

ChatGPT: Your Python Coach

Mastering the Essentials in 100 Prompts

Sergio Rojas-Galeano

ChatGPT: Your Python Coach

Mastering the Essentials in 100 Prompts

Sergio Rojas-Galeano, PhD.

Bogotá, Colombia

About the Author

Sergio Rojas-Galeano is an enthusiastic computer scientist with a passion for the cross-roads between Artificial Intelligence, Machine Learning, Evolutionary Computation, Natural Language Processing, Software Engineering, and Scientific Edition. With professional experience from both academic and industry positions in Colombia and England, he is deeply committed to upholding ethical standards and advancing the field of computer science. Currently, he serves as a Full Professor at Universidad Distrital Francisco José de Caldas (Colombia), where he has taught programming and computational intelligence for over two decades. With a PhD in Computer Science from University College London (UCL) in the UK (awarded in 2009), he has also been a visiting scholar at UCL (in 2022) and Universidad de Málaga in Spain (in 2023). He is author or co-author of over 70 scientific papers and has previously published the book, *Models of Learning and Optimisation for Data Scientists* (LeanPub, 2019).

email: sargaleano@gmail.com

License

This document is licensed as:



Permission to use this document is subject to the License; any other use is strictly prohibited. The contents of this document are provided on an "AS IS" basis without warranties of any kind, either express or implied. The author assumes no responsibility or liability for any consequences arising from the use of this document, including but not limited to errors, omissions, or inaccuracies. The reader assumes full responsibility for the application of the concepts and techniques described in this book.

© 2023 by Sergio Rojas-Galeano.

Colophon

This document was typeset using [L^AT_EX](#) and the [kaobook](#) class. The author crafted the content with the help of OpenAI's [ChatGPT](#) content-generation system, based on his own ideas, purpose and style. Cover design was created by the author using templates and art available at the [Canva](#) graphic design platform. All product names, logos, brands, trademarks and registered trademarks are property of their respective owners. All company, product and service names, as well as celebrity, book, character, band, team or movie names mentioned in this book are used solely for identification and educational purposes. Use of these names, trademarks and brands does not imply endorsement.

First Edition

June 2023

Bogotá, Colombia

Code is like humor. When you *have* to explain it, it's bad.

– Cory House

Preface

Welcome to *ChatGPT: Your Python Coach. Mastering the Essentials in 100 Prompts!* If you're reading this, it means you're about to embark on an exciting journey into the world of programming. But why Python, you may ask? Well, Python is one of the most widely used and versatile programming languages out there, with applications ranging from web development and data analysis to machine learning and artificial intelligence (AI).

In fact, there has been a growing interest in AI and its applications in recent years, including the development of language models like ChatGPT. While there are valid concerns regarding the ethical implications, data biases, and potential for generating fake content, it is worth noting that in the realm of programming, these concerns can be somewhat mitigated as the content generated by ChatGPT primarily consists of code, which can be easily evaluated for its validity and appropriateness by simply running it. This provides an advantage in minimising the risks associated with hallucination or inaccurate information, as the output can be verified through practical execution. As such, language models like ChatGPT have become valuable tools for programming tasks, offering developers enhanced assistance in their coding endeavours.

This guide has been carefully crafted to help you navigate the basic concepts of Python programming with ease, with ChatGPT as your trusty mentor. ChatGPT is a powerful language model designed to assist you with any questions or doubts you may have while learning Python. It will provide you with examples, explanations, and feedback, making your learning experience interactive and engaging.

In this guide, you'll learn the fundamentals of Python programming, including variables, data types, operators, flow control structures such as conditionals and loops, containers such as lists and dictionaries and modularisation and reutilisation concepts. You'll also get to work on fun and exciting challenges that will put your newfound skills to the test.

But this guide is more than just a collection of exercises and examples. It's an invitation to explore the power of programming and its potential impact on our lives, and to see how AI and language models like ChatGPT are expected to leverage the abilities and productivity of programmers to create powerful applications for a promising future. As you delve deeper into the world of Python, you'll witness how programming can be used to solve real-world problems, automate tedious tasks, and innovate products and services.

The majority of the content in this book, including this preface, was initially generated with the assistance of ChatGPT. However, it is important to note that the draft material served as a foundation which I then extensively revised, adjusted, and enriched. This process was driven by my educational expertise, with the goal of creating didactic and informative content for readers. While ChatGPT played a role in generating the initial material, the final content has been carefully curated and tailored to ensure its educational value and correctness.

To promote transparency in this innovative approach to AI-assisted writing, I have deliberately chosen to include a comprehensive list of the 100 prompts I conceived and utilised to feed ChatGPT, ensuring thorough coverage of the intended thematic scope of this guide. This list will be incorporated at the end of the book for easy reference. Furthermore, this *List of Prompts* may empower readers to reproduce a similar yet personalised version of the book that aligns with their individual preferences and requirements.

Besides, I will utilise endnotes to reference each prompt in its corresponding position on the list whenever it is used in the text. For instance, the first prompt employed to generate the initial draft of this preface is referred next¹.

Lastly, I've created a companion repository that houses a collection of Jupyter Notebooks that contain the Python code examples from the book, thoughtfully organised by chapters for convenient browsing and exploration:

<https://github.com/Sargaleano/chatgpt-python-coach>

Moreover, starting from Chapter 4, I've added an exciting feature. For the more engaging code examples, I've included links to the [PythonTutor.com](https://pythontutor.com) website. These links allow you to visualise the step-by-step execution of the code, dynamically observing the program's memory state. Look for these links at the bottom of the code listings and output boxes. It's a fantastic way to enhance your understanding and bring the code to life!

So, if you are eager to learn programming and start creating programs with Python and ChatGPT, I hope you find this guide useful and enjoyable to read.

Sergio Rojas-Galeano
Bogotá, June 23, 2023

Contents

Preface	v
Contents	vii
Introduction	3
Is this the right book for you?	3
About Python	3
Programming cycle	4
The tools	5
Using Colab	6
Using PythonTutor	8
ChatGPT as a programming coach	10
Using ChatGPT	11
ChatGPT prompt template	14
1 Know your Basics	15
1.1 Hello, World!	15
1.2 Strings and string formatting	17
1.3 Hands-on time!	22
2 Data Building Blocks	27
2.1 Data types	27
2.2 Basic Input/Output	30
2.3 A coding buddy	32
2.4 Hands-on time!	36
3 The Operations Room	39
3.1 Arithmetic operators	39
3.2 Comparison operators	41
3.3 Logical operators	43
3.4 Bitwise operators	44
3.5 Assignment operators	48
3.6 Identity operators	49
3.7 Memberships operators	51
3.8 Exponentiation operator	52
3.9 Advanced math operators	53
3.10 Hands-on time!	55
4 The Control Tower	59
4.1 If/Else statements	59

4.2	Match-case statements	64
4.3	For loop statements	66
4.4	While loop statements	69
4.5	Nested control structures	74
4.6	Hands-on time!	81
5	The Containers Zone	91
5.1	Lists	91
5.2	Tuples	99
5.3	Sets	103
5.4	Strings	107
5.5	Dictionaries	113
5.6	Hands-on time!	126
6	Modular Affairs	133
6.1	Modularisation and Reusability	133
6.2	Functions	136
6.3	Classes and Objects	143
6.4	Hands-on time!	153
7	Finale	161
7.1	Digging Further	161
7.2	Suggested Readings	167
	List of the 100 prompts	169

List of Figures

1	Program development cycle	4
2	The Colab home screen	7
3	An untitled notebook	7
4	A notebook with text and code cells	8
5	PythonTutor webpage	9
6	PythonTutor visualisation screen	9
7	ChatGPT welcome page	11
8	ChatGPT login page	12
9	ChatGPT home page	12
10	ChatGPT chat page	13
7.1	A suggested learning track for Python Web Development.	161
7.2	A suggested learning track for Data Science with Python.	162
7.3	A suggested learning track for Python for Automation and Scripting.	162
7.4	A suggested learning track for Python for Game Development.	163
7.5	A suggested learning track for Python for DevOps.	163
7.6	A suggested learning track for Advanced Python Concepts.	164
7.7	A suggested learning track for Testing and Debugging in Python.	164
7.8	A suggested learning track for Additional Resources for Python.	165

Introduction

Chapter contents

Is this book for you?.....	3
About Python.....	3
Programming cycle.....	4
The tools.....	5
Using Colab.....	6
Using PythonTutor.....	8
ChatGPT as a coach.....	10
Using ChatGPT.....	11
Prompt templates.....	13

Is this the right book for you?

This book is designed to help newcomers like you learn the basics of programming with ease, specifically in the Python language, one of the most versatile and widely used programming languages out there.

But this book is more than just a typical programming guide. It's like having a dedicated coach by your side, guiding you through the learning journey with examples, explanations, and feedback. And that coach happens to be an AI language model - ChatGPT. With ChatGPT as your guide, you'll have access to a powerful tool that will help you understand the essential programming concepts. You will receive real-time feedback that will help you learn faster and more effectively, ultimately helping you to become a proficient programmer.²

This book serves as an introductory guide to fundamental concepts in Python, including algorithms, flow control structures, containers, and modularisation. Mastering these basics will provide you with a solid foundation for tackling more advanced topics in future volumes. So, embrace these fundamental concepts and gear up for the thrilling challenges that await you. In the meantime, this chapter introduces the tools that will be used throughout the book. If you are already familiar with these tools and want to dive straight into coding, feel free to skip this chapter and jump to Chapter 1.

About Python

Python is a programming language that's as easy as pie – or rather, as easy as writing down a recipe for your favourite dessert! It was created in the late 1980s by a Dutch programmer named Guido van Rossum, who named it after the Monty Python comedy troupe.³

Today, Python is used by everyone from NASA to Instagram to build everything from space rovers to photo filters. It's known for its simplicity, readability, and versatility. In fact, it's one of the most popular programming languages in the world, and for good reason. It's a language that's easy to learn but has powerful capabilities, making it ideal for beginners and experts alike.

Monty Python

Monty Python is a British comedy group known for their unique and absurdist humour. Their television show, Monty Python's Flying Circus, aired from 1969 to 1974 and gained a cult following. They also created several popular films, including Monty Python and the Holy Grail and Life of Brian. Their influence on comedy and pop culture has been significant and continues to be celebrated by fans worldwide.⁴

Python versions

Some key differences between recent versions are:⁵

- Python 2.x has limited support for non-English symbols whereas Python 3.6+ use Unicode encoding for other languages.
- Python 3.6+ allows for extensive text formatting for printing output, compared to previous versions.
- Python 2.x returns integers for integer operands, while Python 3.6+ provides both default decimal division and a separate integer division command.
- Python 3.5 and later added capability for managing multiple tasks concurrently.
- Python 3.10 handles multiple conditions with a single matching test.
- Python 3.11 is the latest version (April/2023).
- For updated information about Python releases, see: <https://devguide.python.org/releases/>.

Python's future is promising, as it is expected to become even more popular in the years to come due to its increasing use in areas such as machine learning, artificial intelligence, and data science.

There are plenty of Python resources available online. The official Python website (<https://www.python.org/>) has a wealth of information, including tutorials, documentation, and downloads. Other useful websites include Real Python (<https://realpython.com/>), Python for Beginners (<https://www.pythonforbeginners.com/>), and Codecademy (<https://www.codecademy.com/learn/learn-python>).

Programming cycle

Programming is a creative process that requires careful planning, focus, and attention to detail. Whether you're writing a simple script or a complex application, there is a standard cycle of program development that you should follow to ensure your code works as intended. The four key steps of this cycle are shown in Figure 1.⁶

Let's go over each step in detail:⁷

- ▶ **Write (coding):** The first step is to write the code. This is where you turn your ideas into actual instructions that the computer can understand. You'll use a programming language like Python to write your code, and you'll need to pay close attention to the syntax and structure of your code.
- ▶ **Run:** Once you've written your code, it's time to run it. This is where you see your code in action and test whether it works as intended. You might use a program like IDLE to run your code, or you might use a command line interface to execute your code.

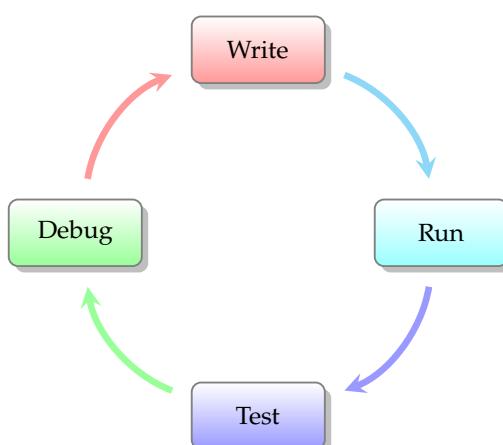


Figure 1: A typical program development cycle

- ▶ **Test:** Testing is an essential part of the program development process. This is where you check whether your code is working correctly and identify any errors or bugs. You'll need to run different test cases and scenarios to ensure that your code is robust and works in a variety of situations.
- ▶ **Correct (debug):** Finally, if you find any errors or bugs during testing, it's time to correct them. This is known as debugging. You'll need to go back to your code and identify the cause of the error. This might involve stepping through your code line by line or using debugging tools to trace the flow of your code. Once you've identified the issue, you can make the necessary changes to your code to fix it.

Remember, programming is a journey of continuous learning and improvement. It takes time and effort to develop good programming skills. Embrace the challenges that come your way, and don't be afraid to ask for help when needed. By adhering to these principles and dedicating yourself to your craft, you'll be well on your way to creating software that not only meets the needs of your users but also may have a positive impact on the world.

The tools

Python is an interpreted computer language, which means that the code is executed directly line by line, without requiring a compilation step. This is different from compiled languages like C++, Java, or Go, where the source code is compiled into machine code that can be executed by a computer.⁸

One advantage of an interpreted language like Python is that it allows for more rapid development and prototyping. Since the code can be executed immediately, developers can quickly test and debug their programs. Nonetheless, interpreted languages can be slower to execute than compiled languages, as the code has to be re-interpreted each time it runs. This can be a concern for programs that require high performance or deal with large datasets.

There are numerous Python Integrated Development Environments (IDEs) available to carry out the development cycle of Figure 1. These are software applications, such as PyCharm, Spyder, and Visual Studio Code, that offer advanced features to assist experienced developers in speeding up the development process and increasing productivity.

In this book, however, we want to focus on the concepts and algorithms behind programming without the inconvenience of software installation.

Therefore, we will turn to online ready-to-use tools as our primary platform to write, run, and test our Python code. These tools are detailed below:

- ▶ **Jupyter Notebooks**: a web-based interactive computational environment that allows you to create and share documents that contain live code, equations, visualisations, and narrative text. Website: <https://jupyter.org/>. No account required to use.
- ▶ **Google Colaboratory (Colab)**: a free cloud-based Jupyter Notebook environment provided by Google. It allows you to run Python code, in your browser with zero configuration required. To run Colaboratory, you simply need to log in to your Google account, go to the website (<https://colab.research.google.com/>) and click "New Notebook" to create a new notebook. From there, you can start writing Python code and running it in real-time. Your work will be saved to your Google Drive account.
- ▶ **PythonTutor**: a web-based visualiser for Python code that helps students understand how the code is executed by the Python interpreter. To use PythonTutor, you need to go to the website (<http://pythontutor.com/>); from there, you can either paste in your Python code or write it directly on the site and click "Visualize Execution". No account is required to use it.
- ▶ **ChatGPT**: a language model that can assist in learning, coding, explaining, debugging, documenting and providing feedback about Python programs. Website: <https://chat.openai.com>. Requires an OpenAI account.

Using Colab

Let's see how to create a new Jupyter notebook in Google Colab, where we can carry out basic Python programming. Firstly, log in to your personal Google account (in case you need it, refer to this link to create one: <https://support.google.com/accounts/answer/27441?hl=en>). Then, follow these steps:⁹

1. Go to <https://colab.research.google.com/> in your web browser to open Colab. The screen shown in Figure 2 will appear.
2. Choose *File* → *New Notebook* in the menu bar. This will create a new, untitled notebook. You can give it a title by clicking on the "Untitled0.ipynb" text at the top of the page and typing in a new name, as shown in Figure 3.

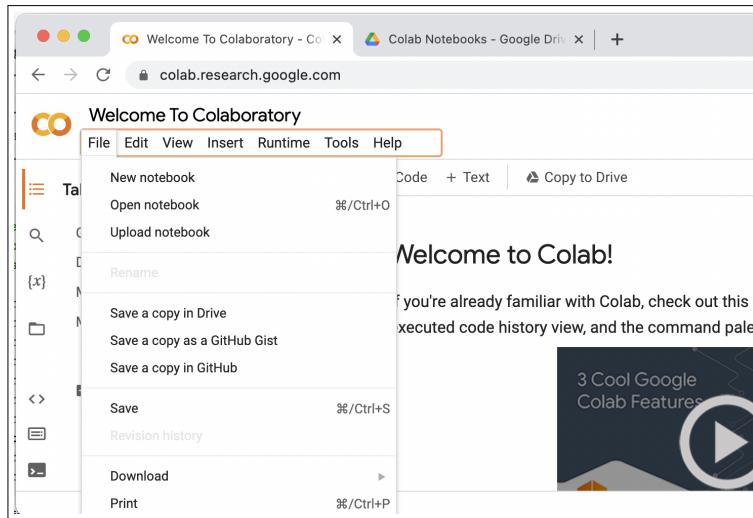


Figure 2: The Colab home screen.

- Now, you can start writing the contents in the cells of the notebook. There are two types of cells in Jupyter Notebook:¹⁰

- ▶ **Code cells:** These cells allow the user to write and execute code in the Notebook. Code cells are denoted by the `In []` prompt on the left-hand side of the cell. When you execute the code in a cell, the output is displayed in a textbox below the cell, and a number appears in the square brackets to the left of the cell indicating in which order it was executed. Colab supports a variety of programming languages, but the default is Python.
- ▶ **Markdown cells:** These cells allow the user to add text, images, and other types of rich media to the Notebook. Markdown cells use the Markdown language, which is a lightweight markup language that allows for easy formatting of text using simple syntax. When you run a Markdown cell, the text is rendered as formatted text or media.

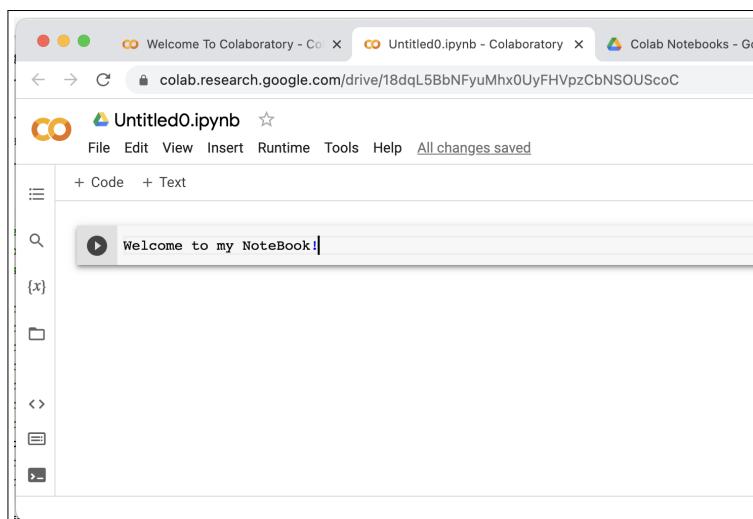


Figure 3: An untitled notebook.

4. You can run the code in the cell by clicking on the  button on the left side of the cell, or by using the keyboard shortcut "Shift+Enter".
5. If you need to add more cells, you can click on the  button on the top left of the page or go to the menu *Insert* → *Code Cell* or *Insert* → *Text Cell*.
6. To save your notebook, you can either click on the *File* → *Save* or use the keyboard shortcut "Ctrl+S".
7. Colab automatically saves your notebook to your Google Drive, so you can access it from any device as long as you are signed in to the same Google account.

Figure 4 shows an example of a notebook, which includes a markdown cell with introductory text, and a code cell with a "Hello, World!" program. The output after running the code can also be seen.

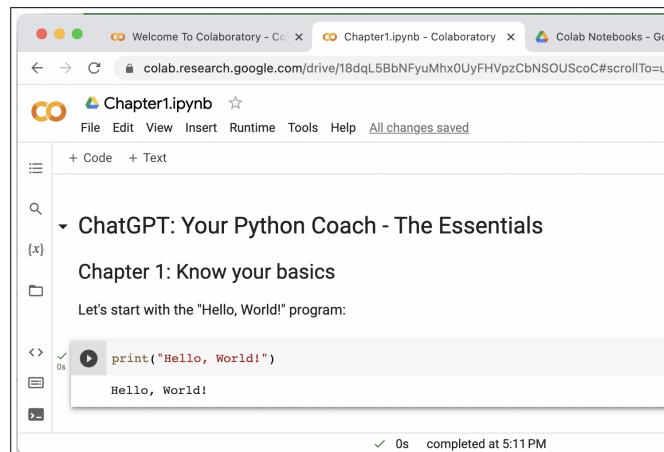


Figure 4: A notebook with text and code cells.

Using PythonTutor

PythonTutor is an online tool that allows you to visualise the execution of Python code step by step. It can help you understand how Python code works by showing you how the code executes line by line and how the values of variables change over time.¹¹

To use PythonTutor, you need to follow these steps:

1. Go to the PythonTutor website at www.pythontutor.com. The screen in Figure 5 will show up.
2. Choose the Python language by clicking on the appropriate button. Select the Python 3.6 version.
3. Enter your Python code into the text box provided.

Learn Python, JavaScript, C, C++, and Java

This tool helps you learn Python, JavaScript, C, C++, and Java programming by [visualizing code execution](#). You can use it to debug your homework assignments and as a supplement to online coding tutorials.

Start coding now in [Python](#), [JavaScript](#), [C](#), [C++](#), and [Java](#)

Over 15 million people in more than 180 countries have used Python Tutor to visualize over 200 million pieces of code. It is the most widely-used program visualization tool for computing education.

You can also embed these visualizations into any webpage. Here's an example showing recursion in Python:

```

Python 3.6
1 def listSum(numbers):
2     if not numbers:
3         return 0
4     else:
5         (f, rest) = numbers
6         return f + listSum(rest)
7
8 myList = (1, (2, (3, None)))
9 total = listSum(myList)

Edit this code
line that just executed
next line to execute
< Prev Next >
  
```

Figure 5: PythonTutor webpage, where you can visualise and step through the execution of Python code.

4. Select the options for visualisation by clicking on the checkboxes and dropdown menus provided. You can choose to see the code output, the values of the variables, and the call stack, among other options.
5. Click the "Visualize Execution" button to start the visualisation.
6. Once the visualisation starts, you will see a timeline on the left-hand side of the screen that shows each step in the execution of your code. You can step through the code one line at a time by clicking the "Next" or "Prev" button or use the slider to go through the execution. An example of the visualisation of a program can be seen in Figure 6.

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

Python 3.6 known limitations

```

1 numerator = int(input("Enter the numerator: "))
2 denominator = int(input("Enter the denominator: "))
3 if denominator == 0:
4     print("Error: Division by zero is not allowed.")
5 else:
6     result = numerator / denominator
7     print(f"The fraction {numerator}/{denominator}
8 is equal to decimal: {result:.02f}")

Edit this code
line that just executed
next line to execute
  
```

Print output (drag lower right corner to resize)

Enter the numerator: 4
Enter the denominator: 0

Frames Objects

Global frame

numerator	4
denominator	0

Figure 6: PythonTutor visualisation screen.

7. As you step through the code, the right-hand side of the screen will show the current state of the program, including the values of the variables and any output generated by the code.
8. You can also interact with the visualisation by clicking on the code or the variables to see more detailed information.

ChatGPT as a programming coach

Disclosure

This book was written with the assistance of ChatGPT. Throughout the book, pieces of automatically generated content served as a draft, which I then reviewed, verified, adjusted, remixed, and augmented until they reached their final form. For the sake of transparency, I've also included the prompts I used to feed the model to generate the drafts, which I labelled with a superscript number directly following the final text fragment where they were used. These numbers indicate their position in the *List of Prompts* provided at the end of the book.

ChatGPT is a language model created by OpenAI that is designed to communicate with humans using natural language. It is a type of artificial intelligence that can understand and respond to text input in a conversational way. It has been trained on a vast amount of data to be able to understand and generate natural language, including structured high-level computer languages, making it an excellent tool to help students learn programming in a conversational entertaining way.¹²

As a student learning programming, ChatGPT can serve as a helpful coach. You can ask it questions about programming concepts, ask for examples of code, or even get feedback on your own code. It can provide explanations and clarifications on programming topics and serve as a tool for practising coding skills.

Concretely, here are some ways ChatGPT can help you learn programming:

- ▶ It can provide explanations and examples of programming concepts and syntax, helping you to better understand them.
- ▶ It can answer coding questions, such as why a particular syntax rule is needed or how to carry out a specific task.
- ▶ It can help you debug your code by suggesting possible solutions and explaining why certain errors occur.
- ▶ It can assist you in learning how to write efficient and optimised code by providing feedback on your code's performance.
- ▶ It can suggest resources and learning materials to help you further your understanding of programming.
- ▶ It can help you document your code so that it is easier for other programmers to understand or for you to remember in future occasions.
- ▶ It can translate code between different programming languages to create new versions of your programs.
- ▶ It can provide contextual references about the meaning, history, or anecdotes related to the development of programming and computer science in general.

Although ChatGPT can be a valuable tool for learning programming, it should not be relied on as the sole source of information or feedback. It's important to use it responsibly, cross-check responses with other sources, and be aware of potential inaccuracies or biases in the data used to train this AI language model.

On the topic of ethical concerns related to the use of ChatGPT, it is essential to note that they cannot replace key skills you need to acquire as a talented programmer, particularly problem-solving and critical thinking. While ChatGPT can be an excellent companion to your learning journey or even a coding buddy when you embark on your programming projects, it cannot replace (at least not at the time of writing this book), the benefits you can get from human ingenuity, interaction, and guidance.

Moreover, you will soon realise the significance of refining your skill to ask well-formed questions to make the most of the ChatGPT in this coaching experience. This is now referred to as *prompt engineering*, which involves clearly stating to the model what you need help with, in a way that reduces inaccuracies or false results.

Overall, ChatGPT is an incredible resource for enhancing your programming skills, but use it responsibly. With a discerning mindset, you can leverage its potential to become a proficient programmer and start a promising career in the field.

Using ChatGPT

Here's a brief overview of how to use ChatGPT:

1. Open your web browser and navigate to chat.openai.com. Once the page loads, you should see the screen in Figure 7:

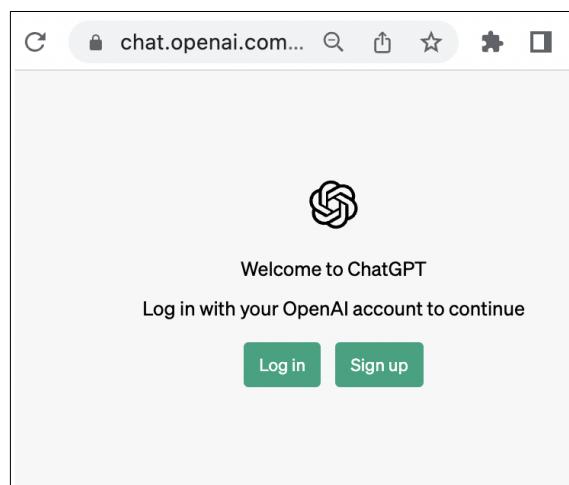


Figure 7: ChatGPT welcome page.

2. Log in to your OpenAI account using the screen shown in Figure 8. If you don't have an account, choose the "Sign up" option and follow the instructions.

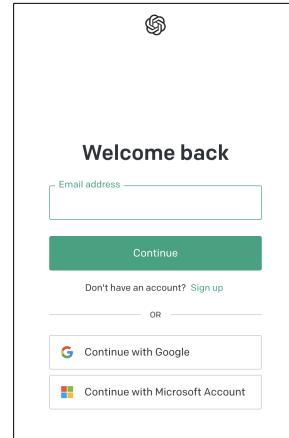


Figure 8: ChatGPT login page.

- Once you've signed in, you'll be taken to the ChatGPT home page, which is shown in Figure 9. In the "Send a message" text box, enter the question (that is, the *prompt*) you'd like to send to ChatGPT, and then hit the "Enter" key or click the arrow button.

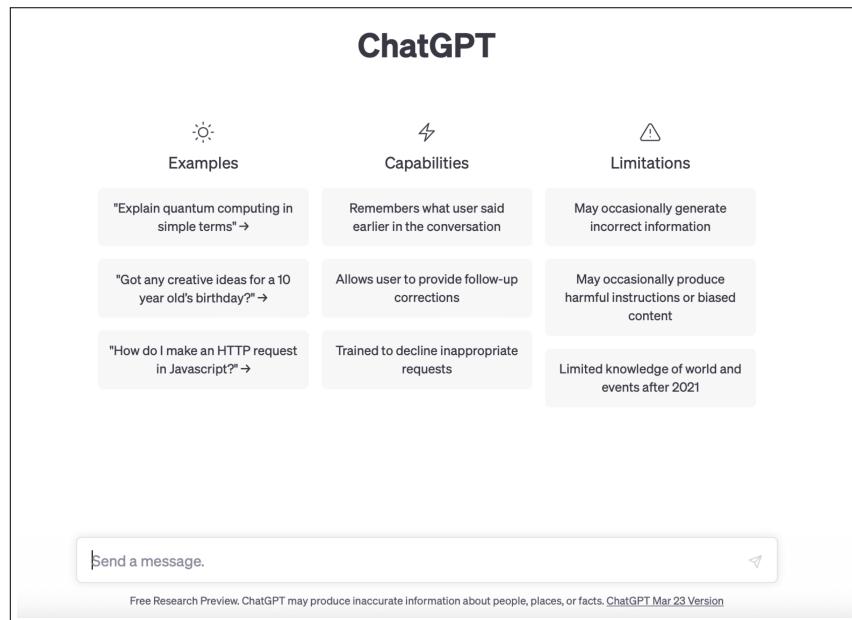


Figure 9: ChatGPT home page.

- Wait for ChatGPT to generate a response. The response will appear in the chat window below your message. Note that the response may include explanatory text as well as programming code. You can click on the "Copy code" button, as illustrated in the screenshot in Figure 10, to copy the code and paste it in a platform such as Colab or PythonTutor to execute the program.
- If the response doesn't fully answer your question or you need more information, type in another prompt, send it to ChatGPT and continue the conversation. The history of past conversations is kept on the left-hand side of your screen, so you can return and resume any of them if you like.

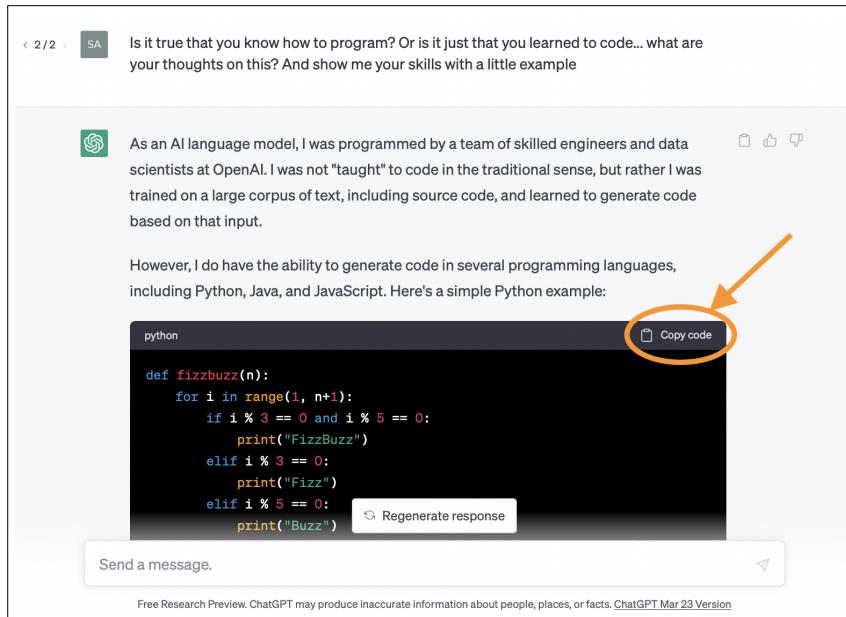


Figure 10: ChatGPT chat page.

In this book, I will use ChatGPT as described before. To make the material more readable, I will format the prompts, explanatory text, code and output of the response, using different coloured text boxes. For the sake of illustration, I have reproduced the above screenshot in the aforementioned formatted style:

Is it true that you know how to program? Or is it just that you learned to code... what are your thoughts on this? And show me your skills with a little example.

As an AI language model, I was programmed by a team of skilled engineers and data scientists at OpenAI. I was not "taught" to code in the traditional sense, but rather I was trained on a large corpus of text, including source code, and learned to generate code based on that input. Here's an example with its output:

```

1 def fizzbuzz(n):
2     for i in range(1, n+1):
3         if i % 3 == 0 and i % 5 == 0:
4             print("FizzBuzz")
5 ...

```

```

1
2
Fizz
4
Buzz
...

```

ChatGPT coach prompt templates

As mentioned earlier, crafting effective prompts when asking ChatGPT for help with programming is a skill that will develop as you interact with the tool and achieve intermediate goals. In the following chapters, we will be using the following prompt templates to cover various programming topics:¹³

1. Introducing and explaining programming concepts:
 - ▶ "Explain [programming concept] in a simple way."
 - ▶ "What's the intuition behind [programming concept]."
 - ▶ "Provide a real-life analogy of [programming concept]."
2. Writing code examples related to programming concepts:
 - ▶ "Write a code example for [programming concept]."
 - ▶ "What's a practical application of [programming concept]?"
 - ▶ "How [programming concept] is used to solve [some problem]?"
 - ▶ "Illustrate the use of [several programming concepts] in a single code example, that [uses/computes/comparisons] [data/information/statistics] about [real-life topic]. Please include explanatory comments in each block of the code."
3. Describing key lines of code:
 - ▶ "What does this line of code mean: [code snippet]?"
 - ▶ "How does [specific line of code] contribute to the program's functionality?"
 - ▶ "What happens if I change [specific line of code] to [different code]?"
4. Documenting and commenting code:
 - ▶ "Write documentation strings for my Python code"
 - ▶ "Add inline comments to describe the functionality of blocks of code in my program"
5. Generating new challenges:
 - ▶ "Suggest some similar exercises to [problem]"
 - ▶ "What variations of [problem] can I try to solve?"
 - ▶ "How can I practice [programming concept] in different scenarios?"
6. Including useful links to resources:
 - ▶ "Recommend any good resources for learning [programming concept]?"
 - ▶ "Where can I find more information on [programming concept]?"
 - ▶ "What are some useful online tutorials for [programming concept]?"
7. Making the topic more enjoyable:
 - ▶ "Tell me a programming anecdote related to [programming concept]?"
 - ▶ "List some fun facts about [programming concept]"

So, without any more hesitation, let's dive straight into our thrilling journey of learning Python with the help of ChatGPT. I hope you find this experience fun and rewarding.

1 Know your Basics

1.1 Hello, World!

Explain the concept of the 'Hello, World!' program in a simple but colorful manner, without using any code. Additionally, include a relatable anecdote to provide beginners in programming with a warm introduction to the topic.¹⁴

Ah, the legendary "Hello, World!" program! This is the perfect way to kick off your journey into programming, and it's an experience that you'll always remember. Imagine yourself sitting at your computer, feeling a mix of excitement and nervousness. You're about to write your first Python program ever!

The "Hello, World!" program is the simplest program you can write, but it's a significant event. It's the equivalent of a newborn baby's first cry or a musician's first note on a new instrument. This program is the foundation of all programming languages, and it's the perfect starting point for anyone who wants to learn to code.

Now, you might be wondering, "Why on earth would I want to write a program that just says 'Hello, World!'?" Well, it's not about the message itself, but rather about the process of writing the program. This simple program is a great way to get familiar with a new language, overcome the fear of starting something new, and gain the confidence to continue learning and exploring.

But why the message "Hello World"? Who invented it? What's the meaning?¹⁵

The origin of the "Hello, World!" program is somewhat unclear, but it is believed to have originated from the Bell Labs in the early 1970s. The first recorded instance of the phrase being used in a programming context was in a 1974 book about the programming language C by Brian Kernighan and Dennis Ritchie.

As for the meaning behind the message, it is simply a way for programmers to test that their program is functioning properly and able to display output. It's a tradition and a rite of passage for beginners in the programming world, signalling the start of their journey towards becoming proficient programmers.

Well, enough with the words. Let's dive into the code and begin the ChatGPT coaching experience.

Ready, set, code!

Write the classic "Hello, World!" program in Python and explain it. Provide the output as well. I will be using Colab.¹⁶

Here's the classic "Hello, World!" program in Python:

```
1 print("Hello, world!")
```

The `print()` function is a built-in function in Python that allows you to output text or other data to the console, in this case the text (also known as *string*) "Hello, World!". The output should look like this:

Hello, world!

To run this program in Google Colab, follow these steps:

1. Click on the "Copy code" button on ChatGPT.
2. Paste it in the first cell of your notebook in Colab.
3. Click the "play" button next to the cell or press "Shift + Enter"

Explain what a syntax error is and demonstrate 3 possible syntax errors in the 'Hello, World!' program with their respective outputs.¹⁷

When writing a program, you need to follow the correct syntax of the programming language. A syntax error occurs when you make a mistake in the syntax of the code, such as missing a punctuation mark, using the wrong case for a keyword, or typing an incorrect operator. These errors prevent the code from being interpreted correctly. Here are three possible syntax errors that could have occurred in the "Hello, World!" program:

1. Missing Parenthesis:

```
print("Hello, World!"
      File "<stdin>", line 1
      print("Hello, World!"
            ^
SyntaxError: unexpected EOF while parsing
```

The error message tells us that the unexpected end of file (EOF) was encountered while parsing the code.

2. Misspelled Keyword:

```
prnt("Hello, World!")
      File "<stdin>", line 1
      prnt("Hello, World!")
            ^
NameError: name 'prnt' is not defined
```

Here, the error message points out that no command `prnt()` exists.

3. Missing Quotes:

```
print(Hello, World!)
  File "<stdin>", line 1
print(Hello, World!
      ^
SyntaxError: invalid syntax
```

Here the error indicates that the string message must be enclosed in quotation marks.

*Why is it called `print()`? Do I need a printer to use it?*¹⁸

The `print()` function in Python is used to display or print a given message or output to the console. It is called "print" because, originally, it referred to printing out information to a physical printer. This was back when computers did not have screens. However, in modern times, it simply means to display information on a computer screen, and there is no need for a printer to use it.

1.2 Strings and string formatting

*Explain in a simple way what a string is, and the syntax rules for defining strings in Python.*¹⁹

In Python, a string is a sequence of characters enclosed in quotes (single quotes or double quotes). It can contain letters, numbers, symbols, and whitespace. Strings are commonly used to store and manipulate text data in programming.

To define a string in Python, you simply enclose the desired sequence of characters in quotes. For example:

```
1 my_string = "Hello, World!"
2 print(my_string)
```

Hello, World!

In Python, you can use either single quotes or double quotes to define a string. The important thing is to be consistent with your choice of quote marks throughout your code.

Here are some examples of defining strings in Python. Each string is assigned, that is, stored in memory in a variable, by means of the assignment operator '='. Notice you can print multiple strings using a comma-separated list of variables. Use the `sep='\\n'` modifier to print in different lines of output:

```

1 string1 = "This is a string using double quotes"
2 string2 = 'This is a string using single quotes'
3 string3 = "This string contains numbers and symbols: $5.99"
4 print(string1, string2, string3, sep='\n')

```

This is a string using double quotes
 This is a string using single quotes
 This string contains numbers and symbols: \$5.99

You can also use triple quotes (either single or double) to define a multi-line string:

```

1 multi_line_string = """This is a
2     multi-line
3         string"""
4 print(multi_line_string)

```

This is a
 multi-line
 string

In addition, you can use escape characters (e.g. '\n' for a new line, '\t' for a tab) to include special characters within a string. For example:

```

1 escaped_string = "This string contains a new line.\nHere's the next line.\n\tAnd here's
   another line indented with a tab to the right"
2 print(escaped_string)

```

This string contains a new line.
 Here's the next line.
 And here's another line indented with a tab to the right

*Explain string formatting in simple terms. Write fun code examples for each of the formatting options.*²⁰

String formatting is the process of creating a new string by embedding the values of variables or expressions into placeholders within a string. It allows you to create dynamic strings that can display different values based on changing input.

In Python, there are multiple ways to format strings:

- **f-strings:** These are the newest and easiest way to format strings in Python. You simply include the variable or expression in curly braces within a string, preceded by the letter "f".

- ▶ **str.format()**: This is a slightly older method of string formatting that is still widely used. It involves calling the `format()` method on a string and passing in the values you want to insert into the placeholders using positional or keyword arguments.
- ▶ **% operator**: This is an older method of string formatting that is less commonly used these days. It involves using the `%` operator to substitute the values of variables or expressions into placeholders in the string.
- ▶ **concatenation**: In this approach, the `+` operator is used to join together a single string.

Here are some examples of each type of formatting:

```

1 # f-strings
2 name = "Alice"
3 age = 30
4 print(f"{name} is {age} years old and loves Python!")
5
6 # str.format()
7 food = "pizza"
8 price = 10.99
9 print("I love {0}, it costs ${1:.2f} per slice.".format(food, price))
10
11 # % operator
12 city = "Paris"
13 country = "France"
14 print("I'd love to visit %s in %s someday." % (city, country))
15
16 # concatenation
17 print('Hello, ' + 'World' + '!')

```

```

Alice is 30 years old and loves Python!
I love pizza, it costs $10.99 per slice.
I'd love to visit Paris in France someday.
Hello, World!

```

*Why are there some green lines in the code that begin with a hash symbol?*²¹

The green lines in the code that begin with a hash symbol are called comments. Comments are used to provide explanatory notes or reminders for the person reading the code. The Python interpreter ignores comments, so they have no effect on the program's execution. Comments are a useful way for programmers to communicate with one another, and they can also help to make code easier to understand and maintain.

Ok. Now provide an example of a program using only print() commands with a real-world analogy.²²

Here's an example of a program that prints a recipe on the screen:

```

1 # Example program: "Cooking Pancakes"
2 # This program demonstrates the steps for cooking pancakes
3 print("-"*30 + " Cooking Pancakes " + "-"*30)
4 print("Step 1: Mix the pancake batter.")
5 print("Step 2: Heat the griddle or frying pan on medium heat.")
6 print("Step 3: Pour batter onto the griddle or frying pan, using a 1/4 cup measuring cup.
    ")
7 print("Step 4: Cook for 1-2 minutes, until the edges start to dry and the surface is
    bubbly.")
8 print("Step 5: Flip the pancake with a spatula, and cook for an additional 1-2 minutes on
    the other side.")
9 print("Step 6: Remove the pancake from the griddle or frying pan and place on a plate.")
10 print("Step 7: Repeat steps 3-6 for the remaining batter.")
11 print("Step 8: Serve with your favorite toppings, such as butter and syrup.")
```

The output is shown below:

```

----- Cooking Pancakes -----
Step 1: Mix the pancake batter.
Step 2: Heat the griddle or frying pan on medium heat.
Step 3: Pour batter onto the griddle or frying pan, using a 1/4 cup measuring cup.
Step 4: Cook for 1-2 minutes, until the edges start to dry and the surface is bubbly.
Step 5: Flip the pancake with a spatula, and cook for an additional 1-2 minutes on
    the other side.
Step 6: Remove the pancake from the griddle or frying pan and place on a plate.
Step 7: Repeat steps 3-6 for the remaining batter.
Step 8: Serve with your favorite toppings, such as butter and syrup.
```

Can I use formatting options such as width and alignment to build strings? Show me engaging examples.²³

You can use width and alignment formatting options to build strings in Python. Width specifies the total width of the string, while alignment specifies how to align the string within that width. Here are some examples:

1. **Centered Text:** You can center a string by using the `{:^width}` pattern to indicate center alignment and specifying the width. Notice how the expression `'-'*8` is used to repeatedly print the character "-".

```

1 text = "Hello, World!"
2 centered_text = f"|{text:^40}|" 
3 print(' - '*8, centered_text, ' - '*8)
4

```

----- | Hello, World! | -----

2. **Left-aligned Text:** You can left-align a string by using the `{:<width}` pattern to indicate left alignment and specifying the width.

```

1 name = "John"
2 age = 30
3 job = "Engineer"
4 output = f" |Name: {name:<10}|Age: {age:<5}|Job: {job:<10}|" 
5 print(output)
6

```

|Name: John |Age: 30 |Job: Engineer |

3. **Right-aligned Text:** You can right-align a string by using the `{:>width}` pattern to indicate right alignment and specifying the width.

```

1 name = "Sarah"
2 age = 28
3 job = "Designer"
4 output = f" |Name: {name:>10}|Age: {age:>5}|Job: {job:>10}|" 
5 print(output)
6

```

|Name: Sarah|Age: 28|Job: Designer|

4. **Truncated Text:** You can truncate a string by specifying the maximum width, using the pattern `{:.width}` within the placeholder.

```

1 text = "This is a very long string that needs to be truncated"
2 truncated_text = f"{text:.20}"
3 print(truncated_text+"...")
4

```

This is a very long ...

5. **Zero-padded Numbers:** You can zero-pad a number by using the pattern `{:0width}` pattern to indicate zero-padding and specifying the width.

```

1 number = 42
2 padded_number = f"{number:05}"
3 print(padded_number)
4

```

00042

6. **Padding and Decimal Places:** To format a floating-point number with a specific number of decimal places and padding, using the pattern `{:width.precisionf}`.

```

1 num = 3.14159
2 formatted_num = "{:08.2f}".format(num)
3 print("The formatted number is:", formatted_num)
4

```

The formatted number is: 00003.14

*What do you mean by "floating-point" number?*²⁴

A floating-point number is a numerical value that has a decimal point. It is often used to represent real numbers in programming languages. Floating-point numbers can be either positive or negative, and can have varying degrees of precision, depending on the number of digits used to represent them. They are called "floating-point" because the decimal point can "float" or move to different positions within the number, depending on the magnitude of the number. They are represented in a computer's memory using a standard format that allows for efficient computation and storage. In Python, floating-point numbers are represented using the `float` data type.

1.3 Hands-on time!

*I'd like to put what I've just learned about `print()` and string formatting into practice. Please provide four code exercises where I need to fix the errors to make them run correctly.*²⁵

Here are four exercises where you have to correct the errors so they can run properly:

Exercise 1:

```

1 print(Hello, World!)

```

Exercise 2:

```
1 message = "I have {} apples"
2 print(message+format(5))
```

Exercise 3:

```
1 name = input("What is your name? ")
2 age = input("How old are you? ")
3 print("Your name is {} and you are {} years old."format(name, age))
```

Exercise 4:

```
1 number = 42
2 print("The answer is {:.10}".format(number))
```

I'd like to put what I've just learned about print() and string formatting into practice. This time, suggest some exercises for me to write the code that can be solved using only print() and string formatting.²⁶

Exercise 1:

Print a shopping receipt such as the one shown below.

RECEIPT		
Item	Quantity	Price
Apples	3	\$1.50
Bananas	2	\$0.80
Oranges	4	\$2.00

Exercise 2:

Create a program that prints a right triangle using the '*' character, see below.

```
*
```

```
* *
```

```
* * *
```

```
* * * *
```

```
* * * * *
```

Exercise 3:

Create a program that prints the following sum in the screen:

```

1
20
300
+ 4000
-----
4321

```

One interesting aspect of ChatGPT is its ability to revise, correct, and provide feedback on a piece of code. This feature can be incredibly helpful for programmers, especially those who are new to coding or who are struggling with a particular issue. By simply submitting a code snippet or a problem to ChatGPT, users can receive insightful feedback, suggestions for improvement, and even corrected code.²⁷

This feature can save time and improve the quality of code by providing a fresh perspective and catching errors that might have been missed. Additionally, ChatGPT's ability to provide explanations for its revisions and feedback can help users better understand programming concepts and best practices.

Overall, ChatGPT's ability to revise, correct, and provide feedback on code is a powerful tool that can benefit both new and experienced programmers. By leveraging the knowledge and expertise of this AI language model, users can improve their coding skills and produce better quality code.

The following template can be useful for this purpose:²⁸

Below is the code I wrote to solve the exercise: [insert the exercise statement here].

Please revise/improve/correct my code and provide feedback on how to make it better, more efficient, or more readable. Also, demonstrate how you would have coded the solution differently, if applicable.

Code:

[Insert code here]

For the sake of illustration, let's apply this template for Exercise 2:

Below is the code I wrote to solve the exercise: "Create a program that prints a right triangle using the '' character.".*

Please revise/improve/correct my code and provide feedback on how to make it better, more efficient, or more readable. Also, demonstrate how you would have coded the solution differently, if applicable.²⁹

Code:

```
1 print('* ')
2 print('* ' * 2)
3 print('* ' * 3)
4 print('* ' * 4)
5 print('* ' * 5)
```

Here is an improved version of your code:

```
1 for i in range(1, 6):
2     print('* ' * i)
```

This code uses a for loop and the range function to print each line of the rectangle triangle. It's more concise and eliminates the need to write out each line separately.

Note that there are many possible ways to solve this problem, and the best solution depends on the requirements and constraints of the problem.

Anyway, great effort on your initial code! Keep practising and learning.

(NB: ChatGPT is correct in suggesting that using a **for loop** would be a more elegant and concise approach to solve the problem. More on loops will be covered in Chapter 4.)

Data Building Blocks 2

2.1 Data types

*What exactly is a program? How does it relate to data?*³⁰

A program is a set of instructions that tells a computer what to do. It is a sequence of statements that are executed by the computer in order to accomplish a specific task. Programs are written in a programming language, which is a set of syntax rules that define how the instructions are to be written.

Data is any kind of information that can be stored and processed by a computer. It can be numbers, text, images, audio, or video. Data is used by programs to perform their tasks. For example, a program that calculates the average of a set of numbers will need to read the numbers from a data file or ask the user to enter them.

The relationship between programs and data is that programs can be used to create, store, retrieve, and manipulate data. They can also be used to analyse data and to generate new information.

Here are some examples of how programs and data are related:

- ▶ A word processor uses data (text) to create documents.
- ▶ A spreadsheet uses data (numbers) to calculate values.
- ▶ A database uses data (records) to store information.
- ▶ A web browser uses data (web pages) to display information on a screen.
- ▶ A game uses data (characters, levels, etc.) to create an interactive experience.

*Ok. Explain with examples and outputs how data is represented in Python. Provide examples from comics characters to better understand the concept.*³¹

In Python, data is represented by assigning values to variables. These values can be of different types, such as strings, integers, and booleans. Here are some examples:

```
1 # Assigning a string to a variable
2 hero = "Spider-Man"
3 print("Character:", hero)
4 # Assigning an integer to a variable
5 age = 23
6 print("Age:", age)
7 # Assigning a boolean to a variable
8 is_villain = False
9 print("Is villain?", is_villain)
```

```
Character: Spider-Man
Age: 23
Is villain? False
```

Variables can also be assigned values of different types:

```
1 # Assigning a string to a variable and then changing it to an integer
2 power = "Spider-sense"
3 print("Power:", power)
4
5 power = 10
6 print("Power:", power)
7
8 # Assigning a boolean to a variable and then changing it to a string
9 is_hero = True
10 print("Is hero?", is_hero)
11
12 is_hero = "yes"
13 print("Is hero?", is_hero) # Output: yes
```

```
Power: Spider-sense
Power: 10
Is hero? True
Is hero? yes
```

Python also allows for multiple assignment in a single line, like this:

```
1 # Assigning multiple variables at once
2 name, team, universe = "Iron Man", "Avengers", "Marvel"
3 print("-"*10)
4 print("Character:", name)
5 print("Team:", team)
6 print("Universe:", universe)
7
8 # Update the values of a variable
9 name, team, universe = "Wonder Woman", "Justice League", "DC Comics"
10
11 print("-"*10)
12 print("Character:", name)
13 print("Team:", team)
14 print("Universe:", universe)
```

```
-----
Character: Iron Man
Team: Avengers
Universe: Marvel
-----
Character: Wonder Woman
Team: Justice League
Universe: DC Comics
```

Overall, you can think of data types as different types of containers that hold different kind of things. Just like in real life, we have different containers for different types of things. For example, we use a glass for holding liquids, a box for holding objects, and a bag for holding groceries. Similarly, in Python, we use different data types for holding different kinds of data.

*So, how can I know the type of a variable at a given point in the program?*³²

You can use the built-in function `type()` to determine the type of a variable in Python. For example:

```
1 name = "Peter Parker"
2 age = 25
3 height = 1.83
4 is_spiderman = True
5
6 print("Type of name:", type(name))
7 print("Type of age:", type(age))
8 print("Type of height:", type(height))
9 print("Type of is_spiderman:", type(is_spiderman))
10
11 # Change value (and type) of a variable
12 is_spiderman = "yes"
13 print("Type of is_spiderman:", type(is_spiderman))
```

```
Type of name: <class 'str'>
Type of age: <class 'int'>
Type of height: <class 'float'>
Type of is_spiderman: <class 'bool'>
Type of is_spiderman: <class 'str'>
```

2.2 Basic Input/Output

How can a program get different types of input data from the user? Use examples from comic books.³³

A program can get different types of input data from the user using different built-in functions depending on the type of input data. Here are some examples using characters from comic books:

1. **Text input:** The `input()` function can be used to get text input from the user. For example, in the Spider-Man comic, the program can ask the user to enter their name:

```
1 name = input("What's your name? ")
2 print("Hello, " + name + "!")
```

What's your name? Peter
Hello, Peter!

2. **Integer input:** The `int()` function can be used to get integer input from the user. For example, in the X-Men comic, the program can ask the user to enter their age:

```
1 age = int(input("What's your age? "))
2 if age >= 18:
3     print("You're old enough to join the X-Men!")
4 else:
5     print("Sorry, you're not old enough to join the X-Men yet.")
```

What's your age? 20
You're old enough to join the X-Men!

3. **Floating-point input:** The `float()` function can be used to get floating-point input from the user. For example, in the Iron Man comic, the program can ask the user to enter the weight of their suit:

```
1 weight = float(input("What's the weight of your Iron Man suit (in pounds)? "))
2 print("Your suit weighs " + str(weight) + " pounds.")
```

What's the weight of your Iron Man suit (in pounds)? 500.5
Your suit weighs 500.5 pounds.

4. **Boolean input:** The `bool()` function can be used to get boolean input from the user. For example, in the Avengers comic, the program can ask the user if they want to join the team:

```

1 answer = bool(input("Do you want to join the Avengers? (True/False) "))
2 if answer:
3     print("Welcome to the team!")
4 else:
5     print("Maybe next time.")

```

Do you want to join the Avengers? (True/False) True
Welcome to the team!

*What are those `if/else` commands that you used in some of the previous examples? I can imagine what they mean, but can you explain shortly and clearly?*³⁴

Sure! An `if/else` statement allows the program to make decisions based on whether a certain condition is `True` or `False`. The basic structure of an `if/else` statement is as follows:

```

1 if condition:
2     # Code to be executed if the condition is true
3 else:
4     # Code to be executed if the condition is false

```

Notice that the two blocks of code are mutually exclusive: if the condition is `True`, then only the code block under the `if` statement is executed, otherwise, if the condition is `False`, then the block under the `else` statement is the only code that is executed.

For example, let's say we want to write a program that checks the length of a string and prints out a message depending on whether the string is longer or shorter than a certain number of characters. We could use an `if/else` statement to do this:

```

1 # Define a string
2 hero = "Mister Fantastic"
3
4 # Check the length of the string
5 if len(hero) > 10:
6     print("The hero's name is really long!")
7 else:
8     print("The hero's name is pretty short.")

```

In this example, the `len()` function is used to get the number of characters (length) of the string, which in this case is 15, thus the program prints out the following message:

The hero's name is really long!

More on conditionals will be covered in Chapter 4.

2.3 A coding buddy

ChatGPT is not only capable of suggesting coding exercises but can also solve them all by itself! This amazing feature opens up a whole new world of exciting learning possibilities for you! You can challenge yourself by asking ChatGPT to propose a coding problem. Once you've solved it using your own knowledge and skills, you can compare your solution with ChatGPT's. This is where the real fun begins! You can get feedback from ChatGPT on the differences between your solution and its own. You can inquire about which solution is more efficient, or learn more about new techniques and tricks that you didn't know before but ChatGPT's got covered! Just imagine how much fun you'll have with your new coding buddy, ChatGPT!

Let's see an example of how this works.

Hi there, ChatGPT! I'd like to practice my coding skills with an exercise that involves user input, data type conversion, and printing output, maybe even including a menu of options. Could you please suggest an exercise for me to work on? Once I have the statement, I'd also appreciate it if you could show me an example of the expected output. Thanks!³⁵

Hi, budding programmer! Let's practice your user input, data type conversion, and printing skills with an entertaining exercise. Write a program that asks the user for at least 5 pieces of personal information, such as name, age, and hobbies, etc. Then, offer the user a menu of two options: "1-Short Bio" and "2-Long Bio". Depending on the user's choice, your program should print either a short or long biography. Get creative with your output and have fun! Here's an example of the output:

Welcome to the Personal Bio Program!

```
Please enter your name: John
Please enter your surname: Smith
Please enter your age: 25
Please enter your favorite hobby: books
Please enter your favorite food: pizza
Please enter your favorite color: red
```

Menu of options:

1. Short Bio
2. Long Bio

Enter your choice (1 or 2): 1

--- Short Bio ---

John, 25, is a programming student whose favourite hobby is reading books.

Ok, now show me the solution.³⁶

Sure! Here's one possible solution to the Personal Bio Program:

```

1 # Personal Bio Program
2 # Ask user for personal data
3 name = input("What's your name? ")
4 age = int(input("How old are you? "))
5 city = input("What city do you live in? ")
6 hobby = input("What's your favorite hobby? ")
7 pet_name = input("What's your pet's name? ")
8
9 # Print menu of options
10 print("Please choose an option:")
11 print("1. Short bio")
12 print("2. Long bio")
13
14 # Ask user for choice
15 choice = input("Enter your choice (1 or 2): ")
16
17 # Print bio based on choice
18 if choice == "1":
19     print(f"Hi, my name is {name}. I'm {age} years old and I live in {city}. My favorite
20         hobby is {hobby} and I have a pet named {pet_name}.")
21 elif choice == "2":
22     print(f"My name is {name} and I'm {age} years old. I currently live in {city} and in
23         my free time, I enjoy {hobby}. I have a pet named {pet_name} who keeps me company.
24             Thanks for reading!")
25 else:
26     print("Invalid choice. Please enter 1 or 2.")

```

Cool. My version of the solution is shown below. What are your thoughts on it? In comparison to yours?³⁷

```

name = input("What is your name? ")
age = int(input("How old are you? "))
city = input("What city do you live in? ")
hobby = input("What is your favorite hobby? ")
food = input("What is your favorite food? ")
color = input("What is your favorite color? ")
credit_card = input("What is your credit card number? ")

```

```

print('+'+'-'*20+'+')
print(f"{'OPTIONS':^20}|")
print('+'+'-'*20+'+')
print(f"{'1. Short Bio':^20}|")
print(f"{'2. Long Bio':^20}|")
print('+'+'-'*20+'+')

choice = int(input("Enter choice number: "))

if choice == 1:
    print(f"\n----- Personal Bio for: {name} -----")
    print(f"{name} is {age} years old and lives in {city}.")
    print("-----")
else:
    print(f"\n----- Personal Bio for: {name} -----")
    print(f"{name} is {age} years old and lives in {city}.")
    print(f"In {name}'s free time, {name} enjoys {hobby} and eating {food}.")
    print(f"A fun fact about {name} is that their favourite color is {color}.")
    print(f"For your information, {name}'s credit card number")
    is:"+'*len(credit_card) )
    print("-----")

```

Your solution looks good and accomplishes the same task as the solution I provided. One difference I noticed is that you added an extra layer of formatting to the menu options by using '+' and '-' characters to create a border around the menu. This adds a nice touch of visual appeal to the program. Additionally, your program also uses f-strings to format the output, which is a concise and elegant way of displaying information. The only thing I would recommend is to avoid asking for sensitive information like credit card numbers in a program like this, especially since the program prints it out partially masked. It's important to prioritize user security and privacy in all applications.

Overall, great job!

But I took care to conceal the credit card number. I was just joking around.³⁸

Ah, I see! I apologize for the confusion. In that case, your code looks great and functions properly. It's always a good idea to be careful with sensitive information, even in code examples. Overall, your implementation is concise and readable, and effectively uses the string formatting options available in Python to make the output look neat and organized.

Good job!

By the way, I like how you commented each section of your code; it helps to clarify things. Please do the same with my code.³⁹

Sure, here's your code with comments:

```

1
2 # Ask for user input on personal data
3 name = input("What is your name? ")
4 age = int(input("How old are you? "))
5 city = input("What city do you live in? ")
6 hobby = input("What is your favorite hobby? ")
7 food = input("What is your favorite food? ")
8 color = input("What is your favorite color? ")
9 credit_card = input("What is your credit card number? ")

10
11
12 # Print the menu options
13 print('+'*20+ '-')
14 print(f"{'OPTIONS':^20}|")
15 print('+'*20+ '-')
16 print(f"{'1. Short Bio':^20}|")
17 print(f"{'2. Long Bio':^20}|")
18 print('+'*20+ '-')

19
20
21 # Ask user to choose an option
22 choice = int(input("Enter choice number: "))

23
24
25 # Print the chosen personal bio
26 if choice == 1:
27     print(f"\n----- Personal Bio for: {name} -----")
28     print(f"{name} is {age} years old and lives in {city}.")
29     print("-----")
30 else:
31     print(f"\n----- Personal Bio for: {name} -----")
32     print(f"{name} is {age} years old and lives in {city}.")
33     print(f"In {name}'s free time, {name} enjoys {hobby} and eating {food}.")
34     print(f"A fun fact about {name} is that their favourite color is {color}.")
35     print(f"For your information, {name}'s credit card number is:{'*'*len(credit_card)} ")
36     print("-----")

```

2.4 Hands-on time!

Provide four code exercises related to data type input and conversion that I need to correct for syntax or logic errors so that they can run properly.⁴⁰

Sure, here are four code exercises related to data type input and conversion, with syntax and/or logic errors that need to be corrected:

Exercise 1:

```
1 age = input("Please enter your age: ")
2 if age >= 18:
3     print("You are old enough to vote!")
4 else:
5     print("You are not old enough to vote yet.")
```

Exercise 2:

```
1 number = input("Enter a decimal number: ")
2 number = bool(number)
3 print(f"The integer part of the {number} is {number}")
```

Exercise 3:

```
1 password = input("Please enter your password: ")
2 if length(password) >= 8:
3     print("Your password is strong!")
4 else:
5     print("Your password is too short. Please choose a password that is at least 8
       characters long.")
```

Exercise 4:

```
1 name = input("Please enter your name: ")
2 age = input("Please enter your age: ")
3 print("Hello, name! You are age years old.")
```

Suggest some exercises for me to write code that can be solved using only user input and data type conversion operations.⁴¹

Exercise 1:

Write a program that asks the user to enter their birth year and calculates their age. An example of the printout message is shown below.

```
Please enter your name: Tony Stark
Please enter your birth year: 1970

Hi Tony Stark! As of 2023, you are 53 years old.
```

Exercise 2:

Write a program that asks the user to enter a temperature in Celsius and converts it to Fahrenheit. The conversion formula is $F = (C * 9/5) + 32$. An example of the printout message is shown below.

```
Please enter the temperature in Celsius: 30
30 C degrees is equal to 86 F degrees.
```

(NB: This exercise requires the use of arithmetic operators including addition, multiplication, and division. More on these and other math operators will be covered in Chapter 3).

Exercise 3:

Create a calculator program that prompts the user to enter two numbers and select an operation to perform on them. The program should display a menu of operation options (addition, subtraction, multiplication, and division) and ask the user to select the corresponding number for their desired operation. Once the user has entered their input, the program should output the result of the selected operation. An example of the printout message is shown below.

```
Welcome to the calculator app!
Please enter the first number: 10
Please enter the second number: 5
----- Menu of operations -----
1. Addition
2. Subtraction
3. Multiplication
4. Division
Enter operation number: 3
10 * 5 = 50
```

(NB: Again, more on these and other math operators will be covered in Chapter 3).

The Operations Room

List all of the Python operators and provide a brief description for each.⁴²

Here is a list of different types of Python operators and their definitions:

1. **Arithmetic operators:** used for performing mathematical operations, such as addition, subtraction, multiplication, division, etc.
2. **Comparison operators:** used for comparing values and returning a Boolean value (True or False), such as equal to, not equal to, greater than, less than, etc.
3. **Logical operators:** used for combining multiple conditions and returning a Boolean value, such as and, or, not.
4. **Bitwise operators:** used for performing operations on individual bits of binary numbers, such as AND, OR, XOR, etc.
5. **Assignment operators:** used for assigning values to variables, such as `=`, `+=`, `-=`, `*=`, `/=`, etc.
6. **Identity operators:** used for comparing the memory location of two objects, such as `is`, `is not`.
7. **Membership operators:** used for checking if a value is present in a sequence, such as `in`, `not in`.
8. **Exponentiation operator:** used for raising a number to a power, represented by `**`.

So, let's take a closer look at each of them.

3.1 Arithmetic operators

Explain the details of arithmetic operators in Python in a colourful tone and include a single code example to illustrate the use of all the operators. The example should compare entertaining statistics about Marvel vs X-men movies and box office.⁴³

In Python, arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, and division. These operators are used to manipulate numerical data types such as integers and floating-point numbers.

Here are the key arithmetic operators in Python:

- **Addition:** The plus sign (+) is used for addition. It adds two values together. For example, $2 + 2$ equals 4.
- **Subtraction:** The minus sign (-) is used for subtraction. It subtracts the second value from the first value. For example, $5 - 2$ equals 3.

- ▶ **Multiplication:** The asterisk (*) is used for multiplication. It multiplies two values together. For example, $3 * 4$ equals 12.
- ▶ **Division:** The forward slash (/) is used for division. It divides the first value by the second value. For example, $8 / 2$ equals 4.
- ▶ **Integer Division:** The double forward slash (//) is used for integer division. It divides the first value by the second value, but rounds down to the nearest whole number. For example, $8 // 3$ equals 2.
- ▶ **Modulo:** The percent sign (%) is used for modulo. It returns the remainder of the first value divided by the second value. For example, $7 \% 3$ equals 1.

Let's have some fun with these operators. Here's an example that compares some statistics about the Marvel vs X-Men movies and box office:

```

1 # Marvel and X-Men movie statistics
2 marvel_movies = 28
3 xmen_movies = 13
4 marvel_box_office = 22.55e9 # in US dollars
5 xmen_box_office = 6.01e9     # in US dollars
6 single_ticket = 15          # in US dollars
7
8
9 # Addition
10 total_movies = marvel_movies + xmen_movies
11 print("Total number of movies from both franchises:", total_movies)
12
13 # Subtraction
14 difference_movies = marvel_movies - xmen_movies
15 print("Marvel has", difference_movies, "more movies than X-Men")
16
17 # Multiplication
18 money_needed = single_ticket * total_movies
19 print("Total cost of watching all movies in the theatre: US$", money_needed)
20
21 # Division
22 total_box_office = marvel_box_office + xmen_box_office
23 average_box_office = total_box_office / total_movies
24 print(f"Average box office revenue per movie: US$ {average_box_office:.2f}")
25
26 # Integer Division
27 years_needed = total_movies // 12    # assuming watching 1 movie per month
28

```

```

29 # Modulo
30 plus_months_needed = total_movies % 12    # assuming watching 1 movie per month
31
32 print(f"Time required to watch all movies (one per month): {years_needed} years plus {
      plus_months_needed} months.")

```

The code is self-explanatory and produces the following output:

```

Total number of movies from both franchises: 41
Marvel has 15 more movies than X-Men
Total cost of watching all movies in the theatre: US$ 615
Average box office revenue per movie: US$ 696,585,365.85
Time required to watch all movies (one per month): 3 years and 5 months.

```

What do those numbers $22.55\text{e}9$ and $6.01\text{e}9$ for the `box_office` variables mean?⁴⁴

The numbers $22.55\text{e}9$ and $6.01\text{e}9$ are in scientific notation, which is a way to express very large or very small numbers using powers of 10. In this case, "e9" means "times 10 to the 9th power," or one billion. Therefore, $22.55\text{e}9$ means 22.55 billion, and $6.01\text{e}9$ means 6.01 billion. These numbers likely represent the box office revenue in US dollars for the Marvel and X-Men movies, respectively.

3.2 Comparison operators

Explain the details of comparison operators in Python in a colourful tone and include a single code example to illustrate the use of all of them, maybe using fun facts about football teams.⁴⁵

Comparison operators, also called relational operators, are used to compare values in Python. The result of the comparison is always a Boolean value: either `True` or `False`.

Here is a list of comparison operators in Python:

- ▶ **`== (equal to)`**: Checks if the values of two operands are equal.
- ▶ **`!= (not equal to)`**: Checks if the values of two operands are not equal.
- ▶ **`< (less than)`**: Checks if the value of the left operand is less than the value of the right operand.
- ▶ **`<= (less than or equal to)`**: Checks if the value of the left operand is less than or equal to the value of the right operand.
- ▶ **`> (greater than)`**: Checks if the value of the left operand is greater than the value of the right operand.
- ▶ **`>= (greater than or equal to)`**: Checks if the value of the left operand is greater than or equal to the value of the right operand.

Let's illustrate the use of all these operators with a fun example about football teams! Imagine we have two teams, Real Madrid and Barcelona, and we want to compare some of their statistics:

```
1 # Real Madrid statistics
2 real_points = 70
3 real_goals_scored = 76
4 real_goals_conceded = 39
5 real_goal_difference = real_goals_scored - real_goals_conceded
6
7 # Barcelona statistics
8 barca_points = 65
9 barca_goals_scored = 85
10 barca_goals_conceded = 38
11 barca_goal_difference = barca_goals_scored - barca_goals_conceded
12
13 # Comparison operators
14 print(f"Real Madrid ({real_points}) and Barcelona ({barca_points}) have equal points:",
15      real_points == barca_points)
16 print(f"Real Madrid ({real_goals_scored}) and Barcelona ({barca_goals_scored}) have
17      different goals scored:",
18      real_goals_scored != barca_goals_scored)
19 print(f"Real Madrid ({real_goals_conceded}) has conceded fewer goals than Barcelona ({barca_goals_conceded}):",
20      real_goals_conceded < barca_goals_conceded)
21 print(f"Real Madrid's goal difference ({real_goal_difference}) is less than or equal to
22      Barcelona's goal difference ({barca_goal_difference}):",
23      real_goal_difference <= barca_goal_difference)
24 print(f"Real Madrid ({real_points}) has more points than Barcelona ({barca_points}):",
25      real_points > barca_points)
26 print(f"Barcelona ({barca_goals_conceded}) has conceded more goals than Real Madrid ({real_goals_conceded}):",
27      barca_goals_conceded >= real_goals_conceded)
```

Real Madrid (70) and Barcelona (65) have equal points: False
Real Madrid (76) and Barcelona (85) have different goals scored: True
Real Madrid (39) has conceded fewer goals than Barcelona (38): False
Real Madrid's goal difference (37) is less than or equal to Barcelona's goal difference (47): True
Real Madrid (70) has more points than Barcelona (65): True
Barcelona (38) has conceded more goals than Real Madrid (39): False

As this example illustrates, these comparison operators are very useful in Python programming as they allow us to check the relationships between different values and make decisions based on those relationships.

3.3 Logical operators

Explain the concept of logical operators in Python in a casual tone. After defining each operator, include a single code example to illustrate the usage of all of them. Use a table of fast food dishes indicating how suitable are for vegetarians and vegans, for this purpose.⁴⁶

Logical operators in Python are tools that allow us to combine and manipulate boolean conditions representing true or false statements. These operators are mainly used with boolean variables, which can only be either `True` or `False`.

The most commonly used logical operators in Python are `and`, `or`, and `not`. The `and` operator returns `True` only if both operands are `True`, while the `or` operator returns `True` if at least one of the operands is `True`. The `not` operator is used to reverse the boolean value of its operand.

Let's consider the following table of fast food dishes indicating how suitable they are for vegetarians and vegans:

Dish	Vegetarian?	Vegan?
Cheeseburger	No	No
Margherita Pizza	Yes	No
Garden Salad	Yes	Yes
Ice Cream	Yes	No

We can use logical operators to answer questions like "Which dishes are suitable for both vegetarians and vegans?" or "Which dishes are not suitable for either vegetarians or vegans?". Here's an example code to demonstrate the usage of logical operators:

```

1 # Define the boolean values
2 # (cb: Cheeseburger, mp: Margherita Pizza, gs: Garden Salad, ic: Ice Cream)
3 cb_veg = False
4 cb_vegan = False
5 mp_veg = True
6 mp_vegan = False
7 gs_veg = True
8 gs_vegan = True
9 ic_veg = True
10 ic_vegan = False
11

```

```

12 # Print the results of logical operators
13 print("Is Cheeseburger both vegetarian and vegan? ", cb_veg and cb_vegan)
14 print("Is Margherita Pizza both vegetarian and vegan? ", mp_veg and mp_vegan)
15 print("Is Garden Salad both vegetarian and vegan? ", gs_veg and gs_vegan)
16 print("Is Ice Cream both vegetarian and vegan? ", ic_veg and ic_vegan)
17 print("Is Cheeseburger either vegetarian or vegan? ", cb_veg or cb_vegan)
18 print("Is Margherita Pizza either vegetarian or vegan? ", mp_veg or mp_vegan)
19 print("Is Garden Salad either vegetarian or vegan? ", gs_veg or gs_vegan)
20 print("Is Ice Cream either vegetarian or vegan? ", ic_veg or ic_vegan)
21 print("Is Cheeseburger not vegetarian? ", not cb_veg)
22 print("Is Cheeseburger not vegan? ", not cb_vegan)

```

The output is shown below:

```

Is Cheeseburger both vegetarian and vegan? False
Is Margherita Pizza both vegetarian and vegan? False
Is Garden Salad both vegetarian and vegan? True
Is Ice Cream both vegetarian and vegan? False
Is Cheeseburger either vegetarian or vegan? False
Is Margherita Pizza either vegetarian or vegan? True
Is Garden Salad either vegetarian or vegan? True
Is Ice Cream either vegetarian or vegan? True
Is Cheeseburger not vegetarian? True
Is Cheeseburger not vegan? True

```

3.4 Bitwise operators

Explain the concept of bitwise operators in Python in a friendly but professional tone. Include a code example after defining each operator to demonstrate how to use them all. Show the results of the operations, including bit sequences and integers of the operands.⁴⁷

Bitwise operators in Python are used to manipulate bits or binary digits of an integer. They are applied to each bit position of an integer independently, hence the name “bitwise”.

There are six bitwise operators in Python: bitwise AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>).

- The bitwise AND operator returns a new integer where each bit is set to 1 only if both bits from the original integers are 1. Otherwise, the bit is set to 0.

Here's an example:

```

1 a = 12 # 1100 in binary
2 b = 25 # 11001 in binary
3 c = a & b # 1000 in binary
4 print(f" {a:08b} (decimal: {a})\n & {b:08b} (decimal: {b})\n ----- \n {c:08b}\n {c} (decimal: {c})")
5

```

Notice I used the `:08b` format modifier to print the integers as 8-bit binary strings for more clarity. The output would be:

```

00001100 (decimal: 12)
& 00011001 (decimal: 25)
-----
00001000 (decimal: 8)

```

- The bitwise OR operator returns a new integer where each bit is set to 1 if at least one bit from the original integers is 1. Otherwise, the bit is set to 0. Example:

```

1 a = 12 # 1100 in binary
2 b = 25 # 11001 in binary
3 c = a | b # 11001 in binary
4 print(f" {a:08b} (decimal: {a})\n | {b:08b} (decimal: {b})\n ----- \n {c:08b}\n {c} (decimal: {c})")
5

```

The output would be:

```

00001100 (decimal: 12)
| 00011001 (decimal: 25)
-----
00011101 (decimal: 29)

```

- The bitwise XOR operator returns a new integer where each bit is set to 1 only if one of the bits from the original integers is 1. Otherwise, the bit is set to 0. Example:

```

1 a = 12 # 1100 in binary
2 b = 25 # 11001 in binary
3 c = a ^ b # 10001 in binary
4 print(f" {a:08b} (decimal: {a})\n ^ {b:08b} (decimal: {b})\n ----- \n {c:08b}\n {c} (decimal: {c})")
5

```

The output would be:

```

00001100 (decimal: 12)
^ 00011001 (decimal: 25)
-----
00010101 (decimal: 21)

```

- The bitwise NOT operator returns the complement of an integer, that is, it changes each bit from 0 to 1 and vice versa. Note that it returns the result in a two's complement form, which is a signed representation of integers. Example:

```

1 a = 12 # 1100 in binary
2 c = ~a # -13 in decimal (complement of 1100 in binary)
3 print(f" ~ {a:08b} (decimal: {a})\n-----\n {c:08b} (decimal: {c})")
4

```

The output would be:

```

~ 00001100 (decimal: 12)
-----
-0001101 (decimal: -13)

```

- The left shift operator moves all the bits of an integer to the left by a given number of positions, and fills the rightmost bits with zeros. This is equivalent to multiplying the integer by 2 raised to the power of the number of positions. Example:

```

1 a = 12 # 1100 in binary
2 c = a << 3 # 110000 in binary, which is 96 in decimal
3 print(f" {a:08b} (decimal: {a})\n<< 3 \n-----\n {c:08b} (decimal: {c})")
4

```

The output would be:

```

00001100 (decimal: 12)
<< 3
-----
01100000 (decimal: 96)

```

- The right shift operator moves all the bits of an integer to the right by a given number of positions, and fills the leftmost bits with zeros (for non-negative integers) or ones (for negative integers). This is equivalent to dividing the integer by 2 raised to the power of the number of positions. Example:

```

1 a = 12 # 1100 in binary
2 c = a >> 3 # 11 in binary, which is 1 in decimal
3 print(f" {a:08b} (decimal: {a})\n>> 3 \n-----\n {c:08b} (decimal: {c})")
4

```

The output would be:

```
00001100 (decimal: 12)
>> 3
-----
00000001 (decimal: 1)
```

Bitwise operators are mostly used in low-level programming, such as embedded systems or device drivers, but can also be used in cryptography, graphics, and data compression.

*Okay, but could you please explain briefly what you mean by "binary digits of an integer". Give me some amusing real-life analogies.*⁴⁸

In computer systems, all information is ultimately represented in terms of binary digits, or bits for short. These are the 0s and 1s that you might have seen in computer code or digital electronics. A binary digit is a *bit*, which is the smallest unit of data that a computer can store or manipulate. A bit can only have two possible values, either 0 or 1.

An amusing analogy to understand bits and bytes could be that of a bunch of light switches. Each switch can be either "on" or "off", just like a single bit can be either 0 or 1. By arranging multiple switches side by side, we can form larger groups of switches, and represent larger values.

For example, 4 switches can represent 16 different combinations, or values ranging from 0 to 15. Similarly, 8 switches can represent 256 different combinations or values from 0 to 255. For example, the number 6 can be represented in an 4-bit binary sequence as 0110, which means $0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$, or $0 + 4 + 2 + 0$. Likewise, the number 15 would be 1111, which means $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$, or $8 + 4 + 2 + 1$. So when we talk about the binary digits of an integer, we're referring to the representation of that integer in binary notation.

The bitwise operators work by manipulating these individual switches or bits, as well as groups of them. This allows us to perform a variety of useful operations on the underlying data, such as shifting it left or right, flipping individual bits, or combining multiple values in various ways.

*Okay, one more thing: please explain what you mean by "two's complement form representation" in a brief and friendly manner.*⁴⁹

In computer systems, we often need to represent negative numbers as well as positive ones. One way to represent negative numbers is to use the sign-magnitude representation, where the leftmost bit represents the sign of the number (0 for positive, 1 for negative) and the remaining bits represent the magnitude. However, this representation has some drawbacks, such as the need for special cases to handle zero and the fact that we have two representations of zero (positive and negative).

Two's complement representation is a more memory-efficient and widely used alternative. It represents negative numbers as the two's complement of their absolute value. This means that to obtain the two's complement of a number, we first take the complement (flip all the bits) of the binary representation of its absolute value and then add 1 to the result. See for example:

1. Find the binary representation of 5: 00000101.
2. Invert all the bits: 11111010.
3. Add 1: 11111011.

So the two's complement form of -5 in 8-bit binary is 11111011.

Two's complement negative numbers are useful in arithmetic operations because they allow the computer to perform addition and subtraction using the same circuitry and algorithms as for unsigned integers. In fact, adding a negative two's complement number is equivalent to subtracting its absolute value from the unsigned integer. This simplifies the implementation of arithmetic operations and reduces the amount of hardware required.

In consequence, two's complement representation provides a simple and efficient way to represent negative numbers in computer systems, which is essential for many applications such as digital signal processing, artificial intelligence, and cryptography.

3.5 Assignment operators

Explain the concept of arithmetic assignment operators in Python in a playful manner.⁵⁰

Arithmetic assignment operators in Python are just a fun way of combining math and shortcuts. You know how sometimes you want to update a value by adding or subtracting another value? Well, arithmetic assignment operators are there to make your coding easier!

Let me give you some real-life examples. Imagine you have a jar filled with candy. You want to keep track of how much candy you eat every day. You start with 10 candies, and on the first day, you eat 3 candies. Instead of updating the value manually, you can use the arithmetic assignment operator "`-=`" to subtract 3 to the current value of the jar. So, instead of writing `jar = 7`, you can write `jar -= 3`.

Similarly, suppose you have a piggy bank full of coins. Assume this piggy bank is a modern magic device that allows you to earn 10% interest on the money you save each month. So you start with \$10 and end up with \$11 at the end of the month. Instead of manually assigning the piggy bank's final value, use the arithmetic assignment operator "`*=`" to add 1 to the current balance. Rather than writing `piggy_bank = 11`, you can write `piggy_bank *= 1.1`:

```

1 piggy_bank = 10
2 print("Money now in the piggy_bank:\t$ ", piggy_bank)
3 piggy_bank *= 1.1
4 print("Money one month later:\t\t$ ", piggy_bank)

```

Money now in the piggy_bank: \$ 10
 Money one month later: \$ 11.0

Here are some other examples of arithmetic assignment operators in Python:

```

1 x = 10
2 x += 5 # equivalent to x = x + 5
3 print(x) # Output: 15
4
5 y = 20
6 y -= 3 # equivalent to y = y - 3
7 print(y) # Output: 17
8
9 z = 7
10 z *= 2 # equivalent to z = z * 2
11 print(z) # Output: 14
12
13 a = 16
14 a /= 4 # equivalent to a = a / 4
15 print(a) # Output: 4.0
16
17 b = 5
18 b %= 2 # equivalent to b = b % 2
19 print(b) # Output: 1

```

As you can see, arithmetic assignment operators are a fun and efficient way to update variable values in Python.

3.6 Identity operators

Explain the concept of identity operators in Python in a playful manner.⁵¹

Basically, identity operators are used to compare the memory locations of two variables in Python. They return a boolean value of **True** or **False**, indicating whether the variables are the same or different in terms of their memory location.

To understand it better, imagine you have two identical-looking cups. At first glance, they might seem exactly the same, but if you look closer, you'll notice that they have different fingerprints on them. In Python, identity operators would be like checking if two objects have the same fingerprint or not.

Here are the two identity operators in Python:

- ▶ **`is`**: returns `True` if two variables refer to the same object in memory, and `False` otherwise.
- ▶ **`is not`**: returns `True` if two variables do not refer to the same object in memory, and `False` otherwise.

Let's see an example. In this code, we define variables with the names of characters from the *Cars* movie. We then use the identity operators to compare them:

```

1 # Define the cars
2 mcqueen = "Lightning McQueen"
3 doc = "Doc Hudson"
4 mater = "Tow Mater"
5
6 # Use identity operators to compare the cars
7 print(f"{mcqueen} is {doc}: ", mcqueen is doc)
8 print(f"{mcqueen} is not {mater}: ", mcqueen is not mater)
9
10 # Assign one variable to the other
11 mcqueen = doc
12 print(f"After assigning, {mcqueen} is {doc}: ", mcqueen is doc)
13
14 # Check if a car is None
15 sally = None
16 print(f"{sally} is None: ", sally is None)
17
18 sally = "Sally Carrera"
19 print(f"After assigning, {sally} is None: ", sally is None)
```

```

Lightning McQueen is Doc Hudson: False
Lightning McQueen is not Tow Mater: True
After assigning, Doc Hudson is Doc Hudson: True
None is None: True
After assigning, Sally Carrera is None: False
```

In the final block, we use the `is` operator to check if the object referred to by the `sally` variable is the special Python object `None`. Initially, the result will be `True`, since the variable has not been given any name, but after assigning a name, the result will be `False`.

So, identity operators in Python allow you to compare two objects and find out if they're the same or not, without actually looking at their values.

Ok. Two questions. First, what exactly do you mean by "memory locations"? I don't see any memory locations in the code; please elaborate in a friendly and colourful manner. Second, what is the Python special variable "None"? Please clarify.⁵²

Sure, memory spaces in programming are like storage units where data is stored in a computer's memory. Think of it like a big grid of post-office boxes, where each box is a memory space located at a given row and column in the grid, and each box can store a piece of information. So, when we create a variable, we're basically reserving one of these memory spaces to store some data, like a number, string, or even another object. When we assign a value to that variable, we're storing that value in the memory space assigned to that variable.

As for your second question, in Python, `None` is a special value that represents the absence of a value. It's used to indicate that a variable has no value or that a function returns nothing. Think of it like an empty box in your storage unit - it's still a box, but there's nothing inside it. When you assign a variable to `'None'`, you're essentially reserving a memory location for that variable, but not assigning any actual value to it.

3.7 Memberships operators

Explain the concept of memberships operators in Python in a playful manner.⁵³

Membership operators in Python are like secret detectives that help us investigate if a particular value belongs to a specific group or not.

The first membership operator is `in`, which checks whether a given value is present in a larger string or not. It's like asking the bouncer of a dance club if your name is on the guest list or not. For example:

```

1 name = "Alice"
2 another_name = "Bob"
3 letter = 'A'
4 print(f"{name} starts with {letter}: ", letter in name)
5 print(f"{another_name} starts with {letter}: ", letter in another_name)

```

Alice starts with A: True
Bob starts with A: False

In this example, the code checks if the letter "A" is the first letter of any of the names (assuming the first letter of a name is uppercase).

The second membership operator is `not in`, which checks whether a given value is not present in a larger string or not. The value can be a letter or a sequence of letters. For example:

```

1 sentence1 = "Java is fun!"
2 sentence2 = "Java is fun, but Python is really fantastic!"
3 language = "Python"
4
5 print(f"'{language}' is not in '{sentence1}': ", language not in sentence1)
6 print(f"'{language}' is not in '{sentence2}': ", language not in sentence2)

```

```
'Python' is not in 'Java is fun!': True
'Python' is not in 'Java is fun, but Python is really fantastic!': False
```

These membership operators are especially useful when working with strings and other data types (such as lists, sets or dictionaries, which will be covered in later chapters).

3.8 Exponentiation operator

Explain the concept of exponentiation operator in Python in a playful manner.⁵⁴

Sure, let's talk about the exponentiation operator! It's like having a magic wand that can make numbers grow super fast. Imagine you're a wizard and you have a wand that can multiply any number by itself as many times as you want. For example, if you wave your wand and say "2 to the power of 3", you get $2 \times 2 \times 2 = 8$. If you wave it again and say "2 to the power of 4", you get $2 \times 2 \times 2 \times 2 = 16$. The exponentiation operator in Python works the same way - it takes a number and raises it to a power (also known as an exponent). So if you want to raise 2 to the power of 3 in Python, you can simply write `2**3`, which will give you the result 8. You can use any number as a base and any integer as an exponent, and Python will calculate the result for you. It's like having a magical power to make numbers grow!

Here's an example of using the exponentiation operator:

```

1 # Calculating the area of a square with side length 5
2 area = 5 ** 2
3 print(area) # Output: 25

```

In this example, we're using the exponentiation operator (`**`) to raise the side length of a square to the power of 2, effectively calculating its area. The result is then stored in the variable `area`, which we print to the console.

*What happens when an expression has multiple operands? In what order are they evaluated?*⁵⁵

When an expression has multiple operands, they are evaluated based on the operator precedence and associativity rules. Operators with higher precedence are evaluated first, followed by those with lower precedence.

If there are operators with the same precedence, their evaluation order depends on their associativity. For example, the arithmetic operators (+, -, *, /) have left-to-right associativity, which means that operands are evaluated from left to right.

Here's an example to illustrate this: `x = 2 + 3 * 4 / 2`. In this expression, the multiplication and division operators have higher precedence than the addition operator. So, `3 * 4` is evaluated first, resulting in 12. Then, `12 / 2` is evaluated, resulting in 6. Finally, `2 + 6` is evaluated, resulting in 8. Therefore, the value of `x` would be 8.

It's important to be aware of the hierarchy of operator precedence and associativity when writing complex expressions in Python to ensure that the code behaves as intended. This hierarchy determines the order in which operators are evaluated in an expression with multiple operators.

To help us remember the order of precedence, we can use the acronym PEMDAS, which stands for Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction. This means that operators inside parentheses are evaluated first, followed by exponents, then multiplication and division (which have equal precedence and are evaluated left to right), and finally addition and subtraction (which also have equal precedence and are evaluated left to right).

However, it's important to note that we can change the order of evaluation by using parentheses to group certain operations together. For example, in the expression `(3 + 4) * 5`, the addition inside the parentheses is evaluated first, resulting in the value of 7, which is then multiplied by 5 to give the final result of 35. In contrast, the expression `3 + 4 * 5` yields 23.

3.9 Advanced math operators

*How can I program more complex math operations such as square roots, logarithms, and so on?*⁵⁶

Python provides a built-in module called `math` that offers many functions for performing complex mathematical operations. Here are a few examples:

```

1
2 # Import the built-in math module for various mathematical operations
3 import math
4
```

```

5 # Calculate the square root of a number
6 num = 25
7 sqrt_num = math.sqrt(num) # Use the sqrt() function to calculate the square root
8 print(f"The square root of {num} is {sqrt_num}")
9
10 # Calculate the logarithm of a number with base 10
11 num2 = 1000
12 log_num2 = math.log10(num2) # Use the log10() function to calculate the base 10
    logarithm
13 print(f"The base 10 logarithm of {num2} is {log_num2}")
14
15 # Get the value of pi, a mathematical constant
16 pi = math.pi
17 print(f"The value of pi is {pi}")
18
19 # Convert degrees to radians
20 deg = 45
21 rad = math.radians(deg) # Use the radians() function to convert degrees to radians
22 print(f"{deg} degrees is equal to {rad} radians")
23
24 # Calculate the sine of an angle in radians
25 angle = math.pi / 4
26 sin_angle = math.sin(angle) # Use the sin() function to calculate the sine of the angle
27 print(f"The sine of {angle} radians is {sin_angle}")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/3r4vb9y5>

```

The square root of 25 is 5.0
The base 10 logarithm of 1000 is 3.0
The value of pi is 3.141592653589793
45 degrees is equal to 0.7853981633974483 radians
The sine of 0.7853981633974483 radians is 0.7071067811865475

```

In this example, we imported the `math` module and used its built-in functions to perform complex mathematical operations. We calculated the square root of a number using the `sqrt()` function, the base 10 logarithm of a number using the `log10()` function, and the sine of an angle using the `sin()` function. We also used the built-in constant `pi` to calculate the value of pi and converted degrees to radians using the `radians()` function.

By using the `math` module, you can easily perform complex mathematical operations in your Python programs.

3.10 Hands-on time!

Suggest some fun and informative coding exercises for practising arithmetic, comparison, logical, bitwise, membership, exponentiation, and math operators.⁵⁷

Here are some coding exercises for practising various operators in Python:

1. Arithmetic operators:

- a) Write a program that calculates the area of a circle with a given radius, using the * operator for multiplication and the ** operator for exponentiation.
- b) Write a program that converts a given number of minutes into hours and minutes using the / and % operators.
- c) Write a program that converts a time in military format to civilian format.
- d) Write a program that computes the final price, including tax, given an item's price and a VAT percentage.

2. Comparison operators:

- a) Write a program that checks if a given number is positive, negative, or zero using the >, <, and == operators.
- b) Write a program that checks if a given number is even or odd using the % operator for modulus and the == operator.
- c) Write a program that checks if a given year is a leap year (i.e., divisible by 4 but not divisible by 100, except for years divisible by 400) using the % and AND operators.

3. Logical operators:

- a) Write a program that, given a temperature reading, checks if it falls within the normal human body temperature range.
- b) Write a program that checks if a given sentence contains both the words "dog" and "cat" using the AND operator and the IN operator.

4. Bitwise operators:

- a) Write a program that counts the number of bits set to 1 in the binary representation of a given integer number in the range [0, 15] (i.e., maximum four bits) using the >>, &, and += operators.
- b) Write a program that flips the bits of the binary representation of a given integer number in the range [0, 15] (to keep the binary representation within 4 bits), using the bitwise XOR operator.
- c) Write a program that, given an integer number, prints out double the amount without using arithmetic operators.
- d) Write a program that, given an even integer number, prints out half the amount without using arithmetic operators.

5. Membership operators:

- a) Write a program that checks if a given name starts with a consonant (not with a vowel) using the NOT, OR and IN operators. Assume a name starts with a capital letter.
- b) Write a program that checks whether an email entered by the user meets the minimum valid criteria, including the presence of the "at" symbol (@) and the "dot com" suffix.

6. Exponentiation operator:

- a) Write a program that calculates the cube of a given number using the ** operator.
- b) Write a program that calculates the square root of a given number using the ** operator and a fraction exponent.

7. Math operators:

- a) Write a program that finds the hypotenuse of a right-angled triangle, given the lengths of the other two sides of the triangle.
- b) Write a program that calculates the distance between two points (x_1 , y_1) and (x_2 , y_2) using the - operator for subtraction, the ** operator for exponentiation and the `sqrt()` math function.
- c) Write a program that converts degrees to radians and vice versa, using the `radians()` and `degrees()` math functions.
- d) Write a program that calculates the height of an object using the length of its shadow and the angle of elevation of the sun, using the `tan()` trigonometric function.

Propose a nice and engaging coding problem to practice binary representation of integer numbers. Assume you have a table of food dishes, with columns representing the ingredients used to prepare them and a final column displaying the dish's integer code as a result of interpreting the ingredient columns as bits of a binary number (show a table with four dishes and five ingredients). The program should ask for the name of a dish and a series of yes/no questions to determine the dish's integer code, and then print out a row of the table with this information (show an example of the program's output).⁵⁸

Problem: Dish Code Decoder

You have a table of food dishes with columns representing the ingredients used to prepare them. Each dish is uniquely identified by an integer code based on its ingredients. You are tasked with building a program that asks the user for the name of a dish and a series of yes/no questions to determine the integer code of the dish based on the ingredients.

Here's an example table with four dishes and five ingredients:

Dish	Cheese	Lettuce	Tomato	Onion	Meat	Code
Cheeseburger	Yes	Yes	Yes	No	Yes	29
Margherita Pizza	Yes	No	Yes	No	No	20
Caesar Salad	Yes	Yes	No	Yes	Yes	27
Ice Cream	No	No	No	No	No	0

Your program should prompt the user to enter the name of a dish and then ask a series of questions to determine the presence of each ingredient. Based on the user's responses, the program should compute and display the corresponding row of the table, including the computed code.

Example Output:

```
Welcome to the Dish Code Decoder!
```

```
Let's find out the dish code based on the ingredients.
```

```
Please enter the name of the dish: Pizza
```

```
Does the dish contain Cheese? (Y/N): Y
```

```
Does the dish contain Lettuce? (Y/N): N
```

```
Does the dish contain Tomato? (Y/N): Y
```

```
Does the dish contain Onion? (Y/N): Y
```

```
Does the dish contain Meat? (Y/N): Y
```

```
The code for the dish is: 11
```

```
Here is the dish you selected:
```

Dish	Cheese	Lettuce	Tomato	Onion	Meat	Code
Cheeseburger	Yes	Yes	Yes	No	Yes	29

```
Thank you for using the Dish Code Decoder! Bon appetit!
```

You can try adding new dishes to the table, such as pasta or omelette, and see how the program accurately determines their integer codes based on the ingredients. It's a great opportunity to strengthen your skills in handling user input and working with binary representations.

Enjoy building your Dish Code Decoder!

(NB: You may need to use the *if/else* statement, which was mentioned in Chapter 2, to process the user input and compute the integer code. More on *if/else* statements will be covered in Chapter 4.)

The Control Tower **4**

List all of the control structures in programming and provide a brief description for each.⁵⁹

In programming, there are several control structures that allow you to control the flow of execution in your code. These control structures can be categorized into three main types: branching structures, repetition structures, and nested structures.

1. Branching Structures (Conditionals):

- ▶ **If-Else Statement:** Executes a block of code if a certain condition is true. It can also include an optional `else` statement to provide an alternative block of code to execute if the condition is false.
- ▶ **Match-case Statement:** Evaluates an expression and matches it against multiple cases, executing the code block associated with the matched case.

2. Repetition Structures (Loops):

- ▶ **For Loop:** Executes a block of code for a specific number of iterations. It is often used when the number of iterations is known in advance.
- ▶ **While Loop:** Executes a block of code repeatedly as long as a specified condition is true. It is useful when the number of iterations is not known beforehand.
- ▶ **Do-While Loop:** Similar to the `while` loop, but it guarantees that the code block is executed at least once before checking the condition.

3. Nested Structures:

- ▶ **Nested Conditionals:** Allows you to have an `if-else` statement within another `if-else` statement. It provides more complex branching based on multiple conditions.
- ▶ **Nested Loops:** Involves placing one loop inside another loop. This allows for more intricate control over repetitive tasks.

These control structures give you the ability to make decisions, repeat actions, and conceive more intricate program logic. By combining these structures effectively, you can create powerful and flexible programs to meet various requirements.

So, let's take a closer look at each of them.

4.1 If/Else statements

Explain everything I need to know about `if/else` statements in Python in a fun and creative way, with playful examples.⁶⁰

Sure! Let's dive into the exciting world of conditional statements in Python.

Imagine you're a detective solving cases in a crime-ridden city. Your job is to make decisions based on the clues and evidence you gather. That's exactly what if/else statements are all about - making decisions in your code!

Here's a playful example: Let's say you're a detective investigating a mysterious crime. You have a suspect, and you need to determine if they're guilty or innocent. You can use an if/else statement to help you. The general structure of this conditional statement is:

```

1 if condition:
2     # Code to execute if the condition is true
3 else:
4     # Code to execute if the condition is false

```

So, let's code the proposed example:

```

1 # Ask the user to enter their name
2 suspect = input("Enter your name: ")
3
4 # Check if the entered name matches "Jack the Ripper"
5 if suspect == "Jack the Ripper":
6     # If the name matches, print an arrest message
7     print(f"You're under arrest, {suspect}! You're guilty!")
8 else:
9     # If the name doesn't match, print a message indicating innocence
10    print(f"You're free to go {suspect}! You're innocent!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/ykdth9bd>

Here you have two different runs of this program:

```

Enter your name: Sherlock Holmes
You're free to go Sherlock Holmes! You're innocent!

```

```

Enter your name: Jack the Ripper
You're under arrest, Jack the Ripper! You're guilty!

```

Moreover, you can also use if/elif/else statements to handle more than two different scenarios. Such scenarios can be defined by combining several conditions using logical operators. Let's say you want to determine if it's a good day to go to the beach based on the weather. Here's an example:

```

1 # Set the initial conditions
2 sunny = True
3 temperature = 22
4

```

```

5 # If it's sunny and the temperature is above 25 degrees
6 if sunny and temperature > 25:
7     print("It's a perfect day for the beach!")
8 # If it's sunny but the temperature is 25 degrees or lower
9 elif sunny and temperature <= 25:
10    print("It's a bit chilly, but you can still enjoy the beach!")
11 # If it's not sunny
12 else:
13     print("Maybe it's better to stay home today.")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/362yh7w6>

In this code, we have two variables: `sunny` and `temperature`. Let's go through the code and understand how the conditionals are evaluated and the role of each variable:

1. The initial conditions are set with `sunny = True` and `temperature = 22`.
2. The first conditional statement is `if sunny and temperature > 25`. This condition checks if it's sunny (`sunny == True`) and if the temperature is above 25 degrees (`temperature > 25`). If both conditions are true, the code inside this block will execute, and it will print "It's a perfect day for the beach!".
3. If the first condition is not met, the program moves to the next conditional statement, `elif sunny and temperature <= 25`. This condition checks if it's sunny (`sunny == True`) and if the temperature is 25 degrees or lower (`temperature <= 25`). If both conditions are true, the code inside this block will execute, and it will print "It's a bit chilly, but you can still enjoy the beach!".
4. If neither the first nor the second condition is met, the program moves to the `else` block. This block will execute only when it's not sunny (`sunny == False`). In this case, it will print "Maybe it's better to stay home today.".

It's a bit chilly, but you can still enjoy the beach!

The order of the conditions is important because the algorithm evaluates them one by one from top to bottom. Once a condition is true, the corresponding block of code is executed, and the program exits the `if/elif/else` structure. This means that only one block of code will be executed, depending on which condition is true first. In this case, the `temperature` variable does not affect the execution of the `else` block because the algorithm flow only reaches this point when the previous `if` and `elif` conditions involving the `sunny` variable are not met, that is, when `sunny == False`. Therefore, the `else` block will execute when it's not sunny, regardless of the value of the `temperature` variable.

Remember, `if/else` statements can handle various conditions and allow your program to make different decisions based on those conditions. They're like the choices you make in a game that determine the outcome.

So, whether you're solving mysteries, planning beach trips, or creating exciting programs, `if/else` statements in Python are your trusty tools for making decisions in your algorithms.

*Is there anything else I should know about conditionals in Python?*⁶¹

Yes, there are a few more things to know about conditionals in Python:

- ▶ **Nested conditionals:** You can have conditionals within conditionals to handle more complex scenarios. This is known as nested conditionals. Inside an `if` or `elif` block, you can have another `if` statement to further refine the conditions with additional criteria.

(NB: We will cover more on nested structures at the end of the chapter.)

- ▶ **Chained conditionals:** In some cases, you may have multiple conditions that overlap, meaning the algorithm may execute several `if` blocks depending on the variable value. The syntax for the chained conditionals is as follows:

```

1 if condition1:
2     # code block
3 if condition2:
4     # code block
5 if condition3:
6     # code block

```

In this case, each `if` statement will be evaluated independently, and their respective code blocks will execute when their conditions are `True`.

Here's an example of chained conditionals where the conditions may overlap, resulting in multiple `if` blocks executing:

```

1 # Prompt the user for their age
2 age = int(input("How old are you? "))
3
4 # Check the age range using chained conditionals
5 if age > 13:
6     print("Congratulations, you've survived your childhood!")
7 if age > 18:
8     print("You've made it through the rollercoaster of teenage years!")
9 if age > 35:
10    print("You're now a fully-fledged adult, and bills are your new best friends!")
11 if age >= 65:
12    print("Welcome to the exclusive club of senior citizens! Discounts await!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2s42xwnc>

In this code the conditions are not exclusive, and each block will execute if the corresponding age condition is met. For example, if the user enters an age of 15, the program will print "Congratulations, you've survived your childhood!". If the user enters an age of 25, the program will print both "Congratulations, you've survived your childhood!" and "You've made it through the rollercoaster of teenage years!". Let's see what happens when the user enters an age of 40:

```
How old are you? 40
Congratulations, you've survived your childhood!
You've made it through the rollercoaster of teenage years!
You're now a fully-fledged adult, and bills are your new best friends!
```

As you can see, these printouts reflect the stages the user has passed based on their age, allowing for multiple conditions to be true simultaneously.

- ▶ **Short-circuit evaluation:** Python uses short-circuit evaluation in conditional statements. This means that if the outcome of a condition can be determined based on the first part of the condition, the second part will not be evaluated. For example, in an `and` condition, if the first operand is `False`, the second operand will not be evaluated because the overall result will always be `False`.
- ▶ **Ternary operator:** Python also provides a compact way to write simple conditional expressions called the ternary operator. It allows you to evaluate a condition and return one value if the condition is true, and another value if the condition is false. The syntax is as follows: `value = true_value if condition else false_value`. Example:

```
1 # Prompt the user for the speed of their vehicle
2 speed = int(input("Officer: What was the speed of your vehicle in mph? "))
3
4 # Use the ternary operator to determine the response in the officer's message
5 response = "speeding ticket, sorry" if speed > 70 else "high, five!"
6 print(f"Officer: Alright, I may have to issue you a...{response}. Drive safe!")
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/e6b9v554>

In this code, the print statement incorporates into the officer's response the result of the ternary operator, yielding a message with a touch of humour by issuing a "speeding ticket" if the speed is above 70 mph, and a playful "high five" if the speed is within the limit. The message also encourages safe driving:

```
Officer: What was the speed of your vehicle in mph? 45
Officer: Alright, I may have to issue you a... high, five!. Drive safe!
```

```
Officer: What was the speed of your vehicle in mph? 85
Officer: Alright, I may have to issue you a... speeding ticket, sorry. Drive
safe!
```

So, the aforementioned aspects are additional considerations to keep in mind when working with conditionals in Python. Having a clear understanding of these concepts will enable you to write code that is more flexible and efficient when dealing with scenarios that involve complex decision-making.

4.2 Match-case statements

Now let's talk about match/case statements in Python. Explain everything I need to know about them in a fun and creative way, using playful examples.⁶²

Starting from Python 3.10, the `match/case` statement is introduced as a new feature called "structural pattern matching". It provides a powerful and concise way to match, extract values and decide cases or branches based on patterns.

The basic syntax of the `match/case` statement in Python is as follows:

```

1 match expression:
2     case value1:
3         # Code to execute if expression matches value1
4     case value2:
5         # Code to execute if expression matches value2
6     ...
7     case valueN:
8         # Code to execute if expression matches valueN
9     case _:
10        # Code to execute if no cases match

```

Here's a breakdown of the syntax:

- ▶ **expression:** The expression whose value will be matched against the cases.
- ▶ **value1, value2, ..., valueN:** The possible values that the expression can match. These can be constants, variables, or even patterns.
- ▶ **_:** The wildcard case, which matches any value that has not been explicitly listed in the previous cases.

The code inside each case block is executed when the expression matches the corresponding value. If multiple cases match, only the code from the first matching case is executed.

The `match/case` statement provides a concise way to handle multiple cases based on the value of an expression, making the code more readable and eliminating the need for long chains of `if/elif/else` statements.

Remember that the `match/case` statement was introduced in Python 3.10, so make sure you have the appropriate version installed to use this feature.

Let's explore the `match/case` statement using a playful example of pizza ordering. Imagine you are a pizza lover, and you want to order a pizza with different toppings. So, let's see how we can use the `match/case` statement to handle this situation:

```

1 pizza = input("What pizza would you like today? (veggie/meat/hawaiian)")

2

3 match pizza.lower():
4     case "veggie":
5         toppings = "Mushrooms, olives, and peppers"
6     case "meat":
7         toppings = "Ground beef, sausage, and pepperoni"
8     case "hawaiian":
9         toppings = "Ham, pineapple, and extra cheese"
10    case _:
11        toppings = "Tomato and cheese only"
12
13 print(f"You ordered a {pizza} pizza with toppings: {toppings}")

```

In this example, we ask the user what type of pizza they want to order first. The corresponding toppings are then determined using the `match/case` statement. Each `case` in the `match` block checks if the pizza name matches one of the specified values. If a match is found, the corresponding ingredients are assigned to the `toppings` variable. If none of the cases match the pizza, the wildcard case `_` is triggered, indicating a basic pizza topping:

```

What pizza would you like today? (veggie/meat/hawaiian) Hawaiian
You ordered a Hawaiian pizza with toppings: Ham, pineapple, and extra cheese

```

```

What pizza would you like today? (veggie/meat/hawaiian) surprise me!
You ordered a surprise me! pizza with toppings: Tomato and cheese only

```

With the `match/case` statement, you can elegantly handle different cases based on the value being matched. It improves readability and reduces the need for intricate `if/elif/else` structures.

*Ok. In the match statement, I see you used a "lower()" command. Do I have to put it every time I code a match/case statement?*⁶³

No, the use of the `lower()` command in the `match` statement was specific to the pizza topping example. It was used to convert the user input to lowercase so that the matching is case-insensitive. This means that regardless of whether the user enters "veggie", "VEGGIE", or "VeGGie", it will be considered a match.

In general, you don't have to use `lower()` in every `match/case` statement. It is only necessary when you want to perform case-insensitive matching. If the case of the input values doesn't matter or if you want case-sensitive matching, you can omit the `lower()` command.

4.3 For loop statements

Please provide a brief technical definition of the for loop.⁶⁴

Certainly! A for loop is a programming construct that allows us to repeatedly execute a block of code for a specific number of times or for each element in a collection. It follows a general syntax:

```
1 for variable in iterable:  
2     # Code block to be executed
```

In this syntax:

- ▶ **variable**: Is a variable that takes on the value of each element in the **iterable** during each iteration (repetition) of the loop.
- ▶ **iterable**: Represents a sequence or collection of elements, such as a range, a list, or a string.
- ▶ **code block**: Is the code that follows, indented under the 'for' statement, consisting of a set of instructions to be executed repeatedly.

During each iteration of the loop, the variable takes on the value of the next element in the iterable, and the code block is executed. This process continues until all elements in the iterable have been processed.

The **for** loop is particularly useful when we know in advance the number of iterations that we want the code block to be repeated, or when we want to iterate over a collection of items. It simplifies repetitive tasks and allows us to perform operations efficiently without writing redundant code.

Now let's see the for loops in action in a fun and creative way, using playful examples.⁶⁵

Certainly! Let's explore the world of **for** loop statements in Python with some playful examples.

1. *Counting with Ease.* The **for** loop allows you to effortlessly count numbers within a specified range. For example, to count from 1 to 10, you can use the **range()** function:

```
1 # range(1, 11) generates a sequence from 1 to 11-1, i.e from 1 to 10  
2 for number in range(1, 11):  
3     print(number)
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/pcvbud98>

This code will iteratively print the numbers from 1 to 10, making counting a breeze.

```

1
2
3
4
5
6
7
8
9
10

```

2. *Stepping to Success.* You can also control the step size when iterating over a range. Suppose you want to count from 1 to 10, but only include even numbers. With the step parameter, you can achieve this:

```

1 # the last parameter in range() is the count step
2 for number in range(2, 11, 2):
3     print(number)

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/yccsvpds>

In this case, the loop starts from 2 and increments by 2 until it reaches 10, printing only the even numbers.

```

2
4
6
8
10

```

3. *Unveiling the Magic Word.* The `for` loop is not limited to numbers; it can also iterate over the characters of a string. Let's say we have a magical word, "abracadabra," and we want to reveal its letters one by one:

```

1 magic_word = "abracadabra"
2 print(f"Let's unveil the magical letters of '{magic_word}': ")
3 for letter in magic_word:
4     print("----+")
5     print(f"| {letter} |")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2p8vdpnm>

Running this code will display each letter of the magic word within magic boxes on separate lines, unveiling its hidden power.

```

Let's unveil the magical letters of 'abracadabra':
+---+
| a |
+---+
| b |
+---+
| r |
+---+
| a |
+---+
| c |
+---+
| a |
+---+
| d |
+---+
| a |
+---+
| b |
+---+
| r |
+---+
| a |

```

4. *Multiplication Spell.* The `for` loop can be used to generate multiplication tables effortlessly. Suppose you want to print the multiplication table for a given number, from 1 to 10:

```

1 number = int(input("Enter a number to cast a multiplication spell: "))
2 for i in range(1, 11):
3     result = number * i
4     print(f"{number} x {i} = {result}")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/3fmfxdap>

Running this code will display the multiplication table for the number entered by the user, showing each multiplication operation along with the corresponding result.

```

Enter a number to cast a multiplication spell: 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28

```

```
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

5. *Transforming Strings.* The `for` loop can also be used to manipulate strings, such as converting the case of each character. Let's say we have a string, "Hello, World!", and we want to convert it to uppercase:

```
1 text = "Hello, World!"
2 converted_text = ""
3 for char in text:
4     converted_text += char.upper()
5 print("Original text : ", text)
6 print("Converted text: ", converted_text)
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2p8kw9ee>

Running this code will print "HELLO, WORLD!", transforming all the characters in the string to uppercase.

```
Original text : Hello, World!
Converted text: HELLO, WORLD!
```

As you can see, the `for` loop is a powerful construct that simplifies iteration and allows you accomplish a wide range of tasks, from generating mathematical patterns to manipulating strings. Its versatility and ease of use make it a valuable tool in your programming journey.

4.4 While loop statements

Please provide a brief technical definition of the while loop.⁶⁶

The `while` loop is a control flow statement in programming that allows you to repeatedly execute a block of code as long as a specified condition is true. It provides a way to create loops that iterate dynamically based on the evaluation of a condition, rather than a predetermined number of iterations. The `while` loop consists of a condition that is checked before each iteration, and as long as the condition remains true, the code block within the loop is executed. The loop continues until the condition becomes `False`, at which point the program exits the loop and continues with the next set of instructions.

The syntax of the `while` loop is as follows:

```

1 while condition:
2     # Code block to be executed

```

In this syntax, the **condition** is a boolean expression that is evaluated before each iteration. As long as the **condition** is true, the **code block** would be executed; if it's false, the loop is exited. Inside the **code block**, you can perform any desired operations or calculations. After the **code block** is executed, the program returns to the beginning of the loop and re-evaluates the condition to determine whether to continue or exit the loop.

It's important to ensure that the **condition** eventually becomes false to avoid infinite loops, which are loops that continue executing indefinitely without stopping. If the **condition** never becomes false, the loop will keep repeating indefinitely, causing the program to hang or crash. To prevent this, it's crucial to design the **condition** in a way that allows for termination. This can be achieved by incorporating variables or statements within the loop that update or modify the **condition**, ensuring that it will eventually evaluate to false and exit the loop.

Now let's see the while loops in action in a fun and creative way, using playful examples.⁶⁷

Sure! Let's explore **while** loops through some playful examples:

1. *Counting Sheep.* Imagine you're trying to fall asleep, but you can't seem to doze off. Let's write a program that counts sheep until you finally fall asleep. Here's the code:

```

1 # Initialize the variable sheep to 0
2 sheep = 0
3
4 # Prompt the user for their sleep status
5 asleep = input("Are you asleep yet (yes/no)? ")
6
7 # Continue the loop as long as the user is not asleep
8 while asleep.lower() != "yes":
9     sheep += 1
10    print(f"Counting sheep #{sheep}: Baaa...\n")
11    asleep = input("Are you asleep yet (yes/no)? ")
12
13 # Once the loop is terminated, acknowledge that the user is asleep
14 print("You've fallen asleep! Sweet dreams!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/4mp2ty8n>

In this code, we simulate counting sheep to help the user fall asleep. The program prompts the user for their sleep status using the `input()` function. The **while** loop continues as long as the user's response is not "yes".

Inside the loop, we increment the `sheep` count by 1 and print a message to help the user visualising a sheep jumping over a fence. Then, we prompt the user again for their sleep status. Once the user responds with "yes", the condition of the `while` loop becomes `False`, and the program exits the loop. Finally, we print a message indicating that the user has fallen asleep.

It's important to note that in this example, we do not know in advance how many times the loop will repeat. The loop continues until the user responds with "yes", and the number of times the loop will iterate depends on the user's input.

```
Are you asleep yet (yes/no)? no
```

```
Counting sheep #1: Baaa...
```

```
Are you asleep yet (yes/no)? no
```

```
Counting sheep #2: Baaa...
```

```
Are you asleep yet (yes/no)? no
```

```
Counting sheep #3: Baaa...
```

```
Are you asleep yet (yes/no)? yes
```

```
You've fallen asleep! Sweet dreams!
```

- Unlocking the Treasure Chest.* Picture yourself as a treasure hunter trying to unlock a mysterious chest. Unfortunately the chest is locked, and you need to keep guessing until you find the secret code that opens it. Let's write a program that simulates this treasure hunt:

```

1 # Define a secret word and generate the secret code
2 secret_word = "OpenSesame"
3 secret_code = secret_word.swapcase()
4
5 # Prompt the user to enter their guess
6 guess = input("Enter the secret code: ")
7
8 # Continue the loop as long as the guess is incorrect
9 while guess != secret_code:
10     print("Wrong code! Try again.\n")
11     guess = input("Enter the secret code: ")
12
13 # Print a message indicating the chest open
14 print("Congratulations! You've unlocked the treasure chest!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/23dudzs>

In this code, we simulate a secret code that needs to be guessed correctly to unlock a treasure chest. The `secret_word` is defined as "OpenSesame". We generate the `secret_code` by swapping the case of the characters in `secret_word` using the `swapcase()` method.

The program prompts the user to enter their guess using the `input()` function. The `while` loop continues as long as the guess is not equal to the `secret_code`. Inside the loop, we print a message indicating that the `guess` is incorrect and prompt the user again to enter their guess. Once the user enters the correct `secret_code`, the condition of the loop becomes `False`, and the program exits the loop. Finally, we print a message congratulating the user for unlocking the treasure chest.

```
Enter the secret code: OpenSesame
Wrong code! Try again.

Enter the secret code: opensesame
Wrong code! Try again.

Enter the secret code: oPENSesAME
Congratulations! You've unlocked the treasure chest!
```

These playful examples demonstrate how while loops can be used to create interactive and engaging programs. By using creative scenarios, we can make programming more fun and enjoyable.

*In the "Counting Sheep" example, you ask the user if they are sleeping twice. Isn't that redundant? Is there a way to write the loop without asking the same question twice?*⁶⁸

You're correct, asking the user if they are sleeping twice in the "Counting Sheep" example is indeed redundant. We can simplify the loop and avoid repeating the same question twice by using a 'while True' loop with a conditional 'break' statement. Here's an updated version of the code that addresses this issue:

```
1 sheep = 0
2
3 while True:
4     asleep = input("Are you asleep yet (yes/no)? ")
5     if asleep == "yes":
6         break
7     sheep += 1
8     print(f"Counting sheep #{sheep}: Baaa...")
9
10 print("You've fallen asleep! Sweet dreams!")
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/3vrcnn2f>

In this revised code, the `while True` loop will continue indefinitely until the `break` statement is encountered. Inside the loop, we prompt the user for their sleep status using the `input()` function. If the user responds with "yes", we break out of the loop and proceed to the next statement after the loop. Otherwise, if the user responds with "no", we increment the `sheep` count by 1 and print a message to visualise a sheep jumping over a fence.

Using this approach, we eliminate the redundant question and achieve the desired behaviour of asking the user if they are sleeping only once in the entire code.

Is there anything else I should know about for or while loops in Python?⁶⁹

Here are a few additional points to keep in mind about for and while loops in Python:

► `for` loops:

- The `range()` function is commonly used in for loops to generate a sequence of numbers that can be iterated over.
- The `break` statement can be used to prematurely exit a for loop before completing all iterations.
- The `continue` statement can be used to skip the current iteration and proceed to the next iteration of the loop.
- You can loop over various data structures in Python, including strings, lists, tuples, and dictionaries.

(NB: More on these structures in the next chapter)

► `while` loops:

- The condition in a while loop is evaluated before each iteration, and the loop continues as long as the condition remains true.
- It's important to ensure that the condition eventually becomes false to avoid infinite loops, which can cause the program to run indefinitely.
- The `break` statement can be used to exit a while loop prematurely based on a certain condition.
- The `continue` statement can be used to skip the current iteration and continue with the next iteration of the loop.
- `while` loops are useful when you don't know in advance how many times the loop needs to iterate, as the number of iterations depends on the condition being evaluated.

Understanding how to use loops effectively can greatly enhance your ability to iterate over data, perform repetitive tasks, and solve complex problems in Python. It's always a good practice to test and debug your loops to ensure they behave as expected and handle different scenarios correctly.

4.5 Nested control structures

What is a nested control structure and why is it useful?⁷⁰

A nested control structure is a control structure (such as an `if` statement, `for` loop, or `while` loop) that is placed within another control structure. It allows for multiple levels of branching or looping within a program.

Nested control structures are powerful tools in programming that allow for the handling of complex logic and the execution of advanced tasks. They enable the creation of intricate decision-making processes and iterative operations, providing flexibility and adaptability to programs. With nested control structures, you can design code that responds to multiple conditions, performs repetitive tasks with varying complexity, and implements sophisticated algorithms. This dynamic approach makes programs more efficient and capable of solving complex problems, as they can adapt to changing conditions, make decisions based on multiple factors, and execute iterative operations at different levels of depth.

However, it's important to use nested control structures judiciously and maintain code readability. As the depth of nesting increases, code can become harder to understand and debug. Therefore, it's crucial to strike a balance between using nested control structures to tackle complex scenarios and keeping the code maintainable and comprehensible.

Let us now dive into nesting control structures in Python. Show me how to use nested conditionals allowing for multiple levels of branching, nested loops including nested double for loops, double while loops and mixed while/for or for/while loops, as well as combinations of loops within conditionals and conditional within loops. Use examples from superheroes and comics to be creative and playful in your explanation.⁷¹

Sure! Let's dive into the exciting world of superheroes to exploring the concept of nested control structures in Python.

1. *Superhero vs Supervillain.* Here's an example program that asks the user to choose a superhero and supervillain, and then determines who beats who using two-level nested conditionals.

In this example, the nested conditionals allow you to handle various combinations of superhero and super-villain choices. Each nested block checks the selected superhero and super-villain, and based on the combination, it prints a corresponding "who beats who" message. If the chosen combination does not match any of the nested conditionals, it displays an appropriate error message.

Let's take a look at the code:

```

1 # Get inputs
2 superhero = input("Choose a superhero: ")
3 supervillain = input("Choose a supervillain: ")
4
5 # Check superhero/supervillain combination
6 if superhero == "Superman":
7     if supervillain == "Lex Luthor":
8         print("Superman beats Lex Luthor!")
9     elif supervillain == "Doomsday":
10        print("Doomsday beats Superman!")
11    else:
12        print("Invalid supervillain choice!")
13 elif superhero == "Batman":
14     if supervillain == "Joker":
15         print("Batman beats Joker!")
16     elif supervillain == "Bane":
17         print("Bane beats Batman!")
18     else:
19         print("Invalid supervillain choice!")
20 elif superhero == "Spider-Man":
21     if supervillain == "Green Goblin":
22         print("Spider-Man beats Green Goblin!")
23     elif supervillain == "Venom":
24         print("Venom beats Spider-Man!")
25     else:
26         print("Invalid supervillain choice!")
27 else:
28     print("Invalid superhero choice!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/yesc8st4>

Here's some examples of outputs of this program:

```

Choose a superhero: Batman
Choose a supervillain: Joker
Batman beats Joker!

```

```

Choose a superhero: Spider-Man
Choose a supervillain: Venom
Venom beats Spider-Man!

```

2. *Superhero Training Sessions.* Now let's see nested **for** loops using a scenario of superhero's training sessions:

```

1 import time
2 num_sessions = int(input("Enter the number of training sessions: "))
3 session_duration = int(input("Enter the duration of each training session (in
4 seconds): "))
5
6 for session in range(1, num_sessions + 1):
7     print(f"\nTraining Session {session}:")
8     for second in range(1, session_duration + 1):
9         print(f"Seconds elapsed: {second}. Training in progress...")
10        time.sleep(1) # Pause for one second
11    print("Session complete. Rest 5 seconds and prepare for the next session...")
12    time.sleep(5) # Pause for five seconds
13
14
15 print("\nWell done! See you soon for more training!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/5yhkw5wd>

In this example, we have nested loops to simulate multiple training sessions with a specific duration. The outer loop iterates over the training sessions, while the inner loop iterates over the duration of each session. It prints the session number and displays the progress for each second. After completing the session, it prints a message indicating the session is complete, and then repeats the process for the next session.

Notice we import the time module, which provides the `sleep()` function that is used to pause the program execution for the specified number of seconds. We use it within the inner loop to create a one-second pause between each minute of training. Here's an example of the output of this program:

```

Enter the number of training sessions: 2
Enter the duration of each training session (in seconds): 2

Training Session 1:
Seconds elapsed: 1. Training in progress...
Seconds elapsed: 2. Training in progress...
Session complete. Rest 5 seconds and prepare for the next session...

Training Session 2:
Seconds elapsed: 1. Training in progress...
Seconds elapsed: 2. Training in progress...
Session complete. Rest 5 seconds and prepare for the next session...

Well done! See you soon for more training!

```

3. *Superhero Cheers*: You can use double `while` loops to repeat a set of actions for a sequence of superheroes. Let's explore an example where we say hoorays for different superheroes:

```

1 import random
2 # Outer loop runs until "exit" is entered
3 while True:
4     superhero = input("\nEnter the name of a superhero (or 'exit' to quit): ")
5     if superhero == "exit":
6         break;
7     # Generate a random number of hoorays using randint()
8     hoorays = random.randint(1, 10)
9     # Inner loop runs as long as there are hoorays left
10    while hoorays > 0:
11        print(f"Hip hip hooray, {superhero}!") # Print the cheer message
12        hoorays -= 1 # Decrement the hoorays count
13 print("\nBye!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/ycc6pryx>

In this code, the `random.randint()` function is used to generate a random number of hoorays for each superhero. The outer while loop allows the user to enter superhero names until "exit" is entered, while the inner while loop controls the number of hoorays to be printed for each superhero. By running this program, you can have some fun cheering for different superheroes with a random number of hoorays each time. Notice how we do not know the number of repetitions in advance in either of both loops. Here's an example of the output of this program:

```

Enter the name of a superhero (or 'exit' to quit): Wonder Woman
Hip hip hooray, Wonder Woman!

Enter the name of a superhero (or 'exit' to quit): Superman
Hip hip hooray, Superman!
Hip hip hooray, Superman!

Enter the name of a superhero (or 'exit' to quit): exit

Bye!

```

4. *Battle Royale.* Combining loop and conditional structures allows you to program more complex logic, such as a superhero battle royale. In the Battle Royale program, the user inputs two heroes chosen from Iron-man, Captain America, or Black Widow. The program assigns random power levels between 1 and 100 to each hero and simulates a battle where the heroes take turns hitting each other, decreasing the power level by up to 10 with each hit. The battle continues until one hero's power level reaches or falls below 0. The program allows the user to repeat the battle if they want to continue:

```
1 import random
2
3 heroes = "iron-man/captain america/black widow"
4 while True:
5     hero1 = input("\nEnter the name of the first hero (or 'exit' to quit): ")
6     if hero1 == "exit":
7         break
8
9     hero2 = input("Enter the name of the second hero (or 'exit' to quit): ")
10    if hero2 == "exit":
11        break
12
13    if hero1.lower() not in heroes or hero2.lower() not in heroes:
14        print('Invalid hero names. Only "Iron-man", "Captain America" or "Black
15        Widow" are allowed. Please try again.')
16        continue
17
18    power1 = random.randint(1, 100)
19    power2 = random.randint(1, 100)
20
21    print(f"\nLet the battle begin between {hero1} (power level: {power1}) and {hero2} (power level: {power2})!")
22
23    while power1 > 0 and power2 > 0:
24        hit1 = random.randint(1, 10)
25        hit2 = random.randint(1, 10)
26        power1 -= hit2
27        power2 -= hit1
28
29        print(f"\n{hero1} hits {hero2} with power {hit1}! {hero2}'s power level: {power2}")
30        print(f"{hero2} counters and hits {hero1} with power {hit2}! {hero1}'s
31        power level: {power1}")
```

```

30
31     if power1 <= 0 and power2 <= 0:
32         print("\nIt's a draw! Both heroes are knocked out.")
33     elif power1 <= 0:
34         print(f"\n{hero2} is the winner!")
35     elif power2 <= 0:
36         print(f"\n{hero1} is the winner!")
37
38 print("\nThank you for playing Superhero Battle Royale. See you next time!")

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2p8z9yap>

Observe how in this code, nested loops are used to control the progression of the battle and repetitions between battles. The outer loop allows the user to continue or exit the program, while the inner loop simulates the battle itself, with heroes taking turns and reducing each other's power levels. The nested conditionals are utilised to validate the user's input, determine the winner/loser of the battle, and control the termination of the program. They ensure that the program ends when the user inputs "exit" and determine the logic for determining the outcome of the battle. Here's an example of the output of this program:

```

Enter the name of the first hero (or 'exit' to quit): Black Widow
Enter the name of the second hero (or 'exit' to quit): Captain America

Let the battle begin between Black Widow (power level: 95) and Captain America
(power level: 72)!

Black Widow hits Captain America with power 9! Captain America's power level:
63
Captain America counters and hits Black Widow with power 8! Black Widow's
power level: 87

Black Widow hits Captain America with power 5! Captain America's power level:
58
Captain America counters and hits Black Widow with power 8! Black Widow's
power level: 79

Black Widow hits Captain America with power 8! Captain America's power level:
50
Captain America counters and hits Black Widow with power 7! Black Widow's
power level: 72

Black Widow hits Captain America with power 10! Captain America's power level:

```

40

Captain America counters and hits Black Widow with power 7! Black Widow's power level: 65

Black Widow hits Captain America with power 6! Captain America's power level: 34

Captain America counters and hits Black Widow with power 5! Black Widow's power level: 60

Black Widow hits Captain America with power 4! Captain America's power level: 30

Captain America counters and hits Black Widow with power 5! Black Widow's power level: 55

Black Widow hits Captain America with power 3! Captain America's power level: 27

Captain America counters and hits Black Widow with power 9! Black Widow's power level: 46

Black Widow hits Captain America with power 9! Captain America's power level: 18

Captain America counters and hits Black Widow with power 3! Black Widow's power level: 43

Black Widow hits Captain America with power 8! Captain America's power level: 10

Captain America counters and hits Black Widow with power 5! Black Widow's power level: 38

Black Widow hits Captain America with power 7! Captain America's power level: 3
Captain America counters and hits Black Widow with power 10! Black Widow's power level: 28

Black Widow hits Captain America with power 3! Captain America's power level: 0
Captain America counters and hits Black Widow with power 10! Black Widow's power level: 18

Black Widow is the winner!

Enter the name of the first hero (or 'exit' to quit): exit

Thank you for playing Superhero Battle Royale. See you next time!

4.6 Hands-on time!

Suggest some fun and engaging coding exercises for practising `if/else`, `if/elif/else`, and `match/case` statements, as well as the ternary operator, including problem descriptions and examples of expected output.⁷²

1. **Emoji Translator:** Write a program that translates a given character emoticon into its corresponding meaning. Use a `match/case` statement to determine the meanings.

Example:

```
Input: ":)"
Output: "Smile"
```

2. **Grade Calculator:** Create a program that calculates and displays the letter grade based on a given numerical grade. Use `if/elif/else` statements to assign the appropriate letter grade.

Example:

```
Input: 85
Output: B
```

3. **Pizza Order:** Write a program that takes user input for pizza size and toppings and calculates the total cost. Use `if/elif/else` statements and `in` operator to determine the cost based on size and toppings.

Example:

```
Input: Size - Medium, Toppings - Pepperoni, Mushrooms
Output: Total cost: $12.50
```

4. **Rock, Paper, Scissors Game:** Implement a simple text-based rock, paper, scissors game. Use `match/case` statements to determine the winner based on the player's choices.

Example:

```
Input: Player 1 - Rock, Player 2 - Scissors
Output: Player 1 wins!
```

5. **Weather Clothing Adviser:** Create a program that advises the user on what clothing to wear based on the weather conditions. Use `match/case` statements to provide appropriate clothing recommendations.

Example:

```
Input: Weather - Sunny, Temperature - 28 C
Output: It's sunny! Wear a T-shirt and shorts.
```

6. **Age Group Music Recommender:** Write a program that classifies a person into different age groups based on their age, and recommends appropriate music bands. Use `if/elif/else` statements to categorise the age group.

Example:

```
Input: Age - 45
Output: Listen to music from: Nirvana, R.E.M
```

7. **Age Comparisons:** Create a program that compares three ages to determine who is the oldest and youngest. Use `if/elif/else` statements to make the comparisons.

Example:

```
Input: Ages - 10, 5, 8
Output: Oldest: 10, Youngest: 5
```

These exercises provide practical scenarios where `if/else`, `if/elif/else`, and `match/case` statements, and the ternary operator can be applied. They help reinforce the understanding of conditional statements and decision-making in programming.

Suggest some fun and engaging coding exercises for practising for and while loops, including problem descriptions and examples of expected output.⁷³

1. **Countdown Timer:** Create a program that accepts an input number and counts down from that number to one, printing each number on a new line with a one-second pause between each number.

Example:

```
Input: 5
Output:
5 (wait a second...)
4 (wait a second...)
3 (wait a second...)
2 (wait a second...)
1 (wait a second...)
Done!
```

2. **Repetition Table:** Create a program that generates the repetition table for a given number. The program should display the table from 1 to 10.

Example:

```
Input: 7
Output:
7 x 1 = 7
7 x 2 = 77
7 x 3 = 777
...
7 x 10 = 7777777777
```

3. **Factorial Calculator:** Write a program that calculates the factorial of a given number using a while loop. The factorial of a number is the product of all positive integers less than or equal to that number. The program should display the entire multiplication sequence.

Example:

```
Input: 5
Output: 5! = 5 x 4 x 3 x 2 x 1 = 120
```

4. **Guess the Number Game:** Create a program that generates a random number between 1 and 100. The user should guess the number, and the program should provide feedback (higher or lower) until the correct number is guessed. Use a nested `while/if` statement and display a message with the number of attempts used.

Example:

```
Output:
Guess a number between 1 and 100: 50
Too high!
Guess a number between 1 and 100: 25
Too low!
Guess a number between 1 and 100: 40
Congratulations! You guessed the number in 3 attempts.
```

5. **Sum of Digits:** Write a program that calculates the sum of the digits of a given number using a `while` loop.

Example:

```
Input: 1234
Output: 10 (since 1 + 2 + 3 + 4 = 10)
```

6. **Rock, Paper, Scissors vs Computer (2 out of 3):** Implement 2 out of 3 text-based rock, paper, scissors game (more precisely, first to win two throws, since there can be an unlimited number of ties). Use nested `match/case` statements within a loop with a `break` clause to determine the winner based on the player's choices on the overall run.

Example:

```
Output:
Make your choice: Rock.
Computer plays: Scissors
Result: Player 1 wins!

Make your choice: Scissors.
Computer plays: Scissors
Result: It's a tie!

Make your choice: Paper.
Computer plays: Rock
Result: Player 1 wins!

Winner: Player 1 is the overall winner!
```

7. **Guess User's Age:** Create a program that guesses the user's age by asking a series of yes/no questions. The program should narrow down the possible age range with each question until it reaches the correct age. The program should make educated guesses based on the user's responses to the questions.

Example:

```
Output:
Are you older than 50 (yes/no): no
Are you older than 25 (yes/no): yes
Are you older than 37 (yes/no): no
Are you older than 31 (yes/no): yes
Are you older than 34 (yes/no): no
Are you older than 32 (yes/no): yes
Are you older than 33 (yes/no): yes

I know! You are 34 years old!
```

Suggest some fun and engaging coding exercises to practice nested loop structures, including problem descriptions and examples of expected output.⁷⁴

1. **Prime Numbers:** Write a program that generates prime numbers up to a given limit using a nested `while`/`for` loop. A prime number is a number that is only divisible by 1 and itself.

Example:

Input: 20

- 2. Pattern Printing:** Create a program that prints various patterns using nested loops. For example, you can print a triangle, a square, or even a diamond pattern. Use nested loops to control the number of rows and columns.

Example:

```
Output:  
Triangle Pattern:  
  
*  
  
* *  
  
* * *  
  
* * * *  
  
* * * * *
```

Square Pattern:

```
* * * * *
* * * * *
* * * * *
* * * * *
```

Diamond Pattern:

*
* *
* * * *
* * * * *
* * * * *
* * * *
* *

3. **Times Table Matrix:** Write a program that generates a times table matrix from 1 to 10. Use nested loops to iterate over rows and columns and calculate the product.

Example:

```
Output:
Times Table Matrix:
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
...
10 20 30 40 50 60 70 80 90 100
```

4. **Grade Calculator:** Create a program that calculates the average grade for a class. Prompt the user for the number of students and the number of subjects. Use nested loops to input and calculate grades, and then display the average grade.

Example:

```
Output:
Enter the number of students: 3
Enter the number of subjects: 4

Enter grades for student 1:
Subject 1: 85
Subject 2: 90
Subject 3: 80
Subject 4: 95

Enter grades for student 2:
Subject 1: 75
Subject 2: 70
Subject 3: 85
Subject 4: 80

Enter grades for student 3:
Subject 1: 95
Subject 2: 90
Subject 3: 92
Subject 4: 88

Average Grade: 85.83
```

5. **Letter Race:** Create a program that simulates a race between two user-defined letters over 5 consecutive moves. Each competitor can shift its position to the right by up to 5 random steps in each move. The program should display the positions of the letters after each move and determine the winner at the end of the last move using the ternary operator. You can use nested loops to control the race moves and the competitor steps, respectively.

Example:

```

Input: A, Z
Output:
[Move 1]
..A
....Z

[Move 2]
.....A
.....Z

[Move 3]
....A
.....Z

[Move 4]
.....A
.....Z

[Move 5]
.....A
.....Z

It's a tie!

```

6. **FizzBuzz Game:** Write a program that prints numbers from 1 to 100. For multiples of 3, print "Fizz" instead of the number. For multiples of 5, print "Buzz". For numbers that are multiples of both 3 and 5, print "FizzBuzz". Use nested `for/if/else` structures to check the conditions.

Example:

Output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

Suggest the problem of writing a console-based calculator program that takes two numbers and a selected operation from a menu (decorated with a frame), computes and displays the result, and asks the user if they want to continue. Include an example of the expected output.⁷⁵

Console-based Scientific Calculator:

Write a console-based calculator program that allows the user to perform arithmetic operations. The calculator should provide a menu with a selection of the following operations:

1. Addition (+): Add two numbers.
2. Subtraction (-): Subtract the second number from the first number.
3. Multiplication (*): Multiply two numbers.
4. Division (/): Divide the first number by the second number.
5. Exponentiation (^): Raise the first number to the power of the second number.
6. Square Root (**sqrt**): Compute the square root of a number.
7. Logarithm (**log**): Compute the base 10 logarithm of a number.

Example output:

```
*****
*          CALCULATOR MENU      *
*****
*  1. Addition                  *
*  2. Subtraction                *
*  3. Multiplication             *
*  4. Division                   *
*  5. Exponentiation            *
*  6. Square Root                *
*  7. Logarithm                 *
*  8. Exit                      *
*****
```

Select an operation (1-8): 3

Enter the first number: 5

Enter the second number: 7

Result: $5 * 7 = 35$

```
*****
*          CALCULATOR MENU      *
*****
*  1. Addition                  *
*  2. Subtraction                *
*  3. Multiplication             *
*  4. Division                   *
*  5. Exponentiation            *
*  6. Square Root                *
*  7. Logarithm                 *
*  8. Exit                      *
*****
```

Select an operation (1-8): 8

Bye!

Implementing this problem will allow you to exercise your skills in handling user input, implementing menu-based systems, utilising mathematical functions from the `math` module and implementing nested control structures. Have fun building your scientific calculator!

The Containers Zone

Explain what a container structure is, then list all of the container structures available in Python and provide a brief description for each, using real-life analogies and a friendly tone.⁷⁶

A container structure in Python is a data structure that is capable of holding multiple values or objects. It allows you to organise and store data in a structured manner. Each container structure has its own unique characteristics and is suitable for different purposes.

1. **Lists:** Think of a shopping bag where you can put various items. It is an ordered collection of elements that can be of different types.
2. **Tuples:** Imagine a sealed envelope that contains multiple pieces of information. A tuple is similar to a list, but it is immutable, meaning it cannot be modified once created.
3. **Sets:** Picture a drawer filled with unique items. A set is an unordered collection of unique elements.
4. **Strings:** Consider a string as a sentence or a sequence of characters. It is an immutable container that holds a collection of characters.
5. **Dictionaries:** Visualise a real-life dictionary where words and their definitions are stored. A dictionary is a key-value pair data structure.

These container structures provide flexibility in organising and manipulating data based on your specific needs. By choosing the right container structure, you can efficiently store and process data in your Python programs.

So, let's take a closer look at each of them.

5.1 Lists

Explain everything I need to know about lists in Python in a fun and creative way, using playful examples.⁷⁷

Sure! Let's embark on a playful adventure into the world of Python lists! Imagine you are a wizard with a magical backpack that can hold all sorts of fantastic items. In Python, lists are just like your magical backpack, where you can keep, add, remove and retrieve items at your convenience!

1. **Creating Lists:** To create a list, imagine it as your magical backpack. You can put anything you want inside, like shiny gems, powerful potions, or enchanted scrolls. Just use square brackets [] and separate the items with commas.

2. **Indexing:** Each item in your magical backpack has a special number called an index. It's like a label that helps you find things quickly. Remember, wizards start counting from 0! So, the first item is at index 0, the second item at index 1, and so on. You can access items in your list using these indexes.
3. **Updating and Adding Items:** Your magical backpack is versatile! You can add new items or change the existing ones anytime. Just imagine finding a rare item during your adventures and replacing it with an even more powerful one. You can use the index to update the items in your list.
4. **Slicing:** Sometimes you want to show off your collection of magical items to other wizards. Slicing allows you to take a subset of your list and display it proudly. It's like showing them a glimpse of your extraordinary inventory.
5. **Length and Membership:** Your magical backpack can do more! It can tell you how many items you have using the `len()` function. You can also ask if a particular item is in your list using the `in` keyword. It's like checking if a specific magical artefact is in your possession.
6. **List Operations:** You can perform magical operations on your list! Want to combine two lists? Just use the `+` operator to merge them into a powerful super-list. You can even multiply your list to create multiple copies of its contents. It's like multiplying your magical items to create an army of artefacts!
7. **List Methods:** Your magical backpack comes with handy spells known as methods. These methods can help you sort your items, add new ones, remove unwanted ones, or even reverse the order of your list. They are like special powers that make your list management easier.

So, grab your magical backpack (i.e. Python list) and start filling it with all sorts of wonders! Remember, the more creative you get, the more fun you'll have!

Ok. Show me a code example that illustrates each of the items listed above. Include comments in each code block to explain its purpose. You are welcome to use the superhero vs. villain theme.⁷⁸

Certainly! Here are code examples illustrating each of the items listed above, using a superhero vs. villain theme. Each code block is accompanied by comments that explain the corresponding item, along with the console output.

```

1 # Creating Lists
2 superheroes = [] # Start with an empty list
3 print("Your magical backpack of superheroes is ready!")
4 print(f"List of superheroes: {superheroes}")

```

Your magical backpack of superheroes is ready!

List of superheroes: []

```

1 # Populate the List
2 superheroes = ["Spider-Man", "Iron Man", "Captain America"]
3 print("\nNow your magical backpack of superheroes is filled!")
4 print(f"List of superheroes: {superheroes}")

```

Now your magical backpack of superheroes is filled!

List of superheroes: ['Spider-Man', 'Iron Man', 'Captain America']

```

1 # Indexing
2 print("\nIndexing Example:")
3 print(f"The first superhero is: {superheroes[0]}")
4 print(f"The second superhero is: {superheroes[1]}")
5 print(f"The last superhero is: {superheroes[-1]}")

```

Indexing Example:

The first superhero is: Spider-Man

The second superhero is: Iron Man

The last superhero is: Captain America

```

1 # Updating and Adding Items
2 print("\nUpdating and Adding Items Example:")
3 superheroes[1] = "Black Widow" # Update an item at index 1
4 print("Updated list of superheroes:", superheroes)
5
6 superheroes.append("Hulk") # Add a new item to the end of the list
7 print("Extended list of superheroes:", superheroes)

```

Updating and Adding Items Example:

Updated list of superheroes: ['Spider-Man', 'Black Widow', 'Captain America']

Extended list of superheroes: ['Spider-Man', 'Black Widow', 'Captain America', 'Hulk']

```

1 # Slicing
2 print("\nSlicing Example:")
3 villains = ["Green Goblin", "Loki", "Red Skull", "Thanos", "Venom"]
4 print("List of villains: ", villains)
5 print("The first three villains are:", villains[:3])
6 print("The last two villains are:", villains[-2:])

```

Slicing Example:

List of villains: ['Green Goblin', 'Loki', 'Red Skull', 'Thanos', 'Venom']

The first three villains are: ['Green Goblin', 'Loki', 'Red Skull']

The last two villains are: ['Thanos', 'Venom']

```

1 # Length and Membership
2 print("\nLength and Membership Example:")
3 print("Number of superheroes:", len(superheroes))
4 print("Number of villains:", len(villains))
5
6 hero = input("Enter a superhero name to check if it's in the list: ")
7
8 if hero in superheroes:
9     print(f"Yes, {hero} is in your magical backpack!")
10 else:
11     print(f"Sorry, {hero} is not in your magical backpack.")

```

Length and Membership Example:

Number of superheroes: 4

Number of villains: 5

Enter a superhero name to check if it's in the list: Superman

Sorry, Superman is not in your magical backpack.

```

1 # List Operations
2 print("\nList Operations Example:")
3 all_characters = superheroes + villains # Merge two lists
4 print("All characters in the superhero universe:", all_characters)
5
6 duplicate_villains = villains * 2 # Create multiple copies of the list
7 print("Duplicate villains:", duplicate_villains)

```

List Operations Example:

All characters in the superhero universe: ['Spider-Man', 'Black Widow', 'Captain America', 'Hulk', 'Green Goblin', 'Loki', 'Red Skull', 'Thanos', 'Venom']

Duplicate villains: ['Green Goblin', 'Loki', 'Red Skull', 'Thanos', 'Venom', 'Green Goblin', 'Loki', 'Red Skull', 'Thanos', 'Venom']

```

1 # List Methods
2 print("\nList Methods Example:")
3 print("Original list of superheroes:", superheroes)
4 superheroes.sort() # Sort the list in alphabetical order
5 print("Sorted list of superheroes:", superheroes)
6
7 superheroes.remove("Captain America") # Remove a specific item from the list
8 print("Updated list of superheroes:", superheroes)
9 superheroes.reverse() # Reverse the order of the list
10 print("Reversed list of superheroes:", superheroes)

```

List Methods Example:

```

Original list of superheroes: ['Spider-Man', 'Black Widow', 'Captain America',
'Hulk']
Sorted list of superheroes: ['Black Widow', 'Captain America', 'Hulk', 'Spider-Man']
Updated list of superheroes: ['Black Widow', 'Hulk', 'Spider-Man']
Reversed list of superheroes: ['Spider-Man', 'Hulk', 'Black Widow']

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2s4bz4sy>

Interesting. Now demonstrate how I can use loops and conditionals on lists. This time, use creative food-related examples, taking advantage of Python's randomness features.⁷⁹

Certainly! Here's an example of how you can use loops and conditionals with lists, using a fun food-related theme that incorporates Python's randomness features. The code demonstrates different ways to iterate over a list of food items and apply conditional statements for various food-related scenarios. Enjoy the delicious examples!

```

1 import random
2 foods = ["pizza", "burger", "sushi", "taco", "pasta"]
3 # Looping over a list
4 print("== Menu ==")
5 for food in foods:
6     print(food)

```

```

== Menu ==
pizza
burger
sushi
taco
pasta

```

```

1 # Looping with conditionals
2 print("\n==== Vegetarian Options ===")
3 for food in foods:
4     if food != "burger":
5         print(food)

```

==== Vegetarian Options ===
pizza
sushi
taco
pasta

```

1 # Randomly selecting a food
2 print("\n==== Random Food ===")
3 random_food = random.choice(foods)
4 print("Enjoy your", random_food)

```

==== Random Food ===
Enjoy your sushi

```

1 # Looping with a random twist
2 print("\n==== Food Roulette ===")
3 for i in range(3):
4     random_food = random.choice(foods)
5     if random_food == "sushi":
6         print(f"Round {i+1}: It's your lucky day! You got sushi!")
7     else:
8         print(f"Round {i+1}: Better luck next time. You got {random_food}")

```

==== Food Roulette ===
Round 1: Better luck next time. You got taco
Round 2: It's your lucky day! You got sushi!
Round 3: Better luck next time. You got pasta

```

1 # Random prices with list comprehensions
2 prices = [random.randint(2, 20) for _ in range(len(foods))]
3 print("\n==== Price list ===")
4 for food, price in zip(foods, prices):
5     print(f"{food}\t${price}")

```

```
==== Price list ====
pizza $7
burger $15
sushi $5
taco $10
pasta $14
```

```
1 # Nested Lists and Nested Loops
2 deals = [["sandwich", "crisps", "juice"], ["burger", "fries", "soda"], ["taco", "fruit",
   "beer"]]
3
4 print("\n==== Meal deals ===")
5 for deal in deals:
6     print(f"\nItems in the {deal[0].upper()} meal deal:")
7     for i, item in enumerate(deal):
8         print(f"Item {i+1}: {item}")
```

```
==== Meal deals ===

Items in the SANDWICH meal deal:
Item 1: sandwich
Item 2: crisps
Item 3: juice

Items in the BURGER meal deal:
Item 1: burger
Item 2: fries
Item 3: soda

Items in the TACO meal deal:
Item 1: taco
Item 2: fruit
Item 3: beer
```

```
1 # List Methods and Operations
2 beverages = ["soda", "juice", "beer"]
3
4 print("\n==== Beverages ===")
5 print("Original list of beverages:", beverages)
6 beverages.insert(1, "water")
7 print("Updated list of beverages:", beverages)
```

```

8
9 all_together = foods + beverages
10 print("\n==== All together ===")
11 print("Combined list of foods and beverages:", all_together)

```

```

==== Beverages ===
Original list of beverages: ['soda', 'juice', 'beer']
Updated list of beverages: ['soda', 'water', 'juice', 'beer']

==== All together ===
Combined list of foods and beverages: ['pizza', 'burger', 'sushi', 'taco', 'pasta',
'soda', 'water', 'juice', 'beer']

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/mumh24r8>

*Wait a minute. In the examples above, you use some neat tricks that I'm not aware of, such as: `random.choice()`, `zip()`, `enumerate()` and `insert()`. Can you pls explain these to me?*⁸⁰

Certainly! Let's go through each of those neat tricks used in the code examples.

► **random.choice():**

- This is a function from the `random` module in Python's standard library.
- It allows you to randomly select an element from a given list.
- In the example, `random.choice(foods)` selects one random food item from the `foods` list.

► **zip():**

- `zip()` is another built-in function in Python that combines multiple iterables (lists, tuples, etc.) into a single iterable object.
- It returns pairs of corresponding elements from the input iterables.
- In the example, `for food, price in zip(foods, prices):`, the `zip()` function is used to iterate over two lists, `foods` and `prices`, simultaneously, and the `food` and `price` variables hold the corresponding elements from each list.

► **enumerate():**

- `enumerate()` is a built-in function in Python that allows you to iterate over a list (or any iterable) while also keeping track of the index of each element.
- It returns pairs of index-element tuples.
- In the example, `for i, item in enumerate(deal):`, the `enumerate()` function is used to iterate over each of the `deal` sublists, and the `i` variable holds the index of each item in the `deal`.

► **insert():**

- The `insert()` method is a built-in list operation in Python that allows you to insert an element at a specific index in a list.

- It takes two arguments: the index at which to insert the element, and the element itself.
- The existing elements at and after the specified index are shifted to the right. The `insert()` method modifies the original list and does not create a new list.

These tricks are powerful and commonly used in Python to enhance functionality and make code more concise and readable. They can be very handy in various scenarios, such as generating random selections, iterating over lists with indices, or iterating over multiple lists together.

5.2 Tuples

Explain everything I need to know about tuples in Python in a fun and creative way, using playful examples.⁸¹

Certainly! Let's dive into the fascinating world of tuples in Python. Imagine you're hosting a grand dinner party for your friends, and you want to keep track of the seating arrangement. Tuples can help you with that! Think of a tuple as a special table setting where each guest has a designated seat. Here's what you need to know about tuples:

1. **Immutable Lineup:** Once you set the seating arrangement using a tuple, it remains fixed. You can't change or swap the seats without creating a new tuple. Just like a fancy dinner, once the guests are seated, you can't move them around.
2. **Ordered Placement:** Tuples preserve the order of elements. Just as you meticulously plan the placement of your guests, tuples ensure that the order of the elements remains intact. The first element will always be in the first seat, the second element in the second seat, and so on.
3. **Mix and Match:** Tuples allow you to have a diverse seating arrangement. Each seat can hold a different type of data, such as strings, numbers, or even other tuples. You can mix and match to create complex seating structures. It's like hosting a party where people from different backgrounds and interests come together.
4. **Efficient and Lightweight:** Tuples are lightweight and efficient. They take up less memory compared to other data structures. It's like having elegant place cards that don't take up much space on the table.
5. **Unpacking Trick:** One useful aspect of tuples is tuple unpacking. You can easily assign the elements of a tuple to individual variables in a single line of code. It's like revealing the seating arrangement by handing out name cards to each guest.

Remember, tuples are best suited for situations where you need an ordered collection of elements that shouldn't change. Whether it's a seating arrangement, coordinates on a map, or any other scenario where immutability and order are important, tuples have got you covered!

*Ok. Now let's look at some code samples to see tuples in action! This time, why not use rock bands to illustrate the items you listed previously?*⁸²

Sure! Let's dive into some code examples using rock bands to illustrate the different aspects of tuples. We'll explore each item previously mentioned in a fun and creative way.

1. **Immutable Lineup:** Say you are organising a rock concert, you can use a tuple to define the lineup of bands. Once the lineup is set, it remains fixed throughout the concert. Let's see an example:

```
1 lineup = ("The Beatles", "Led Zeppelin", "AC/DC", "Guns N' Roses", "Queen")
2 print(lineup)
3
4 # Uncommenting the line below will raise an error
5 # lineup[0] = "Coldplay"
```

Once the arrangement is set using a tuple, it cannot be changed. Attempting to modify an element of the tuple will result in an error, because tuples are immutable, just like trying to replace a band after the concert has started.

```
==== Immutable Lineup ====
('The Beatles', 'Led Zeppelin', 'AC/DC', "Guns N' Roses", 'Queen')
```

2. **Ordered Placement:** When planning a concert, you want to ensure that the bands perform in a specific order. Tuples preserve the order of elements. Let's see how it looks:

```
1 lineup = ("The Beatles", "Led Zeppelin", "AC/DC", "Guns N' Roses", "Queen")
2
3 for band in lineup:
4     print("Next band:", band)
```

In this example, the bands perform in the order defined by the tuple. The loop iterates over the lineup, and each band is printed in the correct order.

```
==== Ordered Placement ====
Next band: The Beatles
Next band: Led Zeppelin
Next band: AC/DC
Next band: Guns N' Roses
Next band: Queen
```

3. **Mix and Match:** Rock bands can have diverse lineups with members playing different roles. Tuples allow you to create complex structures by combining different types of data. Let's see an example with the Beatles:

```

1 john = ("John Lennon", 1940, False)
2 paul = ("Paul McCartney", 1942, True)
3 george = ("George Harrison", 1943, False)
4 ringo = ("Ringo Starr", 1940, True)
5
6 beatles = (john, paul, george, ringo)
7
8 for member in beatles:
9     print("\nBeatles member:", member[0])
10    print("Year of birth:", member[1])
11    print("Is alive?", member[2])

```

In this example, each member of the Beatles is represented by a tuple that contains the name, year of birth, and a boolean indicating if they are alive or not. The `beatles` tuple combines the individual member tuples to form the complete band.

==== Mix and Match ====

Beatles member: John Lennon

Year of birth: 1940

Is alive? True

Beatles member: Paul McCartney

Year of birth: 1942

Is alive? True

Beatles member: George Harrison

Year of birth: 1943

Is alive? False

Beatles member: Ringo Starr

Year of birth: 1940

Is alive? True

- Efficient and Lightweight:** Tuples are lightweight and consume less memory compared to other data structures such as lists. Let's see an example:

```

1 beatles_list = ["John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr"]
2 beatles_tuple = ("John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr")
3
4 print("Size of the list:", beatles_list.__sizeof__())
5 print("Size of the tuple:", beatles_tuple.__sizeof__())

```

In this example, we use the `__sizeof__()` function to determine the memory usage of the `beatles_list` and `beatles_tuple`. You'll notice that the size of the tuple is smaller compared to the list.

```
==== Efficient and Lightweight ====
Size of the list: 72
Size of the tuple: 56
```

5. **Unpacking Trick:** Tuple unpacking allows you to assign the elements of a tuple to individual variables. Let's see an example with the Beatles:

```
1 band = ("The Beatles", "John Lennon", "Paul McCartney", "George Harrison", "Ringo
2   Starr")
3
4
5 band_name, *members = band
6
7 print("Band name:", band_name)
8 print("Band members:", members)
```

In this example, we unpack the `band` tuple into individual variables. The first variable `band_name` holds the band's name, and the `members` variable is assigned a list containing the individual band members.

```
==== Unpacking Trick ====
Band name: The Beatles
Band members: ['John Lennon', 'Paul McCartney', 'George Harrison', 'Ringo
  Starr']
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2mprsujd>

These code examples demonstrate the various aspects of tuples using rock bands fun analogies. Tuples are versatile and can be used in various scenarios beyond rock bands.

Wait! There is a weird '' before the "members" variable in the unpacking trick example. What exactly is that?⁸³*

The asterisk (*) before a variable when tuple unpacking is used to collect any remaining items in the tuple into a list. It is known as the "extended iterable unpacking" syntax in Python. In the example with the Beatles, the asterisk is used to assign the remaining members of the band tuple to the `members` variable. This allows for flexibility in case the number of members in the band is not known beforehand.

However, if you know the quantity of remaining elements in the tuple, you can assign them to individual variables, as shown in the following variation of the example:

```

1 band = ("The Beatles", "John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr")
2
3 band_name, vocalist, guitarist, bassist, drummer = band
4
5 print("Band name:", band_name)
6 print("Vocalist:", vocalist)
7 print("Guitarist:", guitarist)
8 print("Bassist:", bassist)
9 print("Drummer:", drummer)

```

Band name: The Beatles
 Vocalist: John Lennon
 Guitarist: Paul McCartney
 Bassist: George Harrison
 Drummer: Ringo Starr

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2c5durbt>

5.3 Sets

Explain everything I need to know about sets in Python in a fun and creative way, using playful examples.⁸⁴

Of course! Let's dive into the magical world of sets in Python. Imagine you have a collection of unique magical creatures. Each creature possesses distinct abilities and characteristics. That's exactly what sets are all about!

- Unique Collection:** A set is like a special treasure chest where you can store a collection of unique items. It's like having a magical zoo where no two creatures can be the same. Each creature is unique and different from the others.
- No Duplicates:** Sets ensure that there are no duplicate creatures allowed. If you try to add the same creature twice, the set will magically keep only one instance of it. It's as if the set has a magical power to maintain its uniqueness.
- Fast Membership Testing:** Sets excel at answering questions like, "Is this creature already in the collection?" They can perform lightning-fast membership tests. It's like having a magical scanner that quickly detects whether a creature is part of the set or not.
- Magical Operations:** Sets also offer powerful magical operations that you can perform on them. You can combine sets to create new sets, find common creatures between sets, or even find unique creatures that exist only in one set and not in the other.

5. **Mutable and Unordered:** Sets are mutable, meaning you can add or remove creatures from them. However, they have no particular order. It's like having a magical gathering where creatures can freely move around without any fixed arrangement.
6. **Magical Removal:** You can magically remove creatures from a set based on their unique properties. Just say the name of the creature you want to remove, and the set will do its magic and eliminate that creature.
7. **Joining Forces:** Sets allow you to unite forces by performing union operations. You can combine multiple sets together, and it will create a grand collection of all the unique creatures from all the sets. It's like merging magical tribes into one big gathering.

Good. Now let's look at some code samples to see sets in action! This time, let's use planets fun facts to illustrate the items you listed previously.⁸⁵

Certainly! Here are some code samples that demonstrate sets using planets fun facts:

```

1 # Unique Collection
2 print("\n==== Unique Collections ===")
3 planets = set(["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"])
4 moons = set([0, 0, 1, 2, 79, 82, 27, 14])
5 locations = set(["inner", "inner", "inner", "inner", "outer", "outer", "outer", "outer"])
6 galaxies = set(["Milky Way", "Milky Way"])
7
8 print(f"Planets (unique values: {len(planets)}):", planets)
9 print(f"Moons (unique values: {len(moons)}):", moons)
10 print(f"Locations (unique values: {len(locations)}):", locations)
11 print(f"Galaxies (unique values: {len(galaxies)}):", galaxies)

```

In this example, we are creating sets from the list of planets and lists containing the corresponding number of moons, locations within the solar system, and galaxies they belong to.

To create a set in Python, you can use curly braces ({}) or the `set()` function. Simply enclose the elements you want to include in the set within the curly braces or pass them as arguments to the `set()` function. Sets are unordered collections of unique elements, meaning that duplicate values are automatically eliminated. They are useful when you want to store a collection of items without any specific order and ensure uniqueness among the elements.

```
==== Unique Collections ====
Planets (unique values: 8): {'Saturn', 'Mars', 'Mercury', 'Venus', 'Neptune',
    'Jupiter', 'Uranus', 'Earth'}
Moons (unique values: 7): {0, 1, 2, 14, 79, 82, 27}
Locations (unique values: 2): {'inner', 'outer'}
Galaxies (unique values: 1): {'Milky Way'}
```

```
1 # No Duplicates
2 print("\n==== No Duplicates ===")
3 planets = {"Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"}
4 print('Before adding "Earth":', planets)
5 planets.add("Earth") # Adding a duplicate planet will have no effect
6 print('After adding "Earth":', planets)
```

```
==== No Duplicates ===
Before adding "Earth": {'Saturn', 'Mars', 'Mercury', 'Venus', 'Neptune', 'Jupiter',
    'Uranus', 'Earth'}

After adding "Earth": {'Saturn', 'Mars', 'Mercury', 'Venus', 'Neptune', 'Jupiter',
    'Uranus', 'Earth'}
```

```
1 # Fast Membership Testing
2 print("\n==== Fast Membership ===")
3 print("Is Earth in planets? ", "Earth" in planets)
4 print("Is Pluto in planets? ", "Pluto" in planets)
```

```
==== Fast Membership ===
Is Earth in planets? True
Is Pluto in planets? False
```

```
1 # Magical Operations
2 print("\n==== Magical Operations ===")
3 gas_giants = {"Jupiter", "Saturn"}
4 rocky_planets = {"Mercury", "Venus", "Earth", "Mars"}
5 ring_systems = {"Saturn", "Uranus", "Neptune"}
6 common_gas_rocky = gas_giants.intersection(rocky_planets) # Outputs an empty set
7 common_ring_gas = ring_systems.intersection(gas_giants)
8
9 print("Planets that are gas giants: ", gas_giants)
10 print("Planets with rocky surfaces: ", rocky_planets)
```

```

11 print("Planets having ring systems: ", ring_systems)
12 print("Planets with gas and rock in common: ", common_gas_rocky)
13 print("Planets with rings and gas in common: ", common_ring_gas)

```

```

== Magical Operations ==
Planets that are gas giants: {'Jupiter', 'Saturn'}
Planets with rocky surfaces: {'Earth', 'Mars', 'Mercury', 'Venus'}
Planets having ring systems: {'Neptune', 'Saturn', 'Uranus'}
Planets with gas and rock in common: set()
Planets with rings and gas in common: {'Saturn'}

```

```

1 # Mutable and Unordered
2 print("\n== Mutable and Unordered ==")
3 print("Before removing 'Neptune':", planets)
4 planets.remove("Neptune")
5 print("After removing 'Neptune':", planets)
6
7 # Magical Removal
8 print("\n== Magical Removal ==")
9 print("Before discarding 'Pluto':", planets)
10 planets.discard("Pluto") # Removing a planet that doesn't exist has no effect
11 print("After discarding 'Pluto':", planets)

```

The main difference between the `remove()` and `discard()` methods in sets is how they handle the removal of an element that does not exist in the set:

- ▶ `remove()` raises an error if the element is not found in the set, and stops the program's execution.
- ▶ `discard()` does not raise an error if the element is not found; instead, it silently continues without making any changes to the set; thus it avoids program interruption. It is recommended to use `discard()` unless you are sure that the element is in the set.

```

== Mutable and Unordered ==
Before removing 'Neptune': {'Saturn', 'Mars', 'Mercury', 'Venus', 'Neptune',
    'Jupiter', 'Uranus', 'Earth'}
After removing 'Neptune': {'Saturn', 'Mars', 'Mercury', 'Venus', 'Jupiter',
    'Uranus', 'Earth'}
== Magical Removal ==
Before discarding 'Pluto': {'Saturn', 'Mars', 'Mercury', 'Venus', 'Jupiter',
    'Uranus', 'Earth'}
After discarding 'Pluto': {'Saturn', 'Mars', 'Mercury', 'Venus', 'Jupiter',
    'Uranus', 'Earth'}

```

```

1 # Joining Forces
2 print("\n==== Joining Forces ===")
3 dwarf_planets = {"Pluto", "Eris", "Haumea"}
4 all_planets = planets.union(dwarf_planets)
5 print("Original planets: ", planets)
6 print("Dwarf planets: ", dwarf_planets)
7 print("All planets: ", all_planets)

```

```

==== Joining Forces ===
Original planets: {'Saturn', 'Mars', 'Mercury', 'Venus', 'Jupiter', 'Uranus',
'Earth'}
Dwarf planets: {'Haumea', 'Pluto', 'Eris'}
All planets: {'Saturn', 'Eris', 'Pluto', 'Mars', 'Haumea', 'Mercury', 'Venus',
'Jupiter', 'Uranus', 'Earth'}

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/sfhaje>

With sets, you can create a unique collection of planets, perform operations like checking membership, finding common planets, and adding or removing planets from the set. Sets offer a convenient and efficient way to work with collections of unique elements, just like exploring the fascinating world of planets with their distinct characteristics.

5.4 Strings

Explain everything I need to know about strings in Python in a fun and creative way, using playful examples.⁸⁶

Sure! Let's embark on a fun journey through the world of strings in Python!

Imagine strings as a magical collection of letters, symbols, and words that can be manipulated and transformed to create all sorts of exciting things. They are like the colorful building blocks of text in Python.

1. **Mix and Match:** Strings have super-mixing powers! They can be combined using the "+" operator, just like mixing potions to create new spells. For example, imagine combining the strings "Hello" and "World" to create the powerful spell "HelloWorld". Wow!
2. **Mighty Replication:** Strings can also be multiplied, like replicating a spell to make it more powerful. Just imagine multiplying the string "Abracadabra" by 3, and behold, you get "AbracadabraAbracadabraAbracadabra"! It's like casting a spell multiple times!

3. **Case Secrets:** Unlock hidden potential by changing the case of strings, like turning "Hello" into "hELLO" or "hello" into "HELLO". It's like using secret codes to unlock different possibilities.
4. **Character Quest:** "Strings are akin to treasure chests filled with characters. In other words, a string is a sequence of characters, including letters (e.g., "ABC...Zabc...z"), symbols ("!@?:>+..."), or digits ("0123...9"), enclosed within single or double quotation marks (' or "). You can access individual characters within a string using indexing, similar to uncovering hidden treasures on a map. For instance, by accessing the 3rd character of the string 'Python', you will discover the letter 't'. It's like uncovering a concealed gem within a textual chest!"
5. **Transforming Magic:** Guess what? Strings are not just static. They can be transformed using methods. You can split a string into a list of words, replace specific words or characters, or even join multiple strings together. It's like having a magical toolbox to shape and mould textual information.

So, let your creativity soar! Use strings to create whimsical stories, funny jokes, or even your own secret language. Strings in Python are your trusty companions and a powerful tools for text manipulation!

Good. Now let's look at some code examples to see strings in action! Consider using classic children's books to illustrate the items you listed previously.⁸⁷

Certainly! Let's dive into the enchanting world of classic children's books and explore the power of strings. Here are some delightful code examples:

```

1 # Mix and Match (single strings)
2 print("\n==== Mix and Match (single strings) ===")
3 book1 = "Alice in Wonderland"
4 book2 = "Peter Pan"
5 books = book1 + " and " + book2
6 print(f"Book 1: {book1}")
7 print(f"Book 2: {book2}")
8 print(f"{books} together make a terrific reading pack for children!")
9
10 # Mix and Match (list of strings)
11 print("\n==== Mix and Match (list of strings) ===")
12 titles = ["The Fellowship of the Ring", "The Two Towers", "The Return of the King"]
13
14 # Use str.join() instead of "+" to concatenate multiple strings
15 collection = " and ".join(titles)
16 print(f"{collection} are the beloved books of the Lord of the Rings trilogy.")

```

```

==== Mix and Match (single strings) ====
Book 1: Alice in Wonderland
Book 2: Peter Pan
Alice in Wonderland and Peter Pan together make a terrific reading pack for children!

==== Mix and Match (list of strings) ====
The Fellowship of the Ring and The Two Towers and The Return of the King are the
beloved books of the Lord of the Rings trilogy.

```

```

1 # Mighty Replication
2 print("\n==== Mighty Replication ===")
3 phrase = "\tI'll huff and I'll puff and I'll blow your house down\n"
4 wolf_talk = phrase * 3
5 dialogue = "And then,\n" + wolf_talk + "the wolf exclaimed."
6 print(dialogue)

```

```

==== Mighty Replication ===
And then,
    I'll huff and I'll puff and I'll blow your house down
    I'll huff and I'll puff and I'll blow your house down
    I'll huff and I'll puff and I'll blow your house down
the wolf exclaimed.

```

```

1 # Case Secrets
2 print("\n==== Case Secrets ===")
3 book = "Harry Potter and the sorcerer's stone"
4 title_case = book.title()
5 upper_case = book.upper()
6 lower_case = book.lower()
7 swap_case = book.swapcase()
8 print("Original: ", book)
9 print("Title case: ", title_case)
10 print("Upper case: ", upper_case)
11 print("Lower case: ", lower_case)
12 print("Swap case: ", swap_case)

```

```

==== Case Secrets ===
Original: Harry Potter and the sorcerer's stone
Title case: Harry Potter And The Sorcerer'S Stone
Upper case: HARRY POTTER AND THE SORCERER'S STONE

```

Lower case: harry potter and the sorcerer's stone

Swap case: hARRY pOTTER AND THE SORCERER'S STONE

```

1 # Character Quest
2 print("\n==== Character Quest ===")
3 story = "Little Red Riding Hood"
4 print(f"Original text:\t\t {story}")
5 print(f"First character:\t {story[0]}") # Accessing first character with index
6 print(f"First 6 characters:\t {story[:6]}") # First 6 characters using slicing
7 print(f"Every other character:\t {story[::2]}") # Traverse every other character
8 vowels = "aeiou"
9 print(f"Consonants only:\t {''.join([c for c in story if c not in vowels])}")
10 print(f"Vowels found:\t\t {set([c for c in story if c in vowels])}")

```

```

==== Character Quest ====
Original text: Little Red Riding Hood
First character: L
First 6 characters: Little
Every other character: Ltl e iigho
Consonants only: Lttl Rd Rdng Hd
Vowels found: {'o', 'e', 'i'}

```

```

1 # Transforming Magic
2 print("\n==== Transforming Magic ===")
3 quote = "Oh, mirror on the wall, who's the fairest one of all?"
4 split_words = quote.split()
5 replaced_word = quote.replace("all", "[any]")
6 blanks_removed = quote.replace(" ", "")
7 reversed_string = quote[::-1]
8 reversed_words = split_words[::-1]
9 print("Original quote:\t\t", quote)
10 print("Character length:\t", len(quote))
11 print("Split words:\t\t", split_words)
12 print("Words length:\t\t", len(split_words))
13 print("Replaced word:\t\t", replaced_word)
14 print("Blanks removed:\t\t", blanks_removed)
15 print("Reversed string:\t", reversed_string)
16 print("Reversed words:\t\t", reversed_words)
17 print("Sentence completion:")
18 for i in range(len(split_words)):
    print("\t" + ".join(split_words[:i + 1]))"
19

```

```

==== Transforming Magic ====
Original quote:      Oh, mirror on the wall, who's the fairest one of all?
Character length:  53
Split words:        ['Oh,', 'mirror', 'on', 'the', 'wall,', "who's", 'the',
                     'fairest', 'one', 'of', 'all?']
Words length:       11
Replaced word:     Oh, mirror on the w[any], who's the fairest one of [any]?
Blanks removed:    Oh,mirroronthewall,who'sthefairestoneofall?
Reversed string:   ?lla fo eno tseriaf eht s'ohw ,llaw eht no rorrim ,hO
Reversed words:   ['all?', 'of', 'one', 'fairest', 'the', "who's", 'wall,', 'the',
                  'on', 'mirror', 'Oh,']
Sentence completion:
Oh,
Oh, mirror
Oh, mirror on
Oh, mirror on the
Oh, mirror on the wall,
Oh, mirror on the wall, who's
Oh, mirror on the wall, who's the
Oh, mirror on the wall, who's the fairest
Oh, mirror on the wall, who's the fairest one
Oh, mirror on the wall, who's the fairest one of
Oh, mirror on the wall, who's the fairest one of all?

```

```

1 # More Tricks
2 print("\n==== More Tricks ====")
3 books = ['"Beauty and the Beast"', '"The Little Prince"', '"The Wizard of Oz"', '"1001
   Nights"', '"The Jungle Book"']
4 for book in books:
5     # Find
6     print("\nBook title:\t\t", book)
7     print("Oz' found at position:", book.find("Oz"))
8     # StartsWith and EndsWith
9     print("Starts with 'The':\t", book.startswith('The'))
10    print("Ends with 'Book':\t", book.endswith('Book'))
11    # IsAlpha and IsDigit
12    print("Begins with a word:\t", book[1:4].isalpha())
13    print("Begins with a number:\t", book[1:4].isdigit())
14    # Strip
15    print('Stripped of double quotes:', book.strip('"'))

```

```

==== More Tricks ====
Book title:      "Beauty and the Beast"
'Oz' found at position: -1
Starts with 'The': False
Ends with 'Book': False
Begins with a word: True
Begins with a number: False
Stripped of double quotes: Beauty and the Beast

Book title:      "The Little Prince"
'Oz' found at position: -1
Starts with 'The': True
Ends with 'Book': False
Begins with a word: True
Begins with a number: False
Stripped of double quotes: The Little Prince

Book title:      "The Wizard of Oz"
'Oz' found at position: 15
Starts with 'The': True
Ends with 'Book': False
Begins with a word: True
Begins with a number: False
Stripped of double quotes: The Wizard of Oz

Book title:      "1001 Nights"
'Oz' found at position: -1
Starts with 'The': False
Ends with 'Book': False
Begins with a word: False
Begins with a number: True
Stripped of double quotes: 1001 Nights

Book title:      "The Jungle Book"
'Oz' found at position: -1
Starts with 'The': True
Ends with 'Book': True
Begins with a word: True
Begins with a number: False
Stripped of double quotes: The Jungle Book

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/y55mdy6n>

The above code performs several string operations on a list of book titles:

1. It uses the `find()` method to locate the position of the substring `Oz` within each book title.
2. The `startswith()` method checks if a book title begins with the substring `"The`.
3. The `endswith()` method verifies if a book title ends with the substring `Book`.
4. By slicing a portion of the book title, it determines if it consists of alphabetic characters using `isalpha()`, or numeric digits using `isdigit()`.
5. The `strip()` method removes double quotes from the book title.

These operations provide insights into the book titles, such as finding specific substrings, checking prefixes and suffixes, examining the character composition of portions, and removing unnecessary characters. So, go ahead and explore more examples, and have a blast discovering the magical and vibrant world of strings with all their amazing tools!

5.5 Dictionaries

Explain everything I need to know about dictionaries in Python in a fun and creative way, using playful examples.⁸⁸

Sure! Let's embark on an exciting journey into the realm of dictionaries in Python. Imagine you have a superhero universe where each hero possesses unique powers, allies, and arch-enemies. You want to create an encyclopedia of such super-universe, capturing the fascinating details of each hero. Each hero can have their own page in this encyclopedia, a unique biography filled with their characteristics and connections. That's precisely where dictionaries come in handy!

1. **Key-Value Pairs:** Dictionaries consist of key-value pairs, just like secret codes and their corresponding meanings. Each hero's identity serves as the key, and their details, such as powers, allies and enemies, are the values associated to that key. It's like having a powerful indexing system that connects heroes to their own attributes.
2. **Superfast Lookups:** Dictionaries excel at answering questions like, "What are the powers of this hero?" They can perform lightning-fast lookups based on a hero's identity. It's like having a superpowered database that quickly retrieves the details of a specific hero.
3. **No Duplicates:** Dictionaries ensure that each hero is unique. Just like in a super-universe, no two heroes can have the same identity. If you try to add a hero with an existing identity, the dictionary will keep only one instance of them, preserving the uniqueness of each hero.

4. **Dynamic and Versatile:** Dictionaries are dynamic and versatile. You can modify them on the fly, updating their powers or allies, adding new heroes or removing existing ones. They provide a flexible structure to accommodate the ever-evolving world of superheroes.

5. **Cross-referencing:** Dictionaries allow you to unravel the intricate hero-villain relationships and connect heroes with their allies. You can search for villains based on heroes or vice versa, discovering epic battles and rivalries within the hero universe. Dictionaries also enable you to explore the intricate web of relationships between heroes, creating an interconnected superhero community.

Remember, dictionaries are your gateway to building dynamic containers filled with unique items and their associated attributes. They empower you to create, manipulate, and explore their contents in a flexible and versatile manner. So, I encourage you to unlock the secrets of dictionaries in Python and harness their power to unleash your creativity!

Great. Now let's look at some code samples to see dictionaries in action! Let's stick to the superheroes theme to illustrate the items you listed previously.⁸⁹

Certainly! Here are some code samples showcasing the power of dictionaries in the realm of superheroes:

```

1 # Create a superhero dictionary
2 spiderman = {
3     "powers": ["superhuman strength", "wall-crawling", "web-slinging"],
4     "allies": ["Iron Man", "Captain America", "Black Widow"],
5     "enemies": ["Green Goblin", "Venom", "Doc Octopus"]
6 }
7 # Key-value Pairs
8 print("\n==== Key-Value Pairs ===")
9 print("\nKeys in Spiderman dictionary:")
10 for key in spiderman.keys():
11     print(f"\t{key}")
12
13 print("\nValues in Spiderman dictionary:")
14 for value in spiderman.values():
15     print(f"\t{value}")
16
17 print("\nKey-Value pairs in Spiderman dictionary:")
18 for key, value in spiderman.items():
19     print(f"\t{key} -> {value}")

```

```

==== Key-Value Pairs ====
Keys in Spiderman dictionary:
powers
allies
enemies

Values in Spiderman dictionary:
['superhuman strength', 'wall-crawling', 'web-slinging']
['Iron Man', 'Captain America', 'Black Widow']
['Green Goblin', 'Venom', 'Doc Octopus']

Key-Value pairs in Spiderman dictionary:
powers -> ['superhuman strength', 'wall-crawling', 'web-slinging']
allies -> ['Iron Man', 'Captain America', 'Black Widow']
enemies -> ['Green Goblin', 'Venom', 'Doc Octopus']

```

```

1 # Superfast Lookups
2 print("\n==== Superfast Lookups ===")
3
4 # Attributes are indexed by key
5 print("Spider-Man's powers:", spiderman["powers"])
6 print("Spider-Man's allies:", spiderman["allies"])
7 print("Spider-Man's enemies:", spiderman["enemies"])

```

```

==== Superfast Lookups ====
Spider-Man's powers: ['superhuman strength', 'wall-crawling', 'web-slinging']
Spider-Man's allies: ['Iron Man', 'Captain America', 'Black Widow']
Spider-Man's enemies: ['Green Goblin', 'Venom', 'Doc Octopus']

```

```

1 # No Duplicates
2 print("\n==== No Duplicates ===")
3 print("Spider-Man's powers:\t", spiderman["powers"])
4
5 # Duplicate key, overwrite
6 spiderman.update({"powers" : ["stamina", "fancy custome"]})
7 print("Spider-Man's updated powers:", spiderman["powers"])

```

```

==== No Duplicates ====
Spider-Man's powers: ['superhuman strength', 'wall-crawling', 'web-slinging']
Spider-Man's updated powers: ['stamina', 'fancy custome']

```

```

1 # Dynamic and Versatile
2 print("\n==== Dynamic and Versatile ===")
3 print("Spider-Man's enemies:\t", spiderman["enemies"])
4
5 spiderman["enemies"].remove("Green Goblin")
6 spiderman["enemies"].append("Carnage")
7 print("Spider-Man's updated enemies:", spiderman["enemies"])

```

```

==== Dynamic and Versatile ===
Spider-Man's enemies: ['Green Goblin', 'Venom', 'Doc Octopus']
Spider-Man's updated enemies: ['Venom', 'Doc Octopus', 'Carnage']

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/epvbefcm>

What about putting together a superhero encyclopedia? Can I implement it as a dictionary of dictionaries? Some kind of nested dictionary? Use a code example to illustrate.⁹⁰

Certainly! You can create a superhero encyclopedia using a nested dictionary structure. Each superhero can be a key in the outer dictionary, with the corresponding value being an inner dictionary that stores the superhero's details. Here is an example code snippet along with the corresponding execution outputs:

```

1 # Creating the superhero encyclopedia
2 superheroes = {
3     "Wonder Woman": {
4         "powers": ["Superhuman strength", "Lasso of Truth", "Flight"],
5         "allies": ["Superman", "Batman"],
6         "enemies": ["Cheetah", "Ares"],
7         "debut": 1941
8     },
9     "Black Widow": {
10        "powers": ["Master spy", "Hand-to-hand combat", "Espionage"],
11        "allies": ["Captain America", "Hawkeye"],
12        "enemies": ["Taskmaster", "Red Guardian"],
13        "debut": 1964
14    },
15    "Captain Marvel": {
16        "powers": ["Superhuman strength", "Energy projection", "Flight"],
17        "allies": ["Iron Man", "Thor"],
18        "enemies": ["Thanos", "Yon-Rogg"],
19        "debut": 1968
20    },

```

```

21 "Storm": {
22     "powers": ["Weather manipulation", "Flight", "Energy absorption"],
23     "allies": ["Wolverine", "Cyclops"],
24     "enemies": ["Magneto", "Apocalypse"],
25     "debut": 1975
26 },
27 "Superman": {
28     "powers": ["Superhuman strength", "Flight", "Heat vision"],
29     "allies": ["Batman", "Wonder Woman"],
30     "enemies": ["Lex Luthor", "Doomsday"],
31     "debut": 1938
32 },
33 "Spider-Man": {
34     "powers": ["Wall-crawling", "Spider-sense", "Web-slinging"],
35     "allies": ["Iron Man", "Captain America"],
36     "enemies": ["Green Goblin", "Venom", "Doc Octopus"],
37     "debut": 1962
38 },
39 "Hulk": {
40     "powers": ["Superhuman strength", "Invulnerability", "Regeneration"],
41     "allies": ["Thor", "Black Widow"],
42     "enemies": ["Abomination", "Red Hulk"],
43     "debut": 1962
44 },
45 "Batman": {
46     "powers": ["Intelligence", "Riches", "Martial arts"],
47     "allies": ["Superman", "Wonder Woman"],
48     "enemies": ["Joker", "Bane"],
49     "debut": 1939
50 },
51 "Wolverine": {
52     "powers": ["regeneration", "adamantium claws", "heightened senses"],
53     "allies": ["Professor X", "Cyclops", "Storm"],
54     "enemies": ["Sabretooth", "Magneto", "Mystique"],
55     "debut": 1974
56 }
57 }
58 print(superheroes.keys())

```

```

dict_keys(['Wonder Woman', 'Black Widow', 'Captain Marvel', 'Storm', 'Superman',
          'Spider-Man', 'Hulk', 'Batman', 'Wolverine'])

```

```

1 # Printing details of a superhero given a name in the encyclopedia
2 name = input("\nEnter a name to search for in the superhero encyclopedia: ")
3 name = name.title() # Capitalize first letter of each word in name
4
5 if name in superheroes:
6     superhero = superheroes[name]
7     print("-----")
8     print(f"{name:^20}")
9     print("-----")
10    print(f"Powers: {', '.join(superhero['powers'])}")
11    print(f"Allies: {', '.join(superhero['allies'])}")
12    print(f"Enemies: {', '.join(superhero['enemies'])}")
13    print(f"Debut Year: {superhero['debut']}")
14 else:
15     print("Sorry, superhero not found in the encyclopedia.")

```

Enter a name to search for in the superhero encyclopedia: Thor
 Sorry, superhero not found in the encyclopedia.

Enter a name to search for in the superhero encyclopedia: black widow

 Black Widow

 Powers: Master spy, Hand-to-hand combat, Espionage
 Allies: Captain America, Hawkeye
 Enemies: Taskmaster, Red Guardian
 Debut Year: 1964

```

1 # Adding new superheroes
2 superheroes["Captain America"] = {
3     "powers": ["Superhuman strength", "Shield throwing", "Leadership"],
4     "allies": ["Iron Man", "Thor"],
5     "enemies": ["Red Skull", "Winter Soldier"],
6     "debut": 1941
7 }
8 # Removing a superhero
9 del superheroes["Hulk"]
10 # Updating superheroes details
11 superheroes["Wolverine"]["allies"].append("Jean Grey")
12 superheroes["Spider-Man"]["enemies"].remove("Doc Octopus")

```

```

13 # Printing the updated superhero encyclopedia
14 value = input("\nWhich attribute (powers/allies/enemies/debut) would you like to see? ")
15 value = value.lower() # Convert attribute to lowercase
16 print("-----")
17 print(f"{'Superhero Encyclopedia':^50}")
18 print("-----")
19
20 for key in superheroes.keys():
21     if value in superheroes[key]:
22         print(f"{key:^20} -> {superheroes[key][value]}")
23     else:
24         print(f"Sorry, no {value} found for superhero {key}")

```

Which attribute (powers/allies/enemies/debut) would you like to see? allies

Superhero Encyclopedia

Wonder Woman	-> ['Superman', 'Batman']
Black Widow	-> ['Captain America', 'Hawkeye']
Captain Marvel	-> ['Iron Man', 'Thor']
Storm	-> ['Wolverine', 'Cyclops']
Superman	-> ['Batman', 'Wonder Woman']
Spider-Man	-> ['Iron Man', 'Captain America']
Batman	-> ['Superman', 'Wonder Woman']
Wolverine	-> ['Professor X', 'Cyclops', 'Storm', 'Jean Grey']
Captain America	-> ['Iron Man', 'Thor']

Which attribute (powers/allies/enemies/debut) would you like to see? age

Superhero Encyclopedia

Sorry, no age found for superhero Wonder Woman
 Sorry, no age found for superhero Black Widow
 Sorry, no age found for superhero Captain Marvel
 Sorry, no age found for superhero Storm
 Sorry, no age found for superhero Superman
 Sorry, no age found for superhero Spider-Man
 Sorry, no age found for superhero Batman
 Sorry, no age found for superhero Wolverine

Which attribute (powers/allies/enemies/debut) would you like to see? debut

Superhero Encyclopedia

Wonder Woman	-> 1941
Black Widow	-> 1964
Captain Marvel	-> 1968
Storm	-> 1975
Superman	-> 1938
Spider-Man	-> 1962
Batman	-> 1939
Wolverine	-> 1974
Captain America	-> 1941

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/5feexkfe>

Nice! You also mentioned that dictionaries enable us to explore intricate relationships and connections between items within the dictionary. What does that mean? Can dictionaries help me determine relationships such as 'the enemies of my allies are my enemies', 'the allies of my allies are my friends', 'the heroes with the same superpower as mine are my buddies' or 'display a table of superheroes sorted by debut year'? Please provide some examples.⁹¹

Certainly! Here are four examples of cross-referencing data items using dictionaries.

- ▶ Example 1: The enemies of my allies are my enemies.

```

1 # Constructing a dictionary to store extended enemies
2 allies_enemies = {}
3 for hero, details in superheroes.items():
4     allies = details["allies"]
5     enemies = details["enemies"]
6     for ally in [hero] + allies:
7         # Check if the hero is not already present in the allies_enemies dictionary
8         if hero not in allies_enemies:
9             allies_enemies[hero] = []
10        # Check if the ally exists in the superheroes dictionary
11        if ally in superheroes:
12            # Extend the list of enemies for the hero with the enemies of the ally
13            allies_enemies[hero].extend(superheroes[ally]["enemies"])
14
15 # Printing the original enemies
16 print('*'*80)
17 print(f"{'Original Enemies':^80}")

```

```
18 print('-'*80)
19 for hero, details in superheroes.items():
20     print(f"{hero:^10} -> {details['enemies']}")  

21
22 # Printing the extended enemies
23 print('-'*80)
24 print(f"'The Enemies of My Allies are My Enemies':^80}")  

25 print('-'*80)
26 for ally, enemies in allies_enemies.items():
27     print(f"{ally:^10} -> {enemies}")  

28
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/yckcbr4k>

The code aims to construct a dictionary, `allies_enemies`, to store the extended enemies of heroes based on their allies. The algorithm iterates through each hero and their details in the `superheroes` dictionary. It retrieves the lists of allies and enemies for each hero. For each hero and their allies, it checks if the hero is not already present in the `allies_enemies` dictionary. If not, it initialises an empty list for the hero. Then, it checks if the ally exists in the `superheroes` dictionary. If the ally exists, it extends the list of enemies for the hero with the enemies of the ally by using the `extend()` function. Finally, the code prints the original enemies and the extended enemies.

Original Enemies

Wonder Woman -> ['Cheetah', 'Ares']
Black Widow -> ['Taskmaster', 'Red Guardian']
Captain Marvel -> ['Thanos', 'Yon-Rogg']
Storm -> ['Magneto', 'Apocalypse']
Superman -> ['Lex Luthor', 'Doomsday']
Spider-Man -> ['Green Goblin', 'Venom']
Batman -> ['Joker', 'Bane']
Wolverine -> ['Sabretooth', 'Magneto', 'Mystique']
Captain America -> ['Red Skull', 'Winter Soldier']

The Enemies of My Allies are My Enemies

```
Wonder Woman -> ['Cheetah', 'Ares', 'Lex Luthor', 'Doomsday', 'Joker', 'Bane']
Black Widow -> ['Taskmaster', 'Red Guardian', 'Red Skull', 'Winter Soldier']
Captain Marvel -> ['Thanos', 'Yon-Rogg']
```

```

Storm    -> ['Magneto', 'Apocalypse', 'Sabretooth', 'Magneto', 'Mystique']
Superman -> ['Lex Luthor', 'Doomsday', 'Joker', 'Bane', 'Cheetah', 'Ares']
Spider-Man -> ['Green Goblin', 'Venom', 'Red Skull', 'Winter Soldier']
Batman   -> ['Joker', 'Bane', 'Lex Luthor', 'Doomsday', 'Cheetah', 'Ares']
Wolverine -> ['Sabretooth', 'Magneto', 'Mystique', 'Magneto', 'Apocalypse']
Captain America -> ['Red Skull', 'Winter Soldier']

```

- ▶ Example 2: The allies of my allies are my friends.

```

1 # Constructing a dictionary to store extended friends
2 allies_friends = {}
3 for hero, details in superheroes.items():
4     allies = details["allies"]
5     for ally in [hero] + allies:
6         if ally in superheroes:
7             # Finding friends of the current hero's ally, excluding the hero itself
8             friends = set(superheroes[ally]["allies"]) - set([hero])
9             allies_friends.setdefault(hero, []).extend(friends)
10
11 # Printing the original allies
12 print('*'*80)
13 print(f"{'Original Allies':^80}")
14 print('*'*80)
15 for hero, details in superheroes.items():
16     # Printing the allies of each hero
17     print(f"{hero:^10} -> {details['allies']} ")
18
19 # Printing the extended allies (friends)
20 print('*'*80)
21 print(f"{'The Allies of My Allies are My Friends':^80}")
22 print('*'*80)
23 for hero, friends in allies_friends.items():
24     # Printing the extended friends list, excluding duplicates
25     print(f"{hero:^10} -> {set(friends)}")
26

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/4wsy98ra>

The provided code aims to construct a dictionary, `allies_friends`, which stores the extended friends of each hero based on their allies. The algorithm iterates through each hero and their details in the `superheroes` dictionary. For each hero, it retrieves their list of allies. It then checks if each ally exists as a key in the `superheroes` dictionary and, if so, finds their friends by subtracting the hero itself from their allies using set operations.

The `setdefault()` function is used to extend the list of friends for the current hero in the `allies_friends` dictionary. It ensures that if the hero is not yet a key in the `allies_friends` dictionary, a new empty list is created as the default value. This allows for easy extension of the friends list without overwriting existing friends. Finally, the code prints the original allies for each hero and the extended allies (with friends) by iterating over the `allies_friends` dictionary and excluding duplicates using sets.

```

-----  

----- Original Allies -----  

-----  

Wonder Woman -> ['Superman', 'Batman']  

Black Widow -> ['Captain America', 'Hawkeye']  

Captain Marvel -> ['Iron Man', 'Thor']  

    Storm -> ['Wolverine', 'Cyclops']  

    Superman -> ['Batman', 'Wonder Woman']  

Spider-Man -> ['Iron Man', 'Captain America']  

    Batman -> ['Superman', 'Wonder Woman']  

Wolverine -> ['Professor X', 'Cyclops', 'Storm', 'Jean Grey']  

Captain America -> ['Iron Man', 'Thor']  

-----  

----- The Allies of My Allies are My Friends -----  

-----  

Wonder Woman -> {'Superman', 'Batman'}  

Black Widow -> {'Iron Man', 'Captain America', 'Hawkeye', 'Thor'}  

Captain Marvel -> {'Iron Man', 'Thor'}  

    Storm -> {'Cyclops', 'Professor X', 'Wolverine', 'Jean Grey'}  

    Superman -> {'Batman', 'Wonder Woman'}  

Spider-Man -> {'Captain America', 'Iron Man', 'Thor'}  

    Batman -> {'Superman', 'Wonder Woman'}  

Wolverine -> {'Storm', 'Professor X', 'Cyclops', 'Jean Grey'}  

Captain America -> {'Iron Man', 'Thor'}
```

- Example 3: Heroes with same superpowers are buddies.

```

1 # Constructing a dictionary to store superpower buddies
2 superpower_buddies = {}
3 for hero, details in superheroes.items():
4     powers = details["powers"]
5     for power in powers:
6         superpower_buddies.setdefault(power, []).append(hero)
```

```

7 print('*'*80)
8 print(f"{'Heroes with same superpowers are buddies':^80}")
9 print('*'*80)
10 for power, buddies in superpower_buddies.items():
11     print(f"|[{power.title():^20}| {', '.join(buddies):<56}]|")
12     print('*'*80)
13

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/4msck3s>

The code aims to construct a dictionary, `superpower_buddies`, to store superheroes who share the same superpowers. The algorithm iterates through each hero and their details in the `superheroes` dictionary. For each hero, it retrieves their list of powers. It then iterates through each power and uses the `append()` function to add the hero to the list of buddies associated with that power in the `superpower_buddies` dictionary.

The `setdefault()` function ensures that if the power is not yet a key in the `superpower_buddies` dictionary, a new empty list is created as the default value. Finally, the code prints the `superpower_buddies` dictionary, displaying each power and the corresponding buddies. It formats the output in a table-like structure using the `|` character to separate the power and buddies columns.

Heroes with same superpowers are buddies	
Superhuman Strength	Wonder Woman, Captain Marvel, Superman, Captain America
Lasso Of Truth	Wonder Woman
Flight	Wonder Woman, Captain Marvel, Storm, Superman
Master Spy	Black Widow
Hand-To-Hand Combat	Black Widow
Espionage	Black Widow
Energy Projection	Captain Marvel
WeatherManipulation	Storm
Energy Absorption	Storm
Heat Vision	Superman

Wall-Crawling	Spider-Man
Spider-Sense	Spider-Man
Web-Slinging	Spider-Man
Intelligence	Batman
Riches	Batman
Martial Arts	Batman
Regeneration	Wolverine
Adamantium Claws	Wolverine
Heightened Senses	Wolverine
Shield Throwing	Captain America
Leadership	Captain America

► Example 4: Sorting superheroes by debut year.

```

1 # Creating a sorted list of keys based on debut year
2 debut_dict = {key: superheroes[key]["debut"] for key in superheroes.keys()}
3 sorted_keys = sorted(debut_dict.keys(), key=debut_dict.get)
4 # Printing the table of superheroes sorted by debut year
5 print('*'*30)
6 print(f'{\'Heroes sorted by debut year\'}:{^30}')
7 print('*'*30)
8 print(f'{\'Superhero\'}:{^15} | {\'Debut Year\'}:{^15}')
9 print('*'*30)
10 for key in sorted_keys:
11     print(f'{key:<15} | {superheroes[key][\'debut\']:{^15}}')
12 print('*'*30)
13

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/dshc9uf4>

This code first creates a dictionary `debut_dict` to store the debut years of superheroes. The dictionary comprehension `{key: superheroes[key]["debut"] for key in superheroes.keys()}` iterates over the keys of the `superheroes` dictionary and assigns the corresponding debut year to each key in `debut_dict`. This step effectively extracts the debut years for each superhero, similar to the list comprehension we discussed earlier.

The second line of the code sorts the keys in `debut_dict` based on their corresponding values (debut years) using the `sorted()` function. The `key` parameter is set to `debut_dict.get`, which retrieves the value (debut year) for each key. This sorting step ensures that the superheroes are arranged in ascending order based on their debut year. The last block of the code prints the table by iterating through the sorted list of keys and applying some formatting enhancements.

Heroes sorted by debut year		
Superhero		Debut Year
Superman		1938
Batman		1939
Wonder Woman		1941
Captain America		1941
Spider-Man		1962
Black Widow		1964
Captain Marvel		1968
Wolverine		1974
Storm		1975

Come on! Let your imagination soar as you explore more examples and have a fantastic time unravelling the captivating world of dictionaries with all their incredible features and possibilities!. Get ready to experience the magic of data organisation and retrieval!

5.6 Hands-on time!

Suggest some fun and engaging coding exercises for practising lists, tuples, and sets. Each exercise should be provided with a problem statement and examples of expected output.⁹²

1. **List Manipulation:** Given a list of numbers, write a program to perform the following operations:

- ▶ Remove all even numbers from the list.
- ▶ Square each remaining number in the list.
- ▶ Return the updated list.

Example:

```
Input: [1, 2, 3, 4, 5, 6, 7, 8]
Output: [1, 9, 25, 49]
```

- 2. Tuple Concatenation:** Given two tuples, write a program to concatenate them and return a new tuple. Example:

```
Input:
Tuple1: (1, 2, 3)
Tuple2: (4, 5, 6)
Output: (1, 2, 3, 4, 5, 6)
```

- 3. Set Operations with Strings:** Given two sets of strings, perform the following operations and print the results:

- ▶ Union: Find the union of the two sets.
- ▶ Intersection: Find the common strings between the two sets.
- ▶ Difference: Find the strings that are in the first set but not in the second set.

Example:

```
Input:
Set1: {"apple", "banana", "orange"}
Set2: {"orange", "grape", "kiwi"}
Output:
Union: {"apple", "banana", "orange", "grape", "kiwi"}
Intersection: {"orange"}
Difference: {"apple", "banana"}
```

- 4. List Filtering:** Given a list of numbers, write a program to filter out all prime numbers greater than 10 and return a new list. Example:

```
Input: [5, 12, 8, 17, 3, 20, 41]
Output: [5, 12, 8, 3, 20]
```

- 5. Unique Permutations:** Given a list of numbers, write a program to generate all unique permutations of the numbers. Example:

```
Input: [1, 2, 3]
Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

6. **Tuple Sorting:** Given a list of tuples containing names and ages, write a program to sort the tuples based on the ages in descending order. Example:

```
Input: [("John", 25), ("Alice", 30), ("Bob", 22)]
Output: [("Alice", 30), ("John", 25), ("Bob", 22)]
```

7. **Set Operations with Multiple Sets:** Given a list of sets, write a program to perform the following operations and print the results:

- ▶ Union: Find the union of all sets.
- ▶ Intersection: Find the common elements across all sets.
- ▶ Difference: Find the elements that are present in the first set but not in any other sets.

Example:

```
Input:
Sets: [{1, 2, 3}, {2, 3, 4}, {3, 4, 5}]
Output:
Union: {1, 2, 3, 4, 5}
Intersection: {3}
Difference: {1}
```

8. **List Rearrangement:** Given a list of strings, write a program to rearrange the strings in such a way that each string starts with the last character of the previous string. If there is no such string, continue with any non-used string from the list. Example:

```
Input: ["apple", "orange", "banana", "rhino", "elephant", "tiger"]
Output: ["apple", "elephant", "tiger", "rhino", "orange", "banana"]
```

9. **Unique Combinations:** Given a list of numbers, write a program to generate all unique combinations of a specific length **k** from the list. Example:

```
Input: [1, 2, 3], k = 2
Output: [[1, 2], [1, 3], [2, 3]]
```

10. **Tuple Manipulation:** Given a list of tuples containing names and scores, write a program to find the average score for each name and return a list of tuples in the format (**name**, **average_score**), sorted by average score in descending order. Example:

```
Input: [("John", 80), ("Alice", 90), ("Bob", 75), ("Alice", 95)]
Output: [("Alice", 92.5), ("John", 80), ("Bob", 75)]
```

11. **List Partitioning:** Given a list of numbers, write a program to partition the list into sublists such that the sum of each sublist is less than or equal to a given maximum value k . The sublists should be as balanced as possible. Example:

```
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 15
Output: [[1, 2, 3, 4, 5], [6, 7], [8, 9], [10]]
```

12. **Matrix Operations:** Given two matrices represented as lists of lists, write a program to perform matrix operations such as addition, subtraction, and multiplication. Example:

```
Input:
Matrix A: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Matrix B: [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
Output:
Addition: [[10, 10, 10], [10, 10, 10], [10, 10, 10]]
Subtraction: [[-8, -6, -4], [-2, 0, 2], [4, 6, 8]]
Multiplication: [[30, 24, 18], [84, 69, 54], [138, 114, 90]]
```

Suggest some fun and engaging coding exercises for practising with strings and dictionaries, including nested dictionaries. Each exercise should be provided with a problem statement and examples of expected output.⁹³

1. **Word Counter:** Write a program that takes a string as input and counts the frequency of each word in the string. Example:

```
Input: "I love to code, code, and code!"
Output: {'I': 1, 'love': 1, 'to': 1, 'code': 3, 'and': 1}
```

2. **Dictionary Merger:** Write a program that takes two dictionaries as input and merges them into a single dictionary. If a key is present in both dictionaries, the values should be concatenated as a list. Example:

```
Input:
dict1 = {'A': 1, 'B': 2, 'C': 3}
dict2 = {'B': 4, 'D': 5}
Output: {'A': 1, 'B': [2, 4], 'C': 3, 'D': 5}
```

- 3. Nested Dictionary Access:** Write a program that accesses values from a nested dictionary using a list of keys. Example:

```
Input:
data = {'person': {'name': 'John', 'age': 25, 'address': {'city': 'New York',
    'country': 'USA'}}}
keys = ['person', 'address', 'city']
Output: 'New York'
```

- 4. Character Frequency Counter:** Write a program that takes a string as input and counts the frequency of each character in the string. Example:

```
Input: "Hello, World!"
Output: {'H': 1, 'e': 1, 'l': 3, 'o': 2, ',': 1, ' ': 1, 'W': 1, 'r': 1, 'd': 1, '!': 1}
```

- 5. Dictionary Manipulation:** Write a program that performs various manipulations on a dictionary. The program should display a menu of options for the user to choose from. Example:

```
Input:
my_dict = {'A': 1, 'B': 2}
Output:
{'A': 1, 'B': 2, 'C': 3} # After adding {'C': 3}
{'A': 1, 'B': 5} # After updating 'B' to 5
{'A': 1} # After removing 'B'
Key 'A' exists in the dictionary: True
Key 'B' exists in the dictionary: False
```

- 6. Anagram Checker:** Write a program that takes two strings as input and determines if they are anagrams of each other. An anagram is a word or phrase formed by rearranging the letters of another word or phrase. Example:

```
Input: "listen", "silent"
Output: True
```

- 7. Character Replacement:** Write a program that takes a string and replaces all occurrences of a specified character with another character. Example:

```
Input: "Hello, World!", replace 'l' with 'x'
Output: "Hexxo, Worxd!"
```

8. **Dictionary Inversion:** Write a program that takes a dictionary as input and returns a new dictionary where the keys and values are swapped. Example:

```
Input: {'A': 1, 'B': 2, 'C': 3}
Output: {1: 'A', 2: 'B', 3: 'C'}
```

9. **Vowel Counter:** Write a program that takes a string as input and counts the number of vowels (a, e, i, o, u) in the string. Example:

```
Input: "Hello, World!"
Output: 3
```

10. **Nested Dictionary Update:** Write a program that updates the values of a nested dictionary based on a specified condition. Example:

```
Input:
data = {'A': {'score': 80}, 'B': {'score': 90}, 'C': {'score': 70}}
threshold = 75
Output: {'A': {'score': 'Pass'}, 'B': {'score': 'Pass'}, 'C': {'score':
'Fail'}}
```

11. **Nested Dictionary Search:** Write a program that searches for a specific value in a nested dictionary and returns the corresponding key(s) where the value is found. Example:

```
Input:
data = {'A': {'score': 80}, 'B': {'score': 90}, 'C': {'score': 80}, 'D':
{'score': 70}}
target = 80
Output: ['A', 'C']
```

12. **Word Mode Removal:** Write a program that takes a string as input and removes the word(s) with the highest frequency count. If multiple words have the same highest frequency count, remove all of them. Example:

```
Input: "The quick brown fox jumps over the lazy dog. The dog barks loudly."
Output: "quick brown fox jumps over the lazy. barks loudly."
```


6.1 Modularisation and Reusability

Explain briefly the concept of modularisation and reusability, as well as why they are useful in programming projects. Then, using real-world analogies, explain the various approaches for implementing these notions in a friendly but neat manner.⁹⁴

Reusability and modularisation are like secret tricks of the trade in software development, especially for big and complex projects. They come with a bunch of benefits that make life easier for programmers.

When you write reusable code, you don't have to start from scratch every time. You can save lots of time and effort by using existing solutions instead of reinventing the wheel. And with modularisation, you break down your code into smaller, manageable pieces. It's like having puzzle pieces that fit together perfectly. This makes it easier to understand, maintain, and fix any bugs that pop up. Plus, you can reuse those modular pieces in different parts of your project.

So, when it comes to tackling those larger projects, embracing reusability and modularisation is key. It helps you work smarter, collaborate seamlessly, and keeps your code shining bright for the long run. In Python, you have a bunch of cool techniques and practices to put these ideas into action:

- 1. Functions:** Functions are like specialised tools in a kitchen. They encapsulate specific tasks and can be reused across different recipes. Similarly, in programming, functions encapsulate specific operations, promoting code reusability. By utilising functions, programmers can write modular code that can be easily reused, making it easier to handle larger and more complex projects.
- 2. Classes and Object-Oriented Programming (OOP):** Classes in OOP are like specialised workers on a construction site. Each worker has a specific role and expertise. Similarly, classes encapsulate data and behaviour into objects, promoting code reuse. By leveraging classes and objects, programmers can create modular and scalable code, making it easier to manage complexity in bigger projects.
- 3. Modules and Packages:** Modules and packages are like toolboxes with labelled compartments for different tools. They provide a way to organise and package related code components, promoting code reusability. By using modules and packages, programmers can easily find and reuse code functionalities, allowing for efficient development and maintenance of larger projects.

- 4. Libraries and Frameworks:** Libraries and frameworks are like books in a library or bookstore. Instead of writing everything from scratch, programmers can utilise existing libraries and frameworks that offer pre-built code and functionalities for specific tasks. By leveraging these resources, programmers can save time and effort, accelerating the development process of bigger projects.
- 5. Design Patterns:** Design patterns are like architectural blueprints for constructing buildings. They provide proven solutions to common problems in software design. By following established design patterns, programmers can write modular and scalable code that addresses complex requirements. Design patterns promote code reusability and maintainability, making them essential for bigger and more complex projects.
- 6. Separation of Concerns:** Separation of concerns is like dividing a project among experts with specific areas of expertise. By breaking down a program into smaller modules, each responsible for a specific task or functionality, programmers can achieve code modularity and reusability. This approach enables better organisation, collaboration, and maintenance of code in larger projects.

By employing these techniques and practices, Python developers can create reusable and modular code that is easier to understand, maintain, and extend. These concepts also contribute to overall code quality, collaboration, and scalability.

Wow! That's a impressive list of powerful ideas. However, since I am still learning the ropes, please recommend one or two items from the list I should start with as a beginner, and then describe each item's key components in a relaxing and entertaining manner.⁹⁵

Absolutely! As a beginner, I recommend starting with functions and classes. These concepts are fundamental building blocks in programming and will provide you with a solid foundation for understanding more advanced topics.

- 1. Functions:** Functions are like gourmet recipes in the world of programming. Just like a recipe guides you in creating a delicious dish, functions guide the computer in performing specific tasks. Think of functions as mini-programs within your program. They have a name and a set of instructions that perform a particular action. Functions can take inputs (ingredients) and produce outputs (the final dish). The best part? Once you create a function, you can use it over and over again, saving you time and effort. Here are some more technical details about functions in Python:

- ▶ **Specialised Solvers:** Functions are like specialised problem solvers. They take inputs (parameters) and perform specific operations on them to produce outputs (return values). Functions help break down the complexity of the problems into manageable pieces. You can think of them as mini-programs within your code that can be called and reused whenever you need them. Functions are defined using the `def` statement.

- ▶ **Returns:** Functions can return values using the `return` statement. This allows them to produce a result or output that can be used in other parts of the code. It's like a function giving you back a gift after performing its task.
- ▶ **Parameters:** Functions can accept parameters, which are inputs provided when the function is called. Parameters allow you to pass values into the function, providing the necessary data for the function to work with. It's like giving instructions or information to the function to guide its actions.
- ▶ **Lambda Functions:** Lambda functions, also known as anonymous functions, are compact and inline functions that don't require a formal `def` statement. They are defined using the `lambda` keyword and are useful for simple, one-line operations. Lambda functions are like short, agile helpers that can assist you quickly without much ceremony.

2. Classes: Imagine a world where you can create your own superheroes with unique powers and abilities. That's exactly what classes allow you to do in programming. A class is like a blueprint that defines the properties (attributes) and methods (superpowers) of your custom-made superheroes (objects). It encapsulates related data and behaviour into a single entity. With classes, you can create multiple instances (individual superhero copies) that share the same attributes and abilities defined by the class. Here are some more technical details about classes in Python:

- ▶ **Mighty templates:** In OOP, classes act as templates for creating objects. They define the attributes (variables) that describe the object's state and the methods (functions) that define its behaviour. It's like designing the blueprint of a building before constructing it. Classes are defined using the `class` statement.
- ▶ **Constructors:** Constructors are special methods used to initialise objects. They are called when an object is created and allow you to set up the initial state or values of the object's attributes. Constructors ensure that objects are properly initialised and ready to be used. It's like creating a blueprint for creating objects with predefined settings.
- ▶ **Attributes:** In OOP, objects have attributes that store data. Attributes represent the characteristics or properties of an object. They can be variables associated with an object and hold information. It's like an object having different characteristics, such as color, size, or type.
- ▶ **Methods:** Methods are functions defined inside a class and are used to perform operations or actions related to the object. They can manipulate the object's data, interact with other objects, or provide functionality specific to the object's purpose. It's like an object having abilities or behaviours that it can perform.

Starting with functions and classes will give you hands-on experience in writing reusable and modular code. It will empower you to break down complex tasks into smaller, manageable parts and provide a solid foundation for further exploration in the exciting world of programming. So put on your apron and cape, and dive into the delights of gourmet functions and the adventures of heroic classes!

6.2 Functions

Ok. So let's start with some code examples to see functions in action! Consider using social networking-related ideas to illustrate the items you listed previously. Feel free to use entertaining Harry Potter trivia and random features.⁹⁶

Certainly! See the examples below.

- **Specialised Solvers:** Here's some examples to illustrate this concept.

```

1 # Example 1: Tagging Friends
2 import random
3
4 # Define a function to randomly tag a friend using the 'def' statement
5 def tag_friend():
6     friends = ["Alice", "Bob", "Charlie", "David", "Eve"]
7     tagged_friend = random.choice(friends)
8     # Display the result
9     print("You tagged a friend:", tagged_friend)
10
11 # Usage
12 print("\n==== Tagging a friend ====")
13 for _ in range(5):
14     tag_friend()

```

In this example, the function `tag_friend()` is defined using the `def` statement with no input parameters, indicated by the empty parentheses `()`. This function serves as a specialised solver for the problem of randomly selecting and tagging a friend from the user's network. It leverages the `random.choice()` function from the `random` library to accomplish this task.

By calling the `tag_friend()` function multiple times within a loop, you can repeatedly tag different friends, effectively reusing the code and conserving code space. This approach encapsulates the logic of friend selection and tagging into a reusable function, promoting code modularity and maintainability. By encapsulating this functionality within a function, you can easily invoke it whenever needed, avoiding the need to duplicate the code throughout your program.

```

You tagged a friend: Charlie
You tagged a friend: Eve
You tagged a friend: Charlie
You tagged a friend: Bob
You tagged a friend: Alice

```

```

1 # Example 2: Receiving Notifications
2 import datetime, time
3 # Define a function to get the notification datetime
4 def get_notification_datetime():
5     current_datetime = datetime.datetime.now()
6     notification_datetime = current_datetime.strftime("%d/%b/%Y %H:%M:%S")
7     # Display the result
8     print("New notification received at:", notification_datetime)
9     time.sleep(random.randint(0, 10)) # Wait some seconds before the next call
10
11 # Usage
12 print("\n==== Get notification ===")
13 for _ in range(3):
14     get_notification_datetime()

```

In this example, the function `get_notification_datetime()` retrieves the current date-time and formats it as '`dd/mmm/yyyy hh:mm:ss`'. This function serves as a specialised solver for the problem of extracting the timestamp of a received notification. To accomplish this task, it utilises the `datetime.datetime.now()` function from the `datetime` library to obtain the current datetime and the `strftime()` method to format it according to the specified format. The resulting formatted notification datetime is then displayed using the updated printout.

By calling the `get_notification_datetime()` function multiple times in a loop, you can simulate receiving notifications at different times, effectively reusing the code without the need to repeat the same lines of code. This approach promotes code reuse, eliminating redundancy and ensuring a more efficient and concise codebase.

```

==== Get notification ===
New notification received at: 09/Jun/2023 02:49:37
New notification received at: 09/Jun/2023 02:49:45
New notification received at: 09/Jun/2023 02:49:51

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/ye29dv4m>

► **Returns:** Here's some examples to illustrate this concept.

```

1 # Example 1: Get User Information
2 def get_user_info():
3     # Simulate retrieving some user information from a social media platform
4     name = "Harry Potter"
5     age = 17
6     return name, age

```

```

7
8 # Example 2: Get Profile Picture
9 def get_profile_picture():
10    # Simulate retrieving the user's profile picture
11    picture_url = "https://twitter.com/Official_HPWiki/photo"
12    return picture_url
13
14 # Example 3: Get Friends Count
15 def get_friends_count():
16    # Simulate retrieving the number of friends in the user network
17    friends = ["Hermione", "Ron", "Neville", "Luna"]
18    return len(friends)
19
20 # Main function
21 def main():
22    # Get user profile data
23    name, age = get_user_info()
24    picture_url = get_profile_picture()
25    friends_count = get_friends_count()
26
27    # Display user profile
28    print("==== User Information ====")
29    print("Name:", name)
30    print("Age:", age)
31    print("Picture:", picture_url)
32    print("Friends Count:", friends_count)
33
34 # Call the main function
35 main()

```

In Example 1, the `get_user_info()` function simulates retrieving user information from a social media platform. In this example, we assume that we obtain the user's `name` and `age`. By using the `return` statement, the function can simultaneously return multiple values, in this case, a string and a number.

In Example 2, the function `get_profile_picture()` is defined to simulate retrieving the user's profile picture from a social media platform; here, we assume it returns the URL of the picture, that is. the `picture_url` string.

In Example 3, the function `get_friends_count()` is defined to simulate obtaining the number of friends that a user has on a social media platform. The `return` statement in this case returns the count of friends, which corresponds to the length of the `friends` list obtained using the `len()` function.

Lastly, the `main()` function serves as the entry point for the program. It is a common practice in Python to have a `main()` function that acts as a central control point for executing the main logic of the program.

In the provided code snippet, the `main()` function is responsible for coordinating the execution of the other functions. It calls the `get_user_info()`, `get_profile_picture()`, and `get_friends_count()` functions to obtain the user's information, profile picture URL, and friends count. We assign the returned values to variables (`name`, `age`, `picture_url`, `friends_count`) and display the user profile using `print()` statements.

Having a `main()` function promotes code organisation and readability. It separates the main control logic of the program from function definitions, making it easier to understand and maintain the code. Additionally, it allows for better code structure and modularity, promoting good programming practices.

```
===== User Information =====
Name: Harry Potter
Age: 17
Picture: https://twitter.com/Official_HPWiki/photo
Friends Count: 4
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/2p8bj2k9>

► **Parameters:** Here's some examples to illustrate this concept.

```
1 import random
2
3 # Pre-built lists of users, posts, IDs, and comments
4 users = ["Harry Potter", "Hermione Granger", "Ron Weasley", "Luna Lovegood"]
5 posts = {
6     123: "I solemnly swear that I am up to no good.",
7     456: "Happiness can be found, even in the darkest of times, if one only
8         remembers to turn on the light.",
9     789: "It is our choices, Harry, that show what we truly are, far more than our
10        abilities.",
11    101112: "We've all got both light and dark inside us. What matters is the part
12        we choose to act on.",
13 }
14 comments = [
15     ("Hermione Granger", "That's a brilliant quote!"),
16     ("Ron Weasley", "Couldn't agree more, mate!"),
17     ("Luna Lovegood", "Such wisdom in those words."),
18     ("Harry Potter", "I'm glad these quotes resonate with everyone."),
19 ]
```

```

17
18 # Function to send friend request
19 def send_friend_request(sender, receiver):
20     print(f"[{sender}] has sent a friend request to [{receiver}]")
21
22 # Function to print a post
23 def print_post(post_id):
24     print(f"Post ID: {'-'*80}{post_id}")
25     label = ' Post ID: %s ' % (post_id)
26     print(f"{label: '-'*80}{post_id}")
27     print(posts[post_id])
28     print('-'*80)
29
30 # Function to like a post
31 def like_post(user, post_id):
32     print(f"{user} liked post with ID {post_id}")
33
34 # Function to comment on a post
35 def comment_on_post(user, post_id, comment):
36     print(f"{user} commented on post with ID {post_id}: {comment}")
37
38 # Main function
39 def main():
40     # Repeat actions multiple times
41     for _ in range(5):
42         sender = random.choice(users)
43         receiver = random.choice(users)
44         post_id = random.choice([key for key in posts.keys()])
45         commenter, comment = random.choice(comments)
46
47         print_post(post_id)
48         like_post(sender, post_id)
49         comment_on_post(commenter, post_id, comment)
50         send_friend_request(sender, receiver)
51         print()
52
53 # Call the main function
54 main()

```

In the first example, the `send_friend_request()` function takes two parameters: `sender` and `receiver`. These parameters represent the Harry Potter characters involved in the

friend request. By passing the appropriate values to the function, you can simulate sending a friend request from one character to another.

The second example, `like_post()`, demonstrates a function that takes the user and `post_id` parameters. The user parameter represents the character who is liking the post, and the `post_id` parameter specifies the ID of the post. By providing the necessary values, you can simulate a character liking a specific post.

The `comment_on_post()` function accepts the user, `post_id`, and comment parameters. These parameters allow you to specify the user who is commenting, the post ID on which the comment is made, and the content of the comment. By passing the appropriate values to the function, you can simulate a character commenting on a particular post with a specific comment.

Lastly, the `main()` function utilises a for loop to repeat the actions multiple times. Within each iteration, the function randomly selects a sender and receiver from the users list, a post from the posts list, an ID from the ids list, and a comment from the comments list. It then calls the appropriate functions (`send_friend_request()`, `like_post()`, and `comment_on_post()`) with the randomly chosen values.

By repeating this process, the code simulates sending friend requests, liking posts, and commenting on posts with random users, posts, IDs, and comments. This showcases how parameters allow you to provide different values to functions, making them adaptable and reusable in various scenarios.

```
----- Post ID: 789 -----
It is our choices, Harry, that show what we truly are, far more than our
abilities.

-----
Ron Weasley liked post with ID 789
Luna Lovegood commented on post with ID 789: Such wisdom in those words.
[Ron Weasley] has sent a friend request to [Ron Weasley]

----- Post ID: 123 -----
I solemnly swear that I am up to no good.

-----
Luna Lovegood liked post with ID 123
Luna Lovegood commented on post with ID 123: Such wisdom in those words.
[Luna Lovegood] has sent a friend request to [Hermione Granger]

----- Post ID: 456 -----
Happiness can be found, even in the darkest of times, if one only remembers to
turn on the light.
```

```

Hermione Granger liked post with ID 456
Luna Lovegood commented on post with ID 456: Such wisdom in those words.
[Hermione Granger] has sent a friend request to [Ron Weasley]

----- Post ID: 101112 -----
We've all got both light and dark inside us. What matters is the part we
choose to act on.

-----
Luna Lovegood liked post with ID 101112
Harry Potter commented on post with ID 101112: I'm glad these quotes resonate
with everyone.

[Luna Lovegood] has sent a friend request to [Hermione Granger]

----- Post ID: 123 -----
I solemnly swear that I am up to no good.

-----
Luna Lovegood liked post with ID 123
Hermione Granger commented on post with ID 123: That's a brilliant quote!
[Luna Lovegood] has sent a friend request to [Harry Potter]

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/mtu2uha9>

► **Lambda Functions:** Here's some examples to illustrate this concept.

```

1 # Example 1: Sorting friends by name length
2 friends = ["Harry Potter", "Hermione Granger", "Ron Weasley", "Draco Malfoy"]
3
4 # Sort the friends list based on the length of their names using a lambda function
5 sorted_friends = sorted(friends, key=lambda x: len(x))
6
7 # Print the sorted friends list
8 print("\n==== Sorted Friends by Name Length ===")
9 for friend in sorted_friends:
10    print(f"{friend} ({len(friend)})")
11
12 # Example 2: Filtering friends with names starting with 'H'
13 friends = ["Harry Potter", "Hermione Granger", "Ron Weasley", "Draco Malfoy"]
14
15 # Filter the friends list to names starting with 'H' using a lambda function
16 filtered_friends = list(filter(lambda x: x.startswith('H'), friends))
17
18 # Print the filtered friends list
19 print("\n==== Friends with Names Starting with 'H' ===")

```

```

20 for friend in filtered_friends:
21     print(friend)

```

Example 1 demonstrates the use of a lambda function as the key parameter in the `sorted()` function. The lambda function `lambda x: len(x)` takes a friend's name as input `x` and returns the length of the name using `len(x)`. This lambda function serves as the basis for sorting the friends list by the length of their names.

Example 2 showcases the use of a lambda function as the filtering condition in the `filter()` function. The lambda function `lambda x: x.startswith('H')` takes a friend's name as input `x` and returns a boolean value indicating whether the name starts with 'H', using `startswith('H')`. This lambda function is used to filter the friends list and include only those friends whose names start with 'H'.

```

==== Sorted Friends by Name Length ====
Ron Weasley (11)
Harry Potter (12)
Draco Malfoy (12)
Hermione Granger (16)

==== Friends with Names Starting with 'H' ====
Harry Potter
Hermione Granger

```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/3jkk8meu>

Lambda functions are compact and inline functions that are useful for simple, one-line operations. They provide a convenient way to define functions on the fly without the need for a formal `def` statement.

6.3 Classes and Objects

Ok. So let's continue with some code examples to see classes in action! Consider using classic movies and characters trivia to illustrate the items you listed previously, including random features.⁹⁷

Certainly! Below are a few code examples that demonstrate the use of classes in the context of movies trivia.

- **Mighty Templates:** Classes are like mighty templates for creating objects with predefined structures. They define the overall structure and behaviour of objects. Once you have defined a class, you can use it to create as many instances or objects from that class as you need. Each object will have its own set of data and will behave according to the template defined by the class.

To create a class in Python, use the keyword `class` followed by the class name and a colon. To define methods within the class, use the `def` keyword followed by the method name, parentheses, and any parameters required. The first parameter of each method should be `self`, which stores a reference to the actual instance executing the method. Indentation indicates that the methods are part of the class definition.

So, without further ado, let's embark on creating a mighty template for a Movie Franchise!

```
1 class MovieFranchise:
2     # Setter method
3
4     def set_franchise(self, name, movies):
5         self.name = name
6         if movies > 0:
7             self.movies = movies
8         else:
9             print(f"Warning: Number of movies for '{name}' franchise must be
greater than 0.")
10            self.movies = 1
11
12    # Getter method
13
14    def get_franchise(self):
15        return self.name, self.movies
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
```

```

34 # Call the main function
35 main()
36

```

In this example, we define a class `MovieFranchise` with methods to set and get the name and number of movies in the franchise. *Setters* and *getters* methods are defined for that purpose. They provide controlled access to the attributes of an object by encapsulating them within the class.

Setters (or setter methods) are used to set the value of an attribute. They allow you to validate the input or perform additional operations before assigning a value to the attribute. In the code, `set_franchise()` is a setter method that set the values of the name and movies attributes of the `MovieFranchise` class, respectively.

Getters (or getter methods) are used to retrieve the value of an attribute. They provide access to the attribute's value from outside the class. In the code, `get_franchise()` is a getter methods that return the values of the name and movies attributes of the `MovieFranchise` class, respectively.

The `main()` function creates three franchise objects and assigns the data for each one. In case a franchise has a negative number of movies, it is automatically corrected to have one. Afterwards, it displays the franchises as a table in the console with decorative separators. The table is formatted using spaces to align the columns properly.

Notice that in the `for` loop of the above code, the `franchise.get_franchise()` statement retrieves and displays the corresponding data for each franchise. It is in this sense that every object behaves according to the template, but each one in their own way.

Warning: Number of movies for 'Avatar' franchise must be greater than 0.

----- Movie Franchises -----	
Name	Movies

Harry Potter	8
Star Wars	9
Avatar	1

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/4ab5dd89>

- **Constructors:** Constructors are special methods that are automatically called when an object is created. They initialise the object's attributes or perform any setup required. Let's create a class called `Character` with a constructor to initialise their name and the movie they acted in.

```

1 import random
2
3 class Character:
4     # The constructor method
5     def __init__(self, name, movie):
6         self.name = name
7         self.movie = movie
8
9     # A method to introduce the Character to the user
10    def introduce(self):
11        greetings = ["Hi, there!", "What's up?", "Hey!", "Yo!", "Howdy!", "Hello!"]
12        greeting = random.choice(greetings)
13        print(f"{greeting} I'm {self.name} from the movie {self.movie}!")
14
15 def main():
16     # Creating many instances of Character
17     dorothy = Character("Dorothy", "The Wizard of Oz")
18     james = Character("James Bond", "007 Series")
19     sarah = Character("Sarah Connor", "Terminator")
20     indy = Character("Indiana Jones", "Indiana Jones")
21     leia = Character("Princess Leia", "Star Wars")
22
23     # Show how all instances can introduce themselves
24     characters = [dorothy, james, sarah, indy, leia]
25     for character in characters:
26         character.introduce()
27
28 # Call the main function
29 main()
30

```

The `__init__` method is used to define the constructor, which takes `self`, `name`, and `movie` as parameters. Within the constructor, the attributes `name` and `movie` are assigned the values passed as parameters, using the reserved keyword `self` which references the object being created. This ensures that each instance of the `Character` class has its own unique `name` and `movie`. It is common practice to use the same names for the parameters and the attributes, although they can be named differently.

In addition, the class implements the `introduce()` method that randomly selects a different greeting from the `greetings` list and uses it to introduce the character. This dynamic behaviour brings an element of surprise and liveliness to the character's interactions with the user.

Furthermore, the creation of character objects is done within the `main()` function. Each object variable is defined, and the attributes are passed as parameters when calling the constructor. Subsequently, a list of characters is created, and a loop is used to perform their introductions:

```
Howdy! I'm Dorothy from the movie The Wizard of Oz!
Hi, there! I'm James Bond from the movie 007 Series!
What's up? I'm Sarah Connor from the movie Terminator!
Hey! I'm Indiana Jones from the movie Indiana Jones!
Hello! I'm Princess Leia from the movie Star Wars!
```

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/mt8fbzrw>

- ▶ **Attributes:** Attributes are characteristics or properties of an object. They hold the data that determines the state of an object at a particular point during the execution of a program. In computational terms, attributes can be thought of as variables associated with an object. For example, let's define attributes for an Actor class:

```
1 class Actor:
2     def __init__(self, name, country, oscar):
3         # Non-mutable attributes
4         self.name = name
5         self.country = country
6         # Mutable attribute
7         self.oscar = oscar
8
9     def win_oscar(self):
10        if not self.oscar:
11            self.oscar = True
12            print(f"{self.name} has won an Oscar!")
13        else:
14            print(f"{self.name} already has an Oscar.")
15
16    def __str__(self):
17        return f"{self.name:20}\t{self.country:10}\t{'Yes' if self.oscar else 'No'}"
18
```

In this code, the attributes `name` and `country` are considered non-mutable because it doesn't make sense for an actor's name or country to change once it has been set in the constructor. These attributes are set once during object creation and remain constant throughout the object's lifetime.

On the other hand, the attribute `oscar` is mutable because it represents the actor's Oscar status, which can be changed from `False` to `True` when the actor wins an Oscar. The `win_oscar()` method allows this attribute to be modified, but it cannot be changed back to `False` since it is not possible for an actor to be stripped away of their award once it has been granted.

Lastly, notice that this class overwrites the built-in method `__str__()` to provide a string representation of the `Actor` object, which is suitable for a table format printout. The `__str__()` method is called when the object needs to be converted to a string, such as when printing or using string formatting. By defining this method, we can customise the output format of the `Actor` object, in this case, to display the name, country, and Oscar status in a formatted table-like structure.

So now we can write the `main()` function to create a list of actors, display their information in a table, update the Oscar status for some actors, and print the updated table:

```

1 def main():
2     actors = [
3         Actor("Meryl Streep", "United States", True),
4         Actor("Daniel Day-Lewis", "United Kingdom", True),
5         Actor("Cate Blanchett", "Australia", True),
6         Actor("Leonardo DiCaprio", "United States", False),
7         Actor("Anthony Hopkins", "United Kingdom", True),
8         Actor("Kate Winslet", "United Kingdom", True),
9         Actor("Denzel Washington", "United States", True),
10        Actor("Julianne Moore", "United States", True),
11        Actor("Tom Hanks", "United States", False),
12        Actor("Natalie Portman", "Israel", False),
13    ]
14
15    print("\n===== Actors' Information =====")
16    print(f"{'Name':20}\t{'Country':10}\t{'Oscar?':5}")
17    print("-" * 45)
18    for actor in actors:
19        print(actor)
20
21    print("\n===== Updating Oscar Status =====")
22    actors[3].win_oscar()
23    actors[4].win_oscar()
24    actors[8].win_oscar()
25
26

```

```
27 print("\n===== Updated Actors' Information =====")  
28 print("Name\tCountry\tOscar?")  
29 print("*"*45)  
30 for actor in actors:  
31     print(actor)
```

```
===== Actors' Information =====  
Name          Country      Oscar?  
=====  
Meryl Streep    United States Yes  
Daniel Day-Lewis United Kingdom Yes  
Cate Blanchett  Australia    Yes  
Leonardo DiCaprio United States No  
Anthony Hopkins United Kingdom Yes  
Kate Winslet    United Kingdom Yes  
Denzel Washington United States Yes  
Julianne Moore  United States Yes  
Tom Hanks       United States No  
Natalie Portman Israel        No  
  
===== Updating Oscar Status =====  
Leonardo DiCaprio has won an Oscar!  
Anthony Hopkins already has an Oscar.  
Tom Hanks has won an Oscar!
```

```
===== Updated Actors' Information =====  
Name          Country      Oscar?  
=====  
Meryl Streep    United States Yes  
Daniel Day-Lewis United Kingdom Yes  
Cate Blanchett  Australia    Yes  
Leonardo DiCaprio United States Yes  
Anthony Hopkins United Kingdom Yes  
Kate Winslet    United Kingdom Yes  
Denzel Washington United States Yes  
Julianne Moore  United States Yes  
Tom Hanks       United States Yes  
Natalie Portman Israel        No
```

- **Methods:** Methods are functions defined within a class that perform specific actions or operations on objects, utilising their own attributes or external data provided by other objects or functions through input parameters. They can also return data as a result of their execution. Computationally speaking, methods in object-oriented programming encapsulate the algorithmic logic that defines the behaviour of an object.

To illustrate this concept, let's create a class called `Movie` that includes methods to process its attributes and provide useful information to the user. These methods perform operations on the attributes of the `Movie` object to extract relevant information. For example, we can determine its duration, retrieve the list of Oscars awarded to the movie, and identify the century in which it was made.

```

1  class Movie:
2      # Movie constructor requires name, year, length, and optional list of oscars
3      def __init__(self, name, year, length, oscars=None):
4          self.name = name
5          self.year = year
6          self.length = length
7          self.oscars = oscars if oscars else [] # Set empty list if oscars==None
8
9      # Add an Oscar award to the movie. Non-return method with confirmation message
10     def add_oscar(self, award):
11         self.oscars.append(award)
12         print(f"Added {award} award to {self.name}")
13
14     # Return the list of Oscars as a comma-separated string
15     def get_oscar_list(self):
16         return ", ".join(self.oscars)
17
18     # Return the duration category: "Normal", "Long", or "Really long"
19     def get_duration(self):
20         if self.length < 90:
21             return "Normal" # < 90 minutes
22         elif self.length < 180:
23             return "Long" # 90-180 minutes
24         else:
25             return "Really long" # > 180 minutes
26
27     # Calculate the century based on the year and return as formatted string
28     def get_century(self):
29         century = (self.year // 100) + 1
30         return f"{century}th century"

```

```

31
32     # Return formatted string representation of the movie
33     def __str__(self):
34         return f"{self.name:<20}\t{self.get_century():<10}\t{self.get_duration()}"
35         :<15}\t{self.get_oscar_list()}"

```

The constructor `__init__` takes parameters such as `name`, `year`, `length` (in minutes), and a list of `oscars`. The `oscars` parameter has a default value of `None`, allowing it to be omitted when creating the object. If no value is provided for `oscars`, it is set to an empty list ensuring that the object is properly initialised with appropriate values, even if the list of oscars is not specified during object creation.

On the other hand, the `add_oscar` method adds an Oscar award to the movie by appending it to the `oscars` list attribute. It also prints a confirmation message to indicate that the award was added.

Furthermore, the class provides several getter methods that operate on the attributes of the movie object and return new information. The `get_oscar_list` method retrieves the list of Oscars as a comma-separated string by joining the elements of the `oscars` list attribute. This allows easy access to the awards the movie has received.

The `get_duration` method determines the duration category of the movie based on its length attribute. It categorizes movies as "Normal" if their length is less than 90 minutes, "Long" if it is between 90 and 180 minutes, and "Really long" if it exceeds 180 minutes. This provides a concise description of the movie's duration.

The `get_century` method calculates the century in which the movie was made based on its year attribute. It divides the year by 100, adds 1 to the result, and returns the century as a formatted string. This allows users to quickly identify the time period in which the movie belongs.

Lastly, the `__str__` method provides a formatted string representation of the movie object. It returns a string containing the movie's name, year, duration category and the list of Oscars. This string representation allows for easy printing or displaying of the movie's information.

In summary, the `Movie` class provides convenient methods to interact with the attributes of a movie object and obtain additional information derived from those attributes. We can now write a `main()` function to use this class to create several `Movie` objects and display some useful information about them.

```

1 def main():
2
3     # The lists of Oscars are just illustrative, not the real ones

```

```

4     movies = [
5         Movie("Titanic", 1997, 194, ["Best Picture", "Best Actor"]),
6         Movie("The Lord of the Rings", 2003, 201, ["Best Picture"]),
7         Movie("Ben-Hur", 1959, 212, ["Best Music", "Best Actress"]),
8         Movie("Gravity", 2013, 89, ["Best Actor", "Best Music"]),
9         Movie("Gone with the Wind", 1939, 238, ["Best Picture", "Best Music"]),
10        Movie("The Shape of Water", 2017, 123),
11    ]
12
13    print("Most Awarded Movie List:")
14    print("-"*80)
15    print("Name\t\tPeriod\tDuration\t\tOscars")
16    print("-"*80)
17    for movie in movies:
18        print(movie)
19
20    print("\nModifying Awards:")
21    movies[5].add_oscar("Best Actor")
22    movies[5].add_oscar("Best Actress")
23
24    print("\nUpdated Movie List:")
25    print("-"*80)
26    print("Name\t\tPeriod\tDuration\t\tOscars")
27    print("-"*80)
28    for movie in movies:
29        print(movie)
30
31 # Call the main function
32 main()

```

The `main()` function creates a list of movie objects, each representing a different movie with attributes like name, year, length (in minutes), and an illustrative list of Oscars won. In the movies list, the movie "The Shape of Water" is initialised without an initial list of Oscars, so it uses the default value in the constructor, which is an empty list.

The code then proceeds to print a table of the most awarded movies. It iterates over the movies list and calls the `__str__` method of each `Movie` object to obtain the column-formatted string representation. Next, the code modifies the awards for "The Shape of Water" by calling the `add_oscar()` method on the corresponding `Movie` object. Two new Oscars, "Best Actor" and "Best Actress," are added to its list of Oscars.

Finally, the updated movie list is printed, showing the changes made to the Oscars for "The Shape of Water" and the rest of the movies.

Below is the output of this program:

Most Awarded Movie List:

Name	Period	Duration	Oscars	
Titanic	20th century	Really long	Best Picture, Best Actor	
The Lord of the Rings	21th century	Really long	Best Picture	
Ben-Hur	20th century	Really long	Best Music, Best Actress	
Gravity	21th century	Normal	Best Actor, Best Music	
Gone with the Wind	20th century	Really long	Best Picture, Best Music	
The Shape of Water	21th century	Long		

Modifying Awards:

Added Best Actor award to The Shape of Water

Added Best Actress award to The Shape of Water

Updated Movie List:

Name	Period	Duration	Oscars	
Titanic	20th century	Really long	Best Picture, Best Actor	
The Lord of the Rings	21th century	Really long	Best Picture	
Ben-Hur	20th century	Really long	Best Music, Best Actress	
Gravity	21th century	Normal	Best Actor, Best Music	
Gone with the Wind	20th century	Really long	Best Picture, Best Music	
The Shape of Water	21th century	Long	Best Actor, Best Actress	

↗ Visualise a step-by-step execution of this code on PythonTutor: <https://tinyurl.com/3pse4vrt>

These examples demonstrate the use of classes as templates with attributes, methods, and constructors in the context of classical movies. By harnessing the power of classes and object-oriented programming, you can unleash your creativity and embark on an exciting journey into a world full of possibilities!

6.4 Hands-on time!

Suggest some fun and engaging coding exercises for practising functions, return values, parameters, and lambda expressions. Each exercise should be provided with a problem statement and examples of expected output. You can incorporate elements of math or randomness, requiring to import relevant libraries to solve the problems.⁹⁸

- 1. Dice Simulator:** Create a function that simulates rolling a dice. The function should take the number of dice rolls as a parameter and return a list of the outcomes. Example:

```
Input: 5
Output: [3, 6, 1, 4, 2]
```

- 2. Fibonacci Sequence:** Write a function that generates the Fibonacci sequence up to a specified number of terms. The function should take the number of terms as a parameter and return a list of the sequence. Example:

```
Input: 8
Output: [0, 1, 1, 2, 3, 5, 8, 13]
```

- 3. Prime Number Checker:** Implement a function that checks if a given number is prime. The function should take the number as a parameter and return True if it is prime, and False otherwise. Example:

```
Input: 17
Output: True
```

- 4. Palindrome Checker:** Create a function that determines if a given word is a palindrome. The function should take the word as a parameter and return True if it is a palindrome, and False otherwise. Example:

```
Input: "radar"
Output: True
```

- 5. Circle Area Calculator:** Write a function that calculates the area of a circle given its radius. The function should take the radius as a parameter and return the calculated area. Example:

```
Input: 5
Output: 78.54
```

- 6. Sum of Squares:** Create a lambda expression that calculates the sum of squares of a list of numbers. The lambda should take a list of numbers as a parameter and return the sum of their squares. Example:

```
Input: [1, 2, 3, 4, 5]
Output: 55
```

- 7. Word Frequency Counter:** Write a function that counts the frequency of each word in a given sentence, regardless of case. The function should take the sentence as a parameter and return a dictionary where the keys are the words and the values are the frequencies. Implement a menu function that prompts the user to enter a sentence and calls the word frequency counter function. It should display the

resulting dictionary of word frequencies. Create a `main` function that calls the menu function in a loop, allowing the user to execute the word frequency counter multiple times until choosing to exit.

Example of a function call:

```
Input: "The cat in the hat"
Output: {'the': 2, 'cat': 1, 'in': 1, 'hat': 1}
```

8. **Reverse String:** Implement a function that reverses a given string. The function should take the string as a parameter and return the reversed string. Create a menu function that prompts the user to enter a string and calls the reverse string function. It should display the reversed string. Write a `main` function that calls the menu function in a loop, allowing the user to reverse strings multiple times until choosing to exit.

Example of a function call:

```
Input: "Hello, world!"
Output: "!dlrow ,olleH"
```

9. **Unique Elements:** Create a function that takes a list of numbers as input and returns a new list containing only the unique elements in the original list, in the same order. Develop a menu function that prompts the user to enter a list of numbers and calls the unique elements function. It should display the resulting list of unique elements. Design a `main` function that calls the menu function in a loop, allowing the user to find unique elements from different lists of numbers until choosing to exit.

Example of a function call:

```
Input: [1, 2, 3, 2, 4, 1, 5]
Output: [1, 2, 3, 4, 5]
```

10. **Average Grades:** Write a function that calculates the average grade of a student given a list of grades. The function should take the list of grades as a parameter and return the average as a floating-point number. Construct a menu function that prompts the user to enter a list of grades and calls the average grades function. It should display the calculated average grade. Develop a `main` function allowing the user to calculate average grades multiple times until choosing to exit.

Example of a function call:

```
Input: [85, 90, 92, 88, 95]
Output: 90.0
```

11. **Caesar Cipher:** Implement a function that encrypts a given message using the Caesar cipher technique. The function should take the message and the shift amount as parameters and return the encrypted message. Create a menu function that prompts the user to enter a message and a shift amount, and calls the Caesar cipher function.

It should display the encrypted message. Write a `main` function that calls the menu function in a loop, allowing the user to encrypt messages using the Caesar cipher multiple times until choosing to exit.

Example of a function call:

```
Input: ("Hello, world!", 3)
Output: "Khoor, zruog!"
```

- 12. Linear Equation Solver:** Create a function that solves a linear equation of the form $ax + b = 0$. The function should take the values of a and b as parameters and return the solution x . Implement a menu function that prompts the user to enter the values of a and b and calls the linear equation solver function. It should display the solution x . Write a `main` function that calls the menu function in a loop, allowing the user to solve linear equations multiple times until choosing to exit.

Example of a function call:

```
Input: (2, 4)
Output: -2.0
```

Ok. Now suggest some fun and engaging coding exercises for practising OOP, including classes, constructors, attributes and methods. Each exercise should be provided with a problem statement and examples of expected output. You can incorporate elements of math or randomness, requiring to import relevant libraries to solve the problems.⁹⁹

- 1. Student Grade Calculator:** Create a class called `Student` that represents a student's information including their name and a list of grades. Implement methods to calculate the average grade, determine if the student passed (average grade ≥ 60), and display the student's information. Additionally, write a `main` function that allows the user to interact with the `Student` objects through a menu. Example:

► Input:

```
1 student1 = Student("John Doe", [75, 80, 90, 65, 70])
2 student1.display_info() # Display student information
3 print("Average Grade:", student1.calculate_average_grade()) # Show avg. grade
4 print("Passed:", student1.has_passed()) # Check if student passed
5
```

► Output:

```
Student Name: John Doe
Grades: [75, 80, 90, 65, 70]
Average Grade: 76.0
Passed: False
```

► Menu:

1. Add Student
2. Display Student Information
3. Calculate Average Grade
4. Check Pass Status
5. Exit

Enter your choice:

2. **Bank Transaction Log:** Create a class called `BankAccount` that represents a bank account. The class should have attributes such as account number, account holder name, and balance. Implement methods to deposit and withdraw money from the account, as well as a method to display the transaction history (log) of the account. The transaction history should keep track of all deposits and withdrawals made on the account. Write a `main` function that allows the user to interact with the `BankAccount` object through a menu. Example:

► Input:

```

1 account1 = BankAccount("123456789", "John Doe", 1000.0)
2 account1.deposit(500.0) # Deposit $500
3 account1.withdraw(200.0) # Withdraw $200
4 account1.deposit(300.0) # Deposit $300
5 account1.display_transaction_history() # Display transaction history

```

► Output:

```

Transaction History:
-----
Deposit:    $500.0
Withdrawal: $200.0
Deposit:    $300.0

```

► Menu:

1. Deposit
2. Withdraw
3. Display Transaction History
4. Exit

Enter your choice:

3. Inventory Management System:

Create a class called `Product` that represents a product in an inventory management system. The class should have attributes such as product ID, name, quantity, and price. Implement methods to increase or decrease the quantity of a product, calculate the total value of the product (quantity × price), and display the product's information. Additionally, write a `main` function that allows the user to interact with the `Product` objects through a menu. Example:

► Input:

```

1 product1 = Product("P001", "iPhone", 10, 1000.0)
2 product1.increase_quantity(5) # Increase quantity by 5
3 product1.decrease_quantity(3) # Decrease quantity by 3
4 product1.display_info() # Display product information
5
6 print("Total Value:", product1.calculate_total_value()) # Calculate and print
    total value

```

► Output:

```

Product ID: P001
Name: iPhone
Quantity: 12
Price: $1000.0
Total Value: $12000.0

```

► Menu:

1. Add Product
2. Increase Quantity
3. Decrease Quantity
4. Calculate Total Value
5. Display Product Information
6. Exit

Enter your choice:

4. Movie Trivia Game:

You are tasked with creating a Movie Trivia game. The game should test the player's knowledge of popular movies by asking a series of multiple-choice questions. You need to code the necessary classes and implement a menu system to allow the user to play the game. Here are the requirements:

- a) Create a class called `Question` that represents a single trivia question. Each question should have the following attributes:

- ▶ `text`: The actual question to be asked.
 - ▶ `choices`: A list of multiple-choice options.
 - ▶ `correct`: The index of the correct answer in the choices list.
- b) Create a class called `TriviaGame` that manages the game flow. It should have the following methods:
- ▶ `__init__()`: Initialises the game, score and sets up the questions (create a set of at least 10 questions).
 - ▶ `show_menu()`: Displays the game menu options for the player.
 - ▶ `start_game()`: Starts the trivia game.
 - ▶ `ask_question(question)`: Asks a single question and checks the player's answer. The questions should be chosen at random and should not be repeated.
 - ▶ `show_score()`: Displays the player's current score.
 - ▶ `finish_game()`: Ends the game and displays the final score.
- c) Implement a menu system using the `main()` function to allow the player to interact with the game. The menu should have the following options:
- ▶ Start game: Begins the trivia game.
 - ▶ Show last score: Displays the score of the most recent game played during the execution, or "Play first!" if none was played.
 - ▶ Quit: Exits the game.

Example:

```
=====
Welcome to the Movie Trivia Game
=====

1. Start game
2. Show last score
3. Quit

Enter your choice: 1
=====

Question 1:
What movie won the Academy Award for Best Picture in 2020?

A. Parasite
B. Joker
C. 1917
D. The Irishman

Enter your answer: A

Correct!
```

Question 2:

Who played the character of Iron Man in the Marvel Cinematic Universe?

- A. Robert Downey Jr.
- B. Chris Evans
- C. Chris Hemsworth
- D. Mark Ruffalo

Enter your answer: B

Incorrect!

Question 3:

In the movie "The Shawshank Redemption," what is the name of the main character?

- A. Andy Dufresne
- B. Red Redding
- C. Ellis Boyd "Red" Redding
- D. Warden Samuel Norton

Enter your answer: C

Correct!

Your score: 2 out of 3

Thank you for playing the Movie Trivia Game!

=====

Welcome to the Movie Trivia Game

=====

- 1. Start game
- 2. Show last score
- 3. Quit

Enter your choice:

=====

7.1 Digging Further

It's been an exciting ride thus far! But, now that I've grasped the fundamentals of Python programming (data types, variables, console data input/output, operators, conditionals, loops, lists, tuples, strings, dictionaries, functions, classes and objects), where should I go next? Could you please guide me in the proper direction? Provide some lesson plans for digging further into various learning tracks so that I can become a Python superstar! (use a mindmap format such as PlantUML)¹⁰⁰.

Alright, my Python prodigy! Now that you've conquered the fundamentals of Python programming like a boss, it's time to take it up a notch and ascend to the next level of Python mastery! Prepare yourself for the exhilarating journey ahead, as I suggest to you mind-blowing lesson plans that will delve deep into specific Python learning tracks. Get ready to soar through the realms of web development, conquer the secrets of data science, unleash your powers in automation and scripting, dive into the realm of game development, embrace the magic of DevOps, unravel the mysteries of advanced Python concepts, master the arts of testing and debugging, and tap into a treasure trove of additional resources. With these lessons under your belt, you'll be a Python virtuoso, ready to dazzle the world with your coding wizardry! Let's go, Python rockstar!

1. Python Web Development:

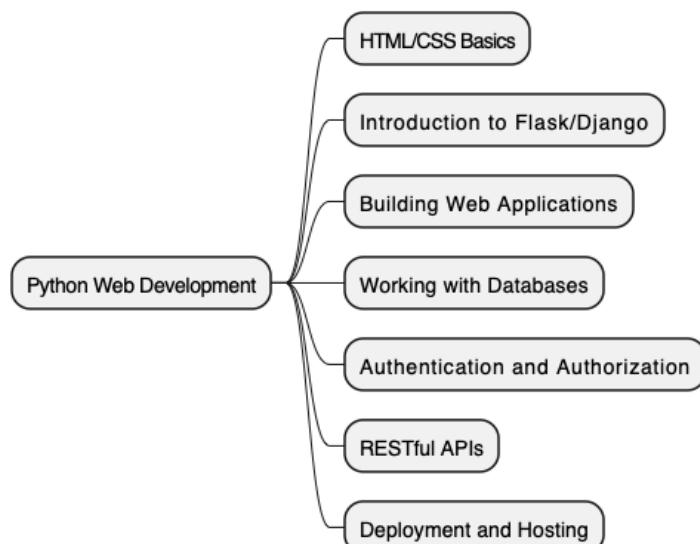


Figure 7.1: A suggested learning track for Python Web Development.

2. Data Science with Python:

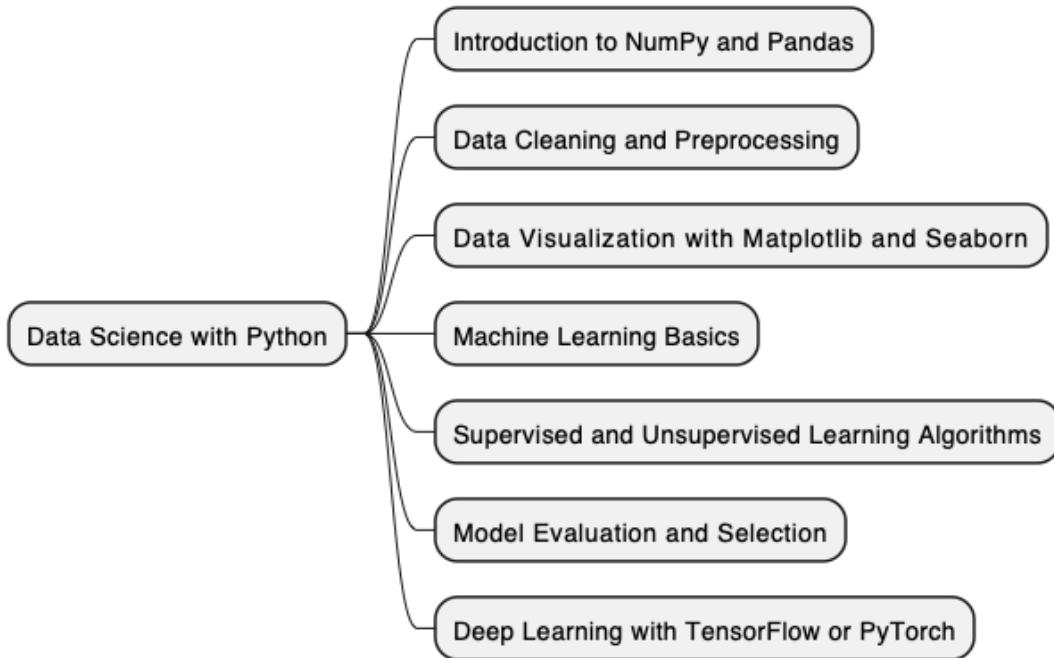


Figure 7.2: A suggested learning track for Data Science with Python.

3. Python for Automation and Scripting:

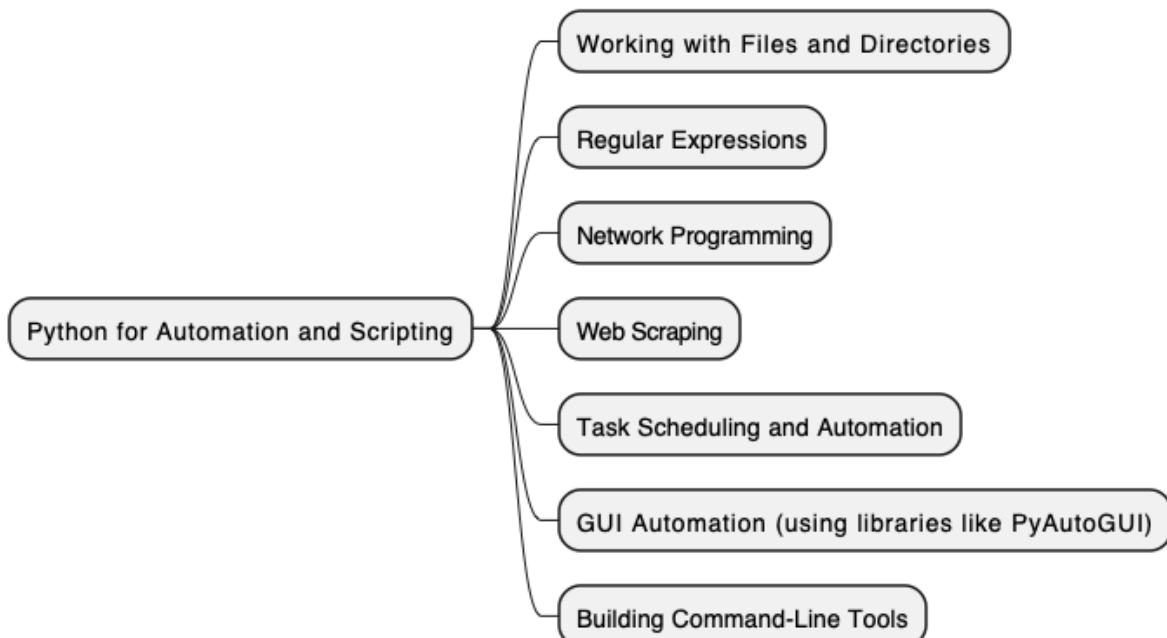


Figure 7.3: A suggested learning track for Python for Automation and Scripting.

4. Python for Game Development:

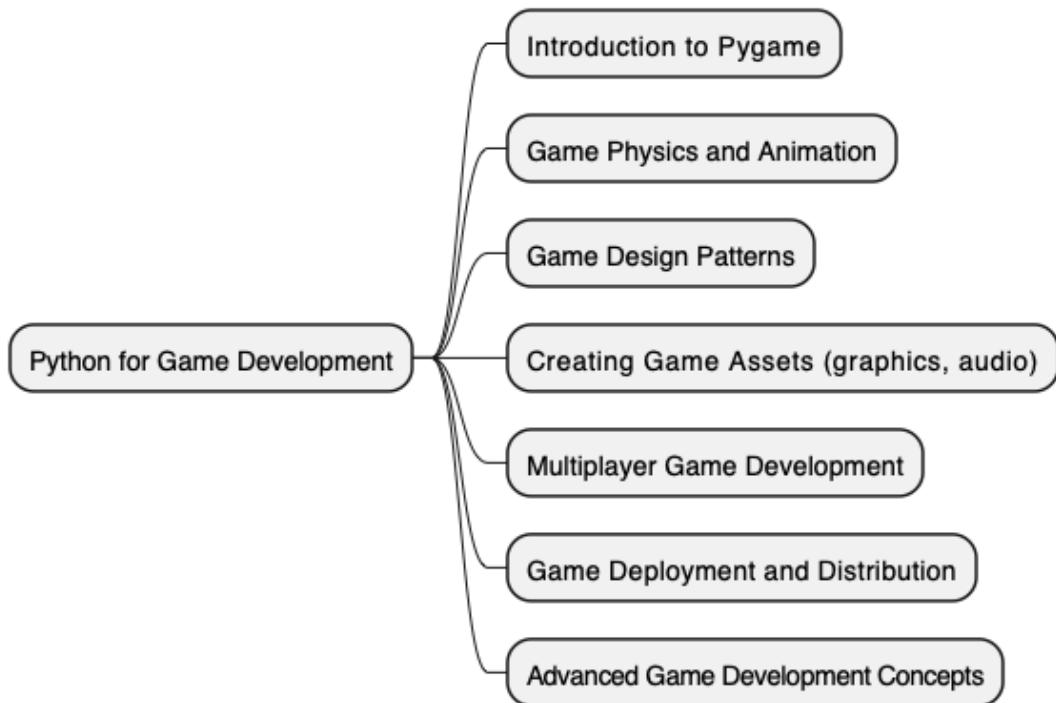


Figure 7.4: A suggested learning track for Python for Game Development.

5. Python for DevOps:

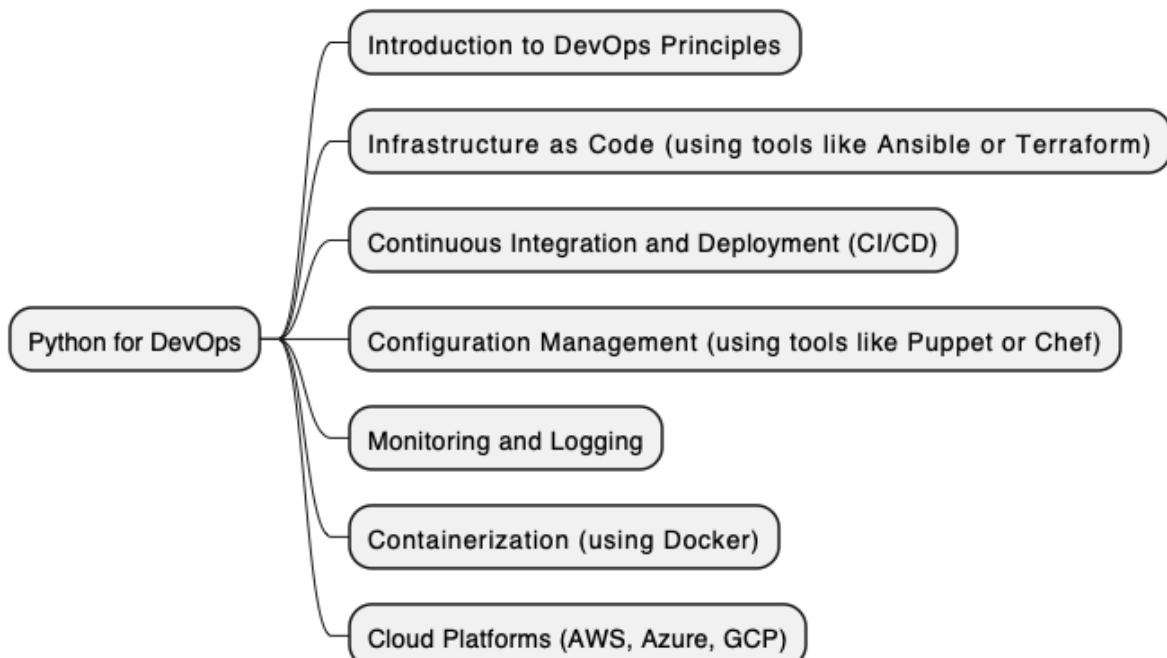


Figure 7.5: A suggested learning track for Python for DevOps.

6. Advanced Python Concepts:

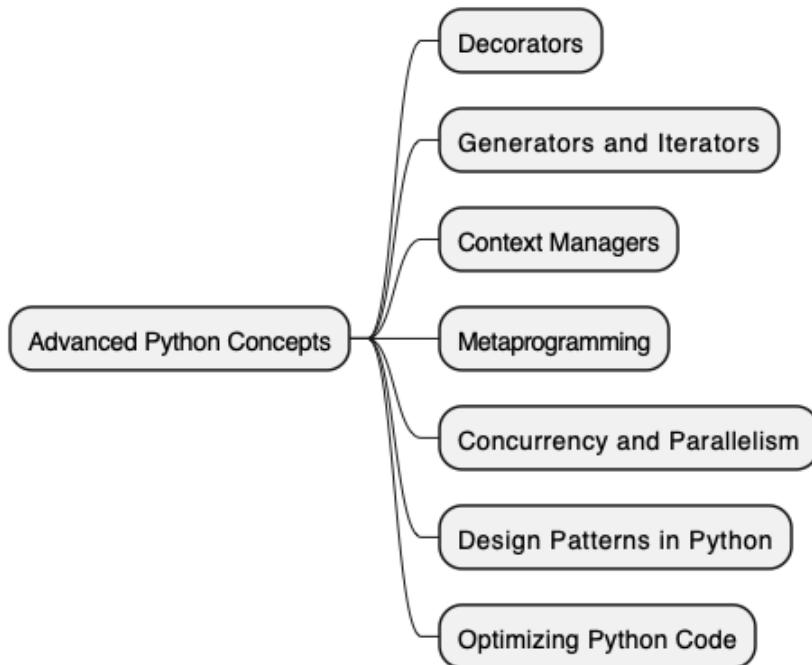


Figure 7.6: A suggested learning track for Advanced Python Concepts.

7. Testing and Debugging in Python:

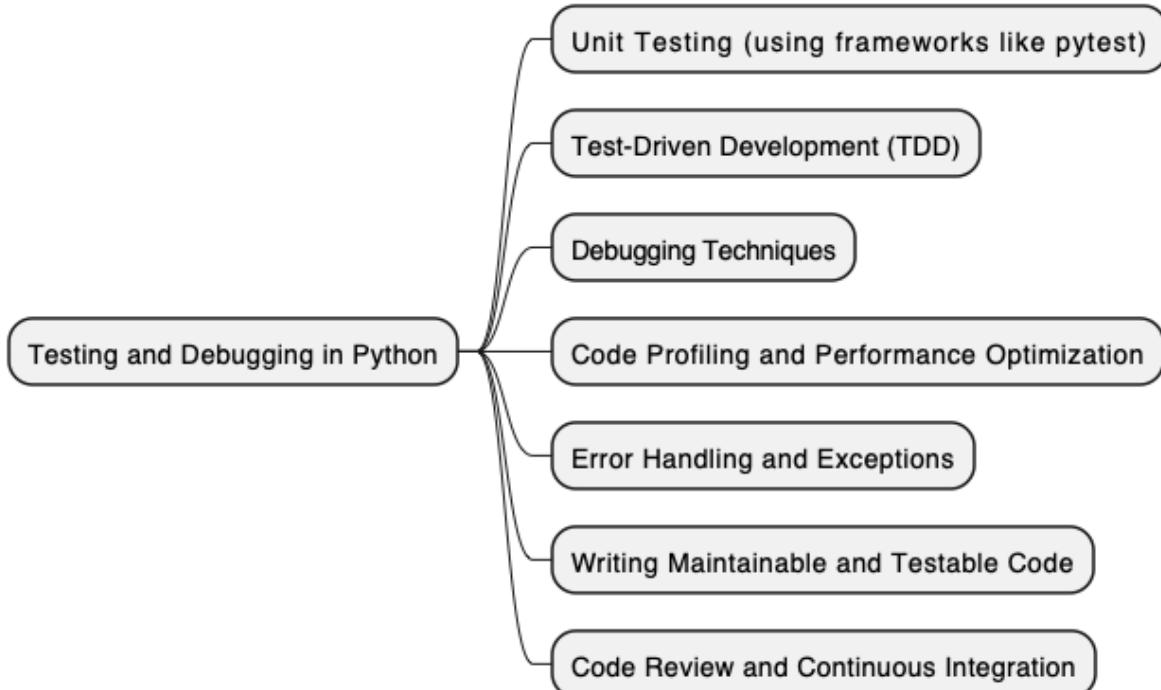


Figure 7.7: A suggested learning track for Testing and Debugging in Python.

8. Additional Resources:

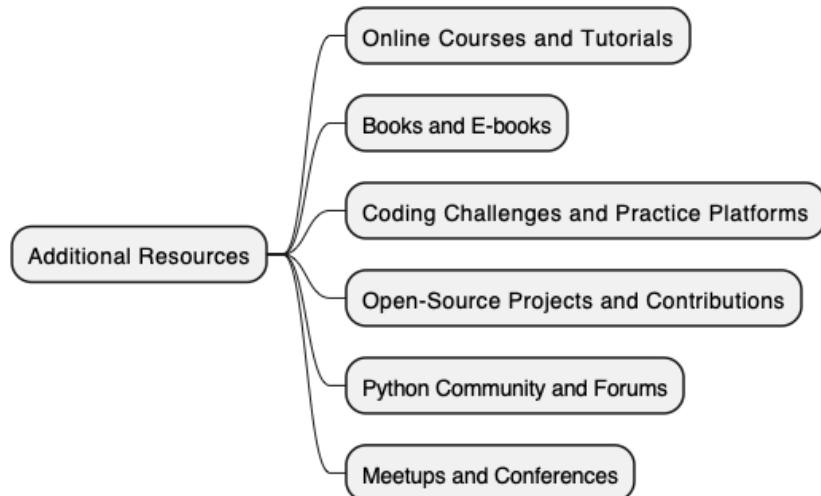


Figure 7.8: A suggested learning track for Additional Resources for Python.

NB. For your information, the output generated by ChatGPT to the previous prompt is shown below. This output was then imported into draw.io as PlantUML input to generate the mindmap diagrams.

```

@startmindmap
* Python Learning Tracks
** Python Web Development
*** HTML/CSS Basics
*** Introduction to Flask/Django
*** Building Web Applications
*** Working with Databases
*** Authentication and Authorization
*** RESTful APIs
*** Deployment and Hosting

** Data Science with Python
*** Introduction to NumPy and Pandas
*** Data Cleaning and Preprocessing
*** Data Visualization with Matplotlib and Seaborn
*** Machine Learning Basics
*** Supervised and Unsupervised Learning Algorithms
*** Model Evaluation and Selection
*** Deep Learning with TensorFlow or PyTorch

** Python for Automation and Scripting
*** Working with Files and Directories
*** Regular Expressions
*** Network Programming
*** Web Scraping
*** Task Scheduling and Automation
*** GUI Automation (using libraries like PyAutoGUI)
*** Building Command-Line Tools
  
```

```
** Python for Game Development
*** Introduction to Pygame
*** Game Physics and Animation
*** Game Design Patterns
*** Creating Game Assets (graphics, audio)
*** Multiplayer Game Development
*** Game Deployment and Distribution
*** Advanced Game Development Concepts

** Python for DevOps
*** Introduction to DevOps Principles
*** Infrastructure as Code (using tools like Ansible or Terraform)
*** Continuous Integration and Deployment (CI/CD)
*** Configuration Management (using tools like Puppet or Chef)
*** Monitoring and Logging
*** Containerization (using Docker)
*** Cloud Platforms (AWS, Azure, GCP)

** Advanced Python Concepts
*** Decorators
*** Generators and Iterators
*** Context Managers
*** Metaprogramming
*** Concurrency and Parallelism
*** Design Patterns in Python
*** Optimizing Python Code

** Testing and Debugging in Python
*** Unit Testing (using frameworks like pytest)
*** Test-Driven Development (TDD)
*** Debugging Techniques
*** Code Profiling and Performance Optimization
*** Error Handling and Exceptions
*** Writing Maintainable and Testable Code
*** Code Review and Continuous Integration

** Additional Resources
*** Online Courses and Tutorials
*** Books and E-books
*** Coding Challenges and Practice Platforms
*** Open-Source Projects and Contributions
*** Python Community and Forums
*** Meetups and Conferences
@endmindmap
```

7.2 Suggested Readings

1. Python.org. (n.d.). Python documentation. <https://www.python.org/doc/>
2. Downey, A. (2015). Think Python: How to Think Like a Computer Scientist. O'Reilly Media, Inc. <https://greenteapress.com/wp/think-python-2e/>
3. Sweigart, A. (2019). Automate the Boring Stuff with Python: Practical Programming for Total Beginners. No Starch Press. <https://automatetheboringstuff.com/>
4. GeeksforGeeks. (2023). Python Tutorial. <https://www.geeksforgeeks.org/python-programming-language/>
5. Real Python (n.d.). <https://realpython.com/>
6. Parlante, N. (2022). Google's Python Class. <https://developers.google.com/edu/python/>

The End

List of the 100 Prompts

1. "ChatGPT: Your Python Coach. Mastering the Essentials in 100 Prompts" is the title of a book I'm writing to teach the fundamentals of Python. Provide a draft version of an engaging and thought-provoking Preface for it. Emphasise the impact AI, and language models in particular, are having in many areas, including programming, and how it is expected to leverage programmers' abilities and productivity to create powerful applications in the near future, mentioning in passing that the issues regarding fake content and hallucination that have been detected on these language models are alleviated in the area of programming because the generated code can be validated by simply running it. Comment also that the initial generated content has been carefully revised and enriched based on my educational expertise.
2. Explain with a colourful tone that the purpose of this book is to initiate a newcomer to the world of programming, in particular, to the Python language, and to help them learn the basic concepts with examples, explanations, and feedback, just like a teacher guides a student through their learning journey. Only that here, that teacher is ChatGPT. Emphasise the new possibilities and advantages of using modern developments in AI to assist the learning and development process of programmers.
3. Provide a witty description of the Python programming language that is suitable for a first-year college student. Make some notes on the language's history and current version, speculate on its future, and provide useful links to organisations or websites where the reader can obtain more technical information.
4. Write a short note on Monty Python.
5. Briefly describe the key differences between Python 2.x and Python 3.x versions. Highlight the latest version's advantages.
6. Create a scheme of the standard cycle of program development, using Latex smartdiagram package with the circular diagram type.
7. Explain to a first-year college reader the standard cycle of program development, including the steps of writing (coding), running, testing, and correcting (debugging) (do it with a bulleted list).
8. Clarify that Python is an interpreted computer language; explain how it differs from a compiled language (advantages and weaknesses). Mention that there are a variety of IDEs available, but in this book, rather than dealing with software installation hassles, I will rely on ready-to-use environments that meet the needs of this book, so we can focus on concepts and algorithms. Jupiter Notebooks, Colaboratory, PythonTutor, and ChatGPT will be used as tools. Make a bullet list of these tools, including a brief description, website, and the steps required to open an account if one is required.
9. Describe in detail how to create a new Jupiter Notebook in Google Colab.
10. Explain the two types of cells in Jupyter Notebook.
11. Explain in detail how to use PythonTutor.
12. Explain to a first-year college student what ChatGPT is, how it can help students learn programming, how to use it responsibly to achieve good learning goals, and whether there are any ethical concerns about using it or relying too heavily on it and how to address them.

13. Propose prompt templates for how a programming student can ask ChatGPT so they can get an accurate and engaging response to the following topics: 1) Introduce and explain programming concepts in an approachable manner. 2) Create code examples that are related to those concepts. 3) Describe the key lines of such code. 4) Write down documentation and comments in the code. 5) Based on the example, come up with new exercises for the student to practice in kind of variations of the solved problem. 6) Include useful links to resources that supplement the previous topics covered. 7) Make fun and witty comments to make the topic more enjoyable.
14. Explain the concept of the 'Hello, World!' program in a simple but colorful manner, without using any code. Additionally, include a relatable anecdote to provide beginners in programming with a warm introduction to the topic.
15. But why the message "Hello World"? Who invented it? What's the meaning?
16. Write the classic "Hello, World!" program in Python and explain it. Provide the output as well. I will be using Colab.
17. Explain what a syntax error is and demonstrate 3 possible syntax errors in the 'Hello, World!' program with their respective outputs.
18. Why is it called print()? Do I need a printer to use it?
19. Explain in a simple way what a string is, and the syntax rules for defining strings in Python
20. Explain string formatting in simple terms. Write fun code examples for each of the formatting options.
21. Why are there some green lines in the code that begin with a hash symbol?
22. Provide an example of a program using only print() commands with a real-world analogy.
23. Can I use formatting options such as width and alignment to build strings? Show me engaging examples.
24. What do you mean by "floating-point number"?
25. I'd like to put what I've just learned about print() and string formatting into practice. Please provide four code exercises where I need to fix the errors to make them run correctly.
26. I'd like to put what I've just learned about print() and string formatting into practice. This time, suggest some exercises for me to write the code that can be solved using only print() and string formatting.
27. Mention in detail that one interesting feature of ChatGPT is that you can ask it to edit a piece of code, correct it, improve it, or provide feedback.
28. Create a simple prompt template for revise/improve/correct a given code.
29. Below is the code I wrote to solve the exercise: "Create a program that prints a right triangle using the '*' character.". Please revise/improve/correct my code and provide feedback on how to make it better, more efficient, or more readable. Also, demonstrate how you would have coded the solution differently, if applicable.
Code: `print('*') print('*' * 2) print('*' * 3) print('*' * 4) print('*' * 5)`

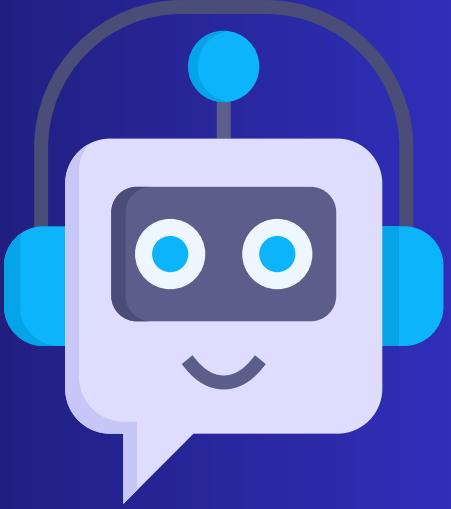
30. What exactly is a program? How does it relate to data?
31. Explain with examples and outputs how data is represented in Python. Provide examples from comics characters to better understand the concept.
32. So, how can I know the type of a variable at a given point in the program?
33. How can a program get different types of input data from the user? Use examples from comic books.
34. What are those if/else statements that you used in some of the previous examples? I can understand the basic idea, but can you provide a clear and concise explanation?
35. Hi there, ChatGPT! I'd like to practice my coding skills with an exercise that involves user input, data type conversion, and printing output, maybe even including a menu of options. Could you please suggest an exercise for me to work on? Once I have the statement, I'd also appreciate it if you could show me an example of the expected output. Thanks!
36. Ok, now show me the solution.
37. Cool. My version of the solution is shown below. What are your thoughts on it? In comparison to yours?
38. But I took care to conceal the credit card number. I was just joking around.
39. By the way, I like how you commented each section of your code; it helps to clarify things. Please do the same with my code.
40. Provide four code exercises related to data type input and conversion that I need to correct for syntax or logic errors so that they can run properly.
41. Suggest some exercises for me to write code that can be solved using only user input and data type conversion operations.
42. List all of the Python operators and provide a brief description for each.
43. Explain the details of arithmetic operators in Python in a colourful tone and include a single code example to illustrate the use of all the operators. The example should compare entertaining statistics about Marvel vs X-men movies and box office.
44. What do those numbers 22.55e9 and 6.01e9 for the box_office variables mean?
45. Explain the details of comparison operators in Python in a colourful tone and include a single code example to illustrate the use of all of them, maybe using fun facts about football teams.
46. Explain the concept of logical operators in Python in a casual tone. After defining each operator, include a single code example to illustrate the usage of all of them. Use a table of fast food dishes indicating how suitable are for vegetarians and vegans, for this purpose.
47. Explain the concept of bitwise operators in Python in a friendly but professional tone. Include a code example after defining each operator to demonstrate how to use them all. Show the results of the operations, including bit sequences and integers of the operands.

48. Okay, but could you please explain briefly what you mean by "binary digits of an integer". Give me some amusing real-life analogies.
49. Okay, one more thing: please explain what you mean by "two's complement form representation" in a brief and friendly manner.
50. Explain the concept of arithmetic assignment operators in Python in a playful manner.
51. Explain the concept of identity operators in Python in a playful manner.
52. Ok. Two questions. First, what exactly do you mean by "memory locations"? I don't see any memory locations in the code; please elaborate in a friendly and colourful manner. Second, what does the Python special variable "None" mean? Please clarify.
53. Explain the concept of memberships operators in Python in a playful manner.
54. Explain the concept of exponentiation operator in Python in a playful manner.
55. What happens when an expression has multiple operands? In what order are they evaluated?
56. How can I program more complex math operations such as square roots, logarithms, and so on?
57. Suggest some fun and informative coding exercises for practising arithmetic, comparison, logical, bitwise, membership, exponentiation, and math operators.
58. Propose a nice and engaging coding problem to practice binary representation of integer numbers. Assume you have a table of food dishes, with columns representing the ingredients used to prepare them and a final column displaying the dish's integer code as a result of interpreting the ingredient columns as bits of a binary number (show a table with four dishes and five ingredients). The program should ask for the name of a dish and a series of yes/no questions to determine the dish's integer code, and then print out a row of the table with this information (show an example of the program's output).
59. List all of the control structures in programming and provide a brief description for each.
60. Explain everything I need to know about if/else statements in Python in a fun and creative way, with playful examples.
61. Is there anything else I should know about conditionals in Python?
62. Now let's talk about match/case statements in Python. Explain everything I need to know about them in a fun and creative way, using playful examples.
63. Ok. In the match statement, I see you used a "lower()" command. Do I have to put it every time I code a match/case statement?
64. Please provide a brief technical definition of the for loop.
65. Now let's see the for loops in action in a fun and creative way, using playful examples.
66. Please provide a brief technical definition of the while loop.

67. Now let's see the while loops in action in a fun and creative way, using playful examples.
68. In the "Counting Sheep" example, you ask the user if they are sleeping twice. Isn't that redundant? Is there a way to write the loop without asking the same question twice?
69. Is there anything else I should know about for or while loops in Python?
70. What is a nested control structure and why is it useful?
71. Let us now dive into nesting control structures in Python. Show me how to use nested conditionals allowing for multiple levels of branching, nested loops including nested double for loops, double while loops and mixed while/for or for/while loops, as well as combinations of loops within conditionals and conditional within loops. Use examples from superheroes and comics to be creative and playful in your explanation.
72. Suggest some fun and engaging coding exercises for practising if/else, if/elif/else, and match/case statements, as well as the ternary operator, including problem descriptions and examples of expected output.
73. Suggest some fun and engaging coding exercises for practising for and while loops, including problem descriptions and examples of expected output.
74. Suggest some fun and engaging coding exercises to practice nested loop structures, including problem descriptions and examples of expected output.
75. Suggest the problem of writing a console-based calculator program that takes two numbers and a selected operation from a menu (decorated with a frame), computes and displays the result, and asks the user if they want to continue. Include an example of the expected output.
76. Explain what a container structure is, then list all of the container structures available in Python and provide a brief description for each, using real-life analogies and a friendly tone.
77. Explain everything I need to know about lists in Python in a fun and creative way, using playful examples.
78. Ok. Show me a code example that illustrates each of the items listed above. Include comments in each code block to explain its purpose. You are welcome to use the superhero vs. villain theme.
79. Interesting. Now demonstrate how I can use loops and conditionals on lists. This time, use creative food-related examples, taking advantage of Python's randomness features.
80. Wait a minute. In the examples above, you use some neat tricks that I'm not aware of, such as: random.choice(), enumerate(), and zip(). Can you pls explain these to me?
81. Explain everything I need to know about tuples in Python in a fun and creative way, using playful examples.
82. Ok. Now let's look at some code samples to see tuples in action! This time, why not use rock bands to illustrate the items you listed previously?
83. Wait! There is a weird '*' before the "members" variable in the unpacking trick example. What exactly is that?
84. Explain everything I need to know about sets in Python in a fun and creative way, using playful examples.

85. Good. Now let's look at some code samples to see sets in action! This time, let's use planets fun facts to illustrate the items you listed previously.
86. Explain everything I need to know about strings in Python in a fun and creative way, using playful examples.
87. Good. Now let's look at some code examples to see strings in action! Consider using classic children's books to illustrate the items you listed previously.
88. Explain everything I need to know about dictionaries in Python in a fun and creative way, using playful examples.
89. Great. Now let's look at some code samples to see dictionaries in action! Let's stick to the superheroes theme to illustrate the items you listed previously.
90. What about putting together a superhero encyclopedia? Can I implement it as a dictionary of dictionaries? Some kind of nested dictionary? Use a code example to illustrate.
91. Nice! You also mentioned that dictionaries enable us to explore intricate relationships and connections between items within the dictionary. What does that mean? Can dictionaries help me determine relationships such as 'the enemies of my allies are my enemies,' 'the allies of my allies are my friends,' 'the heroes with the same superpower as mine are my buddies,' or 'display a table of superheroes sorted by debut year'? Please provide some examples.
92. Suggest some fun and engaging coding exercises for practising lists, tuples, and sets. Each exercise should be provided with a problem statement and examples of expected output.
93. Suggest some fun and engaging coding exercises for practising with strings and dictionaries, including nested dictionaries. Each exercise should be provided with a problem statement and examples of expected output.
94. Explain briefly the concept of modularisation and reusability, as well as why they are useful in programming projects. Then, using real-world analogies, explain the various approaches for implementing these notions in a friendly but neat manner.
95. Wow! That's a impressive list of powerful ideas. However, since I am still learning the ropes, please recommend one or two items from the list I should start with as a beginner, and then describe each item's key components in a relaxing and entertaining manner.
96. Ok. So let's start with some code examples to see functions in action! Consider using social networking-related ideas to illustrate the items you listed previously. Feel free to use entertaining Harry Potter trivia and random features.
97. Ok. So let's continue with some code examples to see classes in action! Consider using classic movies and characters trivia to illustrate the items you listed previously, including random features.
98. Suggest some fun and engaging coding exercises for practising functions, return values, parameters, and lambda expressions. Each exercise should be provided with a problem statement and examples of expected output. You can incorporate elements of math or randomness, requiring to import relevant libraries to solve the problems.

99. Ok. Now suggest some fun and engaging coding exercises for practising OOP, including classes, constructors, attributes and methods. Each exercise should be provided with a problem statement and examples of expected output. You can incorporate elements of math or randomness, requiring to import relevant libraries to solve the problems.
100. It's been an exciting ride thus far! But, now that I've grasped the fundamentals of Python programming (data types, variables, console data input/output, operators, conditionals, loops, lists, tuples, strings, dictionaries, functions, classes and objects), where should I go next? Could you please guide me in the proper direction? Provide some lesson plans for digging further into various learning tracks so that I can become a Python superstar! (use a mindmap format such as PlantUML)



This guide has been carefully crafted to help you navigate the basic concepts of Python programming with ease, with ChatGPT as your trusty mentor.

ChatGPT is a powerful language model designed to assist you with any questions or doubts you may have while learning Python. It will provide you with examples, explanations, and feedback, making your learning experience interactive and engaging.