PYTHON CODING FOR KIDS

LEARN TO PROGRAM YOUR OWN GAMES AND APPS THROUGH SELF-DEVELOPMENT

Harvey Lukes

8+

PYTHON CODING FOR KIDS

LEARN TO PROGRAM YOUR OWN GAMES AND APPS THROUGH SELF-DEVELOPMENT

Harvey Lukes

8+

Python Coding for Kids

Learn to Program Your Own Games and Apps through Self-Development

Harvey Lukes

Table of Contents

Introduction

"*Whether you want to uncover the secrets of the universe, or you just want to pursue a career in the 21st century, basic computer programming is an essential skill to learn.*" - Stephen Hawking

In today's digital world, programming is a basic skill that's as important as reading and writing. It's no longer something used only by engineers and computer scientists.

Back in the old days, computers were massive pieces of equipment that could fill entire rooms. They were difficult to operate and too expensive for regular people to afford. But now, everyone has one. In fact, most people have more than one. Just think about the phone in your pocket, the laptop or desktop lying on your desk, and that new tablet you'd love to have if you don't already. Computers are everywhere and are easier to use than ever before. But still, not that many people learn how to code, and that leaves them at a disadvantage.

Learning how to program isn't that difficult anymore. In fact, modern schools already teach coding as an essential skill to kids who are in the first years of elementary. Did you know that Elon Musk, the founder of SpaceX and Tesla, learned how to code when he was just a kid? He was around 10 years old when he started programming, and by the time he was 12, he sold his first game for $500. Then he grew up and founded a company that will hopefully one day enable us to colonize Mars.

Being a programmer is more than just a job. Having the skill to read and write code will allow you to better understand how computers work. You'll be able to use your creativity to entertain others by creating fun games, or help people by solving their problems with your applications. And before you know it, your mom and dad will call you every few days to come and fix their computer problems.

Coding can be fun and addictive because we all experience pleasure when we manage to solve a problem, and that's what programming is all about. You have a problem, a puzzle to solve, and you need to think of a solution. Once you do, you reap the rewards. So, if you learn how to code today, you'll start your life with opportunities all around you.

But where should you start? After all, if you do a Google search on coding, you're going to be blasted with dozens of programming languages you may not have even heard of. There are actually hundreds of languages out there, some decades old and barely used anymore. Some of them have a specific use, and they focus purely on web-development. Others are the pinnacle of game programming. The decision is a hard one to make, but in the end, the principles behind coding are the same. The only thing that's truly different is the learning curve.

To get you started in the world of programming, this book will teach you Python, one of the most popular and easy to learn languages. Python is being widely taught in schools, universities, and is used by professionals from diverse tech fields.

The best way to learn it is through practice. This book is designed in such a way to guide you through all the steps you need to take to learn the language. Just follow along with the examples and try to do them as well. Then try to create or replicate the games and apps you find in the book. You can give the book a read first to get an idea about how programming works, and then start working through each section. If you don't understand something, don't worry about it, because that's only normal. It just means you need more practice. The more exercises you do, and the more you repeat the steps in this guide, the easier it'll be to learn the difficult concepts. Remember that even professionals spend hours and days trying to figure things out. There's no such thing as the perfect programmer. Accept your mistakes, learn from them, and above all, have fun!

Chapter 1: Starting Your Journey

Not that long ago, programming was an unknown and unpopular activity favored by computer wizards. Things used to be analog, computers weren't used very often, and most households didn't even have one. Fortunately, you were born in the digital age and now everything runs on computers. Chances are, you don't even have an old fashioned clock that keeps ticking every second. Your parents probably don't vacuum or sweep the floors of your home because they have a robot vacuum cleaner doing all the hard

work. Everything runs on computers, whether it’s a humble clock, a watch, a doorbell, or a car driven by artificial intelligence. All of these have to be programmed by someone so that we can all have a better life. This is where the future you comes in to save the day!

You picked up this book because you want to be a programmer, a coder, someone with a vision to share with the rest of us humble mortals. Maybe you love playing games and one day you want to use your creativity to tell a story through a cool game. What if I told you, you could do that after reading this book and doing some of the exercises? You don’t have to wait to go to college to become a programmer. All you need is a desire to learn and to have fun!

But what is programming? What does it really involve? Let’s find out!

What Does It Mean to Program?

Many people get into coding because they love computers and they want to create things. Some want to become game developers, while others want to build cool robots. Programmers can do all that because all they really have to do is write a set of instructions that tell the computer what to do. It’s kind of the same way your mom or dad instructs you to throw out the garbage or do your homework. That’s how a computer works, too. It doesn’t really do much if you don’t order it. Computers are a bit lazy that way. They need a coder to use his or her imagination and write a set of instructions for them to execute.

But how do you boss a computer around? You can try yelling at it, but that won’t do any good. What you need is a programming language, like Python!

To understand programming languages like Python, you first need to learn how a computer communicates. Computers can’t speak our language, so to take your instructions, they need to translate your English into their own language. Don’t worry, this doesn’t mean you have to learn the computer's native tongue, which is machine code written in zeroes and ones. It’s enough for you to learn a programming language like Python, C++, C#, Ruby, and so many more. There are a lot to choose from, but if you learn

just one of them, you can easily pick up another in no time. All languages allow us to talk to the computer and tell it what to do. However, we're going to focus on Python because it's one of the most fun and beginner-friendly languages out there.

Python is a programming language used around the world for nearly everything. That's one of the reasons why it's so popular. You can do anything with it! You can program games, applications, and even robots. This language is taught from elementary school all the way to college. Python is a powerful text-based language, which means that you're going to use English words, symbols, and abbreviations by typing with a keyboard.

So, are you ready to dive in? Or are you worried that maybe coding is too hard for you and you aren't quite ready? Let's get that beginner's fear out of the way!

Can Anyone Become a Programmer?

The short answer is YES! Coding isn't rocket science, even though a lot of people make it sound like you need to be a top notch mathematician. All you need to do is learn a few rules that apply to almost all programming languages, and then learn the most important commands needed to write

instructions. Sure, there are various degrees of difficulty that may require other skills from you, but you don't need any talent to start programming. For example, maybe you're an artistic person and you want to use your creativity to create a game, or a virtual story. Perhaps you're really into science and math and want to create an application that generates all sorts of charts that represent your findings.

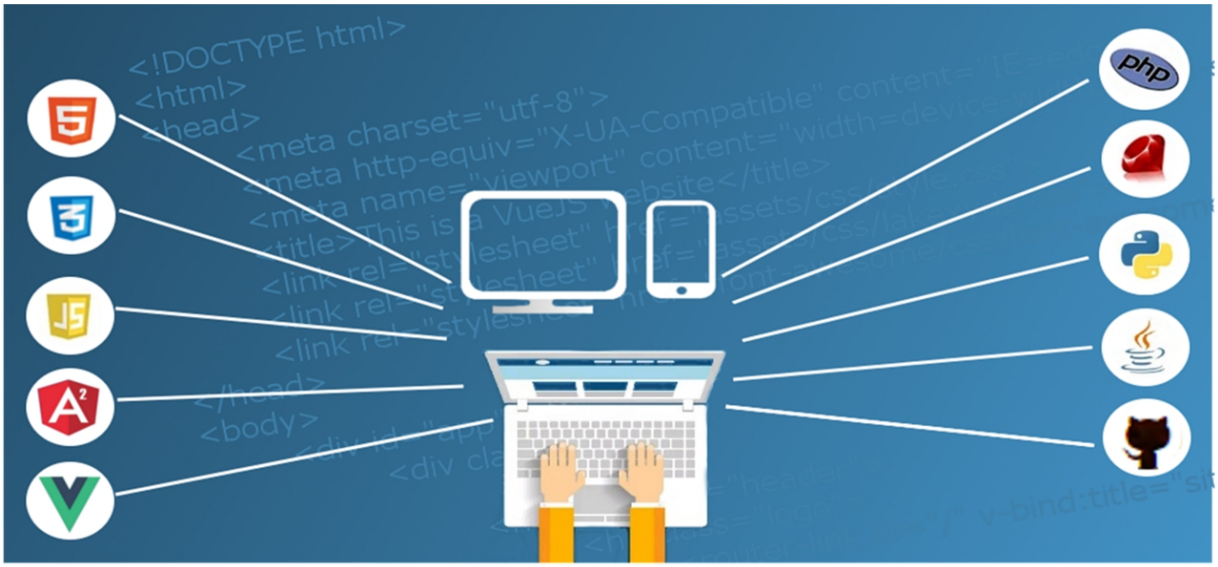
All you really need is the ability to think logically and to be patient enough to focus on the details. Programming is an art as much as it is a science, and you need to think logically, step by step, to write clean code. In other words, you need to be able to create a plan, and then tell the computer in what order it needs to execute certain actions.

But how can you tell if you think logically and you're paying attention to detail? If you like any kind of puzzles, or strategy games like Total War and Starcraft, or empire building games like Civilizations, chances are you have more than enough ability to think things through. A coder with an eye for detail will catch many spelling mistakes and errors, which are referred to as bugs in programming.

Writing clean code and fixing go hand in hand. On a side note, do you know why computer errors are called bugs? Because of an actual bug! In the early days of computers, back in the 1940s, a malfunction occurred inside an electromechanical computer. Nobody could trace the problem, until a year later when engineers found a moth stuck inside one of the computer's relays. They removed the dead moth and recorded the incident as the very first bug. Nowadays, moths don't really affect computers anymore since they can't even fit inside them, but the term "bug" was kept. Every error you encounter in a program is a bug.

With that being said, coding isn't as scary as it seems. You don't need special talents to jump in. All you need is practice and to work on fun projects of your own. If you follow that simple guideline, you'll be able to code anything you want.

Python Alternatives



Just in case you want to open up your horizons from the start and learn what's out there, you can explore the following programming languages along with Python:

1. **Scratch** : Coding usually looks scary to kids and beginners alike because of all the unusual abbreviations and symbols being used. If you or your child are in this situation, there's a lighter alternative to Python called Scratch. This is a visual programming language. Instead of writing all the code yourself, you get to choose from a collection of code blocks that you just drop in the right order. It's like playing with LEGO bricks, but instead of using your creativity with blocks of plastic, you're playing with blocks of code. Scratch is highly interactive and versatile and all you need to do is drag and drop bits and pieces of code. But, this doesn't mean you don't have to learn the rules of programming. You still have to think logically and plan ahead. This is why Scratch is one of the best ways to teach kids how fun programming is and how satisfying it is to solve problems. Scratch can be used to create simple games, animations, applications, and interactive stories. And it's free! Follow this link to learn more about Scratch: https://scratch.mit.edu/ .
2. **JavaScript** : This is an object-oriented programming language that's best used to create web-based applications. Not to be confused with Java, this language is the favorite of web

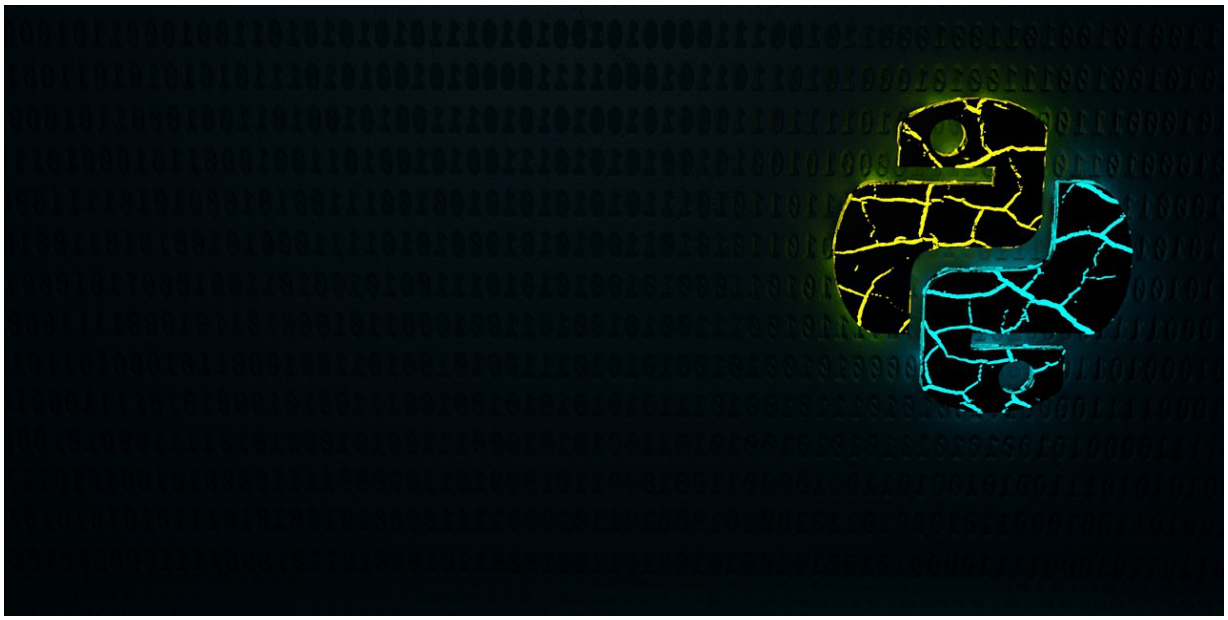
designers, so if you want to create cool online apps you can interact with through a web browser, then this is the right language for you. It's widely used around the world and it's a great place to start learning the basics of programming.

3. **Java** : This language rivals Python in popularity as it's used for a wide variety of purposes. It's somewhat more complex than Python, but it's still a great language to get started with, especially if you love using mobile devices and playing games. Did you know that Minecraft was originally written in Java? Start exploring this amazing language now, or after you learn the fundamentals with Python, and you'll quickly be able to create text-based adventure games and cool mobile programs.
4. **Ruby** : This language is arguably the most easy to read and understand programming language out there. Perhaps even more than Python! If you're worried about reading and writing code, then you should give Ruby a try. Ruby code is self-explanatory. It's so easy to read that you can already read and understand the code of a program written in this language. Kids love Ruby because it allows them to focus on programming concepts without having to worry about the code itself. This way, you can start learning at a faster rate. On top of all that, the language itself is quite popular considering Twitter was written in it.
5. **Lua** : Another language that's as easy to learn as Python, Lua is becoming a popular choice for game developers. If you want to work at Blizzard one day and program game events for World of Warcraft, then you definitely need to check out Lua. It doesn't take long to learn and it will teach you what game programming is all about.
6. **C++** : This is the big daddy of all programming languages. This is the heavyweight champion of software and game development. C++ requires a lot of time and dedication to learn because it's a difficult language to get into. But, once you do, you'll have more than just a basic understanding of coding. You'll know all the complex principles and techniques needed to one day enter the gaming industry or software development industry. C++ is an old language that first appeared in 1985, and to this day it's still being developed and updated for modern use. It is an industry

standard throughout the tech world. You can’t go wrong with this language, but be warned, it has a steep learning curve, so you might want to stick to Python for a while until you master the basics.

No matter which language you learn, what matters is that it teaches you the principles of programming. That’s all you need. The code that you read and write can easily be learned through practice. The logic behind a language is always the same, so master one and be able to expand to others within weeks!

Say ‘Hello’ to Python!



Python started slithering its way in around the 90s, and since then programmers around the world have used it to create millions of applications and games. Even though the language is that old, it’s still more popular than ever, and not just with beginners, but data scientists and analysts as well. Learning Python now means you might still be working with it ten or twenty years later.

On a side note, you need to know that Python wasn’t named after the snake. The man who developed it was a huge fan of Monty Python, a famous comedy group, so he named the programming language in their honor. In

fact, you should probably watch Monty Python's Flying Circus, otherwise you won't notice all the inside jokes made by other Python programmers. Consider it part of your educational material.

With that being said, here are some of the areas where Python is being used daily:

1. **Web development** : Do you love the Internet? I mean, who doesn't!? You'll be happy to learn that YouTube and Google are partially written in Python! The language might not be difficult, but that doesn't mean it's just for kids.
2. **Banking and commerce** : Python is still being used to create financial applications and manage all that money that travels everywhere.
3. **Medical machines** : A lot of medical equipment and robots used to perform surgeries rely on Python for their programming.
4. **NASA** : Do you want to help mankind reach for the stars? Then Python is the right tool for the job. Many tools and programs developed by NASA run thanks to Python.
5. **Entertainment** : Many game developers, artists, and animators use Python to program the logic or to automate tasks that are repetitive.
6. **Robots** : Python is one of the most popular languages used by anyone interested in robotics. You might think this would be insanely complex for a kid to start playing with, but that's not true. In fact, building robots is quite easy, fun, and highly affordable. All you need is a little computer like the Raspberry Pi, a few simple gizmos and LED lights, coupled with a touch of Python, and you can create your own little weather balloon, or temperature monitor.

Python does it all! So, let's jump in and start installing it.

Setting Up Python

Python is a free programming language and you can download it from its official website at https://www.python.org/ . Next, navigate to the

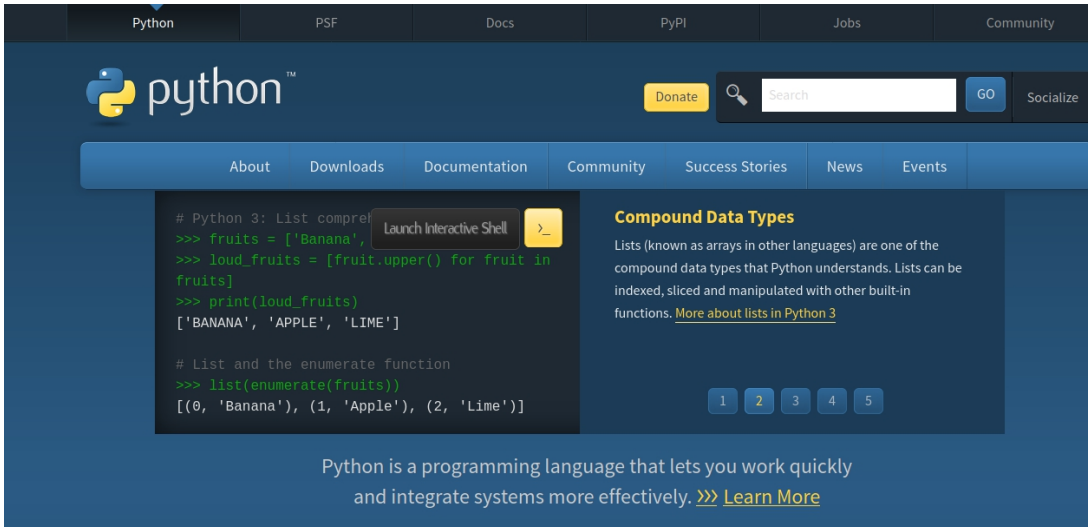
“Downloads” section and choose your computer’s operating system. Python is available for Windows, Mac OS, and other platforms. Select the one you’re using. For the sake of this guide, let’s assume your system runs on Windows. Click on Windows in the Downloads menu, and then select “Latest Python 3 Release - Python 3.8.5”. Python 3 is the latest version of Python. You can also download Python 2, which is older, but you’ll encounter slight differences between the two. In this book, we’ll stick to Python 3 version 3.8.5, which is the latest version at the time of writing.

Now, click on the latest version, 3.8.5 in this case, and then scroll all the way down to the bottom of the newly opened page. There, you’ll see a list of files. Select the right version for your operating system. Keep in mind that you need to choose the right system architecture too, not just the right system. So, in the case of Windows, you need to pick “Windows x86 executable installer” if you have the 32-bit version of Windows, or the “Windows x86-64 executable installer” if your computer runs on the 64-bit version.

Installing Python on a Mac is similar, but instead of choosing between two installers, you only have one option, which is “Download macOS X 64-bit/32-bit installer”. Once you select the correct installer, just double click on it and follow the default steps that are preselected for you.

Congratulations! Python has been installed. Now, you should see a program called IDLE on your desktop, or inside the “All Apps” menu. Execute that program to see if Python is good to go.

But wait! That’s not all. You also have a web-based alternative to the Python program that doesn’t require you to install anything. Just navigate to Python’s homepage at https://www.python.org/ and look for a button with these symbols “>_”. Here’s what it looks like:



If you hover your mouse cursor over that button, it says “Launch Interactive Shell”. In other words, by clicking on the button, you’ll open a program that looks nearly exactly like IDLE, except that it runs on the webpage. You don’t have to download or install anything. Just launch the shell and start coding! But keep in mind that you can’t save your progress, so don’t start working on your game or app through the online terminal. The main advantage is that you can start doing coding exercises or test out your idea wherever you are and no matter what device you’re on.

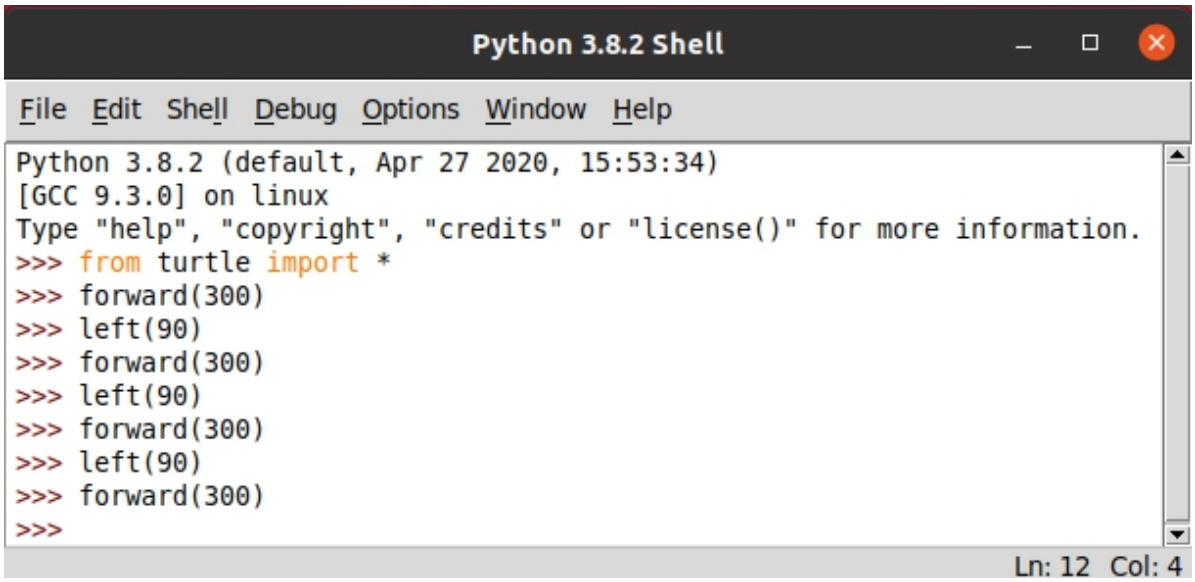
Now, let’s get back to IDLE, because that’s what you’re going to use most of the time.

Using IDLE

The first thing you’ll notice is that IDLE comes with a shell window and an editor window. The shell works exactly like the online interactive shell. It can’t save your code, but it can execute it and show you the results. In the world of programming, a shell is simply a user interface (UI) that allows you access to the services available in your operating system. In fact, the word “shell” is used because this interface is the outer skin of your operating system.

In the shell you can type any Python code and see it run instantly. You’ll also see error messages if any bugs are found. The shell should be used like a notepad. Launch it and try out various ideas to see if they can work at all. You can also use it as a testing area where you try out a block of code on its

own before adding it inside the program or game you're working on. Here's how it looks:



Alright, let's see what we have here:

1. Notice that at the top of the window it says "Python 3.8.2 Shell". This tells us that we're in the shell window and not the editor window.
2. The first line inside the shell tells us the version of Python we're currently using.
3. The second line offers us some information about our computer's operating system. In this example, you can see that the program is running on a Linux computer.
4. After the automatically generated information, we see a block of code. You can see that code is written in front of a ">>>" prompt. In this example, we created a simple drawing program. Its purpose is to draw a line in certain directions and at certain angles. Try it out and see what happens! Then look at the code and figure out what each line did, or go step by step and look at the generated graphics after you execute each single line.

As mentioned, the shell can't save our work. That's not good. We wouldn't want to work for hours and hours and then lose everything when we go to sleep. To avoid losing our progress, we need to jump into the editor

window. This panel allows us to save and load our code and even gives us a few options for dealing with errors.

To run IDLE in the editor window, run the program, navigate to the File menu, and select the New File option. A fresh window will open itself. You'll see that by default it's just an empty screen and without a name. This is your canvas. Now let's start coding! Type the following lines to create your first program:

```
print ('A baby Python came out of a shell and said:')

print ('\n')

print ('Hello World!')
```

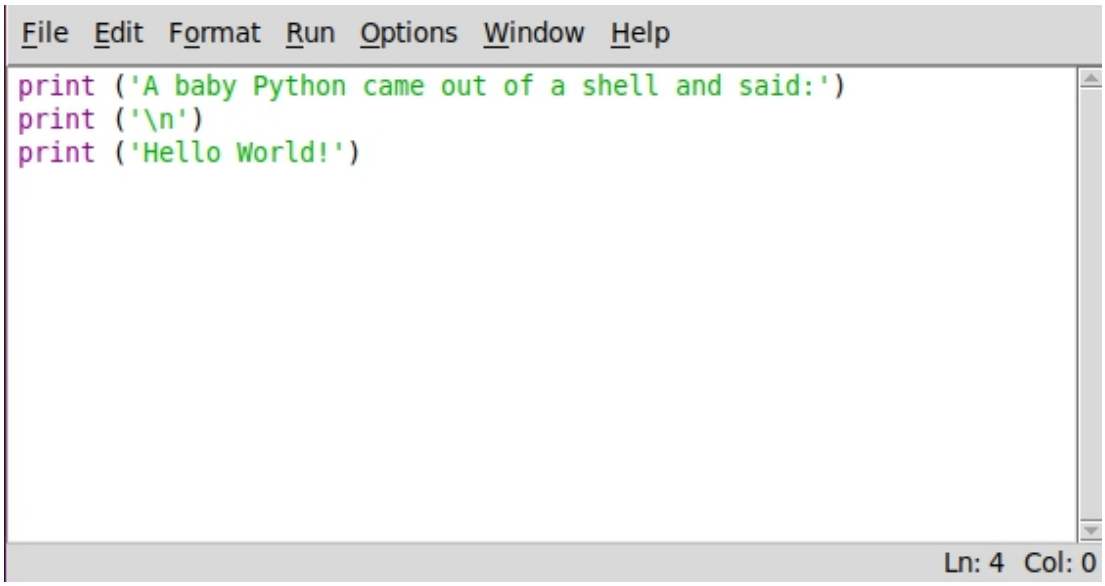
Notice how easy it is to read the code. We have a print command that tells the computer to print something on your computer screen, followed by the messages we want to display. But the way we declare these instructions matters. In this case, the words we want printed need to be written inside single quotes or double quotes. Otherwise the computer will try to read the words as code when they're not. They also need to be declared inside parentheses, or we'll get an error. Finally, we also have a "\n" in the middle. This command translates to "newline", and as the name suggests, it's used to create and leave a new line and continue printing the next line after it. See how logical and easy to remember Python instructions are?

Next, you need to save your file. Go to "File", select "Save As" and choose the location of your program.

Notice that we didn't see the result of the code yet. That's because unlike the shell, the editor doesn't immediately execute your instructions. That's one of the reasons why testing code in the shell is better than doing it in the editor. To run your program, you need to press the F5 key on your keyboard, or navigate to "Run" and select "Run Module". This action will launch a shell window that will show you the result of your code.

A Note on Colors

If you followed along with the code, you may have noticed that some of the words you type come in different colors. Here's an example using our previous program:



IDLE, as well as many other code editors, will color the code in different colors to give us a visual understanding of it.

For example, you can see in the image above that the word “print” is colored purple. This means that the “print” keyword is a Python built-in command. Other special commands like “list” and “int” will also appear purple because they’re part of the programming language. Here, we also have some green code that represents our quoted text that will be printed. The official term for this text is “string”, but we’ll talk more about that later. Here are the other colors you’ll encounter and what they mean:

1. Black: A lot of your code will appear black because the color represents names and symbols.
2. Red: As usual, red means danger, so this color is used to show us the errors in our programming.
3. Blue: This color is used for the output. You probably noticed that the text you told the computer to print in the editor window appears blue in the shell window instead of green. That’s because in the shell window we see the output, or the result of our programming. If you type “2+2” inside the shell, the result, which is 4, will appear blue as well.
4. Orange: Special keywords are colored orange. Did you try out that first program we talked about? The one where we tried to draw something? Notice that the words “from” and “import” are colored orange. That’s because they’re Python keywords. Other

keywords that you'll use frequently, like "if" and "else", will appear orange as well.

As you can see, colors make it a lot easier to read your code. You can immediately figure out what you tried to do. You can see the commands, the keywords, and the output clearly. Often you don't even have to reread your code, you can just glance at it and you'll immediately see if something's wrong.

Chapter 2: Coming Out of Your Shell

Now that you've installed Python and IDLE, you can start building your first application from the ground up.

But before you start writing the code, you need to write down the design of your program. You need to visualize it and figure out what you want it to do. To stick to the programmer's tradition, your first application will be the famous "Hello world" program.

Designing and Coding Your First Program

Here are the steps you need to take to plan your project:

1. **The goal** : Once you have an idea, you need to figure out what the purpose of your program is. Why are you creating it and what should it do? In our example, the goal is to create a program that

welcomes the user with a greeting. Afterwards, it will ask the user what their name is. The user then types his or her name and the program will again say hello, but this time it will also include the name.

2. **The design** : We know what the program is supposed to do, so let's think about each step. Most programmers will create a simple diagram or drawing on a piece of paper. This way they can visualize the steps they need to take when writing the code. Each step is presented with a few details about its purpose before pointing to the next step. The design should be simple and logical. Here's how it would look for our "Hello world" program:

Start Program > Say 'Hi' > Ask the user to type his name > Say 'Hi' including the name > End Program.

That's it! Look how easy it is to follow once you have each step written in front of you. You no longer have to think about the entire program in your head. You can just focus on coding bits and pieces. Working this way will become even more important as your programs grow in complexity. Imagine building a game and all the actions that have to be coded. It's overwhelming just trying to think about it! So, always break down your program in little pieces and work your way up.

What you have so far is known as a software design document. It doesn't have to be anything fancy. Just write the purpose and the structured design of the program on a piece of paper, inside a Word document, Notepad, or create a colorful flowchart in a program like Paint. The bottom line is: plan ahead to save time and effort.

The next step is to start programming! Run IDLE, but remember that the first window it opens is the shell window and we can't save our program inside it. So click on "New File" to open the editor window. Now, let's write some code:

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

Save your file so that you can test this line of code and see if the program runs correctly. If it doesn't and you get an error, don't worry! Even the best of us make mistakes. Fix those bugs by checking if you spelled the word "print" correctly. Remember that this is a built-in Python command, and if

you change one single letter, the computer won't know what you want. Next, check if both parentheses are in place, as well as the quotes. Remember that you can use single quotes or double quotes. They're the same to Python, but it's common for programmers to sometimes forget the closing quote and then they get an error. Fix these mistakes if they exist and run the program again and again until it works.

Next, we're going to follow our design document and start coding the last two steps. Here's how the code will look:

print ('Hello, World!')

user = input ('What's your name?')

print ('Hi,', user)

In the second line, we create a variable called "user" and we store some information inside it, in this case being the question "What's your name?". This "user" is our creation and we can name it whatever we want (we'll talk more about variables in the next section).

The keyword "input" means that the program will ask you to type something before it continues with the last instructions. Once you type your name, the program will greet you using that name. You can change anything you want about the program. You can make it ask funny questions or be rude to the user. You can expand it and instruct it to ask more questions that the user has to answer. With just this bit of code, you can already have some fun and learn!

Using Variables

In our "Hello world" program, we introduced the concept of variables, but what exactly are they? In Python, and in all programming languages in general, variables are used as a method of storing information and labeling it so that we can use it throughout the program. Variables are used everywhere and you'll be playing a lot with them. Imagine variables as books, and each one of them has a title and contains specific information. Whenever you need that information, you pick up the book and use it.

Variables are created and named by the programmer. To create one, you first need to give it a name like we did with “user” in our previous example. The name can be absolutely anything you want, but it should describe the kind of information you’re storing. If you’re storing some information about the user’s name, you should call your variable “user” or “user_name” to make it as descriptive as possible. That way, when you read your code at a later time, you’ll know exactly what that variable does. If you simply named that variable “x”, you’d have to read all the information you stored in it just to have an idea about its purpose.

After naming the variable, we need to assign it a value. Let’s create a new one, like this:

pet = (‘dog’)

The variable is called “pet” and it stores a value, in this case the word “dog”. When you type the following print statement the computer will display the information stored inside the variable.

print (pet)

Now that you have a basic understanding of variables, let’s talk more about how to name them. There are several rules and recommendations you need to follow:

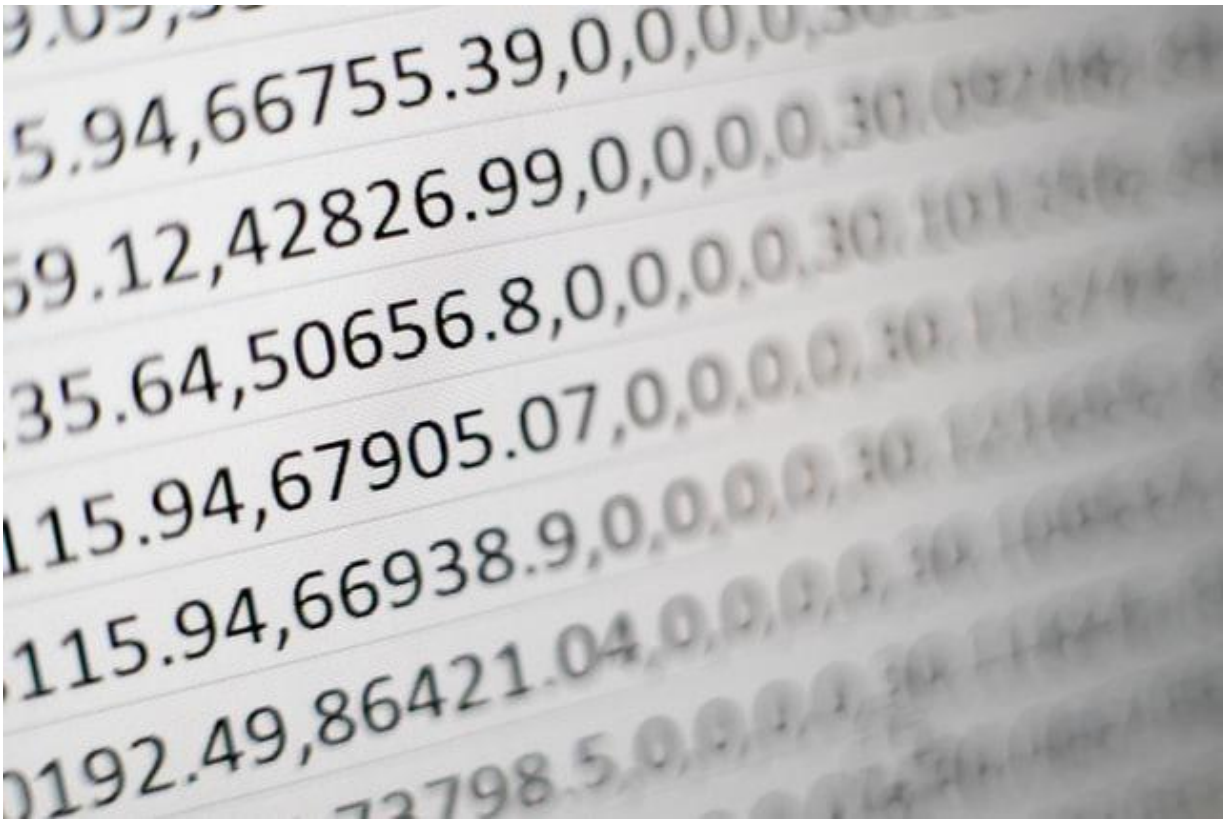
1. The name of a variable should describe the information it stores.
2. Variables can have numbers, letters, and underscores in their names, but they must start with a letter.
3. Spaces are not allowed. If you want to write a longer descriptive name, you should write it using underscores like this: “player_health”, or like this: “playerHealth”. This way the names are easier to read even though we aren’t using a space.
4. Symbols like @ and # are a big no no.
5. You can use lowercase and uppercase letters, and when you do, you should always write the variable as you declared it. The variable “dog” isn’t the same as “Dog” or “DOG”. All three versions are three different variables. So when you get an error, make sure to check that you spelled the variable correctly.
6. Don’t use built-in Python commands and keywords as variable names. This will either cause errors or confusion when you read

your code.

Follow these simple rules and your code will be clean and easy to work with. Keep in mind that once you create a program, you might share it with a coder friend and they need to understand what you meant to do. That's why all programmers follow this set of rules and recommendations no matter where they're from.

Now, let's talk about all the different types of variables. You need to know them by heart because they're at the core of any programming language.

Numbers



When programming, you often have to work with numbers. In coding language, we don't just call them numbers. They are divided in mainly two categories: integers and floats. Integers are the whole numbers like 1, 120, and 42, and in most languages they're represented or declared by using the "int" abbreviation. Floats are the decimal numbers, like 2.33. These numbers are mostly used for accurate measurements where calculations

have to be precise. In most cases you're going to work with integers, especially when starting out.

As mentioned, the purpose of variables is to store information, but the type of data is going to define what they can be used for. In this section we're talking about storing numbers in variables and then using them for a variety of operations, like doing simple mathematical calculations. Just like in math, you can add, subtract, multiply, and more.

Now, open a fresh shell window, or switch to the online version, and let's have some fun with numbers. Try the following code:

```
>>> x = 10

>>> y = x + 5

>>> print (y)

15
```

So what did we instruct the computer to do in this example? First we created a variable called "x" and we assigned a number to it, or an integer to be more precise. Then we declared a second variable called "y" and we told the computer that the value of y is that of x + 5. The result of "x + 5" is then stored in this new variable. Finally, we print the information inside y and the number 15 is displayed as the result. Let's try another exercise:

```
>>> x = 100

>>> y = 15

>>> z = x / y

>>> print (z)

6.666666666666667
```

In this case we create two variables and store some numbers in them. Then we declared a third variable which is the result of dividing x by y. In this case, the result is a float.

Use your creativity and declare as many variables you want. Play with various mathematical operations and see what happens.

Strings

You're already familiar with this data type. Remember your “Hello World” program? We used strings there to make the computer interact with the user. But what are strings exactly?

String is a term that’s used to define any kind of information that’s a sequence of characters. Keep in mind that these characters don’t have to be just letters. They can also be numbers and symbols. But wait, how can numbers be strings? Didn't we just talk about them being their own category of information?

While that’s true, remember that strings are defined in between quotes. The quotation marks are used to tell the computer that it’s dealing with a string and not with numbers or other data types. Everything that’s declared in a quote block is a string, no matter what it contains.

With that being said, strings can be stored inside variables, just like numbers can. The difference is that not all coding operations that work on number variables will work on string variables. The other way around is also valid. Now, let’s store a string inside a variable and simply display it:

>>> player_character = ‘Bob’

>>> print (player_character)

Bob

Here you can see that the variable works exactly the same as the previous ones, except that we stored a string. We created a variable called “player_character” and we stored the string “Bob” inside it. Pay attention to the quotes. If you forget them, you’ll get an error and the program won’t work.

Now, let’s do something that you’ll be using when coding your own games and apps. String variables can be used in combination to create new variables. In other words, we can declare several string variables and then combine their information in another variable. Here’s how this works in code:

>>> player_character = ‘Bob’

>>> welcome = ‘Welcome to the game, ’

>>> start_message = player_character + welcome

>>> print (start_message)

Welcome to the game, Bob

What we did here is simple math, but with strings. We defined the player_character and the welcome strings, and then we combined them to form a new message that’s printed when the program runs. When it comes to strings, we can use the plus sign to merge them together. You should also pay attention to how we left an open space at the end of the “welcome” variable, before placing the end quote. Remember that every type of character counts inside the string, including spaces. If we don’t tell the computer that we want a space at the end of a string, or at the beginning, it won’t know to automatically place it. Computers are both smart and dumb, and we need to tell them everything they need to do.

You can also add a new string that isn’t stored in a variable together with the merged variables. You can see that our result lacks any kind of punctuation mark. To solve that, we can modify the “start_message” variable to look like this:

>>> start_message = player_character + welcome + ‘!’

The exclamation mark will now be added at the end of the message.

Lists

Sometimes, storing numbers and strings just isn’t enough. Sometimes we have to work with a lot of information, or maybe we have a large collection of items that need to be arranged in a specific order. This is where lists come in to save the day.

Lists are collections of data objects that can be ordered however we want. Python automatically assigns a position number, known as an index, to each item. So, we can access or use the list object by using its index value. Just think of a regular shopping list or ingredient list for a recipe. Each item on the list will be separated with the use of a comma, and there are no restrictions when it comes to content. This means that we can store

practically anything, including integers, strings, or even other lists. But before we talk more about lists theory, let's see some action and write some code. Here's a simple list:

>>> shopping_list = ['potatoes', 'onions', 'milk', 'sugar']

That's it! We created a variable called "shopping_list" and we stored a list inside it containing several string objects. Notice how the list items are declared between square brackets, and not round ones. This is important because that's how the computer knows that this is a list. Now, let's access the third item on the list instead of using the whole list:

>>> print (shopping_list [2])

milk

What we did here was print a list item by declaring its position value inside square brackets.

Notice how the output is the third item on the list. But wait, in the code we declared a [2], which should mean that we want the second item, right? Wrong. This is really important and it's valid for all programming languages you'll encounter. In computer programming, numbering doesn't start from 1 like in the real world. Instead, it starts from 0. So when you want the first item in the list, you need to declare 0 as its index, and not 1. In this case, we're interested in the third item in the list, and Python assigns it the position value of 2 because the list order is 0, 1, 2.

Now, let's try something a bit more complex. Let's say you're creating a game and you need to store a collection of weapons that will be available to the player. To achieve that, you might want to do something like this:

>>> sword1 = 'Iron Sword'

>>> sword2 = 'Steel Sword'

>>> sword3 = 'Mithril Sword'

>>> bow1 = 'Short Bow'

>>> bow2 = 'Hunting Bow'

>>> bow3 = 'Longbow'

So far we have a variable for each weapon. But that doesn’t really look right. Since this is a game, we would also need to assign animations to each category of weapons, or sounds, and a lot more. We could save a lot of time by assigning one single animation to each type of weapon. So instead of copy pasting some animation code for every single variable, we should create two lists to represent the two weapon categories. This would make our work a lot easier in the future, so let’s create those lists:

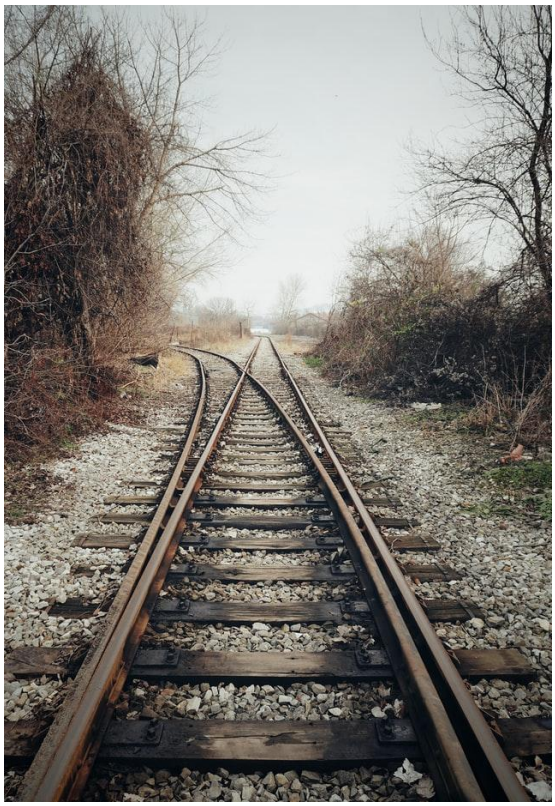
>>> sword_weapons = [‘Iron Sword’, ‘Steel Sword’, ‘Mithril Sword’]

>>> bow_weapons = [‘Short Bow’, ‘Hunting Bow’, ‘Longbow’]

This already looks a lot better, doesn’t it? Now you can pull whatever item you want by declaring the list and the index number of an item.

This is the beauty of coding. There are multiple solutions to the problem, and in our situation you can use both examples. Technically, they both work just fine. But as we continue developing the program, we can improve the code to make our lives easier and write more efficient code. This is another reason why designing your program by creating a basic flowchart is important. It lets you think about future problems so that you can prevent them.

Decision Making Computers



One of the coolest things about programming is that you can create applications that have some freedom to think on their own. Just think about all the smart devices around your home, like the thermostat. It's smart enough to know that when the temperature in your room drops under a certain number it needs to warm it up. Or think about artificial intelligence in games. Ever cast a spell like a fireball, but your enemy knew what you were doing and decided to interrupt your spellcasting? That enemy was programmed by someone to be able to make its own decisions depending on what the player does. But how does this work?

In all of these examples, the program knows that it needs to check whether certain conditions are met. In other words, it's verifying whether something is true or false, or sometimes it's processing a comparison to see if one value is greater than another (room temperature value is below the set value, so the heating turns on). There are a few ways of teaching a program to make decisions, so let's explore them because you'll need to make your programs a lot smarter than they are so far.

Boolean Expressions

Also referred to as Booleans, these expressions are conditional statements that are used all the time in programming whether you're using Python or C++. In simpler terms, a Boolean allows the program to check and see if something is true or false. Therefore, there can only be two results, either True or False. Let's take a look at the most basic example:

```
>>> print (10 > 5)
True
>>> print (10 == 12)
False
```

In the first line, we tell the program that 10 is greater than 5. It analyses this statement and it tells us that it's True. In the second line, we tell the program that 10 is equal to 12, which is of course not true, and it sees that because it gives us a False output.

Now, notice how we used a double equal sign instead of a single one. What's the difference? In programming, one equal (=) means that we are assigning a certain value to a variable or expression. Two equal signs is a comparison operator that compares two values or more.

With that being said, here's a list with all the comparison operators you can use in Boolean expressions and what they mean:

1. >: This symbol translates to "greater than" and checks if a value is bigger than another.
2. <: The opposite of greater than is lesser than and it checks if a value is smaller than the other.
3. ==: This sign means "equal to" and it's used to check if two values are equal to each other. Remember, don't confuse this symbol with the equal sign (=).
4. !=: This weird looking symbol translates to "not equal to". It's used to check if a value isn't equal to the other. So, if we say that 3 != 5, the program tells us it's True.
5. >=: This combination means that something is greater than or equal to a certain value. If one of the conditions is true, then the entire statement is true.

6. <=: This translates to less than or equal to and it works exactly like the previous condition.

In addition, pay attention to how “True” and “False” are written. When you declare that a variable is true or false, you need to use capital letters because the keywords “true” and “false” in small caps are not recognized. Let’s see another example:

```
>>> bananas = 5
>>> dogs = 20
>>> bananas > dogs
False
```

In this case we first stored some values inside two variables and then compared them to each other. We checked to see if there are more bananas than dogs, and the result is false.

Python, as well as other programming languages, allows us to perform more than one comparison at a time. To do that, we use two special operators, “and” and “or”. If we use the “and” operator, the result will only be True if both comparisons are true. If one of them turns out to be false, then the entire expression is False. On the other hand, if we use the “or” operator, only one of the conditions has to be True for the entire comparison to be true. Here’s how it all looks in code:

```
>>> x = 10
>>> y = 5
>>> (x > 10) and (y == 5)
False
```

Even though y is indeed equal to 5, the first statement isn’t true because x isn’t greater than 10, so the whole expression is False. Here’s another example using the “or” operator:

```
>>> (x == 10) or (y < 2)
True
```

The result is True, because only one of the conditions has to be true, and in this case, x is indeed equal to 10.

But how can we use all of these examples to enable the computer to make a decision? In real world programming, computers need to decide on their own what they need to do and what section of their program they need to access. This means that we need to show the computer that it can take several roads to different destinations. In other words, we need to start setting conditions.

Conditional Statements

Imagine you need to decide what to eat for lunch based on which day of the week it is. Is it Monday? Then it's pasta day. Is it Saturday? That's pizza day! So, if it's Monday you're going to eat pasta, and if it's Saturday you're going to have a pizza.

Computer programs need to make similar decisions. Different conditions lead to different outcomes. Based on what's true, the program will activate a certain part of its programming. To do that, your application has to run a test and see what it should do next, exactly the same way you tested yourself to see what you should have for lunch today.

To give those decision making powers to your program, you need to learn the power of the "if" statement. If it's Monday, you'll have pasta for lunch. If the enemy monster hits the player character, the character loses health points. Simple and logical, isn't it? If statements are simple and allow the computer to test if a certain condition is true. Let's see an example:

```
>>> is_monday = input ('Is it Monday? (y / n)')
>>> if is_monday == 'y':
        print ('It's time for some tasty pasta!')
```

In the first line we set an instruction for the user. The program will ask them to answer with a yes or no if it's Monday. The second line is the condition. If the user confirms it's Monday, then the third line of code is triggered and the program will display a message. If the user answers with a "n", then the message will not be displayed because the statement will be False.

Next, we're going to improve this little program by also adding another instruction if the condition turns out to be false. Right now if it's false nothing happens. To change that we need to add an "else" statement after

the “if” block. In programming this is known as the if/else statement. Let’s play with some code:

```
>>> is_dog = input (‘Have you been a good boy? (y/n)’)
>>> if is_dog == ‘y’:
        print (‘Omg, you can talk!’)
>>> else:
        print (‘Woof’)
```

If the condition in the if statement is true, then the first message inside the if block is printed. If it’s false, then the else block will be executed instead.

Finally, you can branch out even more and give the program the ability to choose from multiple paths, not just one or the other. To do that, we have the “elif” statement, which stands for “else if”. Imagine asking your parents to get you a pet and they only have two choices. They can either get you a dog, or nothing. That’s not a very logical outcome. Scenarios like this exist in programming all the time. So, instead of getting a dog or nothing, you can add an ‘elif’ statement to include the option of getting a cat, a parrot, or a ferret.

Now think of another scenario, like an online test that someone actually had to program. The test program can’t have only an if and else block because the student has to be graded with at least A, B, C, D, and F, and not just an A or an F. So the program needs to test whether the user scored enough to get an A, and if not then see if he scored enough for a B, and so on, until all cases have been tested and a result was declared true. Here’s how all of this looks in practice:

```
>>> school_test = eval (input (‘Please type your score from 0 to 100: ’))
>>> if school_test >= 90:
            print (“Excellent, you get an A!”)
>>> elif school_test >= 80:
            print (“Good job, you get a B!”)
>>> elif school_test >= 70:
            print (“Not bad, here’s a C!”)
```

```
>>> elif school_test >= 60:
        print (“Good enough, you get a D!”)
>>> else:
        print (“You get an F. Better luck next time!”)
```

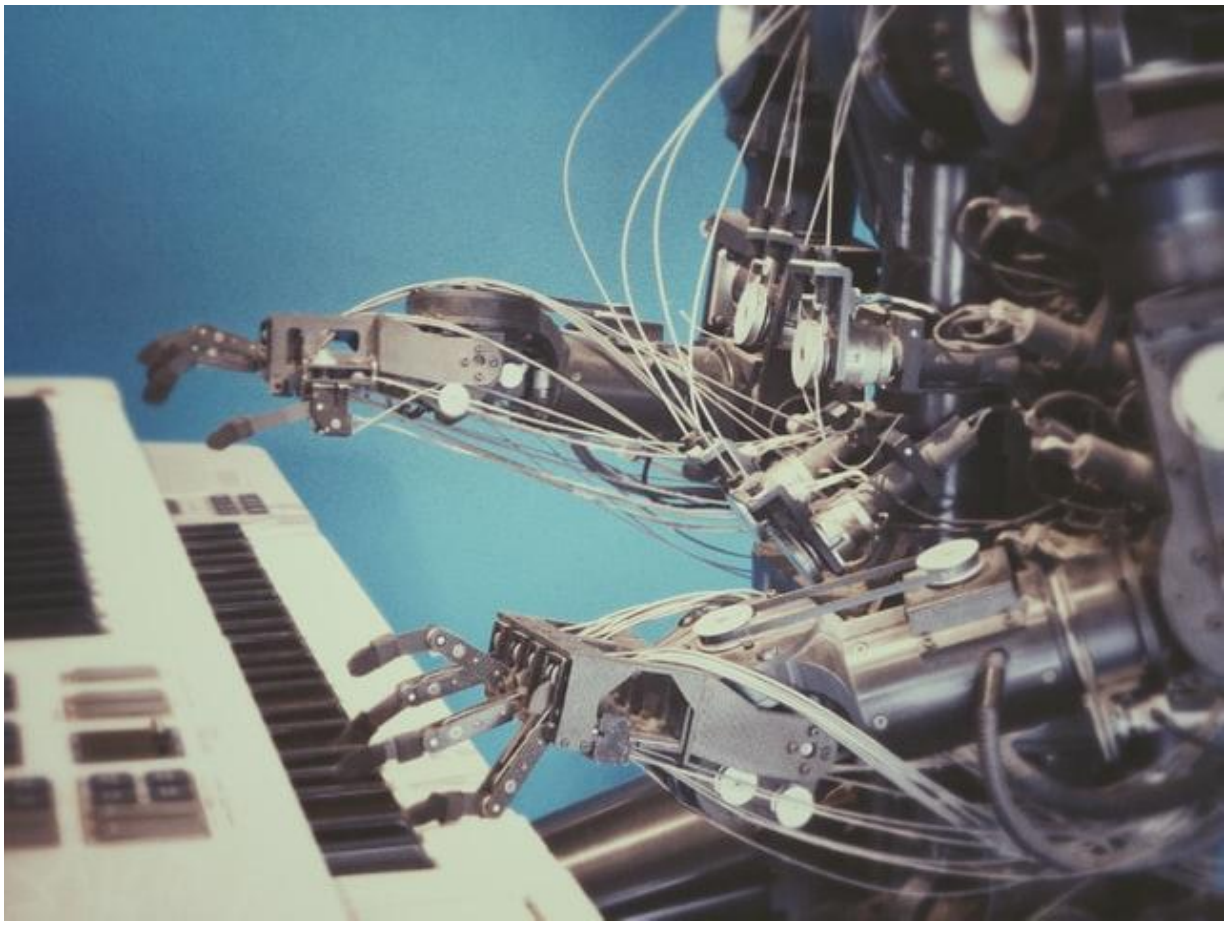
Alright, let’s see what’s going on in this program. But before you continue, try to remake it yourself, or create something similar. At this point you have enough knowledge to create a smart little program.

With that in mind, the first thing we did was instruct the program to ask the user what score he received on the test. The user should type a number from 0 to 100. Then we use “eval”, which is a function (we’ll talk more about them soon). This function is stored in the school_test variable for convenience. The purpose of the function is to evaluate the user input and convert the number to a grade, based on the if/elif statements.

Now we’re all set, and we list all possible situations and conditions. If the user has a score greater than or equal to 90, he gets an A. If it’s greater than or equal to 80, then he gets a B instead, and so on until the final else statement that says that in any other case other than those above, the user gets an F.

Notice how orderly the program processes everything from top to bottom. It all looks so neat and simple, doesn’t it? Go ahead and think of a different scenario where you might have to make such checks. Maybe write a block that checks for the percentage of a player’s health. The if - elif - else code block is used often in the real world because of the number of tests and paths the computer needs to go through. So learn it well and practice it because you’ll use it plenty of times when creating games and apps.

Loops and Automating Boring Tasks



One of the reasons why computers are taking over every aspect of our lives is their ability to automate the boring stuff. Programs don’t care if they have to repeat the same task again and again 24/7. So instead of writing repetitive code and wasting your time instructing the computer to repeat something a few times, you can use a loop.

In essence, loops make the program run in circles and repeat a set of instructions. They are used all the time whether we’re talking about game development, web development, or software engineering. But there’s more than one type of loop, so let’s explore them!

The ‘For’ Loop

Let’s say you want to run a block of code 100 times. If you know for sure how many times a certain statement needs to be executed, then you can use the “for” loop. Here’s an example where we print the “Hello, World!” string 100 times:

```
>>> for counter in range (1, 101):
        print ('Hello, World!')
```

In the first line we declare our "for" loop and instruct it to run 100 times, by counting from a range between 1 and 101. Take note that if we declare a range between 1 and 100, we would get 99 loop executions, because the counter will need to be smaller than the final number. Python considers the first loop execution equal to the first declared number in the range and then it executes all the loops until the last number is reached. So a Python range between 1 and 5 would execute loop number 1, loop number 2, loop number 3, and loop number 4.

Here's another example where we're looping through a list instead of a range of values:

```
>>> shopping_cart = ['milk', 'ham', 'eggs', 'butter']
>>> for x in shopping_cart:
        print (x)
```

First, we declare a list and store it inside a variable, like we did before. Then we write a "for" loop that's meant to print every item inside the list. Try it out, and you'll see that each list object is printed neatly in its own line. This way, you don't have to write multiple print statements, though that would work too. Writing a loop instead is a much cleaner way of coding this kind of program.

The 'While' Loop

We said earlier that 'for' loops are excellent when we know the number of times we need to run a block of code. But what do we do when we don't know? This is where the "while" loops come in.

The 'while' loop is the most versatile technique in programming that can be used to solve a lot of problems. The way it works is simple. Just think of a sentence in English using the word "while" and you'll realize how this loop works. Here's an example: "I can learn a lot while reading this book" or "While I have money in my pocket, I can buy candy". While the condition is true (while you have money), you'll keep running in a loop buying candy.

When you run out of money, you'll exit that loop and stop buying candy because you can't. That's how the 'while' loop works in programming as well.

A 'while' loop doesn't work like a 'for' loop, in the sense that it doesn't need a range condition. Instead, it uses a loop condition. In other words, it relies on a Boolean that can either be true or false. So, while you have money (if true), then you'll buy candy. If that condition is false, the rest of the code inside the loop block is ignored and your program continues with the next block. Here's a simple example.

```
>>> x = 1
>>> while x < 5:
        print (x)
    x += 1
```

What we have here is a variable called 'x' and it is equal to a value of 1. In the second line we write the "while" loop condition that says that while x is smaller than 5, the program has to print x. The last line instructs the program to print x's value + 1 on every loop. If we don't specify that we'll end up with an infinite loop that will never stop (more on that later). So, on the first loop, the code sees 1 is smaller than 5, adds 1, and in the next loop x will be equal to 2, which is still smaller than 5, so we'll have a third loop, and so on. When x reaches a value of 4, the loop will stop running because on the next round x will be equal to 5, which is no longer smaller than 5.

Now, let's see another example where we tell the program to exit the loop when x reaches a value of 3:

```
>>> x = 1
>>> while x < 5:
        print (x)
        if x == 3:
            break
    x += 1
```

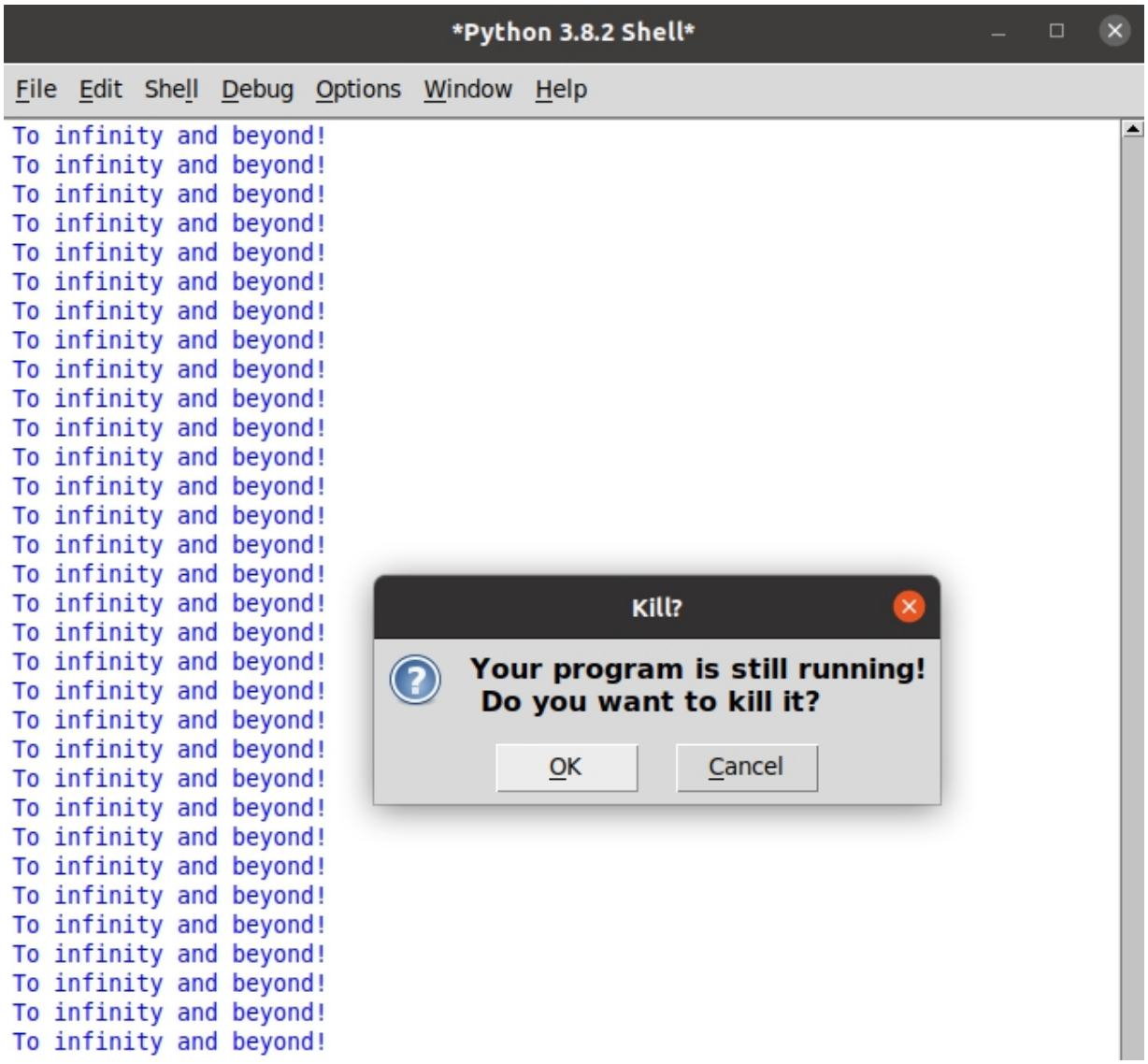
The only difference between this example and the previous one is the “if” condition we added. It tells the program to first check if x is equal to 3 before adding +1 to its value. When that condition turns true and x = 3, then the loop will stop running because of the “break” statement.

Now that you understand how ‘while’ loops work, let’s talk about those pesky infinite loops.

Sometimes we want infinite loops because of some code that needs to run again and again for as long as the program is active. But in most cases, we want to avoid them. So how are infinite loops born? Here’s an example:

```
>>> while True:
        print (‘To infinity and beyond!’)
```

And here’s the result if you run this code in the shell:



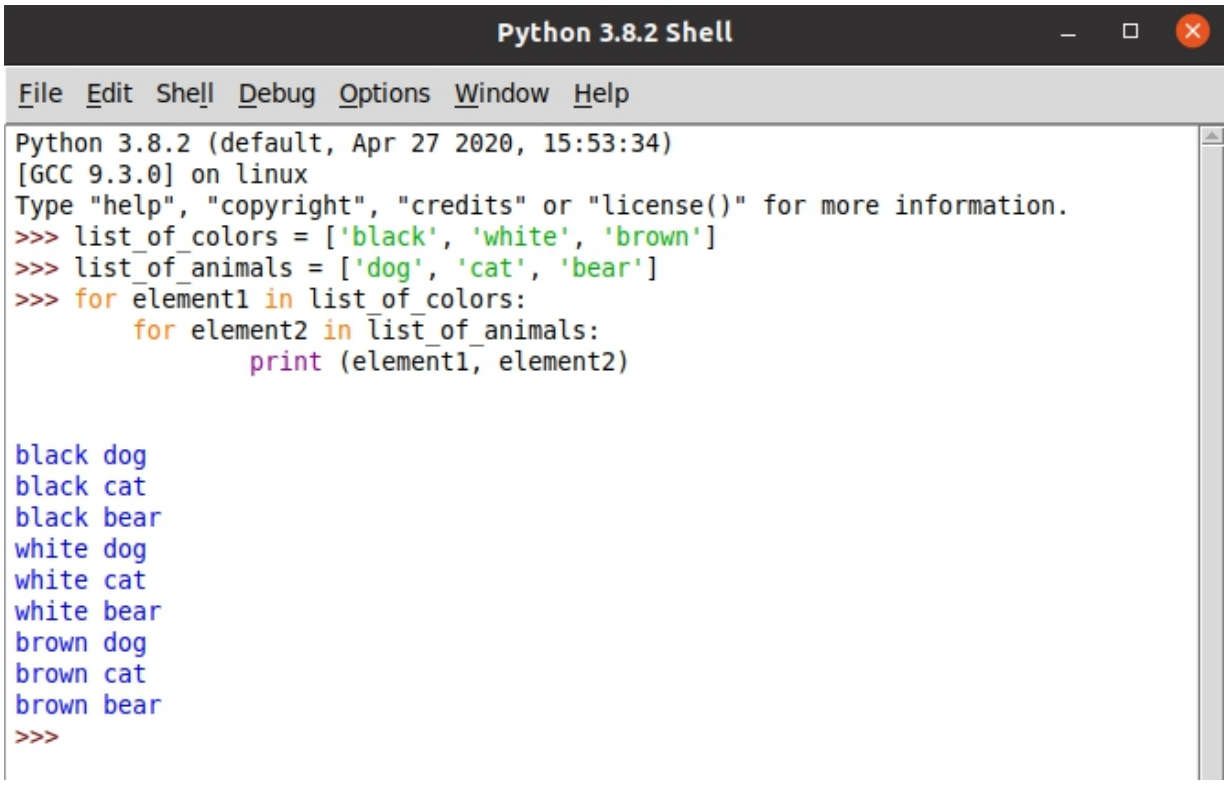
Our code doesn't contain a false option and the condition is always True. So, the program will run that print statement for all eternity, or until it crashes, and our only option is to kill the program. To avoid these situations, we need to write a false condition that can trigger a 'break' statement. So even if we can't know how many times the loop needs to run, we can tell the program to break out of the loop when a certain condition is encountered. For instance, you can tell the program to print this message, but when it reaches 10 prints, the program needs to escape the loop.

Nested Loops

Also known as loops within loops, Python allows us to write loops inside other loops until we're completely lost, confused, and have no idea what's going on. We can program a whole chain of loops. Here's how they work:

```
>>> list_of_colors = ['black', 'white', 'brown']
>>> list_of_animals = ['dog', 'cat', 'bear']
>>> for element1 in list_of_colors:
        for element2 in list_of_animals:
        print (element1, element2)
```

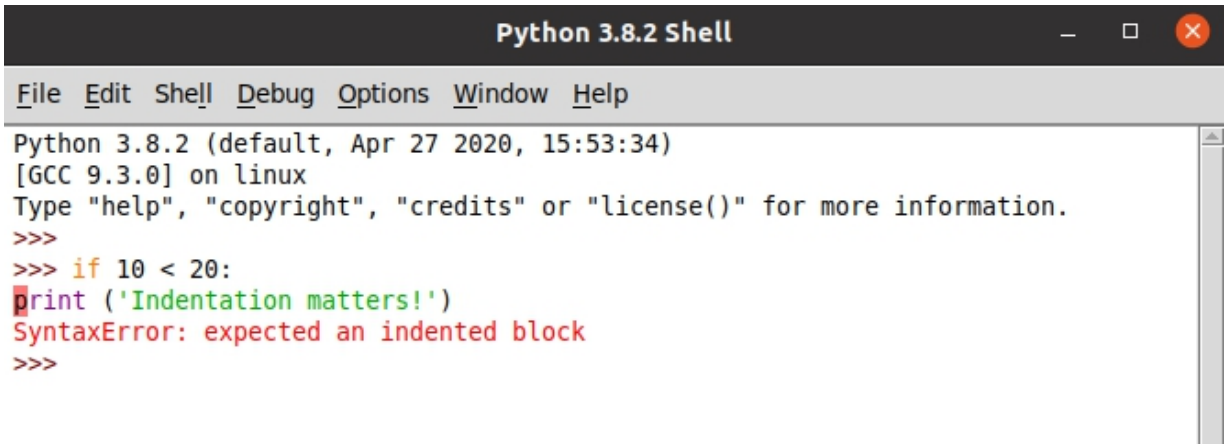
And here's the result:



As you can see, we used two 'for' loops, where one takes a word from the list of colors and afterwards the second loop takes a word from the list of animals. At the end, both words are combined. So, each color is combined with an animal before moving to the next color. Whenever the outer 'for' loop is executed, its inner loop is also executed, before starting over.

Important Note on Python Code Indentation

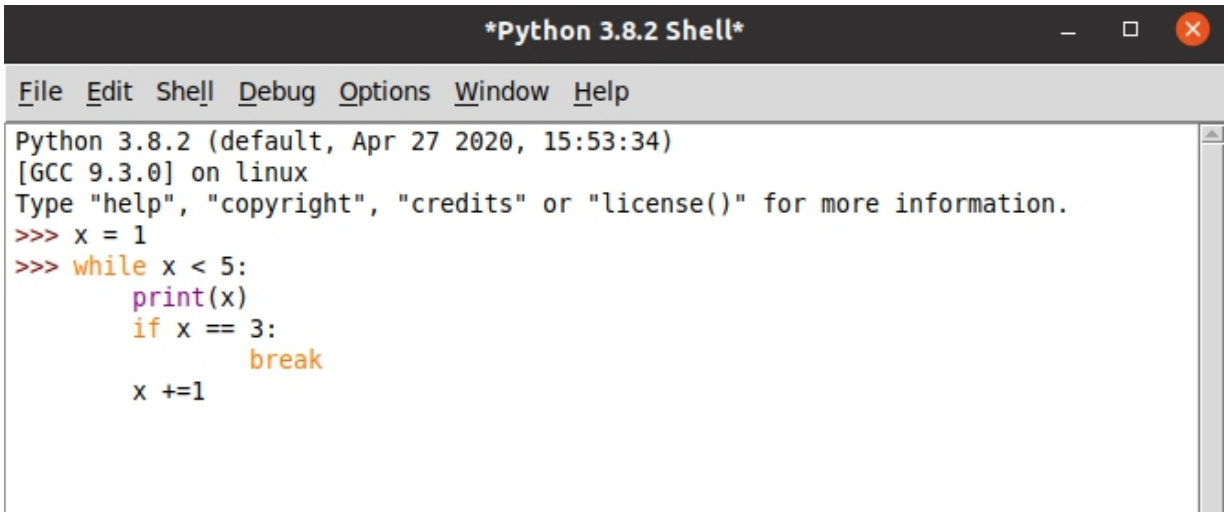
You may have noticed that throughout the chapter we've arranged code in a certain way. Also, if you've been following along with the exercises using IDLE, you probably noticed that those alignments are done automatically by the code editor. These spaces, known as indentation, are very important in Python and if we don't use them properly, this is what happens:



An error appears and it tells us that the 'print' line needs to be indented. That's because Python relies on indentation to define the blocks of code and to know where they belong. Without it, the program can't know if the 'print' statement is part of the 'if' block or outside of it. Other programming languages don't use indentation and instead they use a punctuation mark like a semicolon or curly braces to specify where the lines of code belong.

IDLE automatically sets the indentation for you, but other code editors might not. Besides, you can always delete the indent by accident, like in this example, and you get a syntax error that breaks the program.

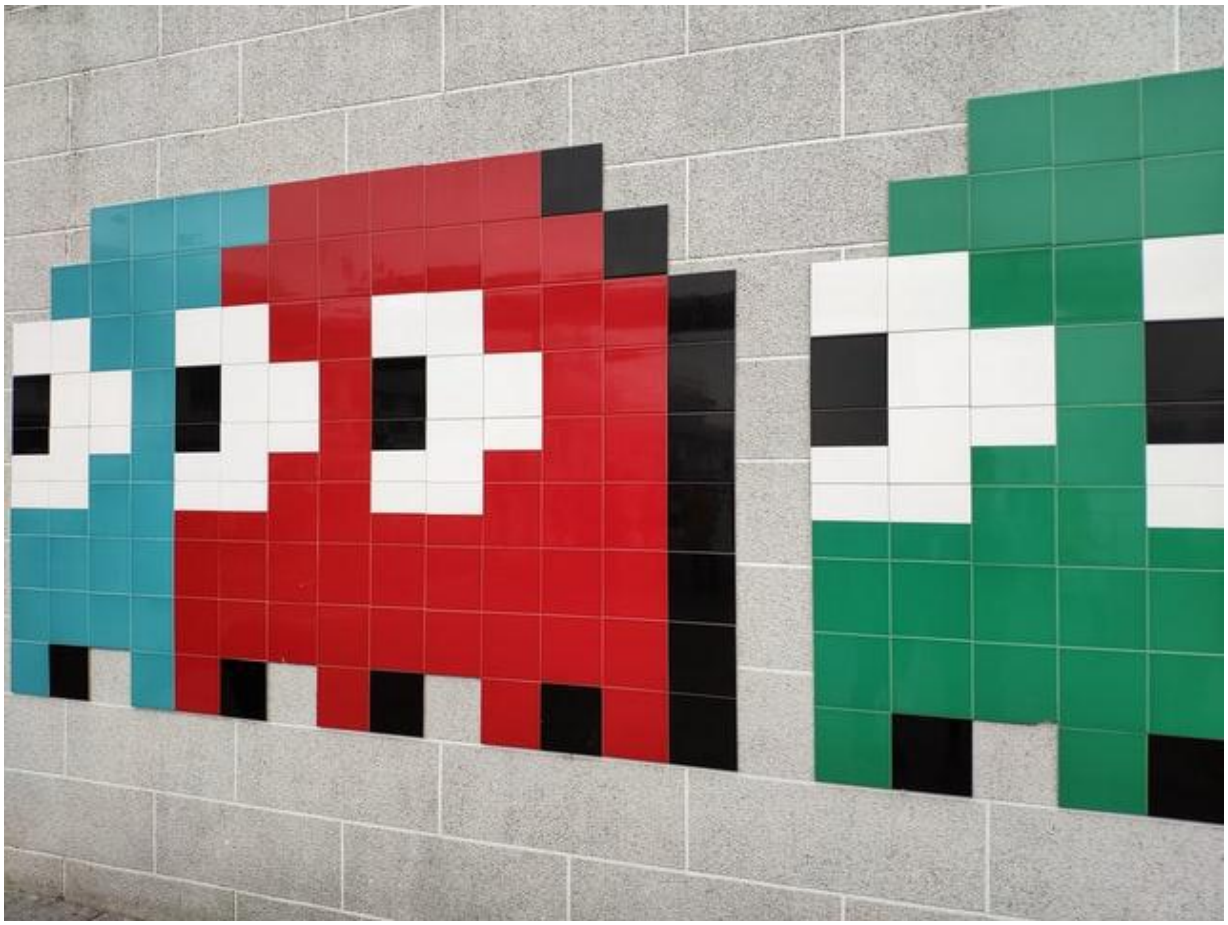
So how does it work? By default, most editors go with fours spaces per indentation, but there is no real rule about the number of spaces. Some programmers change this setting because they like 2 spaces instead. Others prefer 1 space. It all depends on each individual, but the standard is usually 4, which is also quite close to a tab space in size. Here's how a correct indentation should look, using one of our earlier examples:



Notice how it looks neat and it's easy to understand what's happening. The 'while' loop is the boss and the print statement, if statement, and the x+=1 increment statement belong to it. Then we can clearly see that the 'break' statement is only part of the 'if' statement. Without indentation we would get nothing but errors, and even if that didn't happen, the code would become a hard to read puzzle.

Be careful, and keep an eye on your indentation to keep your code organized and error free!

Chapter 3: Coding Games and Apps



First of all, congratulations, you're a programmer! No, you aren't just a student, or someone picking up coding as a hobby, you're a real programmer. You already created a couple little programs, and that makes you worthy of the title. After all, you wrote some code and told the computer to do what you want, right? Think like a programmer, feel like a programmer, be a programmer!

You are now ready to go further and learn how to develop applications and games from start to finish.

In this chapter, you're going to learn slightly more complex coding techniques, like using functions, and you'll find out how games are designed and developed by creating some of your own. In addition, you'll also learn more about those pesky errors that keep popping up when you're just trying to have some fun. Let's get started!

Fun with Functions

Writing code can be time-consuming as programs become more complex. In some cases, certain code blocks have to be repeated several times throughout the program to perform similar tasks but with different values. However, coders don't like a lot of code, so they're always on the lookout to squeeze more out of less. In other words, they want shortcuts, and we want them as well. This is where functions come in.

So, what are functions? Well, they're fun of course! It's in the name. But more importantly, functions allow us to take a block of code that does something, and create a shortcut for it. This means if we need that code later in our program, we won't have to write it again. We'll just point at the shortcut, or call it in programming terms, and that's it!

To create a function, we first give it a name, the same way we name a variable, and then open up a pair of brackets where we type a set of parameters. In programming, a parameter is just some information that the function will use when we call for it. Imagine parameters like variables that are part of something larger, namely a function. But before we dive into function definitions, you should know that Python already has a number of predefined functions. In fact, you've already been working with functions without even knowing it.

Go back to our previous examples and try to guess which bit of code fits the description of a function.

Found it? Take a look at the 'print' function or the 'input' functions. How did we declare them? They have a name, and they clearly do something without us having to tell the computer how to print or ask for an input. All we do is fill the brackets with parameters, like a "Hello, World!" string. As mentioned, we can also pass variables as function parameters. That's exactly what we do when we create an x variable, give it a number or a string, and then we type "print(x)".

But there are other predefined functions as well, for instance the "min()" and "max()" functions. You can use them to find out which number or variable is the smallest or largest. Here are some examples:

>>> min (3, 25, 44, 12, 109)

The result is 3. The ‘max’ function works the same way, but it gives us the largest value.

You already know how to call these predefined functions, but depending on the type of information, there’s another way to access them. Not all data types are created equally and many of them have their own functions. For instance, you might find predefined functions that only work with strings and nothing else. In that case you need to either declare the data type, followed by a dot, the function, and the brackets, or instead of the data type you call your own variable and do the same. Just keep in mind that when you use your variable, you need to know what kind of information it contains. If you have a string variable, then you can use only the predefined string functions. Here’s a simple example:

>>> ‘i love python’.upper()

I LOVE PYTHON

Did you figure out what we did here? We declared a simple string and then called the “upper()” function, which is a predefined string function. This means you can only call it when you use a string, either on its own, or in the form of a string variable. What the function did was turn all of the lowercase string letters to uppercase (capital letters). You should also take note of the empty brackets at the end of the function. That’s where parameters go, but they aren’t always needed, they’re optional. So in this case, the brackets stay empty, but we still have to type them otherwise the program won’t see the function.

So far, we’ve been using the predefined functions. But, how can we make our own so that we can code complex games and apps?

How to Define a Function

To define (create) a function, we need to start by using the “def” keyword, which stands for “define” or “definition”. Launch IDLE, save a new Python file, and give this code a shot:

>>> def greeting():

```
    print (‘Hello, function!’)
```

So, we created a function and attached some instruction to it, but nothing happened. That’s because we need to call the function. Without doing that, we just have a definition that the program stored to be used whenever we want it. To call the function, simply type:

```
>>> greeting()
```

You can call the function from anywhere in the program however many times you want.

With that in mind, here are a couple of tips on how to define your functions to make it easier for yourself and other coders:

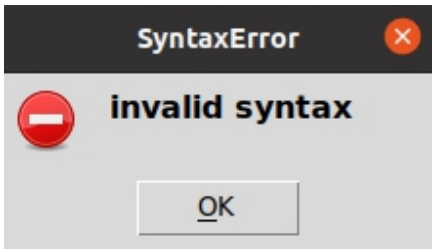
1. **Start from the top** : Since functions can be used throughout the program, you should write their definitions at the top of your code. Even if you’re halfway through your program, when you need a new function, you scroll to the top of your code file, and add in the new function. This way you have them nicely ordered and you can easily make changes when necessary. By spreading out your functions, you’d just create a whole lot of chaos.
2. **Give functions descriptive names** : Just as with variables, you should name your functions in such a way that when you or someone else reads them, you instantly understand what they do. So if the purpose of your function is to convert minutes to seconds, name it “convert_minutes_to_seconds” or “minutes_to_seconds”. The naming rules and recommendations are the same as for variables, so take some time to refresh your memory.

Swatting Bugs

Nothing is more annoying than having a great idea, writing the code for it, and then having your progress stopped by an error. Fortunately, Python likes to give you some clues about what went wrong. Actually, you might even like fixing bugs because they feel like puzzles, and solving a puzzle

might bring you joy. After all, there are a lot of people building a career out of doing that.

If you're following the book and you're using IDLE or the online Python shell, you probably already experienced a few error messages when you made a mistake. These messages are important because they tell you what kind of mistake you made, and then you know where to look. Here's what these messages look like if you were awesome enough not to see one yet:



In this case, Python alerts us that we made a mistake when typing the code. Maybe we misspelled a command or forgot to close a bracket. When an error like this is encountered, the program is stopped, you get the error window like in the above image, and then the code editor highlights the row that's causing the problem. Most editors take you to that row immediately and under it the type of error is mentioned again in more detail. But if you're writing a more complex program, like a game, the error message might be displayed in its own window, depending on the code editor, and then you have to right click on that error and select the "go to line" option.

With that in mind, here are the most common types of bugs:

1. **Syntax errors** : As mentioned, this is probably the most common bug because misspelling code is quite easy to do. They are easy to fix because the editor points out the line with the error, and then when you read it you'll see you missed a letter or forgot a quotation mark to close your string. There are even fancy code editors that spot the mistake as soon as you make it, and they suggest the fix. It's kind of like when you type "dogg" in Microsoft Word or the Google search engine and the system automatically corrects it to "dog". Visual Studio can do that, but that development environment is a bit more complex than our humble IDLE code editor, so stick to this for now.

2. **Indentation errors** : We talked before about the way Python sees blocks of code. It uses the indentation to see how the code is arranged and which line is part of another. Without it, Python doesn't know where a certain instruction is supposed to end. IDLE automatically creates the indentation for you, but if you make a mistake you'll get an indentation error. This is easy to fix; just go back to the troublesome line and change the indentation.
3. **Type errors** : Don't confuse this bug with typing errors because they're not the same thing. Type errors happen when you confuse data types. For example, if you try to multiply two strings, you'll get a type error because the multiplication operation works only with numbers. You can also get this error when you try to combine two different data types. For instance, if you tell your program to see if a number is smaller than a string, you'll get a type error because they're two different data types that don't mix in that operation.
4. **Logical errors** : These are the most challenging ones to deal with because they happen even if your code is 100% correct. Logical errors have to do with program logic, or the way you code the program to run the instructions as you designed them. So, even if you typed the code well, when you run the program it doesn't do what you expected. This means there's a mistake in your logic. Here's an example:

```
>>> player_health = 100
>>> print (player_health)
>>> player_health = player_health - 1
```

Notice anything wrong with the code? It runs perfectly, doesn't it? That's because it is indeed correctly written. But, it doesn't work as it should. We're displaying the player's health before he's taking the -1 damage that's making him lose a health point. The last two lines of code should go the other way around. That's where the logical error is. The code didn't work as we intended through our design.

Finding logical errors is difficult because there's nothing for the code editor to show us. It can't tell us "Hey, your monster is actually supposed to club the player not hug the player" because it doesn't

know. Remember that computers are both smart and dumb and we have to tell them everything they need to do. To find the logical error when your game or program doesn't work like it's supposed to, you have no choice but to slowly go through your code, read it, and see what you did wrong.

Fixing bugs is part of the coder's life. There's no such thing as a bug-free program or game. That's why all software receives regular updates. When was the last time you played a game without any bugs? Did your character get stuck in the wall and couldn't escape? That's a bug. Did your character enter a dungeon but there were no monsters around and you had to restart the game? Yup, bug.

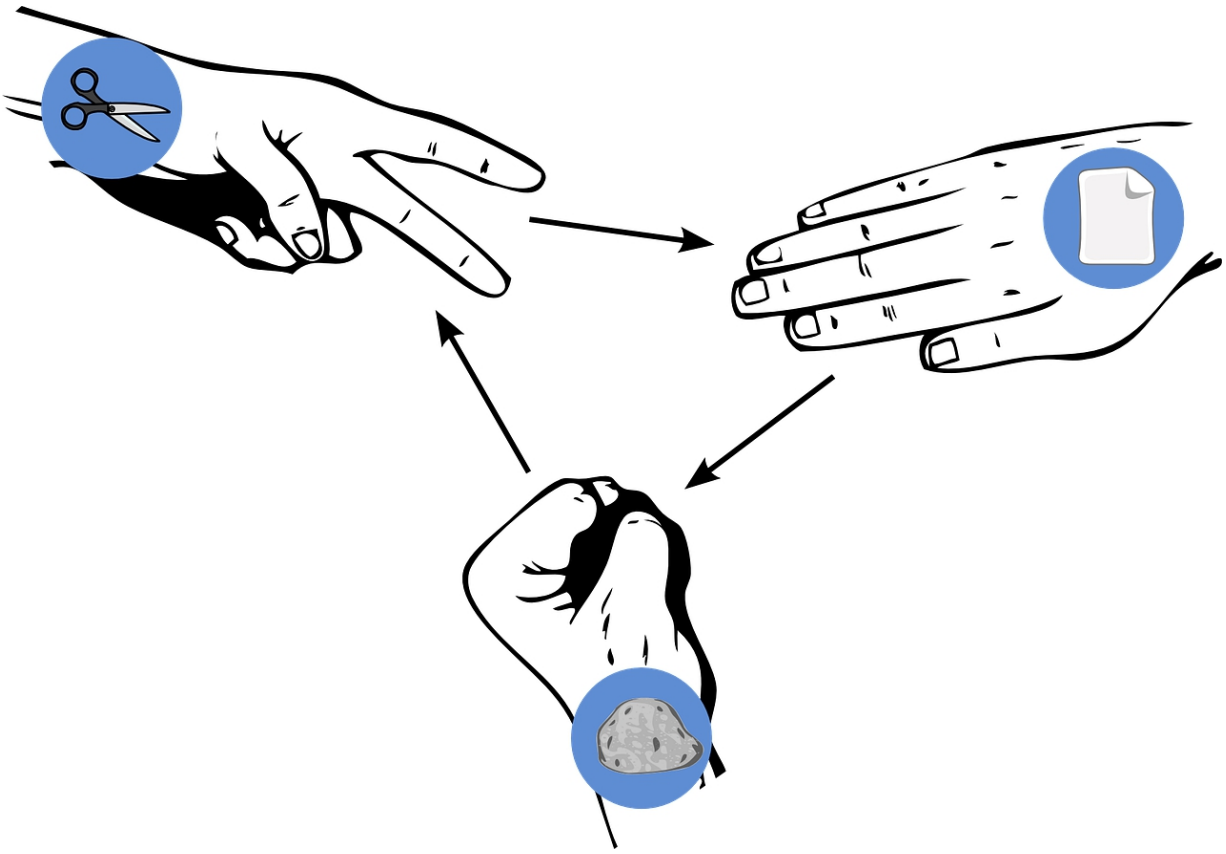
There will always be bugs, but we can make the program or game work well enough by going through the following checklist:

- Is the indentation as it should be?
- Are all keywords spelled correctly and you didn't miss any opening or closing brackets and quotation marks?
- Did you write a number instead of a letter, or a dash (-) instead of an underscore (_)?
- Did you mislabel a variable or function with an uppercase letter when it should be lowercase, or the other way around?
- Did you save your code?
- Does the program work as you designed it?

If you ask yourself these simple questions, you'll be able to solve most errors and deal with the tricky ones after the program is at least functional.

Now that you have all the tools you need to create a more complex program, it's time to use your Python creativity and have some fun!

Project 1: Rock Paper Scissors



This is going to be the first game you'll create using everything you learned so far. But to get started, you first need to think about the design of the game. Remember, you should spend some time with a pen and a piece of paper and write down the rules of your game, and in which order things need to happen. That way, when you open the code editor, you know how to start and avoid future logical errors.

So, what are the rules of 'Rock Paper Scissors'? It requires two players, in this case you and the computer, and both need to pick one of the three items. When both players choose one of the three options, their choices are revealed. The rock smashes scissors, scissors cuts the paper, and paper covers rock. Now that we know the rules, we need to think about how to implement them in code.

The easiest approach is to create a list containing the rock, scissors, and paper items. The player and the computer will pick an object from this list. Now the problem is, how is the computer going to make that choice? We're going to keep it simple to avoid creating Skynet by accident, and give the computer a random choice generator. The player (you) will choose one of

the items, and the computer will automatically pick one randomly. To handle the program's ability to figure out who won, we're going to write a few "if" statements. Let's go through the program and see how it all comes together:

```
>>> import random
>>> game_choice = ['rock', 'paper', 'scissors']
>>> print ('Rock crushes scissors. Scissors cut paper. Paper covers rock.')
>>> human_player = input ("Rock, paper, scissors? (Exit) ?')
>>> while human_player != 'exit':
        human_player = human_player.lower ()
>>> AI_player = random.choice (game_choice)
>>> print ('You picked' +human_player+ ', and the AI picked ' +AI_player+ '.')
>>> if human_player == AI_player:
    print ('It's a tie!')
>>> elif human_player == 'rock':
        if computer == 'scissors':
    print ('You won!')
        else:
            print ('You lost!')
>>> elif human_player == 'scissors':
    if AI_player == 'paper':
    print ('You won!')
        else:
            print ('You lost!')
>>> else:
    print ('Something fishy is going on around here…')
```

>>> print()

>>> human_player = input (‘Rock, paper, scissors? (Exit) ?’)

That’s it! We’re going to break it down, but before that, I encourage you to try and figure everything out on your own. You should already understand 95% of the code, and if you don’t, go back and learn more about ‘if’ and ‘elif’ statements and come back with fresh strength. If there are other parts you don’t understand, like “import random”, which we haven’t covered in the book, then it’s time to use a coder’s greatest weapon: Google. In the programming world there’s an ongoing joke floating around that programmers are simply really really good at using Google’s search engine to learn stuff. Well, that’s true to some extent. A programmer should train their research skills as well because without a doubt you’ll find something you don’t understand. When that happens, you have to figure things out on your own. So Google “import random” and you should find your answer.

Now, just in case you can’t do the exercises right now, or you’re still struggling, we’re going to break things down together. After all, real-world programming often involves teamwork.

With the first line of code we’re importing a Python module. What’s a module? It’s a package that contains some functions and commands. Without the ‘random’ module, the program won’t know what the “choice” function from “random.choice” means. Packages are useful tools that contain additional functions and code that are used to extend Python’s standard library. So, whenever you see the “import” keyword, it means that we’re bringing in a module.

Once we have the module, we can give the computer the power to choose objects randomly. Afterwards, we create the list for the game items and print the rules of the game for the player to know how the game works.

Next, the human player has to choose which item they want to play against the computer. This is where we use loops to check on what the player chose. As a fourth option, the player can also choose to exit the game. After the player picks what to play, it’s the computer’s turn, which is determined randomly. At this point, the program will analyze the choices made by the players to see who wins. We coded a series of conditionals to see who wins based on the choice.

You can also notice that as a final check, we coded the game to do something in case the player decides to type something other than the three keywords. In that case, the player will receive an error telling them that they made a mistake that caused an error. Remember, computers aren't that smart and you need to use your logic and imagination to prevent future problems.

Finally, when the winner is determined, the game will start another round.

Homework

As mentioned in the first chapter, anyone can learn how to program and no such thing as talent is required to be a good coder. It just takes practice, and a lot of it. As you learn from this book, you're probably copying code to try things for yourself, and that's good! But as a second step, you should try to recreate things yourself, but a bit differently to make it more fun. You can't skip this part of the learning process. Even if you make mistakes, with each one of them, you'll learn something that will stick with you and make you a better programmer.

"Experience is the name everyone gives to their mistakes." – Oscar Wilde

So, here's what you can do. Follow all the steps we talked about so far, grab a pen and piece of paper, keep Google handy, and start replicating and even expanding your game using what you learned.

Start by changing the rules if you can think of something more exciting, and then make modifications to the game's code to match the new rules. Add more variables into the mix to make the game more difficult. For example, you can add two more items, and the game already becomes more complex and interesting.

Did you ever watch the *Big Bang Theory?* In that show, the traditional Rock Paper Scissors game was expanded to Rock, Paper, Scissors, Lizard, Spock. You can recreate that game by adding those two additional items on your own, including the conditional statements that come with it. Here are the rules to that game in case you don't already know them: scissors cuts paper, paper covers rock, rock crushes lizard, lizard poisons Spock, Spock smashes scissors, scissors decapitates lizard, lizard eats paper, paper disproves Spock, Spock vaporizes rock, and rock crushes scissors. Quite a

mouthful, isn’t it? Give it a shot on your own, and don’t forget to go through the error checklist.

Project 2: Roll the Dice

In the previous project we used lists to store the game objects and then test to see which one wins against another. For this project we’re going to use lists again, but this time we’ll be generating five random dice and then run a test to check if the player rolled three, four, or five of a kind. Essentially, we’re creating a basic version of Yahtzee. In the actual game, each roll is worth a different number of points, but we’re just going to focus on telling the player what they rolled.

Now that we know the rules, let’s plan how we’re going to program the game. First, we’ll need a loop that allows the player to automatically restart the game after every turn. This is something we also used in our previous project. We only want the game to shut down when the player chooses to quit. Then, we need to create a list with the five dice. To do that, we’ll have

a list with numbers ranging from 1 to 6, and each number will represent the value of the dice. To program the roll of the dice, we need to randomize the values. So we'll have a random number from 1 to 6 for every single dice. Afterwards, we compare the dice to each other, to test if the player rolled three, four, or five identical dice.

This last part of the program is going to be the most difficult. Stop for a moment and try to think how you can do that. Don't forget to use a pen and piece of paper for the design phase to write down your ideas.

What we can do is tell the program to check if all five dice have the same value by first checking if they're all equal to 1, then if they're not we check to see if they're all equal to 2, and so on until we program for every possibility. This sounds like a simple plan because it involves writing a bunch of "if" statements. But it's going to take us a lot of time to write all those statements for each situation. So, how can we simplify the game?

This is the part where brainstorming and that piece of paper become pretty handy. First, what's really important is whether we have a dice roll with five of a kind. It doesn't really matter if we roll five 1s or five 5s. What really matters is that we roll five of a kind. To do that we can simply compare the values of the dice with each other. We can run a test to see if the value of the first die is equal to the value of the second, third, fourth, and fifth. This way we know that we have a five of a kind. But the next question is, how are we going to test for just four of a kind?

The first solution that might pop in your head might involve creating lists that would look something like this [1, 1, 1, 1, 2]. Here we have four of a kind and a fifth die that doesn't match. That's good, but we can also have [1, 1, 2, 1, 1] and many other variations, and that's just when we involve a value of 2. We still have the 3s, 4s, and 5s. That's a lot of possibilities that would mean writing a lot of 'if' statements.

To solve that, we need to look at these lists and see what they have in common. They have four 1s, and a 2 that can be in any of the five spots. But to the game it doesn't matter where that 2 is located. Instead of testing for so many cases, we could just test for two by having all the 1st lumped together, and 2 separately. But to do that, we need the ability to sort the list to go both from the lower value to the highest, and from the highest to the

lowest. This way the dice will look like this: [1, 1, 1, 1, 2] and [2, 1, 1, 1, ,1].

So let’s start with the dice sorting.

Sorting Lists

In Python we can sort our lists using the “sort()” predefined function. This allows us to sort all values from small to large and the other way around. Take note that the sort function has a default setting and if you don’t specify anything, the values will be sorted automatically in an ascending order.

Now, back to our dice game.

The problem: we need to check if the list has four of a kind dice.

The solution: We sort the list so that the program performs only two tests. One will check if we have four identical small values and a higher value, and the second will check if we have a single small value and four identical large values. However, in either case, we’ll have a die that will also be present in the five of a kind roll.

Our solution so far seems quite good, but it comes with a logical problem that we have to fix. What if we roll a five of a kind? We told the program that if we have four identical values, we have four of a kind. But a five of a kind roll also contains four identical values. In other words, we have a four of a kind roll inside the five of a kind roll. We need to solve this by telling the program to ignore the four of a kind if we have a five of a kind roll. This can be achieved with an if/elif statement block that instructs the program to focus on the largest score. So, here’s how this solution will look in code:

```
>>> if myDice[0] == dice [4]:
        print (‘You win!’)
>>> elif (myDice[0] == myDice[3]) or (myDice[1] == myDice[4]):
        print (‘Big Four!’)
```

Don’t forget that in programming 0 is the first item, not 1. With that in mind, notice that when the first dice and the last one have an identical value, we have a five roll win. All dice are the same from 0 to 4. If we don’t

have a five of a kind roll, then we move on to test for the four of a kind roll. In this case, we either have the first four dice or the last four dice with identical values. Here we use a Boolean to see which of the two options is True.

Notice that when we're talking about the dice in programming terms, we use the index to determine the one we're looking at. In other words, when we use lists we usually use the location of the object inside the list in order to use it. So "myDice[0]" is die number one because in programming numbering starts from 0, "myDice[1]" is the second die, and so on. Using this index method we can compare the dice to each other and more.

Now, we still need to program the game to look for a three of a kind roll. To do that, we are going to use an elif statement right after the four of a kind roll. This way, the program will check for each case in a proper order. If we don't have a five of a kind roll, the computer will look for a four of a kind roll, and if we don't have that either, it will test for a three of a kind roll. We're always interested in the roll that brings the highest score. So how do we program the three of a kind roll? There are three cases we need to test. We can either have the first three dice rolling the same number, the three in the middle, or the three at the end of the list. Here's how the code would look:

```
>>> elif (myDice[0] == myDice [2]) or (myDice[1] == myDice [3]) or (myDice[2] == myDice [4])
        print ("It's the three musketeers!")
```

Now that we have solved all of our logical problems and know how to test for each dice roll case, it's time to put everything together.

The Final Game

Again, you should try to create the game on your own using what you learned so far, and using the bits of code we already solved. As a small hint, you'll need to use the "random" module again, so don't forget to import it. It might be difficult, but through trial and error you learn more than anything.

With that in mind, here's the entire solution for the game so that you can compare your own work to it:

```
>>> import random
>>> play_again = True
>>> while play_again:
        myDice = [0, 0, 0, 0, 0]
        for i in range (5):
        myDice[i] = random.randint (1, 6)
        print ('Your dice roll is:', myDice)
        myDice.sort()
        if myDice[0] == myDice[4]"
        print ('You win!')
        elif (myDice[0] == myDice[3]) or (myDice[1] == myDice[4]):
                print ('Big Four!')
                elif (myDice[0] == myDice [2]) or (myDice[1] == myDice [3]) or (myDice[2] == myDice [4])
                        print ("It's the three musketeers!")
                play_again = (input (' Press [Enter] to play again, or press any other key to quit: ') == '')
```

Once we import the necessary module so that we can use the random functions, we need to start the game with a while loop. Remember, we want the player to be able to play as many rounds as they want. Without a loop, the game will shut down after one dice roll.

Inside the loop, we place all of our game code by starting with the "myDice" list that contains the values associated with the dice. At the start they're all set to 0. Then we create a 'for' loop that will run for each die, going between 0 and 4, the index of each die.

In the next step, we assign a random whole number (integer) from 1 to 6 to all the dice. These values represent the dice roll. Then we print the player's

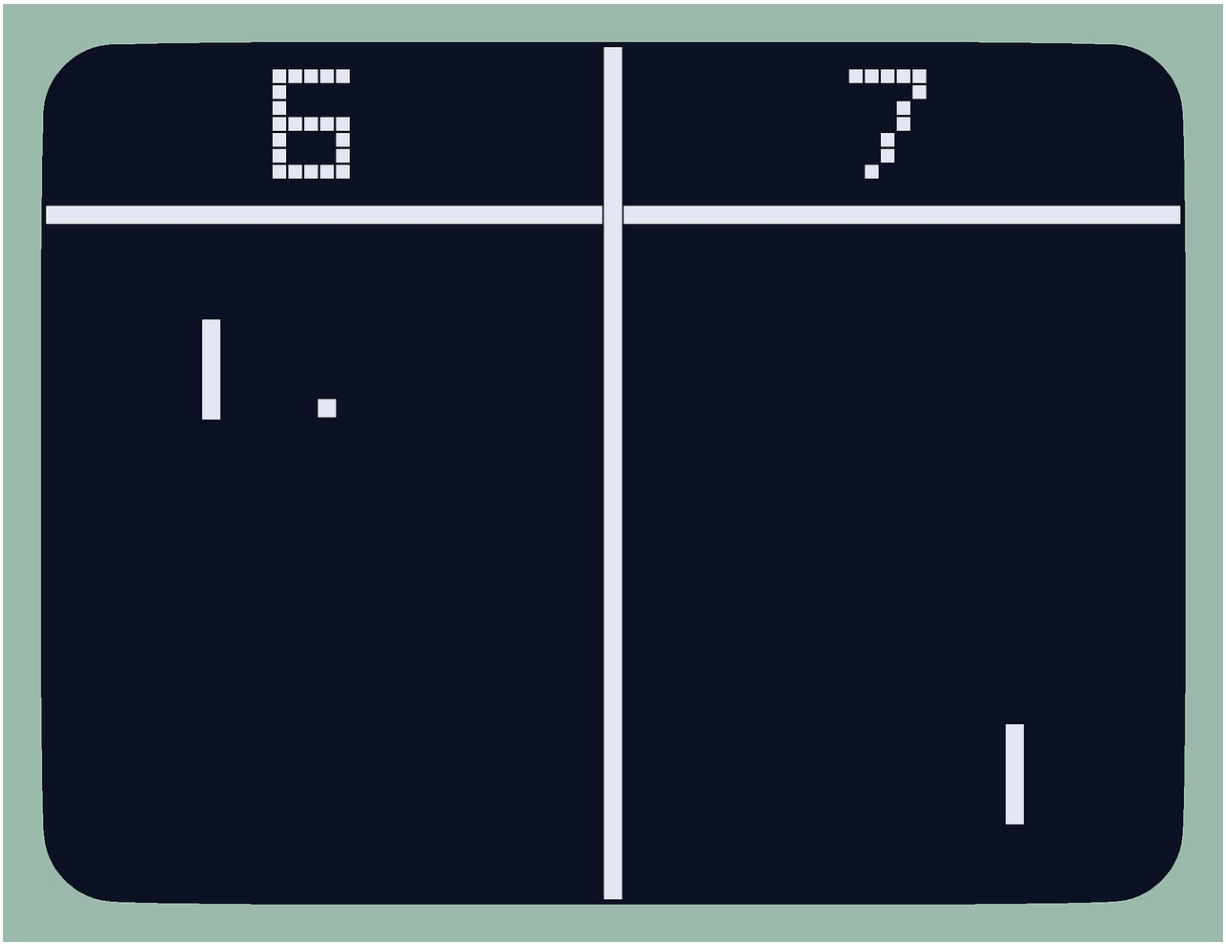
rolls.

Finally, we use the ‘sort’ function on the list of dice so that the program can run the tests to see whether we rolled a three, four, or five of a kind. The values of the dice are sorted from small to large. This way, instead of getting a roll that looks like this: [2, 6, 2, 3, 2], we get this: [2, 2, 2, 3, 6]. The if/elif statements run in an order, testing to see which type of roll we have, like we discussed in the section above.

Now you can run the program and hit the Enter key to keep rolling the dice and see what you get. You might have to roll hundreds of times to get a five of a kind, or you might get a lucky first roll.

It’s not a complicated game and it won’t entertain you for hours and hours, but it’s yours. You should feel proud of every game you create from start to finish.

Project 3: A Simple Game of Pong



In this section you're going to learn how to create your first arcade game. What arcade game is more suitable than one of the first ever created, Pong? You're going to use everything you learned and more for this project. You'll draw graphics, create animations, and program player controls like moving the mouse, but first you'll write down the design as always.

So, here's what we need to create a Pong clone:

1. We need an **environment** , or a game board. To keep it simple, you can use any background you want, but we're going to stick to the classic version of the game that had a simple black background. You can change the color to anything you want.
2. We need to establish the **player's goal** . The player needs to keep scoring against the computer and accumulate a higher score.
3. **Game characters** : In this case they're objects, and they come in the form of a paddle and a ball. The player uses the paddle to hit the ball.

4. **The rules of the game** : Whenever the player hits the ball against the edge of the screen on the opponent's side, he will score a point. If he doesn't hit the ball and it goes on the edge of his side, then the opponent gets a point.
5. **Mechanics** : The player controls the paddle with the mouse.

All of these elements can be found in any game, from Minecraft to Assassin's Creed and Halo. Now, let's get started!

The Skeleton of the Game

Remember to divide your project into a series of steps. It's much easier to focus on small problems at a time instead of constantly worrying about the entire game. So, how do we start? Well, since this is a more complex game, we need a special Python library used for game development. It contains many predefined functions and even graphics and sounds. Its name is 'pygame' and we also need to initialize it for all of its modules to work, like this:

```
import pygame

pygame.init()
```

Next, we can define the colors we're going to use throughout the game. By assigning them to variables instead of using them directly, we can always call on them without thinking about color codes constantly. Since Pong is an old game built in the 70s, we're going to stick to the original black and white color scheme, but you can choose anything you want. Just keep in mind that colors need to be written in their RBG codes, like this:

```
BLACK = (0, 0, 0)

WHITE = (255, 255, 255)
```

Now, let's think from the very beginning. How does the game start? In a window of its own. What will this window look like? Well, we need to write some code to describe it to the computer:

```
size = (800, 600)

screen = pygame.display.set_mode (size)
```

pygame.display.set_caption (‘Pong Clone’)

Now that we have the basic visuals out of the way for now, we need to figure out what to do next. Do we draw and animate the paddles and ball? Do we program some game logic? Not just yet. The easiest approach is to create the main loop that will contain all that information, just like we did with the previous games we created. Here’s how Pong’s shell should look:

```
continue_playing = True
system_clock = pygame.time.Clock()
# Main loop
while continue_playing :
    # Event loop where the player does something
    for event in pygame.event.get():
        #If the player closes the game he will exit
    if event.type == pygame.QUIT:
    # The loop stops running because the player clicked to exit
    continue_playing = False
    # Game logic will go here later
    ----------------------------------
    # Graphics
    # The background will be black
    screen.fill (BLACK)
    # We need to split the screen in half to mark the playing fields
    # Think of the net that splits the ping pong table
    pygame.draw.line (screen, WHITE, [399, 0], [399, 600], 5)
    #Let’s update the screen to see the graphics
    pygame.display.flip()
    # We want the game to refresh/update the screen
```

```
        # at a speed of 60 frames per second
        system_clock.tick(60)
#Close the game once the program loop stops running
pygame.quit()
```

Doesn’t the code just look clear and descriptive? You can understand everything, don’t you? That’s because we used a handy programmer’s tool that becomes necessary when programs become more complicated.

Now save this program file and call it main.py. Since this program is more complicated, we’re going to learn how to divide it into several files that will work together to create the game.

But before we continue, let’s take a quick break and talk about code comments.

Code Comments

Python and other programming languages allow us to write notes in their code editors. These notes are handy because we as coders can explain what certain code blocks are used for. When a program becomes long and complicated, these comments will remind us what we intended to do. Just imagine working on a project for a few weeks and then taking a break. When you come back, you might not know what you wanted to do, or it will take you hours to figure out. By leaving comments, you leave clues for yourself, as well as other coders that might be trying out your code.

In Python, comments are marked with the hashtag ‘#’ symbol. Everything that comes after a hashtag is a comment. Comments are ignored by the program and not executed like commands are. Just keep in mind that each line that’s a comment has to start with a hashtag. That’s why you’ll notice in our code that a comment is broken into two lines, even though it’s all one sentence.

Use comments as often as you want. You can see that our program is easily readable because each line of code is explained. You don’t have to explain every line of course, and most programmers don’t. You should avoid having

too many unnecessary lines and explaining what's obvious. Instead, you might want to create sections and describe what each one does.

Now, let's get back to our Pong game!

Game Objects

The next step is to create the paddles. These paddles are objects that have a certain color, height, and ability to move in various directions. In programming terms, an object is something that has attributes and methods. The attributes are the physical properties, and the methods are that actions the object can perform, like moving. Objects are built like in a factory, out of moulds, which in programming are referred to as classes.

To save time and not create two paddles that would involve copy pasted code, we're going to first build a paddle class. From this class we will simply clone paddle 1 and paddle 2.

Since we'll have two paddles, we're going to allow two players to play the game. The first player will use the 'w' and 's' keys to go up or down, while the second player will use the up and down arrow keys.

Let's get started! Create a new file and name it paddle.py and type the following code, which you can modify however you like:

```
import pygame

BLACK = (0, 0, 0)

#This will be a class that comes from the graphics sprite class

#that's predefined in pygame

Class Paddle (pygame.sprite.Sprite)

#Next, we need to initialize the properties of the paddle

# and then call its parent class, which is Sprite

        def __init__ (self, color, width, height):

                super().__init__()

                #Define the Paddle color, position, width, and height
```

```
        # The background will be black and transparent
        self.image = pygame.Surface ([width, height])
        self.image.fill(BLACK)
        # Using a rectangle to draw the pallet. You can use other shapes as well
        pygame.draw.rect (self.image, color, [0, 0, width, height])
        self.rect = self.image.get_rect ()
```

The class is ready, and now we can use it to generate the paddle1 and paddle2 objects. But before we can do that, we have to open the main.py file and add some more code there. This is how real game development works and why you need to have a good plan, otherwise you might get lost.

```
# The paddle class needs to be imported, just like modules
from paddle import Paddle
# Now we can create the actual paddles
paddle1 = Paddle (WHITE, 10, 100)
paddle1.rect.x = 20
paddle1.rect.y = 200
paddle2 = Paddle (WHITE, 10, 100)
paddle2.rect.x = 770
paddle2.rect.y = 200
# We declare the paddle sprites we'll use
game_sprites = pygame.sprite.Group()
game_spites.add(paddle1)
game_sprites.add (paddle2)
# Now go back to the game logic section and add this
#Game logic
game_sprites.update()
```

Now that we took care of the objects, we need some controls.

Player Controls

To have control over the paddles, we'll go back to the paddle class and add the methods that will move the paddles with the set keyboard strokes. Whenever the player presses a key, the method will be called, just like a function.

Open your paddle file and add these lines of code:

```
def goUp(self, pixels):
        self.rect.y -= pixels
        #Need to limit movement not to go beyond the screen
        if self.rect.y < 0:
                self.rect.y = 0
    def goDown(self, pixels):
                self.rect.y += pixels
        #Need to limit movement not to go beyond the screen
        if self.rect.y > 400:
          self.rect.y = 400
```

Our two methods take two parameters: self and pixels. "Self" refers to the object and "pixels" refers to how many pixels the paddle will move in one go.

Now, let's go back to the main game file and add the player movement interactions inside the main game loop.

```
# Paddle movement
# Player 1 = W and S keys
# Player 2 = arrow keys
movement_keys = pygame.key.get_pressed()
```

```
if movement_keys [pygame.K_w]:
        paddle1.moveUp(5)
if movement_keys[pygame.K_s]:
        paddle1.moveDown(5)
if movement_keys[pygame.K_UP]:
        paddle2.moveUp(5)
if movement_keys[pygame.K_DOWN]:
        paddle2.moveDown(5)
```

Save the file. We now have control over the paddles! Let's program the ball.

The Ball and Movement

We need to create the ball and program it to bounce when it hits something. It has to behave exactly like a real ball would. This means we need to add collision detection so that the game detects what happens when the ball hits the paddle, or the edge of the screen.

Let's start by creating a new file called ball.py and write the following code:

```
import pygame
from random import randint
BLACK = (0, 0, 0)
Class Ball (pygame.sprite.Sprite):
#The ball class works almost exactly like the paddle class we had earlier
        def __init__ (self, color, width, height):
        super().__init__()
         #Same as with the paddles, we need to draw the ball and give it attributes
        self.image = pygame.Surface ([width, height])
```

```
        self.image.fill (BLACK)
        self.image.set_colorkey (BLACK)
        pygame.draw.rect (self.image, color, [0, 0, width, height])
        self.velocity [ randint (4,8), randint (-8, 8)]
        self.rect = self.image.get_rect()
def update (self):
        self.rect.x += self.velocity [0]
        self.rect.y += self.velocity [1]
```

Remember, this is the class that describes the ball and defines it. We now have to use it to create the ball as an object inside the main game file. Open the main file and add the following code:

```
ball = Ball (WHITE, 10, 10)
ball.rect.x = 395
ball.rect.y = 295
#Add the ball to the sprites list
game_sprites.add(ball)
#Perform test to see if the ball bounces against the sides of the screen
if ball.rect.x>=790:
        ball.velocity[0] = -ball.velocity[0]
if ball.rect.x<=0:
        ball.velocity[0] = -ball.velocity[0]
if ball.rect.y>590:
        ball.velocity[1] = -ball.velocity[1]
if ball.rect.y<0:
        ball.velocity[1] = -ball.velocity[1]
```

Finally, we need to make the ball bounce in a random direction when it detects the paddle. Open the ball.py file and add the bouncing method like

this:

```
def bouncing (self):
        self.velocity[0] = -self.velocity[0]
        self.velocity[1] = randint (-8, 8)
```

Now go back to the main file and write these lines to tell the program to detect the ball hitting the paddles:

```
# Collision detection
if pygame.sprite.collide_mask (ball, padde 1) or pygame.sprite.collide_mask (ball, padde 2)
        ball.bouncing()
```

Phew, that’s a lot of code isn’t it? But we aren’t quite done yet. We need to program the game to keep the score, otherwise how do we know who wins?

Keeping Score

The score system is simple. When player one paddles the ball and it hits the right edge of the screen they score a point. The same thing applies to player 2 when the ball hits the left side of the screen. The score will be kept at the top side of the screen.

Start by updating the main program file with these two lines of code:

```
#Player score initialization
score1 = 0
score2 = 0
# Detection when a player obtains a point
# This is an update to the wall bounce check
#Perform test to see if the ball bounces against the sides of the screen
if ball.rect.x>=790:
        score1 += 1
```

```
        ball.velocity[0] = -ball.velocity[0]
if ball.rect.x<=0:
score2 += 1
        ball.velocity[0] = -ball.velocity[0]
# Handles the score display
font = pygame.font.Font (None, 72)
text = font.render (str (score1), 1, WHITE)
screen.blit (text, (300, 10))
text = font.render (str (score2), 1, WHITE)
screen.blit (text, (470, 10))
```

And we're done! Enjoy the game with a friend or with your parents.

But most importantly, go step by step through the entire code and work to understand it. It's more complex than what we used for the 'roll the dice' game, but you can only learn through challenges.

This code is explained only briefly through code comments on purpose. Remember, a huge part of being a programmer is learning to research. So, if you don't understand something, go back through the book one more time or do some research of your own using Google. Solve one problem at a time until you run out of problems. And finally, find a community of fellow Python programmers to learn more and work with others.

Joining a Community

Programming usually involves a team effort, and learning is a lot more fun with others. But what do you do if your parents and friends aren't that much into programming? Join one of the many online communities that cooperate and work together to create cool apps and games!

Online communities aren't just forums where we argue about the best version of Python. People from all over the world gather together to ask each other questions and find solutions to various problems. These

communities are also places where you can make friends or join group projects. Creating something on your own can take a lot of time, so joining other people would speed things up. Plus, you could learn a lot from other programmers. With that in mind, here are some of the most popular communities:

1. Python's community: On Pythons home website, you can find a community section. You can find a lot of advice and learning sources that can help you progress further or clarify things you don't fully understand yet. Follow this link to check it out: https://www.python.org/community/ .
2. PySlacker's community: This group of enthusiasts found at https://pyslackers.com/web contains a lot of learning resources for their members. They also use Slack to communicate with each other and work on community projects.
3. StackOverflow: If you have a question, this is the place for you. Stack Overflow is a website where programmers go to ask and answer questions related to programming, no matter the language. If you can't figure something out, head over to their website at https://stackoverflow.com/ and ask away!

Don't be shy and start exploring on your own! You know the basics, so you're ready to join the world of programming.

Conclusion

If you made it this far, it means that you have what it takes to become a programmer: persistence and determination. Programming isn't about talent, or being a math genius. It's all about being curious, willing to learn, and practicing. A lot of practicing. So, good job for staying on track and getting this far!

The goal of this book was to teach you the fundamentals of Python programming. There's still a long journey ahead of you, but now you have all the tools needed to continue learning and practicing on your own or with others. You can even start exploring other languages like Ruby, Java, C#, C++, and many more.

If you're still not that confident in your abilities, here's what you learned so far:

1. You learned how to work in a code editor and explored all of its features that are found in most development environments.
2. You explored the fundamentals of programming that you'll find in nearly all programming languages. In the second chapter you learned about variables, conditionals, loops, and how to make your program intelligent. All of this knowledge can be applied in other programming languages. The only thing that changes is the syntax, which is only a matter of practice. Mastering the concepts is what's truly important. So go back to this chapter and solidify your knowledge because you'll apply everything you have there no matter what you're coding.
3. In chapter three you started exploring more complex concepts, like functions. Through real projects, you also learned how to apply them and how to combine them with lists and conditionals. Working on personal projects is one of the best ways to learn because you'll encounter challenges. These challenges might make you feel like you don't know much, but a bit of brainstorming and additional research will provide you with the solution. And when you solve that puzzle, you'll experience the best feeling in the world: the feeling of accomplishment.

Don’t underestimate yourself and what you’ve learned. Programming can’t be learned over night, or over a week, or even in a month. Be consistent. Read more, join communities, and ask questions. Create projects with others. That’s how you learn. Before you know it, you’ll be a full time game developer, or create the next revolutionary piece of medical software. The power is in your hands. You just need to write it down and turn it into code.

References

All images sourced from Pixabay.

Using Python on Windows — Python 3.8.5 documentation . (n.d.). Docs.Python.Org. Retrieved July 26, 2020, from https://docs.python.org/3/using/windows.html

Modules — Python 3.8.0 documentation . (2019). Python.Org. https://docs.python.org/3/tutorial/modules.html

Beazley, D. M. (2009). *Python essential reference* . Addison-Wesley.

BeginnersGuide/Download - Python Wiki . (2019). Python.Org. https://wiki.python.org/moin/BeginnersGuide/Download

Miller, B. N., Ranum, D. L., & Anderson, J. (2021). *Python programming in context* . Jones & Bartlett Learning.

Paul Vincent Craven. (2016). *Program arcade game s : with Python and Pygame* . Apress.

Vorderman, C., Steele, C., Quigley, C., Mccafferty, D., & Goodfellow, M. (2018). *Coding games in Python* . Dk Publishing.