Matthew Teens

CODING FOR KIDS

{2 BOOKS IN 1}

Python and Scratch 3.0 Programming to Master Your Coding Skills and Create Your Own Animations and Games in Less Than 24 Hours

Matthew Teens

CODING FOR KIDS

{2 BOOKS IN 1}

Python and Scratch 3.0 Programming to Master Your Coding Skills and Create Your Own Animations and Games in Less Than 24 Hours

Coding for Kids: 2 Books in 1

Python and Scratch 3.0 Programming to Master Your Coding Skills and Create Your Own Animations and Games in Less Than 24 Hours

Matthew Teens

Book Description

Scratch is the ideal introduction to programming for children of all ages! This step by step guide will teach kids the fundamentals of programming and how to create a variety of projects using Scratch 3.0.

Coding for Kids in Scratch 3.0 is an educational book that provides a solid understanding of common coding techniques and concepts that can be later applied when learning other programming languages like Python.

Kids will learn that programming is an exciting, creative activity, which can be fun to learn when using the most popular coding tool for children.

Start by gaining an understanding about how programs work and learn about other programming languages. Not all languages are created equally, and this book will give you a summarized explanation of how they work.

Next, learn the basic programming principles with step by step explanations using Scratch. This guide will show you how to install Scratch and how to set up your development environment. The sooner you start coding, the better.

What else is inside this book?

You will learn how to program by working on real projects. Create graphical elements, manipulate audio effects, create a story book, animate sprites, and develop games!

Computer coding for kids has never been easier or more accessible. Add *Coding for Kids in Scratch 3.0* to your collection and begin your programming journey today!

Coding for Kids in Scratch 3.0

A Step-by-Step Beginners Guide to Master Your Coding Skills and Programming Your Own Animations and Games in Less Than 24 Hours

Matthew Teens

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaged in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of the information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

Table of Contents

Introduction

The desire to learn how to code has blown up in recent years. Everywhere in the world schools have started adding programming classes, even for kids as young as seven years old. Coding clubs and bootcamps started popping up everywhere. More and more jobs require coding skills and people have even started learning how to program just for fun. There are so many devices and tools available to us that we can do so much if we know at least one programming language. Programming is becoming a new must-have skill.

All of this means that it was never easier to learn how to code than it is today. Back in the early days, even before the Internet, programmers had to do everything manually. Their tools couldn't check how correct the code was, couldn't automatically detect errors, and a lot of fundamental computer processes had to be managed by the coder. A single misplaced symbol was enough to give the programmer a headache. Today, all you need is an intelligent tool that handles the boring stuff for you so that you can focus on creating something. One such tool is Scratch. Scratch is a visual scripting language, also known as a drag and drop programming language, and we're going to focus on mastering it throughout this book.

A lot of people think that programming is difficult, it requires math, and some special talent to be able to master it. None of that is true! Programming can be both easy and fun! Scratch is all about making the learning process so fun that you forget you're actually dealing with programming concepts and techniques. It allows you to be creative; it gives you all the tools you need to create art, animations, and cool games to share and play with your friends. If you are a complete beginner, you'll discover a whole world of possibilities.

So let's jump in and start learning about coding. Build up your skills from scratch (pun intended!) and learn the basics of programming through practical projects.

Part 1: General Coding Principles

Chapter 1: Programming Basics

Computers are used for everything. Your favorite games, movies, art, and animations involve the use of computers. Chances are, even your toaster or fridge involved the use of a computer to program all of their smart features. Computers are used in so many ways and by learning how to program, you'll discover a whole new world of possibilities. Let's explore this new realm by first learning how computers think and how programs work.

How Do Programs Work?

Everything that is powered by a computer runs on a sequence of commands. You can look at a program as a long collection of instructions. You might think of computers as something much smarter than us humble mortals, but in fact they're quite dumb. Computers need to be told EXACTLY what to do, otherwise they're useless.

Think about the process of making a sandwich. If I tell you to go make yourself a peanut butter and jelly sandwich, you don't need any more details. You know what that means and how to do it, and even if you aren't quite sure, you can figure it out. But a computer can't do that. It's not enough to tell it to go make a sandwich. You need to tell it to first open the cupboard, fetch the peanut butter jar, open the lid, take a knife from the drawer, and so on. Even ordering it to use peanut butter isn't enough. Every single detail needs to be explained step by step. That's how dumb a computer is, and that's why it needs a programmer.

Programmers use their logic and creativity to write all the commands in an orderly sequence. This is what a program is, except that the commands can't be written in plain English because the computer doesn't understand it. That is why a programming language is used.

But, most languages can't be understood directly either. Computers speak in machine code, which is formed by nothing but long sequences of zeros and ones. So, the programming languages we use are first translated by an interpreter or a compiler so that the computer can read the code. Many languages work like this, such as Python, C, C++, Java, and Ruby. Even Scratch works the same way, though it differs from the other languages because we use visual blocks to create the program instead of writing the code ourselves.

Can Anyone Be a Programmer?

YES!

You decided to start out with Scratch, a visual, building-block based programming language instead of languages that involve writing code like Python or C#. Those languages might feel intimidating because they're much harder to understand, but that doesn't mean you need some special ability or to be a mathematical genius to eventually learn them. Everyone can learn any programming language. So if you're worried that you might not have what it takes, know that with time you can learn other languages and progress. Anyone can be a programmer with enough dedication and desire to learn.

Even if you're not interested in solving complicated problems and developing world-changing software, you can still use programming to focus on your creative side. This knowledge can be used to create cool animations and visual effects, games, interactive stories, virtual reality simulations, and so much more. Don't worry about cold logic and boring math! Programming is pure art. You just need to think in a logical way and in an organized sequence so that you can create a clean program that runs smoothly. This is a skill you will gain in time with practice. That's all it really takes. Practice!

The Process

Before learning the principles and techniques behind various programming languages, you need to understand programming itself. You need to know the process behind everything. The easiest way to look at a program is by imagining it as a recipe. All types of software are created using a collection of components that are connected to each other in various ways. These elements communicate and are processed in a certain order, just like how a recipe contains a collection of ingredients that are used in a certain order to create the final product, your meal.

When you design a program you need to think about its purpose, the same way you "design" your lunch. Then you create it by following a specific sequence. You don't start your breakfast by first eating your eggs and then toasting your bread, right? You first toast the bread and prepare everything on the table. Programs are created through a series of steps, just like your lunch. And just like

with a recipe, you have to have all the instructions with precise details. You can also compare programming with LEGO builds. Place one block on top of another until you have something awesome!

To understand a program and to easily plan it, we can break it down into multiple sets of actions. In programming terms, these sets are called algorithms. Every algorithm has a specific goal, such as the action of toasting bread before setting the table. As a programmer, your main job is to create such algorithms because they're the LEGO blocks of all projects. To better understand algorithms, you need to follow these simple principles:

1. The sequence: All instructions are processed in a specific order, otherwise they don't make much sense. This order is known as a sequence. Code is usually processed and executed by the computer system from top to bottom. This means that the first algorithm at the top goes into action before the second algorithm that follows it. When you program a game character to take damage when hit, you first program the animation and the conditions related to being hit. Afterwards you program the adjustment to the player's health bar, which is the result of being hit. If these two blocks of code were written in reverse, it wouldn't make much sense, would it?
2. Decision-making: This is where programs start being different from cooking recipes. Typical programming languages allow you to give your project options. Instead of just writing a set of instructions that say "Toast two slices of bread," you can also apply conditions and let the program decide what to do. That way the program can, in a way, think on its own and toast the bread only if it's 7 AM and not if it's dinner time. This opens an entire world of possibilities because you can open so many paths. Just using this example, you can also tell the program to completely ignore your bread toasting algorithm if it's Wednesday and instead run the "pour milk on cereal" algorithm. With decision-making abilities, a program can execute only certain actions if the conditions are true. If not, then those actions will be completely ignored even if they are already in your code.
3. Repeatable processes: Algorithms can be automatically repeated as many times as necessary. In programming, we don't want to declare the same algorithm or block of code multiple times throughout our project because it adds up unnecessary lines of code and it can slow down the program. Instead, we use loops to create repeatable actions.

This way we can tell the program how many times a certain action should be restarted at the end of its execution or we can apply a condition that says “repeat this action while this condition is true.” Loops are a crucial building block in all programming languages, including visual ones like Scratch.

These concepts apply to all programming languages. Make sure to go over them until you really understand them and remember them, because they’re the backbone of our future programs and games.

Chapter 2: Programming Languages

Every beginner wonders what programming language to learn and which one is the best or the easiest to start out with. These are normal questions, especially because there are hundreds of programming languages out there. However, out of that impressive number, only about a dozen of them are extremely popular, and only a handful of them power most programs and games you use. That still doesn't make the decision any easier. Some recommend Python, some say C++ is still the best for beginners and experts alike, and others choose to go with a visual scripting language like Scratch.

So what's the best option? There isn't one. It depends on each person. Some want to start off easy because they're intimidated and a bit afraid. Others dive in because that's what their favorite company uses to program their favorite games or software. In either case, it doesn't matter. What matters is that the language allows you to focus on the theory of programming so that you can quickly understand the main concepts. It's difficult to understand how programs and games are made when you constantly have to fight with complicated code that takes a lot of time to remember and understand.

Like the title of this book says, we'll focus on Scratch because it doesn't involve writing code and therefore you can quickly learn how programming works. All you need to do is focus on how programs are structured and how the building blocks like conditionals and loops are built. That doesn't mean you'll always stick to Scratch. You'll eventually get bored of it and seek out a new challenge. So, let's talk a bit about what other programming languages are out there for you to explore.

C

This old programming language might be from the 70s, but it still powers a lot of software today. C is the base language of several other languages, including Java and C++. By learning C, you'll automatically understand how other languages work as well.

However, this is quite a tough language to learn and it can be quite unforgiving for a beginner. C is a low level language that is very close to machine language, so it's not as easy to read and understand as Python or Scratch. Nonetheless, it's

a language that you should consider in the future after you master Scratch. C is mostly used to develop computer operating systems like Windows. It is also used when working with a lot of hardware. For instance, most NASA space probes and satellites rely on this language because it's powerful and extremely fast when operating.

To give you an idea about how C looks, here's a simple program that displays the words "Hello, world!" on the user's screen:

```
#include <stdio.h>

int main ()

{

        printf ("Hello, world!");

        return 0;

}
```

It's probably impossible for you to understand everything that's going on, but this is just to give you an idea about the readability and friendliness of a programming language so that you can compare it with others.

C++

This language was built using C as the foundation, so it's very similar in some ways. Due to additional features and expansions added, C++ is used when creating complex software, as well as games. If you dream about working for a huge tech company like Microsoft, you should one day learn C++. Or maybe your dream is to become a game developer. In this case, C++ is a good option as well because companies like Blizzard and EA Games use this language to create their games.

C++ is a difficult language, so you should leave it as your second or even third language to learn after Scratch and maybe Python. This language let's you control every basic system function manually and that brings a lot of challenges. The more control you have of the underlying computer functions, the easier it is to make mistakes.

With that in mind, here's our "Hello" program coded in C++:

```
#include <iostream>
int main ()
{
        std: : cout << “Hello, World!”;
        return 0;
}
```

The structure of the language is quite similar to C, but there are already some important differences in the execution of the code. C++ is a tricky beast, but it can still be learned even by kids if they have some general knowledge about programming. That’s what you need and what you’re going to get by learning Scratch.

C#

This general purpose programming language is similar to C++ in many ways but it’s also very close to Java. In fact, many beginners who look at some code from Java and C# can’t distinguish between the two of them. While C# (C-sharp) derives from the more complex C++, it’s actually a much easier language to learn and work with. Its code is very English-like and easy to read, and all of those low level system processes are automatically handled. It’s a user-friendly language that you can easily learn with some basic experience with programming principles and concepts.

C# is used in a variety of industries because it’s powerful and versatile. It’s used to create complex applications, but it’s also appreciated by game developers. If one of your dreams is to create games, then C# might be the best step to take after learning the basics of programming and how to work with Scratch. There’s a massive online community of game developers that create amazing things using C# and a free game engine called Unity. The two of them allow you to create high quality games, as well as animations and virtual reality projects.

Here’s how the code looks when recreating our “Hello” program in C#:

```
using System;
namespace Hello
```

```
{
        class Hello
        {
        static void Main (string[] args)
        {
        Console.WriteLine (“Hello, World!”);
        }
        }
}
```

Compared to C and C++, the code is easier to read even though the structure is still the same. All the keywords make sense because they’re just English words and not shortcuts, abbreviations, or unique terms. You only need to learn a bit of theory to know what a class is, what a namespace is, and what words like static and void mean. Once you learn that, you can easily understand the code.

Java

This general purpose programming language is in many ways just like C#, so if you eventually learn one of them, you’ll also be able to understand the other. However, unlike C# which is great for game development, Java is mostly used to create various applications. But it’s still one of the most popular languages out there, so if you’re interested in making cool web or mobile apps, you’ll want Java in your arsenal. Just don’t confuse it with JavaScript. The names might be similar, but JavaScript is used mostly to create websites. Programmers like to say that Java is to JavaScript what a car is to a carpet.

Java is worth learning because it’s very much in demand, so finding a job later in life wouldn’t be that difficult. It’s used to power most online applications. With that being said, let’s take a look at our Hello program written in Java code:

```
public class hello
{
        public static void main (String[] argos)
```

```
    {
        System.out.printIn (“Hello, World!”);
        }
}
```

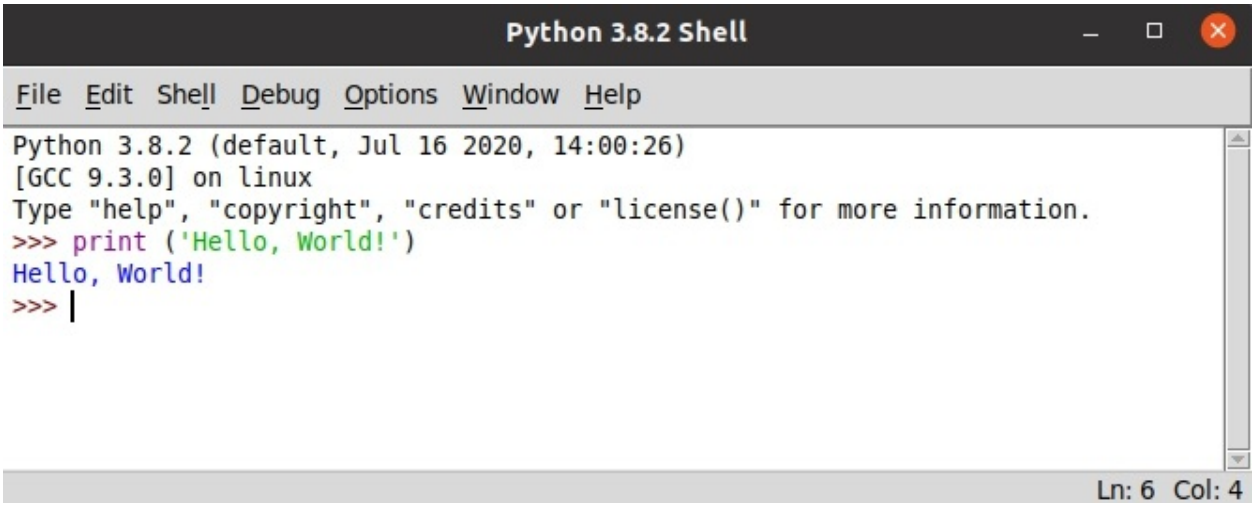
See how it’s barely distinguishable from C#? You might even say it’s a bit simpler. It still uses curly braces to mark the blocks of code, and most keywords are the same.

Python

This is a programming language you’re going to hear about a lot. It’s probably one of the most often used programming languages in schools because of its simplicity, but it’s also great for serious projects. Python is recommended to every beginner, especially after learning how to program using a visual scripting language like Scratch. It’s powerful and easy to learn. That makes it the best option as a first proper coding language.

One of the main reasons why Python is easier to learn is the fact that it’s an interpreted language. This means that we don’t have to declare variables and various parameters the same way we have to when coding in C++, Java, C#, and other such languages. Python recognizes the values we’re using and understands what kind of data type they are, so we don’t have to keep track of them manually. This makes Python code a lot shorter and easier to read and write. Here’s how our Hello program looks in Python:

```
print (‘Hello, world!’)
```



That’s all we need! Just a single line of code and it does the same thing all those weird looking commands did in C++ or C#. You don’t even need to understand terms like “static” or “namespace” for this. It’s basically plain English. You tell Python to print a few words on the screen, and that’s what it does. You don’t even need to be a programmer to understand this statement.

Python is a great starting point in the world of coding because the rules are the same as in Scratch, but the actual code isn’t as scary as in C++, for example. It’s also a multipurpose language, so you can easily use it to create cool animations, games, programs, or whatever you want. Even more importantly, there’s a massive online community of kids just like you that study Python and share their projects. You can easily make new friends and learn how to work in a team with people your age. You can also find a lot of answers to most of your questions so you can have a much easier time learning and fixing any bugs and errors that are giving you problems.

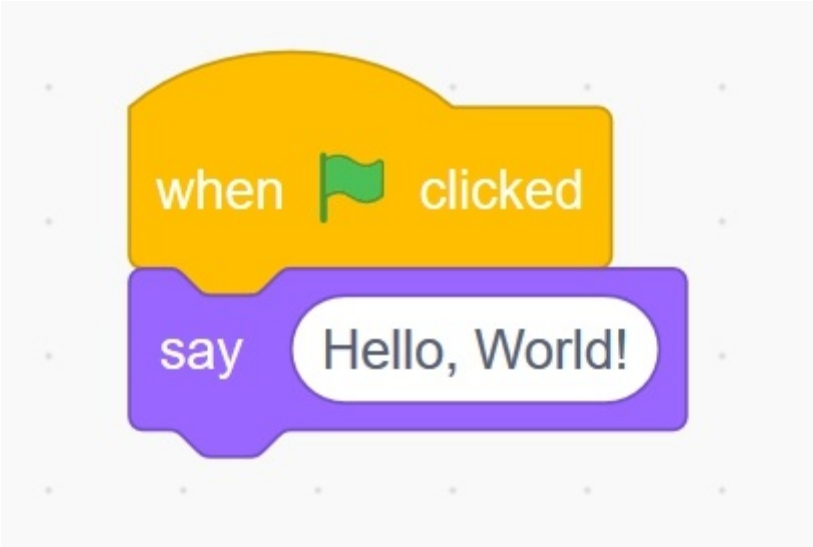
Scratch

This programming language is different because it doesn’t rely on writing code directly, but instead connecting premade building blocks. In other words, Scratch is a visual programming language. We take blocks of code that are visually represented and neatly color-coded and connect them to each other to create a program. It’s like assembling a puzzle.

This makes Scratch an ideal way of learning how to program. Instead of focusing on how to write code and on learning all the keywords, you can focus on the logical structure.

Scratch let’s you immediately create cool projects while teaching you the foundation of all programming languages. It even comes with a collection of graphical elements and audio effects so that you can create cool games and share them with your friends or other programmers like you.

Here’s how the “Hello” program we created in the other programming languages looks in Scratch:



We’re going to discuss Scratch in great detail throughout this book, but for now, notice how we have two connected building blocks. Each block fulfills a function. The first one says that when we click something (in this case the flag representing Play), the program will “say” or print the words “Hello, World!” on the screen. It’s all very simple and easy to understand, even for someone who never programmed anything in their life.

Scratch is probably the most beginner-friendly coding software used in schools and programming classes for kids. It’s a great entry into the world of programming, even if you don’t want to become a programmer. Scratch is a good way to help you understand how computers work and how to think logically and critically. Learning how to solve problems is one of the most important benefits you get out of working in Scratch.

Chapter 3: Coding in Scratch

As mentioned before, Scratch is a visual programming language preferred by kids and beginners because it allows them to easily enter the tech world. But Scratch isn't just for people who want to become programmers. It's much more than that.

Here's what you can do with Scratch:

1. You can learn how to program. Even though you don't have to write code, you still need to understand what a conditional statement is, how booleans work, what variables and functions are, and what kind of data types you're dealing with. All of this applies to any programming language. Scratch just allows you to visualize everything and learn without being distracted by difficult code.
2. Learn mathematical concepts. If you like math, or even if you don't, you'll find that Scratch gives you the tools needed to visualise certain math concepts. For example, you can learn the x y z mathematical coordinates system interactively using Scratch's graphical system. This way you can get better at math because some of us understand these concepts much easier when we can have them in front of us.
3. Creativity! You can turn anything you can imagine into a Scratch project. Got a favorite story? Start recreating it in Scratch with graphics, animations, sound effects, and music. Scratch is a great tool for those that like to read, write, and just create stories. You can also take this to the next level by creating an interactive story where the player makes the decisions when interacting with the characters.
4. Create games and share them with your friends and family. If you love playing games, you might like to also create them. Nothing's more satisfying than bringing your idea to life. Scratch has everything you need to create rich, story-driven games or arcade games. You can play them with your friends or share them with other fellow Scratch-users using the online Scratch platform. You can build a team with other people and create something together. Scratch enables you to experience what game development really feels like, but in a fun way.
5. Learn how to think and learn better. You will never stop learning no matter what you want to do. Through Scratch, you will learn how to think logically, communicate with other people, design projects,

analyze problems, discover solutions, research, and work with others as a team. These are all crucial skills that will give you a head start in life.

Scratch is a school on its own, but maybe more fun and relaxing. It teaches you how to think and how to express yourself while having a good time.

Programming Concepts with Scratch

Before we start diving in, you should learn the basics of the most important concepts and principles you'll encounter in Scratch. Most of them apply in other programming languages or applications as well, so let's go through them briefly so you can understand the fundamental building blocks of programming:

1. The interface: Whatever program or game you design, you need to focus on the way the user interacts with your creation. The interface is what connects the user or player with the project and the actions that can be taken.
2. Booleans: In programming, we have boolean logic that analyzes to see if something is either true or false. A statement can have only one of these two results. In Scratch we're going to use boolean commands to check when a condition is true. For example, we can check if the background color is blue or if the value of 10 is greater than 5 (if the player's chance to hit the monster is greater than his dodge value, for instance).
3. Variables: Information can be stored inside variables. We create them to assign them numbers or text. For instance, we can declare the "x" variable and say it's equal to 10. Now whenever we use x in the program, the program knows that it's supposed to use the value of 10.
4. Lists (or arrays): Sometimes called arrays, they work just like variables. We store information in them. The difference is that we can manipulate that information and change it when needed. Just think of a simple shopping list when you hear the word "array."
5. Events: In Scratch, you'll find a category of code blocks called "events." They are used to tell the program when to perform a certain action. When something happens (an event), an action will be performed.
6. Broadcasting: You can program a sprite (a graphic element) to

receive information from another sprite and then act in some way. Broadcasting messages allows our sprites to communicate with each other.

7. Random numbers: You can specify a range, a minimum value, and a maximum value, and then tell the program to choose a random number in between.
8. Data storage in the cloud: Scratch provides us with cloud variables that let us store some information online, in the cloud. This is very useful when building a score system to keep track of all the scores earned by you and your friends.
9. Functions: Sometimes known as procedures, functions sort of work like variables, except that instead of storing values, we can store lines or blocks of code. In Scratch we can create a function by naming one and then setting up a collection of blocks under it. By doing that, we can later just use the function without specifying everything it contains.
10. Vector graphics: Scratch isn't just a programming tool. It also contains an image editor that lets you create sprites (graphics) or edit existing ones.

Most of these features are the same for other programming tools, such as Python. When you start working in Scratch, make sure to come back to this list when you need to use events or functions for a quick memory refresh.

Setting Up Scratch

The latest version of Scratch, Scratch 3.0, can either be installed on your computer and used offline, or it can be used online without installing anything. The simplest option is using the online version by navigating to https://scratch.mit.edu/ and clicking the "Join" option to set up your personal account. You can also jump straight in without an account by clicking on "Create" instead, but you won't be able to save or share your projects that way. You should join the platform so that you can unlock all the features you need to learn and create. We'll discuss this part in more detail in the next section.

Alternatively, if you don't like online applications or you can't always be online, you can download the desktop application that's available at https://scratch.mit.edu/download . The offline editor is available for Windows,

Mac, Android, and ChromeOS. Choose the correct version based on the OS used by your computer, tablet, or smartphone, and follow the installation instructions. If by any chance you have a Linux-based computer, for instance one that runs on Ubuntu, you should just use the online editor. There's no out of the box Linux installer available on the website.

Once the editor is installed, launch it and you're ready to get started. Since your work will be done offline in this case, you should remember to frequently save. All you need to do is click on "File" and select "Save As."

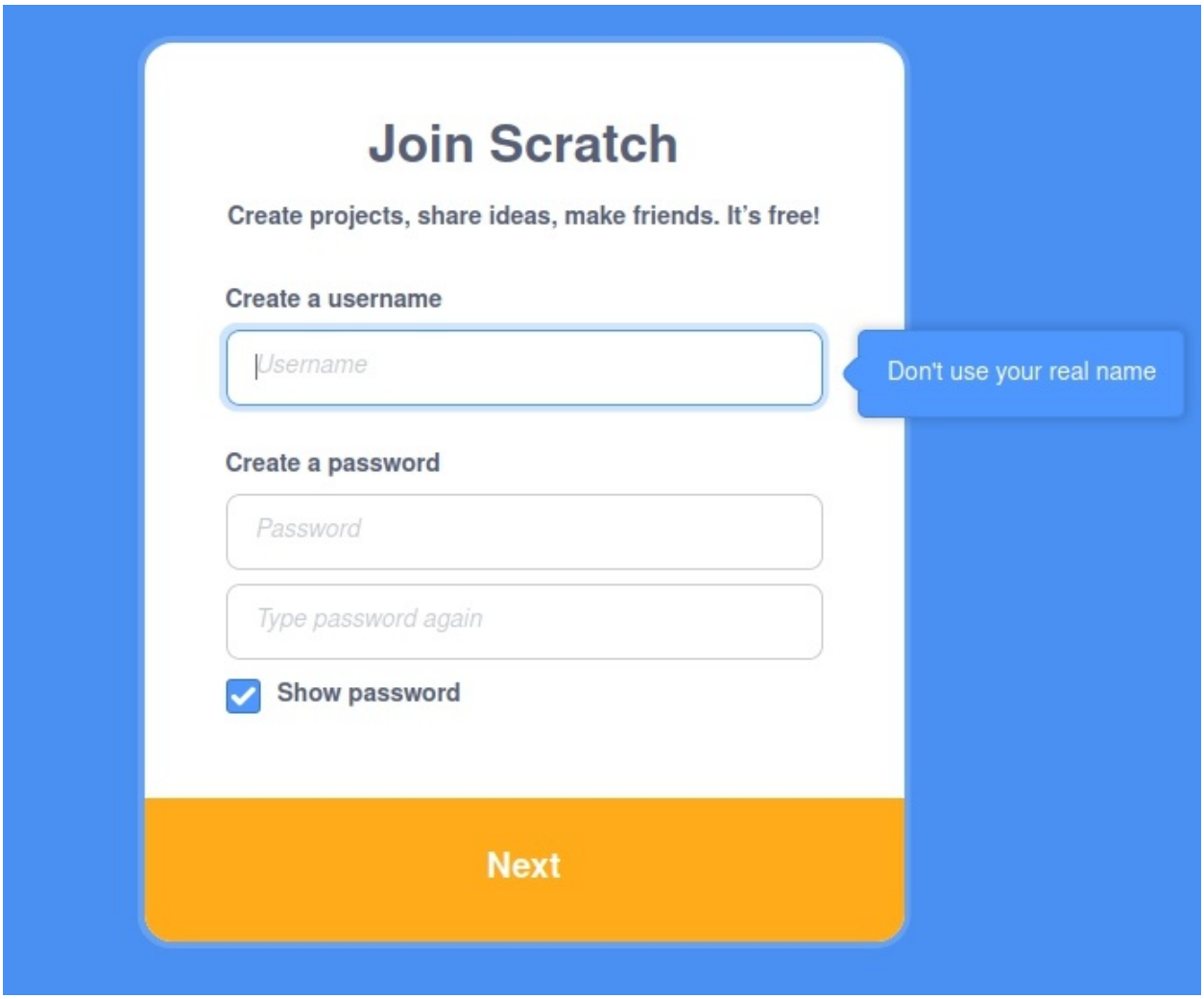
If you created a Scratch account as well, then you can sign in from the homepage, and start creating. All of your work will be automatically saved as long as you're logged in. Remember, that if you click on the "Create" button without logging into an account, your progress will not be saved and you'll lose it as soon as you close your browser. When you're logged in you can find a "My Stuff" button where all of your projects are stored.

Account Setup and Options

While the offline editor can be useful in some cases, the online editor comes with a lot more advantages. You'll be able to log into your account from anywhere and have access to your projects, you'll be able to easily share them with anyone in the community, and you'll have access to the community forums where you can ask questions and discuss various topics.

So let's go through the account creation process and see what kind of options you have once the account has been created.

Create the Account



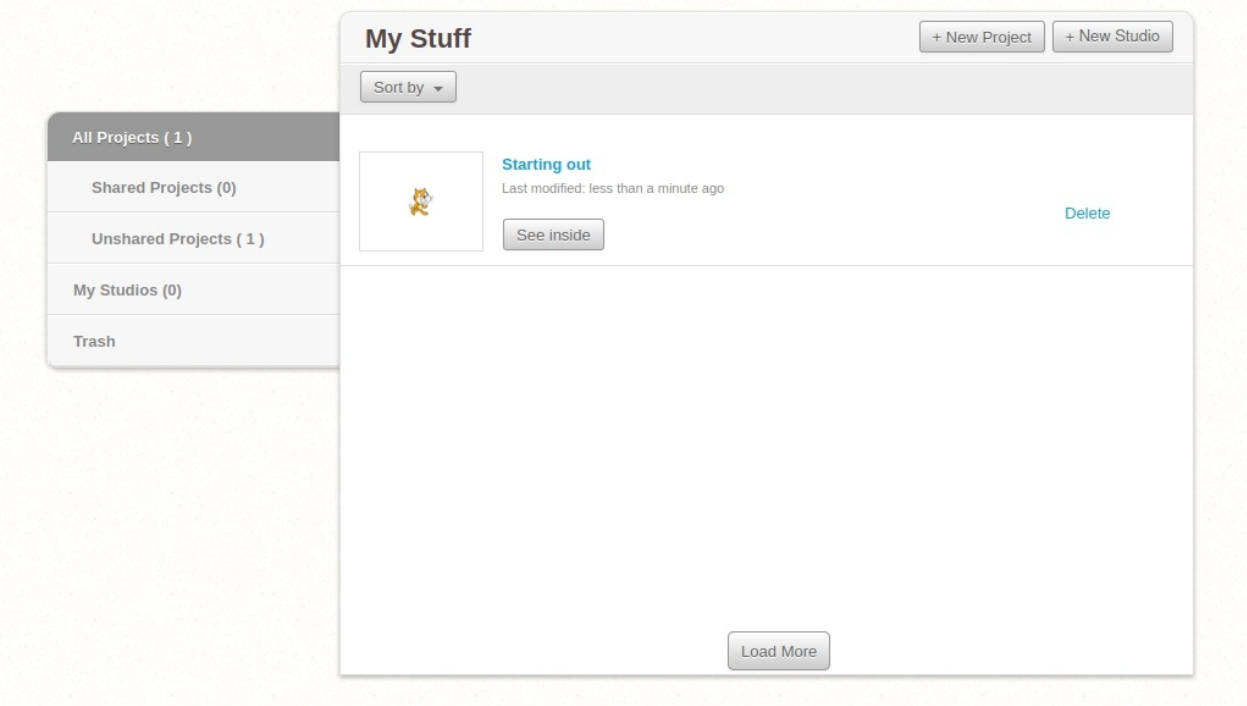
Once you click on the Join Scratch button, this panel will open up. You'll have to create a unique username and a password that you'll remember, as you can see in the image above. In the next panel, you'll have to select the country you live in, followed by your birthdate, and other personal information. Keep in mind some of this information is going to be kept private and will not be shown to anyone. Privacy is important and Scratch only uses your date of birth, email address, and other personal information only when needed to reset your password or confirm your identity. Finally, you fill in your email address and click on the Create Account button.

Account Features

Your user account comes with four important features: saving your projects, receiving notifications and messages, sharing your projects with others, and

participating in a large online community.

Now you can hit the “Create” button and unleash your imagination. All of your projects will be automatically saved. You can find your work by navigating to your account options by clicking on your username and then selecting the “My Stuff” option.

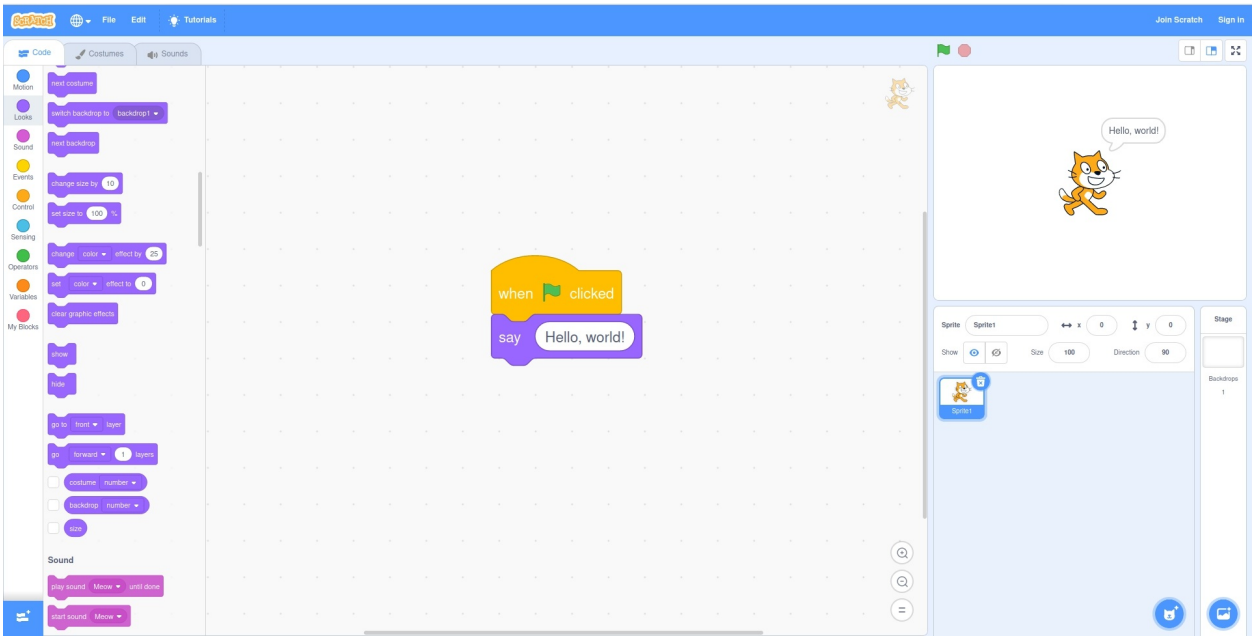


You will also notice that when you’re on your project page you have a new button called “Share.” You can share your project with all fellow scratchers. By default your project is private and can only be seen by you. But if you share it, others can take a look at it, leave comments and suggestions, or even take it themselves to make changes and modify it. If anyone leaves a comment or does something with your project, you’ll be notified in your Messages panel that’s represented by a letter icon.

Finally, you can now go to the Scratch forums to learn new things and talk to people. You’ll find many ideas there and answers to your questions. You can even find other more experienced Scratch users willing to help you out when you’re stuck with something. Don’t be too shy to reach out to others. Programming and development are a team effort most of the time.

The Interface

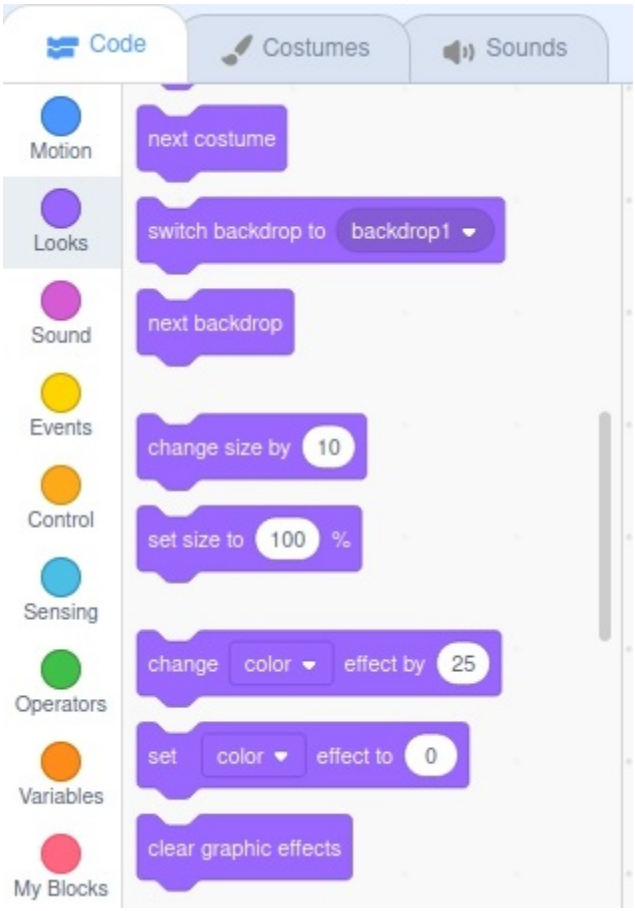
Let’s start exploring the Scratch interface by creating that “Hello” program we saw in all those programming languages we talked about. Here’s how it looks together with all the interface options available:



At this point, you’re probably wondering why we keep creating this boring, useless program that just prints some words on the screen. The reason is simply tradition. This is how every single programmer is initiated into this new world. This is the first program everyone creates no matter what programming language they use.

With that out of the way, spend some time exploring the interface. There are quite a few panels, options, windows, and buttons to play with.

The Code Block Pane



This is where you'll find every single code block you can use to create your project. As you can see, they're color coded into different categories, such as: Events, Control, Operators, Variables, Sound, and more. To use them, all you need to do is click on a code block and drag it to the Script Area, which is the large, blank window in the middle.

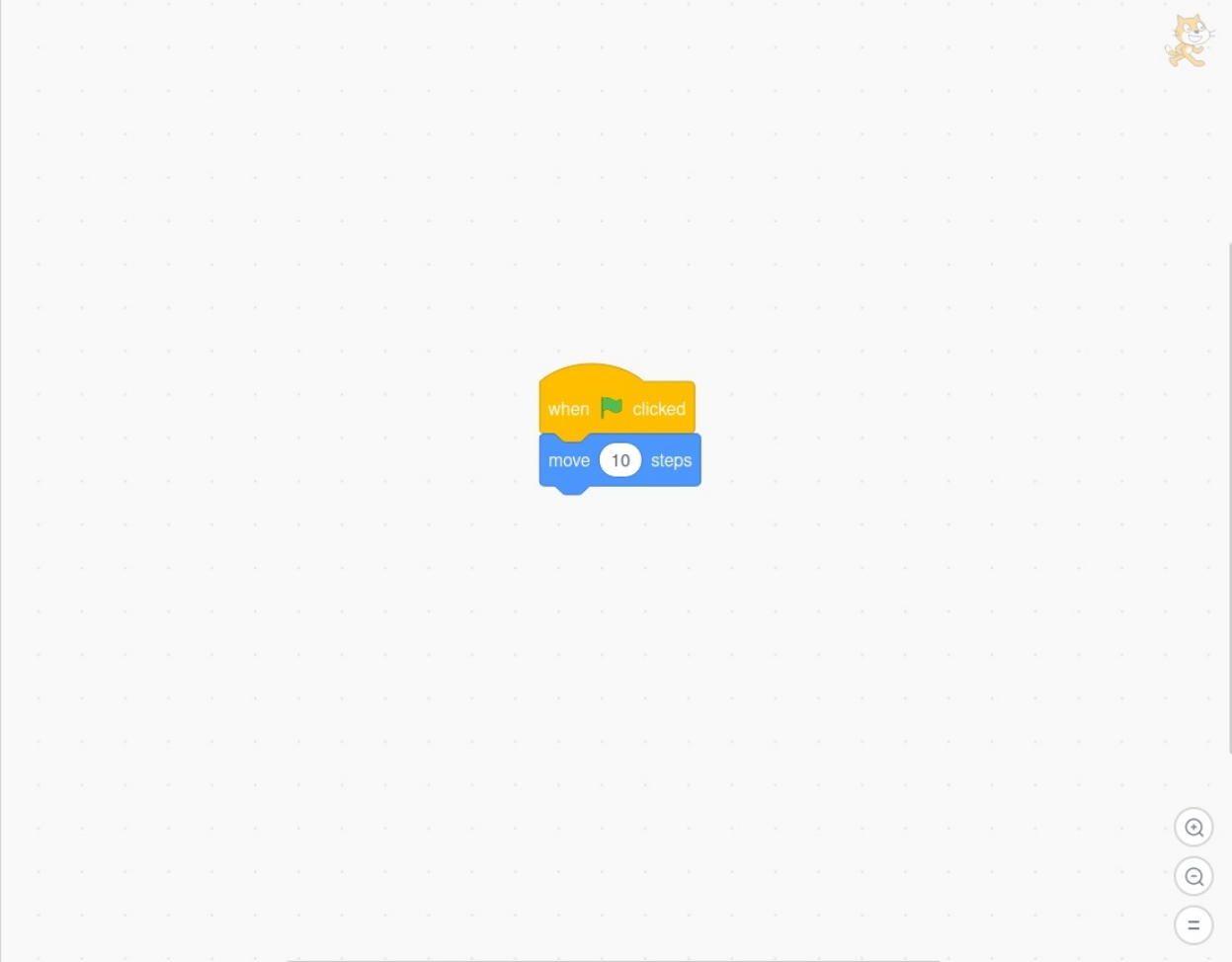
To create our Hello World program, we have to go to the Events code blocks first. That's where we find "When Clicked". It makes sense, doesn't it? After all, the action of clicking something is an event. You'll also notice that this event has a green flag. In Scratch, that little green flag translates to "Start" or "Play," meaning that by clicking it your program or game will run.

Next, we go to the "Looks" code block category where we find the "say" block. In Scratch, "say" is the same as "print" in Python, for example. It prints a message on the screen. Together, the two code blocks simply mean "When Play clicked, say this message." Simple and logical.

Before moving on, explore the categories and the code blocks. Try to figure out what kinds of commands and controls you have in each section. They will all

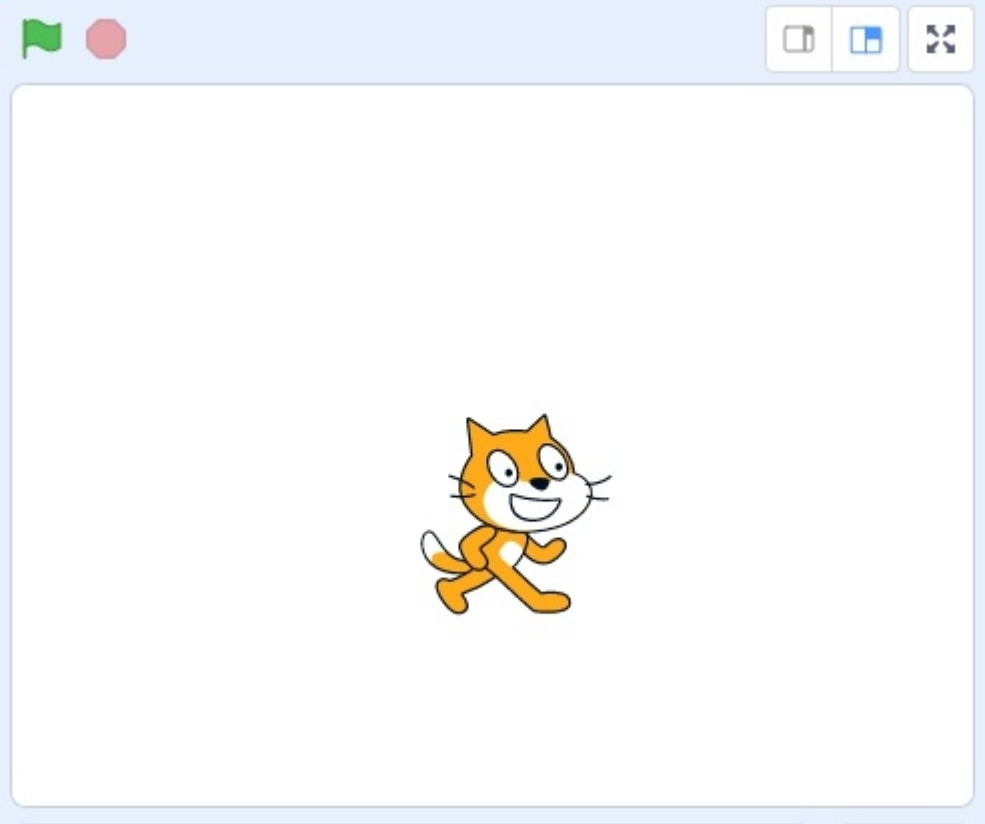
come in handy sooner or later.

The Script Area



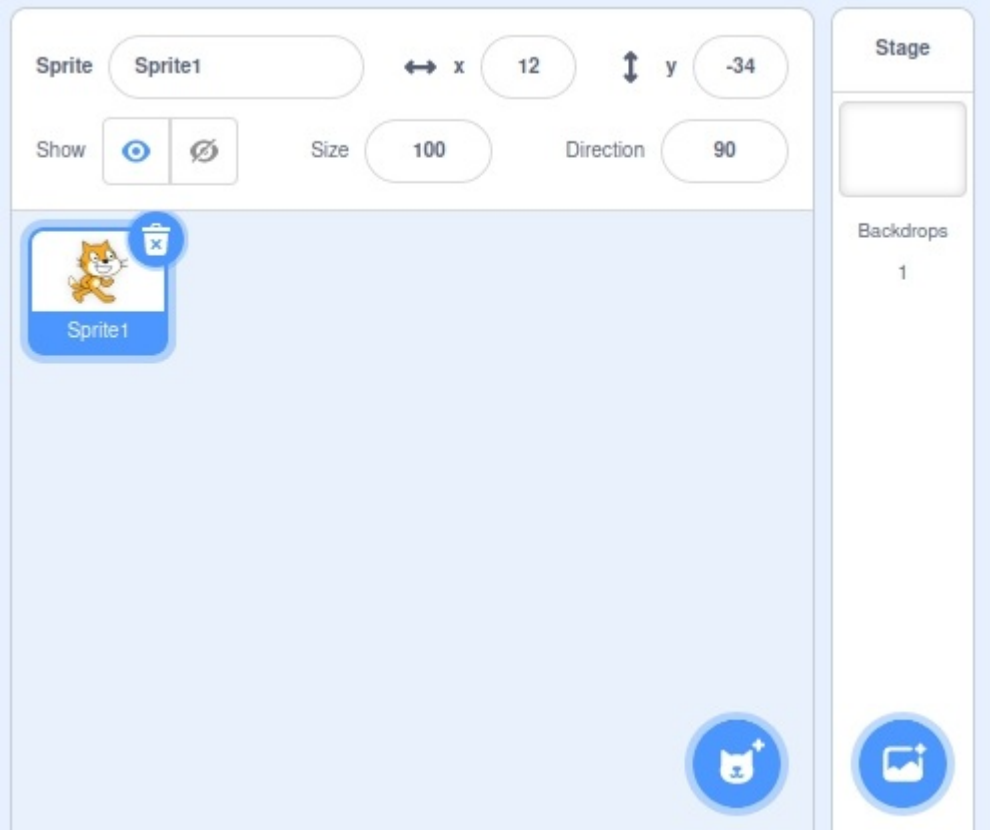
This is where all of your code goes. Drag and drop any code blocks here and link them together. You can also create multiple blocks that aren't connected to each other. It's simply a window where all of your code goes. You can navigate around by dragging the window, and zooming in and out. Use those navigation controls to order your code neatly so that it's clearly visible.

The Stage Area



This is where all your hard word comes to life. Notice the green flag icon. It's the same as the one in our "when clicked" event. When you hit that Play button, your project will run in the window and do whatever you programmed it to do. You also have a full screen option to extend the size of the display. When you're done testing something, you can click on the red stop sign to stop your code from running.

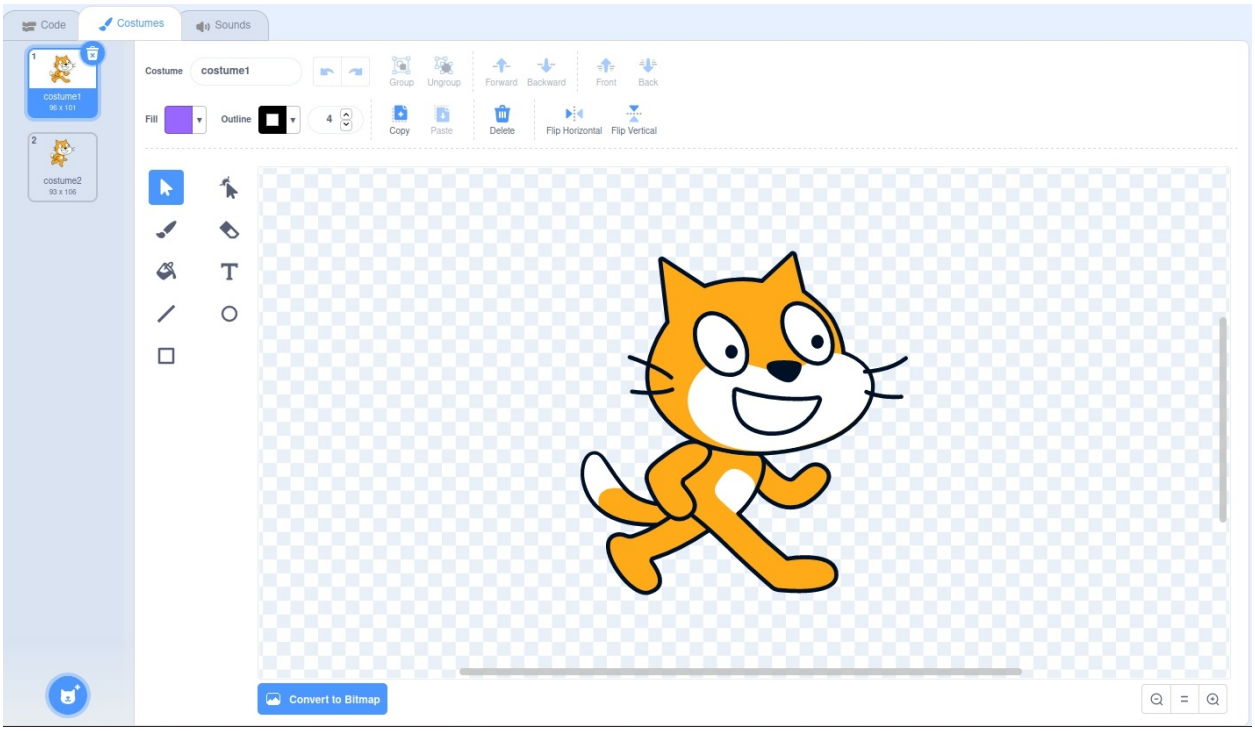
The Sprite Information Window



Next up we have the sprite info pane right under the stage area. This where you'll see all the information about your project graphics and a variety of options that allow you to manipulate it.

You can create new sprites, edit the ones you already use, change the background, upload a sprite you found on another website, and much more. Select the sprite you want to modify and change its size, location, or direction. Just keep in mind that any changes you make apply only to the selected sprite and not the others you might have in this window.

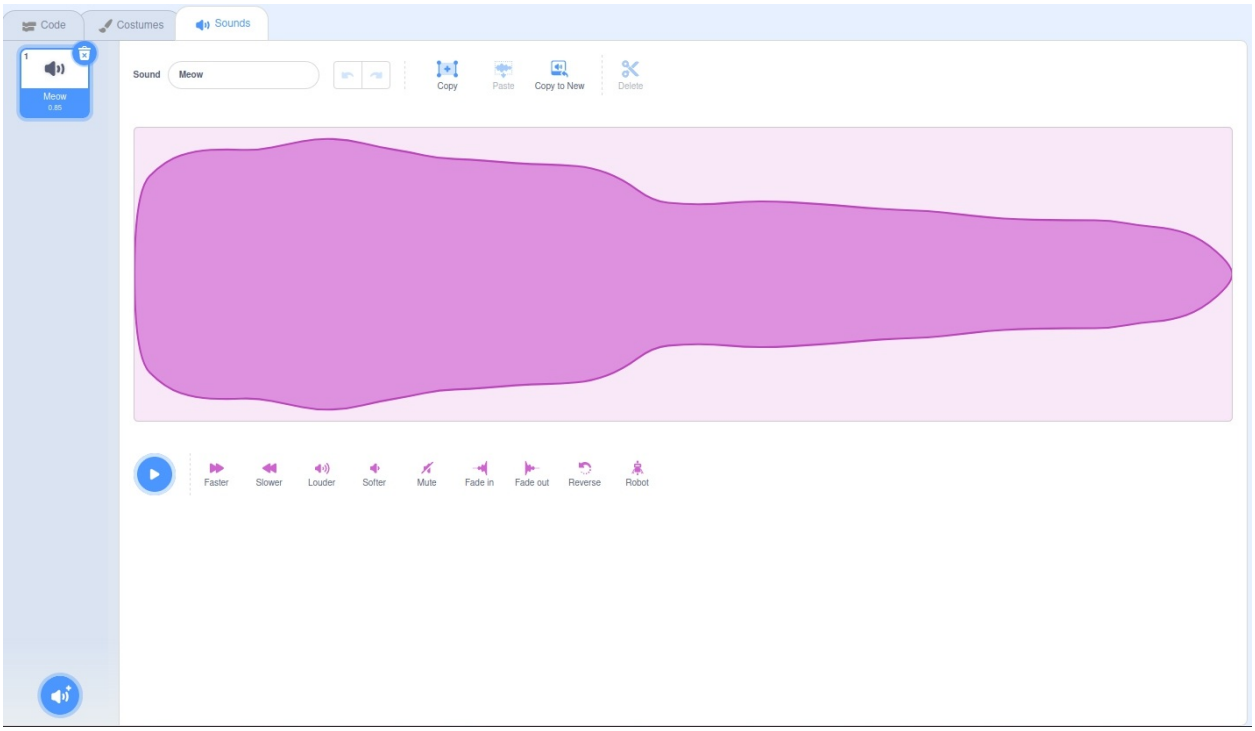
The Costumes Window



This window looks a lot like Paint, but it's even simpler than that. The costumes panel is where you have more control over how you create and edit your sprites. You have various drawing tools such as filling a shape with color using the bucket tool, outlining it, adding text, and more.

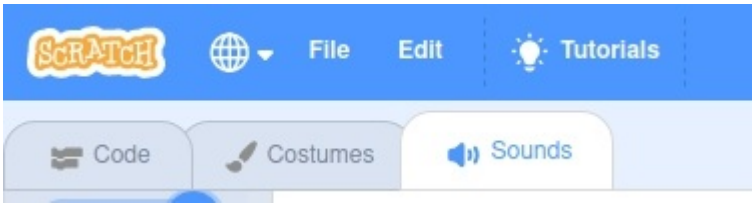
Here you can also upload sprites from open source art sites, like Open Game Art at https://opengameart.org/ , and modify them however you want. Spend some time in this window and create a few things you might use in a game later on.

The Sounds Panel

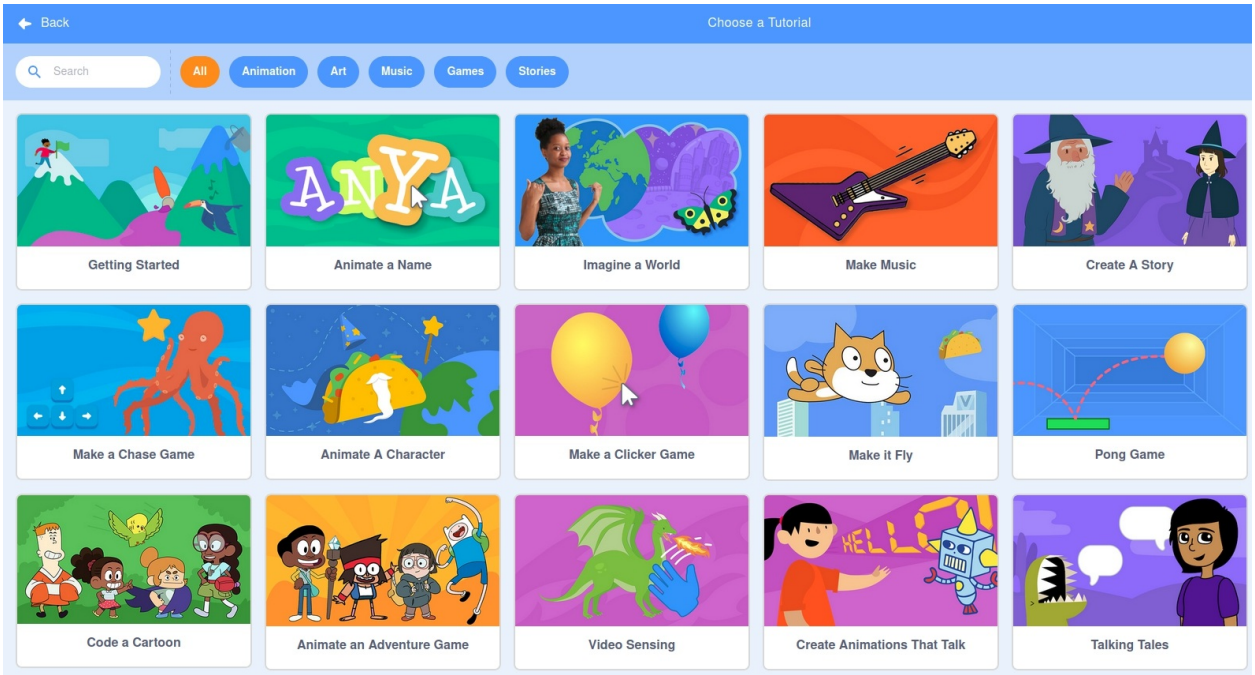


Finally, next to Costumes, we have the Sounds panel. This is where you can find various sounds and edit them however you want. You can also upload your own recorded sounds and manipulate them by speeding them up or making them fade in and out to achieve a smooth effect. Programming isn't just about writing boring code, so put your voice actor hat on and start recording some sound effects for your projects.

The Toolbar and Tutorials



Take note that at the top of the screen there's also a small and easy to miss toolbar. This is where you have options to save and load your project, or undo a mistake you made. But more importantly, you have the Tutorials page. This is extremely useful for a beginner because it contains a collection of video tutorials on anything related to Scratch.



As you can see in this image, there are a lot of tutorials about making animations, music, games, and cartoons. Use them to get new ideas for your project and reinforce what you learn from this book.

Building and Running your First Script

Let’s start creating our first project from start to finish. We already had a couple of examples, but when you start with a new bit of software you might want a detailed explanation of the entire process. Staring at the white, empty script area can be intimidating.

Start by pressing the “Create” button to open a new project. Notice that every new Scratch project already has a sprite ready for you. That’s the Scratch cat and it’s all we need for now.

We’re going to make the cat move across the stage area when we hit the play button (the little green flag). To do that, we need two things. First, we’re dealing with an event, the action of clicking “play.” So go to the code blocks palette and select “Events.” There you’ll find the “when green flag clicked” event. Click on it and while holding the mouse button down, drag the block to the script area.

The second thing we need is the ability to move. Movement is a motion. In the code blocks palette we have the Motion category that contains all the movement

related code blocks. Go there and select the “move (10) steps” block and drag it right at the bottom of the event block so that they click together. The blocks will automatically snap together and form our script. If you make a mistake, don’t worry, you can always use the “Undo” option to go one step back, or you can simply delete the code block and start again.

Finally, click the little green flag above the stage area to run the script. Notice how the cat moves ten steps. You can edit that value inside the “move 10 steps” block. The 10, which is written on a white background, can be changed to anything we want. If you want the cat to move 15 steps, then erase the 10 and add 15. Many code blocks have a value like this that you can change. The value that you see when you look at the blocks inside the code block pane is just a default value.

While this project is very basic, it shows us that the Events code block is important. All of your projects will have to use an event to trigger an action. Most of them however, will use the “when flag clicked”.

Now, let’s add a bit more complexity to this project and make the cat move in a more realistic way. Here are the steps you need to take:

1. Go to the Looks section and add the “switch to costume” code block to the motion block. Make sure they snap together, otherwise you’re creating a seperate code block.
2. By default the value is going to be set to costume2. If you click “play” you’ll see that the cat is going to change its movement position but only for the first movement. All the other times you click on the flag, the cat will remain stuck with costume2. We’ll have to play with the animation to improve it, but first navigate to the Costumes section.
3. Now that you’re in the sprite editor, we can select either costume. Click on the first then the second rapidly a few times to get an idea about the motion. When you switch between two images it’s like looking at an animation. The cat doesn’t go anywhere, but its motion changes due to the change in the pose. We need to use code blocks to simulate this back and forth switching.
4. What we need is another “switch to costume” block, but this time it needs to have costume1 as the value. But if you try the play button now, the cat still doesn’t move like we want it to. It’s still missing something, but what?

5. The switch between the two sprites is actually working as intended. We just can't see it because the computer is processing the transition incredibly fast. Our eyes just can't handle the fast change, so we need to slow it down. To do that we need to play with a timer.
6. Navigate to the Control category and select the "wait secs" block. This block adds a delay between the two actions. Since by default the value is set to 1, that means one second will pass between the costume switches. That's too long though, so play around with the values until you're happy with the animation. 0.2 should be good enough.

The switch costume statement creates the illusion of having an animation because it changes the character between the two sprites. The wait timer adds to the illusion by providing us with a transition. This is a simple way of "animating" a character or an action by using the Costumes panel. The only problem we have in our script right now is that we need to hit the play button multiple times to make the cat move again and again. But now that we have a basis for our project, we can take some time to improve it.

In programming, it's quite normal to start simple and then add things as we go. Rarely can we create something close to perfect from the start. Working that way is slow, frustrating, and time consuming. So, always start simple and improve as you progress.

Refining the Script

To improve the cat animation, we need to make the cat switch between the two costumes over and over again. The change has to repeat forever. In programming terms, our code needs to go in a loop that keeps running as long as the program is still running. In Scratch, we place the blocks of code inside a forever block, which is an infinite loop.

To do that, you need to navigate to the Control section and select the "forever" code block. Just drag it under the "when flag clicked" block and it will automatically snap with the rest of the code blocks inside it. When you run the script now, you'll see that the cat will vanish from the screen. That problem is caused because there are no limits to how far the cat can go. Since the loop goes on forever, that means the cat will keep taking 10 steps forever until it's nowhere to be seen.

To fix this issue, head over to the Motion category and select "if on edge,

bounce” and drag this block right after the second costume switch, before the forever block ends. The cat will now walk until it reappears upside down and starts moving around the bottom right section of the stage area. We can stop the cat from running around by hitting the stop button.

The script has been improved, but we still have some problems. The animation isn’t quite as smooth as we want it to be and then we have the issue of our cat going upside down.

Whenever you need an action to repeat itself, you can use a loop. In our project, we’re using an infinite loop because we don’t have any conditions that tell the program to stop the loop. All the code inside the loop will continue running for as long as the program is up and running. Throughout this book, we’ll be using them in many situations, and in other programming languages they’re one of the most important tools at a programmer’s disposal.

With that in mind, let’s continue improving our script!

Since our character is rotating upside down, we need to make it rotate back up when that happens. What we can use to achieve that is the “set rotation style” code block that you’ll find in the Motion section. Drag and drop it right under the “bounce” block inside our loop. The default value is just what we need, which is “left-right.”

Now let’s fix our sprite animation. In the Looks code block section we have a “next costume” block. Add that to the script area, but don’t attach it to our current block structure. We need to remove a few blocks first. Delete the two “switch costume” blocks, as well as the “wait secs” block from the script. You can do that by either clicking on each one and then pressing the delete key, or by dragging the blocks out of the structure first and then deleting them. Next, insert the “next costume” block right after the “move 10 steps” block. Finally, you can add a “wait secs” block at the end of the loop and play with the values to tweak the animation.

So, you’re probably wondering why we were able to remove all those blocks and just use “next costume” instead. It’s all because of the loop. Before using the forever block, we needed two different blocks to make the switch between the sprites. The switches were actually just one process with a 0.2 second delay in between, and it had to be done manually with each click. Once we replaced the switch blocks with “next costume”, the switch started happening with every single loop. This made the animation a lot more realistic and it works well enough even without a delay block. The loop made the entire script a single

repeatable process instead of a collection of processes.

By having the loop, we obtain the same result but with less code. This is a perfect example of creating a project and then improving it over time by finding new solutions. Sometimes we can't find the perfect solution from the start, so we just go with something that works. Eventually, we can refine that solution. In this case we don't have to know which sprite we need because it doesn't matter. If sprite 1 is active, the next sprite will be sprite 2. If sprite 2 is active, then the next will be sprite 1. This switch will go on and on until we stop running the program.

Adding More Sprites

Every single Scratch project starts out with a cat sprite and nothing else. But that's not enough, and we rarely want to use the cat on its own when creating a game or an interactive story. We always need multiple sprites.

To have more sprites in the stage, we can do several things: we can paint a sprite ourselves using the sprite editor, select a new sprite from the Scratch library, or find one somewhere on the Internet and then upload it. These are the most common ways of adding some graphics to your project.

The options to add a sprite using either method can be found by clicking on the cat icon in the lower left part of the screen. The easiest approach is to just select "Choose a Sprite" because that opens the Scratch sprite library, which is filled with graphical elements.

Now, let's add a new sprite to our project and explore a few more code blocks:

1. Start by adding a sprite from the Scratch library by clicking on the "Choose a Sprite" option. Browse through all the categories to get a feel for the library and what you can find there. Since we're already using a cat, we'll keep things simple by adding a Dog sprite, but you can pick anything you like. Be creative!
2. Once you select the sprite, it will appear in your stage area. To start adding scripts to the new sprite, you have to first select it from the sprite information window. Notice that when you click on the dog, your script area is empty, but if you select your cat again, you'll see your old script. That's because your current script is attached to the cat sprite only. We need to create a new script that controls the dog sprite.

3. Let’s start by dragging a turn block from the Motion category. It can be either block. One of them controls the turn clockwise and the other does it counter-clockwise. We can make adjustments to both of them in such a way to suit our project.
4. Next, go to the Control section and take the “repeat” code block. This will be connected to the turn section.
5. Wrap everything up in a loop by using the “forever” block once again.
6. Afterwards, head over to the Event category and select the “when this sprite clicked” code block. Add it at the top of your code.
7. We’re almost there! Let’s add some more content to the project by adding a “say for secs” block at the end of the repeat block. Make sure to keep it inside the loop.
8. Now that we have the basis of the new script, we can start playing with the values in each code block. We’re not going to keep the default ones because that would be boring. You should experiment with these values, but to give you a rough idea about how they should be, you should add “30” to the turn code, 100 to the repeat block, and any message you want to the say block, like “Woof woof!”
9. Try out your project and make more adjustments. Explore other code blocks and add anything you can think of. Have fun!

Let’s talk about what happened in more detail. We programmed two different sprites to behave in their own way, separate from each other. We added a new “dog” character and created a script for it so that it chases itself in circles. This is what the “turn” code block is for. You can adjust the value to slow down or speed up the action. Then we have the “repeat” code block that controls the number of times the character spins in a circle before saying “woof woof”. This block is in fact a loop, but it’s not infinite like the “forever” block. We have to give it a value and that number represents the number of times the code will repeat. In our case, it’s going to repeat 100 times.

This example teaches you how to combine various actions to make a character more complex and how to write multiple, independent scripts to your projects. The cat starts acting as soon as the play button is clicked, but the dog will only react when you click on it. We’re using different events to make things a little bit more interesting.

Part 2: Projects and Games

Chapter 4: Project 1 - Birthday/Holiday Card Animation

Now that you know the basics of using Scratch and you're familiar with most of the features it has, we're going to start working on a fresh project from design to completion.

In this chapter we're going to animate a digital card that you can give to anyone you want. We're going to use some premade sprites from the Scratch library and create some of our own as well. Creating an animation is a great way of practicing basic programming techniques, and working with graphics will also teach you how to use the sprite editor. Next to creating a fun animation and using your creativity, you will also learn how to design bitmap and vector images in the paint editor, and use various graphical effects.

Working with the Paint Editor

Buying birthday cards for someone doesn't show your appreciation for them. That's why you make a homemade card that you design yourself! The same way you use paper, glue, and scissors to make one, you can use the paint editor in Scratch to make a digital version. So let's get started and at the end, send a card to someone you like and cherish.

Start by creating a new project. The first step is to add or create all the graphical elements we need, such as characters and various objects. We can't animate anything if we don't have any sprites. So, how do we make a digital card? Just like in the real world, we're going to start with the simplest part, the Happy Birthday wishes.

Select the cat character you have by default and delete it because we won't be needing it. We need to draw our first sprite, so go to the Choose a Sprite option and select Paint. This opens the paint editor and creates a new sprite for us. Next, select the Text tool by clicking on the T icon. This will allow us to type some text by opening a text box. Click anywhere on the canvas to open the text box and start typing "Happy Birthday", "Happy Holidays", or anything you want. At the top of the canvas you should also notice a few controls that let us modify the

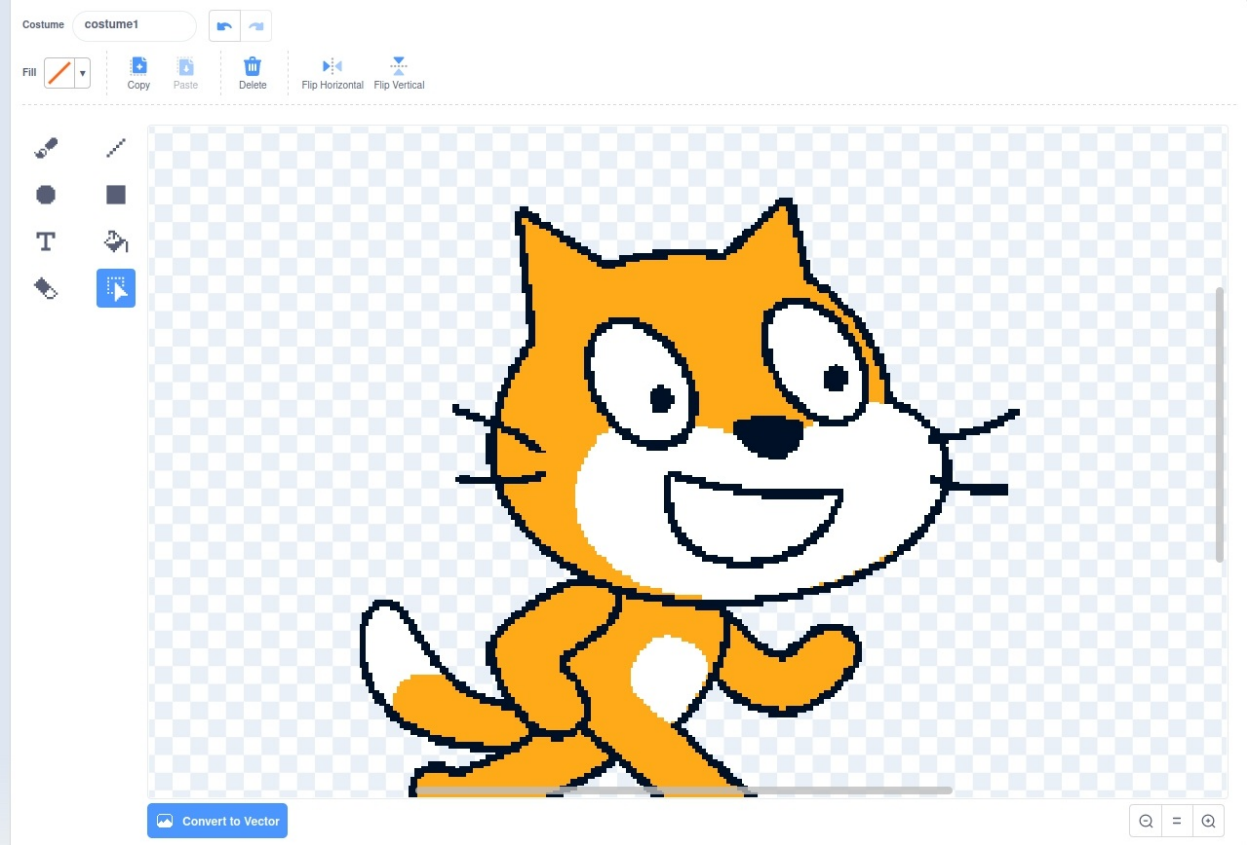
text. Mainly, there's a Fill option that controls the color of the text, and a Font option that modifies the look of the text. Play around with those settings until you're satisfied. You can also start over by deleting the text box or using the eraser tool, if you made some mistakes.

Take note that your sprite is automatically saved and updated. As soon as you write the text you should see your new sprite under the stage area. Even though we wrote some text, it is now converted to an image and it works the same way the cat sprite does. It's no longer text.

Once the text turns into an image, you can resize it just like any other image. Click on one of the corner transformation points and drag it away from the image to increase the size, or in the opposite direction to shrink it. The sprite will be automatically updated. As an alternative, you can also go to the Looks code block category and select "set size to" and then play with the values until you get the size you want.

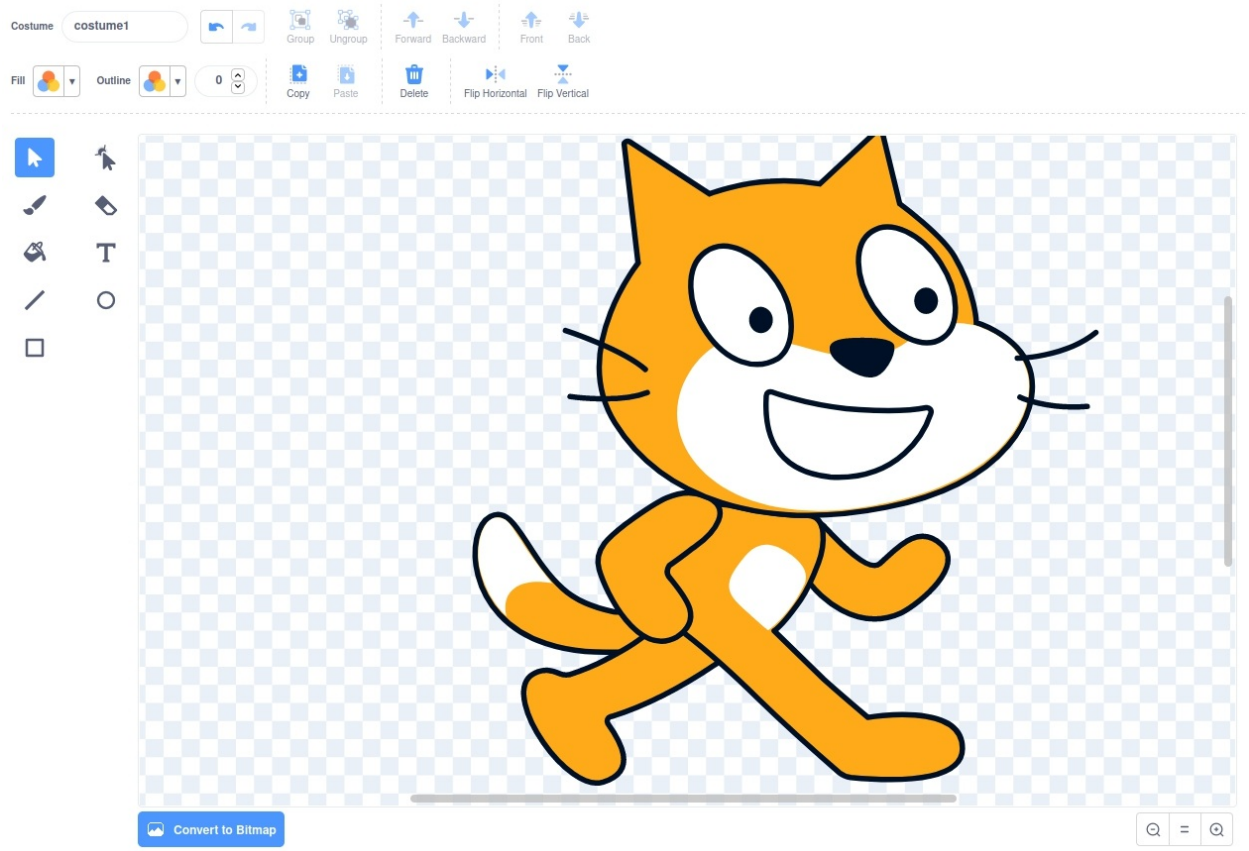
Bitmap or Vector?

When working with graphics in Scratch, you have two options. You can either go with vector graphics or bitmap graphics. A bitmap image is easy to work with, but the more you increase its size, the worse its quality gets. Bitmap images become blurry and rough-looking when you go too far. This happens due to the way bitmap images work. They're essential collections of pixels, or small colored dots in other words, and they're arranged in a certain way. That arrangement of pixels is the image you see. You can see the structure of this type of image by zooming in until you can see the pixels themselves.



When you increase the size of the image, you stretch it so much until the pixels become more and more obvious. This is when the image starts looking blurry.

On the other hand, we have vector images. These can be enlarged as much as you want and you will never lose quality. They're just a bit more difficult to work with, but it's worth it if you need to have the ability to resize an image. Why is that? Vector images work completely differently. The image is generated and displayed with the help of mathematical calculations that create lines and curves. There are no pixels involved here.



Thanks to those algorithms working with lines and curves, we can enlarge a vector image indefinitely and the quality will be the same no matter the size. We're going to see these differences more clearly by actually working with both image types.

The sprite we created so far is in vector mode. We can tell because inside the paint editor Scratch is offering us the option to convert to Bitmap. You can also try to enlarge your image, and you'll see that there's no change in quality.

Now, let's create a new sprite and click on the "convert to Bitmap" button at the bottom of the editor. This action will change the paint editor as well and some of the options will disappear because they can't be applied to bitmap images. Select the text tool like you did before and write anything you want. Now you can select your image in the paint editor and start dragging it from its transform points to increase its size. You can also switch to the script area and use the "set size to" block set to something like 400.

Compare the two images and notice the enormous difference in quality. This is why most images in the Scratch library are actually vector graphics. That's how you can increase the size of the cat to whatever you want without experiencing

that awful pixelation effect.

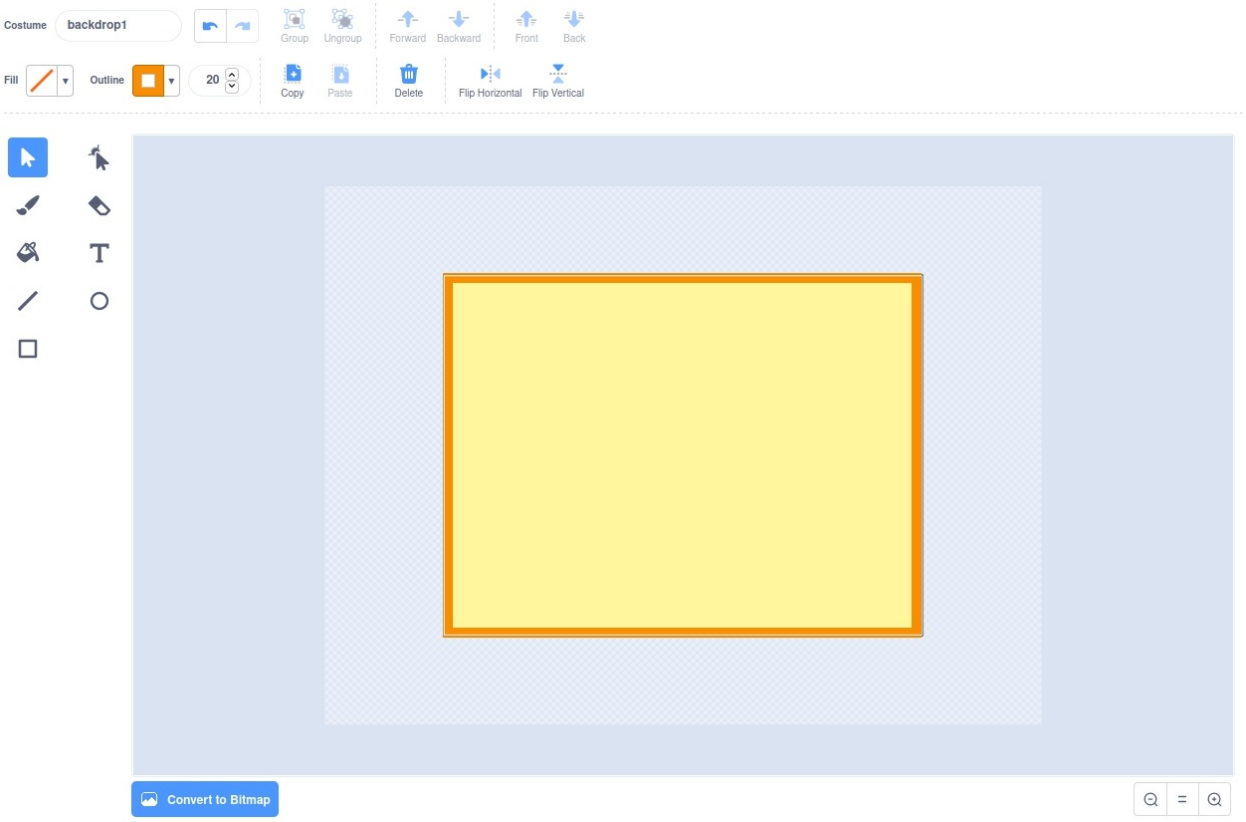
Background Colors

Having fun looking sprites is nice, but our project looks boring with just a plain white background. In Scratch we can upload or create our own backgrounds, so let’s add one for our card.

Next to the Choose a Sprite button, there’s another similar looking one called “Choose a Backdrop”. This is where you find the background settings. You have the same options as you do for your sprites. You can choose a premade backdrop from the Scratch library, upload one of your own, or paint one yourself. Let’s select the paint option, which will open the paint editor.

Notice that most of your paint tools are the same, but what we need here is the bucket tool so that we can fill the entire stage with a color. Choose your color, select the tool, and click on the empty area to fill it with the selected color. Experiment with a few colors until you find something you like. You can always undo or click on the Clear button to get rid of the background.

Now, let’s add some detail to the backdrop so we have more than just a plain color. We can wrap a frame around it and fill it with a second color. To do that, make sure you’re painting in vector mode, which should be the default setting, and select the Rectangle tool. Pick any color you want, preferably one that creates some contrast or complements the main background color.



When you draw the rectangle with the default settings, it will probably be filled with color instead of forming a frame around your current backdrop. To change that, click on the “Fill” option and set it to none. The symbol for no color looks like a white square with a red line passing through it. Afterwards, click on the Outline setting and choose a color. This will form a frame, but you still need to control the thickness. That’s done with the value next to “Outline.” Pick something like 15 or whatever works for you.

Our scene has a bit of life now, but it could still be better. Let’s use a gradient instead of a regular color.

A gradient will have two colors. It gradually progresses from one color to the second one, creating a smooth looking blend. Now, let’s go back to the paint editor and select the gradient option instead of the Fill option.

Click on the Fill color that’s currently selected to open the color options menu. At the top you will see four types of color. The first is Fill, the second is gradient from left to right, the third is gradient from downwards to upwards, and the last one is a gradient that starts from the center moving outwards. Choose whichever gradient type you want because they all work the same. Click on each color box and select your color. For instance, you can pick yellow as your first, and black

as your second to create a smooth gradient going from yellow to dark yellow until it's eventually black. You can also use the arrow buttons to reverse the two colors.

More Sprites!

We haven't added a single bit of code yet because we're still not quite done with the design phase. Sometimes, to be a programmer you also need to learn multiple skills involving design and art. This doesn't mean you have to worry about learning how to draw and paint, but every programmer has to be able to prototype. What does that mean? Prototyping refers to creating a first, rough version of your project using any kind of basic graphics and sound, just enough to see your ideas in practice. It doesn't have to be pretty at first, it just has to work. Afterwards, you can polish your work as much as you want.

With that side note in mind, let's open the paint editor once again and add some more sprites. We're going to spell out the name of the person receiving the card one letter at a time. This means we need a sprite that represents each character. Let's keep it short for the sake of simplicity and make a card for Bob (or anyone else you want). Make each letter sprite the same way you created the first sprite using text.

Once you have the letter sprites, place the message in the stage area somewhere where you have room for the new sprites. Next, position the letters underneath. Adjust the size of the sprites to make the card look nice and orderly.

Now our stage is set and it's finally time to start programming the animation!

Animating the Card

When you think of how to animate something, you should take some inspiration from the real world, or at least your own imagination. What do you want the card to look like? How will the receiver interact with it? What's going to happen when he or she hits the play button? You should try to answer all of these questions and then start placing the code blocks to fit that vision.

The first thing we need to do is hide all the art so that when the user presses Play they don't see everything at once. This way we can slowly animate and reveal each sprite. So, select the Happy Holidays sprite or whatever your main message

is, and drag the “when flag clicked” event block to the script area. Since we want to hide the sprite when the flag is clicked, we need the Hide block from the looks category. See? The code works exactly the way we imagined things.

Next, we need to also hide the receiver’s name, but we have several sprites to deal with. This means we need to hide each sprite, one at a time, by attaching a script to each one. To work faster and not have to drag every block to each script area we can simply select our initial script, right click on it, and select duplicate. This makes a copy of the whole script and you can drag it to another sprite. Repeat the process for each sprite and then test them all to make sure everything works.

So far, the screen is blank because the sprites are hidden. That’s not good and it’s going to confuse Bob. Imagine the sprites being a band waiting for the concert to start and the curtain to lift. That’s what we’re going to do next: bring the band out on stage.

Let’s start by revealing the message first. Select the sprite and add a “wait” block from the Control category. A one second delay should be enough before displaying the message. Now, let’s specify the default size of the sprite with a “set size to” block. Play with the value until you’re happy with the size. Something around 300 should do the trick. To reveal the sprite all we have to add next is the Show block that you can find in the Looks category. Test your script to see what you have so far and make some value adjustments if you don’t like the size and delay.

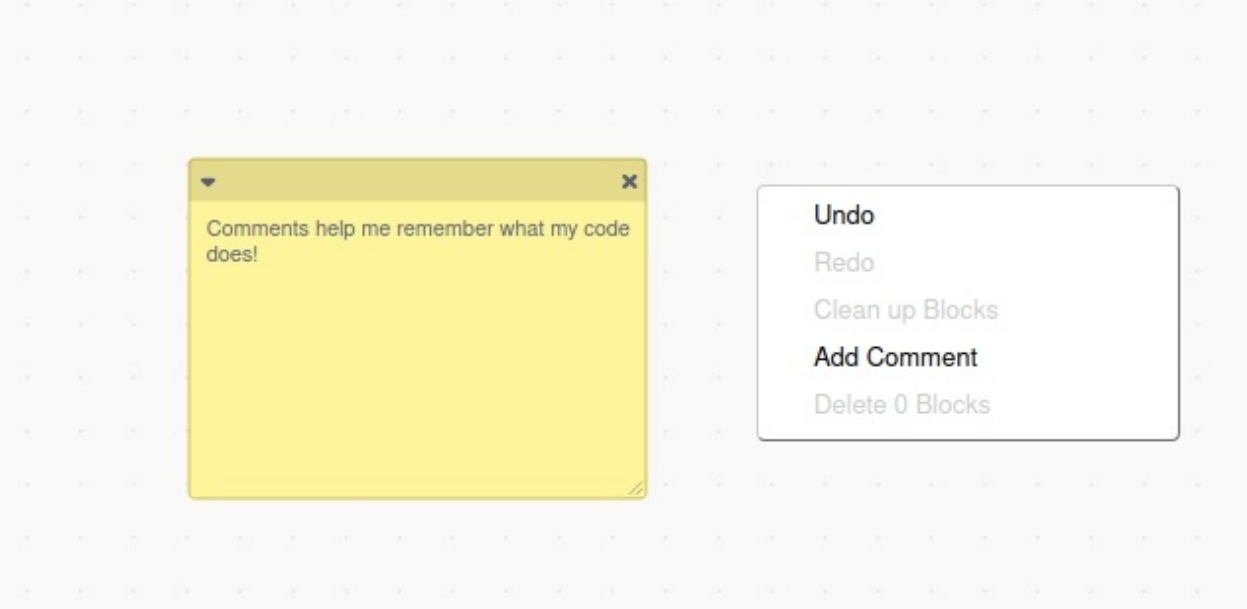
Next, we’re going to make the reveal more interesting by making it blink. We’ll need a loop for this, so add a “forever” block right under the “show” block. To create the blink or pulse effect, we need to add a size change modifier. So, let’s insert a new “change size by” block in the loop, but set it to around -25. Then add a second one but with the opposite value of around 25. In between them we’ll need a delay, so insert a wait block and set it to half a second. To finish up this step, we’ll also need a “broadcast” block right between the show block and the forever loop. You can add any message you want to the broadcast block, like “Hi, Bob!” Test your project by clicking on the green flag and make sure the sprite pulsates as intended.

A Note on Naming Conventions and Comments

So far we didn’t talk about naming the various elements in our project. In programming, keeping track of everything and having your scripts and graphics in order is important. In our project we have several sprites that are named by default with tags like Sprite1 and Sprite2. That’s bad, especially if our project continues to grow. Imagine having 100 sprites inside a game and having to look for everything without proper names. So let’s take a side step and talk about naming conventions and code comments.

The program, or the computer, doesn’t care how an object is named. It’s all the same if it’s Sprite1 or Brown_Dog. We need names that have meaning and that describe our objects. In programming languages we name various parts of code, but in Scratch we can use the same principle to name our sprites.

In addition, we can also leave code comments. Scripts and code can become complicated to understand when the project develops. That’s why each programming language allows the programmer to insert code comments that describe what the code does. In Scratch, we can do the same thing. This is another reason why this fun tool is such a great way to learn how to program. It even uses principles like code commenting in the exact same way as actual programming languages like Python. Here’s how it looks:



With that being said, let’s insert a comment to explain the broadcast section in our script.

Select the message script and then right click wherever you want in the scripts window. A menu will pop up, which contains the “add comment” option. Select it and you’ll see a yellow box opening. You can write anything you want in it to

remind yourself or explain to others what your code does. In this case, you can type something like ‘This broadcast tells the letter B sprite to run its animation. You can drag the comment window anywhere in your scrip area, but you can actually link it to the code you’re describing. Drag the block to the broadcast block and you’ll see that the comment will attach itself to it.

That’s it! You should always take your time to name your sprites so you can tell them apart and to comment your code when it becomes a bit complicated. Just imagine how lost you’ll be when you take a week break from your project and then you return and you don’t have a clue about what you wanted to do.

Special Effects

We have one important block in the Looks category that allows us to manipulate our sprites with various effects. For instance, we can make sudden color changes, change the brightness, pixelate, and create a mosaic effect, among other things.

We’re going to use these effects to play with our letters. Let’s start by selecting the second “B” sprite letter and add a “when I receive” block. To add the effect, we first need a “repeat” loop set to the number of times we want all the effects to repeat. Wrap the “change effect by” block in this loop and set the repeat to 25. Inside the change block you’ll have two values to play with. The first one gives

you a drop-down menu containing all the effects. Let's pick mosaic, but try the rest of them as well to see what they do. The second value represents how much we want this effect to change our sprite. Set it to 5 and also add a "wait secs" block at the end and set it to 0.1.

Next, we'll need a "clear graphics effects" block so that the sprite reappears without the effect, otherwise it would turn invisible. Now, let's add another repeat loop with the value of 25 under the first loop. We'll also need a new "change mosaic effect by -5" block, followed by a 0.1 "wait secs" block.

We're almost done, but we still need a "broadcast" block to tell the next sprite letter to begin its animation. That's it!

Before continuing with your project, you should try the other effects. Here's what they do in a nutshell:

1. Color: This effect is used when handling sprites because it can change the color to anything we want.
2. Fisheye: This adds a distortion effect to the sprite. It rounds the edges of the sprite, making it look as if it's behind a lense or a piece of glass. This effect is useful when creating a transition between sprites.
3. Whirl: This is another distortion effect that twists the sprite around its central point. It makes it look as if it's filled with ripples like those you see when throwing something in the water. This distorting spinning effect is another great choice for sprite transitions.
4. Pixelate: Normally, we want to avoid this effect because it blurs the image. Remember, this effect also happens when increasing the size of a bitmap image. But sometimes it can make an interesting transition effect because it allows you to blur a sprite which lies on top of another sprite.
5. Mosaic: This effect breaks our sprite into a series of smaller sprites.
6. Brightness: You can use this effect to add a flow to your sprite. You have full control over the brightness of the sprite, so you can really make it shine.
7. Ghost: This effect has multiple uses because it can make the sprite transparent. You can use it to completely hide a sprite or make it look like a ghost, transparent but still visible. It's also great as a transition effect because you can make a sprite slowly fade in and then fade out.

All of these effects need to be undone to reset the sprite once you're through with the effect and you want your sprite to be seen normally again. This is done

with the “clear graphic effect” block, like we used in our example.

Now, let’s continue with our project by adding an effect to the first B. So far only the second one is being displayed. We’re going to set up a script that will turn the B around itself. By now you should be able to work this out on your own. Spend some time looking at the other scripts and thinking it through. After you’ve at least given it some thought, go through the guided example below.

Start by adding a “when I receive next b” block, followed by a show block. We need to see the first B sprite as soon as the broadcast for the “next b” is made. Keep in mind that in this case however, we don’t want a graphical effect. We want to rotate the sprite. To do that we need the “turn degrees” block from the Motion category. This command should be inserted inside a repeat loop with 36 repetitions. Why 36? Because we want the sprite to turn by 10 degrees at a time, so we need 36 loops to make it do a full turn, since a circle has 360 degrees. See how basic math finally becomes useful and fun? Finally, we just need to drag a broadcast block at the end of the script, outside of the loop, and broadcast “give me an o” to signal the O sprite.

Now all you need to do is animate the O letter. Choose whichever method you want, and add a different effect. All you really have to do is copy either script from one of the Bs and make some changes to add some excitement.

Final Touches

Let’s add a proper “holiday wishes” greeting on the card and animate a new sprite. Start by selecting a sprite from the library, something appropriate with your card. For this example we’ll go to the Animals section and pick a cute hedgehog, but you can pick absolutely anything you want.

After you select your sprite and click “ok” it will be added to the stage area. Scratch automatically places these sprites in the middle, so click on it and drag it wherever you want it to be. Now, go back in the paint editor to create a new sprite with a congratulatory message. Move the sprite in a suitable place so that all the sprites look good and fit well together. Resize them if needed.

Now that we have a hedgehog and a note, we need to animate them because right now they don’t do anything. While everything else moves around, these two sprites are simply always there. So, let’s play with the ghost effect starting with the note.

Click on the note sprite and add the “when flag clicked” block, followed by a

“change effect by” block. Select your favorite effect, or the ghost effect like in this example, and set it to 100. When the ghost effect is set to 100, it means that our sprite is hidden. That value can only go between 0 and 100, and 0 means it’s fully visible. But we don’t want the note to appear instantly, so we’re going to also use a “wait 9 secs” block right after the “change effect” block.

Next, we need a repeat loop set to a value of 20 and inside it another “change effect by” block set to the mosaic effect and a value of -5, just like we used for our previous sprites. Right now our ghosting (fading) effect is really fast, so we need to slow it down with a “wait 0.2 secs” block inside the loop block.

Finally, you can test the new sprite and see how it fades within three seconds after animating the word “Bob”.

As a final exercise, the hedgehog sprite is up to you. This is your homework because you need to practice on your own as well. Everything else is fully animated, except for this sprite. Use your imagination and come up with new ideas!

Chapter 5: Project 2 - Creating an Interactive Book

Now that you know how to design a whole project and how to work with graphics, you can move on to the next step: a story book. We're going to design a book that allows the user to make some choices and interact with it. You'll learn how to design a book outline, how to work with the "say" block to deliver strings, how to use sound effects, and how to use the coordinate system to navigate.

The Design

When designing this kind of project you should take some inspiration from the real world. Pick up a book, study it, and think about what we might need. The most obvious starting place is the table of contents, of course. After all, that's the first thing you see when you open a book and you use it to navigate. We will also have to make a few buttons and label them so that the user can click on them to open up various chapters.

Start by creating a new project and deleting the default sprite. Click on "choose a sprite" and go to the Scratch library to pick a button sprite. You'll find various buttons inside the Things category. Since this is going to be a more complex project, you should name your sprites from the start. Button1 doesn't say much, so let's name it Hedgehog TOC. Now, go to the Costumes section to launch the paint editor to create the label. Type "Hedgehog" and place it nicely over the button sprite. Make sure you're working with vector images so you can resize them without worrying about quality.

Since we need more than one button, let's duplicate the one we already created and replace the text with "Fox." Feel free to add as many table of contents buttons as you want. But before continuing, we should have an "instructions" sprite that tells the user what these buttons mean. For instance, you can say something like "Click on the buttons to choose a chapter" and then place the sprite at the top of the stage area.

Next up, we need some pages for our book. That means we're going to need two backdrops to represent the pages. We're going to use the backdrop library this time instead of painting our own backgrounds, and we'll use them to create just one image.

Start by going to the backdrop button and selecting “choose backdrop from library. Choose your favorite backgrounds and make them match with the topics you chose. For instance, if you chose a dog, you might want a backyard backdrop or a bedroom backdrop. We’ll also bring a microphone sprite into our scenes to offer some visual help. But first, open the backdrop in the paint editor and then bring the sprite over it and arrange it wherever you want. This way the two graphical elements become one image.

Once you’re done with this step, you’ll notice that we have two microphones in our scene. That’s because one of them is the sprite we imported and the other one is the sprite that we combined with the backdrop. We’ll take care of that soon. The reason we’re going through this process is order. The fewer sprites and elements we have, the cleaner and more manageable the project will be. By combining the sprite with the backdrop, we’ll later be able to just remove the sprite from the sprite list.

Because we want to use the microphone sprite in all of our scenes, we’re not going to just simply delete it, otherwise we would have to re-import it again later. Instead, we’re going to use the Backpack feature. You can find it at the bottom of the page, right under the scripts area. Take note that you can only use this feature if you’re logged into your Scratch account. If you still aren’t, this is the time to sign up.

Open the backpack and move the microphone sprite into it. This will copy the sprite and then we can remove it from the sprite list by just selecting it and pressing the “delete” key. Now, let’s select the second backdrop and add the microphone to it as well. Open your chosen backdrop inside the paint editor and then drag the microphone from the backpack to the editor window. Place it wherever you want and change its size accordingly.

The Content

So far we have two table of content sprites in our scene. Select them, right click, and then hide them from the view. Click on your first backdrop to bring it into the stage area if it’s not already there. Now let’s import a hedgehog and a fox sprite from the library, or any other animals or objects you’re using. Select the hedgehog sprite and drag it to the first backdrop scene, then position it somewhere close to the microphone, and resize it if necessary.

Now, let’s start creating a script and write some text. Select the “say for” block from the Looks category and change the message to anything you want. You can also change the “for” value, which is a time value, to specify for how long that text is being displayed. Test this out to see if the character’s speech bubble is inside the scene view. If the sprite is too close to the edge, the bubble might be partially cut out. Add another “say for” block afterwards, with another bit of text and an appropriate timer. The timer depends on how long your text is. So test things out to see how long it takes you to comfortably read the text you create. Next, add a “wait secs” block with a 3 second delay, followed by another “say for” block.

Now that we have some text in our book, we can start animating our character. We’re going to do that by using the sprite's second costume and triggering another say block. Add a “wait secs” block after the last message, followed by a “next costume” block, and a “say” block without any value inside it. The reason we finish this part of the script with a no-value string block is because we want to clear the character’s speech bubble at the end.

Importing Audio

You can’t make a fun interactive book without adding some audio and sound effects. So, let’s add some sounds to our story using Scratch’s sounds library.

Click on the “choose a sound” button and open the Scratch library. Go to the animals category and pick whichever sound you find appropriate. There are enough effects to fit most animals. Select the sound and click “OK”. You can also duplicate the sound afterwards to add more effects by right clicking on the audio and choosing the “Duplicate” option. Now, select audio inside the sound editor to capture all the effects and use the “softer” effect. To do that, you can click on the left side and drag your mouse all the way to the right. The sound waves are selected the same way you select a block of text.

Let’s name this second sound file “hedgehog soft” so that we know that the sound belongs to this particular character and that we applied the softer effect. Next, we’re going back to our script to insert the sound using the Sound category blocks.

Start with a “play sound until done” right at the top of the script, before the first “say for” command. Then add a second sound block after the second “say for” command. For the first sound you should select the original one that you chose, without the softer effect, and then use the second one we modified in the second

“play sound” block. Finally, let’s add another sound block after the “next costume” block. This way we have sounds throughout the script whenever the animal speaks or does something and it adds some life to our project.

Using the Coordinates System

Our first character is all set, so let’s move on to the fox (or whatever you picked). We’re going to do something different with the fox and move it across the scene by using x and y coordinates. The sprite will be positioned in an exact location based on the coordinates.

Let’s get started by hiding the hedgehog sprite (our first character) from the scene, and selecting our second backdrop. Add the fox sprite close to the microphone and resize it to make it fit in. Now that the sprite is in position, we can find out its precise location in x and y coordinates. By knowing these values we can write the script in such a way to return the sprite to that initial position.

Click on the sprite and then look at the panel above the sprite information window. You’ll notice an X with a directional arrow pointing horizontally, and a Y with a directional arrow pointing vertically. These are the coordinates, and they can have negative or positive values. X represents the horizontal line, which goes from left to right. If you imagine the center of the screen, everything that is left from the center is represented by negative values (like -147) and everything that’s right is in positive values. The center point is 0. The Y value works the same, except that it goes up and down from the center of the window. The upper part is represented by positive values and the lower in negative values. If you set both coordinates to 0, the sprite will be perfectly placed in the middle of the scene.

Now, let’s create the script to return the fox to its initial position, close to the microphone. Go to the Motion category and choose the “set x to” block. Depending on where your character is right now, you should copy that X value and add it to this block. Do that same with the “set y to” block.

After the fox says something, we want it to move somewhere else in the scene. Move the sprite to a position where you want it to be and copy those x and y values so that we can use them in the next block. Create a “glide secs to x: y:” block and paste your coordinate values. Just don’t connect this block to the previous “set x” and “set y” blocks. Also, you should add a value in seconds of around 2.

That’s it! Test out your scripts to make sure the character moves as it’s supposed to and tweak the values a bit more if needed.

New Costumes

If you’re following along with this example, you probably noticed that your character is moving from the microphone backwards, if you set the mic on the right side of the screen and the glide position on the left. Since animals don’t tend to walk backwards, we’re going to create a new costume to make the sprite point the other way. What we’re going to do is take the original sprite, duplicate it, flip it, and then create a script that switches between the two sprites.

Select the sprite and go to the costumes tab. Right click on it and duplicate it. To flip the new sprite, all you need to do is click on the flip button in the editor’s tool tab. Now you have a sprite that faces right and one that faces left.

The next step is to add the “switch costume to” blocks inside the movement script we just created. Add one containing the “face left” sprite first right above the glide block, and another one containing the “face right” sprite above the “set” blocks. And done!

By this point you’re probably asking yourself, what’s the difference between sprites and costumes? Aren’t they one and the same? That’s a good question!

Sprites are simply graphical objects that do something inside our project. Sprites can move and do things when they have an action attached to them. The word “action” has a different meaning here, because it’s not referring purely to the script. An action also involves costumes and other attributes that we link to the script.

Costumes are just a visual appearance of a sprite. You can look at them as frames inside an animation sequence. You can have a complex animation that contains 30 frames, for instance. Each frame is a costume, and all of them together will animate a sprite.

Now that you understand a bit more about Scratch graphics, let’s turn back to the audio system and learn how to create custom sound effects for our project.

Creating Sound Effects Using an Extension

Earlier in our project we used the “play sound” block, and we can have other

variations of audio blocks that we can play with. For instance, we also have the “play (drum) block” that we can further customize by setting its beats per minute value. Let’s do that and create a cool drum effect. But if you look at your code block categories, you won’t find this option. That’s because it’s part of an extension library.

If you look at the bottom of the color-coded code blocks, you’ll see the option to “Add Extension.” Click on it and it will open a new window with several options. For this project we need the “Music” extension. Selecting it will add it as a new block category.

Go to the new Music category and grab a “set tempo to bpm” block. Drag it to the script area but don’t attach it to any other code. Change the default value to 150. Next, drag the “play drum for beats” block and leave it on the default value, which is a snare drum set to 0.25 beats. We’ll also use a “rest for beats” block from the same Music category and set it to 0.4. Now, add another two “play drum for beats” blocks with the default drum values and a beats value set to 0.1. Finally, add a “play drum for beats” block as a last block and set it to “crash cymbals” with 0.5 beats.

We have finished composing a sound effect! Select the script and hit the play button to try out. Make some modifications and play with the values. Try other drum types and see what you can make! Right now you’re wearing the hat of a composer, so use your creativity.

Content Homework

We finished setting up our characters, but our second animal doesn’t have any story yet. Open up a new script and start writing some text and add some motion effects as well. Go back to the hedgehog script to review what we did so far and use some of that inspiration to program the fox’s behavior.

Navigation and Scenes

Since our interactive story is almost finished, all we need is a way to navigate through the scenes. The user needs the ability to go back to the table of contents to choose another scene. Before taking this step, you should spend some time adding additional content. Create your own story and make at least five scenes to have some variety.

Now, let’s go back to the table of contents we created at the start of the project. Hide the characters and change the backdrop back to the plain white background. Bring back the two TOC sprites and the instructions sprite by selecting them and clicking on “show.” Remember that we hid them. We need to make the hedgehog TOC sprite clickable, so select it and let’s start adding to the script.

From the events category we’ll need the “when this sprite clicked” block, followed by a broadcast block, and a hide block. When we click on the TOC button we want the correct scene to load and the table of contents page has to be hidden. We also want the broadcast to work as a sign to hide every sprite. So, let’s go to the Fox TOC button and insert a “when I receive” block and choose the “hide toc” message that we broadcasted in the previous sprite. A “hide” block is needed here as well.

Back to the first sprite, we’re going to work around the broadcast to start making the scene visible. In this case, we’re working with the hedgehog TOC script to bring up the hedgehog’s scene.

Let’s start by loading the scene by adding a “switch backdrop to” block right after the “hide” block, and select the name you used for the first backdrop. Afterwards, we’re going to display the hedgehog sprite by going to the Events category and selecting the “when backdrop switches to” block, with the first scene enabled. Add a “show” block as well to display the sprite.

Now, let’s connect the “when backdrop switches to scene1” script block to the “switch costume” script to combine them. Now that our scene loads, we just need to fine-tune it and perfect it a little. So, let’s add an action delay so that the transition doesn’t happen instantly. Add a “wait” block with a value of 5 after the “set y” block.

That’s it. The first scene is ready and it should be triggered when clicking on the Hedgehog TOC button inside the table of contents scene.

After our story scene plays out, the user needs to be able to switch to another scene. To do that, we need a button in our scene that will lead us back to the table of contents. Here’s how we’re going to add it:

1. We’re going to pick a different button from the Scratch scene. You can also create your own inside the paint editor to make it fit better with your scenes. Add a label like “home” on top of the button, just like we did at the start of the project, and place it somewhere in the

scene where it won’t bother you.

2. From the Events panel, grab a “when this sprite clicked” block and drag it at the start of the button script.
3. Add a broadcast afterwards with a “display TOC” message, followed by a “hide” block.
4. After sending the message to display the table of contents with the help of the broadcast action, we hide the Home button. Then we need to switch our current scene to the table of contents. So, we need a “switch backdrop to” block with the first scene selected.
5. To show the home button, we need a second script block separate from the one we just programmed. Drag a “when I receive hide toc” block, which relies on our broadcast message.
6. Finally, connect a “show” block at the end.

And we’re done! The navigation system is complete and we can now switch between scenes using the table of contents. Of course, we aren’t entirely done because we haven’t taken care of the fox scene, but that’s up to you. The process is the same. Just look at this scene and how the navigation works and try to work it out on your own for the next scene.

Chapter 6: Project 3 - The Caverns of Time

After spending some time practicing with your previous two projects, you should be familiar enough with Scratch to start creating your first game. If you still feel uncomfortable working with the code blocks or graphics, you should spend some more time playing around and going over the previous chapters. You'll get better and better with practice, even though it might be a bit frustrating right now.

With that being said, let's create a game that will keep everyone annoyed enough to keep playing! Here's what we're going to design and build:

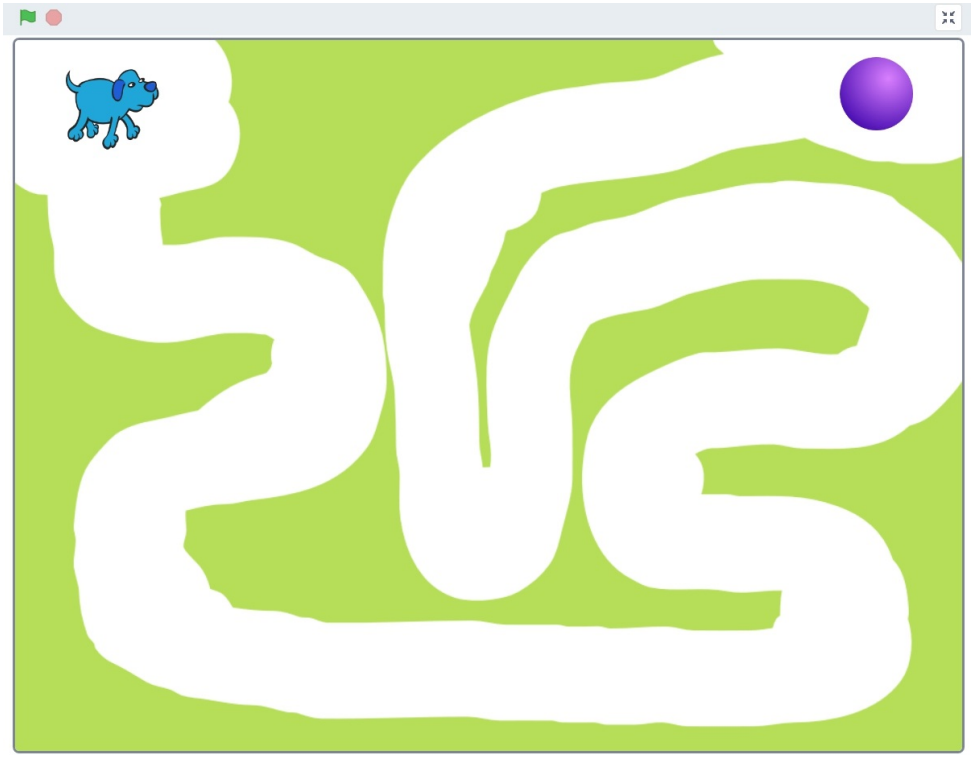
1. We'll make a challenging game that requires nerves of steel, a steady hand, and a lot of patience. One simple mistake will restart the game. This kind of game design is found in a lot of games because it pushes the player to keep trying again and again until he beats the game. It's annoying, frustrating, but also fun when you watch your friends struggle.
2. The game will require the player to move his character through a maze without touching the walls. As soon as he touches the walls, he dies and starts over. At the end of the maze, there's a reward that needs to be collected in order to win the game. We're going to call this game "The Caverns of Time" because it's going to take a lot of time if you're aren't patient enough, and because it's a pretty cool name!
3. But just winning the game isn't fun enough, especially when playing your friends or parents. So we're going to add a timer to see how long it takes each player to complete the game. The top score is going to be recorded and the next player will have to beat it to enter the hall of fame.
4. When it comes to the controls, we'll only use the mouse pointer to move the character through the caverns. When you point the mouse, the player's character will follow it because we will make it stick to it. When the player touches a wall, he'll spawn back at the starting area, but the timer won't reset! The time stops only when the end of the maze is reached.
5. As for the graphical design, we're going to draw a tunnel as our background, and we'll have a dog that is trying to get to his ball. To move the dog, the player needs to move the mouse cursor on top of

the character and that will make it follow. No need for any mouse clicks.

Now that we know where we want to go with the game, we can start the project and create the graphics, sounds, and scripts. Let's start easy by adding some music first and get the atmosphere going. Go to the sounds library and pick something from the "Music Loops" section. Then add your character sprite, in this example a dog, and start to add some script to play the audio.

We're going to need a "when flag clicked" block, followed by a forever loop that will contain the "play sound until done" block. This way the audio will keep starting over until we stop playing the game.

Next, we need a background, which is going to represent the caverns. To create the scene, go to the paint editor and select a fill color. Then with the rectangle tool, draw one big, colorful rectangle that will cover the entire scene. Afterwards, grab the eraser tool and draw the path our character will have to take by erasing parts of the background. Your scene should look something like this, but you're encouraged to spend more time and draw something nicer. Use multiple colors. Try out a gradient instead of a fill color. Get wild!



Finally, we also need to add the ball at the end of the maze, like you can already

see in the upper left corner. Once your scene is set, resize all the sprites to make them fit well. Adjust the character to make sure it can fit through the tunnels without touching the edges, otherwise the game will be unwinnable and that's not fun.

Now, select the background, and let's add a visual effect to make it more interesting. Drag the "when flag clicked" block, and inside a forever loop drag the "change color effect by" with a value of 2. This will create a loop that changes the scene's color over and over again until we exit the game.

Game Mechanics

We have the graphics and some audio, but the player still can't do anything, so let's program the controls. Select the character sprite, the dog in this case, and start with a "when flag clicked" block. Next, we need a "go to front layer" block so that the dog doesn't go behind the background when touching the walls. This is a way of making sure that a sprite is always above all the other sprites. Next, we need to use the coordinates system to tell the game where the character's initial position should be. That will be the starting point where the player will be returned if he fails to reach the end of the maze. Add a "go to x: y:" block and insert the dog's starting coordinates.

Next up, we add a special timer block "wait until" something happens. In this case, we want a "touching mouse pointer." This block is actually made out of two blocks, where one goes inside the other. The "wait until" block is found in the Control section, while the "touching mouse pointer" is inside the Sensing section. Drag the "touching mouse pointer" block on top of the "wait until" block to complete the action. This will make the character move only when the cursor touches it. Until then it stays in position.

Now we need a "repeat until" loop that will contain a "touching" block inside it with a reference to our caverns sprite, followed by a "go to mouse pointer" block. This will allow us to control the player until we touch the wall. Finally, you can add a sound effect that triggers when the dog touches the wall. Make sure to add it outside of the loop. You can also use some special effects, like ghosting, to make the sprite fade out when hitting the wall. Always use your imagination and don't just follow these examples word for word.

We're not quite done yet. Right now the character just stops moving when the

player touches the wall. We need to be able to automatically restart the game when that happens. So we need a forever loop. Just drag a “forever” block from the “go to front layer” all the way to the “start sound” block at the end. Basically, your entire script has to go inside this loop, except for the “when flag clicked” block at the start.

We’re making some good progress, but nothing happens when we reach the end. So let’s add some code that checks and confirms when the dog reaches the ball. We are going to do that using an “if/then” conditional expression.

If statements are one of the most useful tools in programming and you’re going to use them a lot no matter which programming language you use. This enables the program to make some choices. In our project, we’re going to say that if the player touches the ball, then all other actions will stop running and a message with some audio will be played. This part of the code will only run when the player goes over the ball. Until then it will be completely ignored.

So let’s add an “if touching ball then” conditional statement right after the “repeat until” loop. Inside the statement we’ll have the “stop other scripts in sprite” block that will stop the rest of our code from running, and therefore pause the game. Afterwards, add a broadcast block to congratulate the player together with a sound effect. To close the statement, finish off with a “stop this script” block.

That’s it, our game is complete! Play a few rounds and see what you can improve. Draw more complex mazes, place some road blocks, or add more ways of reaching the ball. Use your creativity!

Once you’ve polished your game a little, you can add a timer to keep track of the amount of time the player needs to reach the ball. To do that, we’ll need a new variable called “Timer” that doesn’t exist yet in the code block categories. Remember when we talked about variables? It’s time to see how useful custom ones can be.

Click on the “Variable” category and select the “Make a variable” option. We’ll call it Timer because it makes a lot of sense in this case, since we’re keeping track of time. Stick to the default setting, which is “for all sprites” and then click on your character sprite to add another script. We’re going to start again with “when flag clicked” and then follow up with our new variable “set timer to 0”. This is going to reset it when we launch a new game. Afterwards, we need a forever loop with a “wait 1 secs” delay, followed by a “change Timer by 1”. This will accurately keep track of every second that passes.

Let’s add more detail to the way the player is congratulated when winning. This step is optional, but it makes the game feel more complete and rewarding. The first thing we need to do is hide the congratulatory message from the start, otherwise it will appear throughout the game and we don’t want that. We only need it to appear when the dog reaches the ball. Let’s create it and add some special effects as well.

Start by creating the sprite and add the following script:

when flag clicked

hide

go to x: 0 y: 0

These blocks will position the sprite in the middle of the screen since all coordinates are at 0, and they will keep it hidden throughout the game. Now let’s add another script that will reveal the message when the round is won:

when I receive Congratz

show

go to front

forever

    change color effect by 20

When you win the game, the message will pop up and flash due to the color effect.

All that is left to do now is show the timer during the game and also add a Top Score option to keep track of the best time it took to go through the maze.

We’ll need a new variable that we’ll call Top Score. Go to your character’s script and add a “when I receive congratz” block, followed by an if statement. Next, add an if statement with multiple conditions, like this:

if Top Score = 0 or Timer < Top Score then

    set Top Score to Timer

If/then statements don’t need to have just a single condition. It can have multiple conditions that need to be fulfilled, in which case we would use the “and” operator. But it can also have multiple options where only one of them needs to be fulfilled for the code to be triggered. In this case we’re using the second option with the “or” operator that says the first part will be True as soon as you

play the game, but the second is only true when the Timer has a lower value than the Top Score. Finally, we use a “set to” block to store the top score so that the game remembers it.

Our game is now complete! Use your imagination to improve the game. Create new scenes using what you learned in the previous chapter. You can even combine the story book with the game and add some story and character development. The game doesn’t have to be a simple obstacle course.

Chapter 7: Project 4 - Virtual Simulations

Scratch can do more than animate simple sprites or create games. It can also be used to build intricate simulations. For example, you can create snow, not just as a plain animation, but as a system that behaves exactly like snow in the real world. To demonstrate, we're going to simulate snow by making it fall from the sky, gather on the ground, and stick to various objects.

To achieve such a simulation we'll have a series of snowflake clones that will fall from the top of the scene to the bottom. They will also simulate movement, wiggling from left to right as they fall. No snowflake falls straight down like a rock. When the snowflakes reach the ground they will remain there and gather more snowflakes on top.

Setting the Scene

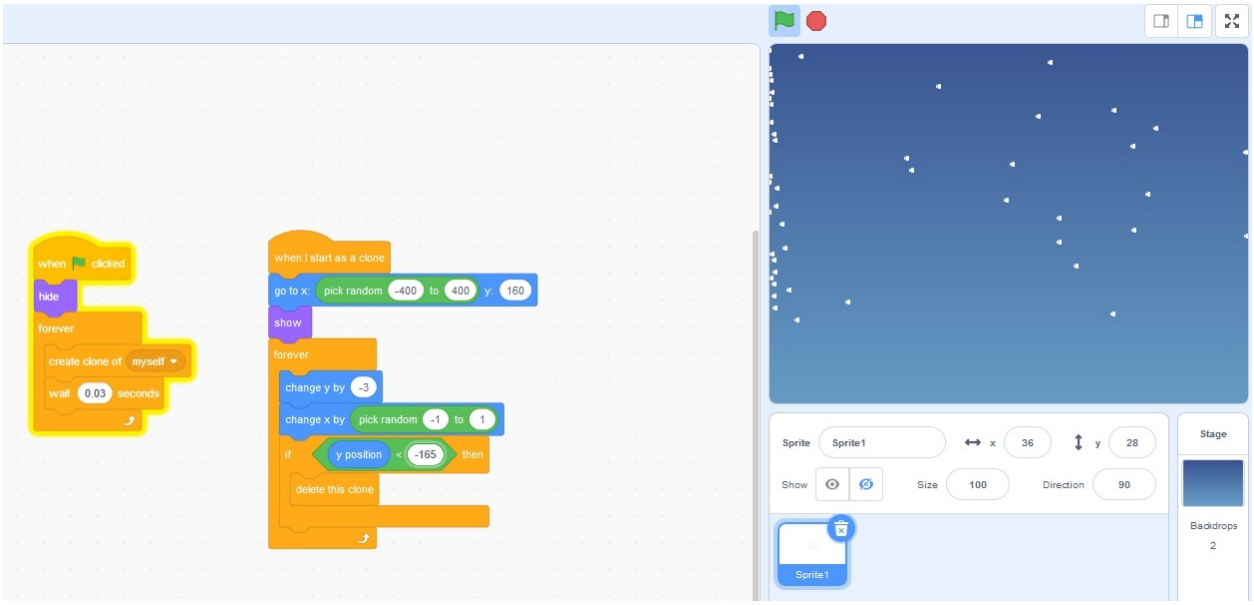
Let's start by designing our scene. Since it's going to snow, we'll want an appropriate winter scene. Of course, if you want you can make it snow on a volcano, or you can create a forest scene and simulate falling leaves. The choice is yours, but in this example we're going to start by adding a snowman to our scene. Then we're going to draw a snowflake by simply creating a white circle.

Launch a new project, remove the cat sprite, and then go to the paint editor. We're going to create the snowflake by selecting the ellipse tool with the fill color set to white. To draw a perfect circle, hold the "shift" key and then click and drag your mouse to create the circle. The snowflake should be really small, so draw it, place it in your scene view, and shrink it until it's the right size.

Next, we need a background. Go back to the paint editor and create a backdrop using the vertical gradient tool. We'll draw a simple blue sky, but the gradient will make it look a lot less plain because it will blend two colors. Pick your colors, for instance a dark blue and a light blue. Fill the scene with color.

Our scene is ready! Let's start simulating some snow!

Scripting the Fall of Snow



Click on the snowflake and switch to the scripts area. We need some code that will clone the snowflake. Otherwise we would have to create hundreds of sprites, and that's not an efficient way to work. All we need is one sprite and some clever code.

Start with a "when flag clicked" event, followed by a clear and hide block. This way the original snowflake will be hidden from view and we'll animate only its clones. If you didn't shrink your snowflake yet, you can do it now by using the "set size to" block. Next, add a forever loop that will contain the cloning block "create clone of myself" and a delay block of "wait 0.03 secs". This will create a new clone rapidly, but not quite instantly.

Now, we need another script that will tell the snow to fall vertically from the top of the scene to the ground and move a bit sideways to jiggle. Start with a "when I start as a clone" conditional that will only trigger the code when the clones are created. Tell the clones to start at the top using the "go to x: y:" block, but also add a "pick random" block inside it so that the snowflakes appear randomly. Otherwise, they will just appear on top of each other. Let's also control the size of the snow to make it more realistic. Not all snowflakes are created identically, so add a "change size by" block with another "pick random" modifier. Experiment with these values to see how it looks. Finish this part of the script with a "show" block to display the flakes.

Next, you need to connect a forever loop that will make the snow fall and jiggle. Inside the loop add a "change y by" block to control the fall of the snowflake with a value of -3, and a "change x by pick random -1 to 1" block to add a slight

jiggle. See how useful coordinates are? You can use them to animate stuff!

Finally, add an if statement that will make the falling snowflake vanish when it reaches the ground. The statement will look like this:

if y position < -175 then

delete this clone

So when the position of the snowflake in our scene is smaller than -175 on the y axis, the snowflake will be removed.

Take a short break to test your project and make sure it works as it's supposed to.

Making the Snow Stick to Objects

Snow doesn't just disappear when it hits the ground, at least not when it's cold enough. It sticks together and forms a layer. Let's do that in our simulation!

The first thing we need to do is make the snow pile up at the bottom of the scene. You would think it would be as simple as that, but Scratch actually has a limit on how many sprites can be on the scene at the same time. So we need to find a way around that. Luckily it's easy, because Scratch has a stamping tool that will stamp the clone to the scene, making it part of the background, and then delete the sprite. So all you need to do is add a "stamp" block inside the if statement before the "delete this clone" block.

The snow should now pile up, but it will only form a row. We need another if statement that includes a "touching color white" block inside it. Add that with the stamp and delete blocks inside it. To select the color, click on the color box and then click anywhere in Scratch where the background is white. This will allow the stamp command to add the clones on top of anything that's white in color.

But now we have another problem. The snow isn't stacking nicely to form a neat carpet of pure white snow. Instead, we're getting clumps that look like branches. To change that, we need to add an "and" operator to the "if touching color" statement and insert a "pick random 1 to 6 = 1" block. We're basically telling the program to roll the dice on which snowflake will stick. Whenever the program rolls 1, the flake will stick. You can increase or decrease the second value, which is 6, to experiment.

Next, you can continue working on your graphics. Draw various objects and characters so that the snow can stick to them. Maybe turn this into a winter scene part of your story book. The sky’s the limit!

Part 3: Glossary

Algorithm: A collection of instructions that fulfill a task. All computer programs are built using algorithms that run in a certain sequence.

Animation: Quickly replacing an image with another creates the feeling of motion. The more sprites you have and the faster they change, the smoother the animation is.

Backpack: This is a Scratch feature that lets us copy a sprite or other objects and store them without having them clutter our active project. This means that we can clean our project when we have too many active sprites. The backpack also lets us move our custom built sprites to other projects.

Bitmap images: This is a type of image that is formed out of a collection of pixels. When we enlarge a bitmap image we will notice a distortion because we're enlarging the pixels. See vector graphics to compare.

Block: Scratch is a visual programming tool which relies on the use of code blocks. These blocks work like LEGO bricks in the sense that you can combine them in order to create a script.

Boolean: This is an expression that can only be true or false, therefore we only have two possibilities. In Scratch you can recognize booleans because they have a hexagonal shape.

Branch: We say that a script is branching out when there is more than one choice. For example, in if/else statements there will be at least two possible decisions.

Clone: In Scratch, a clone is referring to a copy of a sprite that can have different settings and scripts. The clone can act independently from the sprite.

Condition: Statements can be true or false, or require certain parameters to be executed. In other words, a condition needs to be fulfilled for an action to be taken.

Coordinates: Two lines in space that cross each other, one vertical and one horizontal, will intersect at a certain point. We can find that point out by learning the value of the x axis and the y axis.

Costume: In Scratch, a costume is the image of the sprite. The word "sprite" is often used in development as a way to describe any kind of two dimensional

piece of art, but in this case it’s more than that. See sprites.

Ellipse: This is an oval shape used in the paint editor to draw round shapes. By pressing the shift button when using the ellipse, we can get a perfect circle instead.

Event: Any action which allows the program to react with another action or a response is an event. For example, the act of pressing the right mouse button is an event. We can use such an event to trigger an action only when it occurs.

Execute: Making a program or script run.

Function: A block of code that fulfills a specific task. Similar to an algorithm, it’s constructed to be used and reused without having to create it again.

Gradient: When we progressively switch from one color to another, we have a gradient color. Think of the sky during sunset that can go gradually from yellow to orange, blue, and purple.

Input: Any method of inserting information, whether through a mouse, keyboard, or microphone, is an input.

Integer: All whole numbers, including negative ones, are called integers. They cannot contain decimal points.

Library: In programming terms, a library is a collection of objects. They can be blocks of code, sprites, audio files, or costumes.

Loop: An action or function that repeats itself multiple times, or indefinitely, is known as a loop. By using loops, you don’t have to write or create multiple copies of the same code.

Operator: In Scratch, it’s a block that uses information in various ways to obtain a result. For instance, comparing two values to see which one is the greatest involves the use of a “greater than (>)” operator.

Pixels: A collection of tiny dots that come in different colors. They are used to form an image.

Procedure: Another word for function. Procedures are tiny programs that can be reused or recalled throughout the project to fulfil a task.

Python: A programming language that requires you to write the code yourself instead of using visual scripting blocks like Scratch. It is an ideal next step after mastering Scratch.

Random: This is a function that allows the program to choose between two

values in a way that cannot be predicted.

Script: A set of code that controls a part of the program. In Scratch, a script contains all the code blocks that run in the order they're arranged. We can have multiple scripts to form a whole project.

Simulation: Imitating a real action or event, such as raining or snowing. Accurately recreating every step needed to represent something in a realistic way.

Sprite: It's an image that can also contain a script, which is used to control it or transform it in various ways. Sprites are more than just graphics when we're referring to them in Scratch. Otherwise they are normally considered as 2D graphical art made out of a small number of pixels.

Stage: Scratch's game environment, or scene, where all the graphics are located and where the action takes place.

Statement: A command that fulfills an explicit task inside the script.

String: A collection of characters that can be letters, numbers, or symbols. In other words, a string is textual information.

Variable: An object where we can store information. We can then call a variable in other statements when we need that information and we can also manipulate it.

Vector Graphics: Images created from shapes that can be resized without losing quality, unlike bitmap graphics.

Conclusion

After working alongside this book, you gained enough Scratch knowledge to start working on your own projects. Start exploring what the community has to offer, team up with others, and practice everything you learned. Without practice, you can't make any progress. Continue perfecting the projects you already created, share them with others, ask for opinions, and continue to improve.

Scratch offers you all the graphical elements and audio you need to unleash your imagination without being a professional artist. But more importantly, it lets you practice real coding concepts and methods with the help of visual blocks of code. Once you feel like you've learned enough and you understand the fundamentals of programming, you can learn a new programming language. Python is a good place to start, and you should explore it because it relies on the same techniques you learned in Scratch, but it teaches you how to work with real code.

Programming isn't all about boring math, algorithms, and problem solving. It can be used as an outlet for your creativity. You can meet new people and start a coding club where you create cool games. It's a tool that's as fun as you make it. So enjoy the process and never stop learning.

Have fun!

Book Description

The world of programming can seem to be dull and boring, and it’s hard to keep children interested. That’s why Python is a good programming language to start with, as it is easy to learn and through it, children can express their creativity. This book in particular was designed to bring programming closer to its young audience, and inspire them to conduct their own research in the future. The unique and interesting examples used in this fun book will keep the reader’s attention at its peak.

In the chapters of this book you will find puzzles that will make you think and train your brain to work like a true programmer. By the end of the book, you will have a basic understanding which will get you started in the world of programming, and you will feel encouraged to go wrestle with your own ideas and code. Above all, *Coding for Kids in Python* will inspire you to grow and become an independent young programmer who isn’t afraid to continue learning.

Coding for Kids in Python will teach you how to use the fundamental data structures such as variables and functions. You will also learn how to organize your code and even reuse it in your future projects. Using loops and conditional statements will become a breeze, and the Python Turtle module will give you the opportunity to draw shapes and patterns.

With *Coding for Kids in Python* , you will learn basic knowledge which will help you create games, animations, programs, and web-based applications. The possibilities are endless and they should be available to everyone, including kids!

Coding for Kids in Python

A Step-by-Step Beginners Guide to Master Your Coding Skills and Programming Your Own Animations and Games in Less Than 24 Hours

Matthew Teens

Table of Contents

Introduction

So you want to learn how to create cool new applications to help people or use your creativity to build your own virtual universe. Perhaps you want to become the person who keeps us and our computers safe from hackers and other evil doers. What you really want to do is learn how to code, and this book is here to guide you and train you to become a good programmer.

Programming is often misunderstood. Many people think it's all about sitting in front of a computer all day and typing some mumbo jumbo that only makes sense to nerds. That couldn't be further from the truth. Programmers can use their tools to create better and more user-friendly websites, develop entertaining games, create life-like simulations, or give life to intelligent tools. Coding is much more than just staring at a monitor all day. In fact, many programmers and software developers have to travel and investigate the problems they're going to face and the solutions they need to find. This field involves a little bit of everything. Your job is to learn the basics and the tools you need to figure out what you're truly passionate about.

Having the ability to code opens a world of potential, and by working alongside this book you'll be able to unlock your hidden power. In every section you'll learn something useful and fun that you can share with others. You'll be able to read and write code well enough so that you can find others like you and work on a project together. Programming is in most cases a team activity, and you should encourage yourself to join others.

About the Book

For the last twenty years technology has been taking over the world. Everything you use in your daily life has something that someone had to program to make it work. Whether it's your smartphone, smart TV, AC, or thermostat, it works because a programmer told it what to do through code. That's why many schools around the world started teaching programming classes, but a lot of them still didn't catch up with the times. A lot of kids that might be interested in technology, just like you, might never get the chance to learn because classes and school books are outdated. This is where this modern Python programming guide comes in to teach you how to be a good coder.

It’s important to learn something that’s useful and that can be used in the real world. Nobody wants to learn something that only works in theory. After all, how many kids do you know who hate math just because schools don’t teach anything practical? This pushes people away. With the help of *Coding for Kids in Python* , you’ll learn things that you can apply right away! You’ll be able to develop games, applications, and participate in online communities where kids like you discuss programming languages and techniques. To achieve all that, you first need to work your way through these pages and explore the following topics:

1. **The basics of programming** : All programming languages work under the same rules and principles. To become a good coder, it’s not enough to learn a language and nothing else. Sure, you can do that, but then you won’t be able to progress later. This can be a problem since technology advances fast and you need to have the knowledge and skill to catch up and be up to date. To do all that, you need to focus on the fundamentals. You need to learn the basic concepts like variables, loops, conditionals, and functions. Once you know all that, it doesn’t matter if you don’t know the keywords for a certain language. A lot of beginners think that the hard part is learning the language itself, or in the other words the syntax. That may seem challenging in the beginning, but a new language can be learned in a couple of weeks if you master the basis of programming. So, if you want to learn Python, C++, C#, Java, JavaScript, or any other programming language, all you really need is to first focus on one to learn how coding works, then you can apply what you learned in the other languages.
2. **Programming languages** : There are many languages out there and each one has its uses and advantages/disadvantages. Some are very easy to learn, while others are so difficult that they’re left for the master jedis. Some are the best solution to developing games, while others work like magic when creating web applications. That’s why we’re going to explore the most popular languages so you have an idea about how they work, what they’re used for, and how you can learn them in the future.
3. **Python programming** : You’re going to start your programming journey by learning the fundamentals of programming through Python. Python is a powerful and versatile language that can be used for nearly anything. Coders and developers use it to create games and

applications, while data scientists and analysts use it to study vast amounts of data used to train artificial intelligence. Python is a real programming language used for real applications. It's not just something for kids. The reason why Python is taught in many schools is because its syntax, the code itself, is very close to English, and therefore easy to understand for a complete beginner. In this book, you'll learn all the basics of working with Python, from installing it on your computer to creating games.

4. **Game development** : Writing boring code is not a fun way of learning a programming language. Coding should be fun and exciting, and it certainly can be if you mix in your imagination with your ability to solve problems. Reading books about programming and coding languages just isn't enough. You need to practice, practice, and practice some more! The best way to do that is by creating games, and we're going to do that together.

The book is structured in such a way for you to manage to learn on your own. Your parents can also easily follow along as well if they want to join you on your journey. Nobody needs programming knowledge to learn how to code. All you really need is a bit of persistence, determination, and a love for solving puzzles.

Requirements

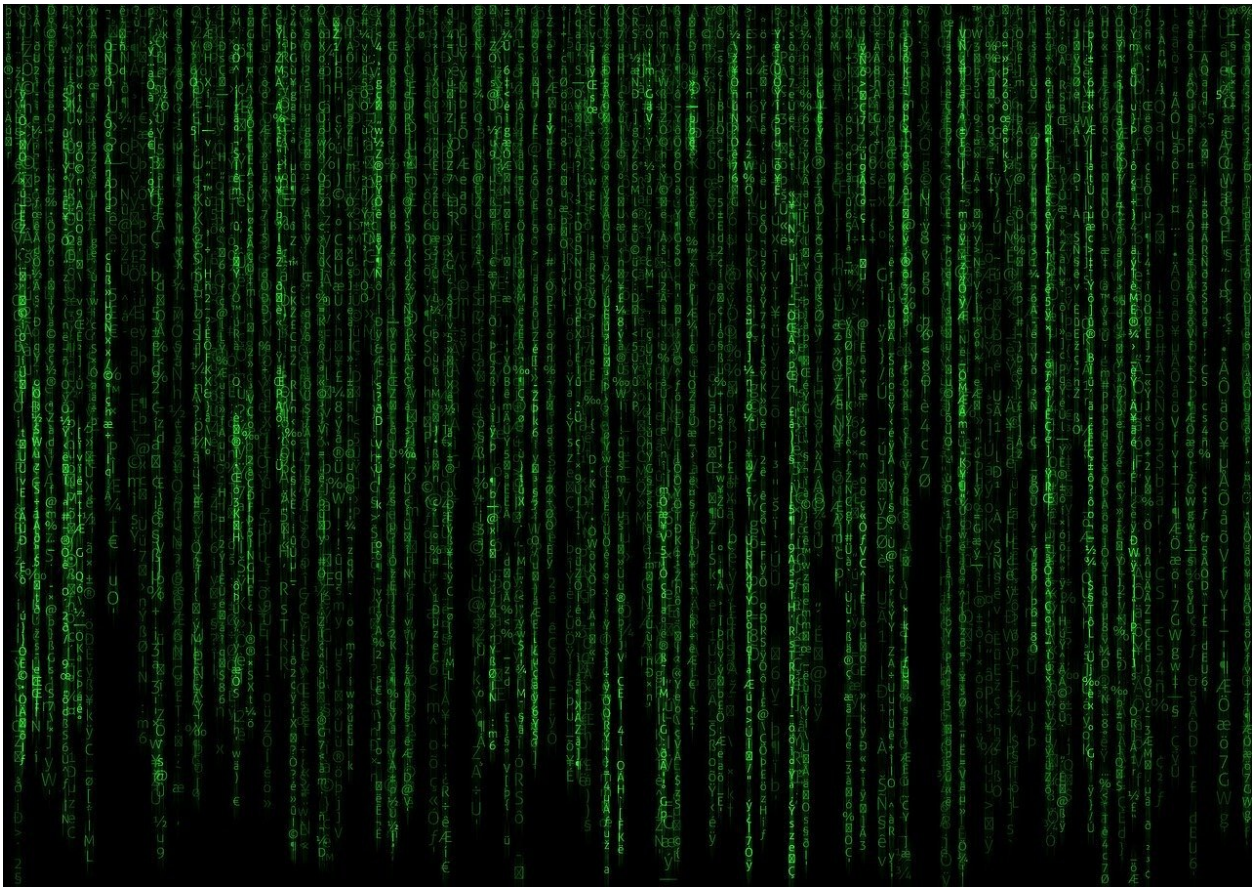
Learning programming languages doesn't require special talents or skills, but there are still some requirements that are obligatory or at least helpful. With that in mind, here's a list of things you'll need to get started:

1. An awesome book to help you get started. Well, you already have that, so scratch that off the checklist. Keep this book at your side at all times because you'll often have to double check things you aren't quite sure about yet. Don't worry if you don't get something right on the first try, that's perfectly normal. Just do your best, check with the guide as often as needed, and practice, practice, practice.
2. A desktop or laptop connected to the Internet. You can't write code without a computer, so hopefully you have one. You'll need to install a couple of tools as well, so having an Internet connection is required as well.

3. Don’t hate math too much. Don’t worry, you won’t be bored to death by doing a lot of math, but knowing some basic math is required to understand some programming concepts. Even if you really don’t like math, just think that if you suffer through it, you’ll enjoy the gift of coding even more.
4. Basic computer-use knowledge. In other words, you should know how to use a web browser and how to download and install applications. Beyond that, knowing how to research a bit on your own can help you a ton. Programmers have to be good at using Google because many solutions to their problems have already been found by other programmers. So, there’s no need to reinvent the wheel.
5. The ability to focus on the details. If you’re already good at noticing the details, you have an advantage over others. Programming basically involves solving puzzles. The more you notice the small things, the fewer errors you’ll have and the better your program will run. Whenever your application or game doesn’t work as planned, you made a mistake somewhere, which can either be a simple misspelling error, or it can be caused by an error in your logic. So, be patient when you have coding problems to solve. There’s an answer to everything if you can focus well enough to find it.

Most importantly, you need to have the motivation to learn! Programming can be fun and rewarding if you’re willing to put in the effort. You won’t learn how to write code in just a couple of days, but if you’re determined to keep reading and practicing, you’ll become better and better over time. So have some fun while you’re on your long journey! Use your imagination to come up with cool new ideas, or try to recreate something that’s already fun for you and give it your own twist. Programming is as fun as you make it.

Part 1: Introduction to Coding



Chapter 1: Programming Fundamentals

A long, long time ago, before the age of the Internet, computer programming was something done by wizards. The coding languages were quite difficult, the code editors didn’t tell you when a mistake was being made, and you even had to manage low level computer elements like memory allocation. Most of these things are now automatically handled by the system. Even code editors tell you when you’re wrong by highlighting your mistake and telling you what kind of mistake it is. You’ll see suggestions that could fix the error and enjoy autocomplete features used to fill in the keywords you're typing. Programming is a lot more accessible nowadays and it no longer forces you to dive too deep into mathematics (though being a math wizard is always helpful).

Leave your worries about the challenge and difficulty at the door because all you need is a bit of determination and love for technology. So, let’s start learning what it truly means to program and write code.

What’s a Program?

Your computer, phone, and probably even your microwave oven are all controlled by programs. All of our technology is controlled by something that

contains a long list of instructions and commands. Programs are just sets of instructions that tell a computer what to do. We normally think of computers as really smart tools that know what they're doing better than us lowly humans, but that's not the case. Computers are actually really dumb. They have no idea what they're doing if a programmer isn't there to tell them exactly what needs to be done. For example, if you told a computer to peel a banana, it wouldn't know how to do it without you first telling it to pick it up. Yes, they're that dumb. As programmers, it's our job to teach them everything they need in order to perform certain tasks for us.

As a programmer, you're going to use your creativity and logic to write sets of instructions and save them as programs. These instructions are written in special code that the computer understands. Actually, the code that we understand and write is first translated by the computer into machine code (a bunch of 0s and 1s). It would be crazy for us to write machine code because it would take a lifetime to create anything meaningful that way. That's why we use programming languages like Python instead. Its code is similar to English, and then a tool called an interpreter translates that code into something that the computer understands, namely machine code. There are many other programming languages out there that work the same way, like C#, Java, Ruby, and others. We're going to stick to Python for the most part because it's more beginner-friendly.

Python is a general programming language that can be used for nearly everything. It is a text-based language which uses keywords that are basic English words or abbreviations. This means you'll have an easy time writing the code because you'll be able to quickly recognize and memorize the language keywords. Besides being easy to use and freely available, you will also find many online communities of student programmers just like you. This is important because you will encounter problems that might be too difficult to solve. This is when teamwork comes in, and easily finding other Python programmers is important.

Is Programming for Everyone?

If you're worried that you might not have what it takes to become a programmer, stop. If you're terrible at math, you can become a programmer. If you're not that good with computers yet, you can become a programmer. If you're a slow learner, you can become a programmer. Anyone can become a programmer!

There’s a myth that you need to be a math wiz or have some special talent to understand code, but none of that is true. All you need is to study and practice.

For instance, if you have an artist’s soul instead of an analytical mind, you can combine your knowledge of coding with your overflowing creativity and create games, stories, and virtual worlds. Coding is often needed to create cool animations and visual sequences, not just the structure of a program. Or maybe you’re in love with mathematics and physics instead, in which case you can apply your programming skills to analyzing data and teaching an artificial intelligence to act like a human.

Programming is a science as well as an art, and most importantly you need to think logically so that you can develop a clean program. In other words, you need to stop and design your program before writing any actual code. Then you have to think of each step like solving a puzzle. Don’t focus on the program as a whole, but instead break everything down into a series of steps that the computer needs to take.

Understanding Programming

A coder needs to understand the process of designing and creating programs. To do that, you can imagine a program as a recipe or furniture assembly instructions. Programs are built out of many components that rely on other components to work. They’re assembled by following a certain order. If this sounds a bit complicated, let’s compare programming to having breakfast.

Does your bowl magically fill with milk and cereal in the morning? Unfortunately, no. There are a series of steps that are taken in a very specific order. You grab the milk from the fridge and pour it over your cereal ,right? Well yes, but that’s still missing several steps. The first thing you do is grab the cereal box, open it, then pour it in the bowl. Then you stop pouring when the bowl is full, head to the fridge, open the fridge door, take the bottle of milk, open it, and then pour it over the cereal. Even now we can still add a couple more steps to truly complete the program. For example, we didn’t include the “twist the bottle cap” detail when opening the milk bottle. Remember, computers are dumb, and we need to explain everything in fine detail, otherwise the program won’t work correctly.

As you can see, a program is nothing but a sequence of instructions. It’s like

working with LEGO bricks. You imagine something cool, then you place a building block on top of another by following the plan. When you code something, you should do the same.

Break the program in a series of actions. These program sections are known as algorithms, and each one of them has a part to play in the grand design of your program. For instance, the act of opening the fridge, grabbing the bottle of milk, opening it, and pouring the milk in your bowl would be an algorithm that's part of the "breakfast" program.

To create these algorithms is part of your job as a coder. All of them are essential building blocks that you connect to other blocks. With that in mind, here are the three main algorithm principles you need to understand before getting started:

- Programming sequence: This represents the order in which your commands are processed by the computer. As described in our breakfast example, all programming languages will execute the code by following the order in which the instructions were written. Therefore, your program isn't going to pour milk before you fill the bowl with cereal.
- Decisions: Programs can choose their own path based on certain conditions. In other words, your program can decide what to do and have a limited ability to think on its own. This works when you tell the program to do something only when a specific condition is met. For instance, let's say your program can choose between multiple breakfasts depending on which day it is. This is done with the help of conditional statements that follow this formula: if x condition is true, then y action will be executed. So if we have a Monday condition, the program will choose to execute the cereal algorithms. In other words, if it's Monday, then cereal will be served and all the cereal related instructions will run while the eggs and bacon instructions, for example, will be ignored unless it's Tuesday.
- Repetitive actions: Some instructions have to run multiple times or even for the entire duration of our program. These repetitive actions are called loops. In all programming languages we have the ability to write a command inside a loop so that it continues running over and over again until we tell it to stop. This way we don't have to type the same line of code several times throughout the program.

These main principles can be applied in all languages. Whether you're using Python, C#, or Java, they all respect the same rules.

General Coding Guidelines

Before you start coding, you should learn a series of good coding habits so you can write clean, high quality code from the start. Many programmers form really bad habits when they start out and then they spend years trying to shake them off. Working with clean, beautiful code will make you a happy programmer and your friends and teammates won't have the urge to strangle you, which is always a bonus.

As a programmer, you'll work on long projects that will take some time to complete. Others you will abandon for a while and then revisit later. In either case, if you don't follow a series of good coding practices, you'll end up staring at your code without having a clue what it does.

But why are you supposed to learn these guidelines before even learning how to write a line of code? Because they apply to all programming languages. So, invest a few minutes now to read these sections, and save days or weeks of panic and frustration later.

Readable Code

Creating programs and games takes time. Your code will also change over time because as you work on your project, you'll realize you need to go back and make a change here and there. After enough time passes, you're going to feel lost when you go back to read your earlier code. This is just one of the reasons why code has to be written nicely and be as descriptive as possible. If you're working with a friend or two, or if you want to share your code online with other learners like you, you'll want them to be able to figure out what you were doing. There's a famous saying among programmers and developers that goes something like this:

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live." - John Woods

Sounds a bit scary, but it certainly motivates us to be better coders. In programming, there are always multiple ways of doing the same thing. Each programmer has his or her own style and techniques. But at the end of the day, other fellow programmers have to be able to understand anyone's code. Since several parts of the code are named by you, such as variables and functions, you can easily confuse others by using made up acronyms and/or meaningless words. Here's an example you should avoid:

```
name1 = car.brand

name2 = car.color

name3 = car.type
```

When you create several variables and label them with generic words like "name" or "label", there's no way for you or other programmers to check and see if these variables are correctly assigned and used by the program. Your programming friends will end up hating you and avoiding you because they have to waste time to figure out what "name1" means and what it does. So instead, you should write code like this:

```
brand_name = car.brand

paint_color = car.color

car_type = car.type
```

Now you don't have to analyze your code for half an hour to understand the meaning behind the variable, function, or other parts of your code. You read the name and the message is clear. So to write clean code, all you need to do is ask yourself one question:

Does this name tell me what the code does without looking into more code?

If the answer is yes, then your code is clear and descriptive.

General Naming Conventions

Naming your variables and functions in descriptive ways isn't quite enough to have clean code. There are a set of rules and guidelines for you to follow to write beautiful code. For example, you can name the variable that contains information about the color of your dog in the following ways: dogColor, dog_color, DogColor, dog_Color. We can read and understand all of them without having to read the rest of our code. But even though you are free to make some choices, there are a few rules and recommendations that you must follow:

1. **Spaces are not allowed** : You can't add a space in the middle of a variable name even if it's made out of several words. If you do, the program will see each word as its own element and that will cause an error. That is why we write all the words together, no matter how long the name is. However, we do use other methods of making the code readable because we don't want this monstrosity "meleeweaponswarrior". It makes us dizzy just trying to read it. Fortunately, using capital letters for each word, or separating them using underscores, makes the variable name a lot easier to read. So instead of that letter soup we would have "meleeWeaponsWarrior" or "melee_weapons_warrior".
2. **Don't use symbols** : You can use underscores (_) as word separators when you give something a longer name. Other than that, you shouldn't be using anything else other than letters and numbers.
3. **Always write the name starting with a letter** : This is a rule since in most languages you can't name your variables starting with a letter. Besides, why would you do that anyway? It would make everything harder to read. You can have numbers at the end or the middle of the name if you need them. Most programmers like to start their variable and function names with a lowercase letter and then use underscore separators or uppercase letters for any other word that follows.
4. **Stick to one naming system throughout the project** : This isn't a rule, but it's heavily recommended. If you like to use underscores, then name all of your variables and functions using underscores

whenever you need to use more than one word. Pick any system you like and then just stick to it. This way your code will look clean and flow nicely, making it easier for anyone to read.

5. **Some names are restricted by the program** : All programming languages have some restrictions when it comes to names because they're already used by the program's library. For example, in Python we have the function keyword "print," which tells the program to print a letter, word, or sentence on the screen. Since the name of this function is already reserved, you can't create a new variable or function and call it "print". This will lead to errors and it will also confuse other programmers because they won't be able to see what's happening unless they start digging through your code. Fortunately, most modern code editors will warn you when you use a reserved name, so you don't have to memorize everything first to know what to use or not use.

Follow these rules to create simple, easy to read code. By following these guidelines, whether you program in Python or C++, you'll be avoiding a lot of headaches, especially when working with friends and teammates.

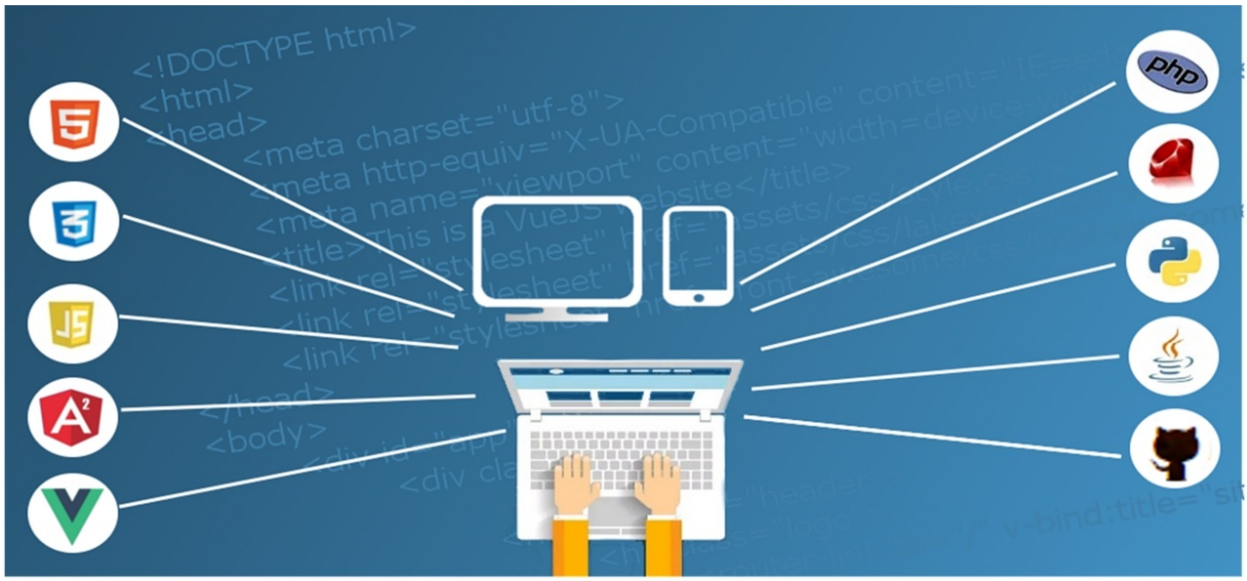
Chapter 2: An Introduction to Programming Languages

Every beginner programmer who's just starting out wonders what programming languages are out there, what they are used for, and which ones they should learn. There are many options and it can be a hard decision to make. The truth is that all languages have a purpose, and some are better than others for certain tasks. In the end, the language you should use is the one you know best, or in this case, the one that can help you learn programming principles and techniques the easiest.

You picked up this book because you're interested in Python, but that doesn't mean you won't learn anything else. Python is versatile, but there are other languages better suited for certain things. Most programmers learn multiple languages over time and they even combine them to create their dream game, app, or the most modern, cutting edge website. So in this chapter, we're going to briefly go through the most used programming languages to give you an idea

about what you can learn together with Python, or after you've mastered the basics of programming.

Web Development and Design



Some programming languages are necessary or best used to create websites and online applications. If you're interested in this side of development, to get started you'll need to explore HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript. The first two are actual programming languages because they can't be used to create programs that actually do something. They don't contain any programming language. In other words, you can't use them to give a program the ability to decide. You can't even use them to figure out how much two plus two is. So what are they used for and how do they work?

HTML

As mentioned, this is not actually a programming language. A lot of beginners make the mistake of calling themselves HTML programmers, but that just annoys some coders. This language is used to build the foundation of every single website. It enables us to use links to navigate from webpage to webpage.

HTML is a markup language because it uses tags to mark certain information. Each tag tells us something about that information and where it's supposed to go on the website. Web browsers like Chrome will analyze these tags and determine how the website should be displayed based on their data. So HTML is used to talk to the browser and make it understand how the website works.

Here's an example of how HTML code looks:

<!DOCTYPE html>

<html>

<body>

<h1> All the text written here will appear in heading 1 format </h1>

</body>

</html>

As you can see, the code is expressed in tags written between angle brackets (<

>). The tags tell the browser where the content (body) of the website begins and where it ends (/body). In between, we have some text that appears formatted using heading 1 according to the h1 tag. There's no programming logic, no special conditions, and no calculations.

HTML is only useful when designing or developing websites because it's necessary for the base structure of a website. So if you think you might be interested in making the Internet a better place, you should consider starting with HTML at some point in the near future.

Cascading Style Sheets (CSS)

HTML isn't enough on its own to create a nice website. You could use it on its own technically, if you really want to achieve the look of websites from the 90s (yuk!), but to create a nice website you need CSS. As the name suggests, CSS is used to refine a website's style. In other words, it allows you to play with colors, fonts, layouts, effects, and a lot more. Similarly to HTML, CSS isn't a proper programming language either because it contains no programming logic. You can't use it to make intelligent applications since it can't do any math, it can't compare a value to another, and it can't set any conditions. It just let's you

beautify your bare bones HTML website. Let’s take a bit of HTML code and then apply some CSS to it, to get a better idea about how it all works:

 Coding is fun

The “p” tag is used to label paragraphs. What we have so far is a bland bit of text, so let’s make it more interesting with CSS:

p { color: blue; font-weight:bold; }

In this case, we’re saving the code as a CSS file that will be used by our HTML file in combination. The code itself is quite easy to understand. We tell the program that we want to turn the paragraph text (p) from the default black to blue and to bolden the text. This instruction is declared inside curly braces.

We can also use CSS inside the HTML document. In that case we don’t need the curly braces anymore, or the additional “p”. Here’s how it works:

<p style = “color: blue; font-weight:bold;”> Coding is fun

The CSS command is inserted directly in the HTML line and it applies only to it. Any other paragraphs that follow won’t have this style unless we declare it again.

HTML and CSS go hand in hand to create a website, but they can’t add any interesting dynamic behaviors to the website because the languages don’t have the ability to process logic operations. For that we need a third tool, like JavaScript.

JavaScript

Now we're getting into some real programming languages. JavaScript is the tool you need to make your websites pop out and do all sorts of cool things. Just don't confuse this language with Java because they're not the same thing.

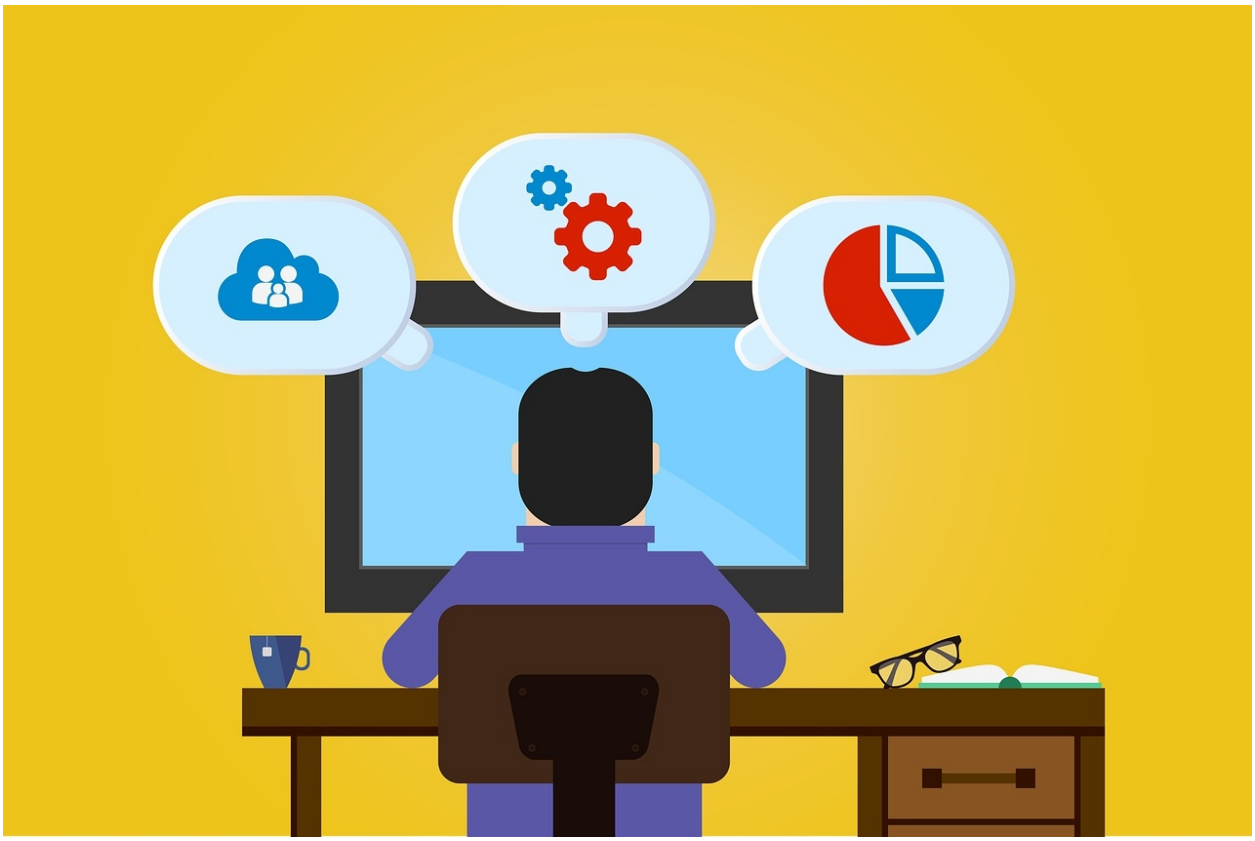
JavaScript operates inside the web browser, and because of that you'll hear from some programmers that it's a scripting language and not a programming language. That's partially correct, especially when comparing the language with more traditional ones like C++, but it's still used to add programming logic to a project, so that makes it a proper programming language.

Without JavaScript added to HTML and CSS, we wouldn't be able to add buttons, special effects and animations, or create intelligent user account forms and panels. Well, we could with other language alternatives like PHP, but the point is, we need a fully fledged programming language in the mix.

Just like Python and other languages you'll learn about, JavaScript allows us to create our own variables and store data inside them. We can also define functions, use various modules and libraries to extend the usability of the program, and even create special events that only happen when the user clicks on something specific. All of this is used to make a modern dynamic website.

So, if you think you want to design awesome websites some day, start exploring HTML, CSS, and JavaScript once you've mastered the basics of programming.

Multipurpose Programming Languages



There are hundreds of programming languages that can be used to create applications and games, or to program robots, medical equipment, and so much more. Choosing one or several can be difficult, but as you progress and learn more about programming, you'll realize you need to explore more tools. That's why in this section you're going to be introduced to a few of the most popular programming languages that are as powerful and versatile as Python, and just as frequently used.

C#

Pronounced C-sharp, this language isn't as easy as Python, but if you learn your programming fundamentals with Python first, you'll be able to learn it quite fast. C# is a powerful language that can be used to develop applications as well as games. In fact, if you're dreaming about becoming a game developer, C# is a great option since it's preferred by many indie game developers. Unity, a free game engine used to create high quality games, virtual reality projects, and

more, uses C# as the programming language that runs everything. There are thousands of people that use this engine to create games, and even big companies like Blizzard use it to create things like Hearthstone. So if you feel like creating cool games is something you want to do in your life, C# is a step in the right direction after Python.

C# is a more complex language. Even though its syntax is also similar to English and fairly easy to understand, the language isn't as forgiving as Python. For example, it takes longer to make the code more readable. In C#, curly braces are used to separate code blocks. If you don't pay attention to them, it's easy to make your code unreadable. Python, on the other hand, doesn't need any curly braces to separate the commands because it relies on indentation. This means that Python requires some empty spaces at the start of the line of code to know that that line is part of a certain block of code. Depending on the code editor, these spaces are automatically placed, so all you need to do is write code.

In addition, when declaring variables like certain types of numbers and text, Python automatically detects them. You don't have to tell the program what kind of information you're declaring. C#, on the other hand, needs to know or it won't be too happy. Here's a simple comparison between the two languages to give you an idea of how they look like:

```
using System;

namespace HelloWorld

{

        class myApp

        {

        static void Main (string[] args)

        {

        Console.WriteLine (“Hello, World!”);

        }

        }

}
```

This simple program written in C# prints the phrase "Hello, World!" on the screen. As you can see, the program requires quite a few elements to do something as simple as that, and you probably don't understand what they do

just by reading the code. You already need to understand what a namespace is, what a class is, and what keywords like static and void mean. The code arrangement is also something that needs to be carefully performed.

Now, here's how the exact same program looks in Python:

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

That's it! No need to bother with namespaces and static void methods to display a message on the screen. All you need is the print command, which is equivalent to WriteLine in C#, and the message you want to print. You can definitely understand what the program does even if you know absolutely nothing about coding.

This is why Python makes a better starting point when learning how to program. You can focus on the programming rules instead of becoming frustrated with complicated syntax. But this doesn't mean C# isn't a great programming language. All languages have their advantages and disadvantages, and it's up to the programmer to decide what to work with for a specific project.

Java

Java is another multipurpose programming language that can be used to create web, mobile, and desktop applications, and more. It's not as suitable for games as C# is, but it's a popular tool used in many tech fields. But don't mistake Java for JavaScript. They're nothing alike.

In some ways Java is similar to C# and if you learn one, you'll usually manage to quickly learn the other as well. It was built to be more beginner friendly than the languages from the C family (C, C++, C#) but it still requires a lot of code for each command. However, Java is immensely popular just like Python because it powers so many applications. It is considered a faster and more efficient language than Python and because of that it's used more often for web-based or network-based applications.

With that in mind, here's our Hello World program written in Java, just to give you a general idea about the language:

```
public class MyApp {
        public static void main (String[] argos) {
        System.out.printIn ("Hello, World!");
```

```
        }
}
```

Notice how it's so similar to C# but slightly simplified. The syntax is almost the same, and Java also uses curly braces to arrange the instructions in blocks of code. However, it's still harder than Python and it requires more lines of code to create even this simple program.

C++

This ancient language that comes from the 70s is a very complex language that is still used by many companies today, such as EA Games, CD Projekt Red, and Microsoft because nothing can really beat it when it comes to resource efficiency. C++ is used to create and manage large programs and most of the high end games you play. It's not recommended for you to start diving into this language until you preferably master another like Python. It's very easy to mess up and in most cases C++ doesn't tell you when and where you made the mistake. The programmer has absolute freedom with this language, but that also means they have the freedom to struggle to fix problems.

Here's our Hello World program written in C++:

```
#include <iostream>

using namespace std;

int main () {
        cout << "Hellow, World!";
        return 0;
}
```

Chances are you don't understand what's happening here. You can see it's somewhat similar to Java and C#, but the language itself is difficult to understand without some C++ knowledge. That's ok, we're not going to dig deep into C++. The most difficult part about this language is the fact that many computer processes need to be manually managed. Python, C#, and other more modern languages manage these things automatically. However, this is partially the reason why C++ is a more efficient language and that's why it's popular. It allows you to manually tweak low-level computer processes and systems like memory allocation.

C++ might sound scary, but it can be learned even by kids. However, it can be a long, difficult road until you can create something. So the general advice is to start with something easier and work on enough projects until you become good at using a programming language. Afterwards you can progress and explore other tools with more courage and determination.

Python

Python is in fact a unicorn. It's a programming language that's simple enough for kids in elementary school to learn, but complex and powerful enough for data scientists and machine learners to use for their projects. Python has been used for the last three decades on a wide variety of projects, ranging from mobile applications to games and medical software.

In programming terms, Python is a dynamic, interpreted language. In other words, we don't need special variable, function, and parameter declarations like we do in C# and Java. This is the main reason why Python code is always a lot shorter and easier to read. Python automatically keeps track of the types of values we insert, and anything that doesn't make sense to the interpreter will be signalled as an error or warning.

Python also comes with a large number of easy to use modules and libraries that allow us to extend its usability. For example, if you want to draw simple graphics by using Python and nothing else, you can import the Turtle graphics module. There's no need for other tools that are outside of the program, unless you need something more complex like high quality 3D graphics.

Finally, Python has a rich community of young learners. As mentioned, this is a language that is being taught in many schools around the world. Unlike other languages that are used almost exclusively by professionals and adult students, Python has a rich young audience. You can learn the language while also making friends that are your age and have the same interests as you. Even if your family or close friends aren't interested in programming and technology, there are many online communities out there that welcome anyone willing to learn and create something with Python.

Chapter 3: Coding in Python

Python is a free programming language and tool which you can download, install, and use for anything you want. In this chapter you're going to learn how to do all that and more. To work on projects, you first need to understand how the Python code editor works, and how to use the interface. Let's get started!

Installing Python

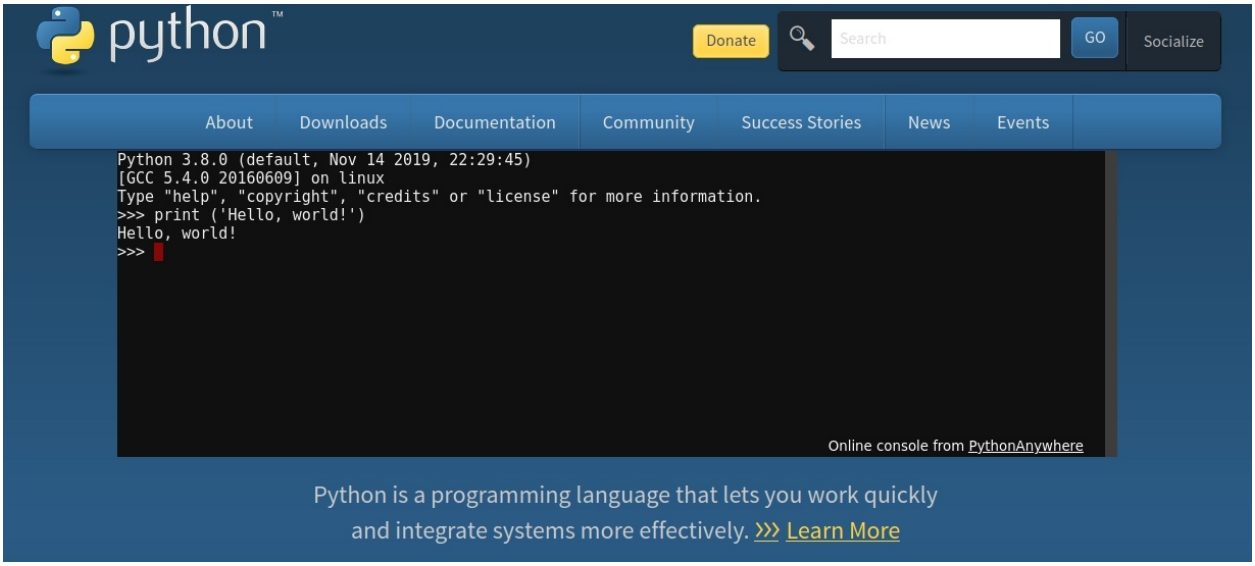
To install Python, head over to the official homepage by following this link: https://www.python.org/ . Click on the Downloads menu, and select the operating system your computer's running on. You can install the program on Windows, Mac, and other platforms. If you have a Linux computer, however, you already have Python and all associated tools installed.

Once you're on the right downloads page for your OS, you can click on the latest Python 3 release, which at the time of writing is version 3.8.5. Make sure you select Python 3 though, because there's also Python 2 which is an older version. You can use it to learn the language of course, but it's a bit outdated and certain features are different from Python 3. In this book we'll stick to Python 3 because it's new and it has a longer shelf life.

Once you click on the version you want, navigate to the bottom of the new page and go through the list of files you can download. Choose the one that matches your operating system as well as its architecture (32-bit or 64-bit). For example, if your computer runs on Windows 32-bit, you need to download the Windows x86 installer. For the 64-bit version, you'll want the Windows x86-64 installer. The same goes for Mac, except that both versions come in the same installer package, which is the macOS X 64-bit/32-bit installer. Select the right installer and then just follow the guide by leaving all settings on default as recommended.

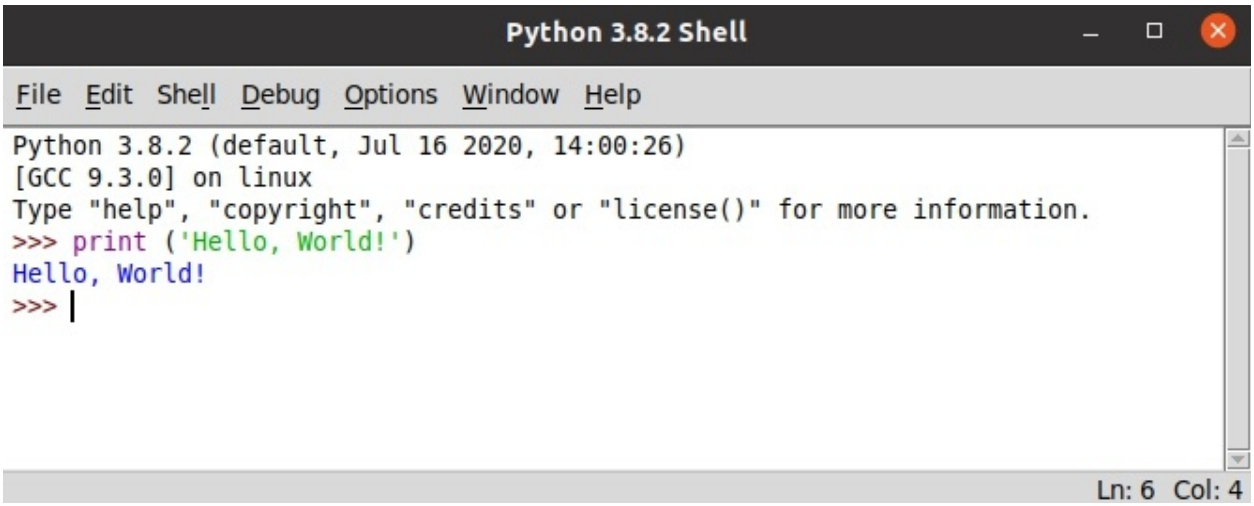
Python is now installed together with a code editor called IDLE. This is where you're going to write your programs and save your program files.

Alternatively, you can use the online Python tool found at https://www.python.org/ when clicking on the ">_" button. This launches a basic code editor that in fact is just a command console.



You can write code and see if it runs, but you can't save it. This makes it useless for more complex programs and games, but it's great for some quick practice. The console executes the code as soon as you hit the Enter key, and you'll see if it gives you the result you want or an error. Just remember that you can't go back and edit any code line, you can just write new ones. So for anything that takes more than a few minutes of practice and exercises, use IDLE instead.

The Code Editor and Its Interface



Take note that when you run the IDLE code editor it opens a shell window. What's a shell you ask? A shell is a basic command line user interface that lets you interact with the processes and services made available by the operating system. The term "shell" refers to the exterior layer, or interface, of the operating system. This means that we can write instructions that use or manipulate these processes and services, but we can't save what we're doing. The shell window works exactly like the online command line interface found on Python's website. You can write code and as soon as you hit the Enter key, it's executed. The shell is useful when testing out some code to see if it gives you the result you want. When you work on a larger program it's sometimes easier to check for errors this way.

Before we continue, let's analyze the shell in a bit more detail using the image above. Here's what we have:

1. Take note that the window is called the Python 3.8.2 Shell. This tells us we're working inside the shell and the code can't be saved. This minor detail is important because once we close the shell, the code is gone.
2. The first three lines in the shell are automatically generated and they give us some information about the operating system we're using and the version of Python.
3. Finally, we have our code. In this case all we did was create our Hello World program once again. Go ahead, try it out on your own. In the shell the output (result) appears in blue. If you make any mistakes, you'll see an error written in red with some details about what triggered it. Code editors use various colors that represent certain parts of code. We're going to talk more about them soon.

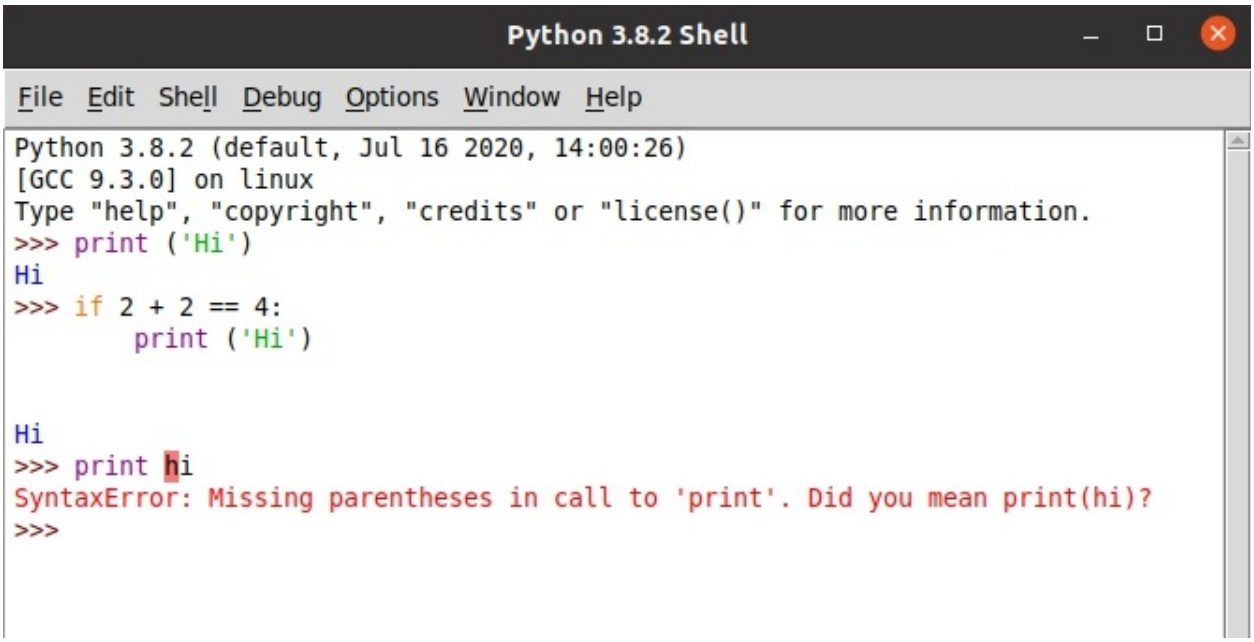
So far we wrote some code, but we can’t save it. That’s bad because we can’t just work on a project for hours and then not save it. We wouldn’t be able to finish anything. What’s the solution? The editor window.

IDLE has two types of windows, namely the shell window and the editor window. The editor window is the one that allows us to write code and save it as a Python file that ends in the .py extension. To open it, click on File and select New File, or use the Ctrl + N shortcut. This launches the editor window and creates an untitled file. This simply means that the file wasn’t saved. To save it, click again on File and navigate to Save As. This will open a new window where you select the name of the file and the location where it’s going to be saved. You should create a dedicated folder for each project to keep things in a neat order and make sure the save file extension is set to .py, which means it’s a Python program file.

Now, write the same Hello World command as before. You’ll notice that the result doesn’t appear anymore as soon as you type it. Remember that the editor window doesn’t run your code immediately. To try out your program, you have to click on the Run button on the navigation bar and then hit the Run Module option. To skip all of that clicking, you can also just press on the F5 button on your keyboard. A shell window will now open and show you the result of your code. If everything runs smoothly, save your progress by clicking on the File menu and selecting Save.

Python Color Coding

Most code editors, including IDLE, highlight keywords and parts of your code using certain colors. Each color represents a programming element. This makes code a lot easier and faster to read because the colors immediately tell you whether you're looking at an output, a string, a built-in function, or an error. Here's an example:



Notice how the "print" command appears in purple. We discussed earlier that programming languages have built-in commands that you can't change. In Python, these commands are represented by the color purple.

Next we have the strings (text) that are color coded using green. First of all, we recognize strings because they're written in between quotation marks. Secondly, we know they're strings because they're green. If you'd forget to add the closing quote, the word "Hi" would still appear in black. Everything that's black means it's a name, number, or symbol. Most of your code will be black as a result. Just look at the 2 + 2 operation, for example.

Next, we have the orange color that is used to represent keywords like "if", "from", "else", "import" and many more. Take note that there's a difference between built-in functions like "print" and "list" and keywords like "if". The keywords don't contain any information or set of instructions inside them. The functions do. A function acts like a shortcut. Instead of writing a collection of statements that explains to the computer how to print a sentence, the statements

are declared within a function. Then that function is used whenever needed by just referring to it.

The output of our code, or the result in other words, appears in blue. It doesn't matter if it's a number, comparison result, or string. All results are represented by the color blue.

Finally, we have the most dreaded color, which is red. You can see in our example that we tried to print the word "Hi" without correctly declaring it as a string inside brackets and quotation marks. This throws an error color coded in red. All errors and warnings appear in red, and most code editors will also highlight the part that's causing the error. In this example, the word "Hi" is highlighted in red and IDLE is offering us a suggestion that will fix the problem. We'll talk more about errors and what they mean in a later section.

Working with the code editor is much easier than the online interactive shell, or other text editing software like Notepad, because of color coding and error signaling features. Once you gain some experience, you won't even have to read your code because the colors already tell most of the story.

Declaring Variables

You installed Python, became familiar with the interface, and now you're fully prepared to dive into some serious programming. Your first stop is "Variables."

Throughout the book we mentioned "variables" several times, but we didn't really discuss what they are and what they do, except very briefly. In most programming languages like Python, C#, and Java, we use variables as a way to store some data, give it a name, and then use it as often as needed inside our program. Instead of writing all that data multiple times whenever we need it, we just call the variable instead. Variables come in different flavours and they're used often, so make sure to spend some time with them to get used to working with them.

Variables are declared and labeled by you, the programmer. This means that you have to give it a name by following all the naming rules and conventions we discussed earlier. After declaring the name of the variable, you have to assign it some data, or values. Let's take a look at an example:

player_sword = ('With this mighty sword I will crush my enemies!')

It’s as simple as that. We created a “player_sword” variable, which tells us the value stored inside it has something to do with the player’s sword in a game we’re making. In this case we’re storing a string. Whenever we want our player to shout that bit of dialogue, we make a reference to the variable instead of writing the entire string again, like this:

```
print (player_sword)
```

There’s no need to repeat a line of code multiple times when we can just store it once inside a variable and then call that variable.

Variables can be used to store any kind of information. We can have any type of numbers or operations with numbers, strings, lists, or anything you can think of. But, each type of variable has to be used by following the rules that apply to the information it contains. For example, we can’t perform certain mathematical operations using string variables. Let’s discuss the variable types you’ll be using.

Numbers (Integers and Floats)

Numbers are frequently used in programming for calculations, comparisons, decision making, and more. Just think about any game you play where numbers are used. The player has things like health, mana, energy, experience, and levels, to name a few elements. All of these stats also apply to various enemies the player encounters. To define all of these values, you need to work with numbers.

In programming, there are two main types of numbers: integers and floating-point numbers (also known as floats). The integers are all whole numbers, both positive and negative, like 3, 100, and -22. Python automatically detects whether a variable contains an integer value, but in other languages like C# the integer is declared with the “int” keyword, and because of that they’re often called ints out of laziness (or optimization as some might say). On the other hand, floating point numbers are decimal numbers and fractions, like 2.5 or 122.345433. Floats are usually used when performing some very accurate calculations. Most of the time, especially if you focus on creating games, you’ll be using integers instead.

Numbers are often stored inside variables which are then used in more complex operations. In a game, for example, you wouldn’t just write “100-10” as an operation that determines how much health a player loses when being hit with a rusty sword. You would first store the 100 value inside a “player_health” variable, and then create another variable for the rusty sword damage. The two variables are then used in various operations like reducing the player’s health

when hit, triggering an animation, a sound, and anything else you see in most games. This means that you're going to be doing some basic math using variables. Variables that store numbers can be used in mathematical operations like addition, subtraction, multiplication, and division. Here's a simple example:

```
x = 20

y = x - 10

print (y)
```

10

We declared two variables, x and y. Inside x we stored the value of 20. Now whenever we use x in an operation, the program knows that it's equal to 20. Next, we told the program that y is equal to the value of x - 10. Since it knows that x is 20, it knows that it needs to store the result of 20 - 10. This is important because it's not the same as declaring that y is equal to 10. Why? Because we might have to change the value stored inside x at some point during development. When we do that and we tell the program that y is equal to 10, we're going to have problems if we forget to manually change y as well. But in this case when we change x, y will change automatically because it's told to use the value of x in the calculation no matter what.

Here's another example:

```
x = 50

y = 3.5

z = x / y

print (z)
```

14.28571428571429

Here we have three variables. Two of them contain some information and the third one uses that information in a division operation. You can tell that x is an integer because it contains a whole number and that y is a float. The result is also float.

String Variables

In programming terms, strings are sequences of characters. In other words, they're text but they can also contain numbers as symbols in the form of written

text, not operators and variables. You already used strings when you created the Hello World program.

A string is declared using quotation marks. They can either be single quotes or double quotes. It doesn't matter to Python. Whatever you write between them is a string even if there are numbers. These symbols tell the computer that it's dealing with a string and not with numerical data or various types of commands.

Like with any other type of data, we can use variables to store strings inside them and then use the variables for various operations. Let's declare a new variable and insert a string inside it:

```
myDog = 'Rex'

print (myDog)
```

Rex

This example is no different than our first example using numbers. We created a 'myDog' variable where we stored a string. When we print the variable, the string it contains is printed. Just make sure you declare the string in between quotes, otherwise you'll get an error. Now let's perform an operation using strings. When creating programs and games, you'll sometimes want to combine various string variables, like this:

```
human_player = "Timmy"

greeting = "Welcome, "

game_welcome = greeting + human_player

print (game_welcome)
```

Just like before, we declared a set of variables and stored some string information in them. The third variable contains the information combined from the two other strings. In this case the plus sign adds the string from one variable next to the string of another. take note of the empty space we left in the greeting string right after the comma. Remember that empty spaces can be part of strings, too. Python doesn't know how to keep words and strings separate from each other if we don't leave a space.

Finally, let's add a separate string to the two string variables. We can merge all of this information without storing everything inside its own variable. Here's how we can modify the "game_welcome" variable:

game_welcome = greeting + human_player + “!!!”

Since we’re probably not going to need those exclamation marks to be in their own variable, we can add them as a simple string that is processed only inside this variable and nowhere else. The process is the same.

Lists

In a lot of cases, string variables just aren’t enough. We have a lot of information and many objects to define and reuse throughout the program. This is when it’s time to use lists.

In programming, lists, also known as arrays in other languages, are used to store data objects. These objects can be ordered however we want, kind of like a shopping list, and then we have various methods of accessing all of them, some of them, or only specific ones, depending on what we need. In Python, the most common way to access a list item is by using its index value. When you create a list, a position number is automatically assigned to every item in the list. It doesn’t matter whether we store strings, numbers, variables, or other lists inside our list. They’re all assigned an index value starting from 0. Remember that in programming we start counting things from 0 and not from 1. So, the first item inside our list will have an index of 0. With that in mind, let’s get back into IDLE and create a list:

breakfast = [‘eggs’, ‘bacon’, ‘bread’, ‘milk’]

In this example we have a ‘breakfast’ variable in which we stored a list. To declare the list, we need to use square brackets and separate each item with commas. Now we can do various operations with the items inside the list. Here’s how we can print one:

print (breakfast [1])

bacon

To print a list item, we need to declare the list variable inside the round brackets, followed by the index value of the item inside square brackets. In this case we want to print the second item, which is bacon. Remember, counting starts from 0, so 1 is the second item in the list.

Now let’s think about how we could use a list in the real world when creating a game, for example. In most games, the player needs to fight some kind of enemies or monsters. How can we add them to the game? Using your current

knowledge of programming, you would probably do it like this:

beast1 = ‘Wild Boar’

beast2 = ‘Brown Bear’

beast3 = ‘Savage Wolf’

humanoid1 = ‘Orc’

humanoid2 = ‘Goblin’

humanoid3 = ‘Dwarf’

First of all, this method isn’t a good idea because it’s going to take a lot of copy pasted code for each type of beast or humanoid and for certain sounds and actions that all of them will have in common. We talked about repetitive code and how it’s not a good thing. We want to keep things short and neat. Secondly, classifying enemy variables this way when we have tens or hundreds of monsters will become confusing in no time. To save ourselves the many headaches, we’re going to use lists instead. Here’s how:

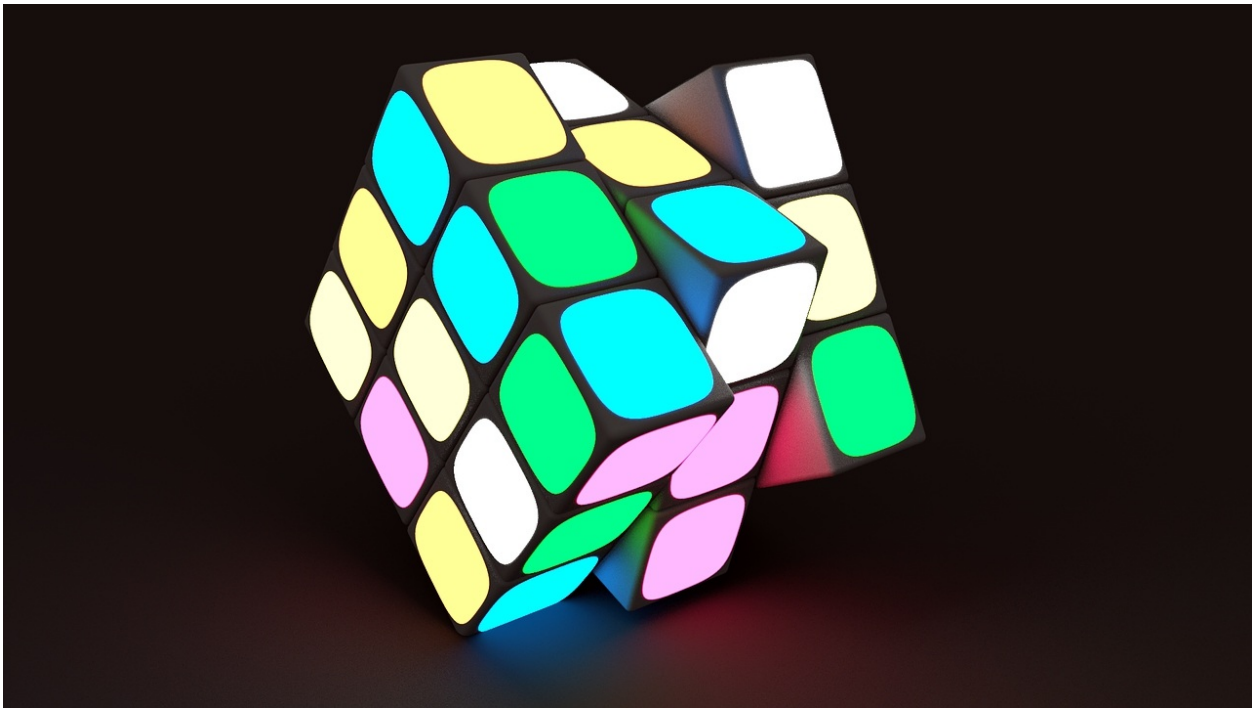
enemy_beasts = [‘Wild Boar’, ‘Brown Bear’, ‘Savage Wolf’]

enemy_humanoids = [‘Orc’, ‘Goblin’, ‘Dwarf’]

This is a huge improvement. We have fewer lines of code and everything is easier to read. Having hundreds of variables like “beast1” would end horribly for any programmer, but lists like this are easier to organize and use in our operations. Now we can simply use the index value for each list item and do whatever we want with them.

This is why coding is so much fun! Notice that both of these methods actually work, even if the first one isn’t recommended. It still does the job. You might encounter some problems later if the project grows large enough, but it’s still a valid solution. Even if you choose to use string variables instead of lists, it’s something you can change later. Sometimes you just want to see if your idea works or you want to get things done without wasting too much time planning. Fortunately, you can always return at a later date and optimize your program. Sometimes it’s easier to improve something that kind of works than to come up with the perfect solution from the start.

Programming Logic



When we discussed all those programming languages, we established that a proper language has to have the ability to program logic and make the game or application somewhat intelligent. Programs need to be able to make choices depending on various situations. They also need to figure out on their own how many times they should process a certain instruction. Just think about all the intelligent devices that are used throughout our homes. Even the humble thermostat is somewhat intelligent. It has sensors that measure the temperature and a smart program that tells it to turn on the heating when the temperature is under a certain value. This means that the program knows to heat the room if the temperature is below the user-set threshold and to stop heating when that threshold is reached.

In other words, programs can make decisions based on a set of conditions. They check whether a condition is true or false, and depending on the result it chooses an action. Programs can also compare values and based on the result, they execute a set of instructions. It’s like telling the computer what it can do when it reaches a crossroads. It does something when going left, and it does something else when going right.

A program without decision-making capabilities isn’t much of a program, so let’s learn how to add some logic to our projects with the help of conditional statements and loops.

Booleans

One of the most common types of expressions used in all programming languages, Booleans enable the program to see if something is true. Based on the result, we can tell the program what to do if the result is true and what to do if it's false. Here are a few examples:

```
print (5 > 6)
```

False

```
print (20 == 20)
```

True

```
x = 5

print (x < 10)
```

True

All of our examples are simple comparisons, and the program tells us whether the statements are true or false. In the first one, we say that 5 is greater than 6, which of course isn't true, so the program tells us that it's False.

In the second example, we say that the value of 20 is equal to 20. That's True, but pay attention to the equal sign. We're using a double equal in this case because we're comparing two values. When we use just one equal, it means that we're assigning a value to a variable or some other element. Remember the difference.

Finally, we also store an integer in a variable and then we compare the variable to the value of 10.

In each case we use special comparison operators to compare a set of values. Here's a full list of these operators and what they represent:

1. Greater than (>): This symbol is used to declare that a value is larger than another.
2. Lesser than (<): This operator means that a value is smaller than another.
3. Greater than or equal to (>=): This is a combination of conditions, and only one of them needs to be true for the result to be true. If a value is greater than another but it's not equal to it, the expression

will still be true because of that “or” condition. So 10 >= 2 is true, even if 10 isn’t equal to 2 because it’s greater.

4. Lesser than or equal to (<=): This works exactly like the greater than or equal to operator except that the first condition checks whether a value is smaller than the other.
5. Equal to (==): Remember not to confuse this sign with the assignment operator (=).
6. Not equal to (!=): This final operator is needed to determine if a value isn’t equal to the other. For example, 5 != 10 is true because 5 isn’t equal to 10.

In programming, we can also run several comparison checks at the same time instead of doing them one by one. In Python this is done using two optional operators, namely “or” and “and”. When the “or” operator is used, the result will be true when at least one condition is met. Even if we have nine comparisons that are false and one that is true, the final result is “true”. When using the “and” operator however, both or all checks have to be true for the condition to be true as a whole. Here’s a simple example:

```
x = 20

y = 10

(x > 10) and (y < 20)

True
```

Both expressions are true because x (20) is indeed greater than 10 and y (10) is less than 2o. Here’s an example using the “or” operator:

```
(x < 10) or (y == 10)

True
```

Even though x isn’t smaller than 10, y is equal to 10, therefore the condition is met.

Boolean expressions aren’t that useful when used like in these basic examples. To really take advantage of them in real world programming, we have to combine them with conditional statements. For complex tasks, we need to allow the computer to decide which road to take depending on the conditions we set.

Conditional Statements

Anything more complex than a Hello World program is going to use some kind of conditional statement. You can't have choices and options without them. For example, if you're making a game, you need to create different scenarios for various events. For instance, if the player is hit with a rusty sword, he will take 10 damage points and a "hit" animation and sound will play. But if he's hit by an arrow, he'll take 20 damage points and a different animation and sound will play. To have these two options, the program needs to check for one of the conditions. So if the player is hit by an arrow, that part of the code will be executed and the other is ignored. If the player is hit by the sword, then the "hit by an arrow" instructions are ignored.

Notice that these explanations repeatedly use the word "if". That's because programs can almost make decisions like we do. We just need to use the "if" statement and show them how. The most simple conditional statement is an "if" statement that tells the program "if this happens then do that". Here's an actual example:

```
is_morning = input ('Is it early in the morning? (y / n)')

if is_morning == 'y':

        print ('Have some breakfast!')
```

The first line of code contains a variable that will require an input from the user. This means that the person using this program will have to answer the question with y for yes or n for no. Next, we have our conditional statement that says if the user presses 'y' then he confirms it's morning, so the program will tell him to have some breakfast. If the user chooses the other option, nothing will happen because the is_morning variable will be equal to "n" and our condition says it has to be equal to "y" for that message to be printed.

What we could do next is tell the program to do something when the statement isn't true. At the moment, nothing happens when the condition turns out false because we didn't issue any instructions. Remember, computers are dumb and they need to be told what to do. So to solve that, we need to add the "else" part of the "if" statement. Let's take a look at an example:

```
is_morning = input ('Is it early in the morning? (y / n)')

if is_morning == 'y':

        print ('Have some breakfast!')

else:
```

```
    print (‘Have an apple’)
```

This translates to “If it’s morning, then have some breakfast, otherwise have an apple”. If the first statement (is morning) is false, then the code inside the else block is triggered and the statement that requires a True output is ignored.

This structure can be extended to give the program as many options as you need. To do that we have another statement called “elif” which is added to the if block. This stands for “else if”, and in our current program it’s quite needed. Our two options so far are having breakfast if it’s morning or having an apple if it’s not. That’s not enough because you would end up having a truckload of apples since most of the day clearly isn’t morning. This means we need more decision paths, and we can add them with the “elif” statement. The structure looks like this:

If this happens then do that, elif (else if) this happens then do this other thing, elif this happens then do a completely different thing, else do this. Let’s consider a new scenario to put this into practice. Let’s say you need a program that calculates the points you got on a test and then assigns you a grade. There are five grades you can get, namely A, B, C, D, and F. So it’s not enough to have only an if and else condition. Here’s how we can give the program the power to choose between these five situations:

```
myTest = eval (input (‘Your test result from 0 to 100: ’)
if myTest >= 90:
    print (“A”)
elif myTest >= 80:
    print (“B”)
elif myTest >= 70:
    print (“C”)
elif myTest >= 60:
    print (“D”)
else:
    print (“F”)
```

Like before, we ask the user for some input. The program asks for a number between 0 and 100 which then will be converted into a grade. To achieve this conversion we use the “eval” function that evaluates the number typed by the

user and then turns it into one of the grades, based on the conditions. The statements themselves are simple and logical. If your score is above 90, you get an A. If that statement isn't true and it's below 90, the next statement is analyzed. If it's above or equal to 80, then it's a B. The program analyzes each condition until the most accurate one is found to be true. Finally, if none of them apply, it means that the score is under 60, in which case it's an F.

Take a moment to fully understand how the program runs. Notice how each statement and instruction is read and executed from top to bottom. It's like the computer is going through a checklist and it sort of does that. That's how code is executed, in an orderly fashion from the beginning to end.

With that being said, you should try to come up with a similar program. This is a lot of information to take in and you need to practice to remember all of it. Imagine a different situation where you'd require a similar if-elif-else structure. Start by writing some ideas on a piece of paper, then work out every section. As a programmer you need to stimulate your creativity as much as you learn languages and tools.

Automation with Loops

One of the most important aspects of programming is the ability to automate tasks. There's an unwritten rule among programmers no matter the language they use. If you have to write a block of code more than twice, you need to find a way to optimize to cut out unnecessary lines of code. That is often done with the help of loops. Instead of manually telling the computer to do something several times, you write a loop and specify how often the command should be executed.

Loops are used to repeat a process or task until it's no longer needed. You're going to use them in Python, C++, C#, Java, and any other language capable of handling logic. So, make sure to learn this part well.

"For" Loops

There are two major kinds of loops: the for loop and the while loop. The first type is used when you know how many times a task needs to be executed. Here's an example:

```
for counter in range (1, 51):
```

```
    print ("Let's go for a ride!")
```

We know we want to print a string fifty times, so we declare a "for" loop and insert a print statement that will be executed every time until the program exits the loop. We use a range to specify how many times we want the loop to run and a counter is used to keep track of all the loops. Notice that our range starts from 1 and it goes to 51. That's because if we declare 50, we would only get 49 loops because the counter value has to be less than the upper range value.

Loops aren't used to just execute a statement multiple times. They can also be used to automatically access or manipulate every object inside a list. For example, if you want to print every item inside a list containing 100 items, you're not going to type 100 print statements, but you're not going to use a range like before either. You'll simply tell the program to process one object at a time, like so:

```
myBreakfast = ['milk', 'bacon', 'eggs', 'toast']

for x in myBreakfast:

    print (x)
```

After we store the list inside a variable, we create a 'for' loop where we say that for every element (x) inside the variable, we print each element (x). The program will loop this block of code until every item in the list has been printed.

As you can see, loops can be useful in more than one way and they keep our code simpler and cleaner. Now, let's talk about the second type of loop.

"While" Loops

Since "for" loops are great when we have a number of times a task has to be executed, what can we do when we don't have that insight? We use the "while" loop, which works exactly like it does in English. Think about it. While you practice coding, you'll learn how to be a better programmer. This means that as long as you keep repeating the task of practicing, you will continue to improve. Or, while you have money, you can afford to buy ice cream. In other words, while the condition is true, the instruction will keep running until the condition is no longer met. It doesn't matter how many times the code has to loop because it's all based on the truth of a condition (or several).

The "while" loop uses Boolean expressions. If you don't remember how they work, you should take a break and go back to refresh your memory. Afterwards,

take a look at the following example:

```
n = 1
while n < 10:
        print (n)
            n += 1
```

Try this exercise yourself and see what happens. The while loop should make sense when you put it in practice. We defined a new variable and stored a value inside it. Then we declared a “while” loop and set a condition. You always need to set a condition, which means it needs to be true for the instructions to be looped. We specified that as long as the value of n is smaller than 10, its value should be printed. In the final line we told the program that after every loop the value of n needs to increase by 1. So with every loop execution the value of n grows by 1, until it reaches 9. The value will still be printed because 9 is smaller than 10, but when the loop restarts one more time and adds 1 to 9, n becomes equal to 10, which is no longer less than 10. Therefore the loop condition is now false and the program will stop executing the loop and continue with other blocks of code that come after it.

Sometimes we need to break out of a loop while the condition is still true. This is done with the help of the ‘break’ statement. We can insert an “if” condition and when it becomes true, we break out of the loop even if the initial condition of the loop is still true. Let’s look at some code because words can be more confusing than Python commands:

```
n = 1
while n < 10:
        print (n)
        if n == 5:
                    break
            n += 1
```

The “if” statement we added says that once “n” is equal to 5, the program will break out of the loop by no longer considering the initial while condition.

You’re going to use loops a lot when coding, so you need to know that there’s a chance of creating an infinite loop by accident. In other words, your code will be executed again and again so many times until the program or computer can no

longer take it and something crashes. Sounds dangerous, but it really isn't. Usually the program crashes first, or you crash it yourself because you can't stop the program otherwise. However, sometimes infinite loops are useful. There are cases when you want something to run in the background for as long as the program is running. Think of a soundtrack looping over and over again while the game is still running. With that in mind, here's an example of an infinite loop:

```
while True:
        print ('Looping forever!')
```

So why is the loop infinite? We set the condition by default to be True without telling the program anything about a 'false' option. This means that the condition is always true no matter what. Therefore, the print command will keep running until you kill the program or it crashes. When you want to avoid this situation, you need to add a false option together with a break statement like we did before. This way the program will stop executing a task when the break condition is encountered. For instance, you can modify this example by telling the computer to break after five print statements. Try to do that on your own. You already know how to write the code.

Nested Loops

In Python and other programming languages, you use the concept of nested loops. In other words, you can put a loop inside another loop, like this:

```
set_color = ['gray', 'black', 'brown']
set_animal = ['dog', 'bear', 'wolf']
for element1 in set_color:
        for element2 in set_animal:
        print (element1, element2)
```

In this program we have declared two lists, one containing a set of colors and the other containing several animals. We want to automatically assign every color to each animal. Since we don't want to manually combine all the possible variations, we're using nested "for" loops. We use a "for" loop that accesses a color object from the list and a second "for" loop inside it that accesses an animal object. The two elements are then matched. These instructions run in a loop until all possible color to animal assignments are made.

Part 2: Coding Real-World Projects

Chapter 4: Using Functions and Error Handling

Programs can quickly become really long and difficult to read. Remember, we want to have only as many lines of code as needed. The more complex a program or game is, the more difficult it is to keep track of everything, especially if you're working with friends on your project. To avoid writing too much code, we can define functions and store commands and instructions inside them.

Functions are kind of like variables in a way because they let you store some information, which you later access by calling the name of the function. It's like creating a shortcut for a block of code and then using it throughout your program. Instead of typing the entire set of instructions, you just type one word and obtain much cleaner code.

So how do you define a function? The same way you start with variables. You give it a name by following the same naming rules and recommendations we talked about earlier. The difference is that functions will also have brackets that can contain parameters. Just think about the print command. It's actually a function that has already been defined and stored inside Python itself. It also has brackets where you insert parameters. In the case of the print function, the parameters are strings or variables that you want to print. They're entirely optional.

The thing with functions is that they're not all the same. There are certain functions that only work with certain data types. This means that a numerical function can't work with a string or a list. These functions are part of the data type and they're called by placing a full stop symbol after the data type or variable. Here's an example of using a special string function:

"hello world".upper()

HELLO WORLD

The "upper" function is attached to the string and it converts the letters from lowercase to uppercase. This predefined function can also take parameters in the brackets, just like any other function. In this example we don't have any parameters to use, so the brackets are left empty.

Besides these predefined functions, we briefly discussed having the ability to create or define our own functions. This is as important as knowing how to

define variables, so let’s dig deeper into functions.

Creating Functions

This is the final bit of programming knowledge you need to start creating real programs and games. Custom functions are a key to programming. In Python, we define a function with the “def” keyword. Here’s how it works:

```
def hello():
        print (‘Hello!’)
```

The function contains a print statement, which means that whenever we use the function, the instruction inside it will be executed. So instead of typing “print (‘Hello!’)” whenever you need that bit of code, you just type the name of the function to call onto its information. To call our function, type the following:

```
hello()
```

The string will now be printed without having to type anything else. These defined functions can be used anywhere in your program once you’ve created them, and you can call them as many times as you need.

To define functions in an easy and clean way, there are a number of recommendations you should follow, just like you do when declaring variables:

1. Functions should have clear, descriptive names that tell you what they do. Don’t worry about writing long names. You should ask yourself if the name is good enough to tell you everything the function does. If it doesn’t, change it. For example, if you write a function that transforms days into hours, you should name it “convert_days_to_hours” and not just “transform” or “conversion”.
2. Functions can be called anywhere in the program. This means it doesn’t really matter where you define them in your code, but it does when it comes to readability. Declare all your functions at the top of the program. Scroll to the top whenever you need to add a new one. Don’t insert new functions in the middle of the code, otherwise when you need to make changes, you’ll have to spend hours going through everything you wrote just to find that pesky function.

Dealing with Bugs

If you followed along and tried to program something on your own, you probably encountered an error or two. That's perfectly alright, because making mistakes is easy and most of them are simple to fix. IDLE and most modern code editors will tell you what's wrong and give you some information about the error and a possible fix. Sometimes these clues aren't quite clear enough and you need to do some research. Look at bugs the same way you do puzzles. You need to find the solution to a problem so you get to the reward. There are people out there that made bug fixing into an art and a career.

The error messages you get from IDLE do half of the work for you. When you know what's wrong and what type of error you're dealing with, you'll know how to apply a solution. Here's an example of an error you might see because it's probably the most common one:

This is an example of what happens when you don't properly spell your keywords or forget to close a bracket or quotation mark. It's called a **syntax error** because there's a mistake somewhere in the sequence of characters. The program stops executing its instructions and IDLE should give you some information about the error together with highlighting the troublemaking code. IDLE and other code editors will also send you straight to the line of code that's causing the problem and offer some suggestions. But if that doesn't happen, you can right click on the error and select the option to "go to line" or something similar.

There are a few other types of errors you can encounter, so let's see what you might be dealing with and what the fix is in each scenario:

- **The indentation error** : Remember that Python doesn't use semicolons or

curly braces to mark where the blocks of code and each statement belong. It's all done through indentation. For example, each line is indented in such a way so that the computer knows when a print command belongs to a particular "if" conditional or outside of it. Indentations are set automatically in IDLE, but you could still delete them by accident. This is what usually leads to the indentation error. Fortunately, it's easy to fix because IDLE will mark the line that's causing the problem. In that case, all you need to do is go to the line, insert or change the indent yourself, and that's it.

- **The type error** : This kind of error has nothing to do with typing. They refer to data types (strings, integers, floats), and they occur when you use a function or some kind of operation that only works with one data type but not the other. For example, you will get a type error when you try to apply a string function to an integer.
- **The logical error** : This type of error is the hardest one to solve. It's not necessarily an error caused by your code. For example, even if your code is technically perfect, the program won't execute the commands the way you want it to. This means there's a mistake somewhere in the design of the program. Here's a simple example:

  ```
  character_hp = 100

  print (character_hp)

  character_hp = character_hp - 1
  ```

  There's nothing wrong with the code itself and it's not throwing an error. The problem is, it doesn't do what we would expect. The health of the character is printed before losing any health points. We want to see the player's HP after taking the damage. To fix this, we need to replace the third line with the second line. That would fix the logic error and the program will work as intended.

  This is why going through the design phase is as important as writing code. If you write your ideas down on a piece of paper, you can use that to figure out why the program isn't doing what you want it to do. Otherwise, finding logic errors will take a long time. There are no clues from the code editor because according to all the programming rules, everything works fine. So take your time and be patient.

Programming is all about solving problems, and the errors are just the icing on the cake. No matter how good you become, there will always be errors to deal

with. So don’t think you’re bad at coding just because you have syntax errors or type errors. Why do you think games and software receive so many updates over time? The developers always find something to fix or improve. It’s all part of the programming cycle.

With that in mind, here’s a shortcut list of things you should check when you’re stuck:

1. Did you spell all your functions and variables correctly? There’s no way for the computer to understand what you mean by “ptint” even though the “t” is next to “r” and it’s an obvious accidental typo.
2. Did you forget a bracket or quotation mark?
3. Are all the indents correctly placed? Even though IDLE places them automatically for you, it’s possible to delete them by accident.
4. Did you mistake a lowercase letter for an uppercase one, or a dash for an underscore? Python is case sensitive and the variable “dog” isn’t the same as “Dog”.
5. Does everything work as you planned in your design document? Check if each block of code performs as it should and there are no logic errors.

Asking yourself a few questions will prevent most errors from occurring. As your program grows, you should go through this checklist more often and test your code occasionally. The more complex the program, the more difficult it will be to search through the code later.

Now that you know all the fundamentals and even some complex programming concepts, we can start creating some cool projects! But remember, all the basic rules apply no matter what you’re working on. You should always design and plan your programs ahead of time, even if all you have is a piece of paper and a pencil. The best way to fix errors is by having as few of them as possible from the very start. Sometimes it’s easier to prevent something than spend hours fixing it.

Chapter 5: Project 1 - Guessing Game

It’s time to put everything you learned to the test by creating a simple game. We are going to develop a game where the computer selects a random number and the player has to guess it. If the player doesn’t guess correctly, they will be told by the computer whether their guess was higher or lower than the randomly generated number. This guessing cycle continues until the perfect match is found.

Now that you know the goal of the game, you should stop for a moment and think what you’re going to need. We’ll have to compare values, so we’re going to use “if” statements. We’ll need input from the player so that they can guess a number, which implies using the input command. We’re also going to need while loops to determine whether the player guessed correctly or not.

But how are you going to generate random numbers? We didn’t talk much about random numbers and the many functions and modules that exist in Python. That would take a whole stack of books because there are so many of them. This is where research comes in! Good programmers need to be good researchers as

well. Python is a popular language and pretty much every question you have about it has already been answered. So before continuing, try putting your Google skills to use.

Did you find the answer? Just in case you're struggling, we need the Random module. As mentioned earlier, Python comes with many modules that extend its functionality. That means that Python doesn't have certain functions and commands activated by default because there are just too many of them. This is why we need to import specific modules for specific tasks. In this case, we need randomly generated numbers, therefore we need the Random module. To import the module just type:

```
import random
```

Now you have access to all the functions that are part of this module. But we need only one, namely the "randint" function. This function stands for random integer, meaning random whole number. Here's how it works:

```
random.randint (1, 10)
```

This line of code will generate a random number between one and ten. The randint function takes two parameters, the minimum number and the maximum number. Between these two values, the program will generate a random number. Try it a few times. See how often you get duplicate values. If you get the same number too many times in a row, then the range is a bit small and you should increase the maximum limit.

Now let's take a look at some real game code and see how to approach this kind of project.

```
import random

random_generated_numbers = random.randint (1, 100)

player_guess = int (input ("Guess anywhere between 1 and 100: "))

while player_guess != random_generated_numbers:

        if player_guess > random_generated_numbers:

                print (player_guess, "is bigger than the number. Try again!")

      if player_guess < random_generated_numbers:

                print (player_guess, "is smaller than the number. Try again!")

player_guess = int (input ("Try again!"))
```

print (player_guess, “Congratulations!”)

Now, let’s go through the code line by line to understand how we used everything we learned so far in practice.

The first step was to import the random module so that we can create a random number generator using the randint function. We added two parameters to the function, 1 and 100, to establish the minimum and maximum values, or in other words, the range. The number generating function with its parameters is stored inside a variable called random_generated_numbers, so the value we obtain is part of this variable as well. Notice that we don’t use a print function here, and therefore this value will not be seen on the screen. That’s the point of the game. The computer needs to generate a number and then hide it from the player.

The next step is to program the player’s functions. We created a variable called “player_guess” where we store the number which the player types when asked for it. We use this variable inside a while loop with the not equal to (!=) operator. This is done so that if the player manages to guess from the first try, the loop doesn’t continue executing any of the rest of the code.

If the player doesn’t guess correctly, we have two “if” conditionals that check the value to see if it’s higher than the randomly generated number or lower. This way the game can tell the player if they guessed too high or low and give them a clue for their next guess. The loop keeps executing these lines until the player gets the correct answer. When they guess the number, the game congratulates the player.

The game itself is fairly simple and not all that exciting. So use what you learned to think of new ways to play this game and make it more interesting. Create a basic score system, add more if statements to mock the player based on the numbers they guess, and unleash your creativity!

Chapter 6: Project 2 - Rock, Paper, Scissors

This is a popular game that you can easily recreate because you probably already know the rules and how it's supposed to work. While practicing code, you should first try to create things you already know to keep things simple but challenging at the same time. As usual, start by designing the game. Describe the rules in as much detail as you can. Think about the order or sequence in which everything needs to happen. Only afterwards you should start coding every little component until you have a complete program.

With that in mind, how do we play Rock Paper Scissors?

First, it takes two players, which means that in our case it's going to be the user against the computer. Each player will have to choose between one of the three options. Once they've picked, they reveal their choices and the game rules decide the winner. As always, the rock beats scissors, scissors beat paper, and

paper beats rock. So how can we write all of this into code?

The simplest method would be to create a list of three items, namely the three objects we're going to play with. Both players can choose one of them by accessing the list. But how are we going to teach the computer to play? We're going to use that handy random module again. The same way we generated numbers randomly, we can use the random functions to make the computer choose randomly from one of the three list items. The player, on the other hand, will choose on their own. Finally, we're going to use a set of conditional statements to see who wins in each case. Give this a try on your own and then check with the code below:

```
import random

list_of_choices = ['rock', 'paper', 'scissors']

print ('Rock beats scissors. Scissors beat paper. Paper beats rock.')

player = input ("Rock, paper, or scissors? Choose your weapon! (Quit) ?")

while player != 'quit':

    player = player.lower()

computer_player = random.choice (list_of_choices)

print ('You chose' + player + ', and your opponent chose ' + computer_player + '.')

if player == computer_player:

        print ('Draw! Play again!')

elif player == 'rock':

    if computer_player == 'scissors':

        print ('You win!')

    else:

            print ('You lose!')

elif player == 'scissors':

        if computer_player == 'paper':

        print ('You win!')

    else:
```

```
            print (‘You lose!’)
else:
        print (‘Something’s not quite right…’)
print()
player = input (‘Rock, paper, or scissors? Choose your weapon! (Quit) ?’)
```

Notice that all the code we needed for this is basic stuff you learned throughout the book. You already know enough programming to work on simple projects, and with a bit of extra research, you can handle even more complex ones. Most of this game consists of conditional statements. But let’s break it all down to refresh your memory and reinforce what you learned.

Just like in the previous project, we start by importing the random module. Remember that Python doesn’t have every single function activated by default because there are just so many of them. That’s why we import modules as needed. In this case, we use the random module once again so that the computer can randomly choose between three options.

After declaring the list and printing the player instructions on how to play the game, we write the code needed for the human player. The player will be able to choose on their own whether to play rock, paper, or scissors. We have a loop here that checks the choice made by the player. However, we do have a fourth choice, which is quitting the game. Once the choice is made, the computer will randomly select one of the objects to play.

Finally, the program will compare the values (the choice you and the computer made), and the winner is established based on conditions. We also prepared ourselves for the possibility of having a player that types something else other than the three keywords that represent the three options. We coded the game to tell the player when they made an invalid choice. In programming you need to also think from the user’s or player’s perspective. Better yet, you should let your family or friends try out your program and see where they get stuck or where they mess up. As a programmer, you think about your project differently and you already know the solutions.

Chapter 7: Project 3 - Fun with Drawings

In Python, you can also play with graphics and animations without needing any complex tools. All you need is the turtle module. This module gives you access to the so-called turtle graphics. It opens a graphics window where you can draw anything you want by using functions and issuing commands through code. The reason why it's called "turtle" is because of the cursor. The graphics are done by moving the cursor not with your mouse or keyboard, but with written instructions. That cursor is the turtle.

We're going to use this module to create a more complex program where we draw various shapes. The most important part you're going to practice here is your ability to plan and design. Let's start simple by creating a basic shape to understand how turtle graphics work.

```
from turtle import *

reset ()
```

left (90)

forward (120)

right (45)

forward (90)

right (90)

forward (90)

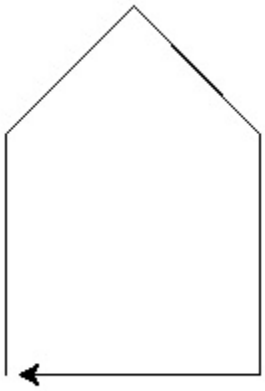
right (45)

forward (120)

right (90)

forward (120)

Here’s the result:



The arrow you see in the image is the turtle pointing in the last direction we sent it. Now, let’s go through the code.

First we imported every function and command part of turtle graphics. Then we issued a reset command so that the position of the cursor is set to default and the pen is placed down. Just like when drawing on a piece of paper, you need to put your “pen” or turtle down so that it can draw something. Afterwards, we have just two types of commands. One of them controls the angle and is measured in degrees. For example, in the first turtle command line we tell the program to turn

the turtle 90 degrees to the left. This moves the cursor straight to the left. Then we tell the cursor to go forward for a distance of 120 points. Next, we change the angle of the cursor again by 45 degrees to the right. This makes it go up. Then once again we tell it to go for a certain distance.

This process is repeated until we have the shape we want. Just keep practicing with the turtle graphics window in front of you to fully understand how the turtle moves.

Now let’s create a more complex drawing program.

We’re going to start by planning everything on paper so we don’t get lost when we’re in the middle of writing code.

We will have three elements: a cursor controller, a string manager, and a user input system. The controller is going to be a function that takes instructions from the user and then converts them into commands understood by the turtle. The string manager is going to be used to divide strings into smaller sections that are then sent to the controller. Finally, the input system will consist of an interface that allows the user to type a series of commands that will be processed by the string manager.

For the controller, we’re going to program it to accept a character that does something and a value that represents a measurement. These elements are then converted into drawing instructions like we had earlier. Basically, we’re creating shortcuts for the user so they don’t have to write any code. For instance, F(50) will be translated as forward (50). When the letter isn’t recognizable, the program will throw an error. We will have several letters to represent each command:

- R - Reset
- U - Pen up
- D - Pen down
- F - Forward
- B - Backward
- R - Turning right
- L - Turning left

Now let’s go through some code and see how the controller works in practice:

```
from turtle import *

def cursor_control (do, val):
```

```
do = do.upper()
        if do == 'F':
                    forward (val)
                elif do == 'B':
                      backward (val)
                elif do == 'L':
                            left (val)
                elif do == 'R':
                            right (val)
                elif do == 'D':
                            pendown()
                elif do == 'U':
                            penup()
                elif do == 'R':
                            reset()
                else:
                             print ('Your command is not recognized. Try
    again!')
```

The code is quite self-explanatory, isn't it? We start by importing all the turtle commands and then we define the controller function that will take a command and an input parameter. Afterwards we convert all the command letters into uppercase letters, and we start defining each command. At the end, we have an else statement that prints a warning message when a letter is used other than the ones we programmed.

Inside the program, you can give the cursor commands like this:

cursor_control ('F', 120)

This will make the turtle go forward by a value of 120, but we aren't done yet. Let's work on the string manager.

Our values need to be converted into simple commands. This is a bit tricky, so we're going to discuss as we go. Let's write some code first:

manager = ‘N-L90-F120-R45-F90-R90-F90-R45-F120-R90-F120’

command_list = manager.split (‘-’)

command_list

[‘N’, ‘L90’, ‘F120’, ‘R45’, ‘F90’, ‘R90’, ‘F90’, ‘R45’, ‘F120’, ‘R90’, ‘F120’]

What we’re doing is using the split function that takes a dash symbol as the separation marker in order to separate the string into a collection of strings, or a list. You can see the final result is a list of separate instructions.

Now let’s complete the programming for the string manager:

```
def string_manager (manager)
        command_list = manager.split (‘-’)
        for command in command_list:
        command_length = len (command)
        if command_length = 0:
        continue
        command_type = command [0]
        if command_length > 1:
        number_string = command [1:]
        number = int (number_string)
        print( command, ‘:’, command_type, number)
        cursor_controller (command_type, number)
```

We define the string manager by first telling the program to divide the string whenever the dash is encountered. Then we loop through all the strings, which are now individual commands. If the length of the command is equal to 0, that means we don’t have a command, so this part is going to be skipped. Next we define the command type by taking every command character to set it as a type (R, L, U, and the rest). Afterwards the number string will take the rest of the command characters by removing the first one. The command is then printed and the cursor controller takes it to execute it and draw something.

Now let’s figure out the final piece of the puzzle, the user interface. Through the interface, the user will be able to type the commands to tell the program what to

draw. We are going to create a window that gives the user a set of instructions, and lets them time as many commands as they want thanks to a while loop. Let's get to work!

```
program_instructions = ‘’’ Send your commands to the program:

like F120-R45-U-F120-L45-D-F120-R90-B50

R = Reset

U = Pen Up

D = Pen Down

F120 = Forward 120

B50 = Backward 50

R90 = Turning to the right at a 90 degree angle

L45 = Turning left at a 45 degree angle’’’

display = getscreen()

while True:

        t_manager = screen.textinput (‘Fun with Drawings’, program_instructions)

    print(t_manager)

    if t_manager == None or t_manager.upper() == ‘END’:

        break

    string_manager (t_manager)
```

Take note of the three single quotes at the start of the “program_instructions” definition and the other three right before the display variable. Triple quotes mean that everything in between is a string, even if it's in a different line and including any line breaks. This block of code is the description of how the program works and what kind of input the user can type. At the end of the instructions set, we define a display variable that contains a “getscreen” function. This opens a new window where all the instructions will appear and where the user will type the commands. Next, we have a while loop that tells the program what to display in the window. The program can be stopped by the user by either pressing the cancel button or by typing the word “END”.

With this program you can now draw much easier without having to restart

everything when you want to create a new drawing. In addition, because of the pen up and pen down commands, you can also draw complex shapes. For instance, you can continue drawing the house we started by creating windows and other details.

Practice using turtle graphics before moving on to more sophisticated ways of playing with graphics and animations, like using game engines. You can also change the color or shape of the pen line you're using, as well as the background and other graphical features. Explore on your own by reading more about turtle graphics and practice by using them for various school projects. You can draw charts, plans, and anything you can think of and present in front of your class. Even if you don't study programming at school, presenting something created with Python will surely impress your teacher and classroom.

Part 3: Glossary

Algorithm: Computer algorithms are collections of instructions that are followed in a specific order or sequence. They fulfill a function or they're used to perform a special task.

Binary code: The basic language of computers is binary. This refers to reading and writing instructions in a sequence of zeros and ones.

Boolean: The boolean expression can only be either true or false. You can think of booleans as questions and based on their answer we can choose to do something or not.

Branch: The more complex programs that rely on the computer being able to make decisions will branch out. This means that the program has multiple paths available depending on various conditions and results.

Bug: Also known as an error, a bug is a problem that makes your program work (or not work at all) in a way you didn't intend.

Call: This term is used when referring to functions. To call a function means to use it whenever necessary. Once a function is defined, you call it anywhere in the program by simply typing its name.

Data: All information is data, whether it's strings, numbers, symbols, or anything else that can be recorded, processed, or used in any other way.

Debug: The process of finding and fixing errors is known as debugging. This is usually done with the help of a debugger.

Debugger: This is either a program, or a function of a program, that looks for errors in your project and offers you solutions or clues to what's going wrong.

Event: Anything a program can directly react to in a way we program it is known as an event. For example, we can tell the computer to play a certain sound only when the right mouse button is pressed. The act of pressing the button is an event.

Execute: Another way of saying "run." A program, instruction, or process is executed on command or based on a set of conditions.

Float: Shortened version of the floating-point number, a float is any number with a decimal in it.

Function: A block of code that contains a collection of instructions and statements. The function fulfills specific tasks whenever it's called.

Index value: This is a number that's automatically assigned when declaring list objects. The numbering system starts from 0, therefore the first item in the list has an index number of 0 instead of 1. The second item is equal to 1, the third to 2, and so on.

Input: Any kind of information that the user enters in the computer or program is referred to as input. Input can come from a variety of systems since it's all about data received from the user. It can be transmitted from the keyboard, mouse, microphone, and video camera, among other devices.

Integer: All numbers that are whole, including negative numbers, are integers. In other words, if there are no decimal points, chances are you're looking at an integer.

Interface: The user needs a way of interacting with the computer without typing any code or commands to do so. That's what the interface is for. It's a way of connecting the user to the computer directly. A graphical interface will also contain pre-programmed buttons that the user can use when interacting with the computer.

Library: All functions and commands that can be used when programming are stored in some kind of library. Python has a large number of libraries, some of which have to be imported in order to have access to their functions.

Loop: Any block of code that repeats itself automatically is a loop. Loops are used to prevent writing the same code again and again, and instead have the computer run the statements as many times as needed based on a set of conditions or factors.

Machine code: This is the actual language that computers understand. A computer doesn't speak in Python or C# by default. An interpreter is used to translate the code that we type into machine code that the computer can read and process.

Module: A collection of functions or even just a section of code that performs a task or several. Modules can be imported into multiple projects to have access to various functions and variables. Modules can also be created by the programmer and transferred to other projects in order to avoid rewriting the same variables and functions.

Operator: A mathematical symbol that performs an action. For instance, the plus

operator is used for addition functions, the minus operator for subtraction functions, comparison operators for comparing values against each other, and so on.

OS: The computer's operating system is the basic platform on which everything else runs. All programs and games you create will use the operating system to have access to the computer's hardware and various functions.

Output: All information that we produce in the form of a result is an output. The output of an operation or a task is created by the program and the user or programmer can see it or use it in other operations.

Program: A program is a collection of instructions that the computer needs to follow in order to achieve a goal of fulfilling a task.

Programming language: The method of creating instructions that the computer can understand and obey involves the use of various programming languages such as Python, C, C++, C#, Ruby, and many more.

Random: This is one of the most popular modules and functions that's particularly used when creating games. It allows the program to randomly generate values, outcomes, or choices in an unpredictable manner.

Statement: This is the smallest block of code that can be written. A function can be broken down into a collection of statements, and a statement is a command that fulfills a specific role.

String: A collection of characters that can contain letters, symbols, and numbers as long as it's declared as such using single or double quotation marks.

Syntax: The collection of rules and structure that determines how the program needs to be written, coded, and arranged, in order to fulfill its function.

Variable: An empty item or place you create to store any kind of data. This data can also change over time as functions and statements can manipulate the variable and update or change the information inside it.

Conclusion

Many people wrongfully think that to be a good programmer, you need to have a knack for it. But programming is a skill, and as with any other skill it can be learned. It is true that everyone can learn programming, but only through persistence and practice can you become exceptional. Today, programming is a very much sought out discipline and a lot of people choose this skill in order to find a good job in the future. With competition being high, it is important to prove that you are the best choice out there. This means that you will have to dedicate time to constantly learn new things and improve. The fact that you finished this book is a certain sign that you have what it takes. However, don't forget this is just the beginning and that the whole world of new information and possibilities is out there for you to grasp.

By reading this book to the very end, you learned the basics of Python programming, and while this means you have a lot more ahead, you already have a head start. Continue learning and above all practicing your programming skills, by yourself or with the help of your teachers. Consider expanding your knowledge and learning such programing languages as C++, Java, Ruby, or C#. Put the newly acquired skills to use every day and start coding.

Lots of young programmers lack self-confidence and they think they don't know that much just yet. But look what you already learned! This book taught you how to explore all the features of various development environments. You learned about loops, variables, and manipulating graphics using nothing but code. If you need to refresh your memory, don't be afraid to go back through the chapters of this book. Let it serve you as a handbook to which you can always come back and remind yourself of the basics. Even already successful programmers have their favorite books on hand at all times to help them in case they forget something.

Start working on a personal project as soon as you feel confident enough. It is the best way to learn. Try applying functions and combine them with conditionals and lists. If you find a tough challenge, that means you are on the good learning path and you are acquiring new skills. Don't be afraid to brainstorm and seek help from others. Some serious research will never fail to provide a solution to your problem. Once you overcome the part which was challenging, the satisfaction will be your greatest reward. What pushes many programmers forward is the feeling of accomplishment, and you will be

surprised at how rewarding it is.

Remember that everything starts with a good idea. Have one? Put it on paper and let it be your start and the goal you will seek to achieve. Ask questions constantly; you'll be surprised by how many people will be there for you to offer you help and guidance. But you also need to be patient. Programming can't be mastered overnight, and it will take years to gather enough knowledge to be able to finish a project. But that doesn't mean you can't start right away. Have an idea for a new game? You can start developing it today, you have the power! Put your idea on a piece of paper, digital or otherwise, and start coding!

References

Images sourced from Pixabay.

Data Structures and Algorithms with Python . (2015). Cham: Springer International Publishing AG.

Pandey, A. K. (2008). *Programming languages: Principles and paradigms* . Oxford: Alpha Science.

Python 3.8.5 documentation. (n.d.). Retrieved from https://docs.python.org/

Thareja, R. (2018). *Python programming: Using problem solving approach* . New Delhi, India: Oxford University Press.

Turtle — Turtle graphics — Python 3.8.5 documentation. (n.d.). Retrieved from https://docs.python.org/3/library/turtle.html