



MicroPython for the Internet of Things

A Beginner's Guide to Programming with Python on Microcontrollers

Breaking the C-like language barrier to make device programming easy and fast

Charles Bell

Apress®

MicroPython for the Internet of Things

A Beginner's Guide to Programming with Python on Microcontrollers



Charles Bell

Apress®

MicroPython for the Internet of Things

Charles Bell
Warsaw, Virginia, USA

ISBN-13 (pbk): 978-1-4842-3122-7
<https://doi.org/10.1007/978-1-4842-3123-4>

ISBN-13 (electronic): 978-1-4842-3123-4

Library of Congress Control Number: 2017960889

Copyright © 2017 by Charles Bell

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Technical Reviewer: Nelson Goncalves
Coordinating Editor: Jill Balzano
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484231227. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

I dedicate this book to my big sister for instilling in me the thirst for knowledge.

Contents

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
■ Chapter 1: What Is the Internet of Things?	1
The Internet of Things and You	2
IOT Is More Than Just Connected to the Internet.....	3
IOT Services	4
A Brief Look at IOT Solutions	5
IOT and Security	15
Python and the IOT	19
Origins	19
Online Python Simulator	20
Summary	24
■ Chapter 2: Introducing MicroPython	27
Getting Started	27
Origins	29
MicroPython Features.....	30
MicroPython Limitations.....	31
What Does MicroPython Run On?.....	31
Experimenting with Python on Your PC.....	32

How MicroPython Works	38
The Run, Evaluate, Print Loop (REPL Console)	39
Off and Running with MicroPython.....	44
Additional Hardware	46
Example 1:.....	50
Example 2:.....	52
Example 3:.....	55
Summary	57
Chapter 3: MicroPython Hardware	59
Getting Started with MicroPython Boards	59
Firmware Updates	59
Networking Issues	60
One Step at a Time!	61
Programming Tools.....	61
Some Assembly Required.....	63
GPIO Pins	64
Other Tips	64
MicroPython-Ready Boards.....	69
Pyboard.....	69
WiPy.....	82
MicroPython-Compatible Boards.....	96
BBC micro:bit.....	96
Circuit Playground Express (Developer Edition).....	99
Adafruit Feather Huzzah	104
Other Boards	108
Breakout Boards and Add-Ons	108
Breakout Boards	109
Board-Specific Shields/Skins	112
Board-Specific Accessories.....	116

Which Board Should I Buy? 121

Summary 123

■ Chapter 4: How to Program in MicroPython 125

Basic Concepts 126

 Code Blocks 126

 Comments 127

 Arithmetic 128

 Output to Screen 129

 Variables 130

 Types 132

Basic Data Structures 133

 Lists 133

 Tuples 134

 Dictionaries 134

Statements 136

 Conditional Statements 136

 Loops 137

Modularization; Modules, Functions, and Classes 138

 Including Modules 138

 Functions 139

 Classes and Objects 140

Learning Python by Example 147

 Example 1: Using Loops 147

 Example 2: Using Complex Data and Files 150

 Example 3: Using Functions 156

 Example 4: Using Classes 162

For More Information 171

Summary 171

■ Chapter 5: MicroPython Libraries	173
Built-In and Standard Libraries	174
Overview.....	174
Common Standard Libraries.....	177
Built-In Functions and Classes	187
Exceptions	192
MicroPython Libraries	195
Overview.....	196
Common MicroPython Libraries	196
Custom Libraries	199
Summary	204
■ Chapter 6: Low-Level Hardware Support	205
Board-Specific Libraries.....	206
Pyboard.....	206
WiPy.....	215
Low-Level Examples	220
Drivers and Libraries to the Rescue!	220
Real-Time Clock (RTC).....	221
Callbacks	224
Using Breakout Boards.....	226
Inter-Integrated Circuit (I2C).....	227
Serial Peripheral Interface (SPI).....	233
Summary	237
■ Chapter 7: Electronics for Beginners	239
The Basics	240
Tools	240
Multimeter	241
Soldering Iron	242

Wire Strippers.....	244
Helping Hands	244
Using a Multimeter	246
Powering Your Electronics	254
Electronic Components	255
Button	255
Capacitor	256
Diode	257
Fuse.....	257
Light Emitting Diode (LED).....	258
Relay.....	260
Resistor	260
Switch.....	261
Transistor.....	262
Voltage Regulator	263
Breakout Boards and Circuits.....	264
Using a Breadboard to Build Circuits	265
What Are Sensors?	268
How Sensors Measure.....	269
Examples of Sensors	271
Summary	282
■ Chapter 8: Project 1: Hello, World! MicroPython Style	283
Overview	284
Required Components.....	285
Set Up the Hardware	288
WiPy.....	290
Pyboard.....	291

- Write the Code..... 293
 - Design..... 293
 - Libraries Needed 293
 - Planning the Code..... 299
 - Test the Parts of the Code..... 304
 - Completed Code 306
- Execute! 309
- Taking It Further 311
- Summary..... 313
- Chapter 9: Project 2: Stoplight Simulator 315**
 - Overview 316
 - Required Components..... 316
 - Set Up the Hardware 319
 - WiPy..... 320
 - Pyboard..... 321
 - Write the Code..... 323
 - Part 1: Stoplight Simulator – Using a Pushbutton 323
 - Part 2: Stoplight Simulator – Remote Control with HTML..... 329
 - Completed Code 335
 - Execute! 341
 - Taking it Further 343
 - Summary..... 344
- Chapter 10: Project 3: Plant Monitoring 345**
 - Overview 345
 - Required Components..... 347
 - Set Up the Hardware 349
 - WiPy..... 350
 - Pyboard..... 351

Write the Code.....	351
Calibrating the Sensor	352
Part 1: Sensor Code Module	354
Part 2: Main Code	364
Execute!	375
Taking it Further	376
Summary	377
■ Chapter 11: Project 4: Using Weather Sensors.....	379
Overview	380
Message Queue Telemetry Transport	380
How It Works.....	381
Getting Started with Adafruit IO.....	382
Required Components	384
Set Up the Hardware	385
Configure Adafruit IO	387
Set Up Feeds	387
Set Up a Dashboard.....	388
Get Your Credentials	396
Write the Code.....	397
MQTT Driver.....	398
BME280 Library	398
Weather Class.....	399
Main Code.....	404
Execute!	407
Taking it Further	412
Summary	412

Chapter 12: Where to Go from Here	415
More Projects to Explore	415
MicroPython Project Samples.....	416
Forums.....	416
Documentation	417
Repositories.....	417
Community Project Sites: Hackster.io.....	418
Knowledge Repositories: learn.adafruit.io.....	420
Join the Community	421
Why Contribute?	422
Which License, Where?.....	422
How We Share	423
Keep Your Designs Original.....	424
Check the License	424
Keep It Appropriate.....	425
Annotate Your Work	426
Be a Good Citizen.....	427
Suggested Communities.....	427
Become a Maker	429
What's a Maker?.....	429
Share Your Ideas.....	430
Attend an Event	430
Summary	431
Appendix	433
Required Components.....	433
Optional Components	435
Recommended Tools	436
Index	439

About the Author



Dr. Charles Bell conducts research in emerging technologies. He is a member of the Oracle MySQL Development Team as a Senior Developer assisting in the development of MySQL high availability solutions. He lives in a small town in rural Virginia with his loving wife. He received his doctorate of philosophy in engineering from Virginia Commonwealth University in 2005. His research interests include database systems, software engineering, sensor networks, and 3D printing. He spends his limited free time as a practicing Maker focusing on microcontroller and 3D printers and printing projects.

About the Technical Reviewer



Nelson Goncalves is a member of the MySQL Developer Team at Oracle, currently working with the InnoDB cluster Team. He is based in Portugal and working for MySQL since 2013. Before joining MySQL, he was an MSc student at Universidade do Minho where he specialized into Formal Methods and Distributed Systems. He loves all things Python related.

Acknowledgments

I would like to thank all of the many talented and energetic professionals at Apress. I appreciate the understanding and patience of my editor, Jonathan Gennick; and managing editor, Jill Balzano. They were instrumental in the success of this project. I would also like to thank the army of publishing professionals at Apress for making me look so good in print with a special thank you to the reviewers for their wise counsel and gentle nudges in the right direction. Thank you all very much!

Most importantly, I want to thank my wife, Annette, for her unending patience and understanding while I spent so much time with my laptop.

Introduction

Internet of Things (IOT) solutions are not nearly as complicated as the name may seem to indicate. Indeed, the IOT is largely another name for what we have already been doing. You may have heard of “connected devices” or “Internet-ready” or even “cloud-enabled.” All of these refer to the same thing — be it a single device such as a toaster or a plant monitor or a complex, multidevice product like home automation solutions. They all share one thing in common: they can be accessed via the Internet to either display data or interact with the devices directly. The trick is applying knowledge of technologies to leverage them to the best advantages for your IOT solution. In this book, we explore how to build IOT solutions using an easy-to-understand programming language named MicroPython running on small, dedicated microcontroller boards.

Intended Audience

I wrote this book to share my passion for Python and IOT solutions. I especially wanted to show how anyone can program their own IOT solutions in Python using MicroPython on small microcontroller boards. The intended audience therefore includes anyone interested in learning how to build IOT solutions, hobbyists, and enthusiasts who don't want to spend a lot of time learning a complicated programming language to control hardware through software in IOT solutions.

How This Book Is Structured

The book was written to guide the reader from a general knowledge of microcontrollers and MicroPython to expertise in developing MicroPython solutions for the IOT. The first several chapters cover general topics including a short introduction to the Internet of Things, what microcontroller boards are available as well as how MicroPython works. Later chapters present a tutorial on programming in MicroPython as well as an introduction to electronics. This is followed by four projects that you can implement to learn how to build MicroPython IOT solutions. Throughout the book are examples of how to implement many of the concepts presented. The following is a brief overview of each chapter included in this book.

- *Chapter 1, “What Is the Internet of Things?”*: This chapter presents and answers the questions of what the IOT is and how IOT solutions are constructed. You are introduced to some terminology to describe the architecture of IOT solutions as well as some examples of well-known IOT solutions. The chapter concludes with a demonstration of MicroPython.

- *Chapter 2, “Introducing MicroPython”*: This chapter presents an overview of what MicroPython is and how you can get started using MicroPython boards.
- *Chapter 3, “MicroPython Hardware”*: This chapter discusses some of the hardware available for MicroPython including the micropython.org (Pyboard) and Pycomm (WiPy) line of microcontroller boards and several other alternative boards. The chapter also presents some of the accessories available for each board.
- *Chapter 4, “How to Program in MicroPython”*: This chapter presents a tutorial on learning to program in MicroPython. It covers all of the basics of the language you need to get started writing your own MicroPython scripts.
- *Chapter 5, “MicroPython Libraries”*: This chapter presents an overview of the various MicroPython libraries available for use in your scripts. It includes many examples of how to get started using the libraries to interface with hardware.
- *Chapter 6, “Low-Level Hardware Support”*: This chapter presents an overview of the low-level hardware abstractions available for the Pyboard and WiPy ports of MicroPython. The differences of the libraries are presented along with several complete examples to demonstrate the functionality.
- *Chapter 7, “Electronics for Beginners”*: This chapter presents a short introduction to electronics including the types of components you will be using in the book along with a list of recommended tools. The chapter concludes with a survey of the types of sensors available for IOT solutions.
- *Chapter 8, “Project 1: Hello, World! MicroPython Style”*: This chapter presents a hands-on project to help get you started programming hardware and building MicroPython solutions. The project is a clock programmed in MicroPython using a real-time-clock (RTC) module.
- *Chapter 9, “Project 2: Stoplight Simulator”*: This chapter presents another hands-on project that interfaces with LEDs and buttons to build a pedestrian stoplight simulation. The project also demonstrates how to control your hardware remotely via a web page.
- *Chapter 10, “Project 3: Plant Monitoring”*: This chapter presents a more complex hands-on project that demonstrates how to generate sensor data and view it over the Internet. The project is a plant monitoring solution that you can expand from one to many plants.

- *Chapter 11, “Project 4: Using Weather Sensors”*: This chapter presents the last hands-on project that combines all that you have learned in the book to build a working IOT solution. The project is a small weather sensor node that uses the new Adafruit IO cloud services to store and visualize the data.
- *Chapter 12, “Where to Go from Here”*: This chapter concludes the tour of MicroPython IOT solutions with suggestions for more projects to explore and where to go to find new project ideas including where to look to find answers to questions or problems you may encounter when developing your own MicroPython IOT projects. The chapter also discusses how you can join the community of IOT, MicroPython, and electronics enthusiasts by becoming a Maker.

How to Use This Book

This book is designed to guide you through learning more about what the Internet of Things is, discovering the power of MicroPython, and seeing how to build your own IOT solutions.

If you already have your own MicroPython board and are familiar with some of the topics early in the book, I recommend you skim them so that you are familiar with the context presented so that the later chapters, especially the examples, are easy to understand and implement on your own. You may also want to read some of the chapters out of order so that you can get your project moving, but I recommend going back to the chapters you skip to ensure you get all of the data presented.

If you are just getting started with MicroPython and microcontrollers, I recommend reading the book in its entirety before developing your own IOT solutions. That said, many of the examples presented in the early chapters are building blocks for what follows in the project chapters.

Downloading the Code

The code for the examples shown in this book is available on the Apress web site, www.apress.com. You can find a link on the book’s information page on the Source Code/Downloads tab. This tab is located in the Related Titles section of the page.

Contacting the Author

Should you have any questions or comments — or even spot a mistake you think I should know about — you can contact the author at drcharlesbell@gmail.com.

CHAPTER 1



What Is the Internet of Things?

If you've been watching the technology world lately, chances are you have encountered numerous mentions of the term, the Internet of Things. Most media references and company advertisements label this or that as the Internet of Things but with little or no explanation about what it means. Even when you do find some depth of what it means, the text tends to focus on either the problems and challenges, or they focus on the promise of making our lives better in the future. Some suggest the Internet of Things will bring about the inevitable evolution of our society as we become more connected to the world around us every day.

However, you need not dive into such heady concepts or recite rhetoric to get started with the Internet of Things. In fact, through the efforts of many open source developers and vendors, you can explore the Internet of Things without intensive training or expensive hardware and software. Best of all, you can explore the Internet of Things without learning a lot about programming or spending months learning how to code!

This book is intended to be a guide to help you understand the Internet of Things and to begin building solutions that you can use to learn more about the Internet of Things. Since this is a beginner's book, we will start by examining the programming language and environment followed by a detailed look at the hardware. We will also learn the basic knowledge of electronics and then explore several projects to help us understand how to work with the software. The final project will bring all the aspects together to help understand what the Internet of Things is, and even how to write custom software for building solutions for the Internet of Things. Best of all, we do so using one of the easiest to use programming languages and easy to use open source microcontroller boards.

So, what is this Internet of Things, hence IOT?¹ Let's begin by explaining what it isn't. The IOT is not a new device or proprietary software or some new piece of hardware, nor is it a new marketing scheme to sell you more of what you already have by renaming it and pronouncing it "new and improved."² While it is true the IOT employs technology and techniques that already exist, the way they are employed, coupled with the ability to access the solution from anywhere in the world, makes the IOT an exciting concept to explore. Now let's discuss what the IOT is.

¹https://en.wikipedia.org/wiki/Internet_of_Things

²For example, everything seems to be cloud-this, cloud-that when nothing was changed.

The essence of the IOT is simply interconnected devices that generate and exchange data from observations, facts, and other data making it available to anyone. While there seems to be some marketing efforts attempting to make anything connected to the Internet an IOT solution or device – not unlike the shameless labeling of everything ‘cloud,’ IOT solutions are designed to make our knowledge of the world around us more timely and relevant by making it possible to get data about anything from anywhere at any time.

As you can imagine, if we were to connect every device around us to the Internet and make sensory data available for those devices, it is clear there is potential for the number of IOT devices to exceed the human population of the planet and for the data generated to rapidly exceed the capabilities of all but the most sophisticated database systems. These concepts are commonly known as addressability and big data and are two of the most active and debated topics in IOT.

However, the IOT is all about understanding the world around us. That is, we can leverage the data to make our world and our understanding of it better.

The Internet of Things and You

How do we observe the world around us? The human body is a marvel of ingenious sensory apparatus that allows us to see, hear, taste, and even feel through touch anything we encounter. Even our brains can store visual and auditory events recalling them at will. IOT solutions mimic many of these sensory capabilities and therefore can become an extension of our own abilities. While that may sound a bit grandiose (and it is), IOT solutions can record observations in the form of data from one or more sensors and make them available for viewing by anyone anywhere via the Internet.

Sensors are devices that produce either analog or digital values. We can then use the data collected to draw conclusions about the subject matter. This could be as simple as a sensor to detect when a door, window, or mailbox is opened. In the case of a switch on a mailbox, the knowledge we gain from a simple switch opening or closing (depending on how it is implemented and interpreted) may be used to predict when incoming mail has arrived or when outgoing mail has been picked up. I use the term predict because the sensor (switch) only tells us the door was opened or closed, not that anything was placed in or removed from the mailbox itself – that would require additional sensors.

A more sophisticated example is using a series of sensors to record atmospheric data such as temperature, humidity, barometric pressure, wind speed, ambient light, rainfall, etc., to monitor the weather that allows us to perform analysis on the data to predict trends in weather. That is, we can predict within a reasonable certainty that precipitation is in the area.

Now, add the ability to see this data not only in real time (as it occurs), but also remotely from anywhere in the world, and the solution becomes more than a simple weather station. It becomes a way to observe the weather about one place from anywhere in the world. This example may seem to be a bit commonplace since you can tune into any number of television, web, and radio broadcasts to hear the weather from anywhere in the world. But consider the implications of building such a solution in your home. Now you can see data about the weather in your own home!

In the same way, but perhaps on a smaller scale, we can build solutions to monitor plants to help us understand how often they need water and other nutrients. Or perhaps we can monitor our pets while we are away at work. Further, we can record data about wildlife in our area to better understand our effect on nature.

IOT Is More Than Just Connected to the Internet

If a device is connected to the Internet, does that make it an IOT solution? That depends on whom you ask. Some believe the answer is yes. However, others (such as myself) contend that the answer is not unless there is some benefit from doing so. For example, if you connected your toaster to the Internet, what could be the benefit of doing so? It would be pointless (or at least extremely eccentric) to get a text on your phone from your toaster stating that your toast is ready. So, in this case, the answer is no.

However, if you have persons such as responsible teenagers or perhaps older adults whom you would like to monitor, it may be helpful to be able to check to see how often they use their toaster and when. That is, you can use the data to help you make decisions about their care and safety.

To me, if there is no use for the data, whether it is something that is viewed in real time or is stored for later processing, then simply connecting it to the Internet does not make it an IOT solution. There must be some gain in the use of the device. Thus, being connected to the Internet doesn't make something IOT. Rather, IOT solutions must be those things that provide some meaning – however small that has benefit to someone or some other device or service.

More importantly, whatever we build IOT solutions to do, they allow us to sense the world around us and learn from those observations. The real tricky part is in how the data is collected, stored, and presented. We will see these in practice through examples in later chapters. See the sidebar for an example of a controversial IOT device - a common household appliance.

However, IOT solutions can often take advantage of companies that provide services that can help enhance or provide features you can use in your IOT solutions. These features are commonly called IOT Services and range from storage and presentation to the infrastructure services such as hosting.

INTERNET-ENABLED APPLIANCES: IOT OR MARKETING HYPE?

One of the ideas or concepts that seems to be becoming popular is the connecting of major household appliances to the Internet. While manufacturers may want you to believe this is a new and exciting IOT device, the truth is it is neither a new idea nor is it a world changing IOT solution.

I was fortunate to participate in a design workshop held on the Microsoft campus in the late 1990s. During our tour of the campus, we were introduced to the world's first Internet-enabled refrigerator (also called a smart refrigerator or simply Internet refrigerator).³ There were sensors in the shelves to detect the weight of food. It was suggested that, with a little ingenuity, that one could use the sensors to notify your grocer when your milk supply ran low, which would enable people to have their grocery shopping not only online but also automatic.

³https://en.wikipedia.org/wiki/Internet_refrigerator

Now, 20 years later, we're seeing manufacturers building refrigerators that connect to the Internet. However, unlike the first smart refrigerator, these new devices are positioned to be a social media focal point for the household. Many don't provide any meaningful data about the contents of the refrigerator outside of the gadget-like ability to see a video image of the contents without opening the door, which could have been solved by installing a glass door.

Suffice to say IOT enthusiasts like me scratch their heads at how something like this could possibly be useful much less sell well. Sadly, these new Internet refrigerators do indeed seem to be selling well, but I wonder if consumers have been sucked into the hype. For an interesting commentary on why the Internet refrigerator isn't for you, do a Google search and you'll find a lot of opinions – most negative (and yet, people still buy these things).⁴

Let's judge the Internet refrigerator with my definition of IOT: does it enhance your life by providing you information about the world around you? Well, if you need to check to see how much milk you have while 3000 miles away from home, then I guess it may be beneficial but for the multitude of us who prefer to just open the door and look before we go to the store, it may not be an IOT device.

IOT Services

Sadly, there are companies that tout having IOT products and services that are nothing more than marketing hype – much like what some companies have done by prepending 'cloud' or appending 'for the cloud' to the name. Fortunately, there are some good products and services being built especially for IOT. These range from data storage and hosting to specialized hardware.

Indeed, businesses are adding IOT services to their product offerings faster than anyone can keep up with the latest. And it isn't the usual suspects such as the Internet giants. I have seen IOT solutions and services being offered by Cisco, AT&T, HP, and countless startups and smaller businesses. I use the term IOT vendor to describe those businesses that provide services for IOT solutions.

You may be wondering what these services and products are and why one would consider using them. That is, what is an IOT service and why would you decide to buy it? The biggest reason you may decide to buy a service concerns cost and time to market.

If your developers do not have the resources or expertise and obtaining them will require more than the cost of the service, it may be more economical to purchase the service. However, you should also consider any additional software or hardware changes (sometimes called retooling) necessary in the decision. I once encountered a well-meaning and well-documented contracted service that permitted a product to go to market sooner than projected at a massive savings. Sadly, while the champions of that

⁴<https://www.howtogeek.com/260896/why-buying-a-smart-fridge-is-a-dumb-idea/>

contract won awards for technical achievement, they failed to consider the fact that the systems had to be retooled to use the new service. More specifically, it took longer to adopt the new service than it would have to write one from scratch. So instead of saving money, the organization spent nearly triple and were late to market. Clearly, one must consider all factors.

Similarly, if your time is short or you have hard deadlines to make your solution production ready, it may be quicker to purchase an IOT service rather than create or adapt your own. This may require spending a bit more, but in this case the motivation is time and not (necessarily) cost. Of course, it is a mixture of both cost and time.

So, what are some of the IOT services available? The following lists a few that have emerged in the last few years. It is likely more will be offered as IOT solutions and services mature.

- *Enterprise IOT Data Hosting and Presentation* – services that allow your users to develop enterprise IOT solutions from connecting to, managing, and customizing data presentation in a friendly form such as graphs, charts, etc. Example: Xively (<https://xively.com/>)
- *IOT Data Storage* – services that permit you to store your IOT data and get simple reports. Example: Sparkfun's IOT Data service (<https://data.sparkfun.com/>)
- *Networking* – services that provide networking and similar communication protocols or platforms for IOT. Most specialize in machine-to-machine (M2M) services Example: AT&T's cellular global SIM service (business.att.com/enterprise/Family/mobility-services/internet-of-things)
- *IOT Hardware Platforms* – vendors that permit you to rapidly develop and prototype IOT devices using a hardware platform and a host of supported modules and tools for building devices ranging from a simple component to a complete device. Example: Intel's IOT gateway development kits (intel.com/content/www/us/en/embedded/solutions/iot-gateway/development-kits.html)

Now that we know more about what IOT is, let's look at a few examples of IOT solutions to get a better idea of what IOT solutions can do and how they are employed.

A Brief Look at IOT Solutions

An IOT solution is simply a set of devices designed to produce, consume, or present data about some event or series of events or observations. This can include devices that generate data such as a sensor, devices that combine data to deduce something, devices or services designed to tabulate and store the data, and devices or systems designed to present the data. Any of these may be connected to the Internet.

If you, or someone you know, has spent any time in a medical facility, chances are a sensor network was employed to monitor body functions such as your body temperature, heart rate, respiratory capacity, or even movement range of your limbs. Modern automobiles also contain sensor networks dedicated to monitoring the engine, climate, and even in some cars road conditions. For example, the lane-warning feature uses sensors (typically a camera, microprocessor, and software) to detect when you drift too far toward lane or road demarcations. Manufacturing plants also employ sensor networks in monitoring and controlling the machines, conveyors, and more. Shipping clearinghouses also employ sensor networks to help route packages to the correct bins and ultimately to the correct trucks or planes for transport.

Thus, sensor networks employ one or more sensors that take measurements (observations) about an event or state and communicate that data to another component or node in the network, which is then presented in some form or another for analysis. Let's look at an example of an important medical IOT solution.

Medical Applications

Medical applications including health monitoring and fitness are gaining a lot of attention as consumer products. These solutions cover a wide range of capabilities such as the fitness features built into the new Apple Watch to Fitness bands that keep track of your workout and even medical applications that help you control life-threatening conditions. For example, there are solutions that can help you manage diabetes.

Diabetes is a disease that affects millions of people worldwide (diabetes.org/). There are several forms: the most serious being type 1 (diabetes.org/diabetes-basics/type-1/?loc=db-slabnav). Those afflicted with type 1 diabetes do not produce enough (or any) insulin due to genetic deficiencies, birth defects, or injuries to the pancreas. Insulin is a hormone the body uses to extract a simple sugar called glucose, which is created from sugars and starches, from blood for use in cells. Failure to monitor your blood sugar can result in dangerously low or high blood sugar levels, both of which can be life threatening and if not controlled can cause long-term damage to internal organs, nerves, and other areas. It is a most serious condition.

ATHLETES AND DIABETES

Professional athletes are some of the most physically fit people in the world. Many are examples of health and admired by fans and fellow athletes alike. In the past, if a professional athlete contracted a disease like diabetes type 1, their career would be over. Now, with modern medical technology, professional athletes are starting to overcome their condition and continue to compete.

One shining example is Ryan Reed, a NASCAR Xfinity stock car racer and driver of the number 16 Lily Diabetes Ford. In 2011, Reed was diagnosed with diabetes type 1 and told he would never race again. Since then, Reed has overcome his handicap through careful monitoring of his condition and has returned to racing.

Not only has Reed returned to the sport he loves, he has won the season opening premier series race at Daytona International Raceway not once, but twice. Reed is proof that education, vigilance, and technology can make our lives better.

Type 1 diabetics must monitor their blood glucose to ensure they are using their medications (primarily insulin) properly and balanced with a healthy lifestyle and diet. If their blood glucose levels drop too low or too high, they can suffer from a host of symptoms. Worse, extremely low blood glucose levels are very dangerous and can be fatal.

One of the newest versions of a blood glucose tester consists of a small sensor that is left in the body for as much as a week along with a monitor that connects to the sensor via Bluetooth. You wear the monitor on your body (or keep it within 20 feet at all times). The solution is marketed by Dexcom (dexcom.com/) and is called a continuous glucose monitor (CGM) that permits the patient to share their data to others via their phone. Thus, the patient pairs their CGM with their phone and then shares the data over the Internet to others. This could be loved ones, those that help with their care, or even medical professionals.

Figure 1-2 shows an example of the Dexcom CGM monitor and sensor. The monitor is on the left and the sensor and transmitter are on the right. The sensor is the size of a small syringe needle and remains inserted in the body for up to a week.



Figure 1-2. *Dexcom Continuous Glucose Monitor with Sensor*

A feature called Dexcom Share permits the patient to make their data available to others via an app on their phone. That is, the patient's phone transmits data to the Dexcom cloud servers, which is then sent to anyone who has the Dexcom Share app and has been given permission to see the data. Figure 1-3 shows an example of the Dexcom Share CGM report from the Dexcom Share iOS app, which allows you to easily and quickly check the blood glucose of a friend or loved one.

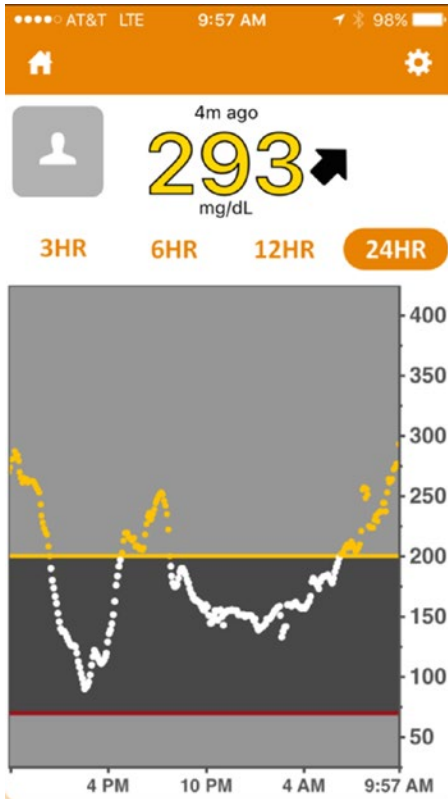


Figure 1-3. Dexcom Share App Report

Not only does the app allow the visualization of the data, it can also relay alerts for low or high blood glucose levels, which has profound implications for patients who suffer from additional ailments or complications from diabetes. For example, if the patient's blood glucose level drops while they are alone, incapacitated, or unable to get treatment, loved ones with the Dexcom Share app can respond by checking on the patient and potentially avoiding a critical diabetic event.

While this solution is a single sensor connected to the Internet via a proprietary application, it is an excellent example of a medical IOT device that can enhance the lives of not only the patient but everyone who cares for them.

Dexcom also provides a free Windows application called Dexcom Studio (<http://dexcom.com/dexcom-studio>) to allow patients to see the data their monitors collect and generate a host of reports they can use to see their glucose levels over time. Reports include averages, patterns, daily trends, and more. They can even share their data with their doctor. Figure 1-4 shows an example of the Dexcom Studio with typical data loaded.



Figure 1-4. Dexcom Studio

WHAT ABOUT BLOOD GLUCOSE TESTERS – GLUCOMETERS?

Until solutions like the Dexcom CGM came about, diabetics had to use a manual tester. Traditional blood glucose testers are single-use events that require the patient to prick their finger or arm and draw a small amount of blood onto a test strip. While this device has been used for many years, it is only recently that manufacturers have started making blood glucose testers with memory features and even connectivity to other devices such as laptops or phones. The ultimate evolution of these devices is a solution like Dexcom, which has become a medical IOT device that improves the quality of life for diabetics.

Combined with the programmable alerts, you and your loved ones can help manage the effects of diabetes. If you have a loved one who suffers with diabetes, a CGM is worth every penny for peace of mind alone. This is the true power of IOT materialized in a potentially lifesaving solution.

Automotive IOT Solutions

Another personal IOT solution is the use of Internet-connected automotive features. One of the oldest products is called OnStar (onstar.com) and is available on most late-model and new General Motors (GM) vehicles. While OnStar is a satellite-based service that has several levels and many fee-based options, it incorporates the Internet to permit communication with vehicle owners. Indeed, the newest GM vehicles come with a WiFi access point built into the car! Better still, there are some basic features that are free to GM owners that, in my opinion, are very valuable.

The free, basic features include regular maintenance reports sent to you via email and the ability to use an app on your phone to unlock, lock, remote start – all the features on your key fob remotely. This is a cool feature if you have ever locked your keys in your car! Figure 1-5 shows an example of the remote key fob app on iOS. Of course, there are even more features available for a fee including navigation, telephone, WiFi, and on-call support.

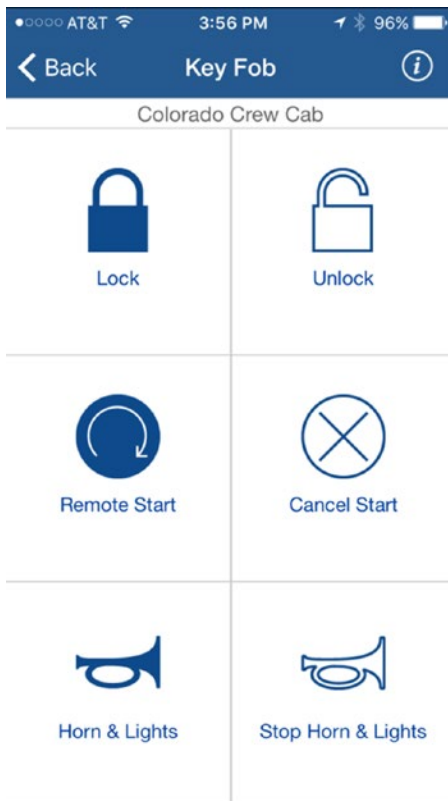


Figure 1-5. OnStar App Key Fob Feature

The OnStar app works by connecting to the OnStar services in the cloud, requesting the feature (e.g., unlock) that is sent to the vehicle via the OnStar satellite network. So, it is an excellent example of how IOT solutions use multiple communication protocols.

The feature I like most is the maintenance reports. You will receive an email with an overview of the maintenance status of your vehicle. The report includes such things as oil life, tire pressure, engine and transmission warnings, emissions, air bag, and more. Figure 1-6 shows an excerpt of a typical email you would receive.

Diagnostics Report from your 2015 Chevrolet Colorado Crew Cab as of

Dear

You are currently enrolled in OnStar Basic Plan. You will continue to receive this Diagnostics Report and Dealer Maintenance Notifications at no cost. To learn more about OnStar plans and services, visit [OnStar.com](#).

DIAGNOSTIC INFORMATION

- Engine and Transmission System
- Emissions System
- Air Bag System
- StabiliTrak® Stability Control System
- Antilock Braking System

REQUIRED MAINTENANCE

Vehicle Maintenance
No oil change due at this time.

Remaining Oil Life: **44%**
Mileage: **27,501**

Odometer-Based Maintenance Items
Based on your current mileage, no items on the additional maintenance list are due at this time.

Tire Pressure: Normal

- No issues found.
- Recommended tire pressure - Front: 35 psi, Rear: 35 psi

Left Front: 34 psi | Right Front: 35 psi
Left Rear: 34 psi | Right Rear: 34 psi

VEHICLE INFORMATION

2015 Chevrolet Colorado Crew Cab
VIN: :
Explore the [Owner Center](#) to learn more about your vehicle.

Recalls and Programs
To check for recalls and programs on your GM vehicle, [click here](#).

Warranty Tracker
Your vehicle has one or more active warranties.

PACKAGES AND SERVICES

OnStar® Subscription

- Account
- Basic Plan
- Expires

You are currently enrolled in Basic Plan. If you'd like to experience more benefits of OnStar, [click here](#) or call 1.888.4.ONSTAR (1.888.466.7827) if you would like to upgrade your plan today.

Plan Add-Ons
You can add individual services to any OnStar plan.
[Purchase add-ons](#)

Data Plan
You are not currently enrolled in a data plan. Please call 1.877.865.7864, or [click here](#) to purchase a plan today.

OTHER INFORMATION

Insurance Benefit
Your mileage makes you eligible for a low mileage discount on auto insurance.
[EXPLORE OPTIONS](#)

OnStar Smart Driver

- Status: Not Enrolled

[Enroll now](#) in our OnStar Smart Driver Program.

Figure 1-6. OnStar Maintenance Report

Notice the information displayed. Actual data is transmitted to OnStar from your vehicle. For example, the odometer reading and tire pressure data are taken directly from the vehicle's onboard data storage. That is, data from the sensors is read, interpreted, and the report generated for you. This feature demonstrates how automatic compilation of data in an IOT solution can help us keep our vehicles in good mechanical condition with early warning of needed maintenance. This serves us best by helping us keep our vehicles in prime condition and thus in a state of high resell value.

I should note that GM isn't the only automotive manufacturer offering such services. Many others are working on their own solutions ranging from an OnStar-like feature set to solutions that focus on entertainment and connectivity.

Fleet Management

Another example of an IOT solution is a fleet management system.⁶ While developed, and deployed well before the coining of the phrase, IOT, fleet management systems allow businesses to monitor their cars, trucks, ships – just about any mobile unit – to not only track their current location but also to use the location data (GPS coordinates taken over time) to plan more efficient routes, thereby reducing the cost of shipment.

Fleet management systems aren't just for routing. Indeed, fleet management systems also allow businesses to monitor each unit to conduct diagnostics. For example, it is possible to know how much fuel is in each truck, when its last maintenance was performed or, more importantly, when the next maintenance is due, and much more. The combination of vehicle geographic tracking and diagnostics is called telematics. Figure 1-7 shows a drawing of a fleet management system.

⁶https://en.wikipedia.org/wiki/Fleet_management

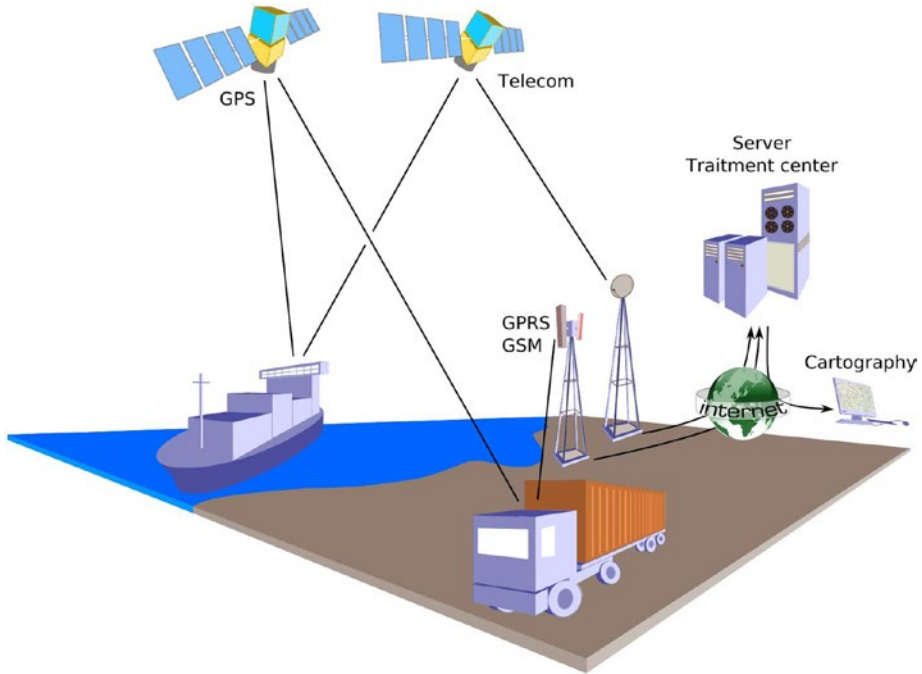


Figure 1-7. Fleet Management Example⁷

In the figure, you will see the application of GPS systems to track location as well as satellite communication to transmit additional data such as diagnostics, payload states, and more. All of these ultimately traverse the Internet and the data becomes accessible by the business analysts.

You may think fleet management systems are only for large shipping companies, but with the proliferation of GPS modules and even the microcontroller market, anyone can create a fleet management system. That is, they don't cost millions of dollars to develop.

For example, if you owned a bicycle delivery company, you could easily incorporate GPS modules with either cellular or wireless connectivity on each delivery person to track their location, average travel time, and more. More specifically, you can use such a solution to minimize delivery times by allowing packages to be handed off from one delivery person to another rather than having them return to the depot each time they complete a set of deliveries.

⁷Éric Chassaing – via CC BY-SA 3.0 (<http://creativecommons.org/licenses/by-sa/3.0/>).

CAMERA DRONES AND THE IOT

One possible use of the IOT is making data that drones generate available over the Internet. Some may feel drones are an invasion of privacy, and I agree in situations where they are misused or established laws are violated. Fortunately, the clear majority of drone owners obey local laws, regulations, and property owners' wishes.⁸

However, there are many legitimate uses of drones be they land-, air-, or sea-based. For example, I can imagine home monitoring solutions where you can check on your home remotely by viewing data from fixed cameras as well as data from mobile drones. I for one would love to see a solution that allowed me to program a predetermined sentry flight path to monitor my properties with a flying camera drone.

While some vendors have WiFi-enabled drones, there aren't many consumer-grade options available that stream data real time over the Internet. However, there are some options that allow you to post video and photos directly from the drone. One of my drones is the Yuneec Breeze (www.yuneec.com/en_US/products/breeze/overview.html), which allows me to post photos and video collected from the drone to social media. Interestingly, these drones are being called "selfie drones" since they have features that allow autonomous modes including a mode where the drone follows you filming your antics, or circling your position, and other interesting features.

While these new drones require a manual action to post data, it is just a matter of time before we see real-time IOT solutions that include drones. Of course, the current controversy and indeed the movement of the U.S. government to register and track drones along with increasing restrictions on their use may limit the expansion of drones and IOT solutions that include drone-acquired data.

IOT and Security

The recent rash of massive data breaches proves that basic security simply wasn't good enough. We've seen everything from outright theft to exploitation of the data stolen from very well-known businesses like Target (over 40 million credit card numbers may have been compromised) and government agencies like the United States Office of Personnel Management (over 20 million social security numbers compromised).

IOT solutions are not immune to security threats. Indeed, as IOT solutions become more and more integrated into our lives, so too will our personal data. Thus, security must be taken extremely seriously and built into the solution from the start.

⁸As of December 21, 2015, drones in the United States that weigh more than 0.55 lbs. must be registered before flying. See <https://registermyuas.faa.gov/>.

This includes solutions we develop ourselves. More specifically, if you design a weather station for your own use, you should take reasonable steps to ensure the data is protected from both accidental and deliberate exploitation. You may think weather data isn't a high risk but consider the case where you include GPS coordinates for your sensors (a reasonable feature) so that people can see where this weather is being observed. If someone could see that information, and determine the solution uses an Internet connection, it is possible they could gain physical access to the Internet device and possibly use it to further penetrate and exploit your systems. Thus, security isn't just about the data; it should encompass all aspects of the solution from data to software to hardware to physical access.

There are four areas where you may want to consider spending extra care ensuring your IOT solution is protected with good security. As you will see, this includes several things you should consider for your existing infrastructure, computers, and even safe computing habits. By leveraging these areas, you will be building a layered approach to security: often called a defense-in-depth method.

DO I REALLY NEED TO WORRY ABOUT SECURITY?

If you're wondering why I've included this section in a beginner's book on the IOT and Python, consider for a moment what you ultimately want to do with the knowledge you gain from this book. If you are only interested in learning how to work with your new MicroPython board and have no aspirations for developing anything more, then you may want to skim these sections. However, if your goals include making MicroPython IOT solutions that you deploy – especially if you plan to connect it to the Internet – you will want to consider security in your solution. Either way, I strongly recommend reading and adhering to these tips for securing your IOT solutions.

Security Begins at Home

Before introducing an IOT solution to your home network, you should consider taking precautions to ensure the machines on your home network are protected. This is important because if someone gets access to your home network, they can achieve all manner of nefarious activities.

The most common mistake made is not securing a home WiFi network. Not only does this mean your neighbors can jump onto your network and hog your bandwidth, it also means they may be able to get to the systems on your home network leaving your IOT devices, computers, appliances, etc., vulnerable to attack.

Fortunately, there are some best practices for securing your home networking to help reduce these risks. These include the following. *Passwords*: This may seem like a simple thing, but always make sure you use passwords on all your computers and devices. Also, adopt good password habits such as requiring longer strings, mixed case, numbers, and symbols to ensure the passwords are not easily guessed.⁹

- *Secure your WiFi*: If you have a WiFi network, make sure you add a password and use the latest security protocols such as WPA2 or, even better, the built-in secure setup features of some wireless routers.
- *Use a firewall*: You should also use a firewall to block all unused ports (TCP or UDP). For example, lock down all ports except those your solution uses such as port 80 for html.
- *Restrict physical access*: Lock your doors! Just because your network has a great password and your computers use super world espionage spy encrypted biometric access, these things are meaningless if someone can gain access to your networking hardware directly. For IOT solutions, this means any external components should be installed in tamper-proof enclosures or locked away so they cannot be discovered. This also includes any network wiring.

Some of these you may know how to do it, but others may require help from a friend who knows more about the devices and networking. For example, if you don't know what a firewall is, ask someone to help you. A little extra security is worth the effort to learn the basics of how to set up a firewall.

Secure Your Devices

As mentioned above, your IOT devices also need to be secured. Some practices to consider include the following.

- *Use passwords*: Always add passwords to the user accounts on any device that has an operating system. This includes making sure you rename any default passwords. For example, you may be tempted to consider a wee Raspberry Pi too small of a device to be a security concern but if you consider these devices run one of the most powerful operating systems available (forms of Linux), a Raspberry Pi can be a very powerful hacking tool.

⁹You also need to balance complexity of passwords with your ability to remember them. If you must write it down, you've just defeated your own security! You may be surprised to learn that the first rule of guessing someone's password is to look under their keyboard. That's where most people put their post-it note with the password. Don't do that.

- *Keep your software up to date:* You should try to use the latest versions of any software you use. This includes the operating system as well as any firmware or programming tools you may be running. Newer versions often have improved security or fewer security vulnerabilities.
- *If your software offers security features, use them:* If you have servers or services running on your devices, and they offer features such as automatic lockout for missed passwords, turn them on. Not all software has these features, but if they are available, they can be a great way to defeat repeated attacks.

Use Encryption

This is one area that is often overlooked. While it is an option normally used only by solutions that transmit confidential data such as commercial IOT devices, if you plan to send data you feel is sensitive, you can further protect yourself and your data if you encrypt both your data as it is stored and the communication mechanism as it is being transmitted. If you encrypt your data, even if someone were to gain physical access to the storage device, the data is useless because they cannot easily decipher the encryption. Use the same care with your encryption keys and passcodes as you do your computer passwords.

Security Doesn't End at the Cloud

There are many considerations for connecting IOT devices to cloud services. Indeed, Microsoft and others have made it very easy to use cloud services with your IOT solutions. However, there are two important considerations for security and your IOT data.

- *Do you need the cloud?:* The first thing you should consider is whether you need to put any of your data in the cloud. It is often the case that cloud services make it very easy to store and view your data, but is it necessary to do so? For example, you may be very concerned and quite keenly eager to view logistical data for where your dog spends his time while you are at work, but who else would care to view this data? In this case, storing the data in the cloud to make it available to everyone is not necessary.
- *Don't relax!:* Many people seem to let their guard down when working with cloud services. For whatever reason, they consider the cloud more secure. The fact is, it isn't! In fact, you must apply the very same security best practices when working in the cloud that you do for your own network, computers, and security policies. Indeed, if anything, you need to be even more vigilant because cloud services are not in your control with respect to protecting against physical access (however remote and unlikely) nor are you guaranteed your data isn't on the same devices as tens, hundreds, or even thousands of other users' data.

Now that we have an idea of how we should include security in our projects, let's take a brief look at the programming language we will use in this book – Python.

Python and the IOT

Python is a high-level, interpreted, object-oriented scripting language. One of the biggest tenants of Python is to have a clear, easy-to-understand syntax that reads as close to English as possible. That is, you should be able to read a Python script and understand it even if you haven't learned Python. Python also has less punctuation (special symbols) and fewer syntactical machinations than other languages.

Here are a few of the key features of Python.

- An interpreter processes Python at runtime. No compiler is used.
- Python supports object-oriented programming constructs by way of classes and methods.
- Python is a great language for the beginner-level programmers and supports the development of a wide range of applications.
- Python is a scripting language but can be used for a wide range of applications.
- Python is very popular and used throughout the world giving it a huge support base.
- Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

Origins

Python was developed by Guido van Rossum from the late 1980s to the early 1990s at the National Research Institute for Mathematics and Computer Science in the Netherlands and maintained by a core development team at the institute. It was derived from and influenced by many languages including Modula-3, C, C++, and even Unix shell scripting languages.

A fascinating fact about Python is it was named after the BBC show, “Monty Python's Flying Circus” and has nothing to do with the reptile by the same name.¹⁰ Quoting Monty Python in source code documentation (and even a humorous diversion for error messages) is very common and while some professional developers may cringe at the insinuation, it's considered by Pythonistas¹¹ as showing your Python street cred. If you like Monty Python, I encourage you to use snippets from the shows in your code. One place I like to have fun with is printing messages. My favorite goes something like this:¹²

```
> DUPLICATE FILE ERROR: He says they've already got one!
```

¹⁰Monty Python refers to a group of comedians and not a single individual. However, the comedy is undeniably brilliant. https://en.wikipedia.org/wiki/Monty_Python

¹¹Pythonistas are expert Python developers and advocates for all things Python.

¹²<http://mzonline.com/bin/view/Python/HolyGrailScene9/>

Some may wonder how a language like Python could possibly be helpful in writing IOT solutions. The answer to that is a cool product called MicroPython. In short, MicroPython is a condensed, optimized code of Python 3 that has been loaded in hardware. This means rather than having to have an interpreter run on an operating system to execute the Python code, the MicroPython chip can run the Python code directly on the hardware. No operating system is needed. In fact, MicroPython has basic file I/O built in.

We'll learn more about MicroPython in the next chapter. It's an exciting new option for those who want to explore IOT but don't want to learn a complex programming language or spend a lot of time learning new operating systems, tools, and hardware. But first, let's look at how easy it is to use MicroPython.

Online Python Simulator

For those of you who are eager to get a taste of what it is like to use Python to control hardware, the good folks over at micropython.org have an online, interactive MicroPython simulator (<http://micropython.org/live>) using one of the most popular MicroPython boards, the pyboard (www.adafruit.com/products/2390). When you visit the site, you will see an interface that has a live video image of a pyboard connected to several devices including a servo, LEDs, an LCD panel, and even an SD drive.

The simulator has several sample scripts (Python programs are called scripts) that you can try out. There are scripts to turn LEDs on, print to the LCD, and even move the servos. The simulator is designed to allow you to select a script and submit it to a queue to run. If there are no other scripts in the queue, you will see the script you selected run almost right away. Figure 1-8 shows an example of a simple script to turn an LED on. In this case, it is the LED on the board itself.

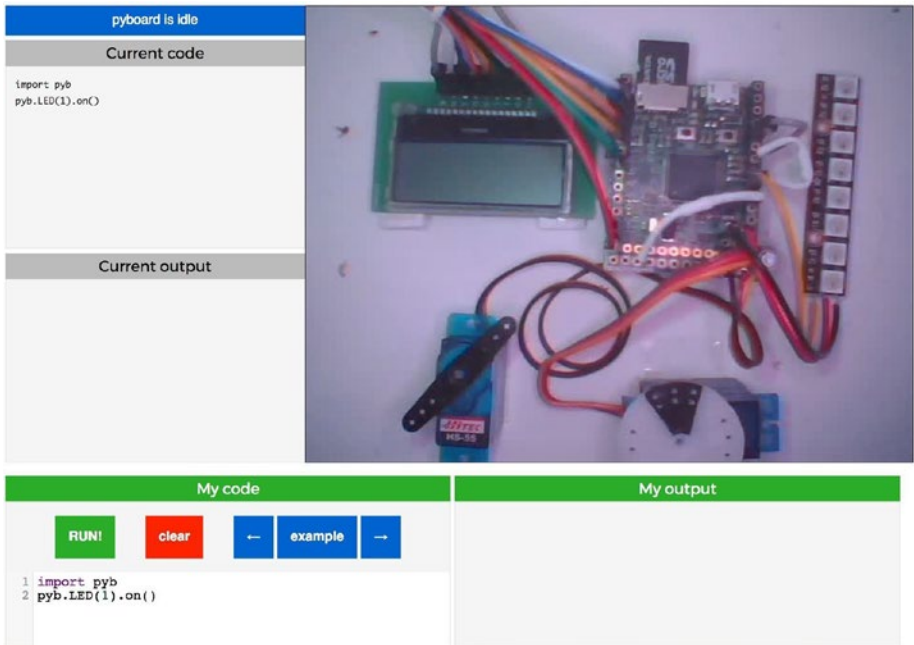


Figure 1-8. MicroPython Simulator - LED on (courtesy of micropython.org)

Notice in the image we see several sections or panels. Clockwise from top left we see the current code (*Current code*) and current output from the code that is running on the board (*Current output*), a video feed of the board in action, an area to display the output of the script (using the `print()` function) for the code you executed (*My output*), and a panel (*My code*) that allows you to scroll through the available example scripts and execute them. The following buttons allow you to control the script.

- *Run!*: Executes the current code in the My Code panel
- *clear*: Clears the My Code panel
- *left arrow*: Cycle back through the list of example scripts
- *right arrow*: Cycle forward through the list of example scripts

One of the really cool things about the simulator is you can use the *My code* panel and write your own code! Just click the *clear* button and position your cursor on line number 2 (below the `import` statement) and start typing. When you're ready, click the *Run!* button. Figure 1-9 shows an example of a script that prints two lines showing code I wrote to run on the board.

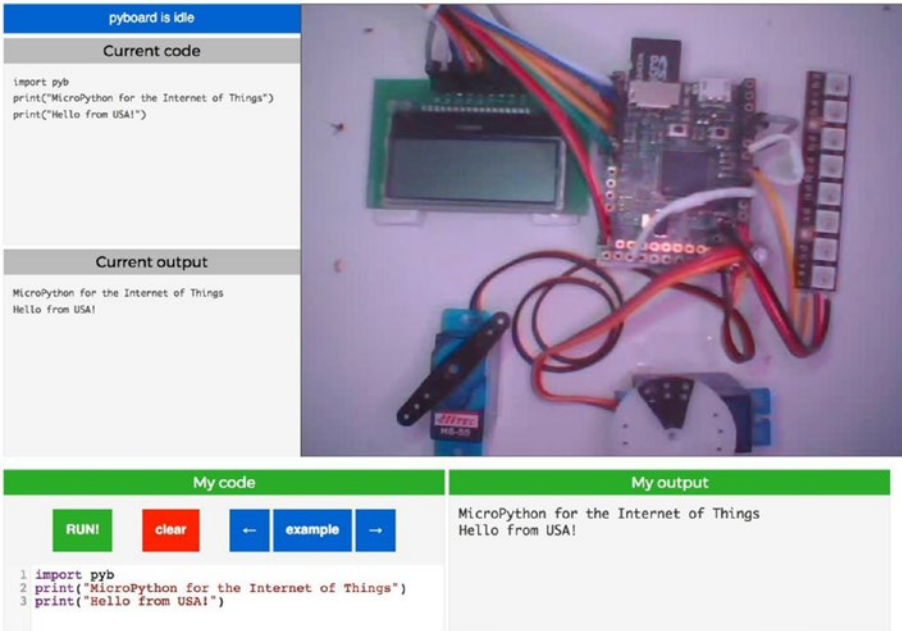


Figure 1-9. *MicroPython Simulator - Custom code (courtesy of micropython.org)*

Figures 1-10 and 1-11 show two additional example scripts.

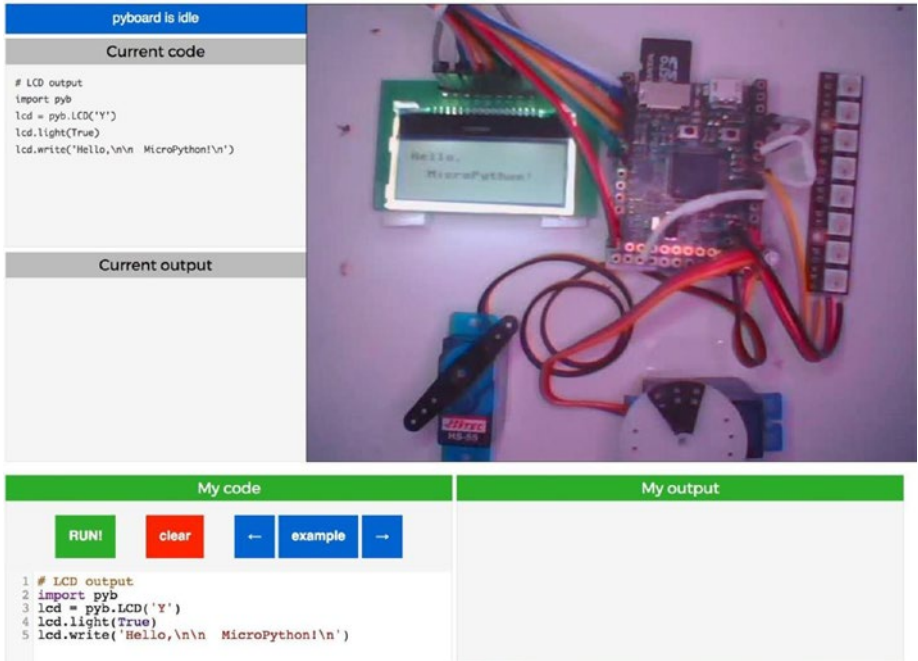


Figure 1-10. MicroPython Simulator – LCD (courtesy of micropython.org)

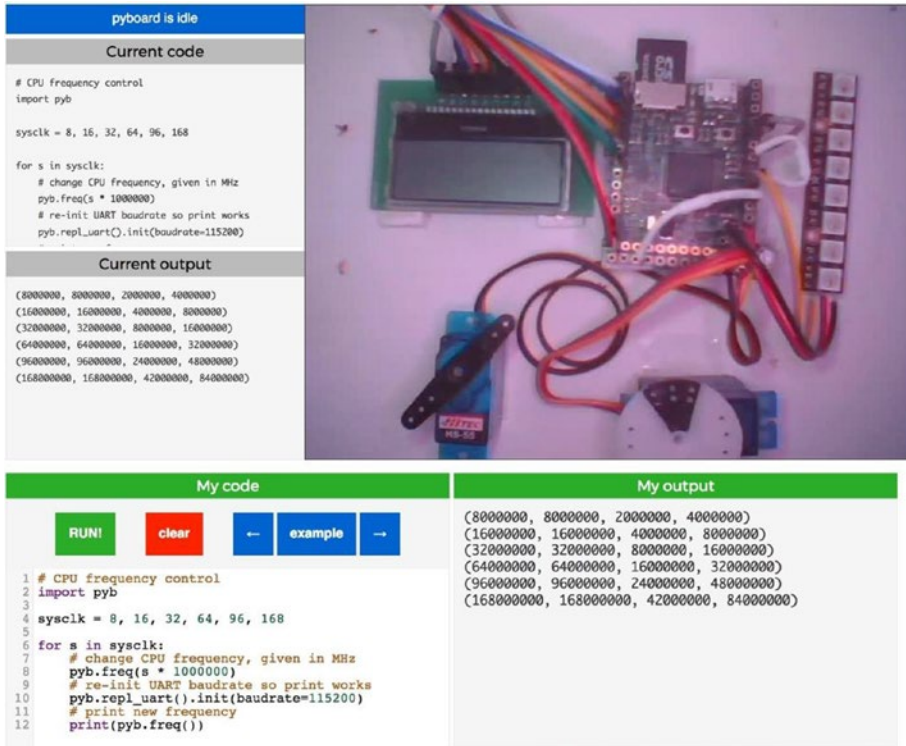


Figure 1-11. MicroPython Simulator – UART Calculations (courtesy of micropython.org)

While you may not fully understand the Python code in the examples, I suggest you take a few moments to try out the scripts so you can see how much fun it is (and how little code it requires) to control hardware with MicroPython. If nothing else, the simulator should give you an appetite to learn more about MicroPython.

Summary

The Internet of Things is an exciting new world for us all. Those of us young in heart but old enough to remember the Jetson’s TV series recall seeing a taste of what is possible in make-believe land. Talking toasters, flying cars that spring from briefcases, and robots with attitude notwithstanding, television fantasy of decades ago is coming true. We have wristwatches that double as phones and video players, we can unlock our cars from around the world, find out if our dog has gone outside, and even answer the door from across the city. All of this is possible and working today with the advent of the IOT.

In this chapter, we discovered what the IOT is and saw some examples of well-known IOT solutions. We also discovered how Microsoft is opening doors for Windows users by expanding its Windows 10 operating system to the IOT via the Raspberry Pi hardware. This is a very exciting opportunity for people who do not want to learn the nuances of a Linux-based operating system to explore the world of hardware and IOT from a familiar and well-understood platform.

In the next chapter, we will discover more about MicroPython including how to program in MicroPython. As you will see, it is not difficult. We will then explore the hardware we will use to run MicroPython in more detail in Chapter 3 to complete our tour of getting started with MicroPython for the IOT.

CHAPTER 2



Introducing MicroPython

Now that we have learned more about the Internet of Things and saw a few demonstrations of MicroPython, it is time to learn more about MicroPython – how we can get started, how it works, and examples of what you can do with your own MicroPython board.

Learning MicroPython is very easy even for those who have not had any programming experience. Indeed, all you need to learn MicroPython is a bit of patience and a little time to get used to the syntax and the mechanisms unique to working with MicroPython, the boards, and the electronics. As we will see, there is a lot you can do with just a little knowledge.

In this chapter, we will learn more about MicroPython including an overview of how to get started with one of the most popular boards. Don't worry if you don't have a board yet; the examples in this chapter are intended to give you a taste of what you can do rather than a detailed tutorial. That being said, we will see a detailed tutorial for the board used in this chapter as well as other boards in Chapter 3. We will also explore programming Python in more detail in Chapter 4.

■ **Tip** I refer to the microprocessor boards that run MicroPython natively or can be loaded with MicroPython binaries as “MicroPython-compatible boards,” or “MicroPython boards.”

Let's start with a look at what MicroPython is, including why it was created and how to get started.

Getting Started

The use of the Python language for controlling hardware has been around for some time. Users of the Raspberry Pi, pcDuino, and other low-cost computers and similar boards have had the advantage of using Python for controlling hardware. In this case, they used full versions of the Python programming language on the native Linux-based operating system.

However, this required special libraries built to communicate with the hardware. These libraries were designed to interface with the general-purpose input output (GPIO) pins. The GPIO pins normally appear on the board in one or more rows of male pins on the board. Some boards used female header pins.

While these boards made it possible for those who wanted to develop electronics projects, it required users to buy the board as well as peripherals like a keyboard, mouse, and monitor. Not only that, but users also had to learn the operating system. For those not used to Linux, this can be a challenge in and of itself.

You may be wondering about microcontrollers such as the wildly popular Arduino (arduino.cc) or Espressif, also known as ESP boards (espressif.com). For those boards, you must use a C-like language¹ to program them, which may be more than some are willing to learn.

The vision for MicroPython was to combine the simplicity of learning Python with the low cost and ease of use of microcontroller boards, which would permit a lot more people to work with electronics for art and science projects. Beginners would not have to learn a new operating system or learn one of the more complex programming languages. MicroPython was the answer. Figure 2-1 shows the MicroPython logo in the form of a skill badge (an iron-on patch)² from Adafruit.



Figure 2-1. MicroPython Logo Skill Badge (courtesy of adafruit.com)

That's pretty cool, isn't it? It's a snake (a python) on an integrated circuit (chip). You can order this skill badge from Adafruit at www.adafruit.com/products/3271. If you don't have anything to attach the patch to, Adafruit also stocks a nifty MicroPython sticker (www.adafruit.com/products/3270). I recommend getting one of these and displaying it proudly when you finish the book.

¹Some may say C++ and I suppose there is some truth in that, but they're more like C for general, basic use.

²Yes, I have one.

Origins

MicroPython³ was created and is maintained by Damien P. George, Paul Sokolovsky, and other contributors. It was designed to be a lean, efficient version of the Python 3 language and installed on a small microcontroller. Since Python is an interpreted language and thus slower (in general) than compiled languages, MicroPython was designed to be as efficient as possible so that it can run on microcontrollers that normally are slower and have much less memory than a typical personal computer.

COMPILED VS. INTERPRETED

Compiled languages use a program, called a compiler, to convert the source code from a human readable form to a binary executable form. There are a few steps involved in this conversion but in general, we take source code and compile it into a binary form. Since it is in binary form, the processor can execute the statements generated directly without any additional steps (again, in general).

Interpreted languages on the other hand, are not compiled but instead are converted to binary form (or an intermediate binary form) on-the-fly with a program called an interpreter. Python 3 provides a Python executable that is both an interpreter as well as a console that allows you to run your code as you type it in. Python programs run one line of code at a time, starting at the top of the file.

Thus, compiled languages are faster than interpreted languages because the code is prepared for execution and does not require an intermediate, real-time step to process the code before execution.

Another aspect is that microcontroller boards like the Arduino require a compilation step that you must perform on your computer and load the binary executable onto the board first. In contrast, since MicroPython has its interpreter running directly on the hardware, we do not need the intermediate step to prepare the code; we can run the interpreted language directly on the hardware!

This permits hardware manufacturers to build small, inexpensive boards that include MicroPython on the same chip as the microprocessor (typically). This gives you the ability to connect to the board, write the code, and execute it without any extra work.

You may be thinking that to reduce Python 3 to a size that fits on a small chip with limited memory that the language is stripped down and lacking features. That can't be further than the truth. In fact, MicroPython is a complete implementation of the core features of Python 3 including a compact runtime and interactive interpreter. There is support for reading and writing files, loading modules, interacting with hardware such as GPIO pins, error handling, and much more. Best of all, the optimization of Python 3 code allows it to be compiled into a binary requiring about 256K of memory to store the binary and runs with as little as 16k of RAM.

³Copyright 2014–2017, Damien P. George, Paul Sokolovsky, and contributors. Last updated on March 5, 2017.

However, there are a few things that MicroPython doesn't implement from the Python 3 language. The following sections give you an idea of what you can do with MicroPython and what you cannot do with MicroPython.

MicroPython Features

The biggest feature of MicroPython is, of course, it runs Python. This permits you to create simple, efficiently specified, and easy-to-understand programs. That alone, I think, is its best advantage over other boards like the Arduino. The following lists a few of the features that MicroPython supports. We will see these features in greater detail throughout this book.

- *Interactive Interpreter:* MicroPython boards have built in a special interactive console that you can access by connecting to the board with a USB cable (or in some cases over WiFi). This console is called a read-evaluate-print loop that allows you to type in your code and execute it one line at a time. It is a great way to prototype your code or just run a project as you develop it.
- *Python Standard Libraries:* MicroPython also supports many of the standard Python libraries. In general, you can expect to find MicroPython supports more than 80% of the most commonly used libraries. These include parsing JavaScript Object Notation (JSON),⁴ socket programming, string manipulation, file input/output, and even regular expression support.
- *Hardware-Level Libraries:* MicroPython has libraries built in that allow you to access hardware directly either to turn on or off analog pins, read analog data, read digital data, and even control hardware with pulse-width modulation (PWM) – a way to limit power to a device by rapidly modulating the power to the device. For example, making a fan spin slower than if it had full power.
- *Extensible:* MicroPython is also extensible. This is a great feature for advanced users who need to implement some complex library at a low level (in C or C++) and include the new library in MicroPython. Yes, this means you can build in your own unique code and make it part of the MicroPython feature set.

To answer your question, “What can I do with MicroPython?,” the answer is quite a lot! You can control hardware connected to the MicroPython board, write code modules to expand the features of your program storing them on an SD card for later retrieval (just like you can in Python on a PC), and much more. The hardware you can connect to include turning LEDs on and off, drive servos, read sensors, and even display text on LCDs. Some boards also have networking support in the form of WiFi radios. Just about anything you can do with the other microcontroller boards, you can do with a MicroPython board.

⁴<https://www.json.org/>

However, there are a few limitations to running MicroPython on the chip.

MicroPython Limitations

The biggest limitation of MicroPython is its ease of use. The ease of using Python means the code is interpreted on-the-fly. And while MicroPython is highly optimized, there is still a penalty for the interpreter. This means that projects that require a high degree of precision such as sampling data at a high rate or communicating over a connection (USB, hardware interface, etc.) may not run fast enough. For these areas, we can overcome the problem by extending the MicroPython language with optimized libraries for handling the low-level communication.

MicroPython also uses a bit more memory than other microcontroller platforms such as the Arduino. Normally, this isn't a problem but something you should consider if your program starts to get large. Larger programs that use a lot of libraries could consume more memory than you may expect. Once again, this is related to the ease of use of Python – another price to pay.

Finally, as mentioned previously, MicroPython doesn't implement all the features of all the Python 3 libraries. However, you should find it has everything you need to build IOT projects (and more).

ARE MY PYTHON SKILLS APPLICABLE TO MICROPYTHON?

If you've already learned how to program with Python, you may be expecting to see something that stands out as different or even odd about MicroPython. The good news is, your Python skills are all you need to work with MicroPython. Indeed, MicroPython and Python use the same syntax; there isn't anything new to learn. As you will see in the next few chapters, MicroPython implements a subset of Python libraries but still is very much Python.

What Does MicroPython Run On?

Due to the increasing popularity of MicroPython, there are more options for boards to run MicroPython being added regularly. Part of this is from developers building processor- and platform-specific compiled versions of MicroPython that you can download and install on the board. This is the fastest growing category.

There are two categories of boards you can use to run MicroPython. First are the boards that have MicroPython loaded from the factory and run only MicroPython. These include the Pyboard (the original MicroPython board) and WiPy. Next are the boards that have available firmware options to install MicroPython on the board including the ESP8266, Teensy, and more. We will see more about these boards in the next chapter.

Next, let's explore Python from our PC in the next section to give you an idea what the language is like and an opportunity to try it out yourself without needing a MicroPython board.

Experimenting with Python on Your PC

Since MicroPython is Python (just a bit scaled down for optimization purposes), you can run Python on your PC and experiment with the language. I recommend loading Python on your PC even if you already have a MicroPython board. You may find it more convenient to try out things with your PC since you can control the environment better. However, your PC won't be able to communicate with electronic components or hardware like the MicroPython boards so while you can do a lot more on the PC, you can't test your code that communicates with hardware. But you can test the basic constructs such as function calls, printing messages, and more.

So, why bother? Simply, using your PC to debug your Python code will allow you to get much of your project complete and working before trying it on the MicroPython board. More specifically, by developing the mundane things on your PC, you eliminate a lot of potential problems debugging your code on the MicroPython board. This is the number one mistake novice programmers make – writing an entire solution without testing smaller parts. It is always better to start small and test a small part of the code at a time adding only those parts that have been tested and shown to work correctly.

All you need to get started is to download and install Python 3 (for example, Python 3.6.2 is the latest but new versions become available periodically). The following sections briefly describe how to install Python on various platforms. For specific information about platforms not listed here, see the Python wiki at: <https://wiki.python.org/moin/BeginnersGuide/Download>.

■ **Caution** There are two versions of Python available – Python 2 and Python 3. Since MicroPython is based on Python 3, you will need to install Python version 3, not Python version 2.

But first, check your system to see if Python is already installed. Open a terminal window (command prompt) and type the following command.

```
python --version
```

If Python is installed, you should see something like the following.

```
$ python --version
Python 3.6.0
```

If you saw a version like Python 2.7.3, there is still a chance you have Python 3 on your machine. Some systems have both Python 2 and Python 3 installed. To run Python 3, use the following command.

```
python3
```

If Python 3 is not installed or it is an older version, use the following sections to install Python on your system. You should always install the latest version. You can download Python 3 from www.python.org/downloads/.

Installing Python 3 on Windows 10

Most Windows machines do not include Python and you must install it. You can download Python 3 for Windows from the official Python website (<https://www.python.org/downloads/windows/>). You will find the usual Windows installer options for 32-bit and 64-bit versions as well as a web-based installer and a .zip format. Most people will use the Windows installer option, but if you must install Python manually, you can use the other options.

Once you download Python, you can launch the installer. For example, on my Windows 10 machine, I downloaded the file under the link named *Latest Python 3 Release – Python 3.6.0*. If you scroll down, you can find the installer you want. For example, I clicked on the installer for Windows 64-bit machines. This downloaded a file named `python-3.6.0-amd64.exe`, which I located in my Downloads folder and executed by double-clicking on the file.

Like most Windows installer installs, you can step through the various screens agreeing to the license, specifying where you want to install it and finally initiating the install. Figure 2-2 shows an example of the installer running.

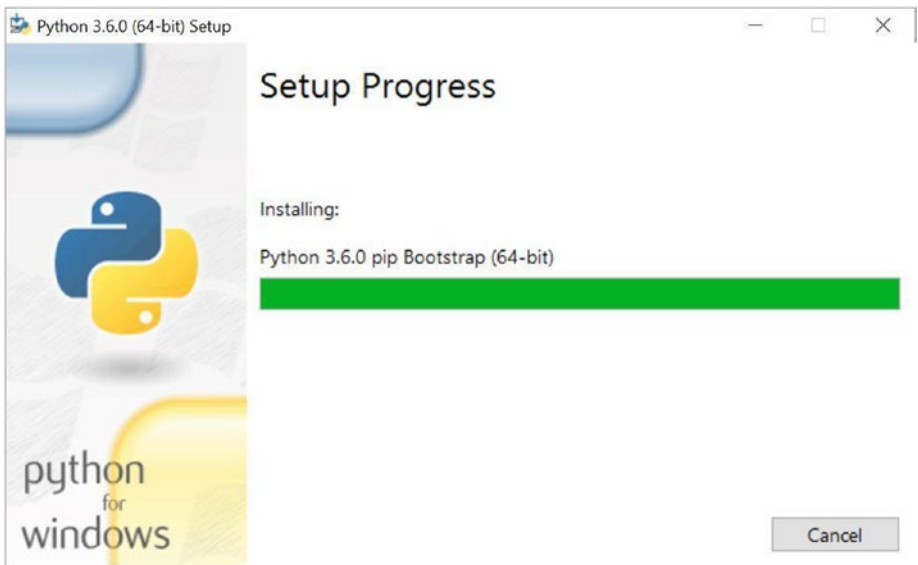


Figure 2-2. Installing Python on Windows 10

■ **Tip** If you get stuck or need more detailed instructions, see the excellent article at How-To Geek www.howtogeek.com/197947/how-to-install-python-on-windows/.

Once the installation is complete, you can try the test in the previous section to verify the installation. If you do not modify your PATH variable, you may need to use the Python console shortcut on the start menu to launch the console.

Installing Python 3 on macOS

If you are using macOS, you probably have Python installed since most releases of macOS install Python by default. However, if you were not able to run the Python version command above or it wasn't the correct version, you can still download the latest Python 3 from the Python website (<https://www.python.org/downloads/mac-osx/>). You will find several versions, but you should download the latest version available.

Once you download Python, you can launch the installer. For example, on my iMac, I downloaded the latest Python 3 file under the link named *Latest Python 3 Release - Python 3.6.0*. If you scroll down, you can find the installer you want. For example, I clicked on the installer for 64-bit machines. This downloaded a file named `python-3.6.0-macosx10.6.pkg`, which I located in my Downloads folder and executed.

Like most installers, you can step through the various screens agreeing to the license, specifying where you want to install it, and finally initiating the install. Figure 2-3 shows an example of the installer running.

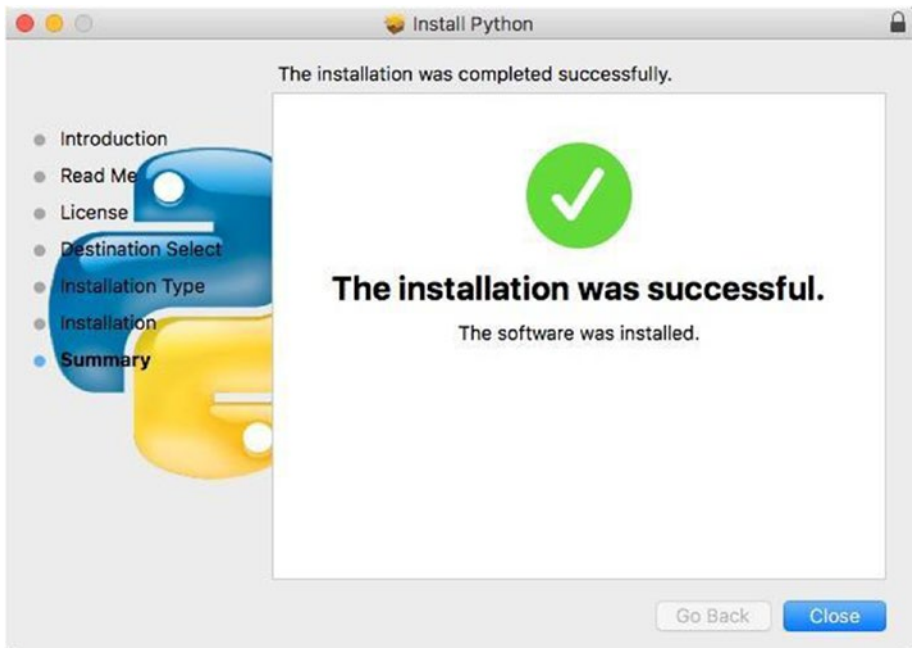


Figure 2-3. Installing Python on macOS

■ **Note** Depending on which version of macOS you are running, and how your security settings are set, you may need to change them to run the installer since it is not signed by an identified developer. See the *Security & Privacy* panel in your *System Preferences*.

Once the installation is complete, you can try the test in the previous section to verify the installation.

Installing Python 3 on Linux

If you are using Linux, the way you install Python will vary based on the platform. For instance, Ubuntu uses apt-get commands while other distributions have different package managers. Use the default package manager for your platform to install Python 3.6 (or later).

For example, on Debian or Ubuntu, we install the Python 3.6 package using the following commands. The first command updates the packages to ensure we have the latest package references. The second command initiates a download of the necessary files and installs Python.

```
sudo apt-get update
sudo apt-get install python3.6
```

■ **Tip** On Ubuntu releases prior to 16.10, you may need to add the apt repository before running the update command. Use the command, `sudo add-apt-repository ppa:jonathonf/python-3.6`, to add the repository and re-run the `sudo apt-get update` command then install Python 3.6. Other platforms may have similar solutions.

Figure 2-4 shows an example of what the installation looks like on Kubuntu (a variant of Ubuntu). Your platform may have slightly different output.

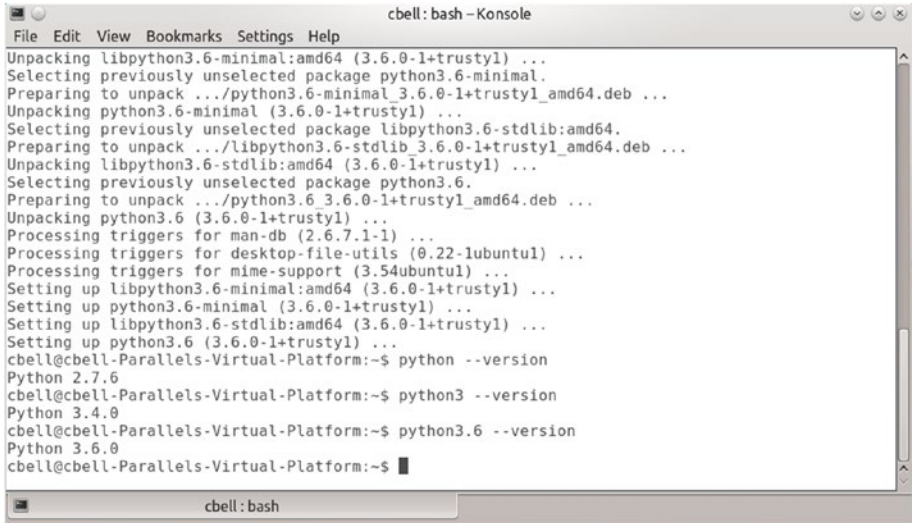


Figure 2-4. Installing Python on Kubuntu (Linux)

Notice in this install, I already had Python 2.7 and Python 3.4 installed. Once I installed Python 3.6, I could run all three simply by using the right interpreter: `python` for 2.7, `python3` for 3.4, and `python3.6` for 3.6. Your platform may have similar constraints if previous versions of Python are installed.

Once the installation is complete, you can try the test in the previous section to verify the installation.

Running the Python Console

Now let's run some tests on our PC. Recall we can open a Python console by opening a terminal window (command prompt) and entering the command `python` (or `python3` or `python3.6` depending on your installation). Once you see the prompt, enter the following code at the prompt (`>>>`). This code will print a message to the screen. The `\n` at the end is a special, non-printing character that issues a carriage return (like pressing *Enter*) to move to a new line.

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

When you enter this code, you will see the result right away. Recall the interpreter works by executing one line of code at a time – each time you press *Enter*. However, unlike running a program stored in a file, the code you enter in the console is not saved. Figure 2-5 shows an example of running the Python console on macOS. Notice I typed in a simple program – the quintessential “Hello, World!” example.

```

MacBook-Pro:~ cbell$ python3
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>> quit()
MacBook-Pro:~ cbell$

```

Figure 2-5. *Hello, World!*

To quit the console, enter the code, `quit()` as shown in the figure above.

While that demonstrates running Python from your PC, it is not that interesting. Let's see something a bit more complicated.

Running Python Programs with the Interpreter

Suppose your project required you to save data to a file or possibly read data from a file. Rather than try and figure out how to do this on your MicroPython board, we can experiment with files on our PC!

In the next example, I write data to a file and then read the data and print it out. Don't worry too much about understanding the code – just read through it – it's very intuitive. Listing 2-1 shows the code for this example. I used a text editor and saved the file as `file_io.py`.

Listing 2-1. File IO Example

```

# Example code to demonstrate writing and reading data to/from files

# Step 1: Create a file and write some data
new_file = open("log.txt", "w") # use "write" mode
new_file.write("1,apples,2.5\n") # write some data
new_file.write("2,oranges,1\n") # write some data
new_file.write("3,peaches,3\n") # write some data
new_file.write("4,grapes,21\n") # write some data
new_file.close() # close the file

# Step 2: Open a file and read data
old_file = open("log.txt", "r") # use "read" mode
# Use a loop to read all rows in the file
for row in old_file.readlines():
    columns = row.strip("\n").split(",") # split row by commas
    print(" : ".join(columns)) # print the row with colon separator
old_file.close()

```

I saved the code to a file to show you how you can execute your Python scripts using the Python interpreter using the following command. If you use Windows, you may need to modify the command like “python.exe file_io.py”.

```
python ./file_io.py
```

Listing 2-2 shows the results of running the script.

Listing 2-2. Output for the File IO Example

```
$ python ./file_io.py
1 : apples : 2.5
2 : oranges : 1
3 : peaches : 3
4 : grapes : 21
```

Notice the code changes the separator in the data by exchanging the comma as originally written to a space, colon, and another space. The code does this by splitting the line (string) read into parts by comma. Hence, the columns data contains three parts, which we use the `join()` method to rejoin the string and print it. Take a moment to read through the code and you will see these aspects. As you can see, Python is easy to read.

Now that we’ve experimented with Python on our PC, let’s see how to use MicroPython on a typical MicroPython board.

How MicroPython Works

Recall that MicroPython is designed to work on small microcontroller platforms. Some of these microcontroller platforms use a special chip that contains the MicroPython binaries (libraries, basic disk IO, bootstrapping, etc.) as well as the microcontroller, memory, and supporting components.

When you use a MicroPython board – like most microcontrollers – you must first write your code and load it onto the board. Most MicroPython boards have a USB flash drive that mounts when you connect it to your computer using a USB cable. This flash drive stores a couple of files you can modify to change its behavior. You can also copy your program (script file) to this drive for execution at boot time. We will see how to do this in a later chapter.

You can also use the MicroPython console, which is very much like the Python console we saw in the last section. The MicroPython console is called the run, evaluate, print loop, or REPL console. The console makes it very easy to get started and to debug (remove errors from) your program before loading it on the board for execution every time the board is powered on.

The Run, Evaluate, Print Loop (REPL Console)

If you have used another microcontroller board like the Arduino, you are likely familiar with some of the following. But if you haven't used such or have not used a terminal program, I provide all the steps you will need for each of the three major platforms: Windows, macOS, and Linux. The following sections walk you through connecting to the board for the first time.

I am using the Pyboard for these examples. You can use any other board that has MicroPython loaded by default, but be sure to check the vendor's documentation before starting to use the board for the first time. Some boards may require loading firmware before first use.

Connect the Board

To get started using the REPL console, connect the board to your computer using a USB to micro USB cable (typically), which provides power over the USB as well as connectivity for the console. Once the board boots (some boards can take up to 1-2 minutes to boot the first time), you can connect to the board using a terminal program. From there, we can use the REPL console in the same way we used the Python console.

■ **Tip** Some boards can take a few moments to boot so if you don't get a connection, wait a few moments and try again.

When the board finishes booting, you can browse the drive on the board using your computer. Once the Pyboard is connected using a USB cable, you should see the USB drive mount. For example, on Windows 10, you can use the file explorer to view the files on the drive. Figure 2-6 shows an example of the files on the Pyboard.

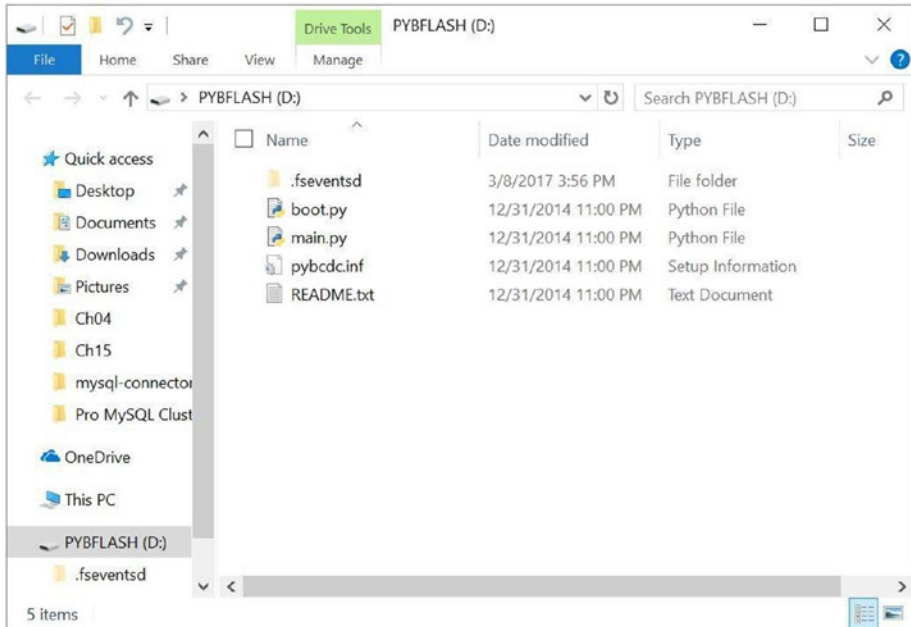


Figure 2-6. USB Drive (Pyboard)

Notice there are two files with a `.py` file extension: `boot.py`, which is executed on boot (hence the name); and `main.py`, which you can replace with your own program if you want to use the board independently. Again, we will see this in action in later chapters. There are also additional files such as the `README.txt` that contains some instructions to using the board; and the `pybcdc.inf`, which is a device driver installer for Windows.

■ **Tip** To use the Pyboard with Windows versions other than Windows 10, you may need to install the USB device driver. The device driver is included on the Pyboard's onboard flash drive in case you need it. If you have an older version of Windows or encounter problems using the terminal application, you can install the device driver as described in the excellent guide at <http://micropython.org/resources/Micro-Python-Windows-setup.pdf>.

Starting the REPL Console (Windows)

To connect to the REPL console using Windows, you will need a terminal program like PuTTY, which was originally developed by Simon Tatham (<http://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>). PuTTY is a simple terminal program that is very easy to use and made for the Windows platform.

To install PuTTY, select the appropriate .msi file (32- or 64-bit), download it, then double-click on the file to launch the installer. Follow the prompts to complete the installation. For example, I downloaded the file named `putty-64bit-0.68-installer.msi`.

Now that you have PuTTY installed, you must know the correct port to use. Open the *Device Manager* and navigate down the tree until you find the *Ports (COM & LPT)* entry and click to open it. Note that the Pyboard must be connected for this to work. You should see one or more entries in the subtree that indicate the COM ports (and printer ports) connected to your PC. The Pyboard will be listed as simply, *USB Serial Device (COMn)*, where n is a number like COM1, COM2, etc. For example, on my PC, it was listed as COM3. Figure 2-7 shows an example from my PC.

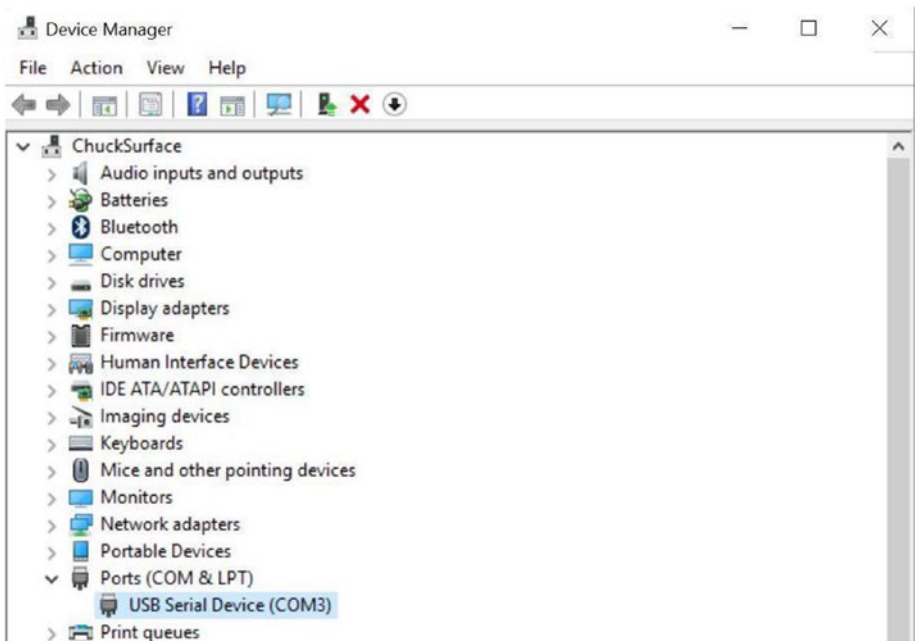


Figure 2-7. Finding the COM Port on Windows 10

Now we can open PuTTY and connect to the Pyboard. Use the shortcut on your Start menu to launch PuTTY or type PuTTY into the search box and click the entry when Cortana finds it. When PuTTY opens, you will see a dialog that you can use to connect via a terminal over the network. Since the Pyboard is connected via the COM port, we must click the small radio button marked *Serial*. We can then enter the COM port in the *Serial line* text box and set the speed to 115200. Figure 2-8 shows a properly configured PuTTY for connecting to the Pyboard on COM3 (as shown in the *Device Manager*).

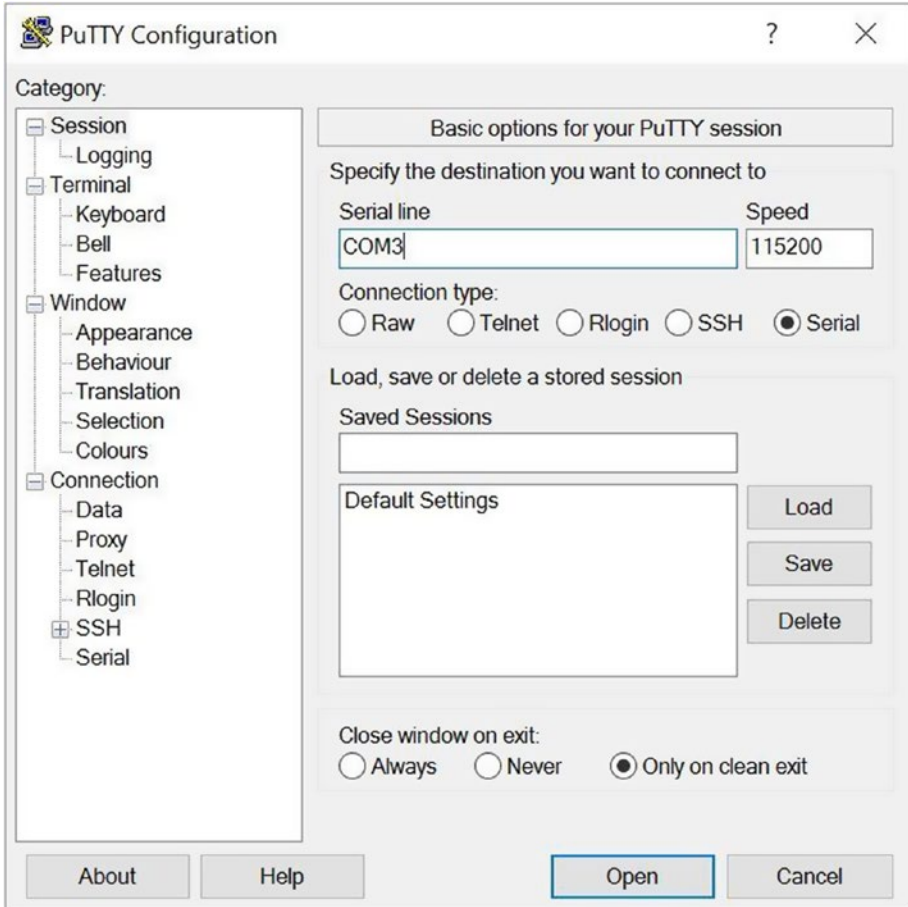


Figure 2-8. Connecting to Pyboard Using PuTTY

When ready, click the Open button. This will open a new terminal that will launch the REPL console as shown in Figure 2-9. I entered a simple statement to demonstrate that the console is working.

```

COM3 - PuTTY
MicroPython v1.8.2 on 2016-07-13; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> print("Hello from Windows 10!")
Hello from Windows 10!
>>>

```

Figure 2-9. *The REPL Console (Windows 10)*

■ **Tip** If you don't like the white (gray) on black color scheme, you can change them by clicking on *Colours* in the tree control. Tread lightly as you must set the values separately (there is no scheme concept).

Starting the REPL Console (macOS and Linux)

To connect using macOS or Linux, you can use the following command. The only thing you may need to do is locate the correct device. I demonstrate this with the first command that lists the devices on my macOS system. Do this after you've connected to the MicroPython board (Pyboard).

```

$ ls /dev/tty.usb*
/dev/tty.usbmodem1422
$ screen /dev/tty.usbmodem1422

```

Go ahead and plug in your board and then open a console following the instructions above. Once the console opens, enter the code shown below. Figure 2-10 shows an example of the REPL console running on a Pyboard.

```

MicroPython v1.8.2 on 2016-07-13; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> print("Hello from Pyboard!")
Hello from Pyboard!
>>> █

```

Figure 2-10. REPL Console (Pyboard)

Can you tell the difference from the Python console on your PC? Look closely. Except for the version statement at the top of the output, they're the same, which is very nice.

There is one oddity with the REPL console. The `quit()` doesn't work. To exit the console for some boards, you will need to reset the board or kill the connection. While this seems odd, I am certain it will be improved in the future, and recall we don't normally use the REPL console for running our projects; rather, we use it to test code so a little hiccup on quitting isn't a big issue.

■ **Caution** You should refrain from simply unplugging your MicroPython board. Some boards, like the Pyboard, present their base file system as a mountable USB drive. Disconnecting without ejecting the board can lead to loss of data.

Now it's time to take a tour of what we can do with MicroPython. The following section uses several example projects to show you what you can do with a MicroPython board. Once more, I introduce the examples with a minimal amount of explanation and details about the hardware. We will learn much more about the hardware in the next chapter.

Off and Running with MicroPython

If you're like me when encountering new technologies, you most likely want to get started as soon as you can. If you already have a Pyboard, you can follow along with the examples in this section and see a few more examples of what you can do with MicroPython. To keep things simple, we will see only those projects that run on the board without need of additional components and do not use the Internet. Figure 2-11 shows the Pyboard 1.1 from MicroPython.org (<https://store.micropython.org/>).

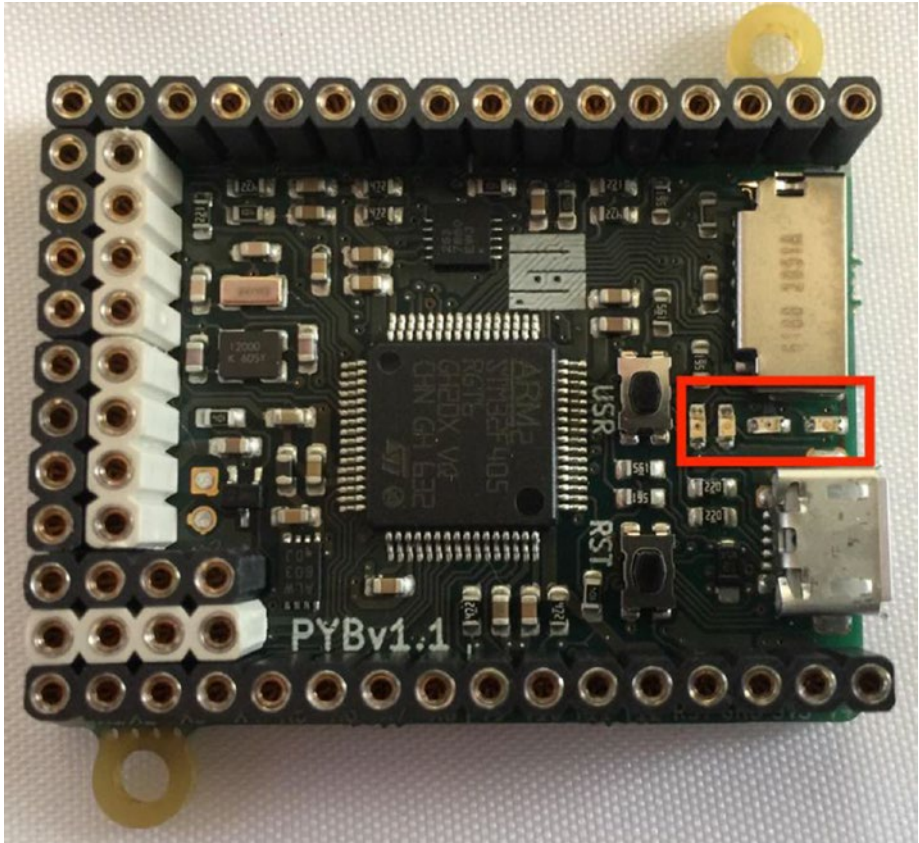


Figure 2-11. Pyboard with Headers

On the right side of the board is the micro USB connector and above that is a micro SD card reader. If you look between those connectors, you will see four, small LEDs as indicated with the square box. To the left of the LEDs is a button marked *USR*. These are the components we will interact with in the following examples.

■ **Note** The Pyboard shown here has GPIO headers (the rows of sockets surrounding the board) pre-soldered from MicroPython.org. If you order a Pyboard, I recommend getting this version, especially if you do not know how or do not want to solder the headers yourself.

What I like most about this board is it has an array of light emitting diodes (LEDs) and a button you can use to experiment with writing MicroPython projects. This makes the Pyboard an excellent first choice for beginners.

However, don't worry if you don't have a board yet. Again, I am including this section to help you learn more about what is possible through examples. Once we've discussed the details of the popular boards and how to program in MicroPython, we will dive into more complex projects that you can experiment with as you learn.

Unfortunately, the Pyboard does not offer any form of networking so it may be rather limited for use in IOT projects. Fortunately, there are ways to connect your Pyboard to the Internet.

■ **Note** There are a growing number of MicroPython compatible boards you can use. I chose the Pyboard to demonstrate MicroPython since the Pyboard is the easiest board to use and is relatively inexpensive. We'll learn more about the other choices in Chapter 3. If you want to buy a Pyboard to run these examples, you can, but you may need either a different board or a networking module for use with the more advanced projects that connect to the Internet.

Additional Hardware

While the following examples don't require any additional electronic components, I would like to introduce several key components that you will need later in this book. If you do not have these components, it is a good idea to order them now so that when you get to the example projects, you will have what you need.

In addition to a MicroPython board, I consider these components as must-have items for anyone wanting to learn how to work with electronics and MicroPython. These include a basic electronics kit that contains the most common components you will need when learning electronics, a breadboard, and jumper wires. I describe each of these in the following sections.

Basic Electronics Kit

The example projects in this book use several common electronics components such as LEDs, switches, buttons, resistors, etc. One of the biggest challenges when learning to work with electronics at the hobby level is what to buy. I've talked to some who have made numerous trips to the local electronics store to get what they need, seeming to never have the right components no matter what they buy.

Fortunately, electronics retailers have caught on to this problem and now offer a basic electronics kit that contains many of the more common components. Both Adafruit (adafruit.com/products/2975) and Sparkfun (sparkfun.com/products/13973) offer such kits. While you cannot go wrong with either kit, I like the Adafruit kit best since it has more components (e.g., more LEDs).

The Adafruit Parts Pal comes packaged in a small plastic case with a host of electronic components. Figure 2-12 shows the Parts Pal kit.

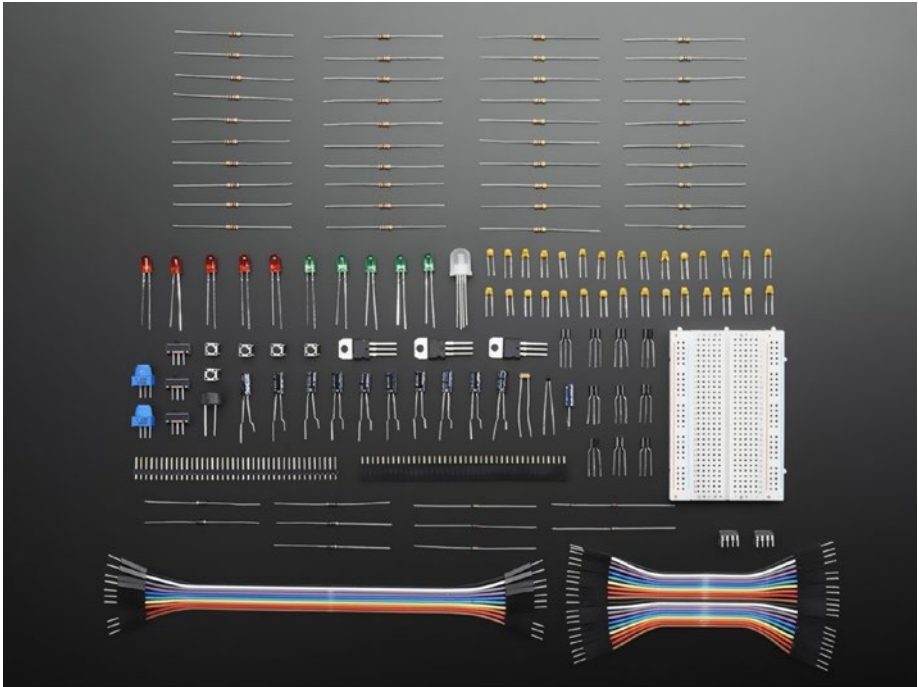


Figure 2-12. *Adafruit Parts Pal (courtesy of adafruit.com)*

The kit includes the following components: prototyping tools, LEDs, capacitors, resistors, some basic sensors, and more. In fact, there are more components in this kit than what you will need for many experiments. Better still, the kit costs only \$19.95, making it a good deal (and the case is a great bonus).

- 1x - Storage box with latch
- 1x - Half-size breadboard
- 20x - Male/male jumper wires - 3" (75mm)
- 10x - Male/male jumper wires - 6" (150mm)
- 5x - 5mm Diffused green LEDs
- 5x - 5mm Diffused red LEDs
- 1x - 10mm Diffused Common-Anode RGB LED
- 10x - 1.0uF Ceramic capacitors
- 10x - 0.1uF Ceramic capacitors
- 10x - 0.01uF Ceramic capacitors

- 5x - 10uF 50V Electrolytic capacitors
- 5x - 100uF 16V Electrolytic capacitors
- 10x - 560 ohm 5% axial resistors
- 10x - 1K ohm 5% axial resistors
- 10x - 10K ohm 5% axial resistors
- 10x - 47K ohm 5% axial resistors
- 5x - 1N4001 Diodes
- 5x - 1N4148 Signal Diodes
- 5x - NPN Transistor PN2222 TO-92
- 5x - PNP Transistor PN2907 TO-92
- 2x - 5V 1.5A Linear voltage regulator - 7805 TO-220
- 1x - 3.3V 800mA Linear voltage regulator - LD1117-3.3 TO-220
- 1x - TLC555 wide-voltage range, low-power 555 Timer
- 1x - Photocell
- 1x - Thermistor (Breadboard version)
- 1x - Vibration sensor switch
- 1x - 10K Breadboard trim potentiometer
- 1x - 1K Breadboard trim potentiometer
- 1x - Piezo buzzer
- 5x - 6mm Tactile switches
- 3x - SPDT slide switches
- 1x - 40-pin Break-away male header strip
- 1x - 40-pin female header strip

Breadboard and Jumper Wires

A breadboard is a special tool designed to allow you to plug in your electrical components and provide interconnectivity in columns so that you can plug the leads of two components into the same column and therefore make a connection. The board is split into two rows, making it easy to use IC in the center of the board. Wires (called jumper wires or simply jumpers) can be used to connect the circuit on the breadboard to the MicroPython board. You will see an example of this later in this chapter.

Fortunately, the Adafruit Parts Pal comes with a breadboard like the one shown in Figure 2-13. This shows a half-sized breadboard from Adafruit. This breadboard is called half since it is one-half the normal length of a standard breadboard. Best of all, it fits in the Parts Pal box.

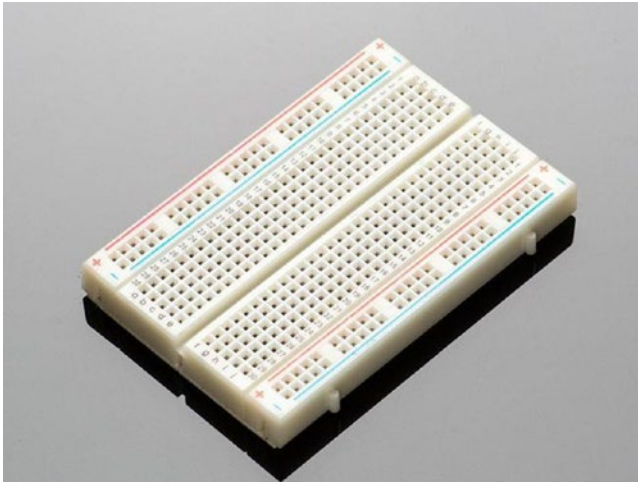


Figure 2-13. *Half-sized Breadboard (courtesy of adafruit.com)*

If you already have some components or decide to buy a different basic electronics kit that doesn't come with a breadboard, you can buy a breadboard separately from Adafruit (adafruit.com/products/64).

If your MicroPython board has male header pins instead of female header pins, you will need to get a different set of jumper wires. Once again, Adafruit has what you need. If you need male/female jumper wires, order the Premium Female/Male Extension Jumper Wires – 20 x 6 (adafruit.com/products/1954). Figure 2-14 shows a set of male/female jumper wires.

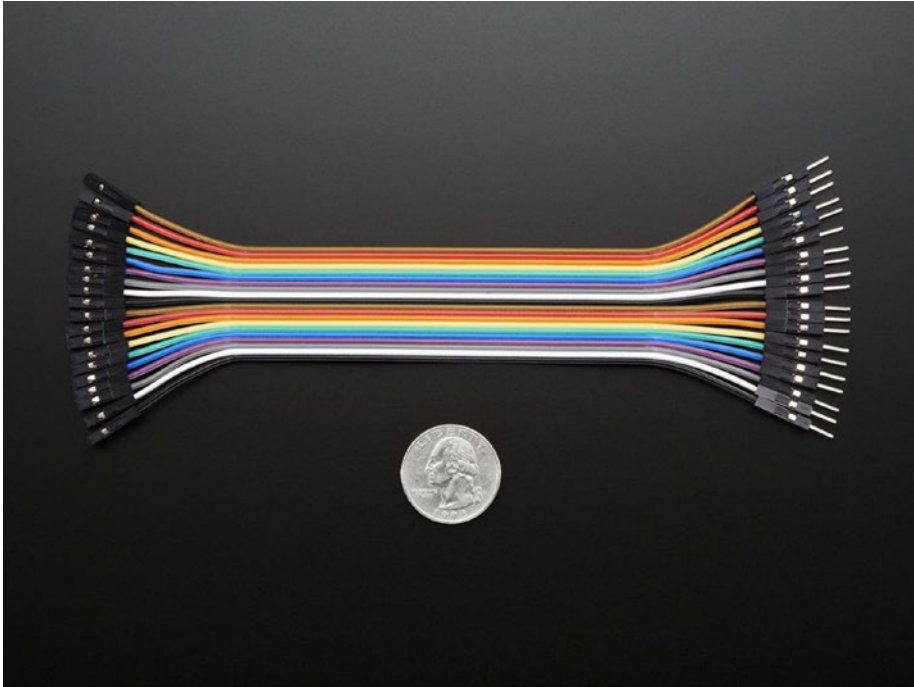


Figure 2-14. Male/Female Jumper Wires (courtesy of adafruit.com)

Now, let's see some hardware in action!

Example 1:

In this example, we will write code to turn on one of the LEDs on the board. If you orient the board as shown in Figure 2-11, the four LEDs are a different color arranged left to right as red, green, yellow (orange), and blue. If you're right-handed like me, you will probably orient the board so that the USB connector is on the left. In that case, the blue LED is on the far left.

Let's write some code to turn the blue LED on. Rather than simply turning it on, let's use a construct called a loop to turn it on and off every 250 milliseconds. Thus, it will flash rather quickly. Before I explain the code, let's look at the completed code. Listing 2-3 shows how the code would look. Don't worry; I'll explain each of the lines of code after the listing.

Listing 2-3. Blink the Blue LED

```

#
# MicroPython for the IOT
#
# Example 1
#
# Turn on the blue LED (4 = Blue)
#
# Dr. Charles Bell
#
import pyb                # Import the Pyboard library

led = pyb.LED(4)         # Get the LED instance
led.off()                # Make sure it's off first

for i in range(0, 20):   # Run the indented code 20 times
    led.on()             # Turn LED on
    pyb.delay(250)       # Wait for 250 milliseconds
    led.off()           # Turn LED off
    pyb.delay(250)       # Wait for 250 milliseconds

led.off()                # Turn the LED off at the end
print("Done!")          # Goodbye!

```

The first lines of code are comment lines. These are ignored by MicroPython and are a way to communicate to others what your program is doing. Feel free to skip the comment lines when entering the code into the REPL console.

Next is a line of code that is used to import the Pyboard hardware library (`pyb`). This library is specific to the Pyboard and makes available all the components on the board. The next two lines of code initialize a variable (`led`) by using the Pyboard library (`pyb`) to get the fourth LED (the blue one). This creates an instance of that object that we can use. In this case, we immediately turn the LED off by calling `led.off()`.

Next is the main portion of the code – a loop! In this case, it is a `for` loop designed to run the block of code below it as indicated by the indentation 20 times. The `for` loop uses a counter, `i`, and the values 0 through 19 as returned by the `range(0, 20)` function. Within the body of the loop (the indented portion), we first turn the LED on with `led.on()`, wait 250 milliseconds using the Pyboard `delay()` method, then turn the LED off again and wait another 250 milliseconds. Finally, we turn the LED off and print a message that we're done.

You can enter this code line by line into the REPL console. You can skip those that start with `#`. Once you enter the `for` loop statement, you will be given a row without the `>>>` prompt. This is normal. Type in the next line using two spaces (for indentation, which establishes the block for the loop). After the second `delay()` call, press Enter on a blank line to end the indentation block. Take a look at the Pyboard now. It should be blinking the blue LED. Remember, the REPL console, like the Python console, runs the code as you enter it. Figure 2-15 shows an example of the code running (I captured it with the blue LED on). If you want it to blink slower, just adjust the `delay()` calls to increase the value to 500 or even 1000.

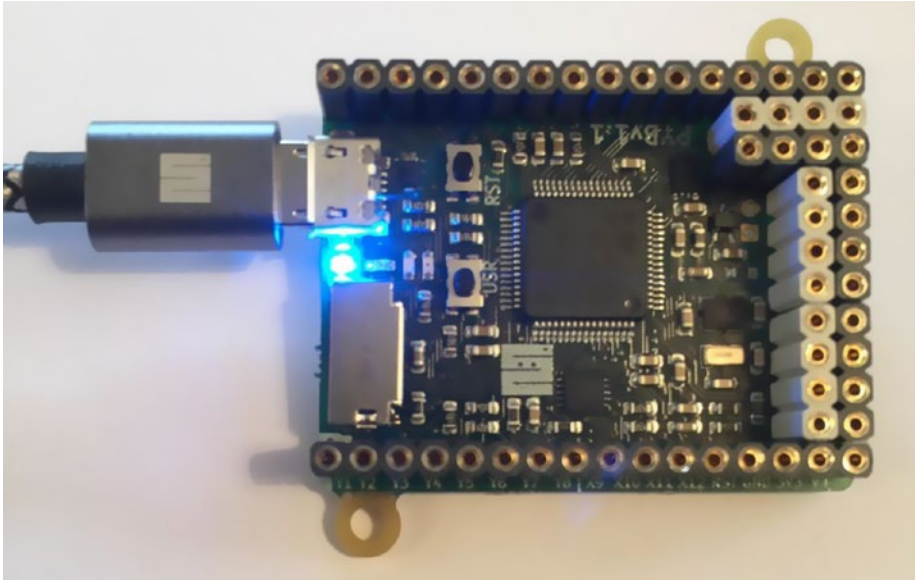


Figure 2-15. Running Example 1

■ **Note** On some platforms, such as macOS and Linux, you can use a text editor to enter the code then copy and paste it into the REPL console. However, this does not work for all platforms or terminal programs. Try it yourself.

If you want to run the code again or you make a mistake, you must first close the PuTTY window and then eject the USB drive. Once ejected, press the reset button (marked RST on the board) to reinitialize the REPL console.

Example 2:

Now, let's turn the LEDs on and off in sequence using a counting variable inside a different loop (a while loop). In this case, we must cycle through the LEDs one at a time, turning each on then off after a short delay. So, it is like the previous example but, as you will see, demonstrates a bit more complexity. Since it is more complex, I will walk through the code in sections before presenting the completed code.

Like all programs for the Pyboard, we begin by importing the Pyboard hardware library.

```
import pyb                # Import the Pyboard library
```

This example uses two loops. First, we loop through the LEDs turning them all off. The following shows how to do this. Notice I use a counting variable in the range of 1 to 4. Inside the body of the loop, I get the LED using the counting variable from the Pyboard hardware library saving it in a variable named `led` (`led = pyb.LED(j)`), then turn it off with `led.off()`. Turning the LEDs off at the start is a good habit and will take care of any events where you leave code running or interrupt code running so that one or more of the LEDs remains on. You can also fix this by resetting the board, but this method is preferred (and good practice).

```
for j in range(1, 5):    # Turn off all of the LEDs
    led = pyb.LED(j)    # Get the LED
    led.off()           # Turn the LED off
```

Next, we use another counting variable initialized to 1. We then start the while loop with an endless condition. That is, the expression following the while is always true. Inside the body of the loop, I do something similar to the last loop by getting the LED from the Pyboard hardware library, turn it on, wait 500 milliseconds, turn it off, and wait another 500 milliseconds. At the end of the body of the loop, I increment the counting variable (while loops do not increment the counting variable like for loops). If the counter gets to 5, I start over again with 1. Check the following code to see these elements.

```
while True:             # Loop forever
    led = pyb.LED(i)    # Get next LED
    led.on()            # Turn LED on
    pyb.delay(500)      # Wait for 1/2 second
    led.off()           # Turn LED off
    pyb.delay(500)      # Wait for 1/2 second
    i = i + 1           # Increment the LED counter
    if i > 4:           # If > 4, start over at 1
        i = 1
```

Listing 2-4 shows the completed code. Read through it a few times until you're convinced it will work.

Listing 2-4. Example 2: Blinking the LEDs

```
#
# MicroPython for the IOT
#
# Example 2
#
# Turn on the four LEDs on the board in order
#
# 1 = Red
# 2 = Green
# 3 = Orange
```

```

# 4 = Blue
#
# Dr. Charles Bell
#
import pyb          # Import the Pyboard library

for j in range(1, 5): # Turn off all of the LEDs
    led = pyb.LED(j) # Get the LED
    led.off()        # Turn the LED off

i = 1               # LED counter
while True:        # Loop forever
    led = pyb.LED(i) # Get next LED
    led.on()        # Turn LED on
    pyb.delay(500)  # Wait for 1/2 second
    led.off()       # Turn LED off
    pyb.delay(500)  # Wait for 1/2 second
    i = i + 1      # Increment the LED counter
    if i > 4:      # If > 4, start over at 1
        i = 1

```

When you run the code, you will see the LEDs turn on in sequence starting with the red, then green, yellow (orange), and blue. Since the loop does not end, it will keep blinking the LEDs until you reset the board (remember to eject the drive first). See Figure 2-16.

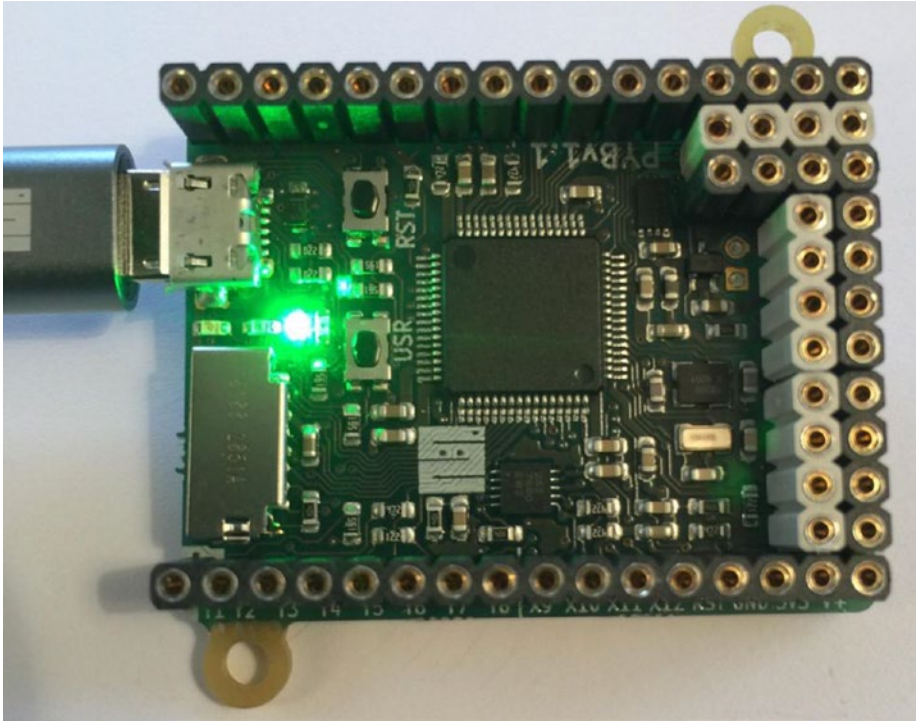


Figure 2-16. Running Example 2

Example 3:

This example demonstrates how to use the button on the board. It also demonstrates how to use an interrupt (called a callback function). That is, we create a function in our code then tell MicroPython to execute that function when an interrupt occurs – in this case, when the button is pressed. Since this example is also complex, I will walk you through the code.

First, we import the Pyboard hardware library and set a variable for the green LED (number 2 in the list) and turn it off.

```
import pyb                # Import the Pyboard library
led = pyb.LED(2)         # Get LED instance (2 = green)
led.off()                # Make sure the LED is off
```

Next, we define a function. It's rather easy. We just use the `def` directive and give the function a name. Let's call it `flash_led`. Inside this function we flash the LED quickly (100 millisecond delay) 25 times. We've already seen code to do this in the previous examples.


```
def flash_led():
    for i in range(0, 25):
        led.on()
        pyb.delay(100)
        led.off()
        pyb.delay(100)
```

Next, we create another variable by getting the user button (called a switch in the library). We then use the method, `callback()` and pass in the name of the function we created. These two statements will establish the connection to run the `flash_led()` function when the button is pressed.

```
button = pyb.Switch()      # Get the button (switch)
button.callback(flash_led) # Register the callback (ISR)
```

Finally, we print a message that the code is ready for testing.

```
print("Ready for testing!")
```

Listing 2-5 shows the completed code.

Listing 2-5. Example 3: Using a Button

```
#
# MicroPython for the IOT
#
# Example 3
#
# Flash the green LED when button is pushed
#
# Dr. Charles Bell
#
import pyb                # Import the Pyboard library
led = pyb.LED(2)         # Get LED instance (2 = green)
led.off()                # Make sure the LED is off

# Setup a callback function to handle button pressed
# using an interrupt service routine
def flash_led():
    for i in range(0, 25):
        led.on()
        pyb.delay(100)
        led.off()
        pyb.delay(100)

button = pyb.Switch()    # Get the button (switch)
button.callback(flash_led) # Register the callback (ISR)

print("Ready for testing!")
```

Once you see, Ready for testing! in the REPL console, you can test it out. If you're timid or have an unusually high propensity for static electricity, use a non-conductive probe and press the button marked USR. Figure 2-17 shows an example of the button press calling the `flash_led()` method. Try it a few times until you're satisfied it works.

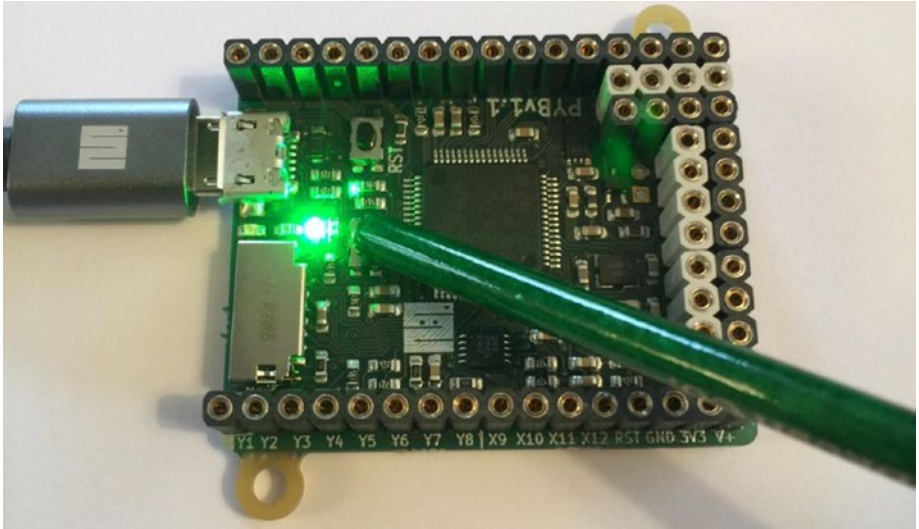


Figure 2-17. Running Example 3

Don't forget to eject the drive before unplugging or resetting your Pyboard.

Summary

MicroPython is a very exciting addition to the microcontroller world. For the first time, beginners do not need to learn a new operating system or a complex programming language like C or C++ to program the microcontroller. MicroPython permits people with some or even no programming experience to experiment with electronics and build interesting projects. Thus, MicroPython provides opportunities for more hobbyists and enthusiasts who just want to get their projects working without a steep learning curve.

In this chapter, we discovered the major features of MicroPython. We also discovered that MicroPython is based on the Python that we find on our PCs. We even tested Python on the PC to show the similarities. Best of all, we saw first-hand how MicroPython works on a microcontroller board. In this case, we used the original MicroPython board, the Pyboard, to demonstrate three examples of Python code written to exercise the components on the Pyboard.

In the next chapter, we will discover a host of hardware we can use to run MicroPython and build our IOT projects. As you will see, there are many options ranging from boards that have MicroPython already loaded allowing you to build your projects without extra setup, to common boards where you can load MicroPython yourself.

CHAPTER 3



MicroPython Hardware

Now that we've had a quick look at how MicroPython boards can be used, including a presentation on several forms of MicroPython projects, it's time to take a tour of the available boards and related hardware. As we will see, there are several boards that you can choose from, including those that have MicroPython already loaded so that they are ready to use out of the box; those that allow you to load MicroPython on the board; and those existing boards that, with a bit of work, can be worth investigating.

This chapter also includes how to get started using each of the boards discussed, including how to connect the board, load firmware, and more. You should read through the entire chapter because while many of the concepts are the same, the procedures may be slightly different from one board to the next. Also, the focus of the chapter is the MicroPython-ready boards. While I briefly discuss alternative boards that you can use, the information in those sections can be useful for using boards not listed in this chapter.

Before we jump into looking at the available boards, let's discuss a few best practices and other practical advice for using MicroPython boards. These apply to all the boards in this chapter and likely any emerging board, and thus you should have these in mind when choosing your board.

Getting Started with MicroPython Boards

While you could just buy one of these boards and jump right into your IOT project, there are some things you should consider and keep in mind when working with your board. Specifically, some boards may require updates, some may require assembling hardware (soldering), and others may have limitations to consider. I discuss these and more in this section. I also include some tips for using the boards on your PC.

Firmware Updates

Those boards that include MicroPython on the chip may require firmware updates. Firmware¹ is the term used to describe software on a chip. Most boards have special chips that allow you to update MicroPython (and other aspects) on your own, allowing you to keep up with changes by the vendor, MicroPython, and more.

¹<https://en.wikipedia.org/wiki/Firmware>

Loading the firmware is board specific, and thus you should refer to your vendor's documentation for how to update the firmware. For example, Pycom documents the update process for their WiPy board at <https://docs.pycom.io/chapter/gettingstarted/installation/firmwaretool.html>.

You should consider updating the firmware when you first receive the board or at any time you find you cannot access some library function, the operation fails, or when new hardware (add-on boards) are released. However, you should avoid updating your firmware too frequently.

My philosophy is simple: I only update firmware once when the board is new and only when something no longer works. If your project is working and you've not encountering any problems, there's no need to update it. One risk to updating the firmware too often is you could render your project useless if something changes in the firmware such as old libraries or hardware no longer supported, changes to the library that breaks your code, and other annoying issues. My only exception to this rule is if the vendor has fixed a bug or improved the user experience, in which case I would update the firmware. The bottom line is, if it works, don't mess with it!

■ **Tip** Update your firmware when your board is new and only when needed later.

A special mention is of those boards where you must load MicroPython yourself. Some of these boards may require you to update other parts of the system such as a bootstrap mechanism, loading MicroPython in non-volatile memory, etc. If you choose to use a board where you must load MicroPython, be sure to check the vendor's documentation for recommendations regarding updating the software on the board (may not be limited to just the firmware). Finally, some boards are new so their firmware may not be in beta or a release candidate and not production quality. In these cases, I like to update the firmware more frequently until the production version is released.

Networking Issues

If your board has networking capabilities, you should take extra care when learning how to use the board. Specifically, take your time to follow the vendor's instructions for setting up networking and connecting your board to your network. For example, some boards have a very specific mechanism for connecting to WiFi networks. Failure to correctly configure your network connection will lead to a lot of frustration, especially if your project is intended to produce data that is accessed over the network.

If your board has networking capabilities, I recommend reading through the documentation and running any examples that show you how to connect the board to your network. The time spent learning how to do this (and repeating it), will eliminate a lot of wasted time. Indeed, the number one mistake beginners make is hooking up their board, writing their code to send data to another system on the network only to discover it doesn't work. In this case, they've failed to ensure the networking portion works before using it for the first time.

This also applies to connecting the board to your PC. For those boards that allow you to access the on-board memory as a file system, this may not be an issue, but for other boards that require connecting over USB via special software, you should ensure you can connect to the board before trying to write your first Python program.

One Step at a Time!

Another very common mistake beginners make is sitting down and writing all their code in one pass without testing anything ahead of time. This creates a situation where if something doesn't work, it can be masked by a host of problems. For example, if there is some logic error or data produced is incorrect, it may cause other parts of the project to fail or produce incorrect results. This is made worse when the project doesn't work at all - there are too many parts to try and diagnose what went wrong. This often places beginners in a desperate situation of confusion and frustration. You students out there know exactly what I am talking about.

This can be avoided easily by building your project one step at a time. That is, build your project one aspect at a time. For example, if you're working with LEDs to signal something, get that part working first. Similarly, if you're reading data from a sensor, ensure you can do that correctly in isolation before wiring it all together and hoping it all works. Even the very experienced can make this mistake, but they are more equipped to fix it if something goes wrong (and they know better but it's a do as I say not as I do situation). We will build the examples in this book one step at a time. Some are small enough that there may be only one step, but the practice is one you should heed for any project you undertake.

Programming Tools

Some microcontroller vendors offer software development (programming) tools. Arguably one of the most successful is the Arduino Integrated Development Environment (IDE). The Arduino IDE provides all the tools you need to work (program) the Arduino - from writing your code to compiling to installing it on the board.

Some vendors offer software to use with their boards. The most promising includes the tools from pycom.io including the PyMakr plugin for popular programming editors² and the PyMate (<https://pycom.io/pymate/>) application for mobile devices. The PyMakr plugin is a special editor and tools for writing Python programs and installing them on the WiPy board. The PyMate application is an exciting new solution that allows you to work with your WiPy remotely, including connecting it to your network and reading devices displaying the data on your mobile device. We will see examples of these when we examine the WiPy board.

²The PyMakr plugin replaces the stand-alone PyMakr application. See <https://forum.pycom.io/topic/635/pymakr-time-of-death-09-02> for more information.

■ **Caution** The PyMakr plugins for Atom, Sublime, Visual Studio Code, and PyCharm are still under development. The intention is to replace them with the PyMakr desktop application. Currently, users are advised to use an FTP client such as FileZilla to upload code/projects to their devices.

If the vendor of your board offers software for use with their board, you should consider using it. However, most of what you will need to do for writing Python scripts and loading them on your board can be done without special software. In fact, there are several programming editors that support Python including Komodo Edit (www.activestate.com/komodo-ide/downloads/edit), PyCharm (www.jetbrains.com/pycharm/), and the PyDev IDE plugin for Eclipse (www.pydev.org/). The PyCharm IDE is offered as a community editor (free) as well as a paid application. Komodo Edit is also a free version of the more powerful, paid Komodo Edit IDE.

The editors I like most are Komodo Edit and PyCharm. Both have Python syntax highlighting – the color of the text changes to indicate syntax, strings, etc., as well as some can complete your statements and even automatically indent depending on what construct you are using.

Both are excellent choices for writing Python. Komodo Edit is a simple editor but it does its job very well. PyCharm is a Python IDE that does quite a bit more, including interactive debugging. While you won't be able to use some of the features with your MicroPython board, if you want to work with Python on your PC, you should consider using the IDE (or one like it). Figure 3-1 shows an example of using Komodo Edit to edit a Python example from the next chapter. While you cannot see the colored text highlights, notice that the editor provides link numbers, tabs for editing multiple files, and more.

```

1 #
2 # MicroPython for the IOT
3 #
4 # Class Example: Inheriting the Vehicle class to form a
5 # model of a pickup truck with maximum occupants and maximum
6 # payload.
7 #
8 # Dr. Charles Bell
9 #
10 from vehicle import Vehicle
11
12 class PickupTruck(Vehicle):
13     """This is a pickup truck that has:
14     axles = 2,
15     doors = 2,
16     __max_occupants = 3
17     The maximum payload is set on instantiation.
18     """
19     occupants = 0
20     payload = 0
21     max_payload = 0
22     __max_occupants = 3
23
24     def __init__(self, max_weight):
25         super().__init__(2,2)
26         self.max_payload = max_weight
27

```

Figure 3-1. Komodo Edit Example

While not required, it is highly recommended that you use an editor that includes Python syntax highlighting. Not only will it help you write the code, it can be helpful to help write more correct code. I find the code completion feature to be a real time saver. You don't have to choose any discussed here as none except PyMakr or PyMate are made for a specific board. Remember, writing Python for a MicroPython board is not different than writing a Python script for your PC – the syntax is the same.

Some Assembly Required

Some vendors such as Adafruit and George Robotics Limited/micropython.org (makers of the original Pyboard) offer boards with and without headers soldered. Not soldering the headers saves on production, and in some cases shipping costs, and makes the boards a bit cheaper. If you know how to solder (or know someone who does), you may be able to save a little by going with the boards without headers.

Another reason you may want a board without headers is if you want to install your board in a project enclosure or some other form of embedded installation. In this case, having the headers soldered may take up more space than you have or make the completed project a bit bulkier.

You may also encounter some add-on boards, breakout boards, or other discrete components that are not soldered with headers (or connectors). If you want to use these, you may have to solder the header or connector yourself. For example, most of the breakout boards from Adafruit (adafruit.com) and Sparkfun (sparkfun.com) do not come with the headers soldered.

GPIO Pins

We learned a bit about general purpose input output (GPIO) pins in the last chapter. One of the things that differentiates the various boards is how the GPIO pins are arranged and even how many pins are provided. While most boards support a list of pins including analog and digital pins, some boards provide fewer pins than others. Figure 3-2 shows a drawing that illustrates the GPIO pins available on the WiPy. You can find similar drawings (also called datasheets, mappings, or pinouts) from the manufacturer or vendor of your board.

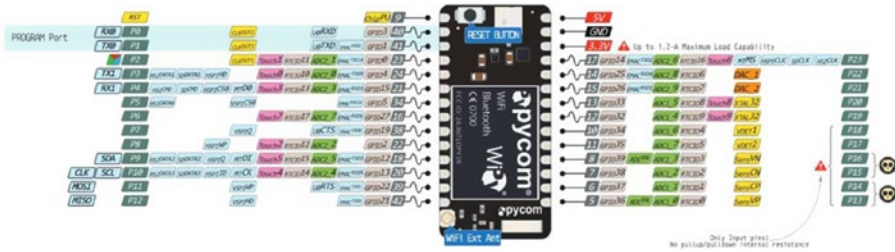


Figure 3-2. *WiPy GPIO Pins (courtesy of pycom.io)*

If your project requires several pins – analog or digital – you should plan to choose a board that can support the number of pins you will need. Fortunately, most will; however, some of the newer boards that you can load MicroPython on may not. For example, some versions of the ubiquitous ESP8266³ (a low-cost WiFi chip) may support only a few pins, making it less than ideal for projects that have many components.

Other Tips

This section includes several tips for things you may encounter when working with your MicroPython board. Consider these tips as things you should do before or during your experiments. Let's begin with the basics.

Visit the Community Forums

The very first thing you should do (even before buying a board) is to visit the community forums for the board(s) of your choice. Many of the vendors host and manage an online message forum for people to ask questions and for people in the community to provide their insights, answers, suggestions, and even help for those who get stuck.

Aside from reading this book, visiting the community forum is an absolute must. In fact, you should consider visiting the forum any time you have a question or problem with your board. You are most likely going to encounter someone who has been there

³<https://en.wikipedia.org/wiki/ESP8266>

before, struggling through the same (or very similar) issue as well as their (and others') suggestions for how to fix it.

The following lists forums for the more popular MicroPython boards. If you do not see your board here, Google for it next time you are online; then, if you like the forum, bookmark it for faster access.

- *Pyboard (and MicroPython) Forum*: <https://forum.micropython.org/>
- *Pycom (WiPy) Forum*: <https://forum.pycom.io/>
- *CircuitPython Forum*: <https://forum.micropython.org/viewtopic.php?f=16&t=2894>
- *General Python Forum (not specific to MicroPython)*: <https://www.python.org/community/>

■ **Note** I include more tips for interacting with community forums in Chapter 12.

Handle with Care!

You should consider your MicroPython board as a very sensitive device susceptible to electrostatic discharge (ESD). Unless you place your board in a case, you should handle your board carefully always placing it on a non-conductive surface before powering it on. ESD can be caused by many things (think back to when you were a child with sneakers on carpet). This discharge can harm the board. Always ensure you handle your board so that ESD is controlled and minimized.

Figure 3-3 shows an example of the Pyboard in a protective case as shipped from the vendor. This clamshell case is just a little larger than the board itself and closes with a snap to keep the board safe while it rattles around in the bottom of your kit.



Figure 3-3. *The Pyboard v1.1 with Headers in its Protective Case*

You should also never move the board when it is powered on. Why? The MicroPython boards have components soldered on with many pins exposed on both sides. Should any two or more of those pins touch something that conducts electricity, you can risk damaging the board.

Also, always store your board in an ESD safe container – one that is expressly made to store electronics. Your average, everyday, inexpensive plastic box should be avoided. However, if you do not have a container made for electronics, you can use static free bags to place the board in while it is being stored. Many of the boards and components you buy come in such packaging. So, don't throw it away!

You should take care to make sure your body, your workspace, and your project is grounded to avoid electrostatic discharge (ESD). The best way to avoid this is to use a grounding strap that loops around your wrist and attaches to an anti-static mat like these uline.com/BL_7403/Anti-Static-Table-Mats.

Finally, be extra careful when connecting your USB cable to your board. Most boards come with a micro USB connector, which is prone to breakage (more so than other connectors). In most cases, it is not the cable that breaks but the connector on the board itself. When this happens, it can be very difficult to repair (or may not be repairable). It is also possible that the cable itself will stop working or only work when you hold the cable in place. If this happens, try a new cable and if that fixes the problem, throw the old one away. If it does not fix the problem, it may be the connector on the board. Fortunately, extra care when plugging and unplugging the cable can avoid these issues. For example,

always plug the micro USB side first and use the full-size USB end to plug and unplug from your PC. The fewer number of times you use the micro USB connector, the less chance you have of damaging it.

PC Doesn't Recognize the SD Drive

One of the things that people most often struggle with when using their board for the first time with a micro SD card/drive is when the operating system does not recognize the on-board USB drive. Should your board come with a USB-ready drive (internal or external), and you cannot see it in your file explorer (or finder), there are several things you can do to fix the problem. First, if the board has a removable micro SD drive, make sure it is inserted properly and that it is formatted as FAT. If you supplied your own micro SD card, this is usually the problem. Reformat it as FAT and it should work.

On the other hand, if the board has a removable micro SD drive as well as an on-board drive, remove the micro SD card, and then try to connect the board to your PC again. Sometimes the external micro SD card takes precedence over the on-board drive.

If your board doesn't show at all or you are having problems connecting to it over USB, there are several possible causes. First, ensure your USB cable is the one the vendor recommends. If you're like me and have accumulated a veritable snake pit of USB cables, it is likely one or more of your "favorite" cables is a charging cable and not a data cable. That is, the cable can be used to charge devices but the pins (wires) for transmitting data are missing or not connected. Check your cable first.

The next likely situation is that your PC does not have the proper driver installed. Be sure to check the vendor's website for which drivers you need to install. Some require special drivers for certain operating systems. For example, you will need a driver for most boards if you use Windows 10.

Another possible issue is the board needs external power. I have seen this on some early prototype boards and in some cases for production boards (but none listed in this chapter). In this case, you need to power the board on first before connecting it to your PC.

Loose or Missing Jumpers

One of the things that can happen when using a board that has jumpers (small plastic connectors designed to complete a circuit) is the loss of one or more jumpers. Jumpers are used to enable or disable certain features. Figure 3-4 shows an example of a pinout for the jumpers on the WiPy Expansion Board.

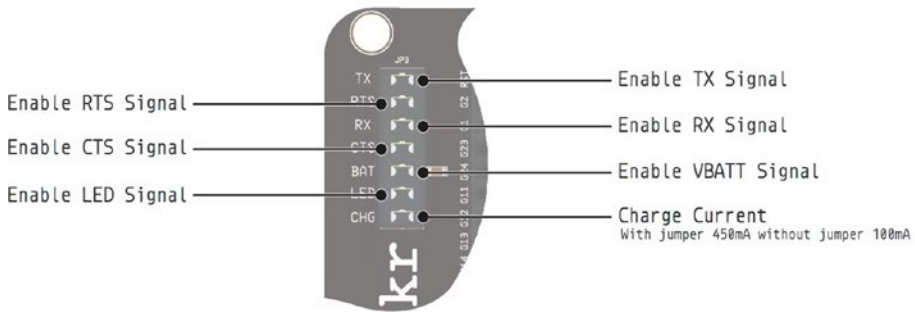


Figure 3-4. Jumpers on the WiPy Expansion Board

Jumpers are so small and that they can be easily lost or misplaced. If you lose your jumpers and need replacements, you can usually get some from a computer repair shop in your area. If there aren't any computer repair shops in your area, if you find a PC enthusiast who builds his own PCs, you should be able to get a few jumpers from them. Why? Because those guys normally have a ton of them on hand. When I managed a computer business, I couldn't give the things away. I still have hundreds lying around somewhere. The bottom line is you shouldn't have to buy them - someone around you, I am certain, has a few spares.

If one or more jumpers fall off, you can tighten them to make them harder to remove. To tighten the jumpers, use a pair of needle-nose pliers and gently (did I mention gently?) compress the female sockets. There's a bit of a trick to doing that. You can also gently bend the jumper pins slightly out at the top of the pin to make the jumpers fit more snugly. Of course, you can find that computer enthusiast and get a new one if it is still too loose or you damage it.

Some boards, such as the WiPy Expansion Board,⁴ have several jumpers that do not affect operation if they are removed. For instance, if you aren't using a lithium polymer (LIPO)⁵ battery, connecting via USB, etc., then you're fine if some of the jumpers are removed and you can use the unused one in place for one that is lost.

Now that we've had a look at some considerations for getting started with MicroPython boards, let's start our tour of the available boards starting with those that have MicroPython loaded and ready to use. Each section includes a brief overview of the board as well as notes on how to get started using the board and where you can buy the board.

⁴https://docs.pycom.io/pycom_esp32/downloads/exp_v03_pinout_v13.pdf

⁵https://en.wikipedia.org/wiki/Lithium_polymer_battery

MicroPython-Ready Boards

The first category of boards we will explore are those boards that come with MicroPython installed and ready to use. These boards are those that you do not need to install any software to use (though you may need to update the firmware periodically or install a driver on your PC). Thus, these boards are the best choices for those new to MicroPython and electronics in general. However, that doesn't mean the boards are not powerful – they are! In fact, the projects in this book are demonstrated using these boards and you will find they are more than adequate for most small- to medium-sized IOT projects.

The MicroPython-ready boards at the time of this writing are the Pyboard and WiPy. Both are good choices, but as we will see one is a bit easier to use for IOT projects.

Pyboard

The Pyboard MicroPython board was one of the first boards produced to host MicroPython. In some ways, it has set the standard for how MicroPython boards should be configured and how it operates. It was created by Damien George and began life in 2013 as a Kickstarter campaign to implement Python on a chip to host on microcontrollers. Pyboard and its accessory boards are produced and sold by George Robotics Limited (micropython.org).

Overview

The most noticeable feature of the board is its small size. The board measures approximately 40mm x 40mm and has two breakaway “ears” for mounting the board. Without the ears, the board is about 32mm by 40mm. That is, if you do not want to mount the board, you can break off the ears and reduce the size even more. The board has space for headers on three sides. One side contains a connector for a micro USB cable for connecting to your PC (or powering the board) as well as a micro SD drive. Figure 3-5 shows a close-up view of the Pyboard v1.1.

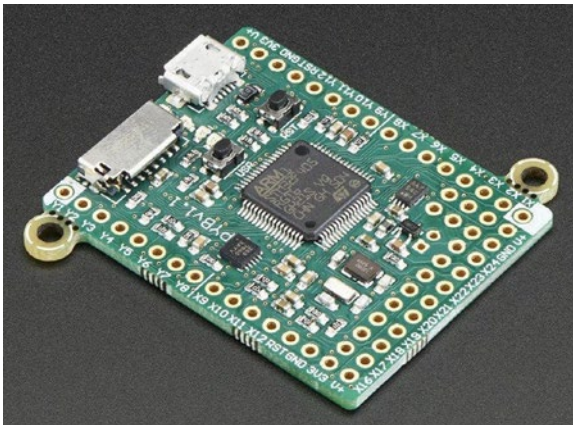


Figure 3-5. The Pyboard v1.1 without Headers (courtesy of adafruit.com)

Notice the small ears in the upper-left and lower-right corners of the board. These are perforated for easy removal. So far, they do not get in the way; so unless you have a smaller, tight fitting case, you can leave them attached.

You can also see two small buttons on the board near the micro USB connector. One is a user-defined button that you can program (USR) and the other is the reset button (RST) for resetting the board. The center of the board is dominated by the ARM processor chip.

Finally, there are four LEDs on the board located next to the micro SD card slot. These are used to communicate errors, power on, and can be turned on or off from your code!

More About the Hardware

Now let's learn a bit more about the hardware – more specifically, some of the details and specifications that you may not know but might be handy to know.

When you use the board with a micro SD card inserted (and the card is formatted as FAT), the board will boot with that drive mounted instead of the internal drive and it will show up when you connect your Pyboard to your PC. You will not see the internal drive, however. This is a source of confusion for many. If you want to use the internal drive to boot, you must create a folder on the micro SD named `flash` and an empty file inside that named `SKIPSD`. Do this and then disconnect and reconnect your board. After that, you can access the micro SD card from your script.

Recall the boot drive is used to hold the `boot.py` and `main.py` files; the `boot.py` file executes on boot and the `main.py` is called once the board is running. Changing these to add your custom programs is how you make the board boot into your project by default.

However, both drives are accessible from Python regardless of which one is used to boot. The internal drive is under the folder `/flash` and the micro SD card (when the `SKIPSD` trick is issued) is under the folder `/sd` (older versions of the board used `0:/` and `1:/` respectfully). Use these paths in your Python code to access the correct drive.

The board also supports alternative boot modes, which are initiated by using the USR button. To change the mode, press and hold the USR button when you plug in (or power on) your board. As you hold the button down, the LEDs on the board will illuminate as follows. Release the USR button when the board cycles to the mode you want.

- When the green LED is illuminated, the board boots normally.
- When the orange LED is illuminated, the board skips running the `boot.py` and `main.py` files on boot.
- When the green and orange LEDs are illuminated, the board resets the filesystem on the internal drive returning the `boot.py` and `main.py` to the factory content.

The last mode is very useful if you manage to brick your board. More specifically, if your board becomes unusable or won't start correctly, try this mode first to see if it fixes the problem.

Another use for the LEDs is indicating there is a Python error. If the red and green LEDs flash, it indicates there is an error in one of your Python scripts called from `main.py`. If all the LEDs are cycling on, it indicates there may be a hardware fault. Try powering off the board and resetting the filesystem. If that doesn't work, your board may be damaged.

The Pyboard is made up of several major components. I list them here for those interested in seeing what capabilities are supported. I begin with the microcontroller and memory specifications.

- STM32F405RG microcontroller
- 168 MHz Cortex M4 CPU with hardware floating point
- 1024KiB flash ROM and 192KiB RAM
- 3-axis accelerometer (MMA7660)
- Real-time clock with optional battery backup
- 24 GPIO pins
- (2) 12-bit digital to analog (DAC) converters, available on pins X5 and X6
- (3) 12-bit analog to digital converters, available on 16 pins, 4 with shielding
- 4 LEDs
- (1) micro SD slot
- (1) micro USB connector for communicating with your PC
- 1 reset (RST) and 1 user (USR) switch
- 3.3V LDO voltage regulator, capable of supplying up to 300mA to components with an input voltage range 3.6V to 10V
- DFU bootloader in ROM for easy upgrading of firmware

If you're interested in learning more about some of these components, the following are links to additional data and datasheets for the major components.

- Microcontroller (STM32F405RGT6): <http://www.st.com/en/microcontrollers/stm32f405rg.html>
- Accelerometer (Freescale MMA7660): <http://micropython.org/resources/datasheets/MMA7660FC.pdf>
- LDO Voltage Regulator (Microchip MCP1802): <http://micropython.org/resources/datasheets/MCP1802-22053C.pdf>

As mentioned previously, the Pyboard is available in several versions included with and without headers soldered, and earlier versions of the board (v1.0) that are somewhat slower and may have less hardware features. I recommend using the v1.1 boards for all your projects.

WHAT, NO INTERNET?

One of the features that you may expect to find is missing from the Pyboard. There is no networking capability. While it is true the latest firmware includes provisions for using two networking hardware (CC3000 and WizNet5000), you must buy modules (breakout boards) that support one of those chipsets/libraries to connect your Pyboard to the Internet (or your local network). Some may find using the Pyboard without networking is fine for most projects, but the projects later in this book will require connecting to the network. After all, this is an IOT book! So, if you intend to build an IOT project, you may want to consider a different board or check out the networking example in a later section for how to connect your Pyboard to your network.

If you want to power your board using an external power source or battery, you must ensure the power source is set to 3.6-10V. Connecting higher power can damage the board. Lower power can make the board unstable (it may fail to run correctly). You can connect power to (positive) to VIN and ground (negative) to GND. Figure 3-6 shows a close-up of where these pins are located near the micro USB connector.

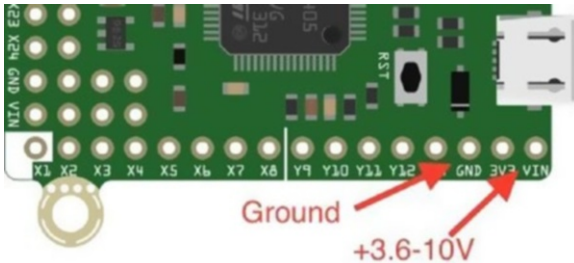


Figure 3-6. Pins for connecting external power (Pyboard)

■ **Tip** Always eject your drive before unplugging it or resetting the board.

Now, let's see the Pyboard in action starting with a simple walkthrough of connecting to our PC and running a simple Python program.

Getting Started with Pyboard

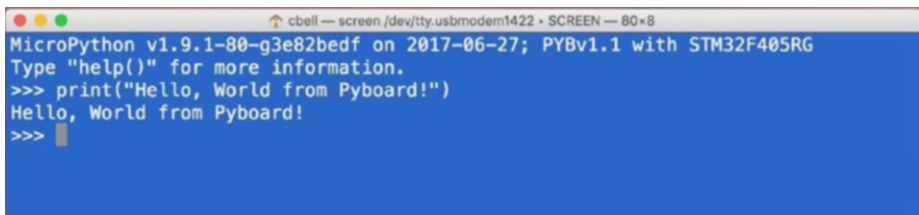
We discussed the REPL console in Chapter 2 including how to connect the board to your PC. Let's run through that again so we can be sure we understand how it works. To connect to your board, use a USB to micro USB cable connecting one end to your

Pyboard and the other end to your PC. What you should see after a few moments is a drive mounted named PYBFASH (or on Windows, a new drive letter appears in the file explorer tree).

■ **Tip** If you use Windows 10 and the Pyboard takes a very long time (more than 5 minutes) to show up as a USB drive, you need to update your firmware. In fact, this is a known issue and has been fixed in a recent firmware release.

You can then open the USB drive and modify the files there by editing them on the drive or by copying files to the drive. For example, if you want to modify the `main.py` script to include your own code or load your own script file, you can make the changes and then eject the drive and reset the board. Once you've done this, the code you added or script you've set to call will fire when the board boots. We will see how to do this in a later chapter.

You can also start the REPL console as we saw in Chapter 2. Figure 3-7 shows an example of the REPL console connected to the Pyboard. In this case, the code is simply printing a statement – the now cliché Hello, World! message.



```

MicroPython v1.9.1-80-g3e82bedf on 2017-06-27; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> print("Hello, World from Pyboard!")
Hello, World from Pyboard!
>>>

```

Figure 3-7. The REPL Console (Pyboard)

Loading the Firmware

Recall that we discussed the need to load firmware periodically. This applies to all boards regardless of whether they are MicroPython ready. This is because the version of MicroPython continues to be refined and defects are being repaired. Thus, to ensure you have the very latest updates, you should update your firmware at least once when you get the board and only when needed later such as when you discover a defect or a new hardware component is added to the library. Loading firmware on each board is a little different, but we will start with the Pyboard in this section in some detail then mention the differences in the procedure for the other boards in this chapter.

Before you get started, you should check the version of the firmware loaded. You can do that with the following Python statements. In this case, the code is being run on a Pyboard connected via the REPL console.

```
>>> import os
>>> os.uname()
(sysname='pyboard', nodename='pyboard', release='1.9.1', version='v1.9.1-80-g3e82bedf on 2017-06-27', machine='PYBv1.1 with STM32F405RG')
```

Here we see the firmware version is 1.9.1, which at the time of this writing was the latest available. It also shows the date the firmware was loaded as well as the name of the firmware file. Finally, we also see the board name.

You can download the firmware modules from <http://micropython.org/download/>. By way of example, the file used to update the Pyboard to the latest version now⁶ is named `pybv10-network-20170629-v1.9.1-95-g1942f0ce.dfu` (latest) and contains the networking hardware support. Figure 3-8 shows an excerpt of the website showing the file and its description. When you visit the site, you will find firmware for a wide variety of boards. Be sure to mark this page as you will need it if you want to use other boards.

Suitable for PYBv1.1 boards:

- `pybv11-20170629-v1.9.1-95-g1942f0ce.dfu` (latest)
- `pybv11-20170611-v1.9.1.dfu`
- `pybv11-20170526-v1.9.dfu`
- `pybv11-20170108-v1.8.7.dfu`
- `pybv11-20161110-v1.8.6.dfu`
- `pybv11-20161017-v1.8.5.dfu`

Suitable for PYBv1.1 boards, with network drivers for CC3000 and WIZ820io included:

- `pybv11-network-20170629-v1.9.1-95-g1942f0ce.dfu` (latest)
- `pybv11-network-20170611-v1.9.1.dfu`
- `pybv11-network-20170526-v1.9.dfu`
- `pybv11-network-20170108-v1.8.7.dfu`
- `pybv11-network-20161110-v1.8.6.dfu`
- `pybv11-network-20161017-v1.8.5.dfu`

Figure 3-8. Excerpt of Firmware Available for the Pyboard

⁶The version number may not change, but the date or build number may change frequently.

Notice the files are named with a file extension of `.dfu`, which stands for Device Firmware Update⁷ – a special binary format for firmware. You should download the latest version of the firmware. Be sure to choose the one that matches your board! For example, I chose the file under the heading *Suitable for PYBv1.1 boards, with network drivers for CC3000 and WIZ820io included* because I want to use my Pyboard with a networking module. You should choose the firmware that has the minimal set of support needed to help save on space (memory), but, so far, this has not been an issue.

Next, you will need a DFU programmer. If you use Windows, download the programmer from <http://www.st.com/en/development-tools/stsw-stm32080.html>. Choose the version that matches your system (32- or 64-bit) and install it. You may need to first register your request via email first.

If you use macOS or Linux, you can download a pure Python version of the DFU loaded from <https://github.com/micropython/micropython>. Click the *Clone or download* button then *Download Zip* button and save the file to your PC. This will download all the tools and utilities for use with MicroPython. Once downloaded, unzip the file and locate the file named `pydfu.py` in the tools folder. See <https://github.com/micropython/micropython/wiki/Pyboard-Firmware-Update> for more information about how to use this DFU programmer on Linux or macOS.

The next thing you must do is place the Pyboard in DFU mode. This is done using a jumper wire placed between the DFU and 3.3V pins as shown in Figure 3-9. The 3.3V pin is marked on the edge of the board and the DFU pin is the one next to it (not marked but located on the second row). Do this before you connect your board to your PC.

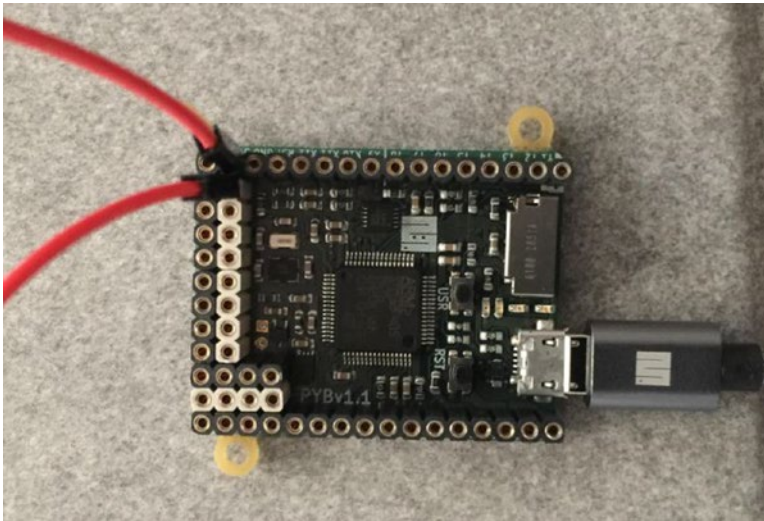


Figure 3-9. DFU Mode Jumper (Pyboard)

⁷https://en.wikipedia.org/wiki/USB#Device_Firmware_Upgrade

■ **Note** The first time you connect your board in DFU mode to your PC, Windows 10 may install a driver. If the driver install starts, select the automatic option and allow Windows to search for the driver. When that process is complete, you are ready to load the firmware.

Now let's see how to update the firmware using Windows 10. Once you've downloaded the firmware you want to load and have installed the DFU programmer, you can start it and then click the *Choose* button in the *Upgrade or Verify Action* section (see below) and search for the file you downloaded and then open it. Once the file is read, click the *Upgrade* button to start the update. The process will begin to update the board and display a complete message as shown in Figure 3-10.

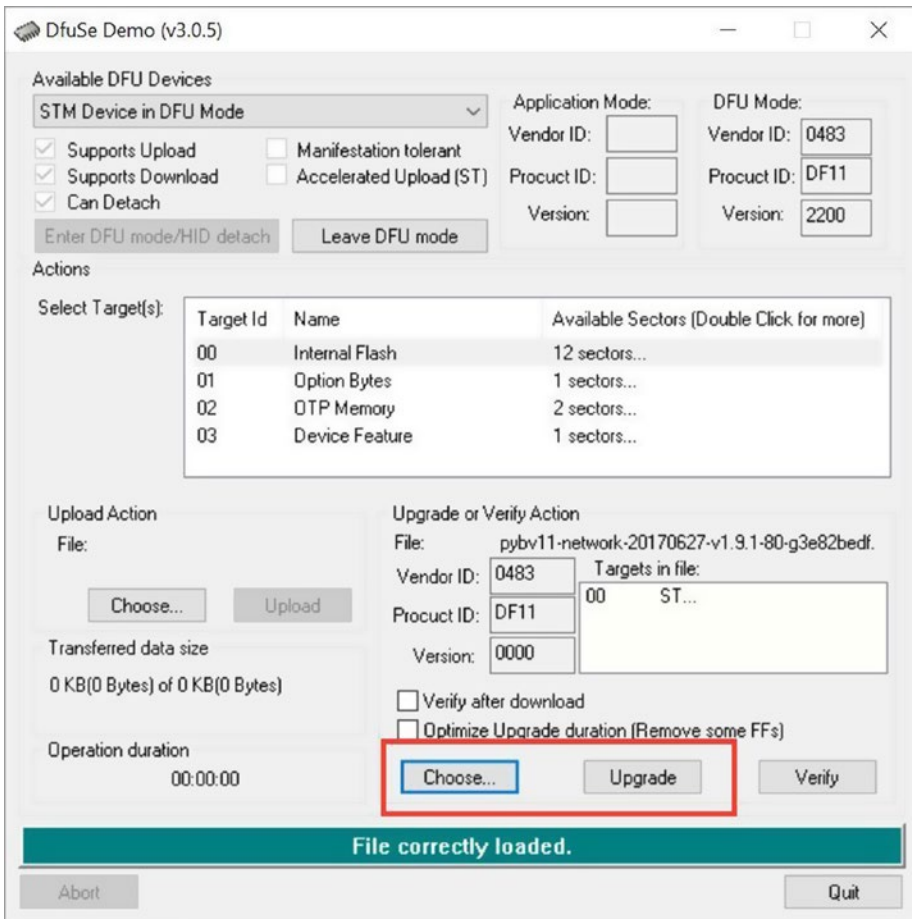


Figure 3-10. DFU Programmer (Windows 10)

When the process is complete, click *Quit* to close the DFU programmer, disconnect your board from your PC, and remove the jumper. The board will now boot with the newest firmware.

Networking with the Pyboard

This section will show you what you need to do to make your Pyboard a network enabled device. We will go through this briefly as only those who own a Pyboard will need to do this step as part of their setup and getting started and only when we get to the example projects that use the Internet.

First, you must purchase a network module. This is the hard part as the only two chipsets/libraries supported to date are the CC3000 (CC3K) for WiFi and WIZNET5000 (WIZNET5K) for Ethernet. Sadly, finding one of the CC3K modules can prove to be a challenge because those retailers that made them have updated to newer chips.

Fortunately, Adafruit made a CC3K module (and Arduino shield) that works. Unfortunately, they no longer make this board having replaced it with a better one that uses a different chipset (and thus is not compatible). You can find similar boards on popular Internet auction sites, but most vendors are in Asia so shipping will take a bit longer. Other places to look for these modules include DigiKey (digikey.com), Mouser (mouser.com), and Amazon (amazon.com), but stock is limited.

Whichever you buy, be sure it has the breakout pins available so you can connect it to your Pyboard. Look for pins labeled MISO, MOSI, CLK/SCK, IRQ, and VBAT_EN/VBEN. They should appear in a row on the module. Even if it isn't exactly like the one I will show you, it should work.

■ **Caution** Be sure you have loaded the firmware with networking support. Standard firmware does not include the networking modules.

To demonstrate that you can indeed connect your Pyboard to the Internet (you may find some posts that suggest it isn't possible), let's repurpose a CC3000 Arduino shield from Adafruit. Yes, this means we can use an Arduino shield without an Arduino! Many of the shields and modules for the Arduino can be used with other microprocessors. You just need to know how to connect the pins to the Pyboard. Don't worry, I will show you exactly how to do this. Figure 3-11 shows the shield without headers installed.

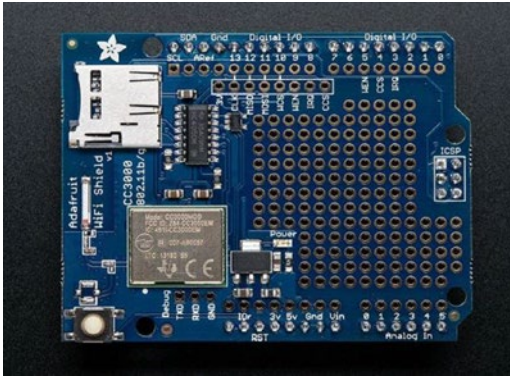


Figure 3-11. Adafruit CC3000 Arduino Shield (courtesy of adafruit.com)

One of the reasons Adafruit is such a fantastic resource is they include datasheets and how-to guides for all their products. They even maintain the links for discontinued products. After a brief review of their documentation, the pins needed to use the shield with the Pyboard are labeled on the shield itself and in the document.

The MicroPython documentation (<https://docs.micropython.org/en/latest/pyboard/library/network.html>) has an excellent guide for how to use the libraries supported. In this case, an examination of the sections for the CC3000 revealed the pins needed to connect the CC3000 module to the Pyboard. Fortunately, since the CC3000 shield was used with an Arduino, the shield had the Arduino headers added. We will use those to connect to the Pyboard.

Table 3-1 shows the correct pins on both boards. Simply use a male-to-female jumper to connect the pins on the Pyboard to the correct pin on the CC3000 shield. Connect the male side to the pin on the shield (which would normally plug into the Arduino) and the female side to the Pyboard pin. You will need a total of eight jumpers. Note that other modules may have a similar arrangement. The pins are labeled the same as shown in the table but may be numbered differently or arranged in a dedicated header.

Table 3-1. Mapping Pins from the Pyboard to the CC3000 Shield

Pyboard		CC3000 Shield	
Pin Name	Pin Number	Pin Name	Pin Number
V+		5V	
GND		GND	
Y3	Y3	IRQ	Digital 3
Y4	Y4	VBAT_EN	Digital 5
Y5	Y5	WCS	Digital 10
Y6	Y6	SCK	Digital 13
Y7	Y7	MISO	Digital 12
Y8	Y8	MOSI	Digital 11

Figure 3-12 shows a connection diagram for all the connections and Figure 3-13 shows what the connections will look like when you're done. It's a bit messy, but it works. Just be sure to place the shield and board on a non-conducting work surface.

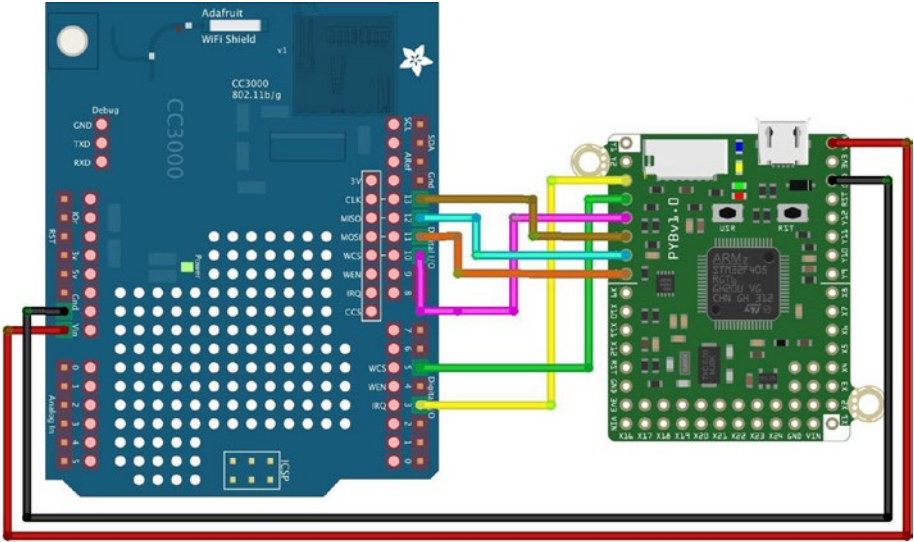


Figure 3-12. Connecting the CC3000 to the Pyboard

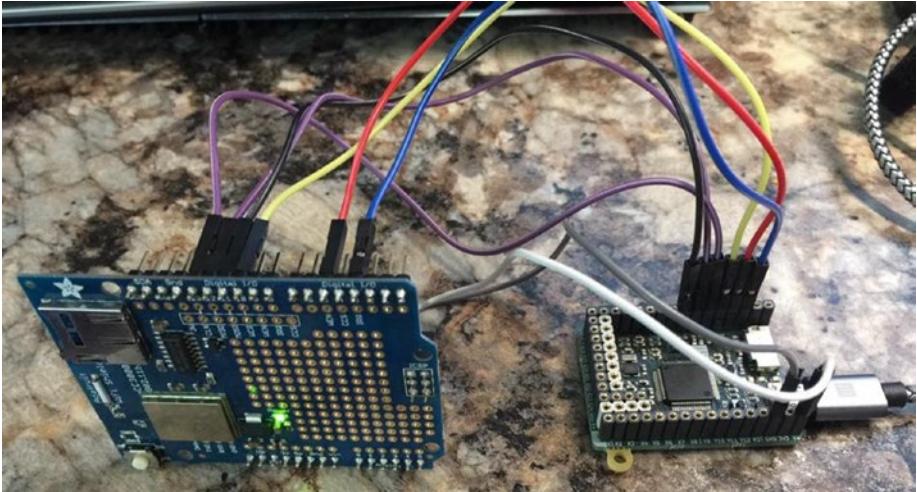


Figure 3-13. Pyboard with Adafruit CC3000 Arduino Shield

■ **Note** If you examine the CC3000 shield carefully, you will see there are holes for a separate header that you could use if you'd rather not use the Arduino headers.

If you have a Pyboard and a CC3000 shield, go ahead and make the connections. Just be sure your Pyboard is not connected to your PC first. Once all the connections are made, you can connect your Pyboard to your PC.

Now, let's write some code to test the module and connect our Pyboard to the Internet. Since this is a WiFi shield, we will need to know the name of your WiFi access point (router) and the password. The code we need to use is shown in Listing 3-1.

Listing 3-1. Using the CC3000 Module on a Pyboard

```
# connect/ show IP config a specific network interface
import network
import pyb

def test_connect():
    nic = network.CC3K(pyb.SPI(2), pyb.Pin.board.Y5, pyb.Pin.board.Y4, pyb.
Pin.board.Y3)
    nic.connect('YOUR_ROUTER_HERE', 'YOUR_ROUTER_PASSWORD_HERE')
    while not nic.isconnected():
        pyb.delay(50)
    print(nic.ifconfig())

    # now use usocket as usual
    import usocket as socket
    addr = socket.getaddrinfo('micropython.org', 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(b'GET / HTTP/1.1\r\nHost: micropython.org\r\n\r\n')
    data = s.recv(1000)
    print(data)
    s.close()
```

Take a moment to read through the code. The code is designed to connect to the `micropython.org` website and return the header data. Notice besides the networking code, there are statements to print the network information and data returned from the website. Also, notice the code is implemented as a function named `test_connect()`. We will call that function from the REPL console. Don't worry, it's very easy.

Go ahead and edit the code to include your router and password and then save it to your Pyboard USB drive with the name `CC3K.py`. Next, connect to your Pyboard with the REPL console and then issue the following commands. This will load the module and then run the code and test the connection.

```
>>> from CC3K import test_connect
>>> test_connect()
```


The connection may take a few moments to execute but when it completes, you should see a line that prints out the connection information and, following that, notice that the data returned from connecting to the website. If you see connection errors, be sure to verify your router name and password.

```
('10.0.1.123', '255.255.255.0', '10.0.1.1', '10.0.1.1', '10.0.1.1',
'08:00:28:59:23:cb', 'MY_ROUTER')
b'HTTP/1.1 200 OK\r\nServer: nginx/1.8.1\r\nDate: Thu, 29 Jun 2017
02:21:50 GMT\r\nContent-Type: text/html; charset=utf-8\r\nTransfer-
Encoding: chunked\r\nConnection: keep-alive\r\nVary: Accept-Encoding\r\
nX-Frame-Options: SAMEORIGIN\r\n\r\n3dc3\r\n<!DOCTYPE html>\n\n\n
<html lang="en">\n  <head>\n    <meta charset="utf-8">\n    <meta http-
equiv="X-UA-Compatible" content="IE=edge">\n    <meta name="viewport"
content="width=device-width, initial-scale=1">\n    <!-- The above 3
meta tags *must* come first in the head -->\n\n    <link rel="icon"
href="/static/img/favicon.ico">\n\n    <title>MicroPython - Python for
microcontrollers</title>\n\n    <link href="/static/bootstrap-3.3.7-dist/
css/bootstrap.min.css" rel="stylesheet">\n\n    <link href="/static/css/sty'
```

If that seemed like a lot of work, don't worry as this is as complicated as it gets for connecting your Pyboard to the Internet. Don't worry too much about understanding all the code now. We will explore Python and the MicroPython libraries in more depth in the next few chapters.

Now let's discover where you can purchase our own Pyboard.

Where to Buy

You can find the Pyboard online from [micropython.org](https://store.micropython.org/store/#/store) (<https://store.micropython.org/store/#/store>). The boards ship from the European Union but shipping costs are reasonable. If you are in North America, you can also buy the boards and related accessories from Adafruit (<https://www.adafruit.com/?q=pyboard&>). Depending on currency exchange and shipping (how quickly you want the board delivered), you may find the boards are slightly cheaper to order from the European Union. You can also find the board on various auction sites and other retail sites like Amazon. You can expect to pay about \$35-\$40 USD for your Pyboard.

You can buy the Pyboard with or without headers installed. Older versions of the board can be purchased the same way and there is a version without the accelerometer, which makes it a bit cheaper. However, I recommend buying the latest, v1.1 board.

As for the networking module, your best bet is to either wait until more hardware support is added to the MicroPython library (and upload the firmware) or find one of the older Adafruit CC3000 modules. Check online auction sites for vendors still offering these or someone who has a used one for sale.⁸

⁸I feel it is only a matter of time before a networking board (skin) or similar is offered by [micropython.org](https://store.micropython.org). Check their site for updates.

WiPy

The WiPy MicroPython board is another of the first MicroPython-ready boards. As the name suggests, the WiPy has WiFi networking built in that acts as a host (think router or access point). In fact, the default way your work with the WiPy is over WiFi. So, you don't have to use another module to make your project Internet ready. This feature alone makes the WiPy a better choice for the projects in this book. As a bonus, the board also has Bluetooth, making it even easier to adapt to your projects.

Overview

The most noticeable feature of the board is its small size. The board measures approximately 40mm x 40mm and has two. The WiPy comes as a small module not much bigger than a pack of gum, measuring a mere 42mm x 20mm and only 3.5mm thick (without headers).

The board comes with or without headers (get the one with the headers or install them yourself). You can use the board on a breadboard or you can also purchase a special board that operates like a dock. Pycom calls it the expansion board and provides a micro SD drive, micro USB connector, LEDs, replicates all the header pins, and includes a battery connector. The expansion board makes using the WiPy considerably easier and there is only one trick to using it as we will see. Figure 3-14 shows the WiPy board from Pycom.

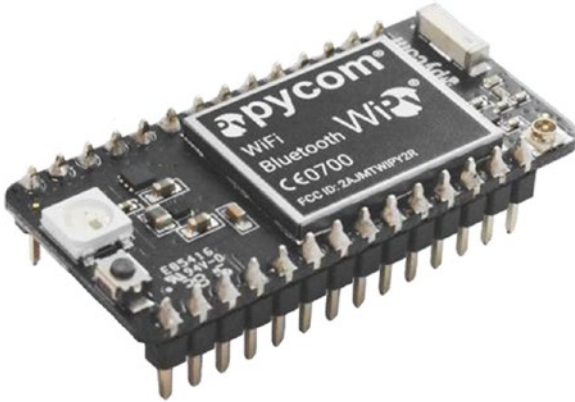


Figure 3-14. Pycom WiPy (courtesy of pycom.io)

The expansion board allows you to connect to your WiPy over USB and power the board with a LIPO battery or via USB. I like the expansion board for its convenience. Plus, it has holes in each corner, making it possible to mount it on risers for easier access to the pins or to mount it permanently to a case or tool holder. It even has a user-accessible (programmable) button and LED. Figure 3-15 shows the Pycom Expansion Board.

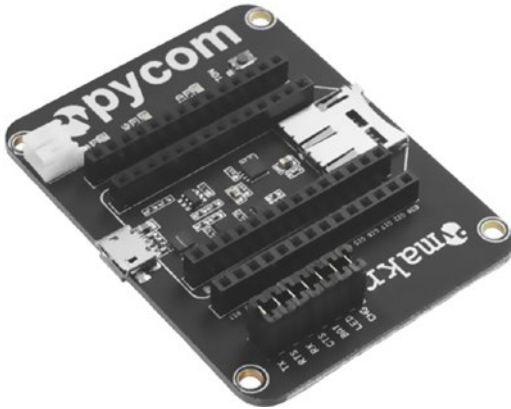


Figure 3-15. *Pycom Expansion Board (courtesy of pycom.io)*

■ **Tip** The current release of the WiPy is version 2.0. If you purchase a WiPy and want the Expansion Board, make sure you also purchase the 2.0 version of the Expansion Board.

Of course, the Expansion Board is quite a bit larger, measuring about 50mm x 65mm and with the WiPy installed about 14mm thick. Still small, but not as small as the WiPy itself.

The micro SD drive and micro USB connector are features you will likely want to have when working with your WiPy. Another feature of the Expansion Board that may make it worth the extra cost is that the LIPO battery connector is also a charger so while your board is plugged into a USB power supply, it charges the LIPO battery.

More About the Hardware

Now let's learn a bit more about the hardware - more specifically, some of the details and specifications that you may not know but might be handy to know.

First, the micro SD card on the Expansion Board (the WiPy does not have an SD drive) does not work the same way as the Pyboard. The WiPy cannot boot from the micro SD drive, but it is still possible to store your files and access them from your programs. We will see more about how to work with the micro SD drive in the next section.

The next thing we should discuss is the boot modes. You can change the boot mode like the Pyboard, which can be helpful if something goes wrong but unlike the Pyboard, the boot modes work with the configuration (firmware) changes rather than how the board boots. More specifically, you can set the board to safe boot, which skips off the boot.py and main.py scripts. The safe boot modes include the following.

- Safe boot with the latest firmware
- Safe boot with the previous user updates
- Safe boot with factory settings

To activate and select the safe boot mode, place a jumper wire between the P12 pin (GPIO20) and 3.3V. This effectively “pulls up” the pin so that the firmware can detect the selection. When holding the board so that the USB connector is on top, the P12 pin is the last pin on the left, and the 3.3V pin is the third from the top on the right. Figure 3-16 shows a properly installed jumper on the WiPy Expansion Board.



Figure 3-16. Jumper Installed for Safe Boot Selection (WiPy Expansion Board)

When you power on the board, the LED will turn orange and start flashing slowly. If you leave the jumper in place after 3 seconds, the LED will start blinking a bit faster and the board will boot with the latest firmware. If you leave the jumper in place for 3 more seconds, the board will boot with the previous user update select. Finally, if you leave the jumper in place 3 more seconds, the LED will flash faster, indicating the board will boot with the factory settings. Remove the jumper at any point during this sequence to select the desired safe boot mode.

The WiPy board supports both WiFi and Bluetooth communication mechanisms making it an excellent choice for an IOT project. It features an Espressif ESP32 chipset and dual processor. Additional features include the following.

- Powerful CPU, BLE and latest WiFi radio
- 1KM WiFi Range
- Fits in a standard breadboard (with headers)
- Ultra-low power usage: a fraction compared to other connected micro controllers
- Hash / encryption: SHA, MD5, DES, AES

- WiFi
 - 802.11b/g/n 16mbps
 - Security: SSL/TLS support and WPA Enterprise security
- Bluetooth: Low energy and classic
- RTC: Running at 32KHz
- Power: 3.3V to 5.5V, 3V3 output capable of sourcing up to 500mA
- Memory
 - RAM: 512KB
 - External flash 4MB
 - Hardware floating-point acceleration
- Python multi-threading

What is interesting is that Pycom makes several other MicroPython boards that you can use for more advanced projects. Each board is designed to support a different communication mechanism (WiFi, radio, cellular, and Bluetooth) and advanced features for embedded solutions. They all have the same form factor as the WiPy. These include the following boards. See the product description pages for more information.

- *SiPy*: supports Sigfox, WiFi, and BLE (<https://www.pycom.io/product/sipy/>)
- *LoPy*: supports LoRa, WiFi and BLE (<https://www.pycom.io/product/lopy/>)
- *GPY*: supports WiFi, BLE and cellular LTE-CAT M1/NB1 (<https://www.pycom.io/product/gpy/>)
- *FiPy*: Sigfox, LoRa, WiFi, BLE and cellular LTE-CAT M1/NB1 (<https://www.pycom.io/product/fipy/>)

Best of all, the Pycom Expansion Board works with these boards. Clearly, Pycom is in the business to supply the IOT world a host of powerful microprocessors – all running MicroPython!

If you want to power your board using an external power source or battery, you must ensure the power source is set to 3.3-5V. Connecting higher power can damage the board. Lower power can make the board unstable (program may fail). You can connect power to (positive) to VIN and ground (negative) to GND. Figure 3-17 shows a close-up of where these pins are located near the large LED.

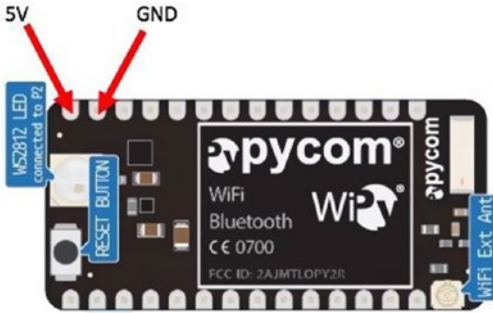


Figure 3-17. Pins for connecting external power (WiPy - courtesy of pycom.io)

Now, let's see the WiPy in action, starting with a simple walkthrough of connecting to our PC and running a simple Python program.

Getting Started with WiPy

Let's start with the basics again – the REPL console. Since the WiPy is a WiFi device, the easiest way to connect to it is over WiFi. If you're using the Expansion Board, you need only power on the board with either a USB cable connected to your PC or a USB power supply. If you are using the WiPy without the Expansion Board, you can power the board as shown above.

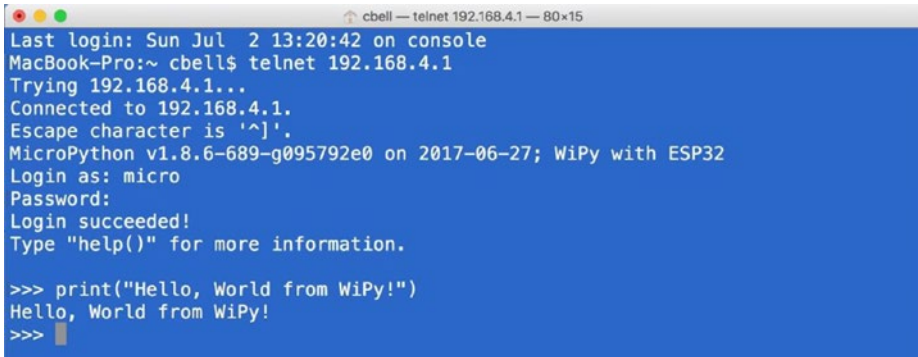
After the WiPy is powered on, it will set up a WiFi network that defaults to 192.168.4.XXX with the WiPy having the IP address of 192.168.4.1. This process can take a few minutes and during that time the LED on the WiPy will be off. When it starts blinking blue, your WiPy network is ready for connections.

To connect to the board over WiFi, you must connect your PC to the WiPy WiFi network. The network SSID should be like `wipy-wlan-b454`. The password is www.pycom.io. Follow the normal procedure for your operating system to connect to the network. If your PC uses WiFi only, you won't be able to access the Internet at the same time, but you won't need to while experimenting with the board.

Once connected to the WiPy WiFi network, you can use telnet to connect to the REPL console as shown below.

```
$ telnet 192.168.4.1
```

Be advised the connection could take a few moments before the REPL console appears. This is normal. Once the connection is made, you will be asked for the user and password. Use `micro` for the user and `python` for the password. Once you've entered the password, you should see the console and can interact with the WiPy as shown in Figure 3-18.



```

cbell — telnet 192.168.4.1 — 80x15
Last login: Sun Jul 2 13:20:42 on console
MacBook-Pro:~ cbell$ telnet 192.168.4.1
Trying 192.168.4.1...
Connected to 192.168.4.1.
Escape character is '^]'.
MicroPython v1.8.6-689-g095792e0 on 2017-06-27; WiPy with ESP32
Login as: micro
Password:
Login succeeded!
Type "help()" for more information.

>>> print("Hello, World from WiPy!")
Hello, World from WiPy!
>>>

```

Figure 3-18. The REPL Console (WiPy)

You can also connect to your WiPy over USB if using the Expansion Board. However, you must duplicate the UART communication on the WiPy first. This is a common oversight and some have complained that their WiPy doesn't connect over USB. What they've failed to do (besides read the online documentation) is run the following code on their WiPy when connected over the network. This code duplicates the UART on the Expansion Board so that you can connect over USB. However, you do not need to do this if you plan to use your WiPy over the network.

```

from machine import UART
import os
uart = UART(0, 115200)
os.dupterm(uart)

```

■ **Note** Newer WiPy boards should already have this code.

If your WiPy does not have this code in the `boot.py` file, you should consider adding it.

Connecting to Your WiFi Network

If you want to connect your WiPy to your WiFi network, you will need to make some small changes to the `boot.py` script. The easiest way to do that is to copy it from the WiPy, modify it, and copy the new version to the WiPy. The best way to do this is with file transfer protocol (ftp). Most operating systems include an ftp client, but if yours does not, you can find several options for download. Use the user `micro` and password `python` to connect. Listing 3-2 shows how to connect to the WiPy and copy the `boot.py` file to your PC.

Listing 3-2. Copying Files From the WiPy

```

$ mkdir wipy_files
$ cd wipy_files/
$ ftp 192.168.4.1
Connected to 192.168.4.1.
220 Micropython FTP Server
Name (192.168.4.1:cbell): micro
Password:
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd flash
ftp> get boot.py
local: boot.py remote: boot.py
227 (192,168,4,1,7,232)
  108      6.33 KiB/s
108 bytes received in 00:00 (6.23 KiB/s)
ftp> get main.py
local: main.py remote: main.py
227 (192,168,4,1,7,232)
  34      2.53 KiB/s
34 bytes received in 00:00 (2.47 KiB/s)
ftp> quit
MacBook-Pro:wipy_files cbell$ ls
boot.py  main.py

```

Notice I created a directory on my local PC and then connected to the WiPy with the command `ftp 192.168.4.1`. I then changed to the `flash` directory and copied the files with the `get` command. Once I copied the files, I quit the `ftp` application and now I have the files on my local machine.

■ **Tip** Make a backup of your original WiPy files so you can revert to factory settings if you need to do so.

Now we can modify the `boot.py` file to connect to our local network. What we need to do is use the `WLAN` class from the `network` module to configure it to scan for networks and connect to your WiFi network by name (SSID) and password. Listing 3-3 shows the code you need to add to your `boot.py` file (new lines shown in bold). Be sure to substitute your SSID and password in the code before you save it to your WiPy (shown in bold).

■ **Caution** I recommend waiting to do this change to your `boot.py` file until you're ready to run your IOT project. Keep using the WiPy in the default mode until you've perfected your project.

Listing 3-3. Enabling WiFi Connection on Boot (WiPy)

```
# boot.py -- run on boot-up
from machine import UART
from network import WLAN
import os
uart = UART(0, 115200)
os.dupterm(uart)
wlan = WLAN(mode=WLAN.STA)
wlan.scan()
wlan.connect(ssid='Your Network SSID', auth=(WLAN.WPA2, 'Your Network Password'))
while not wlan.isconnected():
    pass
print(wlan.ifconfig()) # prints out local IP
```

Notice the code for duplicating the UART is also included. Also, note the print at the end. Before saving the file to your WiPy, you should test the code using the REPL console over USB and paste it to ensure it works. We must use USB because the WiFi will be reset during execution. The following shows the results of running the code from the REPL console.

```
>>> from network import WLAN
>>> wlan = WLAN(mode=WLAN.STA)
>>> wlan.scan()
[[('ssid='SSIDHERE1', bssid=b' \xc9\xd0\x18W\x11', sec=3, channel=1, rssi=-70), ('ssid='SSIDHERE2', bssid=b'\x88\x1f\xa16X\x1c', sec=3, channel=6, rssi=-78), ('ssid='SSIDHERE3', bssid=b'\xb8\x8d\x12bX\xd3', sec=3, channel=1, rssi=-90)]
>>> wlan.connect(ssid="SSIDHERE1", auth=(WLAN.WPA2, "SSIDPASSWORD"))
>>> while not wlan.isconnected():
...     pass
>>> print(wlan.ifconfig())
('10.0.1.128', '255.255.255.0', '10.0.1.1', '10.0.1.1')
```

Notice the IP address returned (the first one in the print statement) is 10.0.1.128. Now you can connect to your WiPy using telnet as shown below.

```
$ telnet 10.0.1.128
Trying 10.0.1.128...
Connected to 10.0.1.128.
Escape character is '^]'.
MicroPython v1.8.6-689-g095792e0 on 2017-06-27; WiPy with ESP32
Login as: micro
Password:
Login succeeded!
Type "help()" for more information.
```

Once you're satisfied it worked (you have the correct SSID and password), you can make the changes and can copy the file to your WiPy as shown in Listing 3-4.

Listing 3-4. Copying Files To the WiPy

```
$ ftp 192.168.4.1
Connected to 192.168.4.1.
220 Micropython FTP Server
Name (192.168.4.1:cbell): micro
Password:
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd flash
ftp> put boot.py
local: boot.py remote: boot.py
227 (192,168,4,1,7,232)
100% |*****| 336      1.84 MiB/s   00:00 ETA
336 bytes sent in 00:00 (0.86 KiB/s)
ftp> quit
```

Now you can reboot your WiPy and wait until the blue LED flashes; then you should be able to see it on your WiFi network. Cool, eh?

Using the SD Drive

Like the Pyboard, the WiPy, when used with the Expansion Board, has a micro SD drive. You can use the SD card to store data, scripts, and more. However, unlike the Pyboard, SD drive is not used when the board is booted (you cannot boot from it) nor is it mountable when connected to your PC via a USB cable.

Neither the internal drive nor the SD card (if connected to the Expansion Board and a card is inserted and formatted as FAT) are mounted on your PC. You must use a file transfer protocol (FTP) application to access your files like we saw in the last section. However, you can mount the SD card through the REPL console. If you always want to access the SD drive, you can add this code to your `boot.py` file on the flash drive.

What we need to do is run some MicroPython code to enable the SD card and mount it. The following shows how to mount the SD card.

```
from machine import SD
try:
    sd = SD()
    os.mount(sd, '/sd')
    print('Card mounted at /sd')
except:
    sd = None
    print('ERROR: Card not mounted.')
```

Now when you access your files, you will see the SD card.

```
$ ftp 192.168.4.1
Connected to 192.168.4.1.
Micropython FTP Server
Name (192.168.4.1:cbell): micro
Password:
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
drw-rw-r--  1 root  root           0 Jan  1  1985 flash
drw-rw-r--  1 root  root           0 Jan  1  1985 sd
```

Loading the Firmware

Loading the firmware on the WiPy is a bit easier than the Pyboard. Pycom has supplied very easy-to-use firmware upgrade tools for Windows, macOS, and Linux that step you through the process automatically selecting the latest version. You can download them from <https://docs.pycom.io/chapter/gettingstarted/installation/firmwaretool.html>. The firmware upgrade process is best done with the Expansion Board.

■ **Tip** If you plan to use a WiPy, be sure to buy the Expansion Board. It is well worth the cost.

You simply download the upgrade tool for your platform, install it, then run the application. For example, on macOS, you can click on the link, download the tool, open the archive, and install it.

Like the Pyboard, you must also jumper one of the pins. In this case, we will pull pin G23 low by connecting a jumper to GND. When holding the board so that the USB connector is on top, the G23 pin is the fourth pin from the top on the left, and the GND pin is the second pin from the top on the right. Figure 3-19 shows a properly installed jumper on the WiPy Expansion Board.

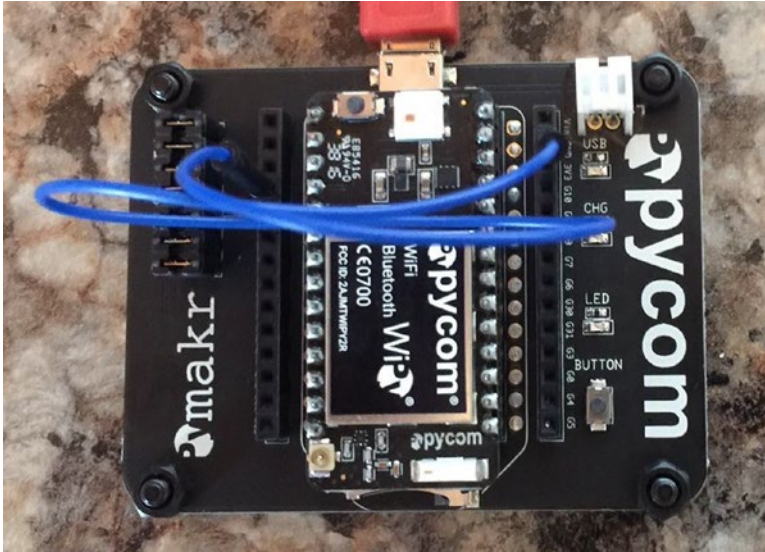


Figure 3-19. Jumper Installed for Firmware Upgrade Tool (WiPy Expansion Board)

Now, let's briefly see how the Pycom firmware upgrade tool runs on macOS. There are four main panels to the dialog: *welcome*, *setup*, *communication*, and *result*. You click the *Continue* button to advance each panel. On the *setup* panel, you are reminded of the connections you need to make and how to set the jumper. On the *communication* panel, you select the communication (USB) port where your WiPy is connected. After the upgrade is complete, the *result* panel will tell you when you can disconnect the board. Figure 3-20 shows the dialogs in order, clockwise from the upper left.

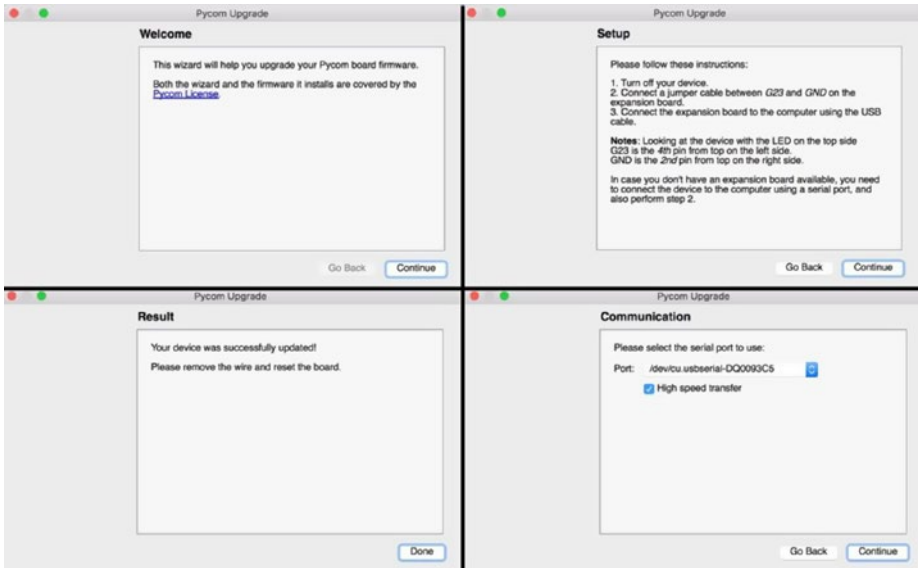


Figure 3-20. *Upgrading the Firmware (WiPy)*

Special Applications

There is one more thing that is unique to the WiPy (and all Pycom boards). Pycom also supplies a special editor plugin for some popular editors that allow you to connect to and work with your files on the WiPy. The plugin is named PyMakr and is currently available for the Atom editor (but will soon be available for other editors). Pycom also provides an app for your mobile device name PyMate, which allows you to interact with your WiPy on your mobile device.

The PyMakr plugin is a nice addition to the WiPy experience. The plugin is a replacement for the original PyMakr editor that Pycom originally released, but they realized that developing a new editor was less productive than developing it as a plugin that would allow users to use their existing favorite editor.

The plugin allows you to open a REPL console inside the editor, move (synchronize) files, open Python scripts and run them, configure the console to connect to your WiPy on other networks (settings), and get information about your WiPy such as the firmware version, SSID, and more. Figure 3-21 shows an example of the plugin running in the Atom editor.

```

boot.py — ~/Documents/wipy_files
Welcome Guide  boot.py  Incompatible Packages  Settings  Welcome

1 # boot.py — run on boot-up
2 from machine import UART
3 from network import WLAN
4 import os
5 uart = UART(0, 115200)
6 os.dupterm(uart)
7 wlan = WLAN(mode=WLAN.STA)
8 wlan.scan()
9 wlan.connect(ssid='StoreLot', auth=(WLAN.WPA2, 'sunhillow'))
10 while not wlan.isconnected():
11     pass
12 print(wlan.ifconfig()) # prints out local IP

Connected ✓
Connecting on 192.168.4.1...
More Reconnect Sync Run Settings Close

>>> print("Hello, World from Atom + PyMakr!")
Hello, World from Atom + PyMakr!
>>>

~/Documents/wipy_files/boot.py* 12:45  CRLF UTF-8 Python 0 files

```

Figure 3-21. PyMakr Plugin (Atom)

If you like the example of the early release here, you can visit the following URL and check on future releases for the other editors: <https://docs.pycom.io/chapter/gettingstarted/installation/pymakr.html>.

The PyMate app for mobile devices also allows you to interact with your WiPy and is like the PyMakr plugin for working with the WiPy, but it can provide widgets you can use to read data from your WiPy and display the data. Widgets can be used to display the data from your script on your mobile device. Widgets include line and bar charts, buttons to control LEDs, and much more.

The setup of PyMate requires downloading it from the Apple or Google application store and configuring it on your mobile device. It also requires registration of the app with Pycom. Setting up your WiPy requires first connecting to your WiFi (or cellular) network to download the correct files and then selecting your device and finally connecting to the WiPy WiFi to upload the files. The process modifies the `boot.py` file so be sure to make a backup of that file first. If you get stuck or you can no longer access your WiPy, you can boot into one of the safe boot modes and copy the original `boot.py` file to the WiPy (or edit the one the PyMate created for you). Once you have set up your device in PyMate, you can connect to it and execute your scripts, configuring the widgets you want to interact with.

The PyMate app is currently very new, and there is very little documentation on how to use it. For that reason, you may want to wait to use the PyMate app until after you've completed the projects in this book. This is mainly due to the modifications to the `boot.py` file but also because you must first learn how to use MicroPython to access hardware so that you can use the widgets. Figure 3-22 shows the PyMate application.

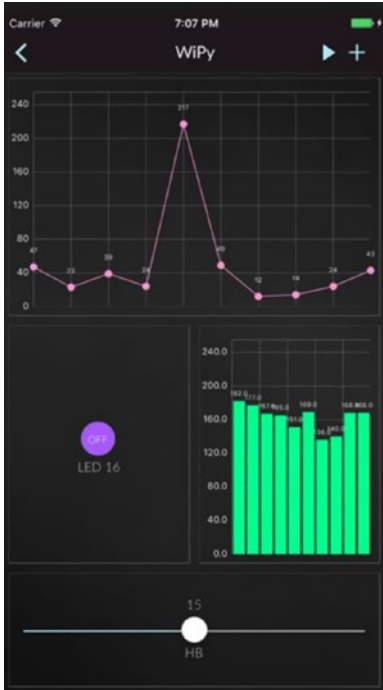


Figure 3-22. PyMate Application (courtesy of pycom.io)

If you have decided to buy the WiPy (and Expansion Board), I encourage you to check out these applications.

DEEP SLEEP ISSUE RESOLVED

Pycom recently announced a minor issue with their boards regarding a deep sleep mode, which is a state you can place the board in programmatically to go into a low-power mode and wake up later. Using deep sleep modes allows the board to conserve power and can be especially handy for projects that run off solar or battery power. The issue was found to be a hardware issue and Pycom is building a special add-on board (called a shield) that you can use to correct the deep sleep issue for older boards. Visit pycom.io for the latest information on the deep sleep shield. The projects in this book do not require deep sleep mode, but your own IOT projects may.

Where to Buy

You can buy the WiPy and other boards directly from Pycom (<https://www.pycom.io/webshop/>). Adafruit also carries the boards but may not have all the latest accessories (<https://www.adafruit.com/?q=pycom&>). Pycom also has a handy link of retailers that you can use to find a reseller closer to you (<https://www.pycom.io/where-to-buy/>). The WiPy costs about \$25 USD and the Expansion Board about \$20 USD. You should consider buying one Expansion Board to make working with your WiPy easier.

You can buy the board with or without headers. For use in this book, it would be best to have the headers installed. This allows you to plug the board into a breadboard.

MicroPython-Compatible Boards

The next group of hardware to explore is those boards that do not come with MicroPython but can be either loaded with MicroPython firmware or can execute MicroPython using special software.

The boards in this category require some effort to configure and may not be for the very beginner (except for the BBC micro:bit). They also may not support the same hardware features such as GPIO pins, have additional hardware that isn't (currently) supported by MicroPython, or in one case may require a special variant of MicroPython to work properly. As such, I will not spend as much time describing the boards or their hardware. Rather, I present the boards with a brief overview and then describe how to get started, including the process needed to load MicroPython. If you decide to use one of these boards, check out the vendor's website for the very latest information on using the board with MicroPython.

The boards covered here are the wildly popular BBC micro:bit boards, which are seeing a growing trend of use in schools, the latest Circuit Playground board from Adafruit called the Circuit Playground Express, and the Adafruit Feather Huzzah. The boards are presented in order of complexity required to get MicroPython working on the board.

BBC micro:bit

The BBC micro:bit board is specially designed to be very easy to use. In fact, it is designed to help teach children more about hardware and software. In that, the BBC micro:bit is a huge success. Part of that success is in how easy the board is to use – its form factor measures only approximately 52mm x 42mm with components on both sides.

On one side is an array of programmable LEDs and two buttons. On the other side are the components including the processor, micro USB connector, reset button, and battery connector. The board has its GPIO header arranged along the bottom edge and is also two sided. A set of large-holed pins (called alligator pins) are spaced evenly that allow you to ground, power (3V), and (3) GPIO pins. This makes using the board with an edge connector simple. Figure 3-23 shows the front and back of the BBC micro:bit board.

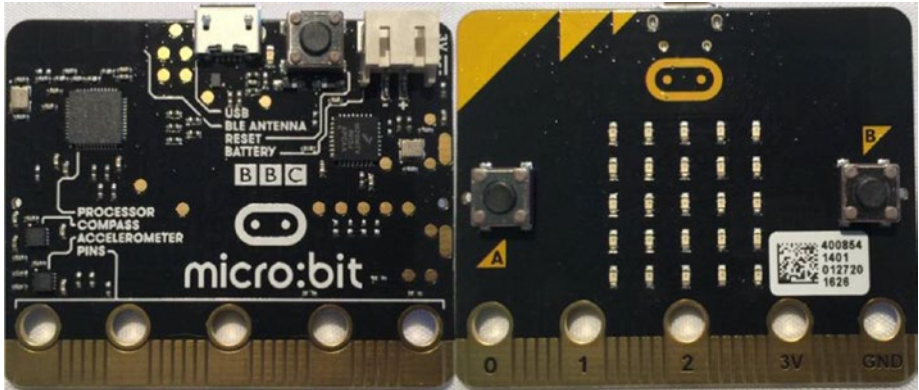


Figure 3-23. The BBC micro:bit Board (front and back)

Another reason for the success of this board is the software created to support it. The developers have created an easy-to-use software for working with the board. Since the board is intended to be used like an Arduino coding in a C-like language, the software available is beyond the scope of this book, but you can read more about it at <http://microbit.org/>. However, we will see shortly a special application that enables us to create, edit, and run MicroPython on the BBC micro:bit. Cool.

The BBC micro:bit does not have networking but does have Bluetooth, which you can use to connect to another device to forward data on to the Internet. So, it can be used for IOT projects but it is not quite as easy as the WiPy or Pyboard with a WiFi module, and may require an intermediate node such as a PC or small computer like a Raspberry Pi or even an Arduino.

The following lists an overview of some of the hardware features of the BBC micro:bit board. You can also access the on-board USB drive when the board is connected to your PC via a USB cable.

- 32-bit ARM Cortex processor
- 16K RAM
- Compass for detecting orientation
- Accelerometer for detecting changes in movement (speed)
- Low-energy Bluetooth (BLE)
- (2) Programmable buttons
- A 5x5 array of programmable LEDs
- (3) Alligator digital/analog pins
- (20) GPIO pins on edge connector
- Battery connector

Now that we've had a brief tour of the hardware, let's look at how to use the board with MicroPython.

Up and Running with MicroPython

The BBC micro:bit board is the easiest alternative board to use with MicroPython. This is due to two software applications – a desktop application named Mu, and a command-line tool named uFlash. Mu is a full editor that you can use on your PC and, when connected to your BBC micro:bit via a USB cable, can save and execute the scripts. The uFlash tool can be used to manually transfer Python scripts to the board. Both options are available for use on Windows, macOS, and Linux. Figure 3-24 shows an example of using Mu to write a short MicroPython script to use the LEDs to scroll a message across them.

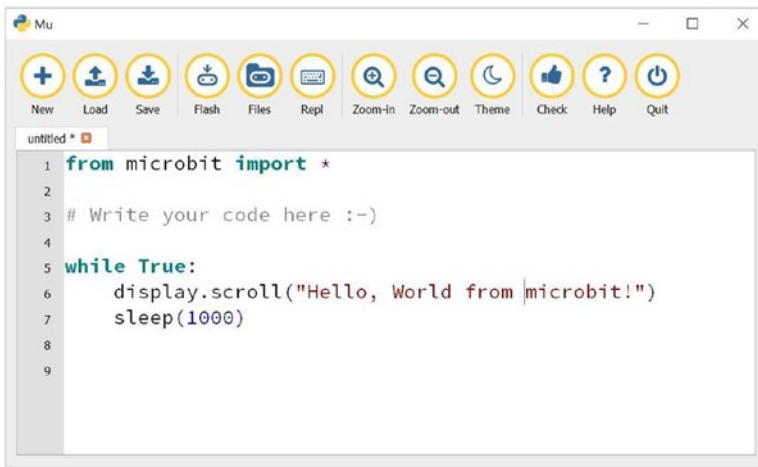


Figure 3-24. The Mu Editor for MicroPython on the BBC micro:bit

Unlike the other boards where we must first install the firmware to use MicroPython, the BBC micro:bit can be used to run MicroPython scripts using one of these tools.

For example, we can use Mu to write our MicroPython script and then “flash” the BBC micro:bit board with a compiled version of the script (called a hex file). Yes, this means you can write your script, compile it, and flash (load the hex file) directly to the board! We simply write the code in the editor and click *Flash* to load it on the board. Best of all, the board is programmed to always run the script on each boot. So, that means we can load our own code directly to the board without extra work. Cool.

You can still access the board with the REPL console through the Mu application by clicking the *REPL* button. Figure 3-25 shows the REPL console for the BBC micro:bit running in Mu on Windows 10.



Figure 3-25. REPL Console via Mu on Windows 10 (BBC micro:bit)

■ **Tip** You can read the very latest on using MicroPython on the BBC micro:bit at <https://microbit-micropython.readthedocs.io/en/latest/>.

Where to Buy

You can purchase the BBC micro:bit as well as several accessories from a few retailers including Adafruit (<https://www.adafruit.com/?q=micro%3Abit&>), Sparkfun (<https://www.sparkfun.com/categories/284>), The Pi Hut (<https://thepihut.com/collections/micro-bit-store>), and Kitronik (<https://www.kitronik.co.uk/microbit.html>). You can also buy the boards from the developer in a special “buy one, give one” promotion that gives free BBC micro:bit boards to schools all over the world at <https://give.microbit.org/>. The board costs about \$25 USD.

If you plan to use the board for IOT projects or with the projects in this book, I recommend also purchasing an edge connector breakout board such as the one from Sparkfun (<https://www.sparkfun.com/products/13989>).

Circuit Playground Express (Developer Edition)

The next easiest alternative board to use is the Circuit Playground Express board from Adafruit. In fact, the Circuit Playground Express (Developer Edition) is a curious alternative board. It’s patterned after several other Adafruit wearable boards where the board is designed so that it can be incorporated into clothing.⁹ As such, it is circular and about 50mm in diameter.

⁹See <https://www.adafruit.com/category/65> for the latest Adafruit wearable products.

■ **Note** The current board has the name Developer Edition appended because it is still in early release. I suspect there will be a more production-complete form of the board in the future.

Around the outside of the board are 10 pins with large holes (called alligator pads) that can be used with alligator clips or used to sew with conductive thread. Adafruit also sells alligator to male jumpers that you may find essential to using the board with a breadboard. See the *Small Alligator Clip to Male Jumper Wire Bundle* at <https://www.adafruit.com/product/3255>. Another interesting feature is a set of 10 RGB NeoPixels, which are bright LEDs that you can change the color through code. The board is also loaded with sensors making a good alternative to experiment with IOT projects. Figure 3-26 shows the Circuit Playground Express board.

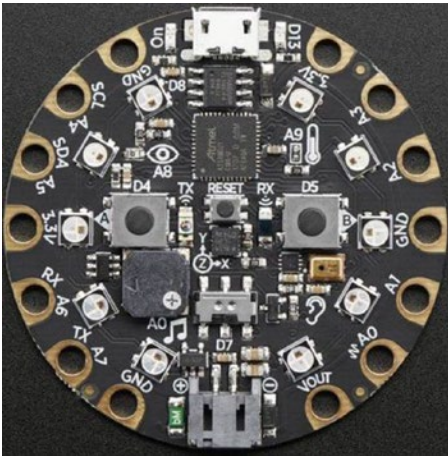


Figure 3-26. *Circuit Playground Express, Developer Edition (courtesy of adafruit.com)*

The following lists an overview of some of the hardware features of the Circuit Playground Express board. You can also access the onboard USB drive when the board is connected to your PC via a USB cable.

- ATSAM D21 ARM Cortex M0 Processor, running at 3.3V and 48MHz
- 2 MB of SPI Flash storage, used primarily with CircuitPython to store code and libraries
- Micro USB port for programming and debugging. The USB port can act like serial port, keyboard, mouse, joystick, or MIDI
- 10 x mini RGB NeoPixels

- Motion sensor
- Temperature sensor (thermistor)
- Light sensor (phototransistor). Can also act as a color sensor and pulse sensor.
- Sound sensor (MEMS microphone)
- A mini speaker!
- (2) Pushbuttons
- (1) Slide switch
- Infrared receiver and transmitter. Can also act as a proximity sensor.
- (8) Alligator-clip friendly input/output pins
- I2C, UART, 8 pins that can do analog inputs, multiple PWM output
- (7) pads can act as capacitive touch inputs and the 1 remaining is a true analog output
- A red “#13” programmable LED
- Reset button

■ **Note** There is also a Circuit Playground Classic board from Adafruit. Do not confuse it with the Circuit Playground Express. The boards are not the same; only the Express board can run MicroPython.

However, like the Pyboard, the Circuit Playground Express (Developer Edition) does not have any networking capability. So, you must use an external module to connect to your network. The best option I’ve found is a Bluetooth module but other modules like the CC3000 may be alternatives.

The board also does not run MicroPython. Instead, it runs a special version of MicroPython called CircuitPython. CircuitPython is a derivative of MicroPython designed and maintained by Adafruit, built especially for the Circuit Playground Express and a host of other boards. CircuitPython is designed to run on a wide array of boards including the Circuit Playground Express, Feather, and other popular boards. CircuitPython is currently compatible with MicroPython version 1.8.4, but it is being updated regularly.

While CircuitPython is compatible with MicroPython, you may need to use different hardware libraries for some features. There are some differences and these are documented at <https://github.com/adafruit/circuitpython#differences-from-micropython>. But for our uses, it will work nearly exactly like what we would expect from MicroPython.

Adafruit has an excellent set of tutorials and blogs to help you out. For more information about CircuitPython, see <https://github.com/adafruit/circuitpython> and <https://learn.adafruit.com/search?q=circuitpython&>.

Now that we've had a brief tour of the hardware, let's look at how to use the board with MicroPython.

Up and Running with CircuitPython

While the Circuit Playground Express board doesn't have a fancy application like the BBC micro:bit, once you install the correct driver, loading CircuitPython binaries (firmware) on the board is surprisingly easy. In fact, all you need to do is download the current binary and copy it to the board's drive. The steps involved are summarized below.

If you use Windows 10, you will need to download Adafruit's special device driver first. You can get the driver at https://github.com/adafruit/Adafruit_Windows_Drivers/releases/download/1.0.0.0/adafruit_drivers.exe. Simply download and run the installer. The driver supports many of the Adafruit boards. When you get the installation options page, you can select which boards you want to support. I recommend selecting all the boards so that you can use any of the Adafruit boards that support CircuitPython. As you will see, there are many such options. Figure 3-27 shows an example of the installation options.

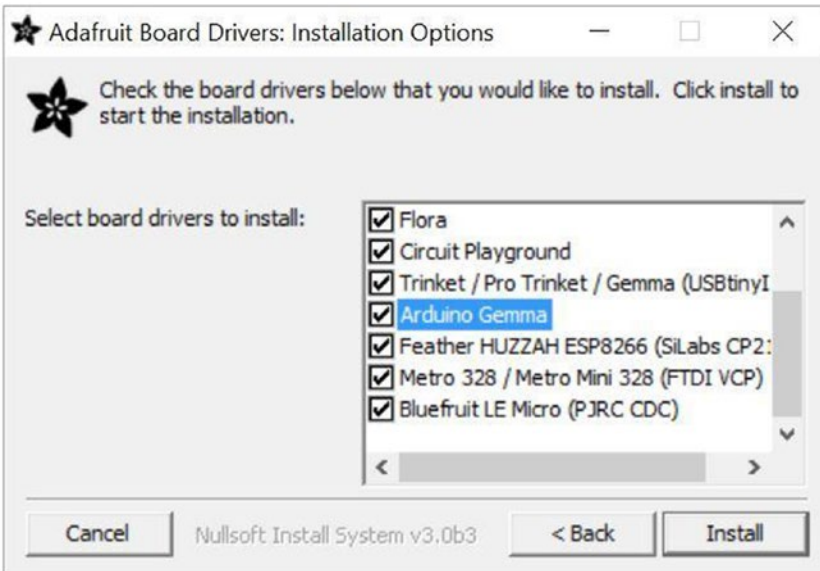


Figure 3-27. Adafruit Boards Driver – Installation Options

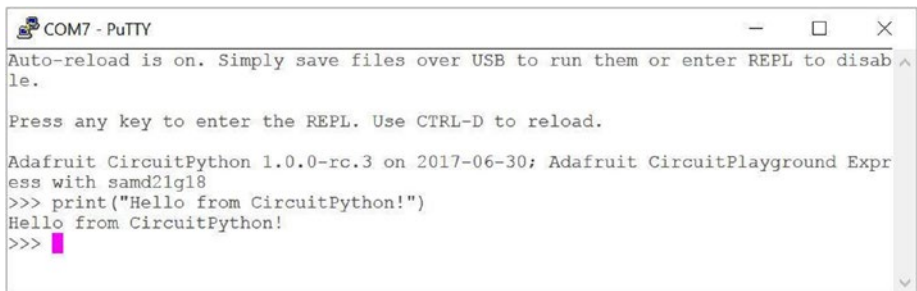
Next, we need to download the firmware. Adafruit built the firmware in a special format called USB Flashing Format (UF2).¹⁰ Go to <https://github.com/adafruit/circuitpython/releases> and download the latest version. For example, the file I downloaded was named `adafruit-circuitpython-circuitplayground_express-2.0.0-beta.1.uf2`.

■ **Note** When you go to that page, you will find there are two options for each of the boards: a `.bin` file and a `.uf2` file. The `.bin` file is used with a command-line tool called `bossac`,¹¹ but we will not use that tool since the UF2 format is so much easier to use.

Now you can connect your Circuit Playground Express to your PC. When the USB drive mounts (it should be named `CIRCUITPY`), copy the old UF2 file named `CURRENT.UF2` to your PC. We will use this as a backup. Next, drag the new UF2 to the `CIRCUITPY` folder. When the file copy completes, the board will reboot and run CircuitPython. That's it! You've just loaded CircuitPython! See, that was much easier than any of the other boards, eh?

■ **Caution** You must save the original UF2 file so that you can reverse the Circuit Python install.

You should now see the drive return but without the UF2 file. For example, my board shows only one file named `boot_out.txt`. Now you can connect to your board using screen or, on Windows, PUTTY. But first, check to see which COM port the board is using. Figure 3-28 shows running the REPL console using PUTTY on Windows.



```
COM7 - PuTTY
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

Press any key to enter the REPL. Use CTRL-D to reload.

Adafruit CircuitPython 1.0.0-rc.3 on 2017-06-30; Adafruit CircuitPlayground Express with samd21g18
>>> print("Hello from CircuitPython!")
Hello from CircuitPython!
>>> █
```

Figure 3-28. REPL Console on Windows (Circuit Playground Express)

¹⁰<https://github.com/Microsoft/uf2>

¹¹<https://learn.adafruit.com/adafruit-feather-m0-express-designed-for-circuit-python-circuitpython/circuitpython#flashing-with-bossac>

Fortunately, returning the board to the default configuration is also easy. You can do so by placing the board into bootloader mode by clicking the reset button twice. When in bootloader mode, the LEDs will turn red, during which time you can copy the original file we saved named CURRENT.UF2 to the drive that pops up (named CIRCUITPY). Once the copy completes, the board will reboot and run the original firmware.

Where to Buy

The Circuit Playground Express (Developer Edition) board is available from Adafruit and resellers of Adafruit products. You can find the board at <https://www.adafruit.com/product/3333>. The cost is approximately \$25 USD.

If you decide to try the Circuit Playground Express board, I recommend getting some alligator clips so you can use the board with a breadboard or connect it to other components. See the Adafruit store for more ideas and accessories for connecting the board to your components.

Adafruit Feather Huzzah

The Adafruit Feather Huzzah w/ESP8266 WiFi is another very popular board but loading MicroPython on it is limited to using a command-line tool and is a bit quirky to use on Windows. Thus, I consider it the hardest to use of the MicroPython-compatible boards. Fortunately, it does use MicroPython rather than CircuitPython like the Circuit Playground Express.

In fact, the board has a lot of features that makes it a great alternative board for MicroPython IOT projects. Chief among these is it has networking and works quite well on a WiFi network, putting it on par with the WiPy.

One of the reasons this board is so popular is because it is based on ESP8266¹² WiFi microcontroller clocked at 80 MHz and is combined with a rich set of features in a small, lightweight package (hence the feather in the name). The board is also very small – about the size of a pack of gum measuring 51mm x 23mm x 8mm and fits on a breadboard also making it an excellent prototyping tool. The board has two rows of GPIO headers including communication (RX, TX), I2C, and SPI protocols. Figure 3-29 shows the Adafruit Feather Huzzah board.

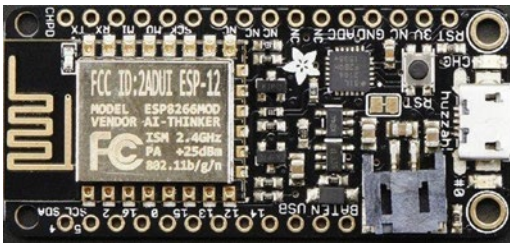


Figure 3-29. *Adafruit Feather Huzzah (courtesy of adafruit.com)*

¹²A most ubiquitous chip indeed!

The following lists an overview of some of the hardware features of the Circuit Playground Express board. You can also access the onboard USB drive when the board is connected to your PC via a USB cable.

- ESP8266 @ 80MHz with 3.3V logic/power
- 4MB of FLASH
- Built-in WiFi 802.11 b/g/n
- 3.3V regulator with 500mA peak current output
- Micro USB connector
- (9) GPIO pins - can also be used as I2C and SPI
- (1) Analog input @ 1.0V max
- 100mA LiPoly charger with charging status indicator LED
- Red programmable LED (#0)
- Blue programmable LED (#2)
- Power/enable pin
- 4 mounting holes
- Reset button

Now that we've had a brief tour of the hardware, let's look at how to use the board with MicroPython.

Up and Running with MicroPython

The process to use MicroPython on the Feather Huzzah (or any ESP8266 board) requires making sure you have the correct drivers loaded, using a Python-based serial bootloader utility in two steps to first erase the flash and then write the new firmware.

First, ensure your PC has the correct drivers loaded to recognize the board. Once again, Adafruit has a driver we can use and can be found at <https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers>. Simply download and install the driver, then reconnect your board to your PC. You should be able to find the COM port (or /dev/ file) on your PC.

One of the nice things about the Feather Huzzah is that it features an auto-reset support for getting into bootloader mode before firmware upload. The feature automatically detects when we attempt to access the bootloader and puts itself in the correct mode. Cool!

■ **Tip** Loading MicroPython on other ESP8266 boards may require setting the board in bootloader mode. Check your documentation for how to do this.

The only other thing we need is the correct firmware file. You can download the latest firmware from <http://micropython.org/download/#esp8266>. I chose the latest version supporting debugging but with the Web-based REPL turned off – a file named `esp8266-ota-20170613-v1.9.1-4-g6ed4581f.bin`. Download the file, and remember where you put it (and its name) as we will need it soon.

To begin the firmware load, we will need the ESP8266 and ESP32 serial bootloader utility from <https://github.com/esp8266/esptool>. Download the tool using the *Clone or Download* button, then unzip the files. You don't have to install the tool. We will run it from the directory using the Python interpreter.

Once the driver is installed, have opened a console (terminal) and changed to the directory where you unzipped the `esptool.py` utility, and your board is connected to your PC with a USB cable, we can begin by erasing the flash drive on the board using the `esptool.py` script and the `erase_flash` command as shown in Listing 3-5.

Listing 3-5. Erasing Firmware (Feather Huzzah)

```
$ python ./esptool.py --port /dev/tty.SLAB_USBtoUART erase_flash
esptool.py v2.0.1
Connecting....._
Detecting chip type... ESP8266
Chip is ESP8266
Uploading stub...
Running stub...
Stub running...
Erasing flash (this may take a while)...
Chip erase completed successfully in 8.8s
Hard resetting...
```

This process takes only a few moments. When it is done, we can upload the firmware using the `esptool.py` script and the `write_flash` command as shown in Listing 3-6.

Listing 3-6. Uploading Firmware (Feather Huzzah)

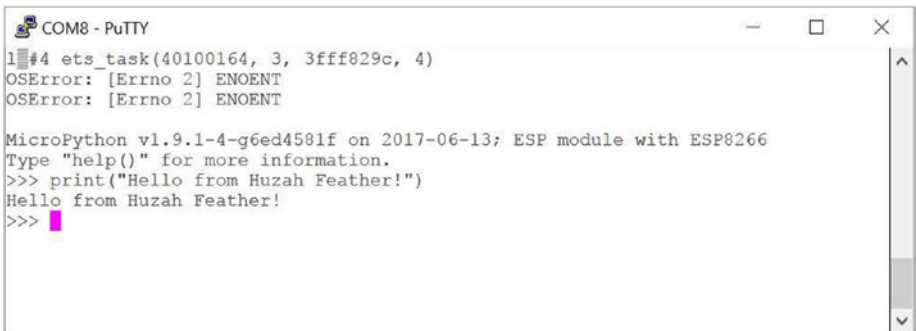
```
$ python ./esptool.py --port /dev/tty.SLAB_USBtoUART --baud 460800 write_
flash --flash_size=detect 0 ~/Downloads/esp8266-ota-20170613-v1.9.1-4-
g6ed4581f.bin
esptool.py v2.0.1
Connecting....._
Detecting chip type... ESP8266
Chip is ESP8266
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Flash params set to 0x0040
```

```

Compressed 819888 bytes to 545209...
Wrote 819888 bytes (545209 compressed) at 0x00000000 in 12.7 seconds
(effective 514.6 kbit/s)...
Hash of data verified.
Leaving...
Hard resetting...

```

This process takes a bit longer but once complete, you can reset the board or disconnect and reconnect; you can connect to the board with the REPL console as shown in Figure 3-30.



```

COM8 - PuTTY
l#4 ets_task(40100164, 3, 3fff829c, 4)
OSError: [Errno 2] ENOENT
OSError: [Errno 2] ENOENT

MicroPython v1.9.1-4-g6ed4581f on 2017-06-13; ESP module with ESP8266
Type "help()" for more information.
>>> print("Hello from Huzzah Feather!")
Hello from Huzzah Feather!
>>>

```

Figure 3-30. REPL Console (Feather Huzzah)

I should note that I encountered some minor communication issues with the latest firmware so there may be some instability in that release. Be sure to use the latest firmware available to avoid issues.

WHAT ABOUT OTHER ESP8266 BOARDS?

If you're wondering if you can load MicroPython on other ESP8266 and similar boards, the answer is yes, you can! See "Firmware for ESP8266 boards" at <http://micropython.org/download/#esp8266> for the latest firmware available.

Where to Buy

The Feather Huzzah board is available from Adafruit and resellers of Adafruit products. You can find the board at <https://www.adafruit.com/product/2821>. You can buy the board without headers, with pin headers, or with stacked headers. I recommend buying it with stacked headers so that you can use male/male jumpers with a breadboard, but the normal pin headers are fine too since you can plug the board directly into a breadboard. The cost is approximately \$20 USD with stacked headers.

Now let's briefly discuss some of the other boards that you could use. As you may suspect, this category of boards is the hardest to use and may not be for everyone.

Other Boards

As the popularity of MicroPython grows, you are likely to see more boards available. In fact, currently there are several efforts underway to make MicroPython available on several boards including the Teensy, Arduino, and several variations of the ESP8266 (Espressif) chipset boards. There are some early, limited versions available, like those for the Teensy 3.X version boards, which can be loaded with MicroPython, but the process requires experience in cross-compilation and thus is not for the beginner to tackle (but you're welcome to try!).¹³

However, keep in mind third-party boards may lag somewhat in capability and documentation is usually spotty at best. But now that you've read about the production boards, you should have the knowledge needed to work with newer boards.

We are also starting to see several variants of the Pyboard appear. To date, I've seen two additional boards out of Asia that appear to be clones of the Pyboard. I have tried one of these clone boards and while the GPIO pin layout differs, it behaves exactly like the Pyboard. If you find you want to use several boards in a large project or one project done many times (like in a teaching setting), you may want to consider the clone boards as they could present a considerable savings.

If the board you want to use is not described in this chapter, you can monitor the MicroPython website (<http://micropython.org/download>) for the very latest on availability of a MicroPython firmware install for your board.

Now that we've seen a few of the MicroPython boards that are available, let's dive into the accessories available for you to use to build your projects.

Breakout Boards and Add-Ons

The last bit of hardware we will explore is those boards available for use with the MicroPython board. These can be special, separate modules (called breakout boards), and boards that are specifically designed to be used with a MicroPython board (depending on the board, called shields or skins). The following sections briefly describe some of the breakout boards, shields/skins, and some accessories you may want to consider depending on which board you choose. The list is neither comprehensive nor is it a list of things you must buy. We will see the recommended hardware in each of the example chapters following the discussion on MicroPython.

¹³If you're adventurous and have a Teensy 3.X board, see <https://github.com/micropython/micropython/wiki/Board-Teensy-3.1-3.5-3.6> for how to load MicroPython, but be forewarned it is not a simple affair.

Breakout Boards

Breakout boards are one of the key elements hobbyists and enthusiasts will use in creating a MicroPython (or any microcontroller-based) IOT solution. This is because breakout boards are small circuit boards that contain all the components needed to support a function such as a sensor, network interface, or even a display. Breakout boards also support one of several communication protocols that require only a few pins to be wired, making them very easy to use. In general, they save the developer a lot of time trying to figure out how to design circuits to support a sensor or chip.

To use breakout boards in our projects, we need only to know which interface to use and how to wire it. Fortunately, most vendors provide a datasheet and other documentation to help you wire your board to it. Even if the vendor only has documentation for an Arduino, it is still helpful to learn how to make the connections. The trick is learning how to write the MicroPython code. We will learn more about the MicroPython libraries and hardware support in Chapters 5 and 6. For now, let's explore some of the breakout boards available. Again, what is shown here is only a very small sample.

In fact, Sparkfun has a huge selection of breakout boards. You can find all manner of boards you may find useful. For a full list, see <https://www.sparkfun.com/categories/20>. Adafruit also has a large selection of breakout boards. You can see their products at <https://www.adafruit.com/category/42>. Either vendor has boards that are known to work well and have ample documentation to support them. But, as noted, you may need to adapt them for use with the MicroPython hardware libraries.

Recall from an earlier discussion that there are breakout boards for providing network connections. Also, recall that MicroPython currently supports breakout boards that use the CC3K WiFi chipset as well as breakout boards that use the WIZNET5K Ethernet chipset. We saw an interesting way to use an Arduino CC3K shield earlier, but Figure 3-31 shows the Adafruit CC3000 WiFi breakout board. I show you this for two reasons: first, that you can see how much smaller it is than the Arduino shield; and second, so you can see the row of pins (currently without a header). Notice there are only a few pins here and (from experience if not from reading the documentation), we can see that it supports the same SPI protocol/interface as the shield.¹⁴ Just like the repurposed Arduino shield, we can use this breakout board to add networking to MicroPython boards that don't have that feature.

¹⁴Notice the presence of MOSI and MISO pins – that's an SPI interface.



Figure 3-31. Adafruit CC3000 Module (courtesy of adafruit.com)

The next breakout board is one we will use in a later chapter to read weather data. There are many breakout boards that include sensors for weather data. The most common and easiest to use are those that read temperature, humidity, and barometric pressure. Figure 3-32 shows an example of two such breakout boards: one from Adafruit (<https://www.adafruit.com/product/992>), and another from Sparkfun (<https://www.sparkfun.com/products/13676>). The Adafruit MPL115A2 reads barometric pressure and temperature while the Sparkfun BME 280 measures barometric pressure, humidity, and temperature. Both use the I2C interface/protocol.¹⁵



Figure 3-32. Weather Breakout Boards (courtesy of adafruit.com and sparkfun.com)

¹⁵Notice the presence of SDA and SCL pins – that’s a I2C interface.

The next breakout board is often considered a sensor in and of itself because it performs one and only one function, but it is still a breakout board. Many sensors are packaged this way and are often called sensors when they're breakout boards. Fortunately, most people will know what you mean if you use the wrong term. The way you can tell if it is a breakout board is if the board the sensor is mounted on contains other discrete components and a row of header pins.

In this case, the breakout board measures soil moisture and can be handy in creating a plant monitoring solution as we will do in a later chapter. Figure 3-33 shows a soil moisture breakout board from Sparkfun (<https://www.sparkfun.com/products/13322>).

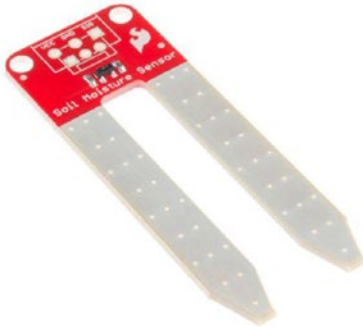


Figure 3-33. Soil Moisture Sensor (courtesy of [sparkfun.com](https://www.sparkfun.com))

Notice the unique shape of the board. The two arms or long prongs are part of the sensor apparatus that measures moisture. Notice at the top is a set of pins (no headers installed) that are used to connect to your board. This breakout board does not use a special interface; rather, the sensor produces voltage that you can measure on one of the analog pins on our MicroPython board. In fact, all we need are three wires: 5V, GND, and signal (which connects to an analog pin on our board). Again, we will see how to use this breakout board in a later chapter.

The last breakout board is a special accessory for the BBC micro:bit board. It is an edge connector that you can use to plug in your BBC micro:bit board to a breadboard, making it easier to wire the pins to other components. You can find this breakout board at Sparkfun (<https://www.sparkfun.com/products/13989>). I highly recommend this or one like it if you choose to use the BBC micro:bit board to run MicroPython.

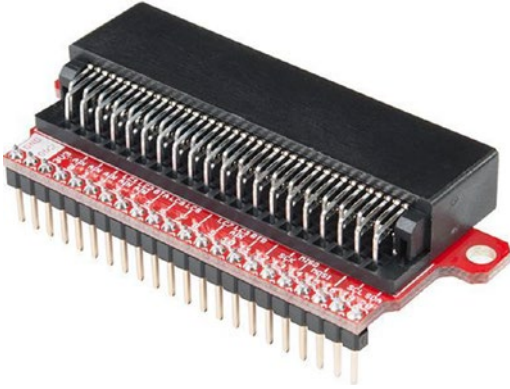


Figure 3-34. BBC micro:bit Edge Breakout Board (courtesy of sparkfun.com)

Board-Specific Shields/Skins

MicroPython board vendors and indeed vendors of a wide array of microcontroller and similar boards package their components so that they can be used for add-on boards called shields or skins or something similar. For example, the Beaglebone add-on boards are called capes and the Raspberry Pi add-on boards are called hats. These are boards that have matching headers and are used to plug into (or stack on top of) the main MicroPython board to add functionality. Many shields or skins have pass-through or stacking headers that enable adding more than one shield at a time. For example, the Arduino and its shield format can be configured with two, three, or more shields added.

In this section, we will see a few of the add-on boards available for the Pyboard and WiPy MicroPython boards. If you decide to use another board for the projects in this book, see the vendor or retailer for availability of add-on boards. Since several of the boards are relatively new, you should check back every few weeks or so to see if any new ones are added.

Pyboard

Pyboard add-on boards are called skins.¹⁶ You can purchase Pyboard skins (and other accessories) from [micropython.org](https://store.micropython.org/store/#/store) (<https://store.micropython.org/store/#/store>). Other retailers may carry them but Adafruit currently does not carry the skins. The skins are normally packaged and sold as unassembled kits so if you want to use one but do not know how to solder, you may need to learn or find someone who can solder the components for you. Fortunately, the small surface-mounted components are already assembled so the most you would have to do is solder larger components like headers.

¹⁶It's just a name. Everyone is trying to be unique but I still call them shields – a habit from my Arduino tinkering.

The first skin is a prototype board that allows you to build a circuit on the skin so that you can use it with your Pyboard. It comes with a complete set of headers. I like it because while it is slightly larger than the Pyboard, there is enough room on there that if you wanted to assemble a circuit you can. Plus, it allows you to remove the skin so that you can use your Pyboard in other projects. Figure 3-35 shows the Protoskin Through-hole XY-Size skin from micropython.org

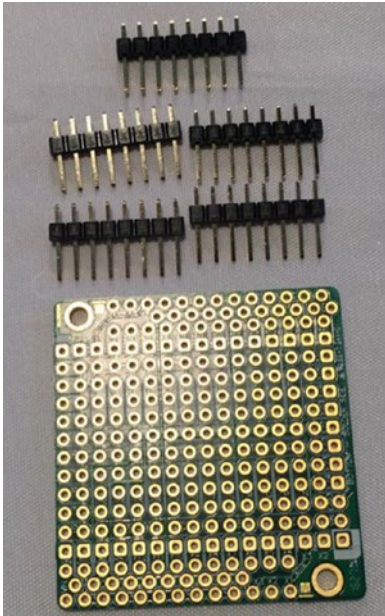


Figure 3-35. *Pyboard Protoskin Through-hole XY-Size*

The next skin is a curious and intriguing feature that adds audio to the Pyboard. It's called simply the Audio Skin and allows you to play and record short sounds as well as play certain waveforms (sounds). I've shown this one to whet your appetite for projects beyond this book. Figure 3-36 shows the Audio Skin from micropython.org.

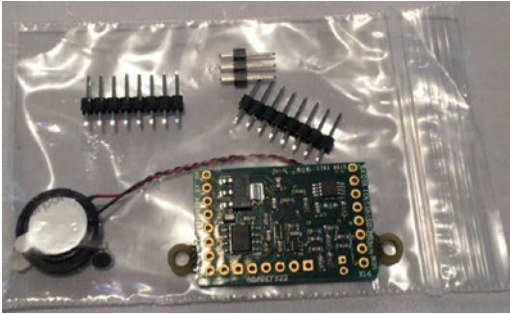


Figure 3-36. *Pyboard Audio Skin*

The next skin is another very intriguing feature that allows you to add a visual element to your Pyboard in the form of a touch LCD. The Color LCD skin with resistive touch and headers adds the ability to add an interactive interface using a 160x128 pixel graphical LCD display with backlight as well as a resistive touch sensor that covers the entire screen. This means you can build small graphical interfaces for your MicroPython projects. Cool! Figure 3-37 shows the front and back of the Color LCD skin with resistive touch and headers skin from micropython.org. Fortunately, you can purchase this skin as a fully assembled kit.

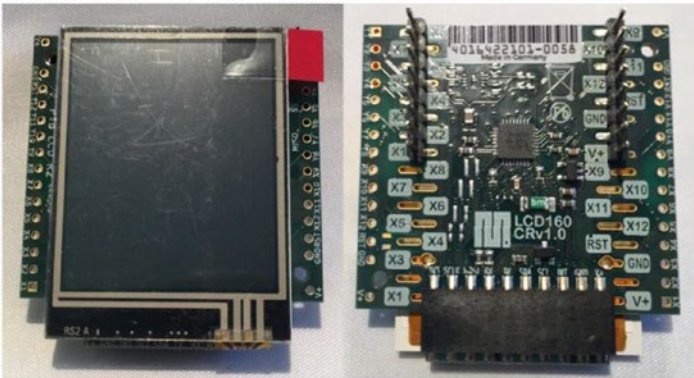


Figure 3-37. *Pyboard LCD Skin*

WiPy and Related

WiPy and Related add-on boards are called shields. Pycom packages and sells their shields fully assembled and ready to use. You can purchase the shields from Pycom (pycom.io) or from other retailers such as Adafruit (adafruit.com). All their shields provide a battery connector with LIPO charger, micro USB for communicating with the board, and a set of headers that fits all Pycom boards (WiPy, SiPy, LoPy, GPy, and FiPy), and a micro SD card reader.

The first shield we saw was the Expansion Board, which I consider a required accessory for the WiPy and related boards. The next is a shield designed to host several sensors called the PySense Shield (<https://www.pycom.io/product/pysense/>). Figure 3-38 shows the PySense Shield.



Figure 3-38. PySense Shield (courtesy of pycom.io)

The list of features for this shield is impressive and includes the following (courtesy of pycom.io). Clearly, there is a lot here that you can use to build upon so if you're considering using the WiPy, this shield and the next may tip the scales. In fact, I will show you how to use this shield in a later chapter to read weather data.

- Ambient light sensor
- Barometric pressure sensor
- Humidity sensor
- 3-axis 12-bit accelerometer
- Temperature sensor
- USB port with serial access
- LiPo battery charger
- MicroSD card compatibility
- Ultra-low power operation (~1uA in deep sleep)

The next (and currently only other) shield available for the WiPy and related boards is the PyTrack Shield (<https://www.pycom.io/product/pytrack/>). This shield is like the PySense shield but instead of a host of sensors (it does have an accelerometer), it has a GPS chip that you can use to record location. Figure 3-39 shows the PyTrack Shield.



Figure 3-39. PyTrack Shield (courtesy of pycom.io)

The list of features for this shield is impressive and includes the following (courtesy of pycom.io). Clearly, there is a lot here that you can use to build upon so if you're considering using the WiPy, this shield and the previous shield may tip the scales.

- Super accurate GNSS + Glonass GPS
- 3 axis 12-bit accelerometer
- USB port with serial access
- LiPo battery charger
- Micro SD card compatibility
- Ultra-low power operation (~1uA in deep sleep)

Board-Specific Accessories

The last category of hardware we will discuss is the accessories available for several of the boards discussed in this chapter. Like almost everything in life, we can accessorize our boards to profound (and costly) ends. While the list of accessories available for MicroPython boards hasn't reached the nerd-like proportions of the Raspberry Pi or Arduino, there are still a few accessories that you should consider.

In this section, I present some of the accessories for each of the Pyboard, WiPy, and BBC micro:bit that I find are essential. That doesn't mean you should rush out and buy them or that they are required for this book; rather, I think you should consider them if you plan to carry on developing with the boards after you complete the examples in this book.

For example, I feel every project should be placed in a case so that the board and any additional components are protected from accidental touches, drops, and other events that can damage the electronics. This is especially true if you plan to mount your board some place for extended operating. Thus, I list a case for each of the boards.

Pyboard

There are not a lot of accessories for the Pyboard other than the essentials such as a USB cable and, of course, the skins. But there is one accessory I find a must for the Pyboard – the case from micropython.org.

■ **Tip** The best source for accessories for the Pyboard is micropython.org (<https://store.micropython.org/store/#/store>). They are based in the European Union and do ship to the United States and other countries.

You can buy a milled aluminum case (called a housing) from micropython.org that is one of the best cases I have ever seen. There are some like it for other boards, but they are typically way overpriced for what you get. That simply doesn't apply to this example. The case is available in three variations: one with an open lid that allows access to the headers, another with a closed lid, and one that allows you to mount the touch-sensitive LCD screen. All of them accommodate the board with the ears so you need not break them off to use this case. Figure 3-40 shows the Pyboard case with the open lid.

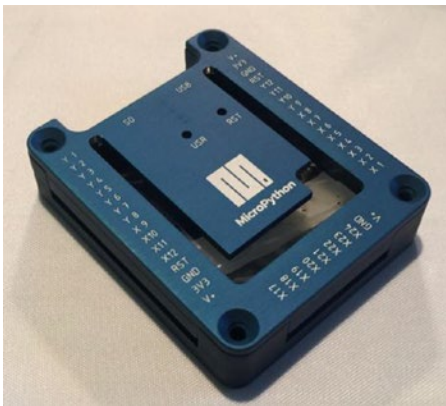


Figure 3-40. PyBoard Case

I cannot say enough about the quality of this case. It is simply the best I've seen anywhere. The open and closed lid variants have lettering on it, labeling all the pins on the header. You won't be disappointed by this case. It is a bit pricey at about \$35 USD, but well worth the price. They are frequently out of stock so check back often for new orders.

WiPy and Related Boards

The list of accessories for the WiPy is growing. You can get the usual essentials such as a USB cable, but Pycom also sells items such as antennas and a wide variety of cases.

■ **Tip** The best source for WiPy accessories is from Pycom (<https://www.pycom.io/webshop#accessories>). While they are based in the European Union, they do ship to the United States with impressive delivery schedules.

Cases for the WiPy vary in color and shape. There are light-duty cases that are sized for the WiPy and shield, cases that are sealed for use outdoors including one large enough to house the WiPy, shield, and LIPO battery. Figure 3-41 shows a smoke-colored case for the WiPy that can accommodate the WiPy and shield.



Figure 3-41. *WiPy Case (courtesy of pycom.io)*

Next is a battery case with the correct connector for attaching to your board. You can get a battery case to contain the correct number of batteries for the correct voltage (check your board specifications first) as well as cases with the correct connectors. I chose to buy a generic case that holds four 1.5V AA batteries and added the correct connector myself. Figure 3-42 shows the battery case I used often in my projects. I've listed the battery case in the WiPy section, but it can be used for other boards such as the BBC micro:bit.



Figure 3-42. Battery Case

■ **Caution** Be sure you get a battery case that contains the right size and power rating for your board.

BBC micro:bit

The board with the most impressive list of accessories available is the BBC micro:bit. I am sure this is due to its popularity, but it favors our efforts too. You can find kits for building robots, power accessories, cases, and much more!

■ **Tip** Two excellent sources for BBC micro:bit accessories are Kitronik (<https://www.kitronik.co.uk/microbit.html>) and The Pi Hut (<https://thepihut.com/collections/micro-bit-store>). Both are in the European Union but ship to the United States with impressive delivery schedules.

One of the power options available for the BBC micro:bit that I like is the MI:power board available from Kitronik (<https://www.kitronik.co.uk/5610-mipower-board-for-the-bbc-microbit.html>). This board is nice because it mounts to your BBC micro:bit board using the large alligator pins (0, 3V, and GND) using only one small 3V coin cell battery for power. It doesn't take up a lot of space when mounted. It includes a switch that allows you

to turn the board off and provides a small speaker (hence the connection via the #0 pin) so you can add sound to your project. Figure 3-43 shows the Power Shield from Kitronik.

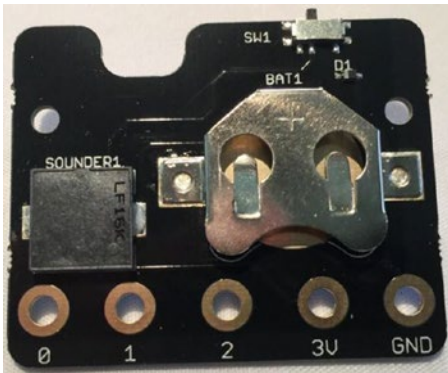


Figure 3-43. MI:power Board for the BBC micro:bit

If you decide to use this battery board, and I highly recommend it over an external battery - especially for the BBC micro:bit, and you decide to use a case, you should know that most cases are not designed to support the MI:power board. However, I did find at least one. As you may surmise, it is made by the same people that made the MI:power!

There are many cases available for the BBC micro:bit with seemingly yet another appearing every day (including those you can print on a 3D printer). However, the one I've found that I like is like others I've used for other boards. It is made from acrylic and uses nylon bolts to join several layers together. It is a clean assembly that allows a clear view of the board, which is nice considering the BBC micro:bit has a lot of LEDs! Figure 3-44 shows the MI:pro case for the BBC micro:bit.



Figure 3-44. BBC micro:bit MI:pro Case Kit

Finally, if you are going to be using the BBC micro:bit a lot to run experiments including those examples in this book as well as the dozens available on the Internet, you should consider the Prototyping System from Kitronik (<https://www.kitronik.co.uk/5609-prototyping-system-for-the-bbc-microbit.html>). This kit comes with a board that you can mount an included edge connector breakout and breadboard making the kit a nice and tidy package. Figure 3-45 shows the Prototyping System from Kitronik. This kit is included in the larger Inventor's Kit as described in the sidebar below.

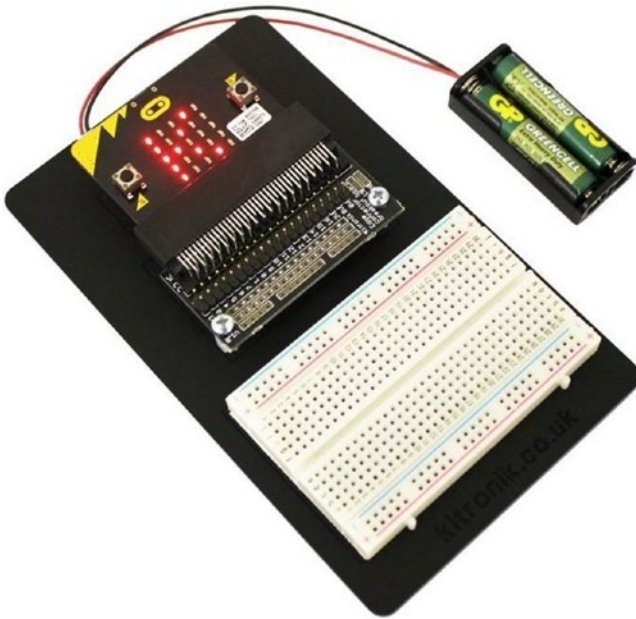


Figure 3-45. Prototyping System for the BBC micro:bit (courtesy of [kitronik.co.uk](http://www.kitronik.co.uk))

Now, let's briefly discuss which board you should buy for use in your MicroPython for the IOT journey.

Which Board Should I Buy?

So, you want to buy a MicroPython board but don't know which to choose. Fortunately, all the boards in this chapter can be an ideal choice. The characteristics that may make some a better fit for some than others include the following.

- *No assembly required* – if you don't know how to solder or don't want to take time to load software, bootstrapping, and firmware, then you should consider a board that is ready-to-go such as the Pyboard or WiPy.

- *Connectivity* – if you plan to make your IOT solution public (as opposed to an academic exercise), you should consider boards that have built-in WiFi or similar networking capabilities such as the WiPy or Feather.
- *Existing Hardware* – if you have already invested money and have a platform and devices (add-on boards, breakout boards, etc.), you may want to consider staying with the board. For example, if you have a lot invested in Arduino, you should consider loading MicroPython on an Arduino board.

Clearly, the choice as to which board to buy requires a bit of forethought. Then again, if you're a true enthusiast, you may decide to buy several of these boards,¹⁷ experimenting for yourself as to which fits your project best. Regardless, the boards I feel are best for learning MicroPython for the IOT are those that have networking such as the WiPy and Feather Huzzah. However, the Pyboard and BBC micro:bit boards are also great boards to learn MicroPython but require an external module to add networking capabilities.

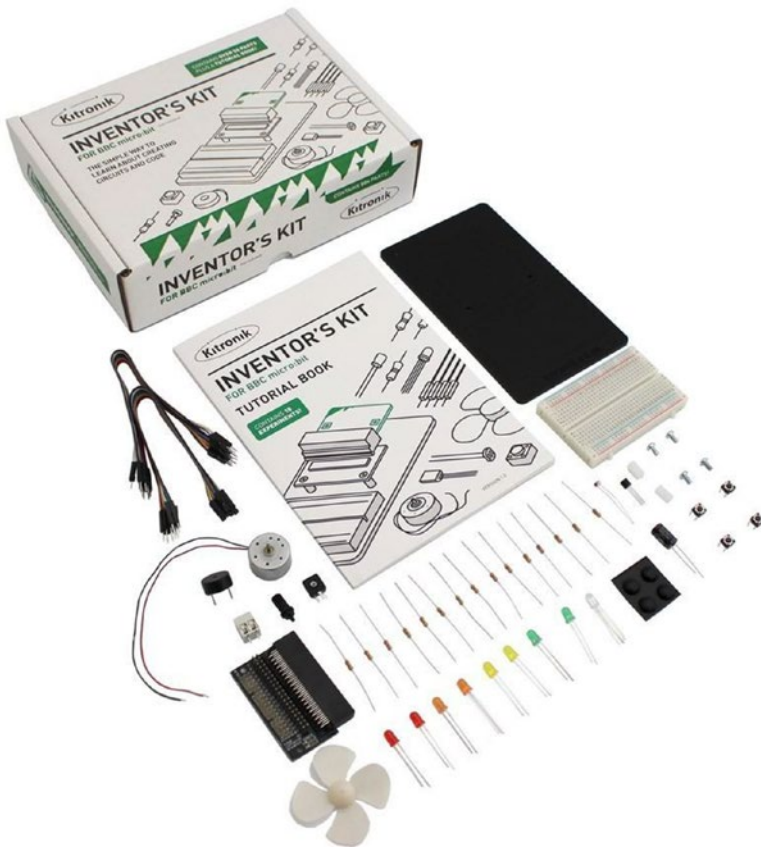
CONSIDER BUYING A KIT

Many of the boards come packaged with a kit containing an assortment of accessories including a getting started kit that includes the board and power adapter. Other kits may include breadboards, and often sensors or add-on boards. For example, you can purchase several kits for the BBC micro:bit board, including two nice options from Sparkfun.

- *BBC micro:bit Go Bundle* – contains the board and battery pack (<https://www.sparkfun.com/products/14336>)
- *Sparkfun Inventor's Kit for BBC micro:bit* – a version of their wildly popular inventor's kit for the BBC micro:bit that includes all the basic electronic components you will need for at least 14 different experiments as well as an easy-to-read instruction booklet (<https://www.sparkfun.com/products/14300>)
- *micro:climate Kit* – contains the board, a weather station add-on board, as well as weather sensors! (<https://www.sparkfun.com/products/14217>)

Other vendors may have additional or similar kits for other boards. For example, Kitronik has an excellent Inventor's Kit for the BBC micro:bit (shown below – courtesy of kitronik.co.uk) that has many of the parts you will need.

¹⁷Yes, I own the boards listed here (some multiple) as well as a growing, shameless horde of other boards.



If you are just starting out and don't have any boards or components, a getting started kit may be the most economical option.

Summary

Wow, that was a lot of information. As you can see, there are several MicroPython boards available. Some like the Pyboard and WiPy come ready to use and do not require anything more than plugging it in and writing your first Python program. Others may require you to load MicroPython first, and others still have a bit of magic needed to make them work. However, all the boards presented here are excellent choices for using in the experiments in this book.

In this chapter, we explored some best practices and tips for using your MicroPython board. We covered the usual things that can go wrong as well as where to go to look for solutions to other problems you may encounter. Finally, we looked at several popular add-ons and breakout boards that you can use to develop your project, including those used later in this book.

In the next chapter, we will dive into a programming tutorial on using Python and MicroPython. The chapter is very much a lightning tour and intended to help guide you to the point where you can write (and understand) the examples in this book.

CHAPTER 4



How to Program in MicroPython

Now that we have a basic understanding of the various MicroPython boards, we can learn more about programming in MicroPython – a very robust and powerful language that you can use to write very powerful applications. Mastering MicroPython is very easy and some may suggest it doesn't require any formal training to use. This is largely true and thus you should be able to write MicroPython scripts with only a little bit of knowledge about the language.

Given that MicroPython is Python, we can learn the basics of the Python language first through examples on our PC. Thus, this chapter presents a crash course on the basics of Python programming including an explanation about some of the most commonly used language features. As such, this chapter will provide you with the skills you need to understand the Python IOT project examples available on the Internet. The chapter also demonstrates how to program in Python through examples that you can run on your PC or your MicroPython board. So, let's get started!

■ **Note** I use the term Python to describe programming concepts in this chapter that apply to both MicroPython and Python. Concepts unique to MicroPython use the term MicroPython.

Now let's learn some of the basic concepts of Python programming. We will begin with the building blocks of the language such as variables, modules, and basic statements, then move into the more complex concepts of flow control and data structures. While the material may seem to come at you in a rush, this tutorial on Python covers only the most fundamental knowledge of the language and how to use it on your PC and MicroPython board. It is intended to get you started writing Python IOT applications.

If you know the basics of Python programming, feel free to skim through this chapter. However, I recommend working through the example projects at the end of the chapter, especially if you've not written many Python applications.

The following sections present many of the basic features of Python programming that you will need to know to understand the example projects in this book.

Basic Concepts

Python is a high-level, interpreted, object-oriented scripting language. One of the biggest goals of Python is to have a clear, easy-to-understand syntax that reads as close to English as possible. That is, you should be able to read a Python script and understand it even if you haven't learned the language. Python also has less punctuation (special symbols) and fewer syntactical machinations than other languages. The following lists a few of the key features of Python.

- An interpreter processes Python at runtime. No external (separate) compiler is used.
- Python supports object-oriented programming constructs by way of a class.
- Python is a great language for the beginner-level programmers and supports the development of a wide range of applications.
- Python is a scripting language but can be used for a wide range of applications.
- Python is very popular and used throughout the world, giving it a huge support base.
- Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- Python code is more clearly defined and visible to the eyes.

Python is available for download (python.org/downloads) for just about every platform that you may encounter or use – even Windows! Python is a very easy language to learn with very few constructs that are even mildly difficult to learn. Rather than toss out a sample application, let's approach learning the basics of Python in a Python-like way: one step at a time.

■ **Note** If you do not have a MicroPython board and have not installed Python on your PC, you should do so now so that you can run the examples in this chapter.

Code Blocks

The first thing you should learn is that Python does not use a code block demarcated with symbols like other languages. More specifically, code that is local to a construct such as a function or conditional or loop is designated using indentation. Thus, the lines below are indented (by spaces or, gag, tabs) so that the starting characters align for the code body of the construct.

■ **Tip** The following shows this concept in action. Python interpreters will complain and could produce strange results if the indentation is not uniform.

```
if (expr1):
    print("inside expr1")
    print("still inside expr1")
else:
    print("inside else")
    print("still inside else")
print("in outer level")
```

Here we see a conditional or if statement. Notice the function call `print()` is indented. This signals the interpreter that the lines belong to the construct above it. For example, the two `print` statements that mention `expr1` form the code block for the if condition (and executes when the expression evaluates to true). Similarly, the next two `print` statements form the code block for the else condition. Finally, the non-indented lines are not part of the conditional and thus are executed after either the if or else depending on the expression evaluation.

As you can see, indentation is a key concept to learn when writing Python. Even though it is very simple, making mistakes in indentation can result in code executing that you did not expect or worse errors from the interpreter.

■ **Note** I use ‘program’ and ‘application’ interchangeably with ‘script’ when discussing Python. While technically, Python code saved in a file is a script, we often use it in contexts where ‘program’ or ‘application’ are more appropriate.

There is one special symbol that you will encounter frequently. Notice the use of the colon (`:`) in the code above. This symbol is used to terminate a construct and signals the interpreter that the declaration is complete and the body of the code block follows. We use this for conditionals, loops, classes, and functions.

Comments

One of the most fundamental concepts in any programming language is the ability to annotate your source code with non-executable text that not only allows you to make notes among the lines of code but also forms a way to document your source code.

To add comments to your source code, use the pound sign (`#`). Place at least one at the start of the line to create a comment for that line, repeating the `#` symbols for each subsequent line. This creates what is known as a block comment as shown. Notice I used a comment without any text to create whitespace. This helps with readability and is a common practice for block comments.

```
#
# MicroPython for the IOT
#
# Example Python application.
#
# Created by Dr. Charles Bell
#
```

You can also place comments on the same line as the source code. The compiler will ignore anything from the pound sign to the end of the line. For example, the following shows a common style of documenting variables.

```
zip = 35012           # Zip or postal code
address1= "123 Main St." # Store the street address
```

Arithmetic

You can perform many mathematical operations in Python including the usual primitives but also logical operations and operations used to compare values. Rather than discuss these in detail, I provide a quick reference in Table 4-1 that shows the operation and example of how to use the operation.

Table 4-1. *Arithmetic, Logical, and Comparison Operators in Python*

Type	Operator	Description	Example
Arithmetic	+	Addition	int_var + 1
	-	Subtraction	int_var - 1
	*	Multiplication	int_var * 2
	/	Division	int_var / 3
	%	Modulus	int_var % 4
	-	Unary subtraction	-int_var
	+	Unary addition	+int_var
Logical	&	Bitwise and	var1&var2
		Bitwise or	var1 var2
	^	Bitwise exclusive or	var1^var2
	~	Bitwise complement	~var1
	and	Logical and	var1and var2
	or	Logical or	var1or var2

(continued)

Table 4-1. (continued)

Type	Operator	Description	Example
Comparison	==	Equal	expr1==expr2
	!=	Not equal	expr1!=expr2
	<	Less than	expr1<expr2
	>	Greater than	expr1>expr2
	<=	Less than or equal	expr1<=expr2
	>=	Greater than or equal	expr1>=expr2

Bitwise operations produce a result on the values performed on each bit. Logical operators (and, or) produce a value that is either true or false and are often used with expressions or conditions.

Output to Screen

We've already seen a few examples of how to print messages to the screen but without any explanation about the statements shown. While it is unlikely that you would print output from your MicroPython board for projects that you deploy, learning Python is much easier when you can display messages to the screen.

Some of the things you may want to print – as we have seen in previous examples – is to communicate what is going on inside your program. This can include simple messages (strings), but can also include the values of variables, expressions, and more.

As we have seen, the built-in `print()` function is the most common way to display output text contained within single or double quotes. We have also seen some interesting examples using another function named `format()`. The `format()` function generates a string for each argument passed. These arguments can be other strings, expressions, variables, etc. The function is used with a special string that contains replacement keys delimited by curly braces `{ }` (called *string interpolation*¹). Each replacement key contains either an index (starting at 0) or a named keyword. The special string is called a format string. Let's see a few examples to illustrate the concept. You can run these yourself either on your PC or your MicroPython board. I include the output so you can see what each statement does.

```
>>> a = 42
>>> b = 1.5
>>> c = "seventy"
>>> print("{0} {1} {2} {3}".format(a,b,c,(2+3)))
42 1.5 seventy 5
>>> print("{a_var} {b_var} {c_var} {0}".format((3*3),c_var=c,b_var=b,a_var=a))
42 1.5 seventy 9
```

¹https://en.wikipedia.org/wiki/String_interpolation

Notice I created three variables (we will talk about variables in the next section) assigning them different values with the equal symbol (=). I then printed a message using a format string with four replacement keys labeled using an index. Notice the output of that print statement. Notice I included an expression at the end to show how the `format()` function evaluates expressions.

The last line is more interesting. Here, I use three named parameters (`a_var`, `b_var`, `c_var`) and used a special argument option in the `format()` function where I assign the parameter a value. Notice I listed them in a different order. This is the greatest advantage of using named parameters; they can appear in any order but are placed in the format string in the position indicated.

As you can see, it's just a case of replacing the `{ }` keys with those from the `format()` function, which converts the arguments to strings. We use this technique anywhere we need a string that contains data gathered from more than one area or type. We can see this in the examples above.

■ **Tip** See <https://docs.python.org/3/library/string.html#formatstrings> for more information about format strings.

Now let's look at how we can use variables in our programs (scripts).

■ **Tip** For those who have learned to program in another language like C or C++, Python allows you to terminate a statement with the semicolon (`;`); however, it is not needed and considered bad form to include it.

Variables

Python is a dynamically typed language, which means the type of the variable (the type of data it can store) is determined by context as it is encountered or used. This contrasts with other language such as C and C++ where you must declare the type before you use the variable.

Variables in Python are simply named memory locations that you can use to store values during execution. We store values by using the equal sign to assign the value. Python variable names can be anything you want, but there are rules and conventions most Python most developers follow. The rules are listed in the Python coding standard.²

However, the general, overriding rule requires variable names that are descriptive, have meaning in context, and can be easily read. That is, you should avoid names with random characters, forced abbreviations, acronyms, and similar obscure names. By convention, your variable names should be longer than a single character (with some acceptable exceptions for loop counting variables) and short enough to avoid overly long code lines.

²<https://www.python.org/dev/peps/pep-0008/>

WHAT IS A LONG CODE LINE?

Most will say a code line should not exceed 80 characters, but this harkens from the darker days of programming when we used punched cards that permitted a maximum of 80 characters per card and later display devices with the same limitation. With modern, widescreen displays this is not as big a deal, but I still recommend keeping lines short to ensure better readability. No one likes to scroll down (or right) to read!

Thus, there is a lot of flexibility in what you can name your variables. There are additional rules and guidelines in the PEP8 standard and should you wish to bring your project source code up to date with the standards, you should review the PEP8 naming standards for functions, classes, and more. See the PEP8 coding guidelines for Python coding at <https://www.python.org/dev/peps/pep-0008> for a complete list of the rules and standards.

The following shows some examples of simple variables and their dynamically determined types.

```
# floating point number
length = 10.0
# integer
width = 4
# string
box_label = "Tools"
# list
car_makers = ['Ford', 'Chevrolet', 'Dodge']
# tuple
porsche_cars = ('911', 'Cayman', 'Boxster')
# dictionary
address = {"name": "Joe Smith", "Street": "123 Main", "City": "Anytown",
"State": "New Happyville"}
```

So, how did we know the variable `width` is an integer? Simply because the number 4 is an integer. Likewise, Python will interpret “Tools” as a string. We’ll see more about the last three types and other types supported by Python in the next section.

■ **Tip** For more information about naming conventions governed by the Python coding standard (PEP8), see <https://www.python.org/dev/peps/pep-0008/#naming-conventions>.

Types

As mentioned, Python does not have a formal type specification mechanism like other languages. However, you can still define variables to store anything you want. In fact, Python permits you to create and use variables based on context and you can use initialization to “set” the data type for the variable. The following shows several examples.

```
# Numbers
float_value = 9.75
integer_value = 5

# Strings
my_string = "He says, he's already got one."

print("Floating number: {}".format(float_value))
print("Integer number: {}".format(integer_value))
print(my_string)
```

For situations where you need to convert types or want to be sure values are typed a certain way, there are many functions for converting data. Table 4-2 shows a few of the more commonly used type conversion functions. I discuss some of the data structures in a later section.

Table 4-2. *Type Conversion in Python*

Function	Description
<code>int(x [,base])</code>	Converts <code>x</code> to an integer. Base is optional (e.g., 16 for hex).
<code>long(x [,base])</code>	Converts <code>x</code> to a long integer.
<code>float(x)</code>	Converts <code>x</code> to a floating point.
<code>str(x)</code>	Converts object <code>x</code> to a string.
<code>tuple(t)</code>	Converts <code>t</code> to a tuple.
<code>list(l)</code>	Converts <code>l</code> to a list.
<code>set(s)</code>	Converts <code>s</code> to a set.
<code>dict(d)</code>	Creates a dictionary.
<code>chr(x)</code>	Converts an integer to a character.
<code>hex(x)</code>	Converts an integer to a hexadecimal string.
<code>oct(x)</code>	Converts an integer to an octal string.

However, you should use these conversion functions with care to avoid data loss or rounding. For example, converting a float to an integer can result in truncation. Likewise, printing floating-point numbers can result in rounding.

Now let’s look at some commonly used data structures including this strange thing called a dictionary.

Basic Data Structures

What you have learned so far about Python is enough to write the most basic programs and indeed more than enough to tackle the example projects later in this chapter. However, when you start needing to operate on data – either from the user or from sensors and similar sources – you will need a way to organize and store data as well as perform operations on the data in memory. The following introduces three data structures in order of complexity: lists, tuples, and dictionary.

Lists

Lists are a way to organize data in Python. It is a free-form way to build a collection. That is, the items (or elements) need not be the same data type. Lists also allow you to do some interesting operations such as adding things at the end, beginning, or at a special index. The following demonstrates how to create a list.

```
# List
my_list = ["abacab", 575, "rex, the wonder dog", 24, 5, 6]
my_list.append("end")
my_list.insert(0, "begin")
for item in my_list:
    print("{0}".format(item))
```

Here we see I created the list using square brackets ([]). The items in the list definition are separated by commas. Note that you can create an empty list simply by setting a variable equal to []. Since lists, like other data structures, are objects, there are several operations available for lists such as the following.

- `append(x)`: add x to the end of the list
- `extend(l)`: add all items to the end of the list
- `insert(pos, item)`: insert item at a position pos
- `remove(value)`: remove the first item that matches (==) the value
- `pop([i])`: remove and return the item at position i or end of list
- `index(value)`: return index of first item that matches
- `count(value)`: count occurrences of value
- `sort()`: sort the list (ascending)
- `reverse()`: reverse sort the list

Lists are like arrays in other languages and very useful for building dynamic collections of data.

Tuples

Tuples, on the other hand, are a more restrictive type of collection. That is, they are built from a specific set of data and do not allow manipulation like a list. In fact, you cannot change the elements in the tuple. Thus, we can use tuples for data that should not change. The following shows an example of a tuple and how to use it.

```
# Tuple
my_tuple = (0,1,2,3,4,5,6,7,8, "nine")
for item in my_tuple:
    print("{0}".format(item))
if 7 in my_tuple:
    print("7 is in the list")
```

Here we see I created the tuple using parenthesis (). The items in the tuple definition are separated by commas. Note that you can create an empty tuple simply by setting a variable equal to (). Since tuples, like other data structures, are objects, there are several operations available such as the following, including operations for sequences such as inclusion, location, etc.

- `x in t`: determine if `t` contains `x`
- `x not in t`: determine if `t` does not contain `x`
- `s + t`: concatenate tuples
- `s[i]`: get element `i`
- `len(t)`: length of `t` (number of elements)
- `min(t)`: minimal (smallest value)
- `max(t)`: maximal (largest value)

If you want even more structure with storing data in memory, you can use a special construct (object) called a dictionary.

Dictionaries

A dictionary is a data structure that allows you to store key, value pairs where the data is assessed via the keys. Dictionaries are a very structured way of working with data and the most logical form we will want to use when collecting complex data. The following shows an example of a dictionary.

```
# Dictionary
my_dictionary = {
    'first_name': "Chuck",
    'last_name': "Bell",
    'age': 36,
    'my_ip': (192,168,1,225),
    42: "What is the meaning of life?",
}
```

```
# Access the keys:
print(my_dictionary.keys())
# Access the items (key, value) pairs
print(my_dictionary.items())
# Access the values
print(my_dictionary.values())
# Create a list of dictionaries
my_addresses = [my_dictionary]
```

There is a lot going on here! We see a basic dictionary declaration that uses curly braces to create a dictionary. Inside that, we can create as many key, value pairs we want separated by commas. Keys are defined using strings (I use single quotes by convention but double quotes will work) or integers, and values can be any data type we want. For the `my_ip` attribute, we are also storing a tuple.

Following the dictionary, we see several operations performed on the dictionary from printing the keys, printing all the values, and printing only the values. The following shows the output of executing this code snippet from the Python interpreter.

```
[42, 'first_name', 'last_name', 'age', 'my_ip']
[(42, 'what is the meaning of life?'), ('first_name', 'Chuck'), ('last_name', 'Bell'), ('age', 36), ('my_ip', (192, 168, 1, 225))]
['what is the meaning of life?', 'Chuck', 'Bell', 36, (192, 168, 1, 225)]
'42': what is the meaning of life?
'first_name': Chuck
'last_name': Bell
'age': 36
'my_ip': (192, 168, 1, 225)
```

As we have seen in this example, there are several operations (functions or methods) available for dictionaries including the following. Together this list of operations makes dictionaries a very powerful programming tool.

- `len(d)`: number of items in `d`
- `d[k]`: item of `d` with key `k`
- `d[k] = x`: assign key `k` with value `x`
- `del d[k]`: delete item with key `k`
- `k in d`: determine if `d` has an item with key `k`
- `d.items()`: return a list (view) of the (key, value) pairs in `d`
- `d.keys()`: return a list (view) of the keys in `d`
- `d.values()`: return a list (view) of the values in `d`

Best of all, objects can be placed inside other objects. For example, you can create a list of dictionaries like I did above, a dictionary that contains lists and tuples, and any

combination you need. Thus, lists, tuples, and dictionaries are a powerful way to manage data for your program.

In the next section, we learn how we can control the flow of our programs.

Statements

Now that we know more about the basics of Python, we can discover some of the more complex code concepts you will need to complete your project such as conditional statements and loops.

Conditional Statements

We have also seen some simple conditional statements: statements designed to alter the flow of execution depending on the evaluation of one or more expressions. Conditional statements allow us to direct execution of our programs to sections (blocks) of code based on the evaluation of one or more expressions. The conditional statement in Python is the `if` statement.

We have seen the `if` statement in action in our example code. Notice in the example, we can have one or more (optional) `else` phrases that we execute once the expression for the `if` conditions evaluate to false. We can chain `if / else` statements to encompass multiple conditions where the code executed depends on the evaluation of several conditions. The following shows the general structure of the `if` statement. Notice in the comments how I explain how execution reaches the body of each condition.

```
if (expr1):
    # execute only if expr1 is true
elif ((expr2) or (expr3)):
    # execute only if expr1 is false *and* either expr2 or expr3 is true
else:
    # execute if both sets of if conditions evaluate to false
```

While you can chain the statement as much as you want, use some care here because the more `elif` sections you have, the harder it will become to understand, maintain, and avoid logic errors in your expressions.

There is another form of conditional statement called a ternary operator. Ternary operators are more commonly known as conditional expressions in Python. These operators evaluate something based on a condition being true or not. They became a part of Python in version 2.4. Conditional expressions are a shorthand notation for an `if-then-else` construct used (typically) in an assignment statement as shown below.

```
variable = value_if_true if condition else value_if_false
```


Here we see if the condition is evaluated to true, the value preceding the if is used but if the condition evaluates to false, the value following the else is used. The following shows a short example.

```
>>> numbers = [1,2,3,4]
>>> for n in numbers:
...     x = 'odd' if n % 2 else 'even'
...     print("{0} is {1}.".format(n, x))
...
1 is odd.
2 is even.
3 is odd.
4 is even.
>>>
```

Conditional expressions allow you to quickly test a condition instead of a using a multiline conditional statement, which can help make your code a bit easier to read (and shorter).

Loops

Loops are used to control the repetitive execution of a block of code. There are three forms of loops that have slightly different behaviors. All loops use conditional statements to determine whether to repeat execution or not. That is, they repeat as long as the condition is true. The two types of loops are while and for. I explain each with an example.

The while loop has its condition at the “top” or start of the block of code. Thus, while loops only execute the body if and only if the condition evaluates to true on the first pass. The following illustrates the syntax for a while loop. This form of loop is best used when you need to execute code only if some expression(s) evaluate to true. For example, iterating through a collection of things whose number of elements is unknown (loop until we run out of things in the collection).

```
while (expression):
    # do something here
```

For loops are sometimes called counting loops because of their unique form. For loops allow you to define a counting variable and a range or list to iterate over. The following illustrates the structure of the for loop. This form of loop is best used for performing an operation in a collection. In this case, Python will automatically place each item in the collection in the variable for each pass of the loop until no more items are available.

```
for variable_name in list:
    # do something here
```

You can also do range loops or counting loops. This uses a special function called `range()` that takes up to three parameters, `range([start], stop[, step])`, where `start` is the starting number (an integer), `stop` is the last number in the series, and `step` is the increment. So, you can count by 1, 2, 3, etc., through a range of numbers. The following shows a simple example.

```
for i in range(2,9):
    # do something here
```

There are other uses for `range()` that you may encounter. See the documentation on this function and other built-in functions at <https://docs.python.org/3/library/functions.html> for more information.

Python also provides a mechanism for controlling the flow of the loop (e.g., duration or termination) using a few special keywords as follows.

- `break`: exit the loop body immediately
- `continue`: skip to next iteration of the loop
- `else`: execute code when loop ends (not executed if the loop was stopped with a `break` statement)

There are some uses for these keywords, particularly `break`, but it is not the preferred method of terminating and controlling loops. That is, professionals believe the conditional expression or error handling code should behave well enough to not need these options.

Modularization; Modules, Functions, and Classes

The last groups of topics are the most advanced and include modularization (code organization). As we will see, we can use functions to group code, eliminate duplication, and to encapsulate functionality into objects.

Including Modules

Python applications can be built from reusable libraries that are provided by the Python environment. They can also be built from custom modules or libraries that you create yourself or download from a third party. These are often distributed as a set of Python code files (for example, files that have a file extension of `.py`). When we want to use a library (function, class, etc.) that is included in a module, we use the `import` keyword and list the name of the module. The following shows some examples.

```
import os
import sys
```

The first two lines demonstrate how to import a base or common module provided by Python. In this case, we are using or importing modules for the `os` and `sys` modules (operating system and Python system functions).

■ **Tip** It is customary (but not required) to list your imports in alphabetical order with built-in modules first then third-party modules listed next, and finally your own modules.

Functions

Python allows you to use modularization in your code. While it supports object-oriented programming by way of classes (a more advanced feature that you are unlikely to encounter for most Python GPIO examples), on a more fundamental level you can break your code into smaller chunks using functions.

Functions use a special keyword construct (rare in Python) to define a function. We simply use `def` followed by a name for the function and a comma-separated list of parameters in parentheses. The colon is used to terminate the declaration. The following shows an example.

```
def print_dictionary(the_dictionary):
    for key, value in the_dictionary.items():
        print("{}{}: {}".format(key, value))

# define some data
my_dictionary = {
    'name': "Chuck",
    'age': 37,
}
```

You may be wondering what this strange code does. Notice the loop is assigning two values from the result of the `items()` function. This is a special function available from the dictionary object.³ The `items()` function returns the key, value pairs: hence the names of the variables.

The next line prints out the values. The use of formatting strings where the curly braces define the parameter number starting at 0 is common for Python 3 applications. See the Python documentation for more information about formatting strings (<https://docs.python.org/3/library/string.html#format-string-syntax>).

The body of the function is indented. All statements indented under this function declaration belong to the function and are executed when the function is called. We can call functions by name providing any parameters as follows. Notice how I referenced the values in the dictionary by using the key names.

```
print_dictionary(my_dictionary)
print(my_dictionary['age'])
print(my_dictionary['name'])
```

³Yes, dictionaries are objects! So are tuples and lists and many other data structures.

This example together with the code above, when executed, generates the following.

```
$ python3
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def print_dictionary(the_dictionary):
...     for key, value in the_dictionary.items():
...         print("{}{}: {}".format(key, value))
...
>>> # define some data
... my_dictionary = {
...     'name': "Chuck",
...     'age': 41,
... }
>>> print_dictionary(my_dictionary)
'name': Chuck
'age': 41
>>> print(my_dictionary['age'])
41
>>> print(my_dictionary['name'])
Chuck
```

Now let's look at the most complex concept in Python – object-oriented programming.

Classes and Objects

You may have heard that Python is an object-oriented programming language. But what does that mean? Simply, Python is a programming language that provides facilities for describing objects (things) and what you can do with the object (operations). Objects are an advanced form of data abstraction where the data is hidden from the caller and only manipulated by the operations (methods) the object provides.

The syntax we use in Python is the `class` statement, which you can use to help make your projects modular. By modular, we mean the source code is arranged to make it easier to develop and maintain. Typically, we place classes in separate modules (code files), which helps organize the code better. While it is not required, I recommend using this technique of placing a class in its own source file. This makes modifying the class or fixing problems (bugs) easier.

So, what are Python classes? Let's begin by considering the construct as an organization technique. We can use the class to group data and methods together. The name of the class immediately follows the keyword `class` followed by a colon. You declare other class methods like any other method except the first argument must be `self`, which ties the method to the class instance when executed.

■ **Note** I prefer to use terms that have been adopted by the language designers or community of developers. For example, some use ‘function’ but others may use ‘method’. Still others may use subroutine, routine, procedure, etc. It doesn’t matter which term you use, but you should strive to use terms consistently. One example, which can be confusing to some, is I use the term method when discussing object-oriented examples. That is, a class has methods not functions. However, you can use function in place of method and you’d still be correct (mostly).

Accessing the data is done using one or more methods by using the class (creating an instance) and using dot notation to reference the data member or function. Let’s look at an example. Listing 4-1 shows a complete class that describes (models) the most basic characteristics of a vehicle used for transportation. I created a file named `vehicle.py` to contain this code.

Listing 4-1. Vehicle class

```
#
# MicroPython for the IOT
#
# Class Example: A generic vehicle
#
# Dr. Charles Bell
#
class Vehicle:
    """Base class for defining vehicles"""
    axles = 0
    doors = 0
    occupants = 0

    def __init__(self, num_axles, num_doors):
        self.axles = num_axles
        self.doors = num_doors

    def get_axles(self):
        return self.axles

    def get_doors(self):
        return self.doors

    def add_occupant(self):
        self.occupants += 1

    def num_occupants(self):
        return self.occupants
```

Notice a couple of things here. First, there is a method with the name `__init__()`. This is the constructor and is called when the class instance is created. You place all your initialization code like setting variables in this method. We also have methods for returning the number of axles, doors, and occupants. We have one method in this class: to add occupants.

Also notice we address each of the class attributes (data) using `self.<name>`. This is how we can ensure we always access the data that is associated with the instance created and not a global variable or other local variable.

Let's see how this class can be used to define a family sedan. Listing 4-2 shows code that uses this class. We can place this code in a file named `sedan.py`.

Listing 4-2. Using the Vehicle class

```
#
# MicroPython for the IOT
#
# Class Example: Using the generic Vehicle class
#
# Dr. Charles Bell
#
from vehicle import Vehicle

sedan = Vehicle(2, 4)
sedan.add_occupant()
sedan.add_occupant()
sedan.add_occupant()
print("The car has {0} occupants.".format(sedan.num_occupants()))
```

Notice the first line imports the Vehicle class from the vehicle module. Notice I capitalized the class name but not the file name. This is a very common naming scheme. Next in the code, we create an instance of the class. Notice I passed in 2, 4 to the class name. This will cause the `__init__()` method to be called when the class is instantiated. The variable, `sedan`, becomes the class instance variable (object) that we can manipulate, and I do so by adding three occupants then printing out the number of occupants using the method in the Vehicle class.

We can run the code on our PC using the following command. As we can see, it tells us there are three occupants in the vehicle when the code is run. Nice.

```
$ python ./sedan.py
The car has 3 occupants.
```

OBJECT-ORIENTED PROGRAMMING (OOP) TERMINOLOGY

Like any technology or concept, there comes a certain number of terms that you must learn to be able to understand and communicate with others about the technology. The following briefly describes some of the terms you will need to know to learn more about object-oriented programming.

Attribute: A data element in a class.

Class: A code construct used to define an object in the form of attributes (data) and methods (functions) that operate on the data. Methods and attributes in Python can be accessed using dot notation.

Class instance variable: A variable that is used to store an instance of an object. They are used like any other variable and, combined with dot notation, allow us to manipulate objects.

Instance: An executable form of a class created by assigning a class to a variable initializing the code as an object.

Inheritance: The inclusion of attributes and methods from one class in another.

Instantiation: The creation of an instance of a class.

Method overloading: The creation of two or more methods with the same name but with a different set of parameters. This allows us to create methods that have the same name but may operate differently depending on the parameters passed.

Polymorphism: Inheriting attributes and methods from a base class adding additional methods or overriding (changing) methods.

There are many more OOP terms, but these are the ones you will encounter most often.

Now, let's see how we can use the vehicle class to demonstrate inheritance. In this case, we will create a new class named `PickupTruck` that uses the vehicle class but adds specialization to the resulting class. Listing 4-3 shows the new class. I placed this code in a file named `pickup_truck.py`. As you will see, a pickup truck is a type of vehicle.

Listing 4-3. Pickup Truck class

```
#
# MicroPython for the IOT
#
# Class Example: Inheriting the Vehicle class to form a
# model of a pickup truck with maximum occupants and maximum
# payload.
```

```

#
# Dr. Charles Bell
#
from vehicle import Vehicle

class PickupTruck(Vehicle):
    """This is a pickup truck that has:
    axles = 2,
    doors = 2,
    __max_occupants = 3
    The maximum payload is set on instantiation.
    """
    occupants = 0
    payload = 0
    max_payload = 0

    def __init__(self, max_weight):
        super().__init__(2,2)
        self.max_payload = max_weight
        self.__max_occupants = 3

    def add_occupant(self):
        if (self.occupants < self.__max_occupants):
            super().add_occupant()
        else:
            print("Sorry, only 3 occupants are permitted in the truck.")

    def add_payload(self, num_pounds):
        if ((self.payload + num_pounds) < self.max_payload):
            self.payload += num_pounds
        else:
            print("Overloaded!")

    def remove_payload(self, num_pounds):
        if ((self.payload - num_pounds) >= 0):
            self.payload -= num_pounds
        else:
            print("Nothing in the truck.")

    def get_payload(self):
        return self.payload

```

Notice a few things here. First, notice the class statement: `class PickupTruck(Vehicle) :`. When we want to inherit from another class, we add the parentheses with the name of the base class. This ensures Python will use the base class allowing the derived class to use all its accessible data and memory. If you want to inherit from more than one class, you can (called multiple inheritance), just list the base (parent) classes with a comma-separated list.

Next, notice the `__max_occupants` variable. Using two underscores in a class for an attribute or a method makes that, through convention, the item private to the class.⁴ That is, it should only be accessed from within the class. No caller of the class (via a class variable/instance) can access the private items nor can any class derived from the class. It is always a good practice to hide the attributes (data).

You may be wondering what happened to the occupant methods. Why aren't they in the new class? They aren't there because our new class inherited all that behavior from the base class. Not only that, but the code has been modified to limit occupants to exactly three occupants.

I also want to point out the documentation I added to the class. We use documentation strings (strings that use a set of three double quotes before and after) to document the class. You can put documentation here to explain the class and its methods. We'll see a good use of this a bit later.

Finally, notice the code in the constructor. This demonstrates how to call the base class method, which I do to set the number of axles and doors. We can do the same in other methods if we wanted to call the base class method's version.

Now, let's write some code to use this class. Listing 4-4 shows the code we used to test this class. Here, we create a file named `pickup.py` that creates an instance of the pickup truck, adds occupants, and payload, then prints out the contents of the truck.

Listing 4-4. Using the `PickupTruck` class

```
#
# MicroPython for the IOT
#
# Class Example: Exercising the PickupTruck class.
#
# Dr. Charles Bell
#
from pickup_truck import PickupTruck

pickup = PickupTruck(500)
pickup.add_occupant()
pickup.add_occupant()
pickup.add_occupant()
pickup.add_occupant()
pickup.add_payload(100)
pickup.add_payload(300)
print("Number of occupants in truck = {0}.".format(pickup.num_occupants()))
print("Weight in truck = {0}.".format(pickup.get_payload()))
pickup.add_payload(200)
pickup.remove_payload(400)
pickup.remove_payload(10)
```

⁴Technically, it is called name mangling, which simulates making something private, but can still be accessed if you provide the correct number of underscores. For more information, see https://en.wikipedia.org/wiki/Name_mangling.

Notice I add a couple of calls to the `add_occupant()` method, which the new class inherits and overrides. I also add calls so that we can test the code in the methods that check for excessive occupants and maximum payload capacity. When we run this code, we will see the results as shown below.

```
$ python ./pickup.py
Sorry, only 3 occupants are permitted in the truck.
Number of occupants in truck = 3.
Weight in truck = 400.
Overloaded!
Nothing in the truck.
```

Once again, I ran this code on my PC, but I can run all this code on the MicroPython board and will see the same results.

There is one more thing we should learn about classes: built-in attributes. Recall the `__init__()` method. Python automatically provides several built-in attributes each starting with `__` that you can use to learn more about objects. The following lists a few of the operators available for classes.

- `__dict__`: Dictionary containing the class namespace
- `__doc__`: Class documentation string
- `__name__`: Class name
- `__module__`: Module name where the class is defined
- `__bases__`: The base class(es) in order of inheritance

The following shows what each of these attributes returns for the `PickupTruck` class above. I added this code to the `pickup.py` file.

```
print("PickupTruck.__doc__:", PickupTruck.__doc__)
print("PickupTruck.__name__:", PickupTruck.__name__)
print("PickupTruck.__module__:", PickupTruck.__module__)
print("PickupTruck.__bases__:", PickupTruck.__bases__)
print("PickupTruck.__dict__:", PickupTruck.__dict__)
```

When this code is run, we see the following output.

```
PickupTruck.__doc__: This is a pickup truck that has:
    axles = 2,
    doors = 2,
    max occupants = 3
    The maximum payload is set on instantiation.

PickupTruck.__name__: PickupTruck
PickupTruck.__module__: pickup_truck
PickupTruck.__bases__: (<class 'vehicle.Vehicle'>,)

```

```
PickupTruck.__dict__: {'__module__': 'pickup_truck', '__doc__': 'This is a
pickup truck that has:\n  axles = 2,\n  doors = 2,\n  max occupants
= 3\n  The maximum payload is set on instantiation.\n  ', 'occupants':
0, 'payload': 0, 'max_payload': 0, '_PickupTruck_max_occupants': 3,
'__init__': <function PickupTruck.__init__ at 0x1018a1488>, 'add_occupant':
<function PickupTruck.add_occupant at 0x1018a17b8>, 'add_payload': <function
PickupTruck.add_payload at 0x1018a1840>, 'remove_payload': <function
PickupTruck.remove_payload at 0x1018a18c8>, 'get_payload': <function
PickupTruck.get_payload at 0x1018a1950>}
```

You can use the built-in attributes whenever you need more information about a class. Notice the `_PickupTruck_max_occupants` entry in the dictionary. Recall that we made a pseudo-private variable, `__max_occupants`. Here, we see how Python refers to the variable by prepending the class name to the variable. Remember, variables that start with two underscores (not one) indicates it should be considered private to the class and only usable from within the class.

■ **Tip** For more information about classes in Python, see <https://docs.python.org/3/tutorial/classes.html>.

Now, let's see a few examples of Python programs that we can use to practice. Like the previous examples, you can write and execute these either on your PC or on your MicroPython board.

Learning Python by Example

The best way to learn how to program in any language is practicing with examples. In this section, I present several examples that you can use to practice coding in Python. You can use either your MicroPython board or your PC to run these examples. I present the first two examples using my PC via the Python console and the second two using the MicroPython board via the REPL console.

I explain the code in detail for each example and show example output when you execute the code as well as a challenge for you to try a modification or two of each example on your own. I encourage you to implement these examples and figure out the challenge yourself as practice for the projects later in this book.

Example 1: Using Loops

This example demonstrates how to write loops in Python using the `for` loop. The problem we are trying to solve is converting integers from decimal to binary, hexadecimal, and octal. Often with IOT projects, we need to see values in one or more of these formats and in some cases the sensors we use (and the associated documentation) uses hexadecimal rather than decimal. Thus, this example can be helpful in the future not only for how to use the `for` loop but also how to convert integers into different formats.

Write the Code

The example begins with a tuple of integers to convert. Tuples and lists can be iterated through (values read in order) using a for loop. Recall a tuple is read only so in this case since it is input, it is fine but in other cases where you may need to change values, you will want to use a list. Recall, the syntactical difference between a tuple and a list is the tuple uses parentheses and a list uses square brackets.

The for loop demonstrated here is called a “for each” loop. Notice I used the syntax “for value in values,” which tells Python to iterate over the tuple named values fetching (storing) each item into the value variable each iteration through the tuple.

Finally, I use the `print()` and `format()` functions to replace two placeholders `{0}` and `{1}`, to print out a different format of the integer using the methods `bin()` for binary, `oct()` for octal, and `hex()` for hexadecimal that do the conversion for us. Listing 4-5 shows examples of converting integers to different forms.

Listing 4-5. Converting Integers

```
#
# MicroPython for the IOT
#
# Example: Convert integer to binary, hex, and octal
#
# Dr. Charles Bell
#

# Create a tuple of integer values
values = (12, 450, 1, 89, 2017, 90125)

# Loop through the values and convert each to binary, hex, and octal
for value in values:
    print("{0} in binary is {1}".format(value, bin(value)))
    print("{0} in octal is {1}".format(value, oct(value)))
    print("{0} in hexadecimal is {1}".format(value, hex(value)))
```

Execute the Code

You can save this code in a file named `conversions.py` on your PC and then open a terminal (console window) and run the code with the command, `python ./conversions.py` (or `python3` if you have multiple versions of Python installed). Listing 4-6 shows the output.

Listing 4-6. Conversions Example Output

```
$ python3 ./conversions.py
12 in binary is 0b1100
12 in octal is 0o14
12 in hexadecimal is 0xc
```

```

450 in binary is 0b111000010
450 in octal is 0o702
450 in hexadecimal is 0x1c2
1 in binary is 0b1
1 in octal is 0o1
1 in hexadecimal is 0x1
89 in binary is 0b1011001
89 in octal is 0o131
89 in hexadecimal is 0x59
2017 in binary is 0b11111100001
2017 in octal is 0o3741
2017 in hexadecimal is 0x7e1
90125 in binary is 0b1011000000001101
90125 in octal is 0o260015
90125 in hexadecimal is 0x1600d

```

Notice all the values in the tuple where converted.

Your Challenge

To make this example better, instead of using a static tuple to contain hard-coded integers, rewrite the example to read the integer from arguments on the command line along with the format. For example, the code would be executed like the following.

```

$ python3 ./conversions.py 123 hex
123 in hexadecimal is 0x7b

```

To read arguments from the command line, use the argument parser, `argparse` (<https://docs.python.org/3/howto/argparse.html>). If you want to read the integer from the command line, you can use the `argparse` module to add an argument by name as follows.

```

import argparse

# Setup the argument parser
parser = argparse.ArgumentParser()

# We need two arguments: integer, and conversion
parser.add_argument("original_val")
parser.add_argument("conversion")

# Get the arguments
args = parser.parse_args()

```

When you use the argument parser (`argparse`) module, the values of the arguments are all strings, so you will need to convert the value to an integer before you use the `bin()`, `hex()`, or `oct()` method.

You will also need to determine which conversion is requested. I suggest use only hex, bin, and oct for the conversion and use a set of conditions to check the conversion requested. Something like the following would work.

```
if args.conversion == 'bin':
    # do conversion to binary
elif args.conversion == 'oct':
    # do conversion to octal
elif args.conversion == 'hex':
    # do conversion to hexadecimal
else:
    print("Sorry, I don't understand, {0}.".format(args.conversion))
```

Notice the last else communicates that the argument was not recognized. This helps to manage user error.

There is one more thing about the argument parser you should know. You can pass in a help string when adding arguments. The argument parser also gets you the help argument (-h) for free. Observe the following. Notice I added a couple of strings using the help= parameter.

```
# We need two arguments: integer, and conversion
parser.add_argument("original_val", help="Value to convert.")
parser.add_argument("conversion", help="Conversion options:
hex, bin, or oct.")
```

Now when we complete the code and run it with the -h option, we get the following output. Cool, eh?

```
$ python3 ./conversions.py -h
usage: conversions.py [-h] original_val conversion

positional arguments:
  original_val  Value to convert.
  conversion    Conversion options: hex, bin, or oct.

optional arguments:
  -h, --help    show this help message and exit
```

Example 2: Using Complex Data and Files

This example demonstrates how to work with the JavaScript Object Notation⁵ (JSON) in Python. In short, JSON is a markup language used to exchange data. Not only is it human readable, it can be used directly in your applications to store and retrieve data to and from other applications, servers, and even MySQL. In fact, JSON looks familiar to programmers

⁵<http://www.json.org/>

because it resembles other markup schemes. JSON is also very simple in that it supports only two types of structures: 1) a collection containing (name, value) pairs, and 2) an ordered list (or array). Of course, you can also mix and match the structures in an object. When we create a JSON object, we call it a JSON document.

The problem we are trying to solve is writing and reading data to/from files. In this case, we will use a special JSON encoder and decoder module named `json` that allows us to easily convert data in files (or other streams) to and from JSON. As you will see, accessing JSON data is easy by simply using the key (sometimes called fields) names to access the data. Thus, this example can be helpful in the future not only for how to use read and write files but also how to work with JSON documents.

Write the Code

This example stores and retrieves data in files. The data is basic information about pets including the name, age, breed, and type. The type is used to determine broad categories like fish, dog, or cat.

We begin by importing the JSON module (named `json`), which is built into the MicroPython platform. Next, we prepare some initial data by building JSON documents and storing them in a Python list. We use the `json.loads()` method to pass in a JSON formatted string. The result is a JSON document that we can add to our list. The examples use a very simple form of JSON documents – a collection of (name, value) pairs. The following shows an example of one of the JSON formatted strings used.

```
{"name": "Violet", "age": 6, "breed": "dachshund", "type": "dog"}
```

Notice we enclose the string inside curly braces and use a series of key names, a colon, and a value separated by commas. If this looks familiar, it's because is the same format as a Python dictionary. This demonstrates my comment that JSON syntax looks familiar to programmers.

The JSON method, `json.loads()`, takes the JSON formatted string then parses the string checking for validity and returns a JSON document. We then store that document in a variable and add it to the list as shown below.

```
parsed_json = json.loads('{"name": "Violet", "age": 6, "breed": "dachshund",
" type": "dog"}')
pets.append(parsed_json)
```

Once the data is added to the list, we then write the data to a file named `my_data.json`. To work with files, we first open the file with the `open()` function, which takes a file name (including a path if you want to put the file in a directory) and an access mode. We use “r” for read, and “w” for write. You can also use “a” for append if you want to open a file and add to the end. Note that the “w” access will overwrite the file when you write to it. If the `open()` function succeeds, you get a file object that permits you to call additional functions to read or write data. The `open()` will fail if the file is not present (and you have requested read access) or you do not have permissions to write to the file.

In case you're curious what other access modes, Table 4-3 shows the list of modes available for the `open()` function.

Table 4-3. *Python File Access Modes*

Mode	Description
<code>r</code>	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
<code>rb</code>	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
<code>r+</code>	Opens a file for both reading and writing. The file pointer is placed at the beginning of the file.
<code>rb+</code>	Opens a file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.
<code>w</code>	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, it creates a new file for writing.
<code>wb</code>	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, it creates a new file for writing.
<code>w+</code>	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
<code>wb+</code>	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
<code>a</code>	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
<code>ab</code>	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
<code>a+</code>	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
<code>ab+</code>	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Once the file is open, we can write the JSON documents to the file by iterating over the list. Iteration means to start at the first element and access the elements in the list one at a time in order (the order they appear in the list). Recall, iteration in Python is very easy. We simply say, “for each item in the list” with the for loop as follows.

```
for pet in pets:
    // do something with the pet data
```


To write the JSON document to the file, we use the `json.dumps()` method, which will produce a JSON formatted string writing that to the file using the file variable and the `write()` method. Thus, we now see how to build JSON documents from strings and then decode (dump) them to a string.

Once we've written data to the file, we then close the file with the `close()` function then reopen it and read data from the file. In this case, we use another special implementation of the for loop. We use the file variable to read all of the lines in the file with the `readlines()` method then iterate over them with the following code.

```
json_file = open("my_data.json", "r")
for pet in json_file.readlines():
    // do something with the pet string
```

We use the `json.loads()` method again to read the JSON formatted string as read from the file to convert it to a JSON document, which we add to another list. We then close the file. Now the data has been read back into our program and we can use it. Finally, we iterate over the new list and print out data from the JSON documents using the key names to retrieve the data we want. Listing 4-7 shows the completed code for this example.

Listing 4-7. Writing and Reading JSON Objects to/from Files

```
#
# MicroPython for the IOT
#
# Example: Storing and retrieving JSON objects in files
#
# Dr. Charles Bell
#

import json

# Prepare a list of JSON documents for pets by converting JSON to a dictionary
pets = []
parsed_json = json.loads('{"name": "Violet", "age": 6, "breed": "dachshund",
"type": "dog"}')
pets.append(parsed_json)
parsed_json = json.loads('{"name": "JonJon", "age": 15, "breed": "poodle",
"type": "dog"}')
pets.append(parsed_json)
parsed_json = json.loads('{"name": "Mister", "age": 4, "breed": "siberian
khatru", "type": "cat"}')
pets.append(parsed_json)
parsed_json = json.loads('{"name": "Spot", "age": 7, "breed": "koi",
"type": "fish"}')
pets.append(parsed_json)
```

```
parsed_json = json.loads('{ "name": "Charlie", "age": 6, "breed": "dachshund",
"type": "dog" }')
pets.append(parsed_json)
```

Now, write these entries to a file. Note: overwrites the file

```
json_file = open("my_data.json", "w")
for pet in pets:
    json_file.write(json.dumps(pet))
    json_file.write("\n")
json_file.close()
```

Now, let's read the JSON documents then print the name and age for all of the dogs in the list

```
my_pets = []
json_file = open("my_data.json", "r")
for pet in json_file.readlines():
    parsed_json = json.loads(pet)
    my_pets.append(parsed_json)
json_file.close()
```

```
print("Name, Age")
for pet in my_pets:
    if pet['type'] == 'dog':
        print("{0}, {1}".format(pet['name'], pet['age']))
```

Notice the loop for writing data. We added a second `write()` method passing in a strange string (it is actually an escaped character). The `\n` is a special character called the newline character. This forces the JSON formatted strings to be on separate lines in the file and helps with readability.

■ **Tip** For a more in-depth look at how to work with files in Python, see <https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files>.

So, what does the file look like? The following is a dump of the file using the `more` utility, which shows the contents of the file. Notice the file contains the JSON formatted strings just like we had in our code.

```
$ more my_data.json
{"age": 6, "breed": "dachshund", "type": "dog", "name": "Violet"}
{"age": 15, "breed": "poodle", "type": "dog", "name": "JonJon"}
{"age": 4, "breed": "siberian khatru", "type": "cat", "name": "Mister"}
{"age": 7, "breed": "koi", "type": "fish", "name": "Spot"}
{"age": 6, "breed": "dachshund", "type": "dog", "name": "Charlie"}
```

Now, let's see what happens when we run this script.

Execute the Code

You can save this code in a file named `rw_json.py` on your PC then open a terminal (console window) and run the code with the command, `python ./rw_json.py` (or `python3` if you have multiple versions of Python installed). The following shows the output.

```
$ python ./rw_json.py
Name, Age
Violet, 6
JonJon, 15
Charlie, 6
```

While the output may not be very impressive, by completing the example, you've learned a great deal about working with files and structured data using JSON documents.

Your Challenge

To make this example more of a challenge, you could modify it to include more information about your pets. I suggest you start with a simple text file and type in the JSON formatted strings for your pets. To increase the complexity, try adding information that is pertinent to the type of pet. For example, you could add some keys for one or more pets, other keys for other pets, and so on. Doing so will show one of the powers of JSON documents; collections of JSON documents do not have to have the same format.

Once you have this file, modify the code to read from the file and print out all the information for each pet by printing the key name and value. Hint: you will need to use special code to print out the key name and the value called "pretty printing." For example, the following code will print out the JSON document in an easily readable format. Notice we use the `sort_keys` option to print the keys (fields) and we can control the number of spaces to indent.

```
for pet in my_pets:
    print(json.dumps(pet, sort_keys=True, indent=4))
```

When run, the output will look like the following.

```
{
  "age": 6,
  "breed": "dachshund",
  "name": "Violet",
  "type": "dog"
}
{
  "age": 15,
  "breed": "poodle",
  "name": "JonJon",
  "type": "dog"
}
```

Example 3: Using Functions

This example demonstrates how to create and use functions. Recall functions are used to help make our code more modular. Functions can also be a key tool in avoiding duplication of code. That is, we can reuse portions of code repeatedly by placing them in a function. Functions are also used to help isolate code for special operations such as mathematical formulae.

The problem we're exploring in this example is how to create functions to perform calculations. We will also explore a common computer science technique called recursion⁶ where a function calls itself repeatedly. I will also show you the same function implemented in an iterative manner (typically using a loop). While some would advise avoiding recursion, recursive functions are a bit shorter to write but can be more difficult to debug if something goes wrong. The best advice I can offer is that almost every recursive function can be written as iterative functions and novice programmers should stick to iterative solutions until they gain confidence using functions.

Write the Code

This example is designed to calculate a Fibonacci series.⁷ A Fibonacci series is calculated as the sum of the two preceding values in the series. The series begins with 1 followed by 1 (nothing plus 1), then $1 + 1 = 2$, and so on. For this example, we will ask the user for an integer then calculate the number of values in the Fibonacci series. If the input is 5, the series is 1, 1, 2, 3, 5.

We will create two functions: one to calculate the Fibonacci series using code that iteratively calculates the series and one to calculate the nth Fibonacci number using a recursive function. Let's look at the iterative function first.

To define a function, we use the syntax `def func_name(<parameters>):` where we supply a function name and a list of zero or more parameters followed by a colon. These parameters are then usable inside the function. We pass in data to the function using the parameters. The following shows the iterative version of the Fibonacci series code. We name this function, `fibonacci_iterative`.

```
def fibonacci_iterative(count):
    i = 1
    if count == 0:
        fib = []
    elif count == 1:
        fib = [1]
    elif count == 2:
        fib = [1,1]
    elif count > 2:
        fib = [1,1]
```

⁶[https://en.wikipedia.org/wiki/Recursion_\(computer_science\)](https://en.wikipedia.org/wiki/Recursion_(computer_science))

⁷https://en.wikipedia.org/wiki/Fibonacci_number

```

    while i < (count - 1):
        fib.append(fib[i] + fib[i-1])
        i += 1
return fib

```

This code simply calculates the first N values in the series and returns them in a list. The parameter `count` is the number of values in the series. The function begins by checking to see if the trivial values are requested: 0, 1, or 2 whose values are known. If the `count` value is greater than 2, we begin with the known series [1, 1] then use a loop to calculate the next value by adding the two previous values together. Take a moment to notice how I use the list index to get the two previous values in the list (`i` and `i-1`). We will use this function and the list returned directly in our code to find a specific value in the series and print it.

Now let's look at the recursive version of the function. The following shows the code. We name this function `fibonacci_recursive`.

```

def fibonacci_recursive(number):
    if number == 0:
        return 0
    elif number == 1:
        return 1
    else:
        # Call our self counting down.
        value = fibonacci_recursive(number-1) + fibonacci_
            recursive(number-2)
        return value

```

In this case, we don't return the entire series; rather, we return the specific value in the series – the *n*th value. Like the iterative example, we do the same thing regarding the trivial values returning the number requested. Otherwise, we call the same function again for each number. It may take some time to get your mind around how this works, but it does calculate the *n*th value.

Now, you may be wondering where you place functions in the code. We need to place them at the top of the code. Python will parse the functions and continue to execute statements following the definitions. Thus, we place our “main” code after our functions.

The main code for this example begins with requesting the *n*th value for the Fibonacci series then uses the recursive function first to calculate the value. We then ask the user if they want to see the entire series and if so, we use the iterative version of the function to get the list and print it out. We print out the *n*th value and give the option again to see the entire series to show the result is the same using both functions. Listing 4-8 shows the completed code for the example. We will name this code `fibonacci.py`.

Listing 4-8. Calculating Fibonacci Series

```

#
# MicroPython for the IOT
#
# Example: Fibonacci series using recursion
#
# Calculate the Fibonacci series based on user input
#
# Dr. Charles Bell
#

# Create a function to calculate Fibonacci series (iterative)
# Returns a list.
def fibonacci_iterative(count):
    i = 1
    if count == 0:
        fib = []
    elif count == 1:
        fib = [1]
    elif count == 2:
        fib = [1,1]
    elif count > 2:
        fib = [1,1]
        while i < (count - 1):
            fib.append(fib[i] + fib[i-1])
            i += 1
    return fib

# Create a function to calculate the nth Fibonacci number (recursive)
# Returns an integer.
def fibonacci_recursive(number):
    if number == 0:
        return 0
    elif number == 1:
        return 1
    else:
        # Call our self counting down.
        value = fibonacci_recursive(number-1) + fibonacci_recursive(number-2)
        return value

# Main code
print("Welcome to my Fibonacci calculator!")
index = int(input("Please enter the number of integers in the series: "))

```

```

# Recursive example
print("We calculate the value using a recursive algorithm.")
nth_fibonacci = fibonacci_recursive(index)
print("The {0}{1} fibonacci number is {2}."
      "".format(index, "th" if index > 1 else "st", nth_fibonacci))
see_series = str(input("Do you want to see all of the values in the series? "))
if see_series in ["Y", "y"]:
    series = []
    for i in range(1, index+1):
        series.append(fibonacci_recursive(i))
    print("Series: {0}: ".format(series))

# Iterative example
print("We calculate the value using an iterative algorithm.")
series = fibonacci_iterative(index)
print("The {0}{1} fibonacci number is {2}."
      "".format(index, "th" if index > 1 else "st", series[index-1]))
see_series = str(input("Do you want to see all of the values in the series? "))
if see_series in ["Y", "y"]:
    print("Series: {0}: ".format(series))

print("bye!")

```

Take a few moments to read through the code. While the problem being solved is a bit simpler than the previous example, there is a lot more code to read through. When you're ready, connect your MicroPython board and create the file. You create the file on your PC for this example and name it `fibonacci.py`. We'll copy it to our MicroPython board in the next section.

■ **Tip** For a more in-depth look at how to create and use your own functions (methods) in Python, see <https://docs.python.org/3/tutorial/controlflow.html#defining-functions>.

Now, let's see what happens when we run this script. Recall, we will be running this code on our MicroPython board so if you're following along, be sure to set up your board and connect it to your PC.

Execute the Code

Recall from Chapter 3, when we want to move code to our MicroPython board, we need to create the file and then copy it to the MicroPython board and then execute it. I will show you how to do this on macOS in this section. An example of how to do it on Windows 10 is shown in example 4.

When you plug your MicroPython board into a USB port on your PC, the flash drive will appear in your Finder and on your desktop. Figure 4-1 shows an example on macOS.

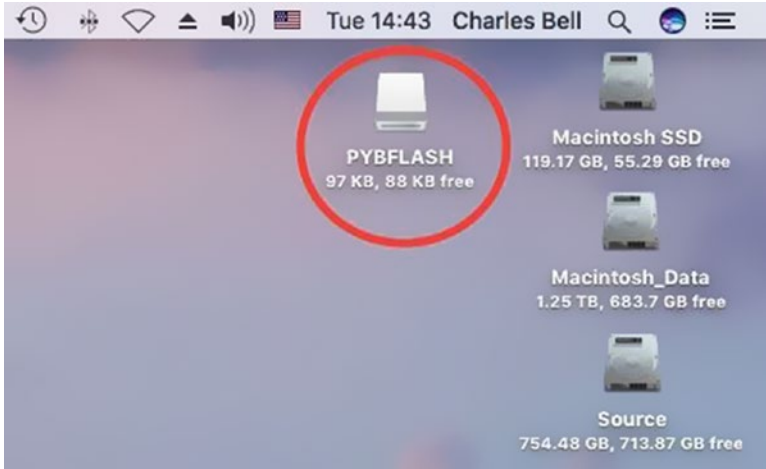


Figure 4-1. MicroPython board Flash Drive Mounted

We now need to copy the `fibonacci.py` file to the flash drive. On macOS, open the flash drive and then drag the files onto the drive. You should see something like Figure 4-2 when you've copied the file.

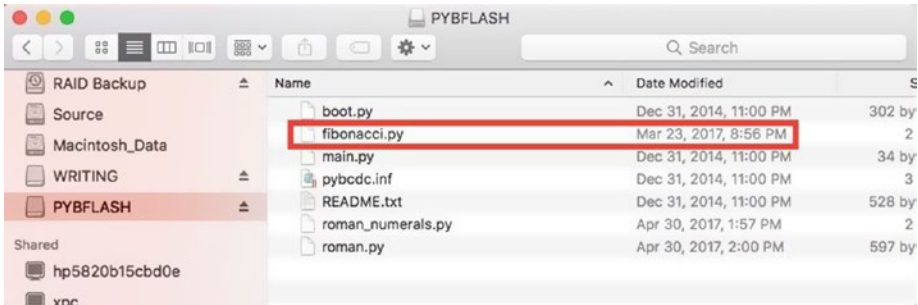


Figure 4-2. Files Copied to the MicroPython board Flash Drive

Notice I annotated the image to show the file. Notice also we see the MicroPython files `boot.py` and `main.py`. We will not modify the `main.py` file to run our example because we don't want the program to run every time we boot the device. Instead, we will run the code from the REPL console.

Open a REPL console using a terminal window. You should see the welcome message from MicroPython followed by the RELP prompt (`>>>`). From the REPL prompt, issue the following code statement.

```
import fibonnaci
```


This code imports the `fibonacci.py` file we just copied and when it does, it executes the code. So, it's like we ran it on our PC from the Python console. Go ahead and test the program by requesting the twelfth value in the Fibonacci series. You should see output shown in Figure 4-3.

```

MicroPython v1.8.2 on 2016-07-13; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> import fibonacci
Welcome to my Fibonacci calculator!
Please enter the number of integers in the series: 12
We calculate the value using a recursive algorithm.
The 12th fibonacci number is 144.
Do you want to see all of the values in the series? y
Series: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]:
We calculate the value using an iterative algorithm.
The 12th fibonacci number is 144.
Do you want to see all of the values in the series? y
Series: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]:
bye!
>>>

```

Figure 4-3. Output of Fibonacci Example (MicroPython board)

Notice I responded to the query to show the entire series. Notice also that we see the code has exercised both versions of the function we wrote: iterative and recursive. Finally, we see that the values or output of both functions is the same data.

■ **Note** If you do not yet have a MicroPython board, you can run the code from your PC with the command, `python ./fibonacci.py`. You will see similar output as shown in the figure.

If you run the code again, just press CTRL-D to do a soft reset then issue the `import` statement again. This will rerun the entire code. Or, if you want to run one of the functions, you can call it again by importing it using the code below. Figure 4-4 shows how this would look when executed on the MicroPython board.

```

from fibonacci import fibonacci_iterative
fibonacci_iterative(6)
from fibonacci import fibonacci_recursive
fibonacci_recursive(6)

```

```

cbell — screen /dev/tty.usbmodemFD14122 115200 • SCREEN — 80x24
MicroPython v1.8.2 on 2016-07-13; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> import fibonacci
Welcome to my Fibonacci calculator!
Please enter the number of integers in the series: 12
We calculate the value using a recursive algorithm.
The 12th fibonacci number is 144.
Do you want to see all of the values in the series? y
Series: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]:
We calculate the value using an iterative algorithm.
The 12th fibonacci number is 144.
Do you want to see all of the values in the series? y
Series: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]:
bye!
>>> from fibonacci import fibonacci_iterative
>>> fibonacci_iterative(6)
[1, 1, 2, 3, 5, 8]
>>> from fibonacci import fibonacci_recursive
>>> fibonacci_recursive(6)
8
>>>

```

Figure 4-4. Experimenting with the Fibonacci Functions

Once imported this way, you can run the functions again. Go ahead and try them with different values to show how they behave. Recall, the functions were implemented differently – the iterative version returns the list and the recursive version returns only the last or nth value.

When you’re done experimenting with the example, remember to close the terminal and eject the drive for the flash before you unplug it.

Your Challenge

To make this example a bit more useful, modify the code to search a Fibonacci series for a specific integer. Ask the user to provide an integer then determine if the value is a valid Fibonacci value. For example, if the user enters 144, the code should tell the user that value is valid and is the twelfth value in the series. While this challenge will require you to rewrite most of the code for the “main” functionality, you must figure out how to use the functions in a new way.

Example 4: Using Classes

This example ramps up the complexity considerably by introducing an object-oriented programming concept: classes. Recall from earlier that classes are another way to modularize our code. Classes are used to model data and behavior on that data. Further, classes are typically placed in their own code module (file) that further modularizes the code. If you need to modify a class, you may need only change the code in the class module.

The problem we’re exploring in this example is how to develop solutions using classes and code modules. We will be creating two files: one for the class and another for the main code.

Write the Code

This example is designed to convert Roman numerals⁸ to integers. That is, we will enter a value like VIII, which is eight, and expect to see the integer 8. To make things more interesting, we will also take the integer we derive and convert it back to Roman numerals. Roman numerals are formed as a string using the characters I for 1, V for 5, X for 10, L for 50, C for 100, D for 500, and M for 1000. Combinations of other numbers are done by adding the character numerical value together (e.g., 3 = III), or a single, lower character before another character to indicate the representative minus that character (e.g., 4 = IV). The following show some examples of how this works.

```
3 = III
15 = XV
12 = XII
24 = XXIV
96 = LXLVI
107 = CVII
```

This may sound like a lot of extra work, but consider this: if we can convert from one format to another, we should be able to convert back without errors. More specifically, we can use the code for one conversion to validate the other. If we get a different value when converting it back, we know we have a problem that needs to be fixed.

To solve the problem, we will place the code for converting Roman numerals into a separate file (code module) and build a class called `Roman_Numerals` to contain the methods. In this case, the data is a mapping of integers to Roman numerals.

```
# Private dictionary of roman numerals
__roman_dict = {
    'I': 1,
    'IV': 4,
    'V': 5,
    'IX': 9,
    'X': 10,
    'XL': 40,
    'L': 50,
    'XC': 90,
    'C': 100,
    'CD': 400,
    'D': 500,
    'CM': 900,
    'M': 1000,
}
```

⁸https://en.wikipedia.org/wiki/Roman_numerals

Notice the two underscores before the name of the dictionary. This is a special notation that marks the dictionary as a private variable in the class. This is a Python aspect for information hiding, which is a recommended technique to use when designing objects; always strive to hide data that is used inside the class.

Notice also that instead of using the basic characters and their values, I used several other values too. I did this to help make the conversion easier (and cheat a bit). In this case, I added the entries that represent the one value previous conversions such as 4 (IV), 9 (IX), etc. This makes the conversion a bit easier (and more accurate).

We will also add two methods; `convert_to_int()`, which takes a Roman numeral string and converts it to an integer; and `convert_to_roman()`, which takes an integer and converts it to a Roman numeral. Rather than explain every line of code in the methods, I leave it to you to read the code to see how it works.

Simply, the `convert to integer` method takes each character and gets its value from the dictionary summing the values. There is a trick there that requires special handling for the lower value characters appearing before higher values (e.g., IX). The `convert to Roman` method is a bit easier since we simply divide the value by the highest value in the dictionary until we reach zero. Listing 4-9 shows the code for the class module, which is saved in a file named `roman_numerals.py`.

Listing 4-9. Roman Numeral Class

```
#
# MicroPython for the IOT
#
# Example: Roman numerals class
#
# Convert integers to roman numerals
# Convert roman numerals to integers
#
# Dr. Charles Bell
#

class Roman_Numerals:

    # Private dictionary of roman numerals
    __roman_dict = {
        'I': 1,
        'IV': 4,
        'V': 5,
        'IX': 9,
        'X': 10,
        'XL': 40,
        'L': 50,
        'XC': 90,
        'C': 100,
        'CD': 400,
```

```

'D': 500,
'CM': 900,
'M': 1000,
}

def convert_to_int(self, roman_num):
    value = 0
    for i in range(len(roman_num)):
        if i > 0 and self.__roman_dict[roman_num[i]] > self.__roman_
            dict[roman_num[i - 1]]:
            value += self.__roman_dict[roman_num[i]] - 2 * self.__roman_
                dict[roman_num[i - 1]]
        else:
            value += self.__roman_dict[roman_num[i]]
    return value

def convert_to_roman(self, int_value):
    # First, get the values of all of entries in the dictionary
    roman_values = list(self.__roman_dict.values())
    roman_keys = list(self.__roman_dict.keys())
    # Prepare the string
    roman_str = ""
    remainder = int_value
    # Loop through the values in reverse
    for i in range(len(roman_values)-1, -1, -1):
        count = int(remainder / roman_values[i])
        if count > 0:
            for j in range(0, count):
                roman_str += roman_keys[i]
            remainder -= count * roman_values[i]
    return roman_str

```

If you're following along with the chapter, go ahead and create a file on your PC for this code and name it `roman_numerals.py`. We'll copy it to our MicroPython board in the next section.

Now let's look at the main code. For this, we simply need to import the new class from the code module as follows. This is a slightly different form of the import directive. In this case, we're telling Python to include the `roman_numerals` class from the file named `Roman_Numerals`.

```
from roman_numerals import Roman_Numerals
```

■ **Note** If the code module were in a subfolder, say `roman`, we would have written the import statement as `from roman import Roman_Numerals` where we list the folders using dot notation instead of slashes.

The rest of the code is straightforward. We first ask the user for a valid Roman numeral string then convert it to an integer and use that value to convert back to Roman numerals, printing the result. So, you see having the class in a separate module has simplified our code making it shorter and easier to maintain. Listing 4-10 shows the complete main code saved in a file named simply `roman.py`.

Listing 4-10. Converting Roman Numerals

```
#
# MicroPython for the IOT
#
# Example: Convert roman numerals using a class
#
# Convert integers to roman numerals
# Convert roman numerals to integers
#
# Dr. Charles Bell
#

from roman_numerals import Roman_Numerals

roman_str = input("Enter a valid roman numeral: ")
roman_num = Roman_Numerals()

# Convert to roman numberals
value = roman_num.convert_to_int(roman_str)
print("Convert to integer:      {0} = {1}".format(roman_str, value))

# Convert to integer
new_str = roman_num.convert_to_roman(value)
print("Convert to Roman Numerals: {0} = {1}".format(value, new_str))

print("bye!")
```

If you're following along with the chapter, go ahead and create a file on your PC for this code and name it `roman.py`. We'll copy it to our MicroPython board in the next section.

■ **Tip** For a more in-depth look at how to work with classes in Python, see <https://docs.python.org/3/tutorial/classes.html>.

Now, let's see what happens when we run this script. Recall, we will be running this code on our MicroPython board so if you're following along, be sure to set up your board and connect it to your PC.

Execute the Code

In this example, we will see how to copy the code to our MicroPython board on Windows 10. Like the last example, we will first create the files on our PC, connect the MicroPython board, copy the files, and then execute them.

When you connect your MicroPython board on Windows 10, the drive will show in the file explorer as shown in Figure 4-5.

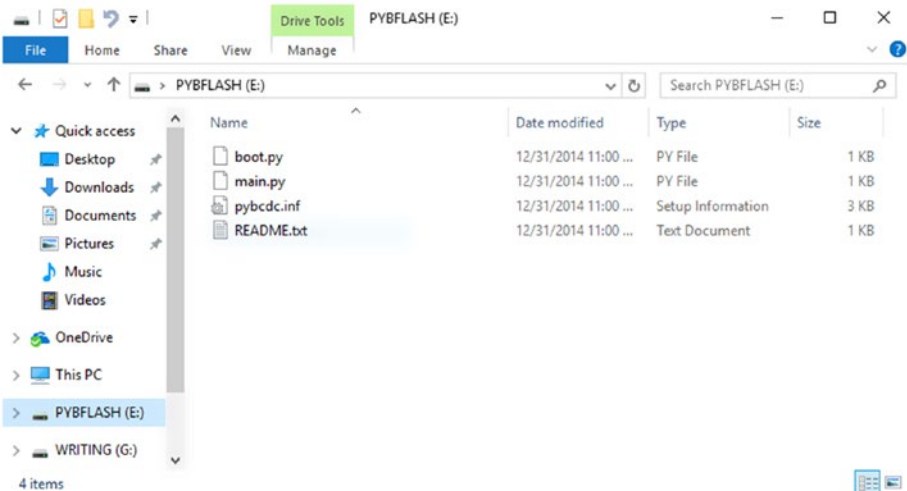


Figure 4-5. MicroPython board Flash Drive Mounted (Windows 10)

If you haven't created the files, do that now. When ready, we'll copy them to the MicroPython board. Rather than use the file explorer, I chose to demonstrate how to copy the files from a console (command prompt). Open a console and then change to the drive letter shown in your file explorer. Next, copy the files `roman_numerals.py` and `roman.py` to the drive. Figure 4-6 shows an example of copying the files.

```

Command Prompt
E:\>dir
Volume in drive E is PYBFLASH
Volume Serial Number is 4621-0000

Directory of E:\

01/01/2015  12:00 AM                34 main.py
01/01/2015  12:00 AM            2,721 pybcd.c.inf
01/01/2015  12:00 AM             528 README.txt
01/01/2015  12:00 AM             302 boot.py
           4 File(s)            3,585 bytes
           0 Dir(s)           90,624 bytes free

E:\>copy c:\Users\Chuck\Documents\source\Ch04\roman*.py e:
c:\Users\Chuck\Documents\source\Ch04\roman.py
c:\Users\Chuck\Documents\source\Ch04\roman_numerals.py
        2 file(s) copied.

E:\>dir
Volume in drive E is PYBFLASH
Volume Serial Number is 4621-0000

Directory of E:\

01/01/2015  12:00 AM                34 main.py
01/01/2015  12:00 AM            2,721 pybcd.c.inf
01/01/2015  12:00 AM             528 README.txt
01/01/2015  12:00 AM             302 boot.py
04/30/2017  02:00 PM             597 roman.py
04/30/2017  01:57 PM            1,542 roman_numerals.py
           6 File(s)            5,724 bytes
           0 Dir(s)           87,552 bytes free

E:\>

```

Figure 4-6. Copying the File to the MicroPython board Flash Drive (Console)

Now, we're ready to connect to the MicroPython board and run the code. Recall one of the ways to connect to the board is to use PuTTY on Windows. Figure 4-7 shows the PuTTY console. Notice I chose COM3 and serial connection. Recall, you can find the COM port from the Device Manager. When ready, click *Open*.

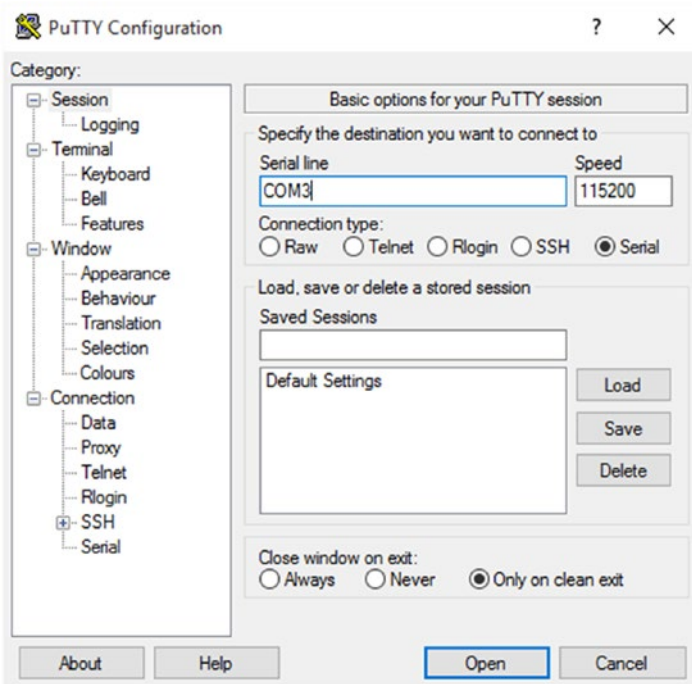


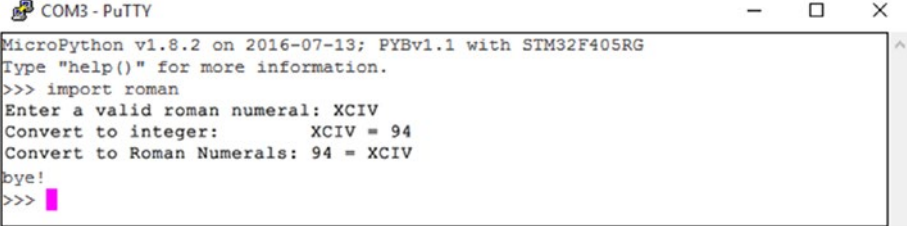
Figure 4-7. Connecting with PuTTY (Windows 10)

PuTTY will open the REPL console when the connection is made. You should see the welcome message from MicroPython followed by the REPL prompt (`>>>`). From the REPL prompt, issue the following code statement.

```
import roman
```

This code imports the `roman.py` file we just copied and when it does, it executes the code, which if you recall will call the code in the `roman_numerals.py` code module. Once again, issuing the import is like how we ran it on our PC from the Python console.

Once you issue the import statement, the code will execute. Go ahead and try it out. Start with the value `LXLIV`, which is `SSS` in Roman numerals. Figure 4-8 shows what the output will look like.



```

COM3 - PuTTY
MicroPython v1.8.2 on 2016-07-13; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> import roman
Enter a valid roman numeral: XCIV
Convert to integer:          XCIV = 94
Convert to Roman Numerals: 94 = XCIV
bye!
>>>

```

Figure 4-8. Executing the Roman Numerals Example

You can also experiment with importing the code and executing it again from the terminal or just eject the drive, reset the board, and reconnect to run it again with other values.

When you're done executing the example, remember to close the terminal and eject the drive for the flash before you unplug it.

Your Challenge

There isn't much to add for this example to improve it other than perhaps some user friendliness (nicer to use). If you want to improve the code or the class itself, I suggest adding a new method named `validate()` used to validate a Roman numeral string. This method can take a string and determine if it contains a valid series of characters. Hint: to start, check the string has only the characters in the dictionary.

However, you can use this template to build other classes for converting formats. For example, as an exercise, you could create a new class to convert integers to hexadecimal or even octal. Yes, there are functions that will do this for us, but it can be enlightening and satisfying to build it yourself. Go ahead, give it a go – create a new class to convert integers to other formats. I would suggest doing a hexadecimal to integer function first and when that is working correctly, create the reciprocal to convert integers to hexadecimal.

A more advanced challenge would be to rewrite the class to accept a string in the constructor (when the class variable is created) and use that string to do the conversions instead of passing the string or integer using the `convert_to*` methods. For example, the class could have a constructor and private member as follows.

```

__roman_str = ""
...
def __init__(self, name):
    self.name = name

```

When you create the instance, you will need to pass the string or else you will get an error that a required parameter is missing.

```

roman_str = input("Enter a valid roman numeral: ")
roman_num = Roman_Numerals(roman_str)

```

For More Information

Should you require more in-depth knowledge of Python, there are several excellent books on the topic. I list a few of my favorites below. A great resource is the documentation on the Python site: python.org/doc/.

- *Pro Python*, 2nd Edition (Apress, 2014) J. Burton Browning, Marty Alchin
- *Learning Python*, 5th Edition (O'Reilly Media, 2013) Mark Lutz
- *Automate the Boring Stuff with Python: Practical Programming for Total Beginners* (No Starch Press, 2015), Al Sweigart

Summary

Wow! That was a wild ride, wasn't it? I hope that this short crash course in Python has explained enough about the sample programs shown so far that you now know how they work. This crash course also forms the basis for understanding the other Python examples in this book.

If you are learning how to work with IOT projects and don't know how to program with Python, learning Python can be fun given its easy-to-understand syntax. While there are many examples on the Internet you can use, very few are documented in such a way as to provide enough information for someone new to Python to understand or much less get started and deploy the sample! But at least the code is easy to read.

This chapter has provided a crash course in Python that covers the basics of the things you will encounter when examining most of the smaller example projects. We discovered the basic syntax and constructs of a Python application including a walkthrough of building a real Python application that blinks an LED. Through that example, we learned how to work with headless applications including how to manage a startup background application.

In the next chapter, we'll dive deeper into MicroPython programming. We will see more about the special libraries available for use in your IOT projects written for running on MicroPython boards.

CHAPTER 5



MicroPython Libraries

Now that we have a good grasp on how to write our code in Python and MicroPython, it is time to look at the supporting libraries that make up the firmware. As you will see, the libraries available in MicroPython mirror those in Python. In fact, the libraries in the firmware (sometimes called the application programming interface or API or firmware API) comprise a great deal of the same libraries in Python.

There are some notable exceptions for standard libraries where there is an equivalent library in MicroPython, but it has been renamed to distinguish it from the Python library. In this case, the library has either been reduced in scope by removing the less-frequently used features or modified in some ways to fit the MicroPython platform - all to save space (memory).

There are also libraries that are specific to MicroPython, and the hardware that provides functionality that may or may not be in some general Python releases. These libraries are designed to make working with the microcontroller and hardware easier.

Thus, there are three types of libraries in the firmware: those that are standard and mostly the same as those in Python, those that are specific to MicroPython, and those that are specific to the hardware. There is another type of library sometimes called user-supplied or simply custom libraries. These are libraries (APIs) we create ourselves that we can deploy to our board and thereby make functionality available to all our projects. We will see an overview of all types of libraries in this chapter.

■ **Note** The word `module`, or `code module`, is sometimes used to refer to a library; however, `module` is commonly a single code file while a library may consist of many code files. Thus, it is ok to interchange the names if the library is a single file (`code module`). Another word sometimes used for library is `package`, which implies multiple files. You may encounter that term too.

Rather than simply paraphrase or (gasp) copy the existing documentation, we will see overviews of the libraries in the form of quick reference tables you can use to become familiar with what is available. Simply knowing what is available can often speed up development and make your programs stronger by not having to reinvent something or spend lots of time trying to find a library that you may (or may not) need. However, there are several libraries that are crucial to learning how to develop MicroPython IOT projects.

We will discover those in more detail with code snippets to help you learn them. So, while this chapter is intended to teach you about the more common libraries, it is also a reference guide to a deeper understanding of the MicroPython firmware with links to the official documentation.

Let's begin with a look at those libraries in MicroPython that are "standard" Python libraries.

Built-In and Standard Libraries

As you know, MicroPython is built on Python. Much work was done to trim things down so that most the Python libraries can fit on a chip. The resulting MicroPython firmware is considerably smaller than Python on your PC but in no way crippled. That is, MicroPython is Python, and as such MicroPython has many of the same libraries as Python.

Some may call these libraries "built-in," but it is more correct to name them "standard" libraries since these libraries are the same as those in Python. More specifically, they have the same classes with the same functions as those in Python. So, you can write a script on your PC and execute it there and then execute the same script unaltered on your MicroPython board. Nice! As you can surmise, this helps greatly when developing a complex project.

Recall we saw this technique demonstrated in the last chapter. There we saw it is possible to develop part of your script - those parts using standard libraries - and debug them on your PC. That is, once you have them working correctly, you can move on to the next part that requires MicroPython libraries or the hardware libraries. This is because developing on our PC is so much easier. We don't need to power the board on, connect it to our WiFi, etc., to get it to work. Plus, the PC is a lot faster. It's just easier all around. By exercising this practice, we can potentially save ourselves some frustration by ensuring the "standard" parts of our project are working correctly.

In this section, we will explore the standard Python libraries beginning with a short overview of what is available, followed by details on how to use some of the more common libraries.

■ **Tip** See <http://docs.micropython.org/en/latest/pyboard/library/index.html#python-standard-libraries-and-micro-libraries> for more details about any of the standard libraries.

Overview

The standard libraries in MicroPython contain objects that you can use to perform mathematical functions, operate on programming structures, work with transportable documents (a document store) through JSON, interact with the operating system and other system functions, and even perform calculations on time. Table 5-1 contains a list of the current standard MicroPython libraries. The first column is the name we use in our `import` statement, the second is a short description, and the third contains a link to the online documentation.

Table 5-1. *Standard Python Libraries in MicroPython*

Name	Description	Documentation
array	Working with arrays	http://docs.micropython.org/en/latest/pyboard/library/array.html
cmath	Complex mathematics	http://docs.micropython.org/en/latest/pyboard/library/cmath.html
gc	Garbage collector	http://docs.micropython.org/en/latest/pyboard/library/gc.html
math	Math functions	http://docs.micropython.org/en/latest/pyboard/library/math.html
sys	System-level functions	http://docs.micropython.org/en/latest/pyboard/library/sys.html
ubinascii	Conversions binary/ASCII	http://docs.micropython.org/en/latest/pyboard/library/ubinascii.html
ucollections	Containers	http://docs.micropython.org/en/latest/pyboard/library/ucollections.html
uerrno	Error codes	http://docs.micropython.org/en/latest/pyboard/library/uerrno.html
uhashlib	Hash algorithms	http://docs.micropython.org/en/latest/pyboard/library/uhashlib.html
uheapq	Heap queue	http://docs.micropython.org/en/latest/pyboard/library/uheapq.html
uio	Streams for input/output	http://docs.micropython.org/en/latest/pyboard/library/uio.html
ujson	JSON documents	http://docs.micropython.org/en/latest/pyboard/library/ujson.html
uos	Operating system	http://docs.micropython.org/en/latest/pyboard/library/uos.html
ure	Regular expressions	http://docs.micropython.org/en/latest/pyboard/library/ure.html
uselect	Stream events	http://docs.micropython.org/en/latest/pyboard/library/uselect.html
usocket	Sockets (networking)	http://docs.micropython.org/en/latest/pyboard/library/usocket.html
ustruct	Pack and unpack structs	http://docs.micropython.org/en/latest/pyboard/library/ustruct.html
utime	Time and date functions	http://docs.micropython.org/en/latest/pyboard/library/utime.html
uzlib	Compression algorithms	http://docs.micropython.org/en/latest/pyboard/library/uzlib.html

Notice the links are for the Pyboard documentation and most of these apply to the other boards too. If in doubt, always visit the MicroPython documentation for your board.

■ **Tip** Most of the libraries presented here are common to the Pyboard and WiPy. We will also see some of the differences in the next chapter.

As you can see, there are many libraries that begin with a `u` to signify they are special versions of the Python equivalent libraries. What is interesting is some platforms may include the original Python library. That is, if you need access to the original Python version - if it exists - you can still access it by using the original name (without the `u` prefix). In this case, MicroPython will attempt to find the module by the original name and if not there, default to the MicroPython version. For example, if we wanted to use the original `io` library, we could use `import io`. However, if there is no module named `io` on the platform, MicroPython will use the MicroPython version named `uio`.

■ **Caution** The format of the `import` statement allows us to specify directories. So, if we use `import mylibs.io`, MicroPython will attempt to find the library (code module) named `io.py` in the `mylibs` folder. This can affect how one uses modules without the `u` prefix. If we used `import io` and `io.py` is not a code module, it will use `io` as the name of a folder and if it exists, look modules in that folder. Thus, you can get yourself in trouble if you use folder names that are the same as the Python library names. Don't do that.

Next, we will look at some of the more commonly used standard libraries and see some code examples for each. But first, there are two categories of standard functions we should discuss.

INTERACTIVE HELP FOR LIBRARIES

A little-known function named `help()` can be, well, very helpful when learning about the libraries in MicroPython. You can use this function in a REPL session to get information about a library. The following shows an excerpt of the output for the `uos` library.

```
>>> help(uos)
<module 'uos'>object is of type module
  __name__ -- uos
  uname -- <function>
  chdir -- <function>
  getcwd -- <function>
  listdir -- <function>
  mkdir -- <function>
```

Notice we see the names of all the functions and, if present, constants. This can be a real help when learning the libraries and what they contain. Try it!

Common Standard Libraries

Now let's look at examples of some of the more commonly used standard libraries. What follows is just a sampling of what you can do with each of the libraries. See the online documentation for a full description of all the capabilities. Like most MicroPython libraries, the common standard libraries may be subsets of the full Python3 (CPython) implementation.

■ **Note** The examples in this chapter are intended to be run on the MicroPython board. You may need to make some changes to run it on your PC.

sys

The `sys` library provides access to the execution system such as constants, variables, command-line options, streams (`stdout`, `stdin`, `stderr`), and more. Most of the features of the library are constants or lists. The streams can be accessed directly but typically we use the `print()` function, which sends data to the `stdout` stream by default.

The most commonly used functions in this library include the following. Listing 5-1 demonstrates these variables and the `exit()` function.

- `sys.argv`: list of arguments passed to the script from the command line
- `sys.exit(r)`: exit the program returning the value `r` to the caller
- `sys.modules`: list of modules loaded (imported)
- `sys.path`: list of paths to search for modules – can be modified
- `sys.platform`: display the platform information such as Linux, MicroPython, etc.
- `sys.stderr`: standard error stream
- `sys.stdin`: standard input stream
- `sys.stdout`: standard output stream
- `sys.version`: the version of Python currently executing

Listing 5-1. Demonstration of the `sys` library features

```
# MicroPython for the IOT - Chapter 5
# Example use of the sys library
import sys
print("Modules loaded: " , sys.modules)
sys.path.append("/sd")
print("Path: " , sys.path)
sys.stdout.write("Platform: ")
```



```

sys.stdout.write(sys.platform)
sys.stdout.write("\n")
sys.stdout.write("Version: ")
sys.stdout.write(sys.version)
sys.stdout.write("\n")
sys.exit(1)

```

Notice we start with the `import` statement and after that, we can print the constants and variables in the `sys` library using the `print()` function. We also see how to append a path to our search path with the `sys.path.append()` function. This is very helpful if we create our own directories on the boot drive (SD drive) to place our code. Without this addition, the `import` statement will fail unless the code module is in the `lib` directory.

At the end of the example, we see how to use the `stdout` stream to write things to the screen. Note that you must provide the carriage return (newline) command to advance the output to a new line (`\n`). The `print()` function takes care of that for us. The following shows the output of running this script on a MicroPython board.

```

>>> import examples.sys_example
Modules loaded: {'examples': <module 'examples'>, 'examples.sys_example':
<module 'examples.sys_example' from 'examples/sys_example.py'>}
Path: ['', '/flash', '/flash/lib', '/sd']
Platform: WiPy
Version: 3.4.0

```

Notice we see the data as we expect and that this example is running on a `WiPy` module. Do you see something strange in the `import` statement? Here, the code is placed in a new directory named `examples` and then copied the code module to that directory on my boot drive. Thus, we can use the directory name in the statement to find the code module and import it (`examples/sys_examples.py`). If you want to run this code on your own MicroPython board, you can write the code to a file such as `sys_example.py`, then copy the file to your boot drive or use `ftp` to copy the file. This allows you to keep your boot (flash) drive tidy and place your code in a common location.

Finally, notice that there are no command-line arguments. This is because we used an `import` statement. However, if we were to run the code on our PC providing command-line arguments, we would see them. The following shows the output of running this script on a PC.

```

$ python ./sys_example.py
Modules loaded: {'builtins': <module 'builtins' (built-in)>, 'sys': <module
'sys' (built-in)>, ... '/sd']
darwin
Version: 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]

```

uio

The `uio` library contains additional functions to work with streams and stream-like objects. There is a single function named `uio.open()` that you can use to open files (but most people use the built-in function named `open()`) as well as classes for byte and string streams. In fact, the classes have similar file functions such as `read()`, `write()`, `seek()`, `flush()`, `close()`, as well as a `getvalue()` function, which returns the contents of the stream buffer that contains the data. Let's look at an example.

In this example, we first open a new file for writing and write an array of bytes to the file. The technique used is passing the hex values for each byte to the `write()` function. When you read data from sensors, they are typically in binary for (a byte or string of bytes). You signify a byte with the escape, `\x` as shown.

After writing the data to the file, we then read the file one byte at a time by passing 1 to the `read()` function. We then print the values read in their raw for (the value returned from the `read(1)` call), decimal value, and hex value. Listing 5-2 shows how to read a file in binary mode using the `uio.FileIO()` class. The bytes written contain a secret word (one obscured by using hex values) - can you see it?

Listing 5-2. Demonstration of the `uio` library features

```
# MicroPython for the IOT - Chapter 5
# Example use of the uio library
# Note: change uio to io to run this on your PC!
import uio
# Create the binary file
fio_out = uio.FileIO('data.bin', 'wb')
fio_out.write("\x5F\x9E\xAE\x09\x3E\x96\x68\x65\x6C\x6C\x6F")
fio_out.write("\x00")
fio_out.close()
# Read the binary file and print out the results in hex and char.
fio_in = uio.FileIO('data.bin', 'rb')
print("Raw,Dec,Hex from file:")
byte_val = fio_in.read(1) # read a byte
while byte_val:
    print(byte_val, ", ", ord(byte_val), hex(ord(byte_val)))
    byte_val = fio_in.read(1) # read a byte
fio_in.close()
```

This is like how you would use the normal built-in functions, which we saw in the last chapter. Listing 5-3 shows the output when run on the WiPy.

Listing 5-3. Demonstration of the `uio` library features (Pyboard)

```
>>> import examples.uio_example
Raw,Dec,Hex from file:
b'_' , 95 0x5f
b'\xc2' , 194 0xc2
b'\x9e' , 158 0x9e
b'\xc2' , 194 0xc2
b'\xae' , 174 0xae
b'\t' , 9 0x9
b'>' , 62 0x3e
b'\xc2' , 194 0xc2
b'\x96' , 150 0x96
b'h' , 104 0x68
b'e' , 101 0x65
b'l' , 108 0x6c
b'l' , 108 0x6c
b'o' , 111 0x6f
b'\x00' , 0 0x0
```

If you're curious what the file looks like, you can use a utility like `hexdump` to print the contents as shown below. Can you see the hidden message?

```
$ hexdump -C data.bin
00000000  5f 9e ae 09 3e 96 68 65  6c 6c 6f 00          |_...>.hello.|
0000000c
```

ujson

The `ujson` library is one of those libraries you are likely to use frequently when working with data in an IOT project. It provides encoding and decoding of JavaScript Object Notation (JSON) documents. This is because many of the IOT services available either require or can process JSON documents. Thus, you should consider getting into the habit of formatting your data in JSON to make it easier to integrate with other systems. The library implements the following functions that you can use to work with JSON documents.

- `ujson.dumps(obj)`: return a string decoded from a JSON object.
- `ujson.loads(str)`: parses the JSON string and returns a JSON object. Will raise an error if not formatted correctly.
- `ujson.load(fp)`: parse the contents of a file pointer (a file string containing a JSON document). Will raise an error if not formatted correctly.

Recall we saw a brief example of JSON documents in the last chapter. That example was written exclusively for the PC but a small change makes it possible to run it on a MicroPython board. Let's look at a similar example. Listing 5-4 shows an example of using the `ujson` library.

Listing 5-4. Demonstration of the `ujson` library features

```
# MicroPython for the IOT - Chapter 5
# Example use of the ujson library
# Note: change ujson to json to run it on your PC!
import ujson

# Prepare a list of JSON documents for pets by converting JSON to a
dictionary
vehicles = []
vehicles.append(ujson.loads('{"make":"Chevrolet", "year":2015,
"model":"Silverado", "color":"Pull me over red", "type":"pickup"}'))
vehicles.append(ujson.loads('{"make":"Yamaha", "year":2009, "model":"R1",
"color":"Blue/Silver", "type":"motorcycle"}'))
vehicles.append(ujson.loads('{"make":"SeaDoo", "year":1997,
"model":"Speedster", "color":"White", "type":"boat"}'))
vehicles.append(ujson.loads('{"make":"TaoJen", "year":2013,
"model":"Sicily", "color":"Black", "type":"Scooter"}'))

# Now, write these entries to a file. Note: overwrites the file
json_file = open("my_vehicles.json", "w")
for vehicle in vehicles:
    json_file.write(ujson.dumps(vehicle))
    json_file.write("\n")
json_file.close()

# Now, let's read the list of vehicles and print out their data
my_vehicles = []
json_file = open("my_vehicles.json", "r")
for vehicle in json_file.readlines():
    parsed_json = ujson.loads(vehicle)
    my_vehicles.append(parsed_json)
json_file.close()

# Finally, print a summary of the vehicles
print("Year Make Model Color")
for vehicle in my_vehicles:
    print(vehicle['year'],vehicle['make'],vehicle['model'],vehicle['color'])
```

The following shows the output of the script running on the WiPy.

```
>>> import examples.ujson_example
Year Make Model Color
2015 Chevrolet Silverado Pull me over red
2009 Yamaha R1 Blue/Silver
1997 SeaDoo Speedster White
2013 TaoJen Sicily Black
```

UOS

The `uos` library implements a set of functions for working with the base operating system. Some of the functions may be familiar if you have written programs for your PC. Most functions allow you to work with file and directory operations. The following lists several of the more commonly used functions.

- `uos.chdir(path)`: change the current directory
- `uos.getcwd()`: return the current working directory
- `uos.listdir([dir])`: list the current directory if `dir` is missing or list the directory specified
- `uos.mkdir(path)`: create a new directory
- `uos.remove(path)`: delete a file
- `uos.rmdir(path)`: delete a directory
- `uos.rename(old_path, new_path)`: rename a file
- `uos.stat(path)`: get status of a file or directory

In this example, we see how to change the working directory so that we can simplify our import statements. We also see how to create a new directory, rename it, create a file in the new directory, list the directory, and finally clean up (delete) the changes. Listing 5-5 shows the example for working with the `uos` library functions.

Listing 5-5. Demonstration of the `uos` library features

```
# MicroPython for the IOT - Chapter 5
# Example use of the uos library
# Note: change uos to os to run it on your PC!
import sys
import uos

# Create a function to display files in directory
def show_files():
    files = uos.listdir()
    sys.stdout.write("\nShow Files Output:\n")
```

```

sys.stdout.write("\tname\tsize\n")
for file in files:
    stats = uos.stat(file)
    # Print a directory with a "d" prefix and the size
    is_dir = True
    if stats[0] > 16384:
        is_dir = False
    if is_dir:
        sys.stdout.write("d\t")
    else:
        sys.stdout.write("\t")
    sys.stdout.write(file)
    if not is_dir:
        sys.stdout.write("\t")
        sys.stdout.write(str(stats[6]))
    sys.stdout.write("\n")

# List the current directory
show_files()
# Change to the examples directory
uos.chdir('examples')
show_files()

# Show how you can now use the import statement with the current dir
print("\nRun the ujson_example with import ujson_example after chdir()")
import ujson_example

# Create a directory
uos.mkdir("test")
show_files()

```

While this example is a little long, it shows some interesting tricks. Notice we created a function to print out the directory list rather than printing out the list of files returned. We also checked the status of the file to determine if the file was a directory or not and if it was, we printed a `d` to signal the name refers to a directory. We also used the `stdout` stream to control formatting with tabs (`\t`) and newline (`\n`) characters.

You will also see how to use `change directory` to improve how we use the `import` statement. Since we changed the directory to `examples`, the `import ujson_example` will attempt to find that module (library) in the current directory first. This is a nice trick you can use in your own projects to deploy your code to your board in a separate directory, which means you can use this to deploy multiple projects to your board placing each in its own directory.

Now let's see the output. Listing 5-6 shows the output when run on the WiPy. Take a few moments and look through the output to see how the functions worked. Also, note the output from the JSON example is shown since we imported that code in the middle of the script. Nice.

Listing 5-6. Demonstration of the uos library features (output)

```
>>> import examples.uos_example
```

```
Show Files Output:
```

```

    name    size
    main.py 34
d      sys
d      lib
d      cert
    boot.py 336
d      examples
    data.bin    15
    my_vehicles.json    377
```

```
Show Files Output:
```

```

    name    size
    sys_example.py 395
    uio_example.py 609
    ujson_example.py    1370
    uos_example.py 1163
    data.bin    15
```

```
Run the ujson_example with import ujson_example after chdir()
```

```
Year Make Model Color
2015 Chevrolet Silverado Pull me over red
2009 Yamaha R1 Blue/Silver
1997 SeaDoo Speedster White
2013 TaoJen Sicily Black
```

```
Show Files Output:
```

```

    name    size
    sys_example.py 395
    uio_example.py 609
    ujson_example.py    1370
    uos_example.py 1163
    data.bin    15
    my_vehicles.json    377
d      test
```

utime

The `utime` library is another popular library used in many projects. The library is used to get the current time, date, and working with time data such as calculating time differentials. Note that some of the functions only work with a real-time clock (RTC) installed either as a hardware extension or through a network time server. See [Chapter 6](#) for more details. The following lists several of the more commonly used functions related

to inserting delays in our scripts. Delays are helpful when we need to pause processing to allow a sensor to read or to wait for data from/to other sources. Another common use of this library is recording the date and time of an event or sensor data read.

- `utime.sleep(seconds)`: put the board in a sleep mode for the number of seconds specified
- `utime.sleep_ms(ms)`: put the board in a sleep mode for the number of milliseconds specified
- `utime.sleep_us(us)`: put the board in a sleep mode for the number of microseconds specified

Let's see a short example of how to use time delays. Here, we use a random function to sleep for a random period and to provide random values. Don't worry about the RTC code; we will see more about that in Chapter 6.

Listing 5-7. Demonstration of the `utime` library sleep features

```
# MicroPython for the IOT - Chapter 5
# Example use of the utime library
# Note: This example only works on the WiPy
# Note: This example will not on your PC.
import machine
import sys
import utime

# Since we don't have a RTC, we have to set the current time
# initialized the RTC in the WiPy machine library
from machine import RTC

# Init with default time and date
rtc = RTC()
# Init with a specific time and date. We use a specific
# datetime, but we could get this from a network time
# service.
rtc = RTC(datetime=(2017, 7, 15, 21, 32, 11, 0, None))

# Get a random number from 0-1 from a 2^24 bit random value
def get_rand():
    return machine.rng() / (2 ** 24 - 1)

# Format the time (epoch) for a better view
def format_time(tm_data):
    # Use a special shortcut to unpack tuple: *tm_data
    return "{0}-{1:0>2}-{2:0>2} {3:0>2}:{4:0>2}:{5:0>2}".format(*tm_data)
```



```

# Generate a random number of rows from 0-25
num_rows = get_rand() * 25

# Print a row using random seconds, milliseconds, or microseconds
# to simulate time.
print("Datetime          Value")
print("-----")
for i in range(0,num_rows):
    # Generate random value for our sensor
    value = get_rand() * 100
    # Wait a random number of seconds for time
    utime.sleep(int(get_rand() * 15)) # sleep up to 15 seconds
    print("{0} {1:0>{width}.4f}".format(format_time(rtc.now()), value, width=8))

```

■ **Note** This example also demonstrates how the firmware can differ among the boards available. This example only works on the WiPy and similar Pycom boards.

Notice there is a lot going on in this example than simply waiting a random number of seconds. This example shows you how to do that but also how to work with time data and formatting data. You typically must do this when logging data. Let's walk through this a bit.

Aside from the RTC code, we create two functions: one to generate a random number using the machine library of the WiPy (this function is not available in the Pyboard firmware), and another to format the datetime in a user-friendly format.

Since the function creates a random number of 24 bits, we divide by the max value for 24 bits to give a value between 0 and 1. We can then multiply that by an integer to get a value for a range. For example, if we wanted a value between 0 and 6, we can use `get_rand() * 6`.

The `format_time()` function returns a string representing the datetime. Here we see some advanced formatting options to format the date with the correct digits per part. Specifically, 4 digits for year and 2 for month, day, hour, minute, and second. The `:0>2` option tells the `format()` function to space the value over 2 positions (digits) with leading zeros. Cool, eh? Notice the last `print()` statement. Notice we use another trick to pass the width with a named parameter (`width`).

Finally, we get to the example code where we generate a random number of lines from 0 to 25, then loop for those iterations generating a random value (0-100) and then wait (sleep) for a random number of seconds (0-15) then print the new datetime and the value. This simulates a typical loop we would use to read data from sensors. As most sensors need time to read the value, the sleep function allows us to wait for that period.¹

¹Also, keep in mind this interval - called a sampling rate - must also make sense for the project. Sampling the ambient temperature in a controlled climate 30 times a second may generate a considerable amount of data that is useless because it rarely changes.

Now that we know more about what this code is doing, let's see an example of how it works. The following shows the output from running this code on the WiPy.

```
>>> import examples.uptime_example
Datetime          Value
-----
2017-07-15 21:32:12 014.1520
2017-07-15 21:32:25 003.5380
2017-07-15 21:32:28 044.8298
2017-07-15 21:32:35 099.0981
2017-07-15 21:32:41 086.8839
2017-07-15 21:32:47 083.8304
2017-07-15 21:32:55 083.0670
2017-07-15 21:32:59 064.3214
```

■ **Note** As mentioned in earlier chapters, there are some elements of MicroPython that are specific to a single MicroPython board or hardware. We will discuss those in the next chapter.

There are also built-in functions that are not part of any specific library and there are exceptions that allow us to capture error conditions. Let's look at those before we dive into some of the more commonly used standard libraries.

Built-In Functions and Classes

Python comes with many built-in functions: functions you can call directly from your script without importing them. There are many classes that you can use to define variables, work with data, and more. They're objects so you can use them to contain data and perform operations (functions) on the data. We've seen a few of these in the examples so far.

Let us see some of the major built-in functions and classes. Table 5-2 includes a short description of each. You can find more information about each at <https://docs.python.org/3/library/functions.html>. While this is the Python documentation, it applies to MicroPython too. Classes are designated with "class"; the remainder are functions.

Table 5-2. *MicroPython Built-in Functions and Classes*

Name	Description
abs(x)	Return the absolute value of a number.
all(iterable)	Return True if all elements of the iterable are true (or if the iterable is empty).
any(iterable)	Return True if any element of the iterable is true.
bin(x)	Convert an integer number to a binary string.
class bool([x])	Return a Boolean value, that is, one of True or False.
class bytearray([source[, encoding[, errors]])	Return a new array of bytes.
class bytes([source[, encoding[, errors]])	Return a new “bytes” object, which is an immutable sequence of integers in the range $0 \leq x < 256$
callable(object)	Return True if the object argument appears callable, False if not.
chr(i)	Return the string representing a character whose Unicode code point is the integer <i>i</i> .
classmethod(function)	Return a class method for function.
class complex([real[, imag]])	Return a complex number with the value $\text{real} + \text{imag} * 1j$ or convert a string or number to a complex number.
delattr(obj, name)	This is a relative of <code>setattr()</code> . The arguments are an object and a string. The string must be the name of one of the object’s attributes.
class dict()	Create a new dictionary.
dir([object])	Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.
divmod(a,b)	Take two (non-complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division.
enumerate(iterable, start=0)	Return an enumerate object. Iterable must be a sequence, an iterator, or some other object that supports iteration.
eval(expression, globals=None, locals=None)	Evaluate an expression using globals and locals as dictionaries in a local namespace.
exec(object[, globals[, locals]])	Execute a set of Python statements or object using globals and locals as dictionaries in a local namespace.

(continued)

Table 5-2. (continued)

Name	Description
<code>filter(function, iterable)</code>	Construct an iterator from those elements of iterable for which function returns true.
<code>class float([x])</code>	Return a floating-point number constructed from a number or string.
<code>class frozenset([iterable])</code>	Return a new frozenset object, optionally with elements taken from iterable.
<code>getattr(object, name[, default])</code>	Return the value of the named attribute of object. Name must be a string.
<code>globals()</code>	Return a dictionary representing the current global symbol table.
<code>hasattr(object, name)</code>	The arguments are an object and a string. The result is True if the string is the name of one of the object's attributes, False if not.
<code>hash(object)</code>	Return the hash value of the object (if it has one). Hash values are integers.
<code>hex(x)</code>	Convert an integer number to a lowercase hexadecimal string prefixed with "0x".
<code>id(object)</code>	Return the "identity" of an object.
<code>input([prompt])</code>	If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.
<code>class int(x)</code>	Return an integer object constructed from a number or string x, or return 0 if no arguments are given.
<code>isinstance(object, classinfo)</code>	Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect, or virtual) subclass thereof.
<code>issubclass(class, classinfo)</code>	Return true if class is a subclass (direct, indirect, or virtual) of classinfo.
<code>iter(object[, sentinel])</code>	Return an iterator object.
<code>len(s)</code>	Return the length (the number of items) of an object.
<code>class list([iterable])</code>	List sequence.
<code>locals()</code>	Update and return a dictionary representing the current local symbol table.

(continued)

Table 5-2. (continued)

Name	Description
map(function, iterable, ...)	Return an iterator that applies function to every item of iterable, yielding the results.
max([iterable arg*])	Return the largest item in an iterable or the largest of two or more arguments.
class memoryview(obj)	Return a “memory view” object created from the given argument.
min([iterable arg*])	Return the smallest item in an iterable or the smallest of two or more arguments.
next(iterator[, default])	Retrieve the next item from the iterator by calling its <code>__next__()</code> method.
class object0	Return a new featureless object. Object is a base for all classes.
oct(x)	Convert an integer number to an octal string.
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)	Open file and return a corresponding file object. Use <code>close()</code> to close the file.
ord(c)	Given a string representing one Unicode character, return an integer representing the Unicode code point of that character.
pow(x, y[, z])	Return x to the power y ; if z is present, return x to the power y , modulo z (computed more efficiently than $\text{pow}(x, y) \% z$).
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)	Print objects to the text stream file, separated by <code>sep</code> and followed by <code>end</code> . <code>sep</code> , <code>end</code> , <code>file</code> and <code>flush</code> ; if present, must be given as keyword arguments.
class property(fget=None, fset=None, fdel=None, doc=None)	Return a property attribute.
range([stop [start, stop[, step]]])	Range sequence.
repr(object)	Return a string containing a printable representation of an object.
reversed(seq)	Return a reverse iterator.
round(number[, ndigits])	Return number rounded to <code>ndigits</code> precision after the decimal point.
class set([iterable])	Return a new set object, optionally with elements taken from <code>iterable</code> .

(continued)

Table 5-2. (continued)

Name	Description
<code>setattr(object, name, value)</code>	This is the counterpart of <code>getattr()</code> . The arguments are an object, a string, and an arbitrary value.
<code>class slice(start, stop[, step])</code>	Return a slice object representing the set of indices specified by <code>range(start, stop, step)</code> .
<code>sorted(iterable[, key][, reverse])</code>	Return a new sorted list from the items in <code>iterable</code> .
<code>staticmethod(function)</code>	Return a static method for <code>function</code> .
<code>class str(object)</code>	Return a str version of <code>object</code> .
<code>sum(terable[, start])</code>	Sums <code>start</code> and the items of an iterable from left to right and returns the total.
<code>super([type[, object-or-type]])</code>	Return a proxy object that delegates function calls to a parent or sibling class of <code>type</code> .
<code>class tuple([iterable])</code>	Tuple sequence.
<code>type(object)</code>	Return the type of an object.
<code>zip(*iterables)</code>	Make an iterator that aggregates elements from each of the iterables.

You should look through this list and explore the links for those you find interesting and refer to the list when developing your projects so that you can use the most appropriate function or class. You may be surprised how much is “built-in.” However, once again, always check the documentation for your chosen MicroPython board for the latest functions and classes available in your firmware.

■ **Tip** See <https://docs.python.org/3/library/functions.html> for more information about the built-in functions and classes. See <http://docs.micropython.org/en/latest/pyboard/library/builtins.html> for the latest list of built-in functions and classes in MicroPython.

Let’s see an example of using one of the classes - the dictionary. The following shows how we can use the built-in class to create a variable of type `dict()` and later use it. Since the class is part of the built-in functionality, it will work on both Python and MicroPython.

```
>>> my_addr = dict()
>>> print(my_addr)
{}
>>> my_addr['street'] = '123 Main St'
>>> my_addr['city'] = 'Anywhere'
>>> my_addr['zip'] = 90125
>>> print(my_addr)
```

```
{'city': 'Anywhere', 'street': '123 Main St', 'zip': 90125}
>>> my_addr = {'street': '201 Cherry Tree Road', 'city': 'Goby', 'zip': 12345}
>>> print(my_addr)
{'city': 'Goby', 'street': '201 Cherry Tree Road', 'zip': 12345}
```

Here we see we can use the dictionary class to create a variable of that type. We can see this in the first `print()` call. Recall that the syntax for defining a dictionary is a set of curly braces. Next, we add values to the dictionary using the special syntax for accessing elements. Finally, we reassign the variable a new set of data using the more familiar dictionary syntax.

Now let's talk about a topic we haven't talked a lot about - exceptions. Exceptions are part of the built-in module for Python and can be a very important programming technique you will want to use. Perhaps not right away, but eventually you will appreciate the power and convenience of using exceptions in your code.

Exceptions

There is also a powerful mechanism we can use in Python (and MicroPython) to help manage or capture events when errors occur and execute code for a specific error. This construct is called exceptions and the exceptions (errors) we can capture are called exception classes. It uses a special syntax called the `try` statement (also called a clause since it requires at least one other clause to form a valid statement) to help us capture errors as they are generated. Exceptions can be generated anywhere in code with the `raise()` function. That is, if something goes wrong, a programmer can “raise” a specific, named exception and the `try` statement can be used to capture it via an `except` clause. Table 5-3 shows the list of exception classes available in MicroPython along with a short description of when (how) the exception could be raised.

Table 5-3. *MicroPython Exception Classes*

Exception Class	Description of Use
AssertionError	an assert() statement fails
AttributeError	an attribute reference fails
Exception	base exception class
ImportError	one or more modules failed to import
IndexError	subscript is out of range
KeyboardInterrupt	keyboard CTRL-C was issued or simulated
KeyError	key mapping in dictionary is not present in the list of keys
MemoryError	out of memory condition
NameError	a local or global name (variable, function, etc.) is not found
NotImplementedError	an abstract function has been encountered (it is incomplete)
OSError	any system-related error from the operating system
RuntimeError	possibly fatal error encountered on execution
StopIteration	an iterator's next function signaled no more values in iterable object
SyntaxError	code syntax error encountered
SystemExit	the sys.exit() function was called or simulated
TypeError	a function or operation is applied to an inappropriate type (like type mismatch)
ValueError	the right type but wrong value found (like out of bounds)
ZeroDivisionError	mathematical function results in a / 0

The syntax for the try statement is shown below. Each part of the construct is called a clause.

```
try_stmt ::= try1_stmt | try2_stmt
try1_stmt ::= "try" ":" code block
              ("except" [expression ["as" identifier]] ":" code block)+
              ["else" ":" code block]
              ["finally" ":" code block]
try2_stmt ::= "try" ":" code block
              "finally" ":" code block
```

Notice there are four clauses: try, except, else, and finally. The try clause is where we put our code (code block) - one or more lines of code that will be included in the exception capture. There can be only one try, else, and finally but you can have any number of except clauses naming an exception class. In fact, the except and else go together such that if an exception is detected running any of the lines of code in the try

clause, it will search the except clauses and if and only if no except clause is met, it will execute the else clause. The finally clause is used to execute after all exceptions are processed and executed. Notice also that there are two versions of the statement: one that contains one or more except and optionally an else and finally, and another that has only the try and finally clauses.

Let's look at one of the ways we can use the statement to capture errors in our code. Suppose you are reading data from a batch of sensors and the libraries (modules) for those sensors raise `ValueError` if the value read is out of range or invalid. It may also be the case that you don't want the data from any other sensors if one or more fail. So, we can use code like the following to "try" to read each of the sensors and if there is a `ValueError`, issue a warning and keep going or if some other error is encountered, flag it as an error during the read. Note that typically we would not stop the program at that point; rather, we would normally log it and keep going. Study the following until you're convinced exceptions are cool.

```
values = []
print("Start sensor read.")
try:
    values.append(read_sensor(pin11))
    values.append(read_sensor(pin12))
    values.append(read_sensor(pin13))
    values.append(read_sensor(pin17))
    values.append(read_sensor(pin18))
except ValueError as err:
    print("WARNING: One or more sensors valued to read a correct value.", err)
except:
    print("ERROR: fatal error reading sensors.")
finally:
    print("Sensor read complete.")
```

Another way we can use exceptions is when we want to import a module (library) but we're not sure if it is present. For example, suppose there was a module named `piano` that has a function named `keys()` that you want to import but the module may or may not be on the system. In this case, we may have other code we can use instead creating our own version of `keys()`. To test if the module can be imported, we can place our import inside a try block as shown below. We can then detect if the import fails and take appropriate steps.

```
# Try to import the keys() function from piano. If not present,
# use a simulated version of the keys() function.
try:
    from piano import keys
except ImportError as err:
    print("WARNING:", err)
    def keys():
        return(['A', 'B', 'C', 'D', 'E', 'F', 'G'])
print("Keys:", keys())
```

If we added code like this and the module were not present, not only can we respond with a warning message, but we can also define our own function to use if the module isn't present.

Finally, you can raise any exception you want including creating your own exceptions. Creating custom exceptions is an advanced topic, but let's see how we can raise exceptions since we may want to do that if we write our own custom libraries. Suppose you have a block of code that is reading values but it is possible that a value may be out of range. That is, too large for an integer, too small for the valid range of values expected, etc. You can simply raise the `ValueError` passing in your custom error message as follows with the `raise` statement and a valid exception class declaration.

```
raise ValueError("ERROR: the value read from the sensor ({0}) is not in
range.".format(val_read))
```

You can then use the `try` statement to capture this condition since you know it is possible and work your code around it. For example, if you were reading data, you could elect to skip the read and move on - continue the loop. However, if this exception was to be encountered when running your code and there were no `try` statements, you could get an error like the following, which even though is fatal, is still informative.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ERROR: the value read from the sensor (-12) is not in range.
```

You can use similar techniques as shown here to make your MicroPython code more robust and tolerant of errors. Better still, you can write your code to anticipate errors and react to them in a graceful, controlled manner.

■ **Tip** For a more in-depth look at Python exceptions, see <https://docs.python.org/3/tutorial/errors.html>.

MicroPython Libraries

There are also libraries that are built expressly for the MicroPython system. These are libraries designed to help facilitate using MicroPython on the hardware. Like the built-in libraries, there are some MicroPython libraries that apply to one board or another and those that differ from one board to another. That is, there are subtle differences in the libraries that prevent them from working on more than one board. Always consult the documentation for the firmware for your board for a complete list of the MicroPython libraries available.

Overview

The MicroPython libraries provide functionality that is specific to the MicroPython implementation of Python. There are libraries with functions for working with the hardware directly, the MicroPython system, and networking. We will see examples of some of the features of each of these libraries in the following sections. Table 5-4 contains a list of the current MicroPython libraries that are common to most boards. The first column is the name we use in our `import` statement, the second is a short description, and the third contains a link to the online documentation.

Table 5-4. *MicroPython-Specific Libraries*

Name	Description	Documentation
framebuf	Frame buffer	http://docs.micropython.org/en/latest/pyboard/library/framebuf.html
machine	Hardware-related functions	http://docs.micropython.org/en/latest/pyboard/library/machine.html
micropython	MicroPython internals	http://docs.micropython.org/en/latest/pyboard/library/micropython.html
network	Networking	http://docs.micropython.org/en/latest/pyboard/library/network.html
uctypes	Access binary data	http://docs.micropython.org/en/latest/pyboard/library/uctypes.html

■ **Tip** See <http://docs.micropython.org/en/latest/pyboard/library/index.html#micropython-specific-libraries> for more details about any of the MicroPython libraries.

Next, we will look at a few of the more common MicroPython libraries and see some code examples for each.

Common MicroPython Libraries

Now let's look at examples of some of the more commonly used MicroPython libraries. What follows is just a sampling of what you can do with each of the libraries. See the online documentation for a full description of all the capabilities. Once again, the MicroPython libraries may vary slightly from one board (firmware) to another. In fact, the Pycom documentation states this quite clearly (<https://docs.pycom.io/chapter/firmwareapi/pycom/>).

These modules are specific to the Pycom devices and may have slightly different implementations to other variations of MicroPython (i.e. for Non-Pycom devices). Modules include those which support access to underlying hardware, e.g. I2C, SPI, WLAN, Bluetooth, etc.

machine

The machine library contains functions related to the hardware, providing an abstraction layer that you can write code to interact with the hardware. Thus, this library is the main library you will use to access features like timers, communication protocols, CPU, and more. Since this functionality is communicating directly with the hardware, you should take care when experimenting to avoid changing or even potentially damaging the performance or configuration of your board. For example, using the library incorrectly could lead to lockups, reboots, or crashes.

■ **Caution** Take care when working with the low-level machine library to avoid changing or even potentially damaging the performance or configuration of your board.

Since the machine library is a low-level hardware abstraction, we will not cover it in-depth in this chapter. Rather, we will see more of the hardware features in the next chapter. In the meantime, let's explore another interesting gem of MicroPython knowledge by showing you how to discover what a library contains through the help function. For example, Listing 5-8 shows an excerpt of what is reported through the REPL console when we issue the statement `help(machine)` on the WiPy when connected to the Expansion Board. The `help()` function will display all the functions and constants that are available for use in the library. While it doesn't replace a detailed explanation or even a complete example, it can be useful when encountering a library for the first time.

Listing 5-8. The machine Library Help

```
>>> import machine
>>> help(machine)
<module 'umachine'>object is of type module
  __name__ -- umachine
  mem8 -- <8-bit memory>
  mem16 -- <16-bit memory>
  mem32 -- <32-bit memory>
  reset -- <function>
  ...
  Pin -- <class 'Pin'>
  UART -- <class 'UART'>
  SPI -- <class 'SPI'>
  I2C -- <class 'I2C'>
  PWM -- <class 'PWM'>
```

```

ADC -- <class 'ADC'>
DAC -- <class 'DAC'>
SD -- <class 'SD'>
Timer -- <class 'Timer'>
RTC -- <class 'RTC'>
WDT -- <class 'WDT'>
PWRON_RESET -- 0
HARD_RESET -- 1
WDT_RESET -- 2
...

```

Notice there is a lot of information there! What this gives us most is the list of classes we can use to interact with the hardware. Here we see there are classes for UART, SPI, I2C, PWM, and more. Let's contrast that to the same output from the Pyboard. The following shows the same excerpt of classes for the Pyboard.

```

Pin -- <class 'Pin'>
Signal -- <class 'Signal'>
I2C -- <class 'I2C'>
SPI -- <class 'SPI'>
UART -- <class 'UART'>
WDT -- <class 'WDT'>

```

Notice there are a few missing. We will explore some of these differences in the next chapter. It is always a good idea to check the output of `help(machine)` on a board you're using for the first time. It may save you a lot of headaches trying to find support for hardware that doesn't exist!

network

The network library on the Pyboard is used to install network drivers (classes for interacting with the networking abstraction) and configuring the settings. While the other libraries discussed are a little different from one board to another, this library is one that is quite different among the Pyboard and WiPy (and other) boards. In fact, the WiPy has the same library but contains different classes.

This is largely since the WiPy has built-in WiFi, it has its own mechanisms for working with networks. Recall we saw the network library for the Pyboard demonstrated briefly in Chapter 3 and we will see it used again later in the example chapters. That is, we learned there are only two network drivers for the Pyboard (and similar) boards: CC3K and WINNET5K. The WiPy network library contains three classes: WLAN, Bluetooth, and Server. Recall also we saw an example of the WLAN class in Chapter 3.

We will see more about the networking classes and the intriguing Bluetooth class in the next chapter.

Custom Libraries

Building your own custom libraries may seem like a daunting task, but it isn't. What is possibly a bit of a challenge is figuring out what you want the library to do and making the library abstract (enough) to be used by any script. The rules and best practices for programming come to play here such as data abstraction, API immutability, etc.

We discussed creating your own modules in the last chapter. In this section, we will look at how to organize our code modules into a library (package) that we can deploy (copy) to our MicroPython board and use in all our programs. This example, though trivial, is a complete example that you can use as a template should you decide to make your own custom libraries.

For this example, we will create a library with two modules: one that contains code to perform value conversions for a sensor, and another that contains helper functions for our projects - general functions that we want to reuse. We will name the library `my_helper`. It will contain two code modules: `sensor_convert.py` and `helper_functions.py`. Recall we will also need a `__init__.py` file to help MicroPython import the functions correctly but we will get back to that in a moment. Let's look at the first code module.

We will place the files in a directory named `my_helper` (same as the library name). This is typical convention and you can put whatever name you want, but you must remember it since we will use that name when importing the library in our code. Figure 5-1 shows an example layout of the files. This image was taken of my Pyboard flash drive.

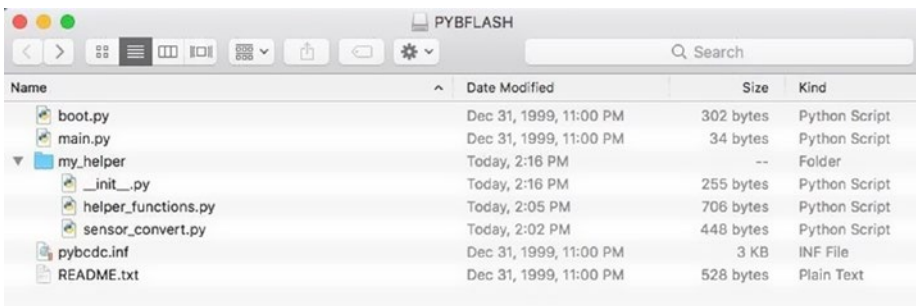


Figure 5-1. Directory/File Layout for the `my_helper` Sample Library on Pyboard

Notice we moved (copied) the directory containing the three files to my Pyboard. If you are copying the files to the WiPy, you will need to use ftp using the commands in Listing 5-9. Notice also we created the directory while connected to the WiPy and that we are in the directory containing the files we want to copy.

■ **Note** If you use Linux, you may need to add the `-p` option (passive) for the ftp command.

Listing 5-9. Using ftp to copy files to the WiPy

```

$ ftp 192.168.4.1
Connected to 192.168.4.1.
220 Micropython FTP Server
Name (192.168.4.1:cbell): micro
Password:
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd flash
250
ftp> ls
227 (192,168,4,1,7,232)
150
-rw-rw-r--  1 root  root           34 Jan  1  1980 main.py
drw-rw-r--  1 root  root            0 Jan  1  1980 sys
drw-rw-r--  1 root  root            0 Jan  1  1980 lib
drw-rw-r--  1 root  root            0 Jan  1  1980 cert
-rw-rw-r--  1 root  root          336 Jan  1  2098 boot.py
drw-rw-r--  1 root  root            0 Jan  1  2098 examples
-rw-rw-r--  1 root  root           15 Jan  1  2098 data.bin
-rw-rw-r--  1 root  root          377 Jan  1  2098 my_vehicles.json
226
ftp> mkdir my_helper
250
ftp> cd my_helper
250
ftp> put __init__.py
local: __init__.py remote: __init__.py
227 (192,168,4,1,7,232)
150
100% |*****| 255      1.24 MiB/s   00:00 ETA
226
255 bytes sent in 00:00 (0.61 KiB/s)
ftp> put helper_functions.py
local: helper_functions.py remote: helper_functions.py
227 (192,168,4,1,7,232)
150
100% |*****| 703      5.49 MiB/s   00:00 ETA
226
703 bytes sent in 00:00 (1.34 KiB/s)
ftp> put sensor_convert.py
local: sensor_convert.py remote: sensor_convert.py
227 (192,168,4,1,7,232)
150
100% |*****| 448      4.10 MiB/s   00:00 ETA
226
448 bytes sent in 00:00 (1.27 KiB/s)

```

Now let's look at the code. The first module is named `helper_functions.py` and contains the two helper functions from a previous example. Since these are generic, we placed them in this code module. However, we want to use this code on all my boards. The problem is, the `machine.rng()` function returns a different size for the random number. On the Pyboard, it's 30 bits but on the WiPy it's only 24 bits. So, we use a `try` statement to detect if that library function is available setting a global variable that can be used in the `get_rand()` to return the correct value. Listing 5-10 shows the complete code for the module.

Listing 5-10. The `helper_functions.py` module

```
# MicroPython for the IOT - Chapter 5
# Example module for the my_helper library
# This module contains helper functions for general use.

try:
    import pyb
    _PYBOARD = True
except ImportError:
    import machine
    _PYBOARD = False

# Get a random number from 0-1 from a 2^24 bit random value
# if run on the WiPy, return 0-1 from a 2^30 bit random
# value if the Pyboard is used.
def get_rand():
    if _PYBOARD:
        return pyb.rng() / (2 ** 30 - 1)
    return machine.rng() / (2 ** 24 - 1)

# Format the time (epoch) for a better view
def format_time(tm_data):
    # Use a special shortcut to unpack tuple: *tm_data
    return "{0}-{1:0>2}-{2:0>2} {3:0>2}:{4:0>2}:{5:0>2}".format(*tm_data)
```

The second code module is named `sensor_convert.py` and contains functions that are helpful in converting sensor raw values into a string for qualitative comparisons. For example, the function `get_moisture_level()` returns a string based on the threshold of the raw value. The datasheet for the sensor will define such values and you should use those in your code until and unless you can calibrate the sensor. In this case, if the value is less than the lower bound, the soil is dry; if greater than the upper bound, the soil is wet. Listing 5-11 shows the complete code for the module.

Listing 5-11. The `sensor_convert.py` module

```
# MicroPython for the IOT - Chapter 5
# Example module for the my_helper library

# This function converts values read from the sensor to a
# string for use in qualifying the moisture level read.

# Constants - adjust to "tune" your sensor

_UPPER_BOUND = 400
_LOWER_BOUND = 250

def get_moisture_level(raw_value):
    if raw_value <= _LOWER_BOUND:
        return("dry")
    elif raw_value >= _UPPER_BOUND:
        return("wet")
    return("ok")
```

Now let's go over the `__init__.py` file. This is a very mysterious file that developers often get very confused about. If you do not include one in your library directory, you should import what you want to use manually. That is, with something like `import my_helper.helper_functions`. But with the file, you can do your imports at one time allowing a simple `import my_helper` statement, which will import all the files. Let's look at the `__init__.py` file. The following shows the contents of the file.

```
# Metadata
__name__ = "Chuck's Python Helper Library"
__all__ = ['format_time', 'get_rand', 'get_moisture_level']
# Library-level imports
from my_helper.helper_functions import format_time, get_rand
from my_helper.sensor_convert import get_moisture_level
```

Notice on the first line we use a special constant to set the name of the library. The next constant limits what will be imported by the `*` (all) option for the import statement. Since it lists all the methods, it's just an exercise but a good habit to use especially if your library and modules contain many internal functions that you do not want to make usable to others. The last two lines show the import statements used to import the functions from the modules making them available to anyone who imports the library. The following shows a short example of how to do that along with how to use an alias. Here, we use `myh` as the alias for `my_helper`.

```
>>> import my_helper as myh
>>> myh.get_rand()
0.2830396
```

We can now access (use) all three functions by that alias as shown below.

```
r_int = myh.get_rand() * 10
print(myh.format_time(value))
print(myh.get_moisture_level(sensor_raw))
```

In case you're wondering, the help function works on this custom library too!

```
>>> import my_helper
>>> help(my_helper)
object <module 'Chuck's Python Helper Library' from 'my_helper/__init__.py'>
is of type module
  __path__ -- my_helper
  helper_functions -- <module 'my_helper.helper_functions' from 'my_helper/
  helper_functions.py'>
  __name__ -- Chuck's Python Helper Library
  sensor_convert -- <module 'my_helper.sensor_convert' from 'my_helper/
  sensor_convert.py'>
  get_rand -- <function get_rand at 0x20004270>
  format_time -- <function format_time at 0x20004300>
  __file__ -- my_helper/__init__.py
  __all__ -- ['format_time', 'get_rand', 'get_moisture_level']
  get_moisture_level -- <function get_moisture_level at 0x20004920>
```

Once you have started experimenting with MicroPython and have completed several projects, you may start to build up a set of functions that you reuse from time to time. These are perfect candidates to place into a library. It is perfectly fine if the functions are not part of a larger class or object. So long as you organize them into modules of like functionality, you may not need to worry about making them classes. On the other hand, if data is involved or the set of functions works on a set of data, you should consider making that set of functions a class for easier use and better quality code.

WAIT, WHAT ABOUT CIRCUITPYTHON?

Recall from Chapter 3 we discussed CircuitPython when we looked at the Circuit Playground board from Adafruit. This chapter does not cover CircuitPython in more detail because it is a port of MicroPython and thus what we learn about MicroPython libraries applies. What is different is some of the board-specific libraries and functions, and CircuitPython has some advanced libraries specific to the Adafruit boards. For more information about CircuitPython, see <https://circuitpython.readthedocs.io/en/stable/>.

Summary

The MicroPython firmware has a lot of capability for IOT projects. Indeed, there are a lot of different classes we can use to write robust and complex MicroPython programs from built-in functions that give the language a breadth of capability to working with data, performing calculations, and even working with time values to interfacing directly with the hardware.

Working with the hardware is one area that differs the most about MicroPython from one board to another. This is because the boards are quite different. Some have networking, some do not. Some have more on-board features than others, and some have less memory and even fewer GPIO pins. It is no surprise then that the firmware differs from one board to the next when it comes to the hardware abstraction layers.

In this chapter, we explored some of the more commonly used built-in and MicroPython libraries that generally apply to all boards. In the next chapter, we will dive a bit deeper into the lower-level libraries and hardware support in MicroPython including those board-specific libraries we can use such as those for the Pyboard, WiPy, and more.

CHAPTER 6



Low-Level Hardware Support

The MicroPython firmware, at the most basic of functionality, is the same from board to board for all the general Python language supported and many of the built-in functions. However, some of the libraries in the MicroPython firmware have a few minor differences from one board to another. In some cases, there are more libraries or classes available than others or perhaps the classes are organized differently, but most implement the same core libraries in one form or another. The same cannot be said to be true at the lower-level hardware abstraction layers. This is simply because one board vendor may implement different hardware than others. In some cases, the board has features that are not present on other boards. For example, the WiPy has WiFi and Bluetooth but the Pyboard has neither.

In this chapter, we will look at several examples of the low-level hardware support in MicroPython. We will learn about the board-specific libraries as well as some of the lower-level specialized libraries such as those for Bluetooth, SPI, I2C, and more. We will also see short code examples to illustrate the capabilities of the board-specific libraries. Some of these will be short snippets of code rather than actual projects you can implement yourself.

However, there are a few that are complete projects, which you may want to explore, but most are explained without a lot of depth and are intended to be examples rather than detailed walkthroughs. Also, keep in mind they will likely require a specific breakout board and MicroPython board as well as other accessories to implement. Again, these are for demonstration purposes. We will see more complete, step-by-step examples in later chapters, including how to assemble the hardware.

To keep things brief, we will explore the board-specific libraries that differ on the Pyboard and WiPy. Other boards may be different further still, but you will need to check the vendor documentation to know how they differ. This chapter should provide you with the insight to find those differences. We will also revisit working with breakout boards to demonstrate some of the libraries and hardware protocols and techniques discussed in previous chapters.

Let's begin with a look at the board-specific libraries for the Pyboard and WiPy.

Board-Specific Libraries

We have already seen that there are differences in the machine library between the Pyboard and WiPy. But there are other differences: libraries that are specific to each board and thus can only be used with another of the same board. These include libraries with functions and classes to work with low-level hardware, board-specific functions, and constants, etc.

Perhaps the one thing you should remember when working with a new MicroPython board for the first time is the firmware at the hardware level is very likely to be different from the last MicroPython board you used. This is especially true when you look at the firmware ports for boards like the BBC micro:bit, Circuit Playground, ESP8266, etc.

■ **Tip** Be sure to visit the documentation for your board for a complete list of functions, uses, and more examples of the board-specific libraries.

Pyboard

The Pyboard-specific libraries are unique in that they contain support for their vendor-specific skins. As we learned in Chapter 3, there are several skins available for the Pyboard. Fortunately, all the skins are supported directly in the firmware with Pyboard-specific classes. This includes the audio skin, LCD, and more.

There are also libraries specific to the Pyboard itself in the form of the `pyb` library, which includes support for several board-specific functions including time, power, interrupt, and more. We will also take a brief tour of that library and then look at one of the libraries for the LCD skin: the `Lcd160cr` class.

The following sections briefly describe the libraries in more detail and include a few commonly found examples of some of the features. See the indicated documentation for complete details of the contents and features of each library.

`pyb`

The `pyb` library is the catch-all library for the Pyboard-related functions and classes. If you're looking for a function or library related to the Pyboard or any of its on-board hardware, this library is the place you should look first. The following lists the function groups and classes available.

- *Time*: time-related functions for delays by millisecond or microsecond and calculating the number of milliseconds or microseconds since an event (saved variable)
- *Reset*: functions that permit you to turn on debug, initiate bootloader mode, or reset the board
- *Interrupt*: enable or disable interrupts

- *Power*: functions to put the board into sleep or deep sleep mode and change the performance characteristics (tread lightly)
- *Miscellaneous*: various functions for information, controlling UART, USB, and mounting block devices

Now let's see a few short examples. The first two show examples of the reset functions, the second two are miscellaneous functions, and the last shows how to use one of the classes to interact with the hardware on the board.

You can use the `pyb.hard_reset()` function to perform a hard reset as follows. If you enter this statement in a REPL console, the board will reset in the same manner as if you pressed the reset button. This could be handy if you need to abort from a serious error or hardware fault.

```
MicroPython v1.9.1-154-g4d55d880 on 2017-07-11; PYBV1.1 with STM32F405RG
Type "help()" for more information.
>>> import pyb
>>> pyb.hard_reset()
```

■ **Note** The `hard_reset()` function can cause your PC to complain that the SD card was ejected without stopping the file system, so use this function sparingly.

The `pyb.bootloader()` function places the board in bootloader mode. Recall from Chapter 3, to install firmware, hard to power off the board, install a jumper wire, and power the board on before we can load firmware. With this function, you can do so from the REPL console. That is, if you can get to a REPL console. If your board is damaged or the firmware is corrupt, you can still perform the jumper wire process to place the board in bootloader mode to load new firmware.

The `pyb.info()` function is used as an informational dump. It prints out lots of low-level information about the board including hardware addresses and more. Most of it is not useful unless you're writing low-level drivers or something similar, but you can pass `True` to the function and get more information in the form of a dump of the GC memory layout as shown in the excerpt below. If you're curious about this data, see the Pyboard hardware specification section in the online reference manual or the forums at forum.micropython.org.

```
>>> pyb.info(True)
...
LFS free: 52224 bytes
GC memory layout; from 20003e40:
0000: h=hhhhhBShh==Sh=hhhh==h==Bh=hBh=hBhThShh=h==hh=BhhhhBhBh=hh=hBh=h
00400: =hTh==Shhh==B.....h=.....h==
00800: ====h=====
00c00: =====
      (92 lines all free)
18000: .....
```

The `pyb.main(filename)` function is one of the most useful functions. It sets the filename of the main script to run after `boot.py` is finished. You can use this function in conjunction with several alternative `main.py` code modules. This can be a useful tool if you wanted to implement a few projects that launch and run automatically. Rather than going into the `boot.py` manually and changing it, you can use this function to tell the Pyboard to start with an alternative file. That way, you can have “profiles” you can use to change the behavior of the board. Cool. Simply call the function with a valid code module path and name (as a string). This setting remains in effect until you change it.

■ **Note** You should only call this function from within `boot.py`.

The `pyb` library also has low-level hardware classes for a host of hardware supported. Note that we saw a similar set of classes for the WiPy in the machine library. On the Pyboard, they’re in the `pyb` library. Table 6-1 shows a list of the available hardware classes. As you can see, there is support for the LEDs, LCD (we’ll see this one in a later section), GPIO pins, and much more. Keep in mind the class names are case sensitive.

Table 6-1. *Pyboard Low-Level Hardware Classes (pyb library)*

Class	Description
Accel	accelerometer control
ADC	analog to digital conversion
CAN	controller area network communication bus
DAC	digital to analog conversion
ExtInt	configure I/O pins to interrupt on external events
I2C	a two-wire serial protocol
LCD	LCD control for the LCD touch-sensor pypskin
LED	LED object
Pin	control I/O pins
PinAF	Pin Alternate Functions
RTC	real-time clock
Servo	3-wire hobby servo driver
SPI	a master-driven serial protocol
Switch	switch object
Timer	control internal timers
TimerChannel	set up a channel for a timer
UART	duplex serial communication bus
USB_HID	USB Human Interface Device (HID)
USB_VCP	USB virtual comm port

Now, let's see one of these in action. We will use the accelerometer to do a simple test of the board. That is, we will write some code that you can use to run and detect when the Pyboard has been physically moved in any of the three directions. The following shows the methods available for the `Accel` class.

- `Accel.filtered_xyz()`: get a 3-tuple of filtered x, y and z values
- `Accel.tilt()`: get the tilt register
- `Accel.x()`: get the x-axis value
- `Accel.y()`: get the y-axis value
- `Accel.z()`: get the z-axis value

An accelerometer is a device used to measure the change in velocity over time: in other words, how an object has moved and how fast. While the accelerometer on the Pyboard is not a precision instrument capable of super-fine detection of minute movement, you can use it to detect when the board has been moved and even to a limited degree which direction (axis of movement). Thus, you can incorporate the accelerometer in your project if you want to know when the project has been moved either for security or to detect or change behavior based on orientation. There are a lot of cool ways you can use an accelerometer!

Now, let's see some code! The following shows how you can use the accelerometer to get its raw values for each of the X, Y, and Z axes. Values range from -32 to 32. With a little observation, we can deduce what values indicate how far the board is moved in each direction. Try the code below and gently pick up the board and move it around in three dimensions by rotating it different directions. Do it slowly, then a bit more quickly and notice how the values change. Then, place the board back in its original position and observe. You can press CTRL-C to stop the loop.

```
import pyb
from pyb import Accel
acc = Accel()
print("Press CTRL-C to stop.")
while True:
    pyb.delay(500)    # Short delay before sampling
    print("X = {0:03} Y = {1:03} Z = {2:03}".format(acc.x(), acc.y(), acc.z()))
```

USE SOFT RESET TO REFRESH IMPORT

If you're like me and like to write your MicroPython code on your PC and test it there, then move it to the board for more development and testing, chances are you will be perplexed by having to reset the board each time you copy a new version of a code module or library you want to include and test. Instead of powering off the board or doing a hard reset, you can do a soft reset using the CTRL+D key when connected in the REPL console. This does a soft reboot that allows you to run the import again. In fact, you must issue the `import` again - it just rebooted! The best part is you won't have to reconnect your board or the REPL console! Try it.

The following shows an excerpt of the output when run in the REPL console.

```
X = 002 Y = 000 Z = 021
X = 000 Y = -01 Z = 023
X = 005 Y = 020 Z = 000
X = -05 Y = 016 Z = -11
X = -11 Y = -13 Z = 014
X = 022 Y = -04 Z = -05
X = -03 Y = 019 Z = 012
...

```

If you'd like a challenge, you can try writing a script to use the accelerometer to detect when the board has been turned upside down. Hint: observe the X, Y, and Z raw values as you experiment and write your code to look for specific ranges.

Caution On the Pyboard, the accelerometer uses I2C(1) so you cannot use the accelerometer and I2C(1) at the same time. You must use different pins for I2C to use the accelerometer and an I2C device at the same time.

These examples are only a very small portion of what is available in this library. If you're working with the Pyboard, you should look at this library first for all your low-level hardware and board-specific control needs. We will see examples of some of the classes available and their use in a later section.

Tip See <http://docs.micropython.org/en/latest/pyboard/library/pyb.html> for a complete list of the functions and classes available in the pyb library.

lcd160cr

The makers of the Pyboard have made an interesting skin that is, so far, unique among the vendors offering MicroPython boards. This skin is called the lcd160cr; hence LCD, which is a touch-sensitive LCD screen that you can connect directly to your Pyboard giving you a very nice touch screen that you can use to make a modern user interface. Think about it – you can build a MicroPython wristwatch, weather sensor, or anything that needs a user interface.

The library that is built in to the Pyboard firmware (named lcd160cr), allows you to sense when the screen is touched (and where), and you can send text or draw shapes. This means, with a little imagination, you can build simple graphical user interfaces. Very nice.

The LCD can be mounted in one of two locations, which are referred to as positions X and Y. These values are then used in the constructor. Figure 6-1 shows the positions for connecting the LCD to the Pyboard in the X and Y positions (shown left to right).

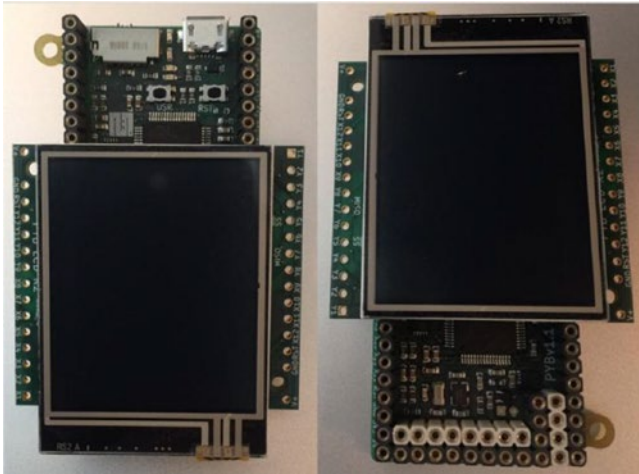


Figure 6-1. Positions for Mounting the LCD (Pyboard)

The difference in the position refers to with how the LCD is oriented on the Pyboard. In the X position, the LCD is mounted on X range of GPIO pins and in the Y position, it is mounted on the Y range of GPIO pins. Note that in the Y position, the LCD must be rotated 180 degrees as shown.

Now, let's see an example of using the LCD. In this example, we will create a simple user interface that detects touch in the four corners of the screen. To make it interesting, we will also turn on a different LED with each touch. This will give you visual feedback that you've touched the screen.

This example is one of the longer examples in this book but it is not difficult to follow. Let's begin with a high-level walkthrough of the code.

The first thing we should do is write the `import` statements so that we can detect when the code is run on a Pyboard (or another board). If it isn't a Pyboard, we should abort since other boards do not have the library we will need (`lcd160cr`). If you adopt this technique in your own code, you can avoid strange import exceptions and other issues that may not be clear as to why the program fails. Plus, it's good programming. The following shows the code for detecting the presence of the libraries needed and how to exit the program if one of them fails to import.

```
# First, make sure this is running on a Pyboard
try:
    import pyb
except ImportError:
    print("ERROR: not on a Pyboard!")
    sys.exit(-1)
```

```
# Next, make sure the LCD skin library in the firmware
try:
    import lcd16ocr
except ImportError:
    print("ERROR: LCD16OCR library missing!")
    sys.exit(-1)
```

Next, we will write a function to turn the LED on. This is an example of how to write code to be reusable. That is, we don't want to repeat the same code repeatedly. In this case, we can turn on a specific LED by color. To help us with that, we can write a function to retrieve the LED based on color (1=red, 2=green, etc.). The following shows the helper function.

```
def led_color(num):
    if num == 1: return "red"
    elif num == 2: return "green"
    elif num == 3: return "orange"
    else: return "blue"
```

The following shows the reusable function for turning on the LED. As you can see, we first turn off the old LED then turn the new (selected) one on.

```
def turn_on_led(num, led_last):
    # turn last LED off
    led = pyb.LED(led_last)
    led.off()
    led = pyb.LED(num)
    led.on()
    sys.stdout.write("Turning off ")
    sys.stdout.write(led_color(led_last))
    sys.stdout.write(" - Turning on ")
    sys.stdout.write(led_color(num))
    sys.stdout.write("\n")
```

Next, we can write the code to read the position on the LCD that was touched and, depending on where the touch occurred, turn on the LED for that corner. We will use red for the upper-left, green for upper-right, orange for lower-right, and blue for lower-right.

The code is simplified by using numbers for the LEDs since the `pyb.LED()` refers to LEDs by numbers. Savvy readers may spot a way to improve the code with enumeration or even the use of constants. If you see these potential improvements, feel free to make those improvements as an exercise. Listing 6-1 shows the completed code for the example.

Listing 6-1. Using the Pyboard LCD (lcd160cr)

```

# MicroPython for the IOT - Chapter 6
# Example module for the LCD skin on a Pyboard
#
# Note: The LCD is positioned in the "X" position.
#
import sys

# First, make sure this is running on a Pyboard
try:
    import pyb
except ImportError:
    print("ERROR: not on a Pyboard!")
    sys.exit(-1)

# Next, make sure the LCD skin library in the firmware
try:
    import lcd160cr
except ImportError:
    print("ERROR: LCD160CR library missing!")
    sys.exit(-1)

# Return color of LED
def led_color(num):
    if num == 1: return "red"
    elif num == 2: return "green"
    elif num == 3: return "orange"
    else: return "blue"

# Use a method to turn off last LED and the next one on
def turn_on_led(num, led_last):
    # turn last LED off
    led = pyb.LED(led_last)
    led.off()
    led = pyb.LED(num)
    led.on()
    sys.stdout.write("Turning off ")
    sys.stdout.write(led_color(led_last))
    sys.stdout.write(" - Turning on ")
    sys.stdout.write(led_color(num))
    sys.stdout.write("\n")

# Setup the LCD in the "X" position
lcd = lcd160cr.LCD160CR('X')

```

```

for j in range(1, 4): # Turn off all of the LEDs
    led = pyb.LED(j) # Get the LED
    led.off()        # Turn the LED off

# Now, let's play a game. Let's change the LEDs to
# different colors depending on where the user touches
# the screen.
print("Welcome to the touch screen demo!")
print("Touch the screen in the corners to change the LED lit.")
print("Touch the center to exit.")
center = False
last_led = 1
while not center:
    pyb.delay(50)
    if lcd.is_touched:
        touch = lcd.get_touch()
        if (touch[0] == 0):
            continue
        # Upper-left corner
        if ((touch[1] <= 60) and (touch[2] <= 60)):
            turn_on_led(1, last_led)
            last_led = 1
        # Upper-right corner
        elif ((touch[1] >= 100) and (touch[2] <= 60)):
            turn_on_led(2, last_led)
            last_led = 2
        # Lower-right corner
        elif ((touch[1] >= 100) and (touch[2] >= 100)):
            turn_on_led(3, last_led)
            last_led = 3
        # Lower-left corner
        elif ((touch[1] <= 60) and (touch[2] >= 100)):
            turn_on_led(4, last_led)
            last_led = 4
        # Center
        elif ((touch[1] > 60) and (touch[1] < 100) and (touch[2] > 60) and
              (touch[2] < 100)):
            led = pyb.LED(last_led)
            led.off()
            center = True

print("Thanks for playing!")
sys.exit(0)

```

If you have an LCD, go ahead and try this example. Once you do, you should see the LEDs light as you touch each corner and you will see in your REPL console output like the following.

```
>>> import pyboard_lcd
Welcome to the touch screen demo!
Touch the screen in the corners to change the LED lit.
Touch the center to exit.
Turning off red - Turning on green
Turning off green - Turning on red
Turning off red - Turning on red
Turning off red - Turning on orange
Turning off orange - Turning on green
Turning off green - Turning on green
Turning off green - Turning on orange
Turning off orange - Turning on blue
Turning off blue - Turning on red
Thanks for playing!
```

■ **Tip** The LCD comes with a thin protector on it. You will want to remove that and use a stylus or similar soft pointer to test the script. If your fingers are large, you may have some difficulty touching the small area designated.

If you decide to try out the example yourself, you should find it gives you a few moments of satisfaction of achieving an interesting doodad. If you like this example and want to get an LCD for your own projects, see the MicroPython store to order one (https://store.micropython.org/store/#/products/LCD160CRv1_0H). You will find you can purchase one with headers or without. You may choose one without if you decide to get the nifty (optional) aluminum case, which requires mounting a different, angled header (you must order and solder them yourself). However, you can use the LCD with headers as shown in this example.

■ **Tip** See <http://micropython.org/resources/LCD160CRv10-refmanual.pdf> for the reference manual for the lcd160cr, which includes the low-level specification for working with the skin.

WiPy

The WiPy-specific libraries are unique in that they contain support for a complete range of Pycom boards including the Expansion Board, PySense, and PyTrack shields.

There are also libraries specific to the Pyboard itself in the form of the `pycom` library, which includes support for changing the heartbeat LED including changing the color. The AES library contains an interesting feature for securing data – AES encryption. We will explore this library in more detail. Also, recall that the WiPy firmware contains the low-level hardware for I2C, SPI, etc., in the machine library as opposed to the Pyboard, which has these in the `pyb` library.

The following sections briefly describe the libraries in more detail including a few commonly found examples of some of the features. See the indicated documentation for complete details of the contents and features of each library.

pycom

The `pycom` library has functions to control specific features of the Pycom devices, such as the heartbeat RGB LED. In fact, you can change the color of the LED, turn it on or off, and get the current state of the LED. While this seems very rudimentary, you can use the heartbeat LED to communicate status information. For example, you can change the color to indicate different actions (or states) such as reading from sensors, saving data, communicating errors, etc.

Let's look at a short example of using the heartbeat LED to set state. Note that the LED color is defined by a 24-bit value representing red, green, and blue (RGB), where the red color is represented by the most significant 8 bits, green the next 8 bits, and blue the least significant 8 bits. We normally represent these values in hexadecimal.¹ For example, bright white is `0xFFFFFF`, `0xFFFF00` is yellow, and `0x00FFFF` is a sea green.

In this example, we will use a set of helper functions and a test loop to work with the heartbeat LED. Note that to turn on (use) the heartbeat LED, you must first turn off the heartbeat internal feature with `heartbeat(False)`. Listing 6-2 shows the example code for working with the heartbeat LED to show status/state.

Listing 6-2. Using the Heartbeat LED for Status/State (WiPy)

```
# MicroPython for the IOT - Chapter 6
# Example for working with the heartbeat LED as a
# status/state indicator
#
import utime

# First, make sure this is running on a Pyboard
try:
    import pycom
except ImportError:
    print("ERROR: not on a WiPy (or Pycom) board!")
    sys.exit(-1)
```

¹<https://en.wikipedia.org/wiki/Hexadecimal>

```

# State/status enumeration
_STATES = {
    'ERROR' : 0xFF0000, # Bright red
    'READING' : 0x00FF00, # Green
    'WRITING' : 0x0000FF, # Blue
    'OK' : 0xFF33FF, # Pinkish
}

# Clear the state (return to normal operation)
def clear_status():
    pycom.heartbeat(True)

# Show state/status with the heartbeat LED
def show_status(state):
    pycom.heartbeat(False)
    pycom.rgbled(state)

# Now, demonstrate the state changes
for state in _STATES.keys():
    show_status(_STATES[state])
    utime.sleep(3)

# Return heartbeat to normal
clear_status()

```

If you run this code, you will notice the LED will present some very bright colors. So, don't stare directly at the LED! You can try reducing the brightness as an exercise using some more subtle color values.

■ **Tip** See the online calculator at http://www.rapidtables.com/web/color/RGB_Color.htm to determine the desired color.

AES

WiPy has a special library named `crypto`, which has the AES library with support for AES (Advanced Encryption Standard) - a symmetric block cipher standardized by NIST. While you may not need this feature, for those of us concerned about protecting sensitive data, this could be a nifty feature worthy of consideration.

For example, you may want to transmit data over the Internet to another system. If you do not protect the data with encryption, it is not too difficult to decipher the data should it be exploited in some manner. If you encrypt the data (and protect the keys!), you can make it much more difficult for people to see the data.

■ **Note** Encryption is an advanced feature that most beginners will not use. Thus, this section is brief. If you want to learn more about this library, see <https://docs.pycom.io/chapter/firmwareapi/pycom/aes.html>.

The AES library offers several constants to control the encryption mode and two functions: one to encrypt and another to decrypt. You will need to provide a key to use in the encryption algorithm for the simplest form of encryption defined by the constant, `AES.MODE_ECB`, which means Electronic Code Book. Other forms may need additional parameters.

There is one caveat. The data you encrypt must be in multiples of 16 byte blocks. Notice how we use spaces to make the strings 16 bytes. So, if you're encrypting data that you've created or read from a sensor or another node on your network, will need to ensure you encrypt with blocks in multiple of 16 bytes.

Let's look at an example. In this example, we will create a file with encrypted data to show how to protect your data for saving or sending to another node (computer). We will then open the file and read the data decrypting it so that we can show how to decipher encrypted data. We will use strings to make it easier but you can encrypt binary data too. Listing 6-3 shows the example code to use AES encryption to protect data.

Listing 6-3. Using Encryption to Protect Data (WiPy)

```
# MicroPython for the IOT - Chapter 6
# Simple example for working with encrypting data
#
import sys

# First, make sure this is running on a WiPy (pycom)
try:
    import pycom
    from crypto import AES
except ImportError:
    print("ERROR: not on a WiPy (or Pycom) board!")
    sys.exit(-1)

# Setup encryption using simple, basic encryption
# NOTICE: you normally would protect this key!
my_key = b'monkeybreadyumy' # 128 bit (16 bytes) key
cipher = AES(my_key, AES.MODE_ECB)

# Create the file and encrypt the data
new_file = open("secret_log.txt", "w") # use "write" mode
new_file.write(cipher.encrypt("1,apples,2.5 \n")) # write some data
new_file.write(cipher.encrypt("2,oranges,1 \n")) # write some data
new_file.write(cipher.encrypt("3,peaches,3 \n")) # write some data
new_file.write(cipher.encrypt("4,grapes,21 \n")) # write some data
new_file.close() # close the file
```

```
# Step 2: Open the file and read data
old_file = open("secret_log.txt", "r") # use "read" mode
# Use a loop to read all rows in the file
for row in old_file.readlines():
    data = cipher.decrypt(row).decode('ascii')
    columns = data.strip("\n").split(",") # split row by commas
    print(" : ".join(columns)) # print the row with colon separator

old_file.close()
```

The code is straightforward and you should have no problem following it. Notice, however, that we combine the write method and the encrypt call to make things a bit shorter. You can do it in two statements using an intermediate variable if you'd like.

When this program runs, it creates a file named `secret_log.txt` on the flash drive on your WiPy. If we examine this file with a hex dumping utility, we can see it is indeed encrypted (we cannot read the data since it was scrambled). The following shows what the file looks like in a hex dump.

```
$ hexdump -C secret_log.txt
00000000 c1 02 97 87 74 28 4f 4e de 83 8d 8d 49 4a f8 93 |....t(ON....IJ..|
00000010 c9 e7 f8 00 f3 ba e2 f8 7c 6e ca 41 13 0c 09 35 |.....|n.A...5|
00000020 a6 83 f6 fc 2c de ba eb f6 3a af fe c0 b5 c6 ee |...,...:.....|
00000030 7a 3b 3a 36 90 da dc 36 3d 61 7e 31 75 a3 ca 96 |z;:6...6=a~1u...|
00000040
```

The code also prints out the strings read from the file to confirm it works. If you run the program via a REPL console, you should see the following output.

```
>>> import wipy_encryption
1 : apples : 2.5
2 : oranges : 1
3 : peaches : 3
4 : grapes : 21
```

Now that we've seen a few examples of the board-specific libraries, let us examine a few examples of working with the hardware features directly. We have seen some of these in previous examples but they were not explained in detail. The following sections take a deeper look at some of the more common low-level hardware access using the Pyboard and WiPy.

Low-Level Examples

Working with the low-level hardware (some would just say, ‘hardware’ or ‘device’) is where all the action and indeed the focus (and relative difficulty) of using MicroPython takes place. MicroPython and the breakout board vendors have done an excellent job of making things easier for us, but there is room for improvement in the explanations.

That is, the documentation online is a bit terse when it comes to offering examples of using the low-level hardware. Part of this is because the examples often require additional, specific hardware. For example, to work with the I2C interface, you will need an I2C capable breakout board. Thus, the online examples provide only the most basic of examples and explanations.

However, that doesn’t mean the rest of the documentation on the low-level hardware is lacking. In fact, the documentation provides an excellent overview of all the classes and functions at your disposal. The challenge then is taking that overview and applying it to solve your programming needs, which is one of the aims of this book!

Except for the on-board sensors that may exist or other components like LEDs, RTC, buttons, etc., most low-level communication will be through I2C, one-wire, analog or digital pins, or even SPI interfaces. The I2C and SPI interfaces are those where you will likely encounter the most difficulty working with hardware. This is because each device (breakout board) you use will require a very specific protocol. That is, the device may require a special sequence to trigger the sensor or features of the device that differs from other breakout boards. Thus, working with I2C or SPI (and some other) type devices can be a challenge to figure out exactly how to “talk” to them.

Drivers and Libraries to the Rescue!

Fortunately, there is a small but growing number of people making classes and sets of functions to help us work with those devices. These are called libraries or more commonly drivers and come in the form of one or more code modules that you can download, copy to your board, and import the functionality into your program. The developers of the drivers have done all the heavy lifting for you making it very easy to use the device.

Thus, for most just starting out with MicroPython wanting to work with certain sensors, devices, breakout boards, etc., you should limit what you plan to use to those that you can find a driver that works with it. So, how do you find a driver for your device? There are several places to look.

First and foremost, you should look to the forums and documentation on MicroPython. In this case, don’t limit yourself to only those forums that cater to your board of choice. Rather, look at all of them! Chances are, you can find a library that you can adapt with only minor modifications. Most of them can be used with very little or even no effort beyond downloading it and copying it to the board. The following lists the top set of forums and documentation you should frequent when looking for drivers.

- *MicroPython Forums*: <https://forum.micropython.org/>
- *MicroPython Documentation*: <https://docs.micropython.org/en/latest/pyboard/>
- *Pycom (WiPy) Forums*: <https://forum.pycom.io/>

- *Pycom (WiPy) Documentation*: <https://docs.pycom.io/>
- *BBC micro:bit MicroPython Forums*: <https://forum.micropython.org/viewtopic.php?t=1042>
- *BBC micro:bit MicroPython Documentation*: <https://microbit-micropython.readthedocs.io/en/latest/>
- *MicroPython Wiki (External docs)*: <http://wiki.micropython.org/Home>
- *Adafruit Learning*: <https://learn.adafruit.com/>

Notice the last entry. If you search for MicroPython, you will find many excellently presented (and complete) tutorials including a growing number of hardware topics.

Second, use your favorite Internet search engine and search for examples of the hardware. Use the name of the hardware device and “MicroPython” in your search. If the device is new, you may not find any hits on the search terms. Be sure to explore other search terms too.

Once you find a driver, the fun begins! You should download the driver and copy it to your board for testing. Be sure to follow the example that comes with the driver to avoid using the driver in an unexpected way.

Which calls to mind one important thing you should consider when deciding if you want to use the driver. If the driver is documented well and has examples – especially if the example includes your board – you should feel safe using it. However, if the driver isn’t documented at all or there is no or little sample code, don’t use it! There is a good chance it is half-baked, old, a work in progress, or just poorly coded. Not all those that share can share and communicate well.

Let’s look at two low-level examples: working with the real-time clock and callbacks through interrupts. We will begin with the real-time clock (RTC). These are only a small sample of what is available and represents the most common things you will need to work with for this book and most small IOT projects.

Real-Time Clock (RTC)

Most boards have a real-time clock (RTC). The RTC is a special circuit (sometimes an integrated circuit or chip) that keeps time. This is because most processors (microcontroller, microprocessor, etc.) operate in synchronization with a crystal or clock chip that keeps the processor running at a certain speed (e.g., Mhz). Sadly, this is often not divisible easily into a time variable (value). The RTC is used to keep time so that we can use the time to record events.

To use an RTC, we first initialize the starting value with the current date and time (like setting a new desktop clock) and we can read the current date and time whenever we want. However, a RTC without a battery backup will lose its values when the board is powered off. Thus, we must set it every time we start the board. Fortunately, there is a time service on the Internet that we can use to get the current date and time. It’s called the network time protocol (NTP).²

²https://en.wikipedia.org/wiki/Network_Time_Protocol

Let's see an example of how to use this service. We will create a program on the WiPy that connects the board to our local WiFi, which is connected to the Internet. Once connected, we will use the NTP to set the current time and then perform a test to see what the current date and time are. We should see the exact date and time when we run the code! Listing 6-4 shows the completed example.

Listing 6-4. Using an NTP Time Server to set the RTC (WiPy)

```
# MicroPython for the IOT - Chapter 6
# Example module for using the ntp time server to set the datetime
# Note: only works on WiPy!

from network import WLAN
from machine import RTC
import machine
import sys
import utime

wlan = WLAN(mode=WLAN.STA)

def connect():
    wifi_networks = wlan.scan()
    for wifi in wifi_networks:
        if wifi.ssid == "YOUR_SSID":
            wlan.connect(wifi.ssid, auth=(wifi.sec, "YOUR_SSID_PASSWORD"),
                timeout=5000)
            while not wlan.isconnected():
                machine.idle() # save power while waiting
            print("Connected!")
            return True
    if not wlan.isconnected():
        print("ERROR: Cannot connect! Exiting...")
        return False

if not connect():
    sys.exit(-1)

# Now, setup the RTC with the NTP service.
rtc = RTC()
print("Time before sync:", rtc.now())
rtc.ntp_sync("pool.ntp.org")
while not rtc.synced():
    utime.sleep(1)
    print("waiting for NTP server...")
print("Time after sync:", rtc.now())
```

Most of this code should be familiar since we've seen WiFi connect in a previous chapter. In this example, we place the code in a method to make it a bit easier to use.³ However, the use of the `RTC()` class is new. Notice all we need to do is, once the network connection is made, is to call the `ntp_sync()` method passing in the name of the NTP service. Yes, it's built into the library! Cool. After that, we need only wait until the RTC synchronizes with the NTP and we're good to go.

■ **Note** The Pyboard provides the `RTC()` class in the machine library but it is a bit different than the one in the WiPy firmware. You could modify this example as an exercise for use on your Pyboard.

When you run this on your WiPy, you will see output like the following. Notice we print the value of the time – which is the starting epoch – then print the time again once the RTC has synched with the NTP.

```
>>> import wipy_ntp
Connected!
Time before sync: (1970, 1, 1, 0, 0, 36, 560190, None)
waiting for NTP server...
waiting for NTP server...
waiting for NTP server...
Time after sync: (2017, 7, 13, 16, 19, 51, 402976, None)
```

This example can be very helpful and in some cases a must when reading data for later analysis. It is often crucial to know when the data was saved or sensor was read. You may want to earmark this code for later use in your IOT projects.

■ **Note** We can use a dedicated RTC module that has an on-board clock that keeps the clock synchronized when operating offline or during periods when the board is powered off. We will see how to do this in Chapter 8.

Now, let's look at callbacks, which are programming mechanisms you can use to work with hardware interrupts.

³Whenever you find yourself making functions for commonly used code, it is time to consider adding it to a library of your favorite functions to make the code available for all your programs as we discussed in Chapter 5.

Callbacks

What do you do if you want to have some bit of code execute in reaction to a sensor or user input? Using what we've learned thus far, we could write our program with a loop to poll the sensor or user actionable device (such as a button) and when triggered, execute the code. This polling technique will work, but there is a better construct called callbacks.

Callbacks are functions we define and associate with the firmware to execute when a certain event occurs. If a hardware abstraction permits the use of a callback, we can use that. Fortunately, the `Switch` class in the Pyboard firmware (`pyb.Switch`) has such a mechanism. We could also use hardware interrupts in much the same way. However, hardware interrupts are an advanced topic. Let's work with the `Switch` class to keep things easier.

The use of callbacks allows us to continue executing code to do work such as reading sensors, displaying data, etc., and when the event (interrupt) occurs, MicroPython will execute the callback function and then return to executing our code. This works by tying the callback function to interrupts. The `switch` class has an interrupt defined for the button press. That is, when the button is pressed, the callback fires (executes).

There are some caveats to using the switch callback. First, the switch callback function cannot take parameters so you cannot define a callback function and pass it any data. In fact, callback functions are not permitted to create data. For example, you cannot create a dictionary, tuple, etc., inside the function. While callback functions can access global variables, they may throw an exception if you use state variables. Finally, you can turn off (disconnect) the callback. For example, you can disconnect the callback for the switch with `switch.callback(None)`.

Now, let's see an example. In this example, we want to create a callback to cycle through the LEDs on the board. With each press of the button, another LED is lit until we cycle through all the LEDs then start over. This means we need to save the last LED lit or the state of the LEDs. To do this, we can use a class that has a local variable, which we access within the callback function.

Setting up the callback is easy. We just call the callback function for the switch and pass in the name of the function. We do this through the constructor of the class. That is, when we create a new instance of the class, we pass in the `pyb.Switch` object and then call the callback on that object passing in the name of the class function.

Let's see the code and you'll see how this works. Listing 6-5 shows the completed code for the callback example for the Pyboard.

Listing 6-5. Callback Example (Pyboard)

```
# MicroPython for the IOT - Chapter 6
# Simple example for working with interrupts using a class
# to store state.
#
import sys
```

```

# First, make sure this is running on a Pyboard
try:
    import pyb
except ImportError:
    print("ERROR: not on a Pyboard!")
    sys.exit(-1)

# Initiate the switch class
switch = pyb.Switch()

class Cycle_LED(object):
    # Constructor
    def __init__(self, sw):
        self.cur_led = 0
        sw.callback(self.do_switch_press)

    # Switch callback function
    def do_switch_press(self):#
        # Turn off the last led unless this is the first time through the cycle
        if self.cur_led > 0:
            pyb.LED(self.cur_led).off()
        if self.cur_led < 4:
            self.cur_led = self.cur_led + 1
        else:
            self.cur_led = 1
        # Turn on the next led
        pyb.LED(self.cur_led).on()

# Initiate the Cycle_LED class and setup the callback
cycle = Cycle_LED(switch)

# Now, simulate doing something else
print("Testing the switch callback. Press CTRL-C to quit.")
while True:
    sys.stdout.write(".") # Print something to show we're still in the loop
    pyb.delay(1000)      # Wait a second...

```

Notice the class we created and how it sets up the callback function for the switch. Notice also how we instantiate the class passing in the switch instance. After that, we set up a simple loop to print out dots until the program is stopped with CTRL-C. This demonstrates that the program continues to run even when the button is pressed.

If you're using the Pyboard, give this example a test. As you will see, using callbacks are quite powerful.

■ **Tip** See https://docs.micropython.org/en/latest/pyboard/reference/isr_rules.html if you want to learn how to write your own interrupt handlers.

COMMUNICATION VIA BLUETOOTH LOW ENERGY (BLE)

Another feature of the WiPy that is interesting is that it comes with Bluetooth Low Energy (BLE) and the firmware supports it as a network class. While you can use a BLE module with other boards via UART (serial communication) – see <https://github.com/dmazzella/uble> for an advanced and complete example, the BLE feature of the WiPy makes it possible to create read/write services that you can use to communicate small packets of data: for example, to sample a sensor and send the data over BLE.

Sadly, the Bluetooth support in the firmware is still very new and is not complete enough for those new to MicroPython (and BLE). Thus, it is too advanced for a beginning book. In fact, the documentation is currently very terse and the examples are difficult to follow. As more people use it and find inventive ways to apply it, you will likely see the documentation enhanced as new features are added and existing features are added. Check the Pycom online documentation for updates to the Bluetooth support (<https://docs.pycom.io/chapter/firmwareapi/pycom/network/bluetooth/>).

Now, let's look at how to communicate with breakout boards using the I2C and SPI protocols.

Using Breakout Boards

We looked at breakout boards briefly in Chapter 3 where we saw an interesting example of using an Arduino shield to connect a Pyboard to a wireless network. That is just one example of a breakout board. You will also find breakout boards that host sensors and other devices that can make your IOT project easier. The trick in using those breakout boards is finding the correct, working driver.

Recall there are two methods for working with breakout boards: finding a driver you can use, or building your own driver. Building your own driver is not recommended for those new to MicroPython and I2C or SPI. It is much easier to take the time to search for a driver that you can use (or adapt) than to try to write one yourself. This is because you must be able to obtain, read, and understand how the breakout board communicates (understand its protocol). Each board will communicate differently based on the sensor or devices supported. That is, a driver for a BMP180 sensor will not look or necessarily work the same as one for a BME280 sensor. You must be very specific when locating and using a driver.

Searching for a driver can be a tedious endeavor, which requires some patience and perhaps several searches on the forums using different search terms such as “micropython BME280.” Once you find a driver, you can tell quickly whether it is a viable option by looking at the example included. As mentioned before, if there is no example or the example doesn't resemble anything you've seen in this book or in the online documentation, don't use it.

■ **Tip** To find a driver, visit <https://forum.micropython.org/viewforum.php?f=14> and search for your board. For example, if you have a BME280 breakout board, use that as the search term. Using precise terms like “Vendor XYZ Model#40113” may be too specific. Try using the general name of the sensor/device first.

Let’s look at two examples of breakout boards: one that uses the I2C protocol, and another that uses the SPI protocol.

ONLINE EXAMPLES

If you want to use a breakout board in your IOT project, be sure to spend some time not only in the forums but also looking at various blogs and tutorials such as those on hackaday.com, learn.sparkfun.com, or learn.adafruit.com. The best blogs and tutorials are those that explain not only how to write the code but also what the breakout board does and how to use it. These online references are few but the ones from these three sites are among the very best. Also, look at some of the videos on the topic too. Some of those are worth the time to watch – especially if they’re from the nice folks at Adafruit or Sparkfun.⁴

Inter-Integrated Circuit (I2C)

The I2C protocol is perhaps the most common protocol that you will find on breakout boards. We’ve encountered this term a few times in previous chapters and thus we only know it is a communication protocol. I2C is a fast-digital protocol using two wires (plus power and ground) to read data from circuits (or devices). The protocol is designed to allow the use of multiple devices (slaves) with a single master (the MicroPython board). Thus, each I2C breakout board will have its own address or identity that you will use in the driver to connect to and communicate with the device.

Let’s look at an example of how to use an I2C breakout board. In this example, we want to use an RGB sensor from Adafruit (<https://www.adafruit.com/product/1334>) to read the color of objects. Yes, you can make your MicroPython board see in color! We will use this breakout board with a WiPy.

Don’t worry if you do not have or do not want to purchase the Adafruit RGB Sensor breakout board (although it is not expensive). This example is provided as a tutorial for working with I2C breakout boards. We will use another I2C breakout board in one of the example projects later in the book. Figure 6-2 shows the Adafruit RGB Sensor.

⁴No, I am not affiliated with either of these – merely a devoted fan and frequent return customer. Check them out!



Figure 6-2. Adafruit RGB Sensor (courtesy of adafruit.com)

Wiring the breakout board is also very easy since we need only power, ground, SCL and SDA connections. SCL is the clock signal, and SDA is the data signal. These pins are labeled on your MicroPython board (or in the documentation) as well as the breakout board. When you connect your breakout board, make sure the power requirements match. That is, some breakout boards can take 5V but many are limited to 3 or 3.3V. Check the vendor’s website if you have any doubts.

We need only to connect the 3V, ground, SDA, SCL, and LED pins. The LED pin is used to turn on the bright LED on the breakout board to signal it is ready to read. We will leave it on for 10 seconds so that there is time to read the color value then display it. We will then wait another 5 seconds to take the next reading.

To connect the wires, you can use five male-to-female jumper wires to plug into the WiPy (or Expansion Board) and the breakout board. Figure 6-3 shows the connections you need to make. Notice the wires are plugged into pins marked as “GXX” rather than “PinXX.” This can be a source of confusion when working with boards. It is always best to refer to the pinout drawings to ensure you’re using the right pins. In this case, we need P9 and P10 for the I2C connection and we will use P8 for the LED.

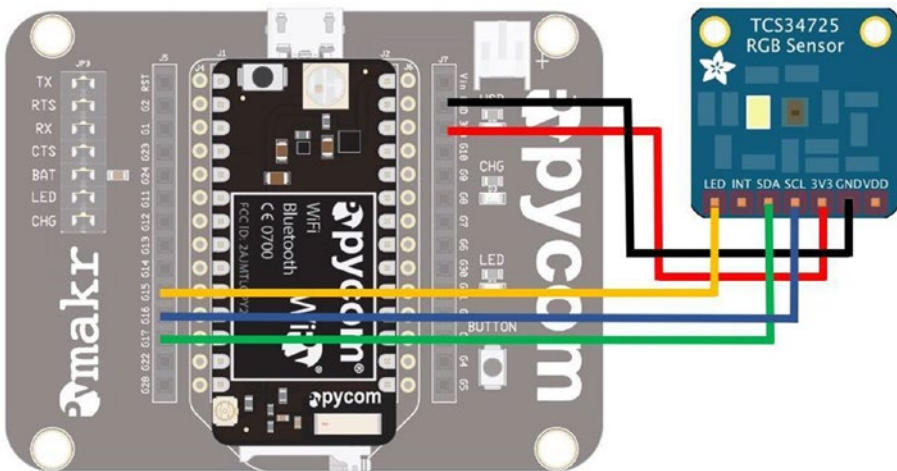


Figure 6-3. Wiring the RGB Sensor (WiPy)

■ **Tip** See <https://docs.pycom.io/chapter/datasheets/development/wipy2.html> for the pinouts for the WiPy board and <https://docs.pycom.io/chapter/datasheets/boards/expansion.html> for the Expansion Board. Note that the Expansion board “mirrors” the WiPy, but its labels are completely different.

Once you have the hardware connected, set it aside. We need to download the driver and copy it to the board before we can experiment further. You can find the driver at <https://github.com/adafruit/micropython-adafruit-tcs34725>. This is a fully working, tested driver that demonstrates how easy it is to use an I2C breakout board. Don't worry about the internals of the library.⁵ The code to which I'm referring is shown below. Notice the default address is 0x29 but since address is a parameter, you can override it if you have another breakout board for the same RGB sensor that is at a different address. This means you can use more than one with the same driver.

```
class TCS34725:
    def __init__(self, i2c, address=0x29):
        self.i2c = i2c
        self.address = address
        self._active = False
        self.integration_time(2.4)
        sensor_id = self.sensor_id()
        if sensor_id not in (0x44, 0x10):
            raise RuntimeError("wrong sensor id 0x{:x}".format(sensor_id))
    ...
```

To download the driver, you first navigate to <https://github.com/adafruit/micropython-adafruit-tcs34725> and then click the Download button and then the Download Zip button. Once the file has downloaded, unzip it. In the resulting folder, you should find the file named `tcs34725.py`. This is the driver code module. When ready, we will copy it to our WiPy using FTP as shown below. Be sure to connect to your WiPy Wi-Fi network first and open a terminal in the same directory as the file.

```
$ ftp 192.168.4.1
Connected to 192.168.4.1.
220 Micropython FTP Server
Name (192.168.4.1:cbell): micro
Password:
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd flash
ftp> put tcs34725.py
```

⁵But feel free if you are curious. If you do, you will see an interesting bit of code that demonstrates the addressability of I2C devices.

```

local: tcs34725.py remote: tcs34725.py
227 (192,168,4,1,7,232)
100% |*****| 5222      23.05 MiB/s      00:00 ETA
5222 bytes sent in 00:00 (6.57 KiB/s)
ftp> quit

```

Now that the driver is copied to our board, we can write the code. In this example, we will set up the I2C connection to the breakout board and run a loop to read values from the sensor. Sounds simple, but there is a bit of a trick to it. We will forego a lengthy discussion of the code and instead offer some key aspects allowing you to read the code yourself to see how it works.

The key components are setting up the I2C, sensor, a pin for controlling the LED, and reading from the sensor. The LED on the board can be turned on and off by setting a pin high (on) or low (off). First, the I2C code is as follows. Here, we initiate an object, then call the `init()` function setting the bus to master mode. The `scan()` function prints out the devices on the bus. If you see an empty set displayed, your I2C wiring is not correct. Check it and try the code again. Note that you can run this code manually once you've done the imports.

```

i2c = I2C(0, I2C.MASTER)           # create and init as a master
i2c.init(I2C.MASTER, baudrate=20000) # init as a master
i2c.scan()

```

The next part is the sensor itself. The driver makes this easy. All we need to do is pass in the I2C constructor function as shown.

```

# Setup the sensor
sensor = tcs34725.TCS34725(i2c)

```

Setting up the LED pin is also easy. All we need to do is call the `Pin()` class constructor passing in the pin name (P8) and setting it for output mode as follows.

```

# Setup the LED pin
led_pin = Pin("P8", Pin.OUT)
led_pin.value(0)

```

Finally, we read from the sensor with the `sensor.read()` function passing in `True`, which tells the driver to return the RGB values. We will then print these out in order. Listing 6-6 shows the completed code. Take a few moments to read through it so that you understand how it works.

Listing 6-6. Using the Adafruit RGB Sensor (WiPy)

```

# MicroPython for the IOT - Chapter 6
# Example of using the I2C interface via a driver
# for the Adafruit RGB Sensor tcs34725
#

```

```

# Requires library:
# https://github.com/adafruit/micropython-adafruit-tcs34725
#
from machine import I2C, Pin
import sys
import tcs34725
import utime

# Method to read sensor and display results
def read_sensor(rgb_sense, led):
    sys.stdout.write("Place object in front of sensor now...")
    led.value(1)          # Turn on the LED
    utime.sleep(5)       # Wait 5 seconds
    data = rgb_sense.read(True) # Get the RGBC values
    print("color detected: {")
    print("    Red: {0:03}".format(data[0]))
    print("   Green: {0:03}".format(data[1]))
    print("    Blue: {0:03}".format(data[2]))
    print("   Clear: {0:03}".format(data[3]))
    print("}")
    led.value(0)

# Setup the I2C - easy, yes?
i2c = I2C(0, I2C.MASTER)          # create and init as a master
i2c.init(I2C.MASTER, baudrate=20000) # init as a master
i2c.scan()

# Setup the sensor
sensor = tcs34725.TCS34725(i2c)

# Setup the LED pin
led_pin = Pin("P8", Pin.OUT)
led_pin.value(0)

print("Reading Colors every 10 seconds. When LED is on, place object in
front of sensor.")
print("Press CTRL-C to quit. (wait for it)")
while True:
    utime.sleep(10)          # Sleep for 10 seconds
    read_sensor(sensor, led_pin) # Read sensor and display values

```

Once you have the code, you can copy it to your board in the similar manner we did for the driver with the ftp utility as shown below.

```

$ ftp 192.168.4.1
Connected to 192.168.4.1.
220 Micropython FTP Server
Name (192.168.4.1:cbell): micro

```

```

Password:
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd flash
ftp> put wipy_RGB.py
local: wipy_RGB.py remote: wipy_RGB.py
227 (192,168,4,1,7,232)
100% |*****| 1202          4.51 MiB/s    00:00 ETA
1202 bytes sent in 00:00 (2.17 KiB/s)
ftp> quit

```

All that is left is running the example and testing it. Listing 6-7 shows how you can run the example on the WiPy as well as a sample of the output. If you are running this example on your WiPy, you can place whatever object you want in front of the sensor, and it will read the color returning it as a tuple representing the RGB value plus clear values as shown.

Listing 6-7. Output from using the Adafruit RGB Sensor (WiPy)

```

>>> import wipy_RGB
Reading Colors every 10 seconds. When LED is on, place object in front of sensor.
Press CTRL-C to quit. (wait for it)
Place object in front of sensor now...color detected: {
    Red: 057
    Green: 034
    Blue: 032
    Clear: 123
}
Place object in front of sensor now...color detected: {
    Red: 054
    Green: 069
    Blue: 064
    Clear: 195
}
Place object in front of sensor now...color detected: {
    Red: 012
    Green: 013
    Blue: 011
    Clear: 036
}
...

```

If you wanted another exercise, you could take these values from the sensor and map them to an RGB LED. Yes, you can do that! Go ahead, try it. See <https://github.com/JanBednarik/micropython-ws2812> for inspiration. Tackle it after you've read the next section on SPI.

■ **Tip** See <https://learn.sparkfun.com/tutorials/i2c> for an in-depth discussion of I2C.

Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) is designed to allow sending and receiving data between two devices using a dedicated line for each direction. That is, it uses two data lines along with a clock and a slave select pin. Thus, it requires six connections for bidirectional communication or only five for reading or writing only. Some SPI devices may require a seventh pin called a reset line.

Let's look at an example of how to use a SPI breakout board. In this example, we want to use the Adafruit Thermocouple Amplifier MAX31855 breakout board (<https://www.adafruit.com/product/269>) and a Thermocouple Type-K sensor (<https://www.adafruit.com/product/270>) to read high temperatures. We will use this breakout board with a Pyboard.

Don't worry if you do not have or do not want to purchase the Adafruit Thermocouple Amplifier MAX31855 breakout board (although it is not expensive). This example is provided as a tutorial for working with SPI breakout boards. We will use another I2C breakout board in one of the example projects later in the book. Figure 6-4 shows the Adafruit Thermocouple Amplifier and Type-H sensor from Adafruit.

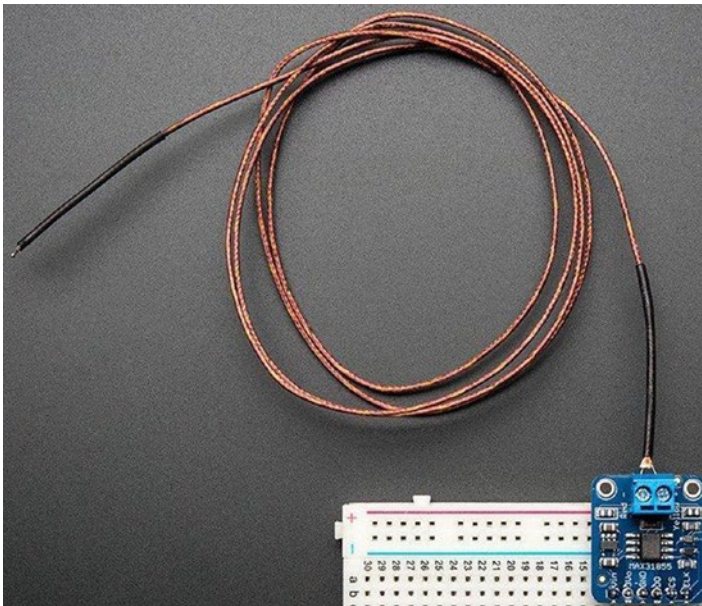


Figure 6-4. Adafruit Thermocouple Breakout Board and Type-K Sensor (courtesy of adafruit.com)

The sensor can be used to measure high temperatures either through proximity or touch. The sensor can read temperature in the range -200°C to +1350°C output in 0.25 degree increments. One possibly use of this sensor is to read the temperature of nozzles on 3D printers or any similar high heat output. It should be noted that the breakout board comes unassembled so you will need to solder the header and terminal posts.

Now, let’s see how to wire the breakout board to our Pyboard. We will use only five wires since we are only reading data from the sensor on the breakout board. This requires a connection to power (ground (GND), the master input (MOSI), clock (CLK), and slave select (SS). Figure 6-5 shows the connections.

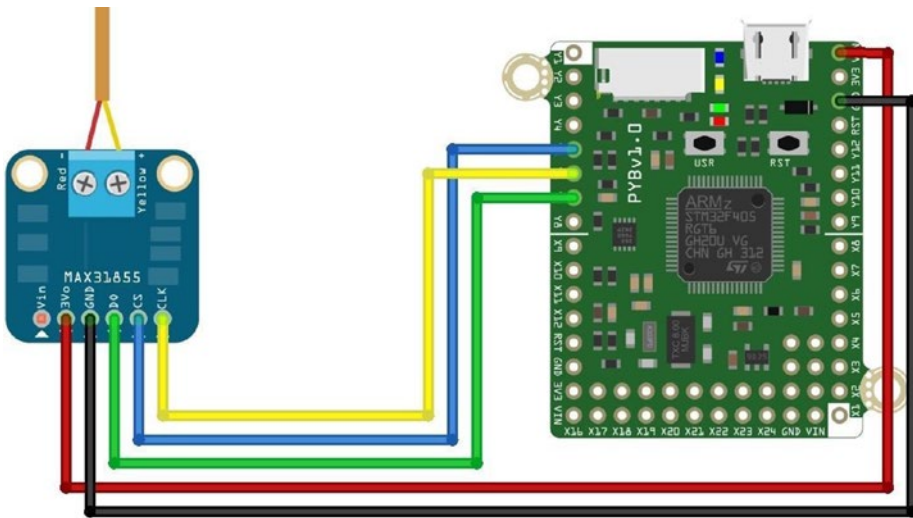


Figure 6-5. Wiring the Adafruit Thermocouple Module (Pyboard)

The correct pins to use on the Pyboard are shown in Table 6-2. You can find these pins on the Pyboard pinout found at <https://docs.micropython.org/en/latest/pyboard/pyboard/quickref.html>. Refer to Figure 6-5 for confirmation.

Table 6-2. Connecting the Thermocouple Breakout Board (Pyboard)

Pyboard	Location on Pyboard	Breakout board
VIN	upper rightmost pin	3V0
GND	on right side, third pin down from the top	GND
Y5	on left side of the board, fifth pin from top	CS (slave select)
Y6	on left side of the board, sixth pin from top	CLK
Y7	on left side of the board, seventh pin from top	D0

Now, let's look at the code! In this example, we are not going to use a driver; rather, we're going to see how to read directly from the breakout board using SPI. To do so, we first set up an object instance of the SPI interface then choose a pin to use for slave select (also called chip or even code select). From there, all we need to do is read the data and interpret it. We will read the sensor in a loop and write a function to convert the data.

This is the tricky part. This example shows you what driver authors must do to make using the device easier. In this case, we must read the data from the breakout board and interpret it. We could just read the raw data, but that would not make any sense since it is in binary form. Thus, we can borrow some code from Adafruit that reads the raw data and makes sense of it.

The function is named `normalize_data()` and it does some bit shifting and arithmetic to transform the raw data to a value in Celsius. This information comes from the datasheet for the breakout board but the nice folks at Adafruit made it easy for us.

Setting up the SPI class is easy. We initiate an SPI object using the class constructor passing in the SPI option. This is unique to the Pyboard and can be values of 1 – use the X position, or 2 – use the Y position. Notice in the connections above we use the Y-pins so we will use a value of 2 for the first parameter. The other parameters tell the SPI class to set up as a master, set the baudrate, polarity, and phase (which can be found on the datasheet). Next, we need only select the pin for reading data and then set the pin high. The following shows the code we need to activate the SPI interface.

```
spi = SPI(2, SPI.MASTER, baudrate=5000000, polarity=0, phase=0)
cs = machine.Pin("Y5", machine.Pin.OUT)
cs.high()
```

Now, let's look at the completed code. Listing 6-8 shows the complete code to use the Thermocouple Amplifier Breakout board from Adafruit.

Listing 6-8. The Adafruit Thermocouple Module Example (Pyboard)

```
# MicroPython for the IOT - Chapter 6
# Simple example for working with the SPI interface
# using the Adafruit Thermocouple Amplifier. See
# https://www.adafruit.com/product/269.
#
# Note: this only runs on the Pyboard
#
import machine
import ubinascii

# First, make sure this is running on a Pyboard
try:
    import pyb
    from pyb import SPI
except ImportError:
    print("ERROR: not on a Pyboard!")
    sys.exit(-1)
```

```

# Create a method to normalize the data into degrees Celsius
def normalize_data(data):
    temp = data[0] << 8 | data[1]
    if temp & 0x0001:
        return float('NaN')
    temp >>= 2
    if temp & 0x2000:
        temp -= 16384
    return (temp * 0.25)

# Setup the SPI interface on the "Y" interface
spi = SPI(2, SPI.MASTER, baudrate=5000000, polarity=0, phase=0)
cs = machine.Pin("Y5", machine.Pin.OUT)
cs.high()

# read from the chip
print("Reading temperature every second.")
print("Press CTRL-C to stop.")
while True:
    pyb.delay(1000)
    cs.low()
    print("Temperature is {0} Celsius.".format(normalize_data(spi.recv(4))))
    cs.high()

```

At this point, you can make the hardware connections and power on your Pyboard. Then, you can copy the file to your Pyboard and execute it as shown below.

```

>>> import pyboard_SPI
Reading temperature every second.
Press CTRL-C to stop.
Temperature is 32.0 Celsius.
Temperature is 31.75 Celsius.
Temperature is 32.0 Celsius.
Temperature is 32.5 Celsius.
Temperature is 33.5 Celsius.
Temperature is 34.0 Celsius.
Temperature is 34.25 Celsius.
Temperature is 34.5 Celsius.
Temperature is 34.5 Celsius.
...

```

Once you run the example, you should see it produce values in degrees Celsius. If you see 0.00, you likely do not have the SPI interface connected properly. Check your wiring against the figure above. If you see values but they go down when you expose the thermocouple tip to heat, you need to reverse the wires. Be sure to power off the board first to avoid damaging the sensor, breakout board, or your Pyboard!

If you would like to run this example with the WiPy, you can! Simply alter the code to use the WiPy SPI class and its initialization sequence as shown at <https://docs.pycom.io/chapter/firmwareapi/pycom/machine/SPI.html>. You will also have to change the includes a bit, but this is good practice for the example project chapters.

■ **Tip** See <https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi> for an in-depth discussion of SPI.

Summary

Accessing the low-level hardware through the firmware is where the true elegance and in some cases complexity of using MicroPython begins. Given that the available boards differ, it should be no surprise that the low-level support in the firmware also differs. Thus, when planning MicroPython IOT projects, we must consider what we want to do and whether our board (and firmware) supports it. We also need to know what breakout boards and devices we want to connect to and if there are drivers or other libraries we can use to access them. In this case, most breakout boards with I2C or SPI interfaces will require some form of driver.

In this chapter, we explored some of the low-level support in the firmware and specialized support for the Pyboard and WiPy. As we discovered, this is where the code becomes very specialized. As we saw, it sometimes is a matter of choosing a different library to import, but sometimes the classes, functions, and even how to use the functions differs from one board to another.

We also saw a lot of code in this chapter – more than any previous chapter. The examples in this chapter are meant to be examples for you to see how things are done rather than projects to implement on your own (although you're welcome and encouraged to do so). We will see more hands-on projects with a greater level of detail in later chapters.

In the next chapter, we take a short detour in the form of a short tutorial on electronics. If you've never worked with electronics before, the next chapter will give you the information you need to complete the projects in this book and prepare you for an exciting new hobby – building MicroPython IOT projects!

CHAPTER 7



Electronics for Beginners

If you are new to working with hardware and have little or no experience with electronics, you may be curious as to how you can complete the projects in this book. Fortunately, the projects in this book walk you through how to connect the various electronic parts together with your MicroPython board. That is, you can complete the projects without additional skill or experience.

However, if you want to know what the components do, you will need a bit more information than “plug this end in here.” This is especially so if something goes wrong. Furthermore, if you want to create projects on your own, you need to know enough about how the components work to successfully complete your project – whether that is completing the examples in this book or examples found elsewhere on the Internet.

Fortunately, you don’t need formal training or even a college degree in theory to learn how to work with electronics. You can learn quite a lot about working with electronics at the hobbyist level without devoting months or years of research.¹ To ensure success even at a basic level, you will need to know more than simply how to plug the components together.

Rather than attempt to present a comprehensive tutorial on electronics, which would take several volumes, this chapter presents an overview of electronics for those who want to work with the types of electronic components commonly found in IOT projects. I include an overview of some of the basics, descriptions of common components, and a look at sensors. If you are new to electronics, this chapter will give you the extra boost you need to understand the components used in the projects in this book.

If you have experience with electronics either at the hobbyist or enthusiast level or have experience or formal training in electronics, you may want to skim this chapter or read the sections with topics on which you may want a refresher.

Let’s begin with a look at the basics of electronics. Once again, this is in no way a tutorial that covers all there is to know, but it will get you to the point where the projects make sense in how they connect and use components.

¹However, there is no substitute for formal training! If you want to explore electronics beyond the tutorial in this chapter, you may want to consider formal training or even a self-paced course as described in the sidebar, “I Want to Learn More!”

The Basics

This section presents a short overview of some of the most common tools and techniques you will need to use when working with electronics. As you will see, you only need the most basic of tools and the skills or techniques are not difficult to learn. However, before we get into those, let's look at some of the tools you will need to work on your IOT projects.

Tools

The clear majority of tools you will need to construct your IOT projects are common hand tools (screwdrivers, small wrenches, pliers, etc.). For larger projects or for creating enclosures you may need additional tools such as power tools, but I will concentrate only on those tools for building the projects. The following is a list of tools I recommend to get you started.

- Breadboard
- Breadboard wires (also called jumpers)
- Electrostatic discharge (ESD) safe tweezers
- Helping hands or printed circuit board (PCB) holder
- Multimeter
- Needle-nose pliers
- Screwdrivers – assorted sizes (micro, small)
- Solder
- Soldering iron
- Solder remover (solder sucker)
- Tool case, roll, or box for storage
- Wire strippers

However, you cannot go wrong if you prefer to buy a complete electronics toolset such as those from Sparkfun (sparkfun.com/categories/47) or Adafruit (adafruit.com/categories/83). You can often find electronics kits at major brand electronics stores and home improvement centers. If you are fortunate enough to live near a Fry's Electronics, you can find just about any electronics tool made. Most electronics kits will have all the hand tools you will need. Some even come with a multimeter but more often you must buy them separately.

Most of the tools in the list do not need any explanation except to say you should purchase the best tools that your budget permits. The following paragraphs describe some of the tools that are used for special tasks such as stripping wires, soldering, and measuring voltage and current.

Multimeter

A multimeter is one of those tools that you will need when building IOT solutions. You will also need it to do almost any electrical repair on your circuits. There are many different multimeters available with prices ranging from inexpensive, basic units; to complex, feature-rich, incredibly expensive units. For most IOT projects, including most IOT kits, a basic unit is all you will need. However, if you plan to build more than one IOT solution or want to assemble your own electronics, you may want to invest a bit more in a more sophisticated multimeter. Figure 7-1 shows a basic digital multimeter (costing about \$10) on the left and a professional multimeter from BK Precision on the right.

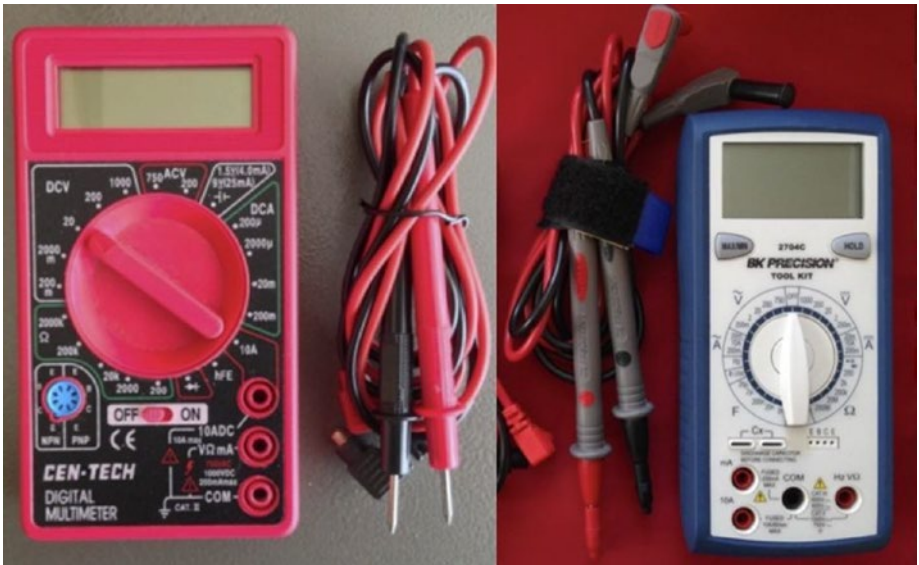


Figure 7-1. Digital multimeters

Notice the better meter has more granular settings and more features. Again, you probably won't need more than the basic unit. You will need to measure voltage, current, and resistance at a minimum. Whichever meter you buy, make sure it has modes for measuring AC and DC voltage, continuity testing (with an audible alert), and checking resistance. I will explain how to use a multimeter in a later section.

■ **Tip** Most multimeters including the inexpensive ones come with a small instruction booklet that shows you how to measure voltage, resistance, and other functions of the unit.

Soldering Iron

A soldering iron is not required for any of the projects in this book because we will use a breadboard to lay out and connect the components. However, if you plan to build a simple IOT solution where you will need to solder wires together, or maybe a few connectors, a basic soldering iron from an electronics store such as Radio Shack is all you will need. On the other hand, if you plan to assemble your own electronics, you may want to consider getting a good, professional soldering iron such as a Hakko. The professional models include features that allow you to set the temperature of the wand, have a wider array of tips available, and tend to last a lot longer. Figure 7-2 shows a well-used entry-level one from Radio Shack. Figure 7-3 shows a professional model Hakko soldering iron.



Figure 7-2. *Entry-level soldering iron*



Figure 7-3. Professional soldering iron

■ **Tip** For best results, choose a solder with a low lead content in the 37%–40% range. If you use a professional soldering iron, adjust the temperature to match the melting point of the solder (listed on the label).

DO I NEED TO LEARN TO SOLDER?

If you do not know how to solder or it has been a while since you've used a soldering iron, you may want to check out the book *Learn to Solder* by Brian Jepson, Tyler Moskowite, and Gregory Hayes (O'Reilly Media, 2012) or Google how-to videos on soldering. Or you could buy the Getting Started Soldering Kit from Maker Shed (makershed.com/products/make-getting-started-kit-soldering), which comes with a soldering iron, wire cutters, supplies, and more – everything you need to learn how to solder.

Wire Strippers

There are several types of wire strippers. In fact, there are probably a dozen or more designs out there. But there are two kinds: ones that only grip and cut the insulation as you pull it off the wire; and those that grip, cut, and remove the insulation. The first type is more common and, with some practice, does just fine for most small jobs (like repairing a broken wire); but the second type makes a larger job — such as wiring electronics from bare wire (no prefab connectors) — much faster. As you can imagine, the first type is considerably cheaper. Figure 7-4 shows both types of wire strippers. Either is a good choice.



Figure 7-4. Wire strippers

Helping Hands

There is one other tool that you may want to get, especially if you need to do any soldering: it's called helping hands or third hand tool. Most have a pair of alligator clips to hold wires, printed circuit boards, or components while you solder. Figure 7-5 shows an example of a simple helping hands tool.



Figure 7-5. Helping hands tool

Now let's look at some of the skills you are likely to need when working with advanced IOT projects.

ESD IS THE ENEMY

You should take care to make sure your body, your workspace, and your project is grounded to avoid electrostatic discharge (ESD). ESD can damage your electronics – permanently. The best way to avoid this is to use a grounding strap that loops around your wrist and attaches to an anti-static mat like these at uline.com/BL_7403/Anti-Static-Table-Mats.

Let's look at how to use the one tool you will likely use more than any other when learning electronics - the multimeter.

Using a Multimeter

The electrical skills needed for IOT projects can vary from plugging in wires on a breadboard — as we saw with the projects so far — to needing to solder components together or to printed circuit boards (PCBs). Regardless of whether you need to solder the electronics, you will need to be able to use a basic multimeter to measure resistance and check voltage and current.

A multimeter is a very useful and essential tool for any electronics hobbyist and downright required for any enthusiast of worth. A typical multimeter has a digital display² (typically an LCD or similar numeric display), a dial, and two or more posts or ports for plugging in test leads with probe ends. Most multimeters have ports for lower current (that you will use most) and ports for larger current. Test leads use red for positive and black for negative (ground). The ground port is where you plug in the black test lead and is often marked with a dash or COM for common. Which of the other ports you use will depend on what you are testing.

One thing to note on the dial is that there are many settings (with some values repeated) or those that look similar. For example, you will see a set of values (sometimes called a scale) for ohms, one or two set of values for amperage, and one or two set of values for volts. The set of values for voltages that has a V with a solid and dashed line is for DC whereas the range that has a V with a wavy line is for AC. Amperage ranges are marked in the same manner. Figure 7-6 shows a close-up of a multimeter dial labeled with the sets of values I've mentioned.



Figure 7-6. *Multimeter Dial (typical)*

²Older multimeters have an analog gauge. You can still find them if you want a bit of old school feel.

■ **Tip** When not in use, be sure to turn your multimeter dial to off or one of the voltage ranges if it has a separate off button.

There is a lot you can do with a multimeter. You can check voltage, measure resistance, and even check continuity. Most basic multimeters will do these functions. However, some multimeters have a great many more features such as testing capacitors and the ability to test AC as well as DC.

Let's see how we can use a multimeter to perform the most common tasks we will need for IOT projects: testing continuity, measuring voltage in a DC circuit, measuring resistance, and measuring current.

Testing Continuity

We test for continuity to determine if there is a path for the charged particles to flow: that is, our wires and components are connected properly. For example, you may want to check to ensure a wire has been spliced correctly.

To test for continuity, turn your multimeter dial to the position marked with an audible symbol, bell, or triangle with an arrow through it. Plug the black test lead into the COM port and the red test lead in the port marked with Hz $V\Omega$ or similar. Now you can touch the probe end of the test leads together to hear an audible tone or beep. Some multimeters don't have an audible tone but instead may display "1" or the like to indicate continuity. Check your manual for how your multimeter indicates continuity. Figure 7-7 shows how to set a multimeter to check for continuity including which ports to plug in the test leads.

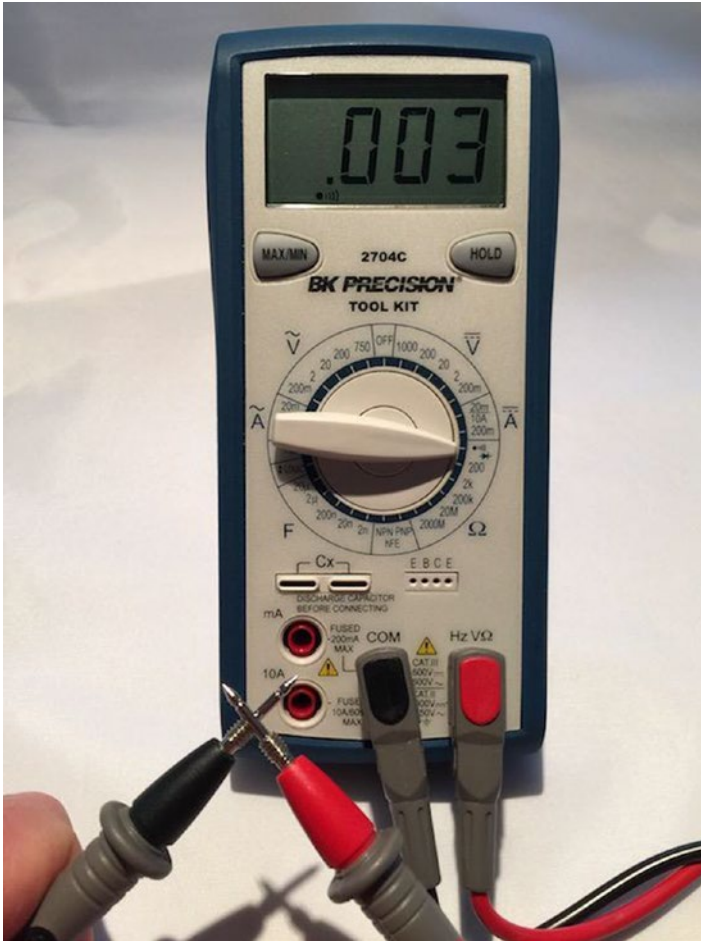


Figure 7-7. Settings for checking continuity

Notice in the photo I simply touched the probes together to demonstrate how to check for continuity. I like to do this just to ensure my multimeter is turned on and on the correct setting.³

Another excellent use for the continuity test is when diagnosing or discovering how cables are wired. For example, you can use the continuity test to discover which connector is connected on each end of the cable (sometimes called wire sorting or ringing out from the old telephone days).

³Yes, a bit of OCD there. Check, double-check, check again.

Measuring Voltage

Our IOT projects use DC. To measure voltage in the circuit, we will use the DC range on the multimeter. Notice the DC range has several stops. This is a scale selection. Choose the scale that closely matches the voltage range you want to test. For example, for our IOT projects we will often measure 3.3-12V so we choose 20 on the dial. Next, plug the black test lead into the COM port and the red test lead into the port labeled Hz V Ω .

Now we need something to measure! Take any battery you have in the house and touch the black probe to the negative side and the red probe to the positive side. You should see a value appear on the display that is close to the range for the battery. For example, if we used a 1.5V battery, we should see close to 1.5V. It may not be exactly 1.5-1.6V if the battery is depleted. So now you know how to test batteries for freshness! Figure 7-8 shows how to measure voltage of a battery.

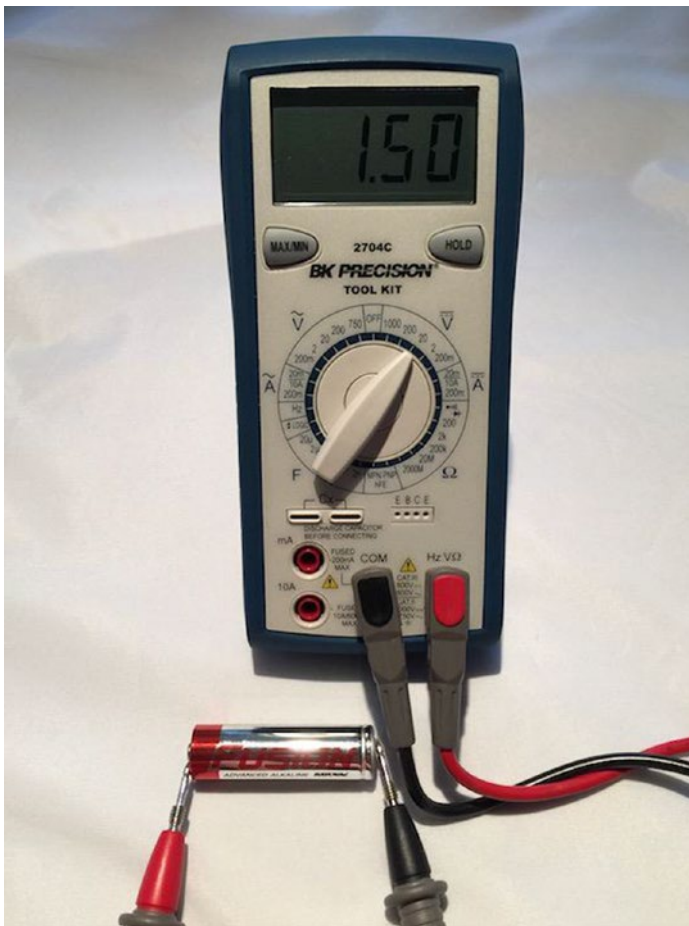


Figure 7-8. Measuring voltage of a battery

Notice the readout displays 1.50, which is the correct voltage for this AA battery. If I had reversed the probes – the red one on negative and the black on positive, the display would have read -1.50. This is Ok because it shows the current is flowing in the opposite direction of how the probes are oriented.

Note If you use the wrong probe when measuring voltage in a DC circuit, most multimeters will display the voltage as a negative number. Try that with your battery. It won't hurt the multimeter (or the battery)!

We can use this technique to measure voltage in our projects. Just be careful to place the probes on the appropriate positions and try not to cross or short by touching more than one component at a time with a single probe tip.

Measuring Current

Current is measured as amperage (milliamps - mA). Thus, we will use the range marked with an A with a straight and dashed line (not the wavy one - that's AC). We measure current in series. That is, we must place the multimeter in the circuit. This can be a little tricky because we must interrupt the flow of current and put the meter inline.

If you are familiar with how to use a breadboard, you can follow along with this experiment. However, if you haven't used a breadboard, you may want to read through this experiment, then return to it once you finish reading this chapter. For this experiment, we will use a breadboard power supply, an LED, and a resistor. We will wire the circuit such that we will use the multimeter to complete the circuit. Figure 7-9 shows how to set up the circuit with the multimeter inline.

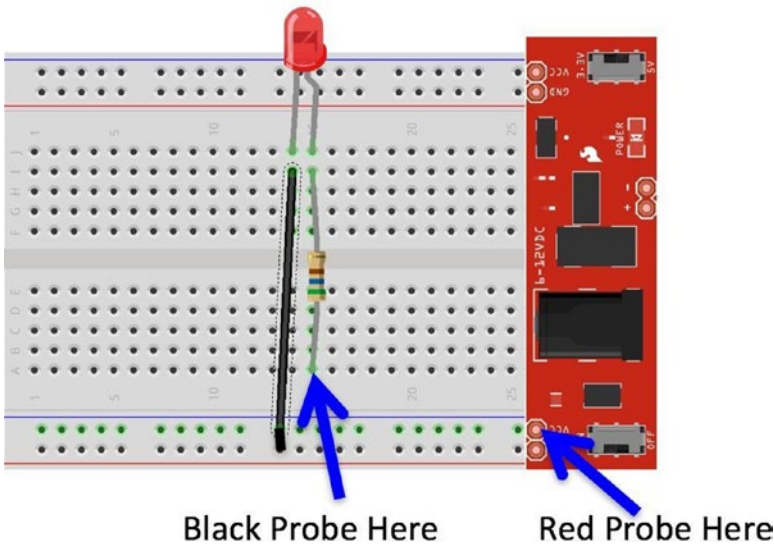


Figure 7-9. Measuring current

Before powering on your breadboard power supply, plug the black test lead into the COM port and the other test leads into the port labeled mA. Some multimeters use the same port for measuring voltage as well as current. Turn the dial on the multimeter to the 200mA setting. Then power on the breadboard power supply and touch the leads to the places indicated. Be careful to touch only the VCC pin on the breadboard power supply. Once the circuit is powered on, you should see a value on the multimeter. Figure 7-10 shows how to use a multimeter to measure current in a circuit

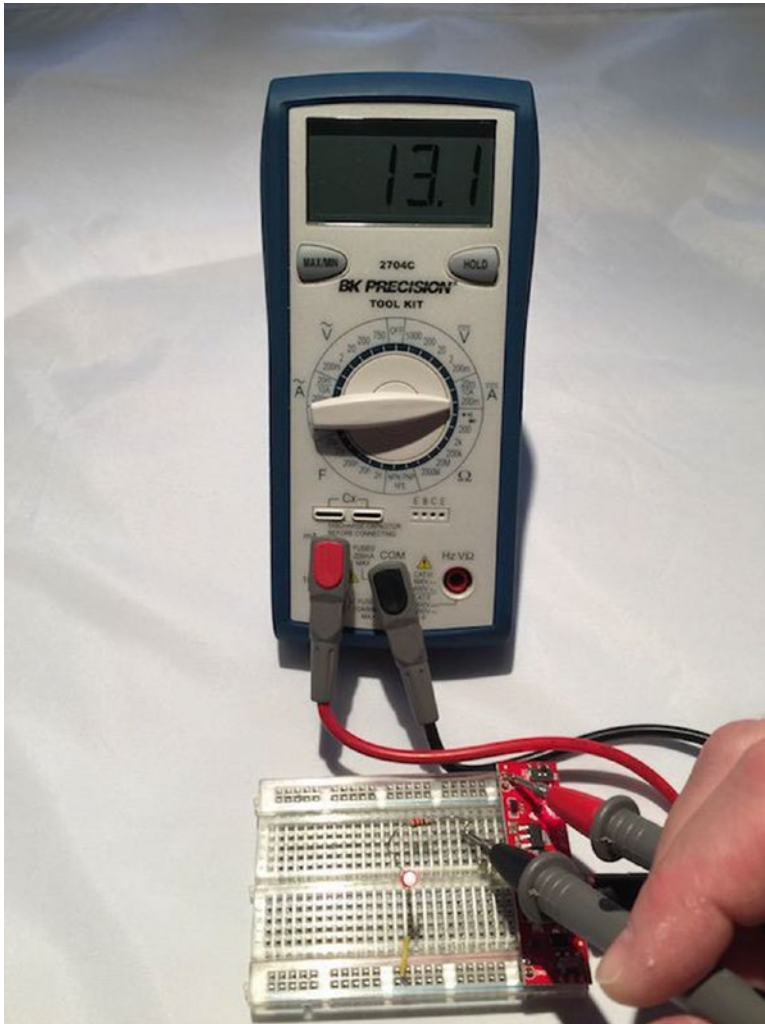


Figure 7-10. *Measuring current*

There is one other tricky thing about measuring current. If you attempt to measure current that is greater than the maximum for the port (for example, the meter in the photo has a maximum of 20mA on the one port). If I exceeded that by, say 5A, I would likely blow a fuse in the multimeter. This is not desirable but at least there is a fuse that we can replace should we make a mistake and choose the wrong port.

Measuring Resistance

Resistance is measured in ohms (Ω). The most common component we will use to introduce resistance in a circuit is a resistor. We can test the resistance of the charge through the resistor with our multimeter. To test resistance, choose the ohm scale that is closest to the rating of the resistor. For example, I am going to test a resistor that I believe is about 200 ohms but since I am not sure, I will choose the 2k setting.

Next, plug the black test lead into the COM port and the red test lead into the port labeled HzV Ω . Now, touch one probe to one side of a resistor and the other probe to the other side. It doesn't matter which side you choose – a resistor works in both directions. Notice the readout. The meter will read one of three things: 0.00, 1, or the actual resistor value.

In this case, the meter reads 0.219, meaning this resistor has a value of 220 Ω . Recall, I used the 2k scale, which means a resistor of 1k would read 1.0. Since the value is a decimal, I can move the decimal point to the left to get a whole number.

If the multimeter displays another value such as 0 or 1, it indicates the scale is wrong and you should try a higher scale. This isn't a problem. It just means you need to choose a larger scale. On the other hand, if the display shows 0 or a small number, you need to choose a lower scale. I like to go one tick of the knob either way when I am testing resistance in an unknown component or circuit.

Figure 7-11 shows an example of measuring resistance for a resistor. Notice the display reads 219. I am testing a resistor rated at 220 ohms. The reason it is 219 instead of 220 is because the resistor I am using is rated at 220 +/- 5%. Thus, the acceptable range for this resistor is 209-231 ohms.



Figure 7-11. *Measuring resistance of a resistor*

Now we know how to test a resistor to discover its rating. As we will see, those rings around the body of the resistor are the primary way we know its rating but we can always test it if we're unsure, someone has painted over it (hey, it happens), or we're too lazy to look it up.

Now, let's discuss the most fundamental concept you must understand when working with electronics – powering your project!

Powering Your Electronics

Electricity⁴ is briefly defined as the flow of electric charge and when used provides power for our electronics – from a common light bulb or ceiling fan to our high definition television or our new tablet. Whether you are powering your electronics with batteries or a power supply, you are initiating a circuit where electrons flow in specific patterns. There are two forms (or kinds) of power you will be using. Your home is powered by alternating current and your electronics are powered by direct current.

The term alternating current (AC) is used to describe the flow of charged particles that changes direction periodically at a specific rate (or cycle) reversing the voltage along with the current. Thus, AC systems are designed to work with a specific range of cycles as well as voltage. Typically, AC systems use higher voltages than direct current systems.

The term direct current (DC) is used to describe the flow of charged particles that do not change direction and thus always flow in a specific “direction.” Most electronics systems are powered with DC voltages and are typically at lower voltages than AC systems. For example, IOT projects typically run on lower direct current (DC) voltages in the range 3.3-24V.

■ **Tip** For more information about AC and DC current and the differences, see <https://learn.sparkfun.com/tutorials/alternating-current-ac-vs-direct-current-dc>.

Since DC flows in a single direction, components that operate on DC have a positive and negative “side” where current flows from positive to negative. The orientation of these sides – one to positive and one to negative is called polarity. Some components such as resistors can operate in either “direction” but you should always be sure to connect your components per its polarity. Most components are clearly marked but those that are not have a well-known arrangement. For example, the positive pole (side) of an LED is the longer of the two legs (called anode whereas the negative or shorter leg is called the cathode).

Despite the lower voltages, we mustn’t think that they are completely harmless or safe. Incorrectly wiring electronics (reversing polarity) or shorting (connecting positive and negative together) can damage your electronics and in some cases cause overheating, which, in extreme cases, causes electronics to catch fire.

■ **Caution** Don’t be tempted to think working with 3.3 or 5.5 volts is “safe.” Even a small amount of voltage improperly connected can lead to potentially devastating results. Don’t assume low DC voltage is harmless.

⁴<https://learn.sparkfun.com/tutorials/what-is-electricity>

I had a lesson in just how real this scenario can be a couple of years ago. I was changing the batteries in our smoke detectors. I took the old batteries out and placed them in my pocket. I had forgotten I had a small penknife in the same pocket. One of the batteries shorted on the knife and within about ten minutes, the battery heated to an alarming temperature. It wasn't enough to burn but had I left something like that unattended, it could have been bad.

That's a scary thought, isn't it? Consider it an admonishment as well as a warning; we should never relax our safe handling practices even for lower voltage projects.

Finally, DC components are often rated for a specific voltage range. Recall from our discussion on the various low-cost computing boards and GPIO headers, some boards operate at 5V whereas others operate at 3.3V (or less). Fortunately, there are several ways we can adapt components that work at different voltages – by using other components!

■ **Note** I have deliberately kept the discussion on power simple. There is far more to electrical current – even DC – than what I've described here. Once you understand these basics, you'll be able to work with the projects in this book and more.

Electronic Components

Aside from learning how to use a multimeter and possibly learning to solder, you also need to know something about the electronic components available to build your projects. In this section, I provide a short list and description of some of the common components in alphabetical order by name that you will encounter when building IOT solutions. I also cover breakout boards and logic circuits, which are small circuits built with a set of components that provide a feature or solve a problem. For example, you can get breakout boards for USB host connections, Ethernet modules, logic shifters, real-time clocks, and more.

Button

A button (sometimes called a momentary button) is a mechanism that makes a connection when pressed. More specifically, a button connects two or more poles together while it is pressed. A common (and perhaps overused) example of a button is a home doorbell. When pressed, it completes a circuit that triggers a chime, bell, tone, or music to play. Some older doorbells continue to sound while the button is pressed.

In IOT projects, we will use buttons to trigger events, start and stop actions, and similar operations. A button is a simple form of a switch but unlike a switch, you must continue to press the button to make the electrical connections. Most buttons have at least two legs (or pins) that are connected when the button is pressed. Some have more than two legs connected in pairs and some of those can permit multiple connections. Figure 7-12 shows several buttons.

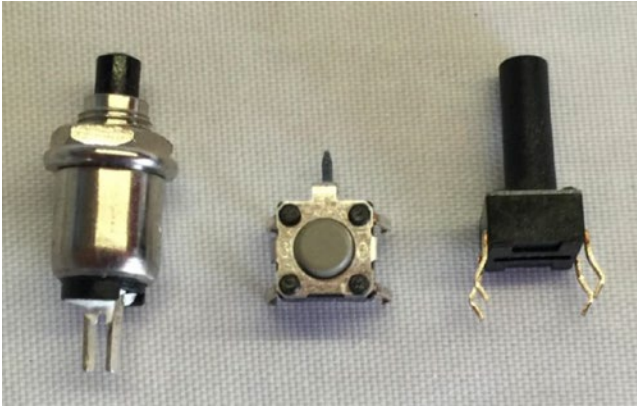


Figure 7-12. Momentary buttons

There is a special variant of a momentary button called a latching momentary button. This version uses a notch or detent to keep the poles connected until it is pushed again. If you’ve seen a button on a stereo or in your car that remains depressed until pressed again, it is likely a latching momentary button.

There are all kinds of buttons from those that can be used with breadboards (the spacing of the pins allow it to be plugged into a breadboard), can be mounted to a panel, or those made for soldering to printed circuit boards.

Capacitor

A capacitor is designed to store charges. As current flows through the capacitor, it accumulates charge and can discharge after the current is disconnected. In this way, it is like a battery but unlike a battery, a capacitor charges and discharges very fast. We use capacitors for all manner of current storage from blocking current, reducing noise in power supplies, in audio circuits, and more. Figure 7-13 shows several capacitors.

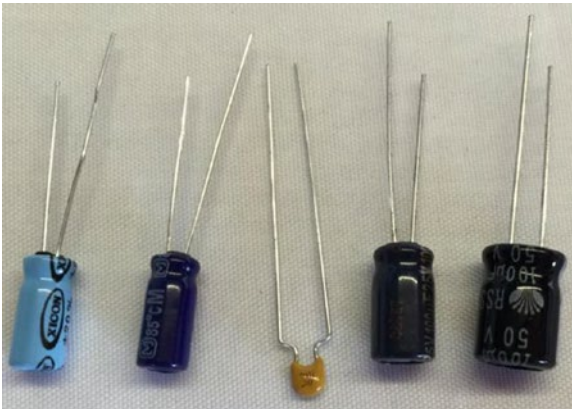


Figure 7-13. Capacitors

There are several types of capacitors but we will most often encounter capacitors when building power supplies for IOT projects. Most capacitors have two legs (pins) that are polarized. That is, one is positive and the other negative. Be sure to connect the capacitor with the correct polarity in your circuit.

Diode

A diode is designed to allow current to flow in only one direction. Most are marked with an arrow pointing to a line, which indicates the direction of flow. A diode is often used as rectifiers in AC-to-DC converters (devices that convert AC to DC voltage), used in conjunction with other components to suppress voltage spikes, or protect components from reversed voltage. They are often used to protect against current flowing into a device.

Most diodes are shaped like a small cylinder, and they are usually black with silver writing and have two legs. They look a little like resistors. We use a special variant called a Zener diode in power supplies to help regulate voltages. Figure 7-14 shows several Zener diodes.

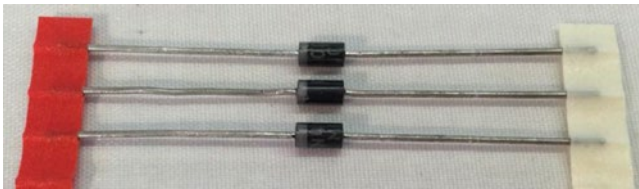


Figure 7-14. Diodes

Fuse

A fuse is designed to protect a device (the entire circuit) from current greater than what the components can safely operate. Fuses are placed inline on the positive pole. When too much current flows through the fuse, the internal parts trigger a break in the flow of current.

Some fuses use a special wire inside that melts or breaks (thereby rendering it useless but protecting your equipment) while other fuses use a mechanism that operates like a switch (many of these are resettable). When this happens, we say the fuse has “blown” or “tripped.” Fuses are rated at a certain current in amperage indicating the maximum amps that the fuse will permit to flow without tripping.

Fuses come in many shapes and varieties and can work with AC or DC voltage. Those we will use are of the disposable variety. Figure 7-15 shows an example of two fuses: an automotive-style blade fuse on the left and a glass cartridge fuse on the right.



Figure 7-15. Fuses

If you are familiar with the electrical panel in your home that house the circuit breakers, they are resettable fuses. So, the next time one of them goes “click” and the lights go out, you can say, “Hey, a fuse has tripped!” Better still, now you know why - you have exceeded the maximum rating of the circuit breaker.

This is probably fine in situations where you accidentally left that infrared heater on when you dropped the toast and started the microwave (it happens), but if you are tripping breakers frequently without any load, you should call an electrician to have the circuit checked.

Light Emitting Diode (LED)

As we learned in Chapter 3, an LED has two legs where the longer leg is positive and the shorter negative. LEDs also have a flat edge that indicates the negative leg. They come in a variety of sizes ranging from as small as 3mm to 10mm. Figure 7-16 shows an example of some smaller LEDs.



Figure 7-16. Light emitting diodes

Recall we also needed to use a resistor with an LED. We need this to help reduce the flow of the circuit to lower the current flowing through the LED. LEDs can be used with lower current (they will burn a bit dimmer than normal) but should not be used with a higher current.

To determine what size resistor we need, we need to know several things about the LED. This data is available from the manufacturer who provides the data in the form of a datasheet or in the case of commercially packaged products, lists the data on the package. The data we need includes the maximum voltage, the supply voltage (how many volts are coming to the LED), and the current rating of the LED.

For example, if I have an LED like the one we used in the last chapter, in this case a 5mm red LED, we find on Adafruit's website (adafruit.com/products/297) that the LED operates at 1.8-2.2V and 20mA of current. Let's say we want to use this with a 5V supply voltage. We can then take these values and plug them into this formula⁵:

$$R = (V_{cc} - V_f) / I$$

Using more descriptive names for the variable, we get the following.

$$\text{Resistor} = (\text{Volts_supply} - \text{Volts_forward}) / \text{Desired_current}$$

Plugging our data in, we get this result. Note that we have mA so we must use the correct decimal value (divide by 1000). In this case, it is 0.020 and we will pick a voltage in the middle.

$$\begin{aligned} \text{Resistor} &= (5 - 2.0) / 0.020 \\ &= 3.0 / 0.020 \\ &= 150 \end{aligned}$$

Thus, we need a resistor of 150 ohms. Cool. Sometimes the formula will produce a value that does not match any existing resistors. In that case, choose one closest to the value but a bit larger. Remember, we want to limit and thus err on the side of more restrictive than less restrictive. For example, if you found you need a resistor of 95 ohms, you can use one rated at 100 ohms, which is safer than using one rated at 90 ohms.

■ **Tip** Always err on the side of the more restrictive resistor when the formula produces a value for which there is no resistor available.

Also, if you use LEDs in serial or parallel, the formula is a little different. See <https://learn.adafruit.com/all-about-leds> for more information about using LEDs in your projects and calculating the size of resistors to use with LEDs.

⁵A variant of Ohm's law (https://en.wikipedia.org/wiki/Ohm's_law).

Relay

A relay is an interesting component that helps us control higher voltages with lower voltage circuits. For example, suppose you wanted to control a device that is powered by 12V from your MicroPython board, which only produces a maximum of 3V. A relay can be used with a 3V circuit to turn on (or relay) power from that higher source. In this example, we would use the MicroPython board output to trigger the relay to switch on the 12V power. Thus, relays are a form of switch. Figure 7-17 shows a typical relay and how the pins are arranged.



Figure 7-17. Relay

Relays can take a lot of different forms and typically have slightly different wiring options such as where the supply voltage is attached and where the trigger voltage attaches as well as whether the initial state is open (no flow) or close (flow) and thus the behavior of how it controls voltage. Some relays come mounted on a PCB with clearly marked terminals for changing its switching feature and where everything plugs in. If you want to use relays in your projects, always check the datasheet to make sure you are wiring it correctly based on its configuration.

You can also use relays to allow your DC circuit to turn AC appliances on and off like those from Adafruit (<https://www.adafruit.com/product/2935>).

Resistor

A resistor is one of the standard building blocks of electronics. Its job is to impede current and impose a reduction in voltage (which is converted to heat). Its effect, known as resistance, is measured in ohms. A resistor can be used to reduce voltage to other components, limiting frequency response, or protect sensitive components from over voltage. Figure 7-18 shows several resistors.

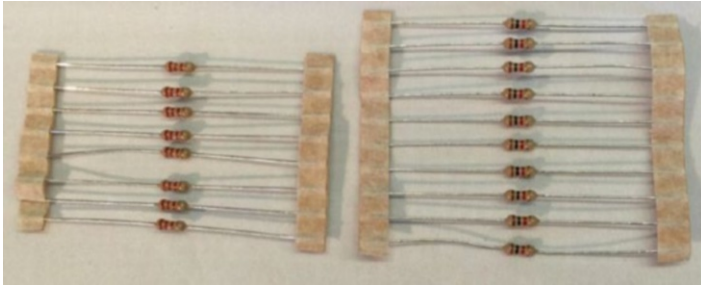


Figure 7-18. Resistors

When a resistor is used to pull up voltage (by attaching one end to positive voltage) or pull down voltage (by attaching one end to ground) (resistors are bidirectional), it eliminates the possibility of the voltage floating in an indeterminate state. Thus, a pull-up resistor ensures that the stable state is positive voltage, and a pull-down resistor ensures that the stable state is zero voltage (ground).

Switch

A switch is designed to control the flow of current between two or more pins. Switches come in all manner of shapes, sizes, and packaging. Some are designed as a simple on/off while others can be used to change current from one set of pins to another. Like buttons, switches come in a variety of mounting options from PCB (also called through hole) to panel mount for mounting in enclosures. Figure 7-19 shows a variety of switches.



Figure 7-19. Various switches

Switches that have only one pole (leg or side) are called single-pole switches. Switches that can divert current from one set of poles to another set are called two-pole switches. Switches where there is only one secondary connection per pole are called single-throw switches. Switches that disconnect from one set of poles and connect to another while maintaining a common input are called double-throw switches. These are often combined and form the switch type (or kind) as follows.

- *SPST*: single pole, single throw
- *DPST*: double pole, single throw
- *SPDT*: single pole, double throw
- *DPDT*: double pole, double throw
- *3PDT*: three pole, double throw

There may be other variants that you could encounter. I like to keep it straight like this; if I have just an on/off situation, I want a single throw switch. How many poles depends on how many wires or circuits I want to turn on or off at the same time. For double-throw switches, I use these when I have an “A” condition and “B” condition that I want A on when B is off and vice versa. I sometimes use multiple throw switches when I want an “A,” “B,” and off situations where I use the center position (throw) as off. You can be very creative with switches!

Transistor

A transistor (a bipolar transistor) is designed to switch current on/off in a cycle or amplify fluctuations in current. Interestingly, transistors used to amplify current replaced vacuum tubes. If you are an audiophile, you likely know a great deal about vacuum tubes. When a resistor operates in switching mode, it behaves like a relay but its “off” position still allows a small amount of current to flow. Transistors are used in audio equipment, signal processing, and switching power supplies. Figure 7-20 shows two varieties of transistors.

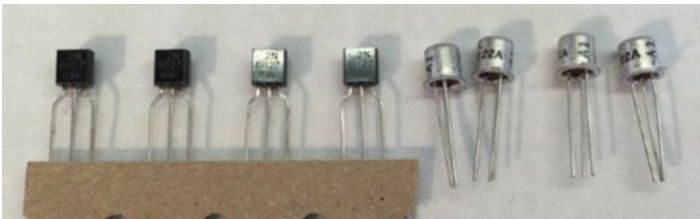


Figure 7-20. *Transistors*

Transistors come in all manner of varieties, packaging, and ratings that make it suitable for one solution or another.

Voltage Regulator

A voltage regulator (linear voltage regulator) is designed to keep the flow of current constant. Voltage regulators often appear in electronics when we need to condition or lower current from a source. For example, we want to supply 5V to a circuit but only have a 9V power supply. Voltage regulators accomplish this (roughly) by taking current in and dissipating the excess current through a heat sink. Thus, voltage regulators have three legs: positive current in, negative, and positive current out. They are typically shaped like those shown in Figure 7-21 but other varieties exist.

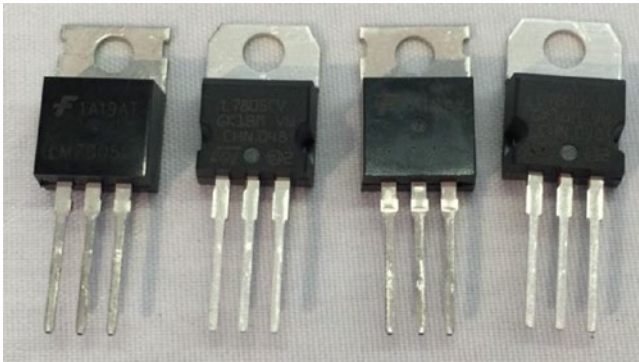


Figure 7-21. Voltage regulators

The small hole in the plate that extends out of the voltage regulator is where the heat sink is mounted. Voltage regulators are often numbered to match their rating. For example, a LM7805 produces 5V whereas a LM7833 produces 3.3V.

An example of using a voltage regulator to supply power to a 3.3V circuit on a breadboard is shown in Figure 7-22. This circuit was designed with capacitors to help smooth or condition the power. Notice the capacitors are rated with μF , which means microfarad.

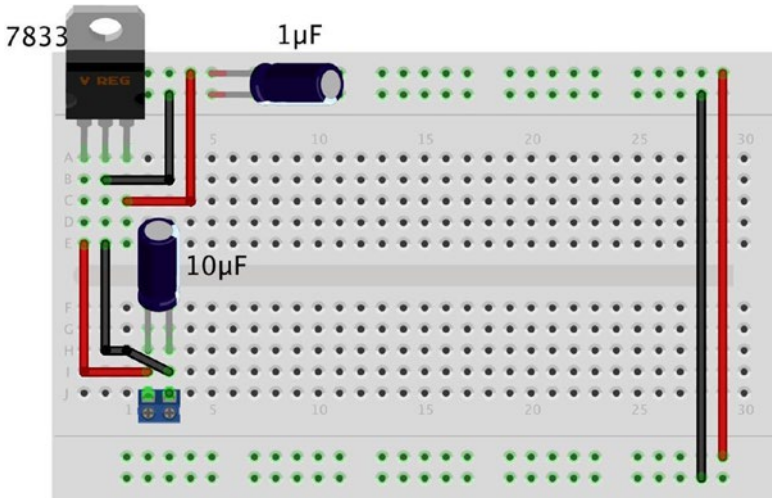


Figure 7-22. Power supply circuit on a breadboard with voltage regulator

Breakout Boards and Circuits

Breakout boards are our modular building blocks for IOT solutions. They typically combine several components together to form a function such as measuring temperature, enabling reading GPS data, communicating via cellular services, and more. Figure 7-23 shows two breakout boards. On the left is an Adafruit analog to digital converter (adafruit.com/products/1083), and on the right is an Adafruit Barometric Pressure Sensor breakout board (<https://www.adafruit.com/product/1603>).



Figure 7-23. Breakout boards

Whenever you design a circuit or IOT solution, you should consider using breakout boards as much as possible because they simplify the use of the components. Take the Barometric Pressure Sensor, for example. Adafruit has designed this board so that all we need to do to use it is attach power and connect it to our IOT Device on its I2C bus. An I2C bus is a fast digital protocol that uses two wires (plus power and ground) to read data from circuits (or devices).

Thus, there is no need to worry about how to connect the sensor to other components to use it - just connect it up like any I2C device and start reading data! We will use several breakout boards in the projects later in this book.

Using a Breadboard to Build Circuits

If you have been following along with the projects thus far in the book, you have already encountered a breadboard to make a very simple circuit. Recall from Chapter 3 that a breadboard is a tool we use to plug components into to form circuits. Technically, we're using a solderless breadboard. A solder breadboard has the same layout only it has only through-hole solder points on a PCB.

WHY ARE THEY CALLED BREADBOARDS?

In the grand old days of microelectronics and discrete components became widely available for experimentation, when we wanted to prototype a circuit, some would use a piece of wood with nails driven into it (sometimes in a grid pattern) where connections were made (called "runs") by wrapping wire around the nails. Some used a breadboard from the kitchen to build their wire wrap prototypes. The name has stuck ever since.

A breadboard allows us to create prototypes for our circuits or simply temporary circuits without having to spend the time (and cost) to make the printed circuit board. Prototyping is the process of experimenting with a circuit by building and testing your ideas. In fact, once we've got our circuit to work correctly, we can use the breadboard layout to help us design a PCB. Figure 7-24 shows several breadboards.

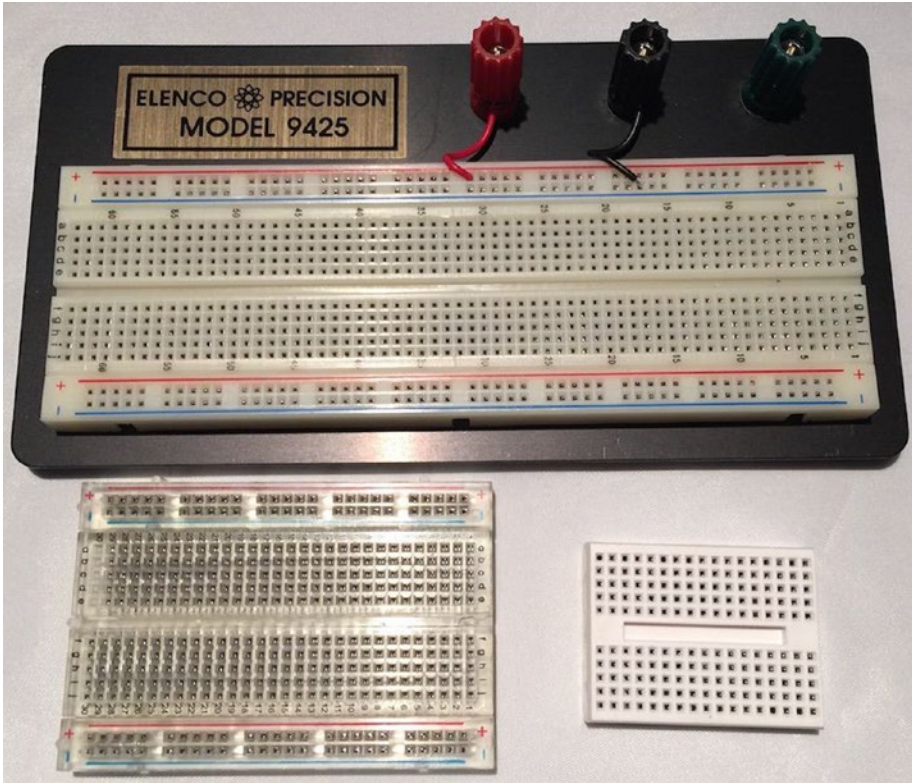


Figure 7-24. Assorted breadboards

Recall that most breadboards (there are several varieties) have a center groove (called a ravine) or a printed line down the center of the board. This signifies the terminal strips that run perpendicular to the channel are not connected. That is, the terminal strip on one side is not connected to the other side. This allows us to plug integrated circuits (IC) or chips that are packaged as two rows of pins. Thus, we can plug the IC into the breadboard with one set of pins on each side of the breadboard. We see this in the example below.

Most breadboards also have one or more sets of power rails that are connected together parallel to the ravine. If there are two sets, the sets are not connected together. The power rails may have a colored reference line but this is only for reference; you can make either one positive with the other negative. Finally, some breadboards number the terminal strip rows. These are for reference only and have no other meaning. However, they can be handy for making notes in your engineering notebook. Figure 7-25 shows the nomenclature of a breadboard and how the terminal strips and power rails are connected together.

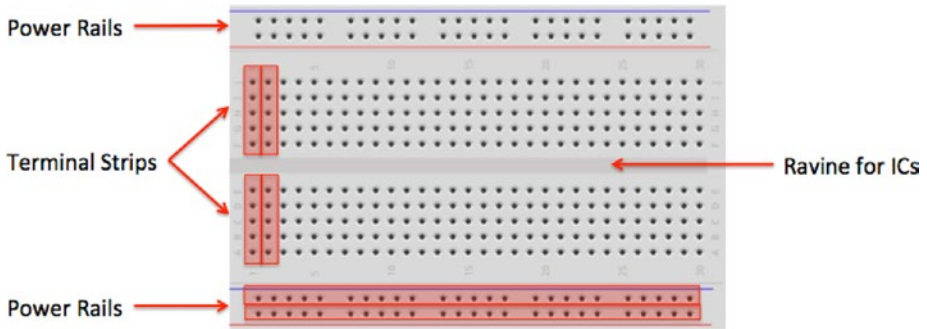


Figure 7-25. Breadboard layout

■ **Note** The sets of power rails are not connected together. If you want to have power on both sides of the breadboard, you must use jumpers to connect them.

FRITZING: A BREADBOARDING SOFTWARE APPLICATION

The drawings of breadboards in this book were made with a program named Fritzing (<http://fritzing.org/home/>). This open source application allows you to create a digital representation of a circuit on a breadboard. It is quite handy to use. If you find yourself wanting to design a prototype circuit, using Fritzing can help save you a lot of trial and error. As a bonus, Fritzing allows you to see the same circuit in an electronic schematic or PCB layout view. I recommend downloading and trying this application out.

It is sometimes desirable to test a circuit out separately from code. For example, if we want to make sure all our devices are connected together properly, we can use a breadboard power supply to power the circuit. This way, if something goes horribly wrong, we don't risk damaging our IOT device. Most breadboard power supplies are built on a small PCB with a barrel jack for a wall wart power supply, two sets of pins to plug into the power rails on the breadboard, an off switch (very handy), and some can generate different voltages. Figure 7-26 shows one of my favorite breadboard power supplies from Sparkfun (sparkfun.com/products/13157).

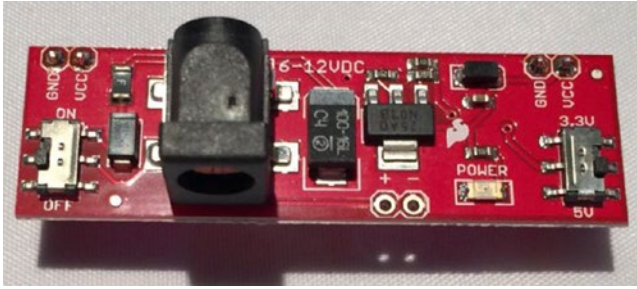


Figure 7-26. Breadboard power supply

Should our circuits require more room than what is available on a single breadboard, you can use multiple breadboards by simply jumping the power rails and continuing the circuit. To facilitate this, some breadboards can be connected using small nubs and slots on the side. Finally, most breadboards also come with an adhesive backing that you can use to mount on a plate or inside an enclosure or similar workspace. If you decide to use the adhesive backing, be forewarned that they cannot be unstuck easily - they stay put quite nicely.

Now that we know more about how breadboards work, let's discuss the component our IOT solutions we will employ to collect data: sensors.

What Are Sensors?

A sensor is a device that measures phenomena of the physical world. These phenomena can be things you see, like light, gases, water vapor, and so on. They can also be things you feel, like temperature, electricity, water, wind, and so on. Humans have senses that act like sensors, allowing us to experience the world around us. However, there are some things your sensors can't see or feel, such as radiation, radio waves, voltage, and amperage. Upon measuring these phenomena, it's the sensor's job to convey a measurement in the form of either a voltage representation or a number.

There are many forms of sensors. They're typically low-cost devices designed for a single purpose and with a limited capability for processing. Most simple sensors are discrete components; even those that have more sophisticated parts can be treated as separate components. Sensors are either analog or digital and are typically designed to measure only one thing. But an increasing number of sensor modules are designed to measure a set of related phenomena, such as the USB Weather Board from SparkFun Electronics (www.sparkfun.com/products/10586).

The following sections examine how sensors measure data, how to store that data, and examples of some common sensors.

How Sensors Measure

Sensors are electronic devices that generate a voltage based on the unique properties of their chemical and mechanical construction. One of the common misconceptions some have about sensors is they do not manipulate the phenomena (change the event or data) they're designed to measure. Rather, sensors sample some physical variable and turn it into a proportional electric signal (voltage, current, digital, and so on).

For example, a humidity sensor measures the concentration of water (moisture) in the air. Humidity sensors react to these phenomena and generate a voltage that the microcontroller or similar device can then read and use to calculate a value on a scale. A basic, low-cost humidity sensor is the DHT-22 available from most electronic stores (<https://www.adafruit.com/product/385>). Figure 7-27 shows a typical DHT-22 sensor.

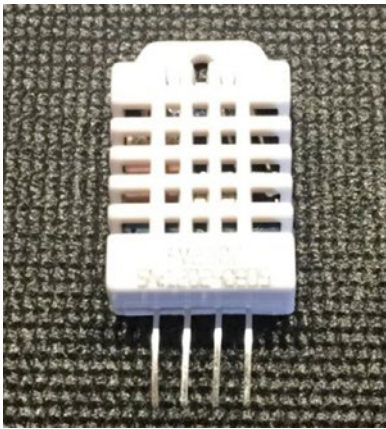


Figure 7-27. DHT-22 Humidity Sensor

The DHT-22 is designed to measure temperature as well as humidity. It generates a digital signal on the output (data pin). Although simple to use, it's a bit slow and should be used to track data at a reasonably slow rate (no more frequently than about once every 3 or 4 seconds).

When this sensor generates data, that data is transmitted as a series of high (interpreted as a 1) and low (interpreted as a 0) voltages that the microcontroller can read and use to form a value. In this case, the microcontroller reads a value 40 bits in length (40 pulses of high or low voltage) — that is, 5 bytes — from the sensor and places it in a program variable. The first two bytes are the value for humidity, the second two are for temperature, and the fifth byte is the checksum value to ensure an accurate read. Fortunately, all this hard work is done for you in the form of a special library designed for the DHT-22 and similar sensors.

The DHT-22 produces a digital value. Not all sensors do this; some generate a voltage range instead. These are called analog sensors. Let's take a moment to understand the differences. This will become essential information as you plan and build your sensor nodes.

Analog Sensors

Analog sensors are devices that generate a voltage range, typically between 0 and 5 volts. An analog-to-digital circuit is needed to convert the voltage to a number. But it isn't that simple (is it ever?). Analog sensors work like resistors and, when connected to GPIO pins, often require another resistor to “pull up” or “pull down” the voltage to avoid spurious changes in voltage known as floating. This is because voltage flowing through resistors is continuous in both time and amplitude.

Thus, even when the sensor isn't generating a value or measurement, there is still a flow of voltage through the sensor that can cause spurious readings. Your projects require a clear distinction between OFF (zero voltage) or ON (positive voltage). Pull-up and pull-down resistors ensure that you have one of these two states. It's the responsibility of the A/D converter to take the voltage read from the sensor and convert it to a value that can be interpreted as data.

When sampled (when a value is read from a sensor), the voltage read must be interpreted as a value in the range specified for the given sensor. Remember that a value of, say, 2 volts from one analog sensor may not mean the same thing as 2 volts from another analog sensor. Each sensor's data sheet shows you how to interpret these values.

As you can see, working with analog sensors is a lot more complicated than using the DHT-22 digital sensor. With a little practice, you will find that most analog sensors aren't difficult to use once you understand how to attach them to a microcontroller and how to interpret their voltage on the scale in which the sensor is calibrated to work.

Digital Sensors

Digital sensors like the DHT-22 are designed to produce a string of bits using serial transmission (one bit at a time). However, some digital sensors produce data via parallel transmission (one or more bytes⁶ at a time). As described previously, the bits are represented as voltage, where high voltage (say, 5 volts) or ON is 1 and low voltage (0 or even -5 volts) or OFF is 0. These sequences of ON and OFF values are called discrete values because the sensor is producing one or the other in pulses — it's either ON or OFF.

Digital sensors can be sampled more frequently than analog signals because they generate the data more quickly and because no additional circuitry is needed to read the values (such as A/D converters and logic or software to convert the values to a scale). Thus, digital sensors are generally more accurate and reliable than analog sensors. But the accuracy of a digital sensor is directly proportional to the number of bits it uses for sampling data.

The most common form of digital sensor is the pushbutton or switch. What, a button is a sensor? Why, yes, it's a sensor. Consider for a moment the sensor attached to a window in a home security system. It's a simple switch that is closed when the window is closed and open when the window is open. When the switch is wired into a circuit, the flow of current is constant and unbroken (measuring positive volts using a pull-up resistor) when the window is closed and the switch is closed, but the current is broken (measuring zero volts) when the window and switch is open. This is the most basic of ON and OFF sensors.

⁶This depends on the width of the parallel buffer. An 8-bit buffer can communicate 1 byte at a time, a 16-bit buffer can communicate 2 bytes at a time, and so on.

Most digital sensors are small circuits of several components designed to generate digital data. Unlike analog sensors, reading their data is easy because the values can be used directly without conversion (except to other scales or units of measure). Some may suggest this is more difficult than using analog sensors, but that depends on your point of view. An electronics enthusiast would see working with analog sensors as easier, whereas a programmer would think digital sensors are simpler to use.

Now let's look at some of the sensors available and the types of phenomena they measure.

Examples of Sensors

An IOT solution that observes something may use at least one sensor and a means to read and interpret the data. You may be thinking of all manner of useful things you can measure in your home or office, or even in your yard or surroundings. You may want to measure the temperature changes in your new sunroom, detect when the mail carrier has tossed the latest circular in your mailbox, or perhaps keep a log of how many times your dog uses his doggy door. I hope that by now you can see these are just the tip of the iceberg when it comes to imagining what you can measure.

What types of sensors are available? The following sections describe some of the more popular sensors and what they measure. I also provide a few hints on how you might want to use the sensor in an IOT project. However, this is just a sampling of the growing array of sensors available. Perusing the catalogs of online electronics vendors like Mouser Electronics (www.mouser.com), SparkFun Electronics (www.sparkfun.com), and Adafruit Industries (www.adafruit.com) will reveal many more examples.

I also include photos of popular examples for some of the sensor types.

Accelerometers

These sensors measure motion or movement of the sensor or whatever it's attached to. They're designed to sense motion (velocity, inclination, vibration, and so on) on several axes. Some include gyroscopic features. Most are digital sensors. A Wii Nunchuck (or WiiChuck) contains a sophisticated accelerometer for tracking movement. Aha: now you know the secret of those funny little thingamabobs that came with your Wii! You may want to add accelerometers if your IOT project involves something in motion and the observation of that motion provides useful information.

Audio Sensors

Perhaps this is obvious, but microphones are used to measure sound. Most are analog, but some of the better security and surveillance sensors have digital variants for higher compression of transmitted data. IOT projects such as home security, child monitoring, ghost hunting, or auditory health can all benefit from integrating audio sensors.

Barcode Readers

These sensors are designed to read barcodes. Most often, barcode readers generate digital data representing the numeric equivalent of a barcode. Such sensors are often used in inventory-tracking systems to track equipment through a plant or during transport. They're plentiful, and many are economically priced, enabling you to incorporate them into your own projects. If your IOT project requires capturing data from an object, you may want to consider barcodes.

For example, if you want to sense when parking lot subscribers enter or exit an unattended parking lot, you could position a barcode reader at the gates that read barcodes that you design and distribute to your subscribers. When the car pulls up to the gate, the barcode reader can read the barcode, log the entry, and raise the gate. If you've ever lived in a large city, worked in a controlled office complex, or were a commuter student, you may have encountered parking solutions like this.

Biometric Sensors

A sensor that reads fingerprints, irises, or palm prints contains a special sensor designed to recognize patterns. Given the uniqueness inherent in patterns such as fingerprints and palm prints, they make excellent components for a secure access system. Most biometric sensors produce a block of digital data that represents the fingerprint or palm print. IOT projects that require a greater level of security may want to include a biometric sensor to help identify the user of the system.

Capacitive Sensors

A special application of capacitive sensors, pulse sensors are designed to measure your pulse rate and typically use a fingertip for the sensing site. Special devices known as pulse oximeters (called pulse-ox by some medical professionals) measure pulse rate with a capacitive sensor and determine the oxygen content of blood with a light sensor.

If you own modern electronic devices, you may have encountered touch-sensitive buttons that use special capacitive sensors to detect touch and pressure. If your IOT project needs to measure any sort of movement, or respond to touch, capacitive sensors can help provide a futuristic non-tactile interface. The Touch Bar on the latest MacBook Pro is an example of such a solution.

Figure 7-28 shows two examples of touch-sensitive modules that you can buy from Adafruit. In this case, we see breakout boards for a momentary switch (www.adafruit.com/products/1374) and a toggle switch (www.adafruit.com/products/1375).

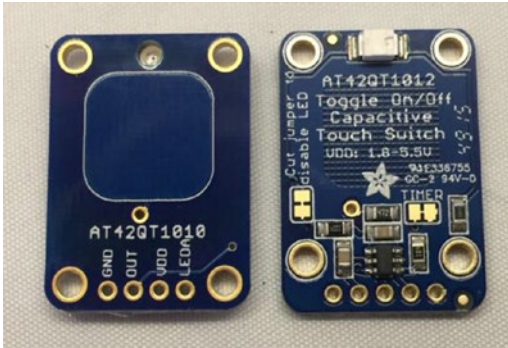


Figure 7-28. Touch Capacitive Sensor Breakout Boards

Coin Sensors

This is one of the most unusual types of sensors. These devices⁷ are like the coin slots on a typical vending machine. Like their commercial equivalent, they can be calibrated to sense when a certain size of coin is inserted. Although not as sophisticated as commercial units that can distinguish fake coins from real ones, coin sensors can be used to add a new dimension to your projects. A great, practical IOT project for parents would be a coin-operated WiFi station where the children have to buy their own Internet time. Not only will this keep them from using the Internet too much, it may also help teach them how to budget their allowance. Now, that should keep the kids from spending too much time on the Internet!

Current Sensors

These are designed to measure voltage and amperage. Some are designed to measure change, whereas others measure load. IOT projects that integrate circuits or need to monitor the flow of electricity will need a current sensor. These may be some of the more esoteric projects, but you can use these sensors to monitor the behavior of existing solutions without modifying them.

For example, if you wanted to adapt sensors to observe a manufacturing machine, you could add sensors that monitor the current to the various components. That is, you may be able to record when voltage is applied to motors, actuators, or even warning lights to determine when (or how much) the devices are activated. However, as a hobbyist, you are more likely interested in building your own multimeter or similar tool.

Flex/Force Sensors

Resistance sensors measure flexes in a piece of material or the force or impact of pressure on the sensor. Flex sensors may be useful for measuring torsional effects or to measure

⁷www.sparkfun.com/products/11719

finger movements (like in a Nintendo Power Glove). Flex-sensor resistance increases when the sensor is flexed. For example, if you want to create an IOT solution that reports your fishing experience in real time, you might want to use a flex sensor on your fishing rod to report every time you cast or got a hit on your lure.

Gas Sensors

There are a great many types of gas sensors. Some measure potentially harmful gases such as LPG and methane and other gases such as hydrogen, oxygen, and so on. Other gas sensors are combined with light sensors to sense smoke or pollutants in the air. The next time you hear that telltale and often annoying low-battery warning beep⁸ from your smoke detector, think about what that device contains. Why, it's a sensor node! If your IOT project needs to observe or detect any form of gas, especially if it involves reacting to certain gases or levels thereof, you will need to use the appropriate gas sensors.

Light Sensors

Sensors that measure the intensity or lack of light are special types of resistors: light-dependent resistors (LDRs), sometimes called photo resistors or photocells. Thus, they're analog by nature.

If you own a Mac laptop, chances are you've seen a photo resistor in action when your illuminated keyboard turns itself on in low light. Special forms of light sensors can detect other light spectrums such as infrared (as in older TV remotes). For example, if you want your IOT project to automatically adjust the brightness of its display, a light sensor is the component you need.

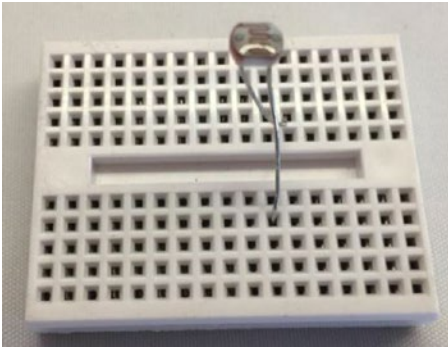


Figure 7-29. *Mini Photocell*

⁸I subscribe to the more is better theory and have many detectors in our home, which is great but when the batteries run down, I can never tell which detector is beeping! This becomes maddeningly frustrating when they beep only once or twice then go silent. Fortunately, I've replaced nearly all of them with the newest 10-year battery variants. No more wild beep goose chases!

The following show two examples of light sensors. Figure 7-29 shows a typical mini photocell (www.sparkfun.com/products/9088) and Figure 7-30 shows a color sensor (www.adafruit.com/products/1334).



Figure 7-30. Color Sensor Breakout Board

Liquid-Flow Sensors

These sensors resemble valves and are placed inline in plumbing systems. They measure the flow of liquid as it passes through. Basic flow sensors use a spinning wheel and a magnet to generate a Hall effect (rapid ON/OFF sequences whose frequency equates to how much water has passed). If your IOT project involves any form of liquid such as a garden pond or irrigation system, knowing the flow of the water may be helpful in learning or observing something.

Liquid-Level Sensors

A special resistive solid-state device can be used to measure the relative height of a body of water. One example generates low resistance when the water level is high and higher resistance when the level is low. Like liquid-flow sensors, liquid-level sensors are typically used in the same solution. Figure 7-31 shows a typical liquid-level sensor that operates as a switch where the float closes the switch when the water level rises.



Figure 7-31. *Water-Level Sensor*

Location Sensors

Modern smartphones have GPS sensors for sensing location, and of course GPS devices use the GPS technology to help you navigate. Fortunately, GPS sensors are available in low-cost forms, enabling you to add location sensing to your project. GPS sensors generate digital data in the form of longitude and latitude, and most can also sense altitude. If your IOT project needs to report its location, a GPS sensor can give you very accurate readings. However, like most sensors, GPS sensors can have a degree of inaccuracy. Depending on how close you need to locate something, you may need to spend a bit more on a more accurate GPS sensor.

Magnetic-Stripe Readers

These sensors read data from magnetic stripes (like that on a credit card) and return the digital form of the alphanumeric data (the actual strings). IOT projects that include a security component may want to use a magnetic-stripe reader to help identify a user. When combined with a password and a biometric sensor, security can be increased considerably. That is, someone would have to know something (a password or pin), possess something (security card with a magnetic-stripe encoded with a key phrase, number, user id, etc.), and be validated as someone (fingerprint) before gaining access.

Magnetometers

These sensors measure orientation via the strength of magnetic fields. A compass is a sensor for finding magnetic north. Some magnetometers offer multiple axes to allow even finer detection of magnetic fields. This is another sensor that you may not encounter very often, but if your IOT project needs to measure magnetic fields from motors or atmospheric phenomena, you may want to look at magnetometers.

Moisture Sensors

Moisture sensors measure the amount of moisture in a substance (such as soil) or in the air. They typically send data in the form of a voltage reading where low values indicate less moisture. You often find moisture sensors in atmospheric projects or even plant monitoring solutions. Figure 7-32 shows a typical soil moisture sensor (). Notice the prongs are the portion of the sensor inserted into the soil.

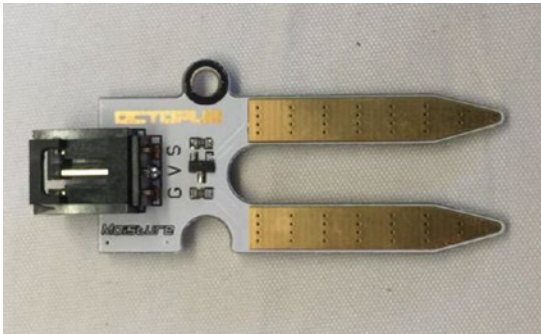


Figure 7-32. Soil Moisture Sensor

Proximity Sensors

Often thought of as distance sensors, proximity sensors use infrared or sound waves to detect distance, movement, or the range to/from an object. Made popular by low-cost robotics kits, the Parallax Ultrasonic Sensor uses sound waves to measure distance by sensing the amount of time between pulse sent and pulse received (the echo). For approximate distance measuring,⁹ it's a simple math problem to convert the time to distance. If you're building an IOT project that detects movement or proximity such as a motion sensing camera, you may want to use proximity sensors.

⁹Accuracy may depend on environmental variables such as elevation, temperature, and so on.

The following figures show two types of proximity sensors. Figure 7-33 shows a popular passive infrared sensor (PIR) motion sensor (www.sparkfun.com/products/13285).

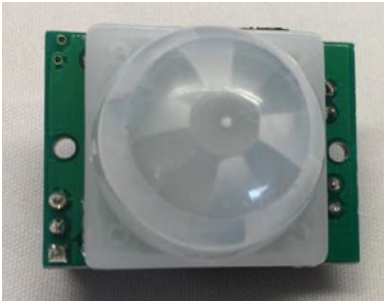


Figure 7-33. PIR Motion Sensor

Figure 7-34 shows an ultrasonic sensor (www.sparkfun.com/products/13959) used in many projects from robots to drones.

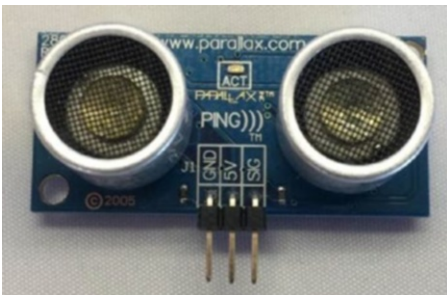


Figure 7-34. Ultrasonic Proximity Sensor

Radiation Sensors

Among the more serious sensors are those that detect radiation. This can also be electromagnetic radiation (there are sensors for that too), but a Geiger counter uses radiation sensors to detect harmful ionizing. In fact, it's possible to build your very own Geiger counter using a sensor and an Arduino (and a few electronic components). This is one sensor that you may not encounter as a hobbyist. However, there are several kits for building your own Geiger counter such as those from Adafruit (www.adafruit.com/products/483).

RFID Sensors

Radio frequency identification uses a passive device (sometimes called an RFID tag) to communicate data using radio frequencies through electromagnetic induction. For example, an RFID tag can be a credit-card-sized plastic card, a label, or something similar that contains a special antenna, typically in the form of a coil, thin wire, or foil layer that is tuned to a specific frequency.

When the tag is placed near the reader, the reader emits a radio signal; the tag can use the electromagnet energy to transmit a nonvolatile message embedded in the antenna, in the form of radio signals which is then converted to an alphanumeric string.¹⁰ RFID sensors are another good choice for security systems. If you have pets, you may want to visit your veterinarian to inquire about RFID sensors that act as hidden owner identification tags. If you know the frequency, you can even use it to help detect when your pet goes through a pet door. Figure 7-35 shows an RFID reader (sensor) that you can use to read RFID tags via USB (www.sparkfun.com/products/9963).

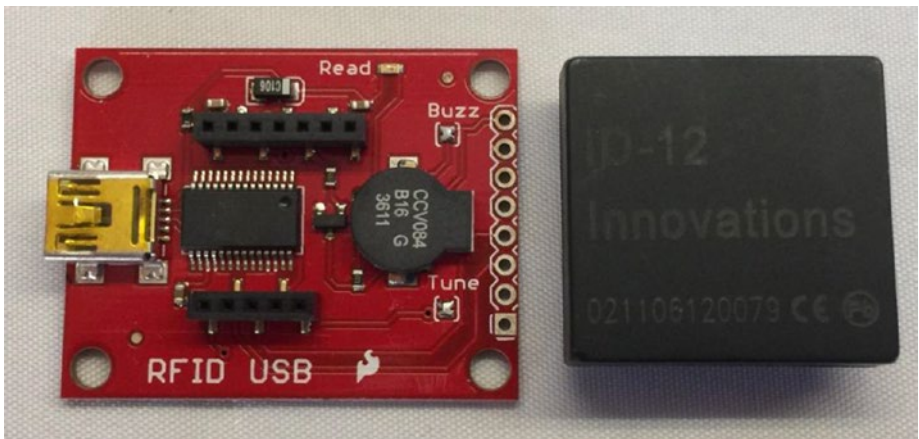


Figure 7-35. RFID Reader

Speed Sensors

Like flow sensors, simple speed sensors like those found on many bicycles use a magnet and a reed switch to generate a Hall effect. The frequency combined with the circumference of the wheel can be used to calculate speed and, over time, distance traveled. If your IOT solution needs to read movement, you can use a magnetic switch and a magnet to detect rotation. For example, bicycle speedometers often use a magnet and magnetic switch to detect the number of rotations, circumference of the wheel, and frequency of the actions to calculate speed.

¹⁰http://en.wikipedia.org/wiki/Radio-frequency_identification

Switches and Pushbuttons

These are the most basic of digital sensors used to detect if something is set (ON) or reset (OFF). Even so, you can use switches and buttons to build a user interface, for controlling other devices, or even turning the thing on!

Tilt Switches

These sensors can detect when a device is tilted one way or another. Although very simple, they can be useful for low-cost motion-detection sensors. They are digital and are essentially switches. If your IOT solution needs to detect when the device is leaning, you can use tilt sensors to trigger at a certain lean angle. For example, some modern motorcycles use tilt sensors to turn on cornering lights – headlamps angled to improve vision around a turn at night.

Touch Sensors

The touch-sensitive membranes formed into keypads, keyboards, pointing devices, and the like are an interesting form of sensor. You can use touch-sensitive devices like these for collecting data from humans. Touch sensors can help you build a user interface for your IOT project that can be presented in a low-profile form or to save space in a console, project box, etc.

Video Sensors

As mentioned previously, it's possible to obtain very small video sensors that use cameras and circuitry to capture images and transmit them as digital data. If you want to incorporate a video element to your IOT project such as a security solution, you can add a camera or video sensor to capture a visual component that can help provide information beyond the incident measurements. That is, you can review a photo and learn more than simply something moved or approached the device. For example, you can build an IOT project that detects movement and takes a photo if something gets close enough or, perhaps, moves faster than a certain threshold.¹¹

Weather Sensors

Sensors for temperature, barometric pressure, rain fall, humidity, wind speed, and so on are all classified as weather sensors. Most generate digital data and can be combined to create comprehensive environmental solutions. Figure 7-36 shows a common breakout board for the BMP280 pressure and temperature sensor (www.adafruit.com/products/2651).

¹¹Have you ever thought it would be great if you could catch a photo of whatever critter is eating your garden? Build your own critter camera with a proximity sensors and an infrared camera!



Figure 7-36. BMP280 Pressure and Temperature Sensor

With this and other easy-to-use sensors, it's possible to build your own weather station from about a dozen inexpensive sensors, your MicroPython board, and a bit of programming to interpret and combine the data. In fact, we'll do that in Chapter 10!

Tip – If you want to see more sensors, you can purchase any number of sensors from Adafruit (www.adafruit.com/category/35) or Sparkfun (www.sparkfun.com/categories/23).

I WANT TO LEARN MORE!

If you find you need or want to learn more about electronics than what I've presented in this chapter or you want to learn more about the electronics you will need for a more advanced IOT project, you may want to consider taking a course at a community college or try a self-paced course on electronics.

One of the best self-paced courses I've found include the set of electronics books by Charles Platt. I've found these books to be very well written, opening the door for many to learn electronics without having to spend years learning the tedious (but no less important) theory and mathematics of electronics. Best of all, they are not written in the dreary textbook fact-fact-fact-question pace. They are written by a world-renown expert with a gift of presenting the material in an easy-to-read and comprehend style. I recommend the following books for anyone wanting to learn more about electronics.

- *Make: Electronics, Second Edition* (O'Reilly, 2015), Charles Platt
- *Make: More Electronics* (O'Reilly, 2014), Charles Platt
- *Encyclopedia of Electronic Components Volume 1* (O'Reilly, 2012), Charles Platt

- *Encyclopedia of Electronic Components Volume 2* (O'Reilly, 2014), Charles Platt
- *Encyclopedia of Electronic Components Volume 3* (O'Reilly, 2016), Charles Platt

The third volume in his encyclopedia series includes an in-depth study of sensors: a must for advanced IOT projects.

Makershed (makershed.com/collections/electronics) sells companion kits that contain all the parts you need to complete the experiments in the *Make: Electronics* and *Make: More Electronics* books. The books together with the kits make for an excellent self-paced learning experience.

Summary

Learning how to work with electronics as a hobby or to create an IOT solution does not require a lifetime of study or a change of vocation. Indeed, learning how to work with electronics is all part of the fun of experimenting with the IOT! I have met many people who have learned electronics on their own and while most will admit formal study is essential for mastering the topic, you can learn quite a lot on your own - enough to become proficient working with the basic electronic components typically found in IOT projects.

This chapter presented the basics of electronics components including the use of breadboards, common components, and example circuits. This and a bit of key knowledge of how to use a multimeter will get you a long way toward becoming proficient with electronics. We also learned about one of the key components of an IOT solution - sensors. We discovered two ways they communicate (digital and analog) and a bit of what types of sensors are available.

In the next chapter, we will dive into our first electronics project - the equivalent of a "hello, world!" project for hardware. We'll see how to connect our MicroPython board to a few components and write a Python program to control them. Cool!

CHAPTER 8



Project 1: Hello, World! MicroPython Style

Here we are at the most fun part of this book - working on MicroPython projects! It is at this point that we have learned how to write in MicroPython and now know a lot more about the hardware and even how to use discrete electronics and breakout boards.

This chapter represents an introduction to building MicroPython projects. As such, there are a few more things we need to learn including techniques and procedures for installing and running our projects on our MicroPython boards. This chapter will introduce those things you need to make your MicroPython projects successful. Thus, the chapter is a bit longer and should be considered required reading even if you do not plan to implement the project.

To help achieve these goals and make things a bit easier (since this is your first real MicroPython project), we will forego connecting this project to the Internet so that we can keep the project easier to implement. Later chapters will have less introductory information and will focus on the project more as they will become more complex as we progress.

As you will see, the format for all the project chapters is the same: an overview of the project is presented followed by a list of the required components and how to assemble the hardware. Once we have a grasp of how to connect the hardware, we then see how to connect everything and begin writing the code. Each chapter will close with how to execute the project along with a sample of it running and suggestions for embellishing the project.

If you're wondering by now which board you need to complete these projects - don't worry as we will see how to implement the project in this chapter on both the WiPy and Pyboard along with demonstrations of the differences.

So, let's get started on our very first MicroPython project!

Overview

In this chapter, we will design and build a MicroPython clock. We will use both an SPI and I2C breakout board. We will use a small organic light-emitting diode (OLED)¹ display that uses an SPI interface and a hardware-based real-time clock (RTC) that uses the DS1307 chip and a battery for keeping time while the project is turned off. Rather than simply connecting to a network time protocol (NTP) server on the Internet, we will use the hardware-based RTC and display the current date and time on the small OLED display. This not only keeps the project a bit smaller but also demonstrates how to use a RTC for projects that may not be connected to the Internet.

While the Pyboard has a RTC circuit that you can connect an external battery to keep the RTC powered while the board is turned off (see the VBAT pin reference at <http://docs.micropython.org/en/latest/pyboard/pyboard/quickref.html>), most systems on a chip (SOC) boards, like the WiPy and other Espressif-based boards, do not support such a feature. Thus, we will use an external RTC so that you can use this project on other boards.

As you will see, there is a fair amount of wiring needed and understanding of the hardware capabilities is imperative to write the code, which is why we spent time in previous chapters talking about the firmware and various low-level hardware controls. You will need those skills and knowledge to complete this project.

While a clock may sound rather simple, this project will walk you through all the steps needed to assemble the hardware and write the code. As you will see, there are some distinct differences between the WiPy and Pyboard beyond the hardware layout and pin numbers. In fact, we will see there are some differences in how we write the code for each.

Further, the project is small and simplistic so we can focus on the process, which we can then apply to more advanced projects. In fact, we will see that even a relatively simple project can have an unexpected level of difficulty. But don't worry, as this chapter documents all the things you need to do to complete the project.

The sources for this project are many. The following links include background data used for this project. Learning how to apply knowledge from sites like these isn't something that is easily explained or learned; however, by looking at these sources (should you desire to do so), it should help you understand how to research and plan your own projects.

- *Source of project ideas and drivers:* <http://micropython-urtc.readthedocs.io/en/latest/examples.html>
- *OLED display information:* <https://learn.adafruit.com/monochrome-oled-breakouts/wiring-128x32-spi-oled-display>
- *OLED display driver:* <https://github.com/adafruit/micropython-adafruit-ssd1306>
- *RTC driver documentation:* <https://github.com/adafruit/Adafruit-uRTC/tree/master/docs>
- *RTC driver:* <https://github.com/adafruit/Adafruit-uRTC>

¹<https://en.wikipedia.org/wiki/OLED>

Notice the sites used. A good practice is to start with the Adafruit and MicroPython learning, blobs, and forums. Then check out the drivers. That is, do the research first and find all the references you can. If you find nice tutorials like those from Adafruit or Sparkfun, you may want to download them to your tablet or print them out for later reading.² More importantly, take the time to read the references so that you understand as much as you can before you start working with the hardware or writing your code. You can save yourself a lot of time by understanding simple things like how to wire your board to the device and how the driver is expected to be used.³

WHICH DRIVER DO I USE?

You may encounter a situation where you find more than one driver for the hardware you want to use. In fact, I found three drivers for the OLED display. The differences among them are subtle, and it appears at least one is written for a specific platform.

In fact, the one listed above is the best one to use. Even so, it needs some minor changes for use with the WiPy and Pyboard. I will show you those changes and, as you will see, they are not too difficult to spot and fix (for example, when MicroPython throws exceptions, it will show you the source of the issue).

If you encounter a similar situation – having more than one driver to choose from, you may want to try each until you find one that works best for your hardware and project. Sometimes, and in this case it is true, one driver may not be viable or another may be lacking features you need. For example, one of the drivers I found does not support text so it is not usable for this project and another needed major modification for use with the Pyboard. The trick is to find the driver that works best with the least amount of modification.

Now let's see what components are needed for this project, and then we will see how to wire everything together.

Required Components

Table 8-1 lists the components you will need. You can purchase the components separately from Adafruit (adafruit.com), Sparkfun (sparkfun.com), or any electronics store that carries electronic components. Links to vendors are provided should you want to purchase the components. When listing multiple rows of the same object, you can

²Yes, I am one of those that prefer the tactile sensor impressions of printed materials. Besides, you can fit a small stack of papers in almost any bag or pocket.

³Which is why I use only those references that are well written and have complete examples. Trying to guess the mind of a driver author is not worth the time or effort.

choose one or the other – you do not need both. Also, you may find other vendors that sell the components. You should shop around to find the best deal. Costs shown are estimates and do not include any shipping costs.

Table 8-1. *Required Components*

Component	Qty	Description	Cost	Links
MicroPython board	1	Pyboard v1.1 with headers	\$45-50	https://www.adafruit.com/product/2390 https://www.adafruit.com/product/3499 https://www.sparkfun.com/products/14413 https://store.micropython.org/store
		WiPy	\$25	https://www.adafruit.com/product/3338 https://www.pycom.io/product/wipy/
OLED display	1	ssd1306-based SPI display	\$18	https://www.adafruit.com/product/661
RTC breakout board	1	RTC module with battery backup	\$15	https://www.sparkfun.com/products/12708
Breadboard	1	Prototyping board, half-sized	\$5	https://www.sparkfun.com/products/12002
Jumper wires	12	M/M jumper wires, 6" (cost is for a set of 10 jumper wires)	\$4	https://www.sparkfun.com/products/8431
Power	1	USB cable to get power from PC		Use from your spares
	1	USB 5V power source and cable		Use from your spares
Coin cell battery	1	See RTC breakout board datasheet	\$3-5	Commonly available

Notice in the source, “use from your spares” for the last two items. This refers to the fact that most of us have these things on hand and in some cases in abundance. For example, if you buy a MicroPython board, you will need to also buy a cable if you don’t already have one. Further, with the proliferation of USB charged devices, it is hard to imagine anyone owning more than one smart device not having an extra USB charger or power supply lying around somewhere.

If you choose to use the WiPy, it is recommended you also purchase the Expansion Board from Adafruit (<https://www.adafruit.com/product/3344>) or Pycom (<https://www.pycom.io/product/wipy-expansion-board/>).

The OLED breakout board used in this project is a small module from Adafruit. It has a tiny but bright display that you can mount on the breadboard. The resolution is 128 pixels wide by 32 pixels high. The OLED breakout board comes without headers installed, but they are easy to add if you know how to solder (now might a good time to practice) or you can get a friend to help you. Figure 8-1 shows the Adafruit OLED SPI breakout board.



Figure 8-1. Monochrome 128x32 SPI OLED Graphic Display (courtesy of adafruit.com)

There are several OLED breakout boards available and so long as they have the SPI interface and use the `ssd1306` controller chip (the description will tell you this), you can use an alternate OLED display. The reason we need to use one with that controller chip is because the driver is written for that controller. Other controller chips will require a different driver.

The RTC breakout board used in this project is a DS1307 breakout board from Adafruit. The board also comes without headers installed (but includes them), nor does it come with a battery so you must purchase a CR1220 coin cell battery. Adafruit has those as well if you want to save yourself a trip to the store. Figure 8-2 shows the RTC breakout board.

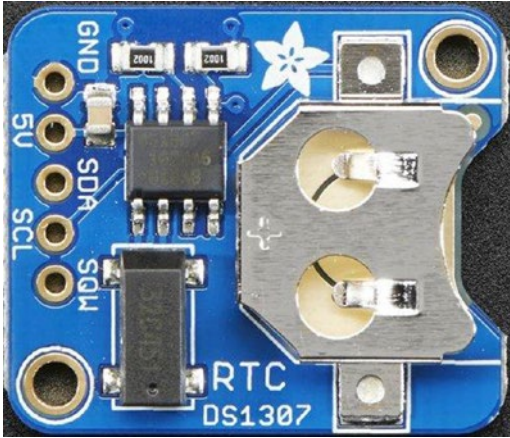


Figure 8-2. DS1307 Real-Time Clock Assembled Breakout Board (courtesy of adafruit.com)

There are several DS1307 RTC clocks available. In fact, Sparkfun has one or you can build your own! See the sidebar, “Building Your Own RTC Module” for more details. Fortunately, the library we will use supports breakout boards with DS1307, DS3231, or PCF8523 RTC chips.

■ **Tip** Small, discrete components like LEDs, resistors, etc., and even jumper wires and breadboards can be found in the kits mentioned in Chapter 2 - the Adafruit Parts Pal (<https://www.adafruit.com/product/2975>) or the Sparkfun Beginner Parts Kit (<https://www.sparkfun.com/products/13973>). I recommend one of these kits.

Now, let’s see how to wire the components together.

Set Up the Hardware

This project has a lot of connections. There are seven needed for the OLED and four needed for the RTC as well as a few additional jumpers for power and ground. To help keep things easier, we will plan for how things should connect. We will use a breadboard to mount the breakout boards making the connections easier. More specifically, we will learn what connections are needed for each component and where they need to be connected to our board, writing them down to keep things straight. Doing this small amount of homework will save you time later (and no small bit of frustration).

In this section and the next, we will see how to implement the project for both the WiPy and Pyboard. In the case of the hardware, the connections on the breakout board side are all the same but the pin numbers differ from one board to another. Both boards are shown in the table, which shows the MicroPython board on the left with the pin label and the breakout board on the right with the pin label on that board. We will use male/male jumper wires to make these connections via a breadboard.

As you will see, mapping out the connections like this makes it easy to check the connections. This table along with a wiring drawing are the tools you will see in this book and other example projects on the Internet or elsewhere. Thus, learning how to read maps and wiring drawings is a skill you should have to make your project successful.

Table 8-2 shows the connections needed for this project. Notice the wire color column. Use this column to record the color of the jumper wire to help manage the connections. It doesn't matter what color wire you use, but using a different color for each connection helps manage the many wires you will have connected. It also makes double-checking your connections easier if you write down which color wire you used for each connection. Traditionally, we use black for ground (negative), red for power (positive) at a minimum.

Table 8-2. *Connections for the MicroPython Clock (WiPy and Pyboard)*

MicroPython Board		Breakout Board		Wire Color
WiPy	Pyboard	Board	Pin Name	
P9	X9	RTC	SCL	
P8	X10	RTC	SDA	
5V	V+	RTC	5V	
GND	GND	RTC	GND	
P6	Y3	OLED	RST	
P5	Y4	OLED	D/C	
P7	Y5	OLED	CS	
P10	Y6	OLED	CLK	
P11	Y8	OLED	DATA	
3V3	3V3	OLED	VIN	
GND	GND	OLED	GND	

Wow, that's a lot of connections! Let's review some tips for wiring components. The best way to wire components to your board is to use a breadboard. As we saw in Chapter 7, a breadboard allows us to plug our components in and use jumper wires to make the connections. This simplifies wiring the project and allows you to move things around if you need to make more room. A breadboard also helps with wiring power and ground connections since the breadboard has power rails on either side that we can use as a common connection. That is, use one jumper to connect power and ground to the breadboard and the other jumper wires to the breakout boards.

When plugging in components, always make sure the pins are mounted parallel to the center channel. Recall breadboards have the pins wired together in rows perpendicular to the center channel. This allows you to make more than one connection to the component (or pin on the board).

■ **Caution** Never plug or unplug jumper wires when the project is powered on.

Finally, always make sure you wire your project carefully double-checking all the connections – especially power, ground, and any pins used for signaling (will be set to “high” or “on”) such as those pins used for SPI interfaces. Most importantly, never plug or unplug jumper wires when the project (or your board) is powered on. This will very likely damage your board or components.⁴

To get started, get out your breadboard and plug the components in first; then, using a different color jumper wire, plug all the needed wires into the breadboard and note the color you used in the chart. This will help you greatly when you begin plugging the other end of the jumpers into your MicroPython board.

For this project, I mounted the OLED on the left side of a half-sized breadboard just below the center channel and the RTC module on the right also below the channel. Notice the boards use a different power connection. The OLED board uses 3.3V and the RTC board 5V. Always check the power requirements of your components before powering on the project. Double-check and triple-check your connections.

Now, let us see how to make the connections shown in the chart for the WiPy and Pyboard.

WiPy

Wiring for the WiPy is best done orienting the board with the USB connector on the expansion board to the left. This will allow you to read the pin numbers on the board even after the wires are plugged into the board. However, this is only a suggestion. Orientation will not matter so long as the wires are connected correctly. Figure 8-3 shows the wiring drawing for the WiPy and the OLED and RTC breakout boards.

⁴Guess how I know this. Sometimes breaking things is the best way to learn.

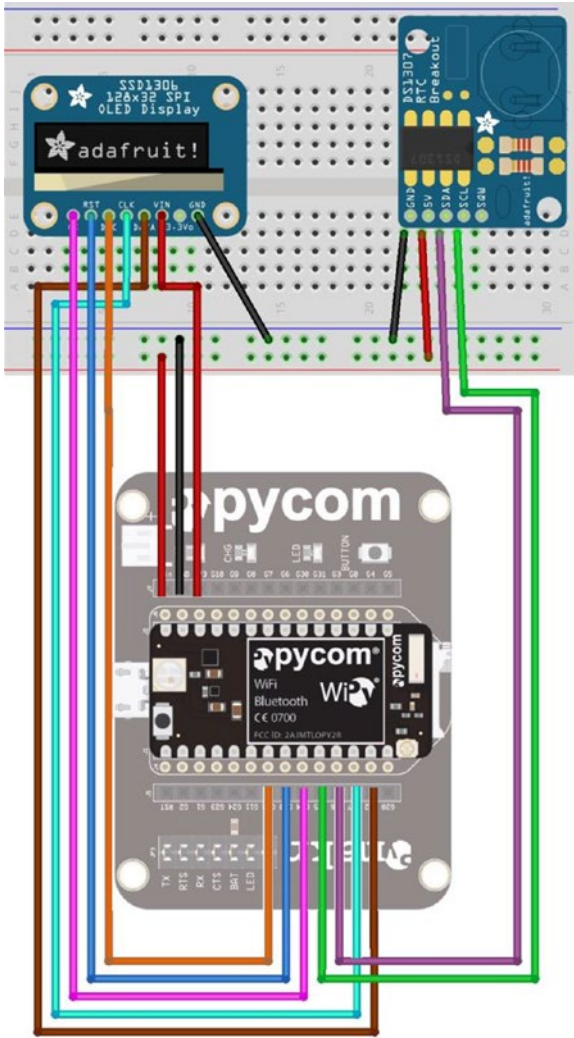


Figure 8-3. Wiring the Clock Project (WiPy)

Pyboard

Wiring for the Pyboard is best done orienting the board with the USB connector to the left. This will allow you to read the pin numbers on the board even after the wires are plugged into the board. Figure 8-4 shows the wiring drawing for the Pyboard and the OLED and RTC breakout boards.

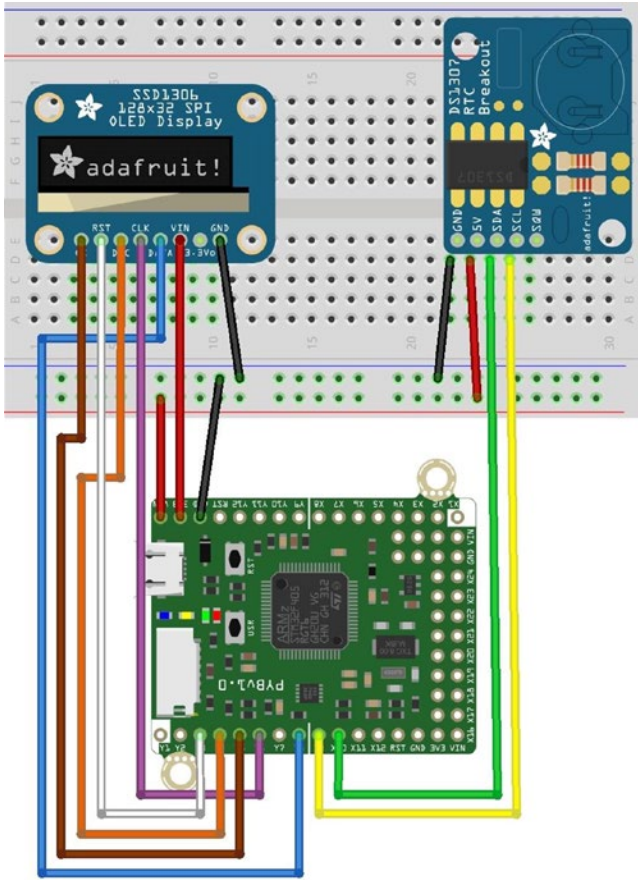


Figure 8-4. Wiring the Clock Project (Pyboard)

■ **Caution** Always double- and triple-check your connections, especially all power and ground connections. Be sure to examine the power connections to ensure the correct power (3V or 5V) is being connected to the components correctly. Connecting the wrong voltage can damage the component.

If you chose a different RTC board than the one shown in the drawing, be sure to adjust the connections as needed. For example, the Sparkfun DS1307 breakout board has the pins in a different order, so don't go by this drawing alone – especially if you use alternative components!

Once again, always make sure to double-check your connections before powering the board on. Now, let's talk about the code we need to write. Don't power on your board just yet – there is a fair amount of discussion needed before we're ready to test the project.

Write the Code

Now it's time to write the code for our project. Since we are working with several new components, I will introduce the code for each in turn. The code isn't overly complicated but may not be as clear as some of the code from previous projects. Let's begin with a look at the design of the project.

Design

Once you have the hardware sorted out and how to connect the components to your board, it is time to start designing the code.⁵ Fortunately, this project is small enough to make the design simple. In short, we want to display the time on our OLED once every second. Thus, the “work” of the code is to read the date and time from the RTC and then display it on the OLED. The following lists the steps that summarize how to design and implement the code for this project or any project for that matter.

1. *Libraries:* we will need to select and import libraries for the RTC and OLED
2. *Set Up:* we will need to set up the interfaces for I2C and SPI
3. *Initialize:* we will need to initialize object instances for the classes in the libraries
4. *Function:* we will need a new function to allow our project to be started and run on boot
5. *Main:* modify the `main.py` file on our MicroPython board to run our code

These five elements are what we will use for all the projects in this book and indeed, it is a good pattern to follow for all your MicroPython projects. The new function step is a new step that we haven't seen yet. In short, we wrap the operational portion of the code in a separate function to make it easy to call from the `main.py` file on boot. We'll see more about this later when we execute and test the project.

Now that we know how the project code will be implemented, let's review the libraries needed.

Libraries Needed

Recall from earlier we need two drivers: one for the OLED display, and another for the RTC. The driver for the OLED display can be found at <https://github.com/adafruit/micropython-adafruit-ssd1306> and the driver for the RTC can be found at <https://github.com/adafruit/Adafruit-uRTC>.

⁵I suppose other engineers may say you should write the code first, but I see it the other way around.

Go ahead and download both drivers now. You should be able to go to the sites and click on the *Clone or Download* link, then the *Download Zip* button to download the files to your PC. Then, open the location of the downloaded files and unzip them. You should find the following files. Copy these to a location on your PC. We copy them because we will need to modify them and copying them allows us to return to the original if we make a mistake or things don't work.

- `ssd1306.py`: the OLED display driver
- `urtc.py`: the RTC driver

There are two primary reasons we would have to modify a library; and in this case, we see examples of both. First, there are differences in the target platform for which the Adafruit driver was written. More specifically, it was written for a different MicroPython board, but we can easily adapt it to our needs. Second, there are differences between the firmware on the boards. This will require changing the driver code to use the correct classes and functions related to the specific MicroPython board.

Again, do not worry because the changes to the libraries are minor. But again, this is a common enough occurrence that you should get used to having to make minor changes like these – especially if you use a board other than a WiPy or Pyboard!

There is one other consideration when using drivers. You should check the driver documentation for any restrictions. For example, drivers may require a newer version of the firmware. In some cases, it may also require an older version of the firmware (but this is rare). If your driver doesn't work with your board and the document claims it does work with your board, check your firmware version in the documentation for any incompatibilities.

We will look at the general changes needed for the drivers for use on the WiPy and Pyboard in the next sections.

Changes to `ssd1306.py` for the WiPy

The OLED driver needs a small set of changes for the WiPy firmware. The `Pin` class on the WiPy does not have a `high()` or `low()` function. Instead, the WiPy firmware uses one function, `value()`, which we pass as value of 1 for high and 0 for low as a parameter. Fortunately, it is just a matter of changing all occurrences of `.high()` to `.value(1)` and `.low()` to `.value(0)`. For those of you who know how to read a difference file (the output of the `diff` command⁶), the difference file is shown in Listing 8-1.

Listing 8-1. Changes for the `ssd1306.py` code module for the WiPy (difference file)

```
--- ./Pyboard/ssd1306.py      2016-10-30 14:06:02.000000000 -0400
+++ ./WiPy/ssd1306.py       2017-07-20 21:39:31.000000000 -0400
@@ -146,23 +146,23 @@
```

⁶https://en.wikipedia.org/wiki/Diff_utility

```

def write_cmd(self, cmd):
    self.spi.init(baudrate=self.rate, polarity=0, phase=0)
-   self.cs.high()
-   self.dc.low()
-   self.cs.low()
+   self.cs.value(1)
+   self.dc.value(0)
+   self.cs.value(0)
    self.spi.write(bytearray([cmd]))
-   self.cs.high()
+   self.cs.value(1)

def write_framebuf(self):
    self.spi.init(baudrate=self.rate, polarity=0, phase=0)
-   self.cs.high()
-   self.dc.high()
-   self.cs.low()
+   self.cs.value(1)
+   self.dc.value(1)
+   self.cs.value(0)
    self.spi.write(self.buffer)
-   self.cs.high()
+   self.cs.value(1)

def poweron(self):
-   self.res.high()
+   self.res.value(1)
    time.sleep_ms(1)
-   self.res.low()
+   self.res.value(0)
    time.sleep_ms(10)
-   self.res.high()
+   self.res.value(1)

```

WHAT'S A DIFFERENCE FILE?

A difference file is the output of the `diff` command. It shows the line-by-line changes or differences between two versions of the same file. In a unified difference file, lines prefaced with a minus sign are those to be removed and those with a plus sign are those to be added. Lines from the file that have no preface (symbol) are used as context to locate the changes (along with a special header). The difference file can be used with the `patch` command to apply the changes to a file. Thus, the difference file is sometimes called a “patch” or “diff” file.

Changes to `uRTC.py` for the Pyboard

The RTC driver will not run correctly on the Pyboard (but works fine without changes on the WiPy). The changes needed are due to the differences in the functions for the I2C class. The driver uses the following methods to read and write data.

```
i2c.readfrom_mem(address, register, num_to_read)
i2c.writeto_mem(address, register, buffer)
```

However, the Pyboard firmware has different function names with a different order of parameters as shown below. Notice the order of parameters is different for the read function.

```
i2c.mem_read(num_read, address, register)
i2c.mem_write(buffer, address, register)
```

Thus, we need to change all the read and write functions in the library to match the firmware of our boards. All you need to do is open the file and make the changes – it is just a matter of renaming the functions and reordering the parameters for the read function. Listing 8-2 shows the changes you need to make.

Listing 8-2. Changes for the `urtc.py` code module for the Pyboard (difference file)

```
--- ./Pyboard/urtc.py    2017-07-20 19:20:38.000000000 -0400
+++ ./WiPy/urtc.py      2017-04-21 16:52:32.000000000 -0400
@@ -40,8 +40,8 @@

    def _register(self, register, buffer=None):
        if buffer is None:
-           return self.i2c.mem_read(1, self.address, register)[0]
-           self.i2c.mem_write(buffer, self.address, register)
+           return self.i2c.readfrom_mem(self.address, register, 1)[0]
+           self.i2c.writeto_mem(self.address, register, buffer)

    def _flag(self, register, mask, value=None):
        data = self._register(register)
@@ -56,8 +56,8 @@

    def datetime(self, datetime=None):
        if datetime is None:
-           buffer = self.i2c.mem_read(7, self.address,
-                                     self._DATETIME_REGISTER)
+           buffer = self.i2c.readfrom_mem(self.address,
+                                         self._DATETIME_REGISTER, 7)

            if self._SWAP_DAY_WEEKDAY:
                day = buffer[3]
                weekday = buffer[4]
```

```

@@ -128,8 +128,8 @@
    def alarm_time(self, datetime=None, alarm=0):
        if datetime is None:
-           buffer = self.i2c.mem_read(3, self.address,
+           buffer = self.i2c.mem_read(3, self.address,
+           self._ALARM_REGISTERS[alarm])
+           buffer = self.i2c.readfrom_mem(self.address,
+           self._ALARM_REGISTERS[alarm], 3)
            day = None
            weekday = None
            second = None
@@ -145,8 +145,8 @@
                if not buffer[1] & 0x80 else None)
        if alarm == 0:
            # handle seconds
-           buffer = self.i2c.mem_read(1,
-           self.address, self._ALARM_REGISTERS[alarm] - 1)
+           buffer = self.i2c.readfrom_mem(
+           self.address, self._ALARM_REGISTERS[alarm] - 1, 1)
            second = (_bcd2bin(buffer[0] & 0x7f)
                if not buffer[0] & 0x80 else None)
        return datetime_tuple(
@@ -219,8 +219,8 @@
    def alarm_time(self, datetime=None):
        if datetime is None:
-           buffer = self.i2c.mem_read(4, self.address,
+           buffer = self.i2c.mem_read(4, self.address,
+           self._ALARM_REGISTER)
+           buffer = self.i2c.readfrom_mem(self.address,
+           self._ALARM_REGISTER, 4)
        return datetime_tuple(
            weekday=_bcd2bin(buffer[3] &
                0x7f) if not buffer[3] & 0x80 else None,

```

Fortunately, there are not that many changes you need to make so it is easy and quick to make the changes. If you know how to apply the difference file with `patch`,⁷ you can do that and all the changes will be made for you. Hint: you will need to use the `patch` command. Otherwise, just open the file and make the changes by hand. Just find all the rows marked with a minus sign (-) and change them to match the rows with the plus sign (+). If you copy and paste, don't forget to remove the plus sign.

Changes to `ssd1306.py` for the Pyboard

The OLED driver will also not run correctly on the Pyboard. The changes needed are due to the same differences in the functions for the I2C class as we saw in the RTC driver. You may be wondering why we need to make these changes if we're using the SPI interface.

⁷[https://en.wikipedia.org/wiki/Patch_\(Unix\)](https://en.wikipedia.org/wiki/Patch_(Unix))

As it turns out, the driver will work for either the I2C or SPI interfaces and since the I2C interface is different, the changes still must be made. If we do not, MicroPython will complain and we won't be able to use the driver. Yes, even if we don't use that part of the code. This is because MicroPython will check syntax of the entire code module that is imported.

Like the RTC driver, it is simply a matter of opening the file and making the changes. Listing 8-3 shows the difference file to making the RTC driver compatible for the WiPy and Pyboard.

Listing 8-3. Changes for the `ssd1306.py` code module for the Pyboard (difference file)

```

--- /Users/cbell/Downloads/Adafruit-uRTC-master_orig/urtc.py    2017-04-21
16:52:32.000000000 -0400
+++ /Users/cbell/Downloads/Adafruit-uRTC-master/urtc.py        2017-07-20
19:20:38.000000000 -0400
@@ -40,8 +40,8 @@

    def _register(self, register, buffer=None):
        if buffer is None:
-           return self.i2c.readfrom_mem(self.address, register, 1)[0]
-           self.i2c.writeto_mem(self.address, register, buffer)
+           return self.i2c.mem_read(1, self.address, register)[0]
+           self.i2c.mem_write(buffer, self.address, register)

    def _flag(self, register, mask, value=None):
        data = self._register(register)
@@ -56,8 +56,8 @@

    def datetime(self, datetime=None):
        if datetime is None:
-           buffer = self.i2c.readfrom_mem(self.address,
-                                         self._DATETIME_REGISTER, 7)
+           buffer = self.i2c.mem_read(7, self.address,
+                                     self._DATETIME_REGISTER)
+
            if self._SWAP_DAY_WEEKDAY:
                day = buffer[3]
                weekday = buffer[4]
@@ -128,8 +128,8 @@

    def alarm_time(self, datetime=None, alarm=0):
        if datetime is None:
-           buffer = self.i2c.readfrom_mem(self.address,
-                                         self._ALARM_REGISTERS[alarm], 3)
+           buffer = self.i2c.mem_read(3, self.address,
+                                     self._ALARM_REGISTERS[alarm])
+
            day = None
            weekday = None
            second = None

```



```

@@ -145,8 +145,8 @@
        if not buffer[1] & 0x80 else None)
    if alarm == 0:
        # handle seconds
-       buffer = self.i2c.readfrom_mem(
-           self.address, self._ALARM_REGISTERS[alarm] - 1, 1)
+       buffer = self.i2c.mem_read(1,
+           self.address, self._ALARM_REGISTERS[alarm] - 1)
        second = (_bcd2bin(buffer[0] & 0x7f)
            if not buffer[0] & 0x80 else None)
    return datetime_tuple(
@@ -219,8 +219,8 @@

    def alarm_time(self, datetime=None):
        if datetime is None:
-           buffer = self.i2c.readfrom_mem(self.address,
-               self._ALARM_REGISTER, 4)
+           buffer = self.i2c.mem_read(4, self.address,
+               self._ALARM_REGISTER)
        return datetime_tuple(
            weekday=_bcd2bin(buffer[3] &
                0x7f) if not buffer[3] & 0x80 else None,

```

Now that we have the libraries modified to work on our boards, let's look at the code we will need to write.

Planning the Code

Now that we have our design and have downloaded and modified the libraries, we can begin writing the code. Rather than show you a long listing and say “comprehend or perish,” let's walk through all the parts of the code first so that we understand each part. As we walk through the code, feel free to test the parts yourself, but if you prefer to wait until the end to test the code, you can. Also, as we walk through the code, we will see the differences needed for the WiPy and Pyboard. Let's begin with a look at the imports section.

Imports

The imports section for the project comes before all other statements but after the comment block at the top of the file. As you have seen in previous examples and in Chapter 4, you should include some level of documentation at the top of the file to explain what the code does. You don't have to write a lengthy tutorial – just a short statement or so that describes the program including your name and other information. This is important if you want to share your code with others and if you ever go back to the code later to reuse it.⁸

⁸I sometimes find myself wondering who wrote a piece of code only to realize it was me! Documenting or “signing” the code may help you remember what you wrote and why.

If you want to type in the code as we go along, you can open a new file named `clock.py` with your favorite code (or text) editor. Recall the best editors are those that feature Python syntax checking like Komodo Edit.

The imports for the WiPy are different than what we need for the Pyboard; however we need the same libraries – we just import them from different places. The following shows the imports for the WiPy. This is one of three areas that are different for the two boards.

```
import urtc
import utime
from machine import SPI, I2C, RTC as rtc, Pin as pin
from ssd1306 import SSD1306_SPI as ssd
```

The imports we need for the Pyboard include the following. Notice we need a lot of libraries. You will see we import the Pyboard library, the RTC library, `ssd1306` library, and the I2C and SPI classes. Notice also I use an alias (the “as XXXX” phrase) for some of the libraries to make typing a bit easier (fewer characters to type).

```
# Imports for the project
import pyb
import urtc
from machine import SPI, Pin as pin
from pyb import I2C
from ssd1306 import SSD1306_SPI as ssd
```

Set Up

Next, we need to set up the interfaces for I2C and SPI for use in the RTC and `ssd1306` libraries. That is, the classes in those libraries need object instances of the interfaces passed to the constructor. The code we will use is like the code we saw in previous examples.

This is another major area of difference between the WiPy and Pyboard. The following shows the interface setup code for the WiPy.

```
# Setup SPI and I2C
spi = SPI(0, SPI.MASTER, baudrate=2000000, polarity=0, phase=0)
```

`i2c = I2C(0, I2C.MASTER, baudrate=100000, pins=("P9", "P8"))` The following shows the setup for I2C and SPI for the Pyboard.

```
spi = SPI(2, baudrate=8000000, polarity=0, phase=0)
i2c = I2C(1, I2C.MASTER)
i2c.init(I2C.MASTER, baudrate=500000)
```

Notice we use different parameters for the SPI and we specify pins for the I2C. The reason we must specify pins for the I2C interface is because the clock pin is shared between SPI and I2C on the WiPy. Thus, we must manually specify the pins to use for the I2C interface. You can use other pins if you'd like, but just remember to use the correct pins when you wire the components together. Finally, notice we do not need the `init()` function on the WiPy.

Initialize

Next, we initialize object instances for the classes in the libraries. This is the point where you need to read the documentation for each library to understand what is needed to initialize the objects. For the `ssd1306` driver, the class constructor requires the number pixels (resolution is number of pixels in rows, columns) for the display, the interface instance (SPI from the last section), and the pins we will use for the D/C, RST, and CS pins. For the RTC driver, we need only pass in interface instance (I2C from the last section). The following shows how to do both steps for the WiPy.

```
# Setup the OLED : D/C, RST, CS
oled_module = ssd(128,32,spi,pin('P5'),pin('P6'),pin('P7'))

# Setup the RTC
rtc_module = urtc.DS1307(i2c)
```

The initialize code is very similar for the Pyboard. The only difference are the pins we specify.

```
# Setup the OLED : D/C, RST, CS
oled_module = ssd(128,32,spi,pin("Y4"),pin("Y3"),pin("Y5"))

# Setup the RTC
rtc_module = urtc.DS1307(i2c)
```

There is one other thing we need to consider. When we first use the RTC or when we replace the battery, we must initialize the date and time. We can use the driver features to do this. In this case, we simply call the `datetime()` function for the RTC instance passing in a tuple containing the new start date and time - the order of the tuple elements are shown below. Once set, we do not need to run it again. In fact, running it again will reset the RTC and we don't need to do that. Thus, we leave this code commented out for normal operation and uncomment it when we need to reset the RTC. The following shows the code needed. When you run your project for the first time, uncomment this code supplying the correct current date and time but later comment it out.

```
#         (year, month, day, weekday, hour, minute, second, millisecond)
#start_datetime = (2017,07,20,4,9,0,0,0)
#rtc_module.datetime(start_datetime)
```

New Function

Now that all the setup or boiler plate work is done, we can create a new function to allow our project to be started and run on boot. That is, a function we can call once we import the code module. For example, we named the file `clock.py` and if we create a function named `run()`, we can start the code with the following statements. We will see more about how to use these statements when we deploy the project to the board.

```
import clock
clock.run()
```

Recall we want the project to read the date and time from the RTC and display it on the OLED once every second. Thus, we expect to see some sort of loop that performs these two steps. However, we must again refer to the driver documentation where we find that the RTC returns data as a tuple (year, month, day, weekday, hour, minute, second, millisecond). This means we must format the date and time to make it easier for humans to read and to fit on the small OLED screen.

This is a perfect candidate for a helper function or three. Fortunately, these functions are the same for both boards so there is no need to anticipate any changes.

Let's create a function named `write_time()` that takes an instance of the OLED display and the RTC and then read the date and time with the `datetime()` function (with no parameters) and print it to the OLED screen using the `text()` function, which takes a starting column (called the X position in the documentation) and row (Y position) for the location on the screen to print the message when the `show()` function is called. This is the essence of the project. Placing it in a separate function allows you to isolate the behavior and make it easier to maintain or modify the code – because the “core” is in one place.

```
# Display the date and time
def write_time(oled, rtc):
    # Get datetime
    dt = rtc.datetime()
    # Print the date
    oled.text("Date: {0:02}/{1:02}/{2:04}".format(dt[1], dt[2], dt[0]), 0, 0)
    # Print the time
    oled.text("Time: {0:02}:{1:02}:{2:02}".format(dt[4], dt[5], dt[6]), 0, 10)
    # Print the day of the week
    oled.text("Day: {0}".format(get_weekday(dt[3])), 0, 20)
    # Update the OLED
    oled.show()
```

Notice we use the `print()` function and the `format()` function to take the data from the RTC and format it in an expected format that most clocks use: `HH:MM::SS` and `MM/DD/YYYY`. Notice there is an additional function here named `get_weekday()`. This function takes the number of the day of the week as returned from the RTC and returns a string for the name of the day. The following shows the code for this function.

```
# Return a string to print the day of the week
def get_weekday(day):
    if day == 1: return "Sunday"
    elif day == 2: return "Monday"
    elif day == 3: return "Tuesday"
    elif day == 4: return "Wednesday"
    elif day == 5: return "Thursday"
    elif day == 6: return "Friday"
    else: return "Saturday"
```

There is one more function added – a function to clear the screen. This function simply blanks the screen to allow us to overwrite the screen with new data. Normally this is not needed, but it is a good practice to clear the screen in case the driver doesn't do it for you. In this case, it does now. This function is named `clear_screen()` and is shown below. It simply uses the `fill()` and `show()` functions from the `ssd1306` driver. Passing in `o` for the `fill()` function tells the driver to fill the screen with no data (blank or off).

```
# Clear the screen
def clear_screen(oled):
    oled.fill(0)
    oled.show()
```

Now we are ready to code the new `run()` function for the project. We have our helper functions developed so we need only call them and wait for a second on each pass. The following shows the `run()` function for our project. This is the last area that differs from the Pyboard to the WiPy. The following is code for the Pyboard. Can you spot the one line that needs to be changed for the WiPy?

```
# Here, we make a "run" function to be used from the main.py
# code module. This is preferable to direct "main" execution.
def run():
    # Display the deate and time every second
    while True:
        clear_screen(oled_module)
        write_time(oled_module, rtc_module)
        pyb.delay(1000)
```

Notice how “clean” this function is – we can see only three statements: clear the screen, show the time, and wait for one second. That one line of code that needs to be changed for the WiPy is the last. On the WiPy, we use a different class as shown below.

```
utime.sleep(1)
```

Now that we've seen a complete walkthrough of the parts of the code, let's talk about testing the code.

Test the Parts of the Code

Now that we have planned the code and know how to code each of the parts, we have one more thing to do – test the breakout boards separately. We do this by wiring one breakout board and testing it, then powering off and unwiring that breakout board, then wiring the other breakout board and testing it.

This is a good practice to get into the habit of doing for one primary reason. You will save yourself a lot of grief by testing the individual parts of the project – especially the hardware – one at a time. This not only makes it easier to narrow down any issues, it also ensures you can identify the source of the problem. That is, if you plugged all the hardware in and wired everything and wrote the code, deployed it, then powered it on and nothing works, how do you know which part is to blame? This is one of my mantras: build and test one piece at a time.

■ **Tip** Testing code one part at a time is a familiar pattern to me and it is highly recommended you adopt the process yourself. That is, coding a part of the project at a time and testing each individually.

For this project, there are two parts – the RTC and the OLED. Let's see how to test them individually. We will see code to test the components for the Pyboard. You can modify the code as needed using the information above to work on your own board. The code presented is intended to be run via a REPL console. I show the code for the Pyboard and leave changing it to run on the WiPy as an exercise. Hint: look back at the previous sections to see the changes needed.

Test the RTC Breakout Board

To test the RTC, use the following code. This is a condensed form of the code we saw in the walkthrough for the Pyboard.

```
import urtc
from pyb import I2C
from machine import Pin as pin
i2c = I2C(1, I2C.MASTER)
i2c.init(I2C.MASTER, baudrate=500000)
i2c.scan()
rtc = urtc.DS1307(i2c)
# year, month, day, weekday, hour, minute, second, millisecond)
start_datetime = (2017,07,20,4,7,25,0,0)
rtc.datetime(start_datetime)
print(rtc.datetime())
```

Notice we set the date and time in this test. When you run this for yourself, you should change the date and time tuple to include the current date and time when you run the test. What you should see in the REPL console is a tuple representing the date and time. It should be the same as what you set since the code will execute much less than a second from the time you set it to the time you query the RTC. Go ahead and reenter that last statement several times to ensure the time changes as you'd expect. That is, wait a few seconds and try it again – several seconds should have elapsed.

If any of the statements fail, be sure to check your wiring and look for any typos. Also, ensure you are using the correct, modified version of the drivers (and that you have copied them to the board).

Test the OLED Breakout Board

To test the OLED, use the following code. This is a condensed form of the code we saw in the walkthrough.

```
import machine
from machine import Pin as pin
from ssd1306 import SSD1306_SPI as ssd
spi = machine.SPI(2, baudrate=8000000, polarity=0, phase=0)
oled = ssd(128,32,spi,pin("Y4"),pin("Y3"),pin("Y5"))
oled.fill(0)
oled.show()
oled.fill(1)
oled.show()
oled.fill(0)
oled.show()
oled.text("Hello, World!", 0, 0)
oled.show()
```

When you run this code, you should see the screen blank (it should be blank from the start), then fill with white – see `fill(1)` – then blank the screen and finally display the text message. If you do not see any output, power off your board, check all the connections, verifying the correct pins are used, and that you have the correct modified version of the driver copied to your board.

■ **Tip** The OLED breakout boards from Adafruit (and presumably others) comes with a protective cover over the lens. You can and should leave that in place to ensure the lens does not get damaged. Plus, the OLED is bright enough to see through the protective cover.

Now, let's look at the completed code for both boards before we deploy it to our MicroPython board.

Completed Code

In this section, we will see the completed code for the WiPy and Pyboard. These listings are provided as a reference for you to ensure you have the correct code for your board. Later projects will show the final code for one of the boards and notes on how to modify it for use on other boards. Listing 8-4 shows the complete code for running the project on the WiPy.

Listing 8-4. MicroPython Clock Code Module `clock.py` (WiPy)

```
# MicroPython for the IOT - Chapter 8
#
# Project 1: A MicroPython Clock!
#
# Required Components:
# - Pyboard
# - OLED SPI display
# - RTC I2C module
#
# Note: this only runs on the WiPy. See chapter text
#       for how to modify this to run on the Pyboard
#

# Imports for the project
import urtc
import utime
from machine import SPI, I2C, RTC as rtc, Pin as pin
from ssd1306 import SSD1306_SPI as ssd

# Setup SPI and I2C
spi = SPI(0, SPI.MASTER, baudrate=2000000, polarity=0, phase=0)
i2c = I2C(0, I2C.MASTER, baudrate=100000, pins=("P9", "P8"))

# Setup the OLED : D/C, RST, CS
oled_module = ssd(128,32,spi,pin('P5'),pin('P6'),pin('P7'))

# Setup the RTC
rtc_module = urtc.DS1307(i2c)
#
# NOTE: We only need to set the datetime once. Uncomment these
#       lines only on the first run of a new RTC module or
#       whenever you change the battery.
#       (year, month, day, weekday, minute, second, millisecond)
#start_datetime = (2017,07,20,4,9,0,0,0)
#rtc_module.datetime(start_datetime)
```



```

# Clear the screen
def clear_screen(oled):
    oled.fill(0)
    oled.show()

# Return a string to print the day of the week
def get_weekday(day):
    if day == 1: return "Sunday"
    elif day == 2: return "Monday"
    elif day == 3: return "Tuesday"
    elif day == 4: return "Wednesday"
    elif day == 5: return "Thursday"
    elif day == 6: return "Friday"
    else: return "Saturday"

# Display the date and time
def write_time(oled, rtc):
    # Get datetime
    dt = rtc.datetime()
    # Print the date
    oled.text("Date: {0:02}/{1:02}/{2:04}".format(dt[1], dt[2], dt[0]), 0, 0)
    # Print the time
    oled.text("Time: {0:02}:{1:02}:{2:02}".format(dt[4], dt[5], dt[6]), 0, 10)
    # Print the day of the week
    oled.text("Day: {0}".format(get_weekday(dt[3])), 0, 20)
    # Update the OLED
    oled.show()

# Here, we make a "run" function to be used from the main.py
# code module. This is preferable to direct "main" execution.
def run():
    # Display the deate and time every second
    while True:
        clear_screen(oled_module)
        write_time(oled_module, rtc_module)
        utime.sleep(1)

```

Now, let's see the completed code for the Pyboard. As mentioned, the changes are mostly in the imports and set up with one minor change in the run function. Listing 8-5 shows the complete code for running the project on the Pyboard.

Listing 8-5. MicroPython Clock Code Module `clock.py` (Pyboard)

```

# MicroPython for the IOT - Chapter 8
#
# Project 1: A MicroPython Clock!
#
# Required Components:
# - Pyboard
# - OLED SPI display
# - RTC I2C module
#
# Note: this only runs on the Pyboard. See chapter text
#       for how to modify this to run on the WiPy
#

# Imports for the project
import pyb
import urtc
from machine import SPI, Pin as pin
from pyb import I2C
from ssd1306 import SSD1306_SPI as ssd

# Setup SPI and I2C
spi = SPI(2, baudrate=8000000, polarity=0, phase=0)
i2c = I2C(1, I2C.MASTER)
i2c.init(I2C.MASTER, baudrate=500000)

# Setup the OLED : D/C, RST, CS
oled_module = ssd(128,32,spi,pin("Y4"),pin("Y3"),pin("Y5"))

# Setup the RTC
rtc_module = urtc.DS1307(i2c)
#
# NOTE: We only need to set the datetime once. Uncomment these
#       lines only on the first run of a new RTC module or
#       whenever you change the battery.
#       (year, month, day, weekday, hour, minute, second, millisecond)
#start_datetime = (2017,07,20,4,9,0,0,0)
#rtc_module.datetime(start_datetime)

# Clear the screen
def clear_screen(oled):
    oled.fill(0)
    oled.show()

# Return a string to print the day of the week
def get_weekday(day):
    if day == 1: return "Sunday"

```

```

elif day == 2: return "Monday"
elif day == 3: return "Tuesday"
elif day == 4: return "Wednesday"
elif day == 5: return "Thursday"
elif day == 6: return "Friday"
else: return "Saturday"

# Display the date and time
def write_time(oled, rtc):
    # Get datetime
    dt = rtc.datetime()
    # Print the date
    oled.text("Date: {0:02}/{1:02}/{2:04}".format(dt[1], dt[2], dt[0]), 0, 0)
    # Print the time
    oled.text("Time: {0:02}:{1:02}:{2:02}".format(dt[4], dt[5], dt[6]), 0, 10)
    # Print the day of the week
    oled.text("Day: {0}".format(get_weekday(dt[3])), 0, 20)
    # Update the OLED
    oled.show()

# Here, we make a "run" function to be used from the main.py
# code module. This is preferable to direct "main" execution.
def run():
    # Display the deate and time every second
    while True:
        clear_screen(oled_module)
        write_time(oled_module, rtc_module)
        pyb.delay(1000)

```

Ok, now we're ready to execute the project.

Execute!

We are finally at the point where we can copy all the files to our board and execute the project code. There are several recommended steps in this process as shown below. You should follow this process each time you want to deploy and test a project.

1. *Connections*: Double-check all hardware connections
2. *Power*: Power on the board
3. *Copy Files*: Copy the drivers and code files to the board
4. *Test*: Use a REPL console to test the code, fix any issues, and recopy the file(s)
5. *Execute*: When satisfied, modify `main.py` and then reboot the board

The first step cannot be overstated. Always check your connections every time you power on the board. This is in case curious hands have wandered by and “examined” your project or you’ve moved it or some other event has occurred to unplug wires. It never hurts to be extra careful.

Next, we power on the board and check for any issues. Yes, this is the smoke test! Simply make sure all LEDs that are supposed to illuminate do (like those on the board) and that things that should not be on are off. For example, if you see a solid bar on the OLED when you power it on, that’s not a good sign. If ever in doubt, pull the power and check your connections. If things still aren’t right, disconnect everything and test your board. Sometimes a damaged component can cause strange behavior.

Next, we copy all the drivers and code we want to use to the board. When that is done, we can test the code. Since we used a `run()` function to contain the main code, we can simply import the code and call that function as follows.

```
>>> import clock
>>> clock.run()
```

Go ahead and power on your board now and run the code as shown. You should see something like Figure 8-5, which shows the project running in all its glory.

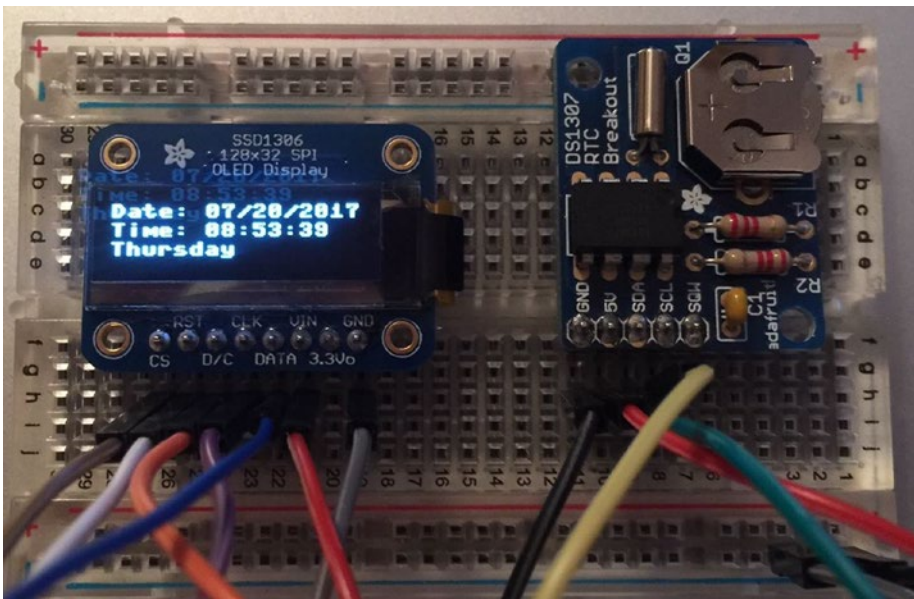


Figure 8-5. A MicroPython Clock!

It is at this point that you should bask in the wonder of your first successful MicroPython hardware project. Take some time to bask in your delight of a job well done.

The final step in the process is to make your project run by default when you power on your board. This is an optional step and one that you only need to take if you want to make your project run every time you power on the board. That is, you will dedicate the board for use in the project for some time. Recall the `main.py` code module on the board will execute on boot, and thus your project will execute when the board is powered on. All you need to do is add the code to that file to start your project as follows. Go ahead and make the changes now (you can always undo the changes later).

```
# main.py -- put your code here!
import clock
clock.run()
```

And now for the ultimate test: if you've modified the `main.py` file, power off your board then power it on again. If the date and time show up after a few seconds, you've done it! You have successfully created a project you can package and run anywhere you want and, so long as the coin cell battery has a charge, it won't lose time.

Taking It Further

This project has a lot of potential for embellishment. If you liked the project, you should consider taking time to explore some embellishments. Here are a few you may want to consider. Some are easy and some may be a challenge.

- Use a different RTC
- Calculate AM/PM and display it
- Connect your board to the Internet and use an NTP service instead of the RTC
- Use a larger display and display the Julian date
- Use a light sensor to turn off or dim the display in direct sunlight
- Add a speaker and implement an alarm feature (hint: some RTCs have this feature)
- Format the date and time using different world standards such as YYYY/MM/DD

Of course, if you want to press on to the next project, you're welcome to do so but take some time to explore these potential embellishments – it will be good practice.

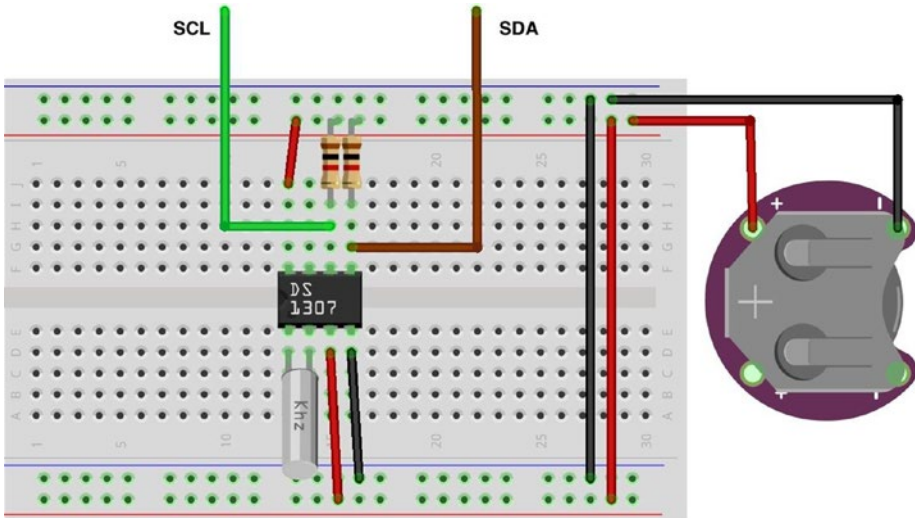
If you're thinking this project is rudimentary now that we have solved the problems with the libraries, consider this: most sensor-based projects and indeed most projects that generate data must be associated with a date and time when the events are sampled. Thus, using a RTC to read the date and time will be a consideration for many IOT projects.

BUILDING YOUR OWN RTC MODULE

If you're like me and like to tinker, you can build your own RTC module using a RTC DS1307 chip, two resistors, a crystal, and a coin cell battery breakout board. You can find these components at most online electronics stores such as Adafruit (www.adafruit.com), Sparkfun (www.sparkfun.com), and Mouser (www.mouser.com). The component list is as follows.

- DS1307 chip
- Coin cell battery breakout board
- 3v coin cell battery
- 32.768kHz Crystal
- (2) 1K Resistor

That's it! The following shows how to connect the components on a breadboard. If you want to implement this side project, see <http://www.learningaboutelectronics.com/Articles/DS1307-real-time-clock-RTC-circuit.php> for an example walkthrough.



If you plan to build a lot of projects that use a RTC, buying these components in bulk and wiring up your own RTC 1307 module may be more cost effective. Plus, it ups the cool factor of your kit.

Summary

Working with hardware such as breakout boards and the drivers we need to talk to them over specialized interfaces such as I2C and SPI can be a challenge. Sometimes, like we saw in this chapter, you need to adapt a driver for use with your MicroPython board. The reason for this is because of the growing array of boards, vendors are creating specialized versions of the MicroPython firmware for use on those boards. In some cases, like the `ssd1306` driver, we only need to make some minor changes to use the driver on our boards. The trick, then, is understanding why the changes are necessary and taking the time to make the changes yourself. It is so easy to just give up when something doesn't work – don't do that! Take your time and understand the problem, and then solve it systematically.

In this chapter, we saw a detailed walkthrough of a MicroPython clock. We used an OLED display to display time we read from a RTC. Along the way, we learned how to plan our projects, make hardware connections, and write code for use in deploying on our MicroPython boards.

In the next chapter, we will explore a project that uses more low-level hardware in the form of discrete components such as LEDs, resistors, and buttons. These are the building blocks you will need to form more complex solutions.

CHAPTER 9



Project 2: Stoplight Simulator

Now that we've had a tutorial of how to design, wire, and implement a MicroPython project, let's now look at a more advanced project. In this case, we will use some very basic components to learn further how to work with hardware. The hardware of choice for this project will be LEDs, resistors, and a button. A button is the most basic of sensors. That is, when the button is pressed, we can make our MicroPython code respond to that action.

Working with LEDs is perhaps more of a "Hello, World!" style project for hardware because turning LEDs on and off is easy and except for figuring out what size of current limiting resistor is needed, wiring LEDs is also easy. Like most LED-based projects, we will be implementing a simulation. More specifically, we will implement a traffic light and a pedestrian walk button. The walk button is a button that pedestrians can use to trigger the traffic signal to change and stop traffic so they can cross the street.

Simulation projects can be a lot of fun because we already have an idea of how it should work and we are simulating something we have encountered. For example, unless you've lived in a very rural area, you most likely have encountered a traffic signal at an intersection that included walk/don't walk signs with a button. If you live in the city, you will have encountered these in various configurations. When a pedestrian (or bicyclist) presses the walk button, the traffic lights all cycle to red and the walk sign is illuminated. After some time (30 seconds or so), the walk sign flashes and then about 15 seconds later, the walk signal cycles to don't walk and the traffic signals resume their normal cycle.

■ **Note** The word "cycle" refers to a set of states that are linear in action. Thus, cycle refers to the changing of one state to another.

We will also add a new twist to the traffic light simulation concept. We will use a web page to trigger the walk request. But first, we will learn how to wire and code the LED lights and add the web interface in a second step.

Overview

In this chapter, we will implement a traffic signal with a pedestrian walk button. This project works with LEDs, which allows us to see the state of our code as it executes. For the traffic light (also called a stoplight), we will use a red, yellow, and green LED to match the same colored lights on the traffic light. We will also use a red and yellow LED to correspond to the don't walk (red) and walk (yellow) lights.

We will use a pushbutton (also called a momentary button) because it triggers (is on) only when pushed. When released, it is no longer triggered (is off). Trigger is the word used to describe the state of the button where triggered means the connections from one side of the button to another are connected (on). A button that remains triggered (latched) is called a latching button, which typically must be pressed again to turn off.

We will simulate the traffic light and walk signal by first turning on only the green traffic light LED and the red walk LED signal. This is the normal state we will use. When the button is pressed, the traffic light will cycle to yellow for a few seconds then cycle to red. After a few seconds, the walk signal will cycle to yellow and after a few seconds will begin flashing. After a few more seconds, the walk signal will cycle back to red and the traffic light to green.

To make things even more interesting, we will also see how to modify this project to use a button simulated from a web page. Yes, we will see how to remotely control the hardware and our code over the network. If you are using the Pyboard or another MicroPython board that does not have any networking capabilities, you will need a network module. We will revisit networking for the Pyboard as we develop the project.

Now let's see what components are needed for this project and then we will see how to wire everything together.

Required Components

Table 9-1 lists the components you will need. You can purchase the components separately from Adafruit (adafruit.com), Sparkfun (sparkfun.com), or any electronics store that carries electronic components. Links to vendors are provided should you want to purchase the components. When listing multiple rows of the same object, you can choose one or the other - you do not need both. Also, you may find other vendors that sell the components. You should shop around to find the best deal. Costs shown are estimates and do not include any shipping costs.

Table 9-1. *Required Components*

Component	Qty	Description	Cost	Links
MicroPython board	1	Pyboard v1.1 with headers	\$45-50	https://www.adafruit.com/product/2390 https://www.adafruit.com/product/3499 https://www.sparkfun.com/products/14413 https://store.micropython.org/store
		WiPy	\$25	https://www.adafruit.com/product/3338 https://www.pycom.io/product/wipy/
LED	2	Red LEDs	kit	https://www.adafruit.com/product/2975
LED	2	Yellow LEDs	kit	https://www.adafruit.com/product/2975
LED	1	Green LEDs	kit	https://www.adafruit.com/product/2975
Resistor	5	220 or 330 ohm resistors	\$8-12	https://www.sparkfun.com/products/10969
Button	1	Momentary button, breadboard friendly	kit	https://www.sparkfun.com/products/12708
Breadboard	1	Prototyping board, half-sized	\$5	https://www.sparkfun.com/products/12002
Networking Module (Pyboard)	1	CC3000 breakout board (or equivalent)	\$15+	various
Jumper wires (WiPy)	9	M/M jumper wires, 6" (cost is for a set of 10 jumper wires)	\$4	https://www.sparkfun.com/products/8431
Jumper wires (Pyboard)	17	M/M jumper wires, 6" (cost is for a set of 10 jumper wires)	\$4	https://www.sparkfun.com/products/8431
Power	1	USB cable to get power from PC		Use from your spares
	1	USB 5V power source and cable		Use from your spares

Notice in the cost, “kit” for the LEDs and button. This refers to the fact that these components can be found in the Parts Pal kit from Adafruit that we saw in Chapter 2. Other vendors may have similar kits. Buying basic components like LEDs, buttons, and resistors are much cheaper when bought in a kit.

Similarly, you can pick up a set of resistors of various sizes much cheaper than if you bought a few at a time. In fact, you most likely will find buying a small set of 5 or 10 of each size resistor that you will eventually need will be far more expensive than if you purchased a set. The set from Sparkfun will provide you all the resistors you need for most projects.

Also, notice we need far fewer jumper wires for the WiPy. This is because we will be using a networking breakout board (in this example, a CC3000 module) to allow the Pyboard to connect to our network.

Finally, notice we need a Pyboard friendly networking module for the Pyboard. Once again, currently this must be a CC3000-based board or a WIZNET5K-based board. Refer to the previous chapters for samples of the networking breakout boards that work with the Pyboard.

Recall from Chapter 7 that LEDs require a current limiting resistor that reduces the current to safe levels for the LED. To determine what size resistor we need, we need to know several things about the LED. This data is available from the manufacturer who provides the data in the form of a datasheet or in the case of commercially packaged products, lists the data on the package. The data we need includes the maximum voltage, the supply voltage (how many volts are coming to the LED), and the current rating of the LED.

For example, if I have an LED like the ones in the Adafruit Parts Pal, in this case a 5mm red LED, we find on Adafruit’s website (adafruit.com/products/297) that the LED operates at 1.8-2.2V and 20mA of current. Let’s say we want to use this with a 5V supply voltage. We can then take these values and plug them into this formula:

$$R = (V_{CC} - V_f) / I$$

Using more descriptive names for the variable, we get the following.

$$\text{Resistor} = (\text{Volts_supply} - \text{Volts_forward}) / \text{Desired_current}$$

Plugging our data in, we get this result. Note that we have mA so we must use the correct decimal value (divide by 1000). In this case, it is 0.020 and we will pick a voltage in the middle.

$$\begin{aligned} \text{Resistor} &= (5 - 1.8) / 0.020 \\ &= 3.2 / 0.020 \\ &= 160 \end{aligned}$$

Thus, we need a resistor of 160 ohms. However, there is no resistor with that rating. When this happens, we use the next size up. For example, if you have only 220 or even 330 ohm resistors, you can use those. The result will be the LEDs will not be as bright but having a higher resistor is much safer than using one that is too small. Too much current and an LED will burn out.

Now, let’s see how to wire the components together.

Set Up the Hardware

While there are a lot of wires you will need to connect for this project using a WiPy and even more for the Pyboard, the components we will use are easy to plug into a breadboard. Table 9-2 shows the connections needed for this project.

Table 9-2. *Connections for the MicroPython (Pyboard and WiPy)*

MicroPython Board			
WiPy	Pyboard	Component	Wire Color
P3	X7	Stoplight: Red LED	
P4	X6	Stoplight: Yellow LED	
P5	X5	Stoplight: Green LED	
P6	X4	Walk signal: Red LED	
P7	X3	Walk signal: Yellow LED	
P23	X1	Button	
GND	GND	Breadboard	

Let's review some tips for wiring components. The best way to wire components to your board is to use a breadboard. As we saw in Chapter 7, a breadboard allows us to plug our components in and use jumper wires to make the connections. In this project, we will use one jumper wire for ground from the MicroPython board to the breadboard and then jumpers on the breadboard to connect to the button. In fact, we will use the ground rail on one side of the breadboard to plug in to one side of the LEDs.

The button works in either position so long as the pins are oriented as shown – with two legs on one side of the center trough. If you orient the button with the legs that can reach either side of the trough, it will be oriented correctly. If you get it off by 90 degrees, the button either will not work or will always be triggered. If you have any doubts, use a multimeter to test the continuity of the button connections. You should find the connections open when not pressed and closed when pressed.

The only component that is polarized is the LED (it has a positive and negative leg). When you look at the LED, you will see one leg (pin) of the LED is longer than the other. This longer side is the positive side. We will plug the LEDs in so that the negative leg is plugged into the ground rail and the positive side is plugged into the main area of the breadboard. We then plug the resistor in to jump over the center trough, connecting the resistor to the GPIO pin on the MicroPython board. It doesn't matter which direction you plug the resistor in – they will work both directions.

If this sounds confusing, don't worry as the wiring diagrams makes the connections more obvious. Let us see how to make the connections shown in the chart for the WiPy and Pyboard.

WiPy

Wiring for the WiPy is also best done orienting the expansion board with the USB connector to the right. Figure 9-1 shows the wiring diagram for the WiPy. Notice how the LEDs, resistors, and button are oriented. You should be able to use the drawing and the wiring plan to wire your own components. Also, notice the orientation of the WiPy. This should help you align the pins with the wires to the breadboard easier, but the physical orientation doesn't matter so long as you are using the correct GPIO pins.

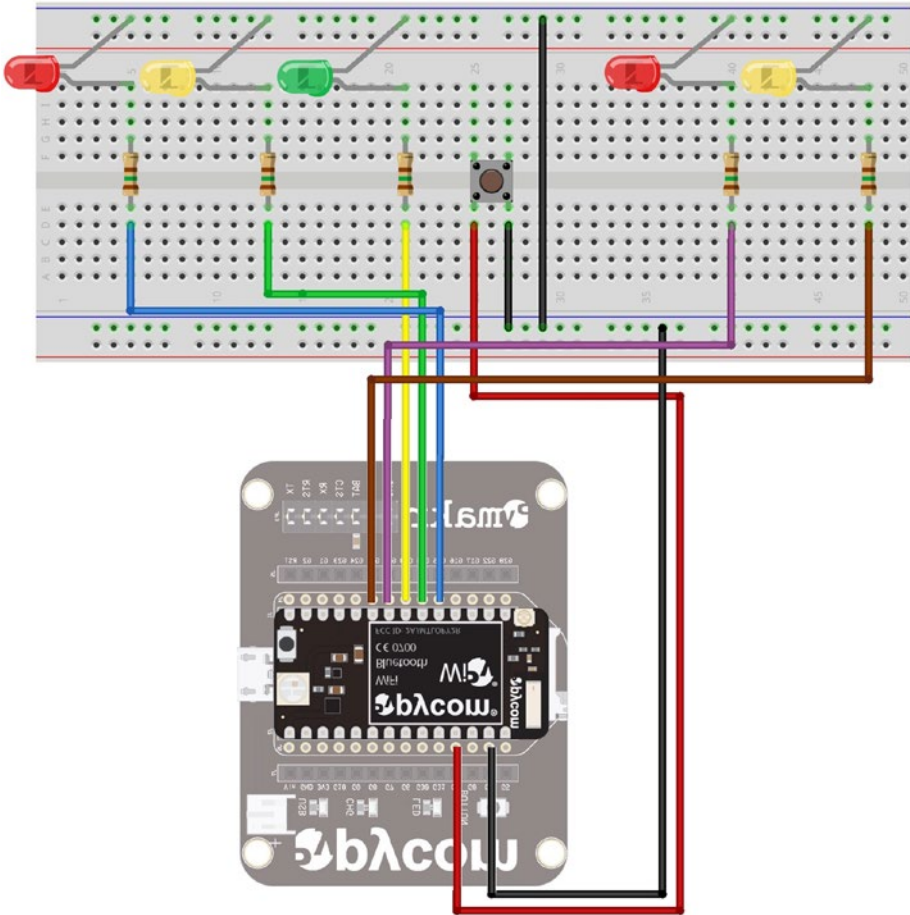


Figure 9-1. Wiring the Stoplight Simulation (WiPy)

Now, let's look at the wiring diagram for the Pyboard. It is made a bit more complex due to the need for adding a networking breakout board.

Pyboard

Wiring for the Pyboard is best done orienting the board with the USB connector to the left. This will allow you to read the pin numbers on the board even after the wires are plugged into the board. There are additional connections needed here for the network module. Recall that the CC3000 breakout board requires additional wiring to connect the SPI interface to the Pyboard. Like we did for the LEDs, resistors, and button, we should plan for how to connect the breakout board. Table 9-3 shows the connections needed for the Pyboard when used with the CC3000 breakout board. If you are using a different breakout board, be sure to annotate this plan with your connections.

Table 9-3. *Additional Connections for the Pyboard and CC3000 Breakout Board*

Pyboard	CC3000	Wire Color
Y3	IRQ	
Y4	VBEN	
Y5	CS	
Y6	CLK	
Y7	MISO	
Y8	MOSI	
V+	VIN	
GND	GND	

With these additional wires needed, the Pyboard wiring diagram is a bit more complicated. Figure 9-2 shows the wiring diagram for the Pyboard with the CC3000 breakout board.

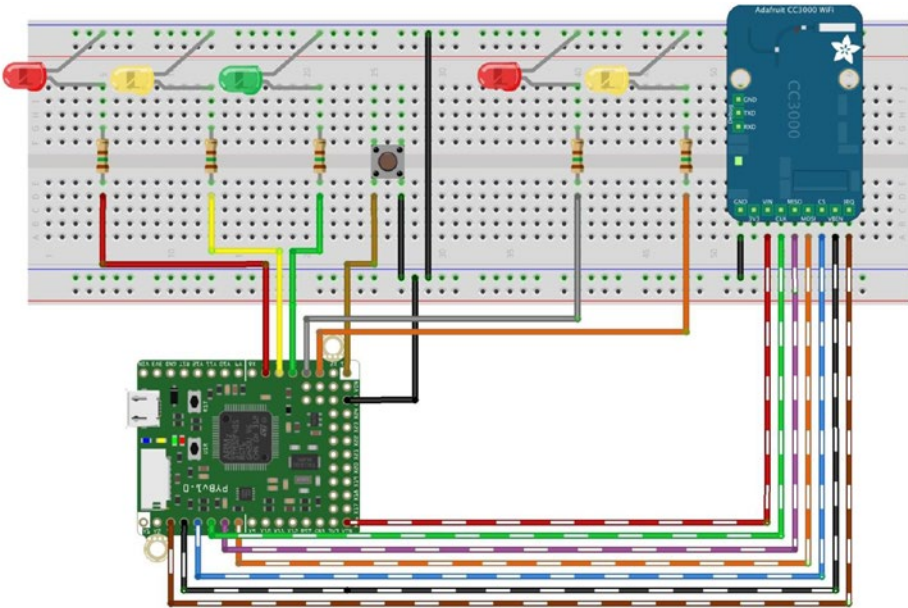


Figure 9-2. Wiring the Stoplight Simulation (WiPy)

Wow, that is a lot of wires! You can see the advantage of using a MicroPython board that has built-in WiFi – you don’t have to dedicate additional wires that can sometimes make a simple project more complex or when you look at your project all wired up – it looks like a nest of wiring.

■ **Note** If you are using the Pyboard, you may be able to use the onboard LEDs instead of separate LEDs.

Finally, always make sure you wire your project carefully, double-checking all the connections – especially power, ground, and any pins used for signaling (will be set to “high” or “on”) such as those pins used for SPI interfaces. Most importantly, never plug or unplug jumper wires when the project (or your board) is powered on. This will very likely damage your board or components.

■ **Caution** Never plug or unplug jumper wires when the project is powered on. You risk damaging your board or the components.

Once again, always make sure to double-check your connections before powering the board on. Now, let’s talk about the code we need to write. Don’t power on your board just yet – there is a fair amount of discussion needed before we’re ready to test the project.

Write the Code

Now it's time to write the code for our project. The code isn't overly complicated but it is a bit longer than the examples thus far. So, we are going to write the code in two parts. In the first part, we will see how to write code to simulate the pedestrian crosswalk button and traffic light. In the second part, we will forgo the use of a hardware button and instead use a web browser to control the button action remotely.

■ **Note** These sections demonstrate the code for the WiPy and describe the differences for the Pyboard. See the complete code listing at the end of the chapter for the code for each board.

As you will see, part 2 will reuse most of the code from the first part but will be a bit more complicated with the HTML server code. Let's begin with a look at part 1 of the project.

Part 1: Stoplight Simulator – Using a Pushbutton

The code for part 1 of the project will need to monitor the button and when pressed, cycle the lights as described above. We also need code to initialize the LEDs setting them to off initially. We can write functions for monitoring the button and cycling the LEDs. We will use an interrupt to tie the function for the button to the hardware so that we can avoid using a polling loop.

Imports

The imports for the project will require the Pin class from the machine library and the utime library. The following shows the imports for the WiPy.

```
from machine import Pin
import utime
```

The imports for the Pyboard also require the Pin class from the pyb library as well as the delay function and ExtInt class. The ExtInt class is used to set up an interrupt for the button to fire when the button is pressed.

```
from pyb import Pin, delay, ExtInt
```

Setup

The setup code for this project will need to initialize the button and LED instances and then turn off all the LEDs (as a precaution) and turn on the green stoplight LED and the red walk signal LED. Listing 9-1 shows the code for setup and initialization.

Listing 9-1. Setup and Initialization of the Button and LEDs (WiPy)

```

# Setup the button and LEDs
button = Pin('P23', Pin.IN, Pin.PULL_UP)
led1 = Pin('P3', Pin.OUT)
led2 = Pin('P4', Pin.OUT)
led3 = Pin('P5', Pin.OUT)
led4 = Pin('P6', Pin.OUT)
led5 = Pin('P7', Pin.OUT)

# Setup lists for the LEDs
stoplight = [led1, led2, led3]
walklight = [led4, led5]

# Turn off the LEDs
for led in stoplight:
    led.value(0)
for led in walklight:
    led.value(0)

# Start with green stoplight and red walklight
stoplight[2].value(1)
walklight[0].value(1)

```

One thing to notice is how the button is initialized. This is a `Pin` object instance that is set up as an input (read) and the pull-up resistors are turned on. This allows the board to detect when the button is pressed because the value of the pin will be a positive value when the connection is made (the button is pressed).

Notice also I create a list that contains the LEDs for the stoplight and walk signal (named `walklight` in the code). This is mostly for demonstration so you can see how to manage lists of class objects. As you can see, it makes it easier to call the same function for all the objects in the list using a loop. Take note of this technique as you will need it from time to time in other projects.

The code for the Pyboard is largely the same. The differences include different pin numbers for the LEDs and buttons (see wiring diagram), different options for the initialization of the pins (`Pin.OUT_PP` instead of `Pin.OUT`), and the use of different functions for the `Pin` class: `high()` for `value(1)` and `low()` for `value(0)`.

Functions

There are two functions needed for this part of the project. First, we need a function to cycle through the lights. Second, we need a function to monitor the button press. Let's look at the cycle light function.

We will name the cycle lights function `cycle_lights()`. Recall we need to control how the lights change state. We do this with a specific cycle as described earlier. To recap, we call this function when we want to simulate changing the stoplight when the walk request button is pressed. Thus, this function will be called from the code for the button.

Listing 9-2 shows the code for the `cycle_lights()` button. As you will see, the code is rather straightforward. The only tricky part may be the loop used to flash the yellow walk LED. Be sure to read through it so that you understand how it works.

Listing 9-2. The `cycle_lights()` Function (WiPy)

```
# We need a method to cycle the stoplight and walklight
#
# We toggle from green to yellow for 2 seconds
# then red for 20 seconds.
def cycle_lights():
    # Go yellow.
    stoplight[2].value(0)
    stoplight[1].value(1)
    # Wait 2 seconds
    utime.sleep(2)
    # Go red and turn on walk light
    stoplight[1].value(0)
    stoplight[0].value(1)
    utime.sleep_ms(500) # Give the pedestrian a chance to see it
    walklight[0].value(0)
    walklight[1].value(1)
    # After 10 seconds, start blinking the walk light
    utime.sleep(1)
    for i in range(0,10):
        walklight[1].value(0)
        utime.sleep_ms(500)
        walklight[1].value(1)
        utime.sleep_ms(500)

    # Stop=green, walk=red
    walklight[1].value(0)
    walklight[0].value(1)
    utime.sleep_ms(500) # Give the pedestrian a chance to see it
    stoplight[0].value(0)
    stoplight[2].value(1)
```

The code for the Pyboard is very similar with the changes as noted previously for controlling the LEDs and use of the `delay()` function instead of the `utime` class `sleep` functions.

We will name the button function `button_pressed()`. This function is used as a callback for the button press interrupt. Technically, we tell MicroPython to associate this method with the pin interrupt but we will see that in a moment. However, there is another element to this function that requires explanation.

When we use a component like a button and the user (you) press the button, the contacts in the button do not go from an off state to an on state instantaneously. There is a very small period where the value read is erratic. Thus, we cannot simply say, “when the pin goes high” because the value read on the pin may “bounce” from low to high (or high

to low) rapidly. This is called bouncing. We can overcome this artificially with code (as well as other techniques) – called debouncing.

In this case, we can check the value of the pin (button) over time and only “trigger” the button press if and only if the value remains stable during that time. Code for debouncing the pin is shown in Listing 9-3. Notice in the loop we wait for a value of 50. This is 50 milliseconds. If the trigger is long enough, we call the `cycle_lights()` function.

Listing 9-3. The `button_pressed()` Function (WiPy)

```
def button_pressed(line):
    cur_value = button.value()
    active = 0
    while (active < 50):
        if button.value() != cur_value:
            active += 1
        else:
            active = 0
            utime.sleep_ms(1)
            print("")
    if active:
        cycle_lights()
    else:
        print("False press")
```

■ **Tip** For more information about debouncing and the techniques available to avoid it, see <http://www.eng.utah.edu/~cs5780/debouncing.pdf>.

Finally, we need to set up the button to call the `button_pressed()` function when the board detects the interrupt. The following sets the callback function on the WiPy.

```
# Create an interrupt for the button
button.callback(Pin.IRQ_FALLING, button_pressed)
```

The code on the Pyboard is also a single line of code but in this case, we must use the `ExtInt` class to set up the interrupt handler as shown below.

```
# Create an interrupt for the button
e = ExtInt('X1', ExtInt.IRQ_FALLING, Pin.PULL_UP, button_pressed)
```

■ **Tip** See the online MicroPython documentation for more information about using the Pin callback for the WiPy and the interrupt for the Pyboard.

Now we're all set to test the code. Go ahead and open a new file named `ped_part1_wipy.py` (or `ped_part1_pyb.py` for the Pyboard) and enter the code above. Listing 9-4 shows the complete code for part 1 of the project. If you're using the Pyboard, it's OK to cheat by looking at the complete listing at the end of the chapter.

Listing 9-4. Stoplight Simulation – Part 1 (WiPy)

```
# MicroPython for the IOT - Chapter 9
#
# Project 2: A MicroPython Pedestrian Crosswalk Simulator
#         Part 1 - controlling LEDs and button as input
#
# Required Components:
# - WiPy
# - (2) Red LEDs
# - (2) Yellow LEDs
# - (1) Green LED
# - (5) 220 Ohm resistors
# - (1) breadboard friendly momentary button
#
# Note: this only runs on the WiPy.
#
# Imports for the project
from machine import Pin
import utime

# Setup the button and LEDs
button = Pin('P23', Pin.IN, Pin.PULL_UP)
led1 = Pin('P3', Pin.OUT)
led2 = Pin('P4', Pin.OUT)
led3 = Pin('P5', Pin.OUT)
led4 = Pin('P6', Pin.OUT)
led5 = Pin('P7', Pin.OUT)

# Setup lists for the LEDs
stoplight = [led1, led2, led3]
walklight = [led4, led5]

# Turn off the LEDs
for led in stoplight:
    led.value(0)
for led in walklight:
    led.value(0)

# Start with green stoplight and red walklight
stoplight[2].value(1)
walklight[0].value(1)
```

```

# We need a method to cycle the stoplight and walklight
#
# We toggle from green to yellow for 2 seconds
# then red for 20 seconds.
def cycle_lights():
    # Go yellow.
    stoplight[2].value(0)
    stoplight[1].value(1)
    # Wait 2 seconds
    utime.sleep(2)
    # Go red and turn on walk light
    stoplight[1].value(0)
    stoplight[0].value(1)
    utime.sleep_ms(500) # Give the pedestrian a chance to see it
    walklight[0].value(0)
    walklight[1].value(1)
    # After 10 seconds, start blinking the walk light
    utime.sleep(1)
    for i in range(0,10):
        walklight[1].value(0)
        utime.sleep_ms(500)
        walklight[1].value(1)
        utime.sleep_ms(500)

    # Stop=green, walk=red
    walklight[1].value(0)
    walklight[0].value(1)
    utime.sleep_ms(500) # Give the pedestrian a chance to see it
    stoplight[0].value(0)
    stoplight[2].value(1)

# Create callback for the button
def button_pressed(line):
    cur_value = button.value()
    active = 0
    while (active < 50):
        if button.value() != cur_value:
            active += 1
        else:
            active = 0
            utime.sleep_ms(1)
            print("")
    if active:
        cycle_lights()

```

```

else:
    print("False press")

# Create an interrupt for the button
button.callback(Pin.IRQ_FALLING, button_pressed)

```

Test and Debug the Code

Testing this part of the project requires copying the source code file to the board and then executing it. Since the code was written without a `run()` function, simply importing it is all you need to do to run the code. Be sure to check all connections before powering on the board. Recall, the command we use to import and run the code once we connect to the board is shown below.

```
>>> import ped_part1_wipy
```

Once imported, the code will run and you can press the button to see the lights cycle through the stages. If you do not see any of the lights on (the green stoplight and red walk signal should be on), check the setup and initialization code. If the lights do not cycle when the button is pressed, check the code for the button to ensure you have it correct. If they do not illuminate in order, you may have the pins wired incorrectly. Be sure to check all wiring connections when encountering problems, always powering off the board before disconnecting or reconnecting any wires or components.

When the project is working, try it out a few times to ensure it works as expected. Then you should congratulate yourself! You've just wired together several discrete electronic components and made a working simulation of a pedestrian walk button and stoplight. Cool!

Now, let's kick this project up a notch and make it accessible over the Internet. After all, that is what IOT is all about!

Part 2: Stoplight Simulator – Remote Control with HTML

The code for this part will use all the code from the first part but we will not need the code for the button. Instead, we will use the `Socket` class to create a listener to listen for a connection from a web browser. The code will send a short HTML-based response (a simple web page) to the client that includes a form containing two buttons – one for the walk request and another for shutting down the server code. The listener will listen on port 80.

■ **Note** If you're using the WiPy, you don't need to add any networking code, but I'll show you how to do so in case you want to run the project on your own network or connect it to the Internet.

The concept of the HTML server is quite simple. The code listens for a connection on the socket port then receives the request (in this case in the form of HTML GET method) and sends an HTML response (the web page). The code then checks to see if the request contains the form data from one of the two buttons: a button to request the walk cycle and another to shut down the server. As you will see, shutting down the server forces us to write the code a bit tidier.

If you've never used HTML code before, don't worry as the example code will provide everything you need. You don't have to learn HTML to use the project in this chapter (or the next), but a basic knowledge would be helpful if you want to elaborate on the project or use the HTML server concept for your own projects. A good source of information about HTML can be found at <https://www.w3schools.com/html/>.

Let's look at the changes needed for the imports section.

Imports

We need a few more libraries in the import section. We need the socket library (renamed with an alias for clarity), the machine library, and the WLAN class from the network library. It is perfectly normal to write the imports this way, but with a little imagination you can simplify them. Do you see how? The following shows the complete import section for the WiPy.

```
from machine import Pin
import usocket as socket
import utime
import machine
from network import WLAN
```

The imports for the Pyboard are a bit shorter. We need to add the SPI library for the network board and the network library as well as the socket library. The following shows the complete import section for the Pyboard.

```
from pyb import Pin, delay, ExtInt, SPI
import network
import usocket as socket
```

Now, let's look at the changes in the setup section.

Setup

For this part of the project, we need to add the code to connect our board to our network (or the Internet) via a wireless connection. We also need to place the HTML code here in the form of a string. This is typically where (and how) one defines a large string for use in Python. Using triple quotes on either side makes the string a documentation string (also called a docstring). See <https://www.python.org/dev/peps/pep-0257/> for more details.

The networking code for the WiPy is the same code we saw earlier in the book. In this case, it sets up the wireless networking feature to connect to an existing wireless network rather than the default access point behavior of the WiPy. Listing 9-5 shows the network code for the WiPy.

Listing 9-5. Wireless Network Setup (WiPy)

```
# Setup the board to connect to our network.
wlan = WLAN(mode=WLAN.STA)
nets = wlan.scan()
for net in nets:
    if net.ssid == 'YOUR_SSID':
        print('Network found!')
        wlan.connect(net.ssid, auth=(net.sec, 'YOUR_PASSWORD'), timeout=5000)
        while not wlan.isconnected():
            machine.idle() # save power while waiting
        print('WLAN connection succeeded!')
        print("My IP address is: {0}".format(wlan.ifconfig()[0]))
        break
```

The networking code for the Pyboard is the same code we saw earlier in the book. In this case, it first sets up the SPI interface to a networking board (in this example a CC3000 breakout board), then initiates a wireless connection to an existing wireless network rather than the default access point behavior of the WiPy. Listing 9-6 shows the network code for the Pyboard.

Listing 9-6. Wireless Network Setup (Pyboard)

```
# Setup network connection
nic = network.CC3K(SPI(2), Pin.board.Y5, Pin.board.Y4, Pin.board.Y3)
# Replace the following with your SSID and password
nic.connect("YOUR_SSID", "YOUR_PASSWORD")
print("Connecting...")
while not nic.isconnected():
    delay(50)
print("Connected!")
print("My IP address is: {0}".format(nic.ifconfig()[0]))
```

■ **Tip** You must change the SSID and password in the code to match your network.

Another part of the setup is the HTML response code. This may seem like a difficult part of the code, but it is simple. We are constructing a string that we will send back to the client through the socket. This string contains HTML code starting with a header. Then, we supply a title (which appears in the title bar of your browser), some text to display on the page, and a form that contains the two buttons. These buttons are in a form that, when each is pressed, will send a HTML GET request through the socket to the server. Simple! Listing 9-7 shows the HTML string for the project. Again, don't worry about the details. You can work on improving the web page later.

Listing 9-7. HTML Response String

```
# HTML web page for the project
HTML_RESPONSE = """<!DOCTYPE html>
<html>
  <head>
    <title>MicroPython for the IOT - Project 2</title>
  </head>
  <center><h2>MicroPython for the IOT - Project 2</h2></center><br>
  <center>A simple project to demonstrate how to control hardware over the
  Internet.</center><br><br>
  <form>
    <center>
      <button name="WALK" value="PLEASE" type="submit" style="height:
        50px; width: 100px">REQUEST WALK</button>
      <br><br>
      <button name="shutdown" value="now" type="submit" style="height:
        50px; width: 100px">Stop Server</button>
    </center>
  </form>
</html>
"""
```

Notice the buttons in the code (`<button></button>` tags). The name and value will be sent in the request to the server. The type of the button is defined as `submit` and when placed on a form, causes the client to post the form data to the server when the button is pressed. Finally, the string before the end tag is the label that will be displayed on the button.

If you're concerned about this being a huge string that takes up memory, you're right it is and it does. If you are planning a project that uses a large HTML response or perhaps several responses, you may want to consider storing those in a file (one response per file) and reading the data from a file before sending it to the client. This will save some data space and could make the difference for larger projects that use more memory.

Don't be concerned about the number of lines used here. The whitespace is used for decoration (mostly) so if you wanted to reduce its overall visual size, you can remove the whitespace but it is common practice to indent the HTML code like this for easier reading.

■ **Tip** If you are concerned about security (who isn't?), you can use secure socket layer connections to make your connection to your MicroPython board more secure. MicroPython provides a class named `ssl` that you can use. See the `ssl` class documentation for more information or you can look at the example at https://github.com/micropython/micropython/blob/master/examples/network/http_server_ssl.py.

There is one other important thing about using HTML on your MicroPython board that should be made clear: this example and many others you may find on the Internet should not be confused with a robust web server. More specifically, this example merely sends HTML back to the client once it detects a HTML GET method (request) from the client while listening on a network socket. So, this example is merely a simple HTML server and not a web server in the sense that it can do all manner of things a web server can do - it doesn't. Thus, you should take care to code your projects to restrict operation to specific GET (or POST) requests and return the appropriate HTML response.

USING WEB PAGES WITH IMAGES

Since there isn't a web server built into MicroPython (yet), serving web pages with images or any other media is problematic. However, you can use a little trick and your PC to use images in your projects. We can use the `SimpleHTTPServer` in Python on our PC to provide a rudimentary web server. In our HTML from our MicroPython board, we can use the `img` tag for the images that uses a URL to point to the file on our PC. For example, the following HTML will reference an image in the same folder as the HTTP server on our PC.

```
<html>
  <head>
    <title>MicroPython for the IOT - Project 2</title>
    <meta http-equiv="refresh" content="10">
  </head>
  <body>
    <p>Pedestrian Stoplight Simulation</p>
    <form>
      
    </form>
  </body>
</html>
```

Notice the `img` tag uses a URL to the localhost on port 8000. We establish that when we run the `SimpleHTTPServer` in Python on our PC as shown below. Run this command in the same folder as the images you want to use.

```
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
127.0.0.1 - - [07/Aug/2017 14:42:30] "GET /red.png HTTP/1.1" 200 -
```

This creates a hybrid solution that will allow you to use images in your MicroPython web-based projects albeit with a little help from our PC (or any other PC on you network).

Now let's look at how to make the `run()` function.

Functions

All that remains to do in the code is to change the code for the hardware button and make a `run()` function to contain the main portion of the code. In this project, that includes the code for setting up the socket and the listen and respond code. You can leave the `button_pressed()` function, but it is not needed. Listing 9-8 shows the code for the `run()` function.

Listing 9-8. The `run()` Function (WiPy)

```
# Setup the socket and respond to HTML requests
def run():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind('', 80)
    sock.listen(5)
    server_on = True
    while server_on:
        client, address = sock.accept()
        print("Got a connection from a client at: %s" % str(address))
        request = client.recv(1024)
        # Process the request from the client. Payload should appear in pos 6.
        # Check for walk button click
        walk_clicked = (request[:17] == b'GET /?WALK=PLEASE')
        # Check for stop server button click
        stop_clicked = (request[:18] == b'GET /?shutdown=now')
        if walk_clicked:
            print('Requesting walk now!')
            cycle_lights()
        elif stop_clicked:
            server_on = False
            print("Goodbye!")
        client.send(HTML_RESPONSE)
        client.close()
```

Notice we begin the function with setting up the socket and binding it to port 80. The `bind()` function uses an empty string to instruct the socket to use the current IP address. Next, we tell the socket to listen on the socket with a five-second timeout using the `listen(5)` function. Then, we set a Boolean variable to loop over the listen and respond code while it is true. In this case, we use the Boolean and set it to False when the shutdown button is pressed.

Inside the loop is where things get interesting. The first thing we do is accept a connection from the socket using `sock.accept()`, which returns two items: the client object instance and the address of the client. We can print the address of the client for debugging purpose.

Next, we tell the client to receive up to 1024 bytes with the `client.recv(1024)` call. This will wait until there is a response from the client, so be aware this is not a preemptive loop – it will wait until the client responds. When the client does send data (like first connect or when a button is pressed), we can then search the string sent for the command.

Notice in the code we check for either the walk request searching for `b'GET /?WALK=PLEASE'` or the shutdown event searching for `b'GET /?shutdown=now'`. There is a bit of trickery going on here that warrants explanation. We use the request string as an array starting our search from the start for the number of characters in the string (17 or 18), which is represented with `[:17]` or `[:18]`.

If the walk request is detected, we call the `cycle_lights()` function, which is unmodified from part 1. If the shutdown event is detected, we set the loop to terminate and then exit the `run()` function.

Take some time to read through the code until you understand how it works. When ready, go ahead and open a new file named `ped_part2_wipy.py` (or `ped_part2_pyb.py` for the Pyboard) and enter the code above.

Finally, you must remove the line of code that sets up the button callback (or on the Pyboard the `ExtInt()` call). It won't hurt the code to leave it in, but since we're not using the button, it is not needed and all unnecessary code should be removed.

Now, let's look at the complete code for both the WiPy and Pyboard.

Completed Code

In this section, we will see the final, completed code for the WiPy and Pyboard. These listings are provided as a reference for you to ensure you have the correct code for your board. Listing 9-9 shows the complete code for running the project on the WiPy.

Listing 9-9. Complete Code for the Stopligh Simulation (WiPy)

```
# MicroPython for the IOT - Chapter 9
#
# Project 2: A MicroPython Pedestrian Crosswalk Simulator
#         Part 3 - controlling LEDs over the Internet
#
# Required Components:
# - WiPy
# - (2) Red LEDs
# - (2) Yellow LEDs
# - (1) Green LED
# - (5) 220 Ohm resistors
#
# Note: this only runs on the WiPy.
#
# Imports for the project
from machine import Pin
import usocket as socket
import utime
import machine
from network import WLAN

# Setup the board to connect to our network.
wlan = WLAN(mode=WLAN.STA)
nets = wlan.scan()
```

```

for net in nets:
    if net.ssid == 'YOUR_SSID':
        print('Network found!')
        wlan.connect(net.ssid, auth=(net.sec, 'YOUR_PASSWORD'), timeout=5000)
        while not wlan.isconnected():
            machine.idle() # save power while waiting
            print('WLAN connection succeeded!')
            print("My IP address is: {}".format(wlan.ifconfig()[0]))
            break

```

```
# HTML web page for the project
```

```
HTML_RESPONSE = """<!DOCTYPE html>
```

```

<html>
  <head>
    <title>MicroPython for the IOT - Project 2</title>
  </head>
  <center><h2>MicroPython for the IOT - Project 2</h2></center><br>
  <center>A simple project to demonstrate how to control hardware over the
  Internet.</center><br><br>
  <form>
    <center>
      <button name="WALK" value="PLEASE" type="submit" style="height:
      50px; width: 100px">REQUEST WALK</button>
      <br><br>
      <button name="shutdown" value="now" type="submit" style="height:
      50px; width: 100px">Stop Server</button>
    </center>
  </form>
</html>
"""

```

```
# Setup the LEDs (button no longer needed)
```

```
led1 = Pin('P3', Pin.OUT)
```

```
led2 = Pin('P4', Pin.OUT)
```

```
led3 = Pin('P5', Pin.OUT)
```

```
led4 = Pin('P6', Pin.OUT)
```

```
led5 = Pin('P7', Pin.OUT)
```

```
# Setup lists for the LEDs
```

```
stoplight = [led1, led2, led3]
```

```
walklight = [led4, led5]
```

```
# Turn off the LEDs
```

```
for led in stoplight:
```

```
    led.value(0)
```

```
for led in walklight:
```

```
    led.value(0)
```

```

# Start with green stoplight and red walklight
stoplight[2].value(1)
walklight[0].value(1)

# We need a method to cycle the stoplight and walklight
#
# We toggle from green to yellow for 2 seconds
# then red for 20 seconds.
def cycle_lights():
    # Go yellow.
    stoplight[2].value(0)
    stoplight[1].value(1)
    # Wait 2 seconds
    utime.sleep(2)
    # Go red and turn on walk light
    stoplight[1].value(0)
    stoplight[0].value(1)
    utime.sleep_ms(500) # Give the pedestrian a chance to see it
    walklight[0].value(0)
    walklight[1].value(1)
    # After 10 seconds, start blinking the walk light
    utime.sleep(1)
    for i in range(0,10):
        walklight[1].value(0)
        utime.sleep_ms(500)
        walklight[1].value(1)
        utime.sleep_ms(500)

    # Stop=green, walk=red
    walklight[1].value(0)
    walklight[0].value(1)
    utime.sleep_ms(500) # Give the pedestrian a chance to see it
    stoplight[0].value(0)
    stoplight[2].value(1)

# Setup the socket and respond to HTML requests
def run():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind('', 80)
    sock.listen(5)
    server_on = True
    while server_on:
        client, address = sock.accept()
        print("Got a connection from a client at: %s" % str(address))
        request = client.recv(1024)
        # Process the request from the client. Payload should appear in pos 6.
        # Check for walk button click
        walk_clicked = (request[:17] == b'GET /?WALK=PLEASE')

```

```

# Check for stop server button click
stop_clicked = (request[:18] == b'GET /?shutdown=now')
if walk_clicked:
    print('Requesting walk now!')
    cycle_lights()
elif stop_clicked:
    server_on = False
    print("Goodbye!")
client.send(HTML_RESPONSE)
client.close()
sock.close()

```

Now, let's see the code for the Pyboard. As you will see, it is largely the same except for the networking code, imports, and pin function calls. Listing 9-10 shows the complete code for running the project on the Pyboard.

Listing 9-10. Complete Code for the Stoplight Simulation (Pyboard)

```

# MicroPython for the IOT - Chapter 9
#
# Project 2: A MicroPython Pedestrian Crosswalk Simulator
#         Part 2 - Controlling the walklight remotely
#
# Required Components:
# - Pyboard
# - (2) Red LEDs
# - (2) Yellow LEDs
# - (1) Green LED
# - (5) 220 Ohm resistors
# - (1) breadboard friendly momentary button
# - (1) CC3000 breakout board
#
# Note: this only runs on the Pyboard.
#
# Imports for the project
from pyb import Pin, delay, ExtInt, SPI
import network
import usocket as socket

# Setup network connection
nic = network.CC3K(SPI(2), Pin.board.Y5, Pin.board.Y4, Pin.board.Y3)
# Replace the following with your SSID and password
nic.connect("YOUR_SSID", "YOUR_PASSWORD")
print("Connecting...")
while not nic.isconnected():
    delay(50)
print("Connected!")
print("My IP address is: {}".format(nic.ifconfig()[0]))

```

```

# HTML web page for the project
HTML_RESPONSE = """<!DOCTYPE html>
<html>
  <head>
    <title>MicroPython for the IOT - Project 2</title>
  </head>
  <center><h2>MicroPython for the IOT - Project 2</h2></center><br>
  <center>A simple project to demonstrate how to control hardware over the
  Internet.</center><br><br>
  <form>
    <center>
      <button name="WALK" value="PLEASE" type="submit" style="height:
        50px; width: 100px">REQUEST WALK</button>
      <br><br>
      <button name="shutdown" value="now" type="submit" style="height:
        50px; width: 100px">Stop Server</button>
    </center>
  </form>
</html>
"""

# Setup the LEDs
led1 = Pin('X7', Pin.OUT_PP)
led2 = Pin('X6', Pin.OUT_PP)
led3 = Pin('X5', Pin.OUT_PP)
led4 = Pin('X4', Pin.OUT_PP)
led5 = Pin('X3', Pin.OUT_PP)

# Setup lists for the LEDs
stoplight = [led1, led2, led3]
walklight = [led4, led5]

# Turn off the LEDs
for led in stoplight:
    led.low()
for led in walklight:
    led.low()

# Start with green stoplight and red walklight
stoplight[2].high()
walklight[0].high()

# We need a method to cycle the stoplight and walklight
#
# We toggle from green to yellow for 2 seconds
# then red for 20 seconds.
def cycle_lights():

```



```

# Go yellow.
stoplight[2].low()
stoplight[1].high()
# Wait 2 seconds
delay(2000)
# Go red and turn on walk light
stoplight[1].low()
stoplight[0].high()
delay(500) # Give the pedestrian a chance to see it
walklight[0].low()
walklight[1].high()
# After 10 seconds, start blinking the walk light
delay(10000)
for i in range(0,10):
    walklight[1].low()
    delay(500)
    walklight[1].high()
    delay(500)

# Stop=green, walk=red
walklight[1].low()
walklight[0].high()
delay(500) # Give the pedestrian a chance to see it
stoplight[0].low()
stoplight[2].high()

```

```
# Setup the socket and respond to HTML requests
```

```

def run():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(('', 80))
    sock.listen(5)
    server_on = True
    while server_on:
        client, address = sock.accept()
        print("Got a connection from a client at: %s" % str(address))
        request = client.recv(1024)
        # Process the request from the client. Payload should appear in pos 6.
        # Check for walk button click
        walk_clicked = (request[:17] == b'GET /?WALK=PLEASE')
        # Check for stop server button click
        stop_clicked = (request[:18] == b'GET /?shutdown=now')
        if walk_clicked:
            print('Requesting walk now!')
            cycle_lights()
        elif stop_clicked:
            server_on = False

```

```

        print("Goodbye!")
        client.send(HTML_RESPONSE)
        client.close()
sock.close()

```

Ok, now we're ready to execute the project.

Execute!

Now is the fun part! We've got the code all set up to control our LEDs and we know it works from part 1. We also have code to set up a socket listener to accept connections over port 80 on our MicroPython board. All we need now is the IP address of that board to point our web browser. We can get that from our debug statements by running the code. Listing 9-11 shows the initial run for the project on a WiPy (results for the Pyboard are similar).

Listing 9-11. Running the Stoplight Simulation (WiPy)

```

MicroPython v1.8.6-694-g25826866 on 2017-06-29; WiPy with ESP32
Type "help()" for more information.
>>> import ped_part3_wipy as w
Network found!
WLAN connection succeeded!
My IP address is: 192.168.42.128
>>> w.run()
Got a connection from a client at: ('192.168.42.127', 49236)
Got a connection from a client at: ('192.168.42.127', 49237)
Got a connection from a client at: ('192.168.42.127', 49243)
Requesting walk now!
Got a connection from a client at: ('192.168.42.127', 49254)
Goodbye!
>>>

```

Notice in this case the IP address is 192.168.42.127. All we need to do is put that in our browser and shown in Figure 9-3.

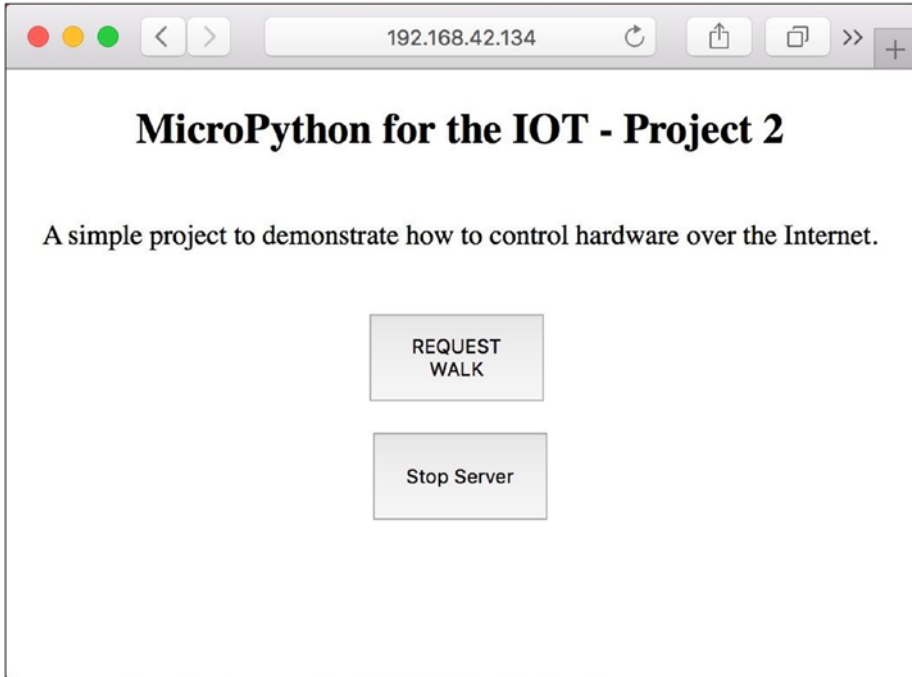


Figure 9-3. *Stoplight Simulation Project*

■ **Caution** If you're using the WiPy and telnet to open a REPL console, be advised that as soon as you run the networking portion of the code, your REPL console will disconnect. This is because the IP address will change! So, when working with the WiPy connecting it to your network, it is best to use the screen or another terminal program over USB to get a REPL console.

Once you enter the URL, you should see a web page like the image shown. If you don't, be sure to check the HTML in your code to ensure it is exactly like what is shown; otherwise, the page may not display properly. You should also ensure the network your PC is connected to can reach the network to which your board is connected. If your home office is set up like mine, there may be several WiFi networks you can use. It is best if your board and your PC are on the same network (and same subnet).

Once you get that sorted out, go ahead and play with the buttons. Remember, the walk button will engage and you will see the lights cycle but you won't be able to do anything until the walk cycle is complete. This is because we don't return the response HTML until after the cycle is complete (see the code to convince yourself). Also, when you click the shutdown button, you will need to restart the code to rerun it. Simply call the `run()` function again.

Take another look at the listing above. Notice there are debug messages printed for each time the client connects (the code accepts the connection and GET request) as well as a statement about what it is doing. You should see something similar in your REPL console.

It is at this point that you should be basking in the wonder of your first successful MicroPython IOT remote control hardware project. Take some time to enjoy a job well done.

The final step in the process is optional for this project since you are unlikely to want it to run more than to just test it. However, if you do want to make it run every time you boot your board, you can modify the `main.py` code module on the board to import your code and call the `run()` function.

WHEN THINGS GO WONKY¹

Sometimes when working with larger scripts or more complicated logic, many libraries (drivers), or even when you use a lot of strings (memory), you can sometimes encounter strange, terse errors. The following is an example.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ped_part2_wipy.py", line 97, in run
OSError: [Errno 12] ENOMEM
```

In this case, the error is an out of memory error meaning your MicroPython board has allocated all (or nearly all) memory to your code and it cannot continue. The only thing you can do at this point is reboot the board. You can try a soft boot (CTRL-D), but that usually doesn't fix the problem, but a hard reboot will.

At this point, you've completed your first real MicroPython IOT project. We can call it an IOT project because it uses the Internet, but it is only a simulation and doesn't actually control a real traffic light, but the techniques we saw in this project permit us to connect our MicroPython board to the Internet and interact with it - which is what the IOT is all about!

Taking it Further

This project shows excellent prospects for reusing the techniques in other projects. This is especially true for the HTML server aspect. If you liked controlling your MicroPython board over the Internet, you should consider taking time to explore some embellishments. Here are a few you may want to consider. Some are easy and some may be a challenge or require more research.²

¹A highly technical term that describes a state of extreme chaos and general failure to execute. Not to be confused with hinky, which indicates a slightly less severe case when something works but isn't quite right.

²I include these so you can grow your knowledge beyond the confines of this book. Try these when you've finished the other projects in the book.

- Use NeoPixels (<https://www.adafruit.com/category/168>) instead of LEDs. These are RGB LEDs so you need only two - one for the stoplight and one for the walk light. See <https://github.com/JanBednarik/micropython-ws2812> for more information and examples.
- Modify the HTML response to show the state of the lights.
- Explore the HTML code to change the web page to your liking. Consider using cascading style sheets to change the background of the button when pressed.
- Connect your board to the Internet and call a friend to connect to your board and try it out.
- Use a display in place of the LEDs for the walk sign to show “WALK” or “DON’T WALK.”

Of course, if you want to press on to the next project, you’re welcome to do so but take some time to explore these potential embellishments – it will be good practice.

If you’re up for a real challenge, you can reuse the code for this project and replace the button logic with a relay board that allows you to use a low-voltage device like the MicroPython board to turn on or off a higher voltage circuit. In this case, you can use the HTML button to turn the relay on or off over the Internet.

Summary

Working with discrete electronic components can be a lot of fun. Just making the circuit work is a real thrill when you’re just starting out with electronics. Now that we know a lot more about MicroPython, we can see how powerful it is to have an easy-to-program language like Python at our disposal to work with directly with hardware – even over the Internet.

In this chapter, we implemented a simulation of a pedestrian crosswalk button and stoplight. We used a series of LEDs to represent the stoplight and walk signal. We also added a hardware button to simulate pressing the real walk button and then turned that into a remote-control button we hosted via simple HTML right from our MicroPython board. If you liked this project, you will enjoy the next two projects even more.

In the next chapter, we will explore a project that uses a sensor to read values and then archive the data and display it on a web page when requested. This is the next-to-last step in our journey of building a true MicroPython IOT project. This chapter and the next one show you how to make data available over the Internet and the final project chapter will show you how to make your sensor data available through cloud services.



Project 3: Plant Monitoring

One of the most common forms of IOT projects are those that monitor events using sensors providing the data either to another machine, cloud service, or local server (like an HTML server). One way to do that is to wire your MicroPython board up to a set of sensors and then log the data. You can find several examples of general data loggers on the Internet, but few combine the logging of data with a visualization component. Indeed, making sense of the data is the key to making a successful IOT solution.

In this project, we will explore combining data logging with data visualization. We will use the same HTML server technique from the last chapter as well as several techniques from previous chapters. We will also see how to use an analog sensor - a sensor that produces analog data that we will then have to interpret. In fact, we will rely on the analog-to-digital conversion (ADC) capabilities of our boards to change the voltage read to a value we can use. We will also see a bit more sophistication in the code as we leverage what we learned about classes and modules from Chapter 4.

The added complexity for this project isn't new hardware or interfaces although we will see how to use the analog-to-digital converter class; the complexity is in the sophistication of the code. As you will see, the code used in this chapter is more modular and uses more functions than previous projects. For that reason alone, it is more complex. But, as you will see, the code isn't difficult to learn and uses concepts we have seen in previous chapters.

Overview

In this chapter, we will implement a plant soil moisture monitoring solution (plant monitor for brevity). This will involve using one or more soil moisture sensors connected to our MicroPython board. We will set up a timer alarm (an interrupt) to run periodically to read the data from the sensors and store it in a comma-separated value (CSV) file.

Figure 10-1 depicts a conceptual drawing of the project. The MicroPython board will read the data from the soil moisture sensors and then display it via a HTML web page upon request.

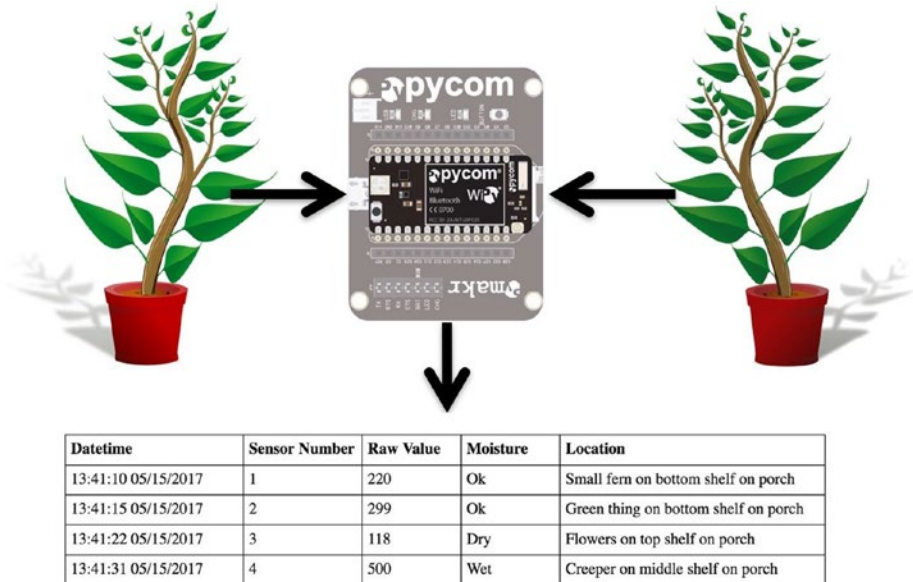


Figure 10-1. Plant Monitoring Project Concept

The user interface for this project is a web page that consists of a table that includes all the data read from the log file. This is how we can overcome potential issues of running the HTML server in a loop. That is, we do not have to interrupt the loop to read the sensors - that’s done via the timer alarm callback. Sadly, this technique only works for the WiPy and similar boards. We will have to use a different technique for the Pyboard.

By separating the sensor reading from the display, we can reuse or modify either without confusing ourselves as we dig into the code. For example, so long as the visualization component reads the sensor data from the file, it doesn’t matter to the sensor reading code how it is used. The only interface or connection between these two parts is the format of the file and since we’re using a CSV file, it is very easy to read and use in our code.

To make things more interesting and to make it easier to code, we will place all the sensor code in a separate code module. Recall, this is a technique used to help reduce the amount of code in any one module thereby making it easier to write and maintain.

Now let’s see what components are needed for this project and then we will see how to wire everything together.

■ **Note** Since we are well into our third project and have seen many of the techniques employed in this project, the discussion on the required components, wiring, and setup of the hardware shall be brief.

Required Components

Table 10-1 lists the components you will need. You can purchase the components separately from Adafruit (adafruit.com), Sparkfun (sparkfun.com), or any electronics store that carries electronic components. Links to vendors are provided should you want to purchase the components. When listing multiple rows of the same object, you can choose one or the other - you do not need both. Also, you may find other vendors that sell the components. You should shop around to find the best deal. Costs shown are estimates and do not include any shipping costs.

Table 10-1. *Required Components*

Component	Qty	Description	Cost	Links
MicroPython board	1	Pyboard v1.1 with headers	\$45-50	https://www.adafruit.com/product/2390 https://www.adafruit.com/product/3499 https://www.sparkfun.com/products/14413 https://store.micropython.org/store
		WiPy	\$25	https://www.adafruit.com/product/3338 https://www.pycom.io/product/wipy/
Soil Moisture Sensor	1+	Soil Moisture Sensor	\$6	https://www.sparkfun.com/products/13637
Networking Module (Pyboard)	1	CC3000 breakout board (or equivalent)	\$15+	various
Real-Time Clock	1	RTC optional for Pyboard and other boards without NTP support	\$10+	https://www.sparkfun.com/products/12708
Jumper wires (WiPy)	3*	M/M jumper wires, 6" (cost is for a set of 10 jumper wires)	\$4	https://www.sparkfun.com/products/8431

(continued)

Table 10-1. (continued)

Component	Qty	Description	Cost	Links
Jumper wires (Pyboard)	15+	M/M jumper wires, 6" (cost is for a set of 10 jumper wires)	\$4	https://www.sparkfun.com/products/8431
Power	1	USB cable to get power from PC		Use from your spares
	1	USB 5V power source and cable		Use from your spares

The number of jumper wires needed will vary depending on how many sensors you are using and whether you are using a MicroPython board that needs a networking module. You will need three jumpers for each sensor and, if you are using the CC3000 SPI breakout board, eight additional wires. If you plan to use the Pyboard, you will also want to add a real-time clock to keep time while powered off.

The Pyboard firmware has no network time protocol (NTP) server support. Thus, you must either initialize the onboard RTC each time you power on the board, attach a backup battery to the board (see the underside of the board – you will see the pins you will need to solder), or add a real-time clock module.

Soil moisture sensors come in a variety of formats but most have two prongs that are inserted into the soil and using a small electrical charge, measure the resistance between the prongs. The higher the value read, the more moisture is in the soil. However, there is a bit of configuration needed to obtain reliable or realistic thresholds. While the manufacturer will have threshold recommendations, some experimentation may be needed to find the right values.

These sensors can also be affected by environmental factors including the type of pot the plant is in, the soil composition, and other factors. Thus, experimenting with a known overwatered soil, dry soil, and properly tended soil will help you narrow down the thresholds for your environment.

Figure 10-2 shows a soil moisture sensor from Sparkfun that has a terminal mount instead of pins. You can find several varieties of these sensors. Just pick the one you want to use, keeping in mind you may need different jumpers to connect it to your board.



Figure 10-2. Soil Moisture Sensor (courtesy of sparkfun.com)

Of special note is how these soil moisture sensors work. If you were to leave the sensors powered on, they can degrade over time. The metal on the prongs can become degraded due to electrolysis thereby dramatically reducing its lifespan. You can use a technique of a GPIO pin to power the sensor by turning the pin on when you want to read a value. Keep in mind there will be a small delay while the sensor settles, but we can use a simple delay to wait and then read the value and turn the sensor off. In this way, we can extend the life of the sensor greatly.

Fortunately, the wiring for this project is less complex than the last two projects. Now, let's see how to wire the components together.

Set Up the Hardware

Table 10-2 shows the connections needed for this project. This shows only two sensors, but you can add several more if you'd like. However, it is recommended you start with one sensor until you get the project working and then add additional sensors. If it is easier, you can use a breadboard to connect the sensors to the MicroPython board, but depending on where you plan to place the board, you may not need it. It is your choice.

Table 10-2. Connections for the MicroPython (Pyboard and WiPy)

MicroPython Board			
WiPy	Pyboard	Component	Wire Color
P13	X19	Sensor 1: VCC	
GND	GND	Sensor 1: GND	
P19	X20	Sensor 1: SIG	
P14	X21	Sensor 2: VCC	
GND	GND	Sensor 2: GND	
P20	X22	Sensor 2: SIG	

Of course, you must insert the soil moisture sensors into the soil of your plants. If your plants are located further away from your power source, you may need to use longer wires to connect the sensors. You should start with a single, small plant and one sensor (or for testing, two sensors in one plant) that you can place close to your PC (or power source).

■ **Caution** You will need soil moisture sensors that can operate at 3.3–5V. Some MicroPython boards may limit output on the pins to 3.3V. The sensors from Sparkfun are compatible.

Let us see how to make the connections shown in the chart for the WiPy and Pyboard.

WiPy

Wiring for the WiPy is best done orienting the board with the USB connector to the left. Figure 10-3 shows the wiring diagram for the WiPy. Notice the use of a small breadboard to help with the ground connections. Notice also Sensor 1 is on the left and Sensor 2 is on the right.

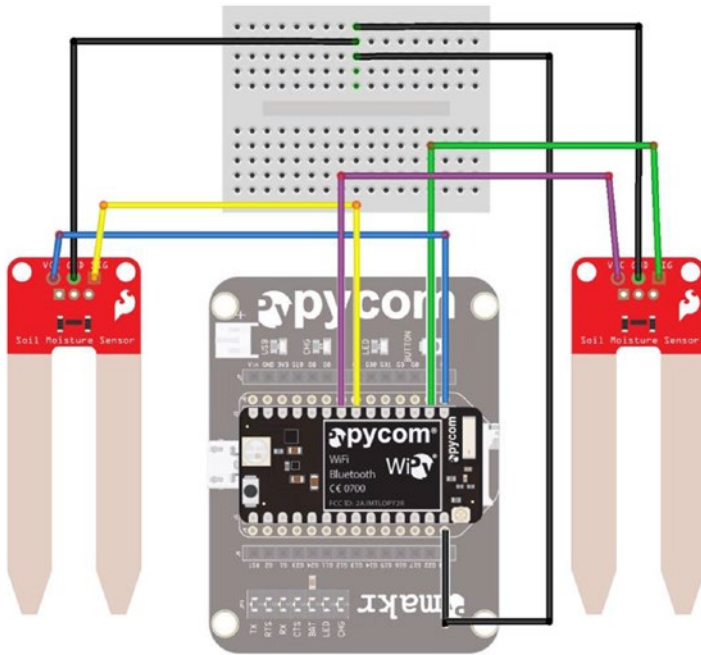


Figure 10-3. Wiring the Plant Monitor (WiPy)

Pyboard

Wiring for the Pyboard is best done orienting the board with the USB connector to the left. Figure 10-4 shows the wiring diagram for the Pyboard. You will likely want to use a breadboard so you can connect your networking module like the one shown. Notice also Sensor 1 is on the left and Sensor 2 is on the right.

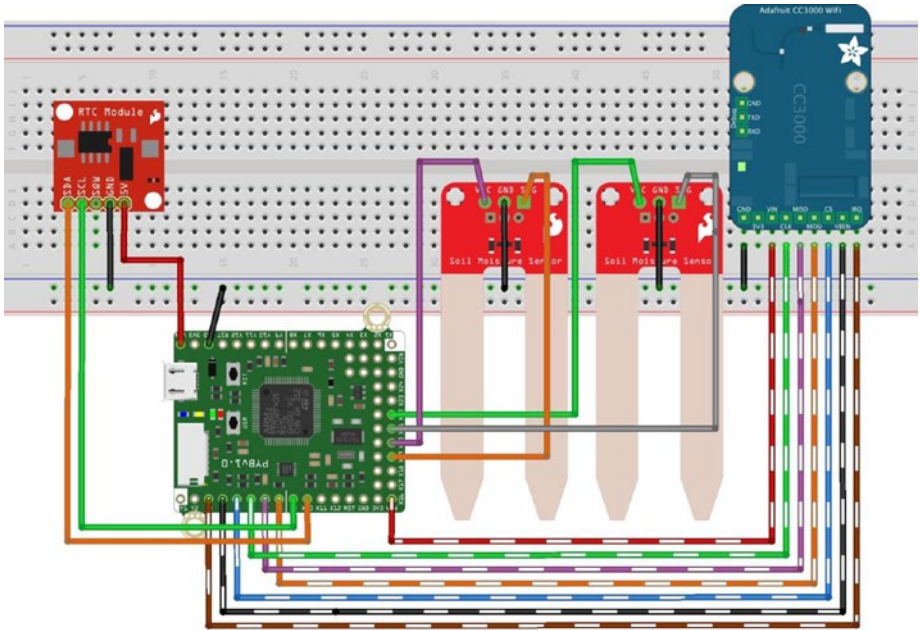


Figure 10-4. Wiring the Plant Monitor (Pyboard)

Once again, always make sure to double-check your connections before powering the board on. Now, let's talk about the code we need to write. Don't power on your board just yet – there is a fair amount of discussion needed before we're ready to test the project.

Write the Code

Now it's time to write the code for our project. The code is longer than what we've seen thus far, and due to all the bits and bobs we're working with, it is best to divide the project into parts. So, we are going to write the code in two stages. We won't have a working project until the end, so most of the discussion will be about the individual parts. We will put it all together before testing the project.

Recall from the overview, we will have two major components: the main code and a code module that encapsulates the soil sensors. We will place the HTML server code and supporting functions in the main code module. However, before we embark on the code for the project, we should calibrate our sensors. Let's do that now.

■ **Note** These sections demonstrate the code for the WiPy. This project is best suited for the WiPy and boards with similar capabilities: namely, WiFi and NTP support. Differences for implementing the project on Pyboard and Pyboard clone boards are given at the end of the chapter. As you will see, it is a bit more work to get the project working on those boards.

Calibrating the Sensor

Calibration of sensors is very important. This is especially true for soil moisture sensors because there are so many different versions available. These sensors are also very sensitive to the soil composition, temperature, and even the type of pot in which the plant lives. Thus, we should experiment with known soil moisture so we know what ranges to use in our code.

More specifically, we want to classify the observation from the sensor so that we can determine if the plant needs watering. We will use the values “dry”, “Ok”, and “wet” to classify the value read from the sensor. Seeing these labels is much easier for us to determine – at a glance – whether the plant needs watering. In this case, the raw data such as a value of 1756 may not mean much, but if we see “dry”, we know it needs water.

Since the sensors are analog sensors, we will use the analog-to-digital conversion on the board. When we read the data from the pin, we will get a value in the range of 0-4096. This value is related to the resistance the sensor reads in the soil. Low values indicate dry soil and high values indicate wet soil.

However, the sensors from different vendors can vary widely in the values read. For example, sensors from Sparkfun tend to read values in the range 0-1024 but sensors from other vendors can read as high as 4096. Fortunately, they all seem to be consistent in that the lower the value, the drier the soil.

So, we must determine thresholds for the three classifications. Again, there are several factors that can influence the values read from the sensor. Thus, you should select several pots of soil including one that you feel is dry, another that is correctly watered, and a third that is overwatered. The best thing to do is select one that is dry, take measurements, then water it until the soil moisture is correct, measure that, then water it again until there is too much water.¹

To determine the threshold, we must first write a short bit of code to set up our board for reading values from the sensor. This includes choosing a GPIO pin that supports ADC. Check the reference drawing for your board to be certain which GPIO pins are included. For example, on the WiPy, there are several ADC GPIO sections including those in the range P13-P18.

We also need to choose a pin to use to power the board. This is also an analog output pin. Finally, we will write a loop to read several values every 5 seconds and then average them. Five seconds is an arbitrary value and it was derived from reading the datasheet for the sensor. Check your sensors to see how much time is needed for the read to settle (may

¹Be sure to choose a plant hearty enough to withstand overwatering.

be under the heading of frequency of reads). Listing 10-1 shows the code needed to setup the analog-to-digital channel, a pin to use for powering the sensor, and a loop for reading 10 values and averaging them.

Listing 10-1. Calibrating the Soil Moisture Threshold (WiPy)

```
# MicroPython for the IOT - Chapter 10
#
# Project 3: MicroPython Plant Monitoring
#
# Threshold calibration for soil moisture sensors
#
# Note: this only runs on the WiPy.
from machine import ADC, Pin
import utime

# Setup the ADC channel for the read (signal)
# Here we choose pin P13 and set attn to 3 to stabilize voltage
adc = ADC(0)
sensor = adc.channel(pin='P13', attn=3)

# Setup the GPIO pin for powering the sensor. We use Pin 19
power = Pin('P19', Pin.OUT)
# Turn sensor off
power.value(0)

# Loop 10 times and average the values read
print("Reading 10 values.")
total = 0
for i in range (0,10):
    # Turn power on
    power.value(1)
    # Wait for sensor to power on and settle
    utime.sleep(5)
    # Read the value
    value = sensor.value()
    print("Value read: {0}".format(value))
    total += value
    # Turn sensor off
    power.value(0)

# Now average the values
print("The average value read is: {0}".format(total/10))
```

If you enter this code in a file named `threshold.py`, you can copy it to your WiPy and execute it. Listing 10-2 shows the output of running this calibration code in a plant that is correctly watered.

Listing 10-2. Running the Calibration Code (WiPy)

```
>>> import threshold
Reading 10 values.
Value read: 1724
Value read: 1983
Value read: 1587
Value read: 1702
Value read: 1634
Value read: 1525
Value read: 1874
Value read: 1707
Value read: 1793
Value read: 1745
The average value read is: 1727.4
>>>
```

Here we see an average value of 1727 (always round the number – you need integers). Further tests running the code on dry soil resulted in a value of 425 and for a wet plant, 3100. Thus, the thresholds for this example are 500 for dry, 2500 for wet. However, your results may vary greatly so make sure to run this code with your sensors, board, and plant of choice.

■ **Tip** To make things easier for calibrating the thresholds, use sensors from the same vendor. Otherwise, you may have to use a different set of thresholds for each sensor supported.

Notice the values read. As you can see, the values can vary from one moment to another. This is normal for these sensors. They are known for producing some jumpy values. Thus, you should consider sampling the sensor more than once to get an average over a short period rather than a single value. Even taking an average can be skewed slightly if one or more of the samples is off by a large margin. However, sampling even 10 values and averaging will help reduce the possibility of getting an anomalous reading. We will do this in our project code.

Now that we have our threshold values for our sensors, we can begin with the code module for the sensors.

Part 1: Sensor Code Module

The first part of the project will be to create a code module to contain a new class named `PlantMonitor` that contains all the functionality to read data from the sensors and save the data to a file. In this section, we will see how to write the code for the module. If you want to follow along writing the code, you can open a new file and name it `plant_monitor.py`. Let's start by looking at the high-level design.

High-Level Design

As we learned earlier, it is a good idea to create a design for each code module (class) we want to use. We will use the code module from the main code. Thus, we need functions to tell the class to read the sensors and a way to get the filename the class uses for the data.

Typically, one would design a code module to completely hide a file and all operations on it, but in this case the class is only concerned with reading the sensors and writing the data. Plus, since the main code needs to read the data and format it with HTML tags, it is more appropriate to place the read function in the main code. That is, you should strive to keep like code functions together, which helps maintaining the code. For example, keeping all the HTML code in one file makes modifying the HTML code (or reusing it) easier.

We will use a timer interrupt to read the sensors. This allows us to set up a function to be called periodically without the need to monitor or poll the time and call the function directly. We provide the clear log function as a convenience. Thus, we will only need two public functions: one to clear the log and another to fetch the filename. Aside from initializing the class, we only need to get the filename when we want to refresh the data (send it to the client). Table 10-3 shows the functions for the Plant Monitoring class.

Table 10-3. *High-Level Design (Functions) Plant Monitor Class*

Function	Parameters	Description
<code>__init__()</code>	<code>rtc</code>	Initialization for the class (the constructor)
<code>clear_log()</code>		Clear the log
<code>get_filename()</code>		Get the filename we're using after the check for SD card
<code>_get_value()</code>	<code>sensor, power</code>	Read the sensor 10 times and average the values read
<code>_read_sensors()</code>	<code>line</code>	Monitor the sensors, read the values, and save them
<code>_convert_value()</code>	<code>value</code>	Convert the raw sensor value to an enumeration

Notice the first function named `__init__()`. This is the constructor for the class and will be called when the class is instantiated from our main code. Notice also the private methods are named with a single underscore.

The following sections explain the initialization code and the functions needed. We will see the complete code in a later section.

Setup and Initialization

In this section, we discuss the code we need to set up and initialize the code module. First, there are a few imports we need including those for the analog-to-digital converter, pin, secure disk (SD), timer, and the operating system libraries.

We will also need some constants defined for the class. Recall we want to classify the soil moisture read with an enumeration. To do so, we will need to use the thresholds we determined for the classification. We can use constants at the top of the file to make it easier to change them later should we need to adjust the code for use with other sensors or the conditions of our plants changes (different pot, soil, environment, etc.). We can use the same philosophy to set the name of the file to contain the data.

We also use a constant to define the frequency for reading the sensors. Since we will use a loop to read the sensor waiting 5 seconds for each read, we will need a minimum of 50-55 seconds to read 10 values. Thus, we cannot set the update frequency to anything less than about one minute. The frequency is in seconds. While you may want to set this to a low value for testing, you certainly do not want to check the soil moisture of your plants every minute. That is, how often to you check your plants normally? Once every few days or once a day? Why check it sooner than normal?

SAMPLING FREQUENCY

How often you sample data from a sensor (also called sampling rate) is often overlooked when designing sensor networks. The tendency is to store as many values as you can; thinking more data is better. But that is not applicable in the general case. Consider the plant monitoring project. If you normally check your plants once per day, how can sampling the sensors once every 5 minutes benefit you? It won't and it only results in excess data!

Sampling rate must be calculated carefully to deliver the data you need to draw conclusions without creating too much data. While more data is always better than too little data, saving data too often at unrealistic frequencies can generate so much data that it could exceed the storage capacity of your device.

You should carefully consider sampling rate when designing projects that sample sensors. Choose a sampling rate that is based on realistic expectations. Generally, if you are sampling data that can change very slowly, the sampling rate should be low. Sampling data that can change more quickly should have a higher (shorter time between samples) sampling rate.

Finally, we need a function that converts a time structure to a string. Recall from an earlier example, we can use a simple format specification. We will use a module-level private function for that feature.

Listing 10-3 shows the code for the setup and initialization section. Place this at the top of the file.

Listing 10-3. Plant Monitor Class Setup and Initialization (WiPy)

```

from machine import ADC, Pin, SD, Timer
import os
import utime

# Thresholds for the sensors
LOWER_THRESHOLD = 500
UPPER_THRESHOLD = 2500
UPDATE_FREQ = 120 # seconds

# File name for the data
SENSOR_DATA = "plant_data.csv"

# Format the time (epoch) for a better view
def _format_time(tm_data):
    # Use a special shortcut to unpack tuple: *tm_data
    return "{0}-{1:0>2}-{2:0>2} {3:0>2}:{4:0>2}:{5:0>2}".format(*tm_data)

```

Constructor

The constructor for the class is where all the major work takes place. There are several things we need to do including the following.

- Normalize the location (path) of the data file
- Set up the sensors in a dictionary stored in a list
- Set up the timer interrupt to read the sensors periodically

We normalize the path to the data file by attempting to use the SD card. If the SD card cannot be found, we default to the flash drive. However, you should avoid writing data to the flash drive since the driver is smaller and can fill up, and writing to the flash drive increases the risk of corrupting the drive or causing problems during execution.

We use a dictionary for each sensor so we can define the pin for the sensor, pin for powering the sensor, sensor number (an arbitrary identification), and the location of the sensor. We then place the dictionaries in a list to make it easy to read all the sensors at the same time using a loop.

Finally, we set up an interrupt via the timer alarm class to read the sensors periodically. Listing 10-4 shows the code for the class constructor.

Listing 10-4. Plant Monitor Class Constructor (WiPy)

```

def __init__(self, rtc):
    self.rtc = rtc

    # Try to access the SD card and make the new path
    try:
        sd = SD()
        os.mount(sd, '/sd')

```

```

        self.sensor_file = "/sd/{0}".format(SENSOR_DATA)
        print("INFO: Using SD card for data.")
    except:
        print("ERROR: cannot mount SD card, reverting to flash!")
        self.sensor_file = SENSOR_DATA
    print("Data filename = {0}".format(self.sensor_file))

    # Setup the dictionary for each soil moisture sensor
    adc = ADC(0)
    soil_moisture1 = {
        'sensor': adc.channel(pin='P13', attn=3),
        'power': Pin('P19', Pin.OUT),
        'location': 'Green ceramic pot on top shelf',
        'num': 1,
    }
    soil_moisture2 = {
        'sensor': adc.channel(pin='P14', attn=3),
        'power': Pin('P20', Pin.OUT),
        'location': 'Fern on bottom shelf',
        'num': 2,
    }
    # Setup a list for each sensor dictionary
    self.sensors = [soil_moisture1, soil_moisture2]
    # Setup the alarm to read the sensors
    a = Timer.Alarm(handler=self._read_sensors, s=UPDATE_FREQ,
                    periodic=True)
    print("Plant Monitor class is ready...")

```

Notice the code for the timer alarm. Here we define the handler (callback) for the interrupt, the frequency using the constant we defined at the top of the file, and set it to fire every *N* seconds (periodically).

Public Functions

There are only two public functions. The first, `clear_log()`, simply opens the file for writing and closes it. This effectively empties the file. The function is provided for convenience. The second function, `get_filename()`, simply returns the name of the file used to store data. This name is not the same as the name in the constant `SENSOR_DATA` because we normalized the path in the constructor as shown in the previous section.

Private Functions

There are three private functions. The `_get_value()` function is the same code from our threshold calibration code where we sample the sensor 10 times and average the value. The `_read_sensors()` function is the callback for the timer alarm interrupt that reads all the sensors we have defined and saves the data to the file. The `_convert_value()`

function is a helper function to determine the classification of the soil based on the sensor data. This function returns a string or “dry”, “Ok”, or “wet”.

Complete Code

Now that we have seen all the parts of the code module, let’s look at the completed code. Listing 10-5 shows the complete code for the Plant Monitor code module. Once again, we can save this file as `plant_monitor.py`.

Listing 10-5. Plant Monitor Code Module Complete Code (WiPy)

```
# MicroPython for the IOT - Chapter 10
#
# Project 3: MicroPython Plant Monitoring
#
# Plant monitor class
#
# Note: this only runs on the WiPy.
from machine import ADC, Pin, SD, Timer
import os
import utime

# Thresholds for the sensors
LOWER_THRESHOLD = 500
UPPER_THRESHOLD = 2500
UPDATE_FREQ = 120 # seconds

# File name for the data
SENSOR_DATA = "plant_data.csv"

# Format the time (epoch) for a better view
def _format_time(tm_data):
    # Use a special shortcut to unpack tuple: *tm_data
    return "{0}-{1:0>2}-{2:0>2} {3:0>2}:{4:0>2}:{5:0>2}".format(*tm_data)

class PlantMonitor:
    """
    This class reads soil moisture from one or more sensors and writes the
    data to a comma-separated value (csv) file as specified in the constructor.
    """

    # Initialization for the class (the constructor)
    def __init__(self, rtc):
        self.rtc = rtc
```

```

# Try to access the SD card and make the new path
try:
    sd = SD()
    os.mount(sd, '/sd')
    self.sensor_file = "/sd/{0}".format(SENSOR_DATA)
    print("INFO: Using SD card for data.")
except:
    print("ERROR: cannot mount SD card, reverting to flash!")
    self.sensor_file = SENSOR_DATA
print("Data filename = {0}".format(self.sensor_file))

# Setup the dictionary for each soil moisture sensor
adc = ADC(0)
soil_moisture1 = {
    'sensor': adc.channel(pin='P13', attn=3),
    'power': Pin('P19', Pin.OUT),
    'location': 'Green ceramic pot on top shelf',
    'num': 1,
}
soil_moisture2 = {
    'sensor': adc.channel(pin='P14', attn=3),
    'power': Pin('P20', Pin.OUT),
    'location': 'Fern on bottom shelf',
    'num': 2,
}
# Setup a list for each sensor dictionary
self.sensors = [soil_moisture1, soil_moisture2]
# Setup the alarm to read the sensors
a = Timer.Alarm(handler=self._read_sensors, s=UPDATE_FREQ,
                periodic=True)
print("Plant Monitor class is ready...")

# Clear the log
def clear_log(self):
    log_file = open(self.sensor_file, 'w')
    log_file.close()

# Get the filename we're using after the check for SD card
def get_filename(self):
    return self.sensor_file

# Read the sensor 10 times and average the values read
def _get_value(self, sensor, power):
    total = 0
    # Turn power on
    power.value(1)

```

```

for i in range (0,10):
    # Wait for sensor to power on and settle
    utime.sleep(5)
    # Read the value
    value = sensor.value()
    total += value
# Turn sensor off
power.value(0)
return int(total/10)

# Monitor the sensors, read the values and save them
def _read_sensors(self, line):
    log_file = open(self.sensor_file, 'a')
    for sensor in self.sensors:
        # Read the data from the sensor and convert the value
        value = self._get_value(sensor['sensor'], sensor['power'])
        print("Value read: {0}".format(value))
        time_data = self.rtc.now()
        # datetime,num,value,enum,location
        log_file.write(
            "{0},{1},{2},{3},{4}\n".format(_format_time(time_data),
                sensor['num'], value,
                self._convert_value(value),
                sensor['location']))

    log_file.close()

# Convert the raw sensor value to an enumeration
def _convert_value(self, value):
    # If value is less than lower threshold, soil is dry else if it
    # is greater than upper threshold, it is wet, else all is well.
    if (value <= LOWER_THRESHOLD):
        return "dry"
    elif (value >= UPPER_THRESHOLD):
        return "wet"
    return "ok"

```

Wow, that's a lot of code! Take some time to read through it until you understand all the parts of the code.

Changes for the Pyboard

There are considerably more changes needed for this project to run on the Pyboard. The main reasons for this include the usual changes we need for the imports, the pin class high/low versus value, and differences in how we use the real-time clock. Finally, the Pyboard does not support timer alarm class interrupts so we must use a polling technique to read the sensors. This last change means we must make the `read_sensors()` function a public function so we can call it from the main code.

Since the changes are numerous, a difference file is nearly the same length as the actual code.² Thus, we will see the complete code for the Pyboard. Listing 10-6 shows the complete code for the code module with the differences needed to adapt the code module to the Pyboard (in bold). While most changes are minimal, if you are using the Pyboard or a Pyboard clone, take note of the pins used for the sensor.

Listing 10-6. Plant Monitor Code Module Complete Code (Pyboard)

```
# MicroPython for the IOT - Chapter 10
#
# Project 3: MicroPython Plant Monitoring
#
# Plant monitor class
#
# Note: this only runs on the Pyboard.
from pyb import ADC, delay, Pin, SD
import os
import pyb

# Thresholds for the sensors
LOWER_THRESHOLD = 500
UPPER_THRESHOLD = 2500
UPDATE_FREQ = 120 # seconds

# File name for the data
SENSOR_DATA = "plant_data.csv"

# Format the time (epoch) for a better view
def _format_time(tm_data):
    # Use a special shortcut to unpack tuple: *tm_data
    return "{0}-{1:0>2}-{2:0>2} {3:0>2}:{4:0>2}:{5:0>2}".format(*tm_data)

class PlantMonitor:
    """
    This class reads soil moisture from one or more sensors and writes the
    data to a comma-separated value (csv) file as specified in the constructor.
    """

    # Initialization for the class (the constructor)
    def __init__(self, rtc):
        self.rtc = rtc
```

²This happens quite often. When it does, it is better to view the actual code as the difference file may be harder to read.

```

# Try to access the SD card and make the new path
try:
    self.sensor_file = "/sd/{0}".format(filename)
    f = open(self.sensor_file, 'r')
    f.close()
    print("INFO: Using SD card for data.")
except:
    print("ERROR: cannot mount SD card, reverting to flash!")
    self.sensor_file = SENSOR_DATA
print("Data filename = {0}".format(self.sensor_file))

# Setup the dictionary for each soil moisture sensor
soil_moisture1 = {
    'sensor': ADC(Pin('X19')),
    'power': Pin('X20', Pin.OUT_PP),
    'location': 'Green ceramic pot on top shelf',
    'num': 1,
}
soil_moisture2 = {
    'sensor': ADC(Pin('X20')),
    'power': Pin('X21', Pin.OUT_PP),
    'location': 'Fern on bottom shelf',
    'num': 2,
}
# Setup a list for each sensor dictionary
self.sensors = [soil_moisture1, soil_moisture2]
# Setup the alarm to read the sensors
self.alarm = pyb.millis()
print("Plant Monitor class is ready...")

# Clear the log
def clear_log(self):
    log_file = open(self.sensor_file, 'w')
    log_file.close()

# Get the filename we're using after the check for SD card
def get_filename(self):
    return self.sensor_file

# Read the sensor 10 times and average the values read
def _get_value(self, sensor, power):
    total = 0
    # Turn power on
    power.high()
    for i in range(0,10):
        # Wait for sensor to power on and settle
        delay(5000)
        # Read the value

```



```

        value = sensor.read()
        total += value
    # Turn sensor off
    power.low()
    return int(total/10)

# Monitor the sensors, read the values and save them
def read_sensors(self):
    if pyb.elapsed_millis(self.alarm) < (UPDATE_FREQ * 1000):
        return
    self.alarm = pyb.millis()
    log_file = open(self.sensor_file, 'a')
    for sensor in self.sensors:
        # Read the data from the sensor and convert the value
        value = self._get_value(sensor['sensor'], sensor['power'])
        print("Value read: {0}".format(value))
        time_data = self.rtc.datetime()
        # datetime,num,value,enum,location
        log_file.write(
            "{0},{1},{2},{3},{4}\n".format(_format_time(time_data),
                                          sensor['num'], value,
                                          self._convert_value(value),
                                          sensor['location']))

    log_file.close()

# Convert the raw sensor value to an enumeration
def _convert_value(self, value):
    # If value is less than lower threshold, soil is dry else if it
    # is greater than upper threshold, it is wet, else all is well.
    if (value <= LOWER_THRESHOLD):
        return "dry"
    elif (value >= UPPER_THRESHOLD):
        return "wet"
    return "ok"

```

Ok, now we're ready to look at the main code.

Part 2: Main Code

The main code is like the code for the last project. However, this time we will use a file to store the HTML code (since it doesn't change) and a single HTML string for populating a HTML table with the data from the file. We will also add code to read the date and time from a network time protocol (NTP) server.

Unlike the last project, the HTML code does not include a button but we can format a command manually on the URL. We can use this technique to allow access to commands without using buttons or other user interface features. It also helps to make these commands harder to use to prevent overuse. For example, we can provide a clear

logs command. We would use a URL like `http://192.168.42.140/CLEAR_LOG`, which submits a GET request to the HTML server. We can capture that command and clear the log when it is issued.

■ **Caution** If you build commands like this, be sure to use them carefully. That is, setting your URL to `http://192.168.42.140/CLEAR_LOG` and pressing enter issues the command. Refreshing the page will reissue the command! When you use the command, be sure to clear your URL before refreshing or, better, use it once and close the page/tab.

The following sections explain the initialization code and the functions needed. We will see the complete code in a later section. Let's start with the HTML code.

HTML Code (Files)

We will store the HTML code needed in files to save memory. Recall by reading a row at a time – we do not have to take up space with the strings in our code. As your projects grow in complexity, this could become an issue. Thus, this project demonstrates a way to save some memory.

The HTML for this project creates a web page with a simple table that includes all the data in the file at the time of the request. To make things easier, we will use three files. The first file (named `part1.html`) will contain the HTML code up to the table rows; the second file (named `plant_data.csv`), which is populated by the `PlantMonitor` class; and the third (named `part2.html`) will contain the remaining HTML code.

The first file, `part1.html`, is shown in Listing 10-7. This file establishes the table HTML code. It also establishes characteristics for the table including text alignment, border size, and padding – all through cascading style (`<style>` tag). Don't worry if this looks strange or alien. You can google for W3C standards to see how we use the tag to control the style of the web page.

Listing 10-7. HTML Code (`part1.html`)

```
<!DOCTYPE html>
<html>
  <head>
    <title>MicroPython for the IOT - Project 3</title>
    <meta http-equiv="refresh" content="30">
    <style>
      table, th, td {
        border: 1px solid black;
        border-collapse: collapse;
      }
      th, td {
        padding: 5px;
      }
    </style>
  </head>
  <tbody>
    <tr>
      <td></td>
    </tr>
  </tbody>
</html>
```

```

    th {
        text-align: left;
    }
</style>
</head>
<center><h2>MicroPython for the IOT - Project 3</h2></center><br>
<center>A simple project to demonstrate how to retrieve sensor data over
the Internet.</center>
<center><br><b>Plant Monitoring Data</b><br><br>
<table style="width:75%">
  <col width="180">
  <col width="120">
  <col width="100">
  <col width="100">
  <tr><th>Datetime</th><th>Sensor Number</th><th>Raw Value</
th><th>Moisture</th><th>Location</th></tr>

```

Notice the table code. Again, don't worry if this seems strange. It works and it is very basic. Those familiar with HTML may want to embellish and improve the code. The last line establishes the header for the table.

The second file, `plant_data.csv`, contains the data. We will use a constant to populate a properly formatted HTML table row. The following shows an example of what a row of data would look like in the file and how that data is transformed to HTML. We will see the HTML for the table row in the next section.

```

# Raw data
2017-08-08 20:26:17,1,78,dry,Small fern on bottom shelf
# HTML table row
<tr><td>2017-08-08 20:26:17</td><td>1</td><td>78</td><td>dry</td><td>Small
fern on bottom shelf </td></tr>

```

The last file, `part2.html`, contains the closing tags so it isn't very large. But since we're reading from files, we include this file. The following shows the code in the second file.

```

</table>
</center>
</html>

```

So, how do we use these files? When we send a response back to the client (the web page), we read the first file sending one row at a time, then read the data file sending one row at a time, then read the last file sending one row at a time. We will use a helper function to read the data file. Listing 10-8 shows the code used to do this.

Listing 10-8. Reading the HTML and Data File (WiPy)

```

# Read HTML from file and send to client a row at a time.
def send_html_file(filename, client):
    html = open(filename, 'r')

```

```

for row in html:
    client.send(row)
html.close()

# Send the sensor data to the client.
def send_sensor_data(client, filename):
    send_html_file("part1.html", client)
    log_file = open(filename, 'r')
    for row in log_file:
        cols = row.strip("\n").split(",") # split row by commas
        # build the table string if all parts are there
        if len(cols) >= 5:
            html = HTML_TABLE_ROW.format(cols[0], cols[1], cols[2],
                                         cols[3], cols[4])
            # send the row to the client
            client.send(html)
    log_file.close()
    send_html_file("part2.html", client)

```

Imports

The imports we need for the project include those for the real-time clock, `sys`, `usocket`, `utime`, `machine`, and `WLAN`. These are now quite familiar. The last line imports the `PlantMonitor` class from the `plant_monitor` code module. The complete list of imports is shown below. If you want to follow along, open a new file and name it `plant_wipy.py`.

```

# Imports for the project
from machine import RTC
import sys
import usocket as socket
import utime
import machine
from network import WLAN
from plant_monitor import PlantMonitor

```

We also need a string we can use to create the rows for the table. The HTML code that occurs before this line is saved in files as described above. The following shows the string used. Notice we use replacement syntax so that we can use the `format()` function to fill in the details.

```

# HTML web page for the project
HTML_TABLE_ROW = "<tr><td>{0}</td><td>{1}</td><td>{2}</td><td>{3}</td><td>{4}</td></tr>"

```

Imports

We also want to connect the project to our network. We use the same code as we used in previous projects and examples but instead make it a function named `connect()`, which we will call from the main `run()` function. Be sure to change the SSID and password to match your network.

Network Time Protocol

Since we are saving data that has a time element (you want to know when you sampled the soil moisture), we need to store the date and time with the data. The easiest way to do this is to use a network time protocol (NTP) server: that is, provided the board is connected to the Internet. If it is not connected to the Internet, we must use a RTC module or initialize the onboard RTC at startup. We saw how to use a NTP server in Chapter 5. We repeat it in this project as a function named `get_ntp()`, which we will call from the main `run()` function.

The `run()` Function

The HTML server portion of the `run()` function is like the last project but instead of processing form requests, we send the web page back to the client by default. The only command supported is the `CLEAR_LOG` command, which requires specifying it on the URL on the client as described above.

Another difference is instead of placing code in the global section of the code file (so that it executes when we import the file in our REPL console or `main.py` file), we use functions to connect to the network, set up the NTP, and send the HTML code to the client. This is an escalation of complexity that you should start using as general practice. We did not see this in earlier projects so that we could concentrate on getting the code complete. When writing your own projects, be sure to use functions to contain code and call the functions from your other code.

Since this is different from the last project, we will look at the code. Listing 10-9 shows the code for the `run()` function.

Listing 10-9. Plant Monitor `Run()` Function (WiPy)

```
# Setup the socket and respond to HTML requests
def run():
    # Connect to the network
    if not connect():
        sys.exit(-1)

    # Setup the real time clock
    rtc = get_ntp()

    # Setup the plant monitoring object instance from the plant_monitoring class
    plants = PlantMonitor(rtc)
    filename = plants.get_filename()
```

```

# Setup the HTML server
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind('', 80)
sock.listen(5)
print("Ready for connections...")
server_on = True
while server_on:
    client, address = sock.accept()
    print("Got a connection from a client at: %s" % str(address))
    request = client.recv(1024)

    # Allow for clearing of the log, but be careful! The auto refresh
    # will resend this command if you do not clear it from your URL
    # line.

    if (request[:14] == b'GET /CLEAR_LOG'):
        print('Requesting clear log.')
        plants.clear_log()
        send_sensor_data(client, filename)
        client.close()
sock.close()

```

Take a moment to read through this code. Notice how we implement the function in a more modular way. Not only does placing common code in functions help with how you break a problem down into parts, it also makes your main code (the `run()` function) shorter.

Let's look at the completed code.

Complete Code

Now that we have seen all the parts of the code module, let's look at the completed code. Listing 10-10 shows the complete code for the Plant Monitor code module. Once again, we can save this file as `plant_wipy.py` or `plant_pyboard.py` for the Pyboard).

Listing 10-10. Plant Monitor Main Code (WiPy)

```

# MicroPython for the IOT - Chapter 10
#
# Project 3: MicroPython Plant Monitoring
#
# Required Components:
# - WiPy
# - (N) Soil moisture sensors (one for each plant)
#
# Note: this only runs on the WiPy.
#
# Imports for the project

```

```

from machine import RTC
import sys
import usocket as socket
import utime
import machine
from network import WLAN
from plant_monitor import PlantMonitor

# HTML web page for the project
HTML_TABLE_ROW = "<tr><td>{0}</td><td>{1}</td><td>{2}</td><td>{3}</td><td>{4}</td></tr>"

# Setup the board to connect to our network.
def connect():
    wlan = WLAN(mode=WLAN.STA)
    nets = wlan.scan()
    for net in nets:
        if net.ssid == 'YOUR_SSID':
            print('Network found!')
            wlan.connect(net.ssid, auth=(net.sec, 'YOUR_PASSWORD'), timeout=5000)
            while not wlan.isdisconnected():
                machine.idle() # save power while waiting
            print('WLAN connection succeeded!')
            print("My IP address is: {0}".format(wlan.ifconfig()[0]))
            return True
    return False

# Setup the real time clock with the NTP service
def get_ntp():
    rtc = RTC()
    print("Time before sync:", rtc.now())
    rtc.ntp_sync("pool.ntp.org")
    while not rtc.synced():
        utime.sleep(3)
        print("Waiting for NTP server...")
    print("Time after sync:", rtc.now())
    return rtc

# Read HTML from file and send to client a row at a time.
def send_html_file(filename, client):
    html = open(filename, 'r')
    for row in html:
        client.send(row)
    html.close()

# Send the sensor data to the client.
def send_sensor_data(client, filename):

```

```

send_html_file("part1.html", client)
log_file = open(filename, 'r')
for row in log_file:
    cols = row.strip("\n").split(",") # split row by commas
    # build the table string if all parts are there
    if len(cols) >= 5:
        html = HTML_TABLE_ROW.format(cols[0], cols[1], cols[2],
                                     cols[3], cols[4])
        # send the row to the client
        client.send(html)
log_file.close()
send_html_file("part2.html", client)

# Setup the socket and respond to HTML requests
def run():
    # Connect to the network
    if not connect():
        sys.exit(-1)

    # Setup the real time clock
    rtc = get_ntp()

    # Setup the plant monitoring object instance from the plant_monitoring class
    plants = PlantMonitor(rtc)
    filename = plants.get_filename()

    # Setup the HTML server
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(('', 80))
    sock.listen(5)
    print("Ready for connections...")
    server_on = True
    while server_on:
        client, address = sock.accept()
        print("Got a connection from a client at: %s" % str(address))
        request = client.recv(1024)

        # Allow for clearing of the log, but be careful! The auto refresh
        # will resend this command if you do not clear it from your URL
        # line.

        if (request[:14] == b'GET /CLEAR_LOG'):
            print('Requesting clear log.')
            plants.clear_log()
            send_sensor_data(client, filename)
            client.close()
sock.close()

```


Changes for the Pyboard

Like the code module previously, the changes needed to run the code on the Pyboard are numerous. The main reasons for this include the usual changes but also the code is quite different because we must use a polling method since there is no (easy) way to set up a timer interrupt like we did on the WiPy.

Since the changes are numerous, a difference file is nearly the same length as the actual code.³ Thus, we will see the complete code for the Pyboard. Listing 10-11 shows the complete code for the code module with the differences needed to adapt the code module to the Pyboard (in bold). While most changes are minimal, if you are using the Pyboard or a Pyboard clone, take note of the pins used for the sensor.

There are two major changes needed in the `run()` function. First, there is not NTP support in the Pyboard firmware so we must use an RTC module or initialize the onboard RTC each time we start the project. Second, since there is no way to set up a timer interrupt that permits access to files, we must change the HTML server to use a non-blocking socket technique. These changes are highlighted in bold.

Listing 10-11. Plant Monitor Main Code (Pyboard)

```
# MicroPython for the IOT - Chapter 10
#
# Project 3: MicroPython Plant Monitoring
#
# Required Components:
# - Pyboard
# - (N) Soil moisture sensors (one for each plant)
#
# Note: this only runs on the Pyboard.
#
# Imports for the project
from pyb import delay, SPI
from pyb import I2C
import network
import urtc
import usocket as socket
from plant_monitor import PlantMonitor

# Setup the I2C interface for the rtc
i2c = I2C(1, I2C.MASTER)
i2c.init(I2C.MASTER, baudrate=500000)

# HTML web page for the project
HTML_TABLE_ROW = "<tr><td>{0}</td><td>{1}</td><td>{2}</td><td>{3}</td><td>{4}</td></tr>"
```

³This happens quite often. When it does, it is better to view the actual code as the difference file may be harder to read.

```

# Setup the board to connect to our network.
def connect():
    nic = network.CC3K(SPI(2), Pin.board.Y5, Pin.board.Y4, Pin.board.Y3)
    # Replace the following with your SSID and password
    print("Connecting...")
    nic.connect("YOUR_SSID", "YOUR_PASSWORD")
    while not nic.isconnected():
        delay(50)
    print("Connected!")
    print("My IP address is: {}".format(nic.ifconfig()[0]))
    return True

# Read HTML from file and send to client a row at a time.
def send_html_file(filename, client):
    html = open(filename, 'r')
    for row in html:
        client.send(row)
    html.close()

# Send the sensor data to the client.
def send_sensor_data(client, filename):
    send_html_file("part1.html", client)
    log_file = open(filename, 'r')
    for row in log_file:
        cols = row.strip("\n").split(",") # split row by commas
        # build the table string if all parts are there
        if len(cols) >= 5:
            html = HTML_TABLE_ROW.format(cols[0], cols[1], cols[2],
                                         cols[3], cols[4])

            # send the row to the client
            client.send(html)
            delay(50)
    log_file.close()
    send_html_file("part2.html", client)

# Setup the socket and respond to HTML requests
def run():
    # Connect to the network
    if not connect():
        sys.exit(-1)

# Setup the real time clock
rtc = urtc.DS1307(i2c)
#
# NOTE: We only need to set the datetime once. Uncomment these
# lines only on the first run of a new RTC module or
# whenever you change the battery.
# (year, month, day, weekday, hour, minute, second, millisecond)

```

```

#start_datetime = (2017,07,20,4,9,0,0,0)
#rtc.datetime(start_datetime)

# Setup the plant monitoring object instance from the plant_monitoring class
plants = PlantMonitor(rtc)
filename = plants.get_filename()

# Setup the HTML server
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind('', 80)
sock.setblocking(False) # We must use polling for Pyboard.
sock.listen(5)
print("Ready for connections...")
server_on = True
while server_on:
    try:
        client, address = sock.accept()
    except OSError as err:
        # Do check for reading sensors here
        plants.read_sensors()
        delay(50)
        continue
    print("Got a connection from a client at: %s" % str(address))
    request = client.recv(1024)

    # Allow for clearing of the log, but be careful! The auto refresh
    # will resend this command if you do not clear it from your URL
    # line.

    if (request[:14] == b'GET /CLEAR_LOG'):
        print('Requesting clear log.')
        plants.clear_log()
        send_sensor_data(client, filename)
        client.close()
sock.close()

```

Notice the try block in the run() function. The sock.accept() function will throw an exception when there is no client connected. This is different than how a blocking call works. A blocking call will simply wait until a client connects. Here, we must attempt to accept a connection and if it fails, continue waiting for a connection. That is, we cannot send the HTML to the client as there is not client!

These changes highlight the added difficulty in using the Pyboard and other MicroPython boards without networking support, those that require RTC modules, and those with limited support for timer alarm interrupts to permit quasi-asynchronous execution (it's not really asynchronous). In other words, we can disconnect some of our code to be called (triggered) by interrupts.

Now, let's run this project!

WAIT, WHAT ABOUT THE DATA FILE?

If you are wondering about the data file, you need not worry. The code is designed to create the file even if it doesn't exist. The following shows a mockup of data you can use in your tests.

```
2017-08-08 20:26:17,1,78,dry,Small fern on bottom shelf on porch
2017-08-08 20:26:32,2,136,dry,Green pot creeper thing on floor in living room
2017-08-08 20:26:47,1,128,dry,Small fern on bottom shelf on porch
2017-08-08 20:27:02,2,112,dry,Green pot creeper thing on floor in living room
```

If you want to start with some sample data, you can do so but just make sure it is comma separated with no spaces and one line of data per row.

Execute!

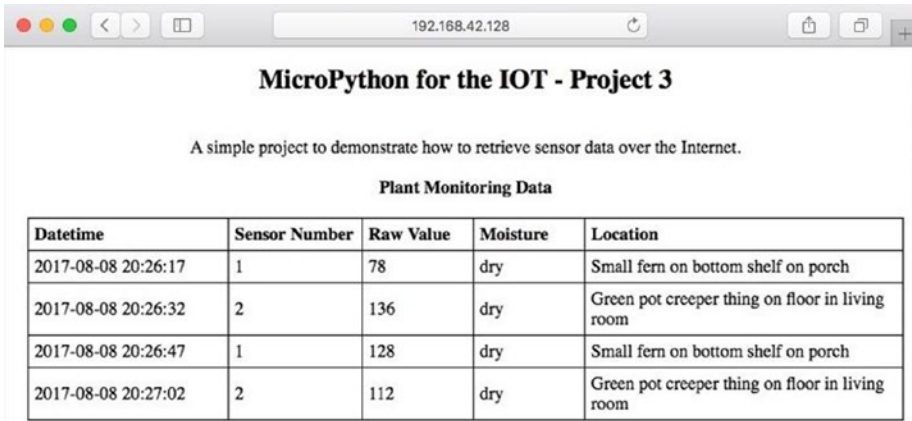
Now is the fun part! We've got the code all set up to read soil moisture from our plants and send all the sensor data collected to the client. Recall, we need to copy the code to our board. We can ftp the two code files (`plant_monitor.py` and `plant_wipy.py`) as well as the two HTML files (`part1.html` and `part2.html`). Do that now and you are ready to test the project.

All we need now is the IP address of that board to point our web browser. We can get that from our debug statements by running the code. Listing 10-12 shows the initial run for the project on a WiPy (results for the Pyboard are similar).

Listing 10-12. Running the Plant Monitor (WiPy)

```
MicroPython v1.8.6-694-g25826866 on 2017-06-29; WiPy with ESP32
Type "help()" for more information.
>>> import plant_wipy as p
>>> p.run()
Network found!
WLAN connection succeeded!
My IP address is: 192.168.42.128
Time before sync: (1970, 1, 1, 0, 1, 44, 382593, None)
Waiting for NTP server...
Time after sync: (2017, 8, 9, 14, 26, 1, 92051, None)
INFO: Using SD card for data.
Data filename = /sd/plant_data.csv
Plant Monitor class is ready...
Ready for connections...
Got a connection from a client at: ('192.168.42.110', 50395)
```

Notice in this case the IP address is 192.168.42.128. All we need to do is put that in our browser and shown in Figure 10-5.



MicroPython for the IOT - Project 3

A simple project to demonstrate how to retrieve sensor data over the Internet.

Plant Monitoring Data

Datetime	Sensor Number	Raw Value	Moisture	Location
2017-08-08 20:26:17	1	78	dry	Small fern on bottom shelf on porch
2017-08-08 20:26:32	2	136	dry	Green pot creeper thing on floor in living room
2017-08-08 20:26:47	1	128	dry	Small fern on bottom shelf on porch
2017-08-08 20:27:02	2	112	dry	Green pot creeper thing on floor in living room

Figure 10-5. *Plant Monitor Project*

Once you enter the URL, you should see a web page like the image shown. If you don't, be sure to check the HTML in your code to ensure it is exactly like what is shown; otherwise, the page may not display properly. You should also ensure the network your PC is connected to can reach the network to which your board is connected. If your home office is set up like mine, there may be several WiFi networks you can use. It is best if your board and your PC are on the same network (and same subnet).

At this point, you've completed another real MicroPython IOT project. In this case, we saw an IOT project that collects and displays data. Cool!

Taking it Further

This project, like the last one, shows excellent prospects for reusing the techniques in other projects. This is especially true for the HTML server aspect. If you liked seeing your sensor data over the Internet, you should consider taking time to explore some embellishments. Here are a few you may want to consider. Some are easy and some may be a challenge or require more research.

- Add more sensors to expand your project to more plants.
- Add a temperature sensor to record ambient temperature and display it on the web page.
- Rewrite the HTML code to produce JSON strings.
- Rewrite the HTML code to produce XML.

- Explore the HTML code to change the web page to your liking. Consider using cascading style sheets to change the background of the button when pressed.
- Connect your board to the Internet and call a friend to connect to your board and try it out.
- Add LEDs to your board to illuminate when the plants need watering.

Of course, if you want to press on to the next project, you're welcome to do so but take some time to explore these potential embellishments – it will be good practice.

Summary

IOT solutions can take many forms. One of the more common forms is those that generate data that we can view over the Internet (sometimes called data collectors). The implementation of data collectors can vary greatly, but they generally store the data in some location and provide a way to view the data. The simplest forms are those that log the data (sometimes called data loggers) locally, on a remote server, in a database, or in a cloud service. The visualization of the data can also vary with the most basic providing the data via a web page.

In this chapter, we saw a MicroPython IOT project that logs data read from a series of soil moisture sensors. We created a plant monitoring solution that saved the data to the local SD card. The project also served the data via a HTML server so that we can see the data at any time. This project can be used as a template for a host of data collection projects. You can simply follow the pattern established in this chapter and build your own HTML-based data logger.

In the next chapter, we will close out our tour of MicroPython IOT projects by making our MicroPython board send data to a cloud-based⁴ storage and visualization service. Cool!

⁴Sadly, some would argue it isn't IOT unless it involves cloud services of one form or another.



Project 4: Using Weather Sensors

Building a full-fledged IOT project requires using technologies that permit your small MicroPython board to collect and send data to services on the Internet that can store, retrieve, and visualize the data. These Internet services are often cloud-based services. Chances are, you've used some of these technologies without knowing. We've already seen an early, simple form of this in the last chapter – using an HTML server to send data to a client over the Internet.

Sending data via HTML may be fine for some projects, but projects where you may need to visualize the data in some other form or if you want to perform analysis on the data, will require a more advanced mechanism to transmit and store the data. Fortunately, there are many technologies we can use, including those that permit the controlled transmission of data.

Such technologies have a defined protocol (a way to communicate) and often are supported by specialized programming interfaces. One of the easiest to use with microcontrollers defines a protocol with publish and subscribe roles. That is, you can publish data (write) or you can subscribe to data (read) and even be notified when new data is available. One of the easiest-to-use publish/subscribe protocols is called message queue telemetry transport (MQTT).

We will use MQTT in this chapter to see an example of a complete IOT project that reads data from sensors and sends the data to a server, which can then be accessed by clients that use MQTT to subscribe to the data. Best of all, some MQTT services like the one we will use in this chapter also provide visualization tools that allow you to see the data as it is being generated.

Fortunately, the sensor used in this chapter is easier to use than those from other chapters. The complexity in this project comes from using cloud services to host the data and MQTT to publish the data. As you will see, it isn't that difficult to build. Let's find out more about the project for this chapter.

Overview

In this chapter, we will implement a simplified weather station IOT solution. We will record temperature, barometric pressure, and humidity using a small sensor on a breakout board. While those three data points aren't exactly a complete, professional (much less hobbyists) grade weather solution, it shows you precisely how easy it is to send data from the sensor to the cloud.

We will send the data to the cloud using MQTT to publish the data. We will also use a subscription to monitor another data element (called a feed) that is connected to a switch. The idea is borrowed from previous projects where we used a button on a web page to trigger an event. In this case, we will use the switch and a subscription to the data to turn our sensor on and off where on means no data is published. In this sense, our MicroPython solution becomes a sensor node.

The user interface for this project will be built using a feature of the MQTT service that permits you to create a site that displays the data using a variety of modular components (called blocks) such as graphs and dials. As you will see, creating the user interface is a lot easier than the HTML we wrote in previous projects.¹ But it has one limitation – it only displays data saved since it was launched. We will learn more about that later when we test the project.

Before we look at the hardware for the project, let's take a short detour to learn about MQTT and the service we will use in the chapter.

Message Queue Telemetry Transport

The publish/subscribe model has been around for some time at least in theory and concept. There are also programming constructs that implement the roles. A publisher publishes data to a location (a server, database, or repository of structured data) that permits subscribers to get the data. Thus, publishers are writers and subscribers are readers.

In the case of IOT projects and sensor networks, we have one or more sensor nodes or data providers sending data to the repository. Rather than use a structured storage mechanism such as SQL, we can use a message queue that records messages that contain the data. When subscribers subscribe to the data, they get the messages in the order that they were received and parse the message for the data. Thus, they do not have to add a data abstraction layer like those we would use with a database server. In this case, the MQTT protocol is all you need.

MQTT is a simple and very lightweight protocol (meaning it doesn't require a huge library with a complex set of steps to use) that you can use with your MicroPython (and other microcontroller platforms). Since MQTT is based on a message queue, the protocol is very tolerant of unreliable delivery of data. And since it doesn't require a lot of memory to use, it can be used on small devices. What this means is MQTT is a way to ensure your small IOT devices can send data to a server (called a broker) with a reasonable assurance

¹Some savvy HTML programmers may beg to differ.

of delivery – both for publishers and subscribers. This makes MQTT a perfect tool for use in IOT projects.

Interestingly, MQTT has been around since 1999. It was invented by Dr. Andy Stanford-Clark of IBM and Alren Nipper of Arcom (Eurotech). It has changed very little since then and has been adapted to a growing number of platforms. For more information about MQTT, see <http://mqtt.org/faq>.

Let's see how MQTT works in a bit more detail.

How It Works

There are three components to a basic MQTT service: sensor nodes or publishers that produce messages containing data, clients or subscribers that read the messages, and a broker or server that stores the messages distributing them to subscribers. Figure 11-1 shows a concept of how the three components work.

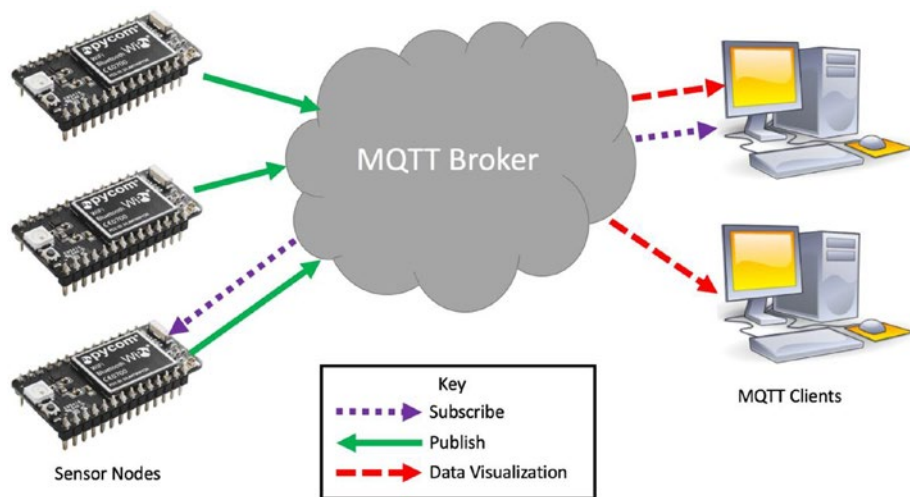


Figure 11-1. MQTT Concept

Notice we have sensor nodes on the left that can publish data (depicted by solid lines), clients on the right that can subscribe to messages (depicted by dotted lines), and a visualization component provided by the broker (depicted as dashed lines). Notice also there are sensor nodes that can publish and subscribe as well as a client that can visualize data and subscribe to messages. In fact, you can have any combination of publish and subscribe. We will see how to both publish and subscribe to messages from our MicroPython project.

Clients

A MQTT client is simply a device (or computer) that has a MQTT library that you can use to program your device to send messages (publish) to or read messages (subscribe) from a broker. Fortunately, the WiPy and some of the other ESP-based microcontroller ports of MicroPython include a MQTT library. Unfortunately, there isn't one for the Pyboard. Thus, we will only look at the WiPy MicroPython board in this chapter.

Brokers

There are several brokers you can use including those that are cloud based as well as one you can use on your own server. Most brokers have their own implementation of the MQTT client (some require specific drivers or libraries) and may run on a specific platform or require other components. Regardless, they support the same MQTT protocol from the sensor nodes and clients. Thus, you can choose to use on MQTT broker and later move to another without having to rewrite your code from scratch.

For IOT projects, you will want to choose a broker that is cloud based so that you can connect your board to the cloud and access the data from anywhere. The broker of choice for this project is Adafruit IO (io.adafruit.com). Adafruit IO is currently in public beta, which means it is available to anyone who wants to use it. There are some restrictions – none of which will affect experiments like the project in this chapter. If you plan to base a commercial product on Adafruit IO, you may want to wait until it is out of beta or consider an alternative broker service.

ALTERNATIVE MQTT BROKERS

The website <https://github.com/mqtt/mqtt.github.io/wiki/servers> contains a list of MQTT brokers. You will find some that cater to a specific platform or programming language such as Java. You will also find some commercial MQTT services you can use for larger projects or projects you want to monetize.

However, if you want to set up your own MQTT server on your network, you can use Mosquito (<http://mosquitto.org/>), which is open source and very popular among hobbyists. There is even a public test server you can use. Better still, you won't have to change the code in this project much to use it. You should consider Mosquito as a stepping stone from a free MQTT broker like Adafruit IO to a commercial MQTT broker.

Getting Started with Adafruit IO

Adafruit IO (io.adafruit.com) is a cloud-based data visualization system that is easy to use, requiring very little programming knowledge to use. This is accomplished with support for representational state transfer (REST) and MQTT APIs. We will be using the MQTT API for this project.

REST is a protocol used by many cloud-based services such as those provided by Amazon Web Services (AWS) as well as many commercial cloud-based solutions. It is best known for use with web-based solutions via HTML operations. See https://en.wikipedia.org/wiki/Representational_state_transfer for more information about REST. We will only use MQTT in this chapter.

The goal of Adafruit IO was to cut out all the complexity of the current data loggers and cloud-based data services solutions and make it easy to use and Adafruit has done so very well. Briefly, we use a MQTT client driver (library), write our code to connect to, and subscribe to data or publish data. The visualization part takes place on the Adafruit IO server itself where we create our own use interface to see the data. There are four steps to getting started with Adafruit IO.

1. Create an account.
2. Set up feeds (message queues) for your data.
3. Set up a dashboard to visualize the data.
4. Connect your devices and start publishing and subscribing.

All you need to get started with Adafruit IO is a user account. To create an account for Adafruit IO, just head over to <https://io.adafruit.com/>. If you already have an account on Adafruit's server, you can use that one and just register for access to Adafruit IO. The process is simple and easy to follow. Once you are logged in, you will see the administrative interface where you can create feeds and dashboards.

A feed is the core component of Adafruit IO. A feed is where you place your data in the form of messages from your devices. Feeds can store data (via publish), and devices (clients) can read the data by subscribing to the feed. You can have more than one feed and each is defined by its name and referenced via your user id.

Dashboards are the views of the data in the feeds. Adafruit IO provides a drag-and-drop interface for building simple views of the data very quickly using predefined user-interface controls called "blocks". Each dashboard can have one or more blocks, which can be connected to your feeds. Data is then shown in the blocks automatically updating when new data arrives.

Once we have our feeds and dashboard set up, we can write our code to use the MQTT library and send (or receive) data. We can then return to our dashboard and view the data. We will see a detailed walkthrough for setting up the feeds and a dashboard for the project in this chapter in a later section.

If you want to learn more about io.adafruit.com, check out these excellent tutorials from Adafruit. You can also find some interesting project ideas for using Adafruit IO at <https://learn.adafruit.com/search?q=io.adafruit.com&>.

- *Adafruit IO*: <https://learn.adafruit.com/adafruit-io/rest-api?view=all>
- *Adafruit IO Basics – Feeds*: <https://learn.adafruit.com/adafruit-io-basics-feeds/resources?view=all>

- *Adafruit IO Basics – Dashboards*: <https://learn.adafruit.com/adafruit-io-basics-dashboards/creating-a-dashboard?view=all>
- *MQTT, Adafruit IO & You*: <https://learn.adafruit.com/mqtt-adafruit-io-and-you/arduino-plus-library-setup?view=all>

Now, let's see what components we need for this project.

Required Components

Table 11-1 lists the components you will need. You can purchase the components separately from Adafruit (adafruit.com), Sparkfun (sparkfun.com), or any electronics store that carries electronic components. Links to vendors are provided should you want to purchase the components. When listing multiple rows of the same object, you can choose one or the other - you do not need both. Also, you may find other vendors that sell the components. You should shop around to find the best deal. Costs shown are estimates and does not include any shipping costs.

Table 11-1. *Required Components*

Component	Qty	Description	Cost	Links
MicroPython board	1	WiPy	\$25	https://www.adafruit.com/product/3338 https://www.pycom.io/product/wipy/
Weather Sensor	1	BME280 (I2C interface)	\$20	https://www.sparkfun.com/products/13676
Jumper wires	4	M/M jumper wires, 6" (cost is for a set of 10 jumper wires)	\$4	https://www.sparkfun.com/products/8431
Breadboard	1	Prototyping board, half-sized	\$5	https://www.sparkfun.com/products/12002
Power	1	USB cable to get power from PC		Use from your spares
	1	USB 5V power source and cable		Use from your spares

BME280 breakout board atmospheric sensors can support either I2C or SPI interfaces. The BME280 from Sparkfun supports both, which makes it ideal for an IOT parts kit. Figure 11-2 shows the BME280 weather sensor from Sparkfun. If you have one from another vendor, make sure it supports I2C (or adjust the code in this project and the wiring diagram accordingly). It does not come with headers so you can add those yourself. Just order a set when you order the sensor and solder them yourself (or get a friend to help you).

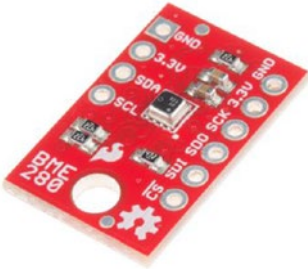


Figure 11-2. Atmospheric Sensor Breakout (courtesy of sparkfun.com)

Fortunately, the wiring for this project is less complex than the last two projects. Now, let's see how to wire the components together.

Set Up the Hardware

The simplicity of this project is that we're using the one sensor and the I2C interface so we do not need many connections. Figure 11-3 shows the wiring diagram for the WiPy.

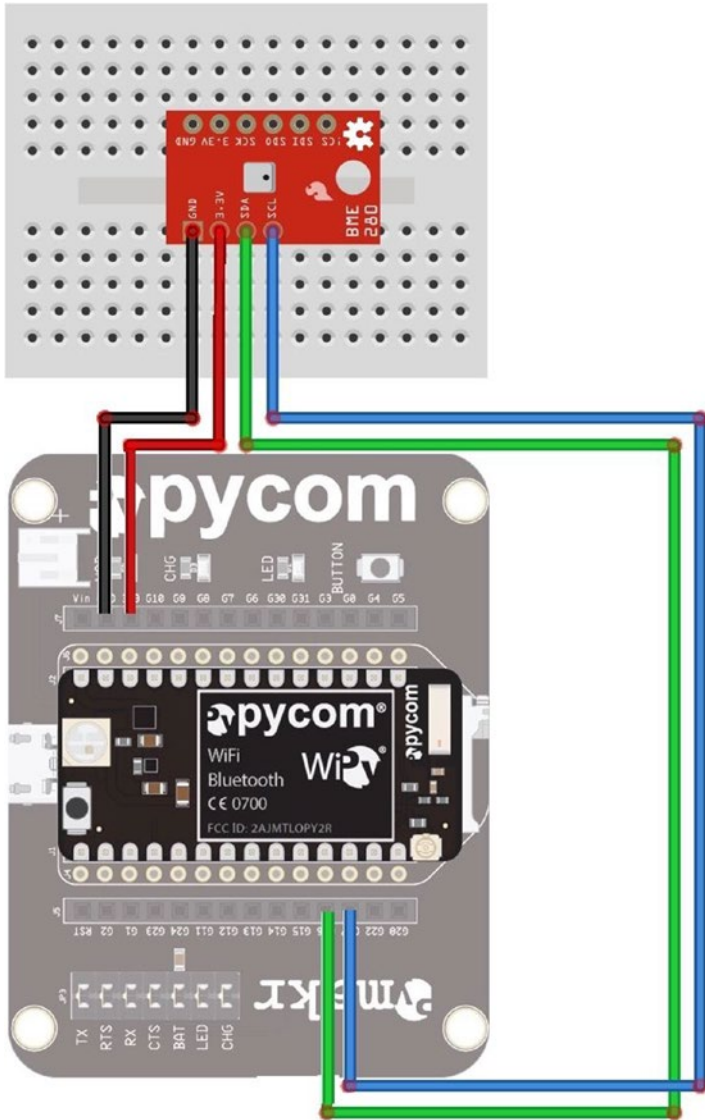


Figure 11-3. Wiring the Weather Sensor (WiPy)

■ **Caution** The Sparkfun BME280 is limited to 3.3V. Be sure to connect it only to the 3.3V pin on the WiPy.

Once again, always make sure to double-check your connections before powering the board on. Don't power on your board just yet – we need to set up our Adafruit IO feeds and dashboard then write the code before we're ready to test the project.

Configure Adafruit IO

Now we are ready to configure Adafruit IO. Recall our project will read three types of data from the BME280 sensor: temperature, humidity, and barometric pressure. Thus, we will create one feed for each of these. We will also use a fourth feed so we can control our MicroPython board via the dashboard. The dashboard will have several blocks to show each of the feeds. Let's see how to create the feeds and dashboard for our project.

Set Up Feeds

To set up a feed, log in to Adafruit IO and select *Feeds* from the list of links on the left of the screen, then click the *Actions* drop-down list and choose *Create a New Feed* as shown in Figure 11-4.

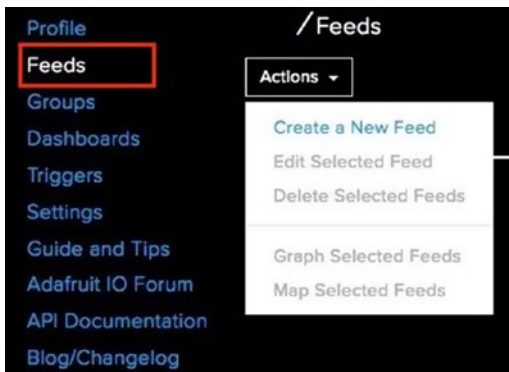


Figure 11-4. Create a new Feed

Next, you can provide a name and description (optional) for the feed. Name the first feed Temperature as shown in Figure 11-5. When you're satisfied with the data entered, click *Create* to create the feed.

Figure 11-5. *Create a New Feed dialog*

Repeat the process and create feeds for storing humidity (named Humidity), barometric pressure (named Pressure), and one for controlling the sensors (named Sensors). When complete, you should see the new feeds as shown in Figure 11-6.

Actions ▾				
<input type="checkbox"/>	Name ▾	Key ▾	Last Value ▾	Recorded ▾
<input type="checkbox"/>	Temperature	temperature	No Data Available	2 minutes ago
<input type="checkbox"/>	Humidity	humidity	No Data Available	a few seconds ago
<input type="checkbox"/>	Pressure	pressure	No Data Available	a few seconds ago
<input type="checkbox"/>	Sensors	sensors	No Data Available	a few seconds ago

Figure 11-6. *Feeds for the Weather Project*

Notice at this point there is no data in the feeds. We won't see any data until we've set up and connected our MicroPython board. Now that the feeds are set up, we can create a dashboard to see the data once we publish it.

Set Up a Dashboard

You can create as many dashboards as you want to view the data in your feeds. In fact, you can make one dashboard for each feed or several dashboards that connect to one or more of your feeds. For this project, we will create one dashboard that connects to all four feeds.

To set up a dashboard, select *Dashboards* from the list of links on the left of the screen, then click the *Actions* drop-down list and choose *Create a New Dashboard* as shown in Figure 11-7.

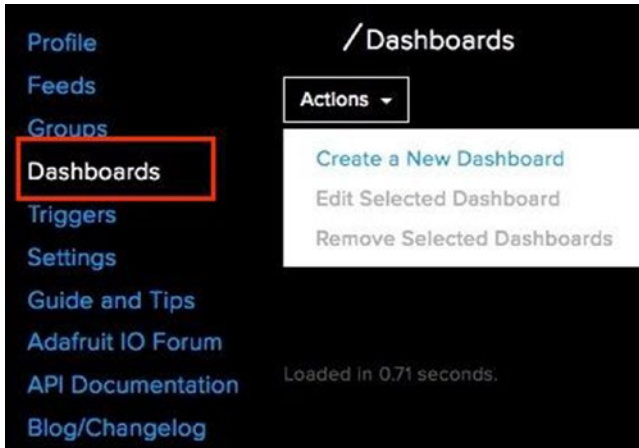


Figure 11-7. Create a new Dashboard

Next, you can provide a name and description (optional) for the dashboard. Name the dashboard *WeatherData* and provide a description (optional) as shown in Figure 11-8. When you're satisfied with the data entered, click *Create* to create the dashboard.

 A dialog box titled 'Create a new Dashboard' with a close button (X) in the top right corner. It contains two text input fields. The first field is labeled 'Name' and contains the text 'WeatherData'. The second field is labeled 'Description' and contains the text 'Weather data including temperature, humidity, and pressure. Control sensors with sensors feed.'. At the bottom right of the dialog are two buttons: 'Cancel' and 'Create'. The 'Create' button is highlighted in blue.

Figure 11-8. Create a New Dashboard dialog

Once you've created the dashboard, we must edit it to add blocks. To add blocks, click your dashboard. At this point, you have a blank dashboard where we can add one or more blocks. The available blocks and their uses are shown in Table 11-2. As you will see, there are blocks that can be used to display data in a variety of ways (called outputs in the Adafruit documentation) and those that can be used to generate data (called inputs) in a feed. Some blocks can be connected to one and only one feed, and other blocks can be connected to multiple feeds.

Table 11-2. Adafruit IO Blocks

Name	Type	Description
Toggle Button	input	Choose between two values either text or numeric. Think of it like a switch.
Momentary Button	input	Send a single value to a feed much like a hardware momentary button.
Number Slider	input	Select a number from a specified range that you define.
Gauge	output	Display the current value of a feed. Can show percentage if you set a min and max value.
Text Box	both	Display static text or text from a feed.
Stream	output	Display messages from one or more feeds.
Image	output	Display images in a feed.
Line Graph	output	Display data in a line graph from a feed.
Color Picker	input	Select a RGB value and send it to a feed.
Map	output	Track the locations of your feed data (if geo data is available).
Remote Control	input	Mimics the Mini Remote Control sold by Adafruit.

For our project, we will add the following blocks.

- Line Graph: one block each to connect the Temperature, Humidity, and Pressure feeds.
- Toggle Button: connect the Sensors feed.
- Stream: connect all four feeds.

To add a block, click on the plus sign icon in the upper-right area of the dashboard edit screen as shown in Figure 11-9.

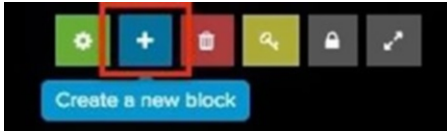


Figure 11-9. Add a Block

When you click the plus sign, you will be shown a list of blocks available. The blocks listed in Table 11-2 are shown in Figure 11-10, listed in order from the top left. To add the block of your choice to your dashboard, click on the block.

Create a new block ✕

Click on the block you would like to add to your dashboard. You can always come back and switch the block type later if you change your mind.

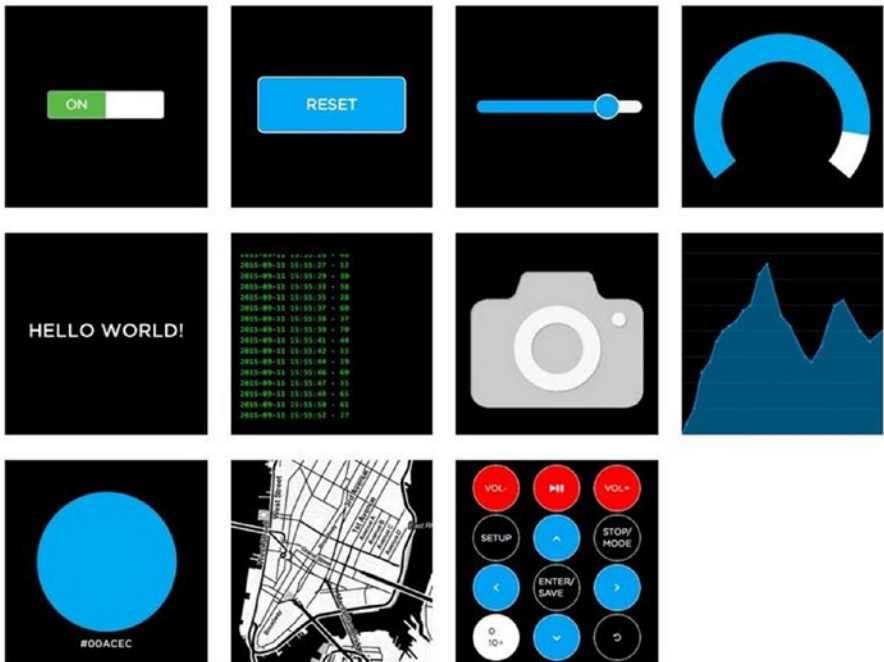


Figure 11-10. Available Blocks

When you select the block, you must select the feed(s) for the block. Figure 11-11 shows an example of selecting the feeds for the line graph for the Temperature feed. As you can see, you could select any number of feeds. However, be sure to consider the scale of each data before using a line graph for multiple feeds. In this project, we have different

scales for the three types of data. If we used one line graph, the feed with the lowest range would appear at the bottom of the graph and may obscure data variation over time (it may appear as a flat line).

Choose up to 5 feeds ✕

The line chart is used to chart one or more feeds.

If you have lot of feeds, you may want to use the search field. You can also create a feed quickly below.

Create

Group / Feed	Last value	Recorded	
<input checked="" type="checkbox"/> Temperature	🔒	8 days ago	1 of 5
<input type="checkbox"/> Humidity	🔒	8 days ago	
<input type="checkbox"/> Pressure	🔒	8 days ago	
<input type="checkbox"/> Sensors	🔒	8 days ago	

< Previous step

Next step >

Figure 11-11. Selecting the feeds for a block

Once you've selected the feeds by ticking the box next to the feed name, click on *Next step* to configure the block. The dialog will look different for each block type. Figure 11-12 shows the settings for the Temperature line graph block.

Block settings

In this final step, you can give your block a title and see a preview of how it will look. Customize the look and feel of your block with the remaining settings. When you are ready, click the "Create Block" button to send it to your dashboard.

Block Title

Show History

X-Axis Label

Y-Axis Label

Y-Axis Minimum

Y-Axis Maximum

Block Preview

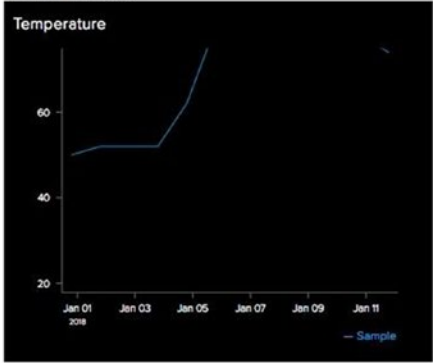


Figure 11-12. Block Settings (Temperature)

Notice I chose to set the name for the block to the same name as the feed and I changed the labels for the X- and Y-axes. I also set the minimum and maximum values for the Y-axis. When you are satisfied with the settings, click *Create block* to add the block to the dashboard.

Go ahead and create two more line graphs for the Humidity and Pressure feeds. You can name the blocks in a similar manner but use the range 0,100 for Humidity and range 80000,120000 for pressure. You may want or need to adjust those if the elevation of your geographic location differs, resulting in slightly different ranges.

Next, we add a block for logging all the data. For this, we will use a Stream block. Add the stream block and select all four feeds. You can name it however you want and the default settings are OK. Figure 11-13 shows the settings I used. This block is a nice block to use when setting up a new dashboard as it lets you see the data as it arrives in each field.

Block settings ✕

In this final step, you can give your block a title and see a preview of how it will look. Customize the look and feel of your block with the remaining settings. When you are ready, click the "Create Block" button to send it to your dashboard.

<p>Block Title</p> <input type="text" value="Data"/>	<p>Block Preview</p> <div style="background-color: black; color: white; padding: 5px;"> <p>Data</p> <pre> 2018-02-01 08:02:00 Temperature 78.34 2018-02-01 08:02:00 Temperature 78.34 2018-02-01 08:02:00 Lamp Color #FF0028 2018-02-01 08:02:00 username/errors Validation failed: Name may contain only letters, digits, underscores, spaces, or dashes 2018-02-01 08:02:00 Temperature 78.34 2018-02-01 08:02:00 username/throttle 58 seconds until reset, current limit is 125 requests every 60 seconds.</pre> </div>
<p>Font Size</p> <input type="text" value="Small"/>	
<p>Font Color</p> <input type="text" value="Green"/>	
<p>Show Feed Name?</p> <input type="text" value="Yes"/>	
<p>Show Timestamp?</p> <input type="text" value="Yes"/>	
<p>Show Errors?</p> <input type="text" value="Yes"/>	
	<input type="button" value="◀ Previous step"/> <input type="button" value="Create block"/>

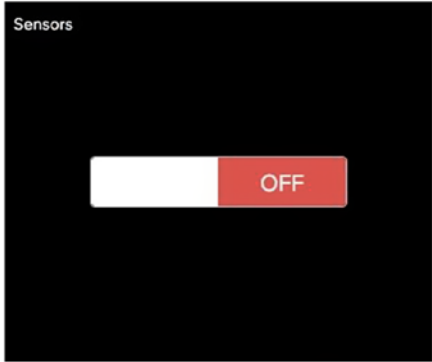
Figure 11-13. Block Settings (Stream Block)

Finally, we add a block for our switch to turn the recording of data on and off. Add a toggle button block and use the settings shown in Figure 11-14.

Block settings



In this final step, you can give your block a title and see a preview of how it will look. Customize the look and feel of your block with the remaining settings. When you are ready, click the "Create Block" button to send it to your dashboard.

Block Title	<input type="text" value="Sensors"/>	Block Preview 
Button On Text	<input type="text" value="ON"/>	
Button Off Text	<input type="text" value="OFF"/>	

[< Previous step](#) [Create block](#)

Figure 11-14. Block Settings (Sensors)

There is one more step. Adding blocks to your dashboard places them on the screen in the order they appear and at default sizes. If you look at your dashboard, you will see each block has edges that you can use to resize each block. You can also click on a block, hold down the mouse button, and drag the blocks around to your liking. Go ahead and do this now. You can use a layout like the one shown in Figure 11-15.

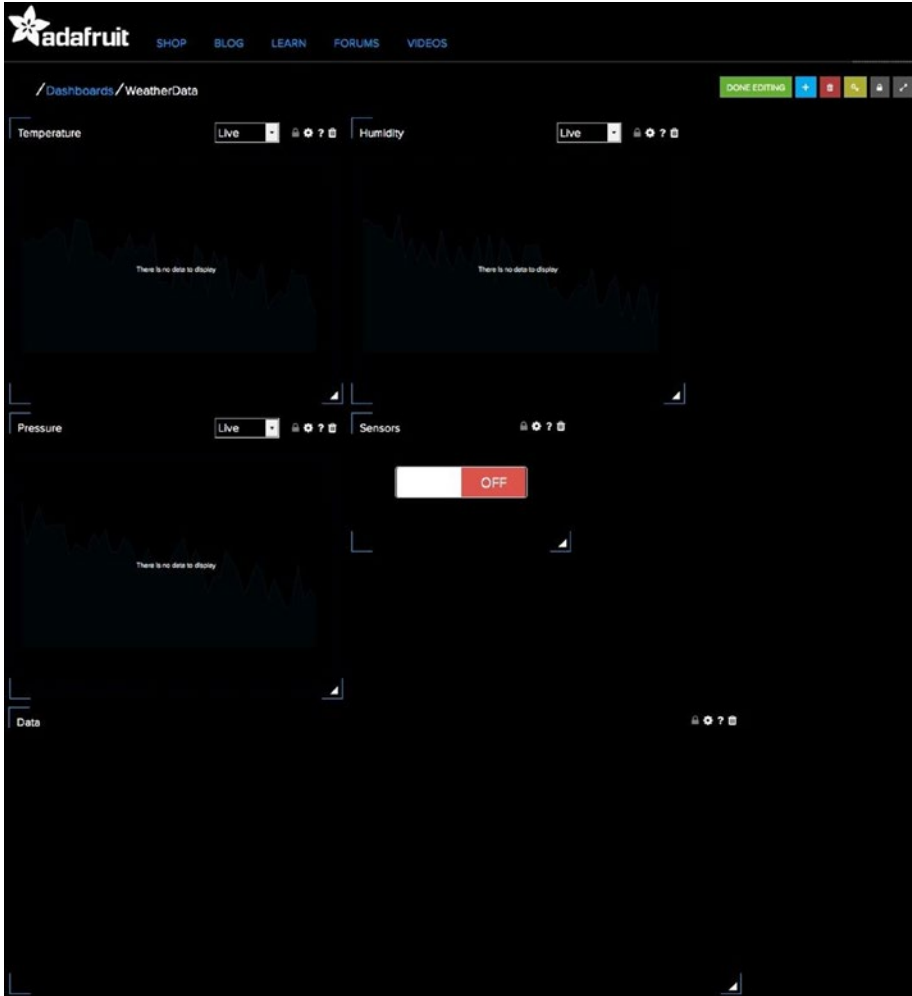


Figure 11-15. *Rearranging Blocks on the Dashboard*

When you are finished editing your dashboard, click on *DONE EDITING*. You can always edit your dashboard later if you want to change scales on one of the blocks or move the blocks around. To edit a block that you've already added, click on the small gear icon on the dashboard edit screen for the block you want to edit.

Get Your Credentials

There is one more thing you need to do. Connection to Adafruit IO is done through your user account and a special key generated by the system. We use this data instead of a user id and password. To retrieve your key, go to the main page and click *Settings*, then the *VIEW*

AIO KEY button. You will see a dialog appear like the one shown in Figure 11-16. Notice I have obscured the data. You should treat this data like you would any other password. If you want to see samples of how to use the values, click on *Show Code Samples*.

YOUR AIO KEY



Your Adafruit IO key should be kept in a safe place and treated with the same care as your Adafruit username and password. People who have access to your AIO key can view all of your data, create new feeds for your account, and manipulate your active feeds.

If you need to regenerate a new AIO key, all of your existing programs and scripts will need to be manually changed to the new key.



Username

Active Key REGENERATE AIO KEY

[Hide Code Samples](#)

Arduino

```
#define IO_USERNAME "XXXXXXXXXXXXXXXXXXXXXX"
#define IO_KEY      "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
"
```

Linux Shell

```
export IO_USERNAME="XXXXXXXXXXXXXXXXXXXXXX"
export IO_KEY="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
```

Figure 11-16. AIO Key

You will need both your user name and your IO Key in the source code so it is best to get these values now and save them some place safe. You can always refer to this page in the future if you do not want to save these values in a file. Also, if you ever need to regenerate your key, you can do so by clicking on *REGENERATE AIO KEY*, but don't do that unless you need a new key!

Ok, now we're ready to start sending some data. But first, we will need to write the code.

Write the Code

The code for this project is relatively easy to write once you understand the MQTT library. As you will see, it pairs well with Adafruit IO in ease of use. To make this project more interesting and more applicable to how you would write IOT projects for the MicroPython platform, we will place all the code for reading the sensors and publishing the data in a class. We will then use that class in a code module that we can call from our `main.py`

code module on boot. Since the code doesn't use any constructs that we haven't already seen, the description of the code will be brief and focus only on the new concepts and the MQTT code specifically.

However, we first need to download a MQTT library and a BME280 driver for the WiPy.

MQTT Driver

The MQTT library for the WiPy is available for download at <https://github.com/pycom/pycom-libraries>. I recommend downloading the entire repository. To do so, click on the *Clone or download* button, then click *Download Zip* and save the file on your PC. Once downloaded, you can unzip the file and then locate the MQTT library in the `/lib/mqtt` folder. The file we want is named `mqtt.py`. You will need to copy this file to your WiPy in a later step.

BME280 Library

The driver for the BME280 is available for download from https://bitbucket.org/oscarBravo/wipy_bme280. To download the driver, click on the download icon on the left side of the screen (looks like a cloud with an arrow pointing down), then click on *Download repository* on the list of downloads available. Once downloaded, unzip the file and find the driver named `bme280.py`.

While the driver will work with the WiPy, the address for the sensor may need to be changed. Be sure to check the address of the BME280 sensor you chose and ensure the address in the driver file (`bme280.py`) is set correctly. Failure to use the correct address will result in strange, fatal errors when you run the code. The following shows the line of code that specifies the address in hexadecimal.

```
# BME280 default address.
BME280_I2CADDR = 0x77
```

Recall, you can find the address of the I2C devices connected to your board using the `scan()` function for the I2C class as shown below. The function returns a decimal value, but you can convert it to hexadecimal as shown below. Be sure to connect your I2C sensor first!

```
MicroPython v1.8.6-694-g25826866 on 2017-06-29; WiPy with ESP32
Type "help()" for more information.
>>> from machine import I2C
>>> i2c = I2C(0, I2C.MASTER, baudrate=100000)
>>> devices = i2c.scan()
>>> for device in devices:
...     print("Address: {}".format(hex(device)))
Address: 0x77
```

Here we see the address is `0x77`. If you look in the `bme280.py` file, you will see the default is `0x76`. Just change that to match your sensor and it will work correctly.

Weather Class

We learned about classes in Chapter 4. There, we learned that we can package our code in such a way as to make a set of operations on data easier to use. We will do that here. We will wrap our code around the BME280 sensor and Adafruit IO to make an easy-to-use class or library for reading weather data and publishing it to the cloud.

However, we aren't going to just throw code that we would normally use in functions into a class willy-nilly. We will design the class to be reusable and easily configured. The best way to do that is to make the constructor – the function called when the class is instantiated – accept parameters.

For example, we will pass in the credentials for our Adafruit IO connection (device id, user name, AIO key) as well as the frequency for how often we want to publish data (recall, this is also called a sampling rate). By passing the information into the class via the constructor, we can make that data available to all the functions in the class and thus we can eliminate the use of global variables!²

Essentially, we will be packaging all the heavy lifting code for initializing the BME280 library, the MQTT library, reading data from the sensor and publishing the data, and subscribing to the Sensors feed.

If you want to follow along, we will name the class `WeatherNode` since it encapsulates the sensor and publishes data. We will name the code module `weathernode.py`. Let's consider the design for the class.

Design

Let's talk a moment about the design of the class. This is a very important step that you should always consider. Having a good design (or at least a plan) for your class in the form of what features you want to implement and making functions to implement them can make a big difference in quality of your code.

For this class, we will create a function that we can call from our main code. This function will be responsible for implementing a loop to read data from the sensors, publish the data, and monitor any messages from the Sensors feed. If a message appears for the Sensors feed, the function will also act on it. Recall, this means turning the publish data on and off based on the switch position message in the feed.

Thus, we will need one function to initiate the read loop, another to be used for reading the data, and a function we can use as a callback for responding to messages in the feed. The following sections describe each of these areas and the functions that perform the operations. Let's begin with a look at the imports section.

²A pet peeve of many computer science and programming teachers. Perfectly legal to use but considered bad form by many.

Imports

The imports section requires several libraries. We need the I2C and MQTT libraries, the BME280 driver, and the time library for delays. The following shows the imports needed for the class.

```
from machine import I2C
from mqtt import MQTTClient
import bme280
import utime
```

Constructor

The constructor needs to accept the Adafruit IO identification tag or device id, user name, AIO key, and frequency (sample rate). The device id can be any string you want to use but it is suggested you use a short name for the device. We will also use a parameter for the port of the MQTT server in case you want to change it, but we will use a default value of 1883, which is the port for Adafruit IO. Listing 11-1 shows the code for the constructor. Constructor functions are always named `__init__()`.

Listing 11-1. Constructor (WeatherNode class)

```
# Constructor
def __init__(self, io_id, io_user, io_key, frequency, port=1883):
    # Turn sensors on/off
    self.sensor_on = False

    # Save variables passed for use in other methods
    self.io_id = io_id
    self.io_user = io_user
    self.io_key = io_key
    self.update_frequency = frequency
    self.port = port

    # Now, setup the sensor
    i2c = I2C(0, I2C.MASTER, baudrate=100000)
    self.sensor = bme280.BME280(i2c=i2c)
    utime.sleep_ms(100)
    print("Weather MQTT client is ready.")
```

Notice this function simply initializes several class variables (designated with `self`.) for storing the connection values and frequency. The constructor also initializes the I2C interface and the BME280 sensor class instance.

Read Data Function

The read data function, which we will name `read_data()`, takes as a parameter a pointer to the BME280 sensor object instance, reads the data from the sensor, and returns it in a tuple. The following shows the `read_data()` function.

```
# Reads the sensor. Returns a tuple of (temperature, humidity, pressure)
def read_data(self):
    utime.sleep_ms(50)
    temperature = self.sensor.read_temperature() / 100.00
    humidity = self.sensor.read_humidity() / 1024.00
    pressure = self.sensor.read_pressure() / 256.00
    return (temperature, humidity, pressure)
```

Notice we perform some scale manipulation on the values read. These calculations were taken from the website that documents the BME280 driver (https://bitbucket.org/oscarBravo/wipy_bme280).

Message Callback Function

The next function we need is a function we can use as a callback for the MQTT messages from the feeds. We will name the function `message_function()`. In this case, we need only the topic and message parameters. The following shows the code for the `message_callback()` function.

```
# A simple callback to print the message from the server
def message_callback(self, topic, msg):
    print("[{0}]: {1}".format(topic, msg))
    self.sensor_on = (msg == b'ON')
```

Notice we print the messages received and if a message to turn the sensors on appears, we set the Boolean class variable to True. We will see how we use this feature in the run function.

Run Function

The last function we need is the one that is going to do all the work for collecting data from the sensor and publishing it on our feeds on Adafruit IO. This is where the core of using the MQTT library occurs so we will go through this bit a little slower.

The first thing we do is get an instance of the `MQTTClient` class. We pass in the device id, the host name of the server (in this case, `io.adafruit.com`), the user id, AIO key, and port.

Once we have an instance of the client, we can set the callback for getting any messages from our subscribed feeds associated with our device id, we use the `set_callback()` function passing in the name of our method to handle the messages. Recall, this is `message_callback()`.

Next, we subscribe to the sensors feed named `<user id>/feeds/sensors` using the `client.subscribe()` function as shown below.

```
client.subscribe(topic="{0}/feeds/sensors".format(self.io_user))
```

We specify the feed using the `topic` parameter. Feed names are always formed using the user id and the feed name. You can think of this like a path. Be sure to check the feed names carefully to ensure you've got the right one. You can always check your Adafruit IO page to double-check.

That completes the setup code for using MQTT. Easy, wasn't it?

Now we write the while loop to read data from the sensors. To make this easy, we will loop indefinitely (but you may want to change that if you plan to make this project more than an experiment). Inside the loop, we will check to see if it is OK to read data from the sensors. Recall, we are using a toggle button on the dashboard to control our board. If the toggle button is "ON", we read, else we do not read data.

If we are reading data (the toggle button is ON), we read the data using our `read_data()` function, then publish each data element to the appropriate feed using the `client.publish()` function. For example, to send data to the temperature feed, we use specify the user id and feed using the `topic` parameter and send the data via the `msg` parameter as shown below.

```
client.publish(topic="{0}/feeds/temperature".format(self.io_user),
msg=str(data[0]))
```

At the end of the loop, we do several things. We sleep for the number of seconds specified in the update frequency parameter. This is how we can avoid sending too much data too frequently and match our data collection to what is a realistic interval. Next, we check for messages using the `client.check_msg()` function. This function simply checks to see if any new messages have arrived. If there are, since we specified a callback function, the callback function will be called for each new message. See, no polling needed! Finally, we merely sleep for one second to allow time for the callback function to fire.

Listing 11-2 shows the completed run function for the `WeatherNode` class.

Listing 11-2. Run Function (`WeatherNode`)

```
def run(self):
    # Now we setup our MQTT client
    client = MQTTClient(self.io_id, "io.adafruit.com", user=self.io_user,
                        password=self.io_key, port=self.port)
    client.set_callback(self.message_callback)
    client.connect()
    client.subscribe(topic="{0}/feeds/sensors".format(self.io_user))

    while True:
        if self.sensor_on:
            data = self.read_data()
            print(" >", data)
```

```

client.publish(topic="{0}/feeds/temperature".format(self.io_user),
               msg=str(data[0]))
client.publish(topic="{0}/feeds/humidity".format(self.io_user),
               msg=str(data[1]))
client.publish(topic="{0}/feeds/pressure".format(self.io_user),
               msg=str(data[2]))
utime.sleep(self.update_frequency)
client.check_msg()
utime.sleep(1)    # Check messages only once per second

```

Now let's look at the completed code for the class.

Completed Code

Listing 11-3 shows the completed code for the WeatherNode class code module (weather_node.py).

Listing 11-3. WeatherNode Class (weather_node.py)

```

# MicroPython for the IOT - Chapter 11
#
# Project 4: MicroPython Weather Node - BME280 MQTT Client class
#
# Note: this only runs on the WiPy.
#
# Imports for the project
from machine import I2C
from mqtt import MQTTClient
import bme280
import utime

class WeatherNode:
    """Sensor node using a BME280 sensor to send temperature, humidity, and
    barometric pressure to io.adafruit.com MQTT broker."""

    # Constructor
    def __init__(self, io_id, io_user, io_key, frequency, port=1883):
        # Turn sensors on/off
        self.sensor_on = False

        # Save variables passed for use in other methods
        self.io_id = io_id
        self.io_user = io_user
        self.io_key = io_key
        self.update_frequency = frequency
        self.port = port

```

```

    # Now, setup the sensor
    i2c = I2C(0, I2C.MASTER, baudrate=100000)
    self.sensor = bme280.BME280(i2c=i2c)
    utime.sleep_ms(100)
    print("Weather MQTT client is ready.")

# Reads the sensor. Returns a tuple of (temperature, humidity, pressure)
def read_data(self):
    utime.sleep_ms(50)
    temperature = self.sensor.read_temperature() / 100.00
    humidity = self.sensor.read_humidity() / 1024.00
    pressure = self.sensor.read_pressure() / 256.00
    return (temperature, humidity, pressure)

# A simple callback to print the message from the server
def message_callback(self, topic, msg):
    print("[{0}]: {1}".format(topic, msg))
    self.sensor_on = (msg == b'ON')

def run(self):
    # Now we setup our MQTT client
    client = MQTTClient(self.io_id, "io.adafruit.com", user=self.io_user,
                        password=self.io_key, port=self.port)
    client.set_callback(self.message_callback)
    client.connect()
    client.subscribe(topic="{0}/feeds/sensors".format(self.io_user))

    while True:
        if self.sensor_on:
            data = self.read_data()
            print(">", data)
            client.publish(topic="{0}/feeds/temperature".format(self.io_user),
                           msg=str(data[0]))
            client.publish(topic="{0}/feeds/humidity".format(self.io_user),
                           msg=str(data[1]))
            client.publish(topic="{0}/feeds/pressure".format(self.io_user),
                           msg=str(data[2]))
            utime.sleep(self.update_frequency)
        client.check_msg()
        utime.sleep(1) # Check messages only once per second

```

Now, let's look at the main code.

Main Code

The main code file is named `weather.py`. As you will see, it is very short and simple. This is because all the work is being done in our class module! The following sections briefly describe the major parts of the main code starting with the imports.

Imports

Since all the work is being done in the `WeatherNode` class, we only need to import that code module (`weather_node`) and those libraries we need for connecting our board to our WiFi network as shown below.

```
from network import WLAN
from weather_node import WeatherNode
import machine
```

Global Definitions

The `WeatherNode` class requires the connection data for our Adafruit IO account. To make it easier to change, we can use some definitions in the main code to contain these variables. The following shows one way to do this. Here, we see definitions for the device id, user id, AIO key, and frequency (for updating the data). Be sure to change these to match your Adafruit IO credentials and set the frequency accordingly.

```
# Define out user id and key
_IO_ID = "YOUR_DEVICE_ID"
_IO_USERNAME = "YOUR_USER_ID"
_IO_KEY = "YOUR_AIO_KEY"
_FREQUENCY = 5 # seconds
```

You may want to keep the frequency at the example 5 seconds so you don't have to wait a long time to see results, but be sure to change this once you deploy the project for any longer-term runs.

Connect Function

The `connect()` function is the same function we've seen in previous chapters. Listing 11-4 shows the code for completeness. As you will see, you must provide your SSID and SSID password.

Listing 11-4. Connection Function (`weather.py`)

```
# Setup the board to connect to our network.
def connect():
    wlan = WLAN(mode=WLAN.STA)
    nets = wlan.scan()
    for net in nets:
        if net.ssid == 'YOUR_SSID_HERE':
            print('Network found!')
            wlan.connect(net.ssid, auth=(net.sec, 'YOUR_WIFI_PASSWORD'),
                timeout=5000)
            while not wlan.isconnected():
                machine.idle() # save power while waiting
```

```

        print('WLAN connection succeeded!')
        print("My IP address is: {}".format(wlan.ifconfig()[0]))
        return True
    return False

```

Now, let's look at the run function.

Run Function

The `run()` function is very simple. All we need to do is call the `connect()` function to connect our WiPy to our WiFi network, then instantiate the `WeatherNode` class, and then call the `run()` function for the class as shown below.

```

connect()
# Run the weather MQTT client
weather_mqtt_client = WeatherNode(_IO_ID, _IO_USERNAME, _IO_KEY, _FREQUENCY)
weather_mqtt_client.run()

```

That's it! Can you see how much easier it makes our code if we use classes to contain the core of our project? Hopefully, you will see the benefits and start constructing your own projects in a similar manner. Now, let's look at the completed code for the `weather.py` code module.

Completed Code

Listing 11-5 shows the completed code for the `Weather` main code module (`weather.py`). Notice you must provide the device id, your AIO user name, AIO key, frequency of updates, SSID name, and SSID password before running this code.

Listing 11-5. Main Code (`weather.py`)

```

# MicroPython for the IOT - Chapter 11
#
# Project 4: MicroPython Weather Node
#
# Required Components:
# - WiPy
# - (1) BME280 Weather Sensor
#
# Note: this only runs on the WiPy.
#
# Imports for the project
from network import WLAN
from weather_node import WeatherNode
import machine

```

```

# Define out user id and key
_IO_ID = "YOUR_DEVICE_ID"
_IO_USERNAME = "YOUR_USER_ID"
_IO_KEY = "YOUR_AIO_KEY"
_FREQUENCY = 5 # seconds

# Setup the board to connect to our network.
def connect():
    wlan = WLAN(mode=WLAN.STA)
    nets = wlan.scan()
    for net in nets:
        if net.ssid == 'YOUR_SSID':
            print('Network found!')
            wlan.connect(net.ssid, auth=(net.sec, 'YOUR_SSID_PASSWORD'),
            timeout=5000)
            while not wlan.isconnected():
                machine.idle() # save power while waiting
                print('WLAN connection succeeded!')
                print("My IP address is: {}".format(wlan.ifconfig()[0]))
            return True
    return False

def run():
    # Setup our Internet connection
    connect()

    # Run the weather MQTT client
    weather_mqtt_client = WeatherNode(_IO_ID, _IO_USERNAME, _IO_KEY, _
    FREQUENCY)
    weather_mqtt_client.run()

```

Now, let's run this project!

Execute!

Now is the fun part! We've got the code all set up to read weather data from the sensor and publish the data in Adafruit IO. You can now copy the two files (`weather.py` and `weather_node.py`) to your WiPy and start the code as follows. Remember, use a USB connection to your WiPy because once the WiFi connection is made, a TCP or WebREPL connection will disconnect.

```

>>> import weather
>>> weather.run()

```

You should see the print messages for connecting to your network and after a few seconds, data read from the sensors. Since we also print out the messages from the feed we've subscribed to, we will see those as well. If you get errors from the MQTT

connection, be sure to double-check your Adafruit IO credentials, make any changes, then reboot your board and try again.

Note You must provide the device id, your AIO user name, AIO key, frequency of updates, SSID name, and SSID password in the code. See those lines marked in bold.

There is one other thing you must do before you start seeing any data in your dashboard. Recall we wrote the class to only publish data when the toggle button on the dashboard is set to “ON”. The mechanism for doing this is reading the messages in the sensors feed looking for a change of status for the toggle button. When you first connect, there may be no such messages in that feed. Thus, you must open your dashboard and turn the toggle button on. If it is already on, turn it off and back on again. You should then see the message appear in the debug messages in your REPL console. Figure 11-17 shows the toggle button in the correct position.

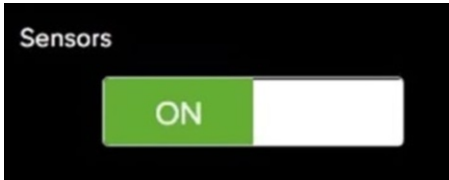


Figure 11-17. Turn Sensors ON (toggle button)

Once you turn on the toggle button, you should start seeing debug messages where data is being published. Listing 11-6 shows an example of running the project on a WiPy. Notice the messages appearing for the sensors feed.

Listing 11-6. Running the Weather Project (WiPy)

```
>>> import weather
>>> weather.run()
Network found!
WLAN connection succeeded!
My IP address is: 192.168.42.128
Weather MQTT client is ready.
[b'my_user_id/feeds/sensors']: b'OFF'
[b'my_user_id/feeds/sensors']: b'ON'
> ['23.09', '41.92773', '101964.1']
> ['23.09', '42.0166', '101972.2']
[b'my_user_id/feeds/sensors']: b'OFF'
[b'my_user_id/feeds/sensors']: b'ON'
> ['23.08', '41.93848', '101974.9']
[b'my_user_id/feeds/sensors']: b'OFF'
```

```
[b'my_user_id/feeds/sensors']: b'ON'
> ['23.07', '41.90332', '101974.9']
> ['23.07', '41.92578', '101980.3']
> ['21.07', '41.90332', '101974.9']
> ['20.97', '41.92578', '101980.3']
> ['20.07', '40.13824', '101974.9']
> ['19.96', '39.32481', '101972.1']
> ['20.06', '40.62481', '101974.8']
> ['21.05', '40.95801', '101974.8']
> ['22.05', '41.93555', '101982.9']
> ['23.05', '41.91308', '101977.5']
...

```

OK, once your project is running and publishing data, you can start to see it in your dashboard. Go ahead and log back into Adafruit IO and select your dashboard. You should now see data in all the line graphs and the stream block. Figure 11-18 shows an example of what your dashboard would look like if you sampled data at the 5 second default frequency.



Figure 11-18. Data Visualization via Adafruit IO Dashboard (Weather Project)

Notice the pressure and humidity lines don't change much. This is expected since this example was run indoors during a very short time. However, you may notice a rise and dip or two in the data for temperature. This was due to artificial changes to the temperature. That is, I moved a heat source close to the sensor to make the area warmer and thus record higher values. Similarly, I used a cool air source to cool the area causing a dip in the values. When I did so, it caused a slight change in the humidity values. Can you think of why that happened?³

You can try this out yourself, but be careful! Don't touch the sensor lest you damage it and don't use any open flame or other heat sources that could cause burns (or worse).

■ **Caution** Resist the temptation to touch the sensor. You could damage the sensor.

You may notice something odd if you exit your dashboard and reopen it. The Adafruit IO dashboard currently only shows the messages (data) in the feeds from the moment the dashboard is opened. Most likely, you ran your project for some time, stopped it, then went back to your dashboard later to proudly show your friend the fruits of your handiwork only to discover there's no data in your dashboard. While this may be a feature of the dashboard, your feeds and the history therein is still there.

This is because when you create a new feed, the default is to store the history (data). You can change that so that only the last value is stored, but most IOT projects will want the data to be around for some time. Thus, using the dashboard to see historical data doesn't work. However, you can still view the data in your feeds and see it graphically by opening the feed directly. Simply click on your feed and you will see the data like that shown in Figure 11-19.

³I used a can of air, which was much drier than the ambient environment.



Figure 11-19. Displaying Historical Data in Feeds

Notice the *Actions* drop-down box near the bottom. Notice also there are checkboxes you can tick to select messages and a checkbox at the top to select all messages. Once you select messages, you can use the *Actions* drop-down list to remove the selected data or download the selected data. You can even add data (one message at a time). This makes it easy to get to your data and modify it. The ability to download the data means you can use it in other projects or tools for analyzing data. Figure 11-20 shows the possible actions available for modifying the data in a feed.

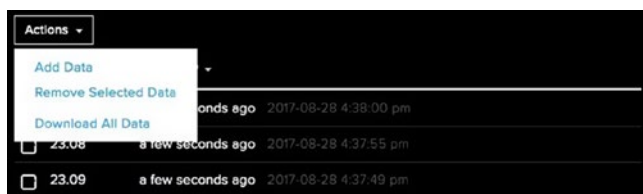


Figure 11-20. Modifying Data in Feeds

If you like this project, you can consider making it run over a longer time span, reducing the sampling frequency to every few hours. You will still see the data but perhaps if the temperature in your environment changes, you'll see the actual variation in data rather than simulated changes.

At this point, you've completed another real MicroPython IOT project. In this case, we saw an IOT project that collects and displays data in the cloud. How cool is that?

Taking it Further

This project, like the last one, shows excellent prospects for reusing the techniques in other projects. This is especially true now that you know how to use MQTT. If you liked seeing your sensor data over the Internet, you should consider taking time to explore some embellishments. Here are a few you may want to consider. Some are easy and some may be a challenge or require more research.

- Add more sensors to expand your project to more weather observations.
- Modify the dashboard to display the data using different blocks.
- Add more sensor nodes to gather data observations from other places, creating new feeds for each.
- Set up your own MQTT broker and attach your sensor nodes.
Hint: change the host name in the `MQTTClient()` call in `weather_node.py`.
- Use MQTT to build a home automation project allowing you to turn lights on and off via the cloud.
- Add another button that you can use to shut down the project.
Hint: you can add a check in the `run()` function to terminate the loop when the switch is turned off.

Summary

Taking a small microcontroller like the WiPy, adding sensors, and writing relatively short MicroPython code to connect to the Internet and produce data that you can view from anywhere in the world demonstrates the concepts of the Internet of Things in a working project. We have now learned how we can leverage our new knowledge of MicroPython and the small MicroPython boards to achieve true IOT solutions.

In this chapter, we demonstrated this by building a weather sensor project that connected our board to the Adafruit IO cloud service and used that service to store and visualize the data. While in retrospect this project seems easier than the previous projects, we gained specialized knowledge in those projects that made this project easier to implement. That is, without the basics of how to build IOT projects, jumping into the deep end of IOT projects would be folly.

Fortunately, with the help of the nice folks at Adafruit, we can now build IOT projects to send data to the cloud and share it with the world. Not only that, but we can also build solutions that allow us to control our projects from the Internet. Everything done quickly with very little effort and no need to learn a complicated API. How cool is that?

With the experience gained in this book, you are now ready to build sophisticated IOT solutions from simple projects that you run for fun to full cloud-based solutions. Now it's time for you to engage your own imagination and put the tools and techniques you learned in this book toward building your own MicroPython IOT solutions.

In the next chapter, we will see how you can take what you've learned in this book further as you plan more MicroPython projects.

I hope you enjoyed the ride, and that reading about and working on the projects in this book were as much fun for you as I had writing them.

CHAPTER 12



Where to Go from Here

Now that you have had a thorough introduction to MicroPython, MicroPython hardware, the Internet of Things, and the types of projects you can create and tutorials as well as examples, it is time to consider what you can do beyond the pages of this book.

In this chapter, we will explore what you can do to continue your craft of building IOT solutions. Most people will want to simply continue to develop projects for themselves either for fun or to solve problems around the home or office. However, some will want to take their skills to the next level.

Whichever the case, there are a few things that you should consider. In the following sections, we will look at sources for more example projects, how to join the community of IOT enthusiasts through social media and other Internet resources, and finally how to become a contributing member of the growing throng of Makers.

More Projects to Explore

If you want to work on more MicroPython IOT projects, you will be happy to learn that there are many examples that you can explore. Most of the examples are either in the various documentation sites or are contributions from the community, ranging from a high-level overview to detailed instructions on how to complete the project. Sadly, most of the examples are presented with little or no instruction. However, now that you have had detailed instructions¹ on working with MicroPython IOT projects and the various MicroPython boards, you should be able to complete the examples with little or no documentation (but documentation always helps).

There are several repositories for MicroPython IOT example projects. Most are either in forums dedicated to MicroPython or the MicroPython board (for example, the Pyboard), but there is also a cool site named Hackster.io, which is a general hardware community forum (www.hackster.io). We will see how to navigate that site in a moment. First, let's look at a couple of resources for MicroPython sample projects.

¹This is one of my major motivations for writing this book.

MicroPython Project Samples

There are several websites that host MicroPython examples. There are three basic types of websites: forums that have categories where people can post their projects, documentation sites that have example projects, and repositories where people can upload their projects. Let's look at each of these.

■ **Tip** The best way to find MicroPython examples is to google for “MicroPython Example” or “MicroPython Sample”. You'll find lots of hits including those resources listed in this chapter.

Forums

The first type, forums, offer a list of categories ranging from announcements, questions and answers, frequently asked questions, technical support, and more. Fortunately, several also have a category for projects. Sometimes the category includes many entries (topics) with most being questions about how to implement a certain project. This can sometimes make looking for a project idea tedious.

The MicroPython forum has such a category (<https://forum.micropython.org/viewforum.php?f=5>). And while there are over 100 topics and most are questions, there are a few gems in there you can explore. Pycom also has a forum for the WiPy with a category for projects (<https://forum.pycom.io/category/28/projects>).

Using a forum to find example projects or samples can take a little work because they are intended to be used by a community and thus contain a lot of questions and commentary, but I feel they are still one of the best resources. However, keep in mind they are intended for a specific board (Pyboard or WiPy). The best way to use them is to navigate to the forum and either scroll through the topics or search for a key phrase. Figure 12-1 shows an excerpt from the Pycom MicroPython project category.

The screenshot shows the Pycom Forum interface. At the top left is the Pycom logo with the tagline 'GO INVENT'. The forum title is 'Pycom Forum'. On the right, there are navigation options: 'A', '7 out of 19', a search icon, 'Q', 'Register', and 'Login'.

Topic Title	Posts	Views	Recent Reply
TTN + MQTT + Node-RED + MySQL = local backup of your LoRaWAN data <small>TTN · LOPY · MQTT · NDBERED · MYSQL · 3 months ago · PiAir</small>	4	507	6 days ago Have a problem with setting up a connection with the new TTN backend with Node-Red. I
LoPy TTN mapper for ttnmapper.org <small>NMEAD183 · TTN · MAPPER · LOPY · 20 days ago · affoltep</small>	2	136	15 days ago Hey @fatcheo Thanks for sharing that! This is really awesome!
TTN LoraWan GW not 100% stable <small>LOPY · LORAWAN · TTN · GATEWAY · 25 days ago · gbalnet</small>	6	297	20 days ago @gmalcalho OK, I will try this in order to get more information when a crash is
LoPy LoraWAN gateway with an ST LoraWAN device <small>LOPY · TTN · GATEWAY · LORAWAN · JOIN · 24 days ago · bergast</small>	5	262	22 days ago @bergast awesome! I updated my project with your solution and it is working perfectly.
Wipy motion and ambient conditions sensor node <small>WIPY · 28 days ago · robmarkcole</small>	8	263	27 days ago @robmarkcole said in Wipy motion and ambient conditions sensor node:

Figure 12-1. Excerpt from the Pycom MicroPython Project Category (courtesy of pycom.io)

Documentation

The MicroPython documentation is also a good resource for finding example code (but not entire projects). They're typically better than the forums because they are much better organized and sometimes better written. However, like the forums, the MicroPython documentation sites, especially the samples, are hardware specific. That is, there is one set of documentation for Pyboard, another for WiPy, and other boards.

However, there are some example projects buried in the documentation. Unfortunately, some are not as well documented or may be partially documented. Regardless, the code in these samples is generally far superior to what you may find in the forums (generally).

The best way to use the documentation sides is to simply navigate to them and browse through the table of contents (that alone makes them better than forums). For example, Figure 12-2 shows an example project from the Pycom MicroPython documentation for the PySense shield.

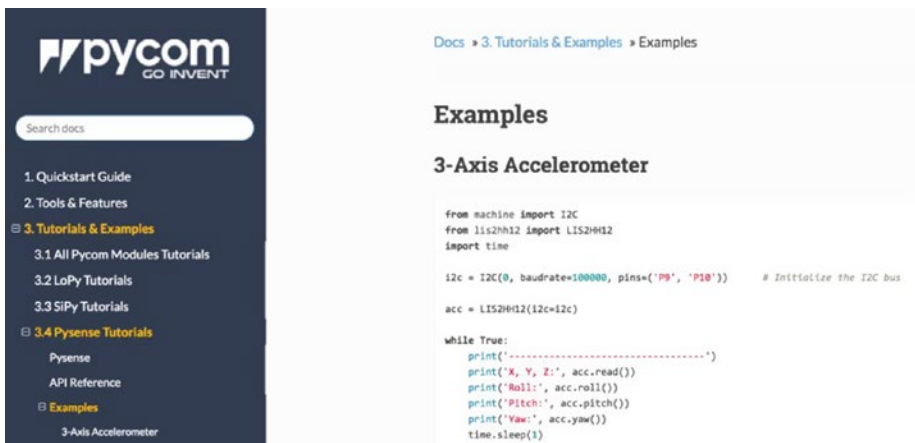


Figure 12-2. PySense Sample Project (courtesy of pycom.io)

Repositories

The best websites for MicroPython projects are repositories. These are typically hosted in a source control service such as GitHub. What makes these sites most useful is that you can navigate directly to the source files and see the code in your browser, skipping the documentation or demonstration page. This makes it nice if you just want to see how to implement some feature rather than walk through a long page of text. Of course, the best way to use the samples is to download the entire set of samples. Simply visit the GitHub main site and download the sample projects. How cool is that?

The main MicroPython site (<https://github.com/micropython/micropython>) is one of the best examples of MicroPython sample project repositories. This repository has a lot of samples and while some are geared to the Pyboard, you can use them as templates

for how to write code for a wide variety of projects. Figure 12-3 shows an excerpt of the repository main page.

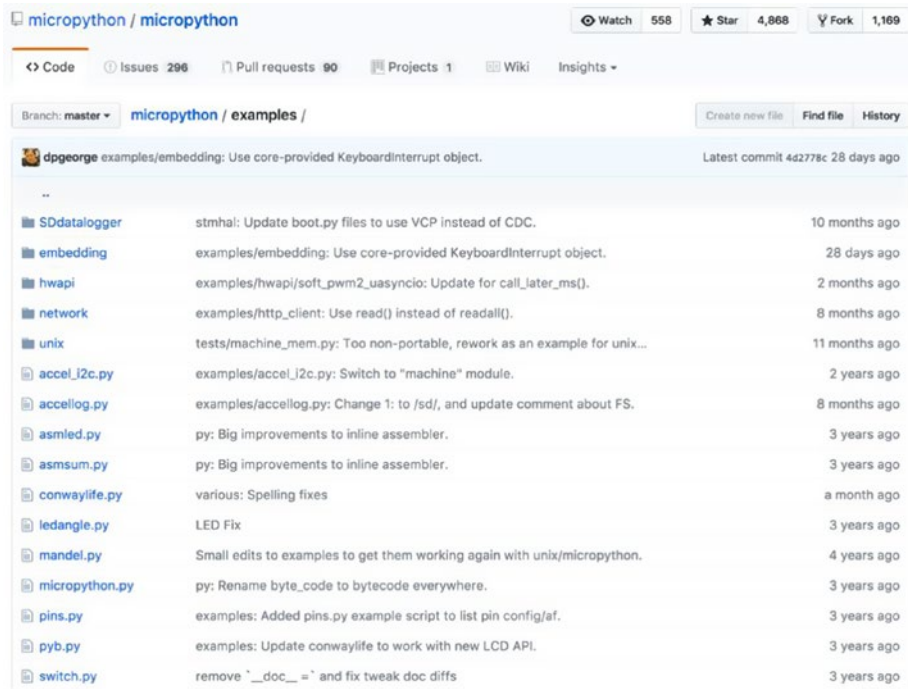


Figure 12-3. MicroPython GitHub Repository Sample Projects

Most (if not all — I haven't checked all) of the samples are licensed under an open source license such as the MIT license, which makes it very convenient for everyone since the MIT license permits you to use and even publish the code. (See <https://opensource.org/licenses/MIT> for a sample of the MIT license.) This is great because I have many times wanted to use a sample or demonstration of a project, only to discover the license doesn't permit it.

Community Project Sites: Hackster.io

The Hackster site is a community dedicated to learning hardware. You can find all manner of hardware sample projects, including many of those for MicroPython, Python, Raspberry Pi, Arduino, and more! There is a small but growing list of MicroPython projects. When you visit the parent site (www.hackster.io), you can search for projects by typing "MicroPython" in the search box. Once you enter the search criteria, the search results page, you see all the projects you can explore. Each is marked with a relative level of difficulty, ranging from easy to advanced. You also see a count of the number of

times the sample project was viewed and the number of thumbs-up ratings the project received from others in the community (you must join Hackster.io to rate a project). Best of all, there is a comments section that you can use to encourage the designer or ask the designer for help with the project.

■ **Tip** Use the golden rule when posting comments or questions in online forums. Resist the temptation to post opinions, fan flames of dissent or ridicule, and stick only to the facts.

What I like most about the Hackster site is that the samples are generally well documented and often include several photos of the project. Due to the unique structure of the site, the samples are organized in parts that make it easy to follow. For example, the web page for “Character LCD over I2C” (www.hackster.io/dzerycz/character-lcd-over-i2c-ba8ee9) contains an overview section that presents a short description of the project, associated tags, difficulty rating, publication date, and even the license. This makes reviewing the project very easy.

For example, there is an excellent example of an intermediate-level project that is not only well documented but also well written. It is the “MicroPython Leak Detector with Adafruit and Home Assistant” by Robin Cole. Figure 12-4 shows an excerpt from the project overview. If you want to see another excellent project, visit <https://www.hackster.io/robin-cole/micropython-leak-detector-with-adafruit-and-home-assistant-a2fa9e>.

The image shows a screenshot of a project page on Hackster.io. The title is "MicroPython Leak Detector with Adafruit and Home Assistant" by Robin Cole. The page includes a photo of the hardware setup, which consists of a Raspberry Pi Zero, a MicroPython sensor node, and a breadboard with various components. The page also features a description of the project, project information, and user interaction buttons.

MicroPython Leak Detector with Adafruit and Home Assistant
 Made by Robin Cole - Published in Adafruit, Home Assistant, Pycom, and Raspberry Pi

pycom®
GO INVENT

ABOUT THIS PROJECT
 Detect water leaks using a MicroPython sensor node and receive mobile notifications via Home Assistant.
 home automation water level

PROJECT INFO

Type	Full instructions provided
Difficulty	Intermediate
Estimated time	1 hour
Published	April 17, 2017
License	GPL3+

1,039 views 6 likes

Respect project I made one

Bookmark Share Give feedback

Figure 12-4. Hackster.io sample project

If you scroll down from the overview, you find sections that demonstrate how to connect the hardware (like how I introduce the projects in this book), a short walkthrough of the code, and descriptions and demonstrations on how to use the project.

Some samples include short videos to demonstrate or explain the project. At the end of the page, you find the comments section, which you can use to read what others have said about the project, as well as the questions that others have had regarding the project. If you get stuck on a sample, be sure to read all the comments — there is a good chance that someone has already asked the question or solved the problem.

WHAT ABOUT HACKADAY.IO?

Another site that is like Hackster.io that is gaining a following is Hackaday.io. It is very like the Hackster.io site in that you can search for MicroPython projects. However, I don't like it as much as Hackster.io, but don't let that stop you from exploring it — try it!

Knowledge Repositories: learn.adafruit.io

Another excellent source of information about MicroPython is learn.adafruit.com. This site contains articles, blogs, and tutorials on a wide range of topics. Much like the user forums, content is added to this site frequently. So, you should visit the site periodically and search for your topic. For example, to find the MicroPython content, simply search for MicroPython. Or, you can use this link: <https://learn.adafruit.com/search?q=micropython&>.

I've often used this site to find ideas for projects. Even if the article, blog, or tutorial isn't exactly what you want or it may not match your hardware, it is often worth reading for tips. For example, if you're planning to build a project using a Charlieplex LED (<https://www.adafruit.com/?q=charlie>), you can find a tutorial at <https://learn.adafruit.com/micropython-hardware-charlieplex-led-matrix/software?view=all>. While this tutorial features the Feather and ESP8266 board, you can still learn quite a bit about using the Charlieplex LED including a driver (that you can modify), hints, and insights in how to make it work with other MicroPython boards.

Figure 12-5 shows an excerpt from learn.adafruit.com showing some of the interesting content for MicroPython. Be sure to check the site for recent additions.

The screenshot shows the Adafruit website's search results for 'micropython'. The top navigation bar includes 'SHOP', 'BLOG', 'LEARN', 'FORUMS', and 'VIDEOS'. A search bar on the right contains the text 'micropython'. Below the search bar, the results are organized into three columns. The first column features the article 'MicroPython Basics: How to Load MicroPython on a Board' by Tony DiCola, with a thumbnail image of various boards. The second column features 'MicroPython Basics: What is MicroPython?' by Tony DiCola, with a thumbnail image of a micro:bit. The third column features 'MicroPython Hardware: SSD1306 OLED Display' by Tony DiCola, with a thumbnail image of an SSD1306 OLED display. To the right of the search results is a 'CATEGORIES' sidebar with a list of categories such as 'Sensors', 'Components', 'Hacks', etc., each with a checkbox.

adafruit SHOP BLOG LEARN FORUMS VIDEOS micropython

Search results for micropython

PRODUCTS | LEARN | BLOG | FORUMS

MicroPython Basics: How to Load MicroPython on a Board
Learn how to load MicroPython firmware on a development board.
by Tony DiCola

MicroPython Basics: What is MicroPython?
What is MicroPython, why would you use it?
by Tony DiCola

MicroPython Hardware: SSD1306 OLED Display
How to use a SSD1306 OLED display with MicroPython boards.
by Tony DiCola

MicroPython Basics: Hardware: Digital I/O

MicroPython for SAMD21
How to use MicroPython with boards like the Feather M0 & Arduino Zero!
by Tony DiCola

CATEGORIES

- Sensors
- Components
- Hacks
- Microcomputers
- Adafruit Products
- Maker Business
- Projects
- LEDs
- Raspberry Pi
- BrainCrafts
- LCDs & Displays
- EL Wire/Tape/Panel
- Tools
- Microcontrollers
- Learn Arduino
- Customer Projects
- BeagleBone
- Circuit Playground
- 3D Printing
- Trinket
- Robotics
- Collin's Lab
- Community Support
- Wearables
- Adafruit IO
- Arduino
- Feather
- Wireless

Figure 12-5. *MicroPython Content on learn.adafruit.com*

Now that you have seen a couple of resources for more sample projects, let's discuss how you can join the community and contribute to the growing repository of all things MicroPython for the IOT.

Join the Community

Once you have mastered the sample projects in this book as well as a few from other resources, it is time to take your hobby a bit further by joining the community of IOT developers and enthusiasts.

In this section, I discuss some of the reasons you may want to share your knowledge, etiquette for sharing and contributing, and a few example communities you may want to join or at least monitor. As you will see, not these are strictly devoted to MicroPython, but can be an excellent source for ideas. Let's begin with why we would want to share.

Why Contribute?

As more and more free thinkers drive hobbies like MicroPython and the IOT, the more prevalent the concept of sharing becomes. This is no accident. Many of the founders and pioneers of the Python, MicroPython, and IOT are open hardware and open source advocates. While this applies not only to hardware and software, it also applies to other intellectual products such as the source code and documentation for published projects.

Many feel their code should be free for anyone to use and modify with reciprocal expectations. For example, if you modify someone else's design or code, you should share not only the improved design but also credit the originator. In some cases, this is as simple as listing the original author, but other times it may mean giving the original author your modifications. So long as you follow the guidelines of the license, all is fair and well in sharing.

However, depending on how the sample code was written (licensed), there may be some limitation to what can be shared. For example, it may not be possible to share code from a proprietary library. While you may be the creator of the code that uses the library, you do not own the library and cannot share it. You most likely can share your code with others, but publication may be restricted.

Sharing your projects also means placing them someplace where others can find them. You may want to make them freely available to anyone or you may want to limit what people can do with your project. Fortunately, there are websites that can handle either quite well.

So, why contribute your project? There are many reasons including the fact that it can be a good feeling to see one of your projects being liked, used, and made by others. Perhaps the most important reason for contributing is to help others learn what you have or, better, learn how to avoid pitfalls or problems. In this way, we all benefit by learning best practices or simply better ways to implement our ideas. Finally, your own project and experiences, when shared, will inspire others to create other projects or perhaps improve on yours.

I've had this happen with my own projects. People have taken what I have made and improved it. Since they, in turn, shared their project, I could incorporate a lot of their improvements in my projects, making them even better than I envisioned.

Which License, Where?

So, how do you know which license is in play? Every website that hosts any form of source code, documentation, examples, etc., will have a clearly marked license. It may appear on the screen at the bottom or some other discrete location or even only in a special page marked a "license" or similar label.

For example, the license on the micropython.org website is located under the heading, "Completely free, open source software" and cites the MIT license for the MicroPython core and paraphrases that license as follows.

You can freely use and adapt MicroPython for personal use, in education, and in commercial products.

Similarly, the Hackster.io website has a section for specifying the license whenever someone uploads a project. Figure 12-6 shows an example of one of the MicroPython projects on their website. As you can see, the license is clearly listed in the project information summary. In fact, you can click on the link and read the license. Fortunately, most projects on Hackster.io and other sites are open source and you should be safe to use them in your projects (but check anyway).


PROJECT INFO	
Type	 Full instructions provided
Difficulty	Intermediate
Estimated time	1 hour
Published	April 17, 2017
License	GPL3+

Figure 12-6. Example Project Information from Hackster.io

■ **Tip** Always check the license of any example you use before you publish it to ensure you are conforming to not only the originator's wishes but also the legal restrictions of the license assigned.

Now, let's focus on how to go about sharing your projects.

How We Share

You may be wondering why anyone would want to give something they have worked on for hours away for free. While it is true that the expectation is that you should share your cool projects with others, it isn't a hard and fast rule. In fact, there are some who have made their projects available for a fee as a precursor to selling the IOT solution in a commercial avenue. However, the clear majority of enthusiasts share their ideas and projects for free.

There are several communities where you can share your projects and we will see some of these in the next section. But first, there are some things you must understand about sharing objects. Believe it or not there is a set of rules – some written, some not – that you are expected to follow should you decide to embrace the community of the IOT or any similar community. The following lists some guidelines (rules) you would do well to heed when sharing your ideas, projects, and commentary with the community.

Keep Your Designs Original

Nobody likes a copycat. You didn't like it when you were 5 and you won't appreciate it when you see something you designed and shared for free being presented as 'design of the month' credited to someone else.

Thus, you must do your homework to make sure your design is unique. You don't have to purposefully alter your design so that it doesn't resemble someone else's, but you should do due diligence and at least search for similar projects. Remember, it is OK if you develop a similar project, but it generally is bad form (or perhaps a violation of the license) to simply reproduce something someone else has published.

In the rare case when it so happens your project is nearly identical to another, so long as your work is your own, there shouldn't be a problem. In fact, this happened to me once. The other designer's responses and mine were something like, "Cool project. Like minds, eh?" Once again, there is nothing wrong with that, provided you both acknowledge the resemblance and there are no licensing issues.

If the other project like yours is truly the same design but was licensed differently, you may have to negotiate with the other designer. This can happen when projects are licensed for ownership (e.g., commercial property), but it is rare given most IOT sample repositories are sites where people share their projects for free.

Let's look at another, non-source code example. What is the likelihood that a dozen different cases for a Raspberry Pi will be similar in size, have the same openings for ports, and perhaps even assemble the same way (snap together)? Very likely, yes? Does this mean there is one original and 11 copies? No, certainly not. This is not what I am talking about.

What I mean by unique is of those 12 cases, you should be able to identify some differences among them. Be that how they print (e.g., orientation on the build platform), if they are made from several parts, whether they have designed with ventilation, etc. Even if all 12 designers started at the same time, there will be some minor differences. More importantly, each is its own work. That is, no one used the design of another to pass off as their own.

In the case of a software project, the source code is most likely going to be a little different among the examples. While how much difference qualifies source code to be deemed different is something for lawyers to sort out, suffice to say if your code and another's are nearly identical, but created without knowledge of the other, it is OK to share your code provided there are no licensing conflicts.

Finally, when you share your project that is based on the work of another, you must annotate your code, documentation, and project web site giving credit to the original designer. That is, you state unequivocally that your project is a derivation of the original. It is also good form to include a link to the original design along with a list of your modifications. Once again, this assumes the license permits it.

Check the License

I have mentioned licensing under the aspect of downloading and using sample projects. Recall that most repositories will require you to specify a license for your project. This permits the repository to host your project and communicate to everyone what your intentions are regarding ownership, permissions to use, etc.

As I stated previously, you need to check the license before using any design. If you plan to modify it, you need to pay close attention to the license. The clear majority of licenses will allow you to use the design and most will allow you to modify it.

However, where some licenses differ concerns the ownership of the modifications. Some open source licenses, like GPL, permit modifications but require you to surrender those modifications to the original owner (the person or organization that created and licensed it) if you plan to distribute those changes. That is, you can modify it at will for personal use but once you distribute those changes, you must give them to the owner of the license.

I have only run into this a couple of times but in those cases the designer was prototyping designs for a commercial product. The license and indeed the text of the project made it clear she was looking for help with the design but that the design would not be made public. Watch out for this and tread lightly. Any work you do could be for the benefit of the owner and not yours to keep or profit.

■ **Tip** When in doubt about a license, contact the originator and ask them directly.

Since most sample IOT projects are licensed for sharing and free modification, you normally don't have to worry too much. However, I recommend you check the license before using any project, especially if you intend to share or publish your derivation.

Keep It Appropriate

Believe it or not, there are hobbyists and enthusiasts with impressively vivid imaginations who have come up with IOT projects that some may consider inappropriate or even obscene. No matter what your own views are, you should strive to tolerate the views of others. That doesn't mean you must compromise your own views; just be aware yours may offend and strive to minimize the offense.

More specifically, don't upload projects with inappropriate themes to sites that are viewable by everyone. It's (may be) fine to upload some project that promotes a theme, ideal, etc. (provided there are no copyright violations); just don't upload projects or commentary that are clearly offensive or intended to cause harm.

For example, if you consider the fact that IOT projects are being used in schools to teach children the technology and techniques of working with hardware and designing software, you shouldn't upload projects with themes that parents may deem as inappropriate. The most obvious of course are offensive language, adult themes, and slanderous images.

You should check the usage and user agreement for the site that hosts your chosen repository as part of the post-no-post decision. Make sure you read the section about what is and is not appropriate and adhere to that.

There is another angle to consider. You should avoid uploading sample projects that are or could potentially be illegal or unlawful. This may be difficult to discern considering the IOT community includes the entire globe. However, most sites will have language to suggest what is and is not permitted. And some have language in the agreement that gives them (the site) the right to remove things they deem inappropriate.

For example, I once saw a project for a radio frequency identification (RFID) reader that could be used to read RFIDs from a distance. This sounds harmless, but consider the number of things that use RFID such as security badges, identification, and even credit cards. Clearly, reading RFID from things you own is fine (indeed, that's what the project demonstrated), but the project could be (and most likely has been) used for evil. Fortunately, others noticed this and the project site has been removed (URL results in a 401 error).

So, before you upload a design or sample project, make sure you understand and agree to the terms of the user agreement as to what is and is not appropriate. Most times a misunderstanding is not something that will get you into trouble, but if you do it more than once, chances are someone at the site will want to speak with you or restrict your access. Which brings me back to the opening of this section – be sure to respect the views of others and especially the intended audience of the site. If you disagree, find another site.

Annotate Your Work

One of the ways I can tell if a sample IOT project is good or of high quality is how it is annotated and documented. That is, how well the designer described the project on its site. If I encounter a project that looks appealing only to discover the designer didn't bother to describe how to connect the hardware, or explain the code with more than seven words (or less), didn't provide any instructions, or worse didn't present any photos of the actual implementation, I won't use it.

Thus, you should strive to provide as full a description as possible. You don't have to write a novel, novella, or a dissertation; but you should provide enough information to describe the intended use; what problem it solves; as well as a set of instructions for how to write the source code, compile, and deploy it.

The only exception is a case where you are still working on a project or you plan to make changes before finalizing it. In this case, you should mark (annotate) the project with some verbiage about a work in progress, being experimental, etc. If your repository has a feature to mark the project as such, use that. This way, others will know your project isn't quite ready for general adoption. One reason for doing this may be to get feedback from others. I've done this myself with mixed results. Mostly people are happy to comment they like it, but don't comment or if they do, however encouraging, don't suggest any changes or improvements.

I would also suggest you provide some level of contact information so that others who have questions can contact you. Most sites make it easy for viewers to contact you through the site but you may want to provide other forms of contact (e.g., email). You may not want to provide your home address and phone number (don't do that), but an email address is a nice way to make yourself open to the community.

For example, I have seen blogs, sample projects, and even tutorials where people have posted their IRC handle, email address, and even in one case their business phone number. While I may not go quite that far, I suggest providing an email address so that you can communicate with people who like the project. Plus, it's nice to connect 1:1 with someone to discuss your work!

Be a Good Citizen

Suppose you run across a sample project that not only isn't high quality but is also (in your opinion) designed or implemented incorrectly. Should you immediately comment and crush the designer's ego with a flippant remark about how dumb their code is? No, certainly not!

What I would do (most likely) is ignore the project altogether. I mean, why make things worse by pointing out the defects? I have found the community at large (there are some exceptions) will likely do the same and not comment. Recall one of the keys to determining whether a project is well designed (good) is how many people use it. Typically, there is a counter you can check for this. If no one has liked it or even downloaded it, you can be sure it won't make it to the top of any search lists or sample project of the month.

On the other hand, if you feel compelled to comment, be sure to either contact the designer privately or be as constructive as you possibly can. The goal should be to help the designer improve his projects, not challenge their intellect (or pride).

When I do comment on projects that I find strange and perhaps flawed (and it is rare), I generally phrase my comments in the form of a question. A question normally doesn't put someone on the defensive and if worded properly should also not offend.

For example, I may ask, "Have you found the code may hang if the user presses the button more than once?" This is a nice way of asking if the designer has tested his project under the conditions you expect it to fail. This is good, constructive criticism in a most intellectual form. I am certain if you think about what you are about to say, you can find other and perhaps more elegant ways of helping people improve their projects.

Now that we've seen why we share and how to share in a responsible manner, let's discover some of the communities you may want to join or at least monitor.

Suggested Communities

There are quite a few general Python and MicroPython web sites and online communities you can visit and even join. Most online communities have repositories that you can search for examples, tips, techniques, and even complete projects that you can explore. Most also have one or more areas where members can comment, ask questions, or generally communicate with others on the forum. You typically must join to be able to post a reply or ask a question, but viewing is typically permitted by anyone.

The best way to use these resources is to visit them periodically. More specifically, you should read the articles (that interest you) and forum posts regularly. This allows you to keep up to date on current events, new techniques, and even new solutions to challenging problems. Table 12-1 presents a brief list of online resources that you should consider visiting to keep abreast of the latest news about Python, MicroPython, hardware, and IOT. I list the general topic, URL, and a brief description for each. However, you neither must join these communities nor are these are the only sites you can or should join. In some cases, you may want to simply monitor the site regularly.

Table 12-1. *Online Resources for Python, MicroPython, IOT, and Hardware*

Topics	URL	Description
MicroPython, Python, and more	hackster.io	General hardware and project site for all manner of projects. Look here for detailed explanations of projects searching for Python or MicroPython
MicroPython, Pyboard	http://docs.micropython.org/en/latest/pyboard/	Documentation for MicroPython on the Pyboard
	https://forum.micropython.org/	Forums for MicroPython and Pyboard
MicroPython, WiPy	https://docs.pycom.io/pycom_esp32/index.html	Documentation for MicroPython on the WiPy and other Pycom boards
	https://forum.pycom.io/	Forums for MicroPython and the Pycom boards
MicroPython, BBC micro:bit	https://microbit-micropython.readthedocs.io/en/latest/	Documentation for MicroPython on the BBC micro:bit
CircuitPython	https://learn.adafruit.com/search?q=circuitpython&	Adafruit's tutorials on CircuitPython
	https://blog.adafruit.com/?s=circuitpython	Adafruit's blogs on CircuitPython
Python	https://www.python.org/	General Python information
	https://www.python.org/doc/	Python documentation
	https://www.python.org/community/	Forums for Python

Notice there are web sites for Python specifically, Raspberry Pi, general hardware, and similar resources. While many do not specifically address or cover MicroPython, most have a growing repository of knowledge that is often surprisingly helpful even with MicroPython.

For example, I try to connect to the Raspberry Pi sites to keep tabs on what is going on there. I can often get ideas for IOT projects or simply ideas for features by seeing projects implemented for the Raspberry Pi and its native operating system. I find it is often the case that while the source code can be quite different, most of the hardware (connections, etc.) apply without modification, which makes sense since the hardware is not tied to the operating system running on the device (but the libraries that drive the hardware does).

Also, don't discount the power of a keyword search using your favorite online search tool. I have often found obscure gems of information that aren't posted on the more popular sites. In most cases, they are well-written blogs. I often start with a keyword search before I visit the sites listed above. If nothing else, it confirms whether what I am researching is unique or ubiquitous. You should also consider using key phrases from error messages as search terms to get help for specific errors.

There are also a few excellent periodicals that you should consider obtaining. I exclude the typical Windows, PC, and general programming periodicals not because they aren't helpful but because they are far too general for IOT-related research. Rather, the following are periodicals that I've found to be very helpful in my IOT research or any electronics or hobby project.

- *The MagPi Magazine*: a monthly magazine devoted to all things Raspberry Pi. Includes many articles on sample projects, news about the Raspberry Pi, peripherals, and general hardware reviews. (raspberrypi.org/magpi)
- *Make*: a magazine devoted to the broad realm of the Maker community presenting sample projects, tutorials, hardware, tools, drones, robots, and more! It truly is a one-stop periodical for all things hacking, tweaking, and general DIY hobbyist, enthusiast, and professional alike. (<http://makezine.com/>)

Now that we've seen several online communities (but in not a complete list as more are being added seemingly weekly) as well as a couple of periodicals you can buy, let's discuss the next step you can take once you have become a productive contributing member of IOT and IOT-related online communities – becoming a maker.

Become a Maker

The next progression in your growth from novice to enthusiast (and even professional) is to practice your craft regularly and share your knowledge with the world. One excellent outlet for this desire is to become a maker. Makers are widely regarded as experts in their areas of interest. The best part is a maker's interests can vary greatly from one to another. That is, becoming a maker isn't about learning a specific set of techniques (academic or otherwise). It is about practicing a craft.

What's a Maker?

Sadly, there is no single definition of what a maker is or should be. This is because a maker is someone with highly creative skills who desires to tinker and work on projects that can range from mechanical sculptures that spit fire to electronic gizmos to new ways to recycle materials to build things on the cheap.

Indeed, a maker is simply an artisan, craftsman, hobbyist, or enthusiast who desires to create things. Hence, Maker. Thus, there are many kinds of Makers. However, what unites them is the willingness to share their techniques and skills with others. Thus, to truly become a maker, you must participate in the community of Makers.

Share Your Ideas

You can become a maker and not contribute at all to the online communities and while that's perfectly fine, if you want to help make the community stronger, you should become more involved. The best way to do this is to join one or more of the online forums and start contributing.

That doesn't mean you must start off by posting some fantastically cool and successful MicroPython or IOT project complete with award-winning documentation and bulletproof code that computer science professors will use someday to teach young minds to mimic your brilliance. Sure, you may scoff (or even laugh) at that, but I have encountered people who are afraid of posting anything lest they be wrong or ridiculed for inaccuracies. Don't worry about that - the best contributors are those that help you grow and you grow by learning better techniques!

The best way to avoid that is to start out slow by first asking questions of your own. You can also start by throwing out a few positive comments for those ideas and projects you like. As you become more involved with the topics and techniques, your knowledge will expand to the point where you can start answering questions of others.

I encourage you to consider this level of involvement if you want to become more involved with MicroPython. I believe interacting with the community and gaining a reputation for being helpful and sharing ideas is one of the things that separates a hobbyist from an enthusiast.

Thus, if you want to become a maker, you should share your ideas with others.

Attend an Event

Another way you can become more involved with the maker community is to attend a Maker Faire (makerfaire.com/). These events are held all over the world. The events allow Makers to showcase their creations, teach others, and generally celebrate all things maker. See the Maker Faire site for events in your area near you.

If you live in or near a larger city, you may find there are local user groups for Python, Raspberry Pi, MySQL, Arduino, and even maker user groups. Try searching for events and groups in your area and find out where and when they meet. Most groups have an open-door policy and invite one and all to attend their meetings. Some, however, do have dues for charter members but this often comes with additional perks such as access to tools, labs, and discounts on bulk purchases.

You may be wondering how a maker event could possibly help you with MicroPython or IOT projects. I am happy to report there is a very strong attendance and many attendees who are interested in IOT and even Python! As we learned from other chapters in this book, a lot of what we can learn to do in Python transfers to MicroPython. The same is true with hardware and IOT. Sure, you may only see demonstrations of IOT projects using other boards like a Raspberry Pi or Arduino, but much of the hardware can be used in MicroPython. If nothing else, you will gain knowledge of what is possible, which you can leverage in your next project idea.

Once you have attended an event - even one that has nothing to do with MicroPython such as a Raspberry Pi meetup, you'll be hooked. You can learn quite a lot from others this way and, who knows, perhaps one day you will be presenting at an event. Speaking from experience, it can be very satisfying sharing your knowledge with others in an open

forum like a conference, user group, meetup, or Maker Faire. That's when you know you've obtained the reputation, skills, and knowledge that a typical maker possesses.

■ **Tip** Many users of the Raspberry Pi use Python in their projects. Don't hesitate to seek out a Raspberry Pi meetup as you can learn quite a lot from Raspberry Pi Python developers.

That doesn't mean that by the time you read this chapter, you're an expert at all things IOT or even profess to be an expert, but you should be well on your way all the same.

Summary

Taking your IOT skills to the next level is what most people will ultimately be inspired to achieve. However, even if you do not want to become a traveling maker teaching the world, you can learn quite a lot by simply joining the online communities that serve Python, IOT, and open hardware.

In this chapter, I have presented suggestions on how best to interact with online communities, where to look to join an online community, and even how to take your skills to the highest level of enthusiasm and become a maker. This chapter therefore rounds out our journey down the road of MicroPython for the IOT that I hope inspires you to continue practicing your skills and ultimately join the community of like-minded enthusiasts.

I sincerely hope that this book has opened many doors for you concerning Python, MicroPython, IOT, and anything open hardware. May your IOT projects all be successful and if they aren't that you learn something in the process. Just remember to share your experiences – good or bad – with the world and give back a little of what you can take from the efforts of others.

Appendix

There are a lot of components and electronics used in this book. To make things a bit easier for you to acquire what you need to complete the projects in this book, I present a set of tables to help you make your own shopping list. I include two sections starting with the minimally required components to complete Chapters 8–11 followed by the list of components for the other examples in the book.

Required Components

This section lists the required components needed to complete the projects in Chapters 8–11. Table A-1 contains a consolidated list of the components needed along with the quantity needed, estimated cost, and links to sources. When more than one option exists, the rows are combined for the component name but you can safely order either of the options.

Table A-1. *Required Components*

Component	Qty	Description	Cost	Links
MicroPython board	1	Pyboard v1.1 with headers	\$45–50	https://www.adafruit.com/product/2390 https://www.adafruit.com/product/3499 https://store.micropython.org/store
		WiPy	\$25	https://www.adafruit.com/product/3338 https://www.pycom.io/product/wipy/
		WiPy Expansion Board	\$23	https://pycom.io/product/expansion-board-2-0/
OLED display	1	ssd1306-based SPI display	\$18	https://www.adafruit.com/product/661

(continued)

Table A-1. (continued)

Component	Qty	Description	Cost	Links
RTC breakout board	1	RTC module with battery backup	\$15	https://www.sparkfun.com/products/12708
Coin cell battery	1	See RTC breakout board datasheet	\$3–5	Commonly available
LED	2	Red LEDs	kit	https://www.adafruit.com/product/2975
LED	2	Yellow LEDs	kit	https://www.adafruit.com/product/2975
LED	1	Green LEDs	kit	https://www.adafruit.com/product/2975
Resistor	5	220 or 330 Ohm resistors	\$8–12	https://www.sparkfun.com/products/10969
Button	1	Momentary button, breadboard friendly	kit	https://www.sparkfun.com/products/12708
Breadboard	1	Prototyping board, half-sized	\$5	https://www.sparkfun.com/products/12002
Networking Module (Pyboard)	1	CC3000 breakout board (or equivalent)	\$15+	various
Soil Moisture Sensor	1+	Soil Moisture Sensor	\$6	https://www.sparkfun.com/products/13637
Weather Sensor	1	BME280 (I2C interface)	\$20	https://www.sparkfun.com/products/13676
Jumper wires	20	M/M jumper wires, 6" (cost is for a set of 10 jumper wires)	\$4	https://www.sparkfun.com/products/8431
Power	1	USB cable to get power from PC		Use from your spares
	1	USB 5V power source and cable		Use from your spares

The exception may be the MicroPython board. You are free to choose a different board than what is listed here as it is your choice and except for some differences in the firmware, most MicroPython boards will work. If you want to run all the projects in Chapters 8–11, you should consider buying the WiPy and Expansion Board. However, if you want to be able to run all the examples in the book, you should consider buying both the WiPy and Pyboard.

You may also want to purchase a basic electronics kit such as the one at <https://adafruit.com/products/2975> or <https://sparkfun.com/products/13973>. These kits normally have all the LEDs and resistors (and more) that you will need for most projects. If you find a kit that does not have a breadboard, be sure to add one to your order. See Chapter 2 for more information about basic electronics kits.

Optional Components

This section lists those components needed to complete the examples shown in the rest of the book. Table A-2 contains a consolidated list of the components needed along with the quantity needed, estimated cost, and links to sources. When more than one option exists, the rows are combined for the component name but you can safely order either of the options.

Table A-2. *Optional Components*

Component	Qty	Description	Cost	Links
Case	1	Case for your MicroPython board	varies	Pyboard: https://store.micropython.org WiPy: https://pycom.io/product/pycase-clear/
LCD160CR	1	Pyboard LCD shield	\$35	https://store.micropython.org/store
RGB Sensor	1	Adafruit RGB Sensor	\$8	https://www.adafruit.com/product/1334
Thermocouple	1	Adafruit Thermocouple	\$15	https://www.adafruit.com/product/269
K-Sensor	1	Adafruit Thermocouple K-Sensor	\$10	https://www.adafruit.com/product/270

These components are optional because the examples in Chapters 1–7 are intended to be informational rather than something you need to do to understand the concepts. That said, I include this table for those that want a complete set of components so that they can experience every example in the book. Even so, these are the minimal set of optional components. For example, I do not include all the various MicroPython boards, shields, and accessories listed in the book. Rather, I only list those you would need to run the examples.

Recommended Tools

While most of the examples in this book require no tools, some of the breakout boards and sensors (and some MicroPython boards) do not come with headers soldered. Thus, you may need a soldering iron such as the Hakko FX-901 Cordless Soldering Iron (<https://www.sparkfun.com/products/13151>) and some solder (<https://www.sparkfun.com/products/9163>). If you already know how to solder, you probably already have everything you need. If you do not want to purchase these tools, you may want to ask around and find someone who knows how to solder and has the tools. After all, you will only need to solder a few headers.

Other than a soldering iron, you may need a small screwdriver or pliers to assemble some components. That is, some boards come with mounts or risers and some cases require some assembly. However, you do not need more than a handy small screwdriver.

On the other hand, if you want to put together a basic electronics kit, see Chapter 7 for the typical tools needed for working with electronics. Figure A-1 shows a sample electronics toolkit from Adafruit (<https://www.adafruit.com/product/136>). This kit costs about \$100 and even has solder, a vice, and a few electronics to get you started! If you have no tools at all, you may want to consider a package like this one.



Figure A-1. Ladyada’s Electronics Toolkit (courtesy of adafruit.com)

Another excellent choice for the budget minded is the Beginner Toolkit from Sparkfun (<https://www.adafruit.com/product/136>). This kit has a more modest set of tools that beginner electronics enthusiasts will need to grow beyond the projects in this book. Plus, it comes in a stylish red box. Figure A-1 shows the basic electronics toolkit from Sparkfun.



Figure A-2. *Beginner's Electronics Toolkit (courtesy of sparkfun.com)*

Index

■ A

- Adafruit Feather Huzzah
 - auto-reset support, [105](#)
 - command-line tool, [104](#)
 - features, [104](#)
 - firmware
 - erasing, [106](#)
 - loading, [106](#)
 - uploading, [106](#)
 - GPIO headers, [104](#)
 - hardware features, [105](#)
 - REPL Console, [107](#)
 - wearable boards, [99](#)
- Advanced Encryption Standard (AES), [217](#)
- Alligator clips, [100](#)
- Analog-to-digital conversion (ADC), [345](#)
- Arduino Integrated Development Environment (IDE), [61](#)

■ B

- BBC micro:bit board
 - array of programmable LEDs, [96](#)
 - Bluetooth, [97](#)
 - hardware features, [97](#)
 - large-holed pins, [96](#)
 - Mu Editor, [98](#)
 - REPL console, [98–99](#)
 - up and running (micropython), [98](#)
 - WiFi module, [97](#)
- Bipolar transistor, [262](#)
- Bluetooth Low Energy (BLE), [226](#)
- Board-specific accessories
 - BBC micro bit, [119–121](#)
 - Pyboard, [117](#)
 - WiPy and related boards, [118–119](#)

Board-specific libraries

Pyboard

- classes, [206](#)
 - lcd160cr, [210](#)
 - pyb, [206](#)
- WiPy, [215](#)
- AES, [217](#)
 - pycom, [216](#)

Board-Specific Shields/Skins

- Pyboard add-on boards, [112–114](#)
- WiPy and Related add-on boards, [114–116](#)

Breakout boards, [226](#)

- Adafruit CC3000 Module, [110](#)
- BBC micro:bit
 - board, [111](#)
 - edge, [112](#)
- hobbyists and enthusiasts, [109](#)
- I2C protocol, [227](#)
- IOT project, [227](#)
- MicroPython, [109](#)
- serial peripheral interface, [233](#)
- soil moisture sensor, [111](#)
- Sparkfun, [109](#)
- weather, [110](#)
- WIZNET5K Ethernet chipset, [109](#)

■ C

- Callbacks, [224](#)
- CC3K module, [77](#)
- Circuit playground express board
 - Adafruit, [99, 102](#)
 - alligator clips, [100](#)
 - Bluetooth module, [101](#)
 - CircuitPython binaries, [102](#)
 - developer edition, [100](#)

- Circuit playground express board (*cont.*)
 - hardware features, 100–101
 - hardware libraries, 101
 - installation options (Adafruit Boards Driver), 102
 - MicroPython, 101
 - REPL console, 103
 - USB drive mounts, 103
- CircuitPython, 203
- Class instance variable, 143

■ D

- Dashboards, 388
 - available blocks, 391
 - blocks, 390
 - creation, 389
 - dialog creation, 389
 - feed selection, 392
 - name and description, 389
 - pressure, 395
 - rearranging blocks, 396
 - stream block, 394
 - temperature setting, 393
- Data structures
 - dictionaries, 134–136
 - lists, 133
 - tuples, 134

■ E

- Electronics
 - alternating current (AC), 254
 - breadboard-circuits, 265
 - application, 267
 - assorted breadboards, 266
 - breadboard layout, 267
 - definition, 265
 - power supply, 268
 - cathode, 254
 - components
 - breakout boards and circuits, 264
 - button, 255
 - capacitor, 256
 - diode, 257
 - fuse, 257
 - LED, 258
 - relay, 260
 - resistor, 260
 - switch, 261
 - transistor, 262
 - voltage regulator, 263
 - definition, 254
 - direct current (DC), 254
 - hand tool, 244
 - multimeter, 241, 246
 - current, 250–251
 - dial-up, 246
 - measure voltage, 249
 - resistance ohms (Ω), 252
 - testing continuity, 247
 - overview of, 239
 - sensors, 268
 - accelerometers, 271
 - analog, 270
 - audio, 271
 - barcode readers, 272
 - biometric sensors, 272
 - capacitive, 272
 - coin, 273
 - current, 273
 - DHT-22 humidity, 269
 - digital sensors, 270
 - Flex/Force, 273
 - gas, 274
 - IOT solution, 271
 - light, 274–275
 - liquid-flow, 275
 - location, 276
 - magnetic-tripe readers, 276
 - magnetometers, 277
 - measures, 269
 - moisture, 277
 - proximity, 277
 - radiation, 278
 - speed, 279
 - switches and pushbuttons, 280
 - tilt switches, 280
 - touch-sensitive membranes, 280
 - video, 280
 - weather, 280
 - soldering iron, 242
 - tools, 240
 - wire strippers, 244
- Electrostatic discharge (ESD), 65–66

■ F, G

- Fibonacci series, 158–159
- File transfer protocol (FTP), 90
- Fleet management, 13–15

■ **H**

Hand tool, [244](#)

■ **I**

Indentation, [127](#)

Inheritance, [143](#)

Instantiation, [143](#)

Inter-Integrated Circuit (I2C), [227](#)

 Adafruit RGB sensor (WiPy), [228](#), [232](#)

 breakout board, [227](#)

 console output, [231](#)

 driver, [229](#)

 init() function, [230](#)

 Pin() class, [230](#)

 RBG sensor (WiPy), [228](#)

 scan() function, [230](#)

 sensor.read() function, [230](#)

 Wi-Fi network (WiPy), [229](#)

Internet of Things (IOT), [415](#)

 automotive, [11–13](#)

 community project sites, [418](#)

 documentation, [417](#)

 fleet management, [13–15](#)

 forums, [416](#)

 internet, [3](#)

 join community, [421](#)

 annotation work, [426](#)

 appropriate, [425](#)

 check license, [424](#)

 contributes, [422](#)

 designs original, [424](#)

 good citizen, [427](#)

 Hackster.io, [423](#)

 license, [422](#)

 online communities, [427](#)

 share, [423](#)

 knowledge repositories, [420](#)

 medical applications, [7–10](#)

 repositories, [415](#), [417](#)

 security

 cloud, [18](#)

 devices, [17–18](#)

 encryption, [18](#)

 firewall, [17](#)

 physical access, [17](#)

 WiFi, [17](#)

 sensors, [2](#), [6](#)

 services, [4–5](#)

 several websites, [416](#)

■ **J, K**

JavaScript Object Notation (JSON), [150](#)

 json.dumps() method, [153](#)

 json.loads() method, [151](#)

■ **L**

lcd160cr classes, [210](#)

 firmware, [212](#)

 helper function, [212](#)

 LCD, [210](#)

 pyb.LED() function, [212](#)

 Pyboard, [211](#)

 REPL console output, [215](#)

 reusable function, [212](#)

learn.adafruit.io, [420](#)

Libraries of MicroPython

 built-in functions, [174](#)

 dict() function, [191](#)

 classes, [187](#)

 overview, [174](#)

 directory/file layout, [199](#)

 exceptions, [192](#)

 helper_functions.py module, [201](#)

 help() function, [176](#), [203](#)

 keys() function, [194](#)

 machine, [197](#)

 metadata, [202](#)

 my_helper() function, [202](#)

 network, [198](#)

 paraphrase/gasp copy, [173](#)

 raise() function, [192](#)

 sensor_convert.py module, [202](#)

 specific functions, [196](#)

 standard libraries

 format_time() function, [186](#)

 get_rand() function, [186](#)

 overview, [174](#)

 sys, [177](#)

 uio, [179](#)

 ujson, [180](#)

 uos, [182](#)

 utime, [184](#)

 user-supplied and custom

 libraries, [173](#)

 WiPy files, [200](#)

Light Emitting Diode (LED), [258](#)

Lower-level hardware

 abstraction layers, [205](#)

 breakout boards (*see* Breakout boards)

Lower-level hardware (*cont.*)

- callbacks, 224
- documentation online, 220
- drivers and libraries, 220–221
- on-board sensors, 220
- real-time clock, 221

Low-level hardware

- board-specific libraries (*see* Board-specific libraries)
- classes, 208

■ M

Maker

- definition of, 429
- event, 430
- progression, 429
- sharing idea, 430

Message queue telemetry transport (MQTT), 379

- Adafruit IO, 382
- brokers, 382
- clients, 382
- concept, 381
- driver, 398
- lightweight protocol, 380
- publish/subscribe model, 380
- website, 382

Method overloading, 143

MicroPython

- breadboard and jumper wires, 48–49, 51–57
- boards
 - assembly requirement, 63
 - community forums, 64
 - ESD, 65–66
 - firmware updates, 59–60
 - flash drive mounted, 160
 - GPIO pins, 64
 - jumpers, 67–68
 - micro SD card/drive, 67
 - networking issues, 60
 - programming tools, 61–63
 - protective case, 66
- categories of boards, 31
- compiled *vs.* interpreted languages, 29
- electronic components, 46–48
- features, 30
- interpreter, 37–38
- limitations, 31
- microcontrollers, 28–29

projects information

- clock (WiPy and Pyboard), 289, 310
- code completion, 306
- components, 285
- design, 293
- driver, 285
- embellishment, 311
- fill() and show() functions, 303
- hardware set up, 288
- imports section, 299
- initialize object instances, 301
- libraries, 293
- links, 284
- modules of RTC, 312
- new function, 302
- OLED breakout board, 305
- overview, 284
- Pyboard, 291
- RTC breakout board testing, 304
- run() function, 303
- set up interfaces, 300
- ssid1306.py, 294, 297
- uRTC.py Pyboard, 296
- WiPy, 290

Pyboard, 45

REPL console

- macOS and Linux, 43–44
- Pyboard, 40
- Windows, 40–43
- simulator, 20–24

Modularization

- built-in attributes, 147
- classes and objects, 143, 145, 147
- __init__() method, 146
- Pickup Truck class, 143–146
- Vehicle class, 141–142
- functions, 139
- modules, 138

Multimeter, 241

■ N

Network time protocol (NTP)

- server, 284, 368

■ O

Object-oriented programming

- language, 140, 143

Optional components, 435

Organic light-emitting diode (OLED), 284

■ **P, Q**

Passive infrared sensor (PIR), 278

Plant monitoring

code writing

calibration, 352

overview, 351

conceptual project, 345

data file, 375

hardware setup

connections, 349

Pyboard, 351

WiPy, 350

HTML code (files), 365

imports, 367–368

IP address, 376

main code

Pyboard, 372

WiPy, 369

network time protocol, 368

plant soil moisture solution, 345

project concept, 346

required components

components, 347

environmental factors, 348

soil moisture sensor, 349

run() function, 368

sensor code (*see* Sensor code module)

WiPy, 375

Polymorphism, 143

Printed circuit boards (PCBs), 246

Programming (MicroPython)

arithmetic, logical and comparison operators, 128–129

challenges

classes, 170

complex data and files, 155

functions, 162

loops, 149–150

code blocks, 126–127

coding

classes, 163–166

complex data and files, 151–154

functions, 156–159

loops, 148

comments, 127–128

conditional statements, 136–137

data structures, 133

data types, 132

executing, code

classes, 167–170

complex data and files, 155

functions, 159–162

loops, 148–149

loops, 137–138

print messages, 129–130

variables, 130–131

pyb library

Accel class, 209

classes, 206

forum.micropython.org, 207

hard_reset() function, 207

low-level hardware classes, 208

pyb.bootloader() function, 207

pyb.hard_reset() function, 207

pyb.info() function, 207

pyb.main(filename) function, 208

REPL function, 210

Pyboard board

Audio Skin, 114

hardware

boot drive, 70

data and datasheets, components, 71

internal drive to boot, 70

microcontroller and memory specifications, 71

pins, external power, 72

USR button, 70

LCD Skin, 114

load firmware, 73–77

networking

Adafruit CC3000 Arduino Shield, 78–79

CC3000 (CC3K), 77

CC3000 connection, 79

code and CC3000 module, 80

connection errors, 81

mapping pins and CC3000, 78

pins, 77–78

REPL console, 80

test_connect(), 80

testing connection, 80

WIZNET5000 (WIZNET5K), 77

overview, 69

PYBFLASH, 73

REPL console, 73

v1.1 headers, 69

PyMakr plugin, 93

PyMate app, 94

PySense Shield, 115

Python

features, 19

file access modes, 152

Python 3

execution, 36–37

Linux, 35–36

macOS, 34

Windows 10, 33

simulator, 20

PyTrack Shield, 116

■ R

Radio frequency identification device
(RFID), 279, 426

Real-time clock (RTC), 184, 221, 284, 288

Recommended tools, 436

Beginner's electronics toolkit, 437

Ladyada's electronics toolkit, 436

Required components, 433

Roman Numeral Class, 164–165

run() function, 368

■ S

Sensor code module

complete code, 359

constructor, 357

high-level design, 355

initialization section, 356

networks, 6

private functions, 358

public functions, 358

Pyboard, 361

sampling rate, 356

setup and initialization, 355

Serial Peripheral Interface (SPI)

Adafruit Thermocouple Amplifier, 233

header and terminal posts, 234

module, 234

Pyboard, 234

source code, 235

Type-H sensor, 233

Shields, 114

Skins, 112

Soldering Iron, 242

Stoplight simulation projects

code written, 322

components, 316

hardware set up, 319

overview, 316

pushbutton, 323

button_pressed() function,
325–326

cycle_lights() function, 325

functions, 324

imports, 323

setup, 323

test and debug code, 329

Pyboard, 320

remote control (HTML), 329

code completed, 335

functions, 333

HTML response string, 332

imports, 330

run() function, 334

setup, 330

web pages, 333

WiPy, 341

wireless network setup, 331

techniques reuse, 343

WiPy, 320

Systems on a chip (SOC) boards, 284

■ T

Teensy 3.X version boards, 108

Type conversion, 132

■ U, V

Ultrasonic Proximity Sensor, 278

■ W, X, Y, Z

Weather sensors

Adafruit IO, 387

AIO key, 397

credentials, 396

dashboard, 388

feeds creation, 387

BME280 library, 398

class code module, 403

classes, 399

constructor, 400

data modification, 411

design, 399

hardware set up, 385

historical data, 411

- imports section, 400
- main code
 - completed code, 406
 - connect() function, 405
 - global definitions, 405
 - imports, 405
 - run() function, 406
- message callback function, 401
- MQTT (*see also* Message queue telemetry transport (MQTT))
 - driver, 398
- overview, 380
- protocol definition, 379
- read data function, 401
- required components, 384
- run function, 401
- TCP/WebREPL connection, 407
- turn sensors ON, 408
- visualization of data, 409
- WiPy MicroPython board
 - firmware upgrade process, 91–93
 - hardware
 - Bluetooth communication mechanisms, 84
 - boot modes, 83
 - features, 84–85
 - Jumper installation, 84
 - micro SD card, 83
 - pins, external power, 86
 - product description, 85
 - WiFi, 84
 - overview, 82–83
 - Pycom Expansion Board, 83
 - Pycom WiPy, 82
 - PyMakr plugin, 93–94
 - PyMate app, 94
 - REPL console, 86–87
 - SD Drive, 90–91
 - WiFi network connection, 87–90
 - Wire strippers, 244