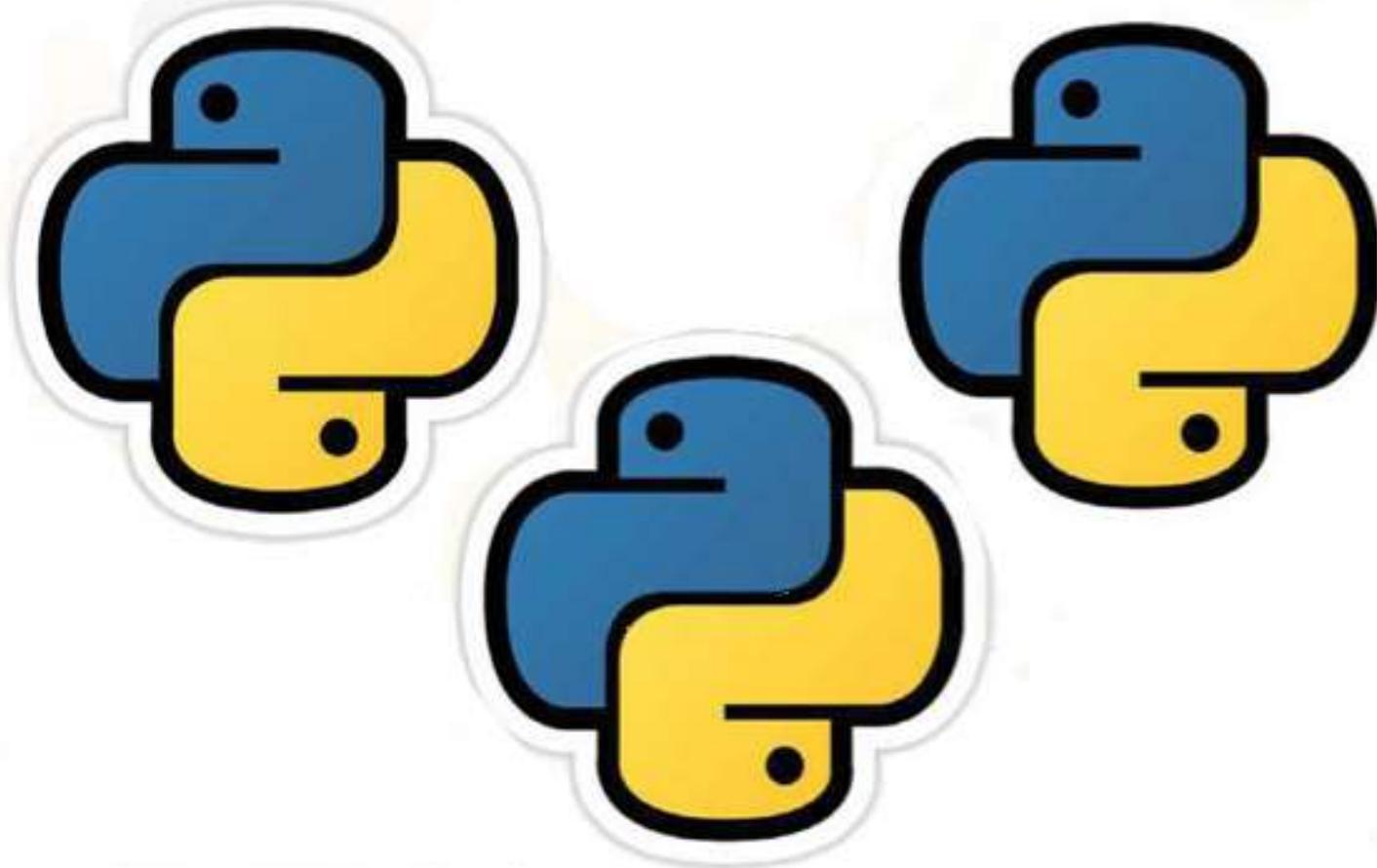


USERS

VOL. III

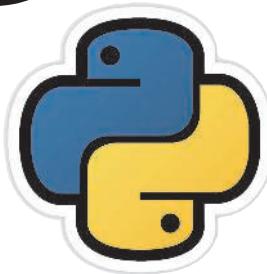
Programación en

python



**PROYECTOS PRÁCTICOS - RASPBERRY PI
MICROPYTHON**

Programación en python



PROYECTOS PRÁCTICOS - RASPBERRY PI MICROPYTHON

USERS

Título: Programación en Python - Vol. III / **Autor:** Edgardo Stasi

Coordinador editorial: Miguel Lederkremer / **Edición:** Claudio Peña

Diseño y Maquetado: Marina Mozzetti / **Colección:** USERS ebooks - LPCU298

Copyright © MMXX. Es una publicación de Six Ediciones. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio actual o futuro, sin el permiso previo y por escrito de Six Ediciones. Su infracción está penada por las leyes 11723 y 25446. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen y/o analizan. Todas las marcas mencionadas en este libro son propiedad exclusiva de sus respectivos dueños. Libro de edición argentina.

Stasi, Edgardo

Programación en Python 3 : proyectos prácticos - Raspberry Pi - Micropython / Edgardo Stasi. - 1a ed . - Ciudad Autónoma de Buenos Aires : Six Ediciones, 2020.

Libro digital, PDF - (Programacion en Python ; 3)

Archivo Digital: online
ISBN 978-987-4958-26-6

1. Lenguajes de Programación. I. Título.
CDD 005.4



RU RedUSERS PREMIUM

- ✓ Nuestros expertos comparten su saber y experiencia
- ✓ Calidad y cantidad por una mínima cuota mensual
- ✓ Cientos de publicaciones con los temas que más interesan
- ✓ Siempre, donde vayas. On Line – Off Line. En cualquier dispositivo
- ✓ Al menos 1 novedad semanal. Lee todo lo que deseas sin límites

SUSCRÍBETE

 usershop.redusers.com

 +54-11-4110-8700

 usershop@redusers.com

PRÓLOGO

Python es un lenguaje de programación que supo ganarse el cariño de la comunidad de programadores tanto por su simplicidad en la escritura como en su potencialidad a la hora de crear un proyecto.

No es extraño, entonces, encontrarnos que cada vez más sistemas sean implementados bajo este lenguaje y que su extensa comunidad aporte continuamente librerías simples pero eficientes.

En este e-book, veremos distintas maneras que existen para llevar todo su potencial a proyectos prácticos, simples pero ilustrativos que, extrapolados, nos pueden dar una idea de cómo encarar proyectar más grande.

Para ello, veremos cómo ponerlos en práctica y obtener resultados en el mundo físico utilizando placas de usos múltiples, como Raspberry pi o pyboard.

También conoceremos MicroPython, una versión acotada del lenguaje, aunque no menos potente, creada exclusivamente para funcionar en componentes donde los recursos son limitados por tratarse de placas de un tamaño reducido, ideales para la implementación de sistemas dentro del mundo del IoT.



Acerca del autor

Edgardo Stasi es un Ingeniero en informática argentino, recibido en el año 2004. Ha desarrollado distintos sistemas de gestión empresarial adquiriendo experiencia con los lenguajes VB6, Progress 4GL, VB.Net. En los últimos años comenzó a desarrollar aplicaciones para entorno web, lo que lo llevó a capacitarse en HTML, CSS, PHP y JS.

Sus primeros encuentros con Python fueron en el laboratorio de programación de la facultad, donde pudo visualizar su potencial y su simplicidad de codificación. Actualmente, es su lenguaje preferido como *hobby*, con el que desarrolla e implementa proyectos de domótica y robótica de manera aficionada.

Sobre este curso

Python es un lenguaje de programación multiplataforma, consistente y maduro, utilizado por numerosas empresas internacionales. Se utiliza en múltiples campos tales como aplicaciones web, juegos y multimedia, interfaces gráficas, networking, aplicaciones científicas, inteligencia artificial y muchos otros.

En esta serie de ebooks sobre programación en Python el lector encontrará todo lo necesario para iniciarse o profundizar sus conocimientos en este lenguaje de programación.

El curso se compone de tres volúmenes, orientados tanto a quien recién se inicia en este lenguaje como a quien ya está involucrado y quiere profundizar sus conocimientos de Python.

Volumen I

Se realiza una revisión de las características de este lenguaje, también se entregan las indicaciones para instalar el entorno de desarrollo y se analizan los elementos básicos de la sintaxis y el uso de las estructuras de control, con una serie de códigos de ejemplo explicados en detalle.

Volumen II

Se presenta el paradigma de programación orientada a objetos con todas sus implicancias: clases, herencia y todo el campo de posibilidades que nos abre comenzar a utilizar este paradigma en Python.

Volumen III

Orientado a la aplicación de Python en proyectos, veremos ejemplos de aplicación en Raspberry Pi y Micropython entre otros.

¡Éxitos en este nuevo desafío!

Sumario

VOL. III

01 - QUÉ ES RASPBERRY PI / 6

- EL SOFTWARE / 6
- EL HARDWARE / 7
- PREPARAMOS PYTHON / 10
- ACCESO A RASPBERRY PI POR SHH / 11
- ¿QUÉ ES SHH?

02 - GPIO / 14

- DISEÑO DEL CIRCUITO / 15
- CONEXIONES PROTOBOARD PLACA /
- CONEXIONES DE ELEMENTOS
- ALTERNATIVAS SI NO
- TENEMOS LA PLACA / 16
- NUESTRO CÓDIGO / 18
- LIBERIA TIME / LIBRERIA GPIO /
- CONTINUEMOS CON EL CÓDIGO

03 - HACKEO A MINECRAFT PI / 22

- INSTALACIÓN DE MINECRAFT / 22
- PRIMEROS PASOS / 23
- HOLA MINECRAFT / 25
- CONECTARNOS CON MINECRAFT / 25
- COMANDOS PYTHON PARA MINECRAFT PI /
- ARMEMOS NUESTRA CASA
- OTROS COMANDOS INTERESANTES / 38
- MINECRAFT.PLAYER / MINECRAFT.CAMERA
- APLICACIONES / 39

04 - MICROPYTHON / 40

- VENTAJAS / 40
- DESVENTAJAS / 40

PLACAS COMPATIBLES

- CON MYCROPYTHON / 41
- PYBOARD / WIPY / ESP8266 / BBC.MICRO.BIT

PLACAS PYBOARD / 44

- PRIMEROS PASOS / MÓDULO PYB / CLASES / MÓDULO MACHINE / SIMULADOR ON LINE DE PYBOARD
- ENCENDAMOS EL LED / 54

05 - PROYECTOS

EN MYCROPYTHON / 56

- HOLA MUNDO / 56
- LA CLASE I2C / LA CLASE FRAMEBUF
- HAGAMOS NUESTRA APLICACIÓN / 60
- CONTROL DE SERVO / 63
- PYTHON Y IOT / 69

Apéndice - DJANGO:

PYTHON EN LA WEB / 70

- QUE ES DJANGO / 70
- SUS ORÍGENES
- INSTALACIÓN EN WINDOWS
- O LINUX / 71
- EN LINUX / EN WINDOWS
- ENTORNOS VIRTUALES / 72
- USOS DE ENTORNOS VIRTUALES
- CREAR UN PROYECTO / 74
- ESTRUCTURA
- AGREGAR UNA APLICACIÓN
- A UN PROYECTO / 78
- LÓGICA DEL FUNCIONAMIENTO /
- ARMADO DE UN TEMPLATE / HOLA MUNDO
- RESUMEN / 87

Qué es Raspberry pi

La tecnología avanza en una dirección indiscutible apuntando a la hiperconectividad, y en este proceso el **IoT** (internet de las cosas) toma relevancia y comienza a vivir su auge. Para acompañar este movimiento, surgen placas que simplifican el diseño y la implementación de proyectos que permiten conectar distintos dispositivos para obtener, procesar y transmitir información en tiempo real.

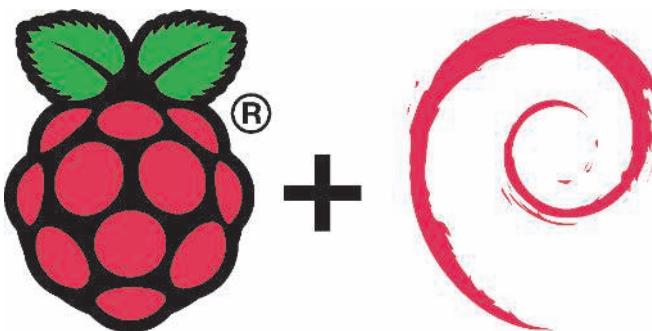
Raspberry pi da un salto a esto e integra en una placa de un tamaño poco mayor que una tarjeta de crédito una computadora funcional con todos sus componentes. Solo basta conectarle una memoria con una imagen del sistema operativo, un teclado, un mouse, un monitor y parlantes, para poder disfrutar de sus capacidades.

EL SOFTWARE

Si bien podemos instalar distintas versiones de sistemas operativos, al menos, para seguir este curso, recomendamos instalar **Raspbian**, una distribución GNU/Linux basado en Debian y optimizada para el hardware disponible.

Una vez que tenemos este sistema operativo funcionando, vamos a verificar que, dentro de los programas preinstalados, se encuentra Python, por lo que solo nos resta

comenzar a escribir código para generar nuestros programas, procesos o automatizaciones.



Raspbian

*Raspbian: distribución
Debian para Raspberry pi.*

EL HARDWARE

Como ya hemos visto, la placa Raspberry pi es una computadora integrada, a la que solo le faltan los periféricos.

Según la versión que tengamos, será el hardware disponible, sin embargo, para los alcances de este e-book, es indistinto con cuál estemos trabajando.



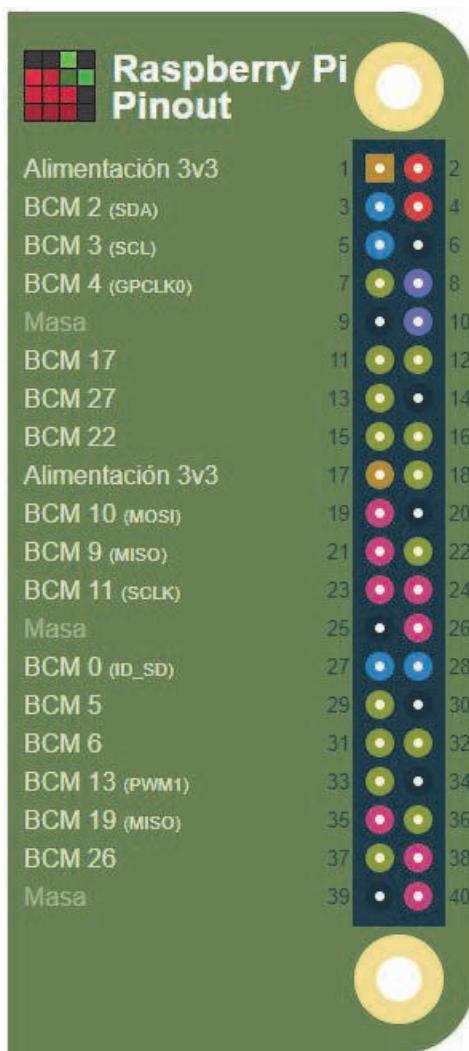
Placa Raspberry pi 2015.

Si observamos detalladamente la imagen anterior, nos vamos a encontrar con una serie de pines generales de entrada/salida, conocidos como **GPIO** (*General Purpose Input/Output*), estos nos van a permitir interactuar con el mundo exterior más allá de los periféricos estándares.



Pines
GPIO.

Desde Python, veremos cómo conectarnos y obtener o transmitir datos a través de ellos. Por tal razón, es importante que sepamos cómo identificarlos y cuál es su función individualmente.



Resultado de pinout.

En las últimas versiones de **Raspbian OS**, podemos ejecutar **pinout** por consola para obtener la configuración de pines sobre la placa que estamos trabajando.

```
pi@raspberrypi:~ $ pinout
 00000000000000000000 J8
 10000000000000000000
Pi Model 3B V1.2
[D] [SoC] [C] [Net]
[S] | | [S] |
[I] +---+ [I] | A | V
[pwr] [HDMI] | I | [V]
Revision : a02082
SoC      : BCM2837
RAM      : 1024Mb
Storage   : MicroSD
USB ports : 4 (excluding power)
Ethernet ports : 1
Wi-fi     : True
Bluetooth : True
Camera ports (CSI) : 1
Display ports (DSI): 1

J8:
 3V3  (1) (2) 5V
 GPIO2  (3) (4) 5V
 GPIO3  (5) (6) GND
 GPIO4  (7) (8) GPIO14
 GND   (9) (10) GPIO15
 GPIO17 (11) (12) GPIO18
 GPIO27 (13) (14) GND
 GPIO22 (15) (16) GPIO23
 3V3  (17) (18) GPIO24
 GPIO10 (19) (20) GND
 GPIO9   (21) (22) GPIO25
 GPIO11  (23) (24) GPIO8
 GND   (25) (26) GPIO7
 GPIO0  (27) (28) GPIO1
 GPIO5  (29) (30) GND
 GPIO6  (31) (32) GPIO12
 GPIO13 (33) (34) GND
 GPIO19 (35) (36) GPIO16
 GPIO26 (37) (38) GPIO20
 GND   (39) (40) GPIO21
```

Para trabajar desde Python, bastará con asignar a una variable un pin determinado. A partir de ese momento, podremos leer o escribir el estado de dicha variable, que va a otorgarnos el estado electrónico del pin en cuestión.

En Raspberry pi podremos hacerlo de dos formas distintas:

- mediante el número físico en la placa, o
- mediante el número de canal asignado en el procesador (BCM).

Para tener en cuenta

Los pines GPIO trabajan a 3.3V.
Es importante no olvidarnos de esto para evitar sobrecargarlo y quemar nuestra placa Raspberry pi.

PREPARAMOS PYTHON

Para comenzar a programar con Python, debemos contar con la librería GPIO. Si nuestra placa tiene instalado Raspian OS, ya viene correctamente instalada. Sin embargo, si por algún motivo no se encuentra, o estamos utilizando alguna otra distribución como sistema operativo, deberemos ejecutar desde consola los siguientes comandos:

```
sudo apt-get install python-dev  
sudo apt-get install python-rpi.gpio
```

Con el primero, instalamos Python, si no está. Mientras que, con el segundo, instalamos la librería GPIO.

Cuando, por algún inconveniente, el paso anterior no funciona, podemos instalar la librería en forma manual.

Lo primero que haremos es descargar el paquete desde:
<https://pypi.org/project/RPi.GPIO/>.

Luego, lo descomprimimos (en caso de cambiar el nombre del archivo, colocamos el correcto):

```
tarzvxf RPi.GPIO-0.7.0.tar.gz
```

Entramos al directorio que acabamos de crear en la descarga:

```
cd RPi.GPIO-0.7.0/
```

Instalamos la librería con:

```
sudo python setup.py install
```

Con todo esto, ya estamos en condiciones de comenzar a programar. Dicha programación podemos hacerla de dos maneras distintas.

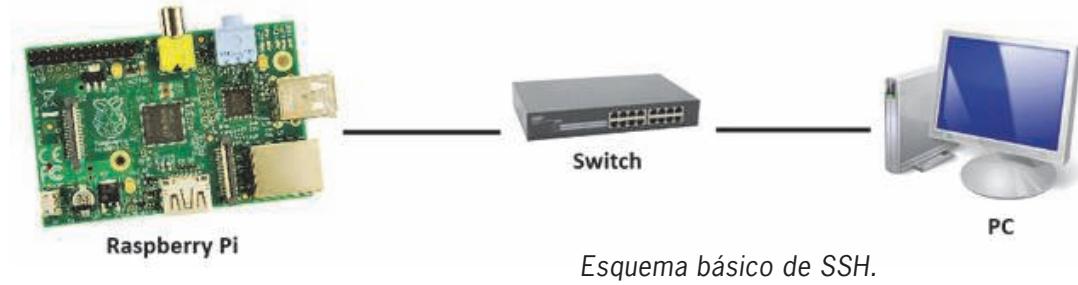
- Desde la misma placa, instalándole los periféricos adecuados para poder manipular y visualizar como si de una computadora se tratase.
- De manera remota, accediendo mediante **SSH** (*Secure Shell*).

ACCESO REMOTO A RASPBERRY PI POR SSH

Si no contamos con los periféricos adecuados o preferimos acceder a nuestra placa de manera remota, podemos utilizar este protocolo y trabajar desde una PC como si estuviéramos haciéndolo desde la misma Raspberry pi.

¿Qué es SSH?

Es un protocolo de acceso remoto seguro, que utiliza la arquitectura cliente/servidor. A diferencia de otros protocolos, SSH encripta la información



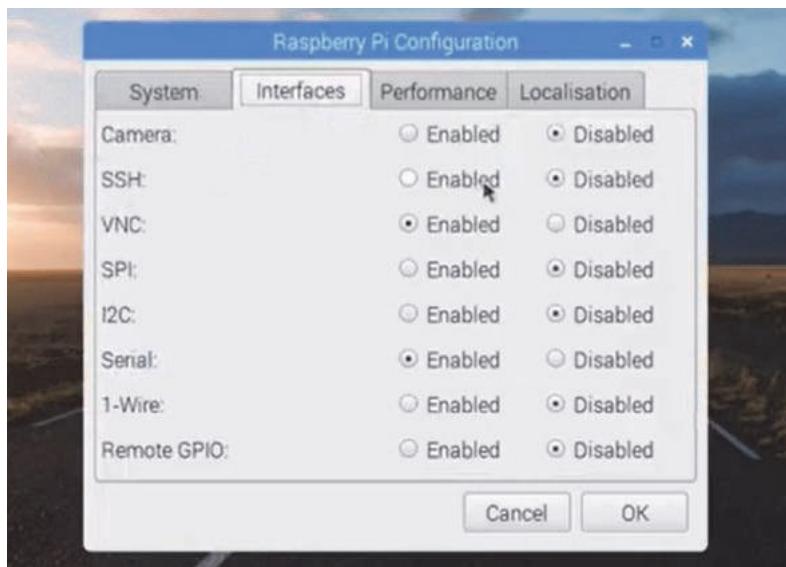
Esquema básico de SSH.



Para poder acceder, es condición que estemos conectados en la misma red y que conozcamos la ip de la Raspberry pi.

Lo primero queharemos es habilitar desde la Raspberry el SSH. Para ello, deberemos conectarle un monitor y un teclado, y acceder a la parte de configuraciones.

Acceso a configuración por SSH.

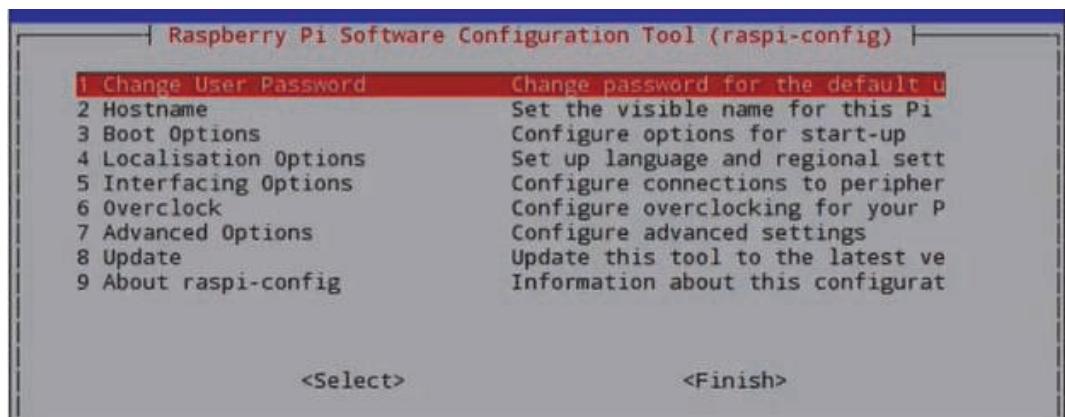


Activación SSH –
Modo gráfico

Luego, en la ventana que se nos abre, presionamos en la solapa de **interfaces**, buscamos la opción SSH y la tildamos en **enable**.

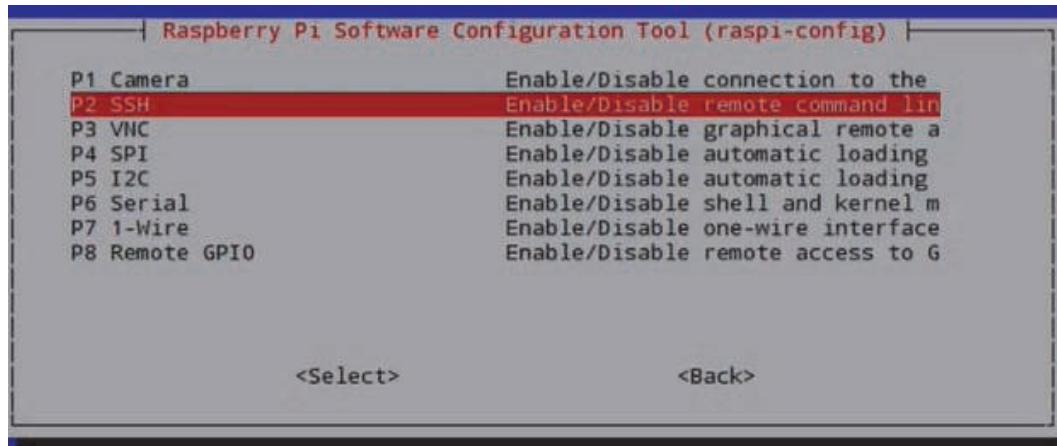
Hecho esto, podremos acceder desde la máquina cliente. Para eso accedemos a la consola de comandos y escribimos **ssh** seguida del número ip y el nombre del usuario **-lusr-name** (por defecto pi). Al presionar **ENTER**, nos pedirá una contraseña, que por defecto es **raspberry**. Una vez cumplidos estos pasos, ya estamos dentro de nuestra placa y podremos enviar los comandos como si estuviéramos de manera local.

En el caso de que tengamos instalada una versión de Raspbian Lite, que no incluye la interfaz gráfica y solo tenemos acceso por consola de comandos, deberemos escribir **sudo raspi-config** y así se nos abrirá un menú.



Menú acceso a configuración por comandos.

Una vez en este menú, accedemos a **Interfacing Options** y seleccionaremos, en la nueva pantalla, la opción **SSH** para habilitarla.



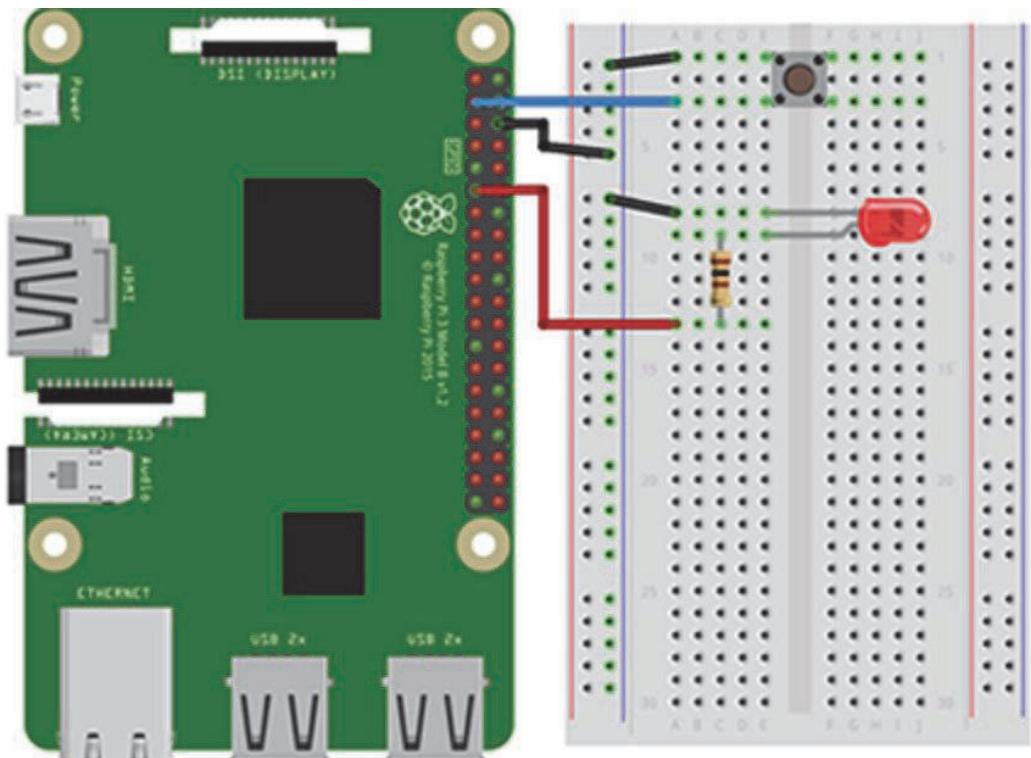
Activación SSH – Modo comandos

Ahora sí. Ya tenemos todo en condiciones para poder comenzar a programar nuestro proyecto.

GPIO

En este capítulo nos introduciremos en el uso de la librería GPIO. Como vimos en el capítulo anterior, podremos transmitir o recibir señales de cada uno de los pines de la placa.

Para poner esto en práctica, diseñaremos un circuito con el cual será posible encender o apagar un led mediante un pulsador.



Diseño eléctrico de nuestro proyecto

02

Los materiales que utilizaremos son:

- Una placa Raspberry pi 3.
- Un diodo led.
- Una resistencia de 100Ω .
- Un pulsador.
- Una protoboard.
- Tres cables macho-hembra.
- Dos cables macho-macho.

Para escribir nuestro código, crearemos una carpeta llamada proyectos y, dentro de esta, un archivo con el nombre proyecto1.py en donde escribiremos nuestro código Python.

DISEÑO DEL CIRCUITO

Para asegurarnos un correcto funcionamiento, conectaremos siguiendo el siguiente detalle:

CONEXIONES PROTOBOARD-PLACA

- Tomaremos un cable macho-hembra y conectaremos la columna 2 de la zona de alimentación de nuestra protoboard al pin 6 (GND).
- Desde el pin 3 (BCM 2), conectaremos la línea 3.
- Luego, desde el pin 11, conectaremos la línea 13.
- Conectaremos la fila 1 y 8 con la columna 2 de la zona de alimentación, que ya habíamos conectado GND de la placa.

CONEXIONES DE ELEMENTOS

- El pulsador lo colocaremos puenteados la fila 1 y 3 de la protoboard.
- El LED, lo conectaremos: positivo en fila 9 y negativo en fila 8.
- Colocaremos la resistencia de manera que haga un puente entre la fila 13 y la fila 9 para alimentar el led de manera segura.

ALTERNATIVAS SI NO TENEMOS LA PLACA

Ya contamos con nuestro circuito armado y con el archivo creado listo para recibir el código que ejecutaremos en nuestra Raspberry pi. Solo nos resta comenzar a codificar. Pero si no tenemos una placa, podemos simular de manera online nuestro proyecto.

Para esto, accederemos a un simulador online gratuito alojado en <https://create.withcode.uk/>,

donde podremos codificar y observar los resultados sin necesidad de tener una placa.

Cálculos de Resistencia

Sabemos que nuestro led funciona con un voltaje de 2,1V, con una corriente máxima de 20mA (0,02A).

También sabemos que los pines GPIO trabajan con 3,3V.

Esto nos indica que el led tiene una sobrecarga de:

$$3,3V - 2,1 = 1,2V$$

Si conectamos el diodo directamente al pin, vamos a quemarlo, entonces necesitaremos una resistencia para obtener una caída de tensión de al menos 1,2V.

$$R = V/I \text{ entonces } R = 1,2V / 0,02A = 60\Omega$$

Es decir, con la resistencia de 100Ω resguardamos nuestro led.



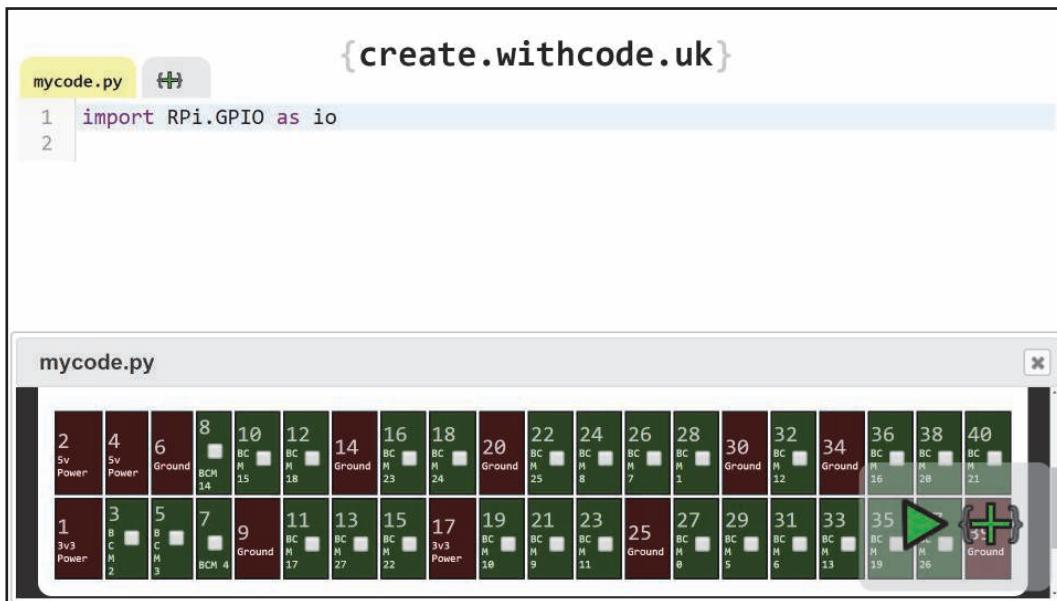
```
mycode.py {create.withcode.uk}
1 name = input("What is your name?")
2 print("Hello " + name)
```

The screenshot shows a simple Python script named 'mycode.py' in a code editor. The code consists of two lines: 'name = input("What is your name?")' and 'print("Hello " + name)'. There is a green play button icon in the bottom right corner of the editor window.

Simulador online de Python.

Su uso es simple e intuitivo, ya que va cargando componentes a medida que importamos bibliotecas en nuestro código.

En el caso de importar GPIO, nos mostrará la siguiente pantalla:



Módulo de GPIO cargado.

En este módulo, visualizaremos una luz, que indica el estado de un pin, y podremos tildar para indicar que está recibiendo una señal y simular así el comportamiento de nuestro proyecto.

NUESTRO CÓDIGO

Lo primero que haremos es importar la librería que nos permitirá codificar y manipular cada uno de los pines GPIO.

```
importRPi.GPIO as io
```

Es importante colocarle un alias (en este caso io), para simplificar el llamado posterior a la librería. Luego, para tener un mejor manejo del tiempo y las esperas, importamos la librería time:

```
import time
```

Antes de avanzar, detengámonos a analizar estas librerías para conocer brevemente de qué se tratan.

LIBRERÍA TIME

Esta biblioteca incluye un conjunto de funciones que nos permiten trabajar con fecha y hora. Entre las más importantes, podríamos mencionar:

- **time()**: nos devuelve la cantidad de segundos que transcurrieron desde las 0 h del 1 de enero de 1970.
- **ctime()**: esta función convierte una expresión devuelta por **time()** en una cadena del tipo **semana mes día hora:minutos:segundos año**. Por ejemplo **sat nov 1 20:30:01 2019**.
- **gmtime()**: obtención del tiempo UTC (tiempo universal coordinado) a partir de los segundos. Esta función nos devuelve un objeto que posee los siguientes atributos:
 - **tm_year**: año
 - **tm_mon**: mes
 - **tm_mday**: día del mes
 - **tm_hour**: hora
 - **tm_min**: minutos
 - **tm_sec**: segundos
 - **tm_wday**: día de la semana [0, 6]
 - **tm_yday**: día del año [1, 366]
 - **sleep(n)**: función que pone en suspensión n segundos la ejecución del programa.

LIBRERÍA GPIO

Como ya se ha dicho, esta librería es la compuerta entre los pines de entrada/salida de la placa y Python. Por tal razón resulta más que interesante el hecho de conocer cuáles son las funciones más importantes que contiene.

- **setmode(modo)**: indica en qué modo vamos a referenciarnos a los pines. Podemos darle dos valores: de **GPIO.BCM** o por la placa (**GPIO.BOARD**).
- **setup(pin,modo)**: configura para qué vamos a utilizar el pin. Podemos darle dos valores: a modo **GPIO.OUT** (salida) o **GPIO.IN** (entrada).
- **output(pin,valor)**: indicamos el valor que sacaremos por el pin señalado. Puede tomar dos valores: **GPIO.HIGH** (emitimos señal, 3.3V) o **GPIO.LOW** (no emitimos, 0V).
- **input(pin)**: lee el valor del pin indicado. El resultado puede estar entre dos **GPIO.HIGH** (se reciben 3.3V) o **GPIO.LOW** (no se recibe nada, 0V).
- **PWM(pin,frecuencia)**: modulación por ancho de pulso. Es un modo en el que se modula la señal para simular una transferencia analógica al pin indicado controlando la cantidad de energía que se entrega o se recibe.
- **cleanup()**: limpia el estado de todos los puertos empleados, devolviéndoles su estado original.

CONTINUEMOS CON EL CÓDIGO

Ya importamos las librerías que vamos a utilizar en nuestro proyecto, ahora nos toca comenzar a configurar de qué modo utilizaremos estos pines.

En este caso, lo referenciaremos según el canal empleado por el procesador, por tal motivo configuraremos el modo en **BCM**.

```
GPIO.setmode(GPIO.BCM)
```

Si miramos nuestro esquema, veremos que el pulsador lo conectaremos al pin 3 (BCM2), y el led, en el pin 11 (BCM17). Para ello, configuraremos el BCM2 como entrada y el BCM17 como salida.

```
GPIO.setup(2, GPIO.IN)
GPIO.setup(17, GPIO.OUT)
```

Pero para no tener que estar continuamente pensando qué número representa cada elemento, vamos a definir una variable para el pulsador y una para el LED. Y referenciaremos el pin en el **setup** con estas variables.

```
pulsador = 2
led = 17
GPIO.setup(pulsador, GPIO.IN)
GPIO.setup(led, GPIO.OUT)
```

Lo que haremos ahora es generar un ciclo de 100 vueltas, con un retardo de medio segundo, que leerá continuamente el estado del pulsador, y en caso de obtener un valor, encenderá el led poniendo el estado del pin en alto. En caso contrario, lo pondrá en bajo.

Una vez finalizado el ciclo, limpiamos el estado de todos los pines utilizados.

```
for i in range(100):
    inputValue = GPIO.input(pulsador)
    if (inputValue == True):
        GPIO.output(led, GPIO.HIGH)

    else:
        GPIO.output(led, GPIO.LOW)

    time.sleep(0.5)

GPIO.cleanup()
```

{create.withcode.uk}

```

mycode.py +++
1 import RPi.GPIO as GPIO
2 import time
3
4 GPIO.setmode(GPIO.BCM)
5
6 pulsador = 2
7 led = 17
8 GPIO.setup(pulsador, GPIO.IN)
9 GPIO.setup(led, GPIO.OUT)
10 for i in range(100):
11     inputValue = GPIO.input(pulsador)
12     if (inputValue == True):
13         GPIO.output(led, GPIO.HIGH)
14     else:
15         GPIO.output(led, GPIO.LOW)
16
17     time.sleep(0.5)
18
19 GPIO.cleanup()

```



Simulador en ejecución y código completo

Hackeo a Minecraft Pi

Minecraft Pi es una versión libre de este popular juego reescrito exclusivamente para las placas Raspberry pi. Está pensado como una herramienta educativa para programadores y se puede descargar gratuitamente bajo licencia GNU desde <https://www.minecraft.net/en-us/edition/pi/>.

Esta versión del juego viene preinstalada en el sistema operativo Raspbian, por eso es recomendable iniciar esta placa con este sistema operativo para simplificar tareas y evitar problemas de compatibilidades.



Minecraft Pi –
Versión del juego para Raspberry

INSTALACIÓN DE MINECRAFT PI

Si por algún motivo nos vemos en la necesidad de instalar Minecraft Pi, deberemos seguir estos pasos.

- Descargarlo desde la página indicada antes.
- Abrir una consola de comandos.
- Con el comando **cd** navegar hasta la carpeta donde hemos descargado el juego, por lo general, en la carpeta **Download**.
- Descomprimir el archivo descargado: **tar -zxvf minecraft-pi-0.1.1.tar.gz**.
- Acceder a la carpeta de instalación: **cd mcpi**.
- Ejecutar Minecraft Pi: **./minecraft-pi**.

También podemos descargar el juego directamente desde la consola de comando, mediante el comando wget de la siguiente manera:
wget https://s3.amazonaws.com/assets.minecraft.net/pi/minecraft-pi-0.1.1.tar.gz.

03

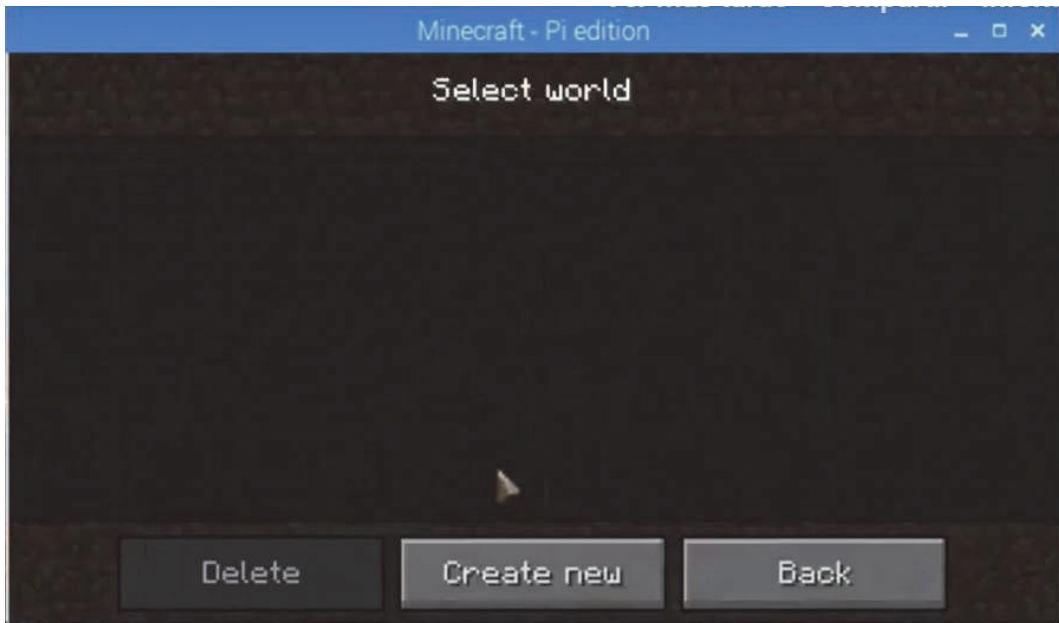
PRIMEROS PASOS

Para programar órdenes en el juego desde Python, lo primero que haremos es iniciararlo. Esto podemos hacerlo desde la consola de comandos, accediendo a la ruta donde se halla guardado el juego, o desde el menú **Game** de nuestra placa Raspberri, seleccionando **Minecraft Pi**



Pantalla inicial del juego.

Una vez iniciado el juego, comenzamos una nueva partida presionando **Start Game** y, en la pantalla que se nos abre a continuación, hacemos clic en **Create new**.



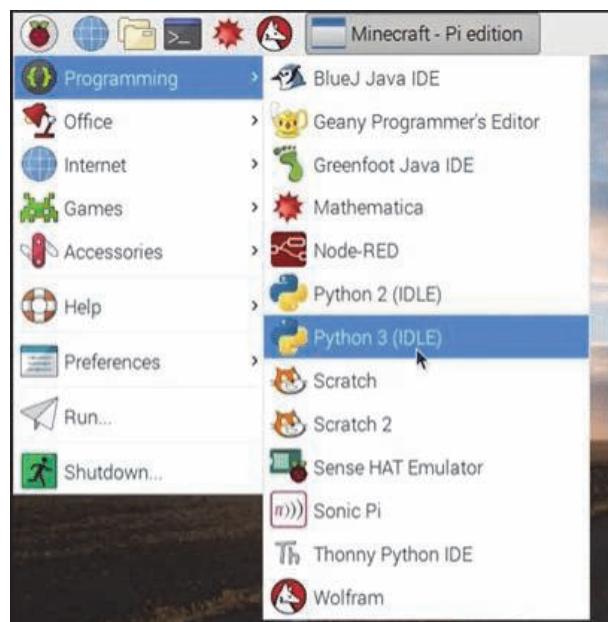
Creación de una nueva partida.

Una vez hecho esto, ya tenemos el juego listo para utilizar. Lo siguiente es abrir el IDLE de Python para poder programar. Para esto, nuevamente desde el menú principal de programas, seleccionamos:

Programming->Python 3 (IDLE).

Es importante que tengamos en todo momento el juego abierto para que el código que ejecutemos desde el IDLE se vea reflejado.

A partir de ahora, tenemos dos opciones: crear un archivo donde pondremos las rutinas para luego ejecutarlas, o ir escribiendo código a medida que jugamos para obtener los resultados.



Ejecución del IDLE de Python.

HOLA MINECRAFT

Vamos a programar nuestro primer script. En este caso, lo vamos a hacer directamente desde el IDLE sin necesidad de crear un archivo aparte.

Para lograrlo, es importante que tengamos abierto tanto el juego como Python.

CONECTARNOS CON MINECRAFT

Para conectarnos con el juego desde Python, debemos importar la librería **Minecraft** perteneciente al paquete **mcpi**. Como ya sabemos, lo realizaremos con el comando **import**.

```
from mcpi.minecraft import Minecraft
```

Luego, deberemos crear un objeto con la función **create()**, que nos servirá de enlace entre el lenguaje y el juego.

```
mc = Minecraft.create()
```

Esta función puede ser llamada con o sin parámetros. Si la llamamos sin parámetros, nos referenciará a una instancia del juego que está en ejecución en el mismo dispositivo que Python y en el mismo instante.

Sin embargo, podemos llamarla pasándole dos parámetros: **dirección** y **puerto**.

```
mc = Minecraft.create("192.168.1.33", 4711)
```

Este código crea una instancia para enlazarse a un mundo en el dispositivo cuya ip es 192.168.1.33, y la comunicación se establecerá mediante el puerto 4711.

A partir de ahora, ya podremos manipular el mundo de Minecraft. Para ello, solo debemos referenciar algún método del objeto **mc**.

Para poder escribir nuestro mensaje en el chat, lo que debemos hacer es llamar al método **postToChat(texto)**. Este método escribe directamente en el chat del juego el texto que le estemos pasando como parámetro, por eso lo utilizaremos con el texto **Hola Minecraft**.

```
mc.postToChat("Hola Minecraft")
```



Resultado de nuestro primer script.

Veremos cómo, con estas tres líneas, pudimos manipular nuestro juego de manera externa.

A continuación, diseñaremos un pequeño refugio simple, pero antes de meternos en el código vamos a ver algunos de los comandos que tenemos disponibles en la librería y que nos permitirán automatizar distintas tareas en el juego.

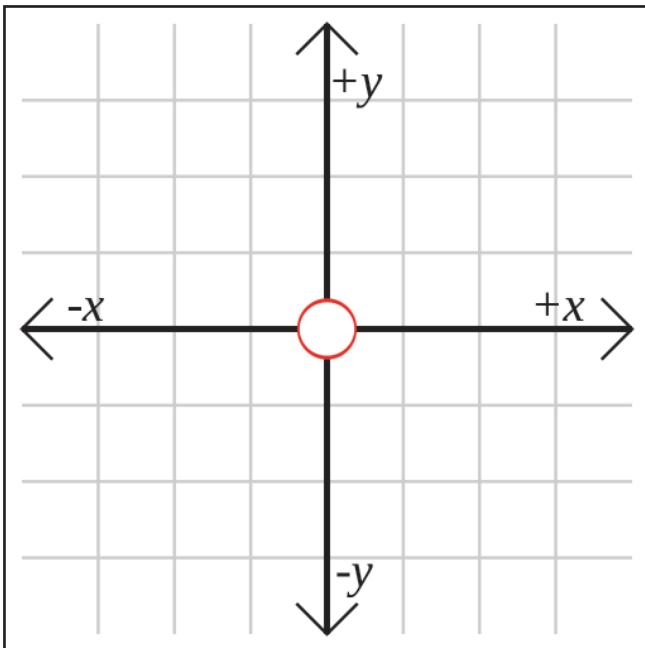
COMANDOS PYTHON PARA MINECRAFT PI

Existen distintos comandos con los que podremos hacer una variedad de acciones instantáneas en el juego solo con escribirlos en Python. De hecho, podremos crearnos una interfaz con botones para ejecutar tareas rutinarias y ponerlas en funciones al hacerle clic.

Muchos de los comandos que se ejecutan hacen referencia a una posición espacial del mundo de Minecraft, por eso es importante que entendamos de qué se trata dicha posición espacial.

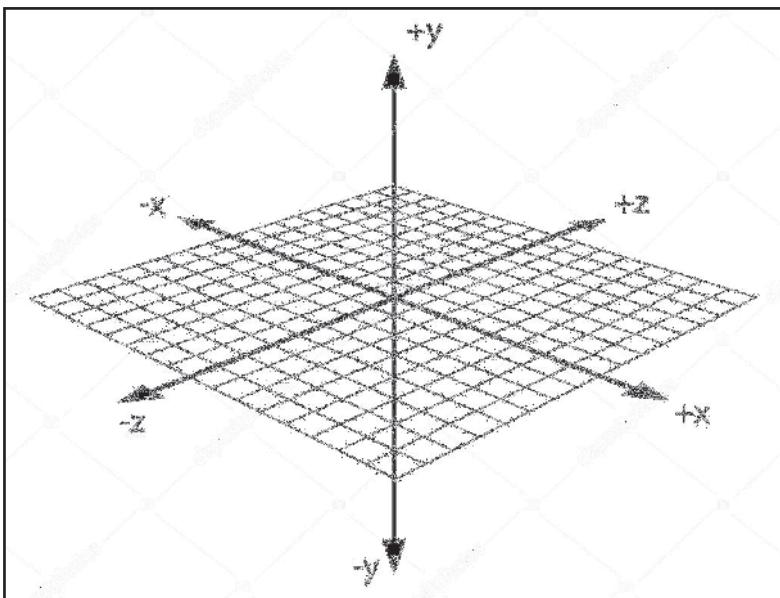
POSICIÓN ESPACIAL

Si nos remontamos a nuestras clases de matemática, recordaremos que, en el momento de querer dibujar en dos dimensiones, nos bastaba con un simple eje de dos coordenadas (**x** e **y**).



Ejes de
coordenadas en 2
dimensiones

Pero si quisiéramos representar algo en tres dimensiones (supongamos un punto), vamos a necesitar un eje más, por lo que obtendríamos un sistema de coordenadas de tres ejes (**x**, **y** y **z**), donde **x** representa el ancho, **y** la altura y **z** la profundidad.



Ejes de
coordenadas en 3
dimensiones.

Como el mundo en Minecraft está en tres dimensiones, es lógico pensar que necesitamos un eje de tres coordenadas para representar cada elemento. Y para simplificar las cosas, la unidad tomada es un bloque. Es decir que cada uno posee una unidad de ancho, por una de profundidad, por una de alto. De esta forma, si nos movemos cuatro posiciones hacia la derecha, lo que estamos haciendo es movernos cuatro bloques a la derecha.

En un principio, el mundo está lleno de bloques vacíos, que se van llenando con bloques de distintos materiales (piedra, césped, aire, etcétera).

Bloque - Id		Bloque - Id		Bloque - Id	
AIR	0	SANDSTONE	24	CRAFTING_TABLE	58
STONE	1	BED	26	FARMLAND	60
GRASS	2	COBWEB	30	FURNACE_INACTIVE	61
DIRT	3	GRASS_TALL	31	FURNACE_ACTIVE	62
COBBLESTONE	4	WOOL	35	DOOR_WOOD	64
WOOD_PLANKS	5	FLOWER_YELLOW	37	LADDER	65
SAPLING	6	FLOWER_CYAN	38	STAIRS_COBBLESTONE	67
BEDROCK	7	MUSHROOM_BROWN	39	DOOR_IRON	71
WATER_FLOWING	8	MUSHROOM_RED	40	REDSTONE_ORE	73
WATER	8	GOLD_BLOCK	41	SNOW	78
WATER_STATIONARY	9	IRON_BLOCK	42	ICE	79
LAVA_FLOWING	10	STONE_SLAB_DOUBLE	43	SNOW_BLOCK	80
LAVA	10	STONE_SLAB	44	CACTUS	81
LAVA_STATIONARY	11	BRICK_BLOCK	45	CLAY	82
SAND	12	TNT	46	SUGAR_CANE	83
GRAVEL	13	BOOKSHELF	47	FENCE	85
GOLD_ORE	14	MOSS_STONE	48	GLOWSTONE_BLOCK	89
IRON_ORE	15	OBSIDIAN	49	BEDROCK_INVISIBLE	95
COAL_ORE	16	TORCH	50	STONE_BRICK	98
WOOD	17	FIRE	51	GLASS_PANE	102
LEAVES	18	STAIRS_WOOD	53	MELON	103
GLASS	20	CHEST	54	FENCE_GATE	107
LAPIS_LAZULI_ORE	21	DIAMOND_ORE	56	GLOWING_OBSIDIAN	246
LAPIS_LAZULI_BLOCK	22	DIAMOND_BLOCK	57	NETHER_REACTOR_CORE	247

Código de id de los bloques en Minecraft Pi.

Cada bloque, cada objeto dentro de nuestro mundo, se encuentra en una coordenada determinada por los valores asignados a **x**, **y** y **z**.

Sabiendo esto, podemos aprovechar Python para agregar, quitar o modificar un bloque. También es posible posicionar nuestro personaje en distintos puntos del espacio cambiándole al menos una de estas coordenadas.

Para comenzar a interactuar, es importante que sepamos cuál es nuestra posición actual, de esa manera podremos alterar el mundo en torno al personaje.

Para obtener la posición, deberemos escribir:

```
pos = mc.player.getPos()
```

La función **getPos()** nos va a devolver un objeto con tres valores, cada uno representa a una de las coordenadas espaciales.

Entonces, resulta eficaz crear tres variables donde almacenamos cada una de estas coordenadas.

```
x = pos.x  
y = pos.y  
z = pos.z
```

Otra forma de hacer lo mismo es utilizar una técnica de Python, que consiste en asignar las variables en una sola línea como sigue:

```
x, y, z = mc.player.getPos()
```

Como se observa, reducimos tres líneas el código de nuestro script. Si consideramos que tenemos muy limitada la cantidad de memoria disponible, comprenderemos rápidamente la importancia de optimizar lo más que podemos el código.

Si queremos trasladar a nuestro personaje como si lo estuviéramos teletransportando, deberemos utilizar la función **setPos(x,y,z)**. Como imaginamos, asignaremos una nueva posición al personaje estableciéndole las tres coordenadas espaciales como parámetro.

Entonces, si quisieramos crear un script para teletransportarnos diez bloques más adelante, bastaría con escribir el siguiente código:

```
from mcpi.minecraft import Minecraft  
  
mc = Minecraft.create()  
x, y, z = mc.player.getPos()  
mc.player.setPos(x+10,y,z)
```

Ahora que sabemos la posición de nuestro personaje y cómo movernos, vamos a agregar un bloque frente a nosotros. Para ello utilizaremos la función **setBlock()**.

Esta función recibe los siguientes parámetros:

- **x**: coordenada en x;
- **y**: coordenada en y;
- **z**: coordenada en z;
- **id**: id de bloque a ingresar;
- **data**: este parámetro es opcional y se utiliza solo en algunos bloques. Su función es asignarle alguna propiedad especial. Por ejemplo, si insertamos un bloque del tipo lana (**id = 35**), podemos variar este parámetro para indicar si es blanco (**data = 0**), naranja (**data = 1**), magenta (**data = 2**), celeste (**data = 3**) o amarillo (**data = 4**).

Insertemos entonces un bloque de piedra frente al personaje. Lo que haremos es:

- Definir una variable **piedra** a la que le asignaremos el valor correspondiente al id del bloque de piedra.
- Obtener la posición actual.
- Insertar el bloque en la misma posición que el personaje, pero aumentando en uno **x** o **z** (si cambiamos **y**, situaríamos el bloque por encima del personaje).

```
from mcpi.minecraft import Minecraft  
piedra = 1  
  
mc = Minecraft.create()  
x, y, z = mc.player.getPos()  
mc.setBlock(x+1, y, z,piedra)
```



Insertamos un bloque tipo piedra frente al personaje.

AVANCEMOS UN POCO MÁS EN NUESTRO CÓDIGO

Lo que vamos a hacer ahora es que florezca todo a nuestro paso, en otras palabras, insertaremos un bloque tipo flor a medida que vamos caminando. Para ello deberemos realizar un script que no finalice nunca, y que cada corto tiempo obtenga la posición de nuestro personaje e intercambie el bloque existente debajo por un bloque del tipo flor (**id = 38**).

Para pausar el código, utilizaremos la función **sleep()**, de la librería **time**, por lo que debemos importarla a nuestro script.

```
from mcpi.minecraft import Minecraft
from time import sleep
flor = 38

mc = Minecraft.create()

while True:
    x, y, z = mc.player.getPos()
    mc.setBlock(x,y-1,z,flor)
    sleep(0.1)
```

Pero si queremos ser realistas, deberíamos poner una validación más, porque nuestro código nos inserta una flor debajo de nosotros sin importar el tipo de bloque en el que estemos parados, y podremos poner una en el agua, en la arena, en el aire o en el césped. Pero lo coherente sería que solo floreciera cuando caminamos por el césped, por este motivo antes de poner una flor, chequearemos si el bloque debajo de nosotros tiene un **id = 2 (Grass)**.

```
from mcpi.minecraft import Minecraft
from time import sleep
flor = 38
cesped = 2

mc = Minecraft.create()
while True:
    x, y, z = mc.player.getPos()
    mipiso = mc.getBlock(x,y-1,z)
    if mipiso == cesped:
        mc.setBlock(x,y-1,z,flor)

    sleep(0.1)
```

Veremos cómo ahora, a medida que caminamos, solo pondremos una flor si lo que estamos pisando es un bloque del tipo césped. Si caminamos sobre cualquier otra cosa, no pasa absolutamente nada.

Como nuestro código es infinito, si queremos frenarlo, deberemos presionar la combinación de teclas **Ctrl + C** en la ventana de Python.

ARMAR UN CUBO SÓLIDO DE BLOQUES

Si quisiéramos crear un cubo de 5 x 5 x 5 bloques, la primera solución que encontraríamos sería la de anidar tres **for** de la siguiente manera:

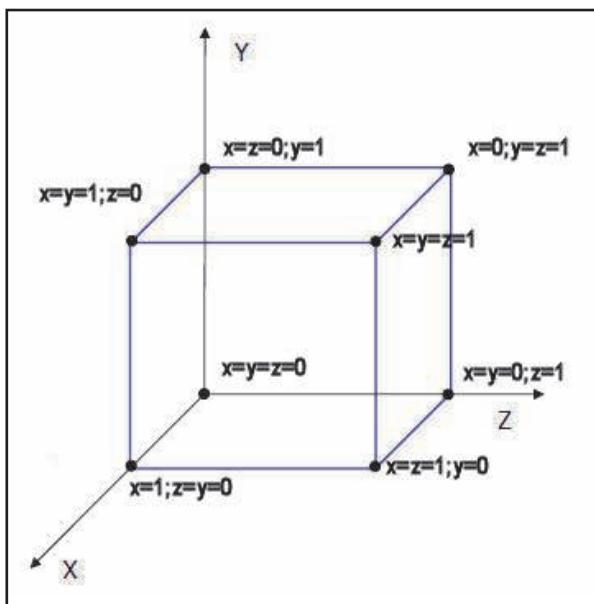
```
piedra = 1
x, y, z = mc.player.getPos()
for iz in range(1,6):
    for iy in range(1,6):
        for ix in range(1,6):
            mc.setBlock(x + ix,y + iy,z + iz,piedra)
```

Pero como la premisa que debemos considerar siempre, atendiendo la memoria limitada que podamos llegar a tener en nuestra placa, es la de reducir el código lo más posible, podemos utilizar la función **setBlocks()**, que hará el mismo trabajo, pero en una sola línea.

Esta función, recibe dos coordenadas, el tipo de bloque y opcionalmente alguna característica del bloque por insertar.

```
mc.setBlocks(x1, y1, z1, x2, y2, z2, bloque,[data])
```

Para entender el desarrollo de esta función, deberemos visualizar un cubo en un eje de coordenadas tridimensional, y centrarnos en dos vértices opuestos. La idea es reconocer las coordenadas de estos vértices, y Python llenará el cubo automáticamente con el bloque indicado.



Coordenadas de vértices de un cubo.

Con esta función, el código anterior quedaría de la siguiente manera:

```
piedra = 1
x, y, z = mc.player.getPos()
mc.setBlocks(x + 1, y + 1, z + 1, x + 6, y + 6, z + 6, piedra)
```

Observemos que, tanto en el primer código como en el segundo, el cubo de bloques comienza a crearse a partir de la posición del personaje más 1, esto es para asegurarnos que no lo aplastaremos al crear nuestra estructura.

ARMEMOS NUESTRA CASA

Ahora armarnos una casa de manera automática. Siguiendo con el concepto minimalista, trataremos de hacerlo con la menor cantidad de código posible para evitar sobrecargar la memoria de nuestra Raspberry pi.

Una forma sencilla de llevarlo a cabo es crear un cubo sólido de madera, y en su interior crear un cubo de aire. Luego, nos deberíamos encargar de agregar una ventana y una puerta en una de sus paredes.

Pasemos al código.

Como siempre, lo primero que haremos es importar las librerías que vamos a emplear:

```
from mcpi.minecraft import Minecraft
```

Luego crearemos una variable que nos permita almacenar el id del bloque que utilizaremos como estructura. Como vamos a utilizar el bloque de madera, trabajaremos con id = 5:

```
estructura = 5
```

Como este tipo de bloque acepta distintos tipos, crearemos otra variable a la cual le asignaremos el número que representa al deseado:

```
tipo = 3
```

Lo que estamos haciendo al crear estas dos variables es parametrizar el bloque que utilizaremos. De manera que, si deseamos realizar la casa con otro material, no tengamos que cambiar todo el código, solo bastará con cambiarle el valor a estas variables.



JUNGLE WOOD PLANK
Minecraft Id: 5:3

Características del bloque a emplear.

Nos conectamos con el juego, creando su instancia y asignándola a una variable. Luego obtenemos la posición del personaje:

```
mc = Minecraft.create()
x, y, z = mc.player.getPos()
```

Para dejar listas las variables de posición, vamos a aumentar el valor de **z** en **15**. De esta manera, no tenemos que preocuparnos si la casa puede llegar a caernos encima:

```
z = z + 15
```

Al aumentar **z** en 15 unidades, estamos definiendo que la casa se va a construir 15 bloques frente a nosotros.

Una vez que tengamos las coordenadas listas, pasamos a crear el primer bloque macizo, el cual lo haremos de 10 x 10 x 4.

```
mc.setBlocks(x, y - 1, z,x + 11, y + 4, z + 11,estructura, tipo)
```

El parámetro **y** lo iniciamos desde la posición **-1**. Lo hacemos así para que el piso de nuestra casa se encuentre a la altura del suelo que estamos pisando y no sobre un escalón de un bloque.

Hasta el momento, construimos un bloque sólido con el material elegido, lo que vamos a hacer ahora es ahuecarlo. Para ello crearemos un cubo de 9 x 9 x 3 desde la posición **x + 1, y y z + 1** hasta la posición **x + 10, y + 3 y z + 10**.

En este caso, el tipo de bloque va a ser aire, por lo que emplearemos el **id = 0**.

```
mc.setBlocks(x + 1, y, z + 1,x + 10, y + 3, z + 10,0)
```

Con esto ya tenemos un cubo hueco realizado con el material escogido, solo nos faltan la puerta y la ventana. Para la puerta, dejaremos una abertura de 2 x 3. Es decir, **id = 0**. Pero para parametrizar el valor, crearemos una variable **puerta** con el fin de que en el futuro podamos cambiar el material. Para la ventana, emplearemos ocho bloques tipo vidrio (**id = 20**), por lo que insertaremos cuatro bloques con **y = 1** en la mitad de la pared donde coloquemos la puerta, y otros cuatro con **y = 2**.

Entonces, sabiendo que en las coordenadas (**x; y; z**) de nuestro mundo comienza una pared, y finaliza en (**x + 10; y + 4; z**), podemos insertar la puerta y la ventana de la siguiente manera:

PUERTA

- Bloque1: coordenadas (`x + 1; y; z`)
- Bloque2: coordenadas (`x + 2; y; z`)
- Bloque3: coordenadas (`x + 1; y + 1; z`)
- Bloque4: coordenadas (`x + 2; y + 1; z`)
- Bloque5: coordenadas (`x + 1; y + 2; z`)
- Bloque6: coordenadas (`x + 2; y + 2; z`)

VENTANA

- Bloque1: en coordenadas (`x + 4; y + 1; z`)
- Bloque2: en coordenadas (`x + 5; y + 1; z`)
- Bloque3: en coordenadas (`x + 6; y + 1; z`)
- Bloque4: en coordenadas (`x + 7; y + 1; z`)
- Bloque5: en coordenadas (`x + 4; y + 2; z`)
- Bloque6: en coordenadas (`x + 5; y + 2; z`)
- Bloque7: en coordenadas (`x + 6; y + 2; z`)
- Bloque8: en coordenadas (`x + 7; y + 2; z`)

Sabiendo esto, luego de ahuecar nuestro cubo, insertaremos el siguiente código:

```
for iy in range(0,3):
    for ip in range(1,3): #bloque para puerta
        mc.setBlock(x + ip, y + iy, z,0)
        if iy> 0:
            for iv in range(4,8): #bloque para ventana
                mc.setBlock(x + iv, y + iy,z,20)
```

Hemos puesto una condición antes de insertar la ventana, porque la queremos despegada del suelo.

Podemos, al principio, crear dos variables con las que parametrizaremos la puerta y la ventana. Entonces:

```
puerta = 0
ventana = 20
```

y la insertamos de la siguiente manera:

```
for iy in range(0,3):
    for ip in range(1,3): #bloque para puerta
        mc.setBlock(x + ip, y + iy, z,puerta)
        if iy> 0:
            for iv in range(4,8): #bloque para ventana
                mc.setBlock(x + iv, y + iy,z,ventana)
```

Con esto, ya hemos terminado de armar toda la estructura de nuestra casa, que quedará como sigue:



Diseño final de la casa armada.

Lo último que learemos al código es teletransportar a nuestro personaje dentro de la casa y para ello insertaremos lo siguiente:

```
mc.player.setPos(x + 5,y,z + 5)
```

CÓDIGO COMPLETO

Luego de haber escrito todos los pasos, nuestro código final es el siguiente:

```
from mcpi.minecraft import Minecraft
estructura = 5
tipo = 3
```

```

puerta = 0
ventana = 20

mc = Minecraft.create()
x, y, z = mc.player.getPos()
z = z + 15
mc.setBlocks(x, y - 1, z,x + 11, y + 4, z +
11,estructura, tipo)
mc.setBlocks(x + 1, y, z + 1,x + 10, y + 3, z + 10,0)

for iy in range(0,3):
    for ip in range(1,3): #bloque para puerta
        mc.setBlock(x + ip, y + iy, z,puerta)
        if iy> 0:
            for iv in range(4,8): #bloque para ventana
                mc.setBlock(x + iv, y + iy,z,ventana)

mc.player.setPos(x + 5,y,z + 5)

```

Mientras estemos jugando, bastará con ejecutar nuestro código para que, en forma automática, aparezca una casa lista para ser habitada con nuestro personaje dentro de ella.

OTROS COMANDOS INTERESANTES

Existen muchos comandos dentro de la librería de Minecraft; a continuación, pasaremos a conocer los más destacados:

- **getBlock(x,y,z)**: en muchas situaciones, nos vemos en la necesidad de saber de qué está hecho un determinado bloque. Esta función nos devuelve el **id** del bloque posicionado en las coordenadas pasadas como parámetros.
- **saveCheckpoint()**: con esta función, grabamos la partida para ser recuperada en cualquier momento, sin temor a que se pierdan los cambios o avances realizados hasta el momento.
- **restoreCheckpoint()**: esta función nos permite recuperar una partida previamente almacenada con **saveCheckpoint()**.

MINECRAFT.PLAYER

El objeto **player** de Minecraft nos entrega una serie de funciones con las cuales podremos interactuar con el personaje principal.

Entre las funciones más importantes podemos nombrar:

- **getPos()** y **setPos()**: obtenemos o seteamos la posición actual del personaje.
- **getRotation()**: nos devuelve en qué dirección está mirando el personaje. El resultado es un valor que va de **0** a **360**, representando los grados del ángulo de rotación.

MINECRAFT.CAMERA

Este objeto nos permite manipular la vista del juego. Entre los comandos más destacados podemos nombrar:

- **setNormal(entityid)**: carga la cámara normal para un determinado jugador que pasamos como parámetro.
- **setFollow(entityid)**: con esta función, la cámara entra en modo para seguir a la entidad especificada.

APLICACIONES

En este capítulo vimos algunos de los comandos que la librería **minecraft** para Python nos ofrece. También conocimos algunos métodos para modificar el mundo que rodea a nuestro personaje. Pero las posibilidades son infinitas, y los límites de qué cosas podemos hacer queda en cada uno de nosotros hasta donde nos lleve nuestra inventiva.

Siguiendo con el ejemplo de la casa, podríamos llamar una subrutina para amueblarla. O hacer que aparezcan y desaparezcan elementos en función del tiempo con las funciones **setBlock()** en posiciones aleatorias, con lo que nos entretendríamos como si de un minijuego se tratase, en el que el objetivo sería obtener la mayor cantidad de determinados elementos.

Si bien vimos herramientas suficientes para interactuar con el mundo, podemos acceder a una guía completa del API desde

<https://www.stuffaboutcode.com/p/minecraft-api-reference.html>.

MicroPython

Según el sitio oficial, **MicroPython** es un pequeño pero eficiente intérprete del lenguaje de programación Python 3 optimizado para que pueda ser ejecutado en microcontroladores y ambientes restringidos.

A pasos lentos pero seguros, este lenguaje supo hacerse camino en el IoT, por un lado, por la sencillez en su uso, por la prolíjidad de su código; y por el otro, porque al ser de código abierto cuenta con un sinfín de librerías para poder implementar cualquier tipo de proyecto, comandando una importante cantidad de placas.

VENTAJAS

- Incluye un RELP (Read-Eval-Print-Loop o consola de lenguaje) que lee las instrucciones, las evalúa y procesa los resultados sin necesidad de compilar o cargar los programas en la memoria del dispositivo.
- Muchas librerías para la ejecución de tareas y el simplificado de codificación. Con estas librerías, podemos conectarnos mediante un socket, leer un JSON (sigla en inglés de notación de objetos de JavaScript) desde un sitio web, entre otras posibilidades, con las cuales simplificamos la programación y reducimos el código por cargar.
- Extensible: podemos extender Python con lenguajes de programación de más bajo nivel con los cuales personalizaremos mejor los procesos y resultados.

DESVENTAJAS

- MicroPython está elaborado con las funciones más relevantes de Python 3 y sus librerías. Si bien está basado en su eficiencia, podemos encontrarnos muchas veces con que no contamos con algo que solemos utilizar en su versión completa.
- Como no es un lenguaje de bajo nivel, su ejecución suele ser más lenta que en homólogos realizados en C o C++.

04

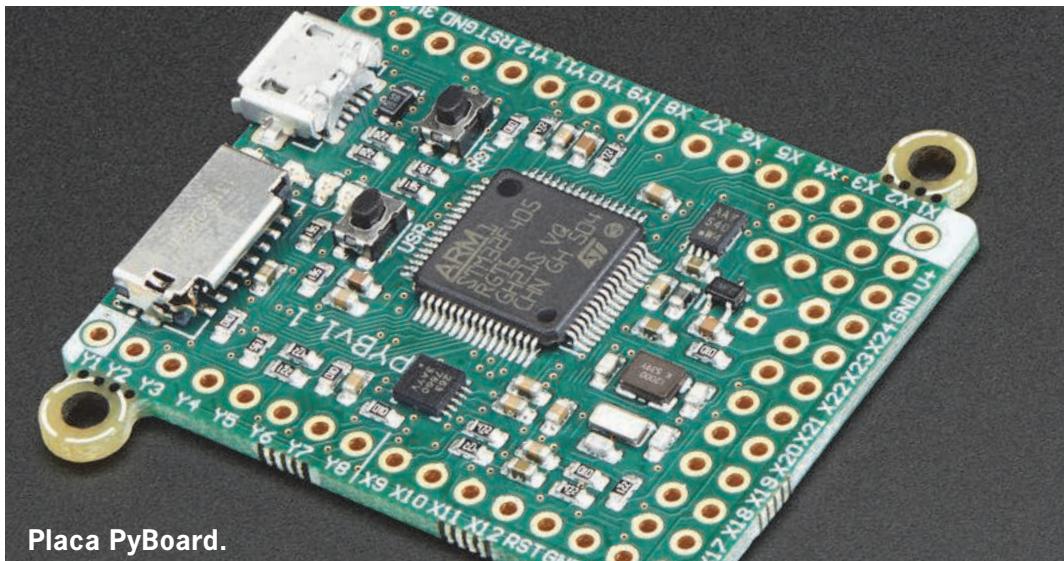
PLACAS COMPATIBLES CON MICROPYTHON

Existen muchas placas compatibles con MicroPython. Si bien cada una tiene sus particularidades, todas se caracterizan por la simplicidad en el uso y la programación de la mano de este potente lenguaje. Algunas de las más empleadas son:

PYBOARD

Esta placa es un diseño compacto pero poderoso que se puede conectar a una PC mediante un cable micro USB, como el de los celulares.

Fue diseñada exclusivamente para ser programada con Python, por lo que posee una integración perfecta con este lenguaje.

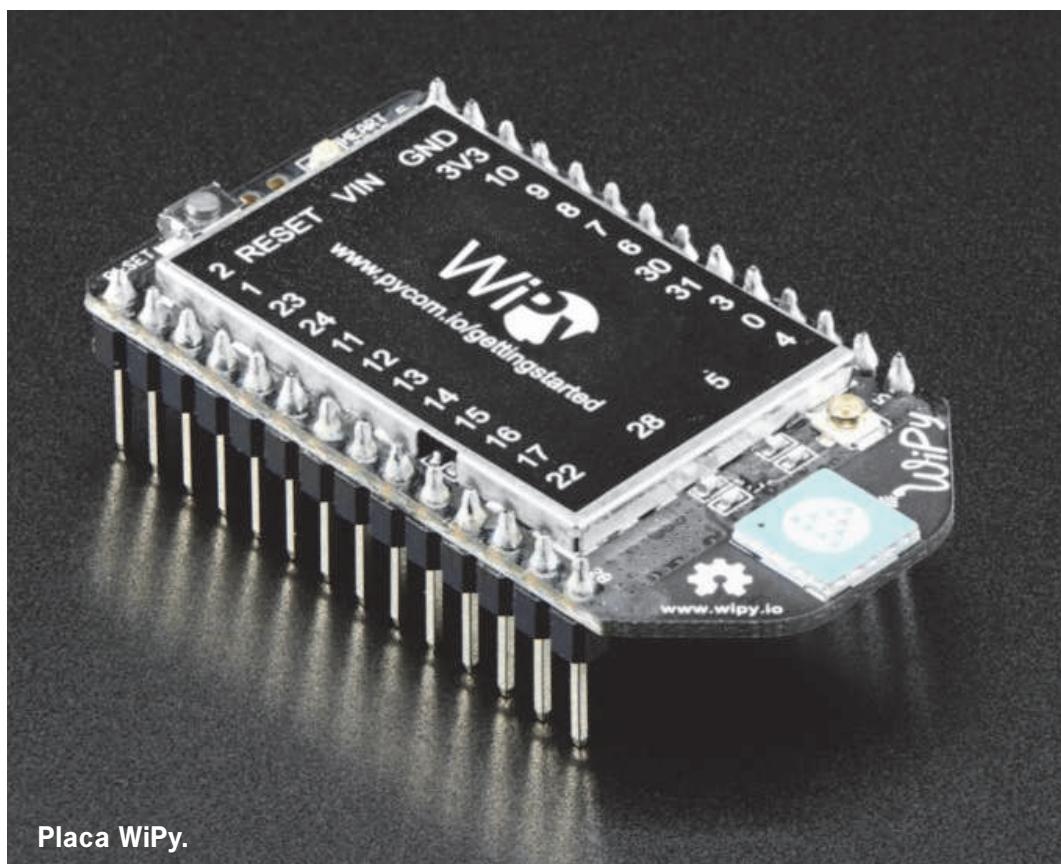


Posee una memoria RAM de 256KB, cuenta con conectividad WiFi y Bluetooth. Si bien no está preparada para que se le instale un sistema operativo como sí sucede con la Raspberry, ofrece una gran versatilidad al permitir ejecutar códigos de manera remota desde un dispositivo anfitrión, o almacenando el código por ejecutar en la memoria interna de la placa para luego hacerla funcionar de manera autónoma desconectada de la PC.

WIPY

Esta placa representa un novedoso desarrollo para IoT, centrado en la conectividad y la ejecución de códigos Python. Está equipada con un procesador de doble chipset **Espressif ESP32** y una placa WiFi capaz de conectar 1 km.

Resulta ideal para proyectos que requieran la conexión remota de un dispositivo con todo el potencial que Python puede ofrecer.



Placa WiPy.

ESP8266

Esta es una placa versátil y poderosa, pensada para ser programada tanto desde el IDE de Arduino como desde Python.

Su grandeza radica en que es posible ejecutar códigos enviados desde un shell a través de internet.

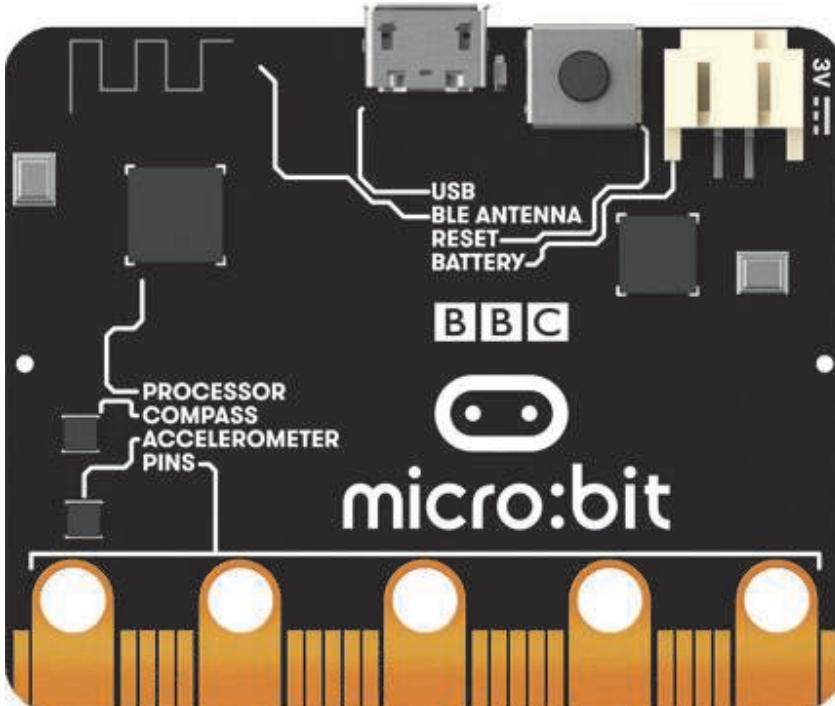
Cuenta con el procesador **ESP8266** de 80Mhz, placa WiFi y una memoria Flash de 4MB para almacenar programas.



BBC MICRO:BIT

BBC micro:bit es un microcomputador programable, útil para distintos proyectos que pueden ir desde un instrumento musical hasta un robot.

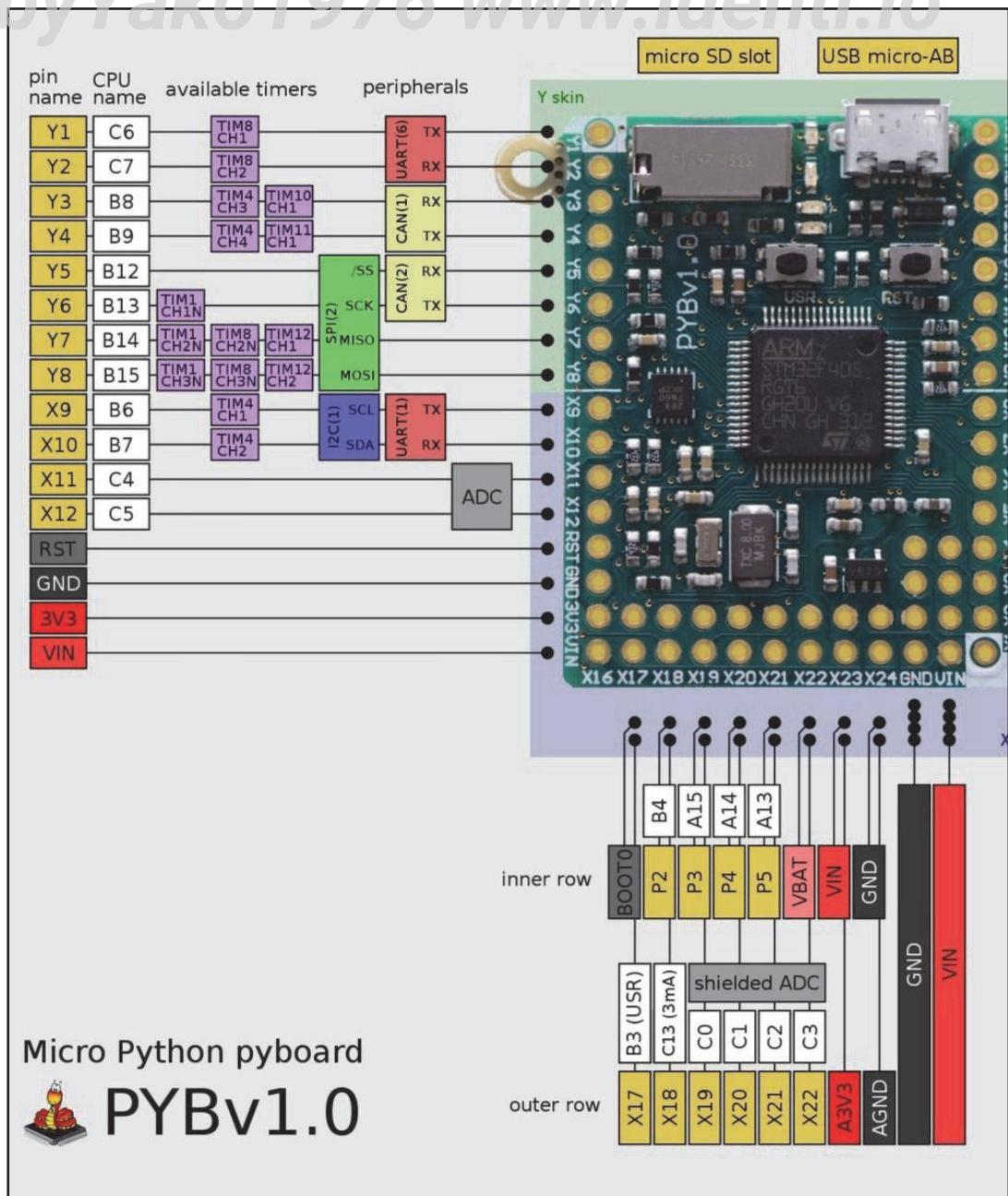
Su punto fuerte es la simplicidad para programar soportando la programación en bloques. Fue diseñada para ser utilizada en ámbitos escolares, y ha obtenido una gran aceptación entre docentes y alumnos.

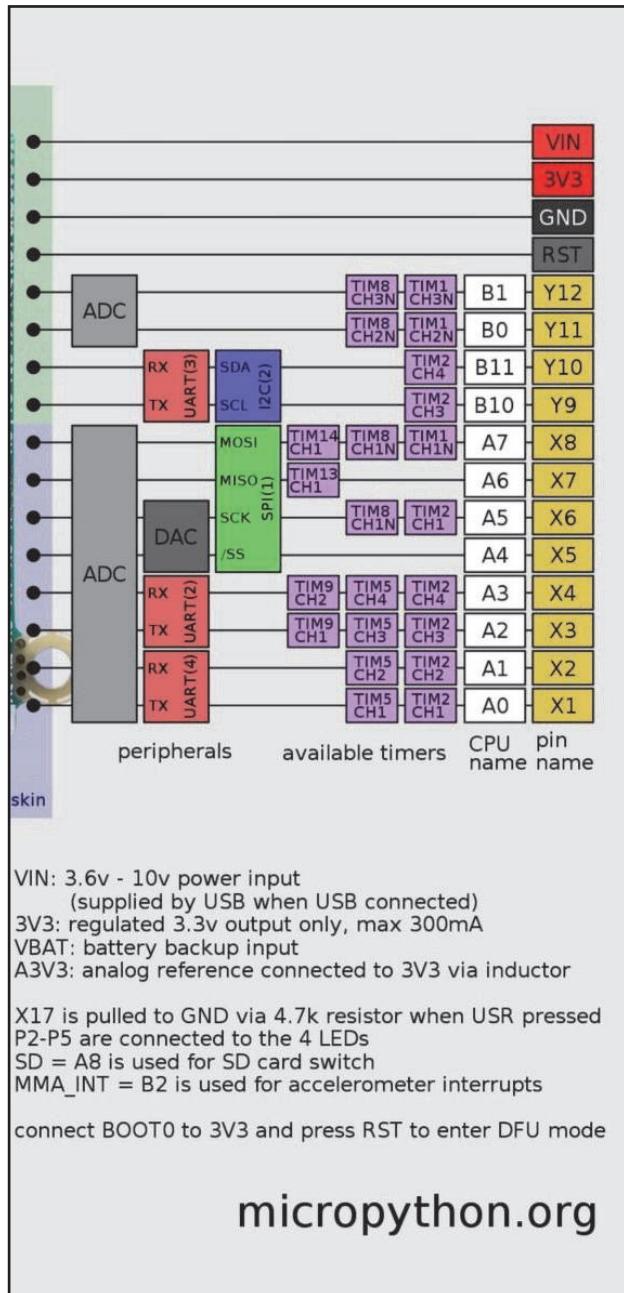


Placa
BBC micro:bit.

PLACAS PYBOARD

by Yako1976 www.identi.io





Como ya hemos visto, esta placa es diseñada exclusivamente para ejecutar códigos Python. Si bien esto parece una limitación, no resulta tal debido a que posee un gran potencial en el hardware. En combinación con un lenguaje simple pero potente, es posible emprender proyectos de gran tamaño gracias a su variedad de pines disponibles con una gran versatilidad de uso y configuraciones.

PRIMEROS PASOS

Cuando conectamos la placa pyboard a nuestra PC, como ya hemos anticipado, tenemos dos maneras de enviarle código: escribiendo en la memoria flash o enviándole código mediante un intérprete de comandos conectado por el puerto serie.

ESCRITURA DIRECTA EN LA MEMORIA FLASH (MSC)

Cuando conectamos la placa a nuestra computadora sin instalar ningún tipo de driver, es reconocida como si se tratase de un pendrive. Si accedemos a la unidad que se crea, nos vamos a encontrar con los siguientes archivos:

- **boot.py**: archivo de configuración de la placa.
- **main.py**: script que va a ejecutar la placa al ser encendida, sin importar si está o no conectada a otro dispositivo.
- **pybcd.inf**: driver necesario para que podamos acceder mediante un puerto serie.
- **README.txt**: archivo de interés.

Para poder introducirle un código a nuestra placa, bastará con abrir y reescribir el archivo **main.py**.

Supongamos que queremos que parpadee el led 1 para indicar que la placa está encendida y funcionando, deberíamos escribir en dicho archivo lo siguiente:

```
import pyb
led = pyb.LED(1)
while True:
    led.toggle()
    pyb.delay(1000)
```

Una vez que hemos actualizado el archivo **main.py**, debemos reiniciar la placa, para ello presionamos el botón de **reset**.

CODIFICACIÓN DIRECTA DESDE UNA CONSOLA SHELL

En este método, podemos ejecutar código directamente desde nuestra computadora, y la placa va a responder sin necesidad de actualizar ningún tipo de archivo ni reiniciarla. Es un buen método para cuando estamos generando un programa y necesitamos chequear el funcionamiento.

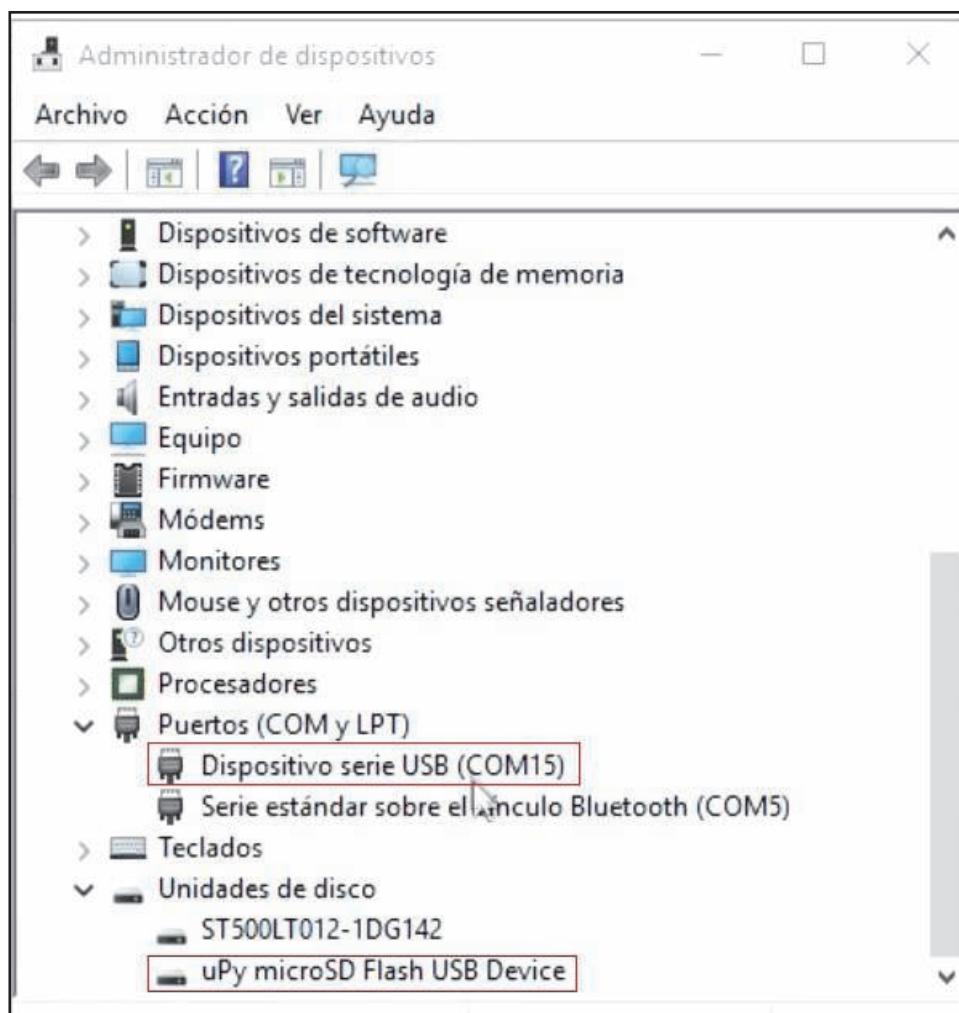
Para poder utilizar este método, deberemos instalar en nuestro equipo:

Recordemos

- **toggle**: cambia de estado el led, si está encendido lo apaga y si está apagado lo enciende.
- **delay**: crea una espera en milisegundos.

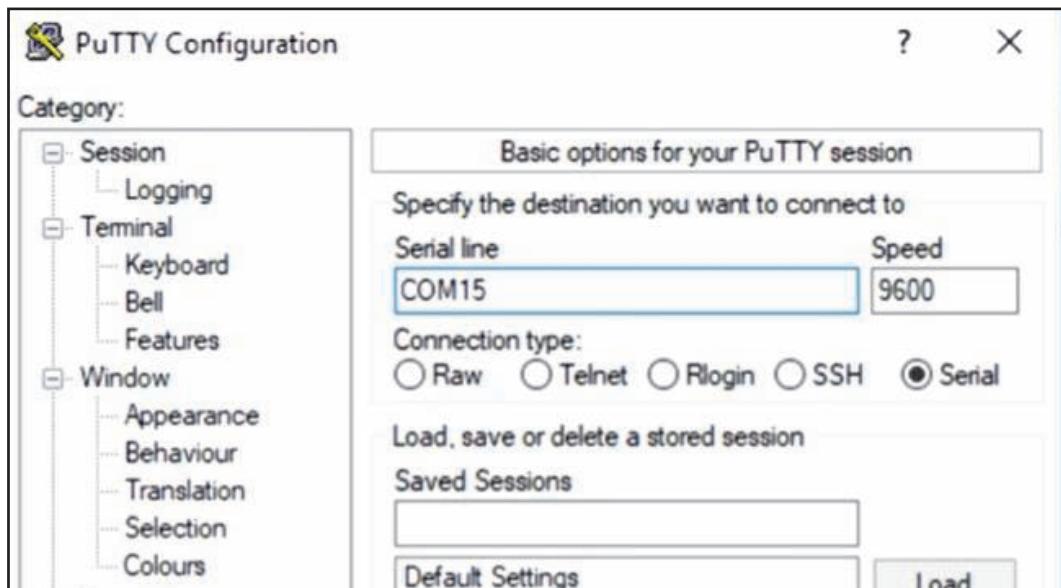
- El driver, que es el archivo **pybcdc.inf** que encontramos en la unidad creada automáticamente al conectar la placa a la PC. Si bien el manual recomienda utilizar solo el archivo incluido en la placa, en Windows 10 la instalación automática suele funcionar correctamente.
- Un Shell de acceso remoto, podríamos utilizar PuTTY por ser de uso gratuito y liviano.

Lo primero que haremos es ir al administrador de dispositivos y chequear en qué puerto serie se nos ha instalado la placa.



Reconociendo placa Pyboard desde administrador de dispositivos.

A continuación, abrimos PuTTY y configuramos para una conexión serial, determinando donde dice **Serial line** el número de puerto que vimos en el administrador de dispositivos.



Configuración serial con Public TTY (PuTTY).

Luego, haciendo clic en **Open** se nos abrirá una consola de comando donde se estará ejecutando MicroPython. En esta consola, podremos ingresar el código directamente y veremos el resultado en la placa de manera inmediata sin necesidad de resetearla.

Cabe aclarar qué, utilizando este método, no se almacena nada en el firmware, por lo que todo lo que escribamos se va a perder al cerrar la consola.

A screenshot of a terminal window titled 'COM15 - PuTTY'. The window displays MicroPython version v1.9.2.2 running on a STM32F405 board. The prompt '=>>>' is visible at the bottom left. The rest of the screen is black.

Consola MicroPython con PuTTY.

MÓDULO PYB

Este módulo posee funciones específicas de la placa pyboard. Entre las funciones que más se destacan podemos enumerar las siguientes:

- **pyb.delay(ms)**: crea una interrupción de **ms** milisegundos.
- **pyb.udelay(us)**: crea una interrupción de **us** microsegundos.
- **pyb.millis()**: devuelve la cantidad de milisegundos desde que la placa ha sido reiniciada.
- **pyb.wfi()**: entra en un estado de bajo consumo esperando una interrupción interna o externa.
- **pyb.stop()**: pone a la placa en un estado de suspensión, y se mantiene hasta que una interrupción externa o una orden interna lo reactive.
- **pyb.have_cdc()**: chequea si el usb está conectado, devuelve **True** si está, o **False** en caso contrario

CLASES

También, podemos encontrarnos con una serie de clases, que nos facilitan el manejo de distintos componentes internos como externos. De esta manera, ya no necesitaríamos programar ni instalar nada adicional, sino que bastará con invocar a la adecuada. Entre las clases más relevantes podemos mencionar:

- **LED()**: nos devuelve un objeto que representa uno de los led incluidos en la placa. Se debe pasar como parámetro el número de led (**1:rojo, 2:verde, 3:amarillo, 4:azul**) y luego darle las opciones **on(),off()** o **toggle()**.

```
import pyb

led1 = pyb.LED(1)      #declaramos la variable led1
                       #con la que manejaremos led
                       #rojo
led1.on()              #la encendemos
led1.off()             #la apagamos
led1.toggle()          #cambiamos es estado, si
                       #está encendida, la apagamos
                       #o encendemos si está apagada
led1.intensity(valor) #indicamos con que intensidad
                       #queremos que brille. Toma un
                       #valor entre 0 y 255
```

- **Accel()**: la placa pyboard incluye un chip acelerómetro, con el cual podemos determinar cualquier movimiento que sufra. Para controlar tales movimientos utilizamos esta clase, que, entre sus métodos más importantes, incluye **x()**, **y()** y **z()** quienes nos devuelven tales coordenadas.

```
import pyb
pos = pyb.Accel()
print (pos.x())           #Imprime coordenada x
print (pos.y())           #Imprime coordenada y
print (pos.z())           #Imprime coordenada z
```

- **Servo()**: nos devuelve un objeto con el cual podremos manipular uno de los tres servos posibles. Una vez declarado el objeto, podremos setear el ángulo de giro y la velocidad, entre otras funciones. Dentro de la placa, cada servo es conectado mediante los pines: GND, VIN y X1 (servo 1), X2 (servo 2) o X3 (servo 3).

```
import pyb
servo = pyb.Servo(1)#Conectamos al servo 1
servo.angle(90, 5000)#Giramos 90º en 5 segundos
```

- **ADC()**: clase para realizar conversión de análogo a digital en cada pin.
- **DAC()**: clase para realizar conversión de digital a análogo en cada pin.
- **Switch**: nos devuelve un objeto que nos permitirá leer el estado del switch de la placa.

```
sw = pyb.Switch()          # Crea el objeto
sw.value()                  # Obtiene el valor
sw.callback(f)              # registra una función para
# llamar cada vez que se
                           # presiona
```

En el sitio oficial de MicroPython (www.micropython.org), es posible encontrar documentación completa sobre el uso de este módulo.

MÓDULO MACHINE

Este módulo tiene asociado declaraciones y funciones relacionadas con cada placa en particular. Cada fabricante crea este módulo respetando la interfaz, de manera que sea posible intercambiar de placa sin variaciones en el código.

ALGUNAS DE LAS FUNCIONES MÁS EMPLEADAS

- **reset()**: resetea el dispositivo como si hubiéramos presionado el botón de reinicio.
- **sleep()**: pone la placa en estado de suspensión, desactivando la CPU y los periféricos, salvo la placa WLAN. Para reactivar la placa, es necesario programar previamente cuál es la interrupción que lo hará.
- **deepsleep()**: similar a **sleep()**, salvo que pone en estado de suspensión total, y para reactivarlo se tiene que presionar el botón de **reset()** o programar previamente el pin que lo reactivará.

CLASES IMPLEMENTADAS

Al igual que pyb, machine implementa una serie de clases con las que automatizamos tareas, y atomizamos el acceso y el funcionamiento de distintos pines y componentes agregados a la placa.

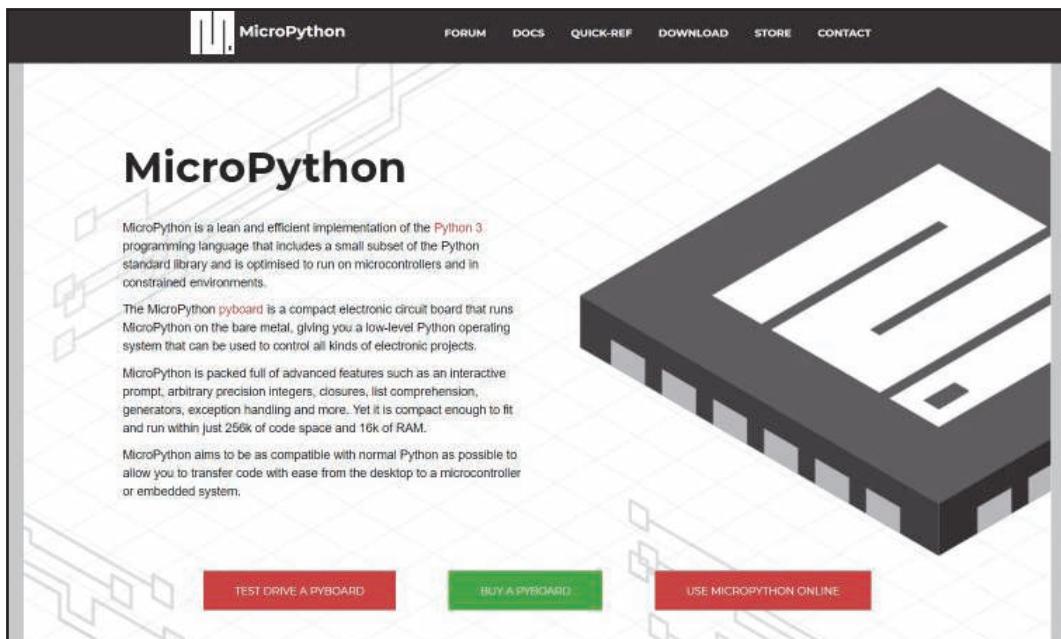
- **Pin()**: devuelve un objeto que nos permitirá interactuar con un pin determinado para leer o setear su valor.
- **WDT()**: controlamos el watchdog de la placa. Con esta clase, podemos evitar un bloqueo y reiniciarla en caso de una falla.

En el sitio oficial de MicroPython (www.micropython.org) es posible encontrar documentación completa sobre el uso de este módulo.

SIMULADOR ONLINE DE PYBOARD

Si entramos en el sitio oficial de MicroPython, www.micropython.org, podemos acceder a un simulador de la placa pyboard, con el cual interactuaremos con distintas librerías sin necesidad de contar con una placa física.

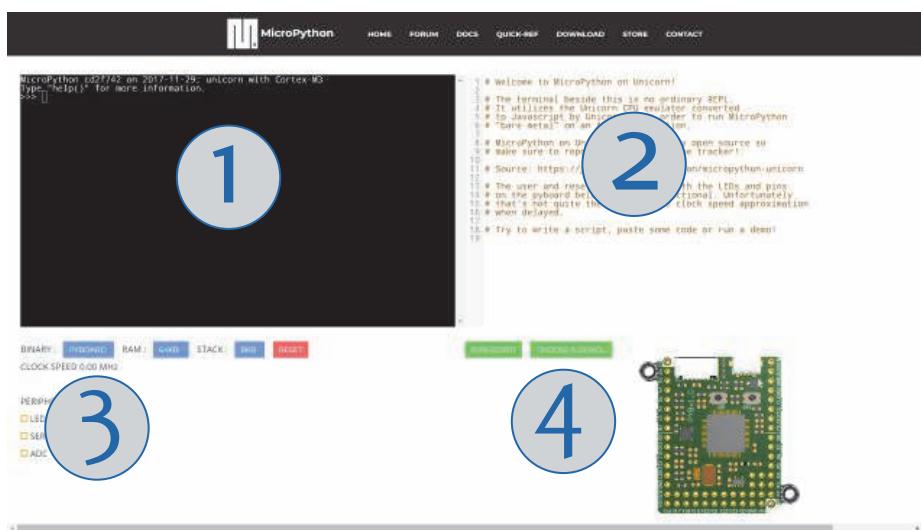
Cabe mencionar que no está completamente implementado MicroPython y todas sus librerías, pero podemos hacer una simulación bastante aproximada para darnos una idea general del funcionamiento de la placa.



The image shows the official MicroPython website homepage. At the top, there's a navigation bar with links for FORUM, DOCS, QUICK-REF, DOWNLOAD, STORE, and CONTACT. The main title "MicroPython" is prominently displayed. Below the title, there's a brief introduction about MicroPython being a lean and efficient implementation of Python 3 for microcontrollers. It also mentions the Pyboard, a compact electronic circuit board, and its compatibility with normal Python. On the right side, there's a large 3D-style illustration of a microcontroller chip. At the bottom, there are three calls-to-action: "TEST DRIVE A PYBOARD" (red button), "BUY A PYBOARD" (green button), and "USE MICROPYTHON ONLINE" (red button).

Sitio oficial de MicroPython.

Para acceder al simulador, deberemos hacer clic sobre el botón que dice **USE MICROPYTHON ONLINE**, esto nos llevará a un simulador online sin necesidad de instalar nada.



The image shows the MicroPython online simulator interface. It features a terminal window displaying a welcome message for MicroPython on Unicorn, along with some sample code. The terminal is highlighted with a large blue circle containing the number 2. Below the terminal, there are tabs for BINARY, PYBOARD (which is selected), RAM, SWD, STACK, and RESET. To the left, there's a sidebar with a circular icon labeled 3 containing "PERIPH" and "ADC". On the right, there's a small image of a green microcontroller board with various components, labeled 4. At the bottom, there are buttons for "RUN" and "RESET".

Simulador online.

Para otorgar mayor realismo y más versatilidad, este simulador posee cuatro sectores con los que podemos interactuar.

1

ESPACIO DE CONSOLA

Este espacio representa una consola de comando donde podremos escribir código como si estuviéramos ejecutando el modo serial. De esta manera, cada sentencia que escribamos la veremos reflejada de manera inmediata.

Por otro lado, si en el código por ejecutar utilizamos la sentencia **print()**, veremos impreso en este sector el resultado, esto nos servirá para hacer un seguimiento de la ejecución del programa.

2

ESPACIO PARA CÓDIGO

Si escribimos el código en un editor externo o queremos simular que estamos creando un archivo para luego actualizar el archivo main.py de la placa, debemos escribir el código en este sector.

Para entrar en modo de ejecución, deberemos presionar sobre el botón que está abajo y que dice **RUN SCRIPT**.

A la par de este botón, veremos otro con el cual podremos cargar algún código de ejemplo de entre los que nos propone el simulador.

3

SELECTOR DE COMPONENTES

En este espacio, tenemos la opción de conectar a la placa 4 componentes distintos:

- Un led: este se conecta al pin **Y12**. Como la clase **Pin** del módulo pyb no está implementada, utilizaremos **machine** para encenderlo y apagarlo.

```
import machine  
y12 = machine.Pin('Y12')  
y12(1)
```

- Un servo que se conecta con los pines **GND**, **VIN** y **X1**.
- Un potenciómetro con el que podremos trabajar con el convertidor de señales analógico y digital (ADC).
- Una pantalla LCD.

Podemos seleccionar uno o varios de estos elementos. Cada vez que tildamos uno, lo vemos reflejado en el simulador de la placa (4).

Por arriba de los componentes, tenemos una serie de opciones para configurar el hardware de nuestra placa llevando la ejecución lo más realista posible a distintas situaciones.

Para que estas configuraciones surtan efecto, cada vez que hagamos una modificación en los parámetros, deberemos presionar el botón que dice **reset**.



SIMULADOR DE PLACA

Este espacio es más que un simple dibujo. En él se ve reflejado cada elemento que seleccionemos del panel 3. También podemos interactuar como si de una placa real se tratase o ver el estado de los led de placa.

A medida que avanza la ejecución, podremos ver cómo cambian los estados de cada componente en este sector. También es posible presionar sobre el botón de **reset**, en el caso de que necesitemos reiniciar la placa, o si un error de código nos ha llevado a una mala ejecución o ciclo infinito.

También podemos presionar sobre el botón de **usuario** o **switch**.

ENCENDAMOS EL LED

A modo de prueba, vamos a realizar un pequeño script que nos permitirá encender las cuatro led de la placa de nuestro simulador.

En primer lugar, importamos las librerías que vamos a utilizar, en este caso emplearemos **time** y **pyb**.

```
import time  
import pyb
```

El paso siguiente es realizar un ciclo de 1000 vueltas que irá cambiando el estado de cada luz en forma alternada definiendo a **i** como iterador.

Recordemos

Para realizar una división cuyo resultado queremos que sea lo más preciso posible, utilizamos el operador **/**. Por el contrario, si la idea es obtener un valor entero, sin importar el resto, utilizaremos **%**. Ej:

8/3 = 2.666

8%3 = 2

Sabiendo que tenemos cuatro luces, vamos a dividir nuestro iterador en cuatro con el operador `%`, lo que nos irá arrojando un valor entero que va a estar entre `0` y `3`.

Con este resultado, invocaremos a la función `toggle` de la clase `LED()`, con lo que iremos encendiendo y apagando alternadamente cada una de las lámparas de la placa.

Para dar un tiempo de espera entre cada suceso, emplearemos la función `sleep_ms` de la clase `time`.

```
for i in range(1000):
    pyb.LED((i%4) + 1).toggle()
    time.sleep_ms(100)
```

Proyectos en MicroPython

En este capítulo, crearemos una serie de proyectos simples escritos en MicroPython. Para ponerlos en práctica sin necesidad de adquirir la placa pyboard, emplearemos el simulador descripto en el capítulo anterior.

Para ello, accederemos a <http://www.micropython.org/unicorn/>, donde nos abrirá el simulador de Python y la placa pyboard.

HOLA MUNDO

En este primer proyecto, emplearemos una pantalla LCD, donde imprimiremos el tradicional **Hola Mundo**.

Para llevar a cabo nuestro proyecto, deberemos conectar una pantalla monocromática en los pines **x9** y **x10**. Si miramos la referencia de pines de la placa pyboard, veremos que x9 corresponde a **SCL** y x10 corresponde a **SDA**. Pero ¿qué significan estas siglas?

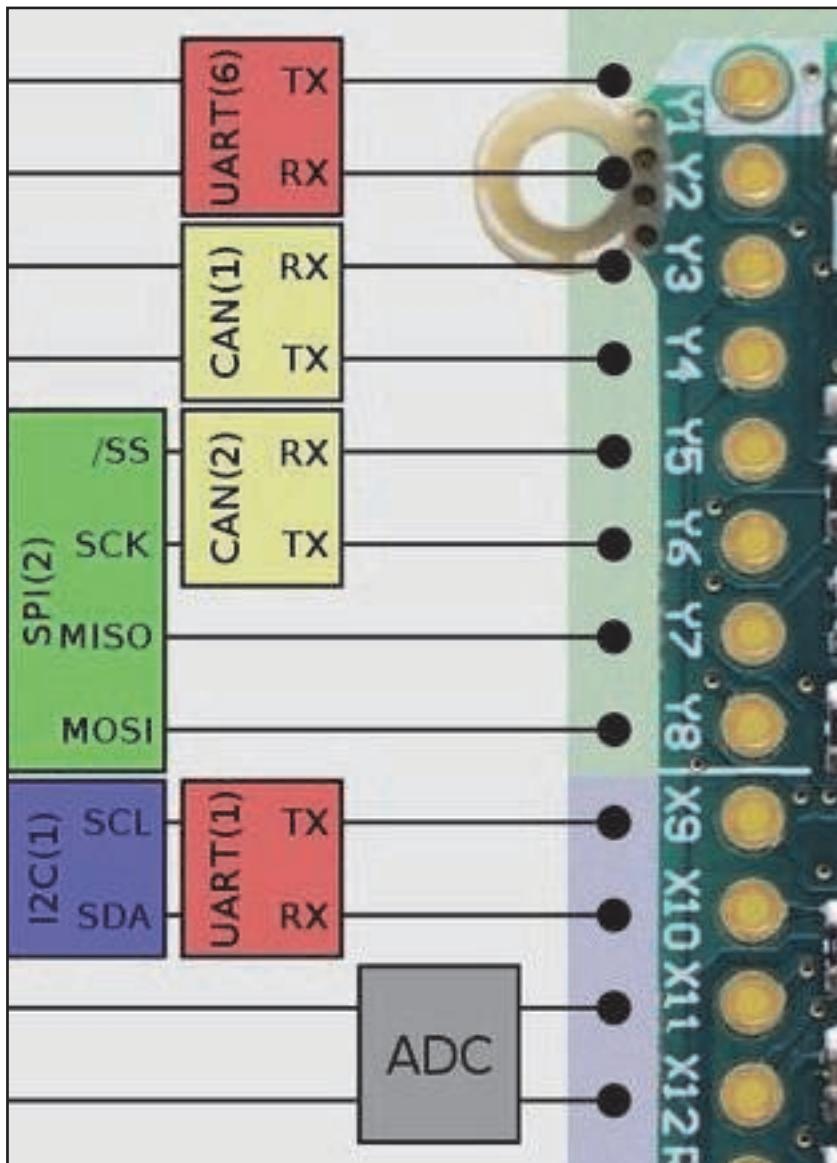
Muchos sensores y periféricos, como la pantalla que queremos conectar, requieren de dos pines en serie para poder funcionar, uno correspondiente a datos (serial data), conocido como **SDA**, y otro correspondiente al clock interno (serial clock), conocido como **SCL**.

MicroPython implementa una clase para poder controlar estos pines, conocida como **I2C**.

Es importante mencionar que este bus permite la conexión de más de un dispositivo, para ello a cada uno que se conecta se le asigna un identificador único con el cual podrá intercambiar datos.

Como esta conexión es serial, la transmisión y recepción de datos queda definida por palabras de 8 bits.

05



LA CLASE I2C

Esta clase es implementada tanto por **pyb** como por **machine**. Si bien el funcionamiento es homólogo en ambos módulos, existen diferencias sutiles en sus implementaciones.

Como el simulador con el cual trabajamos solo implementa el módulo de **machine**, explicaremos algunos de los métodos implementados por esta clase.

DECLARACIÓN

El constructor requiere cuatro parámetros, de los cuales solo dos son requeridos:

```
i2c = machine.I2C(id,scl,sda,frq).
```

Donde:

- **id**: es la identificación única del dispositivo conectado, por defecto es **-1**.
- **scl**: (requerido) corresponde al pin del clock.
- **sda**: (requerido) corresponde al pin de datos.
- **frq**: con este parámetro configuramos la frecuencia mínima del clock. Dicha frecuencia debe ser compatible con el dispositivo que conectamos.

INIT(SCL,SDA,FREQ)

Este método inicializa el bus con los parámetros especificados. Es importante que respetemos los pines scl y sda de la placa en el momento de llamar a este método, en caso contrario nos arrojará un error.

DEINIT()

Finaliza el bus I2C.

MÉTODOS PRIMITIVOS

Estos métodos forman parte del diseño primitivo de la clase y permiten implementar la lectura/escritura de un dispositivo.

- **start()**: inicializa una transacción, pone el pin de datos en bajo, y el del clock, en alto.
- **stop()**: finaliza una transacción, pone el pin de datos en alto mientras el de clock lo sostiene en alto.
- **readinto(buffer, nack=True)**: lee un paquete de bytes del dispositivo y los almacena en la variable **buffer**. Si **nack** es **true**, se indica que ya no

se volverá a leer, por lo que vacía el buffer de lectura. En caso contrario, queda esperando a una siguiente lectura.

- **write(buffer)**: escribe en el dispositivo los byte enviados por **buffer**.

OPERACIONES ESTÁNDARES

Con estos métodos, se pueden llevar a cabo operaciones de lectura y escritura de un dispositivo determinado.

- **readfrom(addr, nbyte, stop)**: lee n bytes del dispositivo cuya id es **addr** y los devuelve para asignarlos a una variable. Si **stop** es **verdadero**, se genera la condición de parada al final de la transferencia.
- **readfrom_into(addr, buffer, stop)**: lee todo el contenido del dispositivo cuya id es **addr** y lo devuelve dentro de la variable **buffer**. Si **stop** es **verdadero**, se genera la condición de parada al final de la transferencia.
- **writetoto(addr, buffer, stop)**: escribe los bytes del buffer pasado como parámetro.
- **writetovto(addr, vector, stop)**: escribe los bytes contenidos en el vector que es pasado como parámetro al dispositivo conectado cuyo id es **addr**.

LA CLASE FRAMEBUF

En nuestro proyecto, también utilizaremos este módulo para crear un buffer donde cargaremos una imagen o un texto y lo pasaremos a nuestra pantalla.

Esta clase nos permite trabajar a nivel pixeles una imagen, generar una línea, un rectángulo o un texto y traducirlo a un buffer de byte.

CONSTRUCTOR

El constructor requiere cinco parámetros:

- **buffer**: un espacio de memoria donde almacenaremos todos los bytes.
- **width**: el ancho expresado en pixeles.
- **height**: el alto expresado en pixeles.
- **format**: corresponde al tipo de los pixeles que se van a almacenar. (**MONO_HLSB**: monocromático; **RGB565**: color RGB de 16 bits; **GS2_HMSB**: escala de grises de 2 bits; **GS4_HMSB**: escala de grises de 4 bits; **GS4_HMSB**: escala de grises de 8 bits).
- **stride**: representa la cantidad de pixeles entre dos líneas horizontales.

MÉTODOS

- **fill(c)**: pinta todo el buffer del color pasado como parámetro.
- **pixel(x,y,c)**: si el color no es pasado, devuelve el color de un determinado pixel. Por el contrario, si se pasa como parámetro el color, se setea dicho color en el pixel determinado por las coordenadas **x** e **y**.
- **hline(x,y,w,c); vline(x,y,h,c); line(x1,y1,x2,y2,c)**: con estas funciones podemos dibujar una línea; la primera la hacemos horizontal; la segunda, vertical; y la tercera, entre dos puntos pasados por las coordenadas.
- **rect(x,y,w,h,c); fill_rect(x,y,w,h,c)**: con estas funciones podemos dibujar un rectángulo con inicio **x**, **y**, de ancho **w** y alto **h**. La diferencia entre una y otra es que la primera solamente dibuja el contorno (un solo pixel como borde) mientras que la segunda realiza un recuadro lleno.
- **text(s,x,y,c)**: escribe un texto (**s**) en las coordenadas indicadas (**x** e **y**), del color **c**. Hasta el momento, el texto se realiza en un tamaño de 8 x 8 sin posibilidad de hacer un cambio en la fuente.

HAGAMOS NUESTRA APLICACIÓN

Para comenzar a llevar a cabo nuestra aplicación, lo primero que haremos es importar las librerías que utilizaremos, estas son: **machine**, **I2C**, y **framebuf**.

```
import machine
import framebuf
from machine import I2C
```

Como la declaración del objeto **I2C** requiere los pines de dato y clock, utilizaremos **Pin** de **machine** para referenciarlos. Como vimos, la placa pyboard reserva al pin X9 como SCL y al pin X10 como SDA.

```
scl = machine.Pin("X9")
sda = machine.Pin("X10")
i2c = I2C(scl=scl, sda=sda)
```

El paso siguiente es crear nuestro buffer, para ello lo primero que tenemos que ver es el tamaño del display y el tipo de pantalla. En nuestro caso, que estamos

utilizando el emulador, tenemos una pantalla monocromática cuya dimensión es de 64 columnas por 32 filas.

```
fbuf = framebuf.FrameBuffer(bytearray(64 * 32 // 8), 64, 32,  
framebuf.MONO_HLSB)
```

Para saber cuál es el tamaño de nuestro buffer, creamos un array de byte de 64 columnas y 32 filas, dividido en grupos de ocho que conforman nuestra palabra, como mencionamos al principio.

Ya tenemos todo definido, solo nos queda empezar a dibujar. El primer paso es pintar todo de negro. De esta forma nos aseguramos que el buffer no incluya basura en su contenido.

```
fbuf.fill(0)
```

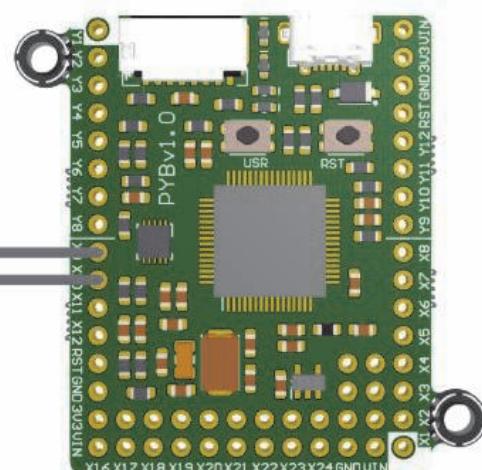
Luego, agregamos el texto **Hola** en la columna 15, fila 8.

Como sabemos que el alto es de 8 pixeles, escribiremos **Mundo** en la columna 15 fila 18

```
fbuf.text('Hola', 15, 8)  
fbuf.text('Mundo', 15, 18)
```

Por último, enviaremos la información a nuestro display.

```
i2c.writeto(8,fbuf)
```



Resultado de nuestro código.

Pero para darle un toque más artístico a nuestro proyecto, vamos a agregarle como detalles un punto en cada extremo del display y le insertaremos un recuadro a la frase **Hola Mundo**.

Para insertar un punto, pintaremos un pixel en la posición que deseemos. En este caso, pondremos uno en las coordenadas correspondientes al extremo izquierdo superior y otro en el derecho inferior. Esto es:

```
fbuf.pixel(0,0,1)  
fbuf.pixel(63,31,1)
```

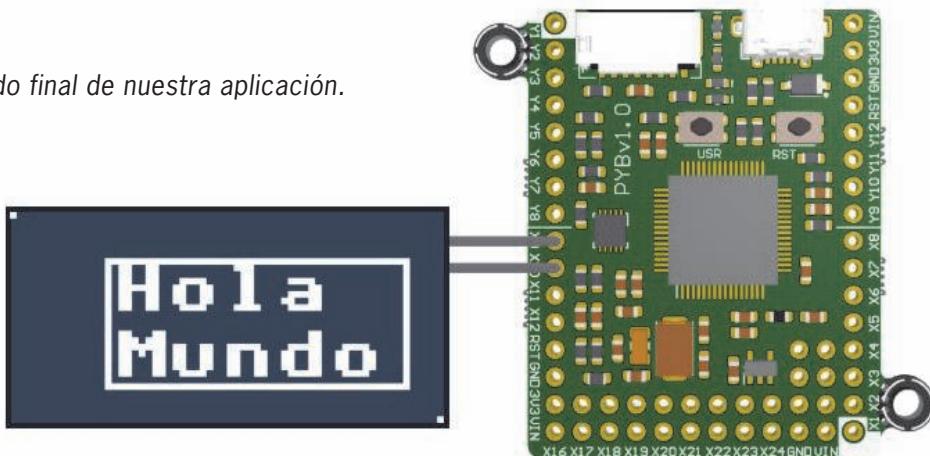
y para el recuadro, utilizaremos la función **rect**.

```
fbuf.rect(14,7,45,20,1)
```

De esta forma, nuestro código finalmente nos queda de la siguiente manera:

```
import machine  
import framebuf  
from machine import I2C  
  
scl = machine.Pin('X9')  
sda = machine.Pin('X10')  
i2c = I2C(scl=scl, sda=sda)  
  
fbuf = framebuf.FrameBuffer(bytearray(64 * 32 // 8), 64, 32,  
framebuf.MONO_HLSB)  
  
fbuf.fill(0)  
fbuf.pixel(0,0,1)  
fbuf.pixel(63,31,1)  
fbuf.rect(14,7,45,20,1)  
fbuf.text('Hola', 15, 8)  
fbuf.text('Mundo', 15, 18)  
  
i2c.writeto(8,fbuf)
```

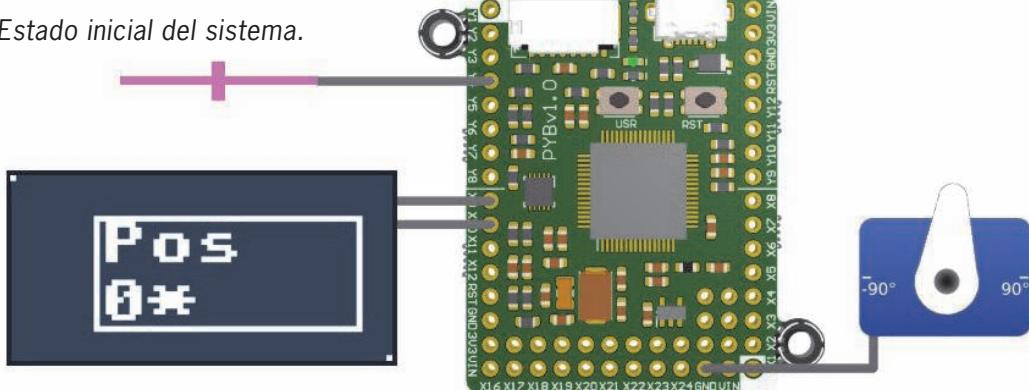
Estado final de nuestra aplicación.



CONTROL DE SERVO

En este proyecto, armaremos un sistema en donde podremos mover un potenciómetro para cambiar la posición de un servo mecánico. Para saber cuál es la inclinación que ha adquirido el servo, mostraremos en un display el ángulo de giro.

Estado inicial del sistema.



Si llevamos a la práctica este proyecto con componentes reales, deberemos instalar dichos componentes de la siguiente manera:

- **Potenciómetro:** en el pin Y4.
- **Display:** SCL en el pin X9 y SDA en X10.
- **Servo:** en los pines X1, GND y VIN, tomando la precaución de que estos dos últimos pertenezcan a la fila de X1.

LAS LIBRERÍAS POR IMPORTAR SON LAS SIGUIENTES

- **machine**: para obtener los pines correspondientes al potenciómetro y al display.
- **I2C**: para trabajar con el display.
- **pyb**: para interactuar con el servo y llevar a cabo la conversión de señal análoga a digital del potenciómetro, y poder interactuar con las led de estados.
- **time**: para realizar una espera antes de volver a chequear el estado del potenciómetro.
- **framebuf**: para cargar la información por mostrar en el display.

Una vez que está todo preparado, definimos las variables con las cuales interactuaremos con el servo, con el potenciómetro y con el display.

SERVO

Como conectamos el servo en la línea correspondiente al pin X1, nuestra variable la definiremos de la siguiente manera:

```
servo = pyb.Servo(1)
```

Si la conexión fuese en la línea del pin X2, usaríamos **Servo(2)**, y si fuera en la línea del pin X3, **Servo(3)**.

POTENCIÓMETRO

Nuestro potenciómetro está en el pin Y4, por tal razón, crearemos primero la variable que represente a dicho pin, y luego la variable que nos permitirá interactuar con el potenciómetro propiamente dicho.

```
y4 = machine.Pin('Y4')
adc = pyb.ADC(y4)
```

DISPLAY

Nuestro display está conectado en los pines X9 y X10, por lo que crearemos las variables para representar ambos pines y la variable correspondiente al display.

```
scl = machine.Pin('X9')
sda = machine.Pin('X10')
i2c = I2C(scl=scl, sda=sda)
```

Por último, definiremos el buffer con el cual enviaremos datos al display. Suponiendo que este fuera monocromático, de 64 x 32 pixeles, nuestra definición sería de la siguiente manera:

```
fbuf = framebuf.FrameBuffer(bytearray(64 * 32 // 8), 64, 32,  
framebuf.MONO_HLSB)
```

FUNCIÓN PARA MOSTRAR

La intención es que, cada vez que movamos el potenciómetro, mostremos en el display el ángulo de giro de nuestro servo.

Para mantener limpio el código y lograr que sea más fácil el mantenimiento, definiremos una función a la cual solamente le pasaremos un texto y se encargará del resto.

Para simplificar aún más las cosas, reutilizaremos el código del proyecto anterior (**Hola Mundo**) y reemplazaremos solo las dos líneas donde escribimos el texto propiamente dicho.

Entonces, nuestra función **mostrar** tendrá el siguiente código:

```
def mostrar(texto):  
    fbuf.fill(0)  
    fbuf.pixel(0,0,1)  
    fbuf.pixel(63,31,1)  
    fbuf.rect(14,7,45,20,1)  
    fbuf.text('Pos', 15, 8)  
    fbuf.text(texto + '*', 15, 18)  
    i2c.writeto(8,fbuf)
```

CÓDIGO PRINCIPAL

Nuestro programa estará inserto en un ciclo infinito, pues la placa deberá estar continuamente esperando a que movamos el potenciómetro.

Dentro de este ciclo, lo primero que haremos es leer su estado y determinar el factor por aplicarle al servo.

Mediante la función **adc.read()**, obtendremos un valor que va desde **0** hasta **255**. Pero nosotros tenemos que convertir esos valores en ángulos de -90° hasta 90°.

Para simplificar la tarea, vamos a suponer que 127 es la mitad, es decir, si la función me devuelve **127**, voy a considerar que el ángulo por aplicar es de 0°.

Entonces si escribo

```
angulo = adc.read() - 127
```

voy a obtener los siguientes resultados:

A partir de esta tabla, podemos armar una regla de tres diciendo que, si el **ángulo leído - 127 >= 127**, estamos en el 100%. Es decir, cuando estamos en el máximo, estamos al 100%, pero nuestro mínimo no es 0, sino que es 127 (la mitad).

Y todo valor menor a eso debería ser negativo hasta llegar al cero, que nos representaría -100%. Así que

```
pangulo = (lectura - 127) * 100 / 127
```

Como pangulo va a variar entre -100% y +100%, y el ángulo de nuestro servo varía entre -90° y 90°, lo que nos queda es determinar qué valor del ángulo le corresponde el porcentaje obtenido.

Para ello, cargaremos en una nueva variable llamada **angulo** el porcentaje que corresponde a los 90°.

Por lo que nuestro código sería el siguiente:

```
pangulo = int(((adc.read() - 127) * 100 / 127))
angulo = int(pangulo * 90 /100)
servo.angle(angulo , 1000)
```

servo.angle()

Esta función recibe dos parámetros:

El primero es el ángulo por aplicar expresado en un valor entero que puede oscilar entre -90° y +90°.

El segundo es el tiempo en milisegundos que queremos que tarde el servo en llegar al ángulo indicado.

adc.read	pangulo
0	-127
< a 127	< 0
127	0
> 127	> 0
255	128

Ahora, lo que haremos es que se encienda el led verde, si el servo apunta a la izquierda (ángulo menor a 0°), y el led rojo, si apunta a la derecha (ángulo mayor a 0°).

Para lograr esto, chequearemos el valor de **pangulo** y, si es mayor a 0, encenderemos el led 1 y apagaremos el led 2. En caso contrario, apagaremos el 1 y encenderemos el 2.

```
if pangulo > 0:  
    pyb.LED(1).on()  
    pyb.LED(2).off()  
else:  
    pyb.LED(2).on()  
    pyb.LED(1).off()
```

Luego llamaremos a la función que hemos creado para mostrar en el display el estado. Como parámetro, le pasaremos la variable **angulo** convertida en string.

```
mostrar(str(angulo))
```

Como último paso nos queda hacer una espera de medio segundo para evitar que el código se apodere del 100% del procesador.

```
time.sleep(0.5)
```

Con esto, ya hemos finalizado nuestro código, que debería ser el siguiente:

```
import machine  
from machine import I2C  
import pyb  
import time  
import framebuf  
  
servo = pyb.Servo(1)  
y4 = machine.Pin('Y4')  
adc = pyb.ADC(y4)  
  
scl = machine.Pin('X9')  
sda = machine.Pin('X10')  
i2c = I2C(scl=scl, sda=sda)
```

```

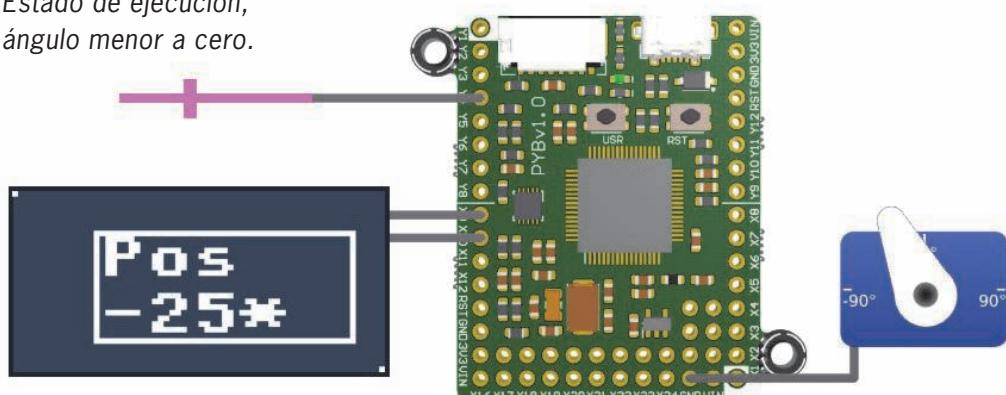
fbuf = framebuf.FrameBuffer(bytearray(64 * 32 // 8), 64, 32,
framebuf.MONO_HLSB)

def mostrar(texto):
    fbuf.fill(0)
    fbuf.pixel(0,0,1)
    fbuf.pixel(63,31,1)
    fbuf.rect(14,7,45,20,1)
    fbuf.text('Pos', 15, 8)
    fbuf.text(texto + '*', 15, 18)
    i2c.writeto(8,fbuf)

while True:
    pangulo = int(((adc.read() - 127) * 100 / 127))
    angulo = int(pangulo * 90 /100)
    servo.angle(angulo , 1000)
    if pangulo > 0:
        pyb.LED(1).on()
        pyb.LED(2).off()
    else:
        pyb.LED(2).on()
        pyb.LED(1).off()
    mostrar(str(angulo))
    time.sleep(0.5)

```

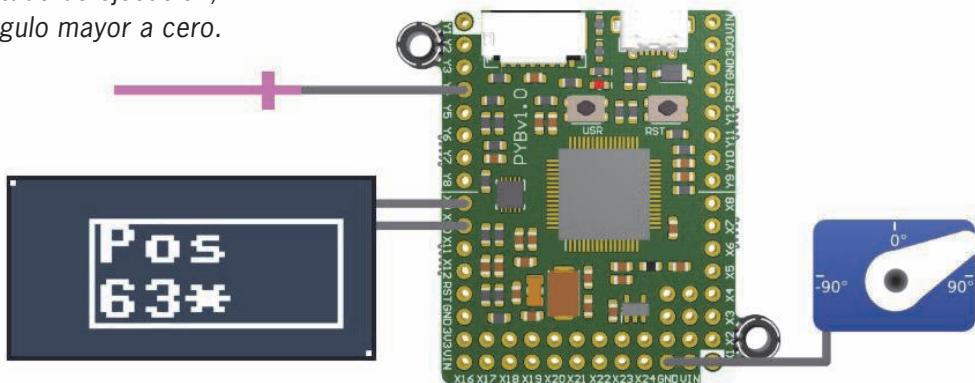
Estado de ejecución,
ángulo menor a cero.



Hola soy Yako1976, si estas leyendo esto y no has descargado de www.identi.io, quiere decir que lo han copiado. Yo solo posteo en www.identi.io, buscarme, descargas directas sin acortadores. Softwares, cursos y más. Que tengáis buena lectura.

Yako1976

Estado de ejecución,
ángulo mayor a cero.



PYTHON Y IOT

Hemos visto cómo un lenguaje de programación de código abierto, libre y gratuito está ganando terreno en el mundo de los programadores. Y no es casualidad que se esté convirtiendo en el preferido de muchos. Solo debemos considerar que posee una sentencia simple, una velocidad de ejecución muy eficiente y que es posible adaptarlo a muchas plataformas. Pero como si esto fuera poco, podemos encontrarnos por internet con mucha documentación gratuita que no por eso es de mala calidad, por el contrario, es creada por aficionados que disfrutan idear y compartir sus logros.

También existen muchas librerías accesibles para cualquiera, que nos permiten trabajar con un sinfín de placas o sensores manteniendo la premisa de simplicidad en la programación.

Con todo esto, hoy en día disputa el primer lugar con lenguajes como C, C++, Java, entre otros, y se convierte cada vez más en el favorito para trabajar en pequeños componentes para automatizar procesos, conectar dispositivos, ayudar a las tareas cotidianas. En otras palabras, junto con un grupo extenso de placas y sensores de hardware libre de fácil acceso y a muy bajo costo, Python da un paso más en la programación del internet de las cosas.

Django: Python en la Web

La popularidad de **Python** ha llevado a que la comunidad de programadores busque la forma de emplearlo para realizar sistemas web. Django es el resultado de uno de estos intentos y no solo nos permite crear un servidor web, sino que nos brinda la plataforma de desarrollo para montar un sistema complejo.

QUÉ ES DJANGO

Django es un **framework** web de alto nivel, diseñado por programadores expertos en Python, quienes buscaron simplificar, de alguna manera, el desarrollo web.

La ventaja fundamental de esta plataforma es que permite que los programadores se enfoquen netamente en el diseño del sistema, dejando la tarea más pesada, el diseño web, al framework.

Creada bajo estrictas normas de seguridad, se convierte en un medio seguro para procesar datos sensibles. Su código está escrito en Python bajo una licencia **open source**, por lo que es gratis para cualquier persona que desee utilizarlo o modificarlo.

SUS ORÍGENES

Surge en el 2003, de la mano de **Adrian Holovaty** y **Simon Willison**, dos programadores que trabajaban en un diario de Kansas. Ellos creaban aplicaciones con Python, y las exigencias del medio los obligaban a desarrollarlas en tiempos límites. Cuando notaron que todas sus aplicaciones tenían mucho en común, comenzaron un sistema que les permitiera simplificar el desarrollo y el mantenimiento. Sin saberlo estaban creando un framework robusto y ágil.

En el año 2005, es liberado su código bajo licencia open source y bautizado como **Django**, en honor al guitarrista de jazz **Django Reinhardt**.

El hecho de que Django naciera de una necesidad laboral lo hace diferente a otros frameworks que surgen en un entorno artificial, teórico o académico, porque su

Apéndice

desarrollo implementa soluciones a problemas reales, cotidianos y actuales. Además, su comunidad está continuamente actualizándolo para que mejore día a día su operatividad.

INSTALACIÓN EN WINDOWS O LINUX

Como ya se ha expresado, Django esta codificado en Python para funcionar exclusivamente bajo este lenguaje y nace de la necesidad de simplificar las tareas. Si partimos de esas premisas, entenderemos que el único requisito que necesitaremos para llevar a cabo un desarrollo web bajo este framework es el conocimiento de Python. De hecho, no necesitamos ser expertos, solo nos bastará conocer cómo importar una librería, cómo se estructura el lenguaje, y el manejo de sentencias básicas como el **if** y el **while**.

Para instalarlo, lo recomendado es que utilicemos la herramienta **PIP** del lenguaje Python.

Ya sea que trabajemos bajo Windows o bajo Linux, la primera acción es determinar si Python está instalado. Para ello ingresaremos a una terminal y escribiremos **python**.

Si después de este comando aparece un mensaje de **command not found**, deberemos instalar el lenguaje antes de continuar. Por el contrario, si aparece un mensaje que nos indique un número de versión, sabremos que ya está instalado.



```
Windows PowerShell
PS C:\> python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Respuesta en consola de comando – Python instalado

Una vez que hemos comprobado que Python está en el sistema, podemos proceder a su instalación; esto puede hacerse o bien descargando los archivos e instalándolos en forma manual, o directamente utilizando la herramienta PIP con el modificador **install**.

```
python -m pip install Django
```

EN LINUX

Python suele venir preinstalado en las distros de Linux, por lo que solo deberemos abrir una terminal y ejecutar el código de instalación.

EN WINDOWS

Si estamos operando bajo el sistema operativo Windows 10, no debemos ejecutar el comando en la ventana de **CMD**, sino que deberemos abrir la consola de comandos PowerShell en modo de administrador para poder llevar a cabo la operación.

Una vez que hemos ejecutado el comando anterior correctamente, ya estamos en condiciones de trabajar con este framework de manera local sin necesidad de una conexión a internet y sin tener que llevar a cabo alguna otra configuración.

ENTORNOS VIRTUALES

Si bien con lo expresado en los apartados anteriores ya podemos crear un proyecto, los propios diseñadores de Django recomiendan crear entornos virtuales para cada proyecto. De esta manera podemos tener varias sentencias de Django operando individualmente sin riesgo de que una se mezcle con otra.

Si no queremos llevar a cabo este paso, podemos diseñar nuestro proyecto y trabajar en un entorno global. Algo que muy probablemente, según indicaciones, nos traerá algún inconveniente cuando llevemos a la práctica más de un proyecto.

Para instalar el entorno virtual:

- Si estamos en Linux, desde la propia terminal ejecutamos:

```
sudo pip3 install virtualenvwrapper
```

Luego, en el archivo oculto **.bashrc -Configuracion de inicio del Shell**- ingresaremos el siguiente texto:

```
export WORKON_HOME=$HOME/.virtualenvs  
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3  
export PROJECT_HOME=$HOME/Devel  
source /usr/local/bin/virtualenvwrapper.sh
```

Con estas líneas, configuraremos dónde vivirá el entorno, dónde se almacenarán los proyectos, y la localización de los scripts instalados con el paquete.

Lo último que haremos es recargar el fichero de inicio y para esto escribiremos en la consola:

```
source ~/.bashrc
```

- Si estamos en Windows 10, el proceso es más simple, lo único que deberemos hacer es abrir una consola PowerShell y ejecutar lo siguiente:

```
pip3 install virtualenvwrapper-win
```

A diferencia de la instalación en Linux, ya no es necesario configurar nada porque la propia instalación define las rutas.

USOS DE LOS ENTORNOS VIRTUALES

Una vez que ya tengamos nuestro entorno virtual instalado, podremos trabajar con ellos. Para esto debemos conocer y hacer uso de los siguientes comandos:

- **mkvirtualenv nombre**: crea un nuevo entorno virtual. Cada vez que necesitamos trabajar con un proyecto, ejecutamos este comando que recibe como parámetro un nombre, el del entorno que estamos creando.
- **deactivate**: con este comando, saldremos del entorno virtual actual.
- **workon**: este comando nos lista todos los entornos virtuales que fueron creados y que están disponibles en este momento.
- **workon nombre**: si tenemos un entorno creado, pero no está en uso, con este comando activamos el entorno virtual especificado.
- **rmvirtualenv nombre**: este comando se emplea para eliminar por completo un entorno virtual.

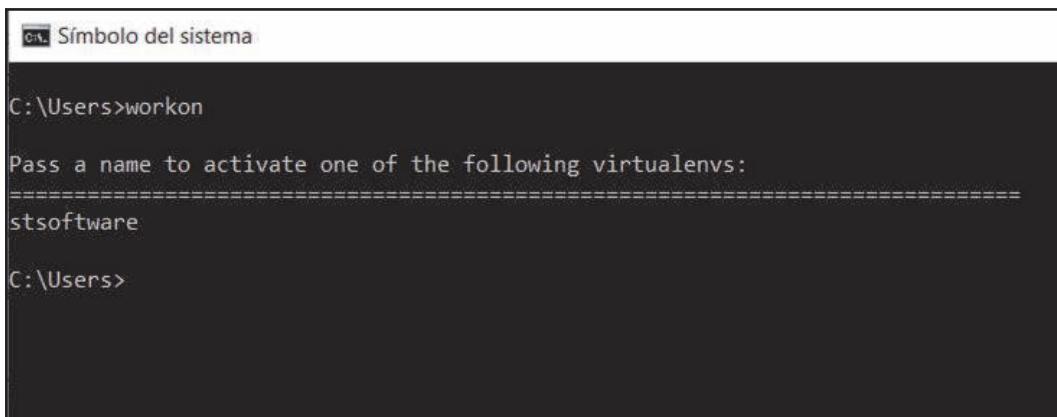
Asimismo, es importante aclarar que existen más comandos disponibles para trabajar con los entornos virtuales, sin embargo, estos representan los de mayor utilidad.

A partir de ahora, deberemos instalar Django en cada entorno virtual que creemos. Gracias a esto podemos utilizar nuestra máquina como un servidor en donde se estén ejecutando distintos proyectos en simultáneo de manera que no se interfieran entre ellos.

CREAR UN PROYECTO

Como primer paso, deberemos crear un entorno virtual para encapsular todo el paquete; para ello respetemos los siguientes pasos.

- **Abrimos CMD:** no es necesario hacerlo en modo administrador.
- **Chequeamos EV:** tipeamos `workon` para ver los entornos virtuales que hemos creado.



```
C:\> Símbolo del sistema  
C:\Users>workon  
Pass a name to activate one of the following virtualenvs:  
=====  
stsoftware  
C:\Users>
```

Listado de los entornos virtuales creados

- **Creamos EV:** si no está el que vamos a utilizar, lo creamos con `mkvirtualenv`.

Atención

Si estamos trabajando en Windows, debemos utilizar CMD y no PowerShell para movernos entre los distintos entornos.

```
C:\ Símbolo del sistema

C:\Users>workon

Pass a name to activate one of the following virtualenvs:
=====
stsoftware

C:\Users>mkvirtualenv redusers
Using base prefix 'c:\\users\\lito\\appdata\\local\\programs\\python\\python38-32'
New python executable in C:\\Users\\Lito\\Env\\redusers\\Scripts\\python.exe
Installing setuptools, pip, wheel...
done.

(redusers) C:\\Users>
```

Entorno virtual redusers creado

Luego de este proceso, se crea una carpeta donde se incluirán todos los archivos necesarios para ejecutar una sentencia de Python aislada.

Dicha carpeta tendrá la siguiente estructura:

```
--Envs
|-- redusers
    |--Include
    |--Lib
    |--Scripts
    |--tcl
```

Como esta nueva instancia es creada limpia, es decir, sin ningún paquete, deberemos instalar todas aquellas librerías que vayamos a emplear en nuestro proyecto. En este caso, vamos a utilizar Django, por lo que deberemos instalarla.

Recordar que

Si vamos a trabajar con otro entorno virtual, deberemos instalar Django para cada entorno.

Tipeamos **pip list** para listar las librerías instaladas.

```
C:\Users>workon
Pass a name to activate one of the following virtualenvs:
=====
stsoftware

C:\Users>mkvirtualenv redusers
Using base prefix 'c:\\users\\lito\\appdata\\local\\programs\\python\\python38-32'
New python executable in C:\Users\Lito\Envs\redusers\Scripts\python.exe
Installing setuptools, pip, wheel...
done.

(redusers) C:\Users>pip list
Package      Version
-----
pip          19.3.1
setuptools   42.0.2
wheel        0.33.6

(redusers) C:\Users>
```

Librerías instaladas en nuestro EV

Como vemos que no está Django, la instalamos con **pip install Django**.

```
C:\Users>mkvirtualenv redusers
Using base prefix 'c:\\users\\lito\\appdata\\local\\programs\\python\\python38-32'
New python executable in C:\Users\Lito\Envs\redusers\Scripts\python.exe
Installing setuptools, pip, wheel...
done.

(redusers) C:\Users>pip list
Package      Version
-----
pip          19.3.1
setuptools   42.0.2
wheel        0.33.6

(redusers) C:\Users>pip install Django
Collecting Django
  Using cached https://files.pythonhosted.org/packages/6a/23/08f7fd7afdd24184a400fcabef921bd09b5b5235cbd62ffa02308a7d35d6/Django-3.0.1-py3-none-any.whl
Collecting asgiref<=3.2
  Using cached https://files.pythonhosted.org/packages/a5/cb/5a235b605a9753ebcb2730c75e610fb51c8cab3f01230080a8229fa36ad/b/asgiref-3.2.3-py2.py3-none-any.whl
Collecting sqlparse>=0.2.2
  Using cached https://files.pythonhosted.org/packages/ef/53/900f7d2a54557cba37886585a91336520e5539e3ae2423ff1102daf4f3a7/sqlparse-0.3.0-py3-none-any.whl
Collecting pytz
  Using cached https://files.pythonhosted.org/packages/e7/f9/f0b53f88060247251bf481fa6ea62cd0d25bf1b11a87888e53ce5b7c8ad2/pytz-2019.3-py3-none-any.whl
Installing collected packages: asgiref, sqlparse, pytz, Django
Successfully installed Django-3.0.1 asgiref-3.2.3 pytz-2019.3 sqlparse-0.3.0

(redusers) C:\Users>
```

Instalación finalizada de Django

Una vez instalado, volvemos a listar las librerías y chequeamos si ya figura en el listado.

cmd Símbolo del sistema

```
(redusers) C:\Users>pip list
Package    Version
-----
asgiref    3.2.3
Django     3.0.1
pip         19.3.1
pytz        2019.3
setuptools  42.0.2
sqlparse    0.3.0
wheel       0.33.6
```

```
(redusers) C:\Users>
```

Listado actualizado de librerías instaladas

- **Creamos el proyecto:** en el caso de que estemos trabajando en Windows, deberemos acceder a la carpeta de nuestro EV, en

C:\Users\%user-name\Envs.

Por lo que tipearemos en la consola:

```
cd C:\Users\%user-name\Envs
```

Luego ejecutaremos el siguiente comando:

```
django-admin.py startproject holausers
```

donde **holausers** va a ser el nombre de nuestro proyecto.

Después de ejecutar este comando, Django nos crea la siguiente estructura de carpetas donde será alojado.

```
|--holausers
  |--manage.py
  |--holausers
    |--__init__.py
    |--asgi.py
    |--settings.py
    |--urls.py
    |--wsgi.py
```

ESTRUCTURA

Para comprender mejor lo que hizo Django, veamos detalladamente la estructura generada.

Primero nos vamos a encontrar con una carpeta contenedora denominada igual que nuestro proyecto y, dentro de esta, otra carpeta con igual nombre, donde se alojarán todos los archivos de nuestro paquete y el archivo **manage.py**.

MANAGE.PY

Es un programa escrito en Python, que nos permitirá interactuar con el proyecto, con el cual podremos crear aplicaciones, lanzar el servicio web o administrar usuarios, entre otras tareas.

CARPETA DE PAQUETE HOLAUSERS

Esta carpeta contendrá todos los archivos del proyecto. En ella nos encontraremos con:

- **__init__.py**: es un archivo que oficia de bandera. En su interior no hay nada, pero le indica a Python que todo el directorio debe considerarse como un paquete.
- **settings.py**: en este archivo se almacena toda la configuración de nuestro paquete como ser: idioma, aplicaciones instaladas, base de datos.
- **urls**: un listado de todas las URL a las cuales podrá acceder el proyecto. Se asemeja al armado de una tabla de contenidos para validar cada acceso.
- **wsgi.py**: archivo que nos permite configurar el acceso web a través de un gateway.

AGREGAR UNA APLICACIÓN A UN PROYECTO

Una vez que tenemos nuestro proyecto listo, el paso siguiente es crearnos una aplicación donde codificaremos.

Cabe aclarar que cada proyecto puede contener más de una aplicación que podrán compartir librerías o datos.

Para poder crear una, desde nuestra consola de comandos deberemos acceder al directorio de nuestro proyecto, donde está el archivo **manage.py**. En nuestro ejemplo, deberemos acceder a **holausers**.

Es importante resaltar que el acceso se realiza a la carpeta contenedora de nuestro proyecto y no a la global.

Una vez que estemos en la carpeta, escribiremos lo siguiente:

```
python manage.py startapp holaapp
```

Con esto, Django nos crea en este directorio una carpeta llamada **holaapp** con el siguiente contenido:

```
|--holausers
  |--holaapp
    |--__init__.py
    |--admin.py
    |--apps.py
    |--models.py
    |--tests.py
    |--views.py
    |--migrations
      |--__init__.py
```

Veamos su contenido:

- **__init__.py**: al igual que antes, es un archivo bandera para decirle a Python que todo el contenido es un paquete.
- **admin.py**: en este archivo, definiremos todos los controladores para el panel de administración generado por Django.
- **apps.py**: definimos estado inicial de nuestra aplicación entre otras cosas relativas a esta.
- **models.py**: en este archivo, vamos a definir cada una de las tablas que va a utilizar nuestra aplicación. Cabe aclarar que cada tabla es representada por una clase dentro de este archivo.
- **test.py**: se definen estados para el testeo de la aplicación.
- **urls.py**: definiremos en este archivo todas las URL que van a ser empleadas dentro de la aplicación.
- **views.py**: definiremos todas las clases que representarán una vista de nuestra aplicación. Cuando agregamos una URL, se suele llamar a una clase definida en este archivo.

- **Fichero migrations:** en esta carpeta, se irán creando automáticamente los archivos relacionados con nuestra base de datos a medida que vayamos creando clases en el archivo **models.py**. Cabe resaltar que, al ser un directorio de generación automática, no es recomendable editar su contenido, para evitar inconsistencias.

LÓGICA DEL FUNCIONAMIENTO

Antes de empezar a codificar, es importante que entendamos cómo funciona Django. Recordemos que este fue creado a partir de la práctica, por lo que puede aparecer complejo al principio, pero una vez que comprendamos correctamente su lógica, veremos que se pueden obtener grandes proyectos con muy poca codificación dejando todo el trabajo pesado a este framework.

Django es un framework **MTV**, esto quiere decir:

- **Modelo (M):** toda la codificación se lleva a cabo mediante clases. A partir de estas clases, realizaremos el almacenamiento de datos, llamado de vistas y cualquier otra tarea que necesitemos para renderizar nuestra aplicación.
- **Template (T):** para simplificar la codificación, se generan plantillas con etiquetas propias de Django. La codificación de estas plantillas puede hacerse como si estuviéramos diseñando una página web tradicional, donde podremos emplear HTML, CSS, JS o cualquier otro lenguaje de diseño web.
- **Vistas (V):** dentro del archivo **views.py**, se definen como función cada una de las vistas que luego serán llamadas para realizar las consultas y la muestra de datos. En esta función se ejecutan todas las tareas previas y las consultas a la base de datos utilizando Python directamente. Es importante resaltar que, en la vista, se preparan los datos y se realiza la llamada al template correspondiente para que este los muestre.

Cuando realizamos un acceso a una página diseñada con Python, Django lee la URL y busca cuál es la vista apropiada para esa petición.

Esto lo hace desde el archivo **urls.py**, donde se hace referencia al archivo **views.py** para chequear qué acción tomar a la petición del usuario.

Desde el archivo **views.py**, podremos renderizar directamente la página que queremos mostrar o, para evitar codificar, llamamos a una plantilla donde ya tenemos la interfaz creada. Solo debemos enviarle los datos correctamente para que esta los muestre.

ARMADO DE UN TEMPLATE

Gracias a las plantillas o templates, Django logra simplificar la programación a muy poco código. Como ya hemos visto, estas son llamadas desde las vistas donde se les pasan los datos por mostrar.

Luego Django se encarga de completar las plantillas mostrando ordenadamente el contenido.

Supongamos que tenemos que hacer un ABM (*Activity-Based Management*) de artículos, categorías, unidades y proveedores. Tradicionalmente deberíamos generar un archivo para cada uno de los módulos. Si bien podríamos reutilizar el código de uno a otro, nos vemos en la necesidad de generar cuatro archivos distintos encargados de gestionar los artículos, las categorías, las unidades y los proveedores.

Con Django, nos bastaría crear un solo template que será llamado desde el archivo **views.py** con distintos parámetros.

Como ya hemos dicho, los template no son otra cosa que archivos codificados en HTML, JS, CSS, entre otros lenguajes, a los que se les agregan etiquetas propias de Django. Cada etiqueta se abre y se cierra con una llave y el signo por ciento (%).

Dentro de las etiquetas más importantes se pueden enumerar las siguientes:

{{CAMPO}}

Con la doble llave, se hace referencia a un campo enviado como parámetro al template. Por ejemplo, si pasamos un registro animal, el cual posee como valor **pato** en el atributo **especie** y el valor **Lucas** en el atributo **nombre**, podríamos escribir lo siguiente:

```
<p> {{animal.nombre}} es un {{animal.especie}}</p>
```

Lo que devolvería:

“Lucas es un pato”

{% IF %}

Esta etiqueta evalúa una variable, si existe, no está vacía y su valor no es falso; inserta el contenido entre **{% if %}** y el **{% endif %}**.

Por ejemplo:

```
{% if edad > "18" %}  
<p> Eres mayor de edad!!</p>  
{% else %}  
    <p> Todavía eres menor!!! <p>  
{% endif %}
```

Es importante resaltar que el bloque **else** es opcional.

{% FOR %}

Esta etiqueta itera para cada elemento de una lista o secuencia enviada al template. En cada iteración, Django mostrará el contenido de las etiquetas **{% for %}** y el **{% endfor %}**.

Por ejemplo:

```
<ul>  
{% for doc in doctores %}  
<li> {{doc.nombre}}-{{doc.especialidad}}</li>  
{% endfor %}  
</ul>
```

Es importante tener en cuenta que Django no admite rupturas en los ciclos, por lo que, si hay valores que no queremos mostrar, debemos limpiar la variable antes de enviarla al template.

Django asigna al ciclo **for** la variable **forloop**, que almacena datos importantes de iteración.

Algunos de los atributos más empleados son:

- **forloop.counter**: cuenta la cantidad de veces que se entra al bucle, comenzando en 1.
- **forloop.counter0**: cuenta la cantidad de veces que se entra al bucle indexando desde el 0.
- **forloop.revcounter**: indica en cada iteración la cantidad de vueltas que faltan para finalizar el bucle. Este atributo, toma el valor **1** cuando itera en la última vuelta.
- **forloop.revcounter0**: indica la cantidad de vueltas faltantes para finalizar el ciclo, tomando como valor **0** en el último ciclo.
- **forloop.first**: es un valor booleano que toma el valor **true** si es el primer ciclo el que se está ejecutando.

- **forloop.last**: este atributo toma el valor **true** si se está ejecutando la última iteración del ciclo.

HOLA MUNDO

Ya hemos visto lo básico para generar nuestro proyecto. Lo que haremos ahora es codificar para obtener el tradicional “Hola Mundo”.

Para ello, generaremos un template con el cual le daremos el formato a nuestra página. Crearemos una función en el archivo **views.py** y redireccionaremos la dirección por defecto de nuestro servidor para que nos salude cada vez que accedamos a la aplicación.

EL TEMPLATE

Para crear nuestro template, accederemos a la carpeta de nuestra aplicación – **holaapp** – y crearemos la siguiente estructura:

```
|--holaapp
  |--template
    |--holaapp
      |--saludo.html
```

Hemos creado un directorio **template** dentro del directorio de nuestra aplicación. Dentro de este, creamos un nuevo directorio con el mismo nombre que la aplicación, y dentro de este el archivo que oficialará de plantilla, al que llamaremos **saludo.html**.

Nuestra plantilla tendrá la estructura básica de un archivo HTML, y dentro de un div, le introduciremos centrado el texto **Hola Mundo!!!**.

El código de nuestro archivo **saludo.html** es el siguiente:

```
<!DOCTYPE html>
<html>
<head>
  <title>Saludo de Red Users</title>
</head>
<body>
  <p>”Hola {{nombre}}!!!”</p>
</body>
</html>
```

LA VISTA

El paso siguiente consiste en crear una función en el archivo vista con el cual direcccionaremos a la plantilla para renderizar la página **saludo.html** como respuesta a la solicitud del usuario.

En el archivo **views.py** escribiremos el siguiente código:

```
from django.shortcuts import render

def index(request):
    return render(request, 'holaapp/saludo.
html',{'nombre':'RedUser'})
```

Básicamente lo que estamos haciendo es:

- Importar una librería para renderizar la página en el navegador.
- Definir la función **index**, que recibe como parámetro la página solicitada.
- Devolver como resultado de la función el renderizado de la plantilla **saludo.html** mediante el llamado de la función **render()**.

REDIRECCIONAMIENTO DE LA URL

Lo último que nos queda por hacer para que nuestro proyecto funcione es configurar el redireccionamiento de las peticiones del usuario.

Para ello, crearemos un archivo **urls.py** dentro del directorio de la aplicación – no del proyecto– y pondremos el siguiente código:

Función **render(request,template_name,context)**

Combina un determinado template y lo relaciona con la información de un contexto, devolviendo un **HttpResponse**, que puede ser renderizado por el navegador.

Parámetros aceptados:

- **request**: el objeto que genera la solicitud;
- **template_name**: el nombre de la plantilla que se va a llamar;
- **context**: el contexto o los valores que se van a combinar con la plantilla.

```
from django.urls import path  
  
from . import views  
  
urlpatterns = [  
    path('', views.index, name='index'),  
]
```

En este caso, lo que hacemos es importar el archivo **views.py**. Como está en el mismo directorio que el archivo que estamos editando, no es necesario que le pongamos la extensión **py**. Y como Python interpreta todo el directorio como un paquete, decimos que lo importe desde el propio paquete agregando **from . import**.

Por último, hacemos llamada a la función **path** para redireccionar la solicitud a la función **index** del archivo **views**.

CONFIGURAR EL PROYECTO

Ya tenemos todos los archivos necesarios, ahora lo que haremos es configurar el proyecto para incluir nuestra aplicación y para que redireccione sus URL.

Para que el proyecto reconozca las URL, deberemos acceder al archivo **urls.py** de la carpeta **holauers** y editarlo de la siguiente manera:

```
from django.urls import include, path  
  
urlpatterns = [  
    path('holaapp/', include('holauers.holaapp.urls')),  
]
```

byYako1976 www.identi.io
Función **path(route, view, kwargs=None, name=None)**

Devuelve el elemento para incluir en urlpatterns:

- **route:** es un string que indica la solicitud del usuario;
- **view:** es la función de la vista llamada para renderizar;
- **kwargs:** son parámetros adicionales por pasar;
- **name:** un nombre de referencia para futuras llamadas.

En resumen, le dijimos al proyecto que, cada vez que se hace una petición a **holaapp**, nos redireccione según lo que esté configurado en el archivo **urls.py** de esa aplicación.

Y como último paso, antes de poner a prueba nuestro proyecto, deberemos incluir nuestra aplicación dentro del archivo **settings.py**.

Para ello, buscaremos la etiqueta **INSTALLED_APPS**, borraremos todo su contenido y le agregaremos **holausers.holaapp**, por lo que debería quedarnos así:

```
INSTALLED_APPS = [  
    'holausers.holaapp',  
]
```

LANZAR EL SERVIDOR Y ACCESO AL SITIO

Ya tenemos todo listo para poner en marcha nuestro sitio, pero para esto debemos lanzar nuestro servidor.

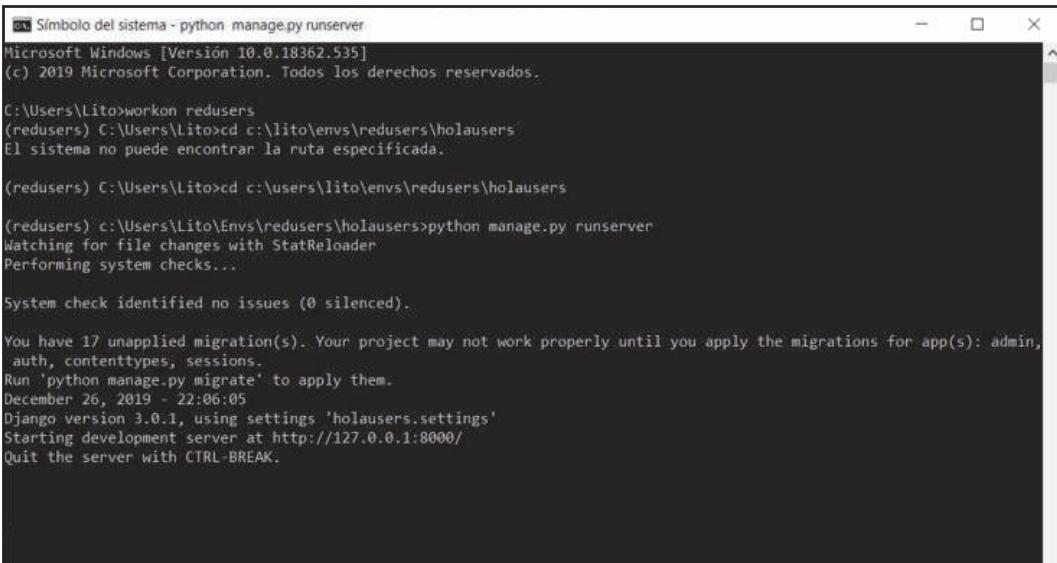
Dicho lanzamiento se lleva a cabo desde la consola de comandos mediante la siguiente orden:

```
python manage.py runserver
```

pero, antes de escribir este código, debemos asegurarnos de que estamos dentro del entorno virtual correspondiente. Y dentro de este entorno, deberemos acceder al directorio de nuestro proyecto, donde está el archivo **manage.py**.

Entonces, abriremos la consola de comandos y escribiremos las siguientes órdenes:

```
workon redusers  
cd c:\Users\%user-name\Envs\redusers\holausers  
Python manage.py runserver
```



```
Símbolo del sistema - python manage.py runserver
Microsoft Windows [Versión 10.0.18362.535]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Lito>workon redusers
(redusers) C:\Users\Lito>cd c:\lito\envs\redusers\holauers
El sistema no puede encontrar la ruta especificada.

(redusers) C:\Users\Lito>cd c:\users\lito\envs\redusers\holauers

(redusers) c:\Users\Lito\Env\redusers\holauers>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
December 26, 2019 - 22:06:05
Django version 3.0.1, using settings 'holauers.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Ejecución del servidor de Django.

Por último, abriremos el navegador y accederemos a **localhost:8000/holaapp/**, y nos abrirá la siguiente página:



Diseño final de nuestro proyecto.

RESUMEN

Como Django es un framework que hace muchas tareas de manera automática y transparente, resulta fundamental que el código esté en correcto diseño, pero también debemos respetar la estructura de los directorios.

Una vez que dominemos esta estructura y la lógica del funcionamiento como ya hemos explicado, veremos que podremos disminuir considerablemente los tiempos de diseños y la eficiencia en la implementación de grandes proyectos.

Para el proyecto que hemos diseñado, esta estructura debe responder al esquema que se muestra en la siguiente imagen.

Carpetas

```
▼ └── holausers
    └── holausers
        └── _pycache_
            └── holaapp
                └── migrations
                    /* __init__.py
                └── templates
                    └── holaapp
                        <> saludo.html
                        /* __init__.py
                        /* admin.py
                        /* apps.py
                        /* models.py
                        /* tests.py
                        /* urls.py
                        /* views.py
                        /* __init__.py
                        /* asgi.py
                        /* settings.py
                        /* urls.py
                        /* wsgi.py
                    └── db.sqlite3
                    /* manage.py
```

Carpeta de General

Carpeta de Proyecto

Carpeta de la Aplicacion

Carpeta para Plantillas

Plantilla del index

Estructura de archivos y carpetas final.

Programación en



ACERCA DE ESTE CURSO

Python es un lenguaje multiplataforma y multiparadigma sumamente versátil que se ha vuelto muy popular en los últimos tiempos, debido, entre otros motivos, a que se convirtió en el lenguaje de elección para las aplicaciones de Inteligencia Artificial. Este curso de Python en tres volúmenes permite aprender desde cero lo necesario para aprovechar el potencial de este lenguaje de programación, contando con ejemplos prácticos que nos ayudarán a comprender, desde las propias bases del lenguaje, hasta temas avanzados como el paradigma de programación orientada a objetos.

EL VOLUMEN III

En este e-book aplicaremos Python implementando proyectos prácticos en placas con microcontroladores. En primer lugar, recorreremos rápidamente la placa Raspberry -pi para conocer sus puertos y formas de conexión. Luego veremos cómo, con la librería GPIO, podremos interactuar con los pines para emitir y/o recibir señales del mundo exterior. Para terminar con la placa Raspberry y aprender lúdicamente, llevaremos a la práctica un pequeño script, con el cual interactuaremos con el conocido juego Minecraft. Por último, en los dos últimos capítulos de este trabajo, conoceremos a MicroPython, veremos su potencialidad y llevaremos a la práctica un proyecto sobre la placa Pyboard donde, con un potenciómetro, cambiaremos la posición de un servo motor indicando en una pantalla el ángulo tomado.

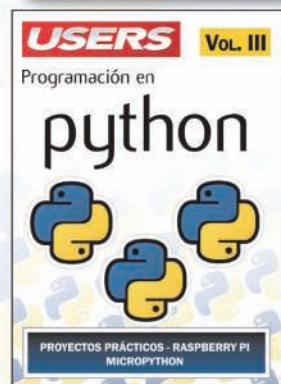
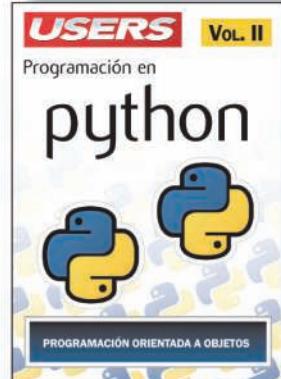


SOBRE EL AUTOR

Edgardo Stasi, es un ingeniero en informática argentino recibido en el año 2004. En la actualidad reside en San Clemente del Tuyú, una ciudad costera de la provincia de Buenos Aires, Argentina. A lo largo de su carrera, fue aprendiendo distintos lenguajes que le permitieron diseñar proyectos para ejecutarse en diversos entornos: Linux, Windows, Web web y dispositivos móviles .

REDUSERS.com

En nuestro sitio podrá encontrar noticias relacionadas y participar de la comunidad de tecnología más importante de América Latina.



REDUSERS.PREMIUM.COM

RedUSERS Premium es la biblioteca digital de USERS. Accederás a cientos de publicaciones: Informes; eBooks; Guías; Revistas; Cursos. Todo el contenido está disponible Online - Offline y para cualquier dispositivo. Publicamos, al menos, una novedad cada 7 días

