Introducción a Python

Laboratorio de Instrumentación Virtual y Robótica Aplicada
UNMdP

Índice

1. Introducción

Python es un lenguaje de alto nivel, de fácil aprendizaje con un simple pero efectivo enfoque orientado a objetos. Cuenta con una sintaxis elegante con tipos dinámicos. El código es interpretado, es decir que pasa por un programa intérprete que lo ejecuta línea por línea.

A partir del intérprete podemos ejecutar instrucciones y obtener el resultado de forma inmediata o ejecutar un programa entero, almacenado con extensión `.py`. Al usar el intérprete, se indica con `>>>` que está esperando una instrucción. También podemos agregar comentarios usando `#` que hace que el resto de la línea a partir del `#` sea un comentario y se ignore para la ejecución.

2. Operaciones con números

Las operaciones numéricas se ingresan de la forma esperada, respetando el orden de operaciones. Los símbolos utilizados son:

- `+` para suma
- `-` para resta
- `*` multiplicación
- `/` división
- `//` división entera (descarta decimales)
- `%` resto de la división
- `**` potencia

Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5
1.6
>>> 8 // 5
1
>>> 8 % 5
3
>>> 3**2
9
```

3. Variables

Una vez obtenidos los resultados, se pueden almacenar en una variable. No es necesario indicar el tipo de la variable como también se pueden almacenar diferentes tipos de valores dentro de una misma variable. Para esto se utiliza el símbolo =, como por ejemplo:

```
>>> a = 1 + 2
>>> b = a / 2
>>> b
1.5
>>> b = "texto"
>>> b
'texto'
```

También se puede combinar un operador con la asignación para modificar una variable:

```
>>> a = 1
>>> a += 1
>>> a
2
>>> a *= 2
>>> a
4
>>> a /= 2
>>> a
2.0
```

Como se observa en el ejemplo, al ingresar como instrucción el nombre de una variable, en el intérprete muestra el valor de la misma. Sin embargo, hay otra forma de mostrar información y es utilizando la instrucción `print()`:

```
>>> print(b)
texto
```

Es posible ingresar varios parámetros y se presentarán todos separados por espacios:

```
>>> print("cadena", 1, 2, b)
texto 1 2 texto
```

4. Cadenas de texto

Existen tres maneras de ingresar texto con Python:

- `"texto"`

- `'texto'`
- `"""texto
  multilínea"""`

Pueden utilizarse de forma indistinta pero en las cadenas de texto encerradas por comillas simples, pueden ingresarse comillas dobles y viceversa.

Existen cadenas de textos especiales que sirven para incluir variables dentro de ellas. Se denominan *f-strings* y se utilizan colocando una f delante de las comillas. Dentro de la cadena se pueden indicar el nombre de las variables, su formato y algunos otros parámetros como por ejemplo:

```
>>> pi = 3.14159
>>> print(f"La variable pi contiene el valor {pi}")
La variable pi contiene el valor 3.14159
>>> print(f"{pi=}")
pi=3.14159
>>> print(f"Ahora con menos decimales: {pi:.2f}")
Ahora con menos decimales: 3.14
```

Las cadenas de texto tienen métodos asociados (funciones pertenecientes a objetos) que permiten manipularlas y obtener información relevante sobre ella. Se puede encontrar en el manual de referencia un listado de todos esos métodos. A continuación se presentan algunos ejemplos útiles:

```
>>> texto = "abc def ghi jkl"
>>> texto.split(" ")
['abc', 'def', 'ghi', 'jkl']
>>> texto.find("def")
4
>>> texto[texto.find("def")]
'd'
>>> texto.replace("def", "abc")
'abc abc ghi jkl'
>>> texto
'abc def ghi jkl'
```

5. Listas

Son similares a lo que en otros lenguajes denominan *arrays* o vectores. Estas estructuras son dinámicas por lo que se pueden agregar o quitar elementos una vez creados. Para crear una lista se utilizan el par de corchetes `[]` con los elementos separados por comas:

```
>>> a = [1, 2, 3]
```

Para acceder a los elementos de una lista, se selecciona el índice comenzando por 0:

```
1
>>> a[0]
>>> a[1]
2
```

También se puede obtener una sublista a partir del rango de los índices indicando desde qué índice, hasta cuál y con qué paso. Estos valores pueden omitirse. El primero corresponde al primer índice de la lista y, en caso de omitirse, corresponde al primero de la lista. El segundo, es el último índice no inclusive de la lista que si se omite es el último de la lista. Por último el paso es en cuánto incrementa el índice que, al omitirlo, es 1. En resumen:

```
lista[inicio:fin:paso]
```

Si se omiten estos valores, por defecto se asume `inicio` como 0, `fin` como la cantidad de elementos de la lista y `paso` como 1. A continuación se muestran unos ejemplos de su uso:

```
>>> a[1:2]
[2]
>>> a[:2]
[1, 2]
>>> a[1:]
[2, 3]
>>> a[0:3]
[1, 2, 3]
>>> a[0:3:2]
[1, 3]
>>> a[::2]
[1, 3]
```

Para agregar un elemento a la lista, se pueden usar los métodos (propios de cada lista) `append()` para agregar al final e `insert()` para agregar en un índice determinado. También se puede utilizar la función `len()` para consultar la cantidad de elementos que contiene. Por último, se puede utilizar la palabra clave **`del`** para eliminar un elemento de la lista:

```
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> len(a)
4
>>> a.insert(1, 100)
>>> a
[1, 100, 2, 3, 4]
>>> len(a)
5
>>> del a[1]
[1, 2, 3, 4]
```

6. Flujo de control

Para controlar la forma en la que se ejecuta una porción de código, podemos encontrar a la palabra clave **`if`** para ejecutar algo bajo ciertas condiciones y las palabras claves **`for`** y **`while`** para repetir un bloque de código.

Otros lenguajes utilizan símbolos como `\{\}` o `begin-end` para encerrar un bloque, sin embargo, Python requiere ubicarlos dentro de cierto nivel de indentación, es decir las instrucciones deben tener la **misma** cantidad de espacios al comienzo de la línea. Una forma de garantizarlo es usando la tecla `<Tab>` donde, según el editor, usa el caracter especial de tabulación o convierte automáticamente a una cantidad de espacios.

6.1. Condicionales

Se componen de la palabra clave **`if`** seguida de una condición que pueda evaluarse como verdadero (**`True`**) o falso (**`False`**) y dos puntos `:`. El bloque siguiente de código se evalúa siempre y cuando la condición sea verdadera:

```
>>> a = 1
>>> if a == 1:
        print("La condición es verdadera")
    print("Esto se ejecuta siempre")
La condición es verdadera
Esto se ejecuta siempre
```

Algunos operadores que se pueden utilizar para generar las condiciones son:

- `==`: compara igualdad
- `!=`: compara desigualdad

- >: compara si es mayor
- <: compara si es menor
- >=: compara si es mayor o igual
- <=: compara si es menor o igual
- `in`: verifica pertenencia en una lista, en una cadena de caracteres, etc.

Estos operadores producen un valor *booleano*:

```
>>> 1 == 2
False
>>> 1 != 2
True
>>> "a" in "abcdef"
True
>>> 4 in [1, 2]
False
```

Si la condición no es verdadera, es posible redireccionar el flujo a otro bloque usando las palabras claves **`else`** y **`elif`**. Ambas se utilizan para los casos en que la condición inicial es falsa pero, en la segunda, vuelve a evaluar una nueva condición como si fuera **`else`** seguido de **`if`**:

```
>>> a = 1
>>> if a == 1:
        print("a es 1")
    elif a > 1:
        print("a no es 1 pero es mayor a 1")
    else:
        print("a no es 1 ni es mayor a 1")
a es 1
```

6.2. Ciclos

Podemos distinguir dos tipos de bucles: **`for`** y **`while`**. El primero repite un bloque de código hasta agotar un conjunto de valores iterables, mientras que el segundo se repite hasta que se cumpla cierta condición. Para ambos casos, el ciclo puede ser interrumpido arbitrariamente usando la palabra clave **`break`**.

Para el caso del ciclo **`for`**, lo que se ingresa a continuación es una (o varias) variables que se asignan al comienzo de cada repetición seguido de la palabra clave `in` y un *iterable*. Esto último se refiere a algo que contenga o genere una cantidad limitada elementos de forma ordenada. Dos formas básicas de utilizarlo es con el iterable que genera la función `range()` y la otra es recorriendo los elementos de una lista. Por ejemplo:

```
>>> for i in range(5):
        print(i)
0
1
2
3
4
>>> for i in range(1, 5):
        print(i)
1
2
3
4
>>> for i in [1, 2, 3, 4]:
        print(i)
1
2
3
4
```

Por otro lado, el ciclo **while** se repite hasta que la condición sea falsa:

```
>>> a = 0
>>> while a < 3:
        print(a)
        a += 1
0
1
2
```

Se puede generar un ciclo infinito colocando siempre el valor verdadero:

```
>>> while True:
        print("Se repite infinitamente")
Se repite infinitamente
Se repite infinitamente
Se repite infinitamente
Se repite infinitamente
Se repite infinitamente
...
```

En este caso, podemos interrumpir la ejecución del ciclo usando la combinación <Ctrl-C>.

6.3. Excepciones

Al producirse algún error durante la ejecución, Python corta la ejecución y muestra una *excepción* que permite dar una idea de dónde puede prevenir el error. Normalmente vienen asociadas a un tipo (**ValueError**, **IndexError**,

NameError, etc.) en relación al error producido. Sin embargo, es posible capturar estos errores utilizando las palabras claves **try** y **except** durante ejecución y realizar algo al respecto. Esto se puede usar para salvar el error y continuar con la ejecución. Por ejemplo, si queremos acceder a una variable que no existe:

```
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' isn't defined
```

En este caso, se produce un **NameError** indicando que la variable no está definida. Si colocamos un bloque **try**, podemos capturar esta excepción:

```
>>> try:
        variable
    except:
        print("La variable no existe, le asignamos un valor")
        variable = 1
La variable no existe, le asignamos un valor
>>> variable
1
```

Es buena práctica luego de **except**, indicar el tipo de excepción a capturar. En caso de omitirlo, captura todas. En nuestro caso sería **NameError**. Esto es útil a la hora de capturar cierto tipo de errores como pueden ser que no exista un archivo, que no pueda conectarse a la red, etc.

7. Funciones

Para evitar repetir bloques de código se puede hacer uso de las funciones que pueden o no devolver un valor. Para definirlas, incluimos en cualquier parte del código la palabra **def** seguida del nombre de la función y sus parámetros:

```
>>> def funcion(a, b, c):
        print("Se llamó a la función con parámetros", a, b, c)
>>> funcion(1, 2, 3)
Se llamó a la función con parámetros 1 2 3
>>> funcion("a", 0, True)
Se llamó a la función con parámetros a 0 True
```

En este caso, la función no devuelve ningún valor, solo muestra información en pantalla. Por otro lado, podemos definir una función que realice una operación específica y devuelva un valor utilizando la palabra clave **return**:

```
>>> def suma_primeros_n_numeros(n):
        total = 0
        for i in range(1, n + 1):
            total += i

        return total
>>> suma_primeros_n_numeros(5)
15
```

8. Módulos

El lenguaje Python permite modularizar código de una forma muy sencilla. Es posible acceder a definiciones en otro archivo a partir de la palabra clave **`import`**. Supongamos que existe un archivo que se llama `modulo.py` en el directorio actual de trabajo con el siguiente contenido:

```
def opuesto(a):
    return -a

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

Podemos acceder a estas funciones de la siguiente manera:

```
>>> import modulo
>>> modulo.saludo()
Hola
>>> modulo.opuesto(1)
-1
```

Una forma alternativa de acceder a estas funciones es importándolas individualmente de la siguiente manera:

```
>>> from modulo import saludo
>>> saludo()
Hola
```

Introducción a MicroPython

Laboratorio de Instrumentación Virtual y Robótica Aplicada
UNMdP

Índice

Manual de referencia

1. Instalación

Antes de comenzar a usar Micropython en una placa ESP32, es necesario descargar el firmware correspondiente. Este puede descargarse desde el sitio oficial. En caso de contar con otro dispositivo, la lista completa se encuentra aquí.

Para programar código MicroPython se sugiere usar el entorno de programación multiplataforma Thonny que, a su vez, facilita la instalación del firmware.

Utilizando Thonny, se puede instalar el firmware desde el menú `Tools ->Options`.

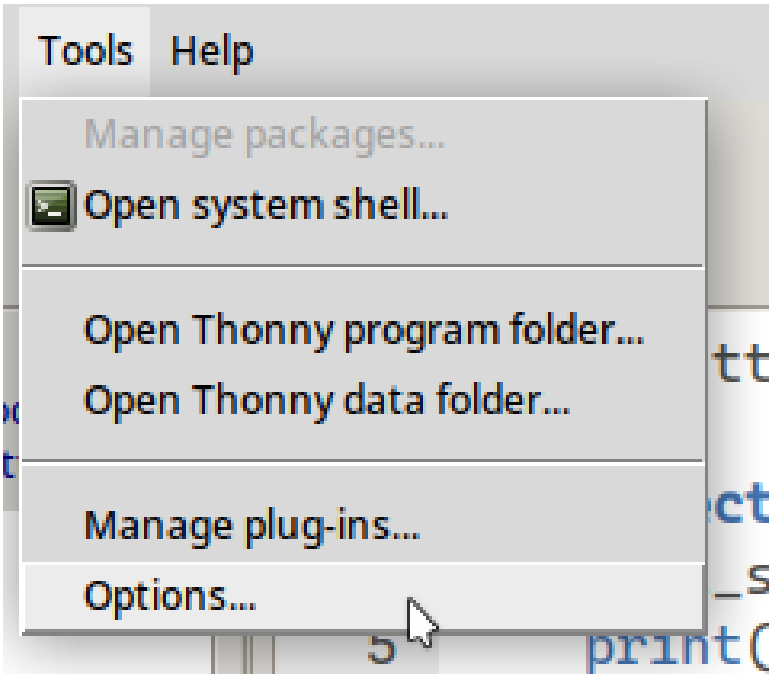

Figura 1: Opciones

Se presentará una ventana con varias pestañas, de las cuales hay que seleccionar la llamada `Interpreter`. Luego, en esa pestaña, seleccionar el intérprete para ESP32. Se debería ver lo siguiente:

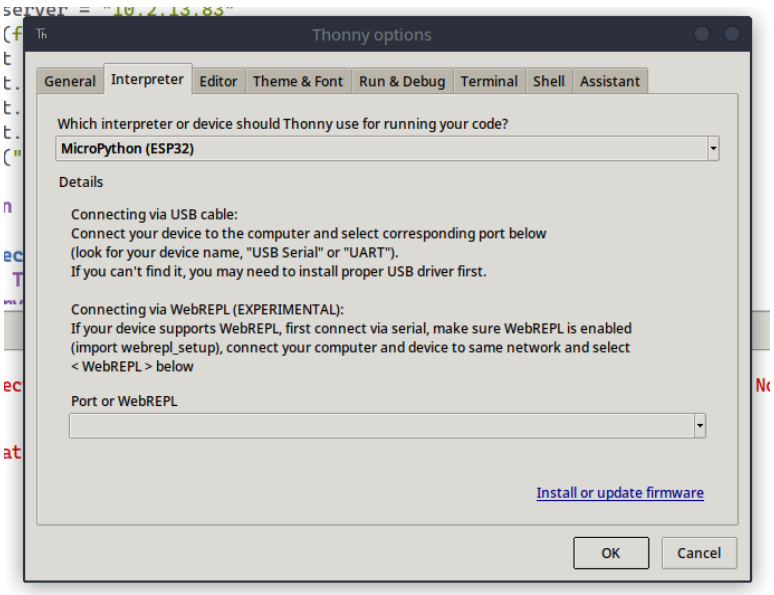

Figura 2: Selección de intérprete

Luego se selecciona abajo a la derecha donde dice *Install or update firmware*:

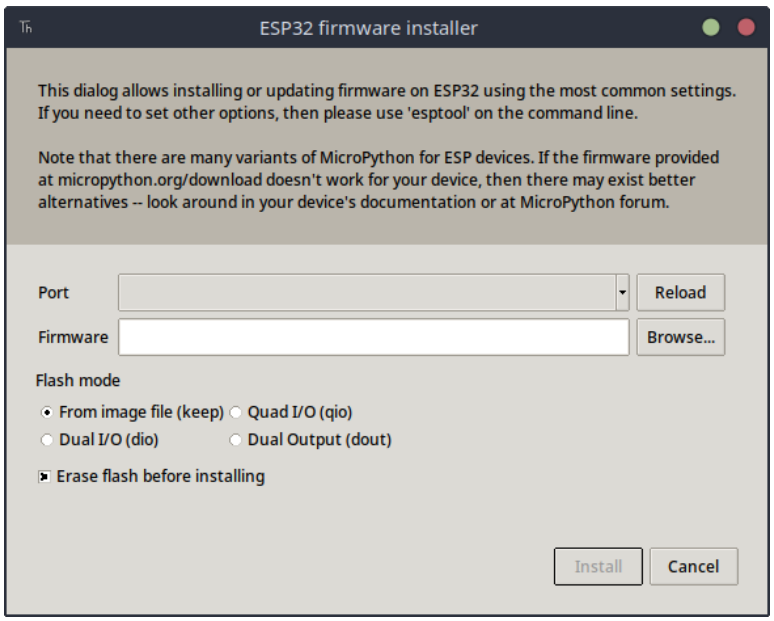

Figura 3: Instalación de firmware

En el campo `port` se debe seleccionar el puerto en el que está conectado la ESP32. En el campo `firmware` se debe seleccionar el archivo `.bin` que

se descargó de la página de Micropython. Luego, se presiona el botón *Install* para comenzar la instalación. Debajo a la izquierda de esa misma ventana, debería aparecer el progreso.

Una vez terminado, se puede cerrar la ventana para volver a la anterior. Si no se hizo anteriormente, se debe seleccionar el mismo puerto que se seleccionó para instalar el firmware donde dice *Port or WebREPL*.

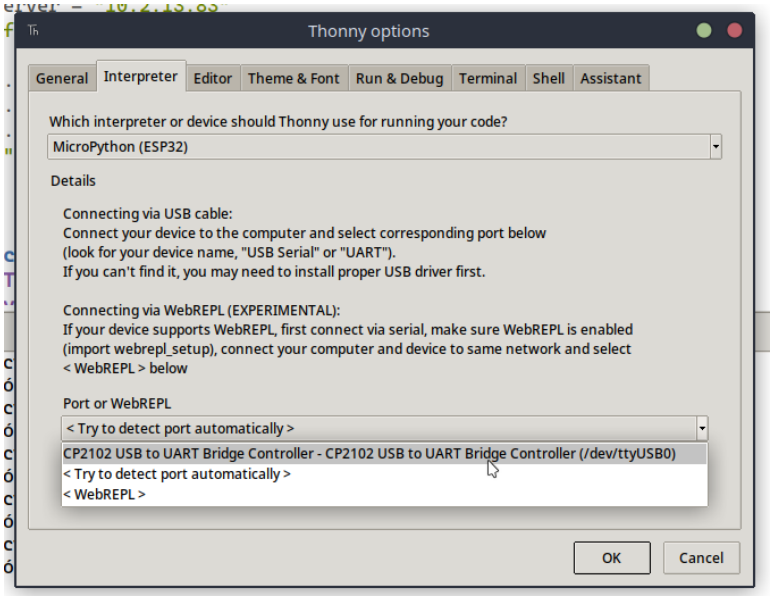

Figura 4: Selección de puerto

Finalmente, se debería ver abajo del editor una pestaña que diga *shell* que permita ingresar instrucciones con la inscripción Micropython como se ve a continuación:

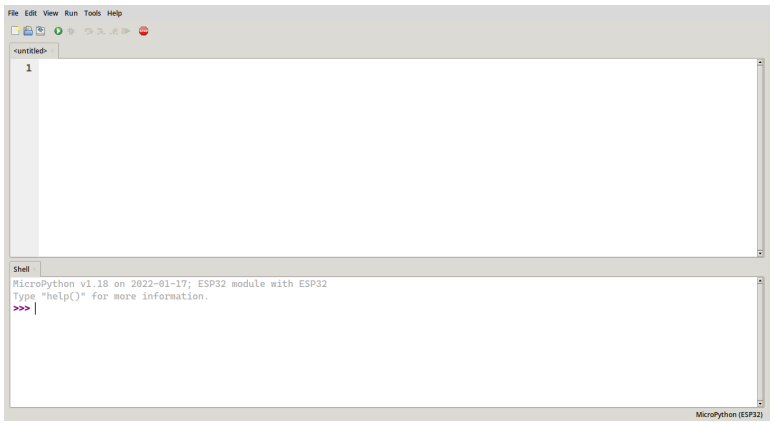

Figura 5: Shell

En el caso que esta pestaña no sea visible, se puede mostrar desde el menú `View ->Shell`.

2. Archivos

Dentro del entorno Thonny, es posible ejecutar código desde la línea de comandos directo sobre la ESP32 o se puede escribir un programa para luego cargarlo en ella. En primer lugar, la línea de comandos sirve para experimentar y probar nuevos conceptos mientras que el programa nos permite repetir código más complejo o almacenarlo directo en su memoria para ser ejecutado cada vez que se inicia.

Para ver los archivos almacenados en la ESP32, se puede abrir el panel `Files` desde el menú `View ->Files`. Allí, muestra los archivos almacenados localmente en la computadora y los archivos almacenados en la ESP32.

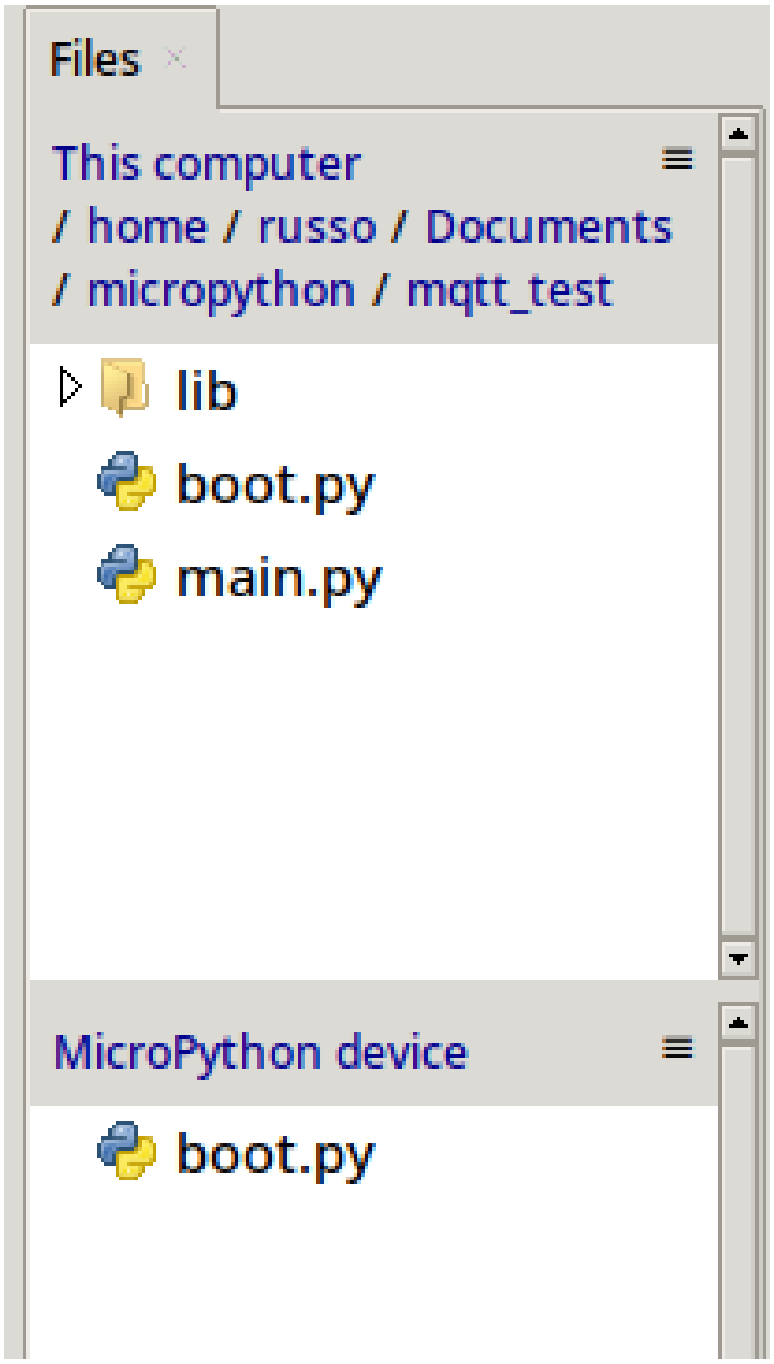

Figura 6: Panel *Files*

Luego de instalar el firmware, en la ESP32, habrá un solo archivo: `boot.py`. Sin embargo, se espera que hayan dos archivos:

- `boot.py` que se ejecuta primero al iniciar el dispositivo

- `main.py` que se ejecuta a continuación

Normalmente, uno esperaría almacenar en `boot.py` aquello que requiera ser configurado una sola vez (como por ejemplo la conexión WiFi) mientras que en `main.py` se puede colocar el programa principal y no generar ningún conflicto al ser modificado y vuelto a ejecutar. Se pueden entender como una especie de `setup` y `loop` de Arduino pero con la particularidad que el archivo `main.py` se ejecuta una única vez.

Los archivos se pueden editar desde su forma local como directo sobre el dispositivo. Luego, con las opciones de botón derecho, se pueden transferir de un lado hacia el otro con las acciones *download* y *upload*.

3. Entradas/Salidas

Para acceder a los pines de entrada/salida de la ESP32, hay que importar del módulo `machine` la definición correspondiente:

```
from machine import Pin

pin = Pin(19, Pin.OUT)
```

Al crear el objeto `pin`, se deben pasar como parámetros el número de pin y su modo (entrada o salida). En caso de que se ingrese solo el número de pin, se asume que es una entrada (que también puede explicitarse con `Pin.IN`).

Para acceder o modificar su valor, se puede usar el método `value()` de la siguiente manera:

```
pin.value()   # devuelve el valor
pin.value(1)  # coloca en alto el pin
```

También podemos configurar una salida como PWM (modulación de ancho de pulso). Para esto usamos la función `PWM` definida en el módulo `machine` y pasamos como argumento uno de los pines compatibles.

```
from machine import PWM, Pin

pwm = PWM(Pin(19))
pwm.freq(10000)  # configuramos la frecuencia a 10kHz
pwm.duty(512)    # configuramos el duty cycle al 50% (0 - 1023)

# También podemos configurarlo todo en una sola línea
pwm = PWM(Pin(19, freq=10000, duty=512))

# Podemos ver la configuración actual imprimiendo el objeto pwm
print(pwm)

# Detenemos la salida
pwm.deinit()
```

4. Control de tiempo

Podemos usar el módulo `time` que nos permite contar tiempos para poder realizar pausas o medir tiempo entre operaciones:

```
import time
time.sleep(1)           # espera 1 segundo
time.sleep_ms(1000)     # ídem a anterior pero en milisegundos
time.sleep_us(1000000)  # ídem a anterior pero en microsegundos

inicio = time.ticks_ms()  # almacena la cantidad de ms
                          # desde el inicio del dispositivo

# se ejecutan varias operaciones que toman tiempo

fin = time.ticks_ms()

# Se muestra la diferencia de tiempos en la misma unidad (ms)
print(time.ticks_diff(fin, inicio))
```

También se pueden programar tareas repetitivas o después de pasado determinado tiempo usando los *Timers* internos de la ESP32.

```
from machine import Timer

def tarea_repetitiva(timer):
    print("Tarea invocada por el timer", timer)

tim = Timer(0)
tim.init(period=1000, mode=Timer.PERIODIC,
         callback=tarea_repetitiva)
```

En este ejemplo, se inicializa un timer que ejecuta una función de forma

periódica cada 1 segundo (1000 ms). La función debe incorporar un parámetro que corresponde al timer que la ejecutó. Alternativamente, se puede cambiar el modo `mode=Timer.ONE_SHOT` que hace que se ejecute una sola vez pasado el tiempo indicado por `period`.

5. Sensores, actuadores y periféricos

5.1. Control de servomotor

Utilizando el concepto de PWM visto anteriormente, es posible controlar el ángulo de giro de un servomotor según el ciclo de trabajo. Para esto, primero se configura una salida PWM a 50Hz y los anchos de pulso según el ángulo requerido. A modo de referencia, deberían tomar aproximadamente los siguientes valores:

- Para la posición 0°: pulso de 1ms
- Para la posición 90°: pulso de 1.5ms
- Para la posición 180°: pulso de 2ms

Luego se ajusta el ciclo de trabajo acorde a la resolución del PWM (0 a 1023) y, teniendo en cuenta que 0ms se configura con 0 y 20ms es el pulso completo, se obtienen los siguientes valores:

- Para la posición 0°: ciclo de trabajo 51
- Para la posición 90°: ciclo de trabajo 76
- Para la posición 180°: ciclo de trabajo 102

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

pwm = PWM(Pin(14))
pwm.freq(50)

for i in range(51, 103):
    pwm.duty(i)
    sleep_ms(10)
```

5.2. Conversión analógico-digital

El ESP32 contiene puertos de conversión de entrada analógica a digital ubicados en los pines 32 a 39 (bloque 1) y los pines 0, 2, 4, 12 a 15 y 25 a 27

(bloque 2). Sin embargo, el bloque 2 es usado también por WiFi por lo que no se pueden usar en simultáneo.

Para configurar un pin, se lo pasamos a la función ADC del módulo `machine` de la misma manera que con PWM. Luego podemos leer su valor con el método `read()` o `read_uv()` para obtener la tensión en µV. También es posible ajustar el factor de atenuación para ampliar el rango de conversión:

- `ADC.ATTN_0DB`: 100mV - 950mV
- `ADC.ATTN_2_5DB`: 100mV - 1250mV
- `ADC.ATTN_6DB`: 150mV - 1750mV
- `ADC.ATTN_11DB`: 150mV - 2450mV

```
from machine import ADC

adc = ADC(Pin(35))
print(adc.read())
```

5.3. DAC

Tambien es posible generar valores analógicos arbitrarios utilizando el conversor digital-analógico integrado al ESP32. Este nos permite convertir valores digitales de 8 bits a tensión de salida.

```
from machine import Pin, DAC

dac = DAC(Pin(25))

dac.write(127)
```

5.4. Sensor de temperatura y humedad DHT11/22

El lenguaje MicroPython incorpora una implementación del driver de los dispositivos DHT11 y DHT22.

```
from dht import DHT11
from machine import Pin

dht = DHT11(Pin(32))

dht.measure()
print(f"Temperatura: {dht.temperature()} °C")
print(f"Humedad: {dht.humidity()} %")
```

5.5. NeoPixel

También se incluye el driver para LEDs `ws2812b`, también conocidos como NeoPixel.

```
from neopixel import NeoPixel
from machine import Pin

np = NeoPixel(Pin(27), 1)

# Orden: G, R, B
np[0] = (0, 255, 0)

# Se aplican los cambios
np.write()
```

6. WiFi

Para conectarnos a una red o crear una red propia en la ESP32, podemos usar el módulo `network`.

6.1. ESP32 como access point

```
import network

ap = network.WLAN(network.AP_IF)  # se configura el modo AP
ap.config(essid="nombre de red")
ap.config(max_clients=4)  # cantidad máxima de conexiones
ap.active(True)
```

6.2. Conexión a una red existente

Para conectarnos a una red existente, podemos hacerlo interactivamente desde la línea de comandos para comprender cómo es el proceso:

```
>>> import network
>>> wlan  = network.WLAN(network.STA_IF)
>>> wlan.active(True)
True
```

En primer lugar, se crea el objeto `wlan` con el cual usaremos para conectarnos a una red WiFi. Con el método `active()` activamos o desactivamos la red. Una vez activa, podemos escanear las redes visibles:

```
>>> wlan.scan()
```

Después de un tiempo, se genera una lista de redes con sus nombres, potencias y otros parámetros. Para ver una simple lista con los nombres, podemos usar un ciclo:

```
>>> for red in wlan.scan():
>>>     print(red[0])
```

Para conectarnos con la red, se puede usar el método `connect`:

```
>>> wlan.connect("nombre de red", "contraseña")
```

Luego de un tiempo, podemos verificar si se conectó exitosamente:

```
>>> wlan.isconnected()
True
>>> wlan.ifconfig()[0]
10.2.212.55
```

Para automatizar este proceso y pueda ser ejecutado en la ESP32 y bloquee la ejecución hasta que logre conectarse, se puede usar el siguiente script:

```
import network
from time import sleep_ms

wlan = network.WLAN(network.STA_IF)
if not wlan.active():
    wlan.active(True)

if not wlan.isconnected():
    wlan.connect("red", "contraseña")

    print("Conectando...")
    while not wlan.isconnected():
        sleep_ms(1000)

config = wlan.ifconfig()
print(f"Conectado con ip {config[0]}")
```

7. MQTT

MQTT es un protocolo de mensajería estandarizado que distribuye la información a través de publicación y suscripción (*publish*/*subscribe*) a determinado tema (*topic*) y es eficiente para situaciones que se transporta poca información ya que consume poco ancho de banda.

Cada dispositivo se puede comportar como publicador o suscriptor en forma simultánea. La comunicación es manejada por un servidor, también

denominado *broker*. Para dirigir la comunicación, se utilizan *topics*, los cuales, para este curso concreto, pueden seguir la siguiente convención:

- Único dispositivo: /<*parámetro*>
- Múltiples dispositivos: /<*dispositivo*>/<*parámetro*>

A modo de ejemplo, estos *topics* pueden ser:

- /humedad
- /cocina/temperatura

El cliente MQTT no está disponible de forma nativa para MicroPython, por lo que es necesario instalarlo. Para esto, se utiliza el gestor de paquetes `upip` desde la línea de comandos de un dispositivo ESP32 con MicroPython y, a la vez, conectado a una red WiFi con acceso a internet (como se realizó en la sección 6).

```
>>> import upip
>>> upip.install("micropython-umqtt.simple")
>>> upip.install("micropython-umqtt.robust")
```

El cliente *simple* implementa el protocolo MQTT, mientras que el cliente *robust* construye sobre el cliente simple un mecanismo de reconexión en caso de pérdida de conexión al servidor. Para crear el cliente se puede ejecutar el siguiente script:

```
from umqtt.robust import MQTTClient
cliente = MQTTClient("nombre", "servidor", keepalive=30)
print("Conectando con servidor MQTT...")
cliente.connect(clean_session=False)
print("Conectado")
```

De esta manera, se establece una conexión al servidor indicando el nombre del dispositivo. Es importante tener en cuenta que los nombres de los dispositivos son únicos en la red por lo que es recomendable utilizar nombres distintos para cada conexión. Luego, se establece una conexión con `clean_session=False` para garantizar persistencia en el caso que se desconecte y sea necesario reconectar.

7.1. Suscripciones

Una vez conectado el cliente, se puede suscribir a diferentes *topic* de forma tal que estará constantemente esperando la llegada de nuevos mensajes. Una vez que llegue un mensaje nuevo, se ejecuta una función *callback* que permita decidir qué hacer según el mensaje.

Para suscribirse a un *topic* se usa el método `subscribe` y como parámetro un *topic*. Para conectarse a más *topics*, se puede repetir la invocación. Por otro lado, para escuchar todo los *topics* se puede utilizar # o `/jerarquía/#` para escuchar *topics* que pertenezcan a determinada jerarquía.

También hay que definir la función `callback`, la cual es una función de Python con dos parámetros: `topic` y `msg` los cuales llegan en formato *bytes* y pueden ser convertidos a cadena de texto con el método decode().

Finalmente, se crea un bucle infinito dentro del cual se realiza la revisión de mensajes nuevos con el método del cliente `check_msg()`.

A continuación se muestra un script de cómo incorporar esto al código anterior.

```
from umqtt.robust import MQTTClient
from time import sleep_ms

def callback(topic, msg):
    topic = topic.decode()
    msg = msg.decode()

    if topic == "/servidor":
        print(f"Llegó {msg} de {topic}")


cliente = MQTTClient("nombre", "servidor", keepalive=30)
print("Conectando a servidor MQTT...")
cliente.set_callback(callback)
cliente.connect(clean_session=False)
print("Conectado")
cliente.subscribe("#")

while True:
    cliente.check_msg()
    sleep_ms(500)
```

7.2. Publicaciones

Para publicar dentro de un *topic*, no es necesaria ninguna configuración extra. Solo se utiliza el método del cliente `publish` con argumentos *topic* y mensaje.

```
cliente.publish("topic", "mensaje")
```