Jugando fuerte con Python

- Maqueta: Sensor Gas
- Maqueta DHT11

Ventajas y desventajas Python

Ventajas

Python es un lenguaje de desarrollo y curva de aprendizaje rápido. Tiene una comunidad amplia con muchas librerías, ejemplos, tutoriales... que para casi todos los problemas, seguro que encuentras una solución escrita en Python

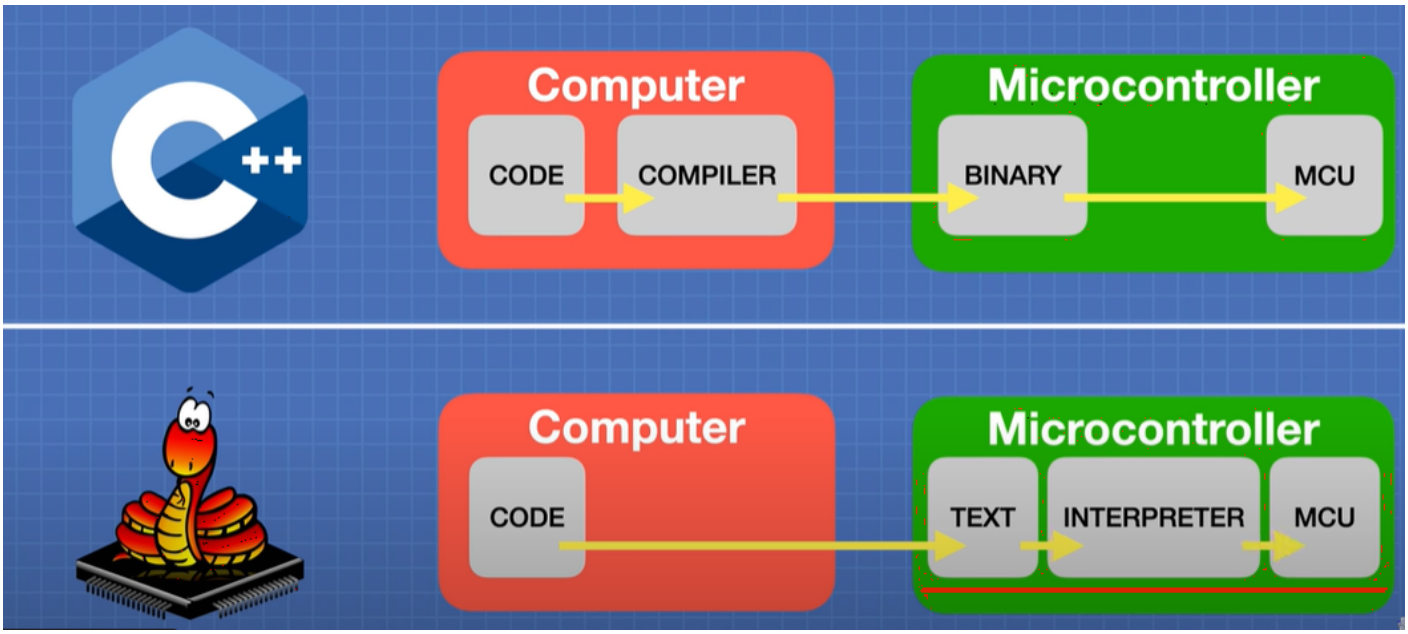
Es un lenguaje de alto nivel, es decir, que se programa igual que los programadores, pero interpretable para los humanos !. Comparándolo con otros lenguajes (Java, C++, etc..) es el más "*humanizado*".

También gestiona la memoria, por ejemplo, si programas en C++, tú eres el responsable de limpiar la memoria de datos que ya no usas, o corres el peligro de quedarte sin memoria. En Python ya lo hace por ti.

Desventajas

La gestión de memoria que antes se mencionaba tiene un precio; bajada de velocidad y paradójicamente coste de memoria.

En otros programas, el compilador esta en tu PC, pero en Python está en el dispositivo (por eso se llama lenguaje **Interpretado**), esto hace que ocupa memoria, y en microbit por ejemplo esto hace que no puedes usar Python y código Bluetooh pues no hay suficiente memoria RAM.



Fuente vídeo Exploring the Arduino Nano ESP32 | MicroPython & IoT

También hay que tener en cuenta que si Python es un lenguaje **interpretado**, siempre será más lento que un lenguaje **compilado** por ejemplo el C++, pues para ejecutarlo el dispositivo, lo ejecuta, pues lo tiene en binario y en paz, pero en Python cada instrucción necesita ser interpretado, decodificado, en binario antes de ejecutarse.

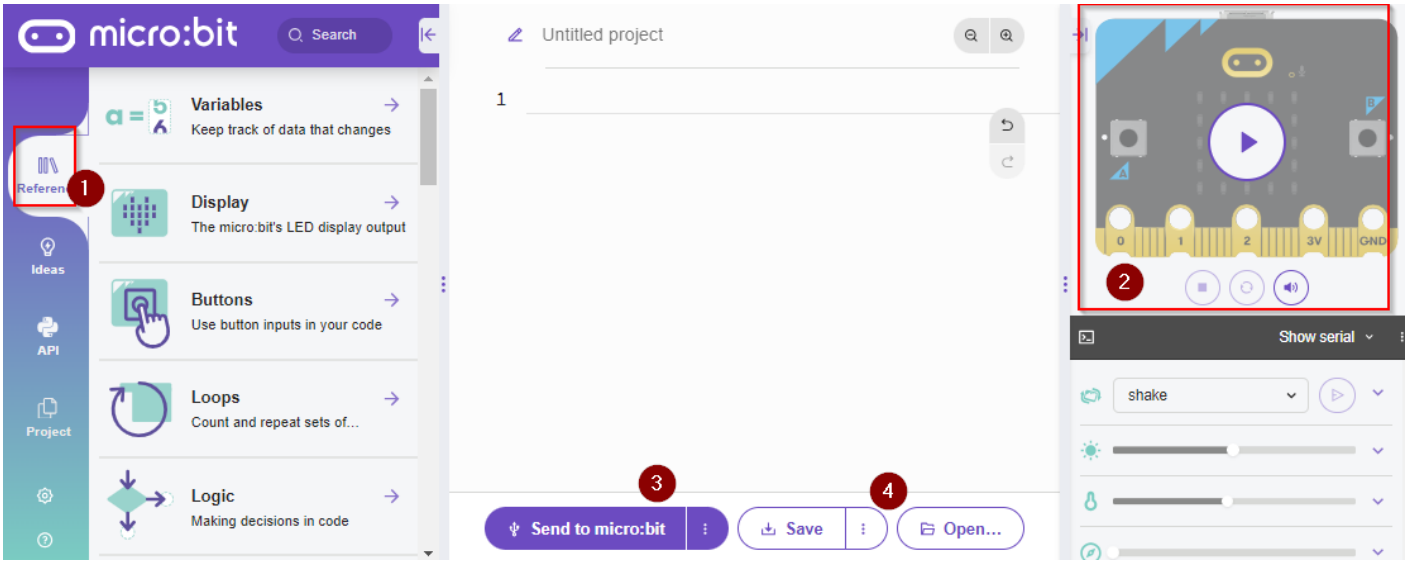
Editores

Tienes dos opciones, online o local :

Programar online con https://python.microbit.org (recomendado)

Entramos en https://python.microbit.org/ y el editor online nos permite trabajar ;

1. Una biblioteca de códigos que nos permitirá seleccionar y usar para programar de forma guiada
2. Un simulador para ver cómo se ejecutaría nuestro código
3. Un botón para enviar a la microbit real
4. Botones para guardar nuestro código de forma local y abrir los existentes.



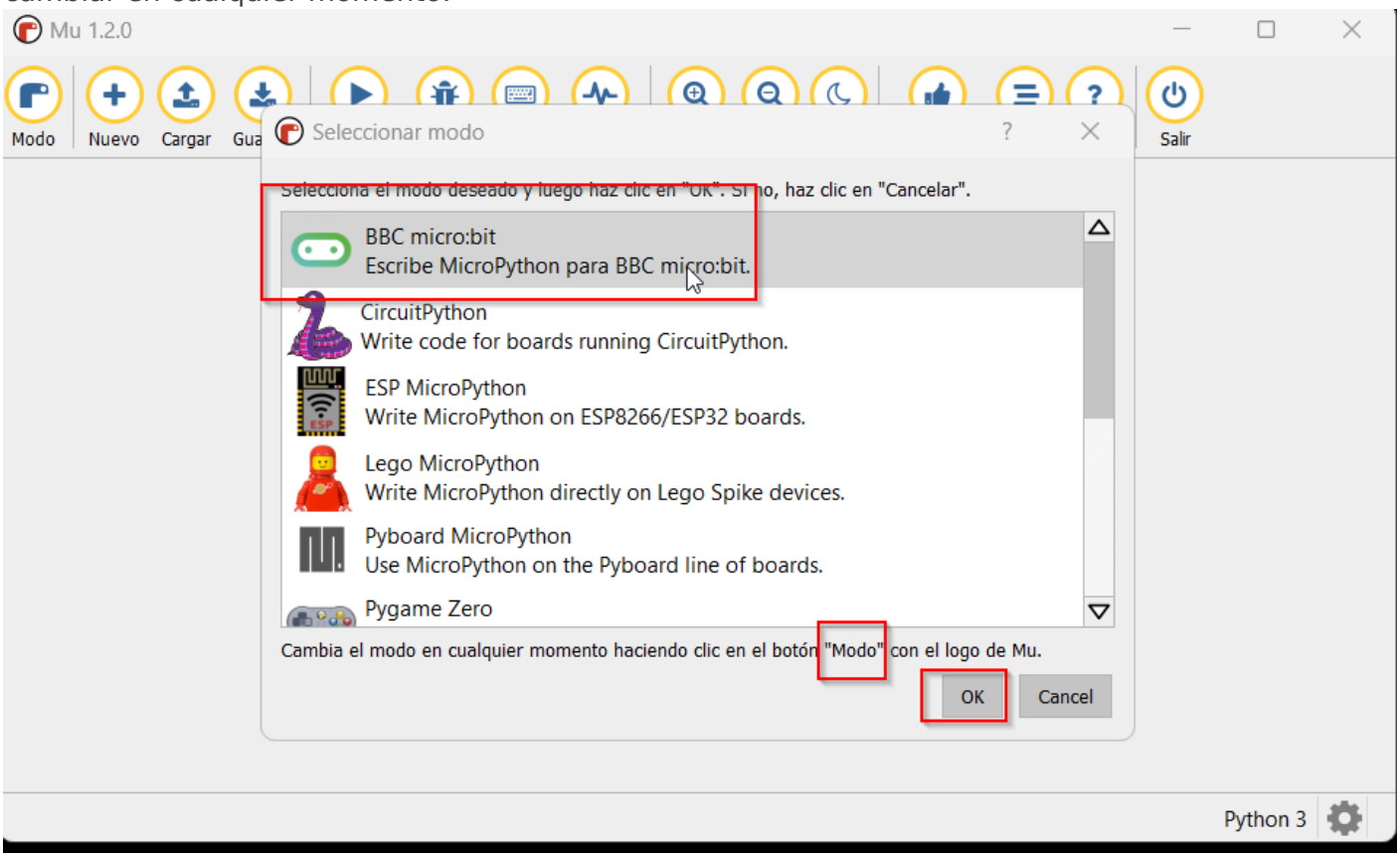
> En este curso utilizaremos el editor online microbit.org

Programar en local con MU

Es un editor muy sencillo, se descarga en https://codewith.mu/ y permite su instalación en Windows, Linux y Apple.
2024-07-04 18_44_27-(1) Exploring the Arduino Nano ESP32 _ MicroPython & IoT Cloud - YouTube.pr

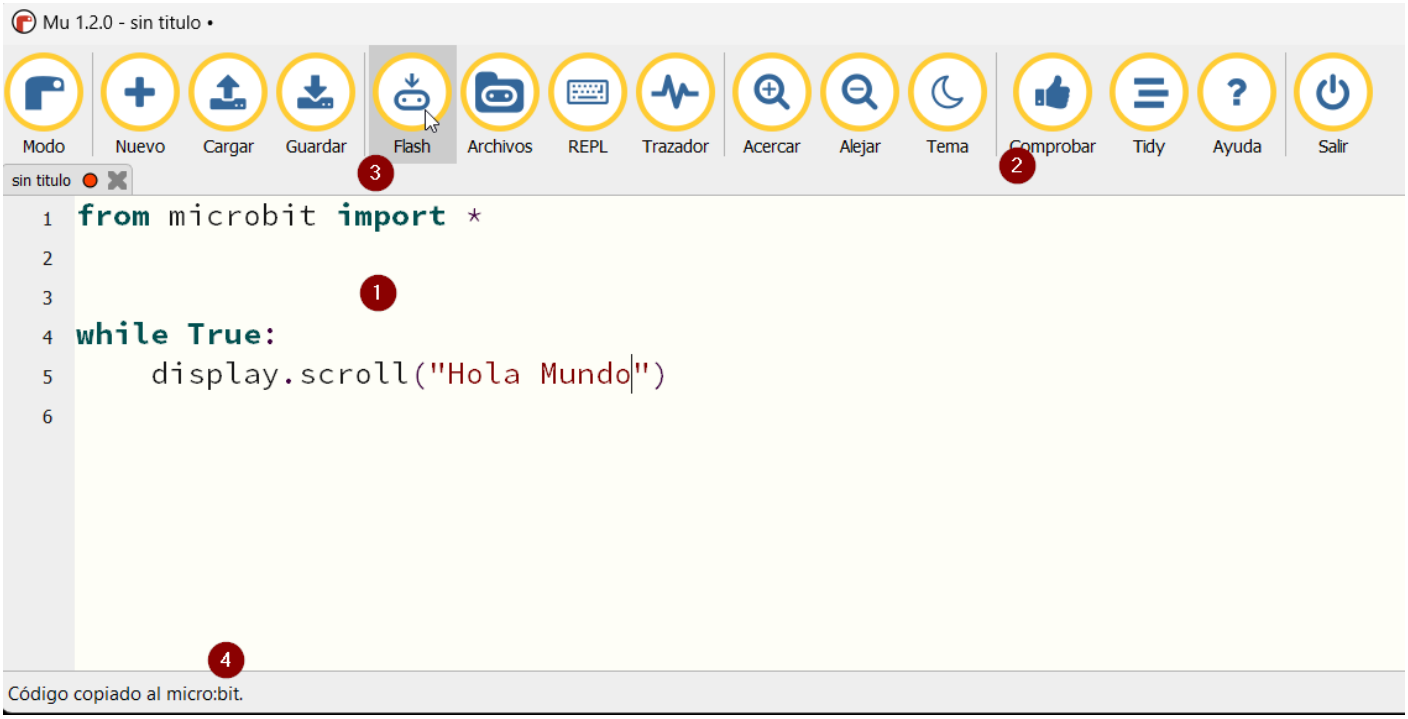
Fuente https://codewith.mu/ CC-BY-NC-SA
La primera vez que lo ejecutamos (tarda algo la primera vez) nos pide el **modo** que se puede

cambiar en cualquier momento:



1 Escribimos el código
2 Lo comprobamos
3 Flasheamos, es decir enviamos el código al Microbit (conectarlo previamente)
4 Cuando sale el mensaje *Código copiado al microbit* **procedemos a resetearlo** para que la placa ejecute el programa.

> ATENCIÓN ES IMPORTANTE **RESETEAR LA MICRO:BIT** tienes un botón de ***reset*** al lado del conector de USB para no estar desconectando y conectando. Una vez reseteado tu programa funcionará.




```
from microbit import *


while True:
  display.scroll("Hola Mundo")
```

OTROS EDITORES DE PYTHON QUE NO SON COMPATIBLES CON PYTHON MICROBIT

Vamos a ver este programa escribo en https://python.microbit.org/

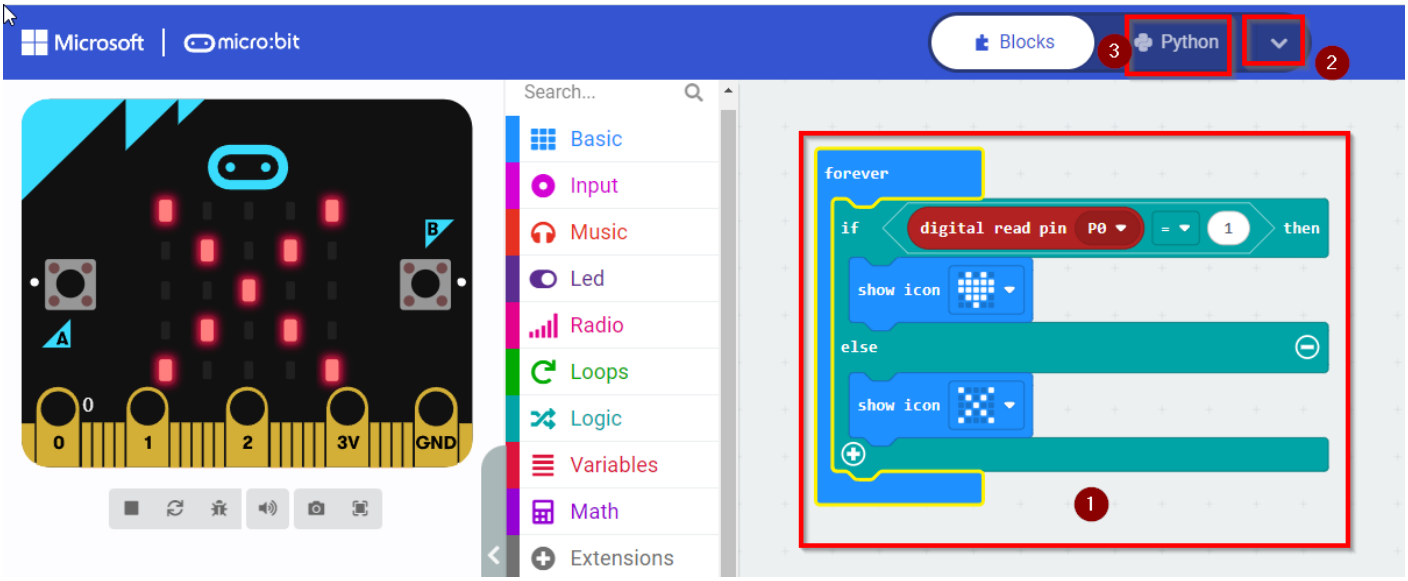
```
# Imports go at the top
from microbit import *
while True:
    if pin0.is_touched():
        display.show(Image.HEART)
    else:
        display.show(Image.NO)
```

Lo que hace es :

https://www.youtube.com/embed/uI2p9HazV1Y

EL MISMO CÓDIGO EN MAKECODE-PYTHON

Makecode a pesar de que esta orientado a programar con bloques, t**iene su sección de Python**



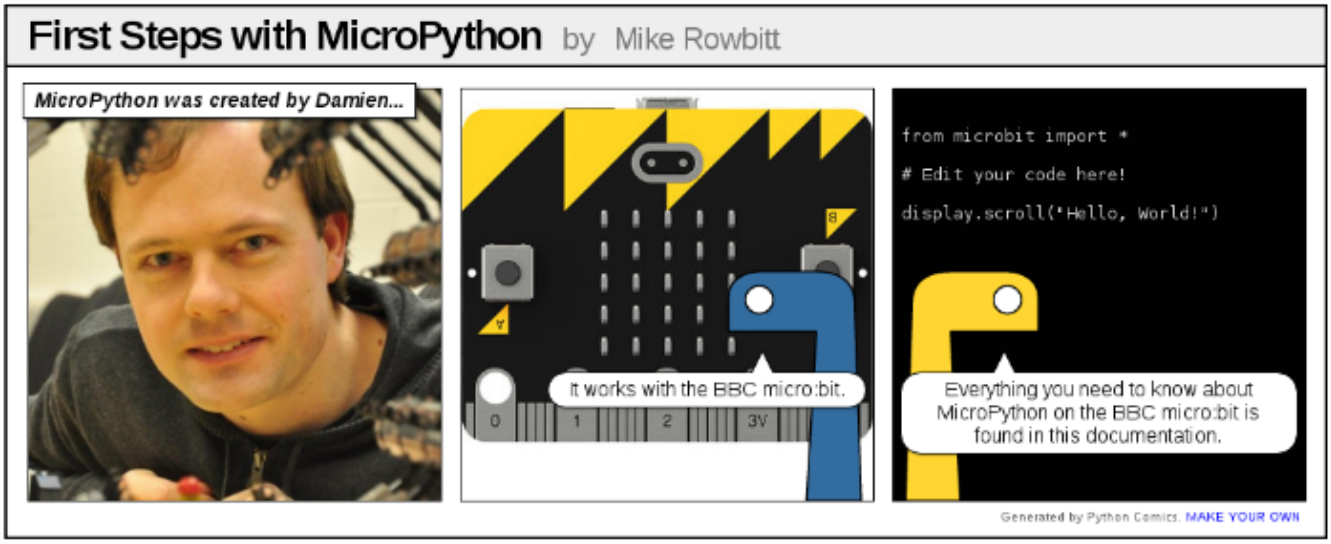
Al darle en Python (arriba a la derecha), muestra este código

```
def on_forever():
    if pins.digital_read_pin(DigitalPin.P0) == 1:
        basic.show_icon(IconNames.HEART)
    else:
        basic.show_icon(IconNames.NO)
basic.forever(on_forever)
```

Como se puede ver **makecode python no es compatible con https://python.microbit.org/** ya lo dice en su tutorial https://microbit-micropython.readthedocs.io/en/v2-docs/

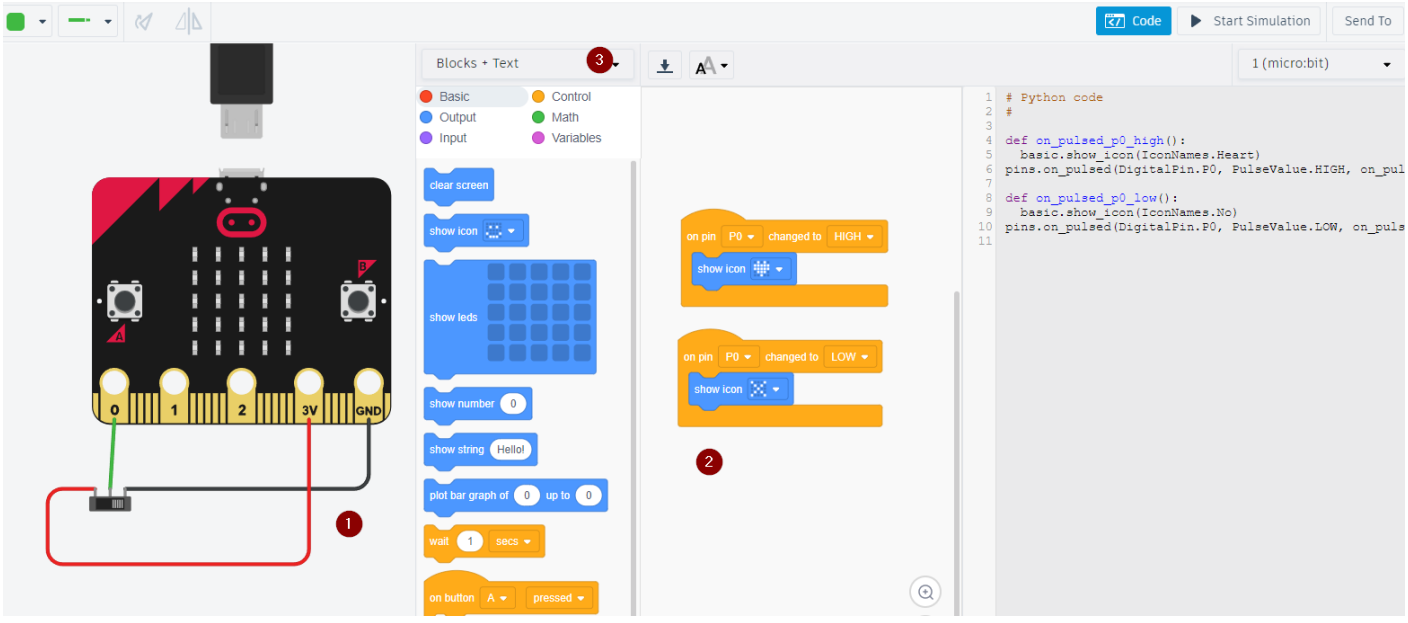
Note

The MicroPython API will not work in the MakeCode editor, as this uses a different version of Python.



EL MISMO CÓDIGO CON PYTHON DE TINKERCAD

Tinkercad https://www.tinkercad.com/ es una herramienta estupenda de simulación pues es muy realístico, igual que Maquecode, este muy orientado a la programación en bloques pero también tiene su sección de código python



Si le das la opción de bloque+código intenta muestra los bloques traducidos a código, pero si le das la opción sólo código **pierdes** la programación en bloques, Esto ya lo vimos en https://libros.catedu.es/books/programa-arduino-mediante-codigo/page/software en los párrafos escritos en naranja.

El código generado vemos que **no es compatible con Python microbit**

```
# Python code
#

def on_pulsed_p0_high():
  basic.show_icon(IconNames.Heart)
pins.on_pulsed(DigitalPin.P0, PulseValue.HIGH, on_pulsed_p0_high)

def on_pulsed_p0_low():
  basic.show_icon(IconNames.No)
pins.on_pulsed(DigitalPin.P0, PulseValue.LOW, on_pulsed_p0_low)
```

Introducción al Python

> Esta es **una muy breve introducción al Python** como recordatorio de algunas instrucciones si ya has utilizado este lenguaje.
>
> Si es la primera vez, te recomendamos que visites nuestro curso **PYTHON PARA TODOS**
>
> **Python for everybody** por Charles R. Severance licencia CC-BY-NCSA que empieza desde cero.

Lenguajes, intérpretes y compiladores

Python es un lenguaje **de alto nivel** destinado a ser relativamente sencillo para que los humanos lean y escriban y para que los ordenadores lean y procesen. Otros lenguajes de alto nivel incluyen Java, C ++, PHP, Ruby, Basic, Perl, JavaScript y muchos más. El hardware real dentro de la Unidad Central de Procesamiento (CPU) no comprende ninguno de estos lenguajes de alto nivel.

La CPU entiende un idioma que llamamos **lenguaje de máquina**. El lenguaje de máquina es muy simple y francamente muy tedioso de escribir porque está representado en ceros y unos:

El lenguaje de máquina parece bastante simple en la superficie, dado que solo hay ceros y unos, pero su sintaxis es aún más compleja y mucho más compleja que Python. Muy pocos programadores escriben lenguaje de máquina. En su lugar, creamos varios traductores para permitir que los programadores escriban en lenguajes de alto nivel como Python o JavaScript y estos traductores convierten los programas al lenguaje de máquina para su ejecución real por parte de la CPU.

Estos traductores de lenguaje de programación se dividen en dos categorías generales: (1) intérpretes y (2) compiladores.

Un **intérprete** lee el código fuente del programa como está escrito por el programador, analiza el código fuente e interpreta las instrucciones sobre la marcha. Python es un intérprete y cuando ejecutamos Python de forma interactiva, podemos escribir una línea de Python (una oración) y Python la procesa de inmediato y está lista para que escribamos otra línea de Python.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
```

```
>>> print(y)
42
>>>
```

Está en la naturaleza de un **intérprete** poder tener una conversación interactiva como se muestra arriba. A un **compilador** debemos entregarle todo el programa en un archivo, y luego ejecuta un proceso para traducir el código fuente de alto nivel al lenguaje de máquina y luego el compilador coloca el lenguaje de máquina resultante en un archivo para su posterior ejecución.

Variables

Las variables son como cajas que puedes meter valores. Y los valores pueden ser de varios **tipos** :

- **int** si son enteros
- **float** si tienen decimales
- **binario**Deben comenzar por 0b. Por ejemplo: 0b110, 0b11
- **string** son frases, son "***cadenas***" de caracteres entre "
- **bool** Solamente hay dos literales booleanos True o False
- **lista** Se pueden declarar variables que son conjuntos por ejemplo Colores = ["verde", "rojo", "naranja"]

Para crear una variable puedes usar cualquier palabra, x, y, z o Nombre_alumno ... pero algunas palabras no puedes usar, ver

Para visualizar variables puedes usar la **instrucción print** poniendo entre paréntesis el valor o variable que quieres visualizar.

En la siguiente ventana puedes dar al botón *play* y ver el resultado

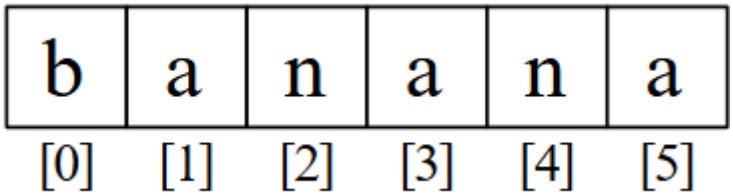
https://trinket.io/embed/python/47afa56dd668

> Modifica los valores como quieras, es un **intérprete**, juega y dale al play para ver el resultado

Como puedes ver se ha introducido un operador el + que realiza la suma del valor de x original (43) y se le incrementa una unidad resultando en la impresión un 44.

Cadenas

Cadenas son secuencias de caracteres, por ejemplo la palabra "banana"



fuente 'Python for Everybody' por Charles R. Severance

Se puede obtener su longitud con la función len, o obtener un carácter ...

https://trinket.io/embed/python/5e023b136db8

Operadores

Este apartado de operadores es adaptado de Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Los **operadores aritméticos** se utilizan para realizar operaciones matemáticas como sumas, restas, multiplicaciones, etc.

Operador	Descripción	Ejemplo
+	Suma o concatenación en textos	`5+3=8`, `"Hola" + "Mundo" = "Hola Mundo"`
-	Diferencia	`6-3=3`
*	Multiplicación	`3*3=9`
/	División	`6/2=3`
//	Parte entera de un cociente	`10//3=3`
%	Resto de un cociente	`10%3=1`
**	Potenciación	`5**2=25`

Los **operadores de asignación** se utilizan para asignar valores a variables.

Operador	Descripción	Ejemplo
=	Asignación	`x=4`, `a = a + 1`

Operador	Descripción	Ejemplo
+=	Suma y asignación	`x+=1` equivale a `x = x + 1`
-=	Diferencia y asignación	`x-=1` equivale a `x = x - 1`
*=	Multiplicación y asignación	`x*=3` equivale a `x = x * 3`
/=	División y asignación	`x/=3` equivale a `x = x / 3`
%=	Asignación de restos	`x%=3` equivale a `x = x % 3`
**=	Asignación de exponentes	`x**=3` equivale a `x = x ** 3`

Los **operadores de comparación** comparan dos valores/variables y devuelven un resultado booleano: Verdadero o Falso `True` o `False`.

Operador	Descripción	Ejemplo
==	Igual a	`2==3` retorna `False`
!=	Distinto de	`2!=3` retorna `True`
<	Menor que	`2<3` retorna `True`
>	Mayor que	`2>3` retorna `False`
<=	Menor o igual que	`2<=3` retorna `True`
>=	Mayor o igual que	`2>=3` retorna `False`

Los **operadores lógicos** se utilizan para comprobar si una expresión es Verdadera o Falsa. Se utilizan en la toma de decisiones.

Operador	Descripción	Ejemplo
and	AND lógica	`a and b #True si a y b son ciertos`
or	OR lógica	`a or b #True si a o b son ciertos`
not	NOT lógica	`not a #True si el operador a es falso`
in	pertenencia	Devuelve True si pertenece
no int	no pertenencia	Devuelve True si no pertenece
is	identidad	Devuelve True si son iguales
is not	no identidad	Devuelve True si no son inguales

Los **operadores bit a bit** o bitwise actúan sobre los operandos como si fueran cadenas de dígitos binarios. Operan bit a bit:

Operador	Descripción	Ejemplo
&	AND bit a bit	`5&6 # 101 & 110 = 110 = 4`
|	OR bit a bit	`5 | 6 # 101 | 110 = 111 = 7`
~	NOT bit a bit	`~3 # ~011 = 100 = -4`
^	XOR bit a bit	`5^3 # 101^011 = 110 = 6`
<<	Desplazamiento izquierda	`4<<1 # 100 << 1 = 1000 = 8`
>>	Desplazamiento derecha	`4 >> 1 # 100 >> 1 = 010 = 2`

> Prueba, juega con este código:

https://trinket.io/embed/python/502063b6b44b

> **Comentarios en Python**
> Una sola línea : Escribiendo el símbolo almohadilla (#) delante del comentario.
> Multilínea: Escribiendo triple comillas dobles (“””) al principio y al final del comentario.

Entradas de teclado

Ya hemos visto salidas por pantalla con **print**, pero ahora con input puede leer variables del teclado, esto es mejor experimentarlo que leerlo :

https://trinket.io/embed/python/34653253eb52

Fíjate que hay que poner las líneas **x = float (x)** e **y = float(y)** para convertirlos a números decimales, en caso contrario las interpreta string y no puede multiplicar en Resultado, pero en el siguiente ejemplo **no es necesario en la variable cel** (celsius) pues se multiplica por números

decimales 32.0 5.0 y 9.0

https://trinket.io/embed/python3/5dbec1550b

> **try** y **except** son dos funciones que son *un seguro para el programador* por si el usuario en vez de teclear un número, mete un string o carácter

> **La sangría es importante en Python**
> La sangría se refiere a los espacios al comienzo de una línea de código. Mientras que en otros lenguajes de programación la sangría en el código es solo para facilitar la lectura, la sangría en Python es muy importante ya que se usa para indicar un bloque de código.

Condicionales

Las instrucciones **if: else:** son las que nos permiten realizar operaciones según las condiciones puestas. *Ojo con la sangría*

https://trinket.io/embed/python/cc1aa3f917a7

> **\n** es un carácter especial que significa "Salto de página"

Bucles

- **while** ejecuta lo contenido en la sangría mientras sea verdadero la condición
- **for** ejecuta lo contenido en la sangría mientras y va recorriendo la variable dentro del rango creado

Para verlo mejor vamos a ver estos ejemplos

- EJEMPLO BUCLE WHILE
  - mientras n sea positivo va ejecutando : imprime n y lo decrementa
  - al decrementar llega un momento que deja de ser positivo y finaliza el bucle
- EJEMPLO BUCLE WHILE INFINITO
  - Es muy típico en robótica, todo el rato hace el bucle (en robótica para que lea los sensores y realice cosas en los actuadores) pero este ejemplo no esta en un robot

sino en tu pc y no queremos que se quede "colgado" luego al teclear "fin" acaba gracias a la instrucción **break**
    - Fíjate que hay una instrucción **continue** para que pase a la siguiente iteración provocando que no imprime lo tecleado
- EJEMPLO BUCLE FOR FRIENDS
    - Va recorriendo la variable friend dentro del cojunto lista friends
    - como puedes ver la diferencia entre for y while es que for además recorre la variable
- EJEMPLO BUCLE FOR
    - mientras n este en el rango de 0 a 5 se ejecuta

Venga pruébalo !!!

https://trinket.io/embed/python/f797a1eaea48

Funciones

No vamos a entrar en detalle, pero observa el siguiente código

- **FUNCIONES PREDEFINIDAS** Si observas, la primera línea llama a importar una librería externa, **import math** donde math es un fichero que tienen funciones predefinidas, vamos a utilizar una de ellas, la raiz cuadrada **sqrt** luego para llamar a esa función que esta definida dentro de math se hace con la instrucción **math.sqrt**
- **FUNCIONES DEFINIDAS POR TI** em este caso, se utiliza la palabra **def** para crear una función, que le vamos a pasar tres argumentos a, b y c y para finalizar la función usamos **return** para devolver el valor que queremos obtener

https://trinket.io/embed/python/900fd133a2a9

Para saber más de Python

CURSO PYTHON FOR EVERYBODY en español	ver
Curso completo de Python 222pag pdf (*)	Descargar
Curso completo de Python 422pag (*)	Descargar
Curso completo de Python desde 0 (*)	Ver
Curso de Python desde 0 (*)	Ver

Manual de referencia Python (*)	Ver
Programación en Python (*)	Ver
Trabajando con ficheros en Python (*)	Ver
Programación orientada a objeto en Python (*)	Ver
un manual para aquellos usuarios con previos conocimientos de Python, como la programación modular y orientada a objetos. También algunos conocimientos de las librerías **tkinter** (Para crear interfaces gráficos y **SQlite3** (para gestionar bases de datos). (*)	Descargar

(*) Agradecimientos a Pere Manel http://peremanelv.com

Micropython de microbit

> Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA

API: El módulo microbit

Todo lo necesario para interactuar con el hardware de la micro:bit está en el módulo *microbit* y se recomienda su uso escribiendo al principio del programa:

```
from microbit import *
```

Las funciones disponibles directamente son:

```
sleep(ms) #1
running_time() #2
temperature() #3
scale(valor_a_convertir, from_=(min, max), to=(min, max)) #4
panic(error_code) #5
reset() #6
set_volume(valor) #7 (V2)
'''
1 Esperar el número de milisegundos indicado
2 Devuelve el tiempo en ms desde la última vez que se encendió la micro:bit
3 Devuelve la temperatura en Celcius
4 Convierte un número de una escala de valores a otra
5 La micro:bit entra en modo pánico por falta de memoria y se dibuja una
cara triste en la pantalla. El valor de error_code puede ser cualquier entero.
6 Resetea la micro:bit
7 Estable el volumen de salida con un *valor* entre 0 y 255
'''
```

Estructuras de datos en Python

Las listas (list)

Se trata de un tipo de dato que permite almacenar series de datos de cualquier tipo bajo su estructura. Se suelen asociar a las matrices o arrays de otros lenguajes de programación.

En Python las listas son muy versatiles permitiendo almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas lo que sea.

Una lista se crea con `[]` y sus elementos se separan por comas. Una gran ventaja es que pueden tener datos de diferentes tipos.

```
lista = [1, "Hola", 3.141592, [1 , 2, 3], Image.HAPPY]
```

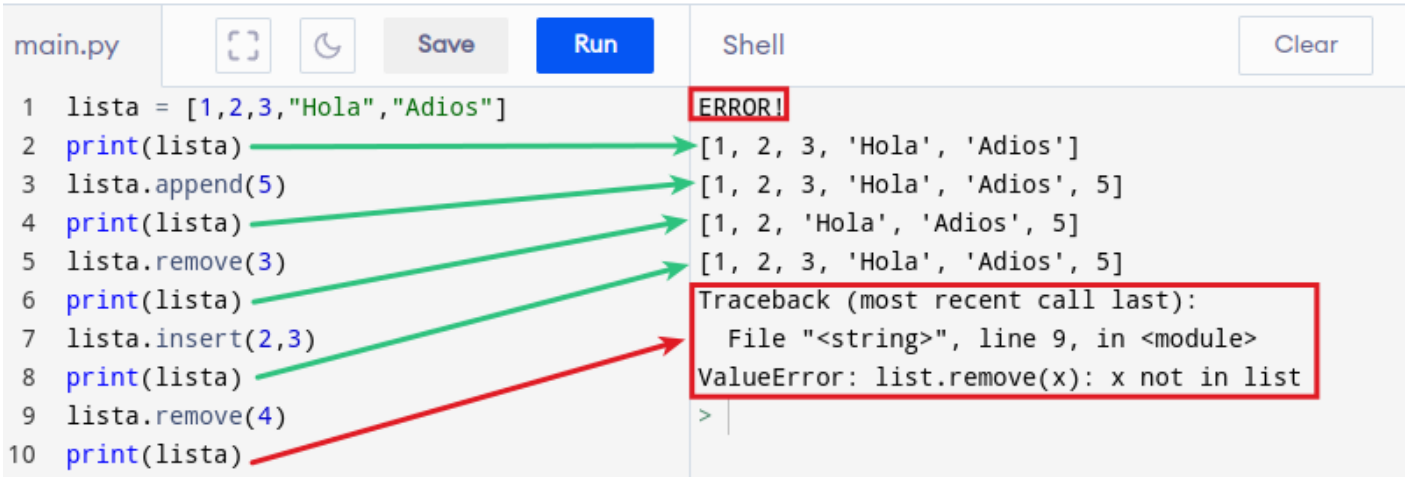
Las de principales propiedades de las listas:

- Son ordenadas, mantienen el orden en el que han sido definidas
- Pueden ser formadas por tipos arbitrarios de datos
- Pueden ser indexadas con [i]
- Se pueden anidar, es decir, meter una lista dentro de otra
- Son mutables, ya que sus elementos pueden ser modificados
- Son dinámicas, ya que se pueden añadir o eliminar elementos

Hay dos métodos aplicables:

- **`append`**. Permite agregar elementos a la lista.
- **`remove`**. Elimina elementos de la lista.
- **`insert(pos,elem)`**. Inserta el elemento `elem` en la posición `pos` indicada.

En el ejemplo vemos el funcionamiento.



Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Con estos conocimientos tendremos suficiente para hacer lo que pretendemos, que no es otra cosa que animar imágenes.

Las tuplas (tuple)

Son muy similares a las listas con una diferencia principal con las mismas y es que las tuplas no pueden ser modificadas directamente, lo que implica que no dispone de los métodos vistos para listas. Una tupla permite tener agrupados un número inmutable de elementos.

Una tupla se crea con `()` y sus elementos se separan por comas.

```
tupla = (1, 2, 3)
```

Principales propiedades:

- Se pueden declarar sin usar los paréntesis, pero no se recomienda. No usarlos puede llevarnos a ambigüedades del tipo print(1, 2, 3) y print((1, 2, 3)).
- Si la tupla tiene un solo elemento esta debe finalizar con coma.
- Se pueden anidar tuplas, por ejemplo `tupla2 = tupla1, 4, 5, 6, 7`.
- Se pueden declarar tuplas vacias, por ejemplo `tupla3 = ()`.
- Las tuplas son *iterables* por lo que sus elementos pueden ser accesados mediante la notación de índice del elemento entre corchetes. Si se quiere acceder a un rango de indices se separan por ":" ambos índices.
- Es posible convertir listas en tuplas simplemente poniendo la lista dentro de los paréntesis de la tupla, por ejemplo, `tupla_lista = ([1, "Hola", 3.141592, [1 , 2, 3], Image.HAPPY])`

A continuación vemos un ejemplo.

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Diccionarios (dict)

Estas estructuras contienen la colección de elementos con la forma `clave:valor` separados por comas y encerrados entre `{}`. Las claves son objetos inmutables y los valores pueden ser de cualquier tipo. Sus principales características son:

- En lugar de por índice como en listas y tuplas, en diccionarios se acceder al valor por su clave.
- Permiten eliminar cualquier entrada.
- Al igual que las listas, el diccionario permite modificar los valores.
- El método `dicc.get()` accede a un valor por la clave del mismo.
- El método `dicc.items()` devuelve una lista de tuplas `clave:valor`.
- El método `dicc.keys()` devuelve una lista de las claves.
- El método `dicc.values()` devuelve una lista de los valores.
- El método `dicc.update()` añade elemento `clave:valor` al diccionario.
- El método `del dicc` borra el par `clave:valor`.
- El método `dicc.pop()` borra el par `clave:valor`.

A continuación vemos un ejemplo

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Bucles

Los **Bucles** son un tipo de estructura de control muy útil cuando queremos repetir un bloque de código varias veces. En Python existen dos tipos de bloques, el bucle ***for*** para contar la cantidad de veces que se ejecuta un bloque de código, y el bucle ***while*** que realiza la acción hasta que la condición especificada no sea cierta.

- While
- for
- Bucle for decontando
- Sentencias break y continue

While

La sintaxis de while es la siguiente:

```
while condicion:
    bloque de codigo
```

donde "*condicion*", que se evalúa en cada iteración, puede ser cualquier expresión realizado con operadores condicionales que devuelva como resultado un valor True o False. Mientra que "bloque de codigo" es el conjunto de instrucciones que se estarán ejecutando mientras la condición sea verdadera (True o '1'). Es lo mismo poner `while true:` que poner `while 1:`.

Para recorrer los bucles se utilizan variables que forman parte de la condición, estableciendose en esta lo que deben cumplir.

Un ejemplo sencillo podría ser el siguiente, controlar el riego de una planta en función del valor de la humedad de la tierra en la que está.

```
from microbit import *

while (humedad() < 45):
    display.scroll(Image.SAD)
    sleep(1000)

display.show(Image.HAPPY)
```

que hará que si la humedad baja por debajo de 45 se muestre una carita triste indicando que hay que regar y si es mayor mostrará una carita feliz. Evidentemente hay que resolver el tema de como obtener la humedad, pero esa es una historia que veremos mas adelante.

El bucle `while` puede tener de manera opcional un bloque `else` cuyas sentencias se ejecutan cuando se han realizado todas las iteraciones del bucle. Un ejemplo lo vemos a continuación:

```
cuenta = 0
while cuenta < 5:
    print("Iteración del bucle")
    cuenta = cuenta + 1
else:
    print("bucle finalizado")
```

for

Son también bucles pero su acción está dirigida a contar el número de veces que ocurre algo o realizar una acción un determinado número de veces. Es especialmente útil para recorrer los datos de una lista, tupla o diccionario.

La sintaxis de este tipo de bucles en Python es:

```
for variable in secuencia:
    declaracion
```

Siendo "variable" la variable que se va a recorrer en el bucle de forma que cuando se alcance el valor establecido se sale del bucle.

La variable puede ser una cadena, un rango de valores que se expresa con `range(n)`, siendo n el número de valores del rango que se inicia en 0 y que pueden ser iterados con una variable. Mas ampliamente, la sintaxis de `range()` es `range(start, stop, step)` siendo `start` y `stop` opcionales.

Veamos un primer ejemplo en el que vamos a utilizar un bucle para encender uno a uno por filas los LEDs de la primera y última columna.

```
from microbit import *
for var in range(5): # var puede tomar 5 valores, del 0 al 4
    display.set_pixel(0, var, 9) # Se ilumina el LED de la fila 0 y el valor de var para
columna
    sleep(300)
    display.set_pixel(4, var, 9) # Se ilumina el LED de la fila 4 y el valor de var para
columna
    sleep(300)
```

Los bucles se pueden anidar, es decir se puede crear un bucle dentro de otro del mismo o diferente tipo, de forma que por cada iteración del bucle mas externo se tienen que producir todas las iteraciones del bucle mas interno. Veamos como ejemplo el de encender todos los LEDs de uno en uno, de izquierda a derecha, utilizando el valor de sus coordenadas x,y. El programa sería:

```
from microbit import *

display.clear()
for y in range(0, 5): # Valor de columna
    for x in range(0, 5): # Valor de fila
        display.set_pixel(x, y, 9) # Encender LED x,y
        sleep(100)
```

En la animación siguiente vemos el programa en funcionamiento.

ejem_dicc.png

El bucle `for` puede tener de manera opcional un bloque `else` cuyas sentencias se ejecutan cuando se han realizado todas las iteraciones del bucle. Un ejemplo lo vemos a continuación:

```
for var in range(5):
    print(var)
else:
    print("bucle finalizado")
```

Bucle for decontando

Se trata del mismo bucle `for` pero ahora la cuenta la realizamos hacia atrás. Hay dos formas sencillas de hacerlo:

- Utilizando la función `range()`. Si queremos darle un enfoque Pythonic simplemente configuramos los argumentos de la función de manera que se indique el principio, el final y el incremento, que será logicamente negativo.

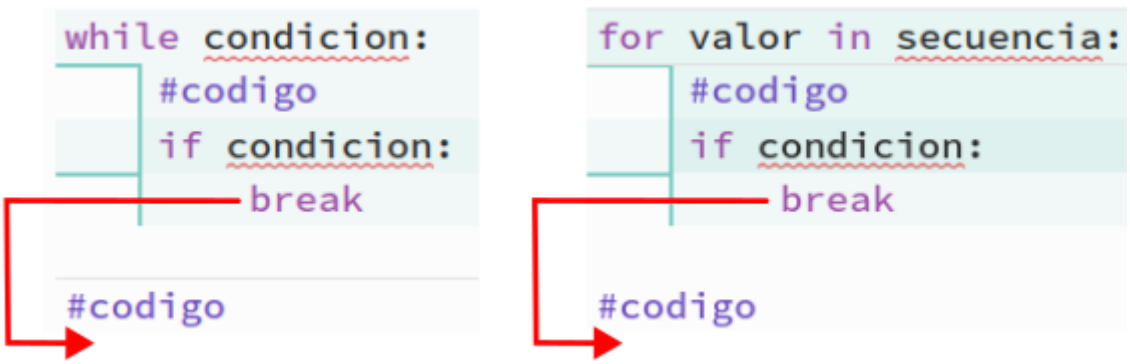
```
for i in range(20, 0, -2): #imprimere 20, 18, 16, ... 0
```

- Utilizando la función `reversed()`. Es una función incorporada en la que hay que indicar como primer argumento el final de la cuenta, como segundo el principio, teniendo en cuenta que se omite, y como tercero el decremento si es ditintos de 1, pero se especifica en módulo. Se utiliza así:

```
for i in reversed(range(0,21,2)): #imprimere 20, 18, 16, ... 0
```

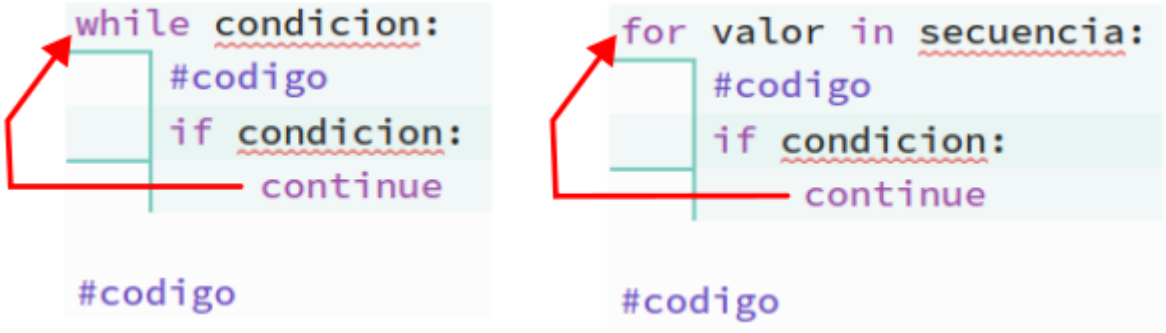
Sentencias `break` y `continue`

La sentencia `break` se utiliza para terminar un bucle de forma inmediata al ser encontrada. En la imagen vemos la sintaxis de la sentencia `break` y su funcionamiento.



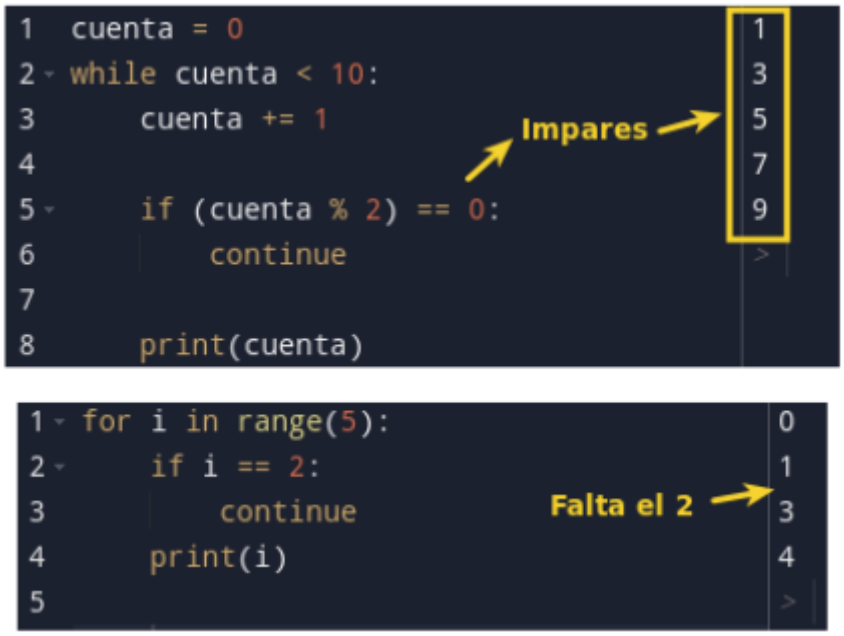
Federico Coca Guia de Trabajo de Microbit CC-BY-SA

La sentencia `continue` se utiliza para saltar la iteración actual del bucle y el flujo de control del programa pasa a la siguiente iteración. En la imagen vemos la sintaxis de la sentencia `continue` y su funcionamiento.



Federico Coca Guia de Trabajo de Microbit CC-BY-SA

En la figura siguiente vemos dos ejemplos de esta sentencia



Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Sentencia condicional `if...else`

En Python hay tres formas de declaración de `if...else`

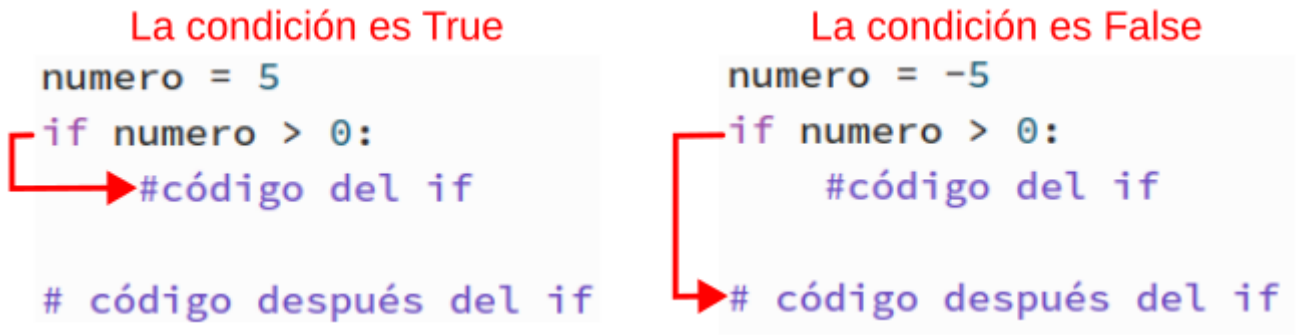
> 1. Declaración `if`
> 2. Declaración `if...else`
> 3. Declaración `if...elif...else`

1. Declaración `if`. La sintaxix de esta declaración en Python tiene la forma siguiente:

```
if condicion:
    # Cuerpo de la sentencia if


# Código después del if
```

Si el resultado de evaluar la condición es cierto (True o 1), el código en "Cuerpo de la sentencia if" y lo estará haciendo mientras se cumpla la condición.

En el momento que la condición sea evaluada como falsa (False o 0) el código en "Cuerpo de la sentencia if" se omite y continua la ejecución del programa por "Código después del if". En la figura siguiente vemos la explicación de forma gráfica.



Funcionamiento de la sentencia if

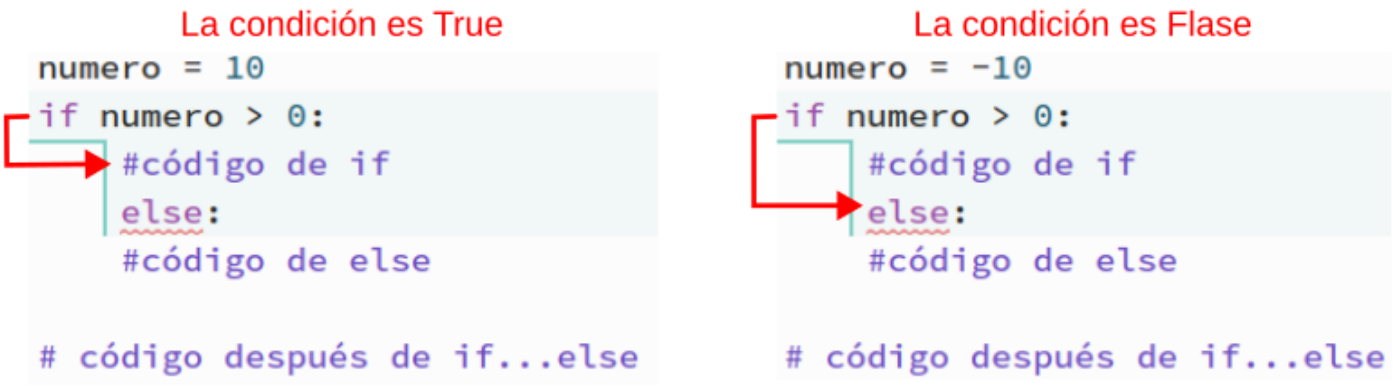
Federico Coca Guia de Trabajo de Microbit CC-BY-SA

1. Declaración `if...else`. Una sentencia `if` puede tener de manera opcional una clausula `else`. La sintaxix de esta declaración en Python tiene la forma siguiente:

```
if condicion:
    # Bloque de sentencias si condicion es True


    else:
    # Bloque de sentencias si condicion es False
```

La sentencia se evalúa de la siguiente forma: Si `condición` es `True` se ejecuta el código dentro del `if` y el código dentro del `else` se omite. Si `condición` es `False` se ejecuta el código dentro del `else` y el código dentro del `if` se omite. Cuando finaliza bien la parte del `if` o bien la del `else` el programa continua con la siguiente sentencia.

En la figura siguiente vemos la explicación de forma gráfica.



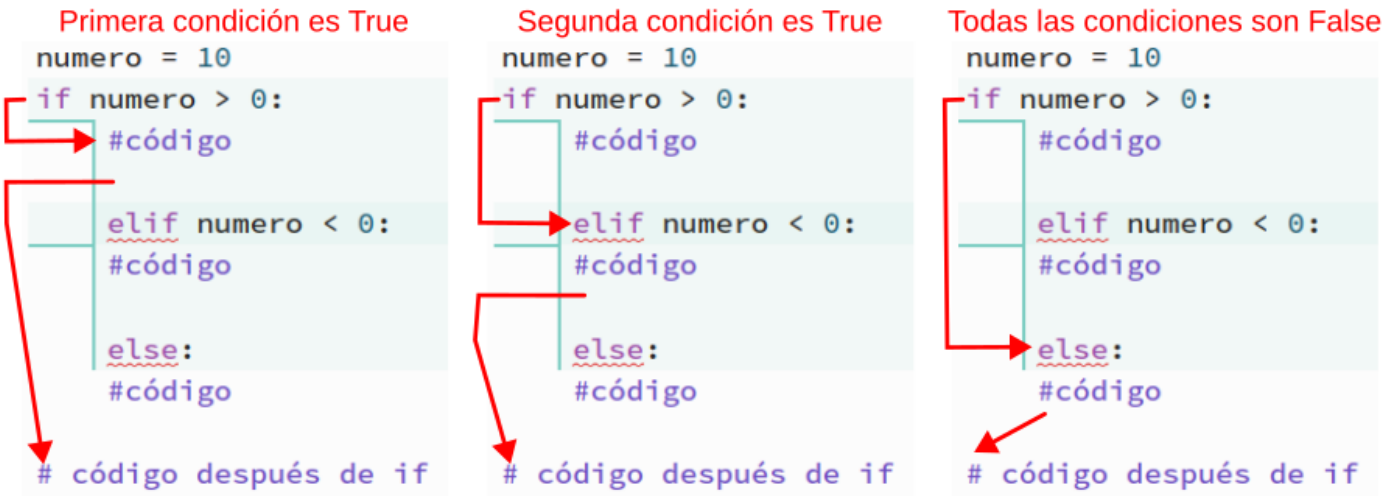
Funcionamiento de la sentencia `if...else` *Federico Coca Guia de Trabajo de Microbit CC-BY-SA*

1. Declaración `if...elif...else`. La sentencia `if...else` se utiliza para ejecutar un bloque de código entre dos alternativas posibles. Sin embargo, si necesitamos elegir entre más de dos alternativas, entonces utilizamos la sentencia `if...elif...else`. La sintaxis de la sentencia `if...elif...else` es:

```
if condicion_1:
    # Bloque 1
elif condicion_2:
    #Bloque 2

    else:
    # Bloque 3
```

Se evalúa así: Si `condicion_1` es `True`, se ejecuta Bloque 1. Si `condicion_1` es `False`, se evalúa `condicion_2`. Si `condicion_2` es `True`, se ejecuta Bloque 2. Si `condicion_2` es `False`, se ejecuta Bloque 3.

En la figura siguiente vemos la explicación de forma gráfica.



Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Funciones en Python

En esta sección vamos a dar solamente una breve introducción a lo que son las funciones y los módulos en Python para estudiar dos funciones concretas definidas en MicroPhyton para micro:bit.

Una función es un bloque de código que realiza una tarea específica.

Supongamos que necesitas crear un programa para crear un círculo y colorearlo. Puedes crear dos funciones para resolver este problema:

- crear una función de círculo
- crear una función de color

Dividir un problema complejo en trozos más pequeños hace que nuestro programa sea fácil de entender y reutilizar.

Existen dos tipos de funciones en Python:

- **Standard library functions (Funciones de biblioteca estándar)**. Son funciones incorporadas en Python que están disponibles para su uso.
- **User-defined functions (Funciones definidas por el usuario)**. Podemos crear nuestras propias funciones para que cumplan con nuestros requisitos.

La sintaxis de una función es la siguiente:

```
def nombre_funcion(argumentos):
    #Cuerpo de la función
```

```
    return
```

Donde,

- `def` es la palabra reservada para declarar una función
- `nombre_funcion` es el nombre que le damos a la función
- `argumentos` es el valor o valores pasados a la función
- `return` retorna un valor desde la función. Es opcional

Veamos un ejemplo sencillo que no manda parametros ni retorna nada.

```
def saludo():
    print("Hola Mundo!")

saludo() #Llama a la función
print("Programa")
saludo()
print("Otra vez programa")
```

Va a generar como salida la cadena "Hola Mundo!" seguida de la cadena "Programa" seguida otra vez de "Hola Mundo!" y finaliza con "Otra vez programa".

Cuando se llama a la función, el control del programa pasa a la definición de la función, se ejecuta todo el código dentro de la función y despés el control del programa salta a la siguiente sentencia después de la llamada a la función.

Como ya se ha mencionado, una función también puede tener argumentos. Un argumento es un valor aceptado por una función. Cuando creamos una función con argumentos necesitamos pasar los correspondientes valores cuando la llamamos.

De forma genérica una función con argumentos tiene la siguiente sintaxis:

```
def funcion(arg1, arg2, ar3,...):
    #Código

#Llamada a la función
funcion(valor1, valor2, valor3, ...)
#Código
```

Cuando llamamos a la función le pasamos los valores correspondiendo valor1 a arg1, valor2 a arg2 y así sucesivamente.

La llamada a la función se puede hacer mencionando el nombre del argumento, que es lo que se conoce como 'argumentos con nombre', siendo el código totalmente equivalente al anterior.

```
funcion(arg1=valor1, arg2=valor2, arg3=valor3, ...)
```

Una función Python puede o no devolver un valor. Si queremos que nuestra función devuelva algún valor a una llamada realizada a función, utilizamos la sentencia `return`.

En el ejemplo siguiente se llama a la función cuatro veces con valores diferentes.

```
def cal_potencia(base, exponente):
    resultado = base ** exponente
    return resultado

#Llamadas a la función
print('Potencia =', cal_potencia(2,8))
print('Potencia =', cal_potencia(3,3))
print('Potencia =', cal_potencia(4,5))
print('Potencia =', cal_potencia(9,6))
```

El resultado es:

```
Potencia = 256
Potencia = 27
Potencia = 1024
Potencia = 531441
```

En Python, las funciones de la biblioteca estándar son las funciones incorporadas que se pueden utilizar directamente en nuestro programa. Por ejemplo,

- `print()`, imprime la cadena entre comillas
- `sqrt()`, devuelve la raíz cuadrada de un número
- `pow()`, devuelve la potencia de un número

Estas funciones están definidas dentro de un módulo. Y, para utilizarlas debemos incluir dicho módulo en nuestro programa. Por ejemplo, `sqrt()` y `pow()` están definidos en el módulo `math`. Para usar las funciones podemos hacer como en el ejemplo siguiente:

```
import math #Carga el módulo math


raiz = math.sqrt(25)
print("La raiz cuadrada de 25 es ", raiz)


potencia = pow(2, 8)
print("2^8 =", potencia)
```

En el ejemplo la variable raiz contendrá el cálculo de la raiz cuadrada y se define por defecto como variable real o decimal y potencia contendrá el resultado de elevar a 8 el número 2. Los resultados obtenidos son:

```
La raiz cuadrada de 25 es 5.0
2^8 = 256
```

Las principales ventajas de utilizar funciones son:

- **Código reutilizable**. Podemos llamar a la misma función tantas veces en nuestro programa como necesitemos, lo que hace que nuestro código sea reutilizable.
- **Código legible**. Las funciones nos ayudan a dividir nuestro código en trozos para que nuestro programa sea mas legible y fácil de entender.

Módulos en Python

A medida que nuestro programa crece, puede contener muchas líneas de código. En lugar de poner todo en un solo archivo, podemos utilizar módulos para separar por funcionalidad los códigos en varios archivos. Esto hace que nuestro código quede organizado y sea más fácil de mantener.

Un módulo es un archivo que contiene código para realizar una tarea específica. Un módulo puede contener variables, funciones, clases, etc. Veamos un ejemplo, vamos a crear un módulo escribiendo algo como lo siguiente:

```
#Definición del módulo suma


def sumar(a, b):


    resultado = a + b
    return resultado
```

Guardamos este programa en un archivo, por ejemplo `modulo_sumar.py` y tendremos definida una función de nombre `sumar` en ese módulo. La función recibe dos valores y devuelve la suma.

Cuando, en un programa diferente, queramos sumar dos números podemos importar la definición creada utilizando la palabra reservada `import`. Para acceder a la función definida en el módulo tenemos que utilizar el operador `.` (punto). Se parece mucho a que el módulo es una clase y la función una instancia de esa clase.

```
# Programa de sumas
import modulo_sumar


modulo_sumar.sumar(4, 5) #devolverá 9
```

Python tiene mas de 200 módulos estándar que pueden ser importados de la misma manera que importamos los módulos definidos por nosotros. En la documentación de Python en español encontramos la referencia a La biblioteca estándar de Python.

Números aleatorios

Este módulo está basado en el módulo `random` de la librería estándar de **Python**. Contiene funciones para generar comportamientos aleatorios.

Para acceder a este módulo es necesario:

```
import random
```

Vamos a ver sus funciones a continuación.

- `.getrandbits(n)`. Retorna un entero con "n" bits aleatorios. La función generadora devuelve como máximo 30 bits, por lo tanto "n" tiene que estar comprendido entre 1 y 30.

```
random.getrandbits(n)
```

* `.seed(n)`. *Inicializa el generador de números aleatorios con un número entero conocido "n". Esto le proporcionará una aleatoriedad determinista reproducible a partir de un estado inicial dado (n).*

```
random.seed(n)
```

- **`.randint(a, b)`**. Devuelve un entero aleatorio **N** tal que a ≤N≤ ba ≤N≤ b.

```
random.randint(a, b)
```

- **`.randrange(stop)`**. Devuelve un número entero seleccionado aleatoriamente entre cero y stop, que no está incluido.

```
random.randrange(stop)
```

- **`.randrange(start, stop)`**. Devuelve un número entero seleccionado aleatoriamente comprendido entre start y stop. El límite stop no está incluido.

```
random.randrange(start, stop)
```

- **`.randrange(start, stop, step)`**. Devuelve un número entero aleatorio entre start y stop separando los valores posibles entre si la distancia establecida por step. Por ejemplo `randrange(3, 30, 5)` devolverá un valor aleatorio de los siguientes posibles: 3, 8, 13, 18, 23, 28.

```
random.randrange(start, stop, step)
```

- **`.choice(secuencia)`**. Devuelve un elemento aleatorio de 'secuencia' que no puede estar vacía. Si 'secuencia' está vacía, genera in `IndexError`.

```
random.choice(secuencia)
```

- **`.random()`**. Devuelve un número aleatorio en coma flotante en el rango [0.0, 1.0).

```
random.random()
```

- **`.uniform(a, b)`**. Devuelve un número aleatorio de coma flotante **N** tal que a≤N≤ba≤N≤b para a≤ba≤b y b≤N≤ab≤N≤a para b<ab<a.

```
random.uniform(a, b)
```

En la imagen vemos ejemplos ejecutados en la shell.

main.py | Save | Run | Shell | Clear

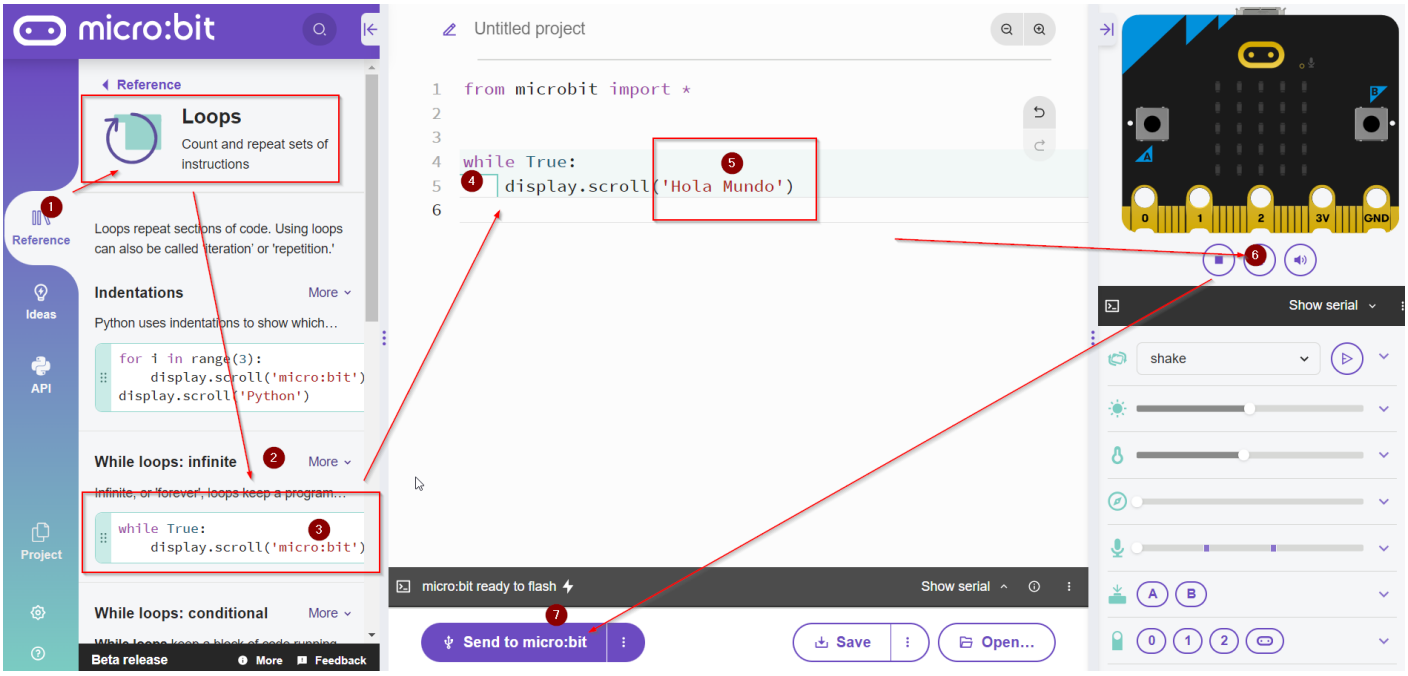
```
1  import random
2  r1 = random.getrandbits(15)
3  r2 = random.randint(10, 40)
4  r3 = random.randrange(3)
5  r4 = random.randrange(3, 5)
6  r5 = random.randrange(3, 30, 5)
7  frutas = ['pera', 'manzana', 'plátano',
       'ciruelas', 'sandia', 'melon']
8  r6 = random.choice(frutas)
9  r7 = random.random()
10 r8 = random.uniform(5, 15)
11 r9 = random.uniform(30, 20)
12 print('.getrandbits(n):', r1)
13 print('.randint(a, b):',r2)
14 print('.randrange(stop):', r3)
15 print('.randrange(start, stop):' ,r4)
16 print('.randrange(start, stop, step):'
       ,r5)
17 print('.choice(secuencia):' ,r6)
18 print('.random():' ,r7)
19 print('.uniform(a, b), (a<=b):' ,r8)
20 print('.uniform(a, b), (b<a):' ,r9)
```

```
.getrandbits(n): 25299
.randint(a, b): 17
.randrange(stop): 1
.randrange(start, stop): 4
.randrange(start, stop, step): 8
.choice(secuencia): ciruelas
.random(): 0.7110020050094659
.uniform(a, b), (a<=b): 11.866913706801071
.uniform(a, b), (b<a): 28.918703017544235
>
```

Solo placa: Hola Mundo

No hay mejor manera para empezar que este sencillo programa

1. Entramos en https://python.microbit.org/
2. Nos vamos a la pestaña de **Reference - Loops** y arrastramos el código de **While loops infinite**
3. Cambiamos el texto por "**Hola Mundo**" y lo simulamos en el microbit virtual de la izquierda
4. ¿Lo ha hecho bien? pues conecta tu microbit a tu ordenador, y **Sent to Microbit** te saldrá un diálogo pidiendo vincular tu microbit, acepta y ya esta !!!!



El código Pytho que ha subido a Microbit es el siguiente, la primera línea importa las librerías para manejar microbit, la segunda es el bucle While y al poner la condición true, se ejecutará siempre, y la instrucción que ejecuta es display.scroll donde visualiza en forma de marquesina el texto que pongamos, también puede ser un número.

```
from microbit import *



while True:
    display.scroll('Hola Mundo')
```

https://www.youtube.com/embed/sCpnOzJnutY

Solo placa: Imágenes

> *Extraído de Federico Coca Guia de Trabajo de Microbit CC-BY-SA*

API: *Display*

Control de la matriz de 5x5 LEDs que en micro:bit se conoce como pantalla. Los métodos de la clase son:

```
display.get_pixel(x, y) #1
display.set_pixel(x, y, val) #2
display.clear() #3
display.show(image, delay=0, wait=True, loop=False, clear=False) #4
display.scroll(string, delay=400) #5
'''
1 Obtiene el brillo [0 (apagado) a 9 (máx))] del pixel (x,y)
2 Establece el brillo [0 (apagado) a 9 (máx))] del pixel (x,y)
3 Borra (apaga) la pantalla
4 Muestra la imagen
5 Desplaza una cadena por la pantalla a la velocidad en ms del *delay*
'''
```

En ambos casos de la API existen otras muchas opciones no incluidas. La funcionalidad de autocompletar nos ayudará para no tener que recordar la sintaxis y conocer las que no aparece aquí. En la animación siguiente vemos un ejemplo de ambos casos.

logicos.png

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Imágenes prediseñadas

MicroPython nos ofrece muchas imágenes integradas para mostrar por pantalla y podemos crear efectos interesantes.

```
from microbit import *

while True:
  display.show(Image.HEART)
  sleep(500)
  display.show(Image.HEART_SMALL)
  sleep(500)
```

Las opciones son múltiples:

```
Image.HEART
Image.HEART_SMALL
Image.HAPPY
Image.SMILE
Image.SAD
Image.CONFUSED
Image.ANGRY
Image.ASLEEP
Image.SURPRISED
Image.SILLY
Image.FABULOUS
Image.MEH
Image.YES
Image.NO
Image.CLOCK12,Image.CLOCK11,Image.CLOCK10,Image.CLOCK9,Image.CLOCK8,Image.CLOCK7,Image.CLOCK6,
Image.CLOCK5,Image.CLOCK4,Image.CLOCK3,Image.CLOCK2,Image.CLOCK1
Image.ARROW_N,Image.ARROW_NE,Image.ARROW_E,Image.ARROW_SE,Image.ARROW_S,Image.ARROW_SW,Image.A
RROW_W,Image.ARROW_NW
Image.TRIANGLE
Image.TRIANGLE_LEFT
Image.CHESSBOARD
Image.DIAMOND
Image.DIAMOND_SMALL
Image.SQUARE
Image.SQUARE_SMALL
Image.RABBIT
```

```
Image.COW
Image.MUSIC_CROTCHET
Image.MUSIC_QUAVER
Image.MUSIC_QUAVERS
Image.PITCHFORK
Image.PACMAN
Image.TARGET
Image.TSHIRT
Image.ROLLERSKATE
Image.DUCK
Image.HOUSE
Image.TORTOISE
Image.BUTTERFLY
Image.STICKFIGURE
Image.GHOST
Image.SWORD
Image.GIRAFFE
Image.SKULL
Image.UMBRELLA
Image.SNAKE Image.ALL_CLOCKS Image.ALL_ARROW
```

Imágenes DIY

Es perfectamente posible crear nuestras propias imágenes configurando cada Pixel o LED de la pantalla. También es posible crear animaciones con imágenes.

Crear nuestras propias imágenes va a resultar una tarea sencilla cuando conozcamos la información para hacerlo. Cada pixel (LED) de la pantalla se puede configurar con diez valores que pueden tomar un valor entre 0 (cero) y 9 (nueve). Cuando le damos valor 0 (cero) es decirle literalmente que el brillo es nulo y sin embargo cuando le damos el valor 9 (nueve) lo ponemos al máximo de brillo posible. Podemos jugar con todos los valores intermedios para crear niveles de brillo.

La forma mas sencilla de definir una imagen consiste en utilizar la *clase microbit.Image* para crearla a partir de una cadena o string que devuelva el pictograma. Es decir utilizando el comando *Image(string)* teniendo que constar de dígitos con los valores 0 a 9 indicados. Para verlo rápidamente hacemos el ejemplos de dibujar una X en relieve asignándola a una variable.

```
mi_imagen_X = Image("90009:"
                    "06060:"
                    "00300:"
                    "06060:"
                    "90009")
```

Los dos puntos indican un salto de línea por lo que se puede usar el ASCII no imprimible "\n" que es precisamente eso, un salto de línea.

```
mi_imagen_X = Image("90009\n"
                    "06060\n"
                    "00300\n"
                    "06060\n"
                    "90009")
```

Los valores de brillo dan la sensación de relieve de profundidas a la X.

En cualquier caso esto no se escribe normalmente así, salvo para hacer mas o menos un gráfico del pixelado, sino en una sola línea.

```
mi_imagen_X = Image("90009\n06060\n00300\n06060\n90009")
```

Ahora parece mas elegante utilizar los dos puntos como indicador de salto de línea.

```
mi_imagen_X = Image("90009:06060:00300:06060:90009")
```

En la imagen vemos el resultado de lo explicado.

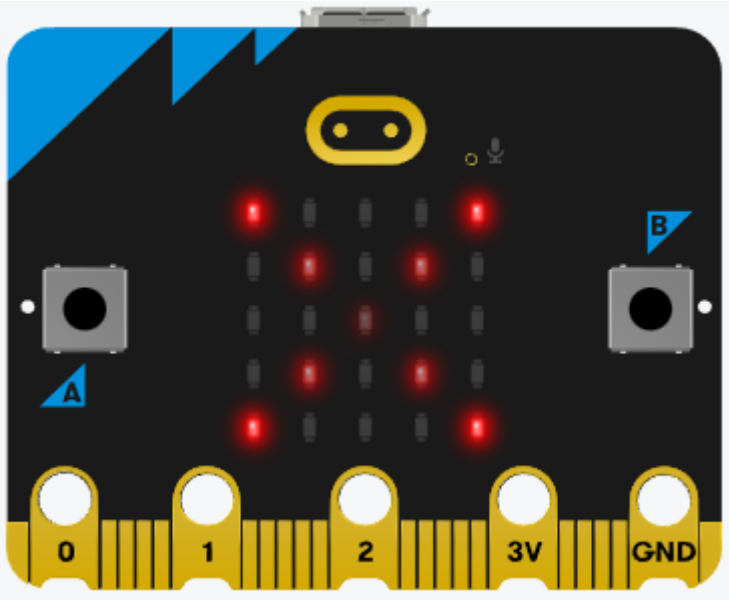


Imagen de una X en relieve

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Este es el código creado:

```
from microbit import *
"""mi_imagen_X = Image("90009\n"
                       "06060\n"
                       "00300\n"
                       "06060\n"
                       "90009")"""
#mi_imagen_X = Image("90009\n06060\n00300\n06060\n90009")
mi_imagen_X = Image("90009:06060:00300:06060:90009")
display.show(mi_imagen_X)
```

Animar imágenes

En micro:bit Python ya disponemos de un par de listas de imágenes incorporadas que se llaman

```
Image.ALL_Clocks
Image.ALL_ARROWS
```

Estas dos ordenes hacen que MicroPython entienda que necesita mostrar cada imagen de la lista, una tras otra.

Cuando queremos mostrar en la pantalla una imagen se nos muestra la siguiente ayuda contextual:

Ayuda contextual para display.show()

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

donde nos indica claramente que **image** puede ser una cadena, un número, una imagen o una lista de imágenes. Además aparecen las opciones que podemos configurar.

Con esta información crear un "reloj" que esté continuamente marcando cada hora es bastante sencillo, basta con poner el siguiente código y darle a simular.

```
# Imports go at the top
from microbit import *
display.show(Image.ALL_CLOCKS, delay=400, loop=True)
```

En la animación vemos el funcionamiento de este "reloj".

ayuda_disp_show.png

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Si cambiamos el reloj por las flechas veremos como van rotando flechas en ángulos de 45 grados.

ayuda_disp_show.png

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Para animar nuestras propias imágenes tendremos que crear cada una sobre un lienzo de 5x5 pixeles y establecer las diferencias para crear la animación. Podemos crear tantas imágenes como creamos oportuno. Creamos una lista con todas las imágenes en el orden que se tienen que reproducir y ya podemos mostrar nuestra lista en la pantalla.

En la animación siguiente vemos un efecto creado de esta forma.

ayuda_disp_show.png

Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Este es el código para crear la animación.

```
# Imports go at the top
from microbit import *
display.clear()
cor1=Image("90000:90000:90000:90000:90000")
cor2=Image("79000:79000:79000:79000:79000")
cor3=Image("57900:57900:57900:57900:57900")
cor4=Image("35790:35790:35790:35790:35790")
cor5=Image("13579:13579:13579:13579:13579")
cor6=Image("01357:01357:01357:01357:01357")
cor7=Image("00135:00135:00135:00135:00135")
cor8=Image("00013:00013:00013:00013:00013")
cor9=Image("00001:00001:00001:00001:00001")
cor10=Image("00000:00000:00000:00000:00000")
todas_las_cortinas=[cor1,cor2,cor3,cor4,cor5,cor6,cor7,cor8,cor9,cor10]
display.show(todas_las_cortinas, delay=100, loop=True)
```

Funciones para la pantalla

- `microbit.display.get_pixel(x, y)`. Devuelve el brillo del LED en la columna x y la fila y como un número entero entre 0 (apagado) y 9 (brillante).
- `microbit.display.set_pixel(x, y, value)`. Establece el brillo del LED en la columna x y la fila y como un número entero entre 0 y 9.
- `microbit.display.clear()`. Apaga (pone el brillo a 0) todos los LEDs.
- `microbit.display.show(image)`. Muestra la imagen.
- `microbit.display.show(image, delay=400, *, wait=True, loop=False, clear=False)`. Si `image` es una cadena, un real o un entero, muestra las letras/dígitos en secuencia. De lo contrario, si `image` es una secuencia iterable de imágenes, muestra estas imágenes en secuencia. Cada letra, dígito o imagen se muestra con un `delay` de milisegundos entre ellos.

Si `wait` es `True`, esta función se bloqueará hasta que la animación termine, de lo contrario la animación ocurrirá en segundo plano.

Si `loop` es `True`, la animación se repetirá para siempre.

Si `clear` es `True`, la pantalla se borrará después de que las iteraciones hayan terminado.

Los argumentos `wait`, `loop` y `clear` deben especificarse utilizando su palabra clave.

- `microbit.display.scroll(text, delay=150, *, wait=True, loop=False, monospace=False)`. Desplaza el texto horizontalmente en la pantalla. Si el texto es un número entero o flotante, se convierte primero en una cadena mediante str(). El parámetro delay controla la velocidad de desplazamiento del texto.

Si `wait` es `True`, esta función se bloqueará hasta que la animación termine, de lo contrario la animación ocurrirá en segundo plano.

Si `loop` es `True`, la animación se repetirá para siempre.

Si `monospace` es `True`, todos los caracteres ocuparán 5 columnas de píxeles de ancho, de lo contrario habrá exactamente 1 columna de píxeles en blanco entre cada carácter mientras se desplazan.

Los argumentos `wait`, `loop` y `monospace` deben especificarse utilizando su palabra clave.

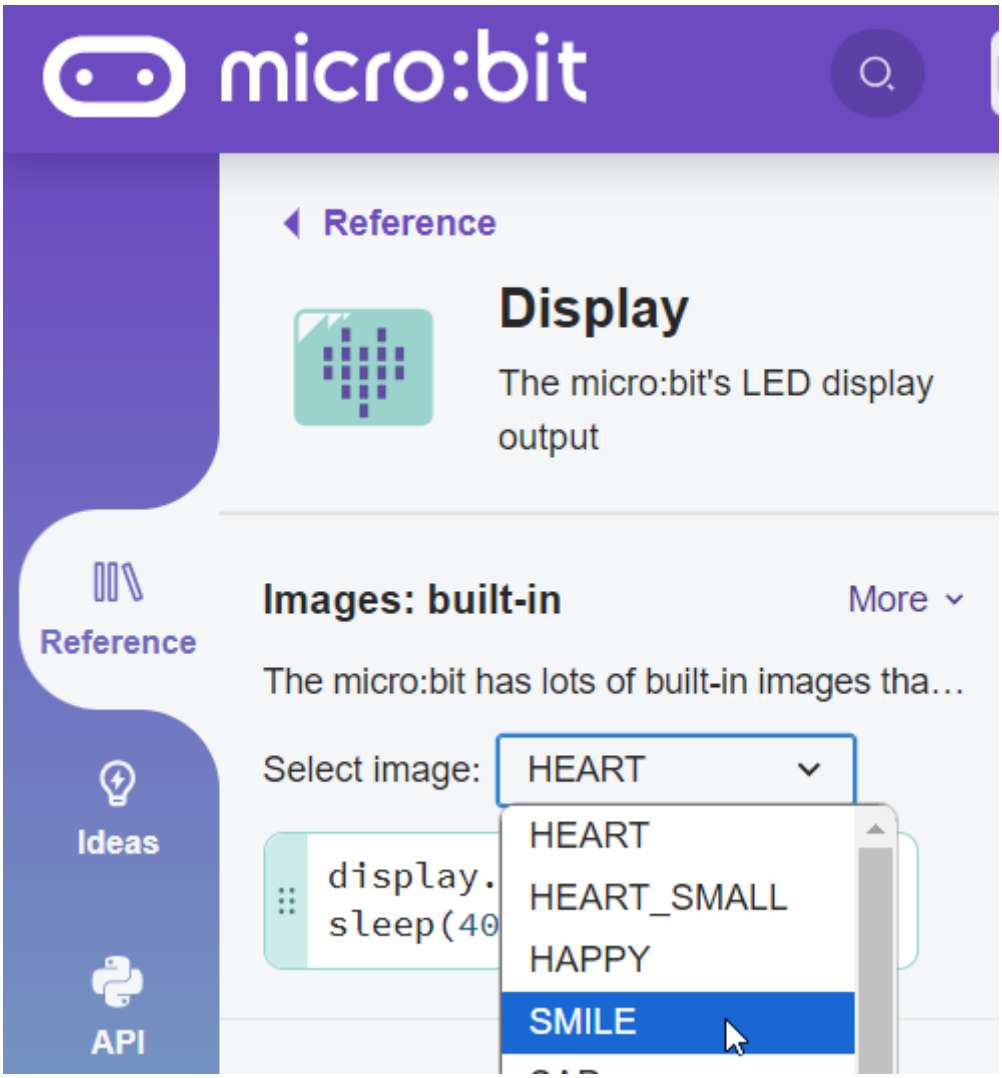
- `microbit.display.on()`. Enciende la pantalla.
- `microbit.display.off()`. Apaga la pantalla. Esto permitirá reutilizar los pines GPIO asociados a la pantalla para otros fines.
- `microbit.display.is_on()`. Devuelve `True` si la pantalla está encendida, en caso contrario devuelve `False`.
- `microbit.display.read_light_level()`. Utiliza los LEDs de la pantalla en modo de polarización inversa para detectar la cantidad de luz que incide sobre la pantalla. Devuelve un número entero entre 0 (oscuridad) y 255 (máximo brillo) que representa el nivel de luz.

> *Extraído de Federico Coca Guia de Trabajo de Microbit CC-BY-SA*

Solo placa: Imágenes estáticas y animadas

Imágenes estáticas

Sin tocar el código anterior, vamos ahora a **Display** y arrastramos el código de **sonrisa**



Vamos a tocar el código para que quede de esta forma, de esta manera practicamos la edición de código

```
# Imports go at the top
from microbit import *



# Code in a 'while True:' loop repeats forever
while True:
    display.show(Image.SMILE)
    sleep(1000)
    display.scroll('Hola mundo')
```

La sonrisa se ve un segundo = 1.000 mseg y luego ejecuta el scroll

> ¿¿ Qué esperas para experimentar otras imágenes ?

Imágenes animadas

Podemos poner las imágenes prediseñadas en la variable **Image** pero también podemos crearlas fácilmente. En el siguiente programa se define qué led de la matriz 5x5 se enciende a la máxima intensidad (0-9)

> Avanzando en la programación, se definen 5 variables **catedux** tipo imagen, y se define una variable **all_catedus** que es tipo array que contienen todas.

```
from microbit import *

catedu1 =  Image("00900:"
                 "09000:"
                 "90000:"
                 "09000:"
                 "00900")

catedu2 =  Image("09000:"
                 "90000:"
                 "09000:"
                 "00900:"
                 "00090")
```

```
catedu3 =  Image("90000:"
                 "09000:"
                 "00900:"
                 "00090:"
                 "00009")

catedu4 =  Image("00009:"
                 "00090:"
                 "00900:"
                 "09000:"
                 "90000")

catedu5 =  Image("00090:"
                 "00900:"
                 "09000:"
                 "90000:"
                 "09000")

all_catedus = [catedu1,catedu2,catedu3,catedu2,catedu1,catedu5,catedu4]
while(True):
    display.show(all_catedus, delay=200)
```

https://www.youtube.com/embed/yLjKgy2NaPI

O jugar con las intensidades: En este juego de **luces del coche fantástico** se utiliza la intensidad media 5 :

> Este ejemplo de regular la intensidad del led es imposible de realizar en programación por bloques.

```
from microbit import *
```

```
catedu1 =  Image("00005:"
                 "00000:"
                 "00000:"
                 "00000:"
                 "00000")

catedu2 =  Image("00009:"
                 "00050:"
                 "00000:"
                 "00000:"
                 "00000")

catedu3 =  Image("00005:"
                 "00090:"
                 "00500:"
                 "00000:"
                 "00000")

catedu4 =  Image("00000:"
                 "00050:"
                 "00900:"
                 "05000:"
                 "00000")

catedu5 =  Image("00000:"
                 "00000:"
                 "00500:"
                 "09000:"
                 "50000")

catedu6 =  Image("00000:"
                 "00000:"
                 "00000:"
                 "05000:"
                 "90000")
```

```
catedu7 =  Image("00000:"
                 "00000:"
                 "00000:"
                 "00000:"
                 "50000")

all_catedus =
[catedu1,catedu2,catedu3,catedu4,catedu5,catedu6,catedu7,catedu6,catedu5,catedu4,catedu3,cated
u2]
while(True):
    display.show(all_catedus, delay=100)
```

https://www.youtube.com/embed/jko4xAxbm2I

¿No sabes lo que es el coche fantástico? eso es que no tienes la edad adecuada para la robótica

https://www.youtube.com/embed/oNeQi8-PXAU

También podemos hacerlo pixel a pixel y no utilizar variables tipo array

```
from microbit import *

display.clear()
while(True):
    for n in range(0, 5):
        display.set_pixel(n, 3, 9)
        if (n<4):
            display.set_pixel(n+1, 3, 5)
        if (1<n):
            display.set_pixel(n-1, 3, 5)
        if (1<n):
            display.set_pixel(n-2, 3, 0)
        sleep(200)
```

```
    for n in reversed(range(0, 5)):
        display.set_pixel(n, 3, 9)
        if (n<4):
            display.set_pixel(n+1, 3, 5)
        if (1<n):
            display.set_pixel(n-1, 3, 5)
        if (n<3):
            display.set_pixel(n+2, 3, 0)
        sleep(200)
```

https://www.youtube.com/embed/a5J1793GCdg

Solo placa: Eventos para los botones

> Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Si trabajamos con versiones anteriores a V2 solamente disponemos de los botones A, B y A+B, pero si tenemos una versión V2 también disponemos del botón táctil incorporado en el logo, aunque a todos los efectos este se considera un pin de entrada.

El logo no es tratado exactamente como un botón, sino como un pin de nombre logo. En el borde existen otros tres pines, los 0, 1 y 2. Por ello la forma de trabajar con el logo va a ser un poco diferente, como veremos en la actividad A04.

La diferencia fundamental, ademas de la forma, es que el logo es un sensor capacitivo y los pines son sensores resistivos. En la práctica esto significa que el logo funciona simplemente tocandolo y los pines necesitan cerrar el circuito con GND, por lo que para que funcionen como pulsador debemos tocar tanto el pinto como GND.

Si queremos que MicroPython reaccione a los eventos de pulsación de los botones, debemos ponerlo en un bucle infinito y comprobar si el botón `is_pressed`.

- **Función** `is_pressed()`

Para trabajar con los botones de la micro:bit tenemos disponibles funciones que se han cargado al importar el módulo `microbit`. Estas funciones están basadas en la función genérica `is_pressed()` pensada para saber que tecla de un teclado se ha pulsado. Sin embargo, en el caso de MicroPython a para micro:bit a estos botones se les ha asignado un nombre a cada uno, `button_a` para el A y `button_b` para el B, de manera que para usarlos se llama al botón y con el operador `.` a la función `is_pressed()`. Por ejemplo, `button_a.is_pressed()` es el código encargado de saber si estamos pulsando el botón A y `button_b.is_pressed()` si lo es el B.

- **Función** `get_pressed()`

Esta función retorna el total acumulado de pulsaciones de botones y restablece este total a cero antes de volver. Es decir, podemos capturar el número de veces que hemos pulsado un botón. El valor de retorno es un número, por lo que, para mostrarlo en la pantalla de LEDs hay que

convertirlo en cadena con la función `str()`.

- **Función** `was_pressed()`

Devuelve `True` o `False` para indicar si se ha presionado el botón desde la última vez que se inicio el dispositivo o se llamó a este método. Llamar a este método borra el estado de que ha sido pulsado, de modo que el botón debe pulsarse de nuevo antes de que este método vuelva a retornar `True`.

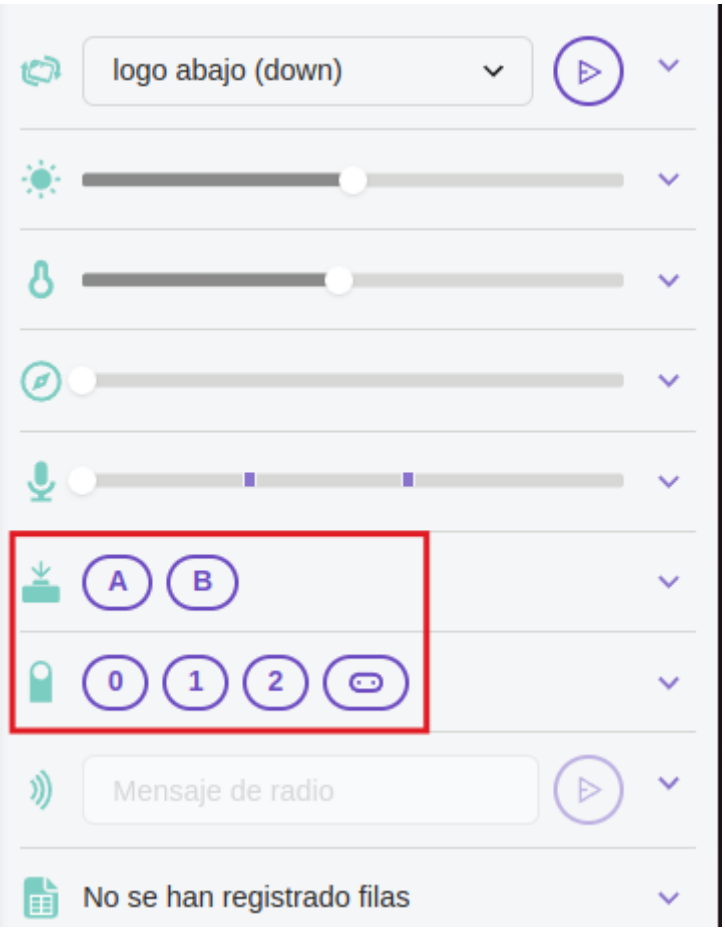
Vamos a hacer un ejemplo que aclarará mejor lo explicado. Se trata de crear un programa (le podremos de nombre Caritas_X) en el que mientras mantegamos pulsado el botón A se muestra una cara sonriente, si no se pulsa ningún botón se muestra una cara triste y si se pulsa el botón B la cara desaparece (se apagan todos los LEDs) y tras 2 segundos aparece una X que se va haciendo cada vez mas grande partiendo del punto central. Finalmente pasados otros 2 segundos el programa vuelve a empezar. El código es:

```
from microbit import *
while True:
    while True:
        if button_a.is_pressed():
            display.show(Image.HAPPY)
        elif button_b.is_pressed():
            break
        else:
            display.show(Image.SAD)

    display.clear()
    sleep(2000)
    mi_X_peque = Image("00000:00000:00900:00000:0000")
    display.show(mi_X_peque)
    sleep(200)
    mi_X_media = Image("00000:09090:00900:09090:0000")
    display.show(mi_X_media)
    sleep(200)
    mi_X_grande = Image("90009:09090:00900:09090:90009")
    display.show(mi_X_grande)
    sleep(2000)
```

En la animación siguiente vemos como funciona

f_ifelifelse.png

Si observamos con cuidado apreciaremos que en algún momento se accionan los botones A y B pero los que aparecen en la parte inferior, debajo de la pantalla de simulación. Están al lado de un logotipo que indica que se pulsen con una flechita. Justo debajo de estos aparecen los citados del borde de placa y el logo junto a ellos, pues es tratado asi, como un pin, y además a su izquierda hay un candado cerrado indicativo de que no se está usando ninguno de ellos. En la imagen siguiente se ve mejor lo indicado.



Vamos a crear otro ejemplo en el que se cuenten las veces que pulsamos el botón A o el botón B durante un tiempo de 3 segundo. El programa es el siguiente:

```
from microbit import *


sleep(3000) #Espera de 3 segundos
```

```
#Convertimos número a cadena con str()
pulsado = str(button_b.get_presses())

display.show(pulsado)

# Por si hemos pulsado mas de 9 veces
display.scroll(pulsado)
```

En la 'Referencia' del compilador, dentro de Botones tenemos un ejemplo que nos indica el botón que hemos pulsado con cuatro opciones posibles, el A, el B, A o B y finalmente A y B. Animamos a cargarlos y probarlos para familiarizarnos todo lo posible con ellos.

> Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Solo placa: Botones

Los botones pueden dar juego, combinándolos con la instrucción if --- else

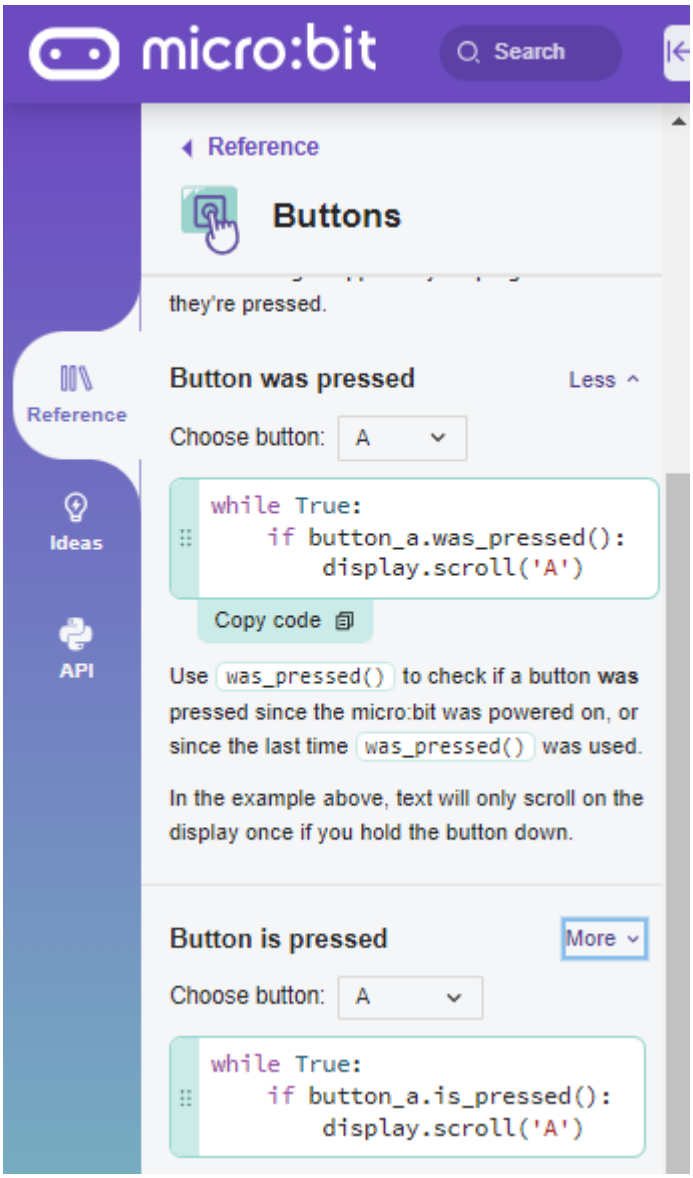
```
from microbit import *

while True:
    if button_a.is_pressed():
        display.show(Image.HAPPY)
    elif button_b.is_pressed():
        break
    else:
        display.show(Image.SAD)

display.clear()
```

Extraido de tutorial https://microbit-micropython.readthedocs.io/en/v2-docs/tutorials/buttons.html

https://www.youtube.com/embed/VgimuhHlRgQ

> ¿Qué pasa si pulsamos el botón B ?

En el apartado **Reference** podemos ir a **Buttons** tenemos diferentes muestras de código :



La diferencia entre este código

```
while True:
    if button_a.was_pressed():
        display.scroll('A')
```

y este otro código

```
while True:
    if button_a.is_pressed():
```

```
        display.scroll('A')
```

es muy sutil, no hay diferencia si apretamos el botón A *excepto si lo mantenemos pulsado*

El siguiente código, visualiza el número de veces que pulsas el botón A durante 3 segundos :

```
from microbit import *
display.scroll('Press A')
sleep(3000)
display.scroll(button_a.get_presses())
```

Solo sensores de la placa

Podríamos continuar, pero **solo con la placa** podemos hacer muchos más programas con Python

Nivel de luz

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA Vamos a ver como utilizar ...

Temperatura

La función en micropython para leer la temperatura de la placa en ºC interna y devuelve un valor ...

Magnetómetro

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA Este módulo permite accede...

Acelerómetro

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA Este objeto permite accede...

Micrófono

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA ATENCIÓN SÓLO VÁLIDO PARA ...

Radio

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA El módulo de radio permite...

Pines de Entrada/salida

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA En MicroPython, cada pin e...

Input output

Una manera rápida de probar las entradas y salidas de microbit es utilizar el código predefinido ...

Música predefinida o crea tu música

SALIDAS DE AUDIO La placa Microbit v2 tiene un altavoz incorporado que se puede anular o activar...

Musica

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA MicroPython de BBC micro:b...

Putty

Putty es un programa que nos permite realizar comunicaciones, normalmente se usa en protocolo SSH...

UART

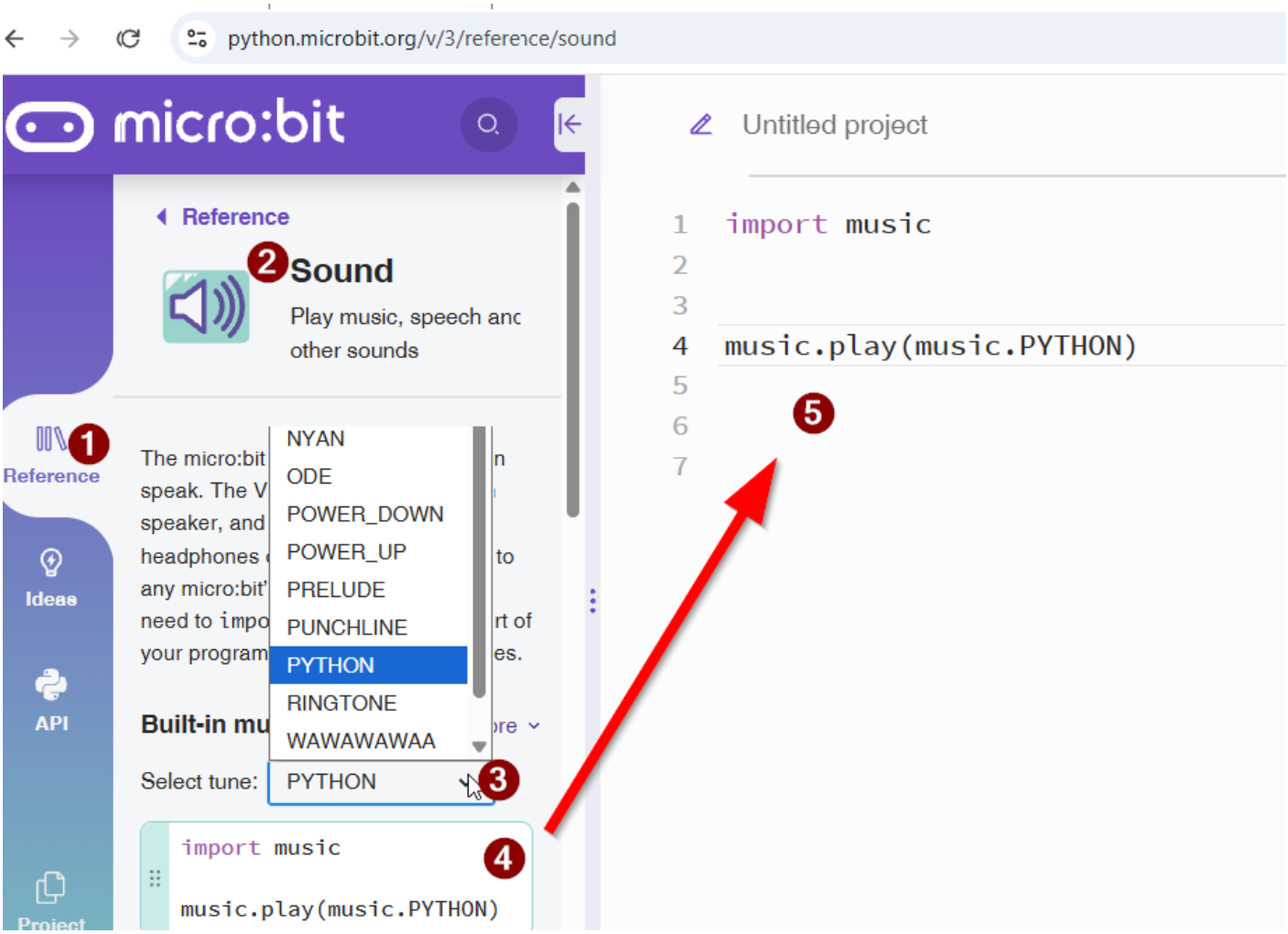
Página extraída de Federico Coca Guia de Trabajo de Microbit https://fgcoca.github.io/Guia-de-tr...

Registro de datos

Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA Grabar datos Para utiliza...

Solo placa: Música

Tan fácil como elegir la música y arrastrar el código:



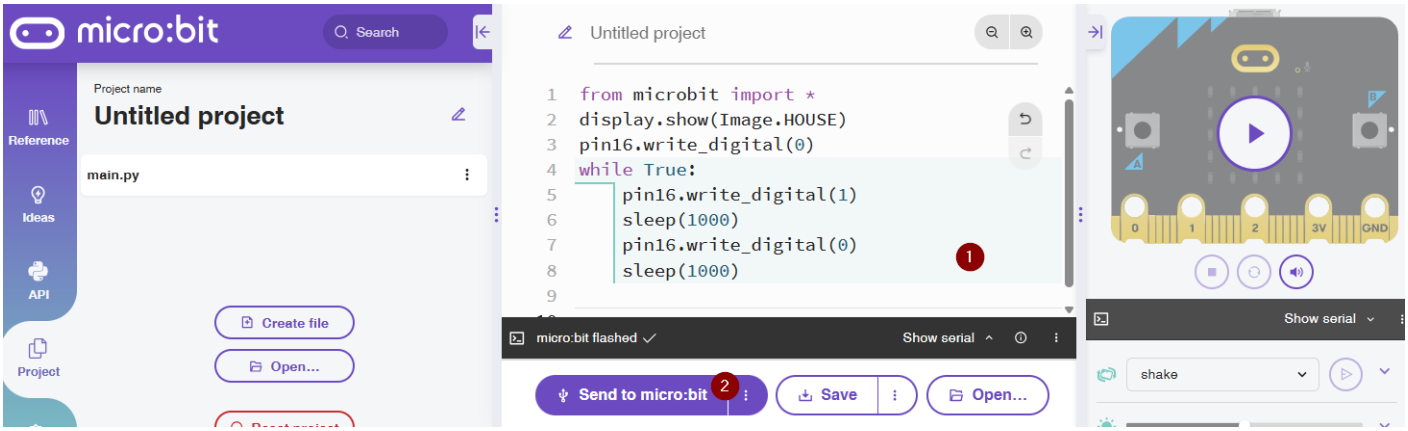
Si te sabe a poco aquí tienes más

- Música predefinida o crear tu música
- Más música

Maqueta: Intermitente led amarillo

Nos vamos a https://python.microbit.org/ y ponemos este código:

```
from microbit import *
display.show(Image.HOUSE)
pin16.write_digital(0)
while True:
    pin16.write_digital(1)
    sleep(1000)
    pin16.write_digital(0)
    sleep(1000)
```



https://www.youtube.com/embed/fjCu82BzQE4

Maqueta: LED amarillo intermitente gradual

Vamos a hace lo mismo pero que el brillo vaya del mínimo 0 al máximo 255 y viceversa, aprovechando las señales PWM

```
from microbit import *
display.show(Image.HOUSE)
pin16.write_digital(0)
brillo=0
while True:
    for brillo in range (0, 255):
        pin16.write_analog(brillo)
        sleep(5)
    while brillo >0 :
        pin16.write_analog(brillo)
        brillo = brillo -1
        sleep(5)
```

https://www.youtube.com/embed/YvICMfKfx9Y

¿Qué es eso de señal PWM?

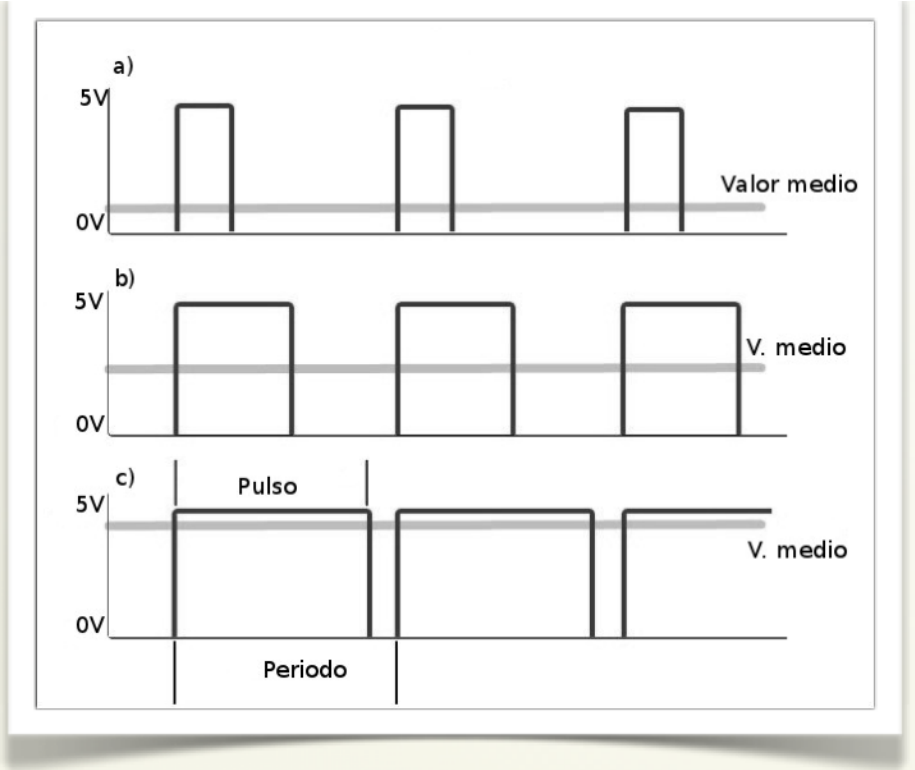
Arduino, ESP32, Micro:bit, PicoW... tienen **entradas** analógicas y digitales. Pero **salidas sólo digitales.**

Para **simular** una salida **analógica** entre 0V y 5V se utilizan señales digitales PWM. En Arduino sólo tiene 6 salidas pseudo-analógicas. En los pines digitales 3, 5, 6, 8, 10 y 11 son PWM

¿Qué es eso de PWM? La señal PWM *(Pulse Width Modulation, Modulación de Ancho de Pulso*) es una señal que utiliza el microcontrolador para generar una señal continua sobre el proceso a controlar. Por ejemplo, la variación de la intensidad luminosa de un led, el control de velocidad de un motor de corriente continua,...

Para que un dispositivo digital, microcontrolador de la placa Arduino, genere una señal continua lo que hace es emitir una señal cuadrada con pulsos de frecuencia constante y tensión de 5V. A continuación, variando la duración activa del pulso (ciclo de trabajo) se obtiene a la salida una señal continua variable desde 0V a 5V.

Veamos gráficamente la señal PWM:



Los pines digitales de la placa Arduino que se utilizan como salida de señal PWM generan una señal cuadrada de frecuencia constante (490Hz), sobre esta señal periódica por programación podemos variar la duración del pulso como vemos en estos 3 casos:

- La duración del pulso es pequeña y la salida va a tener un valor medio de tensión bajo, próximo a 0V.
- La duración del pulso es casi la mitad del período de la señal, por tanto, la salida va a tener un valor medio de tensión próximo a 2,5V.
- La duración del pulso se aproxima al tiempo del período y el valor medio de tensión de salida se aproxima a 5V.

Ejemplo en código ArduinoIDE y Arduino
Para ejecutar una señal PWM, es simplemente **analogWrite(analogOutPin, outputValor);** donde analogOutPin es el número del Pin PWM, acuérdate que sólo puede ser uno de estos 6 : **3, 5, 6, 8, 10 y 11 y** outpuValor es el valor de la señal PWM pero **ojo**

desde 0 a 255 es decir si quieres el valor de 0V tienes que poner 0, si quieres el valor de 5V tienes que poner 255 y si quieres poner un valor medio, haz una regla de tres, por ejemplo 2.5V tienes que poner 255/2=127 o 128 da igual

Otro ejemplo en Python con Micro:bit

pin16.write_analog(brillo) donde brillo puede ir de 0 a 255

Maqueta: Neopixel RGB

Vamos a hacer una discoteca !!!

```
from microbit import *
import neopixel
NEOPIXEL = neopixel.NeoPixel(pin14, 4)
from random import randint

while True:
   for index in range(0, 4):
        NEOPIXEL.clear()
        NEOPIXEL[index] = (randint(10, 255), randint(10, 255), randint(10, 255))
        NEOPIXEL.show()
        sleep(100)
```

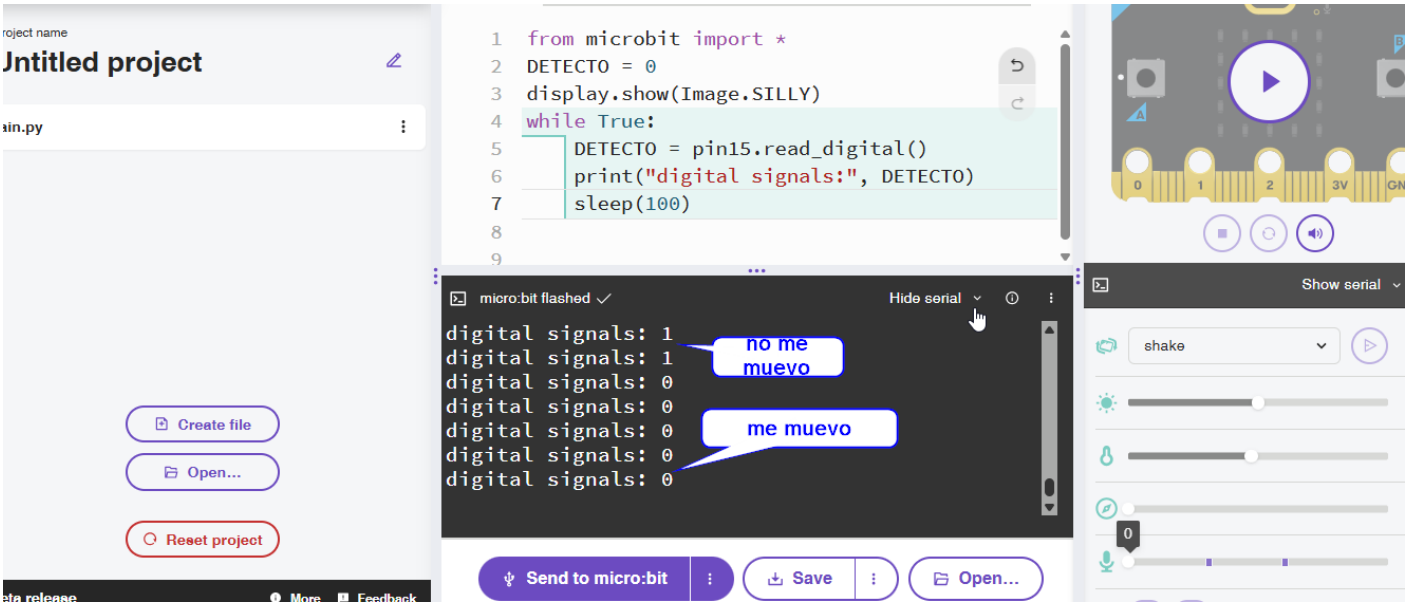
https://www.youtube.com/embed/rJDlR56GRLI

Maqueta : Sensor PIR

Vamos a visualizar la lectura del sensor PIR por el puerto serie:

```
from microbit import *
DETECTO = 0
display.show(Image.SILLY)
while True:
    DETECTO = pin15.read_digital()
    print("digital signals:", DETECTO)
    sleep(100)
```

El resultado :

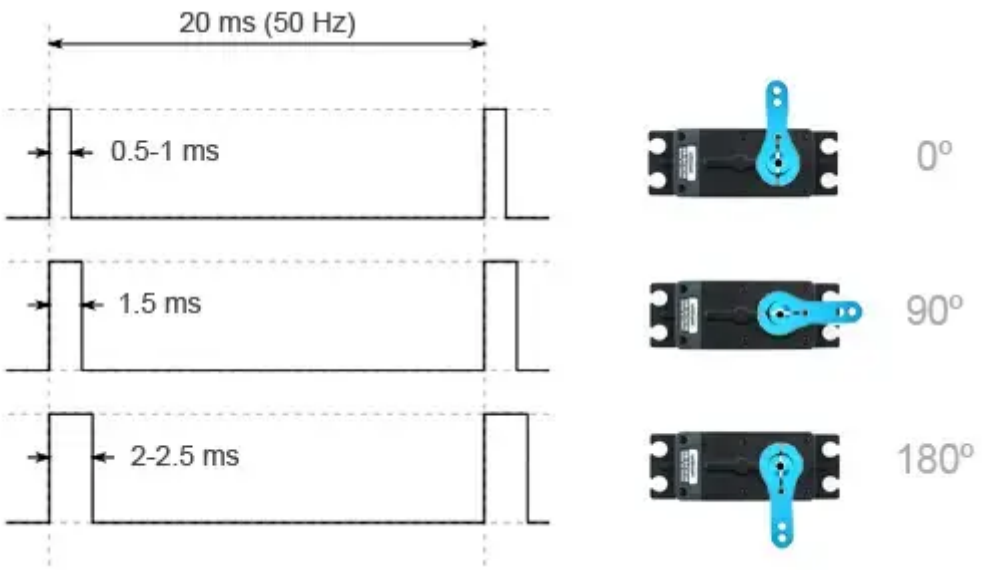


Servos

Hay varias opciones para manejar servos con micro:bit y python

Opción A: Lo más sencillo enviar un pulso adecuado

Los servos funcionan según la anchura del pulso que se envía, siendo los pulsos de 20mseg. Se explica mejor con una imagen :



Autor Luis Llamas CC-BY-SA https://www.luisllamas.es/controlar-un-servo-con-arduino/

1. Por lo tanto podríamos establecer primero pulsos de 20 mseg con la instrucción adecuada (por ejemplo en la pueta pin 8 sería **pin8.set_analog_period(20)**)
2. Enviar pulsos de forma adecuada. Ejemplo en puerta pin 8
    1. si queremos 0º enviamos pulsos de 1mseg que equivale a **pin8.write_analog(50)**
    2. si queremos 90º enviamos pulsos de 1.5mseg que equivale a **pin8.write_analog(75)**
    3. si queremos 180º enviamos pulsos de 2mseg que equivale a **pin8.write_analog(100)**

> Los valores no responden a una regla de tres (en teoría 20mseg serían 255 en formato PWM) sino a **la experiencia-ensayo-prueba-error**. Hemos probado que para la puerta los valores 50-75-100 son correctos. Para la ventana que se ve bien la apertura y cierre los valores son 30-60-100

Mas info en https://support.microbit.org/support/solutions/articles/19000101864-using-a-servo-with-the-micro-bit

Opción B Cargar una librería servo.py

1. Nos vamos a https://github.com/microbit-playground/microbit-servo-class y descargamos servo.py
2. Lo grabamos en la carpeta /mu_code/ donde se ha instalado el editor Mu
3. Utilizamos el código usando esta librería y poniendo los grados como grados

Por ejemplo para la puerta pin 8

```
sv1 = Servo(pin8)
sv1.write_angle(50) # turn servo to 50 degrees
```

> Si quieres saber cómo se instala una librería, consulta la página LCD que ahí se ha instalado una librería https://libros.catedu.es/books/smart-home-para-microbit/page/maqueta-lcd

Opción C Crea tu una librería en tu programa

Esta opción esta extraída del tutorial del fabricante https://docs.keyestudio.com/projects/KS4027-KS4028/en/latest/Python.html#project-6-servo

No explicamos el código pues se extiende de los objetivos del curso

```
from microbit import *

class Servo:
    def __init__(self, pin, freq=50, min_us=600, max_us=2400, angle=180):
        self.min_us = min_us
        self.max_us = max_us
        self.us = 0
        self.freq = freq
        self.angle = angle
        self.analog_period = 0
        self.pin = pin
        analog_period = round((1/self.freq) * 1000)  # hertz to miliseconds
```

```
        self.pin.set_analog_period(analog_period)

    def write_us(self, us):
        us = min(self.max_us, max(self.min_us, us))
        duty = round(us * 1024 * self.freq // 1000000)
        self.pin.write_analog(duty)
        sleep(100)
        self.pin.write_analog(0)

    def write_angle(self, degrees=None):
        if degrees is None:
            degrees = math.degrees(radians)
        degrees = degrees % 360
        total_range = self.max_us - self.min_us
        us = self.min_us + total_range * degrees // self.angle
        self.write_us(us)

Servo(pin8).write_angle(0)
display.show(Image.HAPPY)

while True:
        Servo(pin8).write_angle(0)
        sleep(1000)
        Servo(pin8).write_angle(45)
        sleep(1000)
        Servo(pin8).write_angle(90)
        sleep(1000)
        Servo(pin8).write_angle(135)
        sleep(1000)
        Servo(pin8).write_angle(180)
        sleep(1000)
```

Maqueta puerta

Utilizaremos la versión sencilla de manejo de los servos :

```
from microbit import *

pin8.set_analog_period(20) # pulsos de 20 milisegundos cada uno

while True:

  pin8.write_analog(50)  #equivale a 1mseg de pulso a la derecha
  sleep(1000)
  pin8.write_analog(75) #equivale a a 1.5mseg
  sleep(1000)
  pin8.write_analog(100) #equivale a 2mseg de pulso todo a la izquierda
  sleep(1000)
```

https://www.youtube.com/embed/rYYWD_RAx7o

Maqueta: Ventana

Para la ventana hemos usado el mismo código pero jugando, hemos visto que la ventana cierra mejor a valores más bajos

- **pin9.write_analog(30)** todo abierto
- **pin9.write_analog(60)** media ventana
- **pin9.write_analog(100)** ventana cerrada

el código

```
from microbit import *

pin9.set_analog_period(20) # pulsos de 20 milisegundos cada uno

while True:

  pin9.write_analog(30)  #equivale a 1mseg de pulso a la derecha
  sleep(1000)
  pin9.write_analog(60) #equivale a a 1.5mseg
  sleep(1000)
  pin9.write_analog(100) #equivale a 2mseg de pulso todo a la izquierda
  sleep(1000)
```

https://www.youtube.com/embed/WHRpQ1IjrBo

Maqueta: Motor

El motor tiene un sencillo funcionamiento:

PIN12	PIN13	MOTOR
0	0	PARADO
0	1	ROTACIÓN SENTIDO RELOJ
1	0	ROTACIÓN SENTIDO ANTIRELOJ
1	1	PARADO

Pero con Pytho no sólo podemos poner los pines 12 y 13 a 0 o 1 sino también podemos poner su potencia

Valora qué pasa en cada una de las 4 situaciones siguiente

```
from microbit import *

pin12.write_digital(0)
pin13.write_digital(0)

while True:
    # 1 Que pasa 1 ##########♥1
    display.scroll('1')
    pin12.write_digital(1)
    pin13.write_analog(50)
    sleep(5000)
    # Paramos
    pin12.write_digital(0)
    pin13.write_analog(0)
    sleep(1000)
    # 2 Que pasa 2  ##########♥2
    display.scroll('2')
    pin12.write_digital(1)
    pin13.write_analog(255)
```

```
    sleep(5000)
    # Paramos
    pin12.write_digital(0)
    pin13.write_analog(0)
    sleep(1000)
    # 3 Que pasa 3  ##########♥3
    display.scroll('3')
    pin12.write_digital(1)
    pin13.write_digital(0)
    sleep(5000)
    # Paramos
    pin12.write_digital(0)
    pin13.write_analog(0)
    sleep(1000)
    # 4 Que pasa 4  ##########♥4
    display.scroll('4')
    pin12.write_digital(1)
    pin13.write_digital(1)
    sleep(5000)
    # Paramos
    pin12.write_digital(0)
    pin13.write_analog(0)
    sleep(1000)
    # 5 Que pasa 5  ##########♥5
    display.scroll('5')
    pin12.write_digital(0)
    pin13.write_digital(1)
    sleep(5000)
    # Paramos
    pin12.write_digital(0)
    pin13.write_analog(0)
    sleep(1000)
```

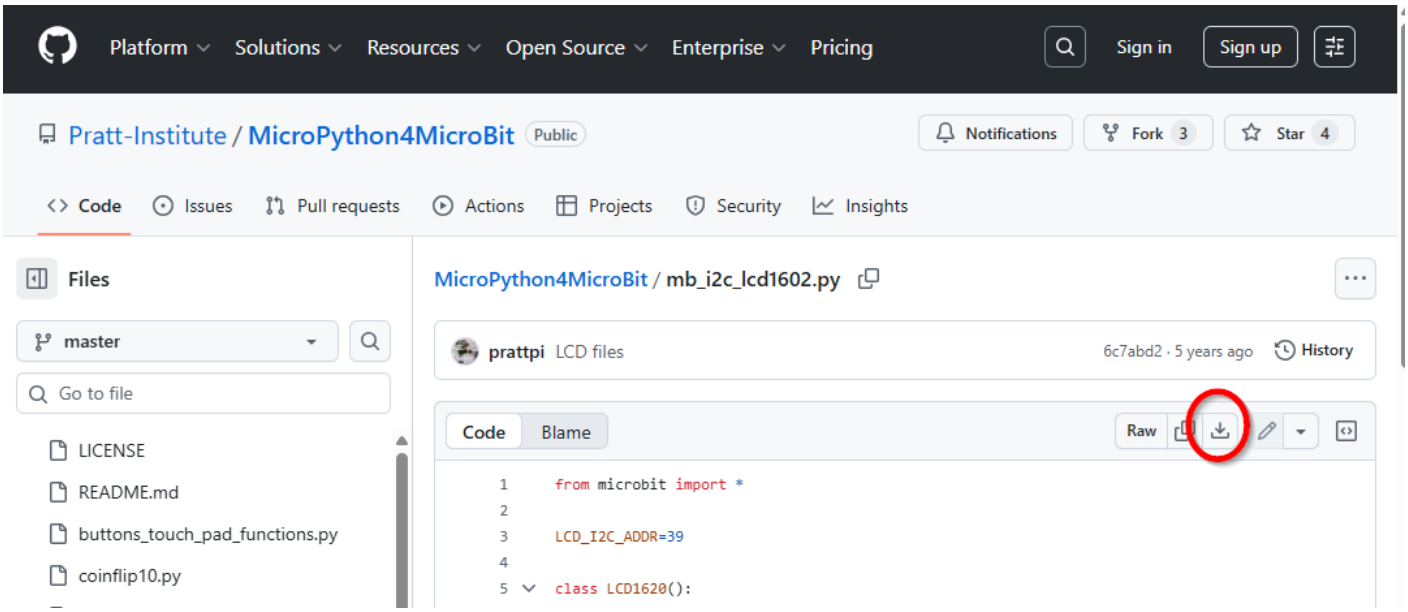
https://www.youtube.com/embed/LpngWgMuGgk

Maqueta: LCD

El display LCD no es nativo, y no hay una solución simple como en los servos (ver https://libros.catedu.es/books/smart-home-para-microbit/page/servos) luego tenemos que incorporar **UNA LIBRERIA EXTERNA** para LCS 16x2 (16 columnas 2 filas)

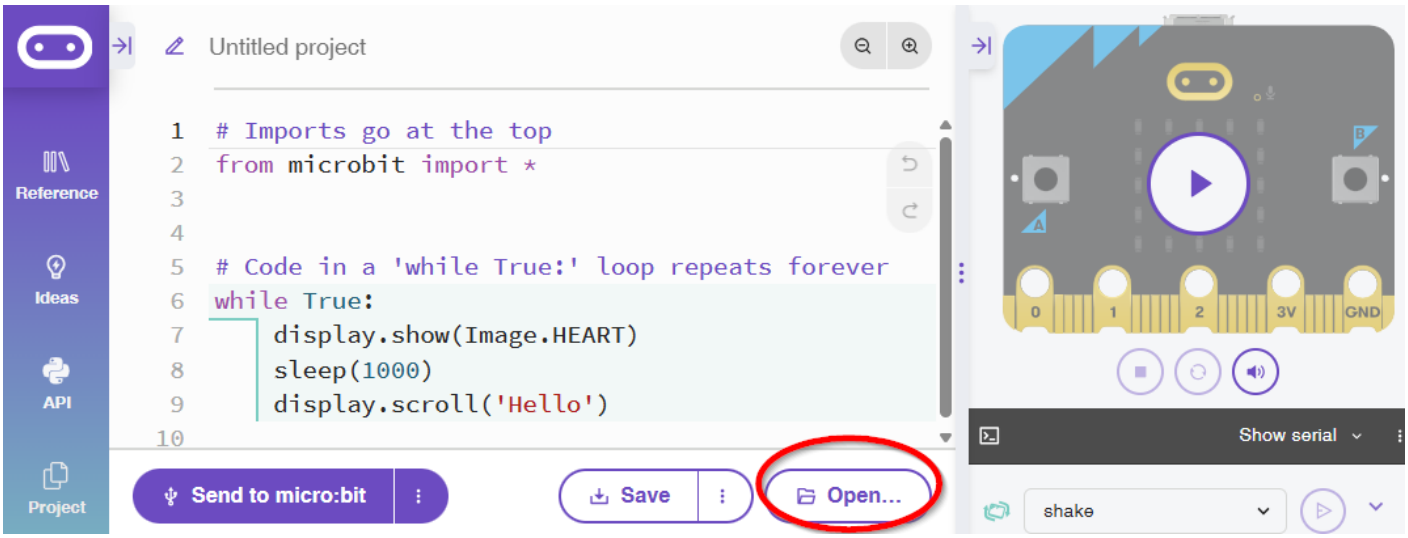
La librería mb_i2c_lcd1602.py

La puedes descargar aquí https://github.com/Pratt-Institute/MicroPython4MicroBit/blob/master/mb_i2c_lcd1602.py



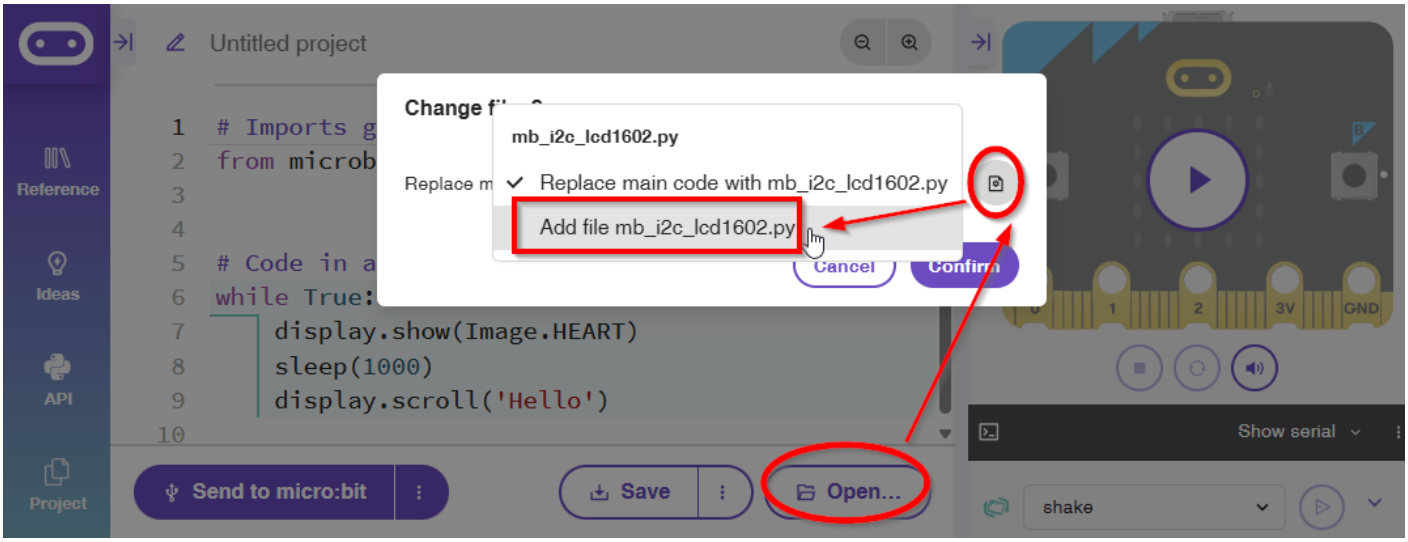
Vamos a utilizar el editor Python online https://python.microbit.org/

Abrimos un nuevo proyecto y le damos a **Open**



Abrimos el fichero **mb_i2c_lcd1602.py** que hemos descargado anteriormente

> nos pide si queremos reemplazar el contenido de main.py **LE DECIMOS QUE NO**, que añada un nuevo fichero mb_i2c_lcd1602.py **OJO QUE HAY QUE DAR AL ICONO PEQUEÑO** que pone en esa imagen



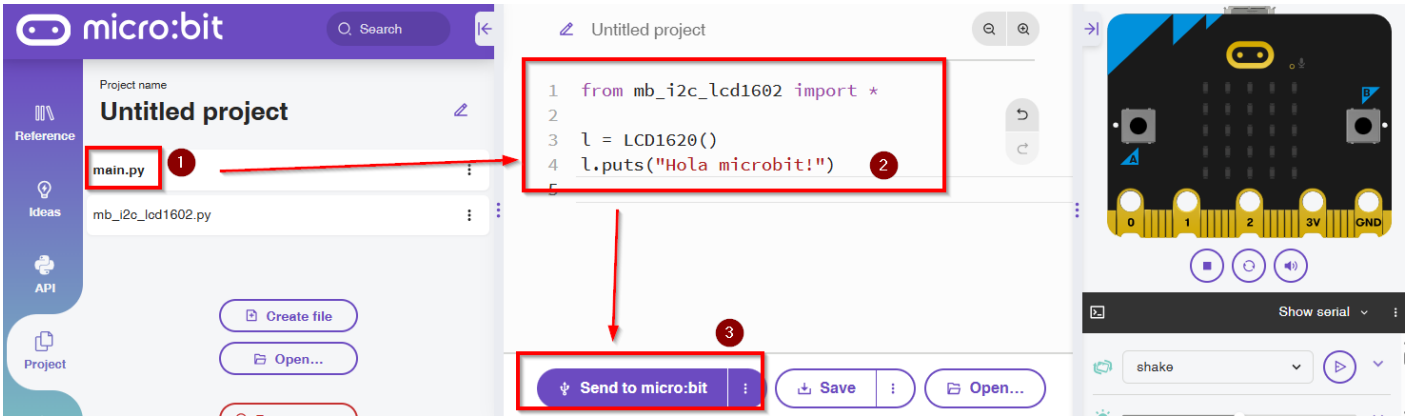
Nuestro programa

Confirmamos, vamos al **main.py** y pegamos este código

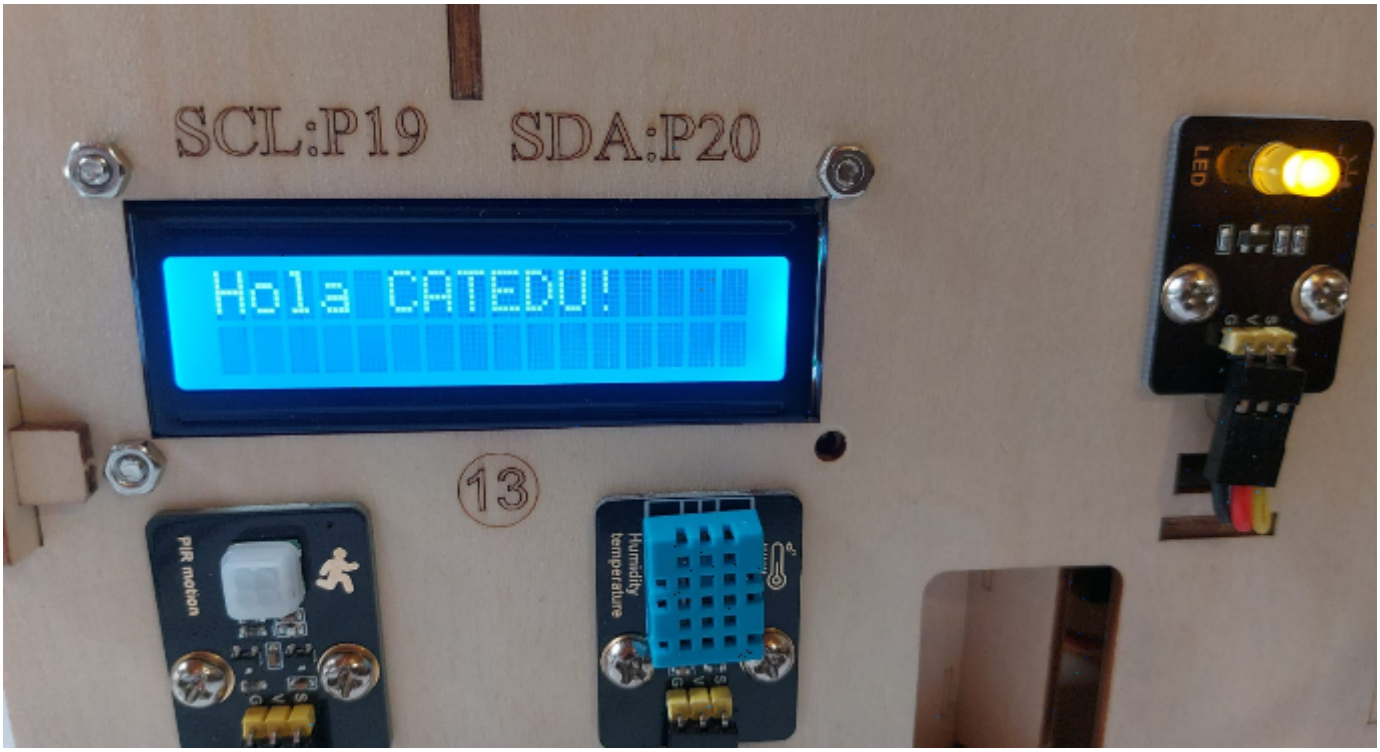
```
from mb_i2c_lcd1602 import *
```

```
l = LCD1620()
l.puts("Hola microbit!")
```

Y **send to micro:bit**



Resultado

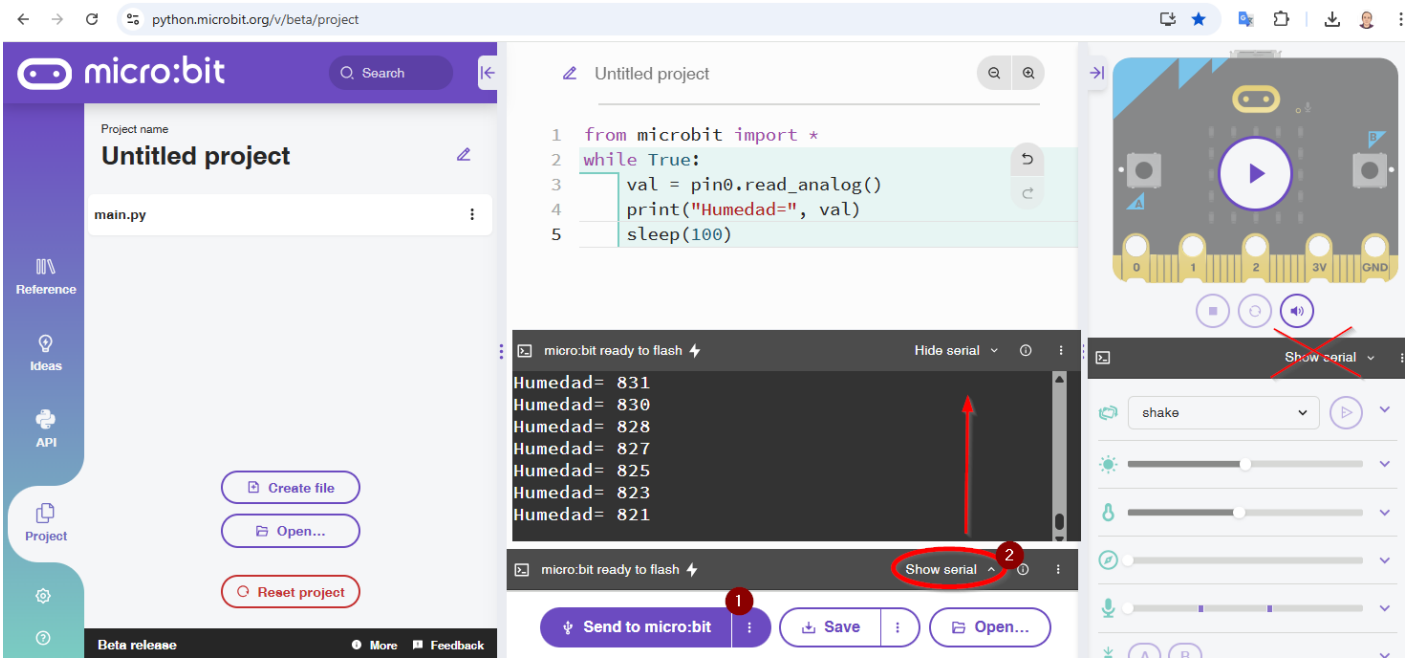


Maqueta Sensor lluvia

No necesitamos ninguna librería especial. Simplemente leer los valores analógicos del Pin 0. En este caso lo visualizamos por el puerto serie :

```
from microbit import *
while True:
    val = pin0.read_analog()
    print("Humedad=", val)
    sleep(100)
```

Para leer el puerto serie en https://python.microbit.org/ lo tienes aquí :



ALARMA LLUVIA

El proyecto pide una alarma. El siguiente código es extraído de https://docs.keyestudio.com/projects/KS4027-KS4028/en/latest/Python.html#project-11-rains-alarm

```
from microbit import *
import music
display.show(Image.HAPPY)
```

```
pin16.write_digital(0)

while True:
    if pin0.read_analog() > 500:
        music.play("C5:4")
        pin16.write_digital(1)
        sleep(100)
        music.reset()
        pin16.write_digital(0)
        sleep(100)
        music.play("C5:4")
        pin16.write_digital(1)
        sleep(100)
        music.reset()
        pin16.write_digital(0)
        sleep(100)
    else:
        music.reset()
        pin16.write_digital(0)
```

Una vez mojado el sensor, si se seca y queda por debajo de 500 se apaga la alarma:

https://www.youtube.com/embed/SuW51rT8lRI

Maqueta: Sensor Gas

Vamos a realizar un detector de gas

```
from microbit import *
import music

pin16.write_digital(0)

while True:

    if pin1.read_digital() == 0:
        music.play("C4:4")
        pin16.write_digital(1)
        sleep(100)
        music.reset()
        pin16.write_digital(0)
        sleep(100)
    else:
        music.reset()
        pin16.write_digital(0)
```

En este caso acercamos una botella de Alcohol

https://www.youtube.com/embed/jBXuNFGVf2Q

> RETO visualiza la lectura del sensor gas por el pueto serie:
> Solución https://docs.keyestudio.com/projects/KS4027-KS4028/en/latest/Python.html#project-12-analog-gas-mq-2-sensor

Maqueta DHT11

Se ha intentado el código de fgcoca y no ha resultado

Se ha intentado con la librería version_2 y con el siguiente código

```
# Imports go at the top
from microbit import *
from version_2 import *

SENSOR = DHT11(pin2)
while True:
    display.show(Image.HEART)
    t , h = SENSOR.read()
    print("Temperatura=",t)
    print("Humedad=",h)
    sleep(2000)
```

Y tampoco ha funcionado, el error lo sigue dando el chequeo de error. Si en la librería version_2 se anula, por el puerto serie no salen los valores correctos.

> Si consigues hacer funcionar el DHT11 de la maqueta con Python dínoslo, https://catedu.es/informacion/

Que curioso que el fabricante no ha puesto código Python en su tutorial https://docs.keyestudio.com/projects/KS4027-KS4028/en/latest/Python.html#expansion-projects