# 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]
```

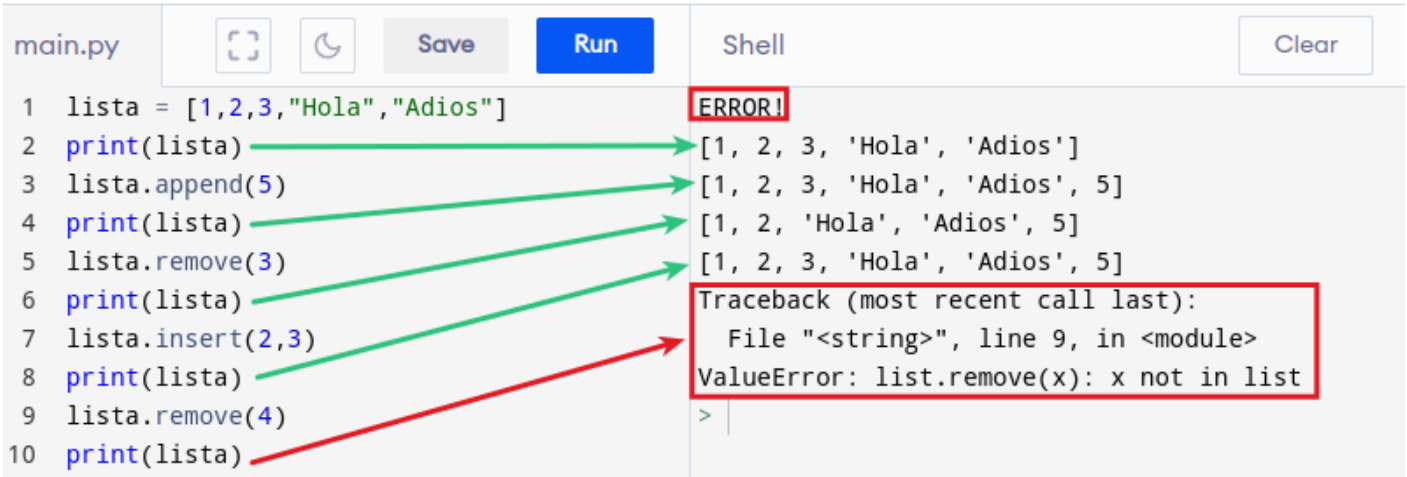
Las de principales propiedades de las listas:

- 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.

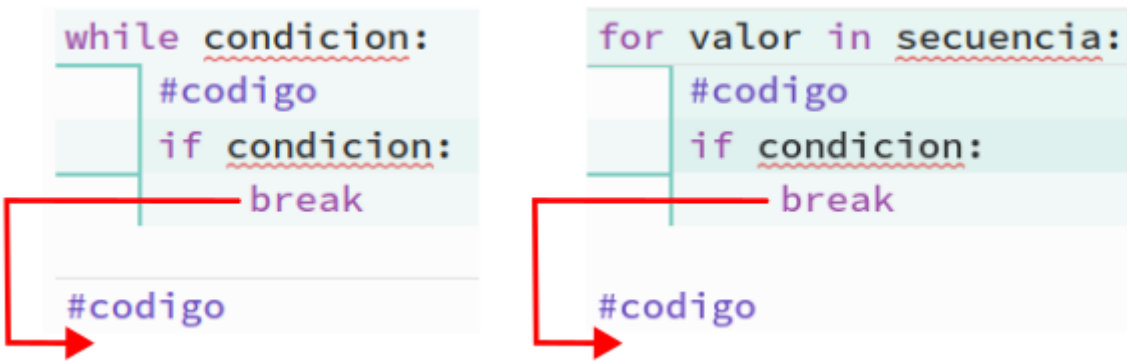
```
for i in range(20, 0, -2): #imprimere 20, 18, 16, ... 0
```

- 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
```

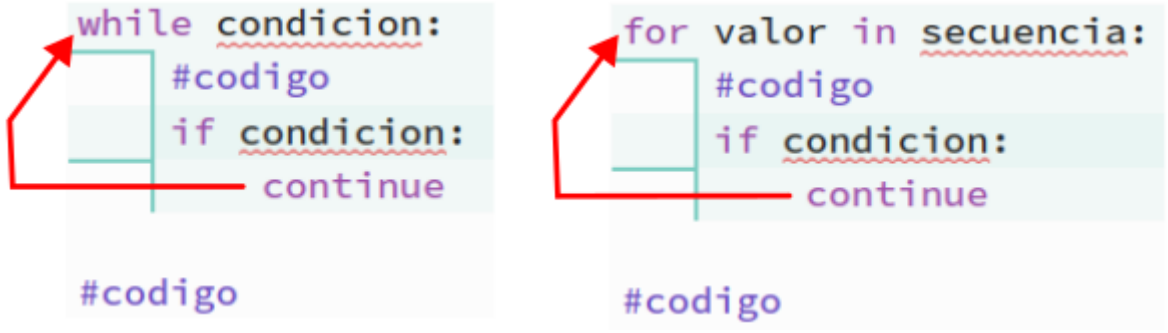
## Sentencias `break` y `continue`

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`

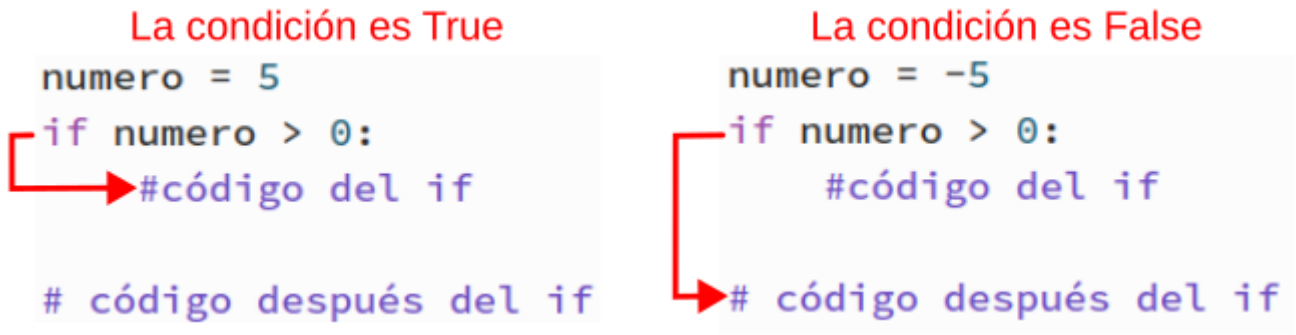
1. Declaración `if`. La sintaxix de esta declaración en Python tiene la forma siguiente:

```
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*

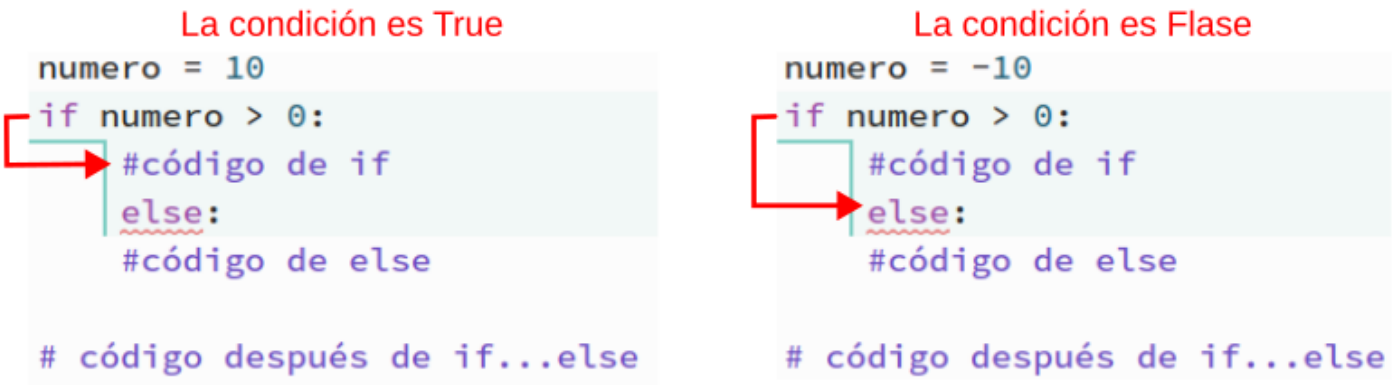
1. Declaración `if...else`. Una sentencia `if` puede tener de manera opcional una clausula `else`. La sintaxix de esta declaración en Python tiene la forma siguiente:

```
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*

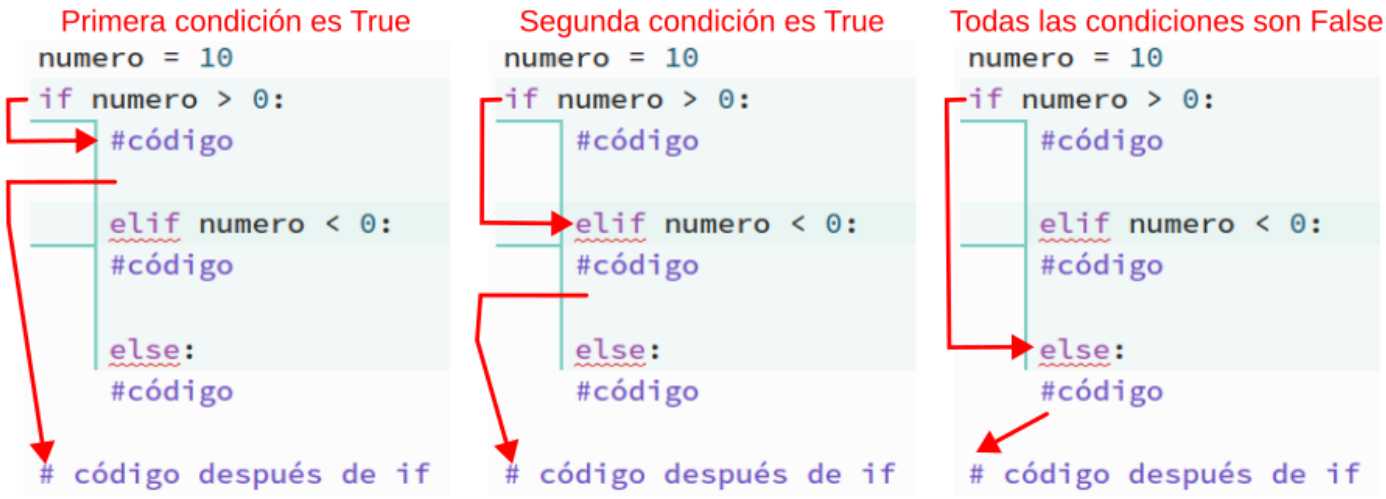
1. Declaración `if...elif...else`. La sentencia `if...else` se utiliza para ejecutar un bloque de código entre dos alternativas posibles. Sin embargo, si necesitamos elegir entre más de dos alternativas, entonces utilizamos la sentencia `if...elif...else`. La sintaxis de la sentencia `if...elif...else`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






## 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

```
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
>
```



Revision #1
Created 2025-11-05 19:39:47 CET by Javier Quintana
Updated 2025-11-05 19:40:00 CET by Javier Quintana