The Ultimate Guide To Python Programming For Beginners

Your 7-Day Gateway To Solve Problems & Express Creativity With Hands-On Exercises To Unlock Career Opportunities

Table of Contents

Preface
 For the Readers
  Writing Conventions
 Unveiling Python: From Its Origins to Global Prominence
 What is Python? A Brief History
 Why Should You Choose Python as Your First Programming Language
  Easy to Learn
  Easy to Read and Write
  Syntax Is Simple
  Abundance of Resources
  Open-Source and Free
  In-Demand
  Large and Supportive Community
 How Python Differs From Other Languages
  Python vs. JavaScript
  Python vs. Node.js
  Python vs. Java
  Python vs. Ruby
  Python vs. PHP
  Python vs. R
  Python vs. C++
  Python vs. C#
 Where Does Python Stand Today
  The Pros and Cons of Python
  Pros
  Cons
Introduction
 But Why Python?
Chapter 1: Python Fundamentals
 Getting Started

Before You Do—Installing Python the Smart Way
For Windows
For Mac
For Linux
Instal Python
Installing Python on Windows
Installing Python on Mac
Installing Python on Linux
Installing Your IDE
IDLE
PyCharm
Visual Studio Code
Sublime Text 3
Atom
Other IDEs and Code Editors
Your First Step to Code
Python Variables
How to Name Your Variables the Right Way
The Data Types in Python
Integers
Float
String
Boolean
List
Dictionary
Tuple
Set
Exercises
Exercise 1
Exercise 2
Chapter 2: Python Basic Operations and Input/Output
Basic Operations in Python
Arithmetic Operations
The Arithmetic Operators

- Addition
- Subtraction
- Multiplication
- Division
- Modulus
- Exponentiation Operator
- Floor Division
- String Concatenation
- Using the + Operator
- Using the * Operator
- Using the join() Method
- Using % Operator
- Using format() Function
- F-string Method
- Python User Input and Output
- Comments
- Exercises
- Exercise 1
- Exercise 2

Chapter 3: Control Structures and Data Collections

- Conditional Statements
- Comparison and Logical Operators
- Logical Operators
- Loops and Iterations
- For
- While
- Break
- Continue
- Pass
- Lists and Dictionaries
- List Operators
- Diving Deeper Into Dictionaries
- Exercises

Chapter 4: Python Functions and Modular Programming

Introduction to Functions
The Syntax of a Function
Parameters and Return Values
Functions With No Parameter
Scope and Variable Lifetime
Exercise
Simple Calculator
Chapter 5: Advanced Data Structures in Python
Tuples
Accessing Items Inside a Tuple
Sets
Modifying Sets
List Comprehension in Python
Set Operations
Set Union
Set Intersection
Set Difference
Set Symmetric Difference
Exercise
Chapter 6: The Basics of Object-Oriented Programming (OOP) Using Python
Basics of Python OOP
Classes and Objects
Objects
Creating Class Methods
Inheritance and Polymorphism
Extending Classes
Method Overriding
Exercise
Chapter 7: Error Handling and Debugging in Python
What Is Error Handling in Programming?
What Is Debugging?
Understanding Exceptions
Types of Exceptions

NameError
TypeError
ValueError
IndexError
IndentationError
ZeroDivisionError
ImportError / ModuleNotFoundError
AttributeError
KeyError
Python Debugging Techniques
Using the Print Statements
Using Python Debugger
Using the Logging Module
The Try and Except Statements
Recommended Workflow of Debugging for Beginners
Exercise
Chapter 8: Python File Operations and Data Storage
Reading and Writing Files
File Access Modes on Python
Opening Files
Closing Files
Writing
Reading
Appending
Working With CSV Data in Python
Opening a CSV File
File Modes
Closing a CSV File
Reading a CSV File
Writing on CSV Files
Read CSV File Into Dictionaries
Writing CSV File From Dictionary
Handling Errors
Working With JSON Data in Python

JSON Serialization
Deserialization
Exercise
Bonus Challenge
Chapter 9: Fundamentals of Python Web Development
Introduction to Python Web Development
The Basics of Framework—Flask
Step 1—Installing the Virtual Environment
Step 2—Creating an Environment
Step 3—Activating the Environment
Step 4—Installing Flask
Step 5—Testing the Development Environment
Building a Simple Python Web App
Routing
Flask HTTP Methods
Using Templates
Exercise
Chapter 10: Introduction to Python Game Development
Project Overview
Preparing for the Project
Pros
Cons
The Right Option
PyGame Tutorial
Import and Initialize
Create a Game Window
Updating Games Using Loops
Drawing Graphics
Creating a Working Game: Circle Escape Challenge
Importing Modules and Initializing Pygame
Setting Up the Display Window and Display Variables
Define Colors, Shapes and Player variables
Game State Control
Create the fonts for the Score and Instructions

Display Functions
Game Loop
Full Code - Circle Escape Challenge
Code Challenge
Conclusion

Preface

For the Readers

Hey there, fellow Python enthusiast! In the pages ahead, we're about to dive deep into the Python universe - its history, its quirks, and how it stacks up against other programming languages. But here's the deal: if you're raring to start coding right this moment, you're in the right place!

The history stuff is awesome, no doubt about it. Python's got a fascinating story, and you'll want to soak it all in eventually. But, if you're itching to get hands-on with Python right now, flip on over to the Introduction section. There, we'll walk you through the process of installing Python and getting your coding groove on.

Think of the history as the 'bonus content' in this Python saga. It's like the deleted scenes on a DVD - cool, but not essential for your main feature experience. So, let's kickstart your coding adventure first, and you can always circle back to the history whenever you're in the mood for a bit of programming nostalgia.

The Python history is like a trusty old friend - it'll be right here waiting whenever you're ready to catch up. So, dive into the Introduction, and let's fire up your Python journey!

Before you move on, the exercise files from the book can be downloaded at the following link: https://www.dropbox.com/scl/fo/9wu4uh8oeccbagdxtzkjv/h?rlkey=pbd0uwdtl2i52uhx9yih6gikm&dl=0

Writing Conventions

Code samples are presented in a special monospaced font with a shaded background to set them apart from regular text. This formatting is designed to make the code stand out and easy to distinguish within the text.

Output examples, which showcase the results of executing code, are also presented in a monospaced font without a shaded background to make them

readily identifiable

- Command line instructions, which guide you through tasks in a terminal or command prompt, are usually displayed in a bullet point. This helps you differentiate them from regular text and code

File names are typically displayed in italics. This style indicates that the text represents a suggested file name you might encounter in your programming journey.

NOTE: E-Readers will not always display these differences. In our testing, code does highlight with a different background color. Other text such as variable names, output and even command line instructions may not have the same format as shown in the print version of the book.

Unveiling Python: From Its Origins to Global Prominence

Learning any language is an exciting prospect. While most people learn languages that they rarely use, programming languages are slightly different. Programming is about being creative, thinking out of the box, and doing something different. It grants you the power to create something, and the great feeling of knowing you can do that is just incredible.

Ever since Python was invented, it has gone on to attract the attention of the world, and the programming community at large, and it continues to draw a lot of attention. While its simplicity is the leading reason why everyone wants to learn Python, the rise of AI and machine learning are two other driving reasons.

If that wasn't enough, almost every leading business school today teaches Python and some of its libraries to business graduates. You might wonder how the two relate, and you wouldn't be wrong to assume that a business graduate has nothing to do with programming. However, Python proves to be a vital piece in their

academic journey. Using Python, business analysts can filter, refine, and extract data. They can work up charts, export data into files, and simplify everything for them and their clients.

Python continues to make a case for itself, and it is safe to assume that Python is here to stay. With all the technological advancements, such as AI and machine learning, knowing Python is becoming a skill garnering a lot of attention and demand. Therefore, let's understand exactly what Python is, where it came from, and why it has become so popular.

What is Python? A Brief History

For years, there have been many programming languages that have been created for a variety of purposes. Each of them boasts an advantage while also lacking in some departments, and it's easy to see why that was the case.

The first language, invented by Ada Lovelace, surfaced in 1843. She created the entire language on paper because computers had not yet been invented. Her programming language was best described as a machine algorithm.

After a century, between the years 1944–45, the first actual programming language was invented by Konrad Zuse. This programming language was called Plankalkül (better known as Plain Calculus). What this language did was to allow the creation of procedures. Procedures were chunks of recallable code that could be used to carry out various operations. Sure enough, it caught on.

Around 1949, just a few years after Zuse's language, another language made itself known to the world: the assembly language. For those who may not know, programming languages have levels, high and low. The low-level languages are generally system-level programming. These are difficult to read and, for someone who has never programmed before, almost impossible to comprehend. Low-level languages create the basic structures for software that eventually interact with hardware.

The assembly language remains the first low-level language, was popularly used, and is still under use today. Of course, this was hard to learn, meaning that it was only natural someone would come up with something slightly better and more understandable. Besides, a low-level language could only carry out limited operations.

During the same year, Shortcode was also introduced. Unlike the assembly language, Shortcode was the first high-level programming language that John McCauley invented. A high-level language is what some of the most popular programming languages are, such as Python. You can use high-level languages to carry out various tasks, operations, and more.

Three years later, in 1952, Alick Glennie developed Autocode, a language he created for use on the Mark 1 computer at the University of Manchester. What truly made Autocode a hit was that it was the first-ever compiled language. Unlike any other language before it, this was the first language ever where a compiler would translate the code into binary code, a series of zeros and ones that the computer understands, without needing to do that manually. This saved a lot of time and allowed programmers to do so much more than before.

Next came the FORTRAN in 1957, which stands for Formula Translation. It was created by John Backus, and it remains the oldest high-level programming language still in use today. This programming language could easily perform scientific, statistical, and mathematical operations.

The world of programming language was now populating faster than ever before. Every bright mind was developing some new and ingenious way to compute and create programs. In the ten years since FORTRAN was invented, four additional languages were invented. These were:

1. ALGOL—Algorithmic language, a joint venture between American and European computer scientists. This language reserves the honor of laying the foundation for some of the most redefining and successful programming languages

ever created, including C, Pascal, C++, and the popularly used Java.

2. LISP—List processor, a language invented by John McCarthy within the walls of Massachusetts Institute of Technology (MIT). This was the first language intended purely for the use of artificial intelligence and is still in use today. Even though it is somewhat complex, it is seen as a great alternative to Python and Ruby. Some popular companies that continue to use LISP include Boeing, Acceleration, Genworks, and more.
3. COBOL—Common Business Oriented Language. As the name implies, this programming language was created solely for processing transactions and is the language of choice for many credit card processors, telephones, ATMs, and even traffic signals. Dr. Grace Murray Hopper was the mind behind this revolutionary language. This language remains the go-to language of major financial and banking institutions and even the gaming industry.
4. BASIC—Beginner's All-purpose Symbolic Instruction Code. Quite a mouthful. However, this is where everything changed. A group of students at Dartmouth College created this programming language. The goal was to create a language for students who weren't the strongest at mathematics or computers. This language, however, went on to catch the attention of one bright mind named Bill Gates. Using this language, Gates and his partner Paul Allen created the world's most popular graphic-based operating system, Windows, which cemented Microsoft's authority in the world of computers.

After Microsoft's massive success at the start, the race to find better programming languages took a new turn. From there, the world went on to welcome PASCAL, a language that was quickly favored by another tech company named Apple. The language was easier to use, offered more flexibility, and was something Apple preferred

using for all its projects.

In 1972, Smalltalk made its debut. While this wasn't such a phenomenal hit, it was still fascinating that you could modify code on the fly. Many of the programming languages of today, such as Python, owe their success to Smalltalk. Even today, companies like Logitech and Leafly continue to use Smalltalk for their tech stacks.

In the same year, the world was also greeted by arguably the most significant programming language phenomenon of its kind: C. Dennis Ritchie developed it at the prestigious Bell Telephone Laboratories. He developed this language to be used with a Unix operating system. By now, A and B, two more programming languages, already existed, but they were not even close to being as powerful and effective as C.

C gave the world an opportunity that it so desperately needed. Because of C, many other off-shoots or derivatives came to life. These included:

1. C#
2. Java
3. Perl
4. Python
5. PHP
6. JavaScript

It was a huge success and, believe it or not, continues to impress many tech giants even today. People at Meta (formerly Facebook) and Google continue to use C. Even Apple switched from PASCAL to C.

With all of these programming languages emerging, a gap was created within the market. With so much data to process, people needed a way to access, modify, alter, or delete databases. While this could be done by manually going into the directories, opening the file, modifying entries, and so on, it was not the most efficient way. Therefore, the same year, IBM researchers Donald Chamberlain and

Raymon Boyce developed another programming language that specialized in database modification and management. They called it SEQUEL, which later went on to become SQL.

We cannot even start to name the number of companies that continue to use SQL today. The fact is that it became an instant hit because it addressed a significant need within the market. Thanks to SQL, database management became extremely easy and efficient. However, after 1972, there was a noticeable downward trend in the number of programming languages that emerged. It seemed the entire tech world had hit the ceiling and was now losing interest.

The next programming language took almost a decade before it was created. In 1981, Ada was created for a US Department of Defense project. It was a structured, imperative, statically typed, object-oriented, high-level programming language that became the worldwide choice language for air traffic management systems. Even space projects use Ada to carry out complex computations effectively and efficiently.

Two years later, C++ was introduced as a modified version of C. C++ is an extension to C for all intents and purposes as it offers additional classes, functions, and other useful templates. The same year also saw Objective-C emerge, a language used to design and create the software behind Apple's wonderful iOS and macOS.

A few years later, Perl made its mark, too. It was initially intended to be a high-level programming language for text editing, but these days, it is mainly used for CGI, system administration, and some database applications.

By now, computers had become more advanced. Older languages were too system-resource-heavy or too inefficient. Some of the languages were disappearing, and some were far too hard to understand. This is where Guido Van Rossum developed a project that would address those issues through a single high-level language. It would draw its inspiration from all the popular languages, such as C and others, to create a far simpler syntax that almost anyone could understand. Van Rossum came up with what

now is the second-most popular programming language on earth and decided to name it after his favorite British comedy troupe, Monty Python. In 1991, the world greeted Python: a language for everybody.

Today, Python is used by Google, Spotify, Yahoo, Pinterest, Quora, Instagram, Uber, and Amazon; the list goes on and on. They all use Python for some solid reasons: it's easy to type, it's highly memory efficient, it's fast, reliable, and it is very flexible.

Think about Python; it is a more mature and robust version of all the popular programming languages. To make things even more attractive for programmers, it is object-oriented, something we will learn more about later.

Python was meant to be a language that could be used for everything and by everyone. Arguably, this is one of those few languages that achieved its goal. From the examples we highlighted above, it shows that Python can be used in a variety of ways. It is the perfect language to learn and master if you aim to seek a career in:

- Web development
- Cybersecurity
- Software development
- Game development
- Data Analytics

Speaking of data analytics, there is no denying that this is the era of data analytics and data sciences. Anyone who aims to utilize this time and make an impact in the world, find a great paying job, or even set up their consultancy would need to learn Python. That is because Python sits at the center of all data sciences. Python is the go-to language for data scientists because of its flexibility. Not only is it a great language to use for your projects, but it is also able to:

- Connect with databases
- Carry out complex mathematical operations

- Work on servers with ease
- Create workflows
- Access, read, and even modify files
- Handle big data

Why Should You Choose Python as Your First Programming Language

That is a very valid question. Of all the programming languages that you can learn from, why Python? There must be something truly unique and rewarding about Python that makes it that much more appealing. Well, there are many, as we shall now see.

Easy to Learn

Python is considered the most beginner-friendly programming language on earth. Whether you're someone migrating from one programming language to another or someone who has never worked with any programming language, it is a great language to start with. Why? Let me show you an example.

We'll use three different programming languages to see and compare how easy or otherwise it is to read the code. We will be using C, C++, and Python. For this example, we'll ask the programming language to print "Hello World." It is a simple program and requires no real skill.

C

```
// C showing
// how to print Hello World
// on screen

#include <stdio.h>

int main()
{
    printf("Hello World");
    return 0;
```

```
}
```

"Oh My...." I know. Hard, isn't it? Let's see how C++ fares.

C++

```
// C++
// code showing how to
// print Hello World

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World";
    return 0;
}
```

"That's just worse." Okay, let's find out how Python differs.

Python

```
print("Hello World")
```

"Wait... is that it?" That is quite literally how you write in a code that asks Python to print the words "Hello World" and write it so that you can practically read it like English. Here is another example to show just how simple things are with Python. This time, we will ask our user to input two values as first number and second number. We will then ask the program to add the two and print the result.

C

```
#include <stdio.h>

int main()
{
    int a, b, c;
    printf("first number: ");
    scanf("%d", &a);

    printf("second number: ");
```

```
    scanf("%d", &b);

    c = a + b;

    printf("Hello World\n%d + %d = %d", a, b, c);

    return 0;
}
```

That's... complicated. Let's see if C++ is any easier.

C++

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c;
    cout << "first number: ";
    cin >> a;

    cout << endl;
    cout << "second number: ";
    cin >> b;

    cout << endl;
    c = a + b;

    cout << "Hello World" << endl;
    cout << a << "+" << b << "=" << c;

    return 0;
}
```

"That makes absolutely no sense." Exactly. For someone who has never coded before, that's just gibberish. However, let's not be quick and see if Python is as easy as they claim.

Python

```
a = int(input("first number: "))
b = int(input("second number: "))
```

```
c = a + b
```

```
print("Hello World")
print(a, "+", b, "=", c)
```

"Well, it kinda makes some sense." It certainly does. Typing code in Python is straightforward because it is the only programming language closest to English, making reading codes in Python much easier. While you may still not get some of that code right away, don't worry. That's only because we are not used to understanding the syntax. We will do that later on in the book. Once done, you can revisit this code and see if it makes sense. You'd be surprised at just how easy of a code this turns out to be.

Easy to Read and Write

From the previous example, it is clear that Python is significantly easier to write and certainly easier to read. It may use some odd quotation marks at times, but even that is understandable enough to the untrained eye.

You cannot deny Python is a thousand times easier to read and make sense of as compared to:

```
printf("Hello World\n%d + %d = %d", a, b, c);
```

That is the beauty of Python. That is precisely what Guido Van Rossum had in mind: Simplicity. It is this simplicity that has gone on to attract people from all walks of life to learn and adopt Python as their go-to language.

While other languages made before Python were aimed to be "general purposes" languages, Python is one that truly lives up to that title. It virtually allows even an untrained professional to read and make sense of the code.

What truly makes it unique is how it is written. You do not have to worry about special symbols at the end of the line, such as in the examples above. You type your line, hit the "Enter/Return" key, and move on to the following line.

Syntax Is Simple

Syntax is the basic structure, the skeletal system of a language. Syntax determines how you write code in any given programming language.

In the above examples we borrowed from C and C++, you can see that they had semi-colons at the very end of every line. These serve as something called terminators. Terminators tell the compiler and the computer, "Hey, this is where the line ends."

Python, on the other hand, does not require any of that. The compiler does not need explicit instructions to determine where the line ends. "Don't worry! I've got you." That's probably what the compiler says because it is smart enough to know when the line ends, making our jobs easier.

It isn't just that, either. As we go deeper into this book, you will learn how simple the entire syntax is. From declaring a variable to recalling values, from taking inputs from users to passing them through lists and dictionaries, everything is done in an easy-to-read-and-write way.

Abundance of Resources

Arguably, Python's most significant selling point is its plethora of resources for all levels of programming. Whether you're a beginner or a veteran programmer, Python offers resources to help you quickly understand and overcome all challenges.

The resources of Python are available for free. You can also request fellow programmers from around the world to help you with custom problems you may be facing with your code. There are places like Stack Overflow, Reddit, GitHub, and many others where you can post your questions and find out exactly what you're looking for.

If you do not find the answer, post your code and request assistance. Someone within the active community will offer their solution, which generally does the trick for everyone.

Open-Source and Free

Another unique selling point for Python is that it is 100% free of cost. Not just that, it is open-source, meaning anyone in the world

can modify the original code and add their contribution to improve the language functionality.

Open-source software and languages have massive communities that are constantly helping languages, users, and the software improve in terms of functionality and ease of use.

An open-source language allows you to stay up-to-date and secure because you're not the only one using Python. Many large organizations also rely on Python to make things work. This means that the language is constantly under supervision. It also means that there will always be programmers constantly working on finding bugs, fixing them, updating the language, and ensuring that it offers efficiency and security.

In-Demand

Python is relatively young, especially when compared to other languages like C and BASIC. It is a language that has only recently seen a massive spike in terms of demand. This is because large corporations and organizations like Google use Python to make things work.

Large and Supportive Community

We touched on that before, but did you know the community is very supportive? Many new programmers feel intimidated by posting a basic question and then find themselves at the receiving end of giggles and laughter. That is not the case with the Python community.

The Python community is caring and respectful and fully understands the needs and requirements of new programmers. Whether you're posting on some official Python forum or places like Stack Overflow, you're bound to be greeted and assisted in the kindest ways possible. You can ask for an explanation of confusing codes. You can ask fellow programmers to write code doing a certain thing and even post your projects to seek reviews and optimization. The possibilities are endless, and the community members are always on hand to assist you through your endeavors.

How Python Differs From Other Languages

Earlier, we looked into how you write code using C, C++, and Python, but those aren't the only languages in question. The fact is that there are many other languages out there, some even more popular, that people continue to use. Therefore, it is only fair to compare Python with those languages and find out what makes Python the language of choice.

Python vs. JavaScript

As it turns out, JavaScript is the most popular programming language, but this will soon change as Python overtakes that crown. This, therefore, poses a bit of a problem. Comparing these two giants of the programming world isn't that easy.

Both of these languages offer essential features, and both of them handle fundamental aspects. Think about these languages like brushes. Some artists may prefer using a particular type of brush, while others may prefer an entirely different one.

That said, programmers don't use a single language to handle the entire project. A good software, for example, may have more than one language. A part of the software may use JavaScript to load the elements of the app's home screen, while the moment you log in, Python may come in to handle integrating your progress with servers.

If you aim to do web apps and work a lot as a front-end developer, JavaScript is your go-to language. You stick to Python if your operation involves a lot of CPU-intensive operations. Let's say you're aiming to do a project on machine learning where you need to train a model to solve a problem most effectively; that's where you'll need Python. If you want to advertise your project on a website and showcase its power, you'll need JavaScript along with HTML and CSS to do that for you.

Therefore, to simplify things, the key differences are:

1. JavaScript is better if you aim to work on the front end.

2. Python is great for all back-end and server-side operations.
3. JavaScript is faster than Python.
4. Python saves time in writing code.
5. Python is significantly better for data analytics, artificial intelligence (AI), and machine learning when compared to JavaScript.
6. Python is much easier to manage, understand, and modify than JavaScript.

Python vs. Node.js

Node.js is another programming language primarily used for web application development7. However, unlike JavaScript, which is complicated compared to Python, Node.js is significantly more advanced to understand, let alone work with.

Node.js, an advanced programming language, has some advantages over Python. These include:

1. A less opinionated ecosystem—Most of the packages of Node.js are very simple, and most of these are designed for a particular purpose. This simplifies the entire project as programmers only use what they need to. However, it also poses a challenge as programmers will need to work out beforehand what packages they need throughout the project, which can often take time.
2. Coding in JavaScript—Using Node.js, you can practically code in both the front and back end using JavaScript. Remember, Node.js is essentially a higher level of JavaScript, not an entirely different language.
3. Large community—Like Python, Node.js also has a huge community willing to help, assist, and guide Node.js users and developers.

With that said, it is worth knowing that you'll need to know JavaScript properly first to be able to use Node.js. Without that,

learning Node.js can be extremely tough and tricky, as most things will not make sense.

It is also worth noting that Node.js struggles to handle and execute multiple tasks simultaneously. Then, there is the fact that if your code isn't written with quality in mind, you may end up ruining your app's performance, and it will work slowly.

Make no mistake, as Python can also run into the same problem. However, some Python frameworks, such as Django, allow users to take advantage of the built-in solutions. These solutions ensure that the code is well-optimized and that operations continue as smoothly as possible. That's yet another reason why Python is the go-to language for all.

Python vs. Java

Java and JavaScript are two different programming languages. They may sound like they are related, but they aren't. JavaScript is used for front-end development, whereas Java is used primarily in application development.

Python is a language that is interpreted and dynamically typed. On the other hand, Java is a compiled language that is statically typed.

On one hand, Python code requires no compilation before running the program. On the other hand, we have Java, which requires the code to be compiled into binary code that the machine can understand before executing it.

Then, there's the fact that Java is yet another advanced language. It has steep learning curves, and ideally, you have previous programming knowledge to fully understand the workings of the language. It takes a lot of time to master the language, too.

Regarding stability, something that matters a lot to big corporations, Java takes the edge ever so slightly. This is because it has a lot of libraries that can help keep the software stable and intact. Make no mistake, though, as Python boasts an ever-growing library that can outperform Java and other languages in many other aspects, especially Financial Technology.

Stability aside, it all boils down to three things:

1. Speed to develop
2. Resources for the language
3. Trends in the industry

As it turns out, Java can take a long time to develop an application and bring it to its Most Viable Product (MVP) stage. Also, Java projects will often require more developers on board, which can easily offset the budget and the project's overall cost. Python, on the other hand, doesn't suffer from any of that.

Resource-wise, and I do not mean the resources you need to learn from, Java will demand a lot of time, energy, and money before fully satisfying you with its results. Python, on the other hand, is significantly less expensive. If you develop a project, Python will help you save a fortune.

Finally, the trend. Python easily takes the cake here because of its ever-growing popularity, incredible ease of use, and versatility. Java is far from being the trending programming language. This means that if you learn Python now, you stand a good chance of finding some lucrative opportunities ahead.

Python vs. Ruby

Ruby is younger than Python but is still a great language to learn, as both languages allow experts and developers to find very similar results, especially if they are developing web applications.

Ruby is primarily aimed at web development, and, unlike Python, it is usually restricted to that domain most of the time. On the other hand, Python can be used in various situations.

Regarding popularity, both of these languages are polar opposites of each other. Ruby is seen as a language that has a declining number of users, whereas Python is a language that has skyrocketed in popularity.

Both languages can handle server-side and scripting. Both languages are dynamically typed, can be used cross-platform, and are both

general-purpose languages. The differences, on the other hand, are apparent.

1. Popularity—There is no denying that Python is significantly more popular than any other language, including Ruby.
2. Philosophy—Ruby and Python both approach problems in their own unique ways. Python offers a more straightforward and somewhat singular solution, while Ruby offers multiple ways to solve the same problem.
3. Flexibility—Python's syntax is straightforward, easy to understand, and uniform. Ruby's syntax is flexible. This is why Ruby users need to find creative ways to solve problems.

Python vs. PHP

PHP is another open-source language that has become quite popular with web developers. Soon, it became the default language of almost every web server technology out there. Around 80% of all the web servers in existence today use PHP.

PHP can be used for other purposes, but that is rarely the case. However, quite like Python, PHP is also very easy to learn. With that said, it goes into very advanced territory, something beginners may struggle to cope with.

When comparing the two, it is easy to see that Python is significantly more versatile, offers a better and easier structure, and has immense popularity.

On the other hand, PHP has many features right out of the box. Even though Python can match that, the fact is that you'll need to rely on importing additional libraries, and that can be a bit of a fuss.

Installing Python is easy, but installing PHP is significantly easier in almost every case except if you're using Linux.

Python vs. R

R is yet another programming language for those who may not know. R and Python are used for data sciences, a field that has garnered much attention recently. Both of these languages are flexible, and both of them are open-source. Both languages come with significant community support, bringing their unique benefits into the mix.

R has 12,000 packages that make it exceptional for reporting. Whatever analysis you aim to carry out, you'll always be able to find a package that does that for you. Then, there's the fact that the output R generates is un-matchable by any other language, thanks to its vast variety of tools.

On the other hand, you go for Python if your project involves machine learning, artificial intelligence, and server-side work. Python, too, can handle almost everything that R can, but what truly makes it stand out is its code's simplicity and strength. While R is used for smaller projects, Python is what people use for large-scale projects.

Python vs. C++

C++ is a high-level and general-purpose programming language. It is also object-oriented. An Object-Oriented Programming (OOP) language uses objects to carry out programming. Through object-oriented programming, you can use real-world entities such as hiding, inheritance, and polymorphism, although these may be too advanced to look into at the moment.

In fairness, Python is also an OOP language, meaning it can do almost everything C++ or any other OOP language can do. However, learning C++ can be quite challenging as its syntax isn't the easiest to understand right away.

Regarding speed, C++ takes the edge because it has a superior memory management system compared to Python. However, that also means that when you use up memory, it must be de-allocated, as failure to do so in C++ will lead to memory leaks and crashes. C++ does not have a built-in garbage collection or dynamic memory management system, meaning you will need to work out when and

where your variables are no longer needed. Once you do that, you must de-allocate these variables and free up memory.

Python has a built-in garbage collecting and dynamic memory management system. This means you do not need to explicitly work out when or where a variable is no longer required. It will be able to work that out for you and ensure that it manages memory utilization at its optimum level automatically.

It is also worth noting that Python requires an interpreter to be active during code compilation. For that very reason, it is called an interpreted language. C++, on the other hand, is a pre-compiled language, and that means it does not require you to have an interpreter.

C++ has a very complex syntax, and that makes reading code quite tricky. Python, on the other hand, offers a significantly simpler syntax. This is why most courses offer Python as the first language to learn. It is a perfect way to get a feel of what it's like to write code, make algorithms, etc.

Finally, when it comes to declaring variables, in C++, you will need to declare what type of variable you want to create, followed by the name of the variable and then the value that you wish to assign to the variable.

For Python, however, things are much more manageable. You type in a name and assign a value. The system will work out what kind of variable you will use.

Python vs. C#

Also known as C Sharp, C# is a popular language that is a hit with the game development community. Just like Python, it is an object-oriented programming language. However, it is a language that is native to Microsoft. It is the optimal programming language to build some products using Microsoft-based platforms.

With that said, if you are already familiar with any variation of C, you'll be right at home with C#. If you've never programmed before, C# may be slightly more challenging.

Where Does Python Stand Today

Python has been around since 1991, or 32 years (when writing this book). During these years, it has become the most popular language. Yes, it is straightforward to use, extremely easy to understand, and very versatile, but what truly makes it stand out is the number of fields and applications where it shines.

At the time of writing this book, Python is used widely in the following sectors and industries:

1. Data sciences
2. Research
3. Game development
4. Software engineering
5. Marketing and advertising
6. Ethical hacking and cybersecurity
7. Data analysis
8. Business reporting
9. Data extraction and filing
10. Web development
11. Machine learning
12. Artificial intelligence
13. Biology and bioinformatics
14. Computer vision
15. Image processing
16. Neuroscience
17. Psychology

Some of the major companies that continue to use Python include, and are not limited to:

1. Google
2. Intel

3. YouTube
4. PayPal
5. Facebook (Now known as Meta)
6. Amazon
7. Spotify
8. Dropbox
9. Netflix
10. Disqus
11. Pinterest
12. Uber
13. Reddit
14. Instagram
15. Instacart
16. IBM
17. JP Morgan Chase
18. Stripe

The Pros and Cons of Python

Of course, Python isn't the holy grail of programming languages, but it is certainly one that comes closest to claiming that title. While Python is a very, very practical language, it does have some drawbacks and shortcomings that you must take into consideration. This is important because it will help you understand whether you can take on a particular project in the future while using Python or not.

Pros

- Easy to read.
- Easy to write.
- Easy to understand.
- Massive learning resources.
- Massive libraries for additional functionality.

- Large, active, and very supportive community.
- Robust.
- Secure.
- Easy to maintain.
- Wide variety of functionality and use cases.
- Has a built-in memory management system.

Cons

- Not as fast as C or Java.
- Memory intensive.
- Does not support multithreading.
- Can be misused or overly used.

With all said and done, it all comes down to one question: should you use Python? The answer to that is "Yes!" You should choose and learn Python as your first, third, or even the 10th programming language. The fact is that Python's popularity is on the rise, and almost every other major firm is switching towards Python. Then, there is the fact that Python can be used for various reasons and projects, which means that you'll probably end up using Python for the project you've been considering.

Learning Python is significantly more beneficial if the entire emergence of artificial intelligence and data sciences inspires you. Python is the go-to language for all these projects, so you will also need to learn Python to make the most of the opportunity. Besides, you may not be willing to spend 3 to 4 years learning a single language and may be in a bit of a hurry to get into the world of programming. Python, as it happens to be, is your quickest entry into programming. It's easy to learn, doesn't take much time, and is very easy to use.

Introduction

Things have changed for everyone in recent years, especially after the lockdown. There was a time when you'd work your day job, and things would be pretty easy to manage. As time went on, things changed. The lockdown only sped up the process.

Today, many professionals realize they may not be making enough money to lead a good life. They may not save fast enough to ensure they have enough for retirement. Besides, the need for career switching and side gigs is higher. It is safe to say that if you are working a daytime job, you'd want to change things around to find better opportunities for yourself and your future.

There was a time when you'd be required to go through years of study and have a proper academic background to find jobs. Today, all you need are skills, making things easier for us all.

Not long ago, a stay-at-home mom had a four-year-old kid to look after and raise. At the same time, she was enrolled in a Masters degree program in media studies. She aimed to ensure that she would then be able to find better opportunities to ensure the household was taken care of easily.

During this time, she decided to learn coding. This was partly due to her studies and partly because everyone talked about how coding experts made good money. Besides, she was a freelancer and was more used to working from home. This allowed her to spend enough time with her child and still be able to study at the same time. Little did she know that she'd learn a skill that would allow her to attract significant opportunities. To make things more interesting, she chose to learn and work with Python.

Today, she's a successful freelancer, a wonderful mom, and a Masters degree holder. She uses her newly acquired skills to assist clients and customers worldwide through her mastery of the language and its application. Needless to say, it worked out well for her.

That is the power of learning a new skill, more specifically, a programming language. While there are so many programming languages, most may require a lot of time, months, or even years of practice, and some may only be learned through degree programs. That's not the case with Python.

Python has gone on to become a go-to language for programmers and experts all around the world, second only to JavaScript. Python is known for being one of the easiest programming languages to learn. You do not need any specialized education, prior programming experience or knowledge, or any of that. You only need a system to work with, time, and interest.

What truly makes this programming language so popular is its use and application. Whenever you learn a language, you must do so with the end results in mind. You must know what that language will allow you to do. Just like Python, there are many other programming languages. However, almost every programming language offers different functionality, versatility, and efficiency.

Some programming languages are great for web applications. Some are renowned for mobile app development. Some are great for those who wish to work with games, and some are almost exclusive to things like web development, styling, and more. So, where does Python stand, you ask?

Python is one of the most popular languages in existence today. When writing this book, it is the second most popular language, just behind JavaScript. What truly makes Python so popular is its usefulness.

Python is the go-to programming language for:

- Data scientists,
- Data mining,
- Web applications,
- Games,
- Machine learning,

- Artificial intelligence,
- Deep learning,
- Hacking and cybersecurity.

If Python was a person, that's one impressive resume.

"But how long does it take for me to learn Python?"

Frankly, Python is unlike any other language out there. It is the most straightforward language to learn partly because it does not involve annoying rules that other languages have but mostly because it looks like English. It feels like you are reading out of a book instead of a cryptic message.

"I get it, but why should I choose this book if Python is so easy?"

Great question. You see, beginners have no idea how to get started properly and how to structure their learning. They may start with loops, but they should start by understanding the syntax. Some go on to learn about something called dictionaries without actually understanding lists or tuples.

Therefore, to ensure you learn Python in the best way possible, we have put together this book. This book offers a step-by-step guide to help you cover all the bases and learn Python the way it's meant to be learned. Not just that, we will go through many examples and exercises to ensure all concepts are reinforced and appropriately understood.

You do not need any prior programming experience to learn Python; you'll see why that's the case. Whether you're aiming to pick up a new skill, explore a new hobby, or switch careers, Python is a great way to get started in the world of programming. To truly make the most of this book, be sure to practice along. Programming is all about practicing. Be sure to practice all the basics as you go along. The more you do, the easier it will become for you to advance your skills.

But Why Python?

It is a valid question. We have heard many beginners and experienced programmers asking what makes Python unique. There must be a reason why it is ranked as the second most popular language. The way things are going, it is soon expected to take over the crown and become the most popular language of all time.

Well, there are many, many reasons. Keeping the technical features aside, the most defining feature of the language is its simplicity. Throughout the book, we will use a variety of code snippets. Throughout, you will notice that Python looks almost identical to English. It is like you can literally read it as sentences. If not, you'll still be able to understand the code easily.

How easy a programming language is does not dictate how functional or otherwise a language can be. Python, despite the easy learning curve, is a very powerful language. How? Think about what's making rounds these days that everyone's talking about - Artificial Intelligence. Most of the artificial intelligence that you see uses Python to function.

Then, there is machine learning, a field that amazes programmers and consumers. The entire field of machine learning sits on top of Python. Think of Python as the very foundation of machine learning. You remove Python and end up with nothing but an idea that just can't work.

If that wasn't enough, data scientists use Python to analyze data to create incredibly detailed files, charts, and visuals. The scope of Python continues to expand wider and wider.

If you really wish to work in cybersecurity, look no further. Most of the tools and scripts you will use require you to know both JavaScript and Python, the two most popular languages. Without these, you will be struggling to do much.

We can go on and on about just how useful, practical, easy to understand, and easy to learn Python happens to be, but that's not the point of this book. The goal of this book is simple: To help you learn Python.

This book won't be diving into extreme details, but it aims to empower you with enough knowledge to get started correctly. We will walk you through some scenarios, encourage you to use your intellect to work out solutions, offer situations requiring users to think matters through, and help you understand the codes. Not just that, we will also be offering some unique projects and exercises. We highly recommend that you pay attention to the steps and details. You'd be surprised how easy they are if you pay close attention to the tiniest things.

How quickly and how much you will go on to master Python or any other programming language entirely depends on how much you practice once you've learned the basics. Ideally, you'd want to practice your code daily and learn more from other resources, such as YouTube, Coursera, Udemy, edX, and any other book that helps you understand more complex and advanced topics. Practice these as much as possible. Check out communities and platforms where other programmers share their codes. These will help you to gain inspiration and learn more.

With that said, it is time to get started, get to the bottom of "Python is easy to learn," and find out if that is the case. At any point, if you feel stuck or unable to understand something, be sure to retrace your steps and learn what you may have skipped. Alternatively, use additional resources to learn from different perspectives.

Chapter 1: Python Fundamentals

> *"Learning to code is learning to create and innovate."* **–Enda Kenny**

Getting Started

Now that you have decided to start your programming journey with Python, it is time to learn how to set things up. To help you set things up nicely, below is a step-by-step guide that will help you install Python, an integrated development environment, and more so that you're ready to start coding your first program.

Before You Do—Installing Python the Smart Way

There are a few pro tips that we'd like to share with you before you go on to install anything on your system. These will help you make the most efficient install and will allow you to use Python more effectively.

For Windows

Be sure to install your Python in a higher directory. This means that you will want to target a directory or disk drive that's higher in value. Therefore, if you have C, D, and E, you'd ideally want to install it in the C drive. Furthermore, you'd want to ensure that you install it in C instead of C:/Folder/Sub-Folder/Python. Make it easier for you to access Python. This will prove to be helpful when you are going to connect Python with the IDE of your choice.

Be sure to use the correct installer as well. You can install Python in many ways, but it is recommended that you stick to the official website and download Python to install using the installation files provided. This will ensure that you do not end up with a version of Python that's unofficial or has bugs in it. You will also want to download the latest version of Python to take advantage of all the latest features and functions.

When installing Python, you'll need to know whether your system runs a 32-bit (x86) version or a 64-bit (x86-64) version. This is

crucial to ensure a correct and functional installation.

"How do I find that out?"

To find out what version of Windows your system is running, you can do the following:

1. Press the Windows key.
2. Type "Run" (without quotation marks) and press Enter/Return.
3. In the new dialogue box, type "dxdiag" (without quotation marks) and hit enter/return.
4. You should now be able to see the version next to "Operating System."

Once you know your version, you can go to Python.org to download Python. There, you'll be presented with several options.

1. Executable installer—This is your typical .exe file you can download and open. The file will execute, and the installer will carry out the rest. This is the default choice that all of us generally go for.
2. Web-based installer—This is essentially the same as the above. The only difference is that this separately downloads all the bits you need to install. This is a good option if you want to save space, as it significantly reduces the installer's size. For this option, you'll need an active internet connection.
3. Embedded zip file—The third option involves downloading a zip file containing all the files needed to install a minimal version of Python. Use this if you only aim to use Python once, perhaps for learning. If you want to stick with Python, this may not be the option to go for.

For Mac

Mac users also have a few options, just like Windows users do. However, before you go ahead and install Python on your Mac, it is good to consider the following:

1. Keep your base installation as clean as possible—You don't want to grab everything you can get and go for a massive installation. You can always add libraries and other functionalities individually if you need them later.
2. Be sure not to use the pip command to install directly into your system.
3. After installation, set up individual directories for each project. This helps organize everything and helps prevent confusion later on, especially if you are working on multiple projects.
4. Before installing Python, it is recommended that you install Python virtual environments first.

For Linux

Installing Python on Linux can be tricky, especially if you do not know what you're doing. Unlike Windows and Mac, Linux has distros or "flavors," which are essentially different versions of the same OS. You will have a unique interface, a different package manager, and so on, depending on your version.

	If you use a Debian-based distro, you probably use an apt package

	manager. If you use a Red Hat based distro, you'll use yum.

Once you know what distro-specific package manager you are using, you can easily install Python into your system.

1. Use apt or yum to download Python into your system.
2. You can also use additional tools, such as asdf-vm and Docker, to help you quickly manage your installation.

With that said, let's dive in and install Python, starting with Windows.

Instal Python

Installing Python on Windows

To install Python on Windows, make sure your Windows is updated. If you have any pending updates, use Windows' update manager first. Once all updates are installed, open your favorite browser and head to www.python.org.

Step 1

The first step is to decide which version of Python you'd like to install on your system. For most, it will be the latest version. However, experienced programmers may require specific and older versions of Python. You can find those under the Downloads section.

Step 2

Once you've decided which version you'd like to install, hover your mouse to where it says "Download" and click the button underneath "Download for Windows" to start the download process. Alternatively, you can click on "Windows" underneath "Download," this will take you to a page where you can see all the various versions available for your chosen operating system. Find the version you need and click the "Download" button to start downloading.

Step 3

Once the download is complete, click on the executable file that has been downloaded earlier. This will start the installer and the installation process.

Be sure to check both the boxes at the bottom before proceeding with the installation.

Step 4

After the installation, click "close" to close the installer window. Now, it is time to verify whether Python is installed successfully or not. To check that,

1. Press the Windows key.
2. Type "cmd" and hit enter to open the command prompt.
3. There, type in Python and press enter/return.

4. If you can see what version of Python is installed on your system, that verifies that the installation was successful.

Step 5

Next, you want to verify that pip was installed successfully as well. Pip is a very powerful and useful package manager that you can use for Python packages. To check,

1. Press the Windows key.
2. Type "cmd" and press enter/return to open the command prompt.
3. Type "pip -V" and press enter/return (note that 'V' is case-sensitive).
4. You should be able to see your pip version, which confirms that pip is installed.

Installing Python on Mac

For Mac, the installation isn't that complicated either. However, before you go ahead, it is important to understand that some macOS versions come with Python pre-installed. Therefore, the first thing you'd want to do is to find out if your system has Python. You can check whether you have Python and which version by doing the following:

1. Open up terminal.
2. Type either "compgen -c python" or "python3 --version" and hit enter.
3. You may also want to try "python --version" as your system may have Python 2 installed.

In most cases, you'd probably have a Python version that is already outdated unless you've recently bought your Mac system. Either way, we will learn how to install the latest version of Python on our systems. To do that, follow steps 1 to 3 from the Windows installation. However, instead of Windows, you'll be choosing

MacOS. The website will also be able to detect your operating system automatically and will provide you with the grey button that directly downloads the .dmg file for your macOS. The installation process will remain the same.

Once you have installed Python, check using the terminal and the above commands to ensure your Python is updated and installed.

Installing Python on Linux

First, you want to find out whether your Linux distro already has Python installed.

1. Press Ctrl + Alt + T to pull up terminal.
2. Type in:
3. Python --version (this is to check if you have Python 2).
4. Python3 --version (this is to check if you have Python 3).
5. Python3.x --version (replace x with the version you're searching for to see if it is installed).

Once you've ensured that you do not have the latest version of Python installed (which happens to be 3.12.0 at the time of writing this book), do the following:

1. In most cases, you will need to use the following command using your terminal window:
2. $ sudo apt-get install Python3.x (replace x with the latest version number).
3. Press Y whenever prompted to allow installation to continue.
4. Once the installation is complete, verify if Python is installed by using your terminal window and entering the command Python3.x (replace x for whichever version you installed). It should execute and start Python.

With that, you should now install Python on whatever system you use. However, we aren't done yet. To use Python properly, we need

an Integrated Development Environment (IDE), a software to help you use and operate Python and other languages.

Installing Your IDE

For every programming language, you generally require an integrated development environment. IDEs are what you'd call software designed to help you harness the true power of any given programming language that it supports.

Many IDEs these days come loaded with additional functionalities and features. The goal of an IDE is to help you code better, debug faster, and identify errors even before you execute a program to ensure that your code doesn't crash. Sometimes, the IDEs can help you learn how to code more efficiently by offering suggestions.

Furthermore, IDEs can help you keep track of your variables, which is quite a good thing. In the past, most programming was done either on Notepad or very old interfaces. All they offered was the ability to code, compile, and export. Finding errors in a code would be like finding a needle in a haystack. Today, all of that is done in seconds.

Through the use of IDEs, you go on to improve your productivity by using powerful features and functions built-in. It is also important to note that not all IDEs can support Python, and not all IDEs support other languages. Some are made exclusively to handle specific types of programming languages, while others offer vast support for various languages. Since we'll be using Python, it is a good idea to familiarize ourselves with some of the most popular and powerful IDEs for Python today.

IDLE

IDLE stands for Integrated Development and Learning Environment. It is the default editor that comes with Python pre-installed. IDLE is the go-to choice for those new to Python who wish to get the hang of things.

You can use IDLE to learn Python, level up your skills, and even work on initial projects before moving on to other IDEs. IDLE is

fully supported by Mac OS, Windows, and Linux, so installing IDLE shouldn't be a problem. For this book, we will also be using IDLE.

PyCharm

PyCharm is an open-source, free-to-use IDE developed by JetBrains. They are considered the masters of their craft, and that can be seen in PyCharm, an IDE designed specifically with Python on their minds. It is one of the most commonly used IDEs for Python today and offers some of the most excellent and powerful tools known to developers.

Now, you might be thinking, why not start with PyCharm if it is so good? Well, the thing is that PyCharm is best suited for large-scale Python projects. Since it has built-in support for JavaScript, CSS, and TypeScript, it is best suited for those who know their way around Python and would like to up their skills and projects to a new level. Once you're done learning Python using IDLE, this is where you need to go.

Visual Studio Code

Just like PyCharm, Microsoft created its own open-source and free IDE. It is another great option for those using Python parallel to other languages. It is lightweight and easy on the hard drive, too. However, it stands out by offering many features that other IDEs offer in their paid versions, and you get all of that for free.

Sublime Text 3

Sublime Text has been around for a while and is considered one of the most popular code editors. With that, it is worth noting that Sublime Text is a code editor and not exactly a full-fledged IDE. Its goal is to help programmers code and edit codes. Because of its niche focus, it offers lightning-fast development speeds and reliability.

Atom

Atom is yet another popular open-source code editor that GitHub develops. It is similar to Sublime Text and has almost the same features you'd find in Sublime Text editor. Like Sublime Text editor,

it is also free to download.

Other IDEs and Code Editors

- Jupyter
- Sypder
- PyDev
- Thonny
- Wing
- Vim
- GNU/Emacs
- Eric
- Dreamweaver
- Visual Studio
- Pyscripter
- Rodeo

While we will be using IDLE for this book, it is a good idea to explore these options as well, as you may find one that you'd like to switch over to once you're done learning. With that out of the way, we head straight into some good news. If you have just installed Python on Windows or Mac, you have already installed IDLE. Linux users, however, will need to install it using your distro's package manager. To do that, type in the following command:

- $ sudo apt-get install idle3 -y

Hit enter/return, and the rest will be done for you. After the installation, you'll see IDLE on your desktop. For Windows and Mac users, search for IDLE, and they should be able to find it easily. It is time to click and open IDLE to get a taste of the interface and type in our first code.

Your First Step to Code

Exciting, isn't it? You're just moments away from typing in some code and seeing a response. As all great programmers have done before, we will start with Hello World. This simple program asks Python to print out the words "Hello" and "World." How do we do that? Let's find out.

Open up IDLE if you haven't done so already. Once it is open, you should see the version of your Python at the top, followed by some other messages. Your cursor, by default, would be placed next to the >>> signs. This is where you will be typing in the following code:

```
print("Hello World")
```

Remember, all codes that we will do are case-sensitive. This means that if you type in Print instead of print, it won't work. Therefore, getting used to this type of typing is a good idea.

Once you've entered the code above, press the enter/return key. VOILA! You just did a thing. If things went well, which they did, you'll be able to see Hello World written in blue. That's the output of the command you just used.

Yes, it's probably not as exciting as you may have thought, but the goal is to take things slowly and go step-by-step. You might also be wondering what is print, why the parenthesis, and what's with the quotation marks. To understand that, let's start learning the basics of Python.

Python Variables

Variables are just like empty containers. When you create them, they can store some form of value in them. They can hold on to that value for as long as you want. Their values can be modified, changed, altered, or removed altogether.

Every programming language uses variables to store and recall values. They serve as the building blocks for any code of any programming language. They come with a reserved memory location where they can store values. If you recall, Python has a built-in garbage collection feature that dis-allocates and frees up memory as you go along. This means we don't have to worry about

using more memory than we need.

Unlike other languages, we can create new variables easily using Python. To create a new variable in Python, you do not need to declare the kind of variable you create. For example, if you created a variable containing whole numbers, such as 10, we'd first need to let the compiler know, "Hey! I want to create a variable with an integer value of 10." Of course, that's not exactly how you'd do that. However, we don't have to worry about that in Python. How? Let's have a look.

In the example below, we will create two new variables, var1 and var2. These variables will have random numbers as values, say 312 and 944, respectively. To do this, return to your IDLE screen and type in the following code.

```
var1 = 312
var2 = 944
```

Now, let's see if these values were correctly stored, and for that, we want the system to print out the values individually. You do that using the print() function, as shown below.

```
print(var1)
print(var2)
```

You should now be able to see the values as a result. It does look easy. That's because it is. With that in mind, there's something every programmer, new or old, must know and practice: naming convention.

How to Name Your Variables the Right Way

You don't want to name variables a, b, xyz, uhd, or any of that. While no one's stopping you from doing that, it's considered bad practice. Why? Let's say you're a developer and have been tasked to handle a project that someone else coded. As you go through the codes, you come across variables with names like:

```
alpha
beta
```

```
var-z
tes1
```

None of these names will make any sense. You will never be able to determine what exact purpose these variables serve unless you go through hundreds or thousands of lines of code before you get an idea. However, this can easily be avoided by correctly naming your variables.

The name of your variables must make sense to any programmer of any level. Therefore, if your variable is a character's name, it should say something like `char_name` so that when you or any other programmer sees it, they immediately know what this variable is for. Similarly, if you are creating a variable for a high score, it should say something like `high_score` or `h_score`.

Now that you have that in mind, it's time to learn how to type those variables correctly using the correct naming convention.

To name a variable the correct way, you will need to do the following:

1. The name of the variable must always start with either a letter or an underscore:
    A. `_example`
    B. `example`
2. The name of the variable must not start with a numerical value:
    A. `1variable` (wrong)
    B. `variable1` (right)
3. The name must only contain alphanumeric characters and underscores:
    A. `my_variable`
    B. `high_score`
    C. `dogs_name`
4. Names are case-sensitive:
    A. `variable` is not the same as `VARIABLE` or `Variable`.

5. Reserved words cannot be used as variable names:
    A. while
    B. if
    C. print
    D. and
    E. while
    F. for

With these rules in mind, let's look at how to type these names properly. There are three types of conventions that you can use to write your variable names. These are:

1. Camel Case
    A. The first word must always be lowercase.
    B. The first letter of every word that will follow is in uppercase.
    C. No space between words.
    D. Example:
        i. myNewVariable
        ii. studentCount
        iii. myNextLevel
2. Pascal Case
    A. The first letter of every word is to be uppercase.
    B. No spaces.
    C. Example:
        i. MyNewVariable
        ii. StudentCount
        iii. MyNextLevel
3. Snake Case
    A. Add an underscore between each word.
    B. No space between words.
    C. Can use upper- and lower-cases at will.

D. Example:

i. my_New_Variable
ii. Student_count
iii. my_next_level

Using these, you can now name your variables. It's best to name your variable with a simple name that shows precisely what they should do. This will always help you and other programmers who may work on your code to know what these variables should do.

The Data Types in Python

Every programming language comes with its own unique data types. A data type defines what kind of value a variable will be storing. For example, in the previous example, we used the word "Integer" instead of a number or a whole number. That is because an integer happens to be a data type found in all programming languages.

When it comes to Python, there are eight data types:

1. Integers
2. Float
3. String
4. Boolean
5. List
6. Dictionary
7. Tuple
8. Set

Integers

Integers are the most basic types of data a variable can store. When a variable stores a particular data type, it changes its type. Therefore, if my_var holds the value of 10, it becomes an integer variable. Of course, you will need to specify this in other languages. On the other hand, Python is smart enough to figure that out independently.

To create an integer variable, type the variable's name and assign it a value.

```
my_var = 10
```

Float

Integers aren't just any numbers, though. Remembering that integers are numbers without any decimal points is a good idea. When you add a decimal point to the value, it is classified as a float.

To declare a float, type in a name for your variable and assign it a value:

```
my_float = 10.0
```

A float is a data type that holds numeric values with decimals. Therefore, 10 is an integer, but 10.0 is not. The latter is classified as a float value. That makes sense, right? However, what if you want to store letters or words as your values? Where do they go? For that, you need a data type called string.

String

String is a data type that stores letters, words, and even sentences. You can store even paragraphs and paragraphs into a single variable as long as the memory allows. Creating a string variable is slightly different from the previous two data types.

To create a string variable, you start as usual and name your variable. After the assignment sign (=), you use double quotes before and after the letters or words you want to store. This lets Python know you're trying to create and store a string value. Therefore, if you want to store the value of "I am a Programmer" in the variable `my_job`, you'll need to type this:

```
my_job = "I am a Programmer"
```

Now, go ahead and print this variable to see how it works.

Boolean

This is an unusual but quite useful data type. Unlike any other data type in Python, this data type can only store two values: True or

False. This data type makes your code behave a certain way using logical inputs and outputs.

Programmers use Boolean (or Bool) to create conditions and check conditions and outputs, which can help make the program more responsive and intelligent. To declare a Boolean variable, you do the following:

```
sit_A = True
sit_B = False
```

If you like, you can use the following code to verify the values.

```
sit_A = True
sit_B = False
print("sit_A =" , sit_A)
print("sit_B =" , sit_B)
```

This will print out the following:

```
sit_A = True
sit_B = False
```

Of course, that's not exactly how we will use Boolean values. We will use these in more complex situations, but we'll do that later.

List

As the name suggests, a list holds a group of values. For example, you would like to store three names in the variable name. To do that, you will need to do the following:

```
name = ["Joey" , "Ross" , "Chandler"]
```

Now, a single variable has three different values stored in it. To recall these values, ask Python to print names, and you will see all the names printed.

Remember, to declare a list, you must use the square brackets [ ].

Lists are handy because you can access individual components and values anytime. For example, if you were only aiming for Python to print "Ross" as an output, you'd do the following:

```
print(name[1])
```

This will print out Ross. You might wonder why Ross was the second value, not the first. However, we will learn more about why that is the case when discussing indexing.

Dictionary

The dictionary works a lot like lists. The difference is that a dictionary stores a group of values in pairs instead of just single values. Let's say you have the following data that you must store:

```
item_1 = butter
item_2 = egg
fruit = banana
```

You'd probably store them in three different string variables in any other circumstances. That will take up more memory than needed. A much more efficient way to store these will be through a dictionary. To do that, you'll need to do the following:

```
shopping_cart = {"item_1" : "butter" , "item_2" : "egg" , "fruit" : "banana"}
```

Now, if you were to ask Python to print shopping_cart, it will show you the following results:

```
{'item_1': 'butter', 'item_2': 'egg', 'fruit': 'banana'}
```

Dictionaries are excellent if you want to store data regarding specific things, such as employee name, ID number, role, etc. It is also important to note that dictionaries use curly brackets unlike lists, which use square brackets { }. Every group is defined by the use of the colon (:) symbol. The comma (,) is used to separate one pair from another.

Tuple

Tuples are also quite like lists and dictionaries. They, too, store multiple values in them. However, these can store multiple types of data in order. Once a tuple is created, you cannot alter its value, something you can do with dictionaries and lists.

To create a tuple, you use the regular brackets ( ), as shown here:

```
Today_Date = ("Sunday", "29th", "October", 2023)
```

This will store both string and integer data type values in the variable Today_Date.

Set

The set is exactly like a tuple. The only difference is that you can modify the values of a set, something you can't do with a tuple. To create a set, use the curly brackets minus the colons and double quotation marks we use for dictionaries.

```
my_set = {1, 2, 3, 4, 5, 6, 7}
```

Exercises

Now that you know how to create variables and store different values, let's put that knowledge to the test before moving toward the next chapter. I know this chapter was quite long, but now, you're ready to move into the actual programming side.

Exercise 1

- Create a variable named my_name and assign your name as its value.
- Create a second variable named my_age and assign it your age as its integer value.
- Print both variables using the print() function.

Exercise 2

- Create a variable named x and give it a value of 10.
- Create another variable named y and give it a value of 20.
- Create a third variable named z and make it the sum of the two variables x and y
- print the value of z.

Once you're done, it's time to move forward to the next chapter and get started with coding.

Chapter 2: Python Basic Operations and Input/Output

> *"Good code is its own best documentation."*
> **–Steve McConnell.**

Now that we know the data types and how to create variables and store values, it is time to start coding. Right away, we will start with basic Python operations, such as arithmetic and concatenation, and then move on to input and output. Once again, be sure to have IDLE open and ready to go. If you previously have values stored, it's a good idea to shut the application down properly and restart IDLE so that you start with a clean slate.

Basic Operations in Python

Arithmetic Operations

Like any other programming language, Python can carry out large arithmetical operations with efficiency and finesse. While we will not be diving into super lengthy arithmetical operations for now, we aim to understand what kind of arithmetic operations you can carry out and how.

The Arithmetic Operators

When it comes to Python, you can use the following arithmetic operators to carry out a variety of operations. Each symbol shown below is called an operator and tells Python the exact kind of arithmetic operation that must be carried out.

Operators	Their Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division (float)

//	Division (floor)
%	Modulus - Returns only the remainder of the Division
**	Power (exponent)

These are pretty easy to understand. However, there's something called the arithmetic operator precedence that you must always consider.

The fact is that if you are using multiple arithmetic operators, some of them will always be solved first, while others will be given a secondary priority. If you do not consider this, you may end up with the wrong results.

The operator precedence is as follows (in order of precedence, top being top priority):

Operator	Purpose	Associativity
**	Power (Exponent)	right-to-left
%, *, /, //	Modulus, multiplication, division, floor division	left-to-right
+, -	Addition and subtraction	left-to-right

As seen above, Python will always compute the power or exponential operator first, followed by modulus, multiplication, and division operators, and finally, addition and subtraction.

With that out of the way, let's look into each operator and see how they work.

Addition

To add two variables or more, you use the + sign as shown below:

```
10 + 20
```

The result will show 30. You can also use variables with unique values and add them to find the result. Alternatively, you can use the resulting number to become a unique variable, as shown below:

```
number_1 = 20
number_2 = 22
number_3 = number_1 + number_2
```

Now, number_3 will store the sum of the first two variables as its value.

Subtraction

Subtraction works in the same fashion as addition does. Therefore, to subtract two numbers, all you need to do is type in the numbers, use the subtract operator, and be presented with the results.

```
20 - 10
10
```

Similarly, you can also create different variables, subtract those, and store the answer in a separate variable, as shown below:

```
sub_1 = 25
sub_2 = 17
sub_3 = sub_1 - sub_2
```

Now, the resulting number will be stored as sub_3's value.

Multiplication

While we are used to calculators with the X symbol for multiplication, things work differently in the programming world. Instead of X, we use the asterisk (*) sign as our multiplication operator. Just as before, to multiply two numbers, all you need to do is the following:

```
100 * 21
2100
```

Here's a quick test. Without using code, what do you think the answer to the following will be?

```
10 + 8 * 2 - 20
```

Common sense would dictate the answer will be 16, but that would be wrong. Why? Because of the operator precedence. Here, Python will first solve 8 * 2, which results in 16, after which it will carry out

the rest of the operations. The answer, therefore, will be 6. Try it yourself.

Division

Once again, we are used to seeing the typical division symbol of ÷, but that's not the case regarding programming. First, you can't type that symbol using a standard keyboard. Second, programmers use the / operator to carry out the division. How? Here's how:

```
x = 49
y = 7
x / y
```

```
7.0
```

Once again, if you are using multiple operators, be sure to factor in the operator precedence.

Modulus

Now, this one is somewhat different. We don't have anything like that in the usual textbooks. However, it isn't that complicated.

Let's say you want to divide 1,000 by 9 and find out the remainder after all the successful divisions. Usually, you'd divide 1000 by 9 and find that you can do that 111 times. That gives you 999. Now, you're left with 1 as a remainder. Using the modulus operator, you can find that out right away. How? Here's how!

```
1000 % 9
```

```
1
```

It's that simple. It may seem like an operator that won't make much sense, but there are many cases where you will need to call upon the modulus operator to help you sort the situation out.

Exponentiation Operator

Exponentiation operators are used in Python to raise the power of a value by a certain number. For example, if you want to find out what's 10 to the power of 10, you'd do the following:

```
10 ** 10
```

```
10000000000
```

Floor Division

Floor division is done using the // operator. This is carried out to find the floor of the quotient after the first value is divided by the second one. In simpler words, you can find out how many times the second number divides successfully before it is no longer possible to divide without entering the decimal ranges.

Therefore, if you want to find out the floor of the quotient when 8 is divided by 5, you do the following:

```
8 // 5
```

```
1
```

This is because 8 can only be divided by 5 once, leaving a remainder of 3. Remember, the // will tell you how many times the second number can divide, whereas the / operator will tell you the exact number required to divide the first number successfully. If you were to use 8 / 5, you'd get 1.6 as the answer.

String Concatenation

The next thing for us to learn is something called string concatenation. We already know what a string is. It is essentially a letter, word, or paragraph stored as a string value in a variable.

String concatenation essentially merges two separate strings into a new one. For example, we have two string variables named msg_1 and msg_2, as shown below:

```
msg_1 = "I am learning "
msg_2 = "Python for Beginners"
```

If you were to print these, you'd need to print them separately. You can concatenate these two into a single string to add both together. We'll store this new value in msg_3.

```
msg_3 = msg_1 + msg_2
```

```
print(msg_3)
```

Now, both strings will be concatenated, and you should be able to see a singular print result.

```
I am learning Python for Beginners
```

With that, you might be wondering why on earth you would ever need to concatenate strings together. Well, you may not be doing a lot of that right away, but as you move further up into the world of programming, you will come across many instances where you will need to do just that.

To concatenate strings, you can use one of the few ways:

Using the + Operator

Just moments ago, we saw exactly how to do just that. You add up two string variables to concatenate a string, and the result will either be printed or stored, depending on how you want the output.

Using the * Operator

When you want to concatenate the same string multiple times, you use the * operator.

```
print(msg_3 * 4)
```

This will concatenate the msg_3 string four times. The result will look like this:

```
I am learning Python for BeginnersI am learning Python for BeginnersI am learning Python for BeginnersI am learning Python for Beginners
```

See how that works?

Using the join() Method

While we haven't touched base on methods and functions, it is still worth knowing that you can use a join() method. This will allow you to join strings in Python and concatenate them together. This is considered the most flexible way of doing things.

```
a = "My "
b = "Name "
c = "Is Cole"
```

```
print("-".join([a,b,c]))
```

```
My -Name -Is Cole
```

Now, all the three will be joined together.

Using % Operator

You can also use the modulus operator to combine strings, as shown below.

```
a = "Orange"
b = "Juice"
print("% s % s" % (a, b))
```

```
Orange Juice
```

Using format() Function

You can also use the format() function for string concatenation. This is a powerful way to not only concatenate but also format strings.

Let's say that you have two variables named a and b, and they hold the following values:

```
a = "Apple"
b = "Juice"
```

If you were to concatenate it regularly by using the + operator, you will end up getting:

```
AppleJuice
```

However, to ensure you format it correctly and concatenate both strings, you can use the following method:

```
print("{} {}".format(a, b))
```

```
Apple Juice
```

F-string Method

Finally, you can use the f-string literals, or formatted strings, to concatenate strings. To do that, you will need two variables, say a and b, with the same values as above, and then you will need to do the following:

```
a = "Apple"
b = "Juice"
print(f' {a} {b}')
```

```
Apple Juice
```

The goal is to understand how each of these methods works. While you are free to use whichever method you prefer, it is still a good idea to know these additional ones as there may be instances where these may come in handy.

Python User Input and Output

So far, we've been providing values to variables. However, what if you want your end user to provide that value for you? Just like games and applications ask you for your name so that they can address you later on, you too can use a variety of ways to take input from users, store the input as a value, and then use that to carry out other operations.

For this, Python comes with two built-in functions: `input()` and `print()`. The `input()` function allows users to provide their input, whereas the `print()` function prints out the values provided by the users.

To use the `input()` function, the syntax is as shown below:

```
input("message")
```

For example, you want to ask your end-user, "Hey, You! What's your name?" Then, you want the end-user to input their name, which is stored in a variable called `player_name`. How do you think you can achieve that? It is reasonably easy to figure out. Here's how to do just that.

```
player_name = input("Hey, You! What's your name?")
```

Now, the end user will see "Hey, You! What's your name?" and they will see a cursor blinking right at the end of the message. That is where they can type in their names. Go ahead and type in your name or any name. Once you're done, hit enter, and that will be stored as a value for `player_name`. Let's see if that worked, and we do

that by using the print() function.

```
print(player_name)
```

You will now be able to see whatever you entered as an input. If you want to greet the user, you can do something like this:

```
print("Hello, " + player_name)
```

Using the input() function is a great way to make your programs more interactive and allow end-users to give their input. In most cases, their input will help run the entire program smoothly.

Users can also input numerical values, which will be stored and used for various arithmetic operations.

```
num_1 = input("Choose a number: ")
num_2 = input("Choose a second number: ")
print("The sum of the two numbers you chose is " + num_1 + num_2)
```

Suppose you chose 20 as your first number and 40 as your second. Let's see what the print results are.

```
The sum of the two numbers you chose is 2040.
```

"Wait a minute! That isn't right."

Exactly. When a user inputs something, it is automatically treated as a string, not a numeric value. To use those numbers correctly, we must specify that we want the input value to be converted into an integer. How do we do that? We add int() before the input, as shown below:

```
num_1 = int(input("Choose a number: "))
num_2 = int(input("Choose a second number: "))
total = num_1 + num_2
print("The sum of the two numbers you chose is: " , total)
```

```
Choose a number: 20
Choose a second number: 40
The sum of the two numbers you chose is: 60
```

Comments

Now and then, you will come across proper English words and sentences. However, they won't do anything to the code for some reason. The moment you read them, you'd know what the code or the following block of code is all about, and that's great, but how do they do it?

Well, what you're looking at are called comments. Python offers the feature to leave comments as notes for yourself and fellow programmers who may later be looking at your code. Leaving comments allows them to understand what they are looking at, what any given piece of code does, and what purpose any particular line serves.

To leave comments, there are three ways:

1. Using the # sign
2. Using the""" """ quotation marks (triple quotation marks)
3. Using the ''' ''' apostrophe (triple apostrophe marks)

The first one, #, is used to leave single-line comments. These are useful if you want to type in short comments or briefly describe a particular line of code. If you want to have a multiple-line comment that usually explains many things, you need to use triple quotation or apostrophe marks.

```
#This is a single line comment
```

```
“””This is a multi-line comment that can stretch on
for as long as you like. There is no limitation to how
much you type here because the compiler will compute
none of this.”””
```

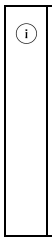
```
‘’’This is also a multi-line comment that can stretch on
for as long as you like.’’’
```

The good thing about using comments is that these are never processed by the compiler, meaning these will not affect your program, the input, or the output in any way, shape, or form. Therefore, these are used purely to make the code more

understandable and to leave notes behind for other programmers.

As a good coding practice, include comments whenever you can. These encourage programmers and developers to understand what's happening and will allow you to gain professional recognition moving forward.

Be sure to use comments clearly and concisely. Try not to write unnecessary words, as these can easily confuse programmers.



For pieces of code that you don't want to use right away, comment them out and use the comment symbols to disable them. When needed, remove the comment symbol to allow the compiler to read the code.

Exercises

It's time to put your thinking cap on and try the following little challenges. The first is relatively simple, while the second is more interactive.

Exercise 1

Code in a program where you take in input from the user for their name. Using that, the program should then greet the user.

Your code must also have comments for at least a few lines to explain what the coded line will do.

This should be reasonably easy for you by now.

Exercise 2

This is an interactive story that you are going to create. The goal is to use basic operations, including input and output, to make the story more interactive. This story, however, will also require some additional things, such as if, elif, and else, which are conditional statements. While we haven't covered those, be sure to return here and continue from where you leave later.

The objective is simple—this story will have branches that depend on the user's answers. Depending on the answers, the story will take a particular direction.

Here are the steps you need to follow:

1. Plan your story.
2. Initialize variables (create them).
3. Capture your user's name.
4. Greet the user and explain the situation, what will happen, and where the user is in the story.
5. Present their choices and let them decide (input function).
6. Continue the story based on the user's choice.

To give you a head's up, here's a sample code:

```
# Initialize variables
user_name = ""
user_choice = ""

# Capture user's name
user_name = input("What is your name, brave adventurer? ")
```

```
# Start the story
print("Welcome, " + user_name + "! You are on a quest to find the hidden treasure.")
```

```
# Present choices
user_choice = input("You come across a fork in the road. Do you go left or right? (left/right) ")
```

```
# Continue the story
if user_choice == "left":
    print("You encounter a friendly wizard who helps you find the treasure. Congratulations!")
elif user_choice == "right":
    print("You get lost in a forest and are never seen again. The end.")
else:
    print("Invalid choice. The end.")
```

Now, go out there and create a turn-based story game that is fun, interactive, and gripping. Once you've done that, we'll move on to

the next chapter to learn about Python's control structures and data collection.

Chapter 3: Control Structures and Data Collections

> *"Much like life itself, success in programming depends on making good decisions."* **–Anonymous**

Programming is fun, and what makes it more interesting is the ability to make your program intelligent. What we mean by "intelligent" is simple: your program must be able to make decisions based on the user's input or choices.

For example, imagine using an app. Every time you open it, it checks the weather outside and then, depending on the weather, suggests to you what kind of clothes are ideal to wear. This sounds simple enough and something we are used to seeing on our smartphones. We are so used to seeing this that we do not see the actual code at work; well, now we will.

Imagine if it's 100 degrees outside, and the app says, "You must wear warm clothes." You'd probably think the app's gone mad. After all, who'd want to boil themselves in 100 degrees while wearing a puffy jacket? Nobody!

However, if the app suggests you wear light-colored and comfortable summer wear, you'd know it's making perfect sense. Well, the app isn't exactly intelligent on its own. We human beings give it that direction. We teach the app what to do "if" a specific condition arrives. If it's winter, wearing warm clothes is ideal. Perhaps something light and breezy should do the trick if it's summer. All of these ifs and buts are what go on to create control structures.

When it comes to programming, these are extremely important to use. If you can go back to the sample code we gave you at the end of the last chapter, you'll see some new pieces of code saying `if`, `elif`, and `else`. They are conditional statements, and they control the flow

of the program. This chapter, therefore, is all about understanding these and learning how they can help us make our programs smarter.

Conditional Statements

Conditional statements are the backbone of every great program created through coding. Whether using Python, JavaScript, or Node.js, you need conditional statements to make your program work more effectively.

Conditional statements work by considering the input and then matching possible outcomes. If a scenario matches, that particular route is taken as a decision. If the first condition isn't applicable, the program checks with the second one, third, and so on. As long as a condition is met, the program will execute. However, if no conditions are met, the program has a final option that it then takes as an output.

For example, when we go to school, we are graded based on how many marks we achieve in our examinations.

- If we get 90 or more, we get an A.
- If we get between 80 to 89, we end up with a B.
- If we get 70 to 79, we get a C.
- If we get 60 to 69, we have a D.
- If we get anything lower than 60, we get an F.

This is something most of us are used to seeing as we've been through these. Now, forget that these are marks or grades. Instead, focus on the conditions here. "If" a value is more than x (x can be any number or string or a Boolean value), we get y as our result.

To further elaborate, let's say three students ended up with the following marks:

1. Clark = 86
2. Barry = 51

3. Bruce = 92

These students acquired their respective marks. Based on this, you then match the conditions to see where they fall. Similarly, a program will also do the same once it knows the values it is being provided with. Whichever condition matches, you'll get the appropriate outcome. If no condition matches, such as scoring below 60, you get an F.

To create conditions in Python, we use the reserved keywords:

- if—For the first condition.
- elif—For all conditions after the first.
- else—For the final condition (when no conditions are met).

So, how do these work? Let's take an example and use the data we already have (marks from our three students).

```
# Grading Software

name = input("Please Enter Your Name: ")
print("Hello, ", name + "! Welcome to Our Grading Software!")
```

```
#marks to be stored as integers
marks = int(input("Enter your marks: "))
```

```
print("Fetching your grade. Please wait!")
```

```
#grade variable initialized, but the value will be decided based on marks
grade = ""
```

```
if marks >= 90:
    grade = "A" #value of grade will automatically update depending on
    marks achieved
    print("Your grade is: ", grade)
elif marks >= 80:
    grade = "B"
    print("Your grade is: ", grade)
```

```
elif marks >= 70:
    grade = "C"
    print("Your grade is: ", grade)
elif marks >= 60:
    grade = "D"
    print("Your grade is: ", grade)
else: #for all other marks that are below 60
    grade = "F"
    print("Your grade is: ", grade)

print("Thank You for Using This Software!")
```

In the above, we have created a simple grading software that asks users for their names and their marks. Based on the marks, the program checks with all the conditions to see where it fits. Depending on that, it returns a value for grade and provides it to us as an output.

Using the code above, try to put in various numbers to see how it works. Be sure that you only try integers and not float values.

Regarding conditional statements, we use if, elif, and else in that specific order. However, you can always use if and else alone when you do not have a third option or do not wish to provide one.

For example, a game character has arrived at a crossroads. The character can either go left, or it can go right. There is no third option here. In such a situation, you will only use if and else. One will define a direction, and the other will be the only alternative.

That said, it is vital to understand how these codes work correctly and what you should do to ensure they are written with the best practices.

For if statements:

```
If (define condition):
    #statement goes here (with indentation)
```

This is the same for elif statements as well. However, for other statements, notice how there is no condition defined. To use else

statements, you do the following:

```
else:
    #statement goes here
```

For all statements, you must end the else condition with a colon (:). IDLE is smart enough to identify that you're typing in a conditional statement. That is why the moment you press enter, it will leave an indent. Having this indent is good as it makes the code more readable.

With that said conditional statements need some other components to work. You may have already noticed how we used > and = symbols. These are just some logical operators conditional statements need to perform flawlessly. Let's learn these and find out what they do.

Comparison and Logical Operators

It is safe to say that all conditional statements are essentially comparisons. A given value is compared with the statements before the program makes a decision. To help do that, we need comparison operators, and you saw three of those earlier.

The following are the comparison operators you can use to compare situations:

Operator symbol	Name	Purpose
==	Equal	This is used to check if a situation is equal to something.
!=	Not equal	This is used to check if a given value or condition is not equal to another.
>	Greater than	To compare a value and see if it is greater than something
<	Less than	To compare a value and see if it is less than something
>=	Greater	When comparing a value to check if

	than or equal to	it is greater than or equal to something
<=	Less than or equal to	When comparing a value to check if it is less than or equal to something

You may be wondering why use the double == symbol instead of the traditional = only, right? When it comes to programming, you need to use == to check if something is equal to something else. The fact is that we are already using the single = sign as an assignment operator. Therefore, using = will assign the value on the right to the variable on the left. It will not compare values.

The program compares given values using comparison operators to see if they match. When they do, a Boolean value of True is passed through the system. You can have the program print out True or False, to check what's happening.

When a condition is true, only then is it executed. If it is false, it will not be executed, and the program will move on to the next one until it either finds a matchable condition or runs out of all options, in which case it will settle with the else condition.

Logical Operators

Besides comparison operators, there are logical operators. As the name suggests, they provide more context to the comparison by adding logic. In short, they make the code work and behave a lot smarter.

Local operators help by further clarifying conditions. We use these to make logical comparisons and arrive at a logical conclusion. For example, if it is summer, you may be told to wear something light, which is perfectly fine. However, if it is summer and raining, you will need a raincoat or an umbrella to prevent getting wet. The usual "Wear something breezy and light-colored" won't be a logical answer. To solve such issues, we use logical operators.

In Python, we have three logical operators. These are:

1. and—This comes in if both conditions are true. For example,

summers "and" raining. In simpler terms, X and Y need to be true.

2. or—This is used when one condition needs to be true. Here, either X or Y must be true for it to work, not both.
3. not—This is used when the condition is not true. "I will go out when it is not raining" is a perfect example.

The and-operator will return a bool value of True only when both conditions are met. The or operator will return as True when either one of the conditions is met. The not operator will return as True when the condition is not met.

Let's look at a simple example and see how logical operators can help us create better logic. For this example, we will create a simple pre-qualification software to decide whether a person is eligible for a loan.

For this, we will need a few things:

1. Age of person (must be between 25–40 years of age).
2. Whether working or not (eligible candidate must be working).
3. Income (must be at least $65,000 a year).
4. Previous bad debts (automatically disqualifies candidates if they have any previous bad debts).

With these in mind, we have a rough sketch of the variables we will use. To make things simple, we will use the following variables to capture this information:

```
age
work_status
income
b_debt
```

Remember, some will remain a string, but some must be changed into integer values. We will also need to capture the applicant's first and last name.

Next, we must create our conditions, use logical operators to carry out this operation, and determine if the person qualifies. We already know our qualification criteria. To qualify, an applicant must:

- Be 25 to 40 years old.
- Be currently working.
- Be making at least $65,000 a year.
- Not have any previous bad debts.

Now, there's a problem. If we type in the usual way, we will see that we can only set one condition, as shown below:

```
if age > 24:
```

Using this will quite literally approve anyone who types their age above 24. They'll be approved even if they have a bad credit history or have no job. That's a problem. Therefore, we can use the and operator here to help us create logic.

```
# Loan Application Software

# Data of applicant (input required by user)
print("Welcome to Your Bank! Please fill out the form below!")
f_name = input("Please, enter your first name: ")
l_name = input("Please, enter your last name: ")
age = int(input("Please, enter your age: "))
work_status = int(input("Are you currently working? 1 for yes, 2 for no: "))
income = int(input("Please, enter your yearly gross salary: "))
b_debt = int(input("Do you have any previous bad debts? 1 for yes, 2 for
no: "))

print("Thank you! Please wait while we process the information.")

if age in range(25, 40) and work_status == 1 and income >= 65000 and
b_debt == 2:
    print("Congratulations! You are eligible to apply for a loan!")
else:
```

```
    print("Unfortunately, your loan application does not meet our
    requirements!")

print("Thank you for choosing Your Bank! Have a great day!")
```

In the above, you might have noticed two things:

1. We used the and operator a few times.
2. We used a range() function.

You can use as many "and" or any other operator in the code as long as it makes sense. As far as the range function is concerned, we used that to ensure the age condition sticks to the range we need. We will talk more about functions and methods later on, but we'll stick to this for now.

Using the code above, try the program with different inputs and see how the program behaves. Of course, this isn't the most convenient way of writing code. So many things can be done to make the code look and feel a lot nicer. However, since we are beginning, it's okay to keep things as neat and sensible as possible without getting into too many complexities.

Logical operators help us keep our codes more relevant and concise. If you were only to use conditional statements, the same code would end up longer, something like this:

```
# Loan Application Software

# Data of applicant (input required by user)
print("Welcome to Your Bank! Please fill out the form below!")
f_name = input("Please, enter your first name: ")
l_name = input("Please, enter your last name: ")

age = int(input("Please, enter your age: "))
if age >= 25:
    if age > 40:
        print("You're age is higher than our maximum required age!")
        exit()
```

```
else:
    print("You're too young for a loan application!")
    exit()

work_status = int(input("Are you currently working? 1 for yes, 2 for no: "))

if work_status == 1:
    print("That's great!")
else:
    print("Since you do not have a job, you are not eligible for a loan!")
    exit()

income = int(input("Please, enter your yearly gross salary: "))
if income >= 65000:
    print("You appear to have a higher income than the required minimum
    level, great!")
else:
    print("Your income currently does not qualify for a loan!")
    exit()

b_debt = int(input("Do you have any previous bad debts? 1 for yes, 2 for
no: "))
if b_debt == 2:
    print("Everything looks good here!")
else:
    print("You are not eligible for a loan!")
    exit()

print("Congratulations! You appear to be eligible for a loan!")
print("Thank you for choosing Your Bank!")
```

See the difference?

Loops and Iterations

Next, we have something called loops. As the name suggests, loops are used when you wish the program to loop through a range of values, carry out multiple iterations of a command in a swift go, and carry on doing this until the condition is satisfied.

To use loops, Python uses two reserved keywords:

1. for
2. while

For

The for loop is used when you wish to iterate over a given range of numbers or sequences. This can be a list, a range, a dictionary, a tuple, a string, or even a set. If you already have exposure to programming, it is worth noting that this for is very different from the one you may be used to seeing.

In Python, the for loop executes a given set of statements for every item within the given list, set, et cetera. The loop will continue to function until the given condition is no longer true or until it is ultimately achieved. Let's see this in action:

```
my_fruits = ["Apple", "blueberry", "cherry", "orange"]
```

We created a list named my_fruits in the above variable and inserted four values. Now, we want Python to print the content of the list. To do that, we will need the for loop.

```
#x is a loop variable that is local to the loop only
for x in my_fruits:

print(x)
```

The outcome will be as shown below:

```
Apple
blueberry
cherry
orange
```

The for loop will go through each value, print it, move on to the next, and continue until all the lists' elements are printed.

You can also go through a string and have individual components of the string printed separately.

```
for x in "pomegranate":
```

```
print(x)
```

This will print out the following:

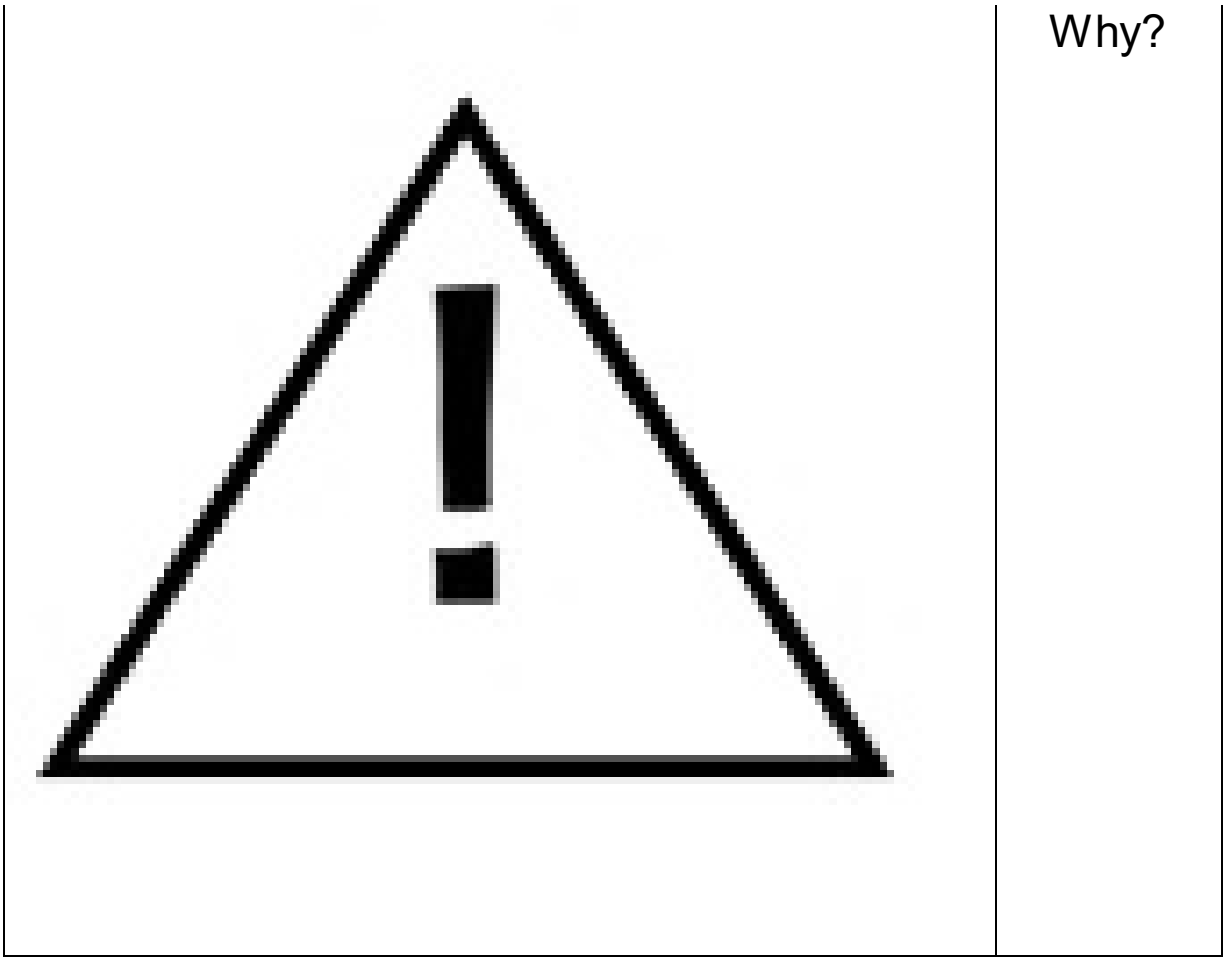
```
p
o
m
e
g
r
a
n
a
t
e
```

While

Besides `for` loops, Python also has `while` loops. This kind of loop continues for as long as the condition remains true. The moment it stops being true, the loop is terminated.

```
i = 0
while i < 10:
    print(i)
```

	Before you execute this, though, it is crucial to understand that this will lead to a system crash.



Reread the code and see if you can find out what the apparent problem is.

Yes! The variable i will always be 0, and it will always be less than 10. This means that the loop will never end, and that will crash IDLE and probably the system. To overcome that problem, we want to ensure that we instruct Python to add increments after every iteration.

In the first go, i will be 0. For the next iteration, we want it to be 1, then 2, and so on. To do that, we must add the following line of code just underneath the print(i) command.

```
i = 0
while i < 10:
    print(i)
    i += 1
```

The += is an incremental sign that tells Python to increase the value of the variable on the left by one every time it loops. You can also do the opposite and have -= 1, which will decrease the value of i by one for every iteration.

Now, if you run the above code, you'll get:

```
0
1
2
3
4
5
6
7
8
9
```

Since the condition stated Python would print i as long as it remained less than 10, it did just that. When it became 10, the condition was no longer true, and the loop was terminated.

With that in mind, let's explore some control statements we use with for and while loops. These are great to ensure that we aren't stuck in a never-ending loop and to do some additional things.

Break

Let's say you have a list of five items. You want the for loop to go through all the iterations and stop when it comes across the value "Orange." You will need to use a break statement to do that.

These statements are beneficial to ensure that the loop doesn't continue forever. When a condition is met, the loop will go to the next line, read break, and stop executing further. This helps prevent your program or system from crashing.

```
cart = ["apple", "banana", "blueberry", "orange", "cheese"]
for x in cart:
    print(x)
```

```
    if x == "orange":
        break
```

Now, if you execute this, you will find the following:

```
apple
banana
blueberry
orange
```

With that said, what do you think will happen if we do the following?

```
cart = ["apple", "banana", "blueberry", "orange", "cheese"]
for x in cart:
    if x == "orange":
        break
    print(x)
```

In this case, Python will only print the first three values and stop the minute it reaches orange. This is because Python goes through its code line by line. Therefore, once it realizes it is dealing with orange, it will understand that the `if` conditional statement is met. It will then move to the following line to check what it must do, and it immediately sees `break`, and that ends the code there. It will not take into account the `print(x)` anymore.

Continue

Besides break, you may sometimes want the for loop to continue even after it has found what you were looking for. In such a case, you can use the `continue` keyword, as shown below:

```
cart = ["apple", "banana", "blueberry", "orange", "cheese"]
for x in cart:
    if x == "orange":
        continue
    print(x)
```

The output, however, will be as shown:

```
apple
```

```
banana
blueberry
cheese
```

Once the value matches the condition, the loop will skip that over and move forward to carry out its process to completion.

Pass

For loops are never meant to be empty. Having one can often cause things to go wrong. However, there will be instances where you may need to write a for loop and then leave it empty until a better use is found or a certain condition is met for them to execute.

For empty loops, we use something called a pass statement. The pass statement lets Python know, "Hey! It's okay! I did that deliberately. Ignore this for now!" Python will skip over it and move on with the rest of the program. How does it work? Quite simple!

```
for x in [10, 11, 12]:
    pass
```

In the above case, Python will skip over this line of code and move on. The pass statement can often be used as a placeholder for future code you will write later. Not having the code at this time is not essential to debugging and testing certain conditions.

Lists and Dictionaries

"Wait, didn't we cover that already?" We did, but we only touched upon what these are and how to create them. Remember we said we'll look into indexing and other aspects later? Now's the time.

Lists and dictionaries, to serve as a refresher, allow us to store multiple values in a single variable. A list can contain numerous values in an ordered manner. A dictionary, on the other hand, contains data in pairs. Lists are created using the [] brackets, whereas dictionaries are created using the { : } format. Now, we will look deeper into these, starting with list operators.

List Operators

You can create a list with an empty value and populate it later. How? Here's how!

```
my_list = []
```

What this will do is create a list variable named my_list. However, it will have no item in it. To add items to the list, we use append. To access this function, we do the following:

```
my_list = []
print(my_list)
print("List is empty")

my_list.append("First item")
print(my_list)
```

Notice how we used a period (.) before the word append. Using a period allows you to access methods and other functionalities available to a given variable. One of the available ones happens to be append. Append is used to add items to a list. It adds the item to the very end of the list. In the above case, the result will now look like this:

```
[]
List is empty
['First item']
```

However, notice what happens here.

```
my_list = []
print(my_list)
print("List is empty")

my_list.append("First item")
print(my_list)

my_list.append("Third item")
my_list.append("Second item")

print(my_list)
```

```
[]
```

```
List is empty
['First item']
['First item', 'Third item', 'Second item']
```

The result shows the first, third, and second items, in that specific order. Why? This is because append only adds one item at a time, and it does so right at the end of the list. This brings us to an important question: how do you remove items from a list? That's simple! Just like append, we can use remove.

```
my_list.remove("Third item")
print(my_list)
```

```
['First item', 'Second item']
```

From there, you can use append to make the adjustments to the list. However, that does seem somewhat time-consuming. Indeed, there must be a more efficient way to remove a value in a particular position in the list. Well, there is.

Let's say you have a long list of items inside a list. You want to remove the item that is entered and stored at position number 4. You don't want to first print out the list, find out what's at number 4, and then restart all over again. Instead, you just want to tell Python "Whatever's in 4th place, remove it." Here's how you can do just that.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
del my_list[3]

print(my_list)
```

"Wait. I said 4th, not 3rd."

We knew you'd say that. However, trust us and execute the code to see what happens.

```
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12]
```

"How?"

This is precisely what we were talking about. Whenever you populate a list, you place each item in a specific index position.

Unlike our numbers that start with 1, the first item in the list has an index value of 0. If you want to remove the 4th item, you'll be targeting index number 3.

index number = n - 1

This is a simple formula for you to remember. This will help you know exactly what index number you want whenever you are aiming to delete or even access individual values. Once you delete the value at index number three, it will replace it with the next element. Therefore, if you were to try and find out what lies at index number 3 after the above, you'll see the following:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
del my_list[3]

print(my_list[3])
```

```
5
```

See what we mean?

Now, let's say you want to remove the value at index position 5, and then you want the system to store it as a separate variable. How do you do that? One way is to print the entire list, find out what value lies at the given index number, and then create a unique variable with that value. This will be time-consuming and somewhat counterproductive as you must remove the value from the list separately to avoid glitches. There's an easier way to do that. We do that by using the pop() function.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

my_num = my_list.pop(5)

print(my_num)
print(my_list)
```

```
6
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12]
```

Python could easily remove the value '6' and store it as a separate variable. With that said, if there's a way to remove specific elements at specific index values, shouldn't there be a way to add specific values at specific positions, too? Well, there is. For that, we use the insert() function.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

my_num = my_list.pop(5)

print(my_num)
print(my_list) # print for comparison
my_list.insert(5, 6) # alternatively my_list.insert(5, my_num)
print(my_list) # includes 6 which was popped earlier
```

```
6
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Through insert(), we could add the value '6' on the 5th index position. Remember, for insert() to work correctly, you must first specify the index number followed by the value you wish to insert.

Let's create another list. This time, we will create a list with random numbers.

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
```

The above list has a lot of elements, and we were not bothered to count how many elements it has. We can ask Python to help us with the len() function.

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
print(len(my_list))
```

This will print out 8, meaning that this list has eight values stored in it. We can do more things with our list as well. For example, we can ask Python to order this list in reverse. To do that, we use the reverse() function.

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
my_list.reverse()
```

```
print(my_list)
```

This will generate the following results:

```
[75, 37, 28, 86, 19, 71, 103, 11]
```

We can also use the sort() function to sort a list in order.

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
my_list.sort()
print(my_list)
```

The list will now be sorted from lower to higher values. You can also ask Python to sort the list out in reverse order by doing the following:

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
my_list.sort(reverse=True)
print(my_list)

[103, 86, 75, 71, 37, 28, 19, 11]
```

Now, the list is sorted in reverse. This can often be very useful in real-life scenarios and programming situations.

Let's say we want to split this list into two separate variables: first_half and second_half. To do that, we will use the slicing method.

```
first_half = my_list[:4] #list[start:end] where we specify end as index 4
second_half = my_list[4:] #we specify starting after index 4

print(first_half)
print(second_half)

[11, 103, 71, 19]
[86, 28, 37, 75]
```

Let's say you now want to combine these two lists. How do you do that? Well, there are a few ways you can do that:

1. You can use the + operator to add together the lists.
2. You can use the for loop to append elements of one list into the other and merge them.

3. Use the extend() function.

The first one is straightforward to do. You simply add the two lists as shown below:

```
my_list = first_half + second_half
```

This will add the two together. The second method is worth looking into as it is a very clever way of adding the lists together using the for loop:

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
my_list.sort(reverse=True)
print(my_list)

first_half = my_list[:4]
second_half = my_list[4:]

print(first_half)
print(second_half)

for x in second_half: #loop will iterate through all elements of second_half
    first_half.append(x) #will add elements of second_half into first_half

print(first_half)
```

Now, your first_half list will be a complete singular list. However, there is a third way to do this as well, and that is by using the extend() function:

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
my_list.sort(reverse=True)
print(my_list)

first_half = my_list[:4]
second_half = my_list[4:]

print(first_half)
print(second_half)

first_half.extend(second_half)
```

```
print(first_half)
```

Once again, your first_half list will have all the elements from second_half.

If you want to remove all the values from a list, you can use the clear() function:

```
my_list = [11, 103, 71, 19, 86, 28, 37, 75]
my_list.clear()
print(my_list)
```

This will take away all the values from the list.

Diving Deeper Into Dictionaries

With the lists sorted, let's see how dictionaries can be accessed, modified, cleared out, and used.

The first thing to note is that dictionaries use keys unlike lists. They do not work with index numbers. Instead, they use keys to recall values.

```
grade_key = {"A" : 90, "B" : 80, "C" : 70, "D" : 60}
min_for_A = grade_key["A"]
print(min_for_A)
```

We created a dictionary with pairs of keys with their values in the above. Key "A" has a value of 90. Key B has a value of 80, and so on. We then created a variable to determine what marks are required to achieve grade A. We assigned this new variable the value of grade_key["A"], which is 90. If you execute the program now, it will show you 90 as the result.

To make it more interesting, let's add another key pair where grade "F" has a value of 0. To do that, we will add the grade value and key using the following method:

```
grade_key["F"] = 0
print(grade_key)
```

```
{'A': 90, 'B': 80, 'C': 70, 'D': 60, 'F': 0}
```

To access a particular value in the dictionary, we rely on keys. Therefore, if you want to find out the value attached to the key pair of C, we will do the following:

```
print(grade_key["C"])
```

This will fetch us the value attached to this key pair. Here, 'C' is the key to access the value. You can also use the get() method to acquire the value, as we did above.

```
print(grade_key.get("C"))
```

This will fetch you the value of the key pair C. You can also find out what keys are stored in a given dictionary, in case you're not the original programmer and wish to find out. To do that, you can use the keys() method:

```
print(grade_key.keys())
```

This will fetch all the keys for you, as shown below:

```
dict_keys(['A', 'B', 'C', 'D', 'F'])
```

Similarly, you can also find all the values stored inside a dictionary. To do that, use the values() method:

```
print(grade_key.values())
```

```
dict_values([90, 80, 70, 60, 0])
```

If you wish to see all the items inside a dictionary, use the items() method:

```
print(grade_key.items())
```

```
dict_items([('A', 90), ('B', 80), ('C', 70), ('D', 60), ('F', 0)])
```

To remove an item from the dictionary, we use the same pop() method as we did in the lists:

```
grade_key.pop("F")
```

This will remove the key pair of "F" and update the dictionary. To clear the entire dictionary, you can use the clear() function:

```
grade_key.clear()
```

This will remove all the key pairs from the dictionary, leaving you with an empty dictionary that you can populate again.

Exercises

Now, it's time to bring all that we have learned and put it to some use. Below is a sample code that you can alter to your liking. The goal is to create a program on your own that has at least one working loop and at least one conditional statement.

```
# Import the random library to generate random numbers
import random

# Initialize variables
target_number = random.randint(1, 100) # Generate a random number
between 1 and 100
user_guess = 0
attempts = 0

# Welcome the user to the game
print("Welcome to the Number Guessing Game!")

# Use a while loop to allow multiple guesses
while user_guess != target_number:

    # Capture the user's guess using the input() function
    user_guess = int(input("Guess a number between 1 and 100: "))

    # Increment the attempts counter
    attempts += 1

    # Use a conditional statement to check the guess
    if user_guess < target_number:
        print("Too low! Try again.")
    elif user_guess > target_number:
        print("Too high! Try again.")
    else:
        print(f"Congratulations! You've guessed the correct number
        {target_number} in {attempts} attempts.")
```

```
# End of the game
print("Thank you for playing the Number Guessing Game!")
```

With that said, let's move on to the next chapter and learn all about functions and modular programming.

Chapter 4: Python Functions and Modular Programming

> *"Don't repeat yourself. Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."* **–Anonymous**

Writing code is fun but can quickly become annoying and cumbersome. This is particularly true when you write large blocks of code repeatedly. If only there were a way to somehow store all that massive block of code into a single line you could recall whenever you wanted. Fortunately for us, there is.

In this chapter, we will learn not only how we can store these blocks of code but also how to use them properly, and to do that, we start by understanding what functions are.

Introduction to Functions

A function is a large block of code designed to carry out a specific job or function. As it turns out, Python has many built-in functions that come pre-loaded. This means you already have hundreds, if not thousands, of functions at your fingertips. What's more, Python allows you to create custom-made functions to help you carry out extensive operations through a single-line function recall.

Functions are reusable, meaning you can use them in a single program as often as you like. These are created to serve particular functions. For example, the print() function prints out information on the console.

Functions also serve another essential purpose; they organize your code correctly and make it more manageable. Yes, the first time you define a function (create), you must type in a few lines, but after that, you'll only use the function's name followed by the parenthesis (). Let's say you have around 50 lines of code to define a custom

function you want to use in your program. The next time, you will use only the name, such as:

```
my_func()
```

With that single line and a few characters, you will recall and execute all the lines without ever having to copy and paste them repeatedly. This helps you save time, energy, and resources.

Besides, functions are great for debugging, a process where you go through code to find out what's not working.

The Syntax of a Function

When creating a function, we use the keyword def, which stands for define. Whenever you define a function, all the code is automatically indented. This is to ensure ease and to facilitate debugging later on. Let's create a sample function to print "Hello Everyone!"

```
def my_func():
    print("Hello Everyone!")

my_func()
```

We first defined a new function in the above code and named it my_func. Next, we coded what we wanted this function to do in the indented code. Once that is done, we type in the name of the function, and it carries out whatever we want it to do. It's that simple.

Let's have a little fun with function by creating a simple game where the computer tells you if your chosen number is odd or even.

```
def my_func():
    var1 = int(input("Enter a number: "))
    if var1 % 2 == 0:
        print("You have chosen an even number")
    else:
        print("You have chosen an odd number")

my_func()
```

We used a simple conditional statement where if the chosen number is perfectly divisible by two and has 0 as a remainder, it is an even number. Anything else is an odd number. Go ahead, try it out!

Generally speaking, most applications and games start with several functions well before the app is used. This is called pre-loading, and this is usually done using functions. Programmers code in functions and call these functions the moment the app is opened.

When you define a function, you can also pass something called arguments. Arguments are additional information or directions for the compiler to follow. The basic syntax to do that looks something like this:

```
def new_func(arg1, arg2):
    #code goes here
```

For example, you have a function where you want to add two numbers together that you input when calling the function. To do that, you first need to define the function in the following manner:

```
def addTwoNum(num1, num2):
    print(num1 + num2)
```

This creates a function where it takes in two arguments, which are two separate numbers. The function is coded to print out the sum of the two numbers. Now, let's recall and see what's what:

```
addTwoNum(19, 22)
```

```
41
```

Do you see how that worked out? Every time you recall this particular function, you must pass two values through for this function to work. This is because we have set two arguments as defaults. We can set as many arguments as we want.

Let's make another function and use what we call a return statement:

```
def multiNum(num1):
    num1 * 10
```

```
res = multiNum(21)

print(res)
```

In the above, we defined a function called multiNum. We passed a single argument. The function will take a value as an argument and multiply that by 10. Next, we created a variable called res to store the resulting number as its value. However, if you execute this code, you end up with the following result:

```
None
```

Why is that? That is because we haven't defined whether it must return a number or a value within the function. In the case of the print() function we used earlier, it was an explicit instruction to print. Sometimes, you may not want the function to print out values. Instead, you'd want it to return a value and send it outside the function. To do that, we'll need to use the return statement, as shown here:

```
def multiNum(num1):
    return num1 * 10

res = multiNum(21)

print(res)
```

Now, the function will return the value of num1 * 10. While it will still not print it, it will store that value successfully in the variable res. Since it exists outside the function block, res will be successfully printed. Try it!

Parameters and Return Values

Functions have both arguments and parameters, and these terms are often used interchangeably. While that isn't completely wrong, they do technically vary from one another.

Arguments are variables passed within the parentheses after the function's name when you are defining the function. There can be as many arguments as you like, each separated by a comma.

```
#arg1 and arg2 are arguments
def sample_func(arg1, arg2)
```

Parameters, on the other hand, are almost identical. The major difference is that these variables pass through a function when the function is called. This is the key and probably the most noticeable difference. Therefore, it is a good idea to familiarize yourself with these two terms to avoid confusion in the future.

```
#par1 and par2 are parameters used during recall of function
sample_func(par1, par2)
```

You can also create functions with default parameter values. Doing so will allow you to call up a function without an argument. In this case, it will use a default value you've passed while defining the function.

```
def my_func(city = "Austin"):
    print("I hail from " + city)
```

```
my_func("New York")
my_func("Chicago")
my_func("Claymont")
my_func() #Here, the default value will be used
```

```
I hail from New York
I hail from Chicago
I hail from Claymont
I hail from Austin
```

Functions With No Parameter

You can also create functions with no parameters. To show this, let's start by creating a dictionary of items for groceries. We will then create a simple function to print out the list of items with their quantities.

```
s_list = {"bread": 2,"eggs": 12,"coffee": 1,"bananas": 6,"sugar": 1}
```

```
def show_list():
    for item, quantity in s_list.items():
```

```
        print(f"{quantity} x {item}")

show_list()
```

Now, as soon as you execute the program, you will be presented with a list that looks like this:

```
2 x bread
12 x eggs
1 x coffee
6 x bananas
1 x sugar
```

In such cases, you don't need to pass any parameters or arguments to a function. Depending on what you aim to do with your function, you can decide whether or not the function requires additional input. To do that, it is recommended that you know exactly what your function is meant to do. The more precise you are, the easier it will be to create a function.

Scope and Variable Lifetime

You may have noticed earlier that we use some variables generally while some are only limited to loops or conditional statements. This is because every variable we create in Python has its scope.

When it comes to Python, the scope is the region of a program where a given variable can be used and accessed. In the case of Python, there are two types of scope variables. These are:

1. Global
2. Local

Global scoped variables are defined within the main program and remain outside the function's body (not defined inside a function's block of code). These variables will remain visible throughout the program and can be accessed at any given time.

A global variable can also be accessed when you are defining a new function, as shown below:

```
#Global variable
a = 20

#declaring a function with local variable
def showMe():
    print("Local scope =", a)

showMe()

print("Global scope =", a)
```

```
Local scope = 20
Global scope = 20
```

A local scoped variable, however, is limited in terms of access and reach. These are generally defined or created during the creation of a function. This variable, being local, cannot be accessed outside a function, meaning you cannot independently use it anywhere else but within the function itself.

```
#local variable
def showMe():
    a = "Morning time"
    print("Local scope =", a)

showMe()

print(a)
```

```
Local scope = Morning time
Traceback (most recent call last):
print(a)
NameError: name 'a' is not defined
```

The error above shows that the variable 'a' is not defined. This is because it was called outside the function's body and never existed in the rest of the program.

There is also a third type of scope called non-local, which exists inside a nested function. A nested function is one where you are defining a function within a function. In such a case, the nested

function's variable will be termed a non-local variable.

Exercise

Using all that we have learned, create a simple function that calculates the average of given numbers and returns the value that can be stored in a variable.

To give you an example, here's a sample to seek inspiration from:

```
def average(numbers):
    return sum(numbers) / len(numbers)

print(average([1, 2, 3, 4, 5])) # Output should be 3.0
```

Simple Calculator

This one is slightly challenging but quite fun. The goal is to create a simple calculator using your coding knowledge. To help you get started, here are the steps you need to take, followed by a sample code:

1. Initialize variables.
2. Create functions for arithmetic operations (addition, subtraction, multiplication, division).
3. Create a menu.
4. Capture the user's input.
5. Create a main program loop.
6. Complete the code.

```
# Initialize variables
first_number = 0
second_number = 0
result = 0

# Function for addition
def add(x, y):
    return x + y

# Function for subtraction
```

```
def subtract(x, y):
    return x - y

# Function for multiplication
def multiply(x, y):
    return x * y

# Function for division
def divide(x, y):
    if y == 0:
        return "Cannot divide by zero."
    return x / y

# Function to display the menu
def display_menu():
    print("Simple Calculator Menu:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Exit")

# Function to capture user input
def get_numbers():
    global first_number, second_number
    first_number = float(input("Enter the first number: "))
    second_number = float(input("Enter the second number: "))

# Main program loop
while True:
    display_menu()
    choice = input("Choose an option (1-5): ")
    if choice in ['1', '2', '3', '4']:
        get_numbers()
        if choice == '1':
            result = add(first_number, second_number)
```

```
    elif choice == '2':
        result = subtract(first_number, second_number)
    elif choice == '3':
        result = multiply(first_number, second_number)
    elif choice == '4':
        result = divide(first_number, second_number)
    print(f"The result is: {result}\n")
elif choice == '5':
    print("Goodbye!")
    break
else:
    print("Invalid choice. Please try again.\n")
```

With that, it's time to move on to the next chapter and learn all about the advanced data structures we use in Python. They're essential, and they are certainly going to come in handy.

Chapter 5: Advanced Data Structures in Python

"Data is the new oil." **–Clive Humby**

While we learned a little about tuples and sets, they are more powerful and somewhat more advanced to cover in just a few paragraphs. That is why we didn't dive too much into those in the previous chapters. This ensured we built a solid foundation and understood some core concepts before diving into these.

When you want to take Python programming up a notch, you must use tuples and sets. Therefore, this chapter will walk you through both, reinforce previously learned knowledge, establish new concepts, and help you understand these better.

Tuples

Tuples, unlike lists, are immutable, meaning they can neither be altered nor modified. Once you have created a tuple, the container is sealed. What remains inside remains with no room for any additional tampering or addition. You can still use the values stored inside tuples, which is why these are handy when you want to use specific values that you know must remain constant.

When you create a tuple, you need to remember that a tuple uses round brackets, just like functions do. You will also need to ensure that you create the tuples correctly as errors can lead to problems. To highlight just how mistypes can cause issues, here's an example:

```
mytup = ("Cat",)
print(type(mytup)) #type() shows the data type of the variable
```

```
mytup = ("Cat")
print(type(mytup))
```

Before you execute the code above, what do you think the outcome will be? Will both the variables be reflected as tuples? Well, try it out and see what happens.

```
<class 'tuple'>
<class 'str'>
```

Surprised? Let me explain. Whenever you are creating a tuple, you not only use the round brackets, but you also need to use a comma. You do this even if you are storing a single value in it.

"Can't I just initialize a tuple and populate it later?"

Recall the critical difference between a tuple and a list: A tuple is immutable, meaning once it is created, it cannot be altered, modified, or changed in any way. We can use the tuple to check the values it stores, use them, and call upon them, but we cannot append, remove, or pop any values.

This is why it is imperative to think through all the scenarios beforehand when creating a tuple. You don't want to create a tuple to find out you need to add more variables or alter values later. Once it is created, it cannot be altered.

Accessing Items Inside a Tuple

This works quite like the list. This means you must rely on the index numbers to find and access individual items within a tuple. There are two ways you can do this:

1. Using positive index.
2. Using negative index.

Let's create another tuple and add five elements to that. These are five random numbers.

```
myTup = (15, 17, 9, 237, 44)
print(type(myTup)) #to verify that myTup is indeed a tuple
```

Let's see how we can print out individual elements at index numbers 0, 2, and 4.

```
myTup = (15, 17, 9, 237, 44)
print(type(myTup))
```

```
print("Value in myTup[0] is ", myTup[0])
print("Value in myTup[2] is ", myTup[2])
print("Value in myTup[4] is ", myTup[4])
```

```
<class 'tuple'>
Value in myTup[0] is 15
Value in myTup[2] is 9
Value in myTup[4] is 44
```

That's done quickly through the use of positive index numbers. However, what's the deal with negative index numbers? Do they even exist? Let's find out.

For this example, we will use the same code above. However, instead of the index values 0, 2, and 4, we will use -4, -2, and -1 to see what happens. You can use other values, too, if you like.

```
myTup = (15, 17, 9, 237, 44)
print(type(myTup))

print("Value in myTup[-4] is ", myTup[-4])
print("Value in myTup[-2] is ", myTup[-2])
print("Value in myTup[-1] is ", myTup[-1])
```

```
<class 'tuple'>
Value in myTup[-4] is 17
Value in myTup[-2] is 237
Value in myTup[-1] is 44
```

Do you see what happened here? The order changed. Instead of going from left to right, the system is now going from right to left. This can often prove to be helpful if you have a larger tuple.

Tuples are great for yet another reason. They are extremely fast when compared to lists. This is because tuples do not occupy a lot of memory and are significantly faster in operation and recall. If you know your variables and which ones will remain unchanged throughout the program, store those in tuples where and when possible.

Sets

We also looked at sets briefly at the start of the book. Lists are ordered collections of items. Sets, however, are an unordered collection of a variety of items. Unlike tuples, these can be created

and then modified. However, the elements, once created, remain immutable. You can still add and remove from the sets, but the stored values cannot be altered or modified.

That said, it is important to note that sets will not take in and store any duplicate items. Therefore, if you try to store a value already in a set, it will end up in an error.

You use the curly brackets, or the {}, to create sets. This does not include any colon ":" symbols as you find in a dictionary. Like tuples, sets are highly optimized and are much more efficient at checking elements inside a set when compared to a list. These are more efficient and do not take up many resources.

The significant difference between sets and lists is that list items can be accessed using the index numbers. However, you cannot use index numbers to access items inside a set. This is because sets are unordered. Anything that is unordered cannot be accessed using index values.

Let's create a sample set for ourselves and use that to see how things work.

```
mySet = {"one", "two", "three", "four"}
print(type(mySet))
```

```
<class 'set'>
```

Modifying Sets

Let's start by looking at the **add()** method.

```
mySet = {"one", "two", "three", "four"}
print(type(mySet))
print(mySet)
```

```
mySet.add("five")
print(mySet)
```

```
<class 'set'>
{'two', 'one', 'four', 'three'}
{'three', 'two', 'one', 'five', 'four'}
```

Your values will continue to jump around every time you run the code. As we mentioned earlier, this list has no specific order, hence the unordered tag. Since the values keep hopping around, there is no way to know which index number a particular value will be on the next time, which is also why you cannot access items within a set using index numbers.

To remove an element from the set, you use the discard() method:

```
mySet.discard("four")
```

This will discard the value of "four" and update the set accordingly.

Things are a little tricky to create an empty set. You see if you were to do the following:

```
newSet = {}
```

That will create an empty dictionary, not a set. To create an empty set, you use the set() function, as shown here:

```
newSet = set()
```

This will initialize a new and empty set that you can populate as you go along.

List Comprehension in Python

We know how to manually create a list, but there's one more way. You can create a list from an existing list or other iterables within your code through a process called list comprehension.

Every list comprehension contains:

1. Some input sequence.
2. Variable that stores members of the above sequence.
3. A predicate expression.
4. A viable output expression (creates an output list while satisfying the predicate expression).

The basic syntax of list comprehension:

```
[ <expression> for item in list if <conditional> ]
```

```
new_list = [x for x in range(1, 9)]
```

```
print(new_list)
```

The numbers within the range will be stored as items in the new list. Notice how we didn't use any `if` condition here. Let's change that and see what's what.

This time, we will use the following snippet of code:

```
new_list = [x for x in range(1, 20) if x % 2 == 1]
print(new_list)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In the above case, instead of storing all the numbers in the range, Python filtered through the numbers and stored only the ones that satisfied the condition. The `if` condition stated to store only the odd numbers or that had 1 as a remainder after division.

Set Operations

We now know that sets are unordered, meaning they cannot be accessed using index numbers. We also know that while the set can be mutated by adding or removing values, the values cannot be changed once stored. However, that's not all there is to learn about sets. There are other things sets can do for you, just like they did in school.

Recall how we once learned about union, difference, intersection, and symmetric difference. Well, we can do that with sets in Python, too, making sets all the more useful than ever.

Set Union

Let's create two sets for this and see how the `union()` method can be used.

```
set_one = {1, 2, 3}
set_two = {4, 5, 6}
new_set = set_one.union(set_two)
```

```
print(new_set)
```

In the above, we have two sets with their values. We then initiated a new variable called new_set, and for this, we want to merge the two previous sets into one through the union() method. If you execute this code, you will see that the new_set holds all the values from both sets.

It is worth noting that the union() does not modify set_one or any other set. Its function is only to unite the two sets. You need to store the values in a new set variable to acquire them. Once again, Python will automatically convert the variable into a set once these values are stored.

There's another way you can do the above without using the union() method. Instead, you can use the | key, as shown below:

```
new_set = set_one | set_two
```

This will do the same thing.

Set Intersection

The set intersection takes in sets and identifies the common elements that these sets share. What it does, then, is allow you to create a new set of all the common values. Once again, there are two ways to do this. You can use the intersection() method to carry this operation out, or you can use the & operator and still do the same thing.

```
set_one = {1, 2, 3, 7, 19, 21}
set_two = {4, 5, 7, 21, 28}
new_set = set_one.intersection(set_two)
```

```
print(new_set)
print(type(new_set))
```

```
{21, 7}
<class 'set'>
```

In the above, the two common elements were identified and then stored in the new_set variable. You can also use the & operator to do this, as shown here:

```
set_one = {1, 2, 3, 7, 19, 21}
set_two = {4, 5, 7, 21, 28}
new_set = set_one & set_two
```

```
print(new_set)
print(type(new_set))
```

```
{21, 7}
<class 'set'>
```

Set Difference

The set difference is the opposite of what set intersection does. As the name suggests, this computes all the elements in sets and identifies the ones that are different.

You will use the difference() method to carry out this operation. Alternatively, you can use the - operator:

```
set_one = {1, 2, 3, 7, 19, 21}
set_two = {4, 5, 7, 21, 28, 30}
new_set = set_one.difference(set_two)
```

```
print(new_set)
print(type(new_set))
```

```
{19, 1, 2, 3}
<class 'set'>
```

Notice how some elements at the end of set two aren't displayed in this set. This is because the first set stopped at 21, meaning it did not have additional elements to compare with. You can, however, do the following using the - operator:

```
set_one = {1, 2, 3, 7, 19, 21}
set_two = {4, 5, 7, 21, 28, 30}
#
new_set = set_two - set_one
print(new_set)
print(type(new_set))
```

```
{28, 4, 5, 30}
<class 'set'>
```

Set Symmetric Difference

The symmetric difference considers all such elements present in the first or the second set but not in both. To check the symmetric difference between the two sets we have created, we can use the symmetric_difference() method. Alternatively, we can also use the ^ operator:

```
set_one = {1, 2, 3, 7, 19, 21}
set_two = {4, 5, 7, 21, 28, 30}
```

```
new_set = set_one.symmetric_difference(set_two)
#Use the comment to disable the above before removing the comment symbol below
#new_set = set_one ^ set_two
```

```
print(new_set)
print(type(new_set))
```

```
{1, 2, 3, 4, 5, 19, 28, 30}
<class 'set'>
```

Exercise

For this chapter, create a list using list comprehension. This list can be about anything you like. As a challenge, try and find ways to manipulate a tuple.

With that, it's time for us to move forward to the next chapter. In the next chapter, we will learn one of the most defining features of Python and understand what exactly OOP is all about.

Share Your Thoughts, Shape a Beginner's Journey

Your Review Can Spark a Coding Adventure

"Sharing knowledge is the most fundamental act of friendship. Because it is a way you can give something without losing something." - Richard Stallman

Imagine you're back at the starting line, curious and eager, with a world of coding ahead of you. Remember the excitement? The slight nervousness? That's where many are today, at the threshold of their Python programming journey, holding *The Ultimate Guide to Python Programming for Beginners* in their hands, wondering if it's the right guide for them.

Our goal at Megabyte Publishing is simple: make Python programming a fun, accessible adventure for everyone. Every step we take is driven by this goal. To reach every aspiring programmer out there, we need your help.

Here's the deal: book covers and reviews are what catch a beginner's eye. So, we have a small, yet significant request for you on behalf of a future coding star you haven't met yet:

Please share your thoughts about this book with a review.

Your review doesn't cost a penny, and it takes less than a minute. But your words have the power to inspire, guide, and reassure a newcomer in the world of Python programming. Your insights could be the nudge that helps...

- A young student discover their passion for technology.
- An aspiring developer take the first confident step in their career.
- A creative mind find a new way to express their ideas.
- A curious soul unlock a new hobby that brings them joy.

To leave your mark and make a difference, just scan the QR code below and share your review:

If the thought of guiding a budding programmer excites you, welcome to our community! You're now part of a special group committed to spreading knowledge and joy.

I can't wait to support you as you continue your Python programming journey with even more amazing tips, tricks, and insights in the upcoming chapters.

Thank you for your generosity and for being a guiding star in someone else's coding journey.

- Your enthusiastic guide, Megabyte Publishing

P.S. - Did you know? Sharing knowledge not only helps others but also reinforces your own learning. If you believe this book could benefit another aspiring Python programmer, why not spread the word? Your recommendation could be the start of their coding adventure!

Chapter 6: The Basics of Object-Oriented Programming (OOP) Using Python

> *"Object-oriented Programming offers a sustainable way to write spaghetti code." – Paul Graham*

Python is an Object-Oriented Programming language. That is the introduction that you will likely come across every time you search for Python. The fact is that not every language is an OOP-based programming language. Fortunately for us, we are learning a language that is, and it is pretty good at that, too.

So, what's the fuss about OOP? What exactly is an object? Why should it matter that Python is an OOP language? Well, this chapter is where we learn all that and learn more about what OOP is and why we should know all about it.

Basics of Python OOP

Object-oriented programming is a utility, power, or paradigm that uses various objects, classes, and more for programming. We can implement real-world entities in our code through the power of OOP. These can include:

- Polymorphism
- Inheritance
- Encapsulation
- And more

The entire goal of OOP is to bring data and functions together as a single unit. This is done so that no other part of the code allows other programmers to access data.

When it comes to OOP, there are six essential components that make up OOP:

1. Class
2. Objects
3. Polymorphism
4. Inheritance
5. Encapsulation
6. Data Abstraction

Considering the scope of this book, we will only be covering the first four aspects, as the latter two can be pretty advanced.

Classes and Objects

Let's first start by understanding objects. An object is nothing more than a code entity that a programmer creates. Every object has attributes and behaviors that allow it to function in a specific manner.

A class, on the other hand, is the blueprint for objects. It is this blueprint that then facilitates the creation of objects. As it stands, a class is a logical entity and holds some methods and attributes.

Many classes may already exist in Python, but you also have the power to create your own classes as you see fit. Let's take the following example to give you an idea of how vital and helpful classes are.

Let's say you want to find out the age and the breed of dogs. Generally, you would create two lists. One will handle the breed, and the other will hold their ages. However, if you were to deal with over 500 dogs, how would you handle that? Creating lists after lists is undoubtedly not how you want to do things, as that can be very hectic and messy regarding clean code writing. Instead, you create classes.

To create a class, there are four rules to remember:

1. A class is always created by using the keyword class.
2. All attributes assigned to the class are variables that belong to this class.

3. Attributes are public, meaning these can be accessed easily using the dot operator (.).
4. A class's name must have the first letter of every word in uppercase.

With that said, let's look into the basic syntax of how a class is created:

```
class SampleClass
    #Your First statement goes here
    .
    .
    .
    .
    .
    .
    #Your nth statement goes here
```

You can also create empty classes by using the pass statement:

```
class SampleClass:
    pass
```

Objects

Now that you know the classes, we move on to the next step and learn all about objects.

We know that objects are created thanks to classes. Their behavior is linked to their parent classes. Everything we have learned about thus far, such as variables, dictionaries, lists, integers, and strings, are all objects.

Objects consist of a few things:

1. Identity—Every object has a unique name that allows the object to interact with other objects.
2. State—The object's state is represented by the attributes assigned to the object. The state also shows the properties of the object in question.

3. Behavior—The methods of the object reflect the behavior of the object.

Let's create a class and call it Dog. We will need this class to create objects to see how the state, behavior, and identity of the object work:

```
class Dog:
    # class default attribute
    attr1 = "furry"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Creation of object
Bruce = Dog("Bruce")
Tom = Dog("Tom")

# accessing attributes from the class
print("Bruce is very {}".format(Bruce.__class__.attr1))
print("Tom is also very {}".format(Tom.__class__.attr1))

# accessing instance attributes
print("This is {}".format(Bruce.name))
print("This is {}".format(Tom.name))
```

In the above code, we started by creating a class named Dog. First, we set a default attribute or state, "furry." Next, we defined an instance attribute. An instance attribute is an attribute that only belongs to one object. This means that as long as the objects are created using the same class, only they can access this attribute. Any other object that isn't created using the same class will not be able to access this attribute.

The __ init __ is called a constructor and is a special method used to initialize an instance of a class. By default, it has one parameter (self). Self is a reference to the instance being created. In our case, we added one more parameter, name. This attribute will be applied to

every instance of the class.

Next, we created two objects using the following syntax:

```
object = class()
```

In our case, we created Bruce and Tom. We passed the names as attributes while calling upon the class. That name will only apply to individual objects and cannot be shared with others.

Finally, we printed the class attributes for each object followed by their names (using the instance attributes).

"That's quite complicated."

Don't worry. We have a simpler way of doing this to get you into the zone. Let's look at another example. For this, we will create a new class named Cat. This time, we won't be using many complicated ways. Instead, we'll use a far simpler approach:

```
class Cat:
    # class attribute
    name = ""
    age = 0

# creating cat1 object
cat1 = Cat()
cat1.name = "Casper"
cat1.age = 2

# creating cat2 object
cat2 = Cat()
cat2.name = "Whiskers"
cat2.age = 5

# accessing attributes
print(f" {cat1.name} is {cat1.age} years old") #print(f"") is formatted
printing - easier to do
print(f" {cat2.name} is {cat2.age} years old")
```

"Yes. This seems simpler."

It really is. In this example, we created a class, gave it some empty but straightforward attributes, and then headed straight to the code. We created two objects and then defined their attributes further. Finally, we printed information out using the attributes of each object.

Creating Class Methods

Objects have attributes, but they also have methods. These are class methods, and they are defined just as functions. The only difference is that these are defined slightly differently:

```
# Adding a method to the Dog class
class Dog:
    def __init__(self, name):
    self.name = name

    def bark(self):
        print("Woof!")

# Using a method
my_dog = Dog("Buddy")
my_dog.bark()
```

Here, we created a class, defined it using the constructor, and then defined a method. Since it was created in this class, this method will now be exclusive to this class and its related objects only.

Inheritance and Polymorphism

Inheritance is precisely what it sounds like. It establishes a relationship between any two given classes. One class is the superclass, better known as the parent, while the other is the subclass, better known as the child.

Every child keeps all the attributes and methods as inheritance from its parent class. Furthermore, it can add new attributes and methods, allowing it to evolve and grow more powerful. Where required, a child class also can override methods of the parent's class as well:

```
class Father(object):
    def buy_pizza(self):
        print("yummy")

class Son1(Father):
    pass

class Son2(Father):
    pass

Father().buy_pizza()
Son1().buy_pizza()
Son2().buy_pizza()
```

In the above, we created a parent class named Father and passed object as its parameter. Next, we defined a simple method, buy_pizza and asked it to print the word yummy.

Then, we created two more classes and passed Father as their parameter. What we did there was tell Python to designate these classes as child classes. As a result, these classes ended up inheriting everything the parent class had. In this case, it was a method.

Once we called the method for each class, they all printed the same message. This is because the child classes inherited the same method, values, and everything else. Besides, a father will always buy enough pizzas for everyone!

POLYMORPHISM

When child classes are created, they inherit all the methods and attributes from their parent class. Naturally, child classes can also end up having their own unique attributes and methods. However, a concept in Python allows child classes to change methods in their parent classes. That process is called polymorphism.

Through polymorphism, you can use a subclass and alter the method originally defined in the parent class. Sounds somewhat impossible, right? Let's see how this works:

```
class Father(object):
    def buy_pizza(self):
        print("yummy")

class Son1(Father):
    def buy_pizza(self):
        print("I want more!")

class Son2(Father):
    def buy_pizza(self):
        print("I don't like pizzas!")

Father().buy_pizza()
Son1().buy_pizza()
Son2().buy_pizza()
```

```
yummy
I want more!
I don't like pizzas!
```

This is how polymorphism works. It is a great tool, especially when you have a lot of child classes, and you'd like to change the behavior of a given superclass from its subclass.

Extending Classes

Let's say you have a class with some methods and attributes you know you need. However, as you continue with your program, you feel like you want another class with all these features. Still, you wish to further add functionality and modifications to the existing methods. One might think, "Ah, yes! You need to create a child class." However, a child class is not the answer here, not on its own.

A child class does inherit everything, but it won't let you add more functionality to the already existing methods and attributes. A child class can undoubtedly make use of polymorphism, but that's just going to change the superclass-defined method. It won't add any additional functionality. That's where you will need to rely on extending your class.

Extending a class allows you to utilize all the existing methods and add more features and functionalities. How do you do that? It's quite easy. Follow these steps, and you'll be able to extend your classes on the go:

Set up a parent class. Be sure to add all the methods and attributes here.

Set up and define a child class. This child class will automatically inherit all the methods and attributes from the parent class.

```
class Parent:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, my name is {self.name}")

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def greet(self):
        super().greet()
        print(f"I am {self.age} years old")

child = Child("Alice", 10)
child.greet()
```

In the above code, we created a parent and a child class. The child class inherited all the attributes and methods. However, notice how we re-defined the "greet" method using the `super()` method. The `super()` method is used to call the superclass's version of the method. Not only did we retain the original method, but we also added extra functionality. The results for this code will look like this:

```
Hello, my name is Alice
I am 10 years old
```

Method Overriding

Using your subclass, or child class, you can morph the original method and override it completely. For this to happen, there are a few things that are required:

1. The method in the subclass must have the same name.
2. It should have the same number of parameters.
3. It should have the same return type.
4. The subclass must have inheritance.

The question here is why. Why would anyone want to override a method?

1. Overriding method allows you to use functions and methods with the same name.
2. It allows you to change the behavior as well as the implementation of the existing method.

Here's a typical example of how a method override takes place:

```
# parent class
class Father:
    # some random function
        def someFeature(self):
        print('Function defined in parent class!')

# child class
class Son(Father):
    # empty class definition
    pass

obj2 = Son()
obj2.someFeature()
```

```
Function defined in parent class!
```

Exercise

Now that you know all about classes and objects, let's put your knowledge to the test. Below, you will find an incomplete code snippet. The goal is to complete this code and make it work:

```
# Define the Book class
class ___:
    def __init__(self, _____, _____):
        self._____ = _____
        self._____ = _____

    def _____(self):
        print(f"The book '{self._____}' is written by {self._____}.")

# Create an object of the Book class
my_book = _____("To Kill a Mockingbird", "Harper Lee")
# Call the describe method to print the book's details
my_book._____()
```

As you can see, the code is incomplete. Here are some hints to get you started:

1. Define a class named Book.
2. Create an object for the class.
3. "Book" class should use ___ init ___ method to set "title" and "author" attributes.
4. The "Book" class should also have a " describe " method.
5. Method "describe" will print/output the book's details.

With that out of the way, let's move on to the next chapter, where we will learn about another important aspect of Python programming—Error handling and debugging.

Chapter 7: Error Handling and Debugging in Python

> *"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."* **–Brian Kernighan**

Whenever you code a program, software, or application, one thing is for certain—there are bound to be errors and crashes. This isn't because you're a beginner, either. The fact is that even the most experienced programmers go through these issues now and then. What truly helps a professional programmer stand out is their ability to spot these errors during the coding process and then handle them accordingly.

Some errors occur due to some issues with the code. However, some other errors emerge for some other reason completely. Either way, programmers respond to these and conduct recovery procedures to ensure the program doesn't crash.

Think about it this way. Whenever you are using your computer, you often end up encountering errors. Does that mean your system crashes completely? Not likely. Your system only crashes when something terrible has happened. In most cases, these errors are picked up and blocked off, and all that happens is a prompt emerging, telling you what possibly went wrong.

Python is no exception to the rule. Whether you code for the first time or the nth time, errors are bound to happen. What you can do, however, is to handle these accordingly, and that is what we will look into in this chapter.

What Is Error Handling in Programming?

Errors happen due to software or hardware. Sometimes, they can happen because both issues emerge simultaneously. Error handling is the art of understanding these errors and ensuring that something's done to allow the application or the program to resume

function while jumping over the error. Since we aren't learning about hardware, we won't be covering how to handle hardware-related errors.

When it comes to software, a programmer can step in and do one of two things:

1. Use some software tool to catch these errors and handle them.
2. Write some code to block off such anticipated errors.

Either way, effective error handling allows a programmer to catch errors and ensure the program continues to operate without terminating. This helps save data, time, and resources.

Speaking of errors, a programmer must be familiar with four main categories of errors. These are:

1. Logical errors
2. Runtime errors
3. Compile-time errors
4. Generated errors

One of the most important things a programmer of any level needs to do is to proofread their code. This is extremely important because when we are preoccupied with coding, we skip a comma here and mistype something there, and just one slightest error can leave you with an unusable code. It can often drive programmers on the edge of sanity to find out what went wrong. Fortunately, many tools can help you identify these errors. Some IDEs, such as Pycharm, have built-in features that help identify potential errors while you are coding. This means you won't have to write the code, execute it, develop an error, and then try to find out where the problem is. You fix the error on the fly.

Logical errors are usually bugs or the use of incorrect logic. This requires debugging by proofreading the code and ensuring everything is working. For runtime errors, there are error-handling

applications that can help to either resolve these issues or minimize their impact. This is done by adopting some countermeasures. As it turns out, most hardware applications come with error-handling mechanisms.

The fact is that errors can be a bit of a trouble to deal with. These can have minimal impacts, but some errors can be fatal (to the system or the application). These errors can lead to system crashes and damage to the hardware itself.

What Is Debugging?

Error handling is anticipating when and where an error may occur. Debugging, however, is the process of going through the code and then fixing the problem. It is a multi-step process that requires a programmer to identify the problem first. Once the problem is identified, it is isolated from the rest of the program to test. Once the findings are confirmed, programmers try to correct the piece of code or try to find a way to create a way to bypass that issue. After this, programmers test the final results to ensure they work seamlessly.

As the name suggests, the general goal of debugging is to find bugs in codes and software. This is why many software companies and gaming studios often pay people to find glitches and bugs in their products. It is an integral part of programming and can often be considered a skill that programmers can further enhance and pursue.

Understanding Exceptions

When it comes to Python, there are two types of errors that you come across. These are called:

1. Syntax error
2. Exceptions

The syntax errors are quite straightforward. They happen when you forget to complete the code, perhaps close the parenthesis, or even mistype the name of a function. When such an instance happens,

Python throws in a syntax error. Here's a sample of such an issue. See if you can spot the problem here:

```
# Scoring Software
name = input("Please Enter Your Name: ")
print("Hello, ", name + "! Welcome to Our Grading Software!")
marks = int(input("Enter your marks: "))
print("Fetching your grade. Please wait!")
grade = ""
if marks >= 90:
    grade = "A"
    print("Your grade is: ", grade)
elif marks >= 80:
    grade = "B"
    print("Your grade is: ", grade)
elif marks >= 70:
    grade = "C"
    print("Your grade is: ", grade)
elif marks >= 60:
    grade = "D"
    print("Your grade is: " grade)
else:
    grade = "F"
    print("Your grade is: ", grade)

print("Thank You for Using This Software!")
```

When you run this, you end up with the following error!

```
SyntaxError: invalid syntax.
```

If you weren't able to find the problem, look at this line:

```
elif marks >= 60:
    grade = "D"
    print("Your grade is: " grade)
```

Here, there is a comma missing. Just because of a simple comma, the entire program failed to function and could not move ahead. It

can be frustrating when you are coding and forget something as small as a comma. However, with most IDEs, you can find the issue, and some IDEs even hint at what may be wrong. Here's another example of a simple line of code that went wrong:

```
print("Hi. My name is" name)
```

```
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

In the above, we deliberately did not use a comma. As a result, the syntax was wrong. IDLE could pick that up and suggest that we may have forgotten to use a comma, which was precisely the problem.

While syntax errors are easy to fix, the following type of error causes issues and needs a deeper understanding.

Exception errors occur when you have done everything correctly, as per the syntax, but their functionality is incorrect. Let's take the following code as an example:

```
name = input("What's your name?")
What's your name? Alex
print(Name)
```

```
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
print(Name)
NameError: name 'Name' is not defined. Did you mean: 'name'?
```

In the code above, we called upon `Name` as opposed to `name`. Recall that Python is a case-sensitive language. This means that `name` and `Name` are two separate things. We did declare "name" but at no point did we ever create a variable named "Name." As a result, an exception was thrown our way.

In the above, we ended up with `NameError`. This generally happens when we try to call upon a variable, function, or method that doesn't exist. As a result, the error was recorded, and the rest of the program was terminated. If this code had additional lines, they wouldn't be executed.

There are quite a few types of Exceptions that exist in Python. Let's go through each of them to understand what's going on. This will help us with our debugging process.

Types of Exceptions

NameError

The first one is NameError. We just came across this above. It happens when you try to call upon a variable that doesn't exist.

TypeError

TypeError occurs when trying to run an operation with an inapplicable data type. Let's say you are trying to print the following:

```
print(2 + "2")
```

In the above, we can see clearly that one data type is an integer value while the other one is a string. These cannot interact with each other unless they are changed into strings or integers.

In such a case, you will be presented with the TypeError, as shown below:

```
print(2 + "2")
```

```
Traceback (most recent call last):
File "<pyshell#7>", line 1, in <module>
print(2 + "2")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

ValueError

A ValueError exception occurs when you are trying to use an invalid value as an argument. Let's say you type in the following:

```
print(int("My name is Cole"))
```

The above shows that int() is called upon, which means the program expects you to input a numerical value. Instead, we ended up passing a string value. As a result, we will end up with the following error:

```
print(int("My name is Cole"))
```

```
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
print(int("My name is Cole"))
ValueError: invalid literal for int() with base 10: 'My name is Cole'
```

IndexError

When you use index numbers and use the wrong ones, you get IndexError, Python's way of saying, "Sorry, but that doesn't exist."

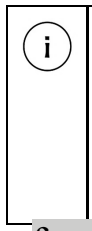
```
dog = [1, 2, 3]
print(dog[3])
```

```
Traceback (most recent call last):
File "<pyshell#11>", line 1, in <module>
print(dog[3])
IndexError: list index out of range
```

In the above, since index values start from 0, the last index number is 2. We asked Python to print whatever is at index number 3. Since that is out of the range of the list itself, we end up getting an IndexError.

IndentationError

When you use the wrong indentations, you end up facing this error. You may have noticed how conditional statements, for and while loops, and definition sections are always indented. This isn't just to make the code look nice, but it is also a requirement.



Sometimes, it may also be reflected as SyntaxError, but the description will clarify what you are looking at:

```
SyntaxError: expected an indented block after 'for' statement on line
1
```

```
for x in range(1, 10):
print(x)
```

```
SyntaxError: expected an indented block after 'for' statement on line 1
```

The print(x) was meant to be indented in this code, yet it wasn't. In some IDEs, this will be reflected as an IndentationError. Here's how

the very same code will generate an indentation error in a different IDE (PyCharm, for this example):

```
File "/Users/PycharmProjects/PythonForBeginners/test.py", line 2
print(x)
^
IndentationError: expected an indented block after 'for' statement on line 1
```

ZeroDivisionError

This one should make sense right away. Sometimes, you may have massive algorithms and complex code to work with. As you process all of that and run it, you may end up with this error. What does it mean? It simply means that there is some instance where something is being divided by 0. Since that is impossible to do, the system will throw an exception your way:

```
print(1/0)
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
print(1/0)
ZeroDivisionError: division by zero
```

Of course, the above is not complicated or complex, but it is a perfect way to show when such an error might pop up.

ImportError / ModuleNotFoundError

You will have this error when you import something from the wrong library. In the future, whenever you are importing specialized libraries for machine learning, game development, or other projects, you may come across this error if you're trying to import the wrong module from the wrong library:

```
from numpy import pandas

Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
from numpy import pandas
ModuleNotFoundError: No module named 'numpy'
```

AttributeError

When you are trying to use an inapplicable attribute for a given Python object, you end up getting this error:

```
print('a'.sum())
```

```
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
print('a'.sum())
AttributeError: 'str' object has no attribute 'sum'
```

KeyError

Just like IndexError, KeyError happens when you are calling upon a key that doesn't exist in a dictionary:

```
animals = {"koala": 1, "panda": 2}
print(animals["rabbit"])
```

```
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
print(animals["rabbit"])
KeyError: 'rabbit'
```

These are some of the most common ones you will come across. You may come across many other exceptions, a detail of which can be found under Python Documentation at www.python.org.

Python Debugging Techniques

Now that you know the errors and exceptions, let's learn how we can overcome these. It is essential to learn all of these to have the best chance at fixing the issues you may encounter during your programming journey.

Using the Print Statements

This is the most basic way to find problems and then try to resolve them. When you have multiple operations, try to print the variables or results out to see if everything is working. If everything works fine, you should get the result you are looking for.

In all the above examples, where we discussed a variety of exceptions, we used print statements to see if the code was working.

Where it wasn't, it was responding with the type of error it encountered and suggestions. It even highlighted the line number to narrow our search down.

This is the very first kind of debugging method you use.

Using Python Debugger

Python comes built-in with a powerful debugging tool. It is called the Python Debugger (pdb). Using pdb helps you greatly because it goes through the code, checks for every value and variable, and then tracks down any hard-to-find bugs. To use pdb, you must first import it into your program and then call it into action. How? Here's an example:

```
import pdb

def factorial(x):
    if x == 0:
        return 1
    else:
        return x * factorial(c-1)

pdb.set_trace()

print(factorial(4))
```

Once you execute this code, the Python debugger will start. It will present you with the following:

```
-> print(factorial(4))
(Pdb)
```

After the parentheses, you can type one of the four values

- n—to execute the command and go to the next one.
- s—to step inside a function call.
- p—to print the value of a variable.
- q—to quit debugger.

If you were to press "n" in the case above, you will find that it returns an error:

```
NameError: name 'c' is not defined
```

Similarly, if you press q, it will detail all the errors it found. This helps eliminate the guesswork, especially since most codes and programs you work with will have hundreds or thousands of lines.

Using the Logging Module

You can also use the logging method to debug your program. This involves importing the logging module into your program, just like we imported pdb earlier. The logging module provides output messages that help you track down and identify problem areas within the code. Here's logging in full action:

```
import logging

def total_price_calculator(unit_price, quantity):

    total = unit_price * quantity
    logging.basicConfig(level=logging.DEBUG)
    logging.debug(f"Total price is: {total}") # add a logging statement to
    debug

    return total

price = total_price_calculator(10, 5)
```

In this code, we first changed the basic configuration of the logging module to DEBUG. This is because, by default, this module does not offer any output. We need to change its level since we want to see what's happening with the code. Next, we carried out a simple program, and it worked. We received the following output:

```
DEBUG:root:Total price is: 50
```

The above does not show any errors and also shows the result. Logging is quite effective, especially when dealing with significantly more complex codes.

The Try and Except Statements

These are the most commonly used and effective ways to debug your program. Using the try and except statements, you can ensure that your code continues operating without running into unnecessary crashes.

The try statement is quite helpful when running a program you know has errors. The try statement is always followed by an except statement. What this does is tell the program, "Hey. Try this first. If it works, use that and move on. If this causes an exception, don't run this and run this instead."

You set up two additional blocks when you spot a troubling piece of code. The first is the try block, where you ask the program to try a new block of code to see if that fixes the problem. You also type in an except block of code in case the try block does not work. It's just like the conditional if and else statements we worked with earlier.

Using these will allow Python to continue running the program instead of terminating the process completely. How? Let's see this in action:

```
try:
    print(a)
except:
    print('Please declare the variable a first')

print(10)
```

```
Please declare the variable a first
10
```

We set up two blocks in the above code, try and except. We first asked Python to try to print 'a', and, as you may have guessed, 'a' is not defined in the code. If you had just asked Python to print 'a' normally, the program would have crashed immediately.

Since we used the try and except, it first tried the try block. Since that led to an error, it skipped that and went for the except block, where we wrote our error to help us know what was causing the problem. As a result, the code continued working and successfully printed 10

in the end.

To make things more interesting, let's try and catch an exception with try and except:

```
try:
    print(a)
except NameError:
    print("You need to declare the variable a first")
```

If this code was run without the try and except block, we'll run into NameError immediately. Instead, now, we end up with a customized error message and not a crash:

```
You need to declare the variable a first
```

As a programmer and debugger, one of your most important jobs is to identify the type of errors you will encounter with a given code. This is important because you can then use that information to create multiple except statements to catch any error that may occur. Here's an example:

```
def print_dict_total(dct):
    print(sum(dct.values()))
```

```
dict = {'a': 1, 'b': 2, 'c': 3}
print_dict_total(dict)
```

```
try:
    print_dict_total(mydict)
except NameError:
    print('Check spelling of dictionary name')
except AttributeError:
    print('Provide a dictionary with numeric values')
except TypeError:
    print('Some dictionary values are not numeric')
```

In the above, we have set up a simple code. Notice how we deliberately used the wrong name in the try block. As a result, it will revert back with NameError. If that happens, the corresponding except block will catch it and ensure it provides a better output that

shows us what we did wrong. We also set up other except statements. To check these, try to change the values or names to see how these come into action.

If all else fails, you can also use the else statement, and if that fails too, you can use the finally statement as your last resort:

```
try:
    print(3/0)
except ZeroDivisionError:
    print('You cannot divide by zero')
else:
    print('The division is successfully performed')
finally:
    print('This message is always printed')
```

Recommended Workflow of Debugging for Beginners

You might be wondering which of these should you use to debug your code. To help you, here is a recommended workflow to use as a step-by-step guide. You can move on to the next step only if the current step fails:

1. Identify the problem yourself.
2. Reproduce and replicate the problem.
3. Use print statements.
4. Use the pdb module.
5. Use a logging module.
6. Use try and except.
7. Test the code once again to finalize everything.

Exercise

Right! It's time to get cracking at some debugging. Below, you will find two sets of code. The goal is to find out the error, fix it, and run the code:

```
print("Welcome to Python Programming!"
```

That's quite simple, isn't it? Well, let's see if you can solve this one.

```
numerator = 10
denominator = 0
result = numerator / denominator
print(f"The result is {result}")
```

With the above code, you are bound to get an error of some kind. The goal is to handle this code exception so that the code does not crash and instead handles the code exception somehow.

Here's an additional code if you're feeling up for the task:

```
numbers = [1, 2, 3, 4, 5]
average = sum(numbers) / len(numbers) - 1
print(f"The average is {average}")
```

There's something wrong with this code. Can you spot the issue and fix it?

With that, it is time for us to move on to the book's next chapter, where we will discuss Python file operations and data storage.

Chapter 8: Python File Operations and Data Storage

> *"The goal is to turn data into information, and information into insight."* **–Carly Fiorina, Former CEO of Hewlett-Packard**

Did you know you can use Python to write files like .csv, text files, and more? Well, that is something Python can do and part of the reason why Python is being taught in almost every leading business and finance school.

This chapter, therefore, will be all about Python file operations and how you can read, write, and store files easily.

Reading and Writing Files

Python is powerful enough for many reasons, and reading and writing files is just one of them. Using Python, you can create, write, and read files easily.

Regarding Python, there are two types of files that Python can handle. These are standard text files and library files. The latter are written in binary language, so we won't be going into those. Keeping the scope of this book in view, we will only focus on text files.

File Access Modes on Python

When you access a file, you have different access modes. Since we are dealing with coding, it isn't as simple as double-clicking and opening a file. Here, we must specify in the code what kind of access we want.

In Python, we have:

1. Read-only—'r' is used to open text files. This is the default mode. Returns error if the file doesn't exist.
2. Read and write—'r+' is used to open and write on the file. Returns error if the file doesn't exist.

3. Write only—'w' is used to open a file for writing. Existing files will have their data truncated, meaning you will overwrite them. Creates the file if the file doesn't exist.
4. Write and read—'w+' is used to open files for reading and writing. Existing files will be overwritten.
5. Append only—'a' is used to open files for writing. A file is created if it doesn't exist. Data is added at the end of the file.
6. Append and read—'a+' is used to open files for both reading and writing. A file is created if it doesn't exist. Data is inserted at the end.

Opening Files

To open a file in Python, we use the following basic syntax:

```
file_object = open('file_name', 'mode')
```

The open() function always takes in two parameters. These are:

1. file_name—this includes the file extension. Python assumes that the file in question is in the same directory.
2. mode—this is an optional parameter, but to use file operations, it is recommended that you use the correct type of mode to access the files.

Depending on the kind of mode you wish to go for, use the corresponding mode letters. Before opening the file, however, you want to ensure that any existing file you wish to work with is in the same directory as the program file itself. If not, you will need to type in the full address of the file instead of the filename.

```
# Open function for "MyFile1.txt"
# file is in the same directory
my_file = open("MyFile.txt", "a")
```

```
# store its reference in the variable my_file
# and "MyFile2.txt" in \Text subdirectory in my_file2
my_file2 = open(r"Text\MyFile2.txt","w+")
```

Note that we used the letter 'r' before the file name. This is done to make the string raw. This ensures that no character of the filename is treated as a special character. This ensures that this does not return an error. You can drop the r if the file happens to be in the same directory and you're not writing down the address.

You'll create my_file as an object for *MyFile1.txt* using the code above. Similarly, you will create my_file2 as an object that will correspond to *MyFile2.txt*.

Closing Files

To close a file, you use the close() function.

Important: Whenever you work with files, always ensure that you close them before moving on to the next one. This is done to ensure the system frees up memory space.

```
my_file = open("MyFile.txt", "a")
my_file.close() #this will close the file
```

Writing

There are two ways to write on a file.

1. Use the write() function—This inserts a string in a single line within the text file.
2. Use the writelines() function—This inserts multiple string elements.

```
file_object.write(str1) #replace str1 with what you wish to enter as a string
file_object.writelines(x) for x = [str1, str2, str3, ..... strN]
```

Reading

To read from a file, you can use three ways.

1. Use the read() function—This returns read bytes in a string format. This reads the entire file if no value is passed through.
2. Use the readline() function—This reads a single line from the file and returns it as a string. This does not read more than

one line if the value passed through exceeds the length of the line itself.

3. Use the readlines() function—This reads all the lines and fetches them back as string elements within a list.

Below is a code that shows you various types of writing and reading functions in use.

```
# Showing various ways to read and write
my_file = open("my_file.txt", "w")
L = ["This is Toronto \n", "This is New York \n", "This is LA \n"]

# \n is placed to indicate EOL (End of Line)
my_file.write("Hi there \n")
my_file.writelines(L)
my_file.close() # closing to change access mode

my_file = open("my_file.txt", "r+")

print("Output of Read ")
print(my_file.read())
print()

my_file.seek(0)

print( "Output of Readline ")
print(my_file.readline())
print()

my_file.seek(0)

# difference between read and readline
print("Output of Read(9) function is ")
print(my_file.read(9))
print()

my_file.seek(0)

print("Output of Readline(9) function is ")
```

```
print(my_file.readline(9))
```

```
my_file.seek(0)
# readlines function
print("Output of Readlines function is ")
print(my_file.readlines())
print()
my_file.close()
```

The output is as follows:

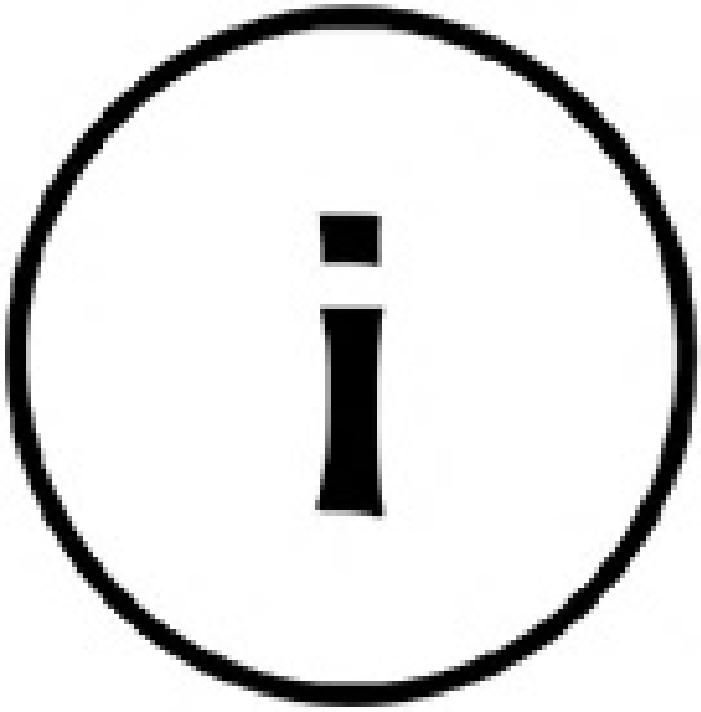
```
Output of Read
Hi there
This is Toronto
This is New York
This is LA

Output of Readline
Hi there

Output of Read(9) function is
Hi there

Output of Readline(9) function is
Hi there
Output of Readlines function is
['Hi there \n', 'This is Toronto \n', 'This is New York \n', 'This is LA \n']
```

	While you already know it is essential to close files you are working with, you



should always double-check that you are doing so. Failure to do so can lead to file corruption, slowing down of the program, system crashing, and other issues.

Appending

We know we must use 'a' or 'a+' to append items to the file. However, let's see how that happens with another example.

```
# Append VS Write
my_file2 = open("file2.txt", "w")
L = ["Say Hello to Japan \n", "Hello Italy \n", "This is England \n"]
my_file2.writelines(L)
my_file2.close()
```

```
# Append-adds at last
my_file2 = open("file2.txt", "a") # append mode
my_file2.write("Today \n")
```

```
my_file2.close()
```

```
my_file2 = open("file2.txt", "r")
print("Readlines after appending")
print(my_file2.readlines())
print()
my_file2.close()
```

```
# Write-Overwrites
my_file2 = open("file2.txt", "w") # write mode
my_file2.write("Tomorrow \n")
my_file2.close()
```

```
my_file2 = open("file2.txt", "r")
print("Readlines after writing")
print(my_file2.readlines())
print()
my_file2.close()
```

The output is:

```
Readlines after appending
['Say Hello to Japan \n', 'Hello Italy \n', 'This is England \n', 'Today \n']

Readlines after writing
['Tomorrow \n']
```

Working With CSV Data in Python

Text files are one thing, and they are pretty standard. However, things are quite different when it comes to CSV. CSV stands for Comma Separated Values file. It is a popular file format used in Excel and spreadsheet software.

Python has a built-in CSV library, allowing us to read, write, and process data to and from any given .csv file easily. Let's see how that works.

Opening a CSV File

Once again, we call upon the open() function to open a CSV file.

```
x = open("file.csv") #Assuming that the file is in the same directory
```

You will need to type in the full address if the file exists in some other directory, as shown below:

```
x = open(r"E:\Python3\Scripts\file.csv")
```

File Modes

Some modes remain the same as before when dealing with CSV files, while some are new.

1. 'r'—this is used for read-only access.
2. 'w'—this is used for write-only access (will overwrite data).
3. 'a'—this is used for writing only (appending data).
4. 'r+'—this is used for reading and writing files.
5. 'x'—this is used to create a new file.

Closing a CSV File

Use the close() function to close an open CSV file.

```
x.close() # assuming that x was already reading or writing on file.
```

It is worth noting that this may not always work. To ensure that the file is closed correctly, it is highly recommended to use the with keyword. Doing so automatically closes the file once the code moves past the with block. Alternatively, you can use the try-finally method:

WITH METHOD

```
with open('file1.csv') as x:
    print(x.read()) # The file will automatically close
```

TRY-FINALLY METHOD

```
x = open('file1.csv')
try:
    # File operations here
finally:
    x.close()
```

Reading a CSV File

To read, you will first need to import the csv module. This will allow you to gain access to its reader() method. The reader() method is excellent as it splits rows into specified delimited. By doing that, it then provides us with an output as a list of strings.

Let's assume that we have a csv file named file1.csv. The file has the following data:

```
name, job, city, age
Ken, Manager, Claymont, 37
George, Actor, Seattle, 44
```

To read from this file, we will need to do the following:

```
import csv

with open('file1.csv') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

The file will close automatically since we used the with keyword. If we run this code, we will end up with three lists, each showing the contents of a line, starting from top to bottom.

```
['name', 'job', 'city', 'age']
['Ken', 'Manager', 'Claymont', '37']
['George', 'Actor', 'Seattle', '44']
```

Writing on CSV Files

If you aim to write on an existing file, the first thing to do is import csv (if you haven't already done so) and then open the file. You can open the file with:

- w—for write only
- a—for append
- r+—read and write

Once done, you must use the writer object and the writerow() method. These will help pass the data through the file as a list of strings.

```
import csv

with open('file1.csv', 'a') as x:
    writer = csv.writer(x)
    writer.writerow(['John', 'Senior Manager', 'San Jose' , '45'])
    writer.writerow(['Samuel', 'Designer', 'Miami' , '40'])
```

This will write the contents in the file *file1.csv*.

Read CSV File Into Dictionaries

Python allows you to read any CSV file directly into a dictionary. This can be done using the DictReader() method. When you do so, it is important to note that the very first row of the file that has the column names will serve as keys for this new dictionary:

```
import csv

with open('file1.csv') as x:
    my_reader = csv.DictReader(x)
    for row in my_reader:
        print(row)
```

If your file does not have names for each column, you can use an optional parameter fieldnames to name keys.

```
import csv

with open('file1.csv') as x:
    my_keys = ['name', 'job', 'city', 'age']
    my_reader = csv.DictReader(x, fieldnames=my_keys)
    for row in my_reader:
        print(row)
```

Writing CSV File From Dictionary

Besides reading, you can also write from a dictionary as well. We will need to use the DictWriter() method to do this. However, unlike

reading, you must use the fieldname parameter to specify the keys.

```
import csv

with open('file1.csv', mode='w') as x:
    my_keys = ['name', 'job', 'city', 'age']
    my_writer = csv.DictWriter(x, fieldnames=my_keys)
    my_writer.writeheader() # adds column names in the CSV file
    my_writer.writerow({'job': 'Senior Manager', 'city': 'San Jose', 'age':
    '45', 'name': 'John'})
    my_writer.writerow({'job': 'Designer', 'city': 'Miami', 'age': '40', 'name':
    'Samuel'})
```

This will add the information in your CSV file.

Handling Errors

When you run across an error, it is best to use the try-except block process to try and resolve the issue:

```
import csv
import sys
file = 'file.csv'
with open(file, newline='') as x:
    my_reader = csv.reader(x)
    try:
        for row in my_reader:
            print(row)
    except csv.Error as error:
        sys.exit('file {}, line {}: {}'.format(file, my_reader.line_num, error))
```

Working With JSON Data in Python

JSON stands for JavaScript Object Notation. This means that it is an executable script file that is made up of text. It is used to store and transfer data. Fortunately, Python supports JSON and does so through its built-in package with the same name: JSON.

If you are to use this package, you will first need to use the import JSON command to import and be able to access all features within

the package.

The text within JSON uses quoted strings containing key-value mapping within the curly brackets {}. It is almost like dictionaries in Python:

```
import json

# {key: value mapping}
x ={"name": "Alan", "age": 29, "Salary": 55000}

# dumps() function called to convert to JSON
y = json.dumps(x)

# printing output
print(y)
```

This produces the following output:

```
{"name": "Alan", "age": 29, "Salary": 55000}
```

JSON supports the primitive Python data types, including strings, numbers, lists, objects, and tuples.

JSON Serialization

Serialization is a process of encoding JSON. The term is used to highlight the data transformation. The given data is converted into a series of bytes that are then stored or transmitted across a network.

Programmers use the dump() function to handle the data flow within a file. The dump() function converts Python objects into JSON objects. This conversion allows programmers to write data to files quickly.

Below is a table that shows Python objects and what they become once they are converted into JSON objects.

Python	JSON
dict	object
list and tuple	array
str	string

int, float, and long	numbers
True	true
False	false
None	null

Here's how you can carry out simple serialization:

```
import json
my_var = {"Subjects": {"Spanish": 75, "Science": 91}}
```

```
with open("Test.json", "w") as a:
    json.dump(my_var, a)
```

We first import JSON into Python. Then, we create a Python object my_var. Next, we create a file named *Test.json*. We use the write mode to access it. Next, we use the dump method. The json.dump() method takes in two arguments. The first is the data object that we want to serialize. The second is the object, which the program will write in Byte format.

Deserialization

This is the opposite of serialization. Therefore, instead of dumping data, it loads the data instead. As a result, the JSON objects are converted into their corresponding Python objects. For this, we use the load() method:

```
with open("Test.json", "r") as read_this:
    my_data = json.load(read_this)
```

Let's look at another example where we create a JSON object and then convert it into a Python object using the load() method:

```
import json
json_obj ="""
{"Country":
    {"name": "Pakistan", "Languages_spoken": [
    {"names":
        ["Urdu", "English", "Sindhi", "Punjabi"]}]}}
```

```
"""
python_obj = json.loads(json_obj)
```

Exercise

So far, you're doing great. Yes, it does take some time to get used to it. However, do not be alarmed if you run into any issues. You can always consult this book and find help from the wonderful community. Alternatively, you can check out Python Documentations to further enhance your knowledge and skills.

With that said, it is time to put your knowledge to the test. For this chapter, create a program that reads and writes from a text file. It's that simple!

Bonus Challenge

If you're up for the challenge, we have one for you.

Create a personal expense tracker. You will need to bring in all the knowledge from this and previous chapters. Below are step-by-step instructions and a completed code to inspire you.

The code must support JSON and CSV formats. The program should include reading and writing expense data to a text file:

1. Initialize an empty list.
2. Load the existing expense list form file.
3. Create functions for operations:
4. Function to add expense.
5. Function to view expense.
6. Function to save expense.
7. Create a menu.
8. Create a main program loop.
9. Complete the code.

The code is as follows:

```
# Initialize expense list
expenses = []
```

```
# Load existing expenses from the file
try:
    with open("expenses.txt", "r") as file:
        expenses = file.readlines()
except FileNotFoundError:
    print("No existing expenses found.")

# Function to add an expense
def add_expense(expense):
    expenses.append(expense)

# Function to view expenses
def view_expenses():
    for expense in expenses:
        print(expense.strip())

# Function to save expenses to file
def save_expenses():
    with open("expenses.txt", "w") as file:
        file.writelines(expenses)

# Function to display the menu
def display_menu():
    print("Expense Tracker Menu:")
    print("1. Add Expense")
    print("2. View Expenses")
    print("3. Save and Exit")

# Main program loop
while True:
    display_menu()
    choice = input("Choose an option (1-3): ")

    if choice == '1':
        expense = input("Enter the expense details: ") + "\n"
        add_expense(expense)
    elif choice == '2':
```

```
        view_expenses()
    elif choice == '3':
        save_expenses()
        print("Expenses saved. Goodbye!")
        break
    else:
        print("Invalid choice. Please try again.")
```

In the next chapter, we will move on to Python for web development, an exciting area that will surely catch your fancy.

Chapter 9: Fundamentals of Python Web Development

> *"If you think math is hard, try web design."* – **Trish Parr**

Web development is exciting. It is a field where programmers and web developers can be as creative as they want. However, with that, there are challenges, too.

Python, a programming language, doesn't necessarily come to mind for web development immediately. With that said, it does have an ace up its sleeve that many developers and professionals know about and love. Therefore, this chapter will help you learn exactly what that is, why Python is great for web development, and how to get started immediately.

Introduction to Python Web Development

Earlier in the book, we talked about how Python was being used by some of the top web-based entities, such as Instagram, Spotify, and Disqus. That is because Python web development offers flexibility. It is a powerful way to develop web-based applications and projects. It can handle simple projects as well as those that are more complex. Because of that, Python continues to be used for projects related to:

- Travel
- Transportation
- Healthcare
- Finance
- Manufacturing

And that is just scratching the surface. There's a lot more that Python web development can do for you. While there are many other well-established platforms and frameworks people use for

web development, Python has certain advantages that most cannot offer:

1. Easy and readable
2. Portability and interactivity
3. Prototyping
4. Visualization options
5. Asynchronous coding
6. OOP
7. Rich standard library
8. Rich ecosystem
9. Enterprise application integration

The list goes on and on. However, there are some cons to this as well that you should know:

1. Fewer specialist developers.
2. Not popular for mobile applications.
3. Speed limitations.
4. High memory consumption.
5. Design restrictions.

Now, for those who are eager to get started, we have some good news. Python has a framework (a platform that provides you with all the tools and functionalities you need to develop applications) that makes your job a lot easier. In this case, that happens to be a framework called Flask.

The Basics of Framework—Flask

A framework is essentially a platform that developers use to develop software applications. These can be offline or web-based. A framework makes the work easier as most fundamental elements are pre-loaded and provided. This helps save time and energy. Not just that, using frameworks helps keep things more organized, too.

Frameworks offer functionality and tools that are used explicitly for application development. This means you won't have to go around the internet to search for what to use to carry out a specific operation. It's all there for you, and Flask is one such example.

Flask is Python's web development framework. It's not the only one, but it is one that people mostly use. It is the most beginner-friendly framework to learn and work with. The other alternative is Django, which requires much time to master.

The good thing is that you don't have to start from scratch. Why? Because by now, you know your way with Python. That helps you save a lot of time and get straight into it.

Flask is based on the Werkzeg Web Server Gateway Interface (WSGI) toolkit and Jinja2 engine. The Werkzeg toolkit is used as a standard for Python web development. It is WSGI that interacts between the server and the web app. Jinja2, on the other hand, is a very popular template engine for Python. Using Jinja allows you to pass Python variables through any HTML code or template easily.

Flask is a lot like Python, meaning you will not have any trouble using it. To give you an example, here's a sample boilerplate code for you to see:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

That makes sense, and it is very readable. Well, before you go on to code anything in Flask, you first need to install something called a virtual environment.

The virtual environment is used to manage your project during development and production. A virtual environment is a group of

independent Python libraries for each project. You can have a set of packages installed for one project, and they will not affect any other project. To use Flask, we need to download a few things.

Here's a step-by-step guide on how to install Flask properly on your system.

Step 1—Installing the Virtual Environment

First, you will need to install the virtual environment. Be sure to follow the steps according to your operating system.

For Linux

If you are using Debian or Ubuntu, do the following:

1. Open Terminal.
2. Use the following apt command to install virtualenv:
    - sudo apt install python-virtualenv

For CentOS, Red Hat, or Fedora users, here are the steps:

1. Open Terminal.
2. Use the following yum command to install virtualenv:
    - sudo yum install python-virtualenv

For MacOS

For Mac users, the installation is equally simple.

1. Launch Terminal
2. Use the following command to install virtualenv using pip.
    - sudo python3 -m pip install virtualenv

For Windows

To install a virtual environment on Windows, do the following:

1. Open the Command Line (aka command prompt) with admin privileges.

2. Use the following pip command to install virtualenv:
    - py -3 -m pip install virtualenv

Step 2—Creating an Environment

Next, we need to create an environment. To do that, you need to do the following:

1. Make a separate directory/folder for every project.
2. Move into the directory. If you're using Terminal or command prompts, use the cd command to enter the destination directory.
3. Once in the directory, create a virtual environment for Flask using the steps below, depending on your operating system.

Linux and MacOS

To create an environment in Linux or MacOS, use the following code in Terminal (be sure you are in the directory of the project):

- python3 -m venv <name of environment>

You can name the environment as per your liking.

Windows

For Windows, use the command prompt to enter the destination folder or directory. Once there, use the following code to create a virtual environment:

- py -3 -m venv <name of environment>

Step 3—Activating the Environment

Once the virtual environment is created, you need to activate it.

Linux and MacOS

Use the following line of code:

- <name of environment>/bin/activate

Windows

Use the following line of code:

- <name of environment>\Scripts\activate

Step 4—Installing Flask

Next, we need to install Flask inside the activated environment. For this, we will need to use the same command in all three operating systems:

- pip install Flask

Step 5—Testing the Development Environment

1. Create a Flask app to test your development environment (it can be as simple as a Hello World program).
2. Create a file named hello.py in the Flask project folder.
3. Use a text editor and code in the "hello world" program.
4. Save and close the file.
5. Use the console to navigate to your project folder (use the cd command).
6. set FLASK_APP environment variable.
    A. Linux and Mac users use the code after the arrow -> export FLASK_APP=hello.py
    B. Windows -> setx FLASK_APP "hello.py"
7. Run the Flask application using the code flask run.
8. Take note of the address (http://x.x.x.x:x) and copy it.
9. Open a browser, paste the address, and press enter to access the web application.

Building a Simple Python Web App

With Flask installed and running, it's time to create a simple web app. We start by learning about routing. From there, we will move

on to templates and some exercises to reinforce what we'll learn.

Routing

Also known as Flask App Routing, it is mapping URLs to specific functions. Modern-day web frameworks, such as Flask, make it easier for users to remember these URLs. Not just that, they aim to make navigation easier as well.

Whenever you go to any URL, you usually find that the URL reads something like www.nameoftheurl.com. If you click on "About Us," it changes to www.nameoftheurl.com/about. Here, the "/" sign is used to signify the root URL. The URL is then mapped so that users know where they are, hence the /about. If we were to click on another section, say blog, it would read /blog. That's routing at work.

Using Flask, we can bind our URL with a function. For that, we use something called app.route decorator. Here's an example of how it works:

```
from flask import Flask

my_app = Flask(__name__)

# Pass the required route to the decorator
@my_app.route("/hello")
def hello():
    return "Hello, Welcome to My Website"

@my_app.route("/")
def index():
    return "Homepage of this website"

if __name__ == "__main__":
    my_app.run(debug=True)
```

Save this file as *main.py* in your development environment directory. We used /hello and / to separate two pages in the above example. One is a welcome page, while the other (/) is a root or home page. You can now run this app by using the following code

on the command prompt or terminal:

```
python main.py
```

Next, open a browser and type 127.0.0.1:5000/hello. You will see the welcome page.

Think about routing as routes that help you navigate more easily.

The World Wide Web (www) is where much information exists. A particular protocol is used to communicate data back and forth—http. You may have already seen this in the address bars of your browsers. Flask, on the other hand, has different decorators when it comes to handling http requests.

There are different methods to retrieve data from a given URL. These are:

Request name	Purpose of Request
GET	Most common method used. GET is a message sent to a server, and the server then returns data
POST	POST sends HTML form data to servers. The server does not cache any data received through POST
HEAD	It is the same as GET but has no response body
PUT	PUT replaces every current representation of the resource with uploaded content
DELETE	Used to delete all representations given by the URL

Flask HTTP Methods

First, we have form. By default, Flask responds to GET requests only. Using the route() decorator, you can change that.

Below, we have set up a simple code to show POST use in a URL route. To do this, we will first create a simple HTML form. We will use the POST method to send out form data from there.

```
<html>
   <body>
      <form action = "http://localhost:5000/login" method = "post">
         <p>Enter Name:</p>
         <p><input type = "text" name = "nm" /></p>
         <p><input type = "submit" value = "submit" /></p>
      </form>
   </body>
</html>
```

Save this as *login.html*. You can use a text editor for this, such as Notepad. In the above, you will notice that we specified the method as post. Using that, we can send form data to the URL easily.

Next, we have GET and POST requests working in tandem. To handle both of these at the same time, we use the app.route() method, as shown below:

```
from flask import Flask, redirect, url_for, request
app = Flask(__name__)

@app.route('/success/<name>')
def success(name):
   return 'welcome %s' % name

@app.route('/login',methods = ['POST', 'GET'])
def login():
   if request.method == 'POST':
      user = request.form['nm']
      return redirect(url_for('success',name = user))
   else:
      user = request.args.get('nm')
      return redirect(url_for('success',name = user))

if __name__ == '__main__':
   app.run(debug = True)
```

This script goes in IDLE and saved as *main.py* in the same directory as the *login.html*. After this, the development server should work.

Once that's started, open the login.html file you saved using your browser. This will open a simple prompt with a text field asking you to "Enter Name." The request.form["nm"] will fetch the form data.

Once you enter the name, you will be taken to another page that says, "Welcome (your name)."

Using Templates

Jinja2 provides templates for Flask. These are great to use as:

1. They are fast and effective.
2. You do not need to install them separately.

In case you do not have Jinja2, all you need to use Jinja2 is to open up your command prompt or terminal and use the following command:

- pip install Jinja2

By default, the Flask framework comes with a pre-installed Jinja2. Using Jinja2, we can work with HTML, CSS, and JS without switching over IDEs or coding on different platforms. You can do all of that in one place. Thanks to Flask, our jobs just became a lot easier.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
@app.route('/hello')
@app.route('/hello/<user>')
def hello_world(user=None):
    user = user or 'Ken'
    return '''
<html>
<head>
<title>Templating in Flask</title>
```

```
</head>
<body>
<h1>Hello %s!</h1>
<p>Welcome to the world of Flask!</p>
</body>
</html>''' % user

if __name__ == '__main__':
    app.run()
```

In the above code, notice how we used both HTML and Python within Flask. This is made possible by a templating engine. Jinja2 is one such templating engine.

When using Flask, note that Flask expects all templates to reside in the templates folder at the root level. If a template is present, Flask can read a template and use it through the render_template() method.

First, you need a main Python file where you will code all your programs. Here's a sample code below:

```
from flask import Flask, render_template, redirect, url_for

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("index.html")

@app.route("/default")
def default():
    return render_template("layout.html")

@app.route("/variable")
def var():
    user = "Programmer"
    return render_template("variable_example.html", name=user)

@app.route("/if")
```

```
def ifelse():
    user = "Practice Programmer"
    return render_template("if_example.html", name=user)
```

```
@app.route("/for")
def for_loop():
    list_of_courses = ['JavaScript', 'Python', 'C#', 'Ruby']
    return render_template("for_example.html", courses=list_of_courses)
```

```
@app.route("/choice/<pick>")
def choice(pick):
    if pick == 'variable':
        return redirect(url_for('var'))
    if pick == 'if':
        return redirect(url_for('ifelse'))
    if pick == 'for':
        return redirect(url_for('for_loop'))
```

```
if __name__ == "__main__":
    app.run(debug=False)
```

You can save this as *app.py* or give it a name you like. Now, we need to declare a variable by using the Jinja template. We will use {{var_name}} inside the text HTML file. Here's a sample HTML file.

```
<html>
<head>
    <title>Variable Example</title>
</head>
<body>
    <h3>Hello {{var_name}}</h3>
</body>
</html>
```

You can also use the if and else conditions. The syntax will vary slightly, but you'll understand almost everything immediately. The basic syntax for conditional statements using Jinja2 template is:

```
{% if condition %}
....
....
....
...
{% endif %}
```

Here's a conditional statement in action (it's used inside the HTML file).

```
<!DOCTYPE html>
<html>
<head>
   <title>If example</title>
</head>
<body>
   {% if(name == "Programmer") %}
      <h3> Welcome </h3>
   {% else %}
      <h3> Unknown name entered: {{name}} </h3>
   {% endif %}
</body>
</html>
```

You can also use loops in Jinja2. It is done in almost the same fashion as conditional statements and inside the HTML file.

```
<!DOCTYPE html>
<html>
<head>
   <title>For Example</title>
</head>
<body>
   <h2> Programmer Ready to Code </h2>
   {% for course in courses%}
   <h4> {{course}} </h4>
   {% endfor %}
```

```
</body>
</html>
```

There is a lot more to Flask and Jinja2. However, the deeper we go, the more advanced it will be, which lies outside this book's scope. Therefore, if you wish to get into Python web development, there are tons of videos, courses, and great resources online that you can use. It takes time to get used to the slight changes here and there, but once you're ready, you'll soon be creating impressive projects and attracting much attention from possible employers.

Exercise

For this chapter, we have a simple but exciting exercise. Fill out the blanks to complete the code snippet using all that you have learned. The code is designed to create a basic web route using Flask. When accessed, the route should display the words "Hello, Flask!".

```
# Import the Flask library

from flask import ____

# Initialize the Flask application
app = ____(__name__)

# Define a route for the root URL
@app.____('/')
def ____():
    return '____, Flask!'

# Run the Flask application
if __name__ == '____':
    app.____()
```

You'd know that you've done this correctly by running this application and then going to your browser and using the address http://127.0.0.1:5000/ and seeing "Hello, Flask!"

Chapter 10: Introduction to Python Game Development

> *"Talk is cheap. Show me the code."* **–Linus Torvalds, creator of Linux operating system**

Yes! Python is very much capable to be used in game development as well. While we will not be diving too deep into the world of game development, the goal of this chapter is to introduce you to yet another possibility that Python presents you. By the end of this chapter, you should be able to code in a simple Python-based game.

Project Overview

For this final chapter, we will be creating a simple Snake game project using all that we have learned throughout the book.

The project will help you refresh all your concepts, apply everything you've learned in a creative way, and improve your coding skills before you take on higher level projects.

In this project, we will be using:

1. Variables and data types.
2. Functions and modules.
3. Loops and conditionals.
4. Object-Oriented Programming.
5. PyGame Library (needs to be imported).

Preparing for the Project

The first step is to learn the fundamentals of Python game development by learning the core, basic concepts of Python, and so on. And, we're already through that!

"How?"

That's what we've been learning so far in this book. Let's waste no time and move on to the next step and discover the pros and cons of

using Python for game development.

Pros

- It's easy for beginners.
- Great for prototyping games.
- Access to a very capable library (PyGame).
- Robust and easy to debug.

Cons

- Slow when compared to other and better established game development languages, such as C#.
- Low compatibility with game engines.
- Limited game libraries.
- Not a great choice for mobile gaming.

Now that we know what Python can do and what it can't, let's move on to the next step: Choosing the right game development framework, library, or engine.

The Right Option

As it turns out, there are a few choices you have that you can choose from.

- Python Arcade
- Panda3D
- HARFANG
- Ren'Py
- Kivy
- PyGame—We'll be using this one for our project
- Ogre
- Pyglet

Once you have made your mind up, or gone for the one we recommend, there are two things to do:

1. Start browsing YouTube and other sources to learn how to work on more complex projects.
2. Work on our project.

You can do both simultaneously. Additionally, you can seek further help from YouTube and other sources to make the code and the overall game even better.

It's perfectly okay to make mistakes. Do not be intimidated by them. In fact, failure is the greatest teacher. If you fail, find out why. Learn from the error and move on better equipped and prepared. Do this time and again and you'll soon be creating games like it's just another walk in the park.

PyGame Tutorial

The PyGame library is used to help programmers create video games. This library is well known within the game development community because it comes with many modules to carry out a variety of functions. They can play sound, draw graphics, take into consideration any mouse input, and so much more. You can also use this library to create a client-side application as well.

First, you will need to download pygame. For windows users, use the following command in command prompt:

```
pip install pygame
```

For mac users, do the following:

1. xcode-select --install
2. pip3 install pygame

Once you've done both, you're ready.

Import and Initialize

To get started, open up IDLE with a fresh screen. Start by importing pygame:

```
import pygame
pygame.init() #this will initialize pygame
```

Create a Game Window

Next, you will need to initialize a window to display. For this, we will use .set_mode() function. This will help create a new game window. All we need is to pass the width and the height of the window as parameters.

```
pygame.display.set_mode((width_of_window, height_of_window)) #e.g, 400, 500
```

Here's a complete code to get you started in the right direction:

```
# import pygame package
import pygame

# initializing imported module
pygame.init()

# displaying a window of height
# 500 and width 400
pygame.display.set_mode((400, 500))

# creating a bool value which checks
# if game is running
running = True

# keep game running till running is true
while running:

    # Check for event if user has pushed
    # any event in queue
    for event in pygame.event.get():

        # if event is of type quit then
        # set running bool to false
        if event.type == pygame.QUIT:
            running = False
```

The window will remain open for as long as you want. To exit, just press the exit button. If you aren't too sure about the screen size, you can use the screen.get_size() method (shown below):

```
# import package pygame
import pygame

# initialize pygame
pygame.init()

# Form screen
screen = pygame.display.set_mode()

# get the default size
x, y = screen.get_size()

# quit pygame
pygame.display.quit()

# view size (width x height)
print(x, y)
```

This will return you the values which you can use as maximum for your game window. You can also change the background color of the window, as shown below:

```
# Importing the library
import pygame

# Initializing Pygame
pygame.init()

# Initializing surface
surface = pygame.display.set_mode((400, 300))

# Initializing RGB Color
color = (255, 0, 0)

# Changing surface color
surface.fill(color)
pygame.display.flip()
```

To have the color you want, we recommend searching on Google the RGB values of the color you seek.

Updating Games Using Loops

Now that the window is created, it is time to program a loop that will constantly update the game's behavior depending on the input, score, and so on.

We will use loops to do the following main tasks:

1. Update game window to show changes.
2. Update game states depending on the user input.
3. Handle different events.
4. Keep game running.

Here's the code sample showing you how to do that.

```
# import pygame package
import pygame

# initializing imported module
pygame.init()

pygame.display.set_mode((400, 500))

# Setting name for window
pygame.display.set_caption('MyGame')

# creating a bool value which checks
# if game is running
running = True

# Game loop
# keep game running till running is true
while running:

    # Check for event if user has pushed
    # any event in queue
    for event in pygame.event.get():
```

```
    # if event is of type quit then set
    # running bool to false
    if event.type == pygame.QUIT:
        running = False
```

Drawing Graphics

Using Pygame, we will now draw objects and shapes. For this, we will use the draw.rect() function. The syntax to do this is simple:

```
pygame.draw.rect(surface, color, rect, width)
```

Here's a sample code to help you draw something.

```
# Import pygame module
import pygame

# initiate pygame
pygame.init()

# create the display 600x600
window = pygame.display.set_mode((600, 600))

# Fill the screen with white
window.fill((255, 255, 255))

# Using draw.rect module to draw the rectangle outline
pygame.draw.rect(window, (0, 0, 255),
                 [100, 100, 400, 100], 2)

# Draws the object to the screen.
pygame.display.update()
```

"Okay. That's just a rectangle."

Wait, there's more. This time, we will use a draw.circle() function to draw a circular shape instead.

```
# Import pygame module
import pygame

# initiate pygame
pygame.init()
```

```
# create the display 600x600
window = pygame.display.set_mode((600, 600))

# Fill the screen with white
window.fill((255, 255, 255))

# Using draw.circle module to draw the circle
pygame.draw.circle(window, (0, 255, 0),
                   [300, 300], 170, 3)

# Draws the object to the screen.
pygame.display.update()
```

Let's now go to some slightly more complex shapes, polygons. For this, we will use the polygon() function.

```
# Import pygame module
import pygame

# initiate pygame
pygame.init()

# create the display 600x600
window = pygame.display.set_mode((600, 600))

# Fill the screen with white
window.fill((255, 255, 255))

# Using draw.polygon module to draw the solid triangle
pygame.draw.polygon(window, (255, 0, 0),
                    [[300, 300], [100, 400],
                    [100, 300]])

# Draws the object to the screen.
pygame.display.update()
```

CHALLENGE: POLYGON

1. Change these coordinates above: [[300, 300], [100, 400], [100, 300]]

2. Add a fourth (or more) set of coordinates inside of the outermost [ ]
    - Note: If your coordinates are larger than 600, you will have to change the screen size to fit your coordinates in window = pygame.display.set_mode(600,600)

Creating a Working Game: Circle Escape Challenge

In this section, we'll put everything we've worked on together, and introduce a few new concepts such as collision detection, score keeping, and increasing game difficulty, among others. We've added comments throughout. You should now be able to gain an understanding of those concepts as we move along.

Importing Modules and Initializing Pygame

Using the following code, we import two Python modules: pygame and random. These modules provide game functionality and random number generation for our falling circles.

Then we initialize pygame.init() to be able to use Pygame features. The code is very simple, and nothing you haven't seen before.

```
import pygame
import random

pygame.init()
```

Setting Up the Display Window and Display Variables

Here we define the width and height of our screen and use pygame.display.set_mode() to those parameters. We also place a caption for the game screen itself using the pygame.display.set_caption() function

```
# Set up the display window, colors, and variables
screen_width = 400
screen_height = 600
screen = pygame.display.set_mode((screen_width, screen_height))
```

```
pygame.display.set_caption("Circle Escape Challenge")
```

Define Colors, Shapes and Player variables

In the section below we:

1. Define colors as RGB tuples using the variables white, player_color and circle_color
2. Set player_width, player_height and starting location in the center of the screen, by defining player_x and player_y
3. Define the dropping circle size with circle_radius, gave it a random starting location for circle_x and set circle_y=0 to start at the top of the screen. Then we set the circle_speed = 0.1. This is the speed at which it drops from the top of the screen. The speed will slowly increase for every 10 points, as we'll see later in the Game Loop
4. Last, we initialize the score variable score = 0

```
# Define colors and other variables
white = (255, 255, 255) # White color for the background
player_color = (0, 0, 255) # Blue color for the player
circle_color = (255, 0, 0) # Red color for the circle
```

```
# Define player rectangle and starting location for x and y
player_width = 50
player_height = 10
player_x = (screen_width - player_width) // 2
player_y = screen_height - player_height
```

```
# Define the Dropping Circle size, give it a random x location and set the
y=0 to start at the top of the screen
circle_radius = 20
circle_x = random.randint(0, screen_width - circle_radius * 2)
circle_y = 0
circle_speed = 0.1 # Set the initial circle speed to a very slow value (e.g.,
0.1)
```

```
score = 0 # Initialize the player's score
```

Game State Control

This is a very simple variable to set, but important to set the game_started variable to False so that the start up instructions are shown in the pygame window.

```
# Add a "game_started" flag to control the game state
game_started = False
```

Create the fonts for the Score and Instructions

```
# Create a font for displaying instructions
font = pygame.font.Font(None, 36)
```

```
# Create a font for displaying the score
score_font = pygame.font.Font(None, 36)
```

Display Functions

This function will display the Score at the top of the screen when the gam is running

```
# Function to display the score on the screen
def display_score(screen, score):
    score_text = score_font.render("Score: " + str(score), True, (0, 0, 0))
    screen.blit(score_text, (10, 10)) # Adjust the position as needed
```

"Hey, what is this blit nonsense?"

Great question. In its simplest form, it is a way to re-draw pixels on the screen. But it is just a little more complicated than that. You don't actually see the re-drawn screen until pygame.display.update() is stated in the code. You will see the above function and the call to update the display at the bottom of the Game Loop.

The following function will display the instructions in the middle of the screen. This will also hold the game state until the SPACE bar is pressed. This function also introduces pygame.KEYDOWN, pygame.K_SPACE and pygame.QUIT. This functionality watches for Key Press events and performs the instructions based on the keys that are pressed. You'll see this again in the game loop to move your

player left and right using the left and right arrow keys.

```
# Function to display instructions and wait for player input to start the game
def display_instructions(screen):
    screen.fill(white)
    instructions_text = font.render("Press SPACE to start", True, (0, 0, 0))
    screen.blit(instructions_text, (screen_width // 2 - 120, screen_height //
    2))
    pygame.display.update()

    waiting_for_start = True
    while waiting_for_start:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()
            if event.type == pygame.KEYDOWN and event.key ==
            pygame.K_SPACE:
                waiting_for_start = False
```

Game Loop

From here, we're going to the loop that will let you play the game. In the game loop we follow the workflow noted below. Events on the same indent level should match the indents in the code that will follow.

1. Set the game variable running to True
2. Create a while loop for running
    A. Check for a Quit condition
    B. Check for key presses (SPACE, LEFT Arrow, RIGHT Arrow)
        i. Space bar starts the game
        ii. Use the Left and Right arrow keys to move the player rectangle
        iii. Fill the screen

iv. Draw the Player rectangle
v. Draw the dropping circle
vi. Move the circle downward by the circle speed variable
vii. When the circle reaches the bottom
    a. Create a random spot for the circle to appear next, set circle_y=0
    b. Increase the score for the successful pass
    c. If the score is divisible by 10 with no remainder, increase the speed
viii. When a collision is detected
    a. End the game
    b. Reset the speed
    c. Set the score = 0
ix. Call the display screen
x. Update the display

C. If the game is not started
    i. Display the instructions

3. End loop (No code, Python knows by the indentation)
4. Quit pygame

Game Loop Code

```
# Game loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    keys = pygame.key.get_pressed()

    if not game_started:
```

```
    # Check if the game hasn't started and the player pressed the SPACE
    key to start
    if keys[pygame.K_SPACE]:
        game_started = True
else:
    if keys[pygame.K_LEFT] and player_x > 0:
        # Move the player to the left when the LEFT arrow key is pressed
        player_x -= 0.1
    if keys[pygame.K_RIGHT] and player_x < screen_width -
    player_width:
        # Move the player to the right when the RIGHT arrow key is pressed
        player_x += 0.1

    screen.fill(white)

    # Draw the Player rectangle and the Dropping Circle

    pygame.draw.rect(screen, player_color, (player_x, player_y,
    player_width, player_height))
    pygame.draw.circle(screen, circle_color, (circle_x, circle_y),
    circle_radius)

    # Use speed variable to decrease verticle position of the circle
    through every game loop
    # Below, when the circle speed increases for every 10 points, this will
    increase the amount
    # that the circle drops on every pygame.display.update()
    circle_y += circle_speed

    if circle_y > screen_height:
        # If the circle reaches the bottom of the screen, reset its position
        circle_x = random.randint(0, screen_width - circle_radius * 2)
        circle_y = 0

        # Increase Score for every successful pass
        score += 1
```

```
            # Increase speed for every 10 points
            if score % 10 == 0:
                circle_speed += 0.05

        if player_x < circle_x < player_x + player_width and player_y < circle_y + circle_radius:
            # Collision detected
            game_started = False
            circle_x = random.randint(0, screen_width - circle_radius * 2)
            circle_y = 0
            circle_speed = 0.1 # Reset circle speed
            score = 0

        # Update and display the score
        display_score(screen, score)

        pygame.display.update()

    if not game_started:
        # Display instructions to start the game
        display_instructions(screen)

pygame.quit()
```

Full Code - Circle Escape Challenge

```
import pygame
import random

pygame.init()

# Set up the display window, colors, and variables
screen_width = 400
screen_height = 600
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption("Circle Escape Challenge")

# Define colors and other variables
white = (255, 255, 255) # White color for the background
```

```
player_color = (0, 0, 255) # Blue color for the player
circle_color = (255, 0, 0) # Red color for the falling circle

# Define player rectangle and start location for x and y
player_width = 50
player_height = 10
player_x = (screen_width - player_width) // 2
player_y = screen_height - player_height

# Define the Dropping Circle size, give it a random x location and set the
y=0 to start at the top of the screen
circle_radius = 20
circle_x = random.randint(0, screen_width - circle_radius * 2)
circle_y = 0
circle_speed = 0.1 # Set the initial circle speed to a very slow value (e.g.,
0.1)

score = 0 # Initialize the player's score

# Add a "game_started" flag to control the game state
game_started = False

# Create a font for displaying instructions
font = pygame.font.Font(None, 36)

# Create a font for displaying the score
score_font = pygame.font.Font(None, 36)

# Function to display the score on the screen
def display_score(screen, score):
    score_text = score_font.render("Score: " + str(score), True, (0, 0, 0))
    screen.blit(score_text, (10, 10)) # Adjust the position as needed

# Function to display instructions and wait for player input to start the game
def display_instructions(screen):
    screen.fill(white)
    instructions_text = font.render("Press SPACE to start", True, (0, 0, 0))
```

```
    screen.blit(instructions_text, (screen_width // 2 - 120, screen_height //
    2))
    pygame.display.update()

    waiting_for_start = True
    while waiting_for_start:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()
            if event.type == pygame.KEYDOWN and event.key ==
            pygame.K_SPACE:
                waiting_for_start = False

# Game loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    keys = pygame.key.get_pressed()

    if not game_started:
        # Check if the game hasn't started and the player pressed the SPACE
        key to start
        if keys[pygame.K_SPACE]:
            game_started = True
    else:
        if keys[pygame.K_LEFT] and player_x > 0:
            # Move the player to the left when the LEFT arrow key is pressed
            player_x -= 0.1
        if keys[pygame.K_RIGHT] and player_x < screen_width -
        player_width:
            # Move the player to the right when the RIGHT arrow key is pressed
            player_x += 0.1
```

```
screen.fill(white)

# Draw the Player rectangle and the Dropping Circle

pygame.draw.rect(screen, player_color, (player_x, player_y,
player_width, player_height))
pygame.draw.circle(screen, circle_color, (circle_x, circle_y),
circle_radius)

# Use speed variable to decrease verticle position of the circle
through every game loop
# Below, when the circle speed increases for every 10 points, this will
increase the amount
# that the circle drops on every pygame.display.update()
circle_y += circle_speed

if circle_y > screen_height:
    # If the circle reaches the bottom of the screen, reset its position
    circle_x = random.randint(0, screen_width - circle_radius * 2)
    circle_y = 0

    # Increase Score for every successful pass
    score += 1

    # Increase speed for every 10 points
    if score % 10 == 0:
        circle_speed += 0.05

if player_x < circle_x < player_x + player_width and player_y <
circle_y + circle_radius:
    # Collision detected
    game_started = False
    circle_x = random.randint(0, screen_width - circle_radius * 2)
    circle_y = 0
    circle_speed = 0.1 # Reset circle speed
    score = 0

# Update and display the score
```

```
            display_score(screen, score)

            pygame.display.update()

        if not game_started:
            # Display instructions to start the game
            display_instructions(screen)

    pygame.quit()
```

These are some of the most basic movements and techniques that you've learned. From here, you know enough to get started and start developing some real games. For that, we highly recommend you search for "Pygame projects for beginners" to learn advanced and complex skills, techniques, and projects. The more you practice, the better you become.

Code Challenge

Before we do part ways, there is but one challenge that we have for you. Pong is a classic game where the player is on the right or left side and tries to return the ball back across the screen. We have a challenge, let's call it "Pong Escape Challenge".

The project is to recreate the game using PyGame using the following guide

- Instead of the circle dropping from top to bottom, have the circle move from left to right.
- The player is on the right side and moves up and down
- HINT: You'll have to change some _x and _y variables as well as some _height and _width variables. Key events need to look for _UP and _DOWN to move the player bar.
- No worries, the code is included in the download package

It's simple. It's elegant, and it's still a classic for everyone. Work on it, improve it, debug it, code it, and play it.

Conclusion

With everything said and done, Python isn't just a programming language. It is far more than that. Python has just started emerging as one of the most popular languages on earth, and it is safe to assume that this language will be quite in demand in the near future.

Considering the fact that Machine Learning, Deep Learning, AI, and Generative AI mostly need Python, you can rest assured that you will have plenty of opportunities ahead of you. Whether you are doing this out of hobby or you wish to add a new skill to your resume, know that it is a great skill to have nonetheless.

This book was meant to help you walk through the basic concepts of Python and empower you with enough knowledge to know and understand how programming works. Now that you know, the world is literally your oyster.

There are many projects you can start working on immediately. For more inspiration, be sure to check out great web resources such as:

- W3 school
- Geeksforgeeks
- Codeacademy
- Coderslegacy
- GitHub
- Stack Overflow
- Reddit

There are tons of ideas, projects, and collaborative opportunities to seek and utilize. Where possible, do collaborate with other programmers to learn even more than what books or videos can teach. You never know. You might just end up working in a project that turns out to be the next big thing like Instagram.

We leave you with our best wishes and we would certainly hope to see you find great fortune and success using this newly acquired skill of yours. Go out there and code your heart out!

KEEPING THE GAME ALIVE

You've just crossed the finish line of *The Ultimate Guide to Python Programming for Beginners*! You're now equipped with the knowledge, skills, and confidence to build an exciting career with Python programming.

But wait, your journey doesn't have to end here. You can pass the torch and light the way for others. Remember how you felt when you first opened this book? Excited? Maybe a little unsure? There are countless others out there right now feeling the same, looking for a guide to start their own Python adventure.

A few words from you could be the guiding star for someone else's journey. Your review can highlight the path for others, showing them where to find the same help and inspiration you received.

Your insights and personal story can make all the difference. They can help others to:

- Gain the confidence to start learning Python.
- Understand the practical applications of what they'll learn.
- See the real-life benefits of persevering through the chapters.
- Feel part of a community of learners and future programmers.

It's simple and takes just a moment of your time. Click the link below or scan the QR code to share your thoughts and experiences:

Your review is more than just words; it's a beacon of encouragement, a signpost of possibility. We deeply appreciate your time and effort in helping to foster a community of learning and growth.

Thank you for being an integral part of this journey. As you continue to explore and grow in the world of Python programming, remember that your journey can inspire countless others. Keep the game alive, and let's keep learning together!

With heartfelt thanks, Megabyte Publishing

P.S. - Sharing your journey not only inspires others but also cements your own understanding. As you move forward, consider mentoring or guiding a fellow beginner. Your journey can be the start of someone else's great adventure in Python programming!

REFERENCES

Dyouri, A. (2022, March 9). *How to use Flask-SQLAlchemy to interact with databases in a Flask application | DigitalOcean*. DigitalOcean. https://www.digitalocean.com/community/tutorials/how-to-use-flask-sqlalchemy-to-interact-with-databases-in-a-flask-application

Gupta, S. (2022, September 14). *What is Python used for? 9 use cases [Overview Guide]*. Springboard. https://www.springboard.com/blog/data-science/what-is-python-used-for/

*Introduction to Python***. (**2019). W3Schools. https://www.w3schools.com/python/python_intro.asp

Karim. (2017, November 10). *Python lists (For absolute beginners)*. Afternerd. https://www.afternerd.com/blog/python-lists-for-absolute-beginners/

Krishna, R. (2020, August 31). *How to format strings using print() in Python?* Numpy Ninja. https://www.numpyninja.com/post/how-to-format-strings-using-print-in-python

Novotny, J. (2023, March 9). *A programmers' guide to Python: Advantages & disadvantages*. Linode Guides & Tutorials. https://www.linode.com/docs/guides/pros-and-cons-of-python/

*Python pygame - The full tutorial***. (**n.d.). CodersLegacy. https://coderslegacy.com/python/python-pygame-tutorial/

Python Software Foundation. (2019). *What is Python? Executive summary*. Python. https://www.python.org/doc/essays/blurb/

*Python vs other programming languages***. (**n.d.). StxNext. https://www.stxnext.com/python-vs-other-programming-languages/

Sarkar, P. (2023, September 4). *How to concatenate strings using Python*. KnowledgeHut.

https://www.knowledgehut.com/blog/programming/concatenate-strings-python

Send data to Flask template (Jinja2) - Python Tutorial. (n.d.). PythonBasics. https://pythonbasics.org/flask-template-data/

7 ways to concatenate strings into a string in Python. (n.d.). Python Tutorial. https://www.pythontutorial.net/python-string-methods/python-string-concatenation/

Taff Inc. (2021, August 23). *Top 12 Python use cases & applications with examples | Python Trends 2023*. TAFF. https://www.taffinc.com/blog/top-python-use-cases-and-applications/

TechBootCamps. (2021, September 16). *Why learn Python? Five reasons to start programming with Python in 2022*. UT Austin Boot Camps. https://techbootcamps.utexas.edu/blog/why-learn-python-get-started-programming/

Yegulalp, S. (2023, January 18). *How to install Python the smart way*. InfoWorld. https://www.infoworld.com/article/3530140/how-to-install-python-the-smart-way.html