # Ciclo de trabajo (duty cycle): Porcentaje del tiempo en que la señal permanece en estado alto en cada período.

    # El ESP32 tiene múltiples canales PWM, lo que te permite generar varias señales PWM simultáneamente en diferentes pines. En MicroPython, puedes controlar el PWM utilizando la clase PWM del módulo machine.

    # Para ver un ejemplo grafico del funcionamiento, podemos ingresar en el siguiente link

}   # https://www.desmos.com/calculator/wsspteh4rc?lang=es
```

Ejemplo 5 – PWM {

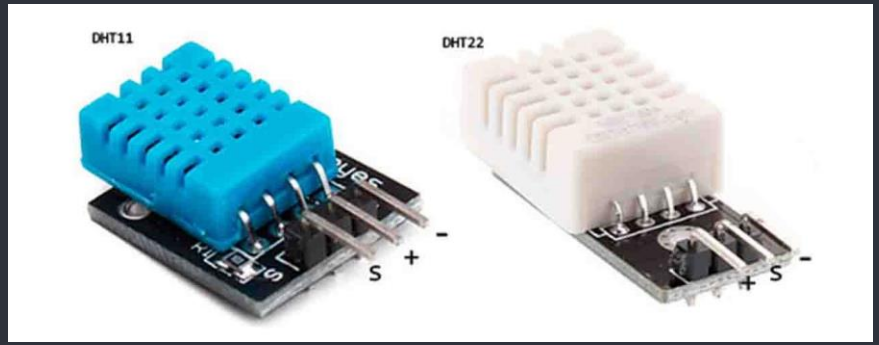
```python
from machine import Pin, PWM
import time

led = PWM(Pin(15))
led.freq(1000)

while True:
    for ciclo in range(0, 1024, 10):
        led.duty(ciclo)
        print(led.duty())
        time.sleep(0.01)
    for ciclo in range(1023, 0, -10):
        led.duty(ciclo)
        print(led.duty())
        time.sleep(0.01)
```

}



Sensores DHT {

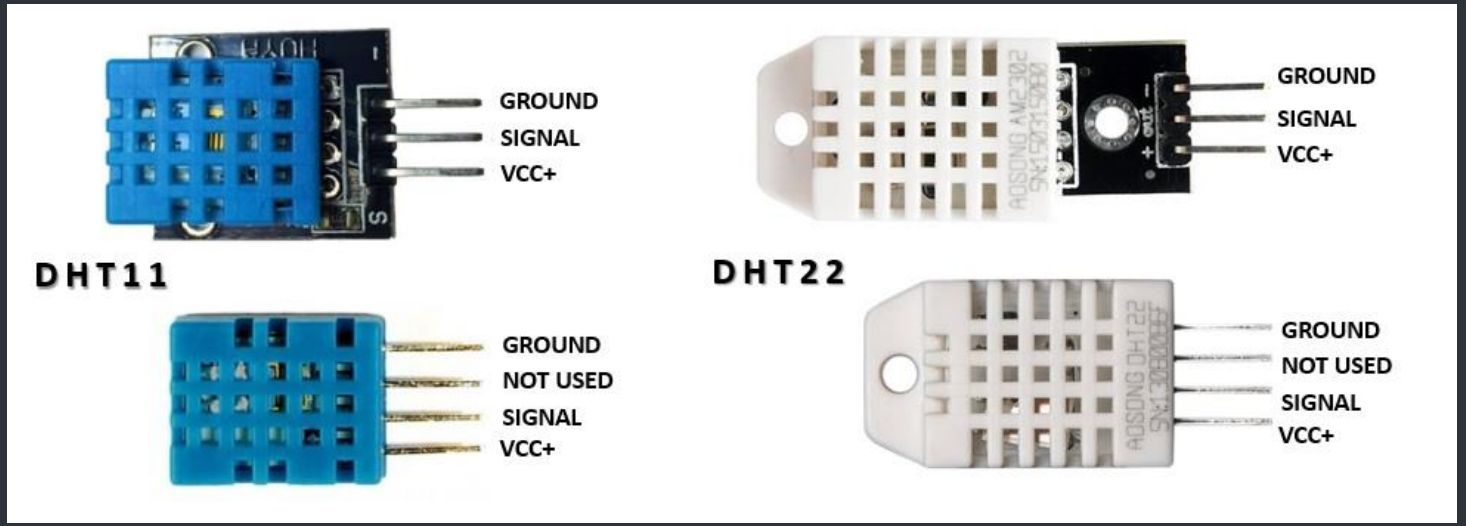
```
# Los sensores DHT (Digital Humidity & Temperature) son sensores digitales
de bajo costo con sensores capacitivos de humedad y termistores para medir
el aire circundante. Cuentan con un chip que maneja la conversión de
analógico a digital y proporciona una interfaz de 1 cable.

# Los sensores DHT11 (azul) y DHT22 (blanco) proporcionan la misma interfaz
de 1 cable, sin embargo, el DHT22 requiere un objeto separado ya que tiene
un cálculo más complejo.
```



}

Sensores DHT {

```
    # Pinout de ambos modelos.
```



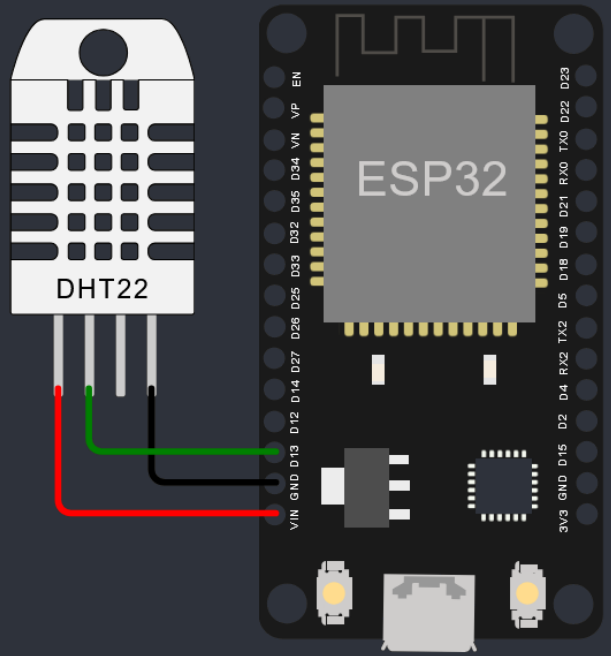
}

Sensores DHT {

```
    # DHT11:

    # Temperatura: Rango de 0°C a 50°C con una precisión de ±2°C.
    # Humedad: De 20% a 80/90% de humedad relativa con una precisión de ±5%.
    # Tiempo de respuesta: 1 Hz. Más lento que el DHT22.
    # Más económico y básico.

    DHT22:

    # Temperatura: Rango de -40°C a 80°C con una precisión de ±0.5°C.
    # Humedad: De 0% a 100% de humedad relativa con una precisión de ±2-5%.
    # Tiempo de respuesta: 0,5 Hz. Más rápido que el DHT11.
    # Mayor precisión y rango.

}
```

Ejemplo 7 – DHT {

```
from machine import Pin
from time import sleep
from dht import DHT22

sensor = DHT22(Pin(13))

while True:
    sensor.measure()
    temperatura = sensor.temperature()
    humedad = sensor.humidity()
    print(f'Temperatura: {temperatura} ºC')
    print(f'Humedad: {humedad} %')
    sleep(1)
}
```



```
1
2
3  Aprender a programar
4  es aprender a pensar.
5
6
7
8
9
10
11 { Steve Jobs; }
12
13
14
```


{ Nos vemos en la proxima clase }