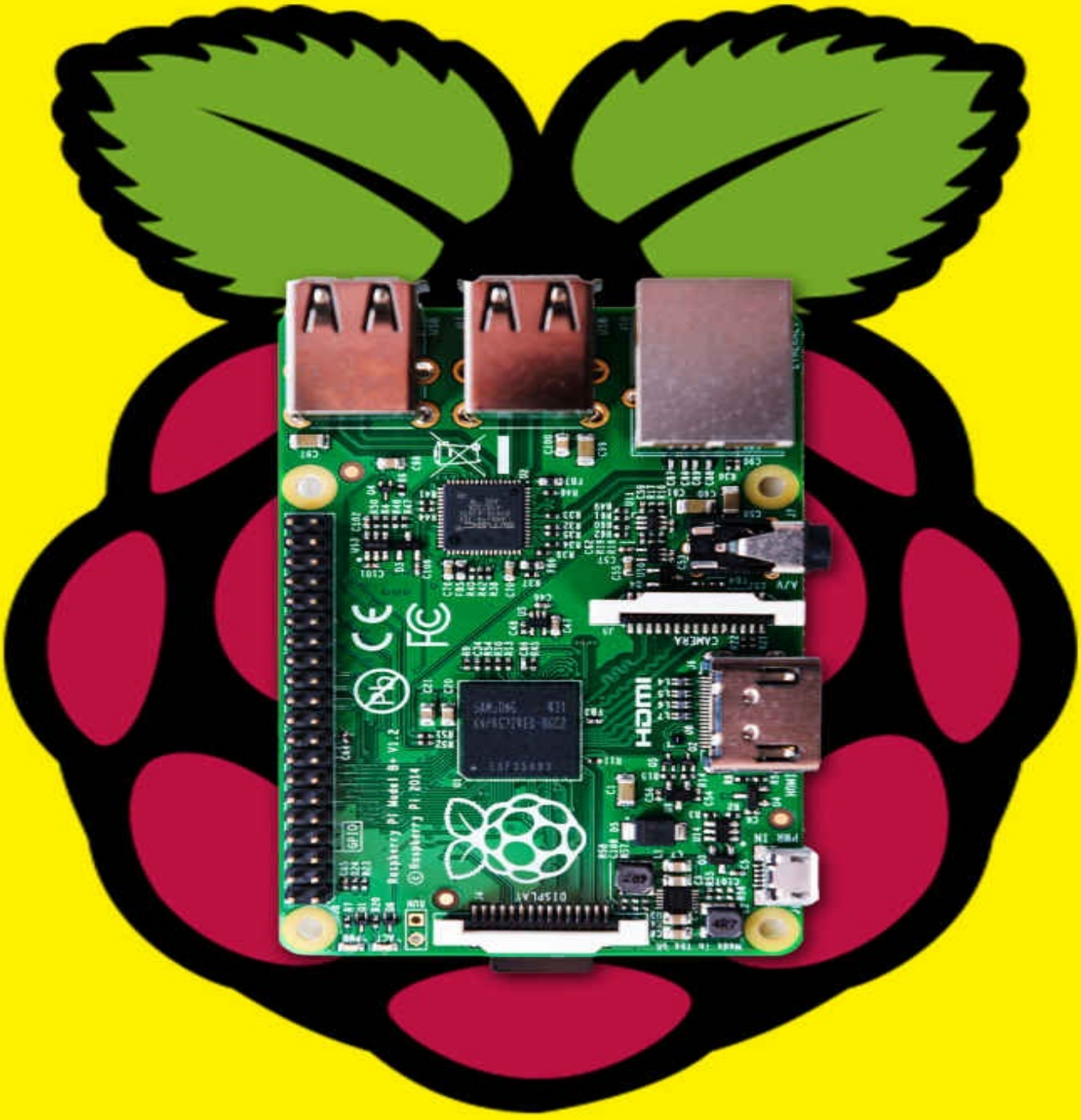
RASPBERRY PI

Step-by-Step Guide To Raspberry PI For Beginners



GABRIEL GREY

Raspberry Pi

Step-by-Step Guide To Raspberry Pi For Beginners

(Raspberry Pi Hardware & Software)

Gabriel Grey

Table of Contents

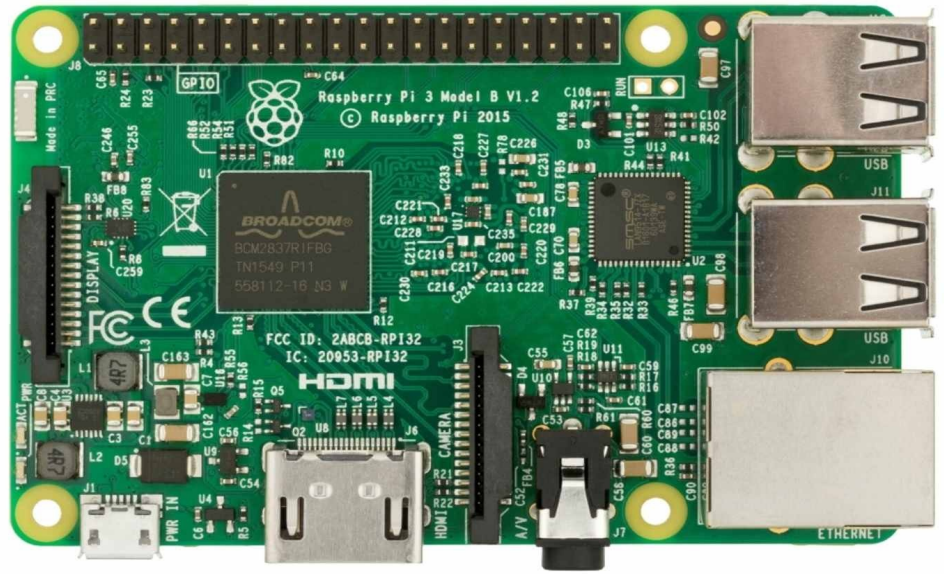
Introduction

Congratulations on downloading *Raspberry Pi* and thank you for doing so.

The following chapters will discuss the Raspberry Pi. The Raspberry Pi is increasingly popular. Why, though? What's all the fuss about? So many people are interested in this little amazing microcomputer and all of the things that it can do. This book is going to be focused around illuminating all of the reasons that people are falling in love with the Raspberry Pi and its various capabilities. We'll also be learning how to program in Python and the plethora of different things that you can use the Raspberry Pi for.

There are plenty of books on this subject on the market, so thanks again for choosing this one! Every effort was made to ensure it is full of as much useful information as possible. Please enjoy!

Chapter 1: What is Raspberry Pi?



Defining the Raspberry Pi

So, you know that you're interested in Raspberry Pi. You probably have some initial questions though: for example, what is a Raspberry Pi, and why is it so popular with *so* many people? Well, let's start with the basics.

The Raspberry Pi is a microcomputer in its purest concept. It was originally developed for sale to developing countries as an extremely cheap and easy-to-use computer that could be used in order to teach the basics of computing architecture and computer programming. However, it ended up growing well beyond the target market, and before the world knew it, the Raspberry Pi was a staple item among tinkerers everywhere.

Many things make the Raspberry Pi incredibly cool. It has myriad of different uses, some of which we'll be discussing in the following chapters. Moreover, there are many different variations that make it accessible to anybody and any project regardless of what exactly somebody is doing.

The Pi itself is easy to use, overall. It doesn't come with a keyboard, mouse, or

anything of the like. In fact, all that ships to the customer is a single motherboard with various little computer parts attached to it. However, this is part of the appeal. This means that the Raspberry Pi is a blank canvas on which you can unleash your creativity.

Many hobbyists will actually pick up the Raspberry Pi explicitly because of how cool and easy to use it is. It provides the perfect engine by which the tinkerer can try to make their wildest—or mildest—dreams come true.

Although its hardware is relatively unimpressive, this is probably what you should expect given that it's so cheap, so small, and so portable. Therefore, in a relative sense, the hardware is actually *very* impressive if you know exactly what you're getting into.

There are multiple different iterations of the Pi. I would personally advise you to get the newest version since it's only $35. Using older versions means using inferior hardware since the hardware is upgraded almost every version. There are some benefits to other versions, like some being a bit smaller, which is perhaps preferable for smaller-scale projects. However, for your average projects, just opt for the $35 version.

The Raspberry Pi offers a lot of functionality for what it is. You can run it as a normal computer, albeit a very underpowered one. You can plug in your keyboard and mouse and have it all display to a monitor or TV and then set up a desktop environment. In fact, that's what we'll be doing later in this book! That's a pretty big deal for a small little microcomputer.

In the next chapter, we'll discuss all the incredible things that one can do with the Raspberry Pi. We'll also look at what other people have made with it. This is going to give you a firm idea of just what this little powerhouse can do.

But if, as this chapter comes to a close, you have a single question about what Raspberry Pi is, the best answer is a tinkerer's paradise. You can do so many things using a Raspberry Pi that it's actually insane.

This is aided by the fact that there is a ton of support for different hardware solutions on the Pi that can enable it to do a plethora of different things that you might not normally expect it to be able to do. We'll be going into more detail on this in the next chapter when we discuss how it's been used and can be used.

The Uses of the Raspberry Pi

Raspberry Pis have a multitude of uses, some of which are more obvious than others are. Because they offer so much in terms of hardware compatibility relative to their small size, they have become the favorite of tinkerers. This makes them an obvious candidate for projects involving complex robotics. However, they've also found their home in a great many other projects.

Gaming Platforms

Perhaps the first thing that really tipped hobbyists off to how neat the Raspberry Pi was video games. Go figure, I suppose. Many people really like video games, and the Raspberry Pi is powerful and cheap enough that you can very easily build a dedicated emulator absolutely packed with games. The more powerful versions of the Raspberry Pi can even play Nintendo 64 games, which is incredible for how small of a computer it really is and how cheap it is.

While more resource-intensive games do stretch its capabilities, it does find a safe home being able to play a multitude of classic NES and SNES games, as

well as games for other consoles like the Sega Genesis, the Amiga, and so on. The first hobbyist implementations of the Raspberry Pi were often in dedicated gaming stations as such.

One popular implementation for the Pi was handheld gaming consoles. These would be things like a portable Super Nintendo and NES that only really had to be about the size of the Raspberry Pi itself. If you affix an LCD or LED screen to the Raspberry Pi and solder some buttons onto it, you would have a super cool portable gaming console with a rechargeable battery. These aren't terribly popular anymore, largely because the battery life wasn't that great and the novelty wore off, but they still are an exciting first project to undertake.

Speaking of video games, many people will actually use the Raspberry Pi as a MAME emulator. MAME emulators are emulators built to run arcade games like those you would find at a classic arcade. Many people will actually build arcade machines from scratch using the Raspberry Pi as their core technology. They'll set up things like arcade buttons and the like and use these as a means of controlling the emulator through some clever scripting or placement. Some seriously cool machines have been made following this paradigm.

Magic Mirror

I think my personal favorite implementation of the Raspberry Pi would be the Magic Mirror. The Magic Mirror is what it sounds like: it's a mirror with a digital display like the ones you would see in old sci-fi shows. All of the information you need is right at your fingertips, and some people have even programmed in voice recognition to these things so that they can control them with their voice. Using motion detectors or light sensors, the mirror would be able to detect when you're around and power on automatically, or it may be always on. It's not a huge power draw, after all—the same as constantly having an alarm clock plugged in.

I’m going to spend a second talking about this because I think it’s super cool and ingenious. The magic mirror is one of my favorite implementations of Raspberry Pi bar none because of the amount of creativity and ingenuity that goes into it. It’s cheap to make (comparatively), and you can actually make an amazing machine out of it—not to mention that there’s nothing else like it. There is no mass-marketed magic mirror. You will be one of the only people with this and you made it yourself, too!

So let’s think about the magic mirror for a second. What happens is that there is a frame. Behind that frame sits a computer monitor, and behind that sits the Raspberry Pi. In front of the computer monitor (or TV, but usually a computer monitor) sits a one-sided mirror. Do you know about those mirrors that they use in the police interrogation rooms in all of the crime shows? It’s a lot like those!

Because of the way the mirror reflects light, any black from behind the mirror won’t be shown. However, any bright things that are directly up against the mirror will shine through. As I said, it’s an ingenious implementation that really doesn’t get enough credit. The Raspberry Pi will be set up to automatically run a full-screen Chromium browser with a custom web page. Other times, the webpage is just opened in a modified Webkit browser that automatically takes up the whole screen, like Chrome OS somewhat. Either way, the Raspberry Pi natively launches into this mode, and from here, we actually start to develop somewhat of an idea of how the thing should work overall.

The customized home page is a black background with white text and images, and this usually has some sort of complex JavaScript code powering it to make it truly customized. One way or another, though, the person ends up with a really cool mirror. The mirror displays stuff like a “Good morning” or “Good evening” message, some have even been programmed to give a different compliment each

day. Some will display stuff for the day like the weather or the latest news, and so on and so forth. It's a seriously neat idea, and in a way, the fact that it was finally implemented indicates in one way or another that the future, at last, is here.

Raspberry Pi, because of the fact that it is cost-efficient and relatively powerful, has also been used in various robotics projects. While it's not as clear-cut for robotics as something like Arduino with its native support for things like motors and stuff of the like, the Raspberry Pi takes on the brainier projects with ease that other architectures may not be such a huge fan of. Raspberry Pi, as a result, has taken up some of the more complex artificial intelligence robotics projects that tinkerers want to take up, and other things similar.

Other Uses

Some people get seriously creative with the Raspberry Pi and will use it as the core of their media center. The Pi is great for this because it's low profile and has the capacity to store a huge amount of information in its small form-factor via its MicroSD slot. More than that, its internet connectivity also means it integrates seamlessly with services such as Spotify and Netflix.

Because of this, many people such as cinephiles and audiophiles will use the Raspberry Pi as a means from which to store and direct their home theater. Several infrastructures like Kodi catered specifically towards this and have been introduced over time and come to play a major part in the overall Raspberry Pi community.

The Bluetooth capabilities of the Raspberry Pi also mean that it's been opened up to use as a media center, speaker, and much more. Many people also take advantage of its WiFi compatibility and choose instead of using it as a whole media center to simplify and use it just as a place to stream stuff throughout the house. There are projects that actually link Raspberry Pis up to a central hub on the WiFi connection and then use this as a means to play music throughout the entire house, which is a seriously cool usage.

The Raspberry Pi also comes with built-in support for a ton of different sensors, thermostats, and things of the like. This means many people are using them to build things like robots that can automatically water their plants for them by detecting when they start to release excess amount of carbon, or even stuff like automatically detecting what nutrients the plant is short on and could use more of.

Some people opt to buy multiple Raspberry Pis and use them in order to build a server farm. These server farms consist of interlinked Raspberry Pis on a central hub that accept and distribute incoming traffic. Network junkies love Raspberry Pis for this reason. A simpler use would be to set up your own dedicated game server using a singular Raspberry Pi.

In the same vein, some people will buy multiple Raspberry Pis in an attempt to build a massive Bitcoin or cryptocurrency mining rig. While this has lost a lot of its efficiency for Bitcoin specifically because Bitcoin now needs an extremely

strong processor and essentially a GPU farm in order to mine anything meaningful, newer cryptocurrencies—especially memory-hard cryptocurrencies—can be easily mined using Pi-based mining operations. While there are better investments if you're serious about starting a mining farm (such as a lot of GPUs), if you just have extra Raspberry Pis lying around from past projects, then it could be a really cool way to make use of them and try it out for yourself. You'll also maybe make a little bit of money on the side.

The Internet of Things

The concept of the Internet of Things, too, is becoming increasingly popular. This can be a bit of a dense topic to grasp, so I'm actually going to start from the very bottom of it and work my way up to the point where Raspberry Pi is relevant. The Internet of Things is the idea of household gadgets being connected to one another via services such as the Internet or an intranet. Some people use a more liberal definition, which is simply everyday devices connecting to the Internet. It isn't explicitly *wrong*, but it fails to carry the same implications as the Internet of Things does.

The Internet of Things is more the idea that your alarm goes off at 7, you press the snooze button 3 times, and when you finally get up and turn off the alarm properly, a signal is sent to your coffee maker and your TV. It starts automatically brewing coffee and turns on the TV so that you can watch the news that morning. This sort of thing is the basic idea behind the internet of things.

So how does Raspberry Pi fall into this equation? I'd say that in one way or another, it's actually self-explanatory. The Raspberry Pi is a beautiful way of providing a relatively powerful computer interface to everyday things. If you have a decent knowledge of electrical engineering, you could solder things, alter relays, and actually program your everyday appliances to work in this manner.

If you lack that sort of background knowledge, then you still have more than enough opportunities to build everything from the ground up, or even just use your Raspberry Pi as a hub that your entire Internet-based items can connect to, as more Internet-based items come out. Many people have already programmed things like automatic settings to their coffee makers or the aforementioned robots

that would do things like automatically water plants or detect nutrient deficiencies. These sorts of things give you a clear idea of the possible uses of the Raspberry Pi in the context of the Internet of Things.

I mentioned earlier that the Raspberry Pi has been used increasingly more for the implementation of some kinds of robotics such as those which are “brainier” and more focused around artificial intelligence. A great example of that rests in the fact that some people have taken up the task of using open-source voice recognition software in order to try to build their own version of things like Amazon Alexa. These projects show how capable the Raspberry Pi is of being used for various different artificial intelligence applications.

Speaking of Amazon Alexa, many Raspberry Pi projects will actually interface with things such as Amazon Alexa or other home assistants in order to make voice-activated Pi commands a reality, among many other things. There are so many of these projects that it’s kind of hard to tell you where to start with them.

There is, in the end, a plethora of different uses for the Raspberry Pi. There are even more than I’ve mentioned here. Really, in the end, the Raspberry Pi is a small and cost-effective computer. Therefore, anything that you can dream about doing with a computer is something that you can do with a Raspberry Pi. It’s a powerful computer in its own right as well, so don’t be afraid to try to push the boundaries of what’s possible with the Raspberry Pi. I guarantee that you won’t be disappointed.

Review Questions

1. What is a Raspberry Pi?
2. What do you get when you purchase a Raspberry Pi?
3. In general, which version of the Raspberry Pi is best to use? Why?
4. In what ways can the Raspberry Pi be used?
5. What is the Internet of Things?
6. Despite being a tiny, underpowered computer, the Raspberry Pi is a very powerful device. Why?

Chapter 2: Setting Up Your Raspberry Pi

In this chapter, we're going to outline everything that you have to do to set up your Raspberry Pi. This will include everything that you need to get it up and running, so pay close attention. It is difficult at first, but it only gets easier from here! This guide assumes that you're working with a Raspberry Pi Model 3 B, the most recent model. However, if you're working with an older model, things will remain largely the same throughout the process.

First off, here's what you're going to need in order to set up your Pi so that it runs as a desktop computer:

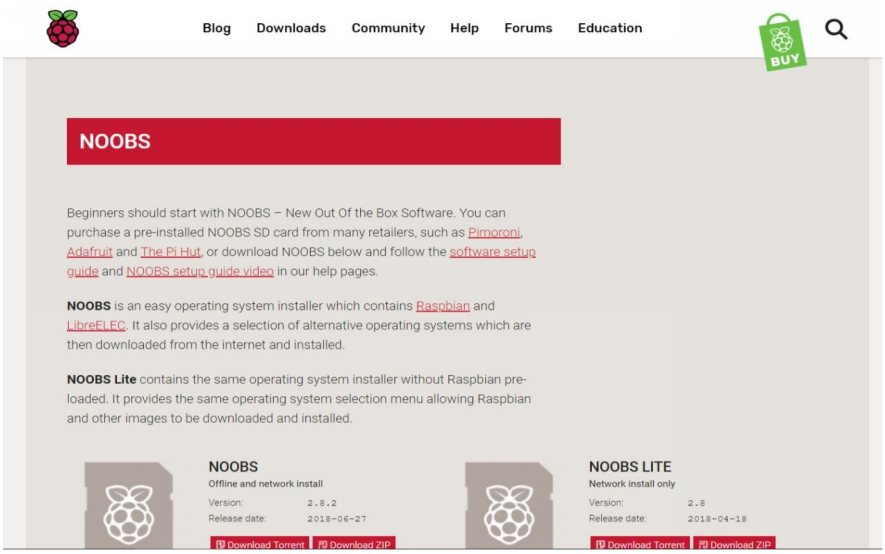
- Monitor (obviously)
- Keyboard and mouse
- MicroSD card
- Operating system

Choosing an Operating System

What operating system should you use on your Raspberry Pi? There are many different answers to this question. Many different companies have made versions of the operating systems that can run on the Raspberry Pi's software. Even Microsoft has released a version of Windows that is able to run on the hardware of the Raspberry Pi. So, bearing all of this in mind, what software specifically should you *use* on your Raspberry Pi?

Personally, I would recommend that you use the operating system *Raspbian* using NOOBS. NOOBS is easy to set up, it runs on Linux (explained later), and it is completely free! Moreover, Raspbian was actually developed by the team that created the Raspberry Pi, so it's designed for the hardware, whereas other operating systems may simply be compatible with the hardware. This means that things will work better out of the box down the line.

Regardless of what operating system you want to use, NOOBS will offer support for it and is an excellent operating system installer. While I would recommend that you install Raspbian, ultimately you have autonomy over whatever you decide to install on your Pi. Now that that's settled, let's talk more about the setup process.


Blog
Downloads
Community
Help
Forums
Education
BUY
NOOBS
Beginners should start with NOOBS – New Out Of the Box Software. You can purchase a pre-installed NOOBS SD card from many retailers, such as Pimoroni, Adafruit and The Pi Hut, or download NOOBS below and follow the software setup guide and NOOBS setup guide video in our help pages.
NOOBS is an easy operating system installer which contains Raspbian and LibreELEC. It also provides a selection of alternative operating systems which are then downloaded from the internet and installed.
NOOBS Lite contains the same operating system installer without Raspbian pre-loaded. It provides the same operating system selection menu allowing Raspbian and other images to be downloaded and installed.
NOOBS
Offline and network install
Version: 2.8.2
Release date: 2018-06-27
Download Torrent
Download ZIP
NOOBS LITE
Network install only
Version: 2.8
Release date: 2018-04-18
Download Torrent
Download ZIP

Installing NOOBS

These instructions will tell you how to set up Raspbian only. If you decide to install a different operating system, then this book will not be able to help you with that. However, as long as the installation is through NOOBS, the process should be the same for all operating systems. Therefore, you shouldn't have any problems installing them. With that said, let's install NOOBS first.

1. In order to set up your Raspberry Pi with an operating system, you will need to grab your SD card and insert it into your computer.
2. Do a search for SD Formatter 4.0. Download and install it. (Kindle formatting doesn't play nice with links so I cannot link you, unfortunately.) It's released by an organization called the SD Association, so as long as you're getting it off their website, you should be in the clear.
3. Install the software and start up the SD Formatter program. Select your SD card's disk drive and then format it exactly as the default settings indicate.
4. Go to your download of the NOOBS installer. You can get it by searching for NOOBS and then heading to the link hosted by the Raspberry Pi organization.
5. Extract the files somewhere, such as your desktop. Copy the files that were extracted over to your SD card, and then you're set.
6. Take out the SD card and put it in your Raspberry Pi's SD card slot.
7. Plug in everything: your monitor via the HDMI port, your keyboard, and your mouse. Ensure that your monitor is on the right setting.

If all goes well, you should be clear to finally plug your Raspberry Pi into the wall and get to some heavy development. If you're using a newer Pi model, it should have built-in WiFi. Older models, however, will require you to connect to

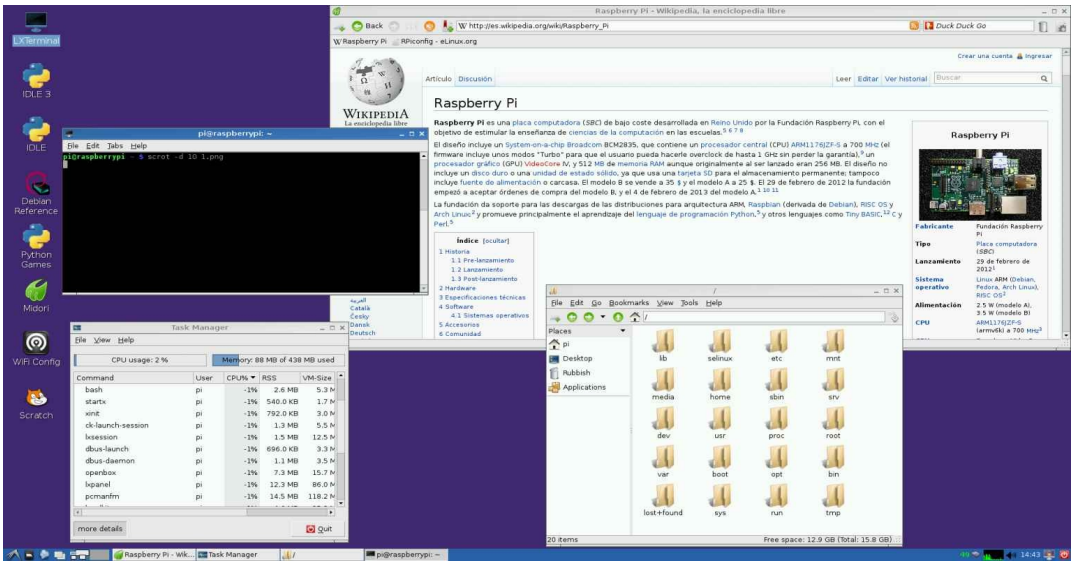
the internet via Ethernet or to plug in a WiFi adapter that is compatible with the Raspberry Pi.

You’ll know your Raspberry Pi is on when the indicator link on the Raspberry Pi is blinking. At this point, you’ll know if everything is going well because the indicator light will be on and you should have video displaying on your monitor.

Installing Raspbian

Now that we've installed NOOBS, we can proceed with installing Raspbian or your operating system of choice. Again, this book will only discuss Raspbian, but the process of installing the operating system is the same for all other operating systems that you can use for the Raspberry Pi.

1. On the main screen upon the operating system powering up, ensure that Raspbian is selected. This is the recommended operating system across most communities, especially for people who are new to Raspberry Pi.
2. Click Install next and then click Yes to confirm that it's going to overwrite your SD card.
3. Wait, and after a bit, your operating system will be installed. It's an extremely easy process like I said.
4. When all is said and done, your Pi will reboot, and you'll be brought to the main screen for the Raspbian operating system.



Getting Started

Now that you're on Raspbian's main screen, it's time to make the magic happen. This is where your adventures with the Raspberry Pi begin. Feel free to poke around a little bit and see what it has to offer. You can see right out the gate that there are quite a few programs built in that aim to help you learn to do various things, which really betray the origin of Raspberry Pi as something intended to help underprivileged people learn about computer science and programming in general.

So now, you're confronted with your operating system. If you've never used a Linux operating system before, you more than likely have a few questions. Moreover, even if you have used a Linux operating system before, there's the chance that this is not anything like what you've used before if you've primarily stuck to KDE distributions. So let's answer some questions first.

1. **What am I looking at?**

   The answer to that question is simple: Raspbian. Raspbian is a distribution of the Linux operating system, which you've most likely

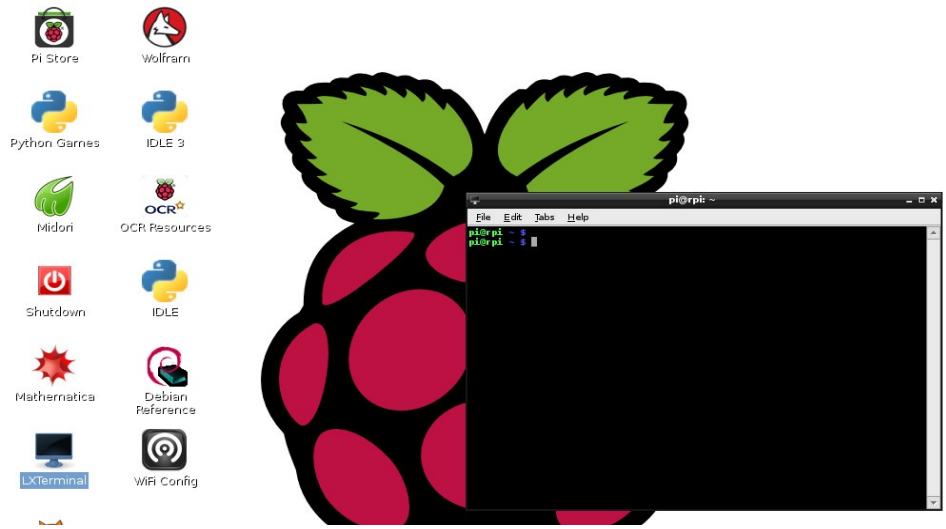
heard of before. It’s an offshoot of a popular Linux distribution called Debian. There are many other Linux distributions, one of which is Ubuntu. Raspbian has been designed to be a perfect match for the Raspberry Pi’s hardware demands and its specific CPU architecture.

2. **What is Linux?**

   Linux itself is an offshoot of another operating system from long, long ago called Unix. Unix was extraordinarily popular for various different reasons that a book could be written about all on its own (and numerous have, and they are quite good).

   Unix itself would inspire many operating systems that people use every day, including Linux (which in itself inspired the extremely popular Android mobile operating system) and MacOS (from which the iOS mobile operating system was derived). In other words, if you have a phone or a non-Windows computer, you’ve already been using a computer inspired by Unix.

   Because of its ubiquity in the ‘80s and the fact that operating systems such as Linux, Minix, and FreeBSD would fit extremely well into the hacker subculture’s belief in freedom of information and free software (free as in speech and free as in beer, both), Unix would remain the king of the software development world for quite a long time. Quite a long time leads up to the current day, where you’re sitting in front of an unfamiliar operating system wondering what to do. How quaint.



Let’s look for a second at our operating system. First, the most important part of a Linux system is the Terminal. If you use an Apple computer, you’ve probably poked around in the Terminal a few times as well. The Terminal was one of the most heavily used features of Unix because it offered an extremely easy way to get packages, manage your system, and do much, much more. This remains true today. Understanding Linux systems means, to an extent, understanding how the Terminal works and all of the many different things that you can do with it. So, if you want to have an idea of how Linux works, poke your head around a guide aimed at teaching you how the Terminal works.

One of the prime pulls of Linux, too, is the fact that it’s completely free, and this applies here no less. You can poke around your system’s information and have complete control over the computer and its motherboard. This is why so many geeks like me love Raspbian: it’s easy to use, but it offers all of the control and autonomy that Linux distributions do.

You’ll notice that Raspbian comes with quite a few different things packed in. Of note are a program geared at helping you with algebra related problems called Mathematica. You’ll also notice that there is a version of Chromium included. This browser is much like Google Chrome. In fact, it is just the open-source

version of Google Chrome. There's a difference between standard versions of Chromium and this version, though; this version is much lighter-weight than other versions, and as a result, it runs much better on the Raspberry Pi's delicate architecture. You can open as many tabs as you want! (Don't actually open as many tabs as you want, it won't go well.)

The last thing of note is the fact that the operating system actually comes with a version of Minecraft, which is referred to as Minecraft Pi. This is actually a lot like Minecraft, except it's geared towards helping kids learn how to program. However, you yourself might find it kind of fun if you poke around and try it. Besides, this is a long book, so it won't hurt to give yourself a bit of a break in order to play a game.

Linux is the best choice for tinkerers. There are many reasons. The first is that it keeps the overall cost of tinkering down since Linux's culture actually endorses the use of open-source and free software. This means that you can spend a lot less money obtaining software and a lot more time actually using your software. The fact that it's completely open is great as well because if you get to be good enough at programming, you can crack open the source code and modify it as you wish. There are no secrets, and you know exactly what you're getting into. In addition, perhaps the biggest pull is the fact that there are so many open-source tools available to you as a Linux programmer. People have been working with Unix-based systems for almost 50 years now, not to mention that the free and open source software movement has been around for more than 40. Believe me, if there is anything you want to do, there is almost certainly a program out there already that's been written to do exactly that. If there isn't, Linux makes it extremely easy for you to make it yourself.

Embedded Linux

Technically speaking, there is no such thing as Embedded Linux. When we talk of embedded Linux, we use it as an umbrella term to refer to an *embedded system that runs Linux.* An embedded system refers to a piece of computing hardware designed for a singular, specific application. In contrast, a Personal Computer has a multitude of purposes—browsing the Internet, playing video games, or writing eBooks about the Raspberry Pi. Lately, though, the line that separates general-purpose computing devices and embedded systems is blurring. In fact, the Raspberry Pi, this book's main topic, can be classified as both. It just depends on the purpose you bestow upon it.

To be clear, embedded systems are still different from general-purpose computers. They have distinct qualities that are theirs alone. These include the following:

- Their purpose is very specific, and they are often dedicated to this purpose.
- They are usually underpowered. They tend to lack the beefy power that personal computers usually have.
- They operate in a larger system, acting as a hub for other sensors and devices. This is in contrast with PC's, which usually act alone.
- Their roles are often quite significant, thus why they are assigned that specific task.
- They process things in real time.

You can have an embedded Linux setup by going nuts on the Terminal in Raspbian. The setup is a complicated process and it assumes you already know a thing or two about technical stuff, so I will not discuss it in this book. Feel free to do research on it yourself. We still have more things to learn about the Pi.

Review Questions

1. Why is Raspbian a great choice for new Raspberry Pi users?
2. How do you install Raspbian or any other operating system for the Raspberry Pi?
3. What is Linux? Why is it relevant to Raspberry Pi?
4. What is an embedded system? How do they differ from personal computers?

Chapter 3: Using Your Raspberry Pi

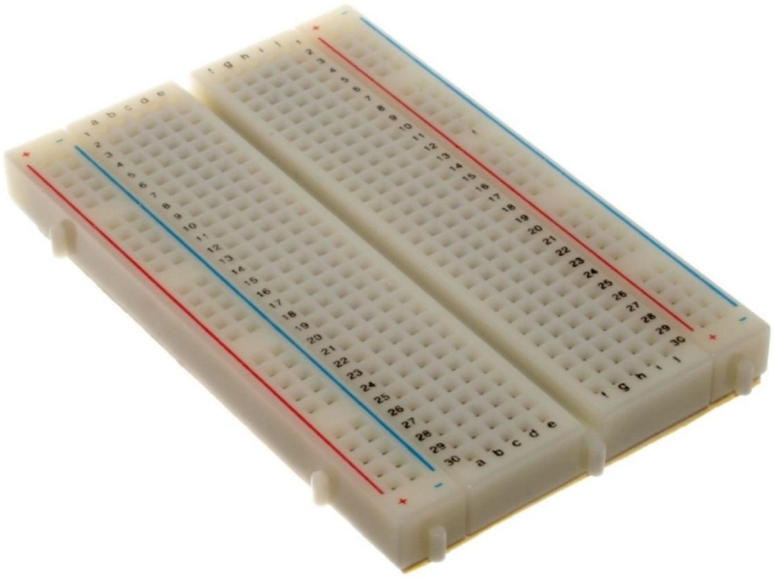
Now that you have learned about basic Python programming, it’s time to move on and actually use your Raspberry Pi. Setting up your Pi to work with sensors, diodes, and other cool tech stuff can be quite a challenge. However, we’ll go through each of the ways you can use your Pi so you can rest assured that you’ll be knowledgeable about it before diving in.

Interfacing Electronics

Your Raspberry Pi would be useless if you weren't able to use it to interact and use other electronic devices, wouldn't it? Here, we'll discuss how to set up your Pi to work with other electronics. First, you'll need to have the proper equipment in order to make sure you won't destroy your circuit or even your Pi.

Digital Multimeter

It is essential that you have one of these before starting to tinker with circuitry. This device measures many things such as voltage, current, resistance, etc. This ensures that you don't accidentally pump your circuit with more than it can handle.



Breadboard

A breadboard is a base for you to use when making prototypes for electronics. Before trying things out on your Raspberry Pi, try it on a breadboard first. Make sure to get the good ones!

Discrete Components

- **Diodes**

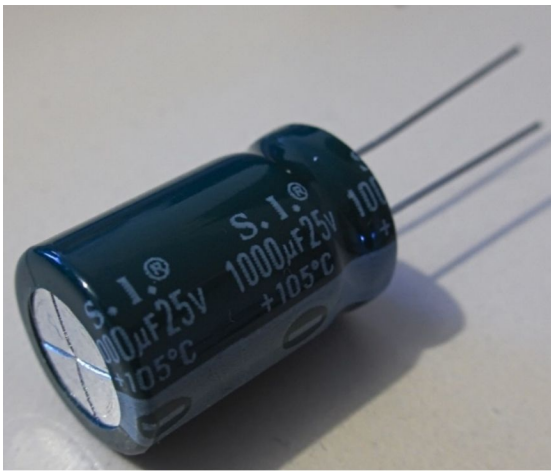
  A diode is a semiconductor component that simply allows one current to flow in one direction but not the other.

- **Light Emitting Diodes (LEDs)**

  An LED acts similarly to a diode, just that it emits light if the current flows in the correct direction. These come in many shapes, sizes, and colors. The length of the leg determines which leg is positive (cathode)

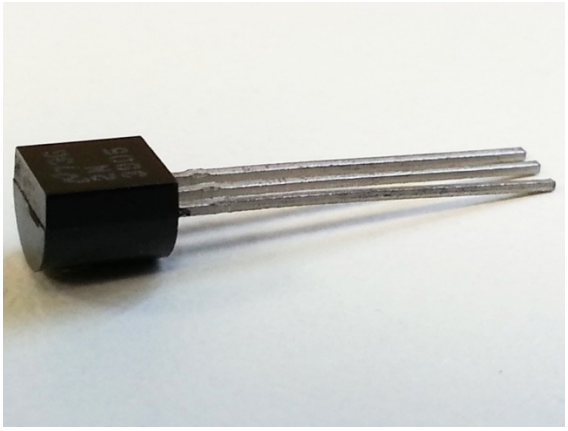
and which is negative (anode).

- **Capacitors**



A capacitor is a component that can be used to store electrical energy. It stores energy when there is a difference in voltage between its two plates. Once the voltage difference dissipates, it releases the stored energy.

- **Transistors**

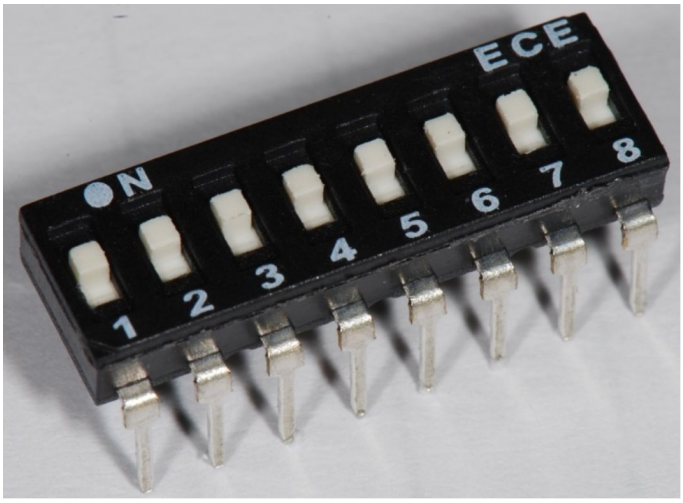


A transistor is a semiconductor component that can be used to amplify or switch electricity or electric signals.

- **Optocouplers**

These are digital switching devices that allow you to isolate two electrical circuits from one another.

- **Buttons and Switches**



These are quite self-explanatory. These are the input devices that you interact with to make your circuit do something. Their basic function is to open or close a circuit. They come in different shapes and forms, depending on what you need.

Communication Protocols

In order for embedded systems to work harmoniously, there needs to be communication between them. This is the way in which data is transferred between embedded systems. There are certain standards that are set in place to make sure there is consistency and coherence in their communication. These are the communication protocols.

A number of communication protocols exist, and the difference between them would be better understood if you learn these few concepts first:

- **Bit rate**

  The bit rate describes the number of bits that are sent per unit of time. This is usually described in bits/sec.

- **Baud rate**

  Whereas the bit rate describes the number of bits sent per unit of time, the baud rate describes the number of *symbols* sent per unit of time. These symbols can each be of any number of bits. This depends on the design. If ever the symbols are only 1 bit, the baud rate would be equal to the bit rate.

- **Parallel Communication**

  In parallel communication, multiple bits are sent at the same time.

- **Serial Communication**

  In serial communication, bits are sent one bit at a time.

- **Synchronous Serial Communication**

  This describes a serial communication protocol wherein data is sent at a steady, continuous stream at a constant rate. This requires that the internal clocks of the two embedded systems be synchronized at the same rate so

that the receiver receives the signal at the same intervals that the transmitter used.

- **Asynchronous Serial Communication**

  This form of serial communication does not require synchronized internal clocks. In place of the synchronization signal, the data stream instead contains start and stop signals before and after the transmission, respectively. When the receiver receives the start signal, it prepares for a stream of data. Conversely, when it receives the stop signal, it resets to its previous state to receive a new stream.

Now that you've learned about the basic concepts of communication between embedded systems, we can now learn about the different communication protocols.

I^2C

I^2C is short for Inter-Integrated Circuit. It is a synchronous serial communication protocol that uses two wires: one for data (SDA), and one for the clock (SCL). It is a multi-master, multi-slave serial computer bus. Most of its uses are confined to attaching lower-speed peripheral integrated circuits to processors and microcontrollers. Because of how it works, I^2C must validate the data passing through it by evaluating whether or not the data on the SDA line changes when the SCL is high. The data on the SDA line should only ever change when the SCL is low. Otherwise, the data is rendered invalid.

- The bus structure is a wired AND gate. This means you can test if the bus is idle or occupied.
- Once a master changes a line's state to HIGH, it always has to check if that line has actually gone to high. Otherwise, it's an indication that the bus is occupied.

- I^2C supports a wide range of voltages.
- I^2C is half-duplex.
- I^2C can support serial 8-bit data transfers up to a speed of 100kbps. This is the standard clock speed of SCL. I^2C is also capable of a higher bitrate: 400 kbps (fast mode) and 3.4 Mbps (high-speed mode).
- I^2C is mainly used for short-distance communication.

UART

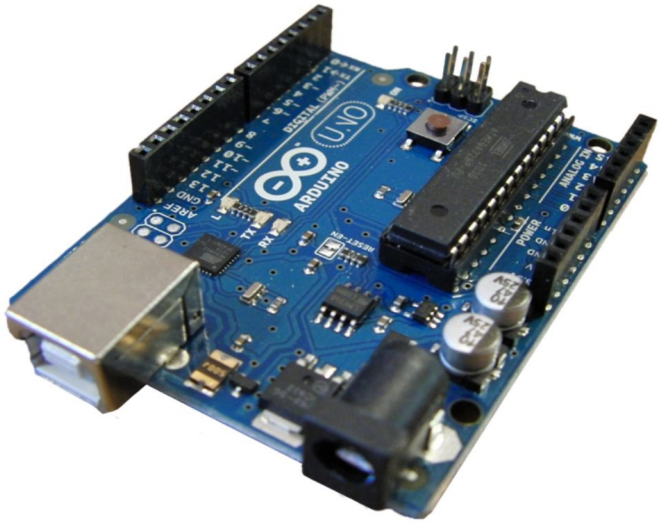
UART is short for Universal Asynchronous Receiver Transmitter. In this protocol, one wire is used for transmitting, and another wire is used for receiving. UART uses a serial type of communication, therefore bits travel in one wire.

- UART supports communication through RS232.
- Standard baud rates for UART include 110, 300, 600, 1200, 4800, and 9600.
- UART can only support communication between two devices at any one time. This is because it is a point-to-point communication protocol.

SPI

SPI is short for Serial Peripheral Interface. It is a synchronous serial communication interface protocol used for short-distance communication. It can operate with one master and several slave devices.

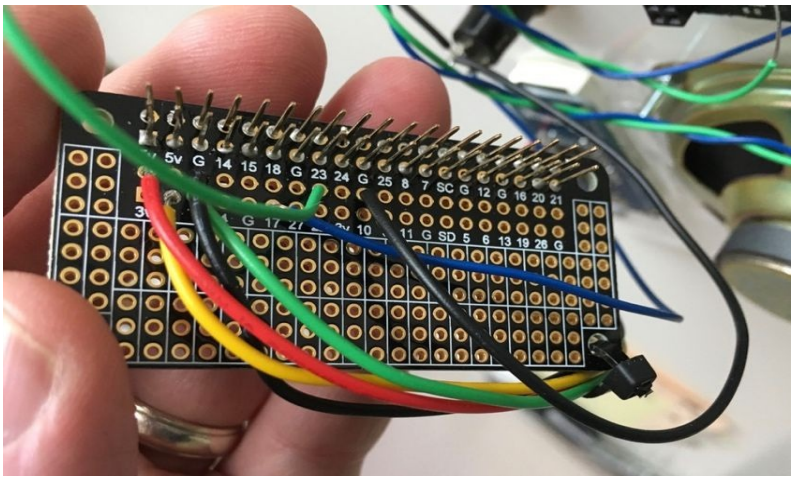
- SPI is a full duplex type of communication protocol.
- SPI protocol has no limit for message size, making it very flexible.



Real-Time Interfacing Using Arduino

In case you aren't familiar, an Arduino is a powerful microcontroller. You can use it in tandem with a Raspberry Pi, creating some impressive projects. Obviously you'll need an Arduino for this to work. You'll need a lot of programming expertise and mastery of interfaces to make use of this, and explaining that will make this book longer than it needs to be, so feel free to do some further research on this topic. However, I'll discuss a few key things.

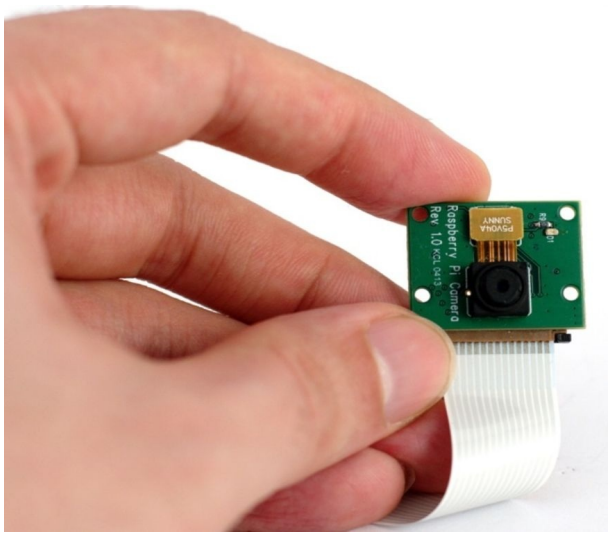
- You can interface with the Arduino using any of the communication protocols discussed above (I^2C, UART, and SPI).
- You can configure the Arduino as an I^2C slave. This means you can connect several Arduinos to one Raspberry Pi.
- A straightforward UART connection can only support one slave at a time.
- If you require a fast, high-level interaction between your Arduino and Pi, configuring the Arduino as an SPI slave will be the way to go. This is because an SPI connection will only be limited by the Arduino's clock speed.



Input and Output

You probably have noticed that row of pins along the top edge of the Raspberry Pi board. These are the GPIO pins. GPIO is short for General Purpose Input/Output. Using the software, you can designate whether each of these pins are for input or output. You can do many things with this. Two of these pins are 5V, and two more are 3.3V. There are also several ground pins, which you cannot configure. The rest are general-purpose 3V3 pins.

If you designate a pin to be an output pin, you can set it to high, at 3V3, or low, at 0V. Conversely, input pins can be read as high (3V3) or low (0V).



Capturing Images, Videos, and Audio

You can use your Raspberry Pi to capture photos and record videos and audio. Obviously, you would need the required peripherals if you want to try this. You'll need a Raspberry Pi Camera ($30) or a USB Webcam and a USB audio and audio HAT.

Images and Video

There are many reasons why you would want to capture images and video using your Pi. There's home security, robotics, automation, image/video streaming, etc. If you have the right combination of peripherals, you can stream high quality video. This stream can be viewed asynchronously. The only limit there is to their durations is the capacity of the available storage on your Pi and any additional attached USB storage devices.

To get started, you need to have a camera. You can use either a USB Webcam which you can purchase in any computer store, or you can buy a camera built specifically for use with the Pi, the Raspberry Pi Camera, which you can purchase for $30. We'll only discuss the Raspberry Pi Camera here to avoid making this too long.

The Raspberry Pi Camera is tiny, only a 1” x 1”square. It attaches to the Pi’s camera serial interface or CSI via a 15cm-long ribbon cable. It captures up to 5 megapixels, supports up to 1080p full HD video at various framerates, and even up to 90 frames per second on VGA resolution. You may also get an infrared filter for it if you wish.

To attach it to the Pi, follow these steps:

- Turn off the Raspberry Pi unit. Do not touch the metal contacts of the ribbon cable. You might ruin it!
- Take off the lens protector.
- Get the CSI connector and gently pull up the housing clip (this is usually black or white).
- Insert the CSI cable into its slot.
- Push down the housing clip to lock it in place.
- Turn on the Pi and configure the camera:
    - Enable the camera with this command:

      pi@erpi ~ $ sudo raspi-config

- Reboot.

To capture images, input the following command:

```
pi@erpi ~ $ raspistill -o image.jpg
pi@erpi ~ $ ls -l image.jpg
```

To capture a 10-second video, input the following command:

```
pi@erpi ~ $ raspivid -t 10000 -o video.h264
pi@erpi ~ $ ls -l video.h264
```

You can do many more things like stream videos and set up a home security system with this feature. However, we'll not discuss those, so feel free to do your own research about it.

Recording and Playing Audio

Most of the time, you will need your video to be accompanied by audio, and sometimes you will need just the audio. You may also want to connect a speaker to the Pi so you can play music, play pranks on friends, or something entirely different. To set up the audio, you will need an audio input or output device. However, the Pi actually comes with a built-in audio output system, which connects via an HDMI port. For input, however, you will need another device.

- **USB Audio**

  You can attach a USB audio input device as long as it supports Linux drivers. You may also use USB webcams that have microphones.

- **Bluetooth Audio**

  You can use a Bluetooth audio input or output system and connect it to the Pi using a Linux-compatible Bluetooth adapter.

- **Raspberry Pi HATs**

  HAT is short for "Hardware Attached on Top." Pretty clever name, isn't it? Anyway, you can attach a HAT that has audio capabilities to the Pi.

To record/play audio, you will need the ALSA utilities software. It contains the aplay and the arecord utilities that you will need to record audio. To install this, simply input this command:

```
pi@erpi ~ $ sudo apt update
```

```
pi@erpi ~ $ sudo apt install alsa-utils
```

Now, let’s move on to using these to record and play audio. To record audio, input the following command:

```
pi@erpi ~/tmp $ arecord -f cd -D plughw:1,0 -d 10 test.wav
```

To play the audio you have just recorded, type this command:

```
pi@erpi ~/tmp $ aplay -D plughw:1,0 test.wav
```

Review Questions

1. What is the most important tool you need to have before starting to experiment with interfacing between your Pi and other electronics?
2. What are the different communication protocols that can be used with the Raspberry Pi? How do they differ from one another? What are their advantages and disadvantages?
3. How do you interface your Raspberry Pi with an Arduino?
4. How do you capture still images, record videos, record audio, and play audio on the Raspberry Pi? What devices would you need to do so? What software is needed?

Chapter 4: Python Programming for the Pi

Most scripting on the Pi happens through Python. However, some other languages occasionally find their way into use in Pi systems. For example, you will occasionally see the language Bash being used in order to write scripts that can be run from the Linux terminal and that can do commands from within the Linux terminal.

However, when it comes to writing full-on scripts, the language of choice is Python. Python has native support on Raspberry Pi, and it's extremely easy to get up and running in no time. However, the challenge comes in actually learning how to program on the Raspberry Pi. This is easier said than done. After all, some people are paid a lot of money to program! There's a very high skill cap to being a good programmer.

This chapter aims to traverse that skill gap and give you all of the useful information that it can regarding programming in Python so that you can start your Raspberry Pi scripting adventures on a strong note. So, with all of that said, let's tackle some essential questions.

What is Python?

Here’s a short and simple answer: Python is a programming language. To elaborate, computers don’t really understand any languages that we speak. Natively, they speak solely in computations. Normally, they’re just permutations of 1’s and 0’s and performing operations upon those. This is popularly known as binary code, and it’s an important foundation of programming and computer science in general.

In order to speak to computers, give them instructions, and tell them what to do, we need a means by which we can start translating what *we* want into *their* language. This means is ultimately called a *programming language*. You use a programming language in order to convert the characters and terminology that *we* use into code that the computer can understand and then execute.

Why Python?

So at this point, you might be wondering—what sets Python apart? Why should we use Python over any other given language? What makes it different? The simple answer is this: Python has a lot of support, is natively supported, and actually works well with the Raspberry Pi platform.

While there are many different scripting languages that are available for you to use on the Raspberry Pi, few are going to be as efficient, well supported, and natively acceptable as the Raspberry Pi. There's a reason that so many different utilities built specifically for the Raspberry Pi are built in the Python programming language. That should be all the reason you need. No language is so good at doing what Python does as Python is.

Additionally, as somebody who is new to programming and is taking in a lot of information, Python is great for you to use because it is simple. Python is easy for a newer programmer to pick up while being powerful enough that a programmer who has been around a time or two will have no problem getting into the Pythonic workflow and learning how to make the most of this powerful language.

Setting up Python

Python comes built into Raspbian, so there isn’t much that you have to do in the way of setting up. You can verify that Python is installed by opening up your Terminal and executing the Python command. If for some reason Python isn’t installed, you can go ahead and navigate to the Python website and download a version for your operating system. If you have Python 2 for some reason, you need to uninstall it and install Python 3 instead. Raspbian should come with Python 3 automatically, which clears up this worry from the get-go.

Data and Variables

The first thing that we're going to look at is mathematical program operations and how you can perform them. In order to understand this, though, you need to understand a thing or two about data.

Python understands data in a relatively unique way, but there's a reason that it's relative; all programming languages understand data in this same kind of detached manner. Data is, for lack of a better term, any singular piece of information that is used in order to represent some given concept. Any individual piece of data is referred to as a *value*. Values can take on several different forms.

However, under it all, computers aren't actually understanding any of these different forms; instead, computers understand the raw idea of ones and zeroes, binary calculations that are happening far, far under the hood of the computer. On top of the binary code is one layer of abstraction, known as Assembly code, which operates upon the bits, or the different sets of binary code that represent individual values. On top of that is another layer of abstraction, known as the operating system. Then there's yet another layer of abstraction, the programming language in use. This works with the operating system to convert something that we can understand into assembly language, which the computer's processor then converts into a set of different calculations. All of this happens in the matter of micro or even nanoseconds.

The key point of all of this is that computers understand things in terms of ones and zeroes, and the way that we see a value means next to nothing to a computer. In order to solve this, programmers long ago decided that computers would actually categorize different values into different *types*. These types of data tell the computer how and in what manner to perform operations to the given values

as they correspond to the ones and zeroes. This is a super complex architecture, so don’t feel too bad if it doesn’t immediately make a whole lot of sense.

Anyhow, in pursuit of properly understanding all of this, it’s necessary that we start to break down these data types a little bit and look at them with a more abstract eye than we currently are. So let’s do that. The next thing that we’re going to do is take a sincere look at the different kinds of data that you can use in Python.

Integer

The integer data type refers to any piece of data that corresponds to a whole number in our abstract understanding. Therefore, these would be numbers such as 7, 39, or -3.

Float

The float data type refers to any piece of data that corresponds to a decimal number in our abstract understanding, so things such as 3.141569 or 94.3332.

Double

Double stands for “double precision” number and refers to a very specific kind of decimal number. You don’t need to understand this *too* in-depth at this point because of float and double act rather synonymously in Python. The reason for this distinction goes back to a time when computers had less RAM and less processing power than they do now, but for our purposes, you can largely ignore this.

Boolean

Boolean means true or false values. This will make a lot more sense later on

when we start to talk about programmatic logic and the way that logic actually plays a part in computer science.

Character

Character stands for any singular alphanumeric or symbolic character that can be printed out in a computer console. These could be things like A, 3, or $. This is a fickle understanding, though, because characters actually correlate to an ASCII value, which means that any given character also has a numeric integer value. For this reason, if you had the character “3” and the integer 3 and tried to see if they were the same, they wouldn’t be. Bear this in mind as you program.

String

We’ll talk about strings more in-depth later, but strings are essentially long chains of characters that are put together. Any set of character values is a string, whether it is 2 characters or 2000 characters long.

These are not all of the values available for you to use in Python. However, they are the ones that you are most likely to use almost immediately, so we’ve covered them here for that reason specifically. These values may be expressed in any given expression in Python. For example:

```
print(3 + 3)
# would print 6

print(“Hey there!\n”)
# printing a string to the console
print(‘C’)
# printing a character to the console
```

In other words, these form the very nucleus of everything that you’re going to be

doing in programming. Every piece of code you ever write will be working with values like these and manipulating them in one way or another. As you work with more and more code, you'll come to appreciate how truly often you actually make use of all of this and how every statement in a program is just the manipulation of data in one way or another. This is the nature of programming, for better or worse.

Sometimes, you're going to want to keep up with these data pieces so that you can recall them or change them at a later point. What can you do for this purpose? The answer is simple. You can use *variables*. Variables offer a method by which you can keep track of values over a long period as you work through a program.

Recall earlier how we talked about data types. Data types were especially useful and a bit more diverse than they are in Python because Python tries to go out of its way to make things easy for you; however, all of these values are stored in the computer's memory, and they're stored in boxes of pre-allocated size depending on how much space any given data type uses.

These individual boxes in the computer's memory can be referred to as variables. Picture it like the overhead view of a given city. You may have a bunch of lots that you can place houses in. You then will refer to any given lot by its address. The lot is like the variable itself, and the address is the *name* of the variable.

Therefore, you can actually store all of these values in variables where you decide the name to refer to it by. So, let's say that you had a variable called something like *dogAge*. If your dog was 4 years old, then you may set this like so:

```
dogAge = 4
```

If your dog's name was *Lucky*, then you may set a string variable like so:

```
dogName = "Lucky"
```

Python makes it extremely easy to name and declare variables. Some other languages have a bunch more hurdles to the process, but Python most definitely does not. This can be both a blessing and a curse. In other languages, you may have to say the *type* of the variable when you declare it, but Python takes this burden off you.

Why would this be a bad thing? Well, simply put, it can be confusing for a newer programmer who doesn't have much experience working with different data types. You may end up forgetting and trying to make a comparison between two pieces of data that aren't of the same time, actually messing up your data in the meantime because the computer doesn't compare different pieces of data in the same exact way.

This is the reason that you actually want to learn what the individual data types are. It will help you realize, for example, that the string "34" and the integer 34 are not the same and should not be compared, and may explain to you why your comparisons may be off at one point or another if you're not careful about this.

Python Math

Of course, working with variables is much more useful if you're actually doing operations on the data in question—for example, if you're actively performing math operations or performing useful equations. In this section, we're going to be exploring the different ways in which you can work with data.

Bear in mind primarily that you can refer back to variables. For example, if you wanted to print a string that you saved in a variable, you could do it like so:

```
print(dogName)
```

Or, if you were ambitious and wanted to print out your dog's name and dog's age both, you could do it like so:

```
print("My dog's name is " + dogName + " and they are " + dogAge + " years old.")
```

But what if something changed? What if, for example, your dog aged by a year? What could you do?

Well, you'd want to take your dog's age and then add one to it. But how can you do this? Well, you can do this by actually assigning it a new value. You can reassign values to variables and manipulate the variables that they have much like you set them and initialized them in the first place. The process is similar for the most part. Let's say, for example, that we wanted to add 1 to the variable dogAge. We could do that like so:

```
dogAge = dogAge + 1
```

The variable dogAge would take the old value of dogAge, 4, ,and add 1 to it, and this would be set as the new value for the variable dogAge. Make sense? Therefore, if you printed the variable *dogAge* now, it would print out the number 5:

```
print(dogAge)
```

Python has numerous different operators that you can use to do math. The Python mathematical operators are like so:

c + d
This is the addition operator. It is used in order to add one number to another.

c - d
This is the subtraction operator. It is used in order to subtract one number from another.

c * d
This is the multiplication operator. It is used in order to multiply one number by another.

c / d
This is the division operator. It is used in order to divide one number by another.

c % d
This is the modulo operator. It is used in order to find the remainder when

you divide c by d. For example, 7 % 3 would yield 1 since 7 divided by 3 has a remainder of 1.

These are the primary different mathematical operators in Python that you need to know. Using this knowledge, you can carry out complex mathematical operations in Python and make some really cool things happen. But this is only the beginning!

Let's note for a second that the way that we reassigned a value earlier wasn't necessarily the best way to do it. That is to say that the statement "dogAge = dogAge + 1" can easily be shortened and made easier to both read and understand. There are a few different shorthand operators in Python for assignment. These are as such:

```
c += d
# This just means c = c + d.

c -= d
# This just means c = c - d.

c *= d
# This means c = c * d.

c /= d
# This means c = c / d.


c %= d
# This means c = c % d.
```

As you can see, these operators aren’t terribly difficult to understand, but they can go a long way for simplifying your code and making it easier to read as a whole.

Comments

Comments are essential for programming unless you want to get lost in your own code. It is especially important when working with a team. Comments are parts of code that, from the computer's perspective, do *absolutely nothing*. Why is it important, then? Comments are important so that you can insert text in your code and not have it affect the program itself. You can use these to tell your fellow programmer to not touch a certain part of the code because it's currently a band-aid solution as you try to fix another part of the code. For our purposes, you can use this as a guide for yourself so you know what part of the code does what and how.

Formatting

If you've ever programmed in another language, then you'll have noticed by now that Python is quite different in many ways. Not the least of these ways is the manner in which Python handles formatting. Many popular languages are ambivalent in regards to whitespace; statements are separated by a semicolon, and you could put your whole program on the same line if you really wanted to. There are even competitions in languages like C and Java to obfuscate code and make it as pretty as possible at the expense of readability.

Python, on the other hand, cares a *lot* about whitespace. Whitespace in Python—that is, line breaks, spaces, and tabs—indicate to Python the hierarchy of the code. This is the main engine by which Python actually starts to understand your code, so you need to pay close attention to your whitespace. Make sure that you're indenting things just as I do and paying attention to how the indentations actually affect the flow of your code as well as how your code works altogether.

User Input and Casting

Here, we're going to spend a brief minute talking about taking in user input. There are going to be many times where you're going to need to retrieve information from the user. For example, you may be asking for the name of the file or for some kind of data necessary to the program from the user. It may even be something as innocuous as a book title if you're writing something like a library or bookkeeping program. One way or another, programs thrive not off just existing but off interaction and their ability to interact with the user and make things happen.

Because of this, it's important that you understand how user input in Python works. It's actually relatively simple.

All user input in Python—at least by means of the console—is handled through the *input* method. The input method allows you to take in information from the console. It will read everything up until the Enter button is pressed and return all of that information as a string.

The *input* method works like so:

```
input("Prompt text")
```

You can set the prompt text to whatever you want or leave it out entirely. All prompt text indicates is that the text that is fed to the input method, as an argument will be displayed to the user in question.

You can set the input method as the value of a variable, and this will set whatever the user enters as the value of that variable. For example, if my text

were like so:

```
food = input(“What is the last thing you ate?”)
```

and the user entered *nachos*, then the value of *food* would be *nachos*. Therefore, if we printed the variable *food*, it would print as nachos:

```
print(food)
# would print as nachos
```

Sometimes, though, this isn’t the end of the line. Let’s say that you were writing a calculator program and you needed to accept numbers that the user entered. Of course, the input method returns a *string*. You know from our discussions earlier about how data types work that strings are not the data type that we really need at the moment; no, we actually need a float value or an integer value. So how can we convert whatever the user entered into one of those values?

You can do this by *casting*. Casting is simply the conversion of one data type to another data type. In Python, variables can hold any data type, so you can actually just set the casted data type as the new value for the old variable, but you don’t really want to do this just for the sake of maintaining clean code and being, well, a good programmer. In fact, it’s probably best that you avoid this particular plan at all costs and just make new variables because it’s more readable and secure anyway. Use a single variable for your user input and then just set your other variables as the casted form of that. For example:

```
in = input(“What is the number?”)
number = #casted in
```

Casting values is easy. All that you do is put the type you’re trying to cast them

to in between parentheses right next to the value, like so:

```
number = (float)in
```

This would set the value of *number* to be the value of *in* casted to a float. Python automatically handles these tricky type conversions for you, for the most part, so you don’t have a whole lot to worry about there.

Introduction to Program Logic

So now, it's time that we start talking about something a little bit deeper. This sort of logic goes by many names; some call it symbolic logic in the liberal arts fields, while people in science and engineering fields prefer to call it discrete mathematics. It also goes by the name of *statement calculus*. Regardless of what you're programming or what language you're programming in, it's important that you have a strong foundation in this sort of logic because it's foundational to the world around you.

Some people say that learning to program changes the way you think, and this is true in one way or another because programming forces you to learn to think like a computer. So, how do computers think? To put it simply, computers think in a very, very black and white manner. They don't think like you or I do. They think in much simpler terms. This is this and that is that, there is no room for grey areas.

This can be a great thing because this is *pure logic*. Pure logic is the state where things either *are* or *are not*. Logic has a long history of being the pursuit of truth through the understanding of things which *are* or *are not*. One of the oldest and boldest examples of this sort of logic lies in the Socratic applications of logic early in the history of Western philosophy; consider, for example, that old Socratic syllogism:

All men are mortal.
Socrates is a man.
Therefore, Socrates is mortal.

This is one of the most basic and easy to understand applications of logic because it makes perfect sense. Logic is ultimately based around two things: *arguments* and *truth*. Logic itself is based entirely around arguments, but logic is used in the deduction of truth.

This section is going to focus on both of these because this question is central to computer science and having a firm understanding of it really can only benefit you as a hopeful programmer.

Arguments

What is an argument? An argument is not just a fight; rather, arguments are a means by which a point is *argued*. An assertion is made, an assertion is checked, and a conclusion is reached. Arguments are composed of two key parts: a premise and a conclusion. For example, in the argument above, we have two premises—*all men are mortal*, and *Socrates is a man*. These can be distilled even further:

For all p, p = q.
There exists p(S).

Of course, this in and of itself is not a revolutionary development. This is just giving an abstract form to the argument, and this is critical in understanding logic. These premises actually give way to a certain conclusion. The first premise is a blanket statement, and the second is our actual assertion, the part which leads us to manifesting our conclusion. Since all *p* is *q*, if there is an instance of *p*, then that singular instance of *p* is *q*.

Therefore:

For all p, p = q.
There exists p(S).
Therefore, p(S) = q.

This is the basic dissection of a logical argument. All logical arguments can be understood in terms very much similar to these. Computer programs are, in a way, logical arguments; they are a way of proceeding throughout logical operations using established rules of conduct, transitivity, and so forth. They are the brain of the computer making arguments constantly, and they are a massive part in the next part of this chapter, which is control flow.

However, you'll notice something about this argument; the entire argument is hinging around the premises being true. While the argument's form is completely correct, if this argument were used for the wrong thing, then the argument itself would be pointless regardless of how solid the form is. For example:

All cats named Mike are tabbies.
My cat is named Mike.
Therefore, my cat is a tabby.

Logically, this argument is completely sound. However, what if my cat named Mike were a Siamese? This couldn't obviously be true. This premise is far too ambitious, it assumes too much, and it has to at one point or another is untrue—right?

Therefore, this is the important part that *truth* plays in logic. Logic is great at the distillation of forms into abstract and easily applicable concepts. However, it still hinges on the discovery of truths at its core, and without the discovery of truths,

the institution of logic as a whole is relatively useless.

Moreover, how do we discover truths? The simple answer is through comparison. For example, in the former argument, without some means to discover truth, then I have no way of proving that the argument is fallacious since the argument's form is consistent and there is no truth to compare it to. Nevertheless, I can compare my cat named Mike and see if it's a tabby. If it isn't a tabby, then clearly the argument is untrue, right?

And this is the foundation of finding truths through comparisons. In computer science, especially beginning computer science, these truths can be simple to understand and easy to work with. However, they do get more complex as time goes on, so just be wary of that as you move forward. Anyway, let's think about comparisons now.

How are comparisons performed in computer science? Comparisons are performed primarily through the usage of expressions. If you think back for a second then you can probably remember a time in your life, maybe middle or high school, when you were working with early algebra and you first started working with expressions. These are basic concepts like less than, greater than, and so forth. You learned that the equals sign itself was just an expression that stated that one thing was equal to another, and following this form, you could actually perform algebra on any expression.

These expressions stick around as you go through math; don't worry. Here is one of the numerous arenas where they actually rear their ugly head again, so that's a lot of fun. Expressions are distilled comparisons between one value and another following a standard form. The different kinds of expressions in Python are like so:

```
b == c
# This means b is equal to c.

b < c
# This means b is less than c.

b <= c
# This means b is less than or equal to c.

b > c
# This means b is greater than c.

b >= c
# This means b is greater than or equal to c.

b != c
# This means b is not equal to c.
```

And of course, the point of expressions—as I said—is to find some meaningful expression of truth. Expressions are representative of relationships between values. Let's say, for example, that I had two values 7 and 3, and I wanted to express them like so:

```
7 < 3
```

This would read as 7 is less than 3. Is this true? Clearly not; 7 is actually greater than 3. Therefore, this expression is *false*.

```
7 > 3
```

This expression on the other hand would be *true*, since 7 is greater than 3. Herein lies the primary purpose of expressions—they aim primarily to serve as a means to evaluate truth through comparisons between values, and they do this job excellently.

Note, of course, that since variables represent values, you can also easily compare two variables, like so:

```
myVariable = 7
myOtherVariable = 3

myVariable > myOtherVariable
# This would be true, of course, since 7 is greater than 3.
```

Note, too, what I said about these expressions having innate truth or false values depending on how they evaluate. Remember earlier when we talked about Boolean values and I mentioned how they were a little tricky? Yeah, this is pretty much what they're for. Boolean values hold true or false information, which can be much more useful than it seems, primarily for reasons that we'll discuss a little bit later on in the book. For right now, though, just understand that Booleans hold true or false values. Expressions also return true or false values. Can you see where I'm going with this?

You can actually *use* variables to store the value of expressions as true or false, like so:

```
myVariable = 7
myOtherVariable = 3
myTruthVariable = myVariable > myOtherVariable
```

```
print(myTruthVariable)
# would print out True, because it's True.
```

While this isn't particularly useful in and of itself much of the time, the real utility in me showing you this is you realizing the exact way that expressions work, because it will make a lot of things click for you mentally down the line if you realize the manner in which they work right now.

Note too that you can actually chain these statements together into an even bigger statement. You do so using *conditional operators*. Conditional operators are fantastic because they give you a means by which to take these simple logical statements and string them together into far more complex logical statements.

In Python, there are three conditional operators:

1. expression1 **and** expression2

   This will evaluate if both expression1 *and* expression2 are true. If so, then the entire statement will come out to be true. If even one of them isn't true, then the whole thing is false.

2. expression1 **or** expression2

   Understanding the logical or can be a little tricky, because it doesn't always mean the way that we understand it. The logical or just means that one of the statements is true. If either expression is true, then the entire statement is true. If both statements are false, then the whole statement is false. However, note that this doesn't mean that both statements can't be true; if both statements are true, then technically the or condition—

wherein only one has to be true—is satisfied, it's just that the condition is technically satisfied twice. See what I'm saying?

3. **not** expression

   This just checks to see whether the given expression is false. If so, then the whole statement is true. If the expression is true, then the whole statement is false. This can be a little tricky, but when you see how people use it in their code it will start to make a bit more sense to you, I promise.

With that, we've laid an important logical foundation for the rest of this chapter. Understanding how computers see logic is truly essential to having an ample and able understanding of computer logic yourself. Stay with me because things are going to get a little bit more confusing from here, but I've given you a great groundwork for the rest of it.

Conditionals

Here, we're actually going to start discussing the very first aspect of control flow: the *conditional statement*. Before we go further, let's talk about what control flow really is for a second.

Control flow is the process by which you give your computer the ability to think. While that might sound a little dramatic, that's really what it is. All control flow solutions are very rudimentary forms of artificial intelligence, and if you look at the structure of actual large artificial intelligence programs, they're all built off of these control flow structures.

And the simple fact is that people *want* intelligent programs. Not only do they want programs to be intelligent, but also the vast majority of programs have to

be intelligent in one way or another. An unintelligent program is hereby defined as a program that doesn't have to make a decision at any point throughout its running duration. Can you imagine how boring these programs would be? Their only purpose would be to open up, run some data, and then exit. These programs have no practical utility outside of unloading files—and in fact, even a good file unloading protocol requires, to one extent or another, the development of architectures for intelligent programs through control flow, unless they're in the exact directory they need to be working with files that they came with.

So, keeping all that in mind, let's start to think a little bit more extensively about what aspect of control flow we're specifically dealing with right now. At this given moment, we're working with the idea of *conditionals*. So, then, what is a conditional? A conditional statement is a statement that makes decisions in the program based off expressions that you provide, and they deal with those expressions in a predetermined way that you write.

There are two different kinds of conditional statements that you're primarily going to be using: passive conditionals and active conditionals. These are primarily delineations that I created myself, but you'll find that they hold true and are also a useful way for describing the flow of a given program. The *passive conditional* is far simpler, so we're actually going to focus on that one first.

What is a passive conditional? A passive conditional is a conditional statement that exists on its own within the code. When you get to the position of the passive conditional within the code, the expression of the conditional statement will be evaluated. If the expression is true, then the code within the conditional statement will be run. If it isn't true, then the whole block of code can be skipped over. This is the nature of the passive conditional. Passive conditionals are expressed as if statements on their own.

An if statement can be written in Python like so:

```
if expression:
    # code goes here, indented once inward
```

So, for example:

```
if myNumber > userNumber:
    print(“My number is bigger!”)
```

In the code above, we are evaluating to see if one number is bigger than the other is, presumably the programmer’s number and the number of the end user. If we get to this point in the code and the programmer’s number is *not* bigger, then the entire code block is skipped through and the program doesn’t actually say anything to the end user.

Of course, sometimes this isn’t exactly what you’re wanting. Sometimes, you want something to happen no matter what. In these cases, you can actually use an *active* conditional. An active conditional stands counter to a passive conditional because it ensures that no matter what, if the tested condition isn’t true, then some code runs.

Note that you’re not always going to want to use an active conditional. There are many cases in which you may need to use a passive conditional instead, such as checking to see whether some piece of data exists or not or something of that nature. However, having an understanding of the distinctions between the two conditionals is important whilst also understanding the nature of them and their usage.

Active conditionals add what's called an *else* statement to your passive conditional. Essentially, it gives your statement an out by giving your code something to do if the condition turns out to not be true. This is a pretty big deal for obvious reasons. It gives you the ability to have backup code to run if your stated conditional turns out not to be true.

Else statements can be added like so:

```
if statement:
    # code goes here, indented once inward
else:
    # backup clause
```

Note that you can't have an else statement without an if statement, that wouldn't make sense obviously. Let's look at the code we worked with before in order to build a firmer understanding of these clauses and how they work, and how we can actually use them in our own code:

```
if myNumber > userNumber:
    print("My number is bigger!")
else:
    print("My number is smaller.")
```

In this variation on the former code, when we arrived at this conditional statement within the code, we would first evaluate the expression in the if statement. We would see whether it was true. If it were true, then we would print out "my number is bigger". Now, we have an additional clause that we can act on if it turns out not to be true—"my number is smaller". If the programmer's number *isn't* bigger, the code won't just skip past the statement—it's going to do its backup, which is to say that the number is smaller than the user's number.

Nifty!

Nevertheless, you may be thinking—sometimes I'm going to need to test more conditions! For example, what if the numbers were *equal*? There's no clause for that! Well, in short, you're wrong! There is most certainly a clause for that. These programmers have thought of everything.

You can actually test as many conditions as you want in a conditional statement. You can add additional expressions to test by way of the *else if* statement, shortened to the *elif* statement. Else if statements are written just like if statements, but are sandwiched between the if and else statement and use the *elif* keyword instead.

Note again that you can't have an else if statement without an if statement and an else statement.

The way that else if statements work is by evaluating every expression one by one and acting accordingly. If, for example, the *if* statement's expression isn't true, then the *elif* statement's expression will be evaluated. This can happen as many times as necessary. If none of the conditions tested turn out to be true, then the program will default over to the else statement.

We can actually add a clause to our former conditionals in order to check for the numbers being equal using an elif statement:

```
if myNumber > userNumber:
        print("My number is bigger!")
elif myNumber == userNumber:
        print("Our numbers are equal!")
else:
```

```
    print(“My number is smaller.”)
```

See now? I hope that it’s beginning to become clear why this is such a fundamental and awesome part of programming. You can actually make your computers start to think and implement their own logic based upon comparisons of data that you influence within the program. That’s incredible! There’s another aspect to control flow, but in order to discuss that, we’re first going to have to talk about lists.

Lists

Earlier in this book, we were talking about data types and variables. Perhaps it wasn’t exactly clear then, but sometimes—even with such a robust type system—that’s simply not going to be good enough for whatever applications you may need. In these cases, you can actually use lists in order to expand the overall functionality of your program. Take, for example, the fact that sometimes you’re going to want to group your data together.

It may not seem immediately obvious why you’d want to do this (or maybe it does), but just in case it doesn’t, allow me to demonstrate. Let’s say that you’re trying to keep a running list of all of the names of the students in your programming class. How would you go about doing this? With what we have so far, we can only create individual variables for every kid:

```
Student1Name = “Bill”
Student2Name = “Rita”
Student3Name = “Jim”
```

This would continue ad nauseum, presumably until you had listed out the name

of every single kid in the class. This is obviously unwieldy for reasons that are painfully clear. First off, how do you easily list off the names of all of the students at once? You'd have to print out each variable one by one. More than that, it would complicate the process of actually accessing and changing the variables in question because it would make it unclear which one is which. That, or there would be *overfitting* of variable names such that there wouldn't be enough abstraction to make a meaningful change in your code.

How could we avoid this situation altogether? The obvious answer is by grouping like data together. It just speaks for itself, really. If two pieces of data are similar and are going to be accessed normally in tandem with one another, then they should be side-by-side.

This idea, as well as its implementation, has its origins in the establishment of arrays in early programming paradigms (that's right, early programmers were thinking a lot about problems like these!). Arrays functioned similarly to the way that lists do.

Remember earlier how I mentioned that when you set up a variable, you're actually setting up a box that holds a value in the computer's memory, generally one that's actually the size to fit the data that you're trying to enter into it? This actually really handily explains the whole concept behind the idea of arrays.

The problem is that computers don't really put these boxes together in an organized way. You ever seen the back of somebody's moving fan, and how it seems like they're just fitting things in there? It's not exactly a great analogy, but it is a little bit similar, if only because in memory, the box tends to be created automatically in the most immediate and available location. Arrays ensure that data which is alike will be placed next to each other. The original definition of an array is the allocation of contiguous memory (that is, the creation of boxes to

hold data which are right next to each other) of the size *elements e * size of data type d*. They would then be partitioned like hostel beds and closed off. The arrays were perfectly sized so as not to waste memory, and they also allowed for contiguous data. Win-win. See how elegant of a solution that really is?

Arrays essentially gave you the ability to name this area of contiguous data and then access the data within it. The name would be whatever you set it as, and you could then get to the data within it with ease. You could set the data in any of the partitions of your array, or even set up the partition and fill it with data later. Arrays were an extremely useful feature in their time and day.

You can actually still use arrays in Python if you really want, but there's no reason, and it's heavily discouraged in the Python community. The thought is generally that if you need such tight allocation of memory, you shouldn't be using Python in the first place, and I have to say that I generally agree with this notion. Arrays in Python have been displaced by lists.

What's the difference between the two? Well, arrays had one big problem. Since they were pre-partitioned, they could only hold data types that you declared them as holding (because different data types need different sizes of partitions) and you could only fit as much data within it as you initially declared yourself as needing.

Lists outgrew this. While they were also implemented in other languages, like C++ through the standard template library or Java through the importing of the utilities package, Python was one of the first major languages to actually place an *emphasis* on lists. Lists differ from arrays in two major ways. First, they are dynamically sized. They can be as big or small as you need them to be, and they'll actually grow up or down as you write more code. This is one of the great advantages of using lists in general.

Lists also, since they don’t have a set memory size, may hold numerous types. This may not seem immediately useful, but as you get more experience, you’ll see why this can be a bit of a godsend, especially when importing data since it simplifies the process many times over.

Lists are in many ways the natural progression of arrays, and they offer an incredibly easy and intuitive way to store data. Declaring a list in Python is simple. Firstly, since you can always add data later, you can declare an empty list and just add data after the fact:

```
myList = []
```

However, you can also start the list *out* with data.

```
names = [“Bill”, “Rita”, “Jim”]
```

See how simple that is? But there’s more to it, we’re only just getting started! You can easily append data to lists by the use of the *append* function. This will add whatever new information to the list that you designate. So, let’s say that a student named Cory joined our class. We called add him to our list of names like so:

```
names.append(“Cory”)
```

Now, if we tried to print our list, it would print like so:

```
names[“Bill”, “Rita”, “Jim”, “Cory”]
```

But what if we wanted to work with an individual piece of data from this list?

How do we do that? Well, first things first, we need a definition. A single piece of data from a list or array is referred to as an *element*. As you'll notice, the elements in a list have their own positions. These are referred to as their *indices*. You can refer to an element of a list by referring to its index. There are also functions for doing a backwards search, e.g. supplying the piece of data in a list and then finding the index of the element matching that data. However, that's a little beyond the scope of what we're working with right now.

Computers are weird, though, and for typically weird computer science-y reasons, computers start counting their indices at 0. So, if you wanted to refer to the first element in a list, you would actually refer to the element at the first index, which is *0*.

You can refer to elements of a list like so:

```
listName[index]
```

So, if we wanted to print the second name in the list *names*, we could do it like so:

```
print(names[1])
```

This would print out *Rita*. And with that, we've built a very rudimentary understanding of lists and how they work. However, there's a little more to it.

Sometimes, you need to remove a piece of data from a list. How can you go about doing this? Well, Python has your back. Python has a really handy keyword called the *del* keyword which is specifically intended for the deletion of data. In order to delete an element from a list, you just have to refer to it like you usually would using the del keyword.

Let’s say that Jim quit our class. We now need to remove him from the roster of names. How do we go about doing this? Simply put, we just have to use the *del* keyword and refer to his index. Since he’s the third student in the list, his index would be *2*.

```
del name[2]
```

This would remove Jim from the list. Also, it would move any students that were ahead of Jim in the list back one position. This means that Cory would now be in the third position. Bear this in mind if you ever use the element keyword, because any code which depends intimately upon the positions of elements and isn’t written so that it scales up and down may become buggy after the deletion of an element in a set. Usually this is only an issue with bigger and less well managed code bases, so if you start out with this in mind, you aren’t going to run into too many problems.

With that, we’ve defined a basic methodology for thinking about lists and the various different uses that they’ll have for your as a Python programmer. In the next section of this chapter, we’re actually going to start thinking about the next aspect of control flow—as well as seeing how lists can apply to them in an extremely important and central way.

Loops

Finally, after a long déluge into a heavy computer science concept, we’ve finally come out the other end and are now able to start talking about loops more in-depth. So, that raises the question—what is a loop? What relevance does this have to us as a programmer?

If you have to ask that question, then you aren’t quite thinking like a

programmer yet, and that's okay, it will happen. Remember that programming is the absolute abstraction of everything that a computer does, as well as the abstraction by extension of everything that the *users* of a computer do. Understanding loops in this way will massively help you in becoming a better and more able programmer capable of writing, well, better programs.

So, bearing all of that in mind, what is a loop? A loop is a method of repeating something over and over under some predefined conditions. This may seem a bit silly or unnecessary at first, but stay with me, because you'll quickly understand why it's necessary.

Let's think about something where we use loop logic but we tend not to *think* about the fact that we're using loop logic. Let's take, for instance, writing a word on a piece of paper, the old-fashioned way.

So, you start out, you think of your word. Thus starts your loop. The first thing that you do is to think about what the first character of the word in question must be. After that, you put your pencil to the paper and you draw the character. After that, you pick your pencil up. So repeats the process until the word is over. On every iteration of the loop, you'll be checking to see if the word is indeed over, and if not, then you'll repeat the process of finding the next character, writing it, and picking up your pencil all over again.

See, while putting it in proper terms as a loop may come across as a bit daft, the simple truth is that loop logic is pretty much everywhere and we tend to vastly underestimate just how much of an impact it really has on our day-to-day lives. As a result, we also tend to underestimate how much of a prevalence it will carry in our programming, despite the fact that loop logic is a *massive* part of programming.

Loops in programming manifest in two primary ways, each with extraordinarily different purposes: while loops and for loops. In Python, the different purposes

of these two loops are more clearly delineated. However, if you look at C++ and Java or languages similar to those, it's much harder to immediately tell as a new programmer which functions both of these loops cater to. Even though it's a little bit clearer, I'm still going to give a fair amount of explanation on the difference between these two different forms of loop logic and how they work so that you as the programmer can have a bit of a better understanding of that.

First, let's focus on the while loop. The while loop is relatively simple conceptually. All that the while loop does is verify whether or not something is true on every iteration of the loop. If it does happen to be true, then the loop will continue. If it doesn't happen to be true, then the loop will terminate and the code will move on to the next part.

Conceptually, the while loop is simple. However, the actual use of the while loop can be a bit trickier. We'll talk about that momentarily, though. For right now, let's just think about the syntax of the while loop. The while loop can be written like so:

```
while expression:
    # loop logic inside, indented once inward
```

So, let's say for example that we wanted to count from 1 to 5.

First, we create our variable:

```
i = 0
```

Next, we create our loop terms:

```
while i < 5:
```

Next, we put our logic within our loop:

```
while i < 5:
    i += 1 # increment i by 1
    print(i) # print i out, whatever its value is
```

On the fifth run of the loop, the number will increment to 5. After the fifth run is finished, the loop will evaluate its condition. It will see that i is now equal to 5 and therefore i is no longer less than 5, and as a result, it will exit the loop. It's a fairly simple implementation, but it's pretty fantastically useful.

The truth is that the while loop really isn't the best for working with data in this way. The for loop is better suited to a purpose like this. The while loop has a very peculiar but very much needed purpose to serve: you use the while loop when you aren't sure how long your loop is going to need to run, plainly.

Some things aren't verifiable. User behavior is one of those. There are other factors too. Sometimes random aspects will be an influence on your program. When you're not sure how many times your loop will need to run or how long something is going to take, your best bet is really to use a while loop in order to effectively "wait it out".

It's for this reason that while loops are commonly used in what's called the *game loop*. The game loop is one specific implementation of the while loop that is rather unique, and its name is quite fitting.

The game loop isn't *only for games*, but it's quite often *used* in games because of the structure thereof. Games are simple. The same process will be repeated over and over until somebody either wins or loses. When that person wins or loses,

the game is over. However, this lack of predictability of when the “when” will finally come lends the while loop to being the perfect candidate for the implementation of this sort of thing.

The game loop is essentially a while loop which is checking a boolean variable on every run. This boolean variable may be something like hasWon, and the loop may be set to run for as long as hasWon == False. When hasWon == false, then the loop will be terminated, of course, and the next thing in the code will happen —the next thing normally being some kind of end game screen, like a “Game Over”.

The variable will be set to false before the while loop starts, and that falsehood will be checked every time the loop completes. Within the logic of the while loop, there will be one (or multiple) win conditions that will actually change this variable so that it’s set to *True*. When the variable is set to true, the loop will go to evaluate its expression and say “Oh, well that variable isn’t true anymore, so clearly this loop is finished.” Then the game will move on to whatever comes next.

Of course, as I said, a game isn’t the only example. This also could be a great model for something like a main menu in a program where the exit condition would be the user entering the word “Exit” or the number 0 or something along those lines. The conditions can vary, but the implementation of the game loop is almost always the same.

Now, let’s shift our focus for a second to the *for loop*. The for loop runs parallel to the while loop because it serves the opposite purpose. The for loop is intended for *iteration*, and in particular, iteration across a set of data to which you already know the size and extent. This is the main purpose of the for loop and is the niche which it fills.

This also, it may be clear now, why we need to spend just a little bit talking about things like lists. Lists and for loops go hand in hand because for loops give you the functionality to actually work through the data contained within lists.

For loops are composed of two essential parts, at least in Python: an iterator, and something to be iterated over. The iterator is a value that is established in the declaration of the for loop which takes on the identity of whatever the current value in the range being iterated over is. If it's a string, then the iterator becomes a string value just like the one in the set. If it's an integer, then the iterator becomes an integer. The iterator value can have any name that you want.

The thing to be iterated over can vary. Much of the time, the thing to be iterated over will be something along the lines of a list or set or dictionary. Other times, it will be something else—a string, for example, can have every word within it iterated over and separated by spaces. With that said, one can also define a numeric range using Python's built-in *range* function. This function serves for the definition, as I said, of numeric ranges. Two numbers are given—the start value of the range and the end value of the range. If only one is given, then the range will start at 0 and will consist of everything from 0 to that number.

A for loop is therefore defined like so:

```
for iterator in iteration:
    # code goes here, indented once inward
```

It's not terribly difficult. So, let's say for example that we wanted to print every name in our list that we just established. How could we do that?

```
for name in names:
```

```
print(name + "\n")
```

Again, there's not a whole lot to it, but it is extremely useful knowledge to have, and as time presses on, you'll find that you use it more and more especially in things like file input and output.

Methods

This is one of the last things that we really need to cover in this book, and it's a pretty essential part of programming in general. This concept is *methods*. Methods go by many different names. The other most common name is *functions*. Admittedly, I started programming on languages that preferred the term *function*, so it's the term that's nearest and dearest to my heart. Forgive me if I slip up and use it instead of method.

However, they are functionally (haha) the same thing in the end. They both offer the same support structure to your code and allow you to create a much cleaner set of code in general. So, with all of that said, what exactly *is* a method?

It's probably easier to understand methods if we first refer to them by their old name, *functions*. Functions is the term preferred in older languages like C and C++ that have functional or procedural programming paradigms, with method being the term preferred in object-oriented languages and scripting languages.

The term function betrays the origins of the concept; think back to your high school and college math courses where you probably worked with the functional form f(x) = y. What was happening when you worked with those? What was the central idea behind them?

The central idea, of course, was the connection between the concept of *f(x)* and *y*, wherein *y* was the result of the function *f(x)*. In essence, *y* was the value that

the function f(x) gave back after everything was computed. *x* was the argument that was given to the function *f(x)*, and was thereby the thing that was manipulated within the actual context of the function. If, for example, we were to write out the function f(x) = 2x + 4 = y, then *x* would be the variable that was manipulated in the function obviously. f(2) would indicate that the function was 2(2) + 4, meaning that *y*, the final value of the function, would be *8*. This was the overall value of the function, or generally described as such at least.

You'll notice that functions offer a means of abstracting an equation. This is pretty much their function in programming as well. But we'll go more into that in a second. There are many similarities between mathematical functions and functions in computer science. For instance, the function that we just dealt with takes a single argument. Functions in computer science take arguments as well. They also don't have to take any at all! Functions in computer science can take 0, 1, or multiple arguments and they will work perfectly fine.

These arguments, much like *x* was manipulated in the function *f(x)*, can then be manipulated in the body of the function in order to yield a certain result. This certain result can be seen as the *end value* of the function, much like *y* was the end value of f(x). Programmers, however, don't really prefer this term; they tend to prefer the term *return value*, which is the value that the function *returns*. Just like a function in math can only give back a singular end value, a function in programming can only give back a singular return value. Moreover, just like a function in math can be undefined, a function in programing doesn't have to give back a return value at all! One could easily make it to the end of the function and have nothing returned and everything would be peachy keen. This is the nature of functions in programming.

So, what good are functions in programming then if they have such a peculiar use standard? They can take many arguments, or they can take none at all. They

can give back data, or they can give back none at all. What gives, exactly?

Well, the simple answer is that functions in programming aren't just confined to the purpose of giving back information. Rather, functions in programming fill the particular role of breaking up the code. If something is reused repeatedly, you should put it in its own method and then just use the method throughout your code as needed.

Let's say for example that we wanted to standardize finding the area of a rectangle. Throughout our code, we actually do this a lot, and it's unwieldy. We want to replace all of this code with one simple way to just give the dimensions of the rectangle and be done with it. So, how can we go about doing this?

Well, the easiest way is to define a method! Defining methods is extremely simple in Python. You define them like so:

```
def function_name(arguments, if you have any):
    # code within
    return value # if there is a return value
```

So, in order to write a function that would compute the area of a rectangle, we just do the following:

```
def areaRec(l, w):
    return l * w
```

This will feed back the value of the given length multiplied by the given width. You can then do whatever you would with this value that you would with another value. For example, you can store it to a variable or print it out to the console.

```
myArea1 = areaRec(3, 7)
myArea2 = areaRec(5, 4)

print(myArea1)
print(myArea2)
```

See how simple and intuitive this all is? Functions offer a great way to break your code into reusable chunks if you know that something is going to happen repeatedly. It's one of the best ways that you can make your code forward thinking. The idea of multiplying length and width to find the area of a rectangle is so small that you're probably thinking, "well, you could just do that with the multiplication and save time", and yes, that's true; however, this is purposefully simple in order to illustrate the idea behind methods and how they can be used. It's not supposed to be incredibly difficult. The purpose of methods will make itself clear in the future when you're working with more advanced programming paradigms.

Object-Oriented Programming

Object-oriented programming is the last major thing that we need to cover in this section of the book before we start looking a little bit forward. It's a massive part of any programming paradigm and any large body of code is going to include object-oriented programming to one extent or another. The extent to which they do varies with the program, but most programs will benefit to some degree from the implementation of object-oriented programming paradigms.

That raises the question, then—what *is* object-oriented programming? Well, that's actually a rather complicated question to answer. Object-oriented programming itself is a *way* of programming. It's more of a programming paradigm than anything else. And yes, there are ways of programming. A language can be more object-oriented, more functional, more imperative, and so forth. These little terms actually define a whole lot about the programs and the way that they work and are written.

However, enough of that—let's focus on what *object-oriented programming* is, and what it means. Object-oriented programming has a long history behind it. It's been in vogue since the 80s, and has come to be increasingly more popular as time has borne on. One of the reasons it's gotten more popular is one of the same reasons that Python has gotten more popular—computers are getting stronger and becoming more capable. Your code doesn't *have* to be extremely simple and efficient anymore. Some performance can be spared for the sake of writing code that is easy to maintain over the long term and easy to build onto. For many companies, the ability to do these things may even be seen as more important as the ability to write efficient code in the first place, since it will save them money over the long run to have easy to read, extend, and maintain code.

This may make it sound like object-oriented programming is inefficient. That isn't necessarily the case. What object-oriented programming is about is using several different philosophical ideas in order to actually build up abstract concepts about programming in general, and then implementing these philosophical ideas *in* your programming in order to make it easy to work with over the long term.

The first major language to be released with object-oriented programming in mind and as its raison d'être was Java in the early 1990s. When the Java programming language was developed and released, so was done with these philosophies in mind: abstraction, encapsulation, polymorphism, inheritance, modularity, and extensibility.

These are easier to understand if you understand object-oriented programming in a really basic way first, though. So let's examine object-oriented programming in the abstract. Object-oriented programming is about exploring and standardizing the relationships between things of common properties. That's the easiest way to put it. It's also about expressing the relationship between classes and objects.

Let's start with classes. Classes are the fundamental cornerstone of object-oriented programming. While there are many object-oriented languages that don't necessarily have classes as their cornerstone, the class *model* nonetheless is used.

Many things are similar to one another. When things have something in common, you can group them according to their similar properties. Let's start with mammals. What do all mammals have in common?

Well, all mammals give live birth. They all produce milk to feed their young. They all have backbones. They all have a name, too, which is pretty important.

They all have a certain number of legs, generally 4.

Using that, we can start building off of our basic idea of what a mammal is and use this in order to define a standard set of properties that every mammal *has*. Let's make a list:

Mammals have:

- Backbone
- Name
- Species name
- 4 legs
- They can produce milk

Perfect, that's a fantastic start. So knowing that we have all of these, and these are the standards by which we classify mammals, we can safely create a *class* of mammals that have all of these properties. Easy enough, right?

A class in programming is an abstract definition of a structure which uses many smaller data pieces in order to define something bigger.

You can define classes in Python like so:

```
class ClassName(inheritor):
    def __init__(self, args):
        # initialization data
        # defines properties when the object is created
```

So, in order to define a Mammal class, we could do this:

```
class Mammal:
    def __init__(self, name, speciesName):
    self.name = name
    self.speciesName = speciesName
    self.legs = 4
    self.hasBackbone = True
```

You can also define methods in your class that all objects will be able to do.

```
class Mammal:
    def __init__(self, name, speciesName):
    self.name = name
    self.speciesName = speciesName
    self.legs = 4
    self.hasBackbone = True

    def produceMilk():
    # logic goes here
```

This is an important starting point, but nowhere near everything that we need to discuss about this. The next thing that we need to talk about is the idea of *objects*. Objects are manifestations of classes. Essentially, imagine your class as the *definition*; objects are the actual *things*. Objects are created and defined like variables. Let's say we wanted to define a new Mammal, the lion:

```
lion = Mammal("lion", "Panthera Leo")
```

With that, you've created an object. See how simple it is? Objects can actually perform any functions of their class.

```
lion.produceMilk()
# lion starts producing milk
```

As you can see, the relation between classes and objects is relatively easy to understand, but still well worth taking the time to fully understand it all in context.

So, with all of that out of the way, it's time that we start focusing on stuff that is a little bit more pressing. What *is* object-oriented programming? What are the ideas? Well, we can give a fair definition of object-oriented programming by analyzing the first core philosophy of object-oriented programming: *abstraction*.

Abstraction is the idea that things should be simpler, more abstracted from the raw hardware of the computer, more difficult to mess up, and easier to understand for *humans*. Computers, as we discussed earlier, haven't always been easy for programmers—or anybody, for that matter—to understand. It's been a very gradual process of making computers easier for the common man to really start to get, and it's been slow at times.

Therefore, abstraction is at the core of object-oriented programming. This is the ability to make simple interfaces and things that are easy to work with and understand in simple English over long periods, as well as create abstract infrastructure for your programs, is essential to the long-term development of programs. You can see it as a bit of a programmatic Keynesian tradeoff. Sure, on one hand, the system expenditure may be a little bit more and code may be a little bit more verbose at first. However, it pays off in the end because of the ease of maintenance. Also, the more code that is added to an object-oriented system, the less that you need in the future. This is because you're constantly abstracting concepts, and the more you abstract things, the easier it is to build cohesive systems out of those abstractions.

Consider how we worked with functions earlier. Functions may, in one way or another, be seen as a kind of abstraction because you're taking code and making it extremely reusable. In this, you can instantly see how abstraction can be a good thing. Instead of constantly having to redefine the same thing, you can define it once as an abstract concept and you're done.

You can also see in plain sight the idea of *encapsulation*. This is one of the other primary considerations behind the development of object-oriented programming paradigms. The idea of encapsulation says that like things should be put together. Things that go together should be kept neat and tidy to keep everything cleaner and better put-together. This is a chief consideration of object-oriented programming, and you can see this in the steady and simple way in which properties of classes are put together.

The next thing that we need to talk about is the idea of *inheritance*. Inheritance is yet another core concept behind object-oriented programming, and the reason why will become extremely plain as we work with it all. Inheritance is the idea that some things are innately connected and derive from other things. As a result, you should be able to build hierarchical systems where one thing can derive from another thing and can derive, in turn from another thing and so forth until we reach the very bottom rung of the ladder. The truth is that the world exists with a surplus of these sorts of hierarchical paradigms, so it makes sense that we should be able to express them in our programs.

When you implement inheritance in your object-oriented code, you're basically recognizing that one class may take on the traits of another. This is true. Let's look at our Mammal class. Sure, this is great and all, but can't we subdivide this even further?

For example, dogs are a *mammal*, and when you're defining dogs, surely you want to take in all of the properties of the mammal class. But that also won't cut it. Dogs have a few more properties than mammals that you might want to define. Let's say, for example, that all dogs bark, which means we need to define an additional method. Let's say also that all dogs have a breed, which is an additional specification that mammals *don't* have. Additionally, when you define a dog, you already *know* it's a dog, so you don't need to give the name and the species name. See what I'm saying?

So, dogs clearly inherit from the Mammal class in a pretty innate way. Now we just need a way to express that in our code. You can express these inheritance relations by citing one class as a base class when you define your function, like so:

```
class Dog(Mammal):
    # code within
```

So, let's actually define our dog class so you can see up close and personal how this would work.

```
class Dog(Mammal):
    def __init__(self, breed):
    self.name = "dog"
    self.speciesName = "canis lupus"
    self.breed = breed

    def bark():
    print("Bark!")
```

Now, you can define new dogs with many properties already defined for you,

and you only need to state its breed:

```
sammy = Dog(“Sammy”)
```

See? It’s super simple. Moreover, because it’s an inherited class, it can actually do everything its parent classes can do:

```
sammy.produceMilk()
    # valid command
sammy.bark()
    # Bark!
```

I hope that it’s beginning to become clearer what role inheritance has in any given object-oriented programming paradigm.
The next thing that we need to talk about is *polymorphism*. Once we’ve talked about this, we’ve actually hit 4 out of 6—hooray! Polymorphism is the idea that things can take on multiple different versions depending upon how they’re called. This might seem like common sense but believe it or not, polymorphism wasn’t considered too often before the advent of object-oriented programming.

The most common form of polymorphism is *method overloading*. Function overloading is when you have two methods of the same name that have different arguments. Since they have different arguments, they actually can be called and run in different ways. Consider, for example, a method which finds the area of a square if given one parameter and a rectangle if given two:

```
def findArea(length):
    return length * length

def findArea(length, width):
```

```
        return length * width
```

It’s a simple idea, but it really makes your code easier to read and understand on the human end. It’s intuitive and makes a lot of sense.

The next thing that we need to discuss is the idea of *modularity*. Modularity is a huge part of object-oriented programming and is one of the primary reasons behind any and all of this. Modularity is the idea that things can actually be broken down and detached. When you make your code modular, what you’re trying to do is take it and make it so that it can be broken down into chunks and those chunks can be reused, or used in different contexts. Additionally, when your code is modular, it means that making a change to one module doesn’t mean refactoring all of your code. While this is inevitable in some cases, generally, code that’s really well-written won’t carry this burden. The best modular code is modular in such a way that its individual modules may be changed and all of the other modules will keep functioning. In other words, every module is a gear in a bigger machine.

The last thing that we really need to discuss about object-oriented programming is the idea of extensibility. Object-oriented programming is heavily based around the idea that your ideas should be able to be easily exported and used as a module in somebody else’s code, should they have the right framework for it. This is, in an essence, the resultant inverse of modularity. Code should not only be modularity, it should be able to be added to with ease.

All of the different object-oriented aspects are about writing code which ultimately is easy to come back to and maintain over a very long period of time. While it’s not always as simple as this, for the most part this goal is graciously reached by conscientious programmers. So what place does object-oriented programming have in Python?

Well, the truth is that Python isn't really built for huge projects. However, when these projects do come around, they're nearly always written in a manner that is at least somewhat object-oriented. While Python in a way is more innately supportive of procedural programming paradigms, it offers an extensive amount of support for object-oriented paradigms as well. When you're writing a script with a complicated structure that you expect yourself to be coming back to a lot, then object-oriented programming offers an obvious solution to a problem that presents itself as a little complicated in nature.

Closing

With that, we've conquered the longest chapter in this book. So, why did we spend so long discussing all of that in a book about Raspberry Pi? Simply put, Raspberry Pi is a tinkerer's tool. Your ability to work with and fully understand Python code is paramount because it's the language most often used on the Raspberry Pi, as we discussed earlier in the chapter. While it's beneficial to learn and work with other languages in addition to Python, Python is by far the most common.

The simple fact is that knowing how to program in Python opens up what you can do with your Raspberry Pi a pretty huge margin. You no longer are confined to just working with other people's code. Instead, you can write your own projects, make your own gadgets, and do your own thing. You can also make meaningful changes to other people's code in order to have it suit your own needs (though don't pass their code off as your own, this is terribly bad form.)

In the chapter to follow, we're going to be discussing where to go from here after discussing everything that we've covered so far.

Review Questions

1. What is Python? Why is it ideal for use with the Raspberry Pi?
2. How do you set up Python?
3. What are the different types of data you can use in Python?
4. What are arrays? Why is their use discouraged in Python?
5. Why are comments important?
6. What is casting?
7. What is object-oriented programming? How can you use it effectively and efficiently?

Chapter 5: Where to Go From Here

So at this point, you probably have a couple of good questions. We've worked our way through all of the complicated stuff and even learned how you can start programming in Python, which is really instrumental. From here, though, what can you start doing? Where can you really go from here? This chapter is going to focus on finding a satisfactory answer to that question, because that question is admittedly really, really hard to answer in its current form.

The simple reason that it's so hard to answer is that there is no straight answer. It all depends on what you want to do. However, there are a number of things that I could recommend you do, so perhaps I'll start with that.

First off, I would heavily recommend that you find some communities dedicated to Raspberry Pi in general. This will be a massive asset to you going forward because you'll be around people who like the same things you like. Doing so will give you a lot of ambition to keep pushing forward with programming on the Raspberry Pi in general and trying to make things happen.

I personally would recommend that you find a project that genuinely excites you and then spend a long while working on that. You're going to probably inevitably run into hiccups along the way, and when that does happen, you can take advantage of the fact that you have your community there to ask questions to. This is where the community starts to come really in handy; when things go awry, they can steer you in the right direction. It's impossible to overstate how important this is to you as a beginner learning to program, especially on architecture so unique as the Raspberry Pi's.

This has yet another benefit to it, though it's a bit hidden: you're also going to

witness the mistakes that others make. The simple fact is that when we mess up, it tends to stick with us. We see things done right all the time, and when something is done wrong, it sticks out more than when everything is done right. It's just basic psychology, really. However, when you do something wrong and are corrected, or when you see somebody corrected about something they're doing wrong and it's relevant to you in some way or another, you're going to be more likely to internalize what went wrong and how to fix it. This is extremely important when you're learning to program.

The thing is, though, that you're not just learning to program. You're here because you heard about the Raspberry Pi, and you wanted to learn about all of the things that you could do with it. The fact is that if you spend long enough looking at all of these cool projects, you're going to eventually come to a point where you have some inspiration and want to make something of your *own* that you can be proud of and say that you did completely of yourself. When this hits, having the feedback net of a community is going to be extremely important.

Along with all of that, I would recommend that you try to take your programming further in general. No programmer is single purpose and no project s only good for itself. During programming a given project, you always learn things that not only apply to the project at hand but to projects in general, and being able to get that varied knowledge and diversify the code that you're working with, you'll be exposing yourself to many more concepts. For example, even though I have a particular interest in the Raspberry Pi projects that are specifically related to natural language processing and open-source voice parsing, I wouldn't have gotten into these particular interests or gotten to know a lot of what I know about them without the additional context of me working on other projects. And that's just the long and short of it: a good programmer is a good programmer, and a good programmer is never a one-trick pony. While they may specialize in one thing or another, programming is built on a whole host of

skills that they inevitably work with and build otherwise, like knowing where to look in order to find information, how to read APIs, how to implement code, and how to know what's worthwhile and isn't. Along the way, you also learn a lot about other different concepts, like biology in the case of working with automatic plant watering, linguistics in the case of working with natural language processing, and general computer science regardless of what you're really working with.

As a result, spreading yourself out across multiple different ideas and multiple different disciplines is a generally great idea because it allows you to build a solid sense of what you're working with in all respects as well as build discipline over time as a programmer. You also need to be taking pains to work with other people's code as a new programmer. This is the best way to learn the best conventions for programming in general. Every language and community has its own programming conventions. For example, the Python community focuses heavily on making its code Pythonic, while the Java community has more of an emphasis on pragmatic and pretty code. There are even more subdivisions within all of this. But in the end, getting exposure to all of these different influences and learning more about the different ways of programming in general will be a massive boon to you as a programmer, which will in turn be a massive boon to you as a Raspberry Pi tinkerer.

This book places a huge emphasis on programming, but the fact of the matter is that programming is absolutely central to working with the Raspberry Pi if you ever want to make your own projects or even update somebody's project that's gone out of date. Without that essential knowledge, you're worthless in terms of what you're able to do on the Raspberry Pi. I hope that that makes a bit more sense in terms of why this book cares so much about you learning to program!

Final Project: Python Game

To end this book, here's a simple Python project you can undertake to enhance your Python skills and consequently get better at programming your Raspberry Pi.

The game you will be programming is an RTD program—roll the dice. Here are the mechanics for the game:

- The number generated must be random. Hint: You will use a certain module for this. Make sure to do some research!
- The minimum number should be 1. The maximum should be 6.
- There must be user input which asks whether or not the dice should be rolled again.
- Whenever the dice is rolled, it outputs a different number between the minimum and the maximum.

Have fun programming this! It's actually pretty simple, so good luck!

Conclusion

Thank you for making it through to the end of *Raspberry Pi*, let's hope it was informative and able to provide you with all of the tools you need to achieve your goals whatever it may be.

We've already discussed where to go from here. Now, I just want to give you some last words of encouragement. You've specifically taken up a path of frustration, and you've done so with intention. Keep that in mind as you go forward and inevitably get frustrated with code not working or your electronics just not working the way that you want them to.

Sometimes, you're going to want to pull your hair out. This is completely and totally normal, and it's all a part of the process of learning. You're not going to be an amazing programmer from day one, and you certainly aren't going to be setting up perfect hardware configurations if you have no background in it. Remember that you just have to keep pushing through and learning.

Be humble in the way that you approach everything. Don't be afraid to admit that you're wrong or that you aren't as capable as what you want to think, and ask for help on one of the communities that we talked about seeking out and joining in the chapter prior. If you can keep doing this, then you'll eventually come out the other side smarter and hopefully with a working gadget.

The Raspberry Pi is altogether one of the coolest pieces of technology out there right now, and it can do so much, but it can also be really frustrating to the programmer because it can do exactly as much as the programmer is able to make it do. Don't let this discourage you; let it *encourage* you. Understand that if you try to start programming on the Raspberry Pi and doing super cool

projects, it’s going to get difficult sometimes. You just have to keep pushing through and trying to become a better programmer.

In the end, hopefully you’ll create something seriously cool of your own. When you do, you should post it on one of the communities you joined and get feedback. People will probably be supportive and tell you how neat it is, and if you happen to be really, really lucky, you just might end up inspiring somebody to pick up a Raspberry Pi and try to do exactly what you’ve done. And then down the line they’ll repeat the cycle.

Well, that’s that. A ton of work has gone into making this book as applicable and useful as possible to people who want to learn about the Raspberry Pi and all of the ridiculously cool things it can do. If this book was able to help you in understanding the capabilities of the Raspberry Pi as well as to better understand computer science in general, then please, leave me a review on Amazon!