Pythonic AI

A beginner's guide to building AI applications in Python

Arindam Banerjee

www.bpbonline.com

First published: 2024

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55515-919

www.bpbonline.com

Dedicated to

My pillars, my parents:
Ashim
&
Sipra Banerjee

About the Author

Arindam Banerjee has been working in software development for more than 13 years, playing central roles as a technical leader and software engineer. He has architected and deployed numerous data-driven AI solutions to cater to both the data and AI requirements of the business and to help organizations succeed in their data endeavors without getting caught up in the hype. Currently, he is a Senior AI consultant at Ernst & Young GDS. He holds a master's degree in Computer Science and Engineering from Vellore Institute of Technology, Vellore. He successfully got many certifications in AI technologies. Furthermore, the author participates as a speaker at international conferences and writes research papers on AI-related topics. He has filed nine patents till now.

About the Reviewer

Shreyas Kulkarni, originally from Maharashtra, India, is a passionate tech enthusiast with a wealth of experience in the dynamic field of Data Science. His journey spans a substantial duration, empowering him with extensive knowledge and expertise in addressing intricate data and technology challenges.

His forte is crafting ingenious solutions, drawing from a diverse skill set encompassing exploratory data analysis, model development, innovative experimentation with cutting-edge AI solutions, and managing the entire data lifecycle. Shreyas excels at simplifying intricate tasks, resulting in the efficient automation of complex processes. His expertise in code delivery ensures a seamless project implementation.

In addition to his professional endeavors, Shreyas maintains an engaging personal blog where he shares his insights, knowledge, and progress in data science through reflective writing and tutorials. This platform offers readers a unique perspective on learning data science effectively, making it a valuable resource for those navigating the ever-evolving tech landscape.

Shreyas actively engages with the tech community, generously sharing his knowledge and insights. He is driven by a deep commitment to responsible AI regulation and ethical technology practices, establishing him as a dedicated advocate for impactful and ethical innovation. His unique blend of skills and an unwavering passion for technology positions him as a valuable asset in any endeavor.

Acknowledgement

Writing a book is a journey that traverses both the realms of creativity and perseverance. This endeavor could not have been accomplished without the support, encouragement, and contributions of many individuals and entities, who have each played a significant role in bringing this work to realization.

First and foremost, I extend my heartfelt gratitude to my family for their unwavering support and encouragement throughout this book's writing. My wife Arismita's presence motivated and enabled me to thrive and grow more. My son Mimo's love sustained me throughout the writing process.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. I extend my appreciation to the dedicated reviewers, technical experts, and editors who contributed their time and expertise to reviewing and revising the manuscript. Your meticulous attention to detail and thoughtful suggestions have played a crucial role in refining the content and ensuring its quality.

I would also like to acknowledge the valuable contributions of my colleagues and co-workers during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Finally, I want to express my profound appreciation to the readers of this book. Your curiosity and engagement drive the pursuit of knowledge and the sharing of ideas, and it is for you that this work was undertaken.

Preface

In an era defined by data-driven innovation and transformative technologies, Artificial Intelligence (AI) holds unprecedented promise. This book is your key to unlocking that promise, catering specifically to those who are new to both Python and the realm of AI.

Imagine being able to harness the potential of AI right from the foundational level without prior experience. Whether you are a student eager to explore the cutting-edge field of AI, a professional venturing into new territories, or an enthusiast with a curiosity for what lies beyond the horizon – this book is crafted for you.

Our journey together will be dynamic and enriching. From the basics of Python programming to crafting intricate AI solutions for computer visions and natural language processing, this hands-on guide will accompany you every step. We understand that venturing into Python and AI simultaneously can be daunting, so we have meticulously structured the content to ease your transition into this exciting realm.

As we delve into the chapters, you will witness how Python, a language known for its simplicity and versatility, blends seamlessly with AI, a technology that is shaping industries and redefining possibilities. Throughout the book, you will learn about the key features of convolutional neural networks, sequence models, attention-based models, transformers, generative adversarial networks, and so on, and how to use them to build enterprise applications that are efficient, robust, and easy to maintain. You will also learn about best practices and will be provided with numerous practical examples to help you understand the concepts. The book does not assume any prior knowledge; instead, it empowers you with a clear understanding of foundational concepts, building your confidence to create AI applications with Python.

We are excited to introduce you to the captivating world of AI, from understanding the fundamental concepts to embarking on hands-on projects that illustrate real-world applications. Along the way, you will develop the technical skills to architect and deploy AI solutions and cultivate a mindset that thrives on problem-solving and innovation.

This book is a transformative journey that equips you with the tools to shape the future. So, brace yourself to embark on this adventure, where curiosity meets capability and Python meets AI.

Get ready to witness the fusion of two powerful forces – Python and AI – and to emerge as a creator, an innovator, and a trailblazer in the realm of technology. Let us dive in and unleash the potential of Pythonic AI together!

Chapter 1: Python Kickstart: Concepts, Libraries, and Coding – It covers the basics of Python, its data structures, and object-oriented design. The chapter explains popular Python libraries, such as NumPy, Pandas, Matplotlib, etc., through examples so that the reader can grasp the following chapters easily. These libraries are widely used in AI and Machine Learning applications and profoundly applied in the subsequent chapters along with TensorFlow 2.

Chapter 2: Setting up AI Lab – This chapter starts by introducing Google Colab, a valuable platform for Python coding that allows users to harness the cloud's capabilities and leverage GPUs without the need for dedicated infrastructure. Simultaneously, this chapter provides step-by-step guidance on establishing a local Anaconda environment, ensuring flexibility in your coding environment. It also explains the potential of Google Colab by teaching you how to seamlessly integrate it with your Google Drive and GitHub repositories, streamlining your workflow.

Chapter 3: Design My First Neural Network Model – focuses on harnessing the power of TensorFlow 2 APIs to craft deep learning models from the ground up, demonstrating the art of saving and loading TensorFlow models, and harnessing the visualization prowess of TensorBoard. This chapter unravels the fundamental concepts of Artificial Neural Networks (ANNs) and explores creating ANN models in TensorFlow 2 and Keras API, training, fine-tuning the architecture for optimal performance, and evaluating the models.

Chapter 4: Explore Designing CNN with TensorFlow – allows the reader to learn the captivating world of image classification, a prominent application of AI. This chapter covers the intricate workings of Convolutional Neural Networks (CNNs) with hands-on experience in constructing CNN models using TensorFlow 2. It explores implementing diverse CNN architectures and harnessing the power of pre-trained CNN models.

Chapter 5: Develop CNN-based Image Classifier Apps – It covers a hands-on journey to construct end-to-end Convolutional Neural Network (CNN) applications. This chapter focuses on crafting an artificial intelligence application capable of accurately identifying images using the CIFAR-10 dataset for training purposes. It explores both building a CNN model from scratch and leveraging pre-trained models for image classification.

Chapter 6: Train and Deploy Object Detection Models – It shows the pivotal realm of object detection, a fundamental task in the field of artificial intelligence. The chapter establishes a solid foundation in the basics of object detection, providing the intuitive understanding necessary to grasp its intricacies. It also explores the inner workings and differences between popular object detection algorithms such as SSD, RCNN, and YOLO. The chapter guides you through the implementation of object detection using pre-trained models within the TensorFlow API.

Chapter 7: Create a Text and Image Reader – It explains AI-powered text recognition or image-to-text applications. The chapter starts with a hands-on guide to utilizing Tesseract for Optical Character Recognition (OCR) applications and equips you with the knowledge and skills to build your own text-reading application. Moving forward, it ventures into the realm of deep learning, harnessing the power of TensorFlow 2 to create cutting-edge image-to-text applications.

Chapter 8: Explore NLP for Advanced Text Analysis – It begins the journey into the fascinating world of Natural Language Processing (NLP), a vital application of AI. This chapter navigates through the practical use of widely used Python libraries like Spacy and NLTK, empowering you to process raw, unstructured text with ease. Furthermore, it unlocks the power of word embeddings, demystifying the concept using GloVe to represent words as vectors. It introduces Word2Vec and guides you through the implementation in TensorFlow.

Chapter 9: Up and Running with Sequence Models – It embarks on a journey through the intricacies of Recurrent Neural Networks (RNNs), Bi-directional RNNs, Long Short-Term Memory (LSTM) models, and Gated Recurrent Units (GRUs) using the TensorFlow 2 framework. This chapter also introduces language modeling and guides you through the implementation.

Chapter 10: Using Sequence Models for Automated Text Classification – It explores how sequence models, specifically LSTM (Long Short-Term Memory) networks, can be harnessed to create a powerful AI application for automatic text classification. This chapter covers understanding the data, performing essential data cleaning and preprocessing, and crafting it into a format suitable for classification. Through hands-on experience, it explains building and training an LSTM model using TensorFlow 2, and alongside, also explores the implementation of a 1-dimensional CNN (Convolutional Neural Network) model.

Chapter 11: Create Attention and Transformer Models – It introduces Attention models, a pivotal concept in the realm of natural language processing (NLP). This chapter navigates through various facets of Attention techniques, including self-attention, bi-directional, and multi-head attention, illuminating their unique roles and building a custom Attention layer in TensorFlow 2. It then unravels the transformative potential of the Transformer network, which leverages Attention to turbocharge the training speed of models. It demystifies the deployment of pre-trained Transformer models in TensorFlow 2 for NLP tasks, offering practical insights into their implementation.

Chapter 12: Generating Captions for Images – It explores the image-to-text AI system to automatically generate descriptive and contextually relevant captions for given input images. This chapter builds the image captioning model consisting of the encoder, sequence decoder, attention, and caption generator components.

Chapter 13: Learn to Build GAN Models – It covers the fascinating world of generative modeling, providing a concise overview of both Discriminative and Generative models. This chapter introduces the Variational Encoder, a key concept in the realm of generative models. It also explains the construction of a powerful Generative Adversarial Network (GAN) model using Tensorflow 2.

Chapter 14: Generate Artificial Faces Using GAN – It covers the conditional Generative Adversarial Networks (cGANs) and their extraordinary ability to generate synthetic face images belonging to specific age categories. This chapter unveils the architecture of the conditional GAN and develops the model using TensorFlow 2 API.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/a3iau6c

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Pythonic-AI**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Python Kickstart: Concepts, Libraries, and Coding

Introduction

Python is crucial for developing AI applications due to its extensive libraries and frameworks. We need to have some basic knowledge of Python to get the best out of this book. This chapter will cover the basics of Python and popular libraries widely used in AI applications through examples. Throughout this chapter, we will use Google Colab notebook for Python coding so that examples can be run on the cloud without having our infrastructure. We will also cover popular Python libraries like NumPy and Matplotlib that will be heavily used in the subsequent chapters, along with TensorFlow.

Structure

In this chapter, we will cover the following topics:

- Introduction to Python
- Basic Python data structures
- Object-oriented design in Python
- NumPy
- Matplotlib

Objectives

By the end of this chapter, we will be able to understand how to write code in Python programming language. We will have a good idea of Python data structures and object-oriented programming in Python. This chapter also covers the hands-on implementation of important Python libraries such as NumPy and Matplotlib.

Introduction to Python

We will have a crash course on Python programming through this chapter. **Artificial Intelligence** (**AI**) technology enables computers to have the intelligence to perform human-level jobs efficiently. We know that computers can perform jobs faster than humans, and we need a language to communicate instructions about these jobs to computers. Programming language serves the purpose of communication language. We write the logic of the intended task to be performed in a human-readable code, and the programming language converts it into binary (0s and 1s) that computers understand. Some commonly used programming languages are Python, Java, C, C++, Javascript, R, Ruby, PHP, and so on.

Python is a general-purpose programming language, and it is used for a variety of applications such as machine learning, web development, game development, general software development, and so on. Python is a free, open-source programming language, and its open-source code is available online. It is a high-level programming language that provides strong abstraction from low-level computing details.

Python is an independent platform and is accepted and executable in all major operating systems. Python's core philosophy is to increase the readability of the code by using proper whitespaces called indentation. It is widely popular in developing AI applications because of the presence of its powerful libraries that support data manipulation and numerical computing in a scalable manner. Let us start learning Python before we jump into developing real-life AI applications.

Using Colab

We will use Google Colab for writing and executing Python code in this chapter. In *Chapter 2, Setting up AI Lab*, we will learn about Google Colab in detail. Hence, if you need help understanding Colab right now, there is no need to worry. We will get it covered in the upcoming chapters.

Colaboratory, or Colab, was developed by the Google research team that enables anyone to write and execute Python code from the browser without any local setup. The Colab setup is hosted in the cloud and is available as a free service. As of now, we need to open our computer's browser and go to **https://colab.research.google.com/**. We will see the following page:

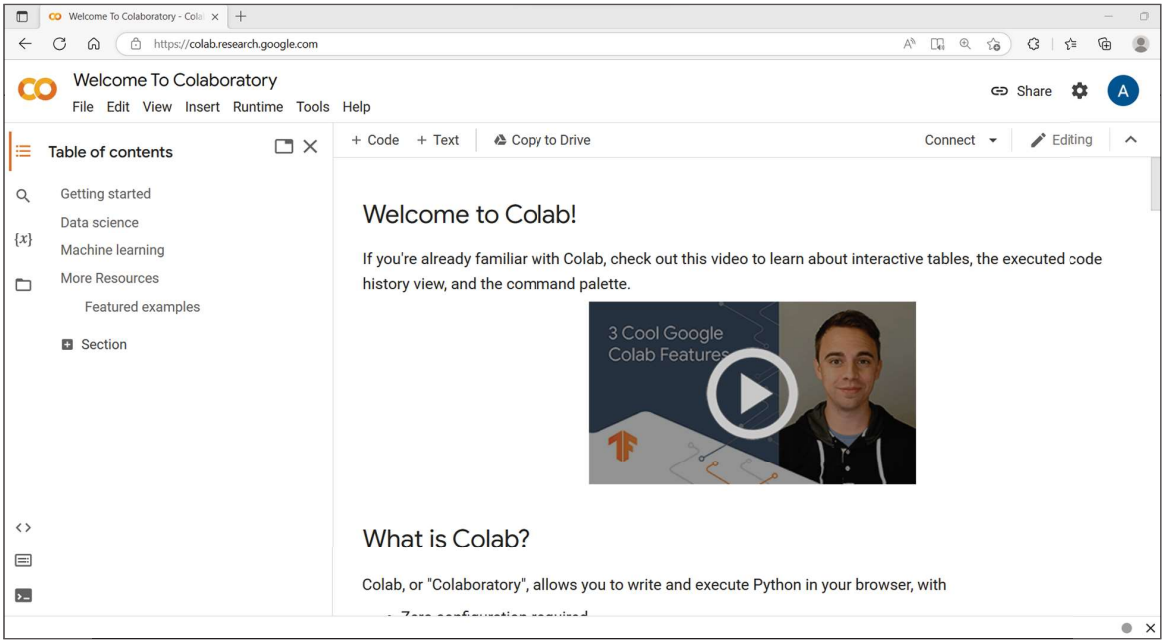

***Figure 1.1**: Colab welcome page*

Once the Google Colab page is opened, we will go to the **File** option towards the top-left corner and click **New Notebook** from the dropdown (as shown with the red arrow). Google Colab needs to be logged in using a Google account. If we are already logged in to our Google account in the browser, clicking on **New notebook** will open a new Colab Notebook. We need to sign in to our Google account, and a new notebook will open like the following figure:

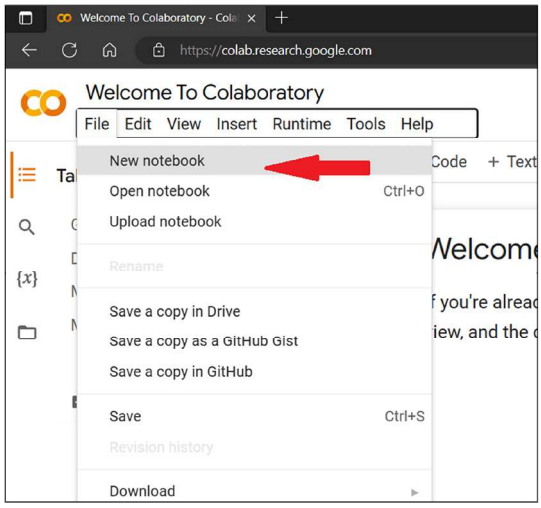

***Figure 1.2**: Create a new notebook*

As shown in the following figure, we will see the cursor blinking on the panel in the opened notebook. This is called a code cell, where we will write the code. On executing the code, if there is any output, it will appear following the code cell. Now, let us start coding Python in the Colab Notebook, as shown in *Figure 1.3*:

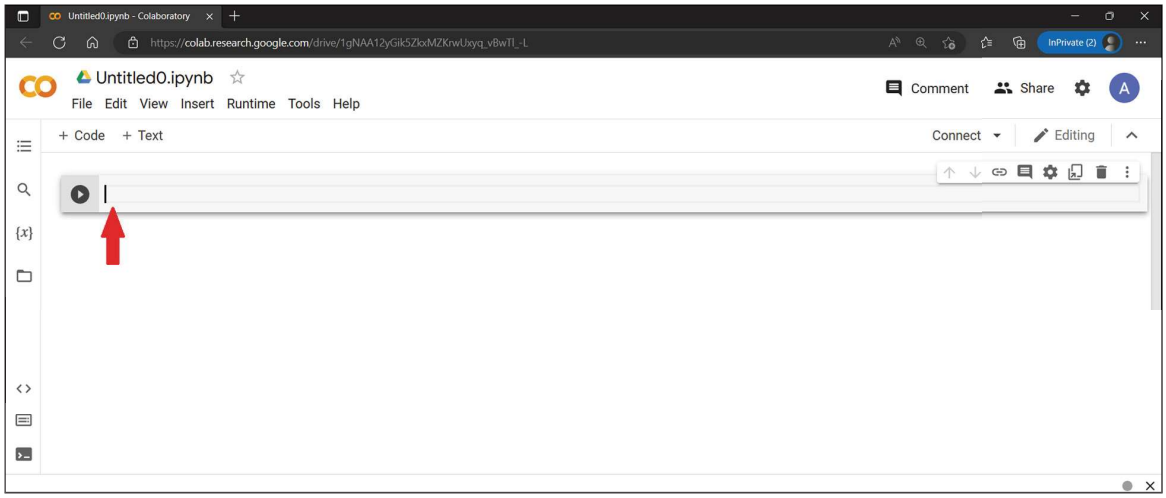

***Figure 1.3**: Code cell in Colab notebook*

Python variables

In a programming language, we need variables to store values. These values can be changed depending on the logic or instructions passed to the program. Unlike Java, C/C++, etc., we do not need to mention the variable type (such as integer, float, string, etc.) in Python. The variable type is automatically created when we assign the value to the variable. First, we create variables and print their values using Python's **`print()`** function. Refer to the following figure:

***Figure 1.4**: Variables*

The variable **`a`** is an integer, **`pi`** is a float, and **`st`** is a string. We write the code in the notebook's code cell and press *Shift+Enter* on the keyboard. That will execute the cell, and the output

will appear. Once the output appears, another new code cell is created, and the cursor starts blinking inside it to accept the next set of codes. Executing the cell may take some time. The number **[1]** written at the top left corner in *Figure 1.4* is the execution number and may not match yours.

While coding with any language, commenting is very important. Commenting increases readability and helps in testing the code. In Python, a single-line comment is created by adding a **#** sign before any line in the code. We can add triple double quotes (**"""**) and keep multi-line comments inside it. The commented part of the code stays in the code but is not executed by the Python interpreter, as shown in *Figure 1.5:*

```
[3]  # This is a single line comment
     """
     This is a multiline string
     and it can be used as a
     multiline comment.
     """
```

***Figure 1.5**: Comments in Python*

Note: Python does not allow starting any variable name with a number, and the name can contain only alpha-numeric characters and underscores.

Tip: The best practice in writing code is to create self-descriptive names for the variables. For example, if we want to create a variable to store temperature data, try to avoid creating it as "t" and instead use "temperature" as the variable name.

Python provides built-in data types for storing numerical values: int, float, and complex.

The int-type variables store integer numbers, the float-type variables store floating-point numbers, and complex-type variables store complex numbers. We can check the variable type using the **type()** function of Python. A string variable in Python is used to store text data. It can be created by assigning the value to the variable using single or double quotes, as shown in *Figure 1.6*:

```
[8]  a = 5
     pi = 3.14
     cn = 2+5j
     st = "Hello World!"
     print(type(a))
     print(type(pi))
     print(type(cn))
     print(type(st))

     <class 'int'>
     <class 'float'>
     <class 'complex'>
     <class 'str'>
```

***Figure 1.6**: The type() function*

Python supports working with Boolean-type variables. The Boolean variables can store only two possible values that are **`True`** and **`False`**, as shown in *Figure 1.7*:

```
[11] flag_1 = True
     flag_2 = False
     print(type(flag_1))
     print(type(flag_2))

     <class 'bool'>
     <class 'bool'>
```

***Figure 1.7**: The Boolean variables*

We can also exclusively define the variable type by using the related function. This process is called casting, as shown in *Figure 1.8*:

```
✓ 0s [13] a = 2
          print(a) # value of a is 2
          print(type(a)) # a is an integer
          b = float(a) # casting variable a to float-type and creating the variable b
          print(b) # value of b is 2.0
          print(type(b)) # b is a float

          2
          <class 'int'>
          2.0
          <class 'float'>
```

***Figure 1.8**: Casting*

Note: Python variables are case-sensitive. Once a variable is created, we cannot change the case in any part of the name of that variable.

Indentation

Indentation (the white space at the beginning of a line) is crucial in writing Python code. Coding readability by indentation is the core Python philosophy. The indentation is used to define the scope of a block of code in Python, and if not properly given, the code will either run the wrong logic or will not run at all.

Python operators

Python operators are different symbols and keywords used to perform some operation on the given variables or values. An operator either changes the value of the given variable or produces a new result. Let us see some of the operators used in Python.

Arithmetic operators

Arithmetic operators are the symbols that can perform different arithmetic operations on the given variables or values. The symbols are self-explanatory. Following, we will perform some common arithmetic operations with the Python arithmetic operators, as shown in the following figure:

```
[18] a = 25
     b = 6
     print(a + b) # addition
     print(a - b) # subtraction
     print(a * b) # multiplication
     print(a/b)   # division
     print(a//b)  # floor division
     print(a%b)   # modulus
     print(a**b)  # exponentiation - a raised to the power of b

     31
     19
     150
     4.166666666666667
     4
     1
     244140625
```

***Figure 1.9**: Arithmetic operators*

Comparison operators

We use the comparison operators to compare two values. These are the mathematical symbols used for comparison, as shown in the following figure:

```
a = 25
b = 5
print(a==b) # equal
print(a!=b) # not equal
print(a>b)  # greater than
print(a<b)  # less than
print(a>=b) # greater than equal
print(a<=b) # less than equal

False
True
True
False
True
False
```

***Figure 1.10**: Comparison operators*

Logical operators

There are three logical operators in Python such as **and**, **or**, and **not**. They combine two conditional statements. Logical **and** is true if both the statements are true; otherwise, it is false. Logical **or** is true if either of the statements is true; otherwise, false. Logical **not** is true if the given statement is false, otherwise true, as shown in the following figure:

```
[24] a = 25
     b = 5
     c = 15
     print((a>b) and (c>b))
     print((b>c) or (a>b))
     print(not(a>b))

     True
     True
     False
```

***Figure 1.11**: Logical operators*

Identity operators

There are two identity operators in Python such as **is** and **is not**. These are used to check if two objects are the same (with the same memory locations) or not, as shown in the following figure:

```
[27] a = 5
     b = a
     c = 6
     print(a is b)
     print(a is not b)
     print(a is c)

     True
     False
     False
```

***Figure 1.12**: Identity operators*

Membership operators

These are used to verify if the given value or variable belongs to a given sequence. In Python, the sequence is represented as a list, tuple, dictionary, set, and so on. We will know more about these in the next section. A string can also be considered as a sequence of characters, and membership operators can be applied to it, as shown in the following figure:

```
[29] a = [4, 1, 6, 9, 2]
     print(4 in a)
     print(12 in a)
     print(10 not in a)

     True
     False
     True
```

***Figure 1.13**: Membership operators*

Python conditional statements

When we try to write logic in Python, we often need to handle decisions by evaluating the given conditions. The conditional statements in Python are used to evaluate the conditions and to route the program's flow accordingly to the correct direction. Keywords used for conditional statements are **`if`**, **`elif`**, and **`else`**.

The **`if`** statement in Python is used with an expression that evaluates the given condition. Inside the **`if`** block, other statements are written to be executed.

The **`elif`** statement is the short form of **`else if`**. An **`elif`** statement is always preceded by an **`if`** statement. If the condition written after the **`if`** keyword is not met, then the program's flow goes inside the **`elif`** block and evaluates the condition given. Remember, an **`elif`** block always comes with an expression to evaluate a given condition.

The **`else`** keyword does not come with any condition to evaluate. If any of the preceding conditions are not met, the program's flow goes inside the **`else`** block. In this example, we can see that b is greater than a. Hence, only the print statement inside the **`elif`** block is executed.

These conditional blocks can be nested too. That means there can be one conditional block inside another, and so on.

Refer to *Figure 1.14:*

```
[32] a = 2
     b = 5
     if a > b:
         print("a is greater than b.")
     elif b > a:
         print("b is greater than a.")
     else:
         print("a and b are same.")

     b is greater than a.
```

***Figure 1.14**: Conditional block*

Python loops

The loops are used to perform repetitive operations or iterations. Python provides two loop commands: **`while`** and **`for`**.

The **`while`** loop evaluates a condition and keeps on iterating the statements written inside the **`while`** block as long as the condition is true. Here, in this example, we initiated the variable **`i`** with the value **`0`**. The while loop evaluates if **`i`** is less than **`5`** or not. Inside the loop, we are printing the current value of **`i`** and incrementing its value by one. After running 5 times, the value of **`i`** becomes 5. So, the **`while`** condition (the value of **`i`** is less than 5) fails, and the loop stops iterating further, as shown in the following figure:

```
[34] i = 0
     while i < 5:
         print(i)
         i = i+1

0
1
2
3
4
```

***Figure 1.15**: While loop*

The **`for`** loop is mostly used in iterating through a sequence. In Python, the sequence is expressed as a list, tuple, dictionary, string (sequence of characters), and so on. In the given example, we are iterating through a list called **`nums`** using a for loop. We will know more about the sequence in the next section. Like conditional blocks, loops can be nested too, as shown in the following *Figure 1.16*:

```
[1] nums = [10, 20, 30, 40, 50]
    for i in nums:
        print(i)

10
20
30
40
50
```

***Figure 1.16**: For loop*

The **`break`** statement in Python is used inside a loop to come out of the loop if a given condition is met. The loop does not iterate through all the items if the **`break`** condition is met early. The

`continue` statement in Python is used to skip the following statements written inside the loop block and continue from the next iteration. Refer to the following figure:

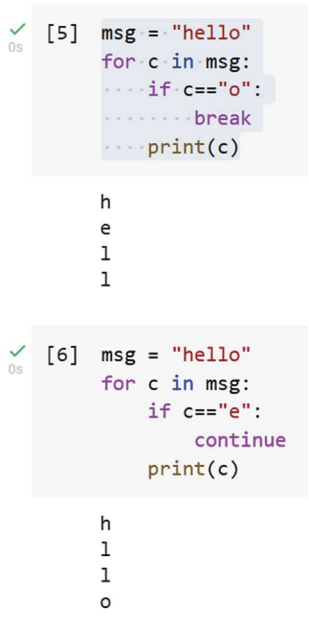

***Figure 1.17**: break and continue*

In the first example of *Figure 1.17*, we are iterating through a string variable called msg and prints every character. The break statement is used when the character **`o`** is reached. Hence, only **`h`**, **`e`**, **`l`**, and **`l`** are printed, and the loop stops from iterating further. So, the character **`o`** is not printed.

In the second example of *Figure 1.17*, we are iterating the same way, but the continue statement is used when the character **`e`** is reached. Hence, when the match happens, the print statement is not executed, and the loop again starts from the beginning. So, only **`h`**, **`l`**, **`l`**, and **`o`** are printed, and **`e`** is not printed.

The **`range()`** function in Python is often used with loops. This function creates a sequence through which the loop can iterate through. The basic syntax of the **`range()`** function is:

range(start, stop, step):

- start: the value where the sequence starts; by default, it is 0.
- stop: the value till the point sequence runs (this value is excluded)
- step: every nth value to select for iteration within the start and stop; by default, 1.

Refer to the following figure:

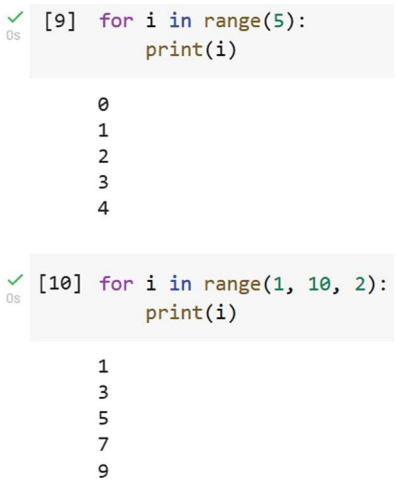

***Figure 1.18**: The range() function*

In the first example of *Figure 1.18*, we are using the range with a single value. Hence, by default, the range starts from 0 and goes up to 5 with step 1. In the second example of *Figure 1.18* above, we have mentioned the start, stop, and step values as 1, 10, and 2. Hence, every second value from 1 to 10 is selected.

Basic Python data structures

Data structures help us in organizing data so that they can be stored and retrieved efficiently. A single data structure cannot be used in every situation. Based on the logic to be implemented, we need to think of the best data structure for storing the data inside the program. The basic data structures that are available in Python are list, tuple, dictionary, and set. These are also called Python collection data types.

List

Python lists are used to store a collection of values in a single variable. Python lists are defined, keeping the values inside square brackets. We can also use the **`list()`** keyword (list constructor) to create a list. Let us create a list called city:

```
1. city = ["Paris", "Mumbai", "New York", "London", "Tokyo"]
```

We can store a mix of data types in a Python list. For example, there can be an int, a float, and a string together in a list. There can also be another list as a member element of a list. Refer to the following:

```
1. items = ["Paris", 2, 5.98765, [3, 7, 9]]
```

List items can be accessed by their index. Python list's index starts with 0.

Refer to the following figure:

```
city = ["Paris", "Mumbai", "New York", "London", "Tokyo"]
print(city[0])
print(city[1])
print(city[-1])
print(city[-2])
```

```
Paris
Mumbai
Tokyo
London
```

***Figure 1.19**: List's index*

As shown in *Figure 1.19*, if we try to access the first element of the list city, we will use the index value 0 within a square bracket as **`city[0]`**. Similarly, the second element can be accessed by **`city[1]`** and so on. Python also allows us to retrieve the elements from the end using the negative index. So, **`city[-1]`** will return the last element of the list. **`city[-2]`** will return the second last element of the list and so on.

Quite often, we may need to slice a small portion of a list. We can get a range of elements by index. Remember that slicing does not need any Python function or keyword to perform. We need to use the indices of the slice range within a square bracket after the list variable's name. The basic syntax of the slice is:

`list_name[start : stop : step]:`

- **`start`**: the value of the index where slicing starts.
- **`stop`**: the value of the index where slicing stops.
- **`step`**: every nth value to select within the start and stop (default value is 1)

The examples in the figure below are shown with different slicing:

```
[16] nums = [2,4,6,8,9,1]
     print(nums[0:3]) # from 0th item up to 3rd item
     print(nums[2:4]) # from 2nd item up to 4th item
     print(nums[::])  # all the items
     print(nums[2:])  # from 2nd items till the end
     print(nums[:2])  # from beginning up to 2nd item
     print(nums[::-1])# all the items in reverse order
     print(nums[::2]) # every second item starting from 0th item
```

```
[2, 4, 6]
[6, 8]
[2, 4, 6, 8, 9, 1]
[6, 8, 9, 1]
[2, 4]
[1, 9, 8, 6, 4, 2]
[2, 6, 9]
```

***Figure 1.20**: Slicing*

Python lists are ordered. We can change, add, and remove values from lists once they are created. Python lists come with some built-in functions.

One important function is **`len()`**, which returns the length of the list. To add an element at the end of the list, we need to use the **`append()`** function. To count the number of occurrences of a value in a list, we need to use the **`count()`** function. Here, the intended value to search should be given as the input to the function. The examples are shown in *Figure 1.21*:

```
[20] nums = [2,4,6,4,8,9,4]
     len(nums)

     7

[21] nums.append(99)
     nums

     [2, 4, 6, 4, 8, 9, 4, 99]

[22] nums.count(4)

     3
```

***Figure 1.21**: Useful functions*

If we want to insert some value to a specific location of the list, we need to use the **`insert()`** function. The index of the location in the list and the value needs to be given as the input to the function. The **`pop()`** function removes an element from the list. By default, the last element of the list is removed. If we specify any index value inside the **`pop()`** function, the element from that location will be removed. The examples are shown in *Figure 1.22*:

```
[26] # insert 7 at 1st position
     nums = [2,4,6,4,8]
     nums.insert(1, 7)
     nums

     [2, 7, 4, 6, 4, 8]

[27] # remove the last item (8)
     nums.pop()
     nums

     [2, 7, 4, 6, 4]

[28] # remove the 2nd item (4)
     nums.pop(2)
     nums

     [2, 7, 6, 4]
```

***Figure 1.22**: Useful functions*

The **`reverse()`** function reverses a list. The **`sort()`** function is very widely used in the list data structure. We often need to sort the list in ascending or descending order. The **`sort`** function takes a binary input called **`reverse`**. If the value of **`reverse`** is true, the sorting is done in ascending way. By default, this value is set to false, and the sorting is done in a descending way. The examples are shown in *Figure 1.23*:

```
[34] # reverse the list
     nums = [2,4,6,4,8]
     nums.reverse()
     nums

     [8, 4, 6, 4, 2]

[35] #sort the list in ascending order
     nums.sort()
     nums

     [2, 4, 4, 6, 8]

[36] #sort the list in descending order
     nums.sort(reverse=True)
     nums

     [8, 6, 4, 4, 2]
```

***Figure 1.23**: Useful functions*

Tuple

Python tuples are also used to store a collection of values in a single variable. Tuples are defined, keeping the values inside parentheses. We can also use the **`tuple()`** keyword (tuple constructor) to create a tuple. The main difference between a tuple and a list is that a list is mutable, but a tuple is immutable. That means once a list is created, we can change the list. We can add new items to it, remove an existing item, reverse and sort the list, and so on. However, tuples cannot be changed once they are created. Tuples are used to keep the collection of the values that are not going to change during the execution of the program. For example, the coordinates (latitude and longitude) of a city, the seven days of a week, the twelve months of a year, and so on. For this reason, functions like **`append()`**, **`pop()`**, **`reverse()`**, **`sort()`**, **`insert()`**, **`remove()`**, etc. are used in the list that is not available for the tuple.

As shown in *Figure 1.24*, let us create a tuple called weekdays. The **`len()`** and **`count()`** functions in tuples work the same as they do in lists. Like lists, tuples also allow duplicate values and a mix of data types. We can retrieve or access the values from the tuples using the index. The index value starts from 0, and slicing is also possible. Refer to the following figure:

```
[38] weekdays = ("sun","mon","tue","wed","thu","fri","sat")
     len(weekdays)

     7

[39] weekdays.count("mon")

     1

[40] weekdays[0]

     'sun'

[41] weekdays[1:5]

     ('mon', 'tue', 'wed', 'thu')
```

Figure 1.24: Tuple

Dictionary

A dictionary is an important data structure of Python. We have seen that the values stored in lists and tuples can be accessed by index. The index starts from zero and increments automatically. Unlike storing data in this way, we often may require storing data in a key and value pair format where values can be accessed from the data structure by using the correct key. So, the key is more like an index but fully customizable by the user. Storing data in the key: value pair makes the dictionary quite optimized for many applications.

A Python dictionary is defined using curly braces inside which comma-separated key:value pairs are written. All the keys in a dictionary must be unique. However, there can be duplicated values in a dictionary. Values can be of any data type, but keys must be of immutable data type. Since Python lists are mutable, they cannot be used as the keys of any dictionary. However, tuples can be used as the keys in a dictionary. Just like the list and tuple, we can also use the **`dict()`** keyword (dictionary constructor) to create a dictionary.

As we used an index to retrieve data from lists and tuples, we can use the key in a dictionary to get the corresponding value. As shown in *Figure 1.25*, the key is used within the square bracket. We can also insert and update a value by giving the appropriate key. We can use the **`keys()`** function to get the list of all the keys in a dictionary. Similarly, the **`values()`** function returns the list of all the values, as shown in the following figure:

```
[43] balls = {"red":1, "blue":6, "yellow":5, "black":3}
     # get the value of key "black"
     balls["black"]

     3

[44] # insert new value for key "green"
     balls["green"]=5
     balls

     {'red': 1, 'blue': 6, 'yellow': 5, 'black': 3, 'green': 5}

     # get all the keys and values
     print(balls.keys())
     print(balls.values())

     dict_keys(['red', 'blue', 'yellow', 'black', 'green'])
     dict_values([1, 6, 5, 3, 5])
```

***Figure 1.25**: Dictionary*

While iterating through a dictionary, we need to use a **`for`** loop. By default, the keys() of the dictionary are returned for the iteration. However, we can explicitly use the **`keys()`** and **`values()`** functions to return the keys and values to be iterated with. We can even use the **`items()`** function to return both the keys and values together, as shown in the following figure:

```
✓ 0s  # print the keys
      for ball in balls.keys():
          print(ball)

      print("**************")
      #print the values
      for key, val in balls.items():
          print(balls[key])

      red
      blue
      yellow
      black
      green
      **************
      1
      6
      5
      3
      5
```

***Figure 1.26**: Iterating a dictionary*

Set

Set is another type of Python data structure that stores a collection of values. A set is defined by writing the comma-separated values inside a curly brace and are unordered. The items inside a set are not positioned and ordered by index. Hence, we cannot retrieve data from a set using an index. Once a set is created, we cannot change any of the items present in that set. However, we can add or remove items from a set. There cannot be any duplicate element present in a set.

If we need to add an item to the set, we need to use the **`add()`** function. An item can be removed from the set using the **`remove()`** method, as shown in the following figure:

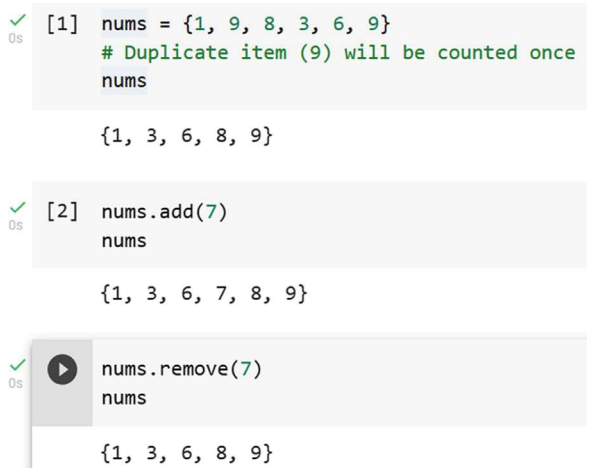

***Figure 1.27**: Set*

Operations from set theory (for example, union, intersection, difference, and so on) can be performed on Python set data structure as shown in *Figure 1.28*:

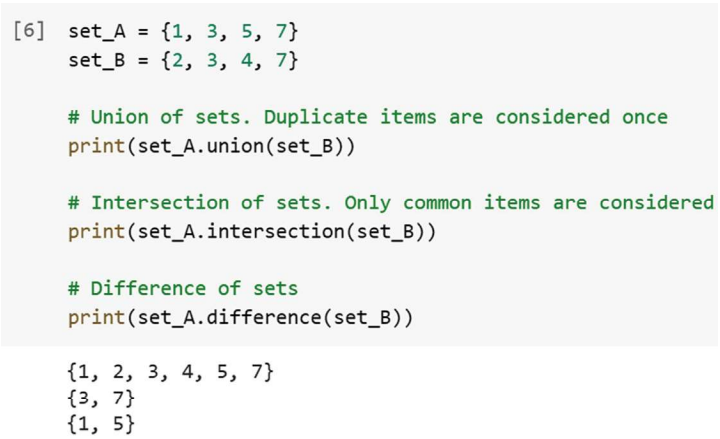

***Figure 1.28**: Set operations*

Revisiting string

String-type variables hold raw text data. Handling raw text is crucial in AI, especially for developing **Natural Language Processing** (**NLP**) related tasks. Let us learn different ways of manipulating strings in Python.

Strings can be considered as a list of characters, and they are iterable like lists. Unlike C/C++ or Java, Python has no separate data type called character. A single character in Python is considered a string with length 1.

Like a Python list, the characters in a string can be accessed by index. The **`len()`** function works on the string too. It returns the length of the string. Like Python list, slicing can be used on strings, too. To check if a substring belongs to a string, we can use the **`in`** keyword (membership operator), as shown in the following figure:

```
[10] msg = "hello world"

     # Get the first (0th) element
     print(msg[0])

     # Length of the string
     print(len(msg))

     # Slicing the string - 1st to 4th
     print(msg[1:4])

     # Check if substring "hell" belongs to the string msg
     print("hell" in msg)

     h
     11
     ell
     True
```

***Figure 1.29**: Strings*

We can change the case of a string using basic functions. The **`upper()`** function converts the entire string into upper-case. The **`lower()`** function does just the opposite. Sometimes, we need to remove the preceding and trailing whitespaces from a string. Functions used for this task are **`rstrip()`**, **`lstrip()`**, and **`strip()`**, as shown in the following figure:

```
[12] # String with a leading and trailing whitespaces
     msg = " Hello World "

     # Uppercase
     print(msg.upper())

     # Lowercase
     print(msg.lower())

     # Remove the leading whitespace
     print(msg.lstrip())

     # Remove the trailing whitespace
     print(msg.rstrip())

     # Remove whitespaces from both sides
     print(msg.strip())
```

```
 HELLO WORLD
 hello world
Hello World
 Hello World
Hello World
```

***Figure 1.30**: String functions*

The **`split()`** function is quite widely used in manipulating strings. This function splits a string based on the given separator and returns a list of split strings. By default, the whitespace is considered the separator. Otherwise, it should be explicitly mentioned. Another widely used function is **`join()`**. It joins all the items of an iterable sequence (list, tuple, dictionary, and so on) into a single string using the given separator, as shown in the following figure:

```
[15] msg = "This is a #hashtag"
     print(msg.split())
     print(msg.split("#"))
```

```
['This', 'is', 'a', '#hashtag']
['This is a ', 'hashtag']
```

```
[16] words = ["a", "fat", "cat", "sat", "on", "the", "mat"]
     sentence = " ".join(words)
     sentence
```

```
'a fat cat sat on the mat'
```

***Figure 1.31**: String functions*

As shown in *Figure 1.32*, concatenating two strings is possible using the **+** sign. However, we cannot concatenate or combine any other datatype with string. If we try to do so, Python will