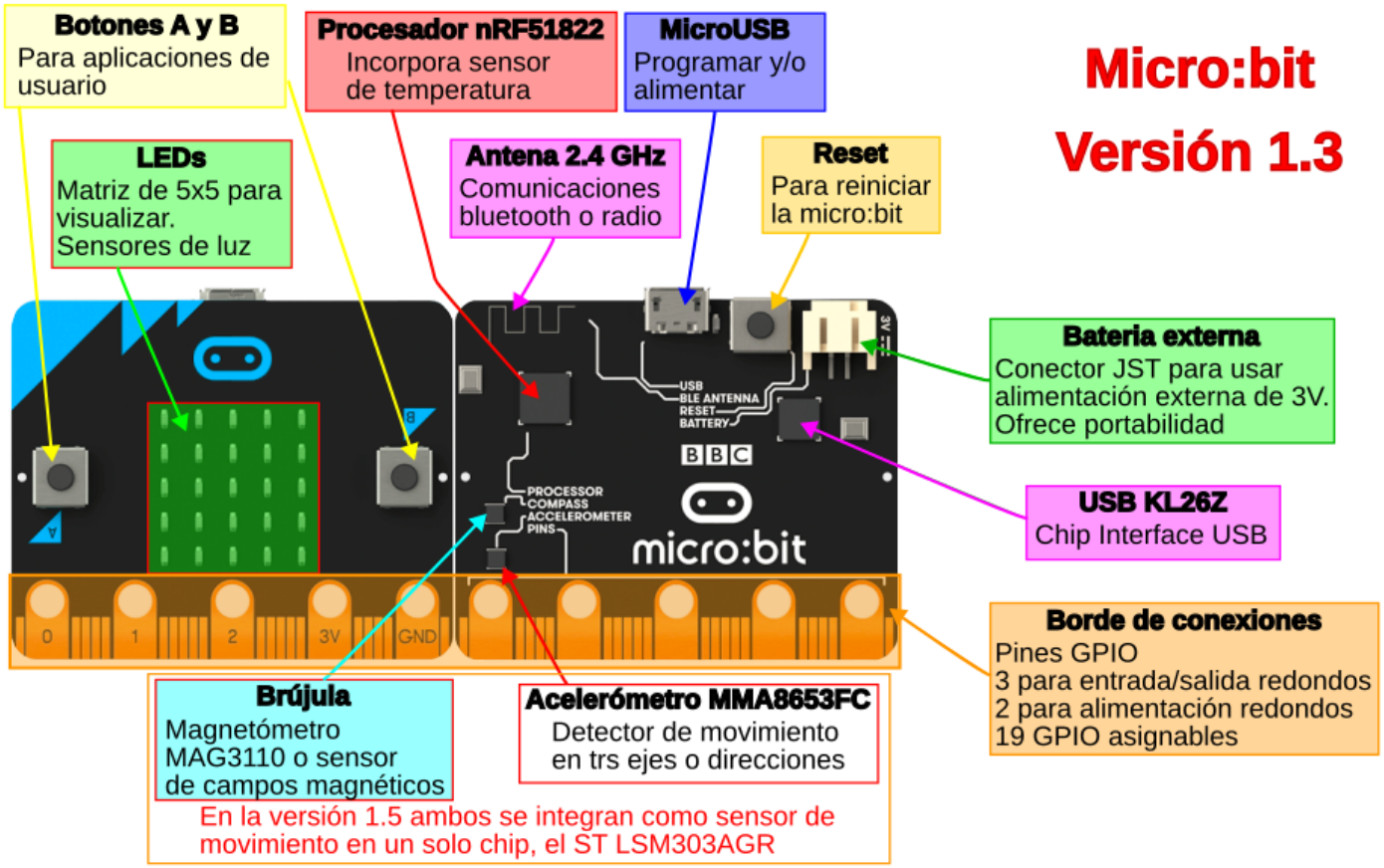
Introducción

- Microbit v2 vs v1
- Ventajas y desventajas Python
- Editores
- Introducción al Python
- Micropython de microbit
- Para saber más Python

Microbit v2 vs v1

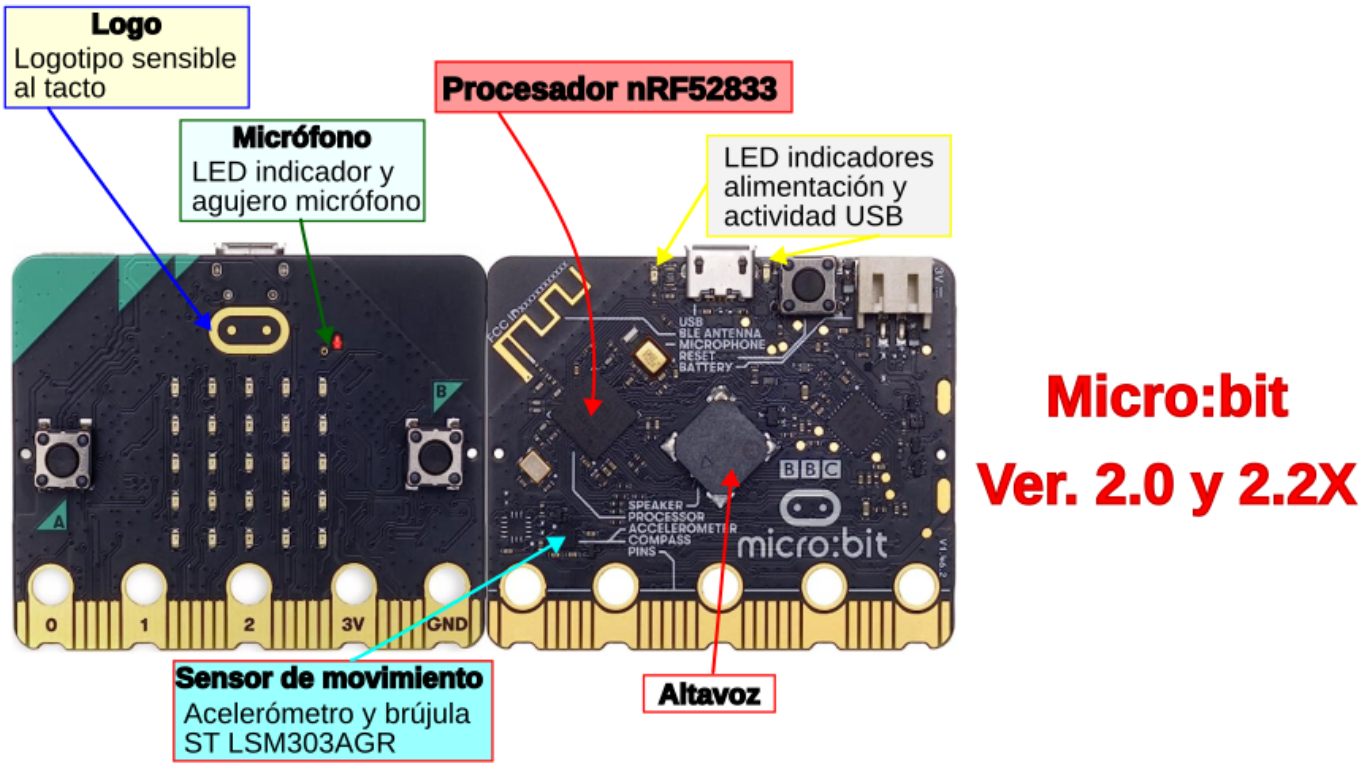
> Página extraída de Federico Coca Guia de Trabajo de Microbit CC-BY-SA

La microbit fué lanzada en el 2015 con la versión 1 que tiene los siguientes sensores y actuadores



Autor Federico Coca Fuente : Guía de Trabajo de Microbit Licencia CC-BY-SA

En el 2020 se lanzó la versión 2 que tiene además incorporó estas características



Autor Federico Coca Fuente : Guía de Trabajo de Microbit Licencia CC-BY-SA

Además de estas que no se ven:

- Modo ahorro de energía. Esta nueva función de ahorro o modo de espera detendrá el programa que se esté ejecutando en la micro:bit hasta que se pulse el botón de reinicio.
- Microprocesador cuatro veces más potente que la que tenía la v1
- Ocho veces más memoria que la v1 (512 KB de memoria Flash y 128 KB de memoria RAM)

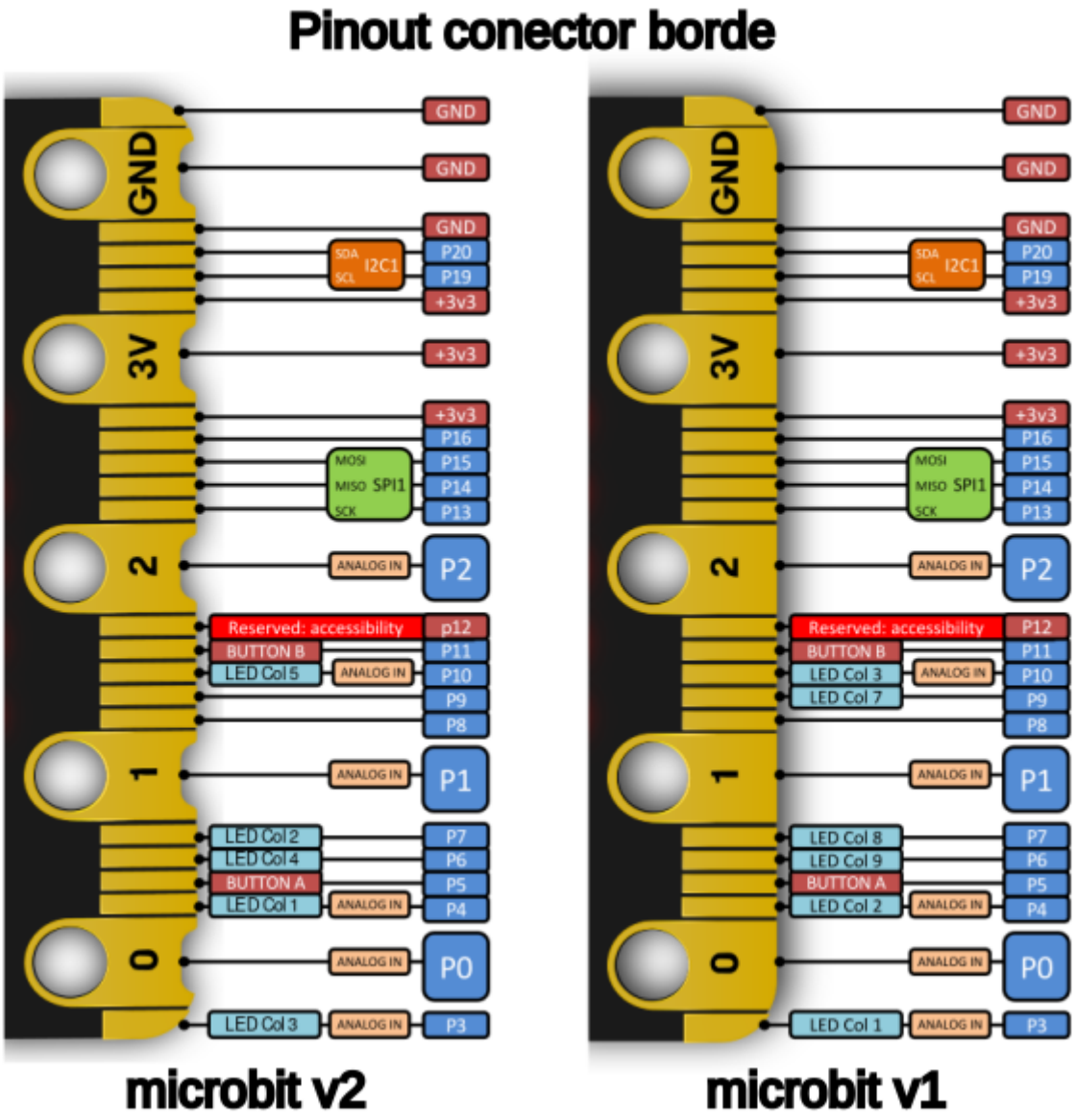
La alimentación es a través de los 5V del puerto USB o 3V a través del conector JST. También es posible alimentar a la micro:bit desde los anillos 3V/GND en el conector de borde.

Pinout

Hay 25 pistas/patillas que incluyen 5 agujeros para usar con clavijas tipo banana de 4 mm o pinzas de cocodrilo. Tres de estos anillos son para entrada y salida de propósito general (GPIO) y también sirven para detección analógica, PWM y táctil, y dos están conectados a la alimentación de la micro:bit.

Solamente tienen conexión las pistas frontales, las posteriores están sin conexión y los anillos posteriores están conectados a los delanteros. Las pistas mas finas están separadas 1,27 mm, algunas son utilizadas por micro:bit y otras están disponibles para su uso mediante cualquiera de los conectores externos existentes, lo que permite un amplio mercado de accesorios externos. En en enlace tenemos una guia de accesorios para micro:bit

En la imagen siguiente tenemos la descripción de pines de la micro:bit v2 a la izquierda y de la v1 a la derecha para poder comparar y establecer las diferencias de una forma sencilla.



Autor Federico Coca Fuente : Guía de Trabajo de Microbit Licencia CC-BY-SA

En microbit.pinout.xyz tenemos un fantástico recurso para obtener más información sobre los pines de la micro:bit y de cómo los utilizan algunos accesorios.

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

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

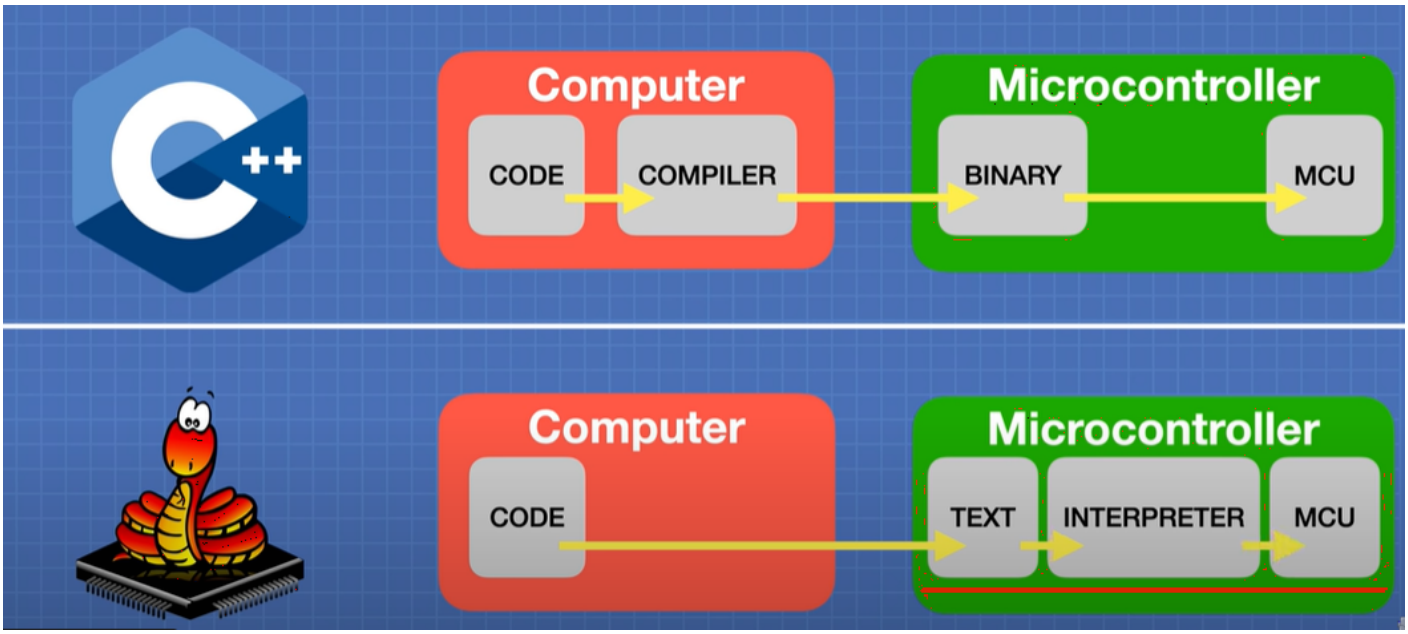
Es un lenguaje de alto nivel, es decir, que se programa igual que los programadores, pero interpretable para los humanos !. Comparándolo con otros lenguajes (Java, C++, etc..) es el más "*humanizado*".

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.

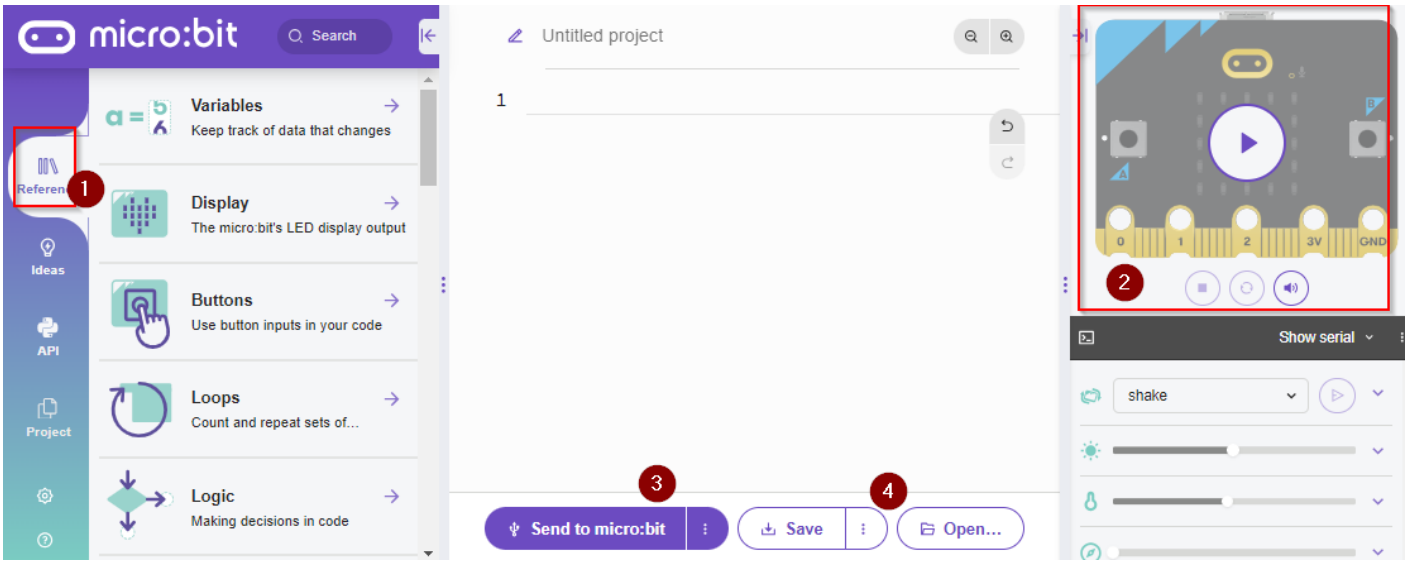
Editores

Tienes dos opciones, online o local :

Programar online con Microbit.org

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.



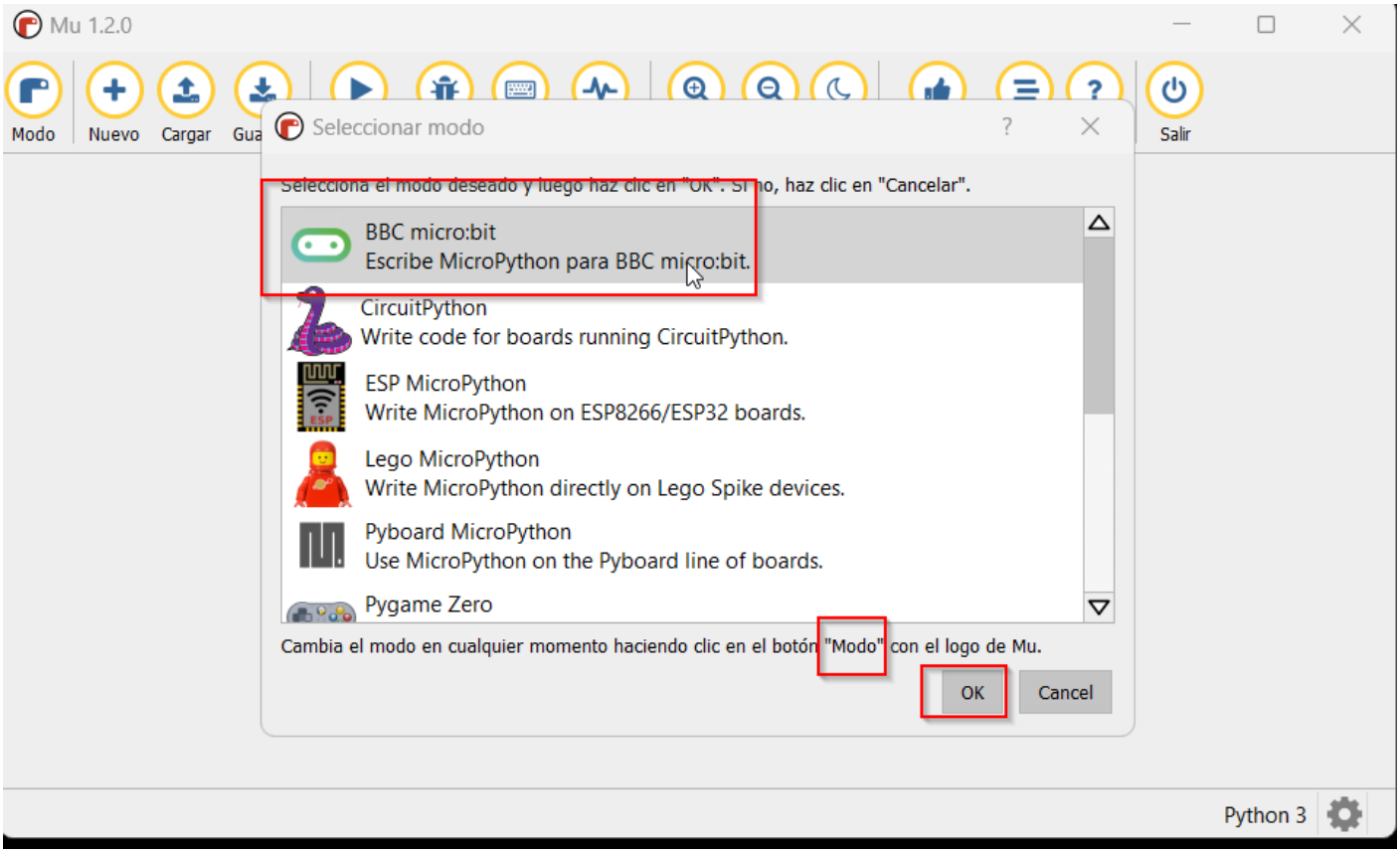
En este tutorial utilizaremos este editor online.

Programar en local con MU

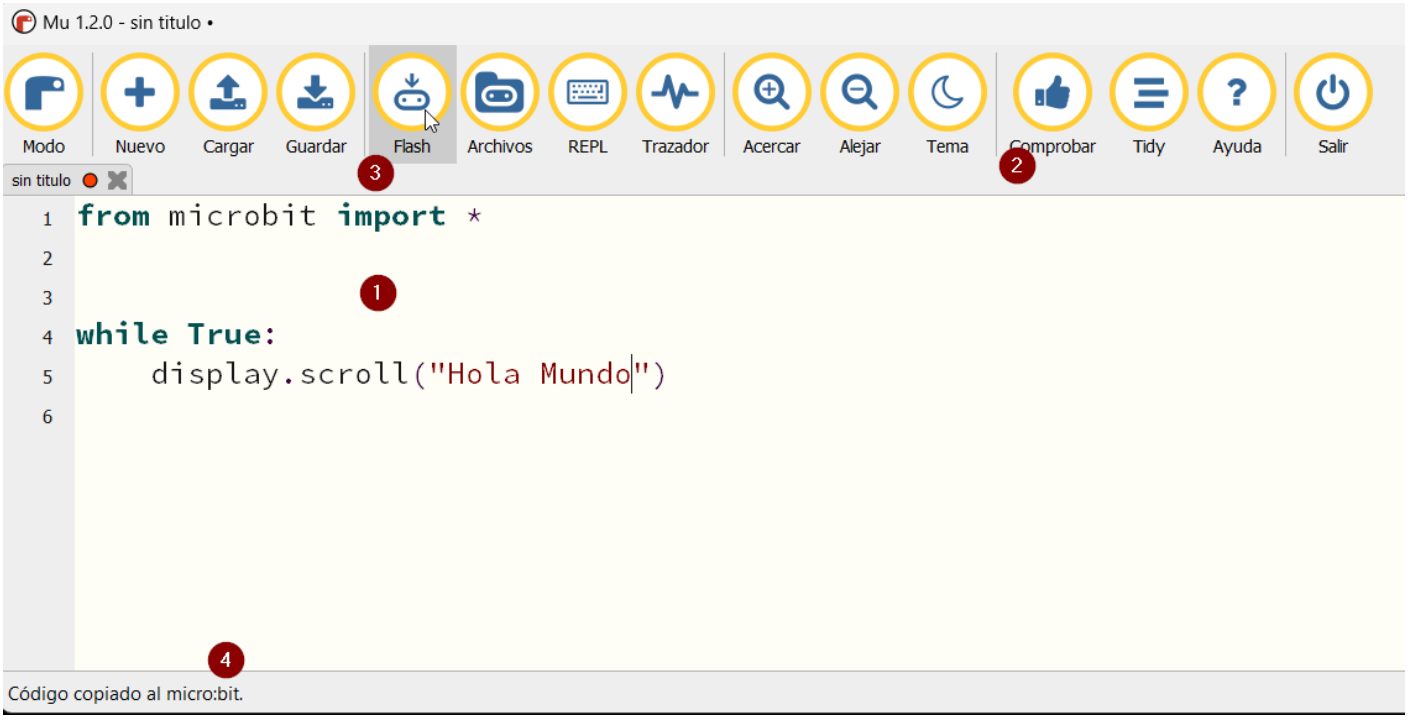
Es un editor muy sencillo, se descarga en https://codewith.mu/ y permite su instalación en Windows, Linux y Apple.
2024-07-04 18_44_27-(1) Exploring the Arduino Nano ESP32 _ MicroPython & IoT Cloud - YouTube.pı

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 (si quieres desconectar y volver a conectar) para que la placa ejecute el programa.



OTROS EDITORES DE PYTHON QUE NO SON COMPATIBLES CON PYTHON MICROBIT

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

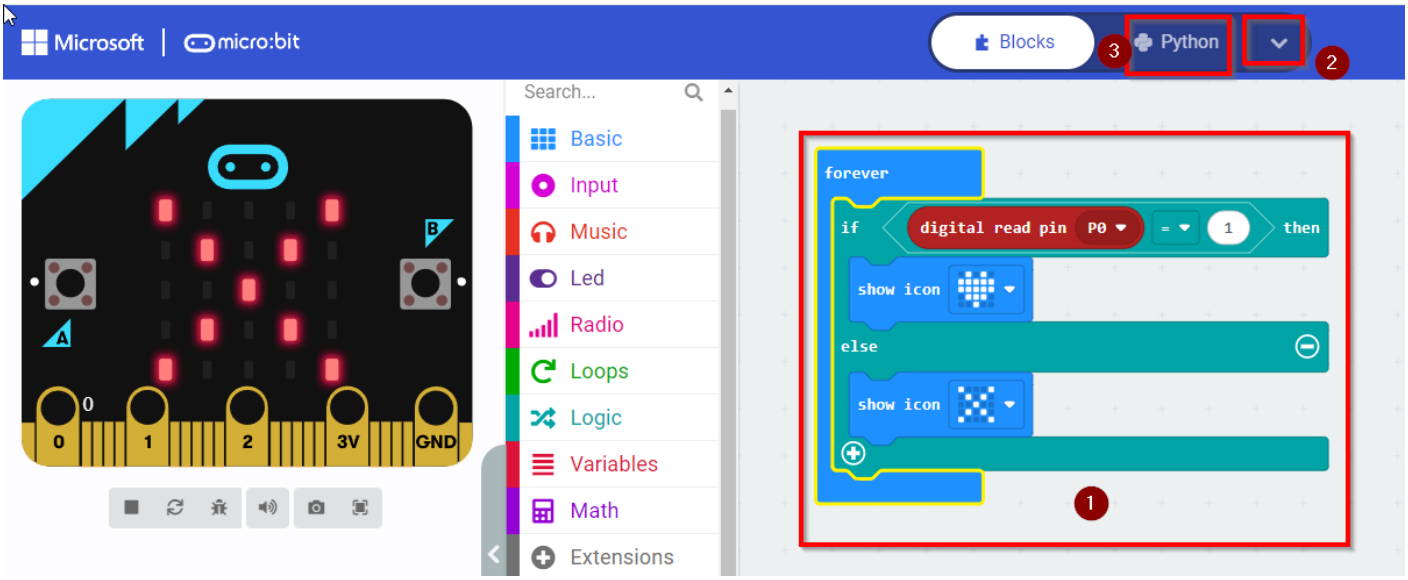
```
# Imports go at the top
from microbit import *
while True:
    if pin0.is_touched():
        display.show(Image.HEART)
    else:
        display.show(Image.NO)
```

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, tiene su sección de Python

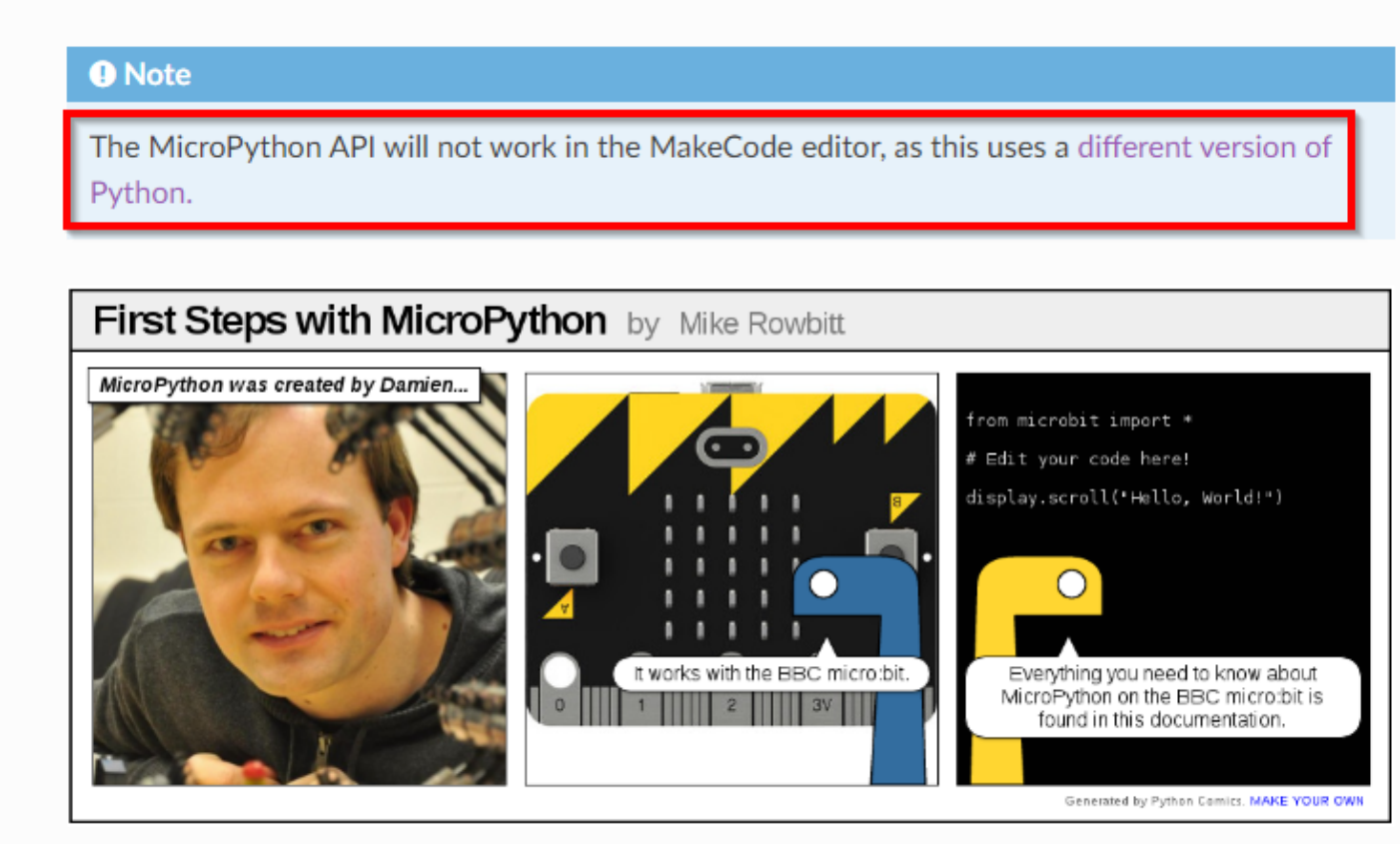


Al darle en Python (arriba a la derecha), muestra este código

```
def on_forever():
    if pins.digital_read_pin(DigitalPin.P0) == 1:
        basic.show_icon(IconNames.HEART)
    else:
```

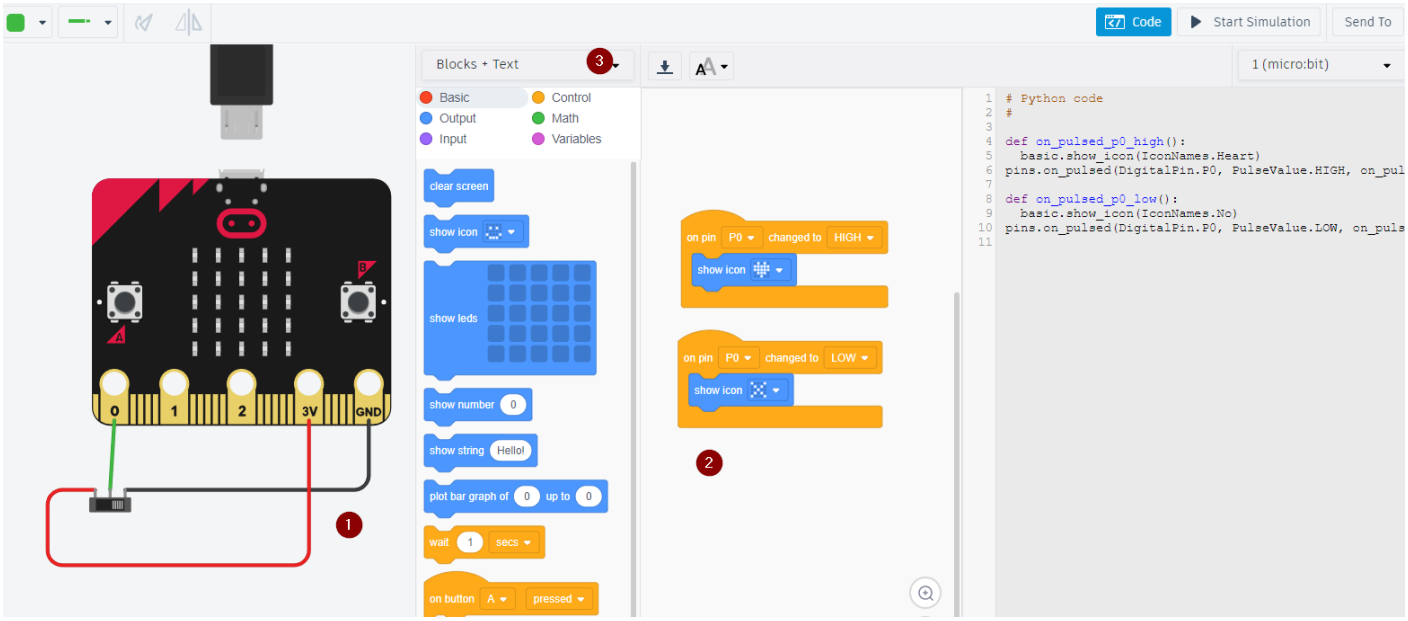
```
        basic.show_icon(IconNames.NO)
basic.forever(on_forever)
```

Como se puede ver **makecode python no es comp0atiuble con https://python.microbit.org/** ya lo dice en su tutorial https://microbit-micropython.readthedocs.io/en/v2-docs/



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

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

Palabras reservadas

Son palabras reservadas que tienen un significado especial para el compilador y que no podemos usar para poner nombres a variables o funciones. Todas las palabras, excepto True, False y None se escriben en minúsculas. A continuación se da un listado de todas las palabras reservadas o keywords:

> False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield

El listado al principio nos puede resultar abrumador, pero imaginemos un lenguaje con tan solo esas palabras y entenderemos que no resultará tan complejo familiarizarse, al menos con las mas usuales.

Identificadores

Los identificadores son los nombres que se dan a variables, clases, métodos, etc. No podemos usar palabras reservadas para estos nombres.

Algunas reglas que nos pueden resultar útiles para nombrar identificadores son:

- Los identificadores son sensibles a mayúsculas y minúsculas
- Los identificadores no pueden ser palabras reservadas
- Los espacios en blanco no están permitidos
- Un identificador puede ser una secuencia de letras y números. Siempre debe empezar por una letra o por el símbolo de subrayado "_".
- El primer carácter de un identificador no puede ser un número.
- No podemos utilizar caracteres especiales como la ñ, ¡, ¿ o letras con acentos.
- No podemos utilizar los símbolos como !, @, #, $, etc.

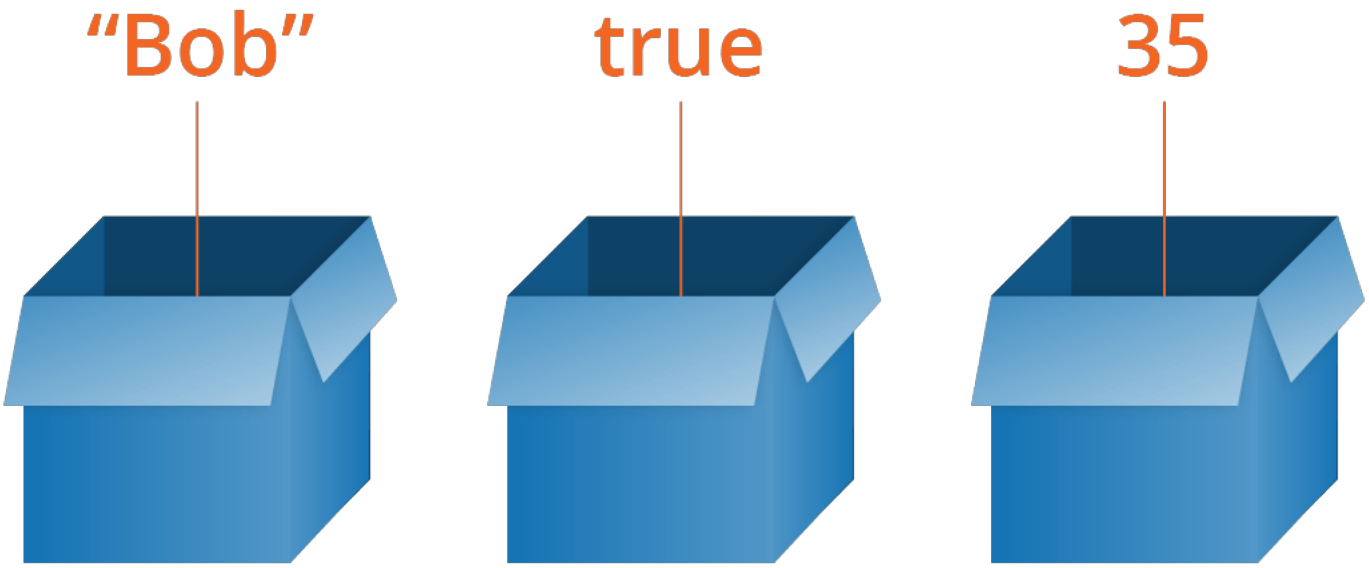
Nos va a resultar muy útil recordar lo siguiente:

- Python es un lenguaje que distingue entre mayúsculas y minúsculas. Esto significa que **Variable** y **variable** no son lo mismo
- Damos siempre a los identificadores un nombre que tenga sentido. Aunque que **c = 10** es un perfectamente válido, escribir **contador = 10** tendría más sentido, y sería más fácil averiguar lo que representa cuando miremo el código pasado un tiempo.
- Las palabras múltiples se pueden separar usando un guión bajo, como por ejemplo **esto_es_un_nombre_de_variable_muy_largo.**

Variables, constantes y literales

Variables

En programación, una variable es un nombre que se utiliza para referirse a una posición de memoria donde se almacena un valor. De forma más abstracta, puede considerarse como una caja que almacena un valor. El nombre de la caja es el nombre de la variable y el contenido su valor. Todas las variables constan de tres partes: un nombre, un tipo de dato y un valor. En la figura siguiente vemos tres variables de distintos tipos:



Licencia CC-BY-SA fuente

La variable `name` contiene la cadena `Bob`, la variable `winner` es cierta y la variable `score` contiene el valor `35`.

Python no dispone de ningún comando para declarar variables. Una variable se crea cuando se le asigna valor por primera vez. La técnica de declarar variables es poner un nombre seguido del signo de asignación (=) y el valor asignado a la variable. En la declaración es importante tener claro que se distinguen mayúsculas de minúsculas y que no están permitidos los caracteres especiales.

En Python no se declara de forma explicita el tipo de la variable pues se trata de un lenguaje inferido. Las variables incluso pueden cambiar de tipo desde el que se establece al asignarle valor

la primera vez. Es decir, si declaro `valor = 5` inicialmente la variable será de tipo entero (int), pero si en el programa se realizan operaciones que al final hacen que `valor = 1.33` ahora valor es de tipo float. Automáticamente sabe que `valor` es un número entero y declara la variable `valor` como un `int`.

Aunque no es necesario si es posible especificar el tipo de dato de una variable, haciendo:

```
x = str(22) # x será la cadena '22'
y = int(22) # y será el entero 22
z = float(22) # z será el número de coma flotante 22.0
```

Algunas reglas para nombrar variables que podemos tener en cuenta son:

> - Los nombres pueden tener una combinación de letras minúsculas o mayúsculas o números o el símbolo de subrayado "_".
> - Crear nombres que tengan sentido, aunque sean largos.
> - Si usamos varias palabras para definir el nombre, estas las separamos por "_"
> - Python es sensible a mayúsculas y minúsculas.
> - Hay que evitar palabras reservadas en nombres de variables.

Constantes

Una constante no es mas que un tipo especial de variable cuyo valor no puede modificarse.

En Python, las constantes suelen declararse y asignarse en un **módulo** (un nuevo archivo que contiene variables, funciones, etc y que se importa al archivo principal).

Veamos cómo declaramos constantes en un archivo separado y lo usamos en el archivo principal,

Creamos un archivo que nombramos constantes.py y que contendrá:

```
PI = 3.141592
FUERZA_GRAVEDAD = 9.82
```

Creamos el archivo principal main.py, que contendrá:

```
import constantes


print(constantes.PI)
print(constantes.FUERZA_GRAVEDAD)
```

En el ejemplo creamos el archivo de módulo constantes.py y asignamos el valor constante a PI y FUERZA_GRAVEDAD.

Después, creamos el archivo main.py e importamos el módulo constantes. Finalmente, imprimimos el valor de cada constante.

La convención es nombrarlas en mayúsculas para distinguirlas de las variables.

Literales

Numéricos

Los literales son representaciones de valores fijos en un programa. Pueden ser números, caracteres, cadenas, etc. Por ejemplo, "¡Hola, mundo!", 12, 23.0, "C", etc.

Los literales numéricos son inmutables (no pueden cambiar) y pueden pertenecer a uno de los tres tipos de datos numéricos posibles: Entero, Coma flotante y Complejo. Los tipos son:

- **Decimal**. Números regulares. Por ejemplo: 5, 22, -40
- **Binario**. Deben comenzar por 0b. Por ejemplo: 0b110, 0b11
- **Octal**. Deben empezar con 0o. Por ejemplo: 0o13, 0o7
- **Hexadedicmal**. Deben empezar con 0x. Por ejemplo 0x13, 0xFF
- **Coma flotante**. Contienen el punto decimal. Por ejemplo 10.2, 3.14
- **Complejo**. Tienen la forma `a + bj`. Por ejemplo: 3 - 2j, -4 + j
- **Booleanos**

Solamente hay dos literales booleanos `True` y `False`

Cadenas de caracteres

Los literales de caracteres son caracteres unicode encerrados entre comillas, por ejemplo `S`. Los literales cadenas de caracteres son cadenas de csaracteres encerradas entre comillas, por ejemplo `Python es divertido`.

Especiales

En Python existe un literal especial, `None`. Podemos usarlo, por ejemplo, para especificar una variable nula, por ejemplo:

```
var = None
print(var)
# El resultado será: None
```

Tipos de datos en Python

En Python, al igual que en programación en general, los tipos de datos especifican el tipo de datos que puede almacenarse en una variable.

Numéricos

Contienen valores numéricos y sabemos que:

> - Los números enteros son de tipo int
> - Los fraccionarios son de tipo float
> - La división (/) siempre devuelve un número en coma flotante
> - Para obtener la parte entera de una división se usa el operador //
> - Para calcular el resto de una división se usa el operador %
> - Para calcular potencias podemos usar el operador **
> - Los paréntesis se pueden usar para agrupar expresiones
> - El signo igual (=) se utiliza para asignar un valor (números, booleanos, cadenas, ...) a una variable
> - El tipo de la variable será el del dato asignado, no se declara el tipo de la variable al crearla
> - Por convención el nombre comienza en minúscula y si son varias palabras se unen por guión bajo

Los tipos básicos de datos son:

> - **`int`**: números enteros con signo sin límite de tamaño, ejemplo: entero = 5
> - **`float`**: números reales, decimales o de coma flotante con precisión de hasta 15 decimales, ejemplo: real = 5.6
> - **`complex`**: números complejos, por ejemplo 5.5 - 5j
> - Para averiguar el tipo de dato usamos la función `type()`.

Podemos realizar conversión de tipos así:

- A entero int(variable)
- A real float(variable)

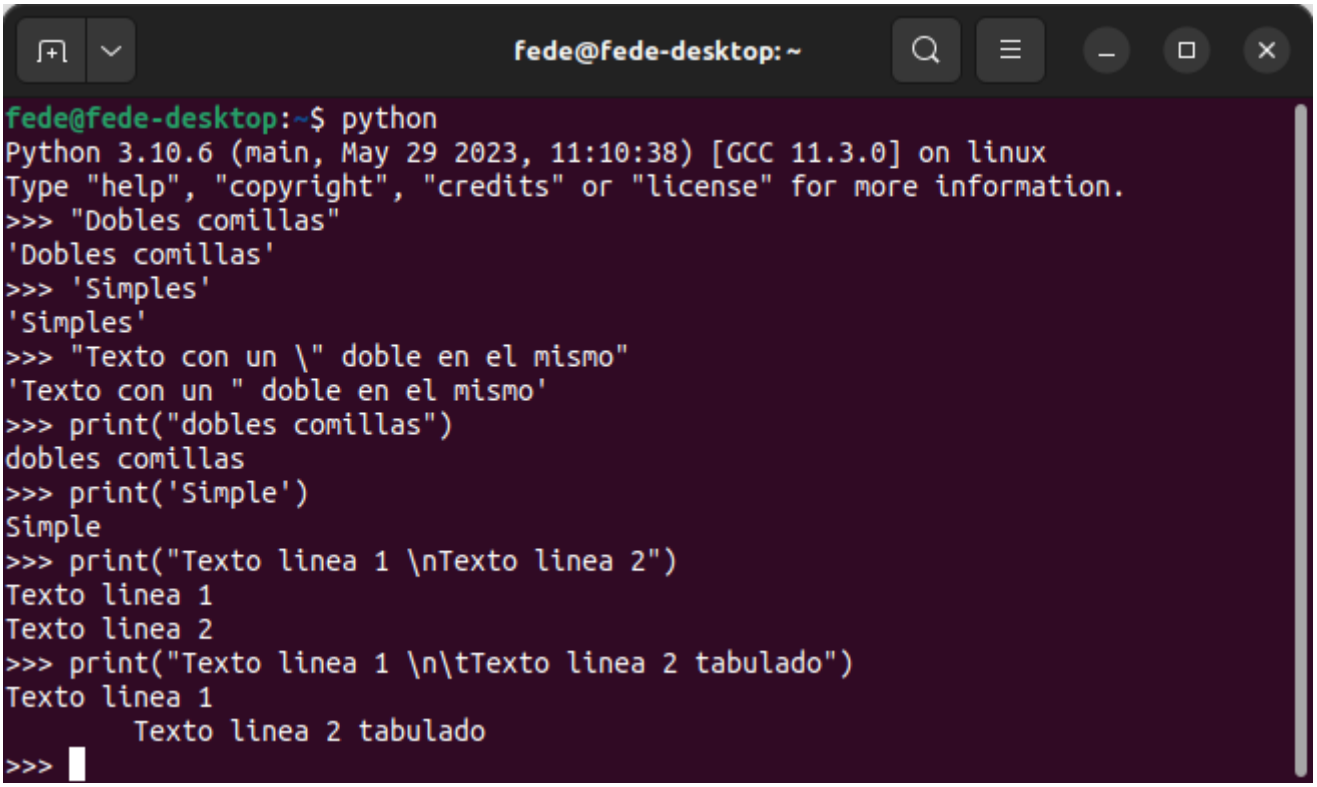
Cadenas

Contienen secuencias de caracteres. Una cadena es una secuencia de caracteres representada entre comillas simples o dobles.

Las cadenas pueden estar encerradas entre comillas simples ('...') o dobles ("...") con el mismo resultado. Podemos usar para incluir comillas en una cadena.

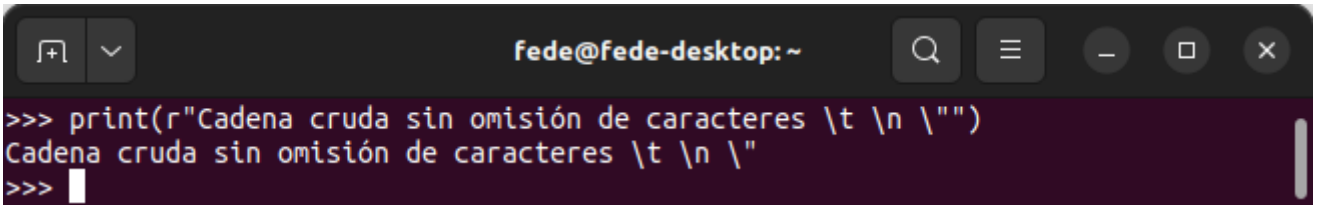
La función print() devuelve la cadena que encierra entre los paréntesis, omitiendo las comillas que la encierran.

En la imagen siguiente se ven varios ejemplos con cadenas utilizando como editor el IDLE que por defecto se instala con Python y que se abre desde una terminal simplemente invocando a Python.



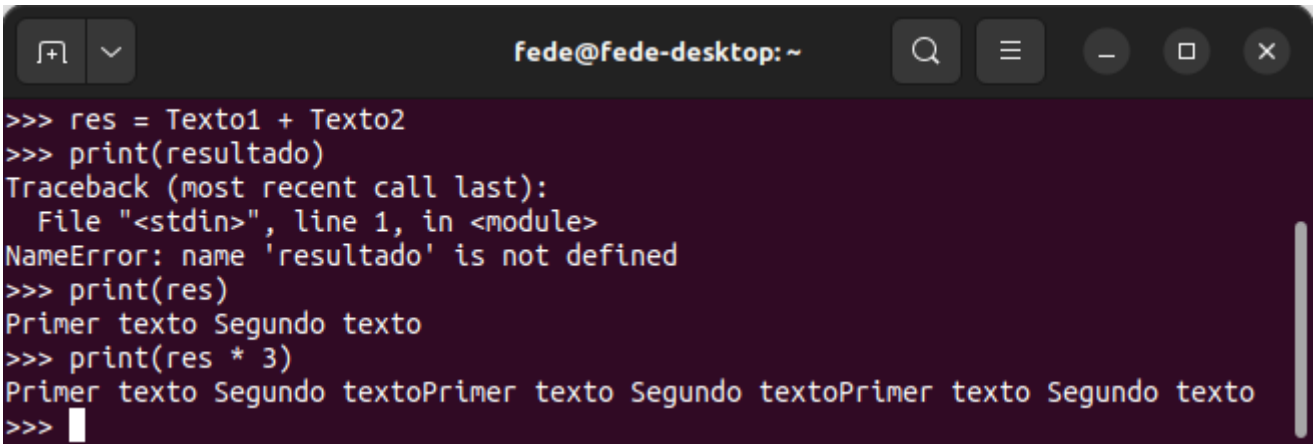
Federico Coca Guia de Trabajo de Microbit CC-BY-SA

Una cadena raw (cruda) se interpreta tal como se escribe, es decir, se omiten los caracteres especiales expresados con . Las cadenas raw se escriben entrecomilladas y van precedidas del carácter 'r'. En la imagen vemos un ejemplo.



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

Es posible aplicar la operación de multiplicar a textos haciendo que estos se repitan. En la imagen siguientes vemos ejemplos de concatenación y multiplicación, así como un error cometido.



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

Secuencias

Contienen colecciones de datos, como las listas, las tuplas, las colecciones de datos (set) o los diccionarios.

Una lista es una colección ordenada de elementos similares o de distinto tipo separados por comas y encerrados entre corchetes [ ].

Tupla es una secuencia ordenada de elementos, igual que una lista. La única diferencia es que las tuplas son inmutables. Una vez creadas, las tuplas no pueden modificarse. En Python, se utilizan los paréntesis () para almacenar los elementos de una tupla.

Las colecciones de datos son un conjunto desordenada de elementos únicos. Una colección de datos se define por valores separados por comas dentro de llaves { }.

Un diccionario es una colección ordenada de elementos. Almacena los elementos como pares clave/valor. Siendo las claves identificadores únicos que se asocian a cada valor.

Estudiaremos estos últimos tipos mas extensamente cuando los necesitemos.

Los datos de tipo booleano solamente pueden contener `True` o `False`.

Dado que en programación Python todo es un objeto, los tipos de datos son en realidad clases y las variables son instancias(objeto) de estas clases.

Comentarios en Python

- Una sola linea : Escribiendo el símbolo almohadilla (#) delante del comentario.
- Multilinea: Escribiendo triple comillas dobles (“””) al principio y al final del comentario.

En los comentarios, pueden incluirse palabras que nos ayuden a identificar además, el subtipo de comentario:

```
# TODO esto es algo por hacer
# FIXME (arreglarme) esto es algo que debe corregirse
# XXX esto también, es algo que debe corregirse
```

Identation o sangría 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.

```
if 5 > 2:
    print("Cinco es mayor que 2")
```

Lo siguiente sería un error de sintaxis.

```
if 5 > 2:
print("Cinco es mayor que 2")
```

El número de espacios de la identation puede ser cualquiera siempre que al menos sea un espacio. **Siempre** hay que usar el mismo número de espacios en el mismo bloque de código.

Operadores en Python

Los operadores son símbolos especiales que realizan operaciones con variables y valores.

A continuación tenemos una lista de los diferentes tipos de operadores de Python:

- Operadores aritméticos
- Operadores de asignación
- Operadores de Comparación
- Operadores Lógicos
- Operadores Bitwise
- Operadores especiales

Operadores aritméticos

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`

Operadores de asignación¶

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

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

Operadores de Comparación

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`

Operadores Lógicos

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`

En la figura siguiente vemos un ejemplo con lo que devuelve en cada caso.

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

Operadores Bitwise¶

Los operadores bit a bit o bitwise actúan sobre los operandos como si fueran cadenas de dígitos binarios. Operan bit a bit, de ahí su nombre.

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`

Operador	Descripción	Ejemplo
^	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`

Operadores especiales

El lenguaje Python ofrece algunos tipos especiales de operadores como el operador de identidad (`identity`) y el operador de pertenencia (`membership`).

- **Operadores `identity`**

En Python, `is` e `is not` se utilizan para comprobar si dos valores se encuentran en la misma parte de la memoria. Dos variables que son iguales no implica que sean idénticas. Algunos ejemplos aclaran mejor lo dicho.

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'

print(x1 is not y1)  # False

print(x2 is y2)  # True
```

Vemos que x1 e y1 son enteros con los mismos valores, por lo que son iguales e idénticos. Lo mismo ocurre con x2 e y2 (cadenas).

- **Operadores `membership`**

En Python, `in` y `not in` son los operadores de pertenencia. Se utilizan para comprobar si un valor o variable se encuentra en una secuencia (cadena, lista, tupla, conjunto y diccionario).

En un diccionario sólo podemos comprobar la presencia de la clave, no del valor.

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]
```

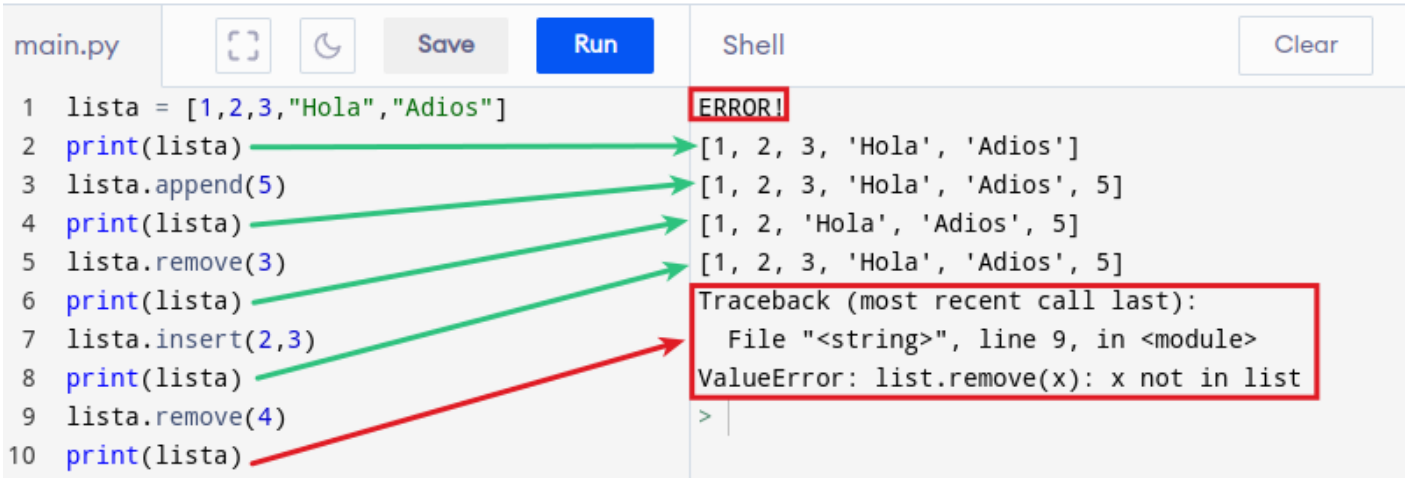
Las de principales propiedades de las listas:

- 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.

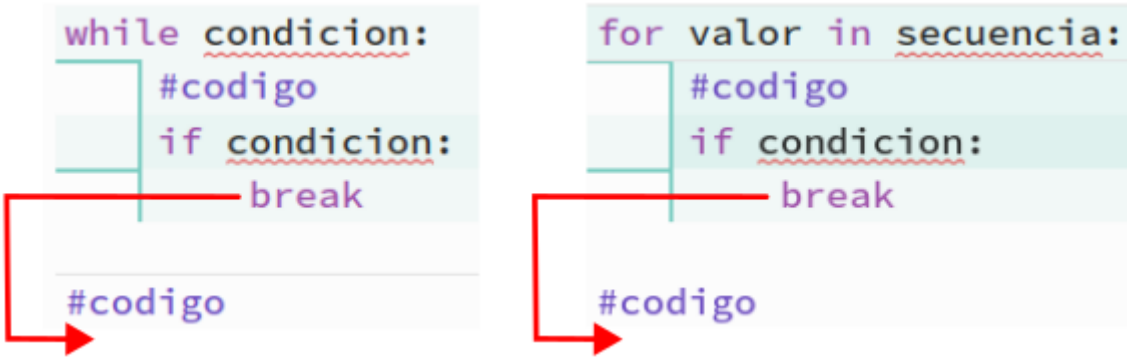
```
for i in range(20, 0, -2): #imprimere 20, 18, 16, ... 0
```

- 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
```

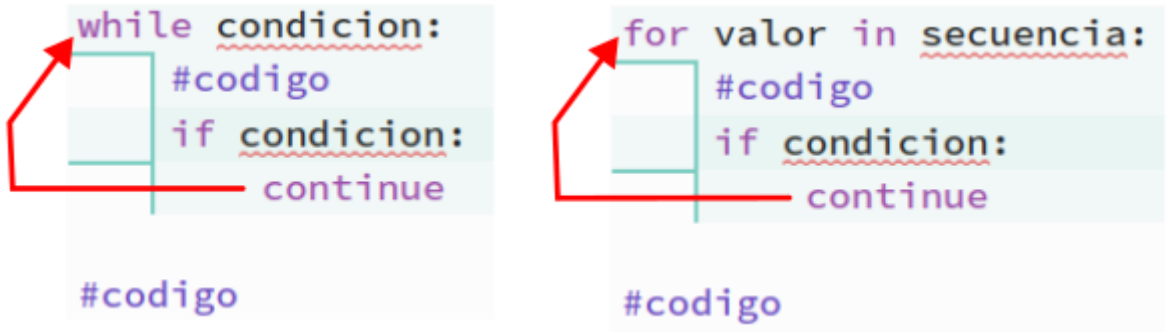
Sentencias `break` y `continue`

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.



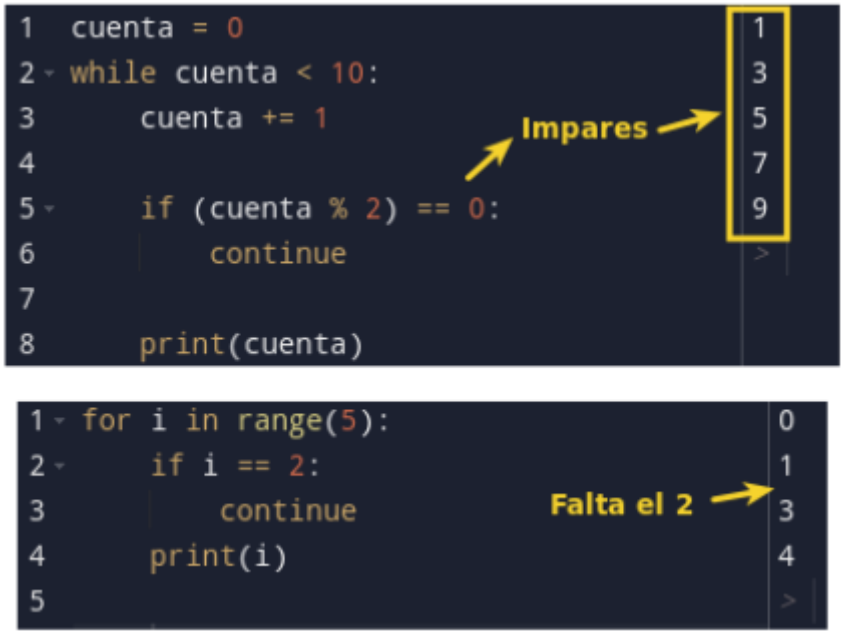
Federico Coca Guia de Trabajo de Microbit CC-BY-SA

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`

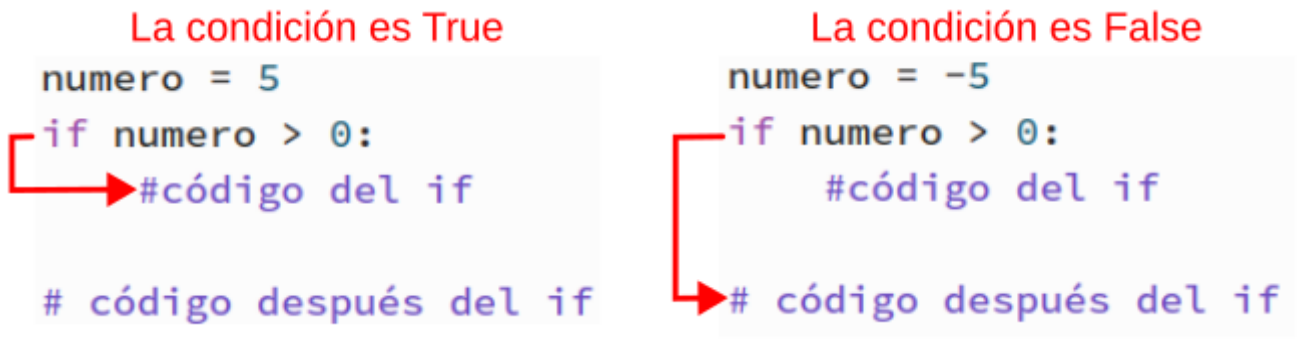
1. Declaración `if`. La sintaxix de esta declaración en Python tiene la forma siguiente:

```
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

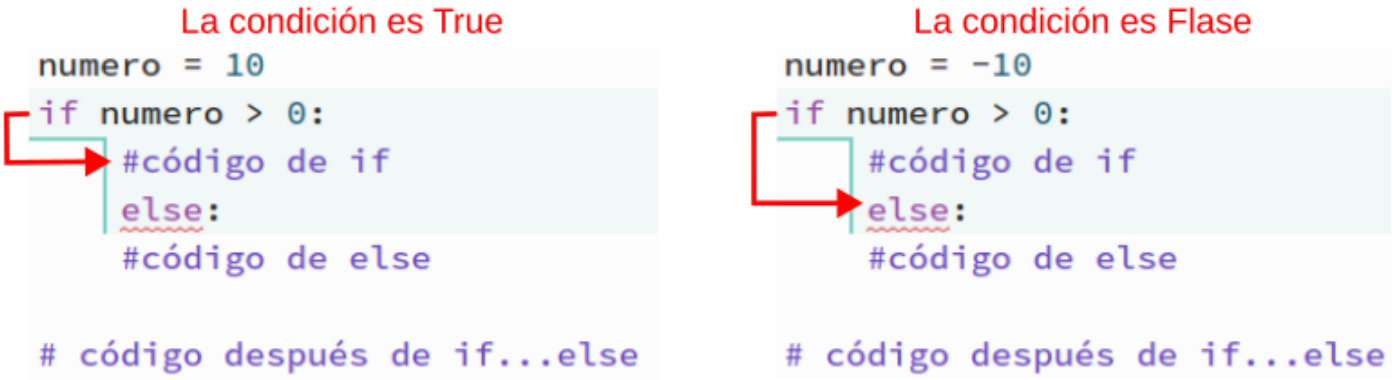
1. Declaración `if...else`. Una sentencia `if` puede tener de manera opcional una clausula `else`. La sintaxix de esta declaración en Python tiene la forma siguiente:

```
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*

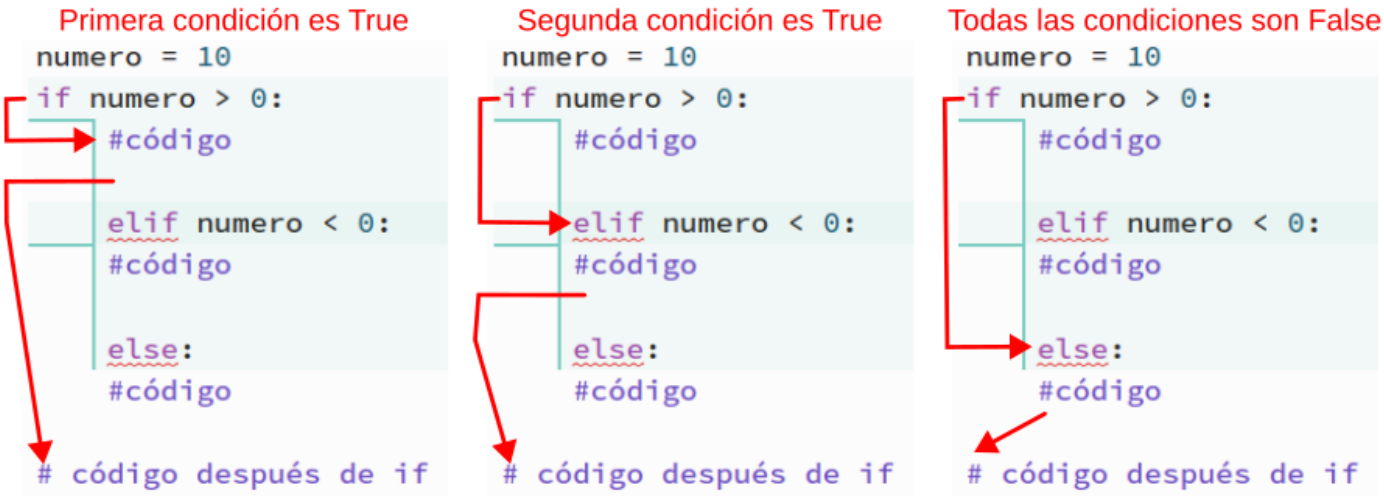
1. Declaración `if...elif...else`. La sentencia `if...else` se utiliza para ejecutar un bloque de código entre dos alternativas posibles. Sin embargo, si necesitamos elegir entre más de dos alternativas, entonces utilizamos la sentencia `if...elif...else`. La sintaxis de la sentencia `if...elif...else`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

```
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

```python
import random
r1 = random.getrandbits(15)
r2 = random.randint(10, 40)
r3 = random.randrange(3)
r4 = random.randrange(3, 5)
r5 = random.randrange(3, 30, 5)
frutas = ['pera', 'manzana', 'plátano',
    'ciruelas', 'sandia', 'melon']
r6 = random.choice(frutas)
r7 = random.random()
r8 = random.uniform(5, 15)
r9 = random.uniform(30, 20)
print('.getrandbits(n):', r1)
print('.randint(a, b):',r2)
print('.randrange(stop):', r3)
print('.randrange(start, stop):' ,r4)
print('.randrange(start, stop, step):'
    ,r5)
print('.choice(secuencia):' ,r6)
print('.random():' ,r7)
print('.uniform(a, b), (a<=b):' ,r8)
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
>
```

Para saber más Python

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