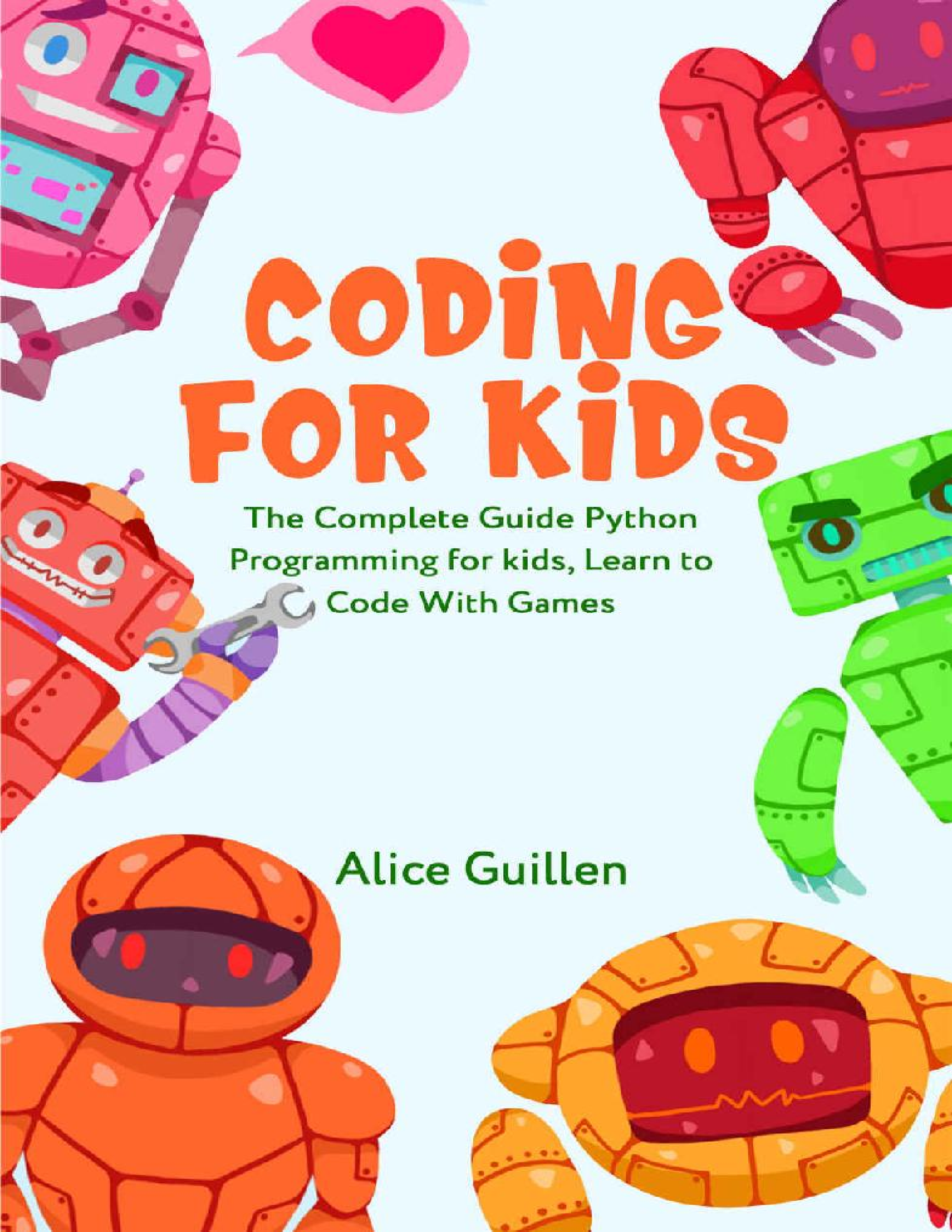


CODING FOR KIDS

The Complete Guide Python Programming for kids, Learn to Code With Games

Alice Guillen

Coding for Kids:
The Complete Guide Python Programming for Kids.

Learn to Code with Games

ALICE GUILLEN

Contents

Introduction: Welcome To (Or Back To) Programming

Welcome. You are the character in this game. Your goal is to finish this course and start thinking like the computer, so that you can give the computer instructions that it will follow and do whatever you please.

You have opened the world that tomorrow has never seen. You are a hero. You are about to learn an excellent skill that will one day save the world.

This book will give you the biggest magic wand because you are curious and cool. Some dumbass doesn't even know anything about Python. This book will help anyone learn how to code in Python.

Coding is used in almost everything we use and love. You can create a program to make games, music, and even develop robots. You can power almost anything electronic. This book gives you that skill.

Programming means translating human ideas and actions into a computer's language. Computers all around the world understand the same set of languages, and Python is one of them. There are many others, like

JavaScript, C#, Ruby, and C++. Programmers use these languages to tell the computer how to do something, but every language is different. This book will teach you Python because it is very similar to the human English language.

Everything you need to become a programmer is this book and a computer. I will guide you through several coding concepts with step-by-step guides and examples so that you will practice on the ground.

When you finish reading this book, you would have become a programmer. Do you have any ideas you want to give life to? Like some cool programs or simple games that you can play with your friends!

We learn coding by practicing. That’s why you would find each chapter in this book structured to walk you through the code as you follow along. As you’ll learn about a concept, you write some code, you understand what that code does, and learn a bit more. You would even be able to fix a bug or two, and see the results of your code in real-time!

This book helps you better understand the big concepts in coding or computer programming and learning Python language.

Have you ever thought of controlling an army? You would sit down behind your desk, giving instructions to a computer so that it can perform whatever task you assign it.

Don’t skip the tasks at the end. You will boost your creativity and brainpower.

Level One: Welcome to Python

Every human wants to be a master. We all want someone or something that will always do our bidding. We might try to talk to pets and even the walls of the bathroom. You may try to talk to your computer.

At this level, your goal is to understand how the computer thinks. You are smart, but your computer is not. It is only as smart as the instructions you put in it. It is the only weapon you have in your hand. LANGUAGE.

Well, while all of us do this, most people may think that everything or everyone we talk to cannot actually understand us. That may be true if we speak a different language from what they already know. If you speak the language that they understand, there is nothing you cannot talk to.

You may have seen in movies people talking to computers, and the computer would most likely talk back. If you have a smartphone, you may know Apple's Siri and Android's Google, and many other interactive software that talks to users. You are smart, eh? Do you find those kinds of conversations unbelievable?

Look, to ask the computer for information is different from providing the computer with certain instructions that it would follow. That is what programming means. This book will teach you one of the languages that the computer understands so that you can tell it to do anything for you or your friends.

Before now, you have always communicated with your computer. For example, when you try to open a folder on your computer, you are giving the computer instructions. When you chat online, for instance, the computer communicates with you. It then communicates with other machines or people to address whatever requests you make.

When you code, you translate human ideas into a language that a machine can understand and use.

Everything around coding is built around the concept of garbage in, garbage out. You give the computer instruction, which is information or data. You expect it to use it for some output, which can be in the form of words, pictures, action, or some other result of processing the data we give it. Sounds interesting, isn’t it?

Let’s talk about how the computer can understand garbage in, garbage out or more professional terms, Input/Output (I/O)?

You press buttons on a controller or tap the screen in a mobile game, that’s input. You have input instruction for the computer. Then, your character jumps, ducks, or moves in whatever direction you intend, that’s output.

Have you ever tried cooking? You can call the ingredients you need to make the recipe input. You must follow the instructions and use the ingredients to get the output, the food!

Cooking Up Programs

One day, I asked some students to write a paper about making toast. I told them that I would use their papers to make a toast in front of the class. I took each person’s paper as input to make the entire class understand the importance of input to output.

When they finished writing, I brought in a toaster and plenty of loaves of bread. I read each paper and demonstrated it to the letter.

The results surprised all of them because none of their procedures worked as expected. It was funny. For example, one would have gotten it all right if he had written that I should remove the bread from the wrapper. It is funny that I dutifully stuffed the bread with its wrapper and everything in the toaster. Do you get it?

When you have an idea in your head, everything plays out fine to you. But imagine that you are going to tell it to a specialist in that field but a dumb person. You must not leave any step out when giving instructions.

Coding is, in many ways, similar to cooking. For example, when you cook, you can create a recipe to make bread from scratch.

In Python, too, you can create code from scratch, using only Python's basic built-in functions.

When cooking, you can use a ready-cooked recipe in another recipe. For example, you can use bread when cooking turkey stuffing uses bread as an ingredient.

The same is said in Python. You can insert a program you'd created for one particular task into another program the same way you add any ingredient to a recipe.

When you want to cook, you can buy premade food ingredients like bread.

Similarly, Python comes with modules (sets of programs that other people have written and made available for you to plug into your program) that you can use to make your program livelier.

Python's even better than that because most of those modules are free!

When you code, you are telling the computer to do something in its language. This book is a step-by-step instruction guide that will help you to understand how to write instructions that match the way a computer "thinks," making computer programming entertaining to future coders.

As I said earlier, computers are pretty stupid. Not as smart as you think. Computers take things literally. Well, people only think that computers are smart because they see them do a few things over and over in different combinations with speed.

Talking to Stupid

You are wearing the apron. Your hands cannot reach the tools, but you have a powerful but stupid assistant. He only does everything you tell him literally. That means, if you, by any means, say, “kick the bucket,” it would kick all the buckets in the stall.

If you want to make bread from scratch, you need to tell your assistant everything, starting with very basic steps.

You have warm water and sugar in a small bowl. You tell your assistance:

“Put three spoons of yeast.”

He says, “I can't find three spoons of yeast.”

“Go to the refrigerator over there. Open it. You will find the little package labeled ‘Yeast.’ Go get it”.

The apprentice gets the package of yeast and tells you, “I’ve got the yeast. Now what?”

“Put the yeast in the bowl.”

He went ahead and put the yeast in the bowl.

“Hey, stop! Open the yeast’s package first!” you yell, “use the spoon to take out the yeast three times.”

This is what you must do with your computer until it registers how to complete the task of baking bread by itself. Once you persevere up to that level, the computer can quickly bake you as much bread as 200 loaves every 2 minutes.

WHY PYTHON?

How many languages do you understand? Humans can understand many different languages. A computer can understand whatever ideas and concepts that we input to it, if it comes in a programming language. In this book, you will learn the Python programming language because Python is cool, and it can be used in many different ways, just like the English language. Many big and important companies and corporations like Google, Instagram, NASA, and Spotify use Python programming language.

Python is also very popular, and it runs on almost every computer in the world.

Applications in Another Light

You must understand certain things before you can create any applications or set of instructions for the computer. For instance, when I made my students write a paper I acted out literally, I am teaching them that the computer takes applications as a procedure.

When you write a program or an application to be run on the computer, the code is a procedure you would write in a well-defined series of steps or instructions on what you want the computer to perform word for word. If the computer performs or acts out the instructions you put out, the result must be consistent with whatever task you have in mind.

If, as you write the code, you leave out a step, the results will not come out as you intend.

The reason is that the computer is stupid. It does not know what you want, but it just acts based on instructions. And it takes everything literally. The only thing the computer knows is that you have given it a specific procedure, and it needs to perform that procedure to create the application.

Computers don’t actually speak any real language. You might know that they use binary codes. 0 and 1 are the computer alphabets that they use to flip internal switches and perform math calculations.

LEVEL ONE COMPLETE.

You have understood that your computer is dumb and takes things literally. You have to write codes as procedures that any dumb person can take literally and follow step by step, just like your dumb assistant at this level.

Open the next level to install Python on your computer.

Level Two: Getting Your Own Copy of PYTHON

At this level, we will equip you with the right tools to create your instructions for the computer.

Let’s get started!

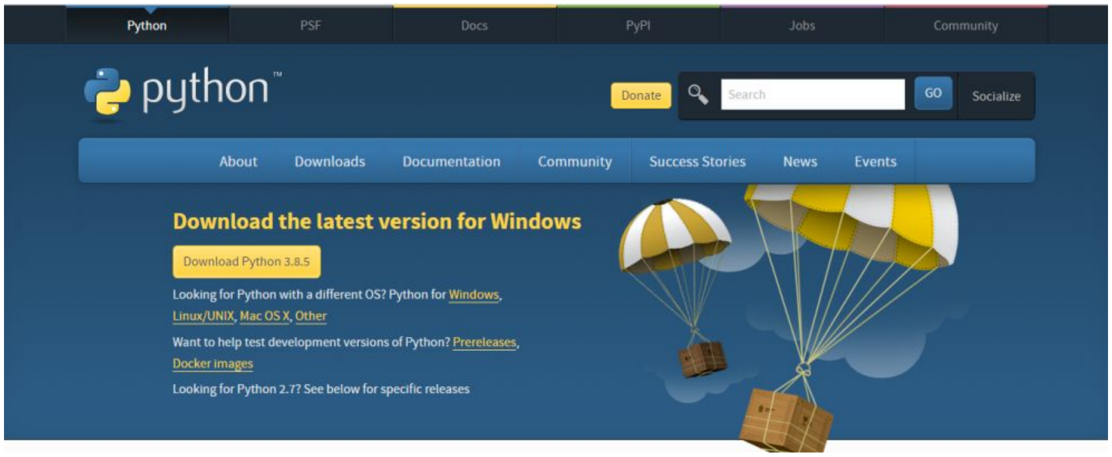
Before you can create applications, you need to get the computer to decode your instructions. Because the computer only uses math language, applications such as Python will help the computer to translate your ideas into understandable instructions by creating the Python and running the code you write.

Downloading the Version You Need

To get the right version for your platform, you need to go to the official Python website at https://www.python.org/downloads/. The download section is a big orange button. At the time of writing this book, the Python 3.8.5 is the latest version.

This book assumes that you are using a PC and that your Mac already has Python pre-installed. Nevertheless, if you have not, or if the version is obsolete, we will see together how you would use Python on your Mac computer.

You can experience the magic of coding when you enter the website using that link up there. Once you enter the Python website, the website will automatically know what type of computer (whether Mac or PC) you entered with. That big orange DOWNLOAD button will help you download the correct version of Python to install on your computer! Go ahead and click the DOWNLOAD button


Python PSF Docs PyPI Jobs Community

python™

Donate Search GO Socialize

About Downloads Documentation Community Success Stories News Events

Download the latest version for Windows

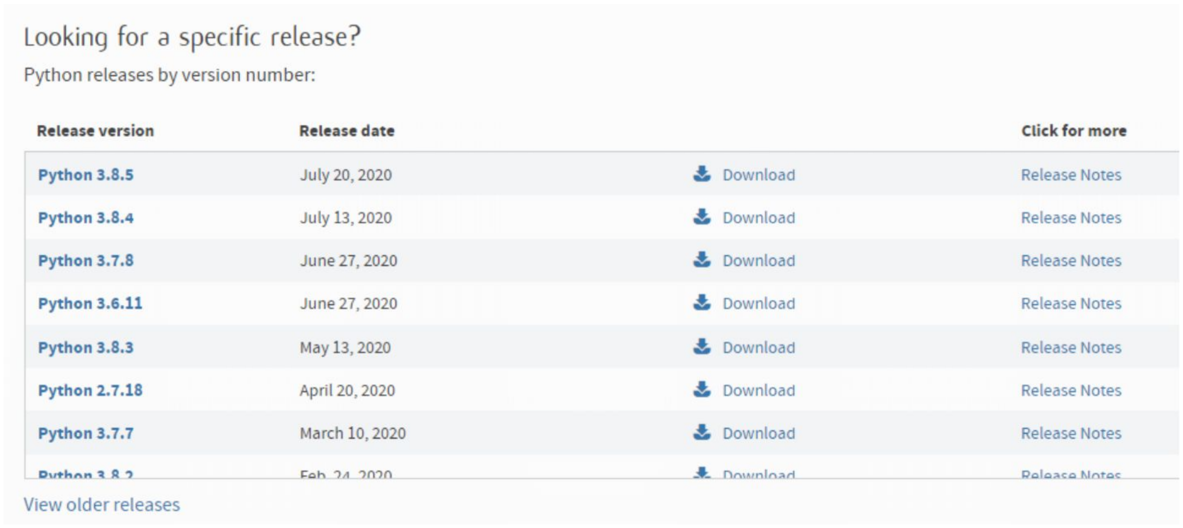
Download Python 3.8.5

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Prereleases, Docker images

Looking for Python 2.7? See below for specific releases

The platform-specific links on the lower part of the page show you alternative Python configurations that you may use when the need arises. Even though I doubt that you may need them for now, for example, you may use a more advanced editor than the one provided with the default Python package, and these alternative configurations can provide one such. You might also want to download earlier releases of the Python software that suits your PC or Mac.


Looking for a specific release?

Python releases by version number:

Release version	Release date		Click for more
Python 3.8.5	July 20, 2020	Download	Release Notes
Python 3.8.4	July 13, 2020	Download	Release Notes
Python 3.7.8	June 27, 2020	Download	Release Notes
Python 3.6.11	June 27, 2020	Download	Release Notes
Python 3.8.3	May 13, 2020	Download	Release Notes
Python 2.7.18	April 20, 2020	Download	Release Notes
Python 3.7.7	March 10, 2020	Download	Release Notes
Python 3.8.2	Feb. 24, 2020	Download	Release Notes

View older releases

The image above shows the other dialog box, where you can download older versions of Python if your PC or Mac does not support the latest version.

Working with PC

Once you have clicked the big download button, the software automatically downloads. You would have to install it by clicking RUN when your PC pops up the dialog.

You would install the Python software just like you would install any other software. Let me walk you through how to do it on your PC, whether you have 64 bit or 32. The process is the same.

Step 1. Find the downloaded file of Python on your PC.

It might come in different filenames such as python-3.8.5.msi for 32-bit systems or python-3.8.5.amd64.msi for 64-bit systems.

Step 2. Open the installation file.

Once you double click or run the installation file, a Security Warning dialog box may pop up asking whether you are sure about running the file. Click Run.

Step 3. Choose All Users and click Next.

You would need to provide the installation directory for Python. I always use the default destination. However, you can install Python anywhere you want on your system.

Step 4. Customize your installation to meet your needs.

Python would then need you to customize its installation.

When you enable the option to "Add python.exe to Path," you will save yourself time later. It allows you to access Python from the Command Prompt window.

Step 5. Click Next to start Installation.

Step 6. Click Finish.

You can now start coding on your Python app.

Working with the Mac

On your Mac computer, you can follow the same steps to install Python from https://www.python.org/downloads/: the official Python website's Downloads section. The process is very similar to Python's website, it will recognize your system as Mac and allow you to download the special software that suits your computer.

Follow the legal documentations and confirm each one that comes up in the installation process.

Click the INSTALL button. You might have to enter your username and password for your account on your computer. Mac OS sometimes asks for this whenever you want to install something. After you have clicked through these steps, you should begin installation into your system.

First, the download may begin with a Python disk image. This download may take time. When it completely downloads, your Mac will open the disk image automatically.

The disk image would look like a system folder. When you click to open the folder, you will find several files, including python.mpkg.

The python.mpkg contains your Python application. The text files are information about the build, licensing, and any other late-breaking notes.

When you double-click Python.mpkg, you would see a Welcome dialog box that tells you about the Python build. Click continue until you no longer can—usually three times.

Then you would select whether to use your hard drive or other media to install Python and click Continue.

Lastly, click Install to complete the installation of Python on your Mac computer.

As you want to install, the installer may need you to input your administrator password. Input the necessary info and click OK.

Python is ready to use once you have seen the Successful! dialog box that shows that your software is ready. You can now close the Python.mpkg disk image and remove it from your system.

Testing Your Installation

To guarantee that your new Python application is usable, we would test it by writing your first Python app.

USING IDLE

As you download and install Python, another supplemental application will be installed too. That one is called IDLE. IDLE means Integrated Development and Learning Environment. We can also call it an integrated development environment or IDE. The IDLE helps us to write Python programs.

You can think of IDLE as your notepad. An electronic notepad that comes with additional tools for you to write, debug, and run whatever Python code you want to create. You cannot work in Python without opening IDLE. Python files cannot be opened directly otherwise!

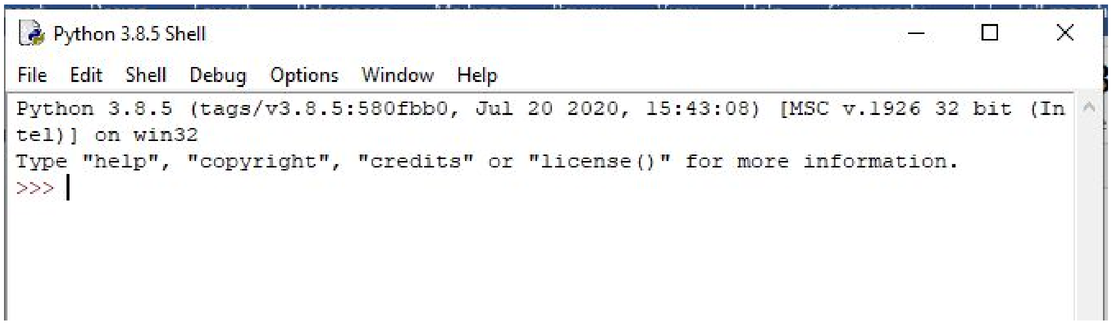
Let’s go in for a drive!

You are the driver on this bus. Let’s run you through how you can install and open your copy of IDLE.

ON A PC

Remember that IDLE comes with the Python app you installed. Hence, the file is already saved in your system. You only need to run it now. You can search IDLE from your Windows Start menu.

Then select the search result IDLE (Python 3.7 64-bit) or IDLE (Python 3.7 32-bit) whichever you see depending on your machine. IDLE automatically displays the Python version and host information when you open it. It looks like the image below:

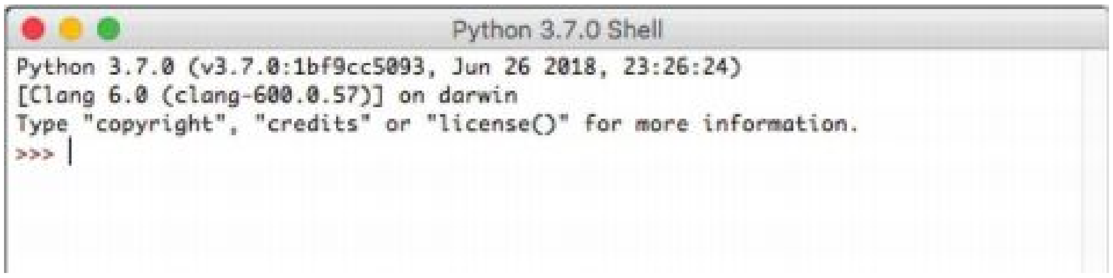


```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
```

ON A MAC

You also already have the file on your system. Hence, you would get to it by navigating to GO > APPLICATIONS. Under Applications, find the Python 3.7 (or whatever version you have downloaded) folder and open it.

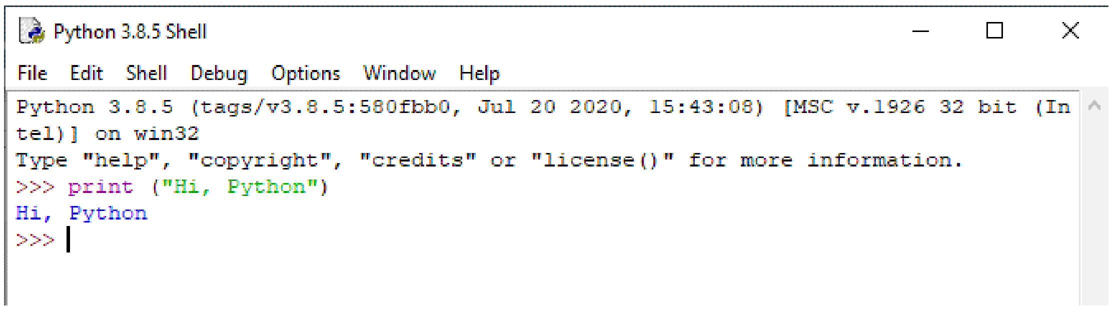
You can find the IDLE app and open it. You will get the pop-up dialog as in the image below:



```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> |
```

Now you've successfully installed Python and IDLE on your computer. You can now write your application.

Open up IDLE on your computer. IDLE opens shell first whenever you open it on your computer. The shell is the window where you can write Python codes and see the results in real-time. The shell that opens looks like the following image:



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print ("Hi, Python")
Hi, Python
>>> |
```

If you look at this shell, you would see that there is a code saying Hi to Python. If you want to do that same code, you can start by typing the following:

Print("Hi Python!")

When you hit the ENTER key, it comes out like that shell up there.

Let's see another code: type print("This is my first Python program.") and press Enter.

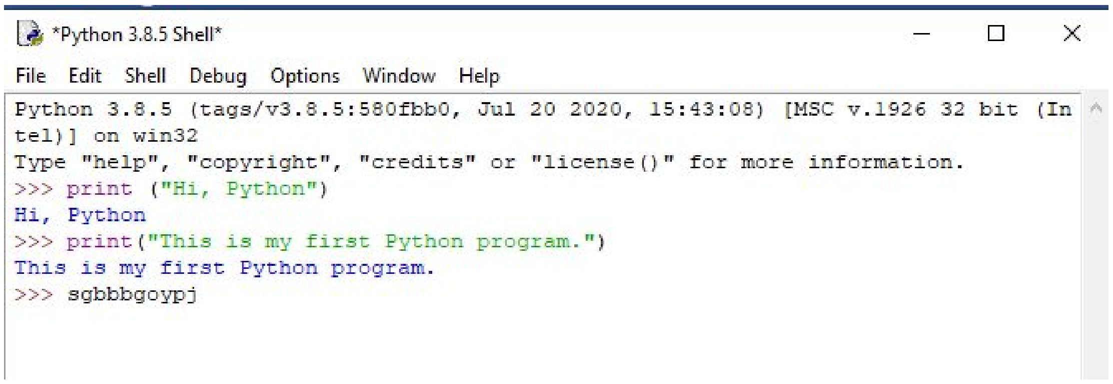
Great job! You have successfully coded your first line of Python code! If I were with you, I would give you a pat on the back, or high-five, but you could do that for me, right? Right. You're about to learn some awesome things.

You would see that Python displays the message you just typed. The print() input you typed will still show whatever you tell it to display on the screen.

Throughout this book, we will use the print() command to display the results of whatever task you want Python to perform.

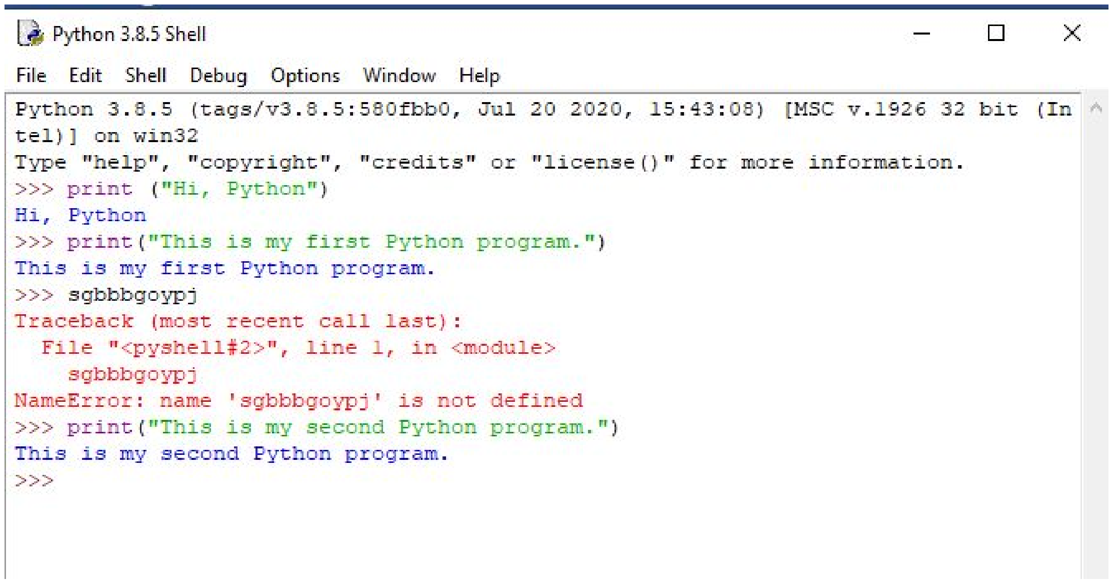
IDLE uses color codes to differentiate the various entries you input so that you can see and understand them easily. When you get an unfamiliar color, that means you are doing something wrong.

For instance, when you type a text, it shows **purple**. That indicates that you are typing a command. **Green** shows the content that you want the command to show or display. **Blue** shows the output from the command you typed in green beneath the command. Whenever a text comes out **black**, it means your command is incorrect or unreadable.



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print ("Hi, Python")
Hi, Python
>>> print("This is my first Python program.")
This is my first Python program.
>>> sgbbbgoypj
```

You will see from this image that the colors are there, but the one in black is the text that Python cannot read.



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print ("Hi, Python")
Hi, Python
>>> print("This is my first Python program.")
This is my first Python program.
>>> sgbbbgoypj
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    sgbbbgoypj
NameError: name 'sgbbbgoypj' is not defined
>>> print("This is my second Python program.")
This is my second Python program.
>>>
```

As you can see in this image, the text cannot be read, and so Python flags it as an error.

Whenever you have finished, you can either save your work or simply kill it. Let’s learn to quit our session first.

Whenever you want to end your IDLE session, type quit() and press Enter. When you click OK, the session dies.

One thing you would have noticed about commands is that they have brackets just after them. All Python commands have parentheses like these two.

That’s how you know they’re commands. However, you don’t need to tell the quit() command anything, so you simply leave the area between the parentheses blank.

SAVING YOUR WORK

If you quit or kill your Python session before you save, you would have to write the code again if you ever need it. If you have written a longer program than the one we just did, you would have to save your progress, so you don’t have to re-type all the code you write.

Even though we have only written a short program, let’s save it to its own file so that you can see how easy it is to save your work.

To begin with, we should make another program:

On the MENU bar in your shell, click the FILE tab to open its setting menu, which is a rundown of activities that you can perform on your shell.

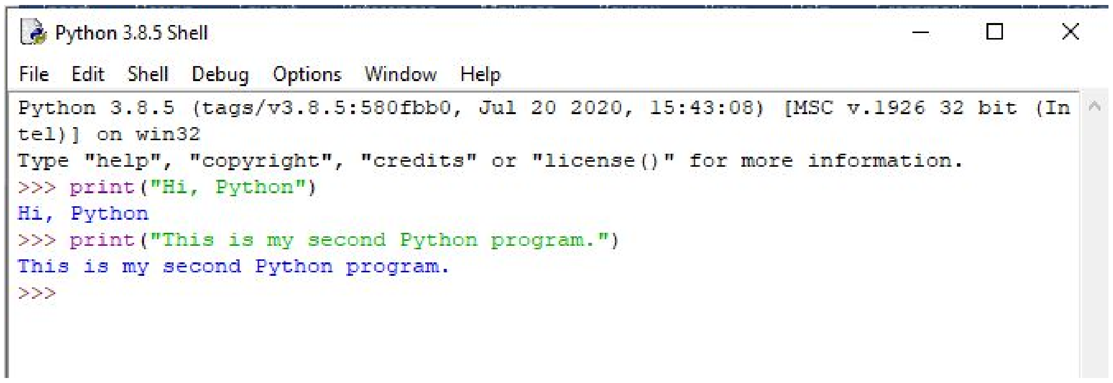
Amazing!

So, let us create a new code. You can write any program using the print() command. You can recreate the code we did along with the “Hi, Python” print. Do you remember?

Simply type the following (or whatever you like to display.)

print("Hi Python!")

Hit the ENTER key. Type print(“This is my second Python program.”) and press Enter.



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hi, Python")
Hi, Python
>>> print("This is my second Python program.")
This is my second Python program.
>>>
```

Let us save our new code now. Here is how to save your Python program. Follow these steps:

On the MENU bar of your shell, click on the FILE tab to open its setting menu. And then you would click on Save. Or you could simply use the shortcut: Ctrl+S.

A window will pop up, requesting that you name your new document. By all means, give your new program any name you wish, just like you name regular files on your computer.

Try to save your new Python program in a folder or place in your system that you cannot forget!

Generally, new Python program documents will be saved in the same folder as the Python file. It is okay to change the "Save" set up to a more convenient folder.

Level complete.

Level Three: Let's Get Inside PYTHON

The basic reason why you are here is that you want to create applications.

You should know that whatever application you create, it has to interact with the computer and the data contained in it.

I am talking about data because every application out there works with data. Programmers use the acronym CRUD to tell what most applications do:

It means that applications can:

Create data

Read data

Update data and

Delete data.

Helpful Hacks

You are a programmer now. You have just written two programs of your own.

If you want to go far in programming, you must think and work like a professional. We have learned two of the professional programmers' tricks. Here is when you need to have more up your sleeve.

The basic thing you should know is that a programmer always finds the easy way to do expert things. For example, since we save files a lot, we have shortcuts to take there. You have met the Programmer's friend, Ctrl+S.

There are many other shortcuts we have created. Use this list of very helpful keyboard shortcuts while coding.

CTRL key + S key: Use this key combination to save a file. You can use these two keys to save your progress while you are working in your shell or to save a whole new file.

CTRL key + N key: This key combination will create a new file for you to start from scratch.

CTRL key + C key: You may have known the Ctrl+C keys. Yes, that is our copycat friend. You can use them in coding to copy any text you have highlighted. You can use your mouse to highlight some code in your shell. To highlight text, if you don’t know, you have to place your cursor before the beginning of the text you want to copy, click, hold the main mouse button, and drag your mouse to the end of the text you want to copy. When you release the mouse button, your text will be selected. After that, Ctrl+C! Your text is copied for pasting!

CTRL key + V key: I call this shortcut the brother of the CTRL+C shortcut. This one is used to paste whatever you copy. Using this shortcut will place the text you have copied to the point where the cursor is or wherever else you choose.

CTRL key + O key: This shortcut will open a file. It will tell your computer to open a document. Your computer will open the Python folder, and you can navigate to where you saved your file. We will use this shortcut soon.

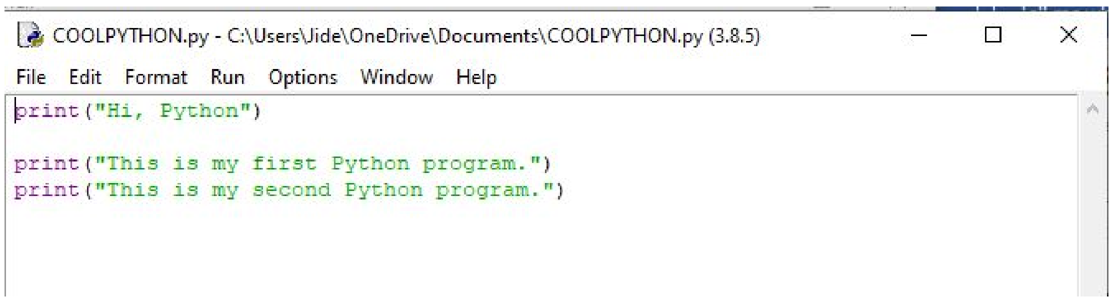
CTRL key + Z key: Everyone makes mistakes. Errors in coding happen to the best of us. We all need a way to turn the hands of time and undo something bad we have done. I sometimes wish we could have this awesome command in real life! This CTRL+Z shortcut performs an undo action. If you need to go back with a step, or to return some code that you have accidentally deleted, this shortcut can help! Press the CTRL key and Z key together, and watch whatever the last action was, undo itself. You can keep pressing this shortcut multiple times if you want to keep going back further and reversing more actions you have performed. The thing is, this shortcut can only do so much—it will only reverse actions up to the point that the computer has stored in its memory.

So, what are we waiting for? Let's run a program and see your code in action. Well, the best part of a programmer's life is seeing their code in action.

After professionals write some code, they save it. Then, they must check how well their code works. Here's how to run your code.

On the MENU bar in your shell, click the FILE tab to open the context menu. And then click Open. You could use the shortcut, Ctrl+O. Once you have picked the folder that you want to open, find your program and select it, then click on the Open button.

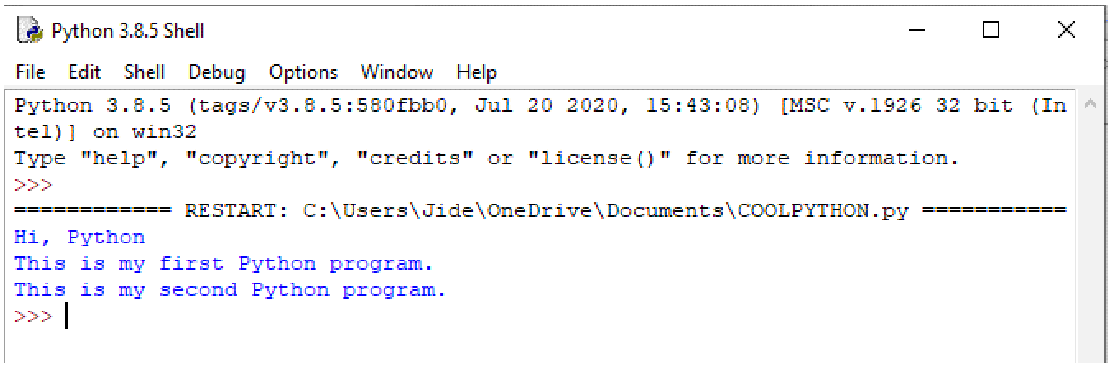
Your program will open in its own entirely new window like this image below:


COOLPYTHON.py - C:\Users\Jide\OneDrive\Documents\COOLPYTHON.py (3.8.5)
File Edit Format Run Options Window Help

```
print("Hi, Python")

print("This is my first Python program.")
print("This is my second Python program.")
```

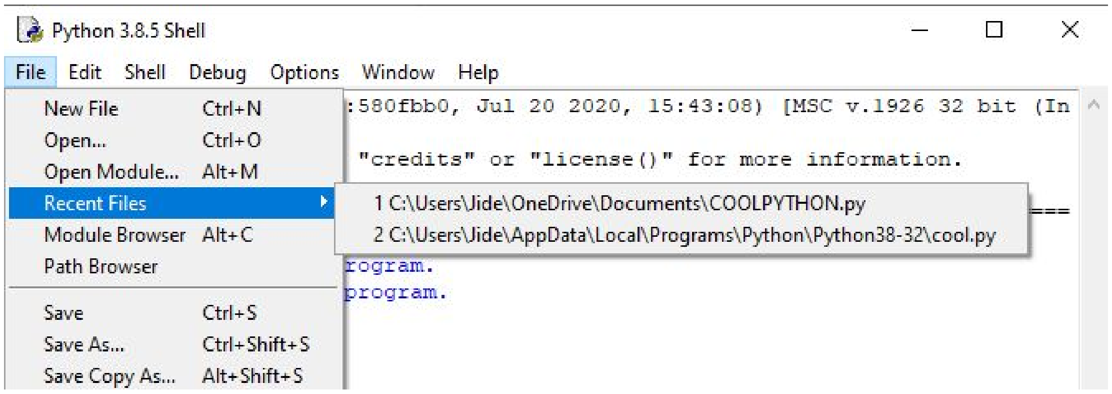
Press the F5 key. That's it! That is another **shortcut**. It means EXECUTE. Once you open a program, the F5 key should execute your code and make the computer carry out the task you asked it to do in code.

In our case, we have asked the computer to print something, and it did! You should see your writing in the shell like this:



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============ RESTART: C:\Users\Jide\OneDrive\Documents\COOLPYTHON.py ===========
Hi, Python
This is my first Python program.
This is my second Python program.
>>> |
```

Here is another guy I must introduce you to. His name is Recent Files! This tab will list all the codes you have saved or worked on in your shell.


Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
New File Ctrl+N
Open... Ctrl+O
Open Module... Alt+M
Recent Files
Module Browser Alt+C
Path Browser
Save Ctrl+S
Save As... Ctrl+Shift+S
Save Copy As... Alt+Shift+S
1 C:\Users\Jide\OneDrive\Documents\COOLPYTHON.py
2 C:\Users\Jide\AppData\Local\Programs\Python\Python38-32\cool.py

```
:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
 "credits" or "license()" for more information.
===
rogram.
program.
```

Commander, Do You Copy?!

In the programming world, Python uses commands from the coder. In this case, you are the commander.

Do you still remember some of the commands we have learned to use?

The print() command. Programmers use this command everywhere. And you, yes! You've already used this command in the previous level!

Remember the **print("Hi Python!")** command?

See, kid, we use the print() command when we want to output a **string**.

What's A String?

A string is a collection of characters. Characters can be letters or numbers or whatever we call a text.

A string in programming is a **type**. A string is a data type used to represent a text. It is a set of characters that can contain spaces and numbers. For example, the word "hamburger" and the phrase "I ate 3 hamburgers" are both strings. Even "12345" could be considered as a string, if specified correctly.

Typically, programmers have to enclose strings in quotation marks for Python to recognize the data. There are other data types used in coding, like integers, Booleans, and lists. Hey, you don’t need all those yet! I will introduce them later guys.

Parameters

A function parameter in Python allows a programmer who is using that function to define variables only within that function. When a function is defined or written, it may have parameters, which are the information (input) you give the function to work with.

The print() function takes a few parameters. For now, let’s stick with only one parameter: the part you put inside the double-quotes. That parameter tells the print() function to take whatever you put in the double quotes and print it out to the console window.

The console window helps us to see the result of the code we write on the go. That is, if you write a basic math operation, the print() function can show your resulting answer in the console.

Debugging

Debugging is a fancy name that programmers give the adventure quest we take to fish out any issues or mistakes (called bugs) in our codes that may cause the program not to work the way we intend. Debugging is the process of detecting and removing existing and potential errors called ‘bugs’ in a code that can cause the program to behave unexpectedly or crash. We use debugging to prevent incorrect operation of a software or system by finding and resolving bugs or defects.

When you are coding, you can use the print() function for debugging. If you want to debug your program, you can print out parts of your codes to double-check that the code will come out as you expect it.

You will need debugging a lot because you will start working with variables and decision-making blocks of code.

Comments

A comment is an expression of a programmer's ideas. In Python, comments are statements that describe the meaning of a block of code to another programmer. Because you cannot possibly remember the names of every variable when you have a hundred-page program or so, you would need comments for you or someone else to read as well as modify your code.

Comments do not get translated by the computer. You use them within your code as messages or markers for yourself or someone else. You may also use them as parts of code you want the computer to ignore.

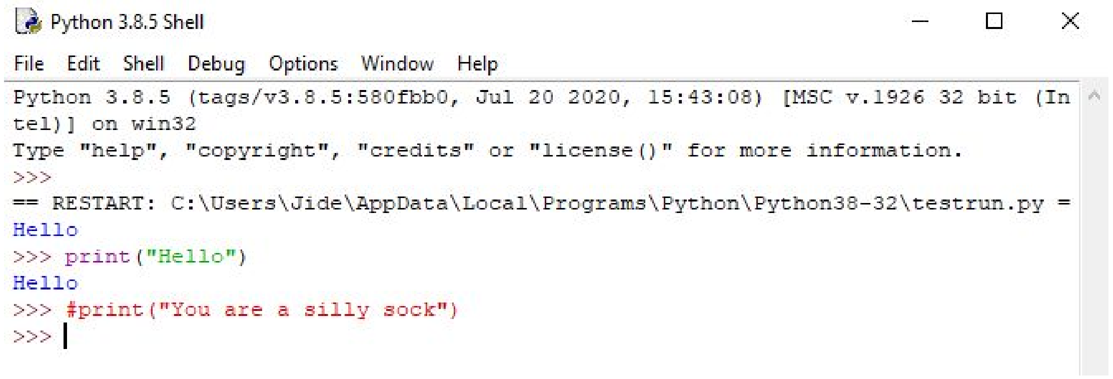
Comments are created by putting a hash (#) before the code line. Comments become a red color to show you they are comments or not executable.

# print("Do not print this!")

So, if there's a part of your code that you think is causing you problems and you don't want to delete it, you can test it by commenting it out:

print("Hello")

print("You are a silly sock!")



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:\Users\Jide\AppData\Local\Programs\Python\Python38-32\testrun.py =
Hello
>>> print("Hello")
Hello
>>> #print("You are a silly sock")
>>> 
```

In the code above, you see that Python printed the first line ("Hello"), but not the second ("You are a silly sock")—because the hash tells the computer, "Hey, bud, don't print this line!"

Isn’t that cool?

Trouble Makers in Python Coding

For the most part, the print() function can write everything you want to display in the console window. Nevertheless, some situations and special characters will not work well with the print() function. They are just like electronic troublemakers. Who are these troublemakers?

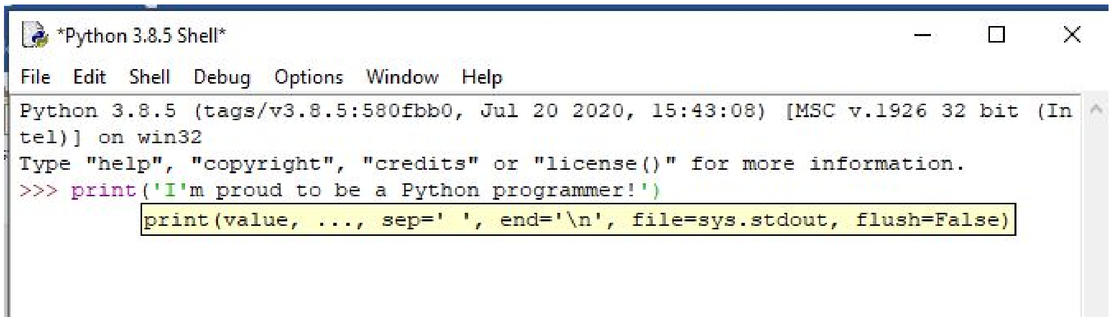
Quotes and Apostrophes

These guys make so much trouble that they can frustrate any newbie programmer when it comes to Python programming.

Let’s say you want to write a code to print the following sentence: “I’m proud to be a Python programmer!”

Just write the code like this:

print('I’m proud to be a Python programmer!')

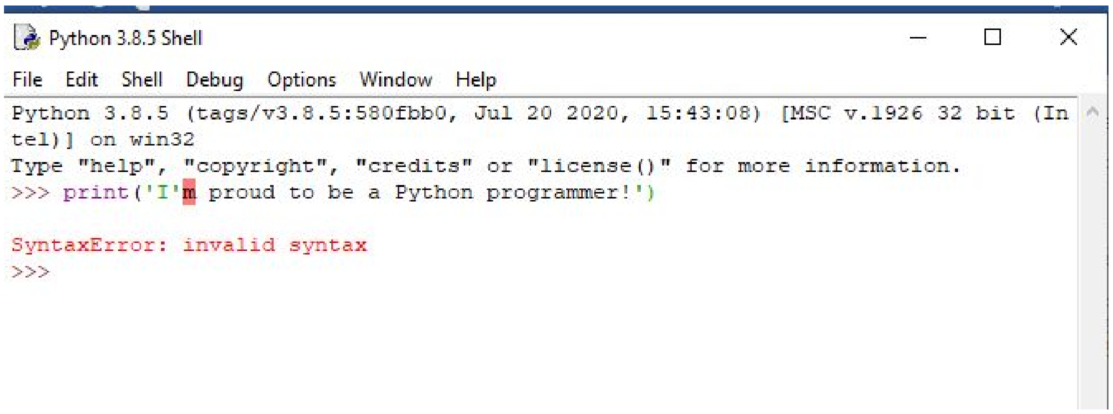


```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('I'm proud to be a Python programmer!')
         print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

When you clicked ENTER, what happened? Did you see the sentence display in your console window?

No?

Haha! Congrats! You got your first syntax error.



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('I'm proud to be a Python programmer!')

SyntaxError: invalid syntax
>>>
```

Here's what happened: Whenever you use the print() function, this is what you tell the computer, "Hey, bud, I need you to write something out to the console window."

The computer says, "Sure! What do you need me to write?"

The computer checks the print() function you gave him, and looks for a starting and ending quotation marks.

To the computer, these marks are "flags" or markers for the beginning and end of the information you are pointing out to. That is why once the computer finds the first and second quotes in your string, it thinks it's done. Remember that your computer is not smart. It is just like that dumb baker assistant we worked with in Level 1.

So, the computer doesn't expect any other characters after the second quote it finds. When you put in another (second, which should be closing) quotation mark to string, the computer sends you a message in the form of a syntax error.

Look closely at the sentence. Did you see where the problem is?

The problem is right at the beginning of the sentence!

Look at the first quote. The computer finds the first quotation mark as the beginning of the statement. The next quotation mark is a single quotation mark or apostrophe within the word "I'm."

At this point, the computer says, "Oh dear, what should I do now? A second quote in the string? This should be the end. I don't know if this extra stuff after it, though. I guess I have to tell the Human that I don't understand what they are trying to make me say."

Then the computer tells you: SyntaxError: invalid syntax

So, how do we fix this?

At least, we want the full sentence. It would not make any sense without the "I'm" still, the dumb computer will mistakenly think our ending is the apostrophe in the word "I'm."

One way to solve this issue is to use double quotes for the entire sentence.

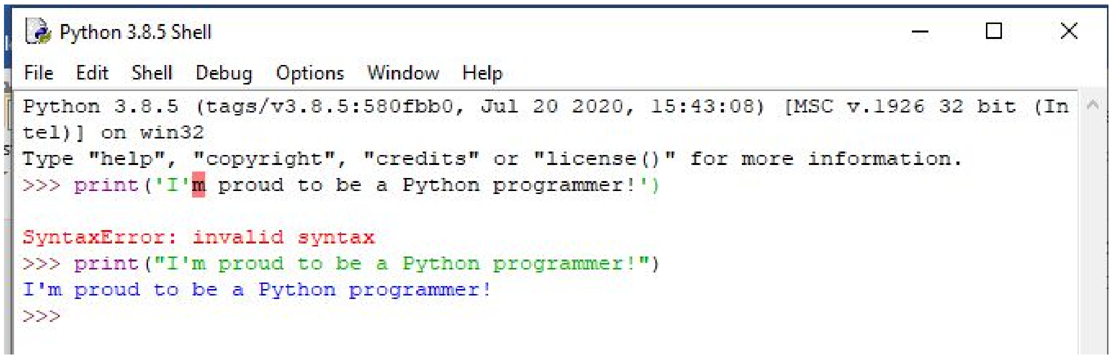
For example:

print("I'm proud to be a Python programmer!")

This one solution works well because the computer sees that the first quote is not the same as the quote after "I."

As it continues looking through the string, it will search out the second matching quote, which must also be a double quote.

Note: In Python, both single and double quotes work when writing strings, but you should be consistent. This double-quote is usually the best way to go when using strings!



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('I'm proud to be a Python programmer!')

SyntaxError: invalid syntax
>>> print("I'm proud to be a Python programmer!")
I'm proud to be a Python programmer!
>>>
```

Solution Method 2

Another solution to this problem is to use **escape characters.**

There are special characters in code called **escape characters.** These special characters are like **Get Out of Jail Free** cards in Monopoly board games.

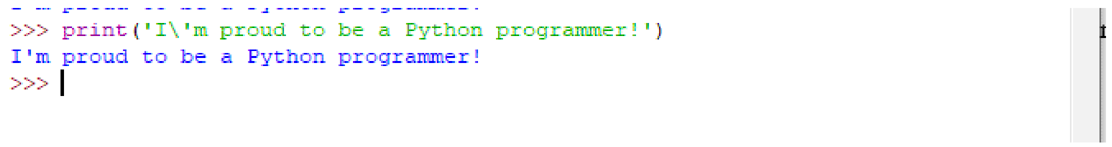
They give the computer a heads-up when the input we are about to give is tricky.

The backslash (\) character is the Python language escape character.

To use this escape character, we simply type a backslash before the tricky character. An escape character gives us an escape route out of the problem!

Let us try and use the escape characters to fix our sentence:

print('I\'m proud to be a Python programmer!')



```
>>> print('I\'m proud to be a Python programmer!')
I'm proud to be a Python programmer!
>>> 
```

Now, look at how I fixed that code. Did it work with you? Yay!

This time, when the computer looks through the statement, it finds the backlash and skips the following character, which is the apostrophe in the word “I’m” because we told it to.

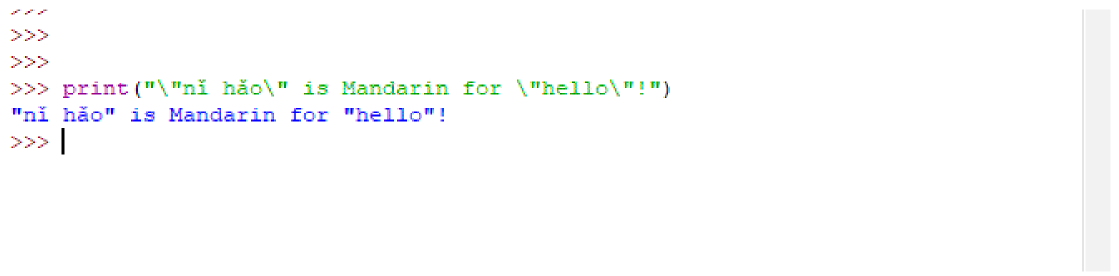
The computer sees the escape character and says, “Oh, that’s nice. The Human was cool enough to put a cover over that. I’d just skip it and go over. I’ll keep looking!”

Escape characters are also really helpful if you need to print more than one troublemaker, like quotes all on one line!

For example, if I wanted to print this statement, "n ĭ h ă o" is Mandarin for "hello," it might come out as an error unless you use the escape character. Try printing this to your console window:

print("\"n ĭ h ă o\" is Mandarin for \"hello\"!")

Do you see how the computer prints the words “n ĭ h ă o” and “Hello” with double quotes? Great! That’s just what I wanted.



```
>>>
>>>
>>> print("\"nǐ hǎo\" is Mandarin for \"hello\"!")
"nǐ hǎo" is Mandarin for "hello"!
>>> |
```

Without the escape characters, the computer will get confused with the multiple use of the same type of quotation marks in the same line of code.

Multiple Lines

When working with the print() function, many newbies have a hard time with multiple lines. How can we print the following sentence, exactly as it appears?

Here is a

short poem

for my Python

program in lines.

Well, you would not have to fret. There is a special escape character that tells the computer to create a new line out of the line of code.

Programmers usually call new lines a line break or line feed. In Python, the escape characters for this action is a combination of the backslash (\) and a lowercase letter “n.”

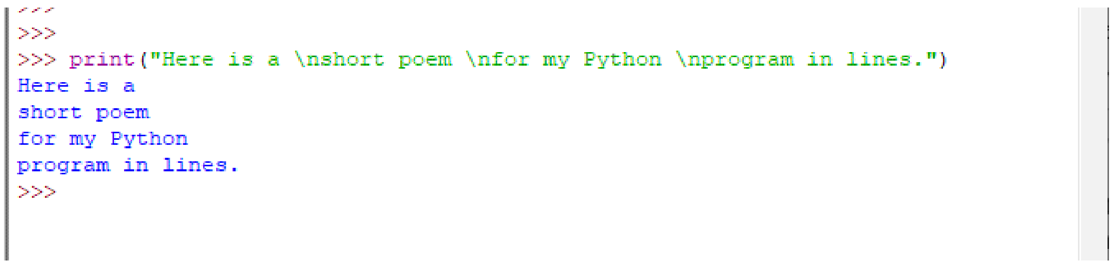
It should look like this: **\n**.

Use what you know about escape characters, try printing that sentence.

If you wrote something like this,

print("Here is a \nshort poem \nfor my Python \nprogram in lines.")

You are correct!



```
>>>
>>> print("Here is a \nshort poem \nfor my Python \nprogram in lines.")
Here is a
short poem
for my Python
program in lines.
>>>
```

Everything between your starting and ending quotes will be printed exactly as the computer understands it. This includes those line spaces!

Just the same way the \ escape character told the computer to ignore the apostrophe in our first example, the \n escape character tells it, “Hey, bud, start writing everything after this \n flag as a new line.”

And the computer will do exactly and just that every time it sees the \n escape character.

Variables

Now, it is a great time to walk through **variables**. That is because they are an important part of coding that every programmer uses all the time! You should get comfortable with them before we move on to the other parts.

A variable is a tag or some way to keep track of information. We use tag/variables every day:

- People wear name tags, so we know who they are.
- There are tags on the food we buy like nutrition labels that tell us information, like how many calories the food has, the grams of sugar it contains, or the list of ingredients used.
- Cloth tags tell us the size, designer, price, and sometimes even the identity of the person who inspected it.

Tags tell us about something. So do variables when it comes to coding.

Think of a variable as a name or a tag attached to a particular object. A variable is a label or a name given to a certain location in the computer memory. That location is where the value you want your program to remember, to retain for later use, will lie.

What's amazing in Python’s variables is that it does not have to follow any specific format before it works. It can be of any type. It can be a string, integer, float, etc.

When we write codes, variables help us to hold pieces of information. Like clothing tags and food labels, coding variables can hold many kinds of information, such as strings, numbers, lists, and more. This means that you tell the computer to do a task once it sees a variable.

So how do you create a variable?

I am thinking about making a variable to keep information about this book. Why not let us start with the name of this book’s author (hey, that’s me!). If we did that, the variable would look like this:

author = "Alice"

And that’s it! You have created a variable.

The variable “author” is now a tag for the string “Alice.”

Here’s what will happen:

If you want to create a variable, you will give it a name, like “author.” This name we give it will help us remember the nature of the information contained in the variable.

Next, we add an equal sign (=). The sign tells the computer that the variable named “author” has the information it should keep. The process of giving the variable something as information is called assignment because we are assigning info to a variable.

Finally, we will input the information that we want our variable to keep track of. In our example, it’s the name of the author, “Alice.”

How about adding your name to the game? Let us create another variable and call it “reader.”

By all means, assign your name to this variable.

For this example, I will just assume that your name is Adam. Then, on the next line, use the print() function to write your variable to the console. Your final code should look something like this:

```
reader = "Adam"

print(reader)
```

Now press ENTER. What happened? Do you see your name displayed in the console window? Sweet!

Now, here’s the cool part about variables: every part of your program where you use the tag Author, the computer will display Alice, and wherever you input reader, it shows Adam (or whatever name you want on it). If you want to change it, for instance, you share this book with your friend, Sharon. You would have to change the variable, so that it assigns the variable to your friend’s name!

So, go and change the reader’s variable, so that it is assigned to your friend’s name, but change nothing else. With the change, your code should look like this:

```
reader = "Sharon"

print(reader)
```

When you press ENTER, your friend’s name should be printed out this time instead of yours.

You see, when the computer sees a variable, it says, “Ooh, this Human wants me to remember this piece of data. I would better store it in my memory. They would ask me later. I would also mark where I am storing this data, so I can quickly get it if Human asks me about it.”

The computer is great at keeping things organized. It keeps everything in the CPU. It’s like a big, grid-like bookshelf with many different cubbies to place things in. That is how the computer marks the location of any data it stores, so that it can quickly remember where to get it if we need it again.

Variables can hold any kind of data type from text to numbers. What if you wanted to create a variable to hold a number, how would you do it?

```
my_number = 6
```

You see that it is still similar to the way we created the other variable:

- First, give your variable a name. In this case, my_number.
- Then we assign it to the information we want it to hold. In this case, it's number 6.

You are a smart observer. You have noticed that we didn't use quotes around our number this time. Can you guess why?

Integers

Remember what we said about **strings**. String types tell the computer that we are giving it a text input. We start and end a string type with quotation marks. Integer types, on the other hand, tell the computer that the information is a whole number. In Python, whole numbers are called integers.

We just type integers out as a plain number. The computer understands math, remember? Hence, no quotes around integers.

The computer only knows a string to start with quotation marks. If you use quotations when inputting integers, you are going to confuse the computer into thinking you are using a string!

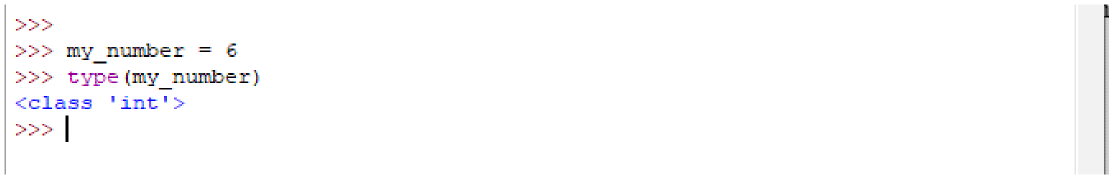
To see what I mean, I will introduce you to a new command. It is called type(). This code will tell us the type of data input we give the computer.

For instance, if you type the following code in your shell:

my_number = 6

type(my_number)

and press ENTER, it should tell you the type.



```
>>>
>>> my_number = 6
>>> type(my_number)
<class 'int'>
>>> 
```

What type did the computer tell you my_number is? If it says <class 'int'> as in the image above, you have seen the work of the type() function! Int is the computer's cool way of saying integer.

Now, let’s see if the computer will recognize an integer when you store your number within quotes:

```
my_number = "3"

type(my_number)
```

What type is it now? It thinks it is a string! That would be disastrous if we need to perform a math operation with the integer. The computer will not be able to find the number.

That is why we can’t use quotes with integers (or any of the other data types that involve numbers).

Good Things to Know About Variables

Because programmers use variables often in coding, you must know everything about it. Plus, you should also start with some good practices whenever you want to create variables for your programs:

Don’t Start Your Variables with a Number

There’s no telling that the computer is only as smart as the input you give it. This means, when you want to name a variable, be as descriptive as possible, but remember to be considerate of the computer.

One of **the rules of Python** is that the names of a variable cannot start with a number.

If you try it, you will get a syntax error.

Python doesn’t want numbers in variable names!

Well, let me tell you why. It started when the computer starts translating your command, it immediately sees the number and lazily assumes that the rest of the code is automatically a number. But there is more to the variable name than just the number that starts it. It gets confused. It gets really confused!

Ensure that Your Variables Have the Same Styling

Different programmers write variables differently. There are many ways to write your variables. It is just that you must learn consistency if you want your codes to work as you intend. Pick one way and stick to it.

You are smart, right? Do you see my own style of writing my variables? I write my variables in all lowercase letters. And even when I need to name my variable more than one word, I separate the words with an underscore, so it is easier to read them.

Look at the other ways many programmers use to write variables:

- **camelCase**: this style means that the first word of the variable name will be in all lowercase, but every other word after it starts with uppercase.

  Here is an example: myHouseNumber

- **PascalCase**: in this case, every word in a variable name starts with a capital letter.

  Here is an example: MyHouseNumber

One thing you should know is that there is no one best method or best way to name your variables. The important thing is to choose whatever feels right to you and stick to it.

Why is it important to stick to one style?

The problem is that the computer doesn't recognize variable's names unless it feels a connection as you type them. What I am saying is that if you have been using my_number as your style and you suddenly write MyNumber instead, you will get an error message, because the computer will see this code as two different variables!

Another thing that Python does not understand is space. So, when you write variable names, avoid space. You can connect words or use underscores (_) between the words. If you use spaces, you'll get an error message!

Variables Should Have Meaning

Lastly, variable's names should make sense. This means that when you or anyone else reads your code, you should know what your variable is and what data is in it.

Look at this list of good variable names:

- mood = "excited"
- age = 13
- my_favorite_color = "red"
- house_number = 1221

Look at this list of error-calling variable names:

- x = 6
- num_pen = 15
- curDay = "Wednesday"
- fAvOrItE_DeSseRt = "pudding"

Did you notice the difference?

The names that have meaning and a consistent style are clear and easy for both humans and computers to read and execute.

Fancy Printing

Hey, you are a good learner. You have learned how to use variables. Now let us learn how to use variables to create some fancier things with the print() command. Let’s walk through some of the fancy stuff now.

Formatted String Literals

Might seem like a tough guy, but the “formatted string literals” is simple guy. Let me introduce you to it.

You see, if you want strings to be even more useful, you can change certain parts of them or move certain parts around. You should remember how most of the strings we printed in earlier sections were full phrases or sentences. Well, we did not make any changes to the lines back there.

What if we did need to change some parts of our code? Let’s look back at our example:

```
print("I'm proud to be a Python programmer!")
```

Imagine that, instead of simply being proud to be a Python programmer, you’re ecstatic! Or overjoyed! Or delighted!

How can you change your sentence to show the word that describes how you feel about being a Python programmer?

You will use formatted string literals!

Formatted strings laterals are also called f-strings, and for the rest of this level, that is what we will call it. Deal? Deal!

We can use the f-strings to produce formatted strings, which are like normal strings but have a specific pattern. F-strings help us to replace any part of a string easily or change the order of any string.

To make it work, we must bring our escape character back. This time, the letter f and some special characters known as braces { } are the escape characters we will use for the replacing or reordering.

We could use an example:

Before the f-string can work, there must be a variable.

Let us make a variable and assign it.

snack = "burgers"

After writing the variable, we will write our actual f-string:

f"I give out {snack}"

Do that in your shell.

Did you see the magic? It is wonderful, right?

Here’s what happened:

As you have input the small letter f before the string, the computer smiled. It knows that you are trying to create an f-string.

So, while the computer looks for ending quotes in the string, it comes across some braces ({}), it goes, “Oh, here’s what the Human wants me to replace. What does it say? ‘snack’? Oh! That is a variable they asked me to keep. I know! Let me get it real quick . . . Got it! It’s actually a tag for ‘burgers.’

“Now, I would just put the word ‘burgers’ in there and remove this f-string placeholder. Nice!”

Then when it replaces all the parts of the string that you have asked it to, it displays the final version to your console window.

Super cool!

So now, let us go to our earlier question about how to change “proud” to “ecstatic.”

How do we use f-strings to change our print() command? I’ll break it down:

Since the adjective (“proud”) is the only part that will change, let us store it in a variable.

```
feeling = "proud"
```

For now, we created a feeling variable and assigned it to “proud,” since that’s how you currently feel!

Next, we know that our sentence will mostly stay the same, except for the adjective we are using to describe how we feel about coding in Python. So, let’s change the parameter in our print() function to be an f-string instead:

```
print(f"I'm so {feeling} to be a Python programmer!")
```

Great! Now our print() function will always print out the current feeling (happy) we have about being a Python programmer!

You should save this code in its separate file. Remember to name the file something cool you will remember easily. Now, open it in its own window and change your feeling variable to a different adjective, like ecstatic and save the code again.

Now press F5 to run your code.

Does your new sentence come with your new adjective? Sweet! The f-string will become extremely helpful when we need to start replacing more parts of our strings. Keep that one in mind, yeah? Let’s learn more fancy!

Easier Multi-Line Strings

Do you still remember how we printed multi-line poetry strings earlier in the Level? You remember how we used the **\n** escape characters, and how funny the code looked when we wrote it. It was also kind of hard to read, like this:

```
print("Here is a \nshort poem \nfor my Python \nprogram in lines.")
```

You can use the f-strings to make your code much cleaner and easier to read. Now, let’s rewrite our multi-line sentence like this:

```
multiline_sentence = """

Here is a

short poem

for my Python

program in lines.

"""

print(f"{multiline_sentence}")
```

The result looks a lot simpler and cleaner, doesn’t it?

Here’s what happened: We create a variable and name it multiline_sentence.

We then assign that variable to the actual multi-line sentence as exactly how we want it to look.

If you notice well, you will see that instead of our normal quotation marks, we use a special type of **escape character** for multi-line strings—these are called **triple quotes**.

That means that we are using a couple of three double quotes or a couple of three single quotes as the starting and ending quotes for our multi-line string. This kind of escape character tells the computer to print out what we put in between these triple quotes exactly the way we have typed it.

After that, we use our f-string to print it!

LEVEL COMPLETED!

Before you leave, use these activities to put what you’ve learned into action!

All these activities are based on the print() function, variables, and the f-string. Try these activities!

ACTIVITY 1: Introduce Yourself

What to Do?

Use the print() function to tell the computer about yourself. You should see your introduction in the console window.

Sample Expected Output

"Hi! My name is Alice."

ACTIVITY 2: Cite a Quote

Sometimes, we want to write something that a great wise person once said, and we will put them in quotes.

What to Do?

Find a quote online or use one of your own. The quote can be a funny line from a movie or an inspiring line or anything. Use the print() function to write a proper quote in the console window.

Remember that you need to escape them with the backslash (\) character properly.

Sample Expected Output

"Coding is a superpower! There is nothing you can do with this new power." — Alice Guillen

ACTIVITY 3: Write a Haiku

Have you ever heard of the Haiku poem? It is a poem from Japan. It has three freestyle lines that can be of anything at all.

What to Do?

Write a Haiku in Python. Here is an example:

Sample Expected Output

Alice Loves

coding and chocolates and

teaches awesome kids!

Level 4: Talking with Numbers

We have worked with strings, and we have seen what Python does with integers. Now, this level is about numeric data types. This type is important in coding because we use them to count, perform math operations, keep track of things, and so on.

It is vital that you know all of the different numeric data types, what you can do with them, and how to use them effectively.

Numeric Data Types

In Python, numeric data have two categories that we will be using: **integers and floats**. You have met integers, and you will use that big guy most of the time. Integers are the whole numbers (positive or negative), the numbers that we know and use in math and other things in life.

The other type is **Floats**. Well, this guy is not as frequently used as integers, and we won't be using it as much, you should still know him well. Let's talk about floats briefly.

Floats

Floats are a cool name given to **floating-point numbers**. They are numbers that can have whole and fractional parts. They are mostly written using decimal points. Do you get the point?

Here is what a float looks like:

pi = 3.14

Floats may look like decimal numbers, but they are not completely the same as decimals. We use floats when we need to make precise calculations, which we often need in math and science.

Did I tell you about modules? There is a **built-in module** in Python called the decimal module, and it is based on the float type.

The decimal module was written with some helpers from the Python language to make the decimal module faster to use, and so, the decimal module gives us numbers to work with that are closer to what we are used to seeing as humans. We also use the decimal module when we want to deal with money or any type of currency calculations.

Operators

Hey, you know integers. Now, let us see how we can use integers with operators.

In coding, operators are special symbols or characters that represent an action. Operators are usually used with **operands.** Operands are the values that you want to act on, that is the numbers or letters.

In this level, our operands will be numbers. The calculator program is written based on a set of operators that are specifically used for math. You are familiar with them. They are called **arithmetic operators.**

Arithmetic Operators

You have heard arithmetic, yeah? Yeah, math! Do you like math? I do too. Arithmetic operators are used to perform basic math functions. I will show you some of the most commonly used arithmetic operators that work just like they do in regular mathematics. You will find that there are only a few differences. I guess that's because the computer does not read math textbooks!

Operator symbol	Operator name	Action	Example	Output result
+	Addition	Sums up values	4 + 5	9
-	Subtraction	Minus a value from another	14 - 5	9

*****	Multiplication	Multiplies values	2*5	10
%	Modulus	Divides values and gives the remainder	14%7 14%6	0 1
/	Division	Divides values	8/4	2.0
******	Exponentiation	Raises to the power of	2 ** 5	32

If you want to see these arithmetic operations work, now, enter your shell and set the following variables:

```
x = 4
```

```
y = 2
```

Now you can start to use any of these different operators on your variables directly in your shell.

Try it out with codes like these:

```
x + y
```

```
y ** x
```

```
x % y
```

Cool, right?

You can try several math operations with Python and see that the computer gives you the answer. You are a genius!

Order of Operations

There is something your math teacher would want you to know. When you calculate math's operations, there is a special set of rules to be followed for arithmetic operators. This set of rules is called the **order of operations**.

Order of operations is the order in which math's operations should be calculated. Python follows this rule too. You must understand and follow it, too, especially if you use more than one arithmetic operators in a single line of code.

Let's say you have more than two calculations for a single variable. You have to follow the order.

Say you want to code this in Python:

total = 10 + (10 * .05) - 1.86 + 3

You may have faced a calculation like this when you want to pay for dinner at a restaurant. You have to multiply the sales' tax and add the result to the price with a subtraction of a coupon and the waiter's tip.

What would the total answer be in this set of calculations? You have to follow the order of operations if you want to reach the correct answer.

Here's how:

1. Brackets

When it comes to equations like this, it is only correct to focus on the brackets or parenthesis first. That is what the computer always does. The brackets shout "**IMPORTANT!**" in the rules of precedence in math. So, in that example, we would calculate the sales tax first (the .05 represents sales tax of 5.0 percent). In this calculation, 10 * .05 would equal 0.5.

2. Exponentiation

After the brackets, the next important calculation is exponentiation. When the computer sees the ** operator, it will raise the beginning number to the power of the second. What this means is, if you type 2 ** 3 in your shell, you'll get 8, because 2 to the power of 3 (also 2 x 2 x 2) equals 8. But our dinner calculation does not have any exponentiation calculations, so we move into the next operation of importance.

3. Multiplication and Division

In the order of operations, multiplication and division follow exponentiation. Both of them are at the same level of importance. That said, if the equation has both a multiplication and division calculation, we start with the calculation on the left and work our way to the right. For example, in this calculation:

8 * 3 / 2

We would first calculate 8 * 3 (which would be 24) because it’s the calculation at the very left, or the first we met. Then, we would calculate the resulting 24/ 2. The final answer would be 12.

Our dinner’s calculation does not have any other multiplication or division calculations. We can move to the next rule of importance.

4. Addition and Subtraction

In the order of operations, addition and subtraction are the last. So far, with what we have calculated, our total dinner’s bill now looks like this, with the sales’ tax calculated first because it was in brackets:

total = 10 + 0.5 - 1.86 + 3

Now, we’re left with quite a few additions and subtraction calculations.

Addition and subtraction have the same level of importance. We use the left-to-right order of calculating them, just like we did with multiplication and division.

Let’s look at the remaining steps:

First, add 10 to 0.5

10.5

Next, subtract the 1.86 from 10.5

8.64

Finally, add the last calculation of 3 to 8.64

11.64

That’s it!

Our total is 11.64 after following the order of operations.

Always remember that all the steps we looked over above will not actually show in your shell. We only went through the same steps that the computer takes so that you can understand how the computer calculates things!

Operators that Compare

Another set of operators we use in programming is what we call **comparison operators**. From the name, you should know what they do. They help us compare one value to another.

When we use these types of operators, they give us back a True or False answer known as a **Boolean type**.

Comparison operators and Booleans are super important in decision-making when it comes to coding.

There are six main comparison operators, and all of them are simple. See, let’s meet them:

Greater Than Operators

Well, just like in math, the greater-than operator looks like this: >. When you use this character in your program, the computer tells you whether or not the value on the left side of the greater-than > symbol is larger than the value on the right side of the symbol.

For example, if you write the following code in your shell:

```
4 > 9
```

the computer looks at the order and says, “Hmm, what does Human want me to figure out? Let’s see what we have here. 3 is greater than 7? Are you kidding me, Human? No! I’d better tell Human that this is False before he goes on to tell his crush!”

You can type that code into your shell and see what the computer tells you. What did it tell you? False?

Right. ??

4 is obviously not larger than 9. So, the computer saves you from believing a lie by telling you that this statement is False. Easy, right?

Less Than

The less-than operator looks like this: <

This time, this one is the opposite of the greater-than operator, and so its symbol is the opposite as well. We want to figure out if the value on the left of the less-than symbol is really less, or smaller than the value on the right side of the < symbol.

Let's try running our code with this operator to see what happens:

```
4 < 9
```

What showed up in your shell? Did it say True?

Nice! That's obviously correct because 4 is less than 9!

The computer says, "that's right, Human. Go on, tell your friends about how genius you are!"

Greater Than or Equal To

What happens when the greater-than sign marries the equal sign?

You don't know too? Well, I guess we will keep using the greater-than-or-equal-to operator to look like this: >=

We're familiar with the equal sign (=) as we have used it to assign pieces of data to variables (remember mood = proud?).

As an operator, we use it to decide, in part, if the value on the left of the >= symbol is equal to the value on the right of the >= operator.

But this operator is special.

Why?

Can't you see that it is a marriage of two symbols?

Well, we are trying to decide if the value on the left of the >= operator is greater than the one on the right or if the value on the left is the same as the value on the right.

The thing is, only one of these cases needs to be true for the computer to decide that the entire expression is True.

So, in code:

5 >= 3

What do you think this will result in? True or False?

If you say True, you're a smartass!

Because 5 is greater than 3, we know that the greater-than operator is correct. So even though the second operator, the equal-to operator, is not correct (because 5 is obviously not the same as 3), the computer still returns True because at least one operator is correct (the greater-than action).

How about this?

2 >= 2

This one is also True!

This time, the equal-to operator is correct, instead of the greater-than operator.

Here's more:

2 >= 3

What do you think? False? That's right!

Both operators are incorrect. The number 2 is not greater than 3, so the greater-than operator is incorrect, and 2 is obviously not equal to 3, which also makes the equal-to operator incorrect.

Because both operators are not right, the computer's final decision for this is False.

Great job!

Less Than or Equal To

This is the less-than-or-equal-to operator: <=

Just like the greater-than-or-equal-to operator, we use this operator to make sure at least one of the operators is correct.

For the less-than-or-equal-to operator, we are looking at the values to see if the value on the left of the <= operator is either smaller than the one on the right of the <= operator or the same as the one on the right. What do you think this code will return when you write it in your shell?

```
2 <= 3
```

That’s right! This returns True because 2 is smaller than 3. This makes the less-than operator correct, even if the equal-to operator isn’t. And since one of the operators is correct, the whole thing returns True.

How about this expression?

```
8 <= 8
```

Yup, same thing. This also returns True, as the equal-to operator is correct.

Not too bad, right?

Equal To

This is the equal-to operator: ==

Are you still looking at the last two operators? This one is a lot simpler. When we use this operator, we are asking the computer to decide if the value on the left of the == symbol is the same as the value on the right of the == symbol.

Easy!

How will this return?

```
33 == 22
```

False!

What about this one?

```
10 == 10
```

True!

Here’s a tricky one. Let us see how you hold up:

```
10 == "10"
```

What do you think the computer will do?

Did you guess False?

You’re right!

It’s okay if you guessed True too. Let me explain.

Here’s what happened: When we use the equal-to operator, we are asking the computer to decide if the values on the left and right of the == symbol are the same. Even if they look the same to us, this is what the computer figures out:

“Hmm, Human wants me to look at this new expression. What do we have here? 10 is equal to “10”? Ha! What is this? How can you say an integer is the same as a string? You’re joking, right? The values aren’t the same because an integer type is never equal to a string type! Sorry, Human, but this one is False!”

That’s okay, see, the computer made a smart decision even though it may seem dumb to us. But remember that the computer takes everything literally. We might think of texts and numbers as the same thing, but computers cannot think! And since you cannot do any calculations with text, you can add 6 and 5, but can you add 6 and hat? Not really. The thought is hilarious.

So, the computer doesn’t see integer and string types as the same types, even if you write them as 20 == twenty.

The computer will check that the values are the same type and the same value in every respect literally.

Not Equal To

Phew! Let’s meet the last one of the comparison operators!

Here’s the not-equal-to operator symbol: !=

This operator asks the computer to decide if the value on the left side of the != symbol is **not** the same as the value on the right side of the != symbol.

You can try to guess the result of these examples before you write the code in your shell:

1. 6 != "six"

2. 20 != "20"

3. 5 != 5

Did you guess correctly?

Let’s look at each example together!

1. The first expression is True. Remember, it’s asking if the two values are not the same. So, since they are not the same, it is True. It might be a tad tricky, but I remember the importance of comparing different types. So, even though our mind is telling us, there’s a number, or integer type, of 6 on the left, and the word, or string type, of “six” on the right, the computer will still return True. Why? Because an integer type is not the same as a string type. And since the not-equal-to operator is deciding exactly that, the expression is True!

2. The next example is True. Even though you might think that they look like the same number, the value on the right is a string type, and that is not the same as the integer on the left! So, the computer sees, “This integer type 10 is obviously not the same as this string type “10”?” And since they are not the same type, the result is True.

3. The third example is False. We see that the value on the left, which is the number 5, is the same value on the right, another number 5. And because the operator we are using is the not-equal-to operator, we have told the computer, “This integer type 9 is not the same as this integer type 9?” And, of course, the computer tells us, “Come on, Human, they are the same. So, this expression is False.”

Logical Operators

We use logical operators in Python programming to help us compare True or False operands.

They are very helpful because they can give our intellect and skill a blast! That means we will be creating smarter code!

There are three main logical operators: **and, or, and not**. Are you ready to meet them? Let’s see what they do.

and

The **and**-operator is used to check that the values on the right and the values on the left are both True.

We use it just by writing “and.”

The reason for the and operator is to help us when there’s a point in our code that we want to run when two conditions are met.

For instance, let’s say that you’re going through a grocery store, and you need to pick only fruits that your mother likes, but she does not like fruit juices that are not natural. She wants you to buy her natural fruit juices.

She wants juices that have apples and oranges and would want you to buy the fruit or the juice that has any of these two fruits. As you walk around the store, you see that there’s only one fruit juice with natural apple but no orange juices. Let’s say we had variables that held this information:

juice_has_apple = True

juice_has_orange = False

To check that the juice you were looking at had both orange and apple ingredients, you’d use the and operator like this:

juice_has_apple and juice_has_orange True

The and operator allows you to check both conditions: that the juice has apple and that it has orange. Only then would you buy it for your mother if both conditions were met!

Unfortunately, you won’t be buying that juice because only one condition is True.

or

The or operator is used to checks that at least one of the values being compared is True.

Going back to our juice example, let’s say that you couldn’t find any juice that had both apple and orange in it.

You need to buy the juice home to momma, so you have to decide that if the juice has either orange or apple in it, you’ll select that juice. Here is where we use the or operator in code.

To check for either orange or apple, you’d write the code like this:

juice_has_apple or juice_has_orange

That way, if the juice you were checking had either apple or orange in it, you’d buy it.

not

We use the not operator to check whether or not the value being compared is False.

Just as you’d take any juice that had apple or orange in it, you definitely would not take any that had onions on it.

Let’s say we had a variable we name juice_has_onions, and its value was True. To make sure you don’t get any juice with onions in it, you could use the not operator:

not juice_has_onion

It looks a little funny if you try to read it out loud, but the computer understands this!

You’re telling the computer, “Hey, bud, make sure that this juice I’m looking at does not have onions.”

LEVEL COMPLETE!

In this Level, we have met numbers and the interesting things we can do with them.

- There are two main numeric data types we will work with most often in Python: integers and floats.
- Operators are special characters that we use to perform actions. We have met arithmetic operators.

- Arithmetic operators are similar to math operations.
- When working with arithmetic operators, we must remember the order of operations to calculate something correctly.
- The order of operations, from most important to least important, is bracket, exponentiation, multiplication, division, addition, and subtraction (BEMDAS).
- If there is more than one calculation with the same level of importance, we calculate from left to right.

We have also met the set of operators called comparison operators. These help us compare two values to each other.

- We have operators to compare if one value is greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equal to (==), and not equal to (!=) another.

We also learned about logical operators, which we can use to make smarter comparisons in our codes.

- We use the and operator to determine if two expressions are both True.
- We use the or operator to check that at least one expression we pass to it is True.
- We use the not operator to determine if the expression we pass to it is False.

In the next level, we'll learn more things we can do with strings and some new data types! Buckle up, champ!

LEVEL ACTIVITIES

COOKIE COMPARISONS

Let's assume that you and some of your friends are eating cookies. As you eat, one of your friends says, “My chocolate chip treat has the most chocolate chips!”

You shake your packet and think, “I figure mine does.”

And now all your other friends are interested. They look down at their chocolate chip packets to see how much their cookies are in comparison to yours and the others.

How might you use coding to confirm who is True? Who has more cookies in their packet?

We could write a small program to help us make that decision. Let’s do it!

Imagine that we built an awesome machine that can scan the cookies and give you variables with the number of cookies each packet contained. How would you use comparison’s operators to help you decide?

What to Do?

For each pair of friends, write a print() command that will show the number of the cookies in the two friends’ packets, the comparison you are using, and if it is True. Here is an example:

Danny and Eddie both think that they have more cookies than the other. Let’s see who is right.

Great! We’ve first scanned their cookies, and we have seen the following variable:

danny_cookies = 15

eddie_cookies = 10

Eddie thinks he has more chocolate chips in his cookie than Danny does in his. How would we write that comparison in code?

danny_cookies > eddie_cookies

Exactly!

Now, how would we print out the results of this chocolate chip battle, including the comparison we are making?

Hint: You can use f-strings!

print(f"Eddie’s cookie has more chocolate chips than Danny's. This is {danny_cookies > eddie_cookies}!")

Awesome! It’s kind of long, but it works!

Look at some other friends who need help. Write a similar print() function for each pair of friends and their claim:

Jim and Flora

Jim says she has less than or equal to the number of chocolate chips as Flora.

jim_cookies = 30

flora_chocolate_chips = 18

Level 5: Strings Attached

In the previous level, we learned about operators and how we use them with numeric data types. Well, we can use operators with strings too. Let’s see how!

Concatenating Strings

Let’s talk about this one that sounds like a Harry Potter spell or potion. It is a fancy word that means stitching or putting things together. We can attach strings, too, you know.

We can do that using the addition (+) operator.

You know that this operator will add numbers. What will happen when we add two characters or words together?

Let’s do it. Try this code in your shell:

```
print("foot" + "ball")
```

Interesting! We have created the new string “football” by adding two separate strings, “foot” and “ball,” together.

Here’s what happened: When the computer sees the addition (+) operator, it says, “Okay, this Human wants me to add some values here. What are the values here?”

Then, it sees that you are trying to add two strings together, it says, “Well, these are two separate strings. I will just stitch them together and give it back to Human as a single string.”

Interestingly everyone that has ever filled a form has used concatenation even if they don’t know it.

Most of us have filled out forms with a section for “First Name” and a separate section for “Last Name.” Well, the people who created that form would use concatenation to show your full name after you submit the form.

Can you try to code that kind of form? It’s very useful and even easier to do. Let’s try it together!

First of all, let us store our first name and last name somewhere. Did I hear you say variables?

You are a smart kid! Variables are just the way to start! We create those like:

first_name = "Alice"

last_name = "Guillen"

Now that we have stored our first and last names, how would we print them out as a full name?

You see, in programming, there is always more than one way to do something.

For one, we can write the program using the addition operator directly in our print() function like this:

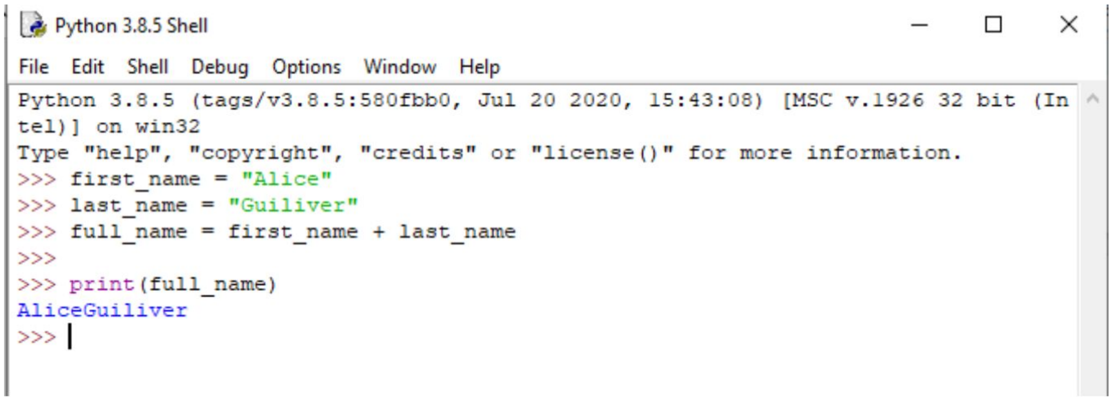
print(first_name + last_name)

That will work.

Or, we can just create another variable to hold the full name and print that one like this:

full_name = first_name + last_name

print(full_name)



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> first_name = "Alice"
>>> last_name = "Guiliver"
>>> full_name = first_name + last_name
>>>
>>> print(full_name)
AliceGuiliver
>>>
```

The funny thing about concatenation is that the attached strings will have no space. Did you notice how your name was printed out a little too combined and close together?

The computer will do exactly what you want it to, a little too literally. We asked it to add the first_name and last_name variables together exactly, and it did it exactly as it found it!

How will you make the computer print the name as a name should normally appear? You need to be exact and add the space between the names.

How?

Again, there are several ways to do this:

We could concatenate an actual space in the code between our first_name and last_name like this:

```
full_name = first_name + " " + last_name

print(full_name)
```

Or, we could just add the space after our first name like this:

```
first_name = "Alice "
```

Or before our last name like this:

```
last_name = " Guillen"
```

That way, when you print out your concatenated name, it will include the space.

```
full_name = first_name + last_name

print(full_name)
```

Can you think of other ways to print out your full name properly?

Confusing the Computer

You know that the computer is dumb and only as smart as your input. What will happen if you add integers and strings? Can we even do that? Try the following code:

```
print(4 + "hats")
```

Could you print that concatenated string? Right.

Here’s what happened: The computer sees your addition operator, and it knows that you want to add some values together. But then, it sees that one value is an integer, and the other is a string, it says, “Human, you know that integers and strings cannot ‘add’ up. Are you sure you are typing the correct instruction? I’m not sure that Human knows what he needs. Better tell them I don’t understand their code.”

And here, you get your first type error (TypeError).

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

That’s right, there is the computer telling you that the two data types cannot be added.

Let us Multiply Strings

Yes, that’s right.

In Python programming, we use the multiplication (*) operator with strings!

Don’t you want to try it out?

Write this code:

```
print(5 * "roller!")
```

What did you get? “rollerrollerrollerrollerroller.”

Neat and very hilarious! You just gave your shell five rollers to play with. How kind!

As you can see, the multiplication (*) operator works similarly with strings as it does with integers. Unlike the case with integers, though, multiplying a string a specific number of times will give you the string in the number of times you give it.

Lists

Let us meet **lists**. This guy is also an important data type for professional Python programmers.

A list is a collection of objects. A list is a set of information in a specific order that can be changed. Yes, just you know, lists.

Lists are very useful because we can use them when we work with a lot of data at the same time. Programmers work with a lot of data very often when coding. In code, we can create a list by giving it a name and then assigning it to a collection of objects we would like it to hold.

That collection of objects will be stored inside straight brackets [ ], and commas separate the objects.

When you list string objects in Python, make sure to place each object within single quotes.

The action of putting objects in a list is called **declaration** in programming. It means that we declared the my_favourite_fruits variable (as you would see below). This list stores a set of words. The two square brackets are important to define the list. We can use commands to access information about the list and to edit the data in the list.

Look at this example where I use a list to hold a collection of my favorite fruits:

```
my_favorite_fruits = ['Mango', 'Carrot', 'Orange', 'Apple']
```

There is nothing you cannot hold within a list. Including strings:

```
our_pets = ['Cat', 'Lemur', 'Puppies', 'Rabbit']
```

Or a list of integers:

```
house_numbers = [3, 5, 2, 8, 4, 5, 4, 3, 3]
```

We can even have a list of Booleans:

```
machine_answers = [True, False, False, True, True]
```

You can even have a list containing objects or stuff that are not even related or that are not even the same data type. You can have a list of mixed objects like:

```
facts_about_me = ['Alice', 'Teacher', 25, True]
```

Lists are useful and very popular because of their flexibility. There are many more interesting features about lists that make them useful.

You want to know about them?

Let’s go together!

Lists Are Ordered

One fun fact about lists is that when we create them, we use them to store a collection of objects as well as their order.

The order or arrangement of the objects in a list is important because when we need to change the list, we need that order to know and access the objects, and compare it with other lists.

For instance, try to write this code in your shell:

```
fruit_list = ["apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango"]
```

```
more_fruits = ["banana", "orange", "melon", "cherry", "kiwi", "mango",
"apple"]
```

```
fruits_list == more_fruits
```

Did you get a True?

No? See?

They are not equal lists.

Here’s what happened: As we learned in the previous level, when the computer sees the == operator, it knows that we are asking it to compare two values.

Then it looks at the first value, and says, “Okay, so we have a list of fruits_list here. It has an apple, banana, cherry, orange, kiwi, melon, and mango stored in it.”

It then goes on to check the other value and says, “Now, the second value is a list of more_fruits. It has a banana, orange, melon, cherry, kiwi, mango, and apple. So far, so good. Human is saying that both values are the same? Come on, Human! Value 1 has apple at index 1, but value 2 starts with banana. How is that similar? Since these two lists don’t have the same order, they aren’t really equal in my eyes. Time to tell Human that this is False.”

Now that you know that lists must have the same objects and the same order to be truly equal, can you create another list that would return True if we compared them?

How can we access information from a list?

```
another_list = ["I", "don’t", "like", "mushroom", "in", "my", "pizza"]
```

Let’s say we want to get some basic information about this list, like how long is it? What is the first piece of data stored on the list? How about the last? What type of data do we have stored in it?

Let us now learn a variety of python commands to access information from our list.

List Length

If we want to know the length of our list, we enter this command:

```
print (len(another_list))
```

You should see something that looks like this:

```
>>> print (len(another_list))

7
```

That result shows the length of your data, or how many objects are in the list.

Lists Can Be Accessed with A List Index

When we work with lists in our code, we usually use a single object from a list at a time in the program. That means, we need an easy way to tell the computer which objects we want to choose from a list, regardless of its position.

Indices (the plural of index) is what we use to do just that. An index is a number that represents the position of an object within a list.

The items on our list are **indexed** so that we can retrieve them easily. We can use the index operator [ ] to find any object in our list. To look for a piece of data in our list, we write the name of the list, followed by the number (position) of the object in the list in square brackets.

Now, if you want to look for the first object in our list, what number would you tell the computer to choose?

Did you say another_list [1]? Why don't you try it in your shell and see what comes out?

What do you notice? Is it what you expected?

Did you get this?

```
>>> print(another_list [1])
don’t
```

Well, that is what you will get. Why?

Try to find out how to get “I” as an answer.

The correct way is to command the console to print another_list [0]. This shows that lists in Python are indexed starting with 0.

But why?

Even though we count starting from one, computers don’t. Did I tell you that computers don’t read math textbooks?

When the computer is looking at lists, the computer starts at zero because it says, “This first object is quite literally zero spaces away from the first bracket.”

It makes sense, as the first item in a list is always the one closest to the opening bracket of a list!

So, indexing your lists looks like this to the computer:

```
I| don’t| like| mushroom|in| my| pizza

0   1      2             3     4    5 6
```

Looking at that, we see that even though the length of the list is 7, the last object in the index is 6. Hence if you want to access the last object in the list, you should understand that it is not the length of the list.

What type of data is stored within a list?

Python is cool! Are you still feeling it? You’re pretty cool too, kid.

Here is one cool stuff we want to do with our list. Enter the following code in your shell:

```
type(another_list).
```

This will return something like the following:

```
>>> type(another_list)
```

```
<class 'list'>
```

Can you guess what I am trying to make the computer tell us?

Correct! I want the computer to help us guess what type of information is stored inside the list.

Let’s try another code like this:

```
>>> print(type(another_list [1]))

<class 'str'>
```

That looks better! ‘Str’ stands for string.

We have met strings. Remember, bits of text. You can tell that a variable is a string when it has single or double quotes around it.

If you look back at your previous commands, you’ll see that we declared the another_list list entries that are all enclosed in quotes.

Lists, strings, and integers! How do we tell them apart?

Let’s look in detail at 3 different types of data: lists, strings, and integers.

You can play around with this and become more familiar by defining two more variables in lists.

Look at the following codes and type them in your shell.

```
test = ‘I don’t like mushroom in my pizza’

test2= [“I don’t like mushroom in my pizza”]
```

Now run the type and len commands on each of these and compare it with the results of another_list.

Here is what we get when we run the len commands on our two variables:

```
>>> print(len(test))

33

>>> print(len(test2))

1
```

Here is what we get when we run the type commands on our two variables:

```
>>> type(test)
<class 'str'>
>>> type(test2)
<class 'list'>
```

Ultimately, the point is that the test variable is a string, not a list. The square brackets define a list. This variable will have a length of 33 because the len function counts the characters or letters in the string.

On the other hand, test2 is a list with one element in it, surrounded by quotes, this is why it has a length of one.

Modifying Lists

We have seen in our recently completed steps that lists have:

Indexing that begins with 0

Built-in attributes like length

Now we will look at existing commands and methods we can use with lists to modify their information.

Let’s return to our test list, which we will declare again as:

```
another_list = ["I", "do not", "like", "mushroom", "in", "my", "pizza."]
```

We can add words to the list without stress! Here is an easy way to modify a list.

Let’s try to add some words. Enter the following commands:

```
another_list.insert(4,"or")
another_list.insert(5,"tomatoes")
```

Now type in another_list again and press ENTER to look at the contents of the list.

```
another_list
```

What did you see?

Amazing, right?

```
>>> another_list
['I', 'do not', 'like', 'mushroom', 'or', 'tomatoes', 'in', 'my', 'pizza.']
```

Try to type in the command again, this time with a different number and a different word.

What happens? Is it possible for the number to be too big? Let them experiment as much as they want.

Python will insert the words you give it in the position you tell it exactly. That is sweet!

Lists Can Be Sliced

Err…it is not as bad as it sounds. When programmers slice a list, it is just like slicing a piece of pie to take out the part we want to eat.

Slicing in programming is the method of selecting a specific range of items within a list.

This is similar to how we use the index to access objects in a list. It is just that we can choose more than one item when slicing.

Instead of placing a single index within the list's brackets, we give it what we call **a slice range**, which includes a starting index, the colon (:) character in the middle, and an ending index.

Here's what it looks like:

```
another_list [2:6]
```

This tells the computer, "Hey, bud, I need some items from the another_list list I gave you. Give me all items starting at the second index up to, but not including, the item at the sixth index."

So, this code would result in this output:

```
>>> another_list [2:6]
```

```
['like', 'mushroom', 'or', 'tomatoes']
```

Here’s what happened: The first index we give in the another_list list is 2. We have told the computer that this is our starting index, which is the location of the first item we want to slice from, or our slice range. The colon (:) character tells the computer we are slicing the list.

Once the computer knows that we want to slice, it will look for the ending index, which is the location of the last item we want it to cut. This lets the computer know when to stop selecting items. In this case, the ending index is 6. The computer will keep selecting items until the ending index. Still, it will not include the item at the ending index itself. This is why “in” is not part of this sliced range.

Removing Parameters

We have learned how to insert objects into a list, and how to slice a list. Now another command I want you to meet is remove().

The remove method takes one parameter, which is the value of the entry to remove. By values, we mean the information stored in each list entry. Sometimes, you may want to make a twin of your list so that you can work on that win while keeping the original. That is what we will be doing with our list now.

Go over to your shell and make a copy of our list by typing

```
test_list=another_list
```

Here is what we want to do. We will try to remove an object from this list. Let us remove “do not” from our test_list. Can you do that?

Did you guess remove ( ) like we used insert ( )?

You are very smart!

We use commands like insert and remove to modify existing lists. So, we know that our command for remove will look like test_list.remove().

We will surely have to add a parameter that will be removed because, otherwise, the computer would not know which list entry to remove!

Therefore, our resulting command is

```
>>> test_list.remove("do not")

>>> test_list

['I', 'like', 'mushroom', 'or', 'tomatoes', 'in', 'my', 'pizza.']
```

Now is a good time for us to discuss an important part of Python syntax. After that, we will do some more practice with list modification.

Other Modification to Lists: Mutation

When we create a list, we can modify the list, as we have seen. We can add, remove, and move objects around.

You know that some species can mutate their genes and become bigger or smaller or whatever. You too, you can change a list in several ways. That means that some lists are mutable.

So far, we have met other data types—like strings, integers, and Booleans. These guys cannot change. They are not mutable in this way once we've created them. These kinds of data types that cannot change are called **immutable**. Lists are mutable, though.

Earlier, we have created a list where I told you about what I don't like in my pizza. Let us create another list just to make it easy. This time, we will create a list of my favorite fruits:

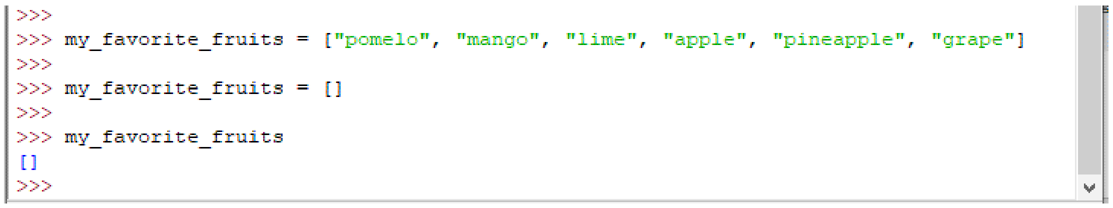
```
my_favorite_fruits = ["pomelo", "mango", "lime", "apple", "pineapple", "grape"]
```

These are my own fruits. Since lists are mutable, let's change the my_favorite_fruits list to store your own favorite fruits!

For this change, let's empty my list by assigning it to an empty list:

```
my_favorite_fruits = []
```

By doing this, we've made a mutation, or change, to our my_favorite_fruits list. If you look into your list through the shell, it should now be empty:


```
>>>
>>> my_favorite_fruits = ["pomelo", "mango", "lime", "apple", "pineapple", "grape"]
>>>
>>> my_favorite_fruits = []
>>>
>>> my_favorite_fruits
[]
>>>
```

Now, let's make another mutation.

Once you have typed in my own list, add your favorite desserts!

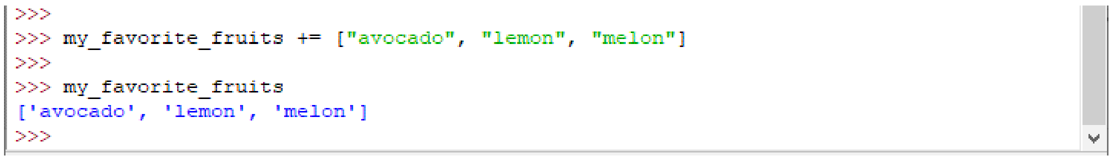
I will just assume that I know your favorite fruits and add some of them here to continue with the example but feel free to add your actual favorite fruits while coding along.

To do this, we can use something called the addition assignment operator (+=) to give our list some new desserts.

You will do that by adding it in front of the variable's name and adding the new list. Just like this:

my_favorite_fruits += ["avocado", "lemon", "melon"]

When you check your list again, what do you see? My favorite fruits have been replaced with yours!


```
>>>
>>> my_favorite_fruits += ["avocado", "lemon", "melon"]
>>>
>>> my_favorite_fruits
['avocado', 'lemon', 'melon']
>>>
```

Membership Operators

It's time I show you to a common friend of Python programmers. When we create lists, we usually check out to see if something is or is not within it. The special set of operators that do this for us is called **membership operators**.

These operators go through some input we give them and will tell us if something we are looking for is in or is not in the list. Let us talk to each of them and see what they do:

in

If we wanted to check that a specific item was within a list, we'd use the in operator. This looks for a positive confirmation that the object we type exists in the list.

So, if we wanted to make sure that avocado was in our my_favorite_fruits list, we'd write:

```
>>> "avocado" in my_favorite_fruits
```

And the result will be

```
True
```

not in

Conversely, if we want to make sure some item is not in our list, we use the not in operator. This operator looks for confirmation that something does not exist.

Let's say we wanted to make sure that no desserts were in our my_favorite_fruits list. We'd use the not in operator like this:

```
'donuts' not in my_favorite_fruits
```

Here, we also get True, which is correct, as there are no 'donuts' on our list.

These operators will be very useful in the upcoming chapters when we need to filter through lots of data in a collection of items!

Making More Changes to Lists

We have already used one method to add to a list, which is the addition assignment operator (+=).

Do you remember when I said that professional Python programmers have several easy ways to do a task? I am going to show you a few more cool methods, including methods that Python already has built-in for you to make changes to lists.

Let's go in together!

append()

You have used the insert() function. This represents one cool way to add an item to the end of a list using the built-in append() function.

Let’s say you forgot to add another fruit to your “my_favorite_fruits list”.

You can add it quickly, like this:

```
my_favorite_fruits.append ("orange")
```

Which would result in:

```
>>> my_favorite_fruits.append ("orange")

>>>

>>> my_favorite_fruits

['avocado', 'lemon', 'melon', 'orange']

>>>
```

del

We have used the remove() function to remove items from the list. Another cool way to do the same thing even more easily is using the del keyword.

As you have guessed, del is short for delete.

We use this method together with list indices. So, if you need to remove the item at the first index, you will write:

```
del  my_favorite_fruits[0]
```

And that’s it!

Since avocado is the item at the first index, that’s the item that is deleted.

Keep in mind that slice ranges work too!

So, we could write something like this:

```
del my_favorite_desserts[1:]
```

That would remove every item from the range of 1 to the end. But, why would we want to do that? In case you deleted them by mistake, you can use either the append() function or the addition assignment operator (+=) so as to add them back!

Changes Using Indices and Slice Ranges

Do you remember slicing? How we use indices and slice ranges to select one or more items in a list? We can use them to modify our list too!

For example, if we want to add “guava” as the second item in our “my_favorite_fruits list”, we can do so in the following way:

```
>>> my_favorite_fruits[1:1] = ["guava"]
```

We write code in this way, because there is already an item at the first index. If we write it some other way, we will confuse the computer, and it will not know what you’re asking it to do.

For example, if we write the code this way:

```
my_favorite_desserts[1] = “guava”
```

This will replace the item which is already at that index and put guava there instead.

So, you have to be careful when inserting new items at indices that already have items in them.

You need to use the slice range of the same starting and ending index to tell the computer to simply add a new item at that index, without changing the rest of the items in the list. After inserting a new item correctly, our list now looks like this:

```
>>> my_favorite_fruits

['avocado', 'guava', 'avocado', 'lemon', 'melon']
```

Tuples

Tuples are a lot like lists, and that's why, we can define them in a pretty similar way as we did to define the lists. Simply put, a tuple is a sequence of data.

What makes them different from lists is that tuples are immutable, i.e., the data inside the tuples can't be modified, which is opposite in the case of lists. Other than this, tuples are very similar to lists, and that would make it much easier for us to understand as we already know about lists.

Everything you know about lists is most likely the same for tuples! This means: they are ordered, as such: they can be accessed with indices, work with slice ranges, and can be made of the same or different types of items. However, there are two major differences between tuples and lists:

Tuples Use Parentheses

Tuples use curved brackets () to hold their items, instead of the squared brackets [ ] that are used with lists. This means they are created like this:

science = (“chemistry”, “biology”, “physics”)

Another difference is that:

Tuples Are Immutable

You cannot add, remove, or change the contents of tuples. So, that means methods like the append () and remove () functions, and del will not work with tuples.

When to Use Tuples Instead of Lists

Well, when we talk about tuples and lists, people wonder when to use tuples and when to use a list.

For the most part, you should choose lists when dealing with collections of items.

You can use tuples when you want to store a collection of objects that shouldn’t be changed. Our earlier tuple is a great example of this, as the science subjects are always revolving around those three.

if Statements

Every human makes decisions every day. Even as you read this book, you are always deciding whether or not you want to try out the codes I put out in your shell or not.

Though, it may seem like a hassle. Our lives are much more flexible and interesting because we can make so many decisions. Well, it is that decision-making ability that makes our Python programs more flexible and interesting—and therefore smarter.

Just as we make decisions in life, we can force the computer to make decisions in code by using what we call “**if statements”**.

An “if statement” is a block of code that gives you control of the path. The computer will take when it executes your code. This is important when we write more complex and longer programs.

When you write several codes and complex programs in the same shell, you don’t want the computer to run all of the code at a go. You would only want to run certain parts of your code when it makes sense, or when we decide it’s the right time to do so.

“If statements” allow us to set up a condition that needs to be met before any additional code is executed. This condition is usually a Boolean expression, which is a condition that the computer evaluates and decides is either True or False.

You know that Boolean data types are True or False statements. When your if statement’s Boolean expression is True, it tells the computer that it should keep going on to the next line of code. The next line of related code is usually right after the “if statement” is indented.

You know what we call the space that comes in between some lines of code so that the computer does not write them together? We call it **Indentation.** It helps the computer to group the blocks of code that belong together.

Now, let us write an if statement:

```
>>> if mood == "happy":

        dance = True

        print ("Alice is happy. She will dance.")
```

Pretty logical, right? If our mood is happy, we will probably choose to dance.

Here’s what happened: In this scenario, we are using the Boolean expression to determine our mood. When the computer gets to that line of code, it asks itself, “Is the mood equal to 'happy'?”

Then it will either answer, “Yes, the mood is definitely equal to 'happy,' I guess that means this Boolean expression is True. Also, it means: I can keep going to the next line of code,”

Or if the mood is not equal, it says, “No, the mood is not equal to 'happy,' which means this Boolean expression is False.

That means: I can’t keep going onto the next line of code. I’ll have to skip to the next line of code I see that has the same indentation as this line.”

It is important for you to remember that the computer will only keep going to the next line of code after the colon (:) if it can answer “Yes” to the question (or condition) you give it. If it can’t, it skips that code and finds the next line that isn’t indented along with the same paragraph or block of code.

What if we aren’t happy, though? What if we want to do something else rather than dancing because there is some error we want to get over?

We can add that decision into our code, too, using an **else if statement.** In coding, we give this statement a cool name: elif.

The code would look like this:

```
if mood == 'happy':

dance = True

print ("Alice is happy. She will dance.")

elif mood == 'calm':

sit_down = True

print ("Alice is calm. She sits down!")
```

Here, we have added an elif statement to our code. We use the elif after a regular if statement. The elif statement allows you to make a different decision if a different condition is met!

It is just like asking a different question if the first one you asked was answered with a “No.” Using an elif statement is perfect for our example, since we are checking for a different condition (mood == 'calm'), and are doing something completely different if that condition is True (sitting down instead of dancing).

You should also remember that the code after our “if statement” is indented.

It is important to indent codes in Python because the computer uses these spaces to figure out which blocks of code belong together.

If you want to indent a block of code, move your cursor to the beginning of the line that you want to indent and press the Tab key on your keyboard. This will add the space you need in front of your code.

Most of the time, the computer automatically indents for you, but there will be times you will need to do this yourself.

After adding our elif statement, the computer sees that we will only do one out of the two actions we have listed. We will never do both!

Here’s why: When the computer checks our Boolean expressions, it will check each of our codes until it finds one that is True. Once it finds that one, it will move to the next line of code that is indented along with it, run all other lines of code that have the same indentation, and then ignore the rest of the Boolean expressions.

In our example, the computer will be able to answer “Yes” when determining if our mood is happy or not.

Since it answered “True,” it moves onto the next line of code that sets the dance variable to True. It will also print out our message ("Alice is happy. She will dance.") since that line of code is also indented like the one before.

Since these are the only two lines of code that belong to that indentation group and seeing that the next line of code is not indented in the same way, the computer will know that it is finished with the if statement.

LEVEL COMPLETE!

Phew! We learned a lot more about strings in this Level. We have also learned how they work with some of the operators we learned about before.

- Strings can be stitched together to create new strings.
- Strings cannot be stitched with numeric data types.
- Strings can be multiplied. We have also met lists.
- Lists are a collection of items that can be changed or modified.
- Lists use brackets [ ] to store their items.
- Lists are ordered and start at an index of 0.
- You can grab specific objects within a list using the index.
- You can modify lists because you can add, reorder, and delete objects within a list. We have also met tuples, which are an immutable data type sibling of lists.
- Tuples use curve brackets () to hold their items instead of square brackets [ ].
- Tuples cannot be modified.
- You can use tuples when the collection of items you want to hold shouldn't change.

Finally, we also learned how to control the path of our code through if statements.

- If statements allow us to make decisions in code.
- Indentation is important and helps us group lines of code that belong together.
- If statements let us tell the computer which parts of our code to run and how.
- If statements use Boolean expressions to figure out what path to take in our code.

In the next Level, you will meet loops!

Loops are what programmers use when it comes to repeating blocks of code or going through larger sets of input data. See you there!

LEVEL ACTIVITIES

Activity 1: Some of My Favorite Things

Now that you how to create lists, try creating one with five of your favorite things!

Remember that you can have anything on your list so far you create it correctly.

What to Do:

Create a list named my_favorite_things and add six objects to it. Print out a message that says, “These are {your name}’s favorite things.

Use an f-string to print out this message with your name and your list of favorite things!

Sample Expected Output

```
'These are Alice's favorite things: ['Pink', 8, 'Fruits', 'Coding', 45].'
```

Level 6: Loops

One reason why the computer is so powerful is that it can repeat many actions or calculations very quickly. In programming, one way we can tell a computer to do this is through loops.

A loop is a special kind of programming statement that allows you to repeat a block of code.

Loops contain a set of instructions that are continually repeated until a specific set of conditions are met.

Let us learn about for and while loops:

For Loops

Let’s get started learning about and understanding for loops with the range command.

The for loop repeats a block of code a specific number of times. We usually use for loops with lists, and when we know how many times we need to repeat a block of code.

Creating a For Loop

Try this out in your shell:

```
for x in range(1,6):
   print (x)
```

and run the program.

Make sure to have an indent on the second line!

This is what you should see:

```
1

2

3

4

5

>>>
```

Can you guess what happened?

Try to change the numbers in the range () method.

What happens when you set the range to (1,3) what about (1,100)?

Do you remember when we talked about indexing? That is almost the same thing that the range () function does.

Loops give computers a set of instructions that are continually repeated. In a for loop, the computer executes the command for a fixed number of times. In our case, this is defined by the range.

We can also have our program list and numbers in reverse order. Have your students enter the following text?

```
for x in range(6,1,-1):
   print (x)
```

Did you see what happened there? Now we can use this method to help us code a popular children’s song. Have your students enter the following text:

```
for x in range(5,0,-1):
   print (x, 'little monkeys jumping on the bed, 1 fell off and bumped his head, momma called the doctor and the doctor said, no more monkeys jumping on the bed')
```

You should see the following:

```
5 little monkeys jumping on the bed, 1 fell off and bumped his head, momma called the doctor and the doctor said, no more monkeys jumping on the bed

4 little monkeys jumping on the bed, 1 fell off and bumped his head, momma called the doctor and the doctor said, no more monkeys jumping on the bed

3 little monkeys jumping on the bed, 1 fell off and bumped his head, momma called the doctor and the doctor said, no more monkeys jumping on the bed

2 little monkeys jumping on the bed, 1 fell off and bumped his head, momma called the doctor and the doctor said, no more monkeys jumping on the bed

1 little monkeys jumping on the bed, 1 fell off and bumped his head, momma called the doctor and the doctor said, no more monkeys jumping on the bed
```

Now, let’s say we create a list of numbers and we want to add 1 to each number in that list and then print the new numbers out.

How do we do that? A for loop to the rescue!

First, we have to declare our numbers list. We need this because loops always need a group of items to go through.

The process that the computer takes to go through a group of items is also called **iterating through a loop**.

Here is our new list of numbers:

```
numbers = [1, 2, 3, 4, 5]
```

You have declared the list in your shell, right? Awesome!

The thing is, if you forget to tell the computer that this is going to be a for loop, it would not know. You have to type in the keyword just after the list like this:

```
numbers = [1, 2, 3, 4, 5]

for
```

Great!

When we discussed the for loop, I told you that the computer uses it to go through your input.

Now with the for you added, the computer knows you want to make a loop, but it’s like, “Hey Human, you want me to make a loop, but what do you want me to loop through?”

You will say, “Hey, machine, we are just getting started!”

Alright then, let’s tell the computer which group of items to go through and how many times to do it.

For this example, we want to go through every number in our numbers list, so we write the loop to do that:

numbers = [1, 2, 3, 4, 5]

for number in numbers:

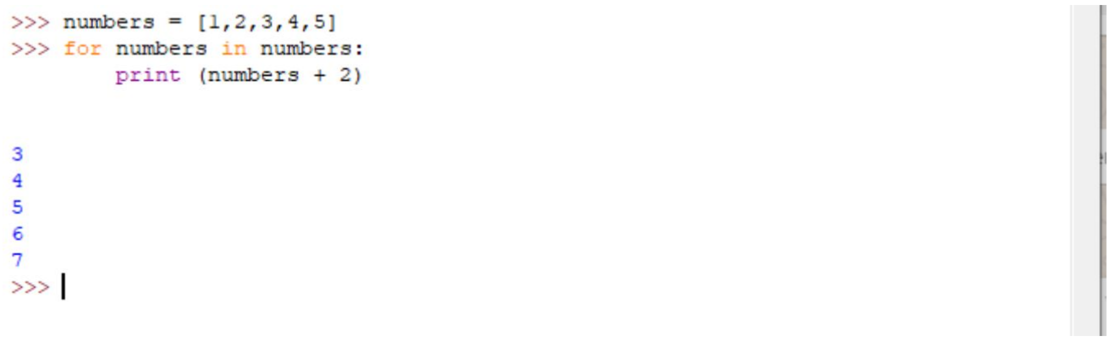
The code we just wrote is the same as telling the computer, for every number in the numbers list, do something. Cool! Now the computer knows which group of items to iterate through. Finally, let’s tell the computer to iterate through each number in our list, add 2 to it (because that’s the cool thing to do), and print that new number to the console. Remember, the block of code that comes after a colon (:) means it belongs to the related line of code above it and should always be indented:

numbers = [1, 2, 3, 4, 5]

for number in numbers:

print (number + 2)

And that’s it! If you run this code, you should see the results of your for loop in your shell.



```
>>> numbers = [1,2,3,4,5]
>>> for numbers in numbers:
        print (numbers + 2)


3
4
5
6
7
>>> |
```

While Loop

The second type of loop is a while loop. This kind of loop also repeats a block of code over and over. Still, it will keep repeating as long as a Boolean expression continues to be True to the computer. We also use this type of loop with groups of items, just like in for loops. However, the while loop is very different than a for loop, because we tend to use the while loop when we don’t know how many times we need to repeat a block of code. Remember that in for loops, we know exactly the number of times a block of code needs to be repeated.

Creating a while loop

Let’s move on now to understanding while loops. Unlike for loops, which typically stop after a fixed number of times, while loops will stop only when a specific condition is met.

Enter this text in your shell to create a While Loop:

```
x=0

while x is not 10:

    x=x+1

    print (x)

print('done!')
```

You should see the following:

```
1

2

3

4

5

6

7

8

9

10

done!
```

Now you have seen that the variable in the example is x and x starts at 0 and increased by 1 each time the loop is run according to the formula x=x+1. Once the computer reaches 10, the condition to end the loop has been met, and the loop is finished. You will then see ‘done!’ printed.

The last code we ran was a for loop – this is called a while loop. Loops are useful because they can control our progress through our code; the ‘done!’ will not print until the loop has stopped running.

For another example, let’s say that our numbers list from the previous section suddenly contained a lot more numbers:

numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

Now, instead of adding 2 to every number in the list and printing all of them out to our shell, we want to go through the list and only print specific ones.

For instance, we only want to print the numbers that, if we add 2 to them, will become a new number that is less than 20.

How can we do this? Should we use a for loop?

Probably not. We don’t know beforehand exactly how many times we will be repeating the code that adds 2. So for this kind of problem, we will use a while loop!

Let’s, first of all, declare our numbers list and assign numbers up to 20 like this:

Now, we will tell the computer what our variable is. We have used x as our variable in the previous example. We can use any letter, just like in algebra. Many programmers like to use the letter i, but let us keep to using x.

So, our iterator, the variable that’s used to keep track of the number of loops we run, is x.

numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

x = 0

We need to declare this variable because it is with a for loop. That’s the way we tell a for loop exactly how many times to repeat itself. With while loops, we need to help the computer a little.

So, this variable will be used with our Boolean expression. It will act as a signal to the while loop to keep going because we haven’t stated exactly how many times to repeat its code.

Now, let’s start our while loop. Remember that we have to tell the computer the kind of loop we want. So, we will write while after we declare our variable.

```
numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

x = 0

while
```

Great!

Now we have to give our while loop a Boolean expression to check against.

This helps the computer decide if it should continue repeating the code and when it should stop. In this example, we want to iterate through all of the numbers in our list.

Because we are keeping track of the number of loops we do, we have to use a little bit more logic to tell the computer if we have gone through each object in our numbers list.

How can we do this? Try this code:

```
numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

x = 0

while(x < len(numbers)):
```

Do you remember the job of the len() function?

We know that our numbers list has a certain number of objects in it. If we make the computer to iterate through our list the same number of times as the total amount of objects in it, then we know we have gone through them all.

That is a cool math equation we have given to the computer.

This means the Boolean expression is asking the computer to check that our iterator variable is less than the total amount of objects in our numbers list.” If it is as we said, that means we have more numbers to iterate through. Besides, it also means that we will repeat our loop.

Once our iterator is no longer less than the total amount of objects, it means we have iterated through them all, and we can finally stop repeating the loop.

Now that we have our Boolean expression in place, we can begin writing the code. We want the computer to repeat in our loop.

Now we must check and see if the new number we create with every loop after adding 2 will be less than 20. We will use the if statement! You are smart!

```
numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

x = 0

while(x < len(numbers)):

if (numbers[x] + 2) < 20:
```

Keep in mind that we only want to print the number if the Boolean expression in our if statement is True! That’s why, we write that code before our actual print statement.

Lastly, we will indent our if statement, since we need it only to run after it passes the Boolean expression in our while loop.

This time, we want to use indices to iterate through each object in our list. Because we have an iterator variable, we can use it to access the next object in the list every time we repeat the loop.

Let me explain. We know that we will be iterating through all objects in the numbers list. Since we started our iterator variable at 0, the first time we enter the loop, our Boolean expression in our if statement will be like this:

```
if (numbers[0] + 2) < 20:
```

The first time we enter the loop should run the program on the first number on the list. Don’t forget, lists start at 0!

When we are done with our repeated code, we increase or add one count to our iterator variable. That means our iterator variable is now set to 1 to pick on the second number.

So, the next loop that repeats means our Boolean expression will now look like this:

```
if (numbers[1] + 2) < 20:
```

Get it?

Because our iterator variable was increased the last time, we ran the loop. Because we are also using the iterator as our index, we will get the next item in the numbers list!

And, just like before, we will pick on the iterator variable when we are done with our repeated code, so when we repeat the loop the next time, the index we are using also changes.

Pretty interesting!

As we iterate through our loop, we ask the computer the same question: “Hey, add 2 to the next number in the numbers list and see if we will get a result that is less than 20.”

If the result is less than 20, the computer will continue to add until we finally get to the print () function and print that number that is less than 20.

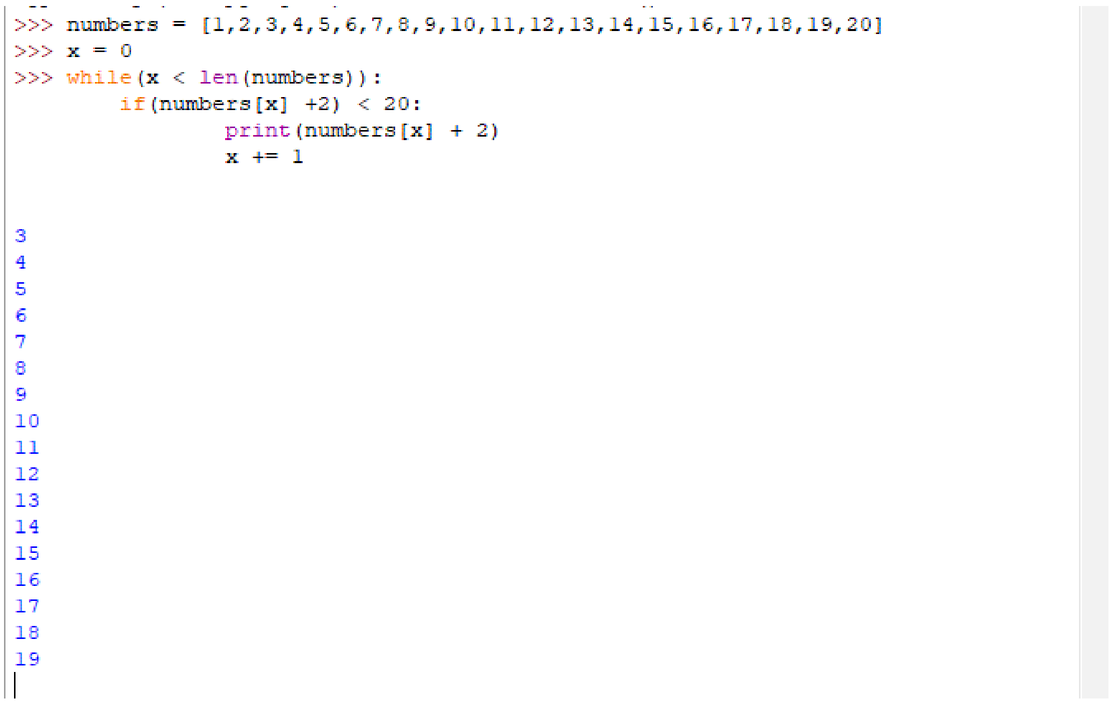
Again, we indent this code, as it is a new block that only runs after passing the first two levels of Boolean expressions.

```
numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

x = 0

while(x < len(numbers)):

if (numbers[x] + 2) < 20:

print(numbers[x] + 2)
```

Finally, we need to add the code to add or increase our iterator variable! Remember, we are the ones keeping track of how many loops we have repeated. We also know the importance of the iterator variable, because we use it in our while loop’s Boolean expression and as our index in our if statement’s Boolean expression:

```
>>> numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

>>> x = 0

>>> while(x < len(numbers)):

        if(numbers[x] +2) < 20:

        print(numbers[x] + 2)

        x += 1
```

Notice how can I put this code on the same indentation level as the "if statement"? Because there is no other code that needs to be repeated, and because we always want to keep count of how many times we have repeated our loop, we have to place the increment code (x +=1) here right before it starts the loop all over again.



```
>>> numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
>>> x = 0
>>> while(x < len(numbers)):
        if(numbers[x] +2) < 20:
                print(numbers[x] + 2)
                x += 1


3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

It is important to put the increment code when running a while loop. Otherwise, it will not give the computer a point to stop!

Try running the while loop without the code to increment our iterator variable.

What happens?

That is what we call infinite loop!

Your shell will just keep printing the same number over and over again forever:

On no!

If your program gets stuck, you can press ctrl-c in the console to cancel the program, or click the red square to stop operation.

An infinite loop is a loop that will repeat itself forever! And we don’t want that.

LEVEL COMPLETED!

In this level, we have learned a lot about for loop and while loop. We have seen how we can use them to repeat programs on the computer.

To decide or to control the flow of a program, we have branching and Looping techniques in Python. To perform decision making, we use the if-else statement in Python. To iterate over a sequence of elements, we use for loop, and when we want to iterate a block of code repeatedly as long as the condition is true, we use the while loop.

We often use a loop, and if-else statement in our program, so a good understanding of it is necessary. This Python loop exercise aims to help Python developers to learn and practice if-else conditions, for loop, range () function, and while loop. All questions are tested on Python 3.

Exercise Question 1: Print First 10 natural numbers using while loop

What To Do

Use the while loop to set the variable and the Boolean expression to be greater than 10. Remember to indent the print function before you run the program.

Expected output:

0

```
1

2

3

4

5

6

7

8

9

10
```

Exercise: Print a Pyramid

What to Do

Set your variable for what you want to print. Then, set the range with the last number as 6. That will make the computer know that it will stop at 5. Now, use the for loop to repeat the program while adding 1 to the variable in the rows. In the end, print

```
        print(column, end=' ')

    print("")
```

Expected OutPut

Print the following pattern

```
1

1 2

1 2 3
```

1 2 3 4

1 2 3 4 5

Exercise: Accept number from user and calculate the sum of all number between 1 and given number

For example, user given 10 so the output should be 55

Exercise: Print multiplication table of a given number

For example, num = 2 so the output should be

2

4

6

8

10

12

14

16

18

20

Exercise: Reverse the following list using for loop

list1 = [10, 20, 30, 40, 50]

Expected output:

50

```
40
30
20
10
```

Exercise Question 9: Display -10 to -1 using for loop

Expected output:

```
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
```

Exercise Question 10: Display a message “Done” after successful execution of for loop

For example, the following loop will execute without any error.

```
for i in range(5):
    print(i)
```

So, the Expected output should be:

```
0
1
2
3
4
Done!
```

Level 7: Importing a Library

In this level, we will start to create some awesome cool games with your newly acquired Python skills. We will be using everything you have learned and some more new skills to be added as we go.

First, let us start turning our computer into a digital dice!

Cool, right?

Type in this code:

```
from random import randint
x = randint(1,4)
print("dice roll:")
print(x)
```

The library is random, and the method we are taking from it is randint. **random** is one of the modules in Python that gives us several functions available for use.

.randint(x, y) is a type of function available through random. This function takes two parameters (two variables: x and y, which must represent any number), it will select a random number between x and y, including x and y.

You can set x and y to whatever numbers you like. In this example, we chose 1,6, just like in a dice!

Let us try to change the minimum and maximum of the numbers that can be produced or deciding only to roll again if the number is less than or equal to five.

This might look like this:

```
from random import randint

roll=randint(1, 6)

print(roll)

if roll < 5 :

    repeat=roll

    print(roll)

else:

    print("You lose")
```

Amazing game, right? Not exciting enough? Okay, let us enter some more amazing games. The TURTLE!

Turtle, Turtle, Turtle.

Python comes with a lot of prewritten code that you can play with! These ready-made codes that anyone can use are called modules. In the previous exercise, we have imported the randint module.

The other one, we will be using for this next exercise is the **turtle module**.

A module is a Python file that contains code blocks that work with another block of code that is usually grouped with them. Modules are like code libraries that give us access functions we may want to include in our program.

Third-party modules are always useful when it comes to development. The Python Package Index (PyPI) contains a wide range of third-party python modules that are capable of interacting with other languages and platforms.

For now, we want to use the turtle module to create a little turtle, make it move, change its color, and so much more. We might create our first interactive game. Let's get started!

To start using a module, you must import it to your shell. It is easy. We will import the turtle module to show you how.

Using the Turtle Module

To start using the turtle module, let us import it. Importing a module means that we are going to make the code in the turtle module available for use.

It is simple. We only type import, and we follow it by the module we want. In this case, let us import the turtle module.

Import turtle

Take it easy, tiger. When you type import, and you follow it with the name of the module you want, nothing happens.

This is it. When you tell the computer to import a module, you are telling it to grab a specific instruction manual and have it ready before we continue with the rest of our code.

In this example, we are saying, "Hey computer, we want to draw some turtles and play around with them. This kind of code is already written in the turtle module. So, I need you to grab the code that belongs in that module. That way, when we ask you to do something, you can look up how to do it in the turtle module!"

Yes, that is what we tell the computer in a single import statement!

Let's Create A Turtle

As we write the import turtle code, the computer is doing some work behind the scenes, and we will not see anything yet.

Nonetheless, we now have access to the different pieces of code in the turtle module, so that means we can create a turtle!

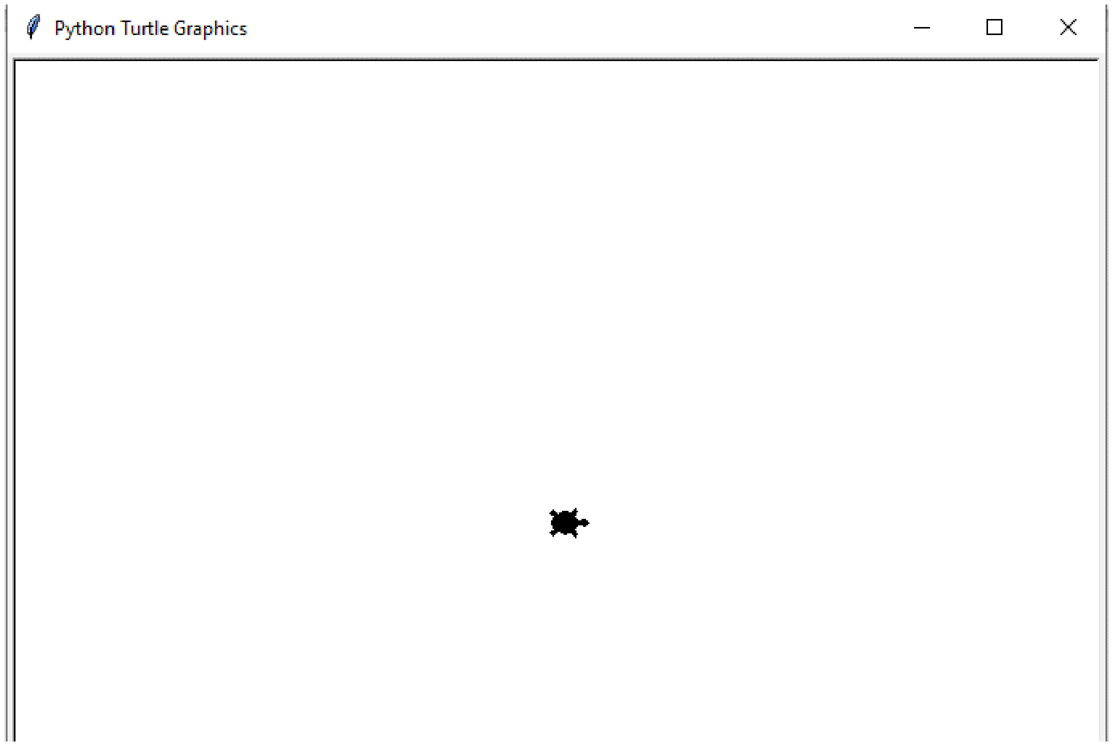
To start creating a turtle, we need to draw it. So we have to tell the computer to give us a turtle shape. We will use the turtle module's shape ()

function to do this.

Go ahead and type the following code into your shell:

turtle. shape('turtle')

Press ENTER. What happened?



You just got your first little turtle! It is so cute, isn’t it?

Let’s each of us name our turtles. I will call my turtle Kali. What will you call yours?

A separate window has opened where Kali just sits there, chilling and enjoying the whiteness of his new home.

Whenever you use the turtle module, you have the control over the Screen object, which is the window that will hold your turtle, and the turtle object, which is the little turtle you have created.

Since the turtle module creates these two things, and since it is a ready-made module with all kinds of code already written for us to interact with these objects, we can get creative!

So, let us get creative with the turtle and his boring house. Let us first resize Kali’s house.

Use this code to resize your turtle’s house:

```
turtle.setup(500, 500)
```

This code will tell the computer to reduce the turtle window size a little bit so that it would be easier to work with and make sure that Kali would not leave a trail behind.

Now, let us make Kali’s home less boring.

Kali’s home is boring. The poor turtle is just sitting alone there in the white space, not knowing what to do.

We can help Kali, or whatever you call your turtle to make it a bit more fun, though! Let’s help him get some new paint to change the color of his home.

There is a function we use to modify the color of the turtle screen interface. We call it the Screen object’s bgcolor() function. The bgcolor() function is a prewritten block of code that changes the background color of the turtle’s screen to whatever color you decide!

We use it like this:

```
turtle.Screen().bgcolor("red")
```

Let me explain this code. There are two parts to the code line. The first part tells the computer which object we want to interact with. In this case, it is the Screen.

Because the Screen object belongs to the turtle module, we make this connection known using dot notation.

In Python programming, the dot notation is a way to show that certain blocks of code are related to each other.

So, to tell the computer that we want to use the Screen object from the turtle module specifically, we use a dot (.) in between them.

The second part of the code line is where we tell the computer to use a specific function that belongs to the Screen object to change the color.

In our case, we are working with the bgcolor() function. Just as before, we put a dot in between the Screen object and the name of the function we want to use:

```
turtle.Screen().bgcolor()
```

Finally, we give the bgcolor() function a color:

```
turtle.Screen().bgcolor("red")
```

When you press ENTER, you will see how your screen color changes to red or whatever color you like.

Amazing, right?

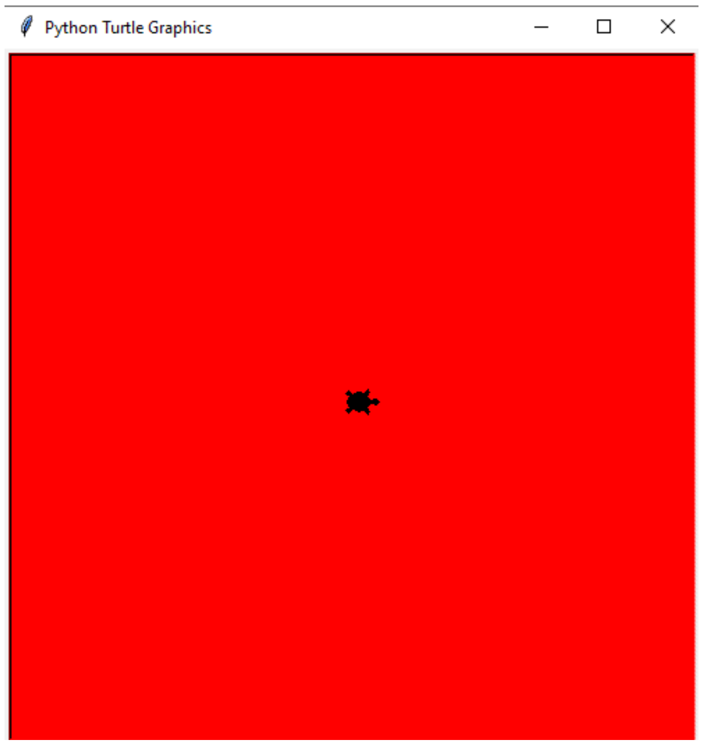
So, altogether, the computer understands our code to mean, “Please, bud, find the turtle module’s Screen object. Then find the bgcolor() function that belongs to it. Then, do what the bgcolor() function says to do with the color we’ve given it.”

In this case, it is the color red. Remember, we didn’t write the code for this; it is already written for us in the turtle module. That’s why, we needed to import the turtle module first before using it. Now, the computer can go through the turtle module’s code, find the objects and functions we are asking it to use, and run the code that is already written for us. Sweet!

If you have followed along so far, Kali’s home should now be red. This means that writing this code:

```
>>> import turtle

>>> turtle.shape('turtle')

>>> turtle.penup ( )

>>> turtle.setup(500, 500)

>>> turtle.Screen().bgcolor("red")

>>>
```

The code should result in something like this:



Whoa! That's really red. Also, not the kind of red that suits a turtle's home, and not the kind I hoped to give to Kali's home.

Well, I think that turtles will like it in the ocean, so maybe let's choose a blue color.

All we have to do is run the same code and change red to blue.

Still, that blue is not so nice. I want a nicer shade of blue that would make Kali feel more at home like ocean water somewhat blue.

We can modify the color of the code. But before we do this, let's talk a little bit about how colors work.

Colors Are Just A Little R, G, And B

Science says that we see only three basic colors. On a computer too, all colors are just combinations of the three primary additive colors, which are red, green, and blue.

Computers use additive color, which means colors are created by adding different ***levels*** or shades of red, green, and blue together.

Because computer screens give off light and can only combine levels of light to make colors, we can tweak these levels!

When choosing colors you want a computer to display, you have to tell it exactly how much of each primary color to show to get the resulting color we want.

We call the primary color tweaking the **RGB color model**. The RGB color model stands for the **red, green, blue color** model. We represent the levels of each color with their corresponding numbers. That means colors are represented by numbers, with each number representing how much red, green, and blue should be used:

(R, G, B)

Each number in the model represents the level of red, green, and blue, contained in the specific color you want. The first number is how strong you want the red color to be. If you wanted the strongest red and absolutely no other color, you would give the RGB model the maximum amount of red, zero for green, and zero for blue. 255 is the maximum number a color can get. So, for red, you have:

(255, 0, 0)

Similarly, for the strongest green color, you would give the maximum amount of green, and no red or blue:

(0, 255, 0)

And lastly, to create a total blue, you would have no red and no green:

(0, 0, 255)

Why is the maximum amount 255?

Let me show you a little something. Let us first talk about how computers store information.

The computer uses the numbers 0 and 1 to process information. A **bit**, which is a cool name for binary digit, is the smallest unit of data a computer can hold.

A bit represents either a 0 or a 1. To the computer, those numbers mean “off” or “on,” respectively.

Another unit of measurement that the computer uses to represent and store information like letters or numbers is a byte. One byte is equal to eight bits.

And that is also exactly the value of one RGB!

So, in 8-bit binary, or more specifically, in the eyes of the computer, the number 0 is the same as 00000000, and the number 255 is equal to 11111111.

You get the drift.

So, the highest amount of data we can store in the computer for one value will use up all eight bits in a single byte. And since an RGB value is exactly one byte of data, this translates to the maximum number of 255 for RGB values.

New knowledge! Cool.

I told you that the computer does not read math textbooks, but it uses math language. Let me show you a peep into the computer’s brain.

The Hexadecimal System

The color I wanted for Kali’s home is #1DA2D8.

Sounds like a magic spell for I dunno, right?

Yeah! What color is #1DA2D8? That is the very specific shade of blue that the ocean deep has written in hexadecimal form.

The hexadecimal system is a numbering system that uses 16 symbols to represent numbers instead of 10. With 16 symbols to represent unique numbers, we consider hexadecimal to be a base-16 number system.

We humans used a base-10 number system, which we call the decimal system.

That means we use exactly 10 symbols to create numerical figures, which you are probably familiar with: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. These are the only symbols we use to create all of our numbers.

Hexadecimal is also called hex or base 16 system. Hex, like decimal, combines a set of digits to create large numbers.

It just happens that hex uses a set of 16 unique digits. Hex uses the standard 0-9, but it also incorporates six digits you wouldn't usually expect to see creating numbers: A, B, C, D, E, and F.

There are many (infinite!) other numeral systems out there. Binary (base 2) is also popular in the engineering world because it is the language of computers. The base 2, binary, system uses just two-digit values (0 and 1) to represent numbers.

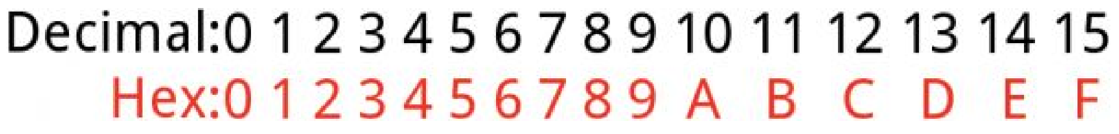
Hex, along with decimal and binary, is one of the most commonly encountered numeral systems in the world of programming. It is important to understand how hex works, because, in many cases, it makes more sense to represent a number in base 16 than with binary or decimal.

The Digits: 0-9 and A-F

Hexadecimal is a base-16 number system. That means there are 16 possible digits used to represent numbers. Ten of the numerical values you are probably used to seeing in decimal numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9; those values still represent the same value you're used to. The remaining six digits are represented by A, B, C, D, E, and F, which map out to values of 10, 11, 12, 13, 14, and 15.

Hex digits

You will probably encounter both upper and lower case representations of A-F. Both work. There isn't much of a standard in terms of upper versus lower case. A3F is the same number as a3f is the same number as A3f.



Decimal:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

From the moment we reach 10 in the decimal system, we are already required to use two characters. In hexadecimal, we still only use one character for the number 10, which is “A.” This means we are already saving the space of one character.

So, in this system, the different characters go together to create a 6-digit hexadecimal color. The first pair of characters in a hexadecimal color is the

R-value, the second pair is the G value, and the third pair is the B value.

The big difference between this system and the decimal system is that we just use six characters—“1DA2D8” instead of a possible 9 (if you translate this color, though, it is only eight: 29, 162, 216).

To make these numbers a hexadecimal color, we add a # sign in front of the numbers. The hash sign in front of hex numbers tells the computer that you are representing a color in hex.

Paint for Kali

Now that we know how to get specific colors using the RGB model, let’s give Kali the right shade of blue for his new home.

The first thing we must do is to tell the turtle module that we want to use the RGB scale of colors, instead of the standard “named” colors like we first did. The one that got us red and then blue.

So, write the code like this:

```
turtle.Screen().colormode(255)
```

Nice! Now the computer will be expecting us to give it a specific value for red, green, and blue!

So, we will do that now with the bgcolor () function like this:

```
turtle.Screen ().bgcolor(29, 162, 216)
```

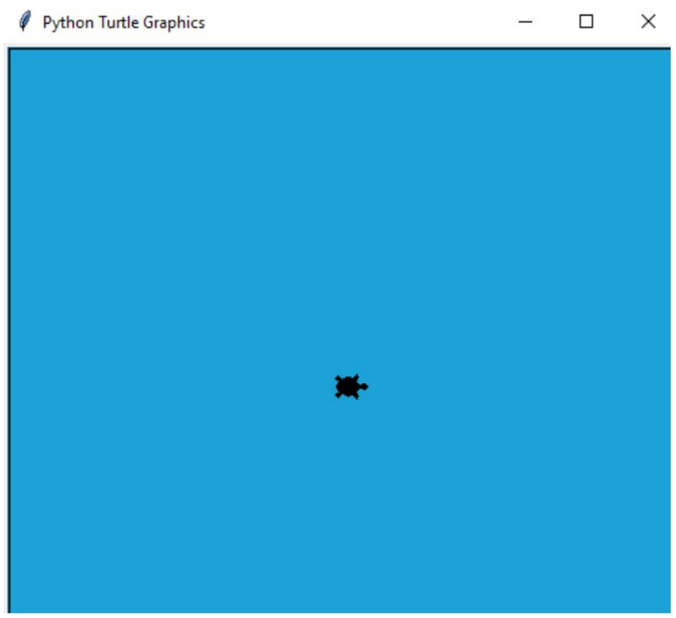
Sweet!

Now, press ENTER.

In my shell, Kali now has an awesome ocean blue scene for his new home. Kali is happy now.

```
>>> turtle. Screen().bgcolor("red")

>>> turtle. Screen().bgcolor("blue")

>>> turtle.Screen().colormode(255)

>>> turtle.Screen().bgcolor(29, 162, 216)
```

```
>>>
```



Setting Turtle Window Background Image

You can also set background images in the turtle window. For this, write the following code

```
import turtle

my_turtle = turtle.Turtle()

my_turtle.shape("turtle")

my_turtle.screen.bgpic("turtle1.png") # set background image

turtle.done()
```

bgpic() method is used to set the window background image.

It takes an argument that is an image name and must provide the format of the image.

The image must be placed in the same folder where your python file is placed.

Kali’s True Colors

Now that we have changed Kali’s home to be a nicer shade of blue, let’s make him more fun to look at. Who likes a black turtle anyway?

Let us make Kali green!

The code we will use to change the turtle's color is similar to our earlier example, except, we are changing the turtle object instead of the Screen object.

Can you guess the code to make Kali a bit green? Remember that we must make up 255, and we don't want him all green. So, we will make him a bit red and blue too. Like this:

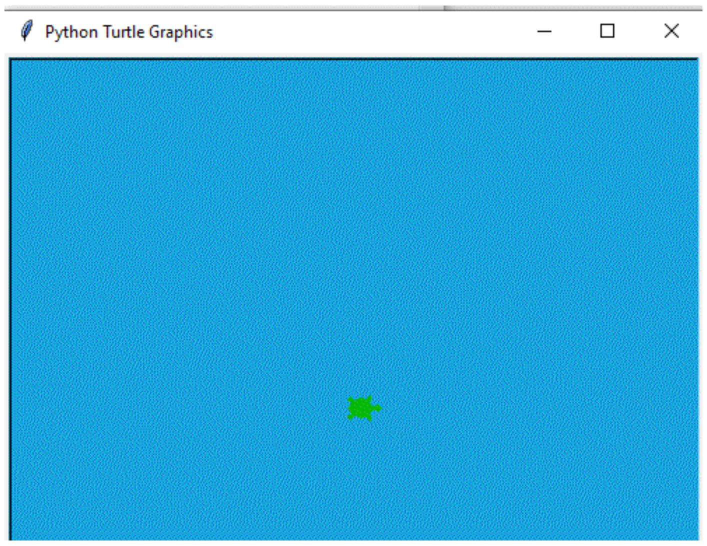
turtle.color(9, 185, 13)

Exactly!

Because we are changing the turtle object itself, we had to call our turtle object directly. Then, the turtle module's color () function will help to give it a specific shade of green, using RGB values.

Remember, you can always choose your colors, so feel free to pick a different color for your turtle!

Kali should be a nice green now:



But even with these color changes, it's still quite hard to see Tooga, so let's make him a little bigger. Yes, we can!

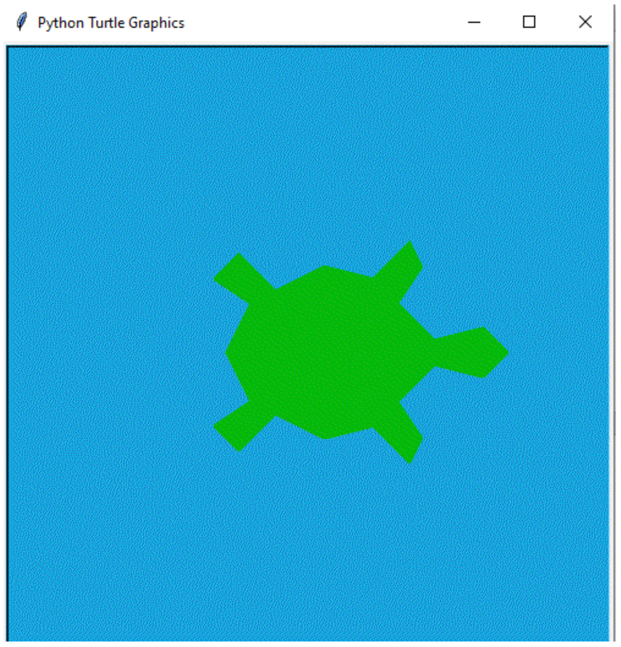
Big or Little Kali?

Let’s make Kali a little bigger so that it’s easier to find him! We can do this by using the turtlesize() function:

```
turtle.turtlesize(10, 10, 2)
```

The turtlesize() function uses three numbers as its input: The numbers represent the objects’ length up/down and sides. The third number sets the size of the turtle’s outline. The outline of an object is the part of it that is outside. In our case, it is the part that is a darker green.

As you can see, our code has made Kali bigger. Do you think he is a little too big?



If you ever need to reset your turtle to the original size, you can do so by using the resizemode() function. Just write the code like this:

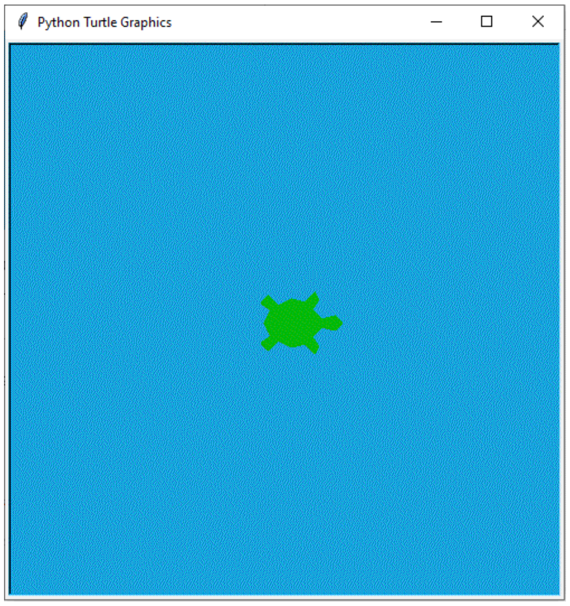
```
turtle. resizemode('auto')
```

The 'auto' parameter tells the computer to use the default values that came from the turtle module.

Now, let us try resizing Kali again, but not as big this time!

```
turtle.turtlesize(3, 3, 2)
```

Ahh, see? That is better.



We can use the third parameter in that code to tweak Kali's outline, or we can change his outline directly by using this code function:

turtle.turtlesize(outline=10)

When you do this, Kali's outline will be a bit thicker by 10 levels!

Turtle Movement

Kali is quite enjoying his home. There is so much water to swim around in!

Let me make it real. Let's move Kali around using the forward() and back() functions. Both functions take a number as an input and will be the number of pixels Kali will move across the screen. Pixels, a cool name to call picture element, are the small little dots that make up what we see on a computer screen.

Pixels are the most common unit of measurement for pictures and drawings! So, to move Kali forward 200 pixels, you'd write:

turtle.forward(200)

And to move him backward 350 pixels, you'd write:

turtle.back(350)

Now he is moved back toward the left side of the screen! The same goes for moving him left and right.

Play around the codes like this:

turtle. right(150)

You will see your turtle move around in his home.

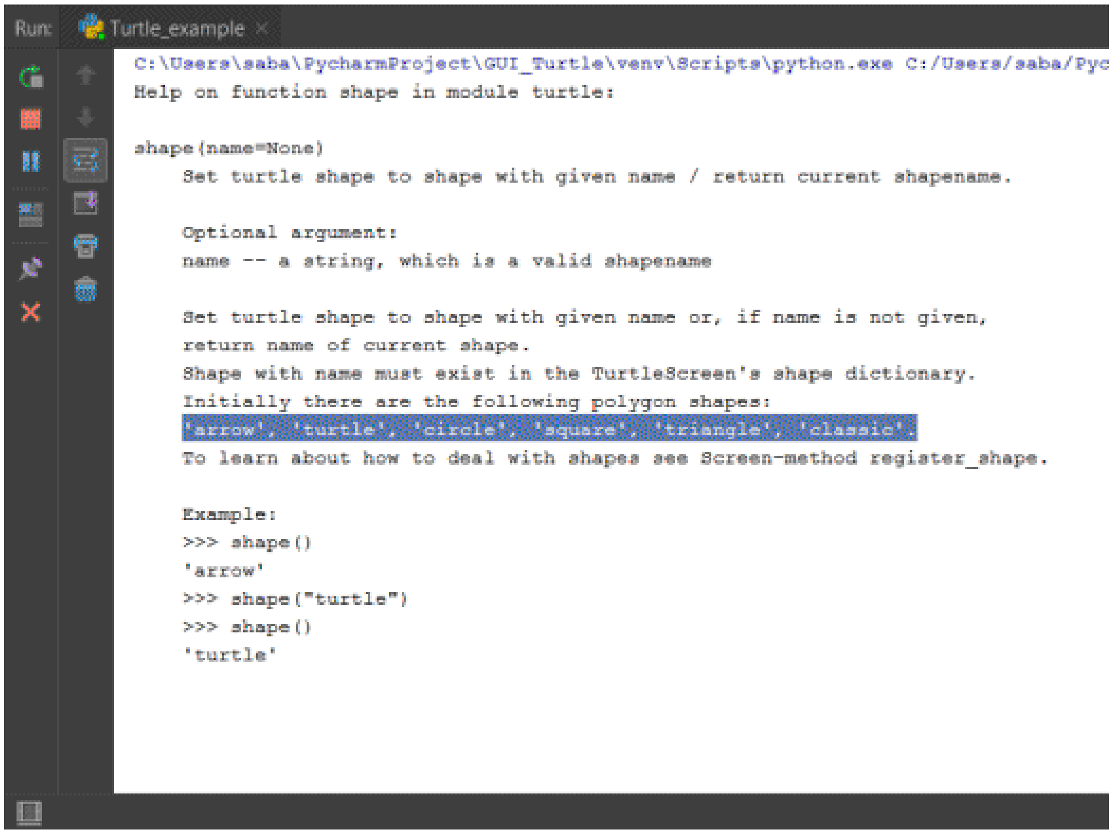
Finding Available Shape Of The Turtle

If you want to know which type of shape you can draw, then run the following command for checking supported shapes by turtle module.

import turtle

help(turtle.shape)

On running this code, you will get the following result –



```
C:\Users\saba\PycharmProject\GUI_Turtle\venv\Scripts\python.exe C:/Users/saba/Pyc
Help on function shape in module turtle:

shape(name=None)
    Set turtle shape to shape with given name / return current shapename.

    Optional argument:
    name -- a string, which is a valid shapename

    Set turtle shape to shape with given name or, if name is not given,
    return name of current shape.
    Shape with name must exist in the TurtleScreen's shape dictionary.
    Initially there are the following polygon shapes:
    'arrow', 'turtle', 'circle', 'square', 'triangle', 'classic'.
    To learn about how to deal with shapes see Screen-method register_shape.

    Example:
    >>> shape()
    'arrow'
    >>> shape("turtle")
    >>> shape()
    'turtle'
```

Doodles And Shapes

Even though it is called the turtle module, this module can be used to create basically any shape in Python. Let’s see how we can do that!

Creating A Pen

To start drawing, we need a drawing tool! For that, we can create an instance of the turtle object and call it pen.

```
pen = turtle.Turtle()
```

Remember that the computer gives us everything in the module we imported, so we get all of the pre-built functions that come with the original object! In this sense, our pen variable can use the same functions we used earlier. That’s how we can write code like this:

```
pen.color("blue")

pen.pensize(5)

pen.forward(100)
```

Do those functions look familiar? That’s because they are! We used them with Kali too!

Now, since we want to use the turtle object to draw, we reuse these same functions to help us draw instead of simply moving an object around!

Creating A Shape

Let’s keep using our pen to draw. How would you draw a red square? You would probably pick a red pen to start. In code, we would do the same thing! We would “pick” our color by changing the color of our pen:

```
pen.color("red")
```

Next, we would move our pen in the shape of a square just like you would move your hand in real life.

To start, we would draw the first line. So in code, it could be something like this:

```
pen.forward(100)

pen.left(90)

pen.forward(100)

pen.left(90)
pen.forward(100)

pen.left(90)

pen.forward(100)
```

After this code, our square should be complete!

Awesome!

You may notice that the arrow shape is blocking our cool new square, making it look like a map. Luckily, we can hide the arrow so we can see our full square in all its glory.

To do that, use the hideturtle() function:

```
pen.hideturtle()
```

The hideturtle() function does just that: it hides the shape of the turtle object you are currently using.

Although we are using a copy of the turtle module that we just named “pen,” the names of the functions that it comes with will still be related to the original turtle object. That’s why, the function is called hideturtle() and not hidepen, for example:

What easier way can we create this shape? You got it. Loops! Let us loop it with this code:

```
for i in range(1, 5):

pen.forward(100)

pen.left(90)
```

That’s much better!

So, we have created a program that draws a line 100 pixels long. We can draw various shapes and fill different colors using turtle methods. There is a plethora of functions and programs to be coded using the turtle library in python. Let’s learn to draw some of the basic shapes.

Draw a Star

filter_none

edit

play_arrow

brightness_4

```
# Python program to draw star
# using Turtle Programming
import turtle

star = turtle.Turtle()

for i in range(50):
    star.forward(50)
    star.right(144)

turtle.done()
```

Shape 3: Hexagon

filter_none

edit

play_arrow

brightness_4

Hexagon

```
import turtle

polygon = turtle.Turtle()


num_sides = 6

side_length = 70

angle = 360.0 / num_sides


for i in range(num_sides):

    polygon.forward(side_length)

    polygon.right(angle)


turtle.done()
```

Some More Amazing Turtle Programs

1. Spiral Square Outside In and Inside Out

filter_none

edit

play_arrow

brightness_4

```
# Python program to draw
# Spiral Square Outside In and Inside Out

# using Turtle Programming

import turtle   #Outside_In

wn = turtle.Screen()

wn.bgcolor("light green")

wn.title("Turtle")

skk = turtle.Turtle()

skk.color("blue")


def sqrfunc(size):

    for i in range(4):

        skk.fd(size)

        skk.left(90)

        size = size-5


sqrfunc(146)

sqrfunc(126)

sqrfunc(106)

sqrfunc(86)

sqrfunc(66)

sqrfunc(46)

sqrfunc(26)

filter_none
```

edit

play_arrow

brightness_4

```
import turtle  #Inside_Out
wn = turtle.Screen()
wn.bgcolor("light green")
skk = turtle.Turtle()
skk.color("blue")

def sqrfunc(size):
    for i in range(4):
        skk.fd(size)
        skk.left(90)
        size = size + 5

sqrfunc(6)
sqrfunc(26)
sqrfunc(46)
sqrfunc(66)
sqrfunc(86)
sqrfunc(106)
sqrfunc(126)
```

sqrfunc(146)

Output:

2. User Input Pattern

filter_none

edit

play_arrow

brightness_4

```
# Python program to user input pattern
# using Turtle Programming
import turtle   #Outside_In
import turtle
import time
import random

print ("This program draws shapes based on the number you enter in a uniform pattern.")
num_str = input ("Enter the side number of the shape you want to draw: ")
if num_str.isdigit():
    squares = int(num_str)

angle = 180 - 180*(squares-2)/squares
```

```
turtle.up


x = 0

y = 0

turtle.setpos(x, y)



numshapes = 8

for x in range(numshapes):

    turtle.color(random.random(), random.random(), random.random())

    x += 5

    y += 5

    turtle.forward(x)

    turtle.left(y)

    for i in range(squares):

        turtle.begin_fill()

        turtle.down()

        turtle.forward(40)

        turtle.left(angle)

        turtle.forward(40)

        print (turtle.pos())

        turtle.up()
        turtle.end_fill()

time.sleep(11)
turtle.bye()
```

3. Spiral Helix Pattern

filter_none

edit

play_arrow

brightness_4

```
# Python program to draw
# Spiral  Helix Pattern
# using Turtle Programming

import turtle
loadWindow = turtle.Screen()
turtle.speed(2)

for i in range(100):
    turtle.circle(5*i)
    turtle.circle(-5*i)
    turtle.left(i)
turtle.exitonclick()
```

Filling Shapes With Color

For most of this chapter, we have been drawing shapes that are only outlined. Python can also color shapes!

To color a shape, we have to tell the computer what color we want to fill our shape with

```
pen.fillcolor('blue')
```

Then we signal to the computer that we want to fill the shape we are about to draw:

```
pen.begin_fill()
```

Now, you can begin drawing any shape you would like. Free draw by giving the pen directions to move it around, or read on and use one of the built-in functions (which we cover right after this section). For this section, I will choose a circle:

```
pen.circle(50)
```

And now, we will tell the computer that we have finished drawing our shape so that it can finish the color-filling process:

```
pen.end_fill()
```

That is a nice orange circle:

write()

Another fun built-in function the turtle module provides is the write() function.

We use it to write text on the screen. It’s similar to the print() function, just a little more sophisticated.

```
pen = turtle.Turtle()

pen.write("Alice rocks!")
```

This code will use the current pen size and color for the text. If we want to change the font, which is called typestyle, and the size of the text we write, we can give the write() function a second parameter!

I am going to change my font to one that’s easy to read:

```
pen.write("Alice rocks!", font=("Open Sans", 60, "normal"))
```

See what we did there? The first parameter is the text we want to output, and the second parameter is a tuple that holds details about the font!

LEVEL COMPLETED!

Activity: Let’s Write a Story!

Now, we are going to use our new Python skills to make something that is unique and all our own! Let’s write a story where the nouns and adjectives change each time we create the story.

What To Do

1. Start with a simple story.

Here is an example of what we will be doing:

```
name1="Anna"

adj1="happy"

sentence1=name1+" woke up in the morning feeling very "+adj1+"."

print sentence1
```

After running this code, the variable sentence1 now has the value ‘Anna woke up in the morning feeling very happy.’ The variables noun1 and adj1 are both strings – so is sentence 1. However, sentence 1 uses noun1 and adj1 within its value to combine the strings!

2. Create a fill-in-the-blank story

Now for the fun part! You can now create your own fill-in-the-blank story or use mine to create a funny story. Write four to five sentence variables that use, in total, three noun strings, three adjective strings, and three-place strings.

Or, if you want, you can use these:

```
sentence1= "Last year, I went on a "+adj1+" trip to "+place1+"."

sentence2= "The weather there was "+adj2+", and I couldn't wait to eat a big "+noun1+" while I was there."

sentence3="Next year, I want to go to "+place2+", because I've always wanted to see the "+adj3+" "+noun2+"."
```

Notice that we have to include spaces before and after using a string variable! Otherwise, the words will be smashed together.

Before you run this code, you have to declare the variables noun1, adj1, place1.

Come up with a random list of nouns, places, and adjectives and randomly declare them to each variable like this:

```
adj1="smelly"

adj2="silly"

adj3="adorable"

place1="Toronto"

place2="Texas"

place3="Mexico"

noun1="chair"

noun2="shoulder"

noun3="statue"
```

Once you have finished setting up the assignments for each variable, combine it with your sentence variables. The order in which we enter commands matters in Python, so if you define your sentences first before adding the pieces of random variable assignment code we just finished, your code will throw an error. Make sure you define all your variables before you try to use them! To print all your sentences together at the end, you can use the print command like this:

```
print sentence1,sentence2,sentence3
```

Here is what the final code looks like:

```
adj1="smelly"
adj2="silly"
adj3="adorable"
place1="Toronto"
place2="Texas"
place3="Mexico"
noun1="chair"
noun2="shoulder"
noun3="statue"
sentence1= "Last year, I went on a "+adj1+" trip to "+place1+"."
sentence2= "The weather there was "+adj2+", and I couldn't wait to eat a big "+noun1+" while I was there."
sentence3="Next year, I want to go to "+place2+", because I've always wanted to see the "+adj3+" "+noun2+"."
print (sentence1,sentence2,sentence3)
```

Here is how it looks when you run the code:

```
Last year, I went on a smelly trip to Toronto. The weather there was silly, and I couldn't wait to eat a big chair while I was there. Next year, I want to go to Texas, because I have always wanted to see the adorable shoulder.
```

Now let’s change the story each time we run the code

The above method is the simplest way to create a fill-in-the-blank story using python. However, we want to create a story that changes every time we run the code. We want our program to be able to choose a random noun, adjective, or place from our list and automatically place it in our story.

Storing variables in a list

We could declare adj1 always to be “Smelly,” but then our story would not change each time we ran our code. We want a variety of names for our story to choose from! Let’s store the options in a list.

```
adj_list=["wild","fluffy","hilarious"]
```

Now, we want to randomly choose which adjectives will be assigned to adj1, adj2, and adj3. Do you remember how to do that? We will be using the random library again, back from our lessons on numbers.

An example of random being used is as follows:

```
from random import randint

roll=randint(1, 6)

print(roll)
```

Using the random library to obtain a random adjective

How can we use the randint method to obtain a random adjective? Think about this for a while. What kind of variable type is the index of a list? When you have an idea, try writing out a bit of code that assigns a random name to the variables adj1, adj2, and adj3.

When you are ready, compare your idea to this following bit of code:

```
minindex=0

maxindex=len(adj_list)-1

index1=randint(minindex,maxindex)

adj1=adj_list[index1]


index2=randint(minindex,maxindex)

adj2=adj_list[index2]
```

```
index3=randint(minindex,maxindex)

adj3=adj_list[index3]
```

In writing this code, my goal was to generate a random number that corresponds to an index of every adjective in the list of possible adjectives. I did this by defining the minimum and maximum possible index values (lines 1 and 2). Because indexing begins with 0, the highest number that we can index is always the length of the list minus 1.

defining an index which takes a random number with these minimum and maximum values (line 3) storing the adjective at this index as the variable to be used (line 4) repeating this three times

Now that we've taken care of the adjective variable assignments do the same with the other variables! You will want to create a place_list and a noun_list and use the randint method to select a random variable from your list.

Putting it all together

Once you have finished setting up the random assignments for each variable, combine it with your sentence variables. The order in which we enter commands matters in Python, so if you define your sentences first before adding the pieces of random variable assignment code we just finished, your code will throw an error. Make sure you define all your variables before you try to use them!

Your final code will look something like this:

```
from random import randint

adj_list=["wild","fluffy","hilarious"]

place_list=["Chicago","China","Brazil"]

noun_list=["telephone", "karate", "toilet"]

minindex=0

maxindex=len(adjList)-1
```

```
index1=randint(minindex,maxindex)
adj1=adj_list[index1]
index2=randint(minindex,maxindex)
adj2=adj_list[index2]
index3=randint(minindex,maxindex)
adj3=adj_list[index3]
minindex=0
maxindex=len(place_list)-1
index1=randint(minindex,maxindex)
place1=place_list[index1]
index2=randint(minindex,maxindex)
place2=place_list[index2]
index3=randint(minindex,maxindex)
place3=place_list[index3]
minindex=0
maxindex=len(adj_list)-1
index1=randint(minindex,maxindex)
noun1=noun_list[index1]
index2=randint(minindex,maxindex)
noun2=noun_list[index2]
index3=randint(minindex,maxindex)
noun3=noun_list[index3]
sentence1= "Last year, I went on a "+adj1+" trip to "+place1+"."
```

```
sentence2= "The weather there was "+adj2+", and I couldn't wait to eat a big "+noun1+" while I was there."

sentence3="Next year, I want to go to "+place2+", because I've always wanted to see the "+adj3+" "+noun2+"."

print (sentence1,sentence2,sentence3)
```

Here is what I got the first time I ran this code:

```
Last year, I went on a fluffy trip to Brazil. The weather there was fluffy, and I couldn't wait to eat a big telephone while I was there. Next year, I want to go to Brazil, because I have always wanted to see the hilarious karate.
```

And here is what I got the second time I ran this code:

```
Last year, I went on a hilarious trip to China. The weather there was fluffy, and I couldn't wait to eat a big telephone while I was there. Next year, I want to go to Brazil, because I have always wanted to see the wild karate.
```

As you can see, the story changes each time you run the code. The more variables you define, the more options for your story, and the more hilarious combinations you may create!

Level 8: Reusable Codes

Coding is really about reusability, or how easy it is to use something again and again. Programmers write codes that can do repetitive, complex, or time-consuming things. Still, if we had to write it every single time we needed to use it, coding would be stressful.

We are usually referring to doing a repetitive task that can be automated but isn't.

Functions and modules are a way to write reusable codes. And yes, we have already used so many in this book!

Most programs are made up of one or many modules, and each of those modules is usually made up of several functions.

Let's see how writing code in this way helps us have smarter programs.

Functions

As we have learned, functions are reusable blocks of code that can do something specific or return a value.

Let's say you want to greet a person every time they used your program. You could write a print() function every single time we needed to greet them or move this action of greeting a person into a function with this code:

```
def greet():

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

That code will work any time by writing code like this:

```
greet()
```

To create a function, we first need to describe what it will be called and what it will do. We start by using the def keyword, which tells the computer that we are writing a function.

We use that keyword to define what our function will do when we use the def keyword.

Next, we name our function. Because we will be greeting people when we use this function, the name “greet” is a good choice, as it clearly describes what our function is doing.

We then add some brackets () to our function name. We may add parameters in the parentheses later, but for now, we don’t have any.

Lastly, a colon (:) shows that the following indented lines of code will be part of our function. That’s it!

An important thing to know about functions is that they don’t run on their own. That means that whenever a computer comes across a function, it automatically skips the code within it.

To use a function, you must call it. That means you must tell the computer to start executing the called function’s code. If we don’t call functions, the code within them will never be run!

Parameters

Our greet() function is pretty normal. That means whenever we call it, it says, “Hello, person!”

If we wanted to greet the person by their name, we would use parameters.

We add a parameter to a function by placing it in between the parentheses that come after the function name, like this:

```
def greet(name):

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

By adding this parameter to the function, we are now able to use it within our function. That means we can do this:

```
def greet(name):

print(f"Hello, {name}!")
```

Now, whenever we call our greet() function, it will use the parameter you pass into it, meaning this code:

```
greet("Alice")
```

This will result in this output:

```
'Hello, Alice!'
```

Pretty cool! You know what, though? We can even change our greeting depending on the person. We might say, “What’s up, Alice? Nice to see you again!” if we are greeting someone we know very well, or “Hello, Dave! Nice to meet you!” if it’s someone new.

Remember, coding is all about reusability, so we are already ahead of the game by putting our greeting into a function. We just have to change it a little bit to do these other things we mentioned!

We will have to add another parameter to our greet() function. We will add a parameter called is_new, which can tell the function whether the person we are greeting is someone we know:

```
def greet(name, is_new):

print(f"Hello, {name}!")
```

Great!

All we need to do is add some logic to our function. We want to print a different greeting for the people we know than the one we print for the people we don’t know.

We can use our newly added is_new parameter to help us make this decision! So, if we don’t know the person, we can use a specific greeting:

```
def greet(name, is_new):

if(is_new):

print(f"Hello, {name}! Nice to meet you!")

else:

print(f"What's up, {name}? Nice to see you again!")
```

That’s it!

What Will You Build?

Congratulations, Python programmer! You have officially learned how to code in the Python language!

You should be proud!

Now you have the proper tools and knowledge for coding in Python, what will you build?

We have created a few games and have gone through some silly and fun activities, so those are just a starting point.

But there is so much more that you can do. How about trying some other games in the next section? The possibilities are endless. Just imagine it—then code it!

GAMES TO PRACTICE

Python Coding Game #1: Rock, Paper, Scissors

Rock, Paper, and Scissors. Are you familiar with these three words being spoken at the same time? I think you are. Rock, Paper, and Scissors is a popular game. My brother and I used to play rock, paper, scissors to decide who would get to choose the morning cartoons before school!

Rock, paper scissors is played between two people, both of them choosing one out of the three options. Then, according to certain rules, one of the two is declared the winner. We will discuss these rules further, and we will implement Rock, Paper, and Scissors in the python programming language.

Nervous? Well, no need to get nervous because it is not as difficult as it sounds! As I mentioned earlier, Python is one of the easiest programming languages, and it does not take much time to understand it.

So let's start!

Let's decompose Rock, Paper, Scissors

Before moving to the coding part, the first rule of programming and development is always to analyze and break down what we are going to implement. Breaking down our problems into smaller steps is called decomposition. Here, we are going to create a program for the game, Rock, Paper, and Scissors. Let's divide the problem into parts.

1. The players. Remember, we need two people in this game. For this program, we will have one user, and the other will be the computer. Both the user and computer are going to enter their inputs. Don't worry about the computer's input. It is going to be the most interesting part.

2. The rules. Next, we have to create example scenarios to decide who is the winner of each turn. Let's discuss these scenarios briefly:

– It is a tie when both the user and the computer makes the same choice.

– Rock wins over Scissors and loses to Paper

– Paper wins over Rock and loses to Scissors

– Scissors wins over Paper and loses to Rock

3. Exit strategy. There will always be an option to finish the game.

4. Who won? We also need to keep the count of points earned by both. When individuals play rock, paper, scissors, they often make it a best of 3 or best of 5 games!

So, now we know what we are going to do. Let’s get coding!

Coding a Python Game: Step-By-Step Instructions

I have created a fully working code for Rock, Paper, and Scissors. You can find the full code at the end of this post. Let’s discuss it step by step. I would suggest going through each line of code together in your classroom and discuss it as you go. In the end, you will have a fully working Rock, Paper, Scissors game that you can play against the computer!

This game was simply created using **variables, list, while loop, if-elif-else ladder, and a special randint method that we imported from random module**. We will review all of these beginner coding concepts below!

Step 1: Importing modules

```
from random import randint
```

The very first line of the code is: from random import randint.

What does this line mean? Read it, and you might get an idea. We are importing randint function from the random module using ‘from’ and ‘import’ keywords. The randint function has some special abilities that we are going to use in our program. We will discuss it later. Now that we have imported this function, we can use it in our code.

Step 2: Creating a list of available options

Earlier, we discussed data types. As I mentioned earlier, data types in python are user-friendly and very easy to understand. Let’s look at our next line of code for our first data structure, a list.

```
game  = ["Rock", "Paper", "Scissors"]
```

Observe the above line of code. We created a list data type and named it, ‘game’. Then we initialized it with the three available choices we have, Rock, Paper, and Scissors.

Step 3: Getting the computer to choose randomly

This is how the computer is making its choices randomly in this program.

```
computer = game[randint(0,2)]
```

We created a variable named computer, and this variable will hold the computer’s choice. We already have a list of choices, and we need to think of something so that the computer can choose one of them randomly. Randomly? Yes! This is where the randint function will come into play.

The randint function is used to generate a random value. The lower and upper limits are passed to this function, and the returned value is always between this range. Here, we will pass 0 and 2 because of our list. The game has three values. Don’t forget that 0 is considered the first value. Therefore 0, 1, and 2 make three options in total!

```
randint(0,2)
```

This piece of code can return 0, 1, or 2. It can be unique or the same. It just depends upon the python’s mood! Joking. It is totally random! We are going to use the returned number as the index of the list.

```
game[randint(0,2)]
```

We write the whole randint function inside the square brackets, so the returned random number can work as the index. Now, the computer’s choice is made and the variable, the computer holds it. We will use this line of code again later.

Step 4: Let’s define a few more variables before we start our game

For our Rock, Paper, Scissors game, there are a few other variables to consider.

```
players_point = 0
```

```
computers_point = 0
```

These two variables will keep the count of points for the user and computer. This is why they are initialized to zero.

```
go_on = True
```

Remember, I mentioned that along with options, the user could also finish the game. The above variable will end the game when user types ‘Finish’. The go_on variable is set to True, and it will be a condition of the while loop.

When the user enters ‘Finish’, the value of this variable will be changed to false, and the game will end.

Step 5: Creating a While loop

```
while(go_on):
```

In the parenthesis, go_on variable is used. Remember, the go_on variable has True as its value. This means the condition in while loop is true, and it will run until the condition is false. And when will the condition become false? Exactly when the go_on variable’s value is changed to false. Later on, we will tell our program to change the go_on variable to false when the user types “Finish.”

Step 6: Allowing User input

The first line of code within the while loop is:

```
    player = input("Rock, Paper or Scissors? or enter Finish to end!\n")
```

This is where we are asking the user for their choice. The four options are Rock, Paper, Scissors, and Finish. The user has to enter one of these, or the program will not go further. To get input from the user, you will use the input function.

The choice is stored in the player variable. The text written inside the function appears on the screen. Remember, we have to enter the values exactly as asked! This is uppercase and lowercase sensitive!

Step 7: Define the Scenarios

We have two variables: computer and player, holding choices of the computer and user, respectively.

Now it is time to decide the winner. We will do it by using the if-elif-else ladder. Have a look at the ladder we used in the program, and then we will discuss it.

```
if(player == 'Finish'):
    go_on = False
elif(player == computer):
    print("Tie!")
elif(player == "Rock"):
    if(computer == "Paper"):
        print("You lose!", computer, "covers", player)
        computers_point = computers_point + 1
    else:
        print("You win!", player, "smashes", computer)
        players_point = players_point + 1
elif(player == "Paper"):
    if(computer == "Scissors"):
        print("You lose!", computer, "cut", player)
```

```
            computers_point = computers_point + 1
        else:
            print("You win!", player, "covers", computer)
            players_point = players_point + 1
    elif(player == "Scissors"):
        if(computer == "Rock"):
            print("You lose...", computer, "smashes", player)
            computers_point = computers_point + 1
        else:
            print("You win!", player, "cut", computer)
            players_point = players_point + 1
    else:
        print("That's not a valid play. Check your spelling!")
```

Let’s understand how the ‘if-elif-else ladder’ works. Each if and elif has a condition part in the parenthesis. The program will enter that block whose condition is true. Once the true condition is found, it will ignore all other conditions. But what happens when none of the conditions is true? The execution goes into the else block. Observe the above code once more. The else block has no condition.

We will discuss all the conditions step by step.

1. The first condition checks whether the user entered ‘Finish’ or not. If this condition is true, the value of the go_on variable will be changed to false, and the program will end.

```
   if(player == 'Finish'):
        go_on = False
```

2. The second condition checks where the choice of both the user and computer is the same. If this condition is true, no points are awarded to anyone.

```
elif(player == computer):
    print("Tie!")
```

3. The following three conditions work according to the basic concepts of the game. The winner is decided, and points are awarded accordingly. Earlier, we initialized two variables with 0, computers_point and players_point. If the computer wins, computers_point variable is incremented by 1 and if the user wins, players_point variable is incremented by 1.

```
elif(player == "Rock"):
    if(computer == "Paper"):
        print("You lose!", computer, "covers", player)
        computers_point = computers_point + 1
    else:
        print("You win!", player, "smashes", computer)
        players_point = players_point + 1
  elif(player == "Paper"):
    if(computer == "Scissors"):
        print("You lose!", computer, "cut", player)
        computers_point = computers_point + 1
    else:
        print("You win!", player, "covers", computer)
        players_point = players_point + 1
  elif(player == "Scissors"):
    if(computer == "Rock"):
        print("You lose...", computer, "smashes", player)
        computers_point = computers_point + 1
    else:
        print("You win!", player, "cut", computer)
        players_point = players_point + 1
```

4. Last is the else block. There is no condition. It only displays a message when the user has entered an invalid choice.

```
    else:
        print("That's not a valid play. Check your spelling!")
```

Step 8: Allowing the Game to Continue

At the end of the while loop, we used the randint function once again to assign a choice for the computer.

```
    computer = game[randint(0,2)]
    print('********Next Turn********')
```

We have to repeat this even though we had a similar line of code at the beginning of our game. Remember, in every turn, the computer has to make a choice. This is why we write this line of code again.

Step 9: Displaying the final score

When the user ends the game by typing ‘Finish’, the program will display the final score. The final score is stored in the two variables we used earlier, computers_point and players_point.

```
print("********Final Points********")
print("Player: ", players_point)
print("Computer: ", computers_point)
```

Rock Paper Scissors – Full Code

There is the full code you will need to run a Rock, Paper, Scissors game in Python.

```
from random import randint

#List of options

game  = ["Rock", "Paper", "Scissors"]


#Assigning a random option to computer

computer = game[randint(0,2)]


#Keep count for points

players_point = 0

computers_point = 0

go_on = True


#Loop goes on until goOn is false

while(go_on):

    #Ask for user input

    player = input("Rock, Paper or Scissors? or enter Finish to end!\n")


    #Check for scenarios

    if(player == 'Finish'):

        go_on = False
```

```
elif(player == computer):
    print("Tie!")
elif(player == "Rock"):
    if(computer == "Paper"):
        print("You lose!", computer, "covers", player)
        computers_point = computers_point + 1
    else:
        print("You win!", player, "smashes", computer)
        players_point = players_point + 1
elif(player == "Paper"):
    if(computer == "Scissors"):
        print("You lose!", computer, "cut", player)
        computers_point = computers_point + 1
    else:
        print("You win!", player, "covers", computer)
        players_point = players_point + 1
elif(player == "Scissors"):
    if(computer == "Rock"):
        print("You lose...", computer, "smashes", player)
        computers_point = computers_point + 1
    else:
        print("You win!", player, "cut", computer)
        players_point = players_point + 1
    else:
        print("That's not a valid play. Check your spelling!")
    #Assigning a random option to computer
    computer = game[randint(0,2)]
    print('********Next Turn********')

#Printing final points
print("********Final Points********")
print("Player: ", playersPoint)
print("Computer: ", computers_point)
```

Let’s look at the final code in action!

A user can enter ‘Rock’, ‘Paper’, or ‘Scissors’ to play, or ‘finish’ to end the game. Then, the value is matched with the computer’s choice, and the points are distributed accordingly. When the game is ended, final points are displayed on the screen.

So, this is how we can create a simple Rock, Paper, and Scissors game in python.

Python Coding Game #2: Create a Magic 8 Ball Fortune Teller

We have built a simple Rock, Paper, and Scissors game in python. The game was simply created using variables, list, while loop, if-elif-else ladder, and a special randint method that we imported from random module. In this tutorial, we are going to create another game that mimics a class toy from my childhood: The Magic 8 ball – fortune teller. This will be fun!

We will use the basic concepts we learned last time and combine them with another new concept of programming language that is known as functions.

We will focus on functions in the magic 8 ball game. But first, let’s discuss a few advantages of using functions.

Advantages of functions in coding

1. Ease in program development.

2. Testing becomes easy.

3. Code reusability.

4. Better program readability.

5. Sharing of code becomes possible.

What is the Magic 8 Ball?

What is magic 8 ball? It is a toy that was developed in the 1950s. The user asks a question and a reply appears on the surface of the ball. If a user asks, “Am I going to die today?” the reply could be, “Certainly yes!”.

Don’t worry! You can play this game because there is no truth in its replies. It just randomly picks an answer from a set of answers and displays it on the screen whenever input is received.

So, it is basic programming. What comes in your mind when I say, “It just randomly picks an answer from a set of answers”? Yes!

You guessed it right, the randint method. Last time we similarly used the randint method to pick a choice for computer from the array containing three values – Rock, Paper, and Scissors. This time will do something similar to this. Let’s see what we are going to do in this game.

Let’s Decompose the Magic 8 Ball Game!

1. Again, there are two parties involved. One is the user, and the second is the magic 8 ball.

2. The user will type a question, and magic 8 ball will give a reply.

3. There will be a set of answers, and one answer per question will be picked from this set.

4. There will always be an option to finish the game.

Magic 8 Ball – Full Working Code

So here is the full working code of Magic 8 ball – fortune teller game in python. We will discuss this code step by step, but first, make sure you go through it properly.

```
#importing randint method

from random import randint

#List of answers

answers = ['Outlook good', 'Yes Signs point to yes', 'Reply hazy', 'try again', 'Ask again later', 'Better not tell you now','It is certain', 'It is decidedly so', 'Without a doubt', 'Yes – definitely', 'You may rely on it', 'As I see it, yes', 'Most likely', 'Cannot predict now', 'Concentrate and ask again', 'Dont count on it', 'My reply is no', 'My sources say no', 'Outlook not so good', 'Very doubtful']


print('Hello stranger!, I am the Magic 8 Ball')

print('**********')


#The magic 8 ball function

def Magic8Ball():

    print('Ask me a question.')

    input()


    #using randint method
    print (answers[randint(0, len(answers)-1)] )
    print('I hope that helped!')

    #calling Replay function
    choice = Replay()

    if(choice == 'Y'):
        #Calling Magic8ball function
        Magic8Ball()
    elif(choice == 'N'):
        return
    else:
        print('I did not understand! Please repeat.')
        #Calling Replay function
        Replay()


#Function for user's decision
def Replay():
    print ('Do you have another question? Enter Y if yes or N if no.')
    choice = input()

    #Returning user input
    return choice
```

```
#Calling Magic8ball function
Magic8Ball()
print('*****I hope you got your answers*****')
```

Importing the randint method

```
#importing randint method
from random import randint
```

Here, we imported the randint method from random module. We will use this method for choosing answers later.

List of answers

```
#List of answers
answers = ['Outlook good', 'Yes Signs point to yes', 'Reply hazy', 'try again',
'Ask again later', 'Better not tell you now','It is certain', 'It is decidedly so',
'Without a doubt', 'Yes – definitely', 'You may rely on it', 'As I see it, yes',
'Most likely', 'Cannot predict now', 'Concentrate and ask again', 'Dont count
on it', 'My reply is no', 'My sources say no', 'Outlook not so good', 'Very
doubtful']
```

This is quite a big list! There are 20 answers on this list. Magic 8 ball can choose any answer from this list. You can also personalize this to your own set of answers! Get creative!

Calling the Magic8Ball function

Furthermore, there is a simple printing part and two functions. These two functions won’t do anything until they are called. So let’s skip this part for the time being and jump directly to the ending part of the code.

Magic8Ball()

This is where everything begins. One of the two functions is named Magic8Ball, and here we are calling it. The program execution will skip the two function, and when this line get’s executed, the magic8ball function will get invoked. There are no parameters for the magic8ball function. So there is nothing passed in the parenthesis.

Working of Magic8Ball function

Now let’s understand the working of the Magic8Ball function.

```
def Magic8Ball():
    print('Ask me a question.')
    input()

    #using randint method
    print (answers[randint(0, len(answers)-1)] )
    print('I hope that helped!')

    #calling Replay function
    choice = Replay()

    if(choice == 'Y'):
        #Calling Magic8ball function
        Magic8Ball()
    elif(choice == 'N'):
        return
    else:
        print('I did not understand! Please repeat.')
        #Calling Replay function
        Replay()
```

Pay attention to the very first line of the above code – def Magic8Ball(). This is how functions are defined in python using the def keyword. First, we ask for the user's questions. Here, we are not storing the user's question anywhere because it does not matter what the user asks. This is the concept of the game.

Pay attention to the next line of code:

```
print(answers[randint(0, len(answers)-1)] )
```

This is the most important part of the game – Answer from Magic8Ball. So what is happening here? The randint method is used to get a random number.

```
randint(0, len(answers)-1)
```

Last time in the Rock, Paper, and Scissors game, we did something similar to this. There, we passed 0 and 2 in the randint method because there were only 3 values to choose from. But in this program, we have 20 or we can have 30 or as much as answers we want. So, instead of using a static value, we used a method that counts the length of the list.

```
len(answers)
```

This will return 20 because there are 20 values in the answers list. But, remember the index of a list always starts from 0, which means, the last element is on the 19th index. You can learn more about this method of counting in our basic Python Tutorial. That is why we subtracted 1 from the returned value. Now the randint method will return a random number from 0 to 19.

```
answers[randint(0, len(answers)-1)]
```

The randint method is used in the square brackets. This is how a random answer will appear.

The Replay() function

This is the part where the user decides, should the game continue or not? We have a separate function for this input – Replay().

```
def Replay():
    print ('Do you have another question? Enter Y if yes or N if no.')
    choice = input()


    #Returning user input
    return choice
```

As mentioned earlier, each function has a specific task. The magic8ball function asks the question and gives an answer. The replay function asks what does the user want to do next – play or quit. The user has to enter ‘Y’ for yes and ‘N’ for no.

The return keyword is used to return a value from a function. But we need to store this returned value. But wait a minute! where did we call this replay function? In the magic8ball function. Let’s go back there.

Calling the replay function in magic8ball function

In the magic8ball function, we call the Replay function and store the returned value in the choice variable.

```
choice = Replay()
```

The choice variable will be used next in the if-elif-else ladder.

The if-elif-else Ladder

```
if(choice == 'Y'):
        #Calling Magic8ball function
        Magic8Ball()
    elif(choice == 'N'):
        return
    else:
        print('I did not understand! Please repeat.')
        #Calling Replay function
        Replay()
```

Let’s understand the if-elif-else ladder step by step.

1. If the value of the choice variable is ‘Y’, the magic8ball function will be called again.

2. If the value of the choice variable is ‘N’, the program will exit the magic8ball function. Now, pay attention here.

```
elif(choice == 'N'):
        return
```

We only used the return keyword, but no value is returned with it. This is a way of exiting from a function.

3. If the value of the choice variable is anything else except ‘Y’ and ‘N’, The reply function will be called again for appropriate user input.

This is how we can build a simple Magic 8 ball – fortune teller game in python.

Let’s look at the final code in action!

When the code is running, the computer will start by asking the user a question. The user enters a question and the computer will randomly generate an answer, just like the toy magic 8 ball!

HAPPY CODING!!!