SunFounder Thales Kit for Raspberry Pi Pico

Release 1.0

Jimmy, SunFounder

Apr 30, 2021

You can check the latest tutorial by visiting the link below:
thales-kit.rtfd.io

CONTENTS

1 Introduction to Raspberry Pi Pico **3**
- 1.1 Features . . . 3
- 1.2 Pico's Pins . . . 4

2 What is Included in This Kit **7**
- 2.1 Components List . . . 9
- 2.2 Components Introduction . . . 10

3 For MicroPython User **39**
- 3.1 Geting Started with MicroPython . . . 39
- 3.2 Thonny Python IDE . . . 42
- 3.3 Your First MicroPython Program . . . 45
- 3.4 Projects . . . 49
- 3.5 MicroPython Basic Syntax . . . 108

4 For Arduino User **135**
- 4.1 Getting Started with Arduino . . . 135
- 4.2 Projects . . . 141

5 FAQ **153**
- 5.1 Q1: NO MicroPython(Raspberry Pi Pico) Interpreter Option on Thonny IDE? . . . 153
- 5.2 Q2: Cannot open Pico code or save code to Pico via Thonny IDE? . . . 154
- 5.3 Q3: Can Raspberry Pi Pico be used on Thonny and Arduino at the same time? . . . 154
- 5.4 Q4: Code upload failed in Arduino IDE? . . . 154

6 Copyright Notice **155**

SunFounder Thales Kit is a basic learning kit based on Raspberry Pi Pico.

The kit contains various types of components, such as displays, sounds, drivers, controllers, and sensors, allows you to learn electronic devices comprehensively.

we have prepared many interesting and practical projects for you, and collected a lot of authoritative related information, just turn on your computer and you can complete programming learning in one stop.

In addition, we provide 2 programming languages: MicroPython and Arduino (C/C++). You can view different tutorials according to your needs

If you want to learn another projects which we don't have, please feel free to send Email and we will update to our online tutorials as soon as possible, any suggestions are welcomed.

Here is the Email: cs@sunfounder.com.

CHAPTER

ONE

INTRODUCTION TO RASPBERRY PI PICO



The Raspberry Pi Pico is a microcontroller board based on the Raspberry Pi RP2040 microcontroller chip.

Whether you want to learn the MicroPython programming language, take the first step in physical computing, or want to build a hardware project, Raspberry Pi Pico – and its amazing community – will support you every step of the way. In the project, it can control anything, from LEDs and buttons to sensors, motors, and even other microcontrollers.

1.1 Features

- 21 mm × 51 mm form factor
- RP2040 microcontroller chip designed by Raspberry Pi in the UK
- Dual-core Arm Cortex-M0+ processor, flexible clock running up to 133 MHz
- 264KB on-chip SRAM
- 2MB on-board QSPI Flash
- 26 multifunction GPIO pins, including 3 analog inputs
- 2 × UART, 2 × SPI controllers, 2 × I2C controllers, 16 × PWM channels
- 1 × USB 1.1 controller and PHY, with host and device support

- 8 × Programmable I/O (PIO) state machines for custom peripheral support
- Supported input power 1.8–5.5V DC
- Operating temperature -20°C to +85°C
- Castellated module allows soldering direct to carrier boards
- Drag-and-drop programming using mass storage over USB
- Low-power sleep and dormant modes
- Accurate on-chip clock
- Temperature sensor
- Accelerated integer and floating-point libraries on-chip

1.2 Pico's Pins



Name	Description	Function
GP0-GP28	General-purpose input/output pins	Act as either input or output and have no fixed purpose of their own
GND	0 volts ground	Several GND pins around Pico to make wiring easier.
RUN	Enables or diables your Pico	Start and stop your Pico from another microcontroller.
GPxx_ADCx	General-purpose input/output or analog input	Used as an analog input as well as a digital input or output – but not both at the same time.
ADC_VREF	Analog-to-digital converter (ADC) voltage reference	A special input pin which sets a reference voltage for any analog inputs.
AGND	Analog-to-digital converter (ADC) 0 volts ground	A special ground connection for use with the ADC_VREF pin.
3V3(O)	3.3 volts power	A source of 3.3V power, the same voltage your Pico runs at internally, generated from the VSYS input.
3v3(E)	Enables or disables the power	Switch on or off the 3V3(O) power, can also switches your Pico off.
VSYS	2-5 volts power	A pin directly connected to your Pico's internal power supply, which cannot be switched off without also switching Pico off.
VBUS	5 volts power	A source of 5V power taken from your Pico's micro USB port, and used to power hardware which needs more than 3.3V.

The best place to find everything you need to get started with your Raspberry Pi Pico is here

Or you can click on the links below:

- Raspberry Pi Pico product brief
- Raspberry Pi Pico datasheet
- Getting started with Raspberry Pi Pico: C/C++ development
- Raspberry Pi Pico C/C++ SDK
- API-level Doxygen documentation for the Raspberry Pi Pico C/C++ SDK
- Raspberry Pi Pico Python SDK
- Raspberry Pi RP2040 datasheet
- Hardware design with RP2040
- Raspberry Pi Pico design files
- Raspberry Pi Pico STEP file

CHAPTER

TWO

WHAT IS INCLUDED IN THIS KIT

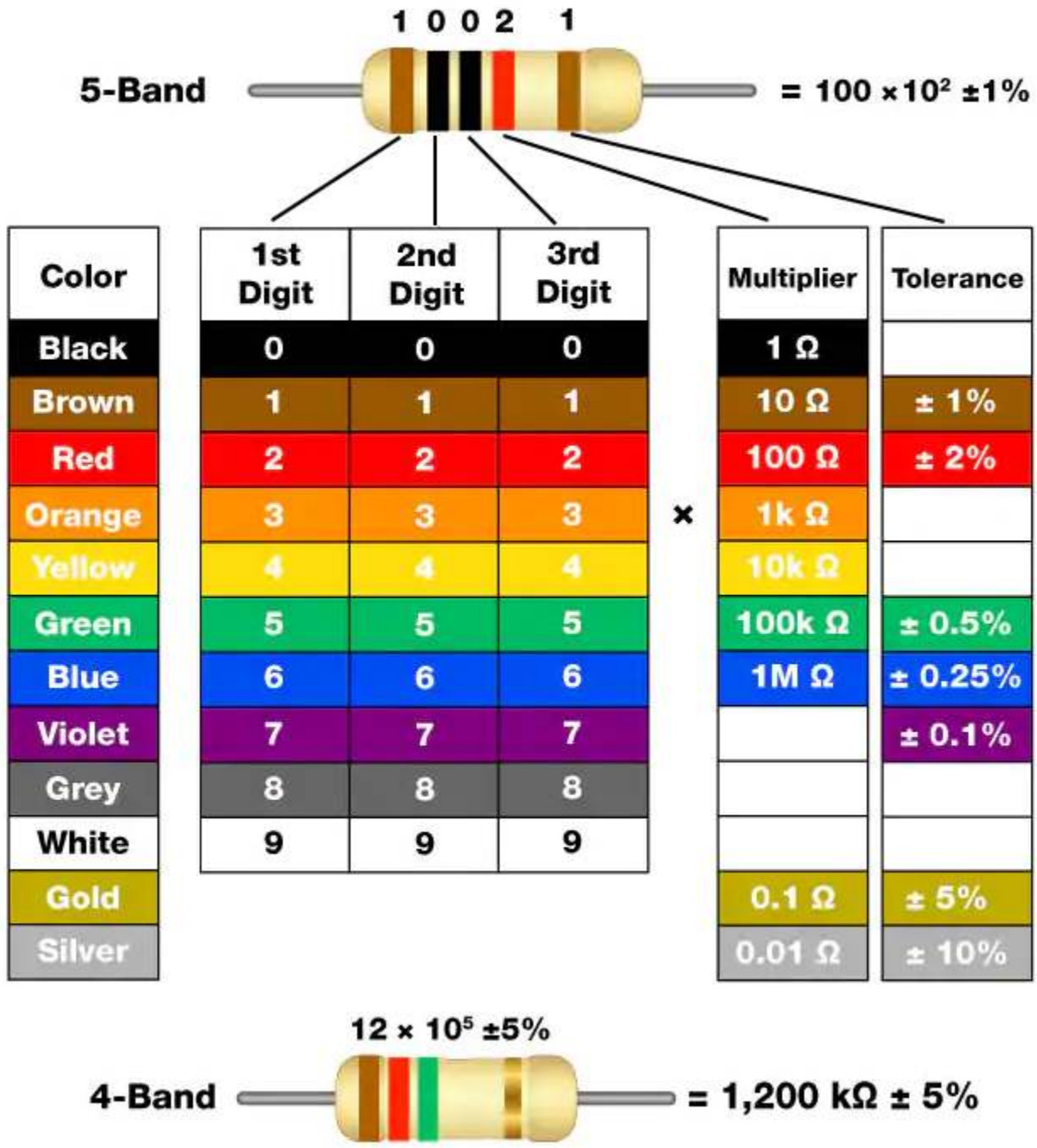
The following is a list of this kit, after you receive the goods, you can check the contents of the list first.

Some of the components on the kit are very small and look the same, the worker may omit or send it wrong when packing. Please do not hesitate to send us the picture of the kit and the names of the missing or wrong components via email.

Here is the Email: cs@sunfounder.com.

2.1 Components List





As shown in the card, each color stands for a number.

Black	Brown	Red	Orange	Yellow	Green	Blue	Violet	Grey	White	Gold	Silver
0	1	2	3	4	5	6	7	8	9	0.1	0.01

The 4- and 5-band resistors are frequently used, on which there are 4 and 5 chromatic bands.

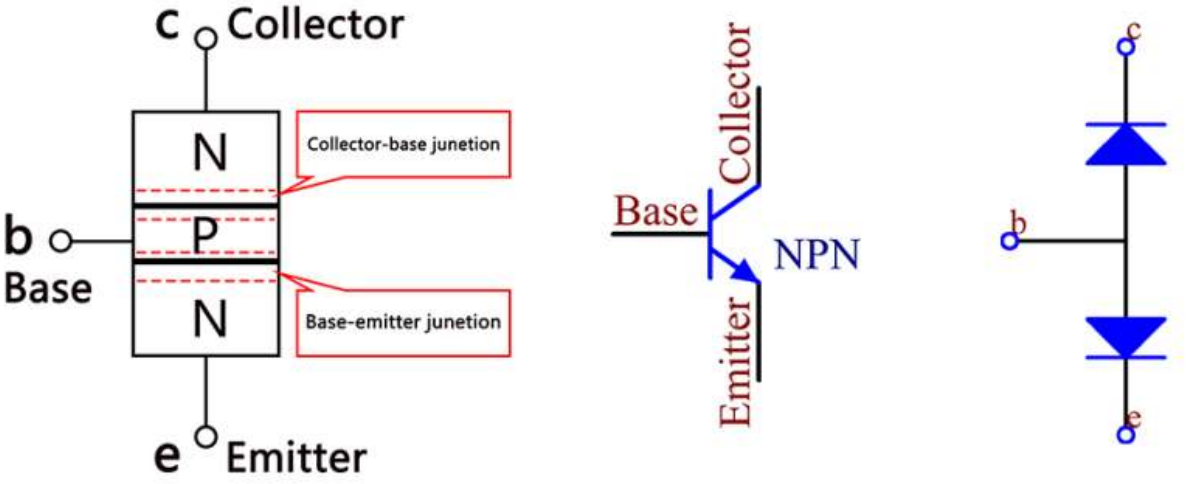
Normally, when you get a resistor, you may find it hard to decide which end to start for reading the color. The tip is that the gap between the 4th and 5th band will be comparatively larger.

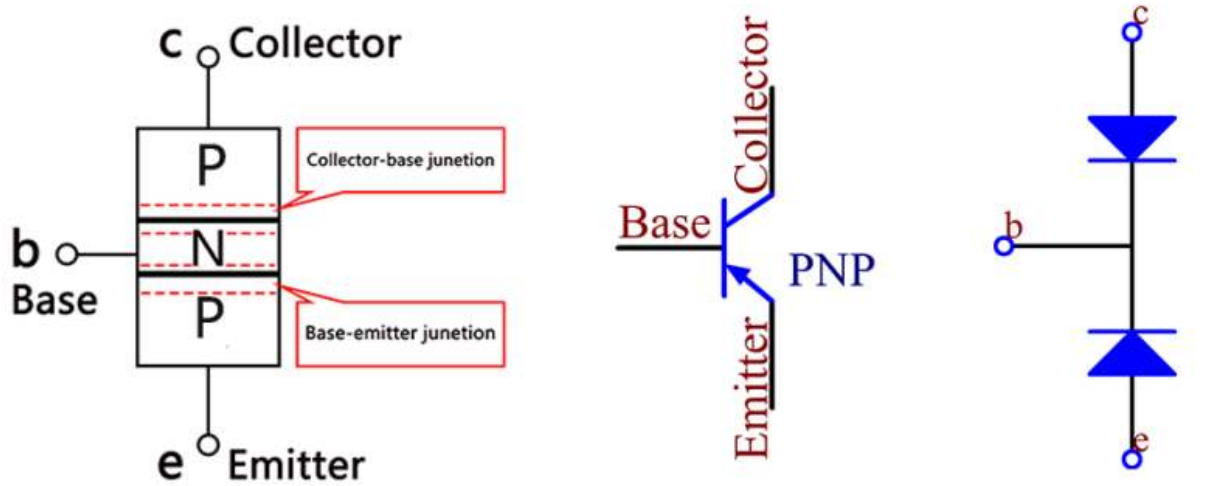
base (b), emitter (e), and collector (c). They form two P-N junctions, namely, the emitter junction and collection junction. The direction of the arrow in the transistor circuit symbol indicates that of the emitter junction.

- P–N junction - Wikipedia

Based on the semiconductor type, transistors can be divided into two groups, the NPN and PNP ones. From the abbreviation, we can tell that the former is made of two N-type semiconductors and one P-type and that the latter is the opposite. See the figure below.

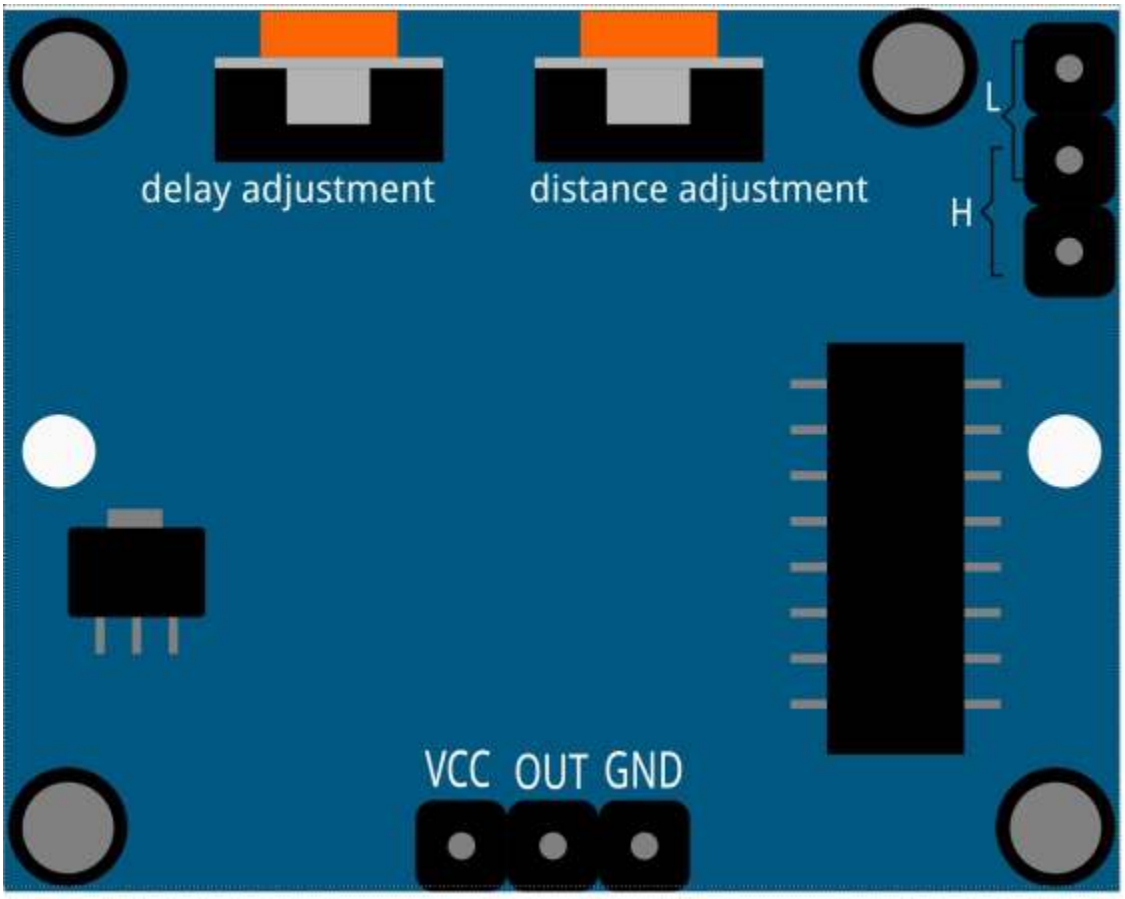
Note: s8550 is PNP transistor and the s8050 is the NPN one, They look very similar, and we need to check carefully to see their labels.





When a High level signal goes through an NPN transistor, it is energized. But a PNP one needs a Low level signal to manage it. Both types of transistor are frequently used for contactless switches, just like in this experiment.

- S8050 Transistor Datasheet
- S8550 Transistor Datasheet



Distance Adjustment

Turning the knob of the distance adjustment potentiometer clockwise, the range of sensing distance increases, and the maximum sensing distance range is about 0-7 meters. If turn it anticlockwise, the range of sensing distance is reduced, and the minimum sensing distance range is about 0-3 meters.

Delay adjustment

Rotate the knob of the delay adjustment potentiometer clockwise, you can also see the sensing delay increasing. The maximum of the sensing delay can reach up to 300s. On the contrary, if rotate it anticlockwise, you can shorten the delay with a minimum of 5s.

Two Trigger Modes

Choosing different modes by using the jumper cap.

- **H**: Repeatable trigger mode, after sensing the human body, the module outputs high level. During the subsequent delay period, if somebody enters the sensing range,the output will keep being the high level.
- **L**Non-repeatable trigger mode, outputs high level when it senses the human body. After the delay, the output will change from high level into low level automatically.

Example

Intruder Alarm

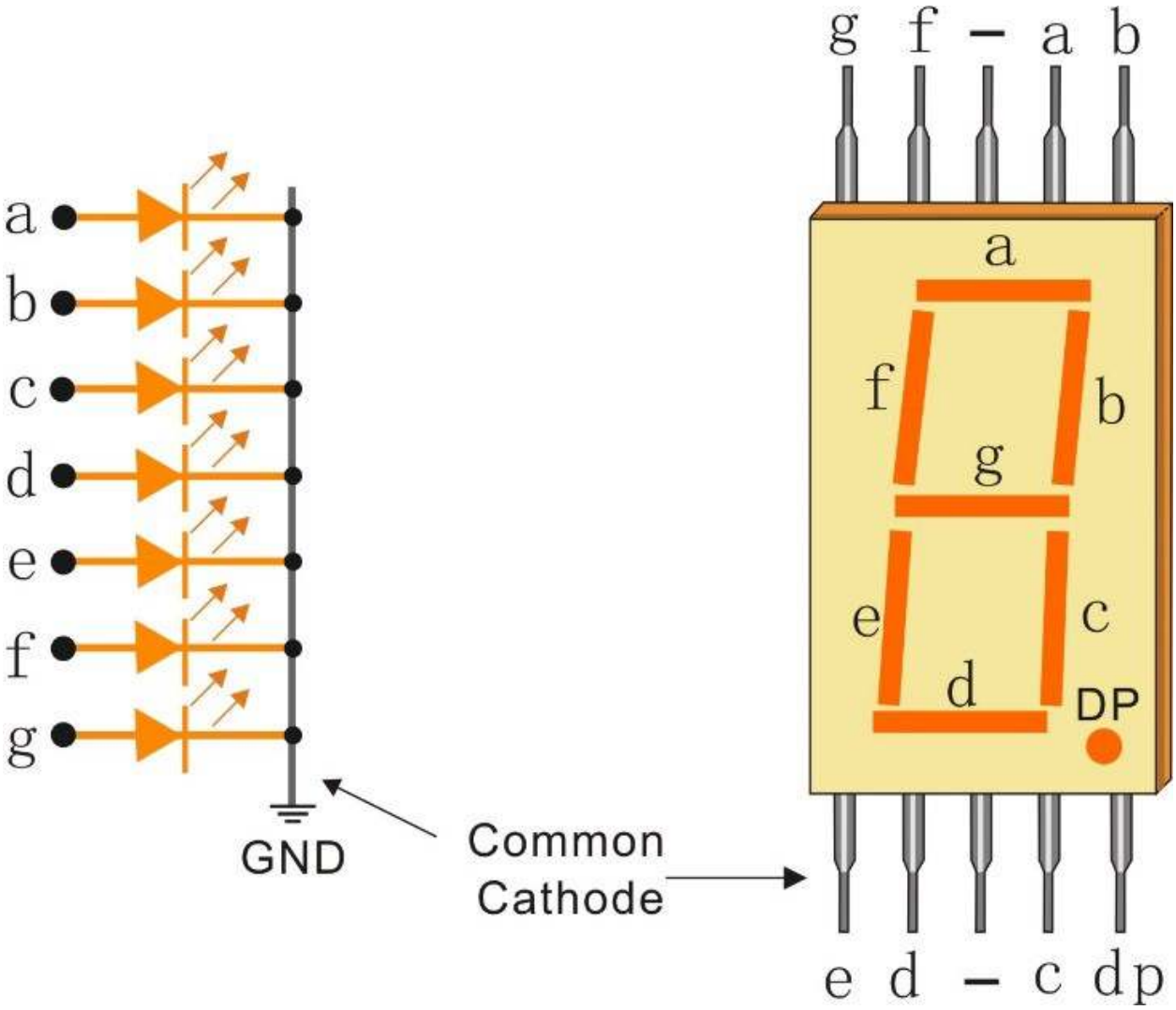
2.2.18 7-segment Display



A 7-segment display is an 8-shaped component which packages 7 LEDs. Each LED is called a segment - when energized, one segment forms part of a numeral to be displayed.

There are two types of pin connection: Common Cathode (CC) and Common Anode (CA). As the name suggests, a CC display has all the cathodes of the 7 LEDs connected when a CA display has all the anodes of the 7 segments connected.

In this kit, we use the Common Cathode 7-segment display, here is the electronic symbol.



Each of the LEDs in the display is given a positional segment with one of its connection pins led out from the rectangular plastic package. These LED pins are labeled from “a” through to “g” representing each individual LED. The other LED pins are connected together forming a common pin. So by forward biasing the appropriate pins of the LED segments in a particular order, some segments will brighten and others stay dim, thus showing the corresponding character on the display.

Display Codes

To help you get to know how 7-segment displays(Common Cathode) display Numbers, we have drawn the following table. Numbers are the number 0-F displayed on the 7-segment display; (DP) GFEDCBA refers to the corresponding LED set to 0 or 1, For example, 00111111 means that DP and G are set to 0, while others are set to 1. Therefore, the number 0 is displayed on the 7-segment display, while HEX Code corresponds to hexadecimal number.

CHAPTER

THREE

FOR MICROPYTHON USER

This section contains the history of MicroPython, installing MicroPython in Pico, the basic syntax of MicroPython and a dozen interesting and practical projects to help you learn MicroPython quickly.

We recommend that you read the chapters in order.

3.1 Geting Started with MicroPython

MicroPython is a full Python compiler and runtime that runs on the microcontroller's hardware like Raspberry Pi Pico. The user is presented with an interactive prompt (the REPL) to excute supported commands immediately. Included are a selection of core Python libraries; MicroPython includes modules which give the programmer access to low-level hardware.

- Reference: MicroPython - Wikipedia

3.1.1 The Story Starts Here

Everything changed in 2013 when Damien George launched a crowdfunding campaign (Kickstarter).

Damien was an undergraduate student at Cambridge University and an avid robotics programmer. He wanted to reduce the world of Python from a gigabyte machine to a kilobyte. His Kickstarter campaign was to support his development while he turned his proof of concept into a finished implementation.

MicroPython is supported by a diverse Pythonista community that has a keen interest in seeing the project succeed.

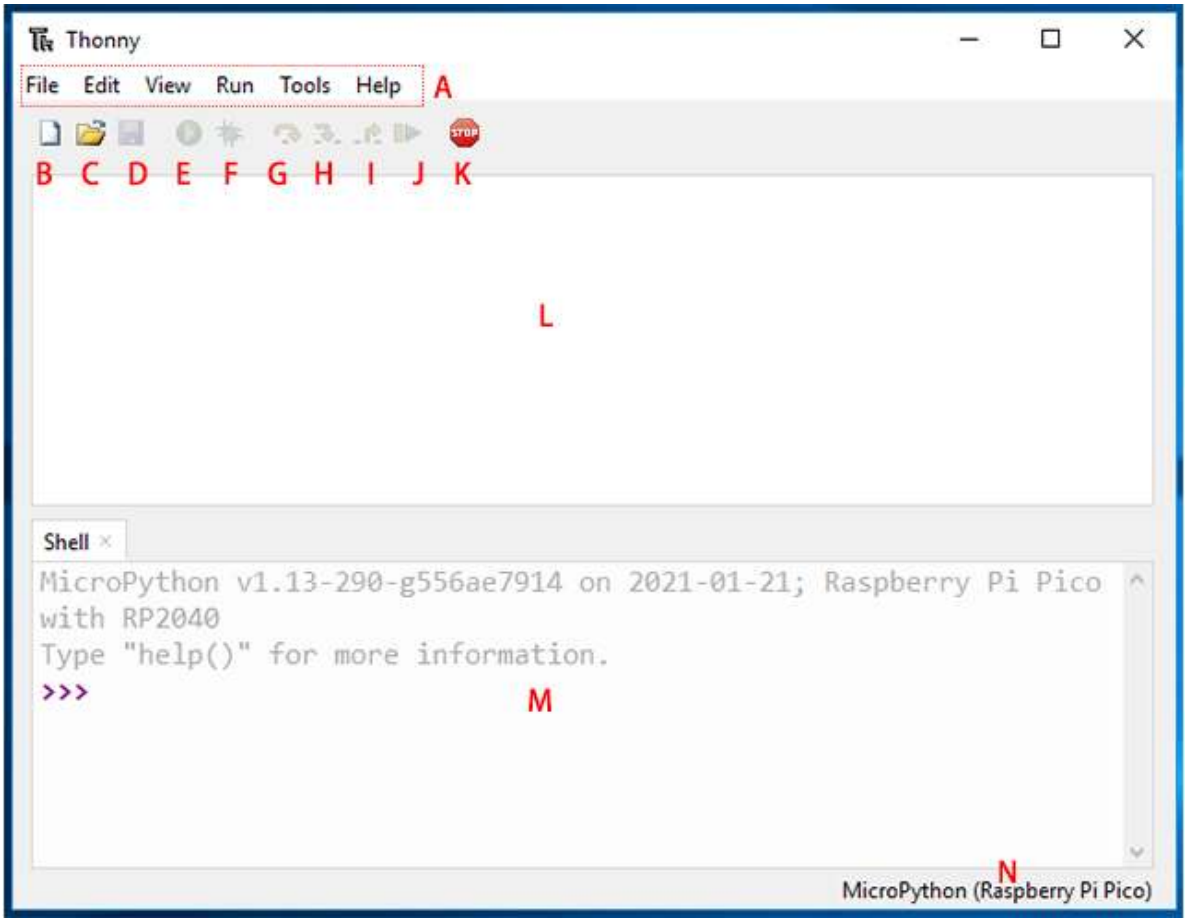
In addition to testing and supporting the code base, the developers provided tutorials, code libraries, and hardware ports, making the project far more than Damien could have done alone.

- Reference: realpython

3.1.2 Why MicroPython

Although the original Kickstarter campaign released MicroPython as a development board "pyboard" with STM32F4, MicroPython supports many ARM-based product architectures. The mainline supported ports are ARM Cortex-M (many STM32 boards, TI CC3200/WiPy, Teensy boards, Nordic nRF series, SAMD21 and SAMD51), ESP8266, ESP32, 16bit PIC, Unix, Windows, Zephyr and JavaScript. Second, MicroPython allows for fast feedback. This is because you can use REPL to enter commands interactively and get responses. You can even tweak code and run it immediately instead of traversing the code-compile-upload-execute cycle.

While Python has the same advantages, for some Microcontroller boards like the Raspberry Pi Pico, they are small, simple and have little memory to run the Python language at all. That's why MicroPython has evolved, keeping the main Python features and adding a bunch of new ones to work with these Microcontroller boards.

- 

- **A**: The menu bar that contains the file New, Save, Edit, View, Run, Debug, etc.
- **B**: This paper icon allows you to create a new file.
- **C**: The folder icon allows you to open files that already exist in your computer or Raspberry Pi Pico, if your Pico is already plugged into your computer.
- **D**: Click on the floppy disk icon to save the code. Similarly, you can choose whether to save the code to your computer or to the Raspberry Pi Pico.
- **E**: The play icon allows you to run the code. If you have not saved the code, save the code before it can run.
- **F**: The Debug icon allows you to debug your code. Inevitably, you will encounter errors when writing code. Errors can take many forms, sometimes using incorrect syntax, sometimes incorrect logic. Debugging is the tool for finding and investigating errors.

Note: The Debug tool cannot be used when MicroPython (Raspberry Pi Pico) is selected as the interpreter.

If you want to debug your code, you need to select the interpreter as the default interpreter and save as to your computer after debugging.

Finally, select the MicroPython (Raspberry Pi Pico) interpreter again, click the save as button, and re-save the debugged code to your Raspberry Pi Pico.

- The G, H and I arrow icons allow you to run the program step by step, but can only be started after clicking on the Degug icon. As you click on each arrow, you will notice that the yellow highlighted bar will indicate the line or section of Python that is currently evaluating.

  - **G**: Take a big step, which means jumping to the next line or block of code.
  - **H**: Take a small step means expressing each component in depth.
  - **I**: Exit out of the debugger.
- **J**Click it to return from debug mode to play mode.
- **K**Use the stop icon to stop running code.
- **L**Script Area, where you can write your Python code.
- **M**Python Shell, where you can type a single command, and when you press the Enter key, the single command will run and provide information about the running program. This is also known as REPL, which means "Read, Evaluate, Print, and Loop."
- **N**Interpreter, where the current version of Python used to run your program is displayed, can be changed manually to another version by clicking on it.

Note: NO MicroPython(Raspberry Pi Pico) Interpreter Option ?

- Check that your Pico is plugged into your computer via a USB cable.
- The Raspberry Pi Pico interpreter is only available in version 3.3.3 or higher version of Thonny. If you are running an older version, please update.

3.3 Your First MicroPython Program

In MicroPython, there are two options/methods for running code:

- *Interactive Mode*
- *Script Mode*

Interactive mode involves running your code directly from the Python shell, which can be accessed from the operating system's terminal. In script mode, you must create a file, save it with .py extension, and then run your code. Interactive mode is suitable when you are running a few lines of code. Script mode is recommended when you need to create large codes and save them for next time.

3.3.1 Interactive Mode

Interactive mode, also known as the REPL provides us with a quick way to run blocks or a single line of MicroPython code via the Python shell.

Click the Python Shell area, type the following command after >>>, and then Enter.

```
print("Hello, World!")
```

How it works?

The onboard LED is connected to the GP25 pin, if you carefully observe the Pico pinout, you will find that GP25 is one of the hidden pins, which means that we cannot use this pin (even if GP25 is used in exactly the same way as other pins). The advantage of this design is that even if you don't connect any external components, you can still have an OUTPUT to test the program.

The machine library is required to use GPIO.

```
import machine
```

This library contains all the instructions needed to communicate between MicroPython and Pico. Without this line of code, we will not be able to control any GPIOs (Of course including GP25).

The next thing to notice is this line:

```
led_onboard = machine.Pin(25, machine.Pin.OUT)
```

An object named `led_onboard` is defined here. Technically, it can be any name, it can be x, y, banana, Micheal_Jackson, or any character, but it is best to use a name that describes the purpose to ensure that the program is easy to read.

The second part of this line (the part after the equal sign) calls the Pin function in the machine library. It is used to tell Pico's GPIO pins what to do. The Pin function has two parameters: the first parameter (25) means the pin you want to set; the second parameter (machine.Pin.OUT) tells that the pin should be used as an output instead of an input.

The above code has "set" the pin, but it will not light up the LED. To do this, we also need to "use" the pin.

```
led_onboard.value(1)
```

We have set up the GP25 pin before and named it led_onboard. The function of this statement is to set the value of `led_onboard` to 1 to turn the on-board LED on.

All in all, to use GPIO, these steps are necessary:

- **import machine library**: This is necessary, and it is only executed once in the entire program.
- **Set GPIO**: Each pin should be set before use.
- **Use**: Assign a value to the pin, each assignment will change the working state of the pin.

If we follow the above steps to write an example, then you will get code like this:

```
import machine
led_onboard = machine.Pin(25, machine.Pin.OUT)
led_onboard.value(1)
```

Run it and you will be able to light up the onboard LED.

Next, we try to add the "extinguished" statement:

```
import machine
led_onboard = machine.Pin(25, machine.Pin.OUT)
led_onboard.value(1)
led_onboard.value(0)
```

According to the code line, this program will make the onboard LED turn on first and then turn off. But when you use it, you will find that this is not the case. The onboard LED never seems to light up. This is because the execution speed between the two lines is very fast, much faster than the reaction time of the human eye. The moment the onboard LED lights up is not enough to make us perceive the light. To fix that, we need to slow down the program.

Insert the following statement into the second line of the program:

```
import utime
```

Like machine, the `utime` library is introduced here, which handles all time-related things, including the delay we need to use. Let's insert a delay sentence between `led_onboard.value(1)` and `led_onboard.value(0)`, let them be separated by 2 seconds:

```
utime.sleep(2)
```

Now, the code should look like this. Run it, we will be able to see that the onboard LED turns on first and then turns off:

```
import machine
import utime
led_onboard = machine.Pin(25, machine.Pin.OUT)
led_onboard.value(1)
utime.sleep(2)
led_onboard.value(0)
```

Finally, we should make the LED blink. Create a loop, rewrite the program, and it will be what you saw at the beginning of this chapter.

```
import machine
import utime

led_onboard = machine.Pin(25, machine.Pin.OUT)
while True:
    led_onboard.value(1)
    utime.sleep(2)
    led_onboard.value(0)
    utime.sleep(2)
```

What More?

Usually, the library will have a corresponding API (Application Programming Interface) file. This is a concise reference manual that contains all the information needed to use this library, detailed introduction to functions, classes, return types, parameters, etc., and even comes with a tutorial.

In this article, we used MicroPython's `machine` and `utime` libraries, we can find more ways to use them here.

- machine.Pin
- utime

The following is also an example of making the LED blink, please try to read the API file to understand it!

```
import machine
import utime

led_onboard = machine.Pin(25, machine.Pin.OUT)
while True:
    led_onboard.toggle()
    utime.sleep(1)
```

3.4.2 Hello, Breadboard!

To use extended electronic components, a solderless breadboard will be the most powerful partner for novice users.

The breadboard is a rectangular plastic plate with a bunch of small holes in it. These holes allow us to easily insert electronic components and build electronic circuits. The breadboard does not permanently fix the electronic components, which makes it easy for us to repair the circuit and start over when we make a mistake.

Note: We can use the breadboard without using special tools. But many electronic components are very small. A pair of tweezers can help us pick up small parts better.

There are many detailed breadboard knowledge on the Internet, let us use it wisely.

- How to Use a Breadboard - Science Buddies
- What is a BREADBOARD? - Makezine

For breadboards, what you need to know clearly is:

1. Each group of half rows inside the breadboard (such as column A-E in row 1 or column F-J in row 3) is connected. This means that when an electrical signal flows in from A1, it can flow from B1, C1, D1, E1, but not from F1 or A2.
2. Both sides of the breadboard are usually used as power buses, and the holes in each column (about 50 holes) are connected. Generally speaking, the hole near the red wire is used to connect the positive power supply, and the hole near the blue wire is used to connect the negative power supply.
3. When building a circuit, the current flows from the positive pole and must first flow through the consumer before it can flow into the negative pole. Otherwise, a short circuit may occur.

Now we should have a general impression of the breadboard circuit, why not try to build a “Hello, LED!” expansion circuit?

Code

The method of controlling the extended LED is the same as the method of controlling the on-board LED, the only difference is that the operating pin is changed to 15.

```
import machine
import utime

led = machine.Pin(15, machine.Pin.OUT)
while True:
    led.toggle()
    utime.sleep(1)
```

`toggle()` can switch the pin between high level and low level, and you can see the LED blinking.

Also see reference here:

- machine.Pin

3.4.3 Reading Button Value

From the name of GPIO (General-purpose input/output), we can see that these pins have both input and output functions. In the previous two lessons, we used the output function, in this chapter we will use the input function to input read the button value.

> Buttons require pull-up resistors or pull-down resistors. If there is no pull-up or pull-down resistor, the main controller may receive a 'noisy' signal which can trigger even when you're not pushing the button.

6. Connect the negative power bus of the breadboard to Pico's GND.

Code

When the button is pressed, the current will flow from 3V3 through the button to GP14, in other words, GP14 will read a high-level signal '1'; otherwise, it will read a low-level signal '0'.

```
import machine
import utime
button = machine.Pin(14, machine.Pin.IN)
while True:
    if button.value() == 1:
        print("You pressed the button!")
        utime.sleep(1)
```

Pull-up Working Mode

Next is the wiring and code when the button in the pull-up working mode, please try it.

How it works?

In the previous lesson, we simply read the button value. This time, we tried a flexible way of using buttons: interrupt requests, or IRQs.

For example, you are reading a book page by page, as if a program is executing a thread. At this time, someone came to you to ask a question and interrupted your reading. Then the person is executing the interrupt request: asking you to stop what you are doing, answer his questions, and then let you return to reading the book after the end.

MicroPython interrupt request also works in the same way, it allows certain operations to interrupt the main program. It is achieved through the following two statements (the highlighted 2 lines):

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN)

def button_press(pin):
    button.irq(handler=None)
    rection_time = utime.ticks_diff(utime.ticks_ms(), timer_light_off)
    print("Your reaction time was " + str(rection_time) + " milliseconds!")

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
timer_light_off = utime.ticks_ms()
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_press)
```

Here, a callback function (button_press) is first defined, which is called an interrupt handler. It will be executed when an interrupt request is triggered. Then, set up an interrupt request in the main program, it contains two parts: `trigger` and `handler`.

- In this program, the `trigger` is `IRQ_RISING`, which means that the value of the pin rises from low level to high level (That is, pressing the button).
- `handler` is the callback function `button_press` we defined before.

In this example, you will find a statement `button.irq(handler=None)` in the callback function, which is equivalent to canceling the interrupt.

In order to better understand the interrupt request, we change the above code to the following (Use the same circuit):

```
import machine
import utime

button = machine.Pin(14, machine.Pin.IN)
count = 0

def button_press(pin):
    print("You press the button!")
    utime.sleep(1)

button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_press)

while True:
    count+=1
    print(count)
    utime.sleep(1)
```

When the program runs, it will start counting and print the numbers in the shell. When we press the button, it will stop counting and enter the callback function to print "You press the button!".

Go back to the original example. We need to make the LED turn off in a random time of 5 to 10 seconds, which is achieved by the following two lines:

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN)

def button_press(pin):
    button.irq(handler=None)
    rection_time = utime.ticks_diff(utime.ticks_ms(), timer_light_off)
    print("Your reaction time was " + str(rection_time) + " milliseconds!")

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
timer_light_off = utime.ticks_ms()
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_press)
```

The `urandom` library is loaded here. Use the `urandom.uniform(5,10)` function to generate a random number, the 'uniform' part referring to a uniform distribution between those two numbers.

If needed, try running the following example of random number generation:

```
import machine
import utime
import urandom

while True:
    print(urandom.uniform(1, 20))
    utime.sleep(1)
```

The last two statements you need to understand are `utime.ticks_ms()` and `utime.ticks_diff()`.

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN)

def button_press(pin):
    button.irq(handler=None)
    rection_time = utime.ticks_diff(utime.ticks_ms(), timer_light_off)
    print("Your reaction time was " + str(rection_time) + " milliseconds!")

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
timer_light_off = utime.ticks_ms()
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_press)
```

- The `utime.ticks_ms()` function will output the number of milliseconds that have passed since the `utime` library started counting and store it in the variable `timer_light_off`.

5. Insert the Button, connect one side of it to the GP16 pin, then use a 10K resistor to connect the same side and the negative bus. Do not forget to connect the other side to the positive power bus.

Note: The color ring of 220 ohm resistor is red, red, black, black and brown.

The color ring of the 10k ohm resistor is brown, black, black, red and brown.

Code

When the program is started, the traffic light will switch in the order of red for 5 seconds, yellow for 2 seconds, green for 5 seconds, and yellow for 2 seconds. If we (pedestrians) press the button, the red LED will be extended to 15 seconds, which will give us more time to cross the road.

```
import machine
import utime
import _thread

led_red = machine.Pin(15, machine.Pin.OUT)
led_yellow = machine.Pin(14, machine.Pin.OUT)
led_green = machine.Pin(13, machine.Pin.OUT)
button = machine.Pin(16, machine.Pin.IN)

global button_status
button_status = 0

def button_thread():
    global button_status
    while True:
        if button.value() == 1:
            button_status = 1

_thread.start_new_thread(button_thread, ())

while True:
    if button_status == 1:
        led_red.value(1)
        utime.sleep(10)
    global button_status
    button_status = 0

    led_red.value(1)
    utime.sleep(5)
    led_red.value(0)

    led_yellow.value(1)
    utime.sleep(2)
    led_yellow.value(0)

    led_green.value(1)
    utime.sleep(5)
    led_green.value(0)

    led_yellow.value(1)
    utime.sleep(2)
    led_yellow.value(0)
```

How it works?

In the previous lessons, we have successfully made the LED blink. In other words, it is very simple for us to write a code that makes the traffic light cycle color. What we need to do is to add a judgment on the state of the button. But if we directly write the statement that reads the button value into the main program, we will find that it doesn't fit anywhere. Even if it is written in, it is difficult for us to read this value. This is because the program is stuck when executing `utime.sleep()`, and the statement to read the button value is not executed at this time.

Of course, we can read the button value through the IRQ in the previous lesson. But this time we take another approach-multithreading.

Multi-threading can be simply understood as dividing a thing into multiple parts, which are executed by different people (or processors). Just like when the chef is frying the steak, the assistant chef makes the sauce so that the newly prepared sauce can be poured on the properly prepared steak to make the best cooking.

Look at these lines:

```
import machine
import utime
import _thread

led_red = machine.Pin(15, machine.Pin.OUT)
led_yellow = machine.Pin(14, machine.Pin.OUT)
led_green = machine.Pin(13, machine.Pin.OUT)
button = machine.Pin(16, machine.Pin.IN)

global button_status
button_status = 0

def button_thread():
    global button_status
    while True:
        if button.value() == 1:
            button_status = 1

_thread.start_new_thread(button_thread, ())

while True:
    if button_status == 1:
        led_red.value(1)
        utime.sleep(10)
    global button_status
    button_status = 0

    led_red.value(1)
    utime.sleep(5)
    led_red.value(0)

    led_yellow.value(1)
    utime.sleep(2)
    led_yellow.value(0)

    led_green.value(1)
    utime.sleep(5)
    led_green.value(0)

    led_yellow.value(1)
    utime.sleep(2)
    led_yellow.value(0)
```

Here, the `_thread` library is imported first. This module implements multithreading support. Then define a thread `button_thread()`, which is independent of the main thread. It is used here to read the state of the button. Finally use `_thread.start_new_thread(button_thread, ())` to start the thread.

The following sample code can help you better understand multithreading:

```
import machine
import utime
import _thread

led_red = machine.Pin(15, machine.Pin.OUT)
led_yellow = machine.Pin(14, machine.Pin.OUT)
button = machine.Pin(16, machine.Pin.IN)

def led_yellow_thread():
    while True:
        led_yellow.toggle()
        utime.sleep(2)

_thread.start_new_thread(led_yellow_thread, ())

while True:
    button_status = button.value()
    if button_status == 1:
        led_red.value(1)
    elif button_status == 0:
        led_red.value(0)
```

In the main thread, the button is used to control the red LED on and off. In the new thread (led_yellow_thread()), the yellow LED will change every 2 seconds. The two threads work independently of each other.

Let's go back to the traffic signal program. We let the main thread change the light and let the new thread read the button value. However, the threads are independent of each other, and we need a way for the new thread to pass information to the main thread, which requires the use of global variable.

The variables we have used before are all local variables, acting only in a certain part of the program (Variables declared in the main function cannot be used in sub-functions, and variables declared in the main thread cannot be used in the new thread). The global variable can be used anywhere, we change it in one thread, and the other can get its updated value.

Global variables are in these places:

```
import machine
import utime
import _thread

led_red = machine.Pin(15, machine.Pin.OUT)
led_yellow = machine.Pin(14, machine.Pin.OUT)
led_green = machine.Pin(13, machine.Pin.OUT)
button = machine.Pin(16, machine.Pin.IN)

global button_status
button_status = 0

def button_thread():
    global button_status
    while True:
        if button.value() == 1:
            button_status = 1
```

(continues on next page)

(continued from previous page)

```
_thread.start_new_thread(button_thread, ())

while True:
    if button_status == 1:
        led_red.value(1)
        utime.sleep(10)
    global button_status
    button_status = 0

    led_red.value(1)
    utime.sleep(5)
    led_red.value(0)

    led_yellow.value(1)
    utime.sleep(2)
    led_yellow.value(0)

    led_green.value(1)
    utime.sleep(5)
    led_green.value(0)

    led_yellow.value(1)
    utime.sleep(2)
    led_yellow.value(0)
```

- When the program is just running, `button_status` is assigned a value of 0, which means that the button has not been pressed.
- In the new thread–button_thread, when the program detects that the button is pressed, `button_status` is assigned the value 1.
- At the beginning of each cycle, it will detect whether the button has been pressed, if the button is pressed (button_status == 1), the red light will be on for 10 seconds. Then `button_status` switch to 0, and wait for the next button press.

The function of `global button_status` is to tell the program that we are going to modify the value of `button_status`, but if we just want to read the variable value, this line is not needed.

3.4.6 Two Kinds of Transistors

Transistor is a semiconductor device that controls a large current through a small current. Its function is to amplify weak signals into larger amplitude signals, and can also be used as a non-contact switch. It is the core component of electronic circuits.

This sounds a bit complicated. In simple words, some components use high-current (such as Buzzer). If the power is directly supplied from the GPIO of the microcontroller, the power may be insufficient or the microcontroller may be damaged. Then, the transistor has played a "dam" role here. Transistor receives the weak electrical signal from the GPIO pin to control the turn-on and turn-off of the large current (from VCC to GND). In this way, high-current components can be driven and the microcontroller can be protected.

Transistor

This kit is equipped with two types of transistors, S8550 and S8050, the former is PNP and the latter is NPN. They look very similar, and we need to check carefully to see their labels. When a High level signal goes through an NPN transistor, it is energized. But a PNP one needs a Low level signal to manage it. Both types of transistor are frequently used for contactless switches, just like in this experiment.

3.4.7 Intruder Alarm

In the previous chapters, we used simple electronic components (such as LED, button). This time we will use the sensor module - PIR.

Passive infrared sensor (PIR sensor) is a common sensor that can measure infrared (IR) light emitted by objects in its field of view. Simply put, it will receive infrared radiation emitted from the body, thereby detecting the movement of people and other animals. More specifically, it tells the main control board that someone has entered your room.

PIR Motion Sensor

Now, let's use PIR and active buzzer to build an Intruder Alarm.

Wiring

Two types of buzzers are included in the kit. We need to use active buzzer. Turn them around, the sealed back (not the exposed PCB) is the one we want.



The buzzer needs to use a transistor when working, here we use S8050.

(continued from previous page)

```
pir_sensor = machine.Pin(14, machine.Pin.IN)
buzzer = machine.Pin(15, machine.Pin.OUT)

def motion_detected(pin):
    for i in range(50):
        buzzer.toggle()
        utime.sleep_ms(100)

pir_sensor.irq(trigger=machine.Pin.IRQ_RISING, handler=motion_detected)
print("Intruder Alarm Start!")
```

What more?

PIR is a very sensitive sensor. In order to adapt it to the environment of use, it needs to be adjusted. Let the side with the 2 potentiometers facing you, turn both potentiometers counterclockwise to the end and insert the jumper cap on the pin with L and the middle pin.

Copy the following code into Thonny and run it, let us analyze its adjustment method along with the experimental results.

```
import machine
import utime

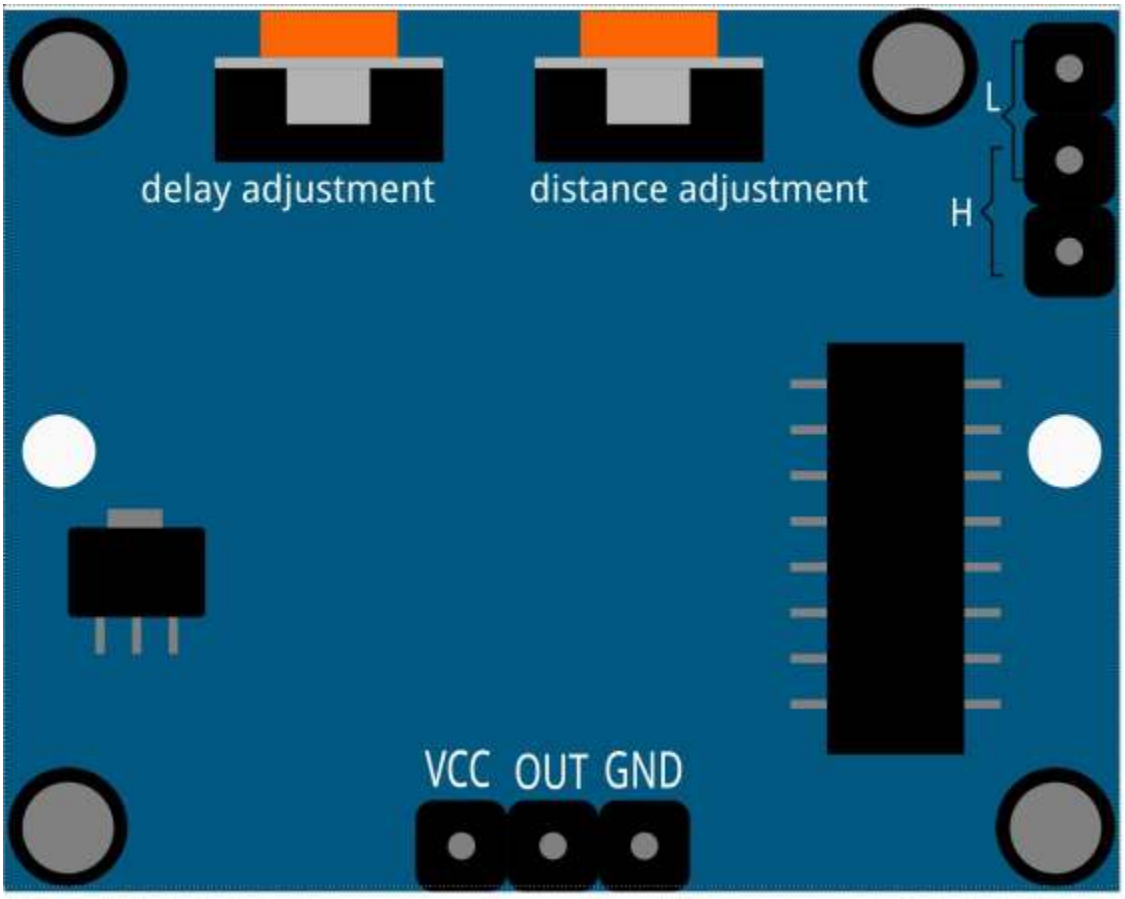
pir_sensor = machine.Pin(14, machine.Pin.IN)

global timer_delay
timer_delay = utime.ticks_ms()
print("start")

def pir_in_high_level(pin):
    global timer_delay
    pir_sensor.irq(trigger=machine.Pin.IRQ_FALLING, handler=pir_in_low_level)
    intervals = utime.ticks_diff(utime.ticks_ms(), timer_delay)
    timer_delay = utime.ticks_ms()
    print("the dormancy duration is " + str(intervals) + "ms")

def pir_in_low_level(pin):
    global timer_delay
    pir_sensor.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_in_high_level)
    intervals2 = utime.ticks_diff(utime.ticks_ms(), timer_delay)
    timer_delay = utime.ticks_ms()
    print("the duration of work is " + str(intervals2) + "ms")

pir_sensor.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_in_high_level)
```



1. Trigger Mode

   Let's take a look at the pins with jumper cap at the corner. It allows PIR to enter Repeatable trigger mode or Non-repeatable trigger mode

   At present, our jumper cap connects the middle Pin and L Pin, which makes the PIR in non-repeatable trigger mode. In this mode, when the PIR detects the movement of the organism, it will send a high-level signal for about 2.8 seconds to the main control board. We can see in the printed data that the duration of work will always be around 2800ms.

   Next, we modify the position of the lower jumper cap and connect it to the middle Pin and H Pin to make the PIR in repeatable trigger mode. In this mode, when the PIR detects the movement of the organism (note that it is movement, not static in front of the sensor), as long as the organism keeps moving within the detection range, the PIR will continue to send a high-level signal to the main control board. We can see in the printed data that the duration of work is an uncertain value.

2. Delay Adjustment

   The potentiometer on the left is used to adjust the interval between two jobs.

   At present, we screw it counterclockwise to the end, which makes the PIR need to enter a sleep time of about 5 seconds after finishing sending the high level work. During this time, the PIR will no longer detect the infrared radiation in the target area. We can see in the printed data that the dormancy duration is always no less than 5000ms.

   If we turn the potentiometer clockwise, the sleep time will also increase. When it is turned clockwise to the end, the sleep time will be as high as 300s.

3. Distance Adjustment

```
import machine
import utime

led = machine.PWM(machine.Pin(15))
led.freq(1000)

for brightness in range(0,65535,50):
    led.duty_u16(brightness)
    utime.sleep_ms(10)
led.duty_u16(0)
```

How it works?

Here, we change the brightness of the LED by changing the duty cycle of the GP15's PWM output. Let's take a look at these lines.

```
import machine
import utime

led = machine.PWM(machine.Pin(15))
led.freq(1000)

for brightness in range(0,65535,50):
    led.duty_u16(brightness)
    utime.sleep_ms(10)
led.duty_u16(0)
```

- `led = machine.PWM(machine.Pin(15))` sets the GP15 pin as PWM output.
- The line `led.freq(1000)` is used to set the PWM frequency, here it is set to 1000Hz, which means 1ms (1/1000) is a cycle. The PWM frequency can be adjusted, for example, the steering wheel needs to work at 50Hz, the passive buzzer can change the tone by changing the PWM frequency. However, there is no limit when using LEDs alone, we set it to 1000Hz.
- The `led.duty_u16()` line is used to set the duty cycle, which is a 16-bit interger(2^16=65536). When we assign a 0 to this function, the duty cycle is 0%, and each cycle has 0% of the time to output a high level, in other words, turn off all pulses. When the value is 65535, the duty cycle is 100%, that is, the complete pulse is turned on, and the result is equal to '1' as a digital output. If it is 32768, it will turn on half a pulse, and the brightness of the LED will be half of that when it is fully turned on.

3.4.9 Colorful Light

As we know, light can be superimposed. For example, mix blue light and green light give cyan light, red light and green light give yellow light. This is called "The additive method of color mixing".

- Additive color - Wikipedia

Based on this method, we can use the three primary colors to mix the visible light of any color according to different specific gravity. For example, orange can be produced by more red and less green.

In this chapter, we will use RGB LED to explore the mystery of additive color mixing!

RGB LED is equivalent to encapsulating Red LED, Green LED, Blue LED under one lamp cap, and the three LEDs share one cathode pin. Since the electric signal is provided for each anode pin, the light of the corresponding color can be displayed. By changing the electrical signal intensity of each anode, it can be made to produce various colors.

1. Connect the GND pin of the Pico to the negative power bus of the breadboard.
2. Insert the RGB LED into the breadboard so that its four pins are in different rows.
3. Connect the red lead to the GP13 pin via a 330 resistor. When using the same power supply intensity, the Red LED will be brighter than the other two, and a slightly larger resistor needs to be used to reduce its brightness.
4. Connect the Green lead to the GP14 pin via a 220 resistor.
5. Connect the Blue lead to the GP15 pin via a 220 resistor.
6. Connect the GND lead to the negative power bus.
7. Connect the negative power bus to Pico's GND.

Note:

- The color ring of the 220 resistor is red, red, black, black and brown.
- The color ring of the 330 resistor is orange, orange, black, black and brown.

Code

Here, we can choose our favorite color in drawing software (such as paint) and display it with RGB LED.

```
import machine
import utime

red = machine.PWM(machine.Pin(13))
green = machine.PWM(machine.Pin(14))
blue = machine.PWM(machine.Pin(15))
red.freq(1000)
green.freq(1000)
blue.freq(1000)

def interval_mapping(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def color_to_duty(rgb_value):
    rgb_value = int(interval_mapping(rgb_value,0,255,0,65535))
    return rgb_value

def color_set(red_value,green_value,blue_value):
    red.duty_u16(color_to_duty(red_value))
    green.duty_u16(color_to_duty(green_value))
    blue.duty_u16(color_to_duty(blue_value))

color_set(255,128,0)
```

Note: The color ring of the 1k ohm resistor is brown, black, black, brown and brown.

Code

```
import machine
import utime

buzzer = machine.PWM(machine.Pin(15))

def tone(pin,frequency,duration):
    pin.freq(frequency)
    pin.duty_u16(30000)
    utime.sleep_ms(duration)
    pin.duty_u16(0)

tone(buzzer,440,250)
utime.sleep_ms(500)
tone(buzzer,494,250)
utime.sleep_ms(500)
tone(buzzer,523,250)
```

How it works?

If the passive buzzer given a digital signal, it can only keep pushing the diaphragm without producing sound.

Therefore, we use the `tone()` function to generate the PWM signal to make the passive buzzer sound.

This function has three parameters:

- **pin**, the GPIO pin that controls the buzzer.
- **frequency**, the pitch of the buzzer is determined by the frequency, the higher the frequency, the higher the pitch.
- **Duration**, the duration of the tone.

We use the `duty_u16()` function to set the duty cycle to 30000(about 50%). It can be other numbers, and it only needs to generate a discontinuous electrical signal to oscillate.

What more

We can simulate the specific tone according to the fundamental frequency of the piano, so as to play a complete piece of music.

- Piano key frequencies - Wikipedia

```
import machine
import utime

NOTE_C4 = 262
NOTE_G3 = 196
NOTE_A3 = 220
NOTE_B3 = 247

melody =[NOTE_C4,NOTE_G3,NOTE_G3,NOTE_A3,NOTE_G3,NOTE_B3,NOTE_C4]
```

(continues on next page)

(continued from previous page)

```
buzzer = machine.PWM(machine.Pin(15))

def tone(pin,frequency,duration):
    pin.freq(frequency)
    pin.duty_u16(30000)
    utime.sleep_ms(duration)
    pin.duty_u16(0)

for note in melody:
    tone(buzzer,note,250)
    utime.sleep_ms(150)
```

3.4.11 Swinging Servo

In this kit, in addition to LED and passive buzzer, there is also a device controlled by PWM signal, Servo.

Servo is a position (angle) servo device, which is suitable for those control systems that require constant angle changes and can be maintained. It has been widely used in high-end remote control toys, such as airplanes, submarine models, and remote control robots.

Now, try to make the servo sway!

4. Connect the yellow lead of the servo to the GP15 pin with a jumper wire.
5. Connect the brawn lead of the servo to the negative power bus with a jumper wire.

Code

When the program is running, we can see the Servo Arm swinging back and forth from 0° to 180°. The program will always run because of the `while True` loop, we need to press the Stop key to end the program.

```
import machine
import utime

servo = machine.PWM(machine.Pin(15))
servo.freq(50)

def interval_mapping(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def servo_write(pin,angle):
    pulse_width=interval_mapping(angle, 0, 180, 0.5,2.5)
    duty=int(interval_mapping(pulse_width, 0, 20, 0,65535))
    pin.duty_u16(duty)

while True:
    for angle in range(180):
        servo_write(servo,angle)
        utime.sleep_ms(20)
    for angle in range(180,-1,-1):
        servo_write(servo,angle)
        utime.sleep_ms(20)
```

How it works?

We defined the `servo_write()` function to make the servo run.

This function has two parameters:

- pin, the GPIO pin that controls the servo.
- Angle, the angle of the shaft output.

In this function, `interval_mapping()` is called to map the angle range 0 ~ 180 to the pulse width range 0.5 ~ 2.5ms.

```
pulse_width=interval_mapping(angle, 0, 180, 0.5,2.5)
```

Why is it 0.5~2.5? This is determined by the working mode of the Servo.

Servo

Next, convert the pulse width from period to duty. Since duty_u16() cannot have decimals when used (the value cannot be a float type), we used `int()` to force the duty to be converted to an int type.

```
duty=int(interval_mapping(pulse_width, 0, 20, 0,65535))
```

Finally, write the duty value into duty_u16().

(continued from previous page)

```
thermistor = machine.ADC(28)

while True:
    temperature_value = thermistor.read_u16()
    Vr = 3.3 * float(temperature_value) / 65535
    Rt = 10000 * Vr / (3.3 - Vr)
    temp = 1/(((math.log(Rt / 10000)) / 3950) + (1 / (273.15+25)))
    Cel = temp - 273.15
    Fah = Cel * 1.8 + 32
    print ('Celsius: %.2f C  Fahrenheit: %.2f F' % (Cel, Fah))
    utime.sleep_ms(200)
```

How it works?

Each thermistor has a normal resistance. Here it is 10k ohm, which is measured under 25 degree Celsius. When the temperature gets higher, the resistance of the thermistor decreases. Then the voltage data is converted to digital quantities by the A/D adapter.

The temperature in Celsius or Fahrenheit is output via programming.

```
import math
```

There is a numerics library which declares a set of functions to compute common mathematical operations and transformations.

- math

```
temperature_value = thermistor.read_u16()
```

This function is used to read the value of the thermistor.

```
Vr = 3.3 * float(temperature_value) / 65535
Rt = 10000 * Vr / (3.3 - Vr)
temp = 1/(((math.log(Rt / 10000)) / 3950) + (1 / (273.15+25)))
Cel = temp - 273.15
Fah = Cel * 1.8 + 32
print ('Celsius: %.2f C  Fahrenheit: %.2f F' % (Cel, Fah))
utime.sleep_ms(200)
```

These calculations convert the thermistor values into centigrade degree and Fahrenheit degree.

```
Vr = 3.3 * float(temperature_value) / 65535
Rt = 10000 * Vr / (3.3 - Vr)
```

In the two lines of code above, the voltage is first calculated using the read analoge value, and then get Rt (the resistance of the thermistor).

```
temp = 1/(((math.log(Rt / 10000)) / 3950) + (1 / (273.15+25)))
```

Note: Here is the relation between the resistance and temperature:

RT =RN expB(1/TK – 1/TN)

- RT is the resistance of the NTC thermistor when the temperature is TK.

- RN is the resistance of the NTC thermistor under the rated temperature TN. Here, the numerical value of RN is 10k.
- TK is a Kelvin temperature and the unit is K. Here, the numerical value of TK is 273.15 + degree Celsius.
- TN is a rated Kelvin temperature; the unit is K too. Here, the numerical value of TN is 273.15+25.
- And B(beta), the material constant of NTC thermistor, is also called heat sensitivity index with a numerical value 3950.
- exp is the abbreviation of exponential, and the base number e is a natural number and equals 2.7 approximately.

Convert this formula TK=1/(ln(RT/RN)/B+1/TN) to get Kelvin temperature that minus 273.15 equals degree Celsius.

This relation is an empirical formula. It is accurate only when the temperature and resistance are within the effective range.

This code refers to plugging Rt into the formula TK=1/(ln(RT/RN)/B+1/TN) to get Kelvin temperature.

```
temp = temp - 273.15
```

Convert Kelvin temperature into centigrade degree.

```
Fah = Cel * 1.8 + 32
```

Convert the centigrade degree into Fahrenheit degree.

```
print ('Celsius: %.2f °C Fahrenheit: %.2f ' % (Cel, Fah))
```

Print centigrade degree, Fahrenheit degree and their units in the shell.

3.4.14 Light Theremin

Theremin is an electronic musical instrument that does not require physical contact. It produces different tones by sensing the position of the player's hand.

The instrument's controlling section usually consists of two metal antennas that sense the relative position of the thereminist's hands and control oscillators for frequency with one hand, and amplitude (volume) with the other. The electric signals from the theremin are amplified and sent to a loudspeaker.

We cannot reproduce the same instrument through Pico, but we can use photoresistor and passive buzzer to achieve similar gameplay.

- Theremin - Wikipedia

(continued from previous page)

```
light_low=65535
light_high=0

def interval_mapping(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def tone(pin,frequency,duration):
    pin.freq(frequency)
    pin.duty_u16(30000)
    utime.sleep_ms(duration)
    pin.duty_u16(0)

# calibrate the photoresistor max & min values.
timer_init_start = utime.ticks_ms()
led.value(1)
while utime.ticks_diff(utime.ticks_ms(), timer_init_start)<5000:
    light_value = photoresistor.read_u16()
    if light_value > light_high:
        light_high = light_value
    if light_value < light_low:
        light_low = light_value
led.value(0)

# play
while True:
    light_value  = photoresistor.read_u16()
    pitch = int(interval_mapping(light_value,light_low,light_high,50,6000))
    if pitch > 50 :
        tone(buzzer,pitch,20)
    utime.sleep_ms(10)
```

When the program runs, the LED will light up, and we will have five seconds to calibrate the detection range of the photoresistor. This is because we may be in a different light environment each time we use it (e.g. the light intensity is different between midday and dusk).

At this time, we need to swing our hands up and down on top of the photoresistor, and the movement range of the hand will be calibrated to the playing range of this instrument.

After five seconds, the LED will go out and we can wave our hands on the photoresistor to play.

3.4.15 Microchip - 74HC595

Integrated circuit (integrated circuit) is a kind of miniature electronic device or component, which is represented by the letter “IC” in the circuit.

A certain process is used to interconnect the transistors, resistors, capacitors, inductors and other components and wiring required in a circuit, fabricate on a small or several small semiconductor wafers or dielectric substrates, and then package them in a package , it has become a micro-structure with the required circuit functions; all of the components have been structured as a whole, making electronic components a big step towards micro-miniaturization, low power consumption, intelligence and high reliability.

The inventors of integrated circuits are Jack Kilby (integrated circuits based on germanium (Ge)) and Robert Norton Noyce (integrated circuits based on silicon (Si)).

This kit is equipped with an IC, 74HC595, which can greatly save the use of GPIO pins. Specifically, it can replace 8 pins for digital signal output by writing an 8-bit binary number.

2. Insert 74HC595 across the middle gap into the breadboard.
3. Connect the GP0 pin of Pico to the DS pin (pin 14) of 74HC595 with a jumper wire.
4. Connect the GP1 pin of Pico to the STcp pin (12-pin) of 74HC595.
5. Connect the GP2 pin of Pico to the SHcp pin (pin 11) of 74HC595.
6. Connect the VCC pin (16 pin) and MR pin (10 pin) on the 74HC595 to the positive power bus.
7. Connect the GND pin (8-pin) and CE pin (13-pin) on the 74HC595 to the negative power bus.
8. Insert 8 LEDs on the breadboard, and their anode leads are respectively connected to the Q0~Q1 pins (15, 1, 2, 3, 4, 5, 6, 7) of 74HC595.
9. Connect the cathode leads of the LEDs with a 220 resistor in series to the negative power bus.

Code

```
import machine
import time

sdi = machine.Pin(0,machine.Pin.OUT)
rclk = machine.Pin(1,machine.Pin.OUT)
srclk = machine.Pin(2,machine.Pin.OUT)

def hc595_shift(dat):
    rclk.low()
    time.sleep_ms(5)
    for bit in range(7, -1, -1):
        srclk.low()
        time.sleep_ms(5)
        value = 1 & (dat >> bit)
        sdi.value(value)
        time.sleep_ms(5)
        srclk.high()
        time.sleep_ms(5)
    time.sleep_ms(5)
    rclk.high()
    time.sleep_ms(5)

num = 0

for i in range(16):
    if i < 8:
        num = (num<<1) + 1
    elif i>=8:
        num = (num & 0b01111111)<<1
    hc595_shift(num)
    print("{:0>8b}".format(num))
    time.sleep_ms(200)
```

When the program is running, `num` will be written into the 74HC595 chip as an eight-bit binary number to control the on and off of the 8 LEDs. We can see the current value of `num` in the shell.

1. Connect 3V3 and GND of Pico to the power bus of the breadboard.
2. Insert 74HC595 across the middle gap into the breadboard.
3. Connect the GP0 pin of Pico to the DS pin (pin 14) of 74HC595 with a jumper wire.
4. Connect the GP1 pin of Pico to the STcp pin (12-pin) of 74HC595.
5. Connect the GP2 pin of Pico to the SHcp pin (pin 11) of 74HC595.
6. Connect the VCC pin (16 pin) and MR pin (10 pin) on the 74HC595 to the positive power bus.
7. Connect the GND pin (8-pin) and CE pin (13-pin) on the 74HC595 to the negative power bus.
8. Insert the LED Segment Display into the breadboard, and connect a 220 resistor in series with the GND pin to the negative power bus.

> **Note:** The color ring of the 220 ohm resistor is red, red, black, black and brown.

9. Follow the table below to connect the 74hc595 and LED Segment Display.

Table 1: Wiring

74HC595	LED Segment Display
Q0	a
Q1	b
Q2	c
Q3	d
Q4	e
Q5	f
Q6	g
Q7	dp

Code

```
import machine
import time

SEGCODE = [0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f]

sdi = machine.Pin(0,machine.Pin.OUT)
rclk = machine.Pin(1,machine.Pin.OUT)
srclk = machine.Pin(2,machine.Pin.OUT)

def hc595_shift(dat):
    rclk.low()
    time.sleep_ms(5)
    for bit in range(7, -1, -1):
        srclk.low()
        time.sleep_ms(5)
        value = 1 & (dat >> bit)
        sdi.value(value)
        time.sleep_ms(5)
        srclk.high()
        time.sleep_ms(5)
    time.sleep_ms(5)
```

(continues on next page)

(continued from previous page)

```
    rclk.high()
    time.sleep_ms(5)

while True:
    for num in range(10):
        hc595_shift(SEGCODE[num])
        time.sleep_ms(500)
```

When the program is running, you will be able to see the LED Segment Display display 0~9 in sequence.

How it works?

`hc595_shift()` will make 74HC595 output 8 digital signals. It outputs the last bit of the binary number to Q0, and the output of the first bit to Q7. In other words, writing the binary number "00000001" will make Q0 output high level and Q1~Q7 output low level.

Suppose that the 7-segment Display display the number "1", we need to write a high level for b, c, and write a low level for a, d, e, f, g, and dg. That is, the binary number "00000110" needs to be written. For readability, we will use hexadecimal notation as "0x06".

- Hexadecimal
- BinaryHex Converter

Similarly, we can also make the LED Segment Display display other numbers in the same way. The following table shows the codes corresponding to these numbers.

Table 2: Glyph Code

Numbers	Binary Code	Hex Code
0	00111111	0x3f
1	00000110	0x06
2	01011011	0x5b
3	01001111	0x4f
4	01100110	0x66
5	01101101	0x6d
6	01111101	0x7d
7	00000111	0x07
8	01111111	0x7f
9	01101111	0x6f

Write these codes into `hc595_shift()` to make the LED Segment Display display the corresponding numbers.

3.4.17 Liquid Crystal Display

LCD1602 is a character type liquid crystal display, which can display 32 (16*2) characters at the same time.

As we all know, though LCD and some other displays greatly enrich the man-machine interaction, they share a common weakness. When they are connected to a controller, multiple IOs will be occupied of the controller which has no so many outer ports. Also it restricts other functions of the controller. Therefore, LCD1602 with an I2C bus is developed to solve the problem.

- Inter-Integrated Circuit - Wikipedia

(continued from previous page)

```
        self.bus.writeto(self.addr, bytearray([0x08]))

    def write_word(self, data):
        temp = data
        if self.blen == 1:
            temp |= 0x08
        else:
            temp &= 0xF7
        self.bus.writeto(self.addr, bytearray([temp]))

    def send_command(self, cmd):
        # Send bit7-4 firstly
        buf = cmd & 0xF0
        buf |= 0x04                 # RS = 0, RW = 0, EN = 1
        self.write_word(buf)
        time.sleep(0.002)
        buf &= 0xFB                 # Make EN = 0
        self.write_word(buf)

        # Send bit3-0 secondly
        buf = (cmd & 0x0F) << 4
        buf |= 0x04                 # RS = 0, RW = 0, EN = 1
        self.write_word(buf)
        time.sleep(0.002)
        buf &= 0xFB                 # Make EN = 0
        self.write_word(buf)

    def send_data(self, data):
        # Send bit7-4 firstly
        buf = data & 0xF0
        buf |= 0x05                 # RS = 1, RW = 0, EN = 1
        self.write_word(buf)
        time.sleep(0.002)
        buf &= 0xFB                 # Make EN = 0
        self.write_word(buf)

        # Send bit3-0 secondly
        buf = (data & 0x0F) << 4
        buf |= 0x05                 # RS = 1, RW = 0, EN = 1
        self.write_word(buf)
        time.sleep(0.002)
        buf &= 0xFB                 # Make EN = 0
        self.write_word(buf)

    def clear(self):
        self.send_command(0x01) # Clear Screen

    def openlight(self):  # Enable the backlight
        self.bus.writeto(self.addr,bytearray([0x08]))
        # self.bus.close()

    def write(self, x, y, str):
        if x < 0:
            x = 0
        if x > 15:
            x = 15
        if y < 0:
```

(continues on next page)

(continued from previous page)

```
            y = 0
        if y > 1:
            y = 1

        # Move cursor
        addr = 0x80 + 0x40 * y + x
        self.send_command(addr)

        for chr in str:
            self.send_data(ord(chr))

    def message(self, text):
        #print("message: %s"%text)
        for char in text:
            if char == '\n':
                self.send_command(0xC0) # next line
            else:
                self.send_data(ord(char))
```

Then, create a new file, and call the lcd1602 library stored before in this file.

```
from lcd1602 import LCD
import utime

lcd = LCD()
string = " Hello!\n"
lcd.message(string)
utime.sleep(2)
string = "    Sunfounder!"
lcd.message(string)
utime.sleep(2)
lcd.clear()
```

After the program runs, you will be able to see two lines of text appear on the LCD in turn, and then disappear.

How it works?

In the lcd1602 library, we integrate the relevant functions of lcd1602 into the LCD class.

Import lcd1602 library

```
from lcd1602 import LCD
```

Declare an object of the LCD class and name it lcd.

```
lcd = LCD()
```

This statement will display the text on the LCD. It should be noted that the argument must be a string type. If we want to pass an integer or float, we must use the forced conversion statement `str()`.

```
lcd.message(string)
```

If you call this statement multiple times, lcd will superimpose the texts. This requires the use of the following statement to clear the display.

(continued from previous page)

```
    nop().side(0)[T2 - 1]

class WS2812():

    def __init__(self, pin, num):
        # Configure the number of WS2812 LEDs.
        self.led_nums = num
        self.pin = pin
        self.sm = StateMachine(0, ws2812, freq=8000000, sideset_base=self.pin)
        # Start the StateMachine, it will wait for data on its FIFO.
        self.sm.active(1)

        self.buf = array.array("I", [0 for _ in range(self.led_nums)])

    def write(self):
        self.sm.put(self.buf, 8)

    def write_all(self, value):
        for i in range(self.led_nums):
            self.__setitem__(i, value)
        self.write()

    def list_to_hex(self, color):
        if isinstance(color, list) and len(color) == 3:
            c = (color[0] << 8) + (color[1] << 16) + (color[2])
            return c
        elif isinstance(color, int):
            value = (color & 0xFF0000)>>8 | (color & 0x00FF00)<<8 | (color & 0x0000FF)
            return value
        else:
            raise ValueError("Color must be 24-bit  RGB hex or list of 3 8-bit RGB")

    def hex_to_list(self, color):
        if isinstance(color, list) and len(color) == 3:
            return color
        elif isinstance(color, int):
            r = color >> 8 & 0xFF
            g = color >> 16 & 0xFF
            b = color >> 0 & 0xFF
            return [r, g, b]
        else:
            raise ValueError("Color must be 24-bit  RGB hex or list of 3 8-bit RGB")

    def __getitem__(self, i):
        return self.hex_to_list(self.buf[i])

    def __setitem__(self, i, value):
        value = self.list_to_hex(value)
        self.buf[i] = value
```

Then, create a new file, and call the stored ws2812 library here.

```
import machine
from ws2812 import WS2812

ws = WS2812(machine.Pin(0),8)
```

(continues on next page)

(continued from previous page)

```
ws[0] = [64,154,227]
ws[1] = [128,0,128]
ws[2] = [50,150,50]
ws[3] = [255,30,30]
ws[4] = [0,128,255]
ws[5] = [99,199,0]
ws[6] = [128,128,128]
ws[7] = [255,100,0]
ws.write()
```

Let's select some favorite colors and display them on the RGB LED Strip!

How it works?

In the ws2812 library, we have integrated related functions into the WS2812 class.

You can use the RGB LED Strip with the following statement.

```
from ws2812 import WS2812
```

Declare a WS2812 type object, named "ws", it is connected to "pin", there are "number" RGB LEDs on the WS2812 strip.

```
ws = WS2812(pin,number)
```

ws is an array object, each element corresponds to one RGB LED on the WS2812 strip, for example, ws[0] is the first one, ws[7] is the eighth.

We can assign color values to each RGB LED, these values must be 24-bit color (represented with six hexadecimal digits) or list of 3 8-bit RGB.

For example, the red value is "0xFF0000" or "[255,0,0]".

```
ws[i] = color value
```

Then use this statement to write the color for the LED Strip and light it up.

```
ws.write()
```

You can also directly use the following statement to make all LEDs light up the same color.

```
ws.write_all(color value)
```

What more?

We can randomly generate colors and make a colorful flowing light.

```
import machine
from ws2812 import WS2812
import utime
import urandom

ws = WS2812(machine.Pin(0),8)
```

(continues on next page)

(continued from previous page)

```
def flowing_light():
    for i in range(7,0,-1):
        ws[i] = ws[i-1]
    ws[0] = int(urandom.uniform(0, 0xFFFFFF))
    ws.write()
    utime.sleep_ms(80)

while True:
    flowing_light()
    print(ws[0])
```

3.5 MicroPython Basic Syntax

3.5.1 Indentation

Indentation refers to the spaces at the beginning of a code line. Like standard Python programs, MicroPython programs usually run from top to bottom: It traverses each line in turn, runs it in the interpreter, and then continues to the next line, Just like you type them line by line in the Shell. A program that just browses the instruction list line by line is not very smart, though – so MicroPython, just like Python, has its own method to control the sequence of its program execution: indentation.

You must put at least one space before print(), otherwise an error message "Invalid syntax" will appear. It is usually recommended to standardise spaces by pressing the Tab key uniformly.

```
if 8 > 5:
print("Eight is greater than Five!")
```

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<stdin>", line 2
SyntaxError: invalid syntax
```

You must use the same number of spaces in the same block of code, or Python will give you an error.

```
if 8 > 5:
print("Eight is greater than Five!")
        print("Eight is greater than Five")
```

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<stdin>", line 2
SyntaxError: invalid syntax
```

3.5.2 Comments

The comments in the code help us understand the code, make the entire code more readable and comment out part of the code during testing, so that this part of the code does not run.

Single-line Comment

Single-line comments in MicroPython begin with #, and the following text is considered a comment until the end of the line. Comments can be placed before or after the code.

```
print("hello world") #This is a annotationhello world
```

```
>>> %Run -c $EDITOR_CONTENT
hello world
```

Comments are not necessarily text used to explain the code. You can also comment out part of the code to prevent micropython from running the code.

```
#print("Can't run it")
print("hello world") #This is a annotationhello world
```

```
>>> %Run -c $EDITOR_CONTENT
hello world
```

Multi-line comment

If you want to comment on multiple lines, you can use multiple # signs.

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

```
>>> %Run -c $EDITOR_CONTENT
Hello, World!
```

Or, you can use multi-line strings instead of expected.

Since MicroPython ignores string literals that are not assigned to variables, you can add multiple lines of strings (triple quotes) to the code and put comments in them:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

```
>>> %Run -c $EDITOR_CONTENT
Hello, World!
```

As long as the string is not assigned to a variable, MicroPython will ignore it after reading the code and treat it as if you made a multi-line comment.

3.5.3 Print()

The `print()` function prints the specified message to the screen, or other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Print multiple objects:

```
print("Welcome!", "Enjoy yourself!")
```

```
>>> %Run -c $EDITOR_CONTENT
Welcome! Enjoy yourself!
```

Print tuples:

```
x = ("pear", "apple", "grape")
print(x)
```

```
>>> %Run -c $EDITOR_CONTENT
('pear', 'apple', 'grape')
```

Print two messages and specify the separator:

```
print("Hello", "how are you?", sep="---")
```

```
>>> %Run -c $EDITOR_CONTENT
Hello---how are you?
```

3.5.4 Variables

Variables are containers used to store data values.

Creating a variable is very simple. You only need to name it and assign it a value. You don't need to specify the data type of the variable when assigning it, because the variable is a reference, and it accesses objects of different data types through assignment.

Naming variables must follow the following rules:

- Variable names can only contain numbers, letters, and underscores
- The first character of the variable name must be a letter or underscore
- Variable names are case sensitive

Create Variable

There is no command for declaring variables in MicroPython. Variables are created when you assign a value to it for the first time. It does not need to use any specific type declaration, and you can even change the type after setting the variable.

```
x = 8       # x is of type int
x = "lily" # x is now of type str
print(x)
```

```
>>> %Run -c $EDITOR_CONTENT
lily
```

Casting

If you want to specify the data type for the variable, you can do it by casting.

```
x = int(5)     # y will be 5
y = str(5)     # x will be '5'
z = float(5)   # z will be 5.0
print(x,y,z)
```

```
>>> %Run -c $EDITOR_CONTENT
5 5 5.0
```

Get the Type

You can get the data type of a variable with the *type()* function.

```
x = 5
y = "hello"
z = 5.0
print(type(x),type(y),type(z))
```

```
>>> %Run -c $EDITOR_CONTENT
<class 'int'> <class 'str'> <class 'float'>
```

Single or Double Quotes?

In MicroPython, single quotes or double quotes can be used to define string variables.

```
x = "hello"
# is the same as
x = 'hello'
```

Case-Sensitive

Variable names are case-sensitive.

```
a = 5
A = "lily"
#A will not overwrite a
print(a, A)
```

```
>>> %Run -c $EDITOR_CONTENT
5 lily
```

3.5.5 If Else

Decision making is required when we want to execute a code only if a certain condition is satisfied.

if

```
if test expression:
    statement(s)
```

```
x = 10

while x > 0:
    print(x)
    x -= 1
```

```
>>> %Run -c $EDITOR_CONTENT
10
9
8
7
6
5
4
3
2
1
```

Break Statement

With the break statement we can stop the loop even if the while condition is true:

```
x = 10

while x > 0:
    print(x)
    if x == 6:
        break
    x -= 1
```

```
>>> %Run -c $EDITOR_CONTENT
10
9
8
7
6
```

While Loop with Else

Like the *if* loop, the *while* loop can also have an optional *else* block.

If the condition in the *while* loop is evaluated as *False*, the *else* part is executed.

```
x = 10

while x > 0:
    print(x)
    x -= 1
else:
    print("Game Over")
```

```
>>> %Run -c $EDITOR_CONTENT
10
```

(continues on next page)

(continued from previous page)

```
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

```
>>> %Run -c $EDITOR_CONTENT
The sum is 10
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

```
numbers = [1, 2, 3, 4]
sum = 0

for val in numbers:
    sum = sum+val
    if sum == 6:
        break
print("The sum is", sum)
```

```
>>> %Run -c $EDITOR_CONTENT
The sum is 6
```

The continue Statement

With the *continue* statement we can stop the current iteration of the loop, and continue with the next:

```
numbers = [1, 2, 3, 4]

for val in numbers:
    if val == 3:
        continue
    print(val)
```

```
>>> %Run -c $EDITOR_CONTENT
1
2
4
```

The range() function

We can use the range() function to generate a sequence of numbers. range(6) will produce numbers between 0 and 5 (6 numbers).

We can also define start, stop and step size as range(start, stop, step_size). If not provided, step_size defaults to 1.

In a sense of range, the object is "lazy" because when we create the object, it does not generate every number it "contains". However, this is not an iterator because it supports in, len and __getitem__ operations.

This function will not store all values in memory; it will be inefficient. So it will remember the start, stop, step size and generate the next number during the journey.

To force this function to output all items, we can use the function list().

```
print(range(6))

print(list(range(6)))

print(list(range(2, 6)))

print(list(range(2, 10, 2)))
```

```
>>> %Run -c $EDITOR_CONTENT
range(0, 6)
[0, 1, 2, 3, 4, 5]
[2, 3, 4, 5]
[2, 4, 6, 8]
```

We can use *range()* in a *for* loop to iterate over a sequence of numbers. It can be combined with the len() function to use the index to traverse the sequence.

```
fruits = ['pear', 'apple', 'grape']

for i in range(len(fruits)):
    print("I like", fruits[i])
```

```
>>> %Run -c $EDITOR_CONTENT
I like pear
I like apple
I like grape
```

Else in For Loop

The *for* loop can also have an optional *else* block. If the items in the sequence used for the loop are exhausted, the *else* part is executed.

The *break* keyword can be used to stop the *for* loop. In this case, the *else* part will be ignored.

Therefore, if no interruption occurs, the *else* part of the *for* loop will run.

```
for val in range(5):
    print(val)
else:
    print("Finished")
```

```
>>> %Run -c $EDITOR_CONTENT
0
1
2
3
4
Finished
```

The else block will NOT be executed if the loop is stopped by a break statement.

```
for val in range(5):
    if val == 2: break
    print(val)
```

(continues on next page)

(continued from previous page)

```
else:
    print("Finished")
```

```
>>> %Run -c $EDITOR_CONTENT
0
1
```

3.5.8 Functions

In MicroPython, a function is a group of related statements that perform a specific task.

Functions help break our program into smaller modular blocks. As our plan becomes larger and larger, functions make it more organized and manageable.

In addition, it avoids duplication and makes the code reusable.

Create a Function

```
def function_nameparameters)
    """docstring"""
    statement(s)
```

- A function is defined using the `def` keyword
- A function name to uniquely identify the function. Function naming is the same as variable naming, and both follow the following rules.
  - Can only contain numbers, letters, and underscores.
  - The first character must be a letter or underscore.
  - Case sensitive.
- Parameters (arguments) through which we pass values to a function. They are optional.
- The colon (:) marks the end of the function header.
- Optional docstring, used to describe the function of the function, we usually use triple quotes so that the docstring can be expanded to multiple lines.
- One or more valid Micropython statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
- Each function needs at least one statement, but if for some reason there is a function that does not contain any statement, please put in the pass statement to avoid errors.
- An optional `return` statement to return a value from the function.

Calling a Function

To call a function, add parentheses after the function name.

```
def my_function():
    print("Your first function")

my_function()
```

```
>>> %Run -c $EDITOR_CONTENT
Your first function
```

The return Statement

The return statement is used to exit a function and return to the place where it was called.

Syntax of return

```
return [expression_list]
```

The statement can contain an expression that is evaluated and returns a value. If there is no expression in the statement, or the `return` statement itself does not exist in the function, the function will return a `None` object.

```
def my_function():
        print("Your first function")

print(my_function())
```

```
>>> %Run -c $EDITOR_CONTENT
Your first function
None
```

Here, `None` is the return value, because the `return` statement is not used.

Arguments

Information can be passed to the function as arguments.

Specify arguments in parentheses after the function name. You can add as many arguments as you need, just separate them with commas.

```
def welcome(name, msg):
    """This is a welcome function for
    the person with the provided message"""
    print("Hello", name + ', ' + msg)

welcome("Lily", "Welcome to China!")
```

```
>>> %Run -c $EDITOR_CONTENT
Hello Lily, Welcome to China!
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 parameters, you have to call the function with 2 arguments, not more, and not less.

```
def welcome(name, msg):
    """This is a welcome function for
    the person with the provided message"""
    print("Hello", name + ', ' + msg)

welcome("Lily", "Welcome to China!")
```

Herethe function welcome() has 2 parameters.

Since we called this function with two arguments, the function runs smoothly without any errors.

If it is called with a different number of arguments, the interpreter will display an error message.

The following is the call to this function, which contains one and one no arguments and their respective error messages.

```
welcome("Lily")Only one argument
```

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
TypeError: function takes 2 positional arguments but 1 were given
```

```
welcome()No arguments
```

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
TypeError: function takes 2 positional arguments but 0 were given
```

Default Arguments

In MicroPython, we can use the assignment operator (=) to provide a default value for the parameter.

If we call the function without argument, it uses the default value.

```
def welcome(name, msg = "Welcome to China!"):
    """This is a welcome function for
    the person with the provided message"""
    print("Hello", name + ', ' + msg)
welcome("Lily")
```

```
>>> %Run -c $EDITOR_CONTENT
Hello Lily, Welcome to China!
```

In this function, the parameter `name` has no default value and is required (mandatory) during the call.

On the other hand, the default value of the parameter `msg` is “Welcome to China!”. Therefore, it is optional during the call. If a value is provided, it will overwrite the default value.

Any number of arguments in the function can have a default value. However, once there is a default argument, all arguments on its right must also have default values.

This means that non-default arguments cannot follow default arguments.

For example, if we define the above function header as:

```
def welcome(name = "Lily", msg):
```

We will receive the following error message:

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: non-default argument follows default argument
```

Keyword Arguments

When we call a function with certain values, these values will be assigned to arguments based on their position.

For example, in the above function welcome(), when we called it as welcome("Lily", "Welcome to China"), the value "Lily" gets assigned to the `name` and similarly "Welcome to China" to parameter `msg`.

MicroPython allows calling functions with keyword arguments. When we call the function in this way, the order (position) of the arguments can be changed.

```
# keyword arguments
welcome(name = "Lily",msg = "Welcome to China!")

# keyword arguments (out of order)
welcome(msg = "Welcome to China",name = "Lily")

#1 positional, 1 keyword argument
welcome("Lily", msg = "Welcome to China!")
```

As we can see, we can mix positional arguments and keyword arguments during function calls. But we must remember that the keyword arguments must come after the positional arguments.

Having a positional argument after a keyword argument will result in an error.

For example, if the function call as follows:

```
welcome(name="Lily","Welcome to China!")
```

Will result in an error:

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
SyntaxError: non-keyword arg after keyword arg
```

Arbitrary Arguments

Sometimes, if you do not know the number of arguments that will be passed to the function in advance.

In the function definition, we can add an asterisk (*) before the parameter name.

```
def welcome(*names):
    """This function welcomes all the person
    in the name tuple"""
    #names is a tuple with arguments
    for name in names:
        print("Welcome to China!", name)

welcome("Lily","John","Wendy")
```

```
>>> %Run -c $EDITOR_CONTENT
Welcome to China! Lily
Welcome to China! John
Welcome to China! Wendy
```

Here, we have called the function with multiple arguments. These arguments are packed into a tuple before being passed into the function.

Inside the function, we use a for loop to retrieve all the arguments.

Recursion

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

```
def rec_func(i):
    if(i > 0):
        result = i + rec_func(i - 1)
        print(result)
    else:
        result = 0
    return result

rec_func(6)
```

```
>>> %Run -c $EDITOR_CONTENT
1
3
6
10
15
21
```

In this example, rec_func() is a function that we have defined to call itself ("recursion"). We use the `i` variable as the data, and it will decrement (-1) every time we recurse. When the condition is not greater than 0 (that is, 0), the recursion ends.

For new developers, it may take some time to determine how it works, and the best way to test it is to test and modify it.

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

3.5.9 Data Types

Built-in Data Types

MicroPython has the following data types:

- Text Type: str
- Numeric Types: int, float, complex
- Sequence Types: list, tuple, range
- Mapping Type: dict
- Set Types: set, frozenset
- Boolean Type: bool
- Binary Types: bytes, bytearray, memoryview

Getting the Data Type

You can get the data type of any object by using the *type()* function:

```
a = 6.8
print(type(a))
```

```
>>> %Run -c $EDITOR_CONTENT
<class 'float'>
```

Setting the Data Type

MicroPython does not need to set the data type specifically, it has been determined when you assign a value to the variable.

```
x = "welcome"
y = 45
z = ["apple", "banana", "cherry"]

print(type(x))
print(type(y))
print(type(z))
```

```
>>> %Run -c $EDITOR_CONTENT
<class 'str'>
<class 'int'>
<class 'list'>
>>>
```

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Date Type
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = str("Hello World")	str
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

You can print some of them to see the result.

```
a = float(20.5)
b = list(("apple", "banana", "cherry"))
c = bool(5)

print(a)
print(b)
print(c)
```

```
>>> %Run -c $EDITOR_CONTENT
20.5
['apple', 'banana', 'cherry']
True
>>>
```

Type Conversion

You can convert from one type to another with the int(), float(), and complex() methods: Casting in python is therefore done using constructor functions:

- int() - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```
a = float("5")
b = int(3.7)
c = str(6.0)

print(a)
print(b)
print(c)
```

Note: You cannot convert complex numbers into another number type.

3.5.10 Operators

Operators are used to perform operations on variables and values.

- *Arithmetic Operators*
- *Assignment operators*
- *Comparison Operators*
- *Logical Operators*
- *Identity Operators*
- *Membership Operators*
- *Bitwise Operators*

Arithmetic Operators

You can use arithmetic operators to do some common mathematical operations.

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor division

```
x = 5
y = 3

a = x + y
b = x - y
c = x * y
d = x / y
e = x % y
f = x ** y
g = x // y

print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
print(g)
```

```
>>> %Run -c $EDITOR_CONTENT
8
2
15
1.666667
2
125
```

(continues on next page)

(continued from previous page)

```
1
8
2
15
>>>
```

Assignment operators

Assignment operators can used to assign values to variables.

Operator	Example	Same As
=	a = 6	a =6
+=	a += 6	a = a + 6
-=	a -= 6	a = a - 6
*=	a *= 6	a = a * 6
/=	a /= 6	a = a / 6
%=	a %= 6	a = a % 6
**=	a **= 6	a = a ** 6
//=	a //= 6	a = a // 6
&=	a &= 6	a = a & 6
|=	a |= 6	a = a | 6
^=	a ^= 6	a = a ^ 6
>>=	a >>= 6	a = a >> 6
<<=	a <<= 6	a = a << 6

```
a = 6

a *= 6
print(a)
```

```
>>> %Run test.py
36
>>>
```

Comparison Operators

Comparison operators are used to compare two values.

Operator	Name
==	Equal
!=	Not equal
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to

```
a = 6
b = 8
```

(continues on next page)

(continued from previous page)

```
print(a>b)
```

```
>>> %Run test.py
False
>>>
```

Return **False**, beause the **a** is less than the **b**.

Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description
and	Returns True if both statements are true
or	Returns True if one of the statements is true
not	Reverse the result, returns False if the result is true

```
a = 6
print(a > 2 and a < 8)
```

```
>>> %Run -c $EDITOR_CONTENT
True
>>>
```

Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

Operator	Description
is	Returns True if both variables are the same object
is not	Returns True if both variables are not the same object

```
a = ["hello", "welcome"]
b = ["hello", "welcome"]
c = a

print(a is c)
# returns True because z is the same object as x

print(a is b)
# returns False because x is not the same object as y, even if they have the same␣
↪content

print(a == b)
# returns True because x is equal to y
```

```
>>> %Run -c $EDITOR_CONTENT
True
False
True
>>>
```

Membership Operators

Membership operators are used to test if a sequence is presented in an object.

Operator	Description
in	Returns True if a sequence with the specified value is present in the object
not in	Returns True if a sequence with the specified value is not present in the object

```
a = ["hello", "welcome", "Goodmorning"]

print("welcome" in a)
```

```
>>> %Run -c $EDITOR_CONTENT
True
>>>
```

Bitwise Operators

Bitwise operators are used to compare (binary) numbers.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

```
num = 2

print(num & 1)
print(num | 1)
print(num << 1)
```

```
>>> %Run -c $EDITOR_CONTENT
0
3
4
>>>
```

3.5.11 Lists

Lists are used to store multiple items in a single variable, and are created using square brackets:

```
B_list = ["Blossom", "Bubbles","Buttercup"]
print(B_list)
```

List items are changeable, ordered, and allow duplicate values. The list items are indexed, with the first item having index [0], the second item having index [1], and so on.

```
C_list = ["Red", "Blue", "Green", "Blue"]
print(C_list)             # duplicate
print(C_list[0])
print(C_list[1])          # ordered
C_list[2] = "Purple"      # changeable
print(C_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Red', 'Blue', 'Green', 'Blue']
Red
Blue
['Red', 'Blue', 'Purple', 'Blue']
```

A list can contain different data types:

```
A_list = ["Banana", 255, False, 3.14]
print(A_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Banana', 255, False, 3.14]
```

List Length

To determine how many items are in the list, use the len() function.

```
A_list = ["Banana", 255, False, 3.14]
print(len(A_list))
```

```
>>> %Run -c $EDITOR_CONTENT
4
```

Check List items

Print the second item of the list:

```
A_list = ["Banana", 255, False, 3.14]
print(A_list[1])
```

```
>>> %Run -c $EDITOR_CONTENT
[255]
```

Print the last one item of the list:

```
A_list = ["Banana", 255, False, 3.14]
print(A_list[-1])
```

```
>>> %Run -c $EDITOR_CONTENT
[3.14]
```

Print the second, third item:

```
A_list = ["Banana", 255, False, 3.14]
print(A_list[1:3])
```

```
>>> %Run -c $EDITOR_CONTENT
[255, False]
```

Change List Items

Change the second, third item:

```
A_list = ["Banana", 255, False, 3.14]
A_list[1:3] = [True,"Orange"]
print(A_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Banana', True, 'Orange', 3.14]
```

Change the second value by replacing it with two values:

```
A_list = ["Banana", 255, False, 3.14]
A_list[1:2] = [True,"Orange"]
print(A_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Banana', True, 'Orange', False, 3.14]
```

Add List Items

Using the append() method to add an item:

```
C_list = ["Red", "Blue", "Green"]
C_list.append("Orange")
print(C_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Red', 'Blue', 'Green', 'Orange']
```

Insert an item as the second position:

```
C_list = ["Red", "Blue", "Green"]
C_list.insert(1, "Orange")
print(C_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Red', 'Orange', 'Blue', 'Green']
```

Remove List Items

The remove() method removes the specified item.

```
C_list = ["Red", "Blue", "Green"]
C_list.remove("Blue")
print(C_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Red', 'Green']
```

The pop() method removes the specified index. If you do not specify the index, the pop() method removes the last item.

```
A_list = ["Banana", 255, False, 3.14, True,"Orange"]
A_list.pop(1)
print(A_list)
A_list.pop()
print(A_list)
```

```
>>> %Run -c $EDITOR_CONTENT
255
['Banana', False, 3.14, True, 'Orange']
'Orange'
['Banana', False, 3.14, True]
```

The `del` keyword also removes the specified index:

```
C_list = ["Red", "Blue", "Green"]
del C_list[1]
print(C_list)
```

```
>>> %Run -c $EDITOR_CONTENT
['Red', 'Green']
```

The clear() method empties the list. The list still remains, but it has no content.

```
C_list = ["Red", "Blue", "Green"]
C_list.clear()
print(C_list)
```

```
>>> %Run -c $EDITOR_CONTENT
[]
```

CHAPTER

FOUR

FOR ARDUINO USER

This chapter includes installing Arduino IDE, uploading code to Raspberry Pi with Arduino IDE and a dozen interesting and practical projects to help you learn Arduino code quickly.

We recommend that you read the chapters in order.

4.1 Getting Started with Arduino

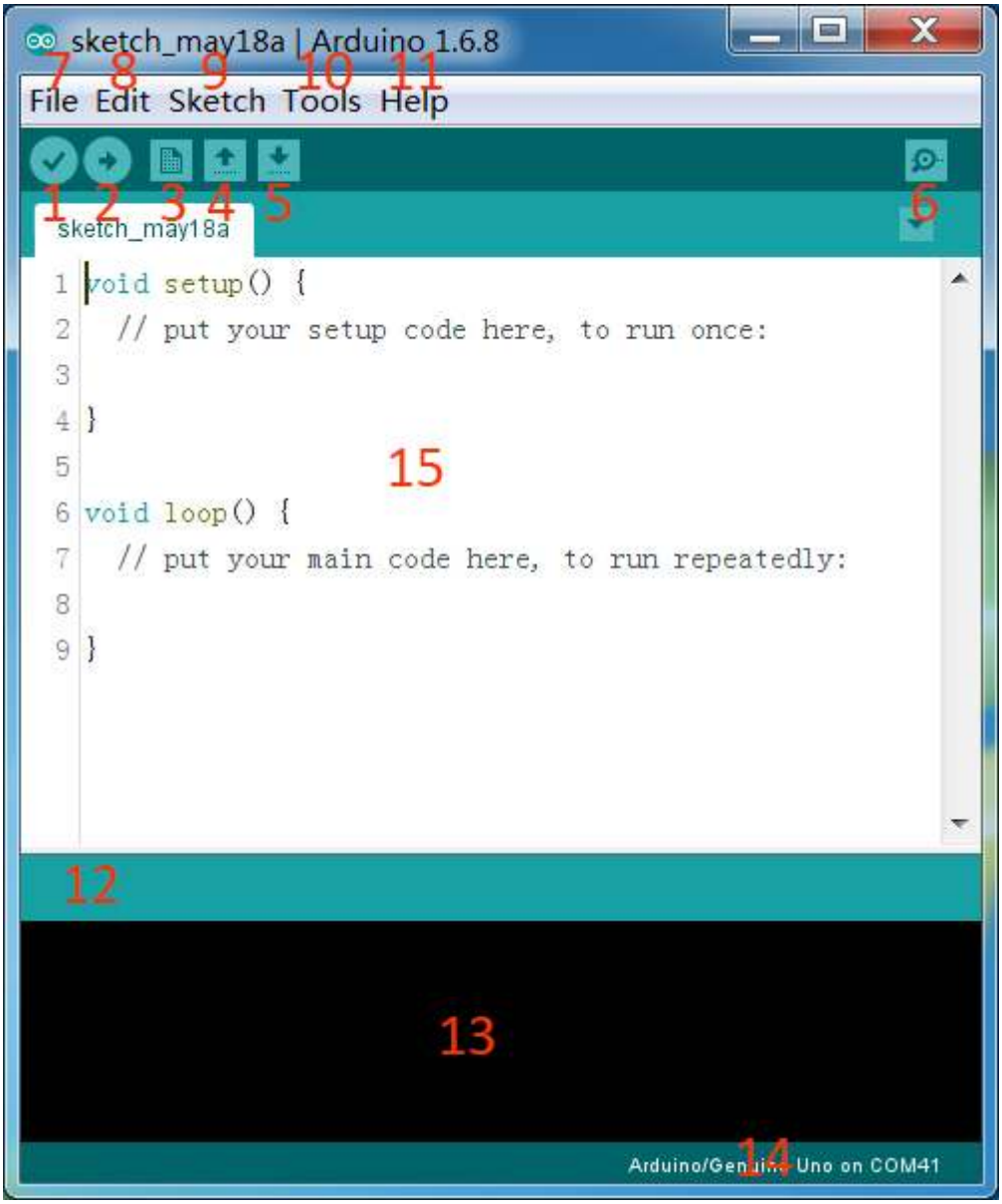
Arduino is an open source platform with simple software and hardware. You can pick it up in short time even if you are a beginner. It provides an integrated development environment (IDE) for code compiling, compatible with multiple control boards. So you can just download the Arduino IDE, upload the sketches (i.e. the code files) to the board, and then you can see relative experimental phenomena.

For more information, refer to Arduino Website.

4.1.1 Install Arduino IDE

Go to Arduino Software Page to download the Arduino IDE accordingly to your operating systemthen follow the instructions to install it.

Click here to get step-by-step instructions accordingly to your operating system.



1. Verify: Compile your code. Any syntax problem will be prompted with errors.
2. Upload: Upload the code to your board. When you click the button, the RX and TX LEDs on the board will flicker fast and won't stop until the upload is done.
3. New: Create a new code editing window.
4. Open: Open an .ino sketch.
5. Save: Save the sketch.
6. Serial Monitor: Click the button and a window will appear. It receives the data sent from your control board. It is very useful for debugging.
7. File: Click the menu and a drop-down list will appear, including file creating, opening, saving, closing, some parameter configuring, etc.
8. Edit: Click the menu. On the drop-down list, there are some editing operations like Cut, Copy, Paste, Find, and so on, with their corresponding shortcuts.

9. Sketch: Includes operations like Verify, Upload, Add files, etc. More important function is Include Library – where you can add libraries.
10. Tool: Includes some tools – the most frequently used Board (the board you use) and Port (the port your board is at). Every time you want to upload the code, you need to select or check them.
11. Help: If you're a beginner, you may check the options under the menu and get the help you need, including operations in IDE, introduction information, troubleshooting, code explanation, etc.
12. In this message area, no matter when you compile or upload, the summary message will always appear.
13. Detailed messages during compile and upload. For example, the file used lies in which path, the details of error prompts.
14. Board and Port: Here you can preview the board and port selected for code upload. You can select them again by Tools -> Board / Port if any is incorrect.
15. The editing area of the IDE. You can write code here.

4.2 Projects

Note: The Pico Arduino core is a recent development, so it may take a long time to compile and upload, or the COM port may suddenly disappear during use.

- For the former, we have to wait patiently until a new Arduino core is released.
- For the latter case, you can unplug the USB cable, then press and hold the BOOTSEL button to plug it in, unplug it and plug it in again to see the COM port.

4.2.1 Hello LED

The digitalWrite() statement here is used to write high level or low level to pins and to let LED and active buzzer work or stop. In this lesson, we will take LED as an example to introduce the experiment phenomenon.

Code

- You can click the **Download Sketch** icon to download the code and open it with the desktop Arduino IDE.
- Or **OPEN IN WEB EDITOR**.
- Then upload the code to your Pico(*Setup the Raspberry Pi Pico*).

How it works?

Here, we connect the LED to the GPIO15, so we define a variable `ledPin` to represent GPIO15. You could also just replace all the `ledPin` pins in the code with 15, but if you were to replace 15 with other pins you would have to modify 15 one by one, which would add a lot of work.

```
#define ledPin 15
```

Now, you need to set the pin to OUTPUT mode in the `setup()` function.

```
pinMode(ledPin, OUTPUT);
```

- pinMode()

The above code has "set" the pin, but it will not light up the LED. Here, we use the digitalWrite() function to assign a high level signal to `ledpin`, which will cause a voltage difference between the LED pins, causing the LED to light up.

```
digitalWrite(ledPin, HIGH);
```

If the level signal is changed to LOW, the ledPin's signal will be returned to 0 V to turn LED off.

```
digitalWrite(ledPin, LOW);
```

An interval between on and off is required to allow people to see the change, so we use a `delay(1000)` code to let the controller do nothing for 1000 ms.

```
delay(1000);
```

4.2.2 Fading LED

In last projects, we have used only two output signals: high level and low level (or called 1 & 0, ON & OFF), which is called digital output.

However, in actual use, many devices do not simply ON/OFF to work, for example, adjusting the speed of the motor, adjusting the brightness of the desk lamp, and so on. In the past, a slider that can adjust the resistance was used to achieve this goal, but this is always unreliable and inefficient.

Therefore, Pulse width modulation (PWM) has emerged as a feasible solution to such complex problems. A digital output composed of a high level and a low level is called a pulse. The pulse width of these pins can be adjusted by changing the ON/OFF speed.

Simply put, when we are in a short period (such as 20ms, most people's visual retention time), Let the LED turn on, turn off, and turn on again, we won't see it has been turned off, but the brightness of the light will be slightly weaker. During this period, the more time the LED is turned on, the higher the brightness of the LED. In other words, in the cycle, the wider the pulse, the greater the "electric signal strength" output by the microcontroller. This is how PWM controls LED brightness (or motor speed).

- Pulse-width modulation - Wikipedia

2. Connect one end (either end) of the 220 ohm resistor to GP15, and insert the other end into the free row of the breadboard.
3. Insert the anode lead of the LED into the same row as the end of the 220 resistor, and connect the cathode lead across the middle gap of the breadboard to the same row.
4. Connect the LED cathode to the negative power bus of the breadboard.
5. Insert the slide switch into the breadboard.
6. Use a jumper wire to connect one end of slide switch pin to the negative bus.
7. Connect the middle pin to GP14 with a jumper wire.
8. Use a jumper wire to connect last end of slide switch pin to the positive bus
9. Use a 10K resistor to connect the middle pin of the slide switch and the negative bus.
10. Use a 104 capacitor to connect the middle pin of the slide switch and the negative bus to realize debounce that may arise from your toggle of switch.
11. Connect the negative power bus of the breadboard to Pico's GND.

When you toggle the slide switch, the circuit will switch between closed and open.

- *Slide Switch*
- *Capacitor*

Code

After the code has run, toggle the Slide switch to one side and the LED will flash. Toggle to the other side and the LED will go out.

How it works?

For switchs, we need to set their mode to `INPUT` in order to be able to get their values.

```
void setup() {
    // initialize the LED pin as an output:
    pinMode(ledPin, OUTPUT);
    // initialize the switch pin as an input:
    pinMode(switchPin, INPUT);
}
```

Read the status of the `switchPin` in loop() and assign it to the variable `switchState`.

```
switchState = digitalRead(switchPin);
```

- digitalRead()

If the `switchState` is HIGH, the LED will flash.

```
if (switchState == HIGH)
{
    // turn LED on:
    digitalWrite(ledPin, HIGH);
    delay(1000);
    digitalWrite(ledPin, LOW);
    delay(1000);
}
```

- Serial

In the loop function, the value of the potentiometer is read, then the value is mapped from 0-1023 to 0-255 and finally the value after the mapping is used to control the brightness of the LED.

```
void loop() {
    int sensorValue = analogRead(sensorPin);
    Serial.println(sensorValue);
    int brightness = map(sensorValue, 0, 1023, 0, 255);
    analogWrite(ledPin, brightness);
}
```

- analogRead() is used to read the value of the sensorPin (potentiometer) and assigns it to the variable `sensorValue`.

```
int sensorValue = analogRead(sensorPin);
```

- Print the value of SensorValue in Serial Monitor.

```
Serial.println(sensorValue);
```

- Here, the map(value, fromLow, fromHigh, toLow, toHigh) function is required as the potentiometer value read is in the range 0-1023 and the value of a PWM pin is in the range 0-255. It is used to Re-maps a number from one range to another. That is, a value of fromLow would get mapped to toLow, a value of fromHigh to toHigh, values in-between to values in-between, etc.

```
int brightness = map(sensorValue, 0, 1023, 0, 255);
```

- Now we can use this value to control the brightness of the LED.

```
analogWrite(ledPin,brightness);
```

The new projects for Arduino are still being updated continuously.

CHAPTER

SIX

COPYRIGHT NOTICE

All contents including but not limited to texts, images, and code in this manual are owned by the SunFounder Company. You should only use it for personal study,investigation, enjoyment, or other non-commercial or nonprofit purposes, under therelated regulations and copyrights laws, without infringing the legal rights of the author and relevant right holders. For any individual or organization that uses these for commercial profit without permission, the Company reserves the right to take legal action.