International Islamia School - Otsuka

Knowledge • Faith • Etiquette

INTRODUCTION TO PYTHON

For Grades 8 & 9

ABDULMALEK SALEM SHEFAT ©2024

Introduction to Python [1]

International Islamia School Otsuka [2]

Abdulmalek Salem Shefat[3]

September 3, 2024

[1]`https://www.python.org`
[2]`https://www.iiso-edu.jp`
[3]malekshefat@gmail.com

Contents

Part I

Core Python Concepts

Chapter 1

Syntax and Basic Structure

1.1 Introduction

Overview

Welcome to the world of Python programming! Python is a powerful and easy-to-learn programming language that's used by many people around the world to create everything from simple games to complex websites and even robots!

In this first chapter, we're going to learn the basics of Python. Think of Python as a special language that you use to give instructions to a computer. Just like you follow certain rules to write sentences in English, we follow specific rules, called *syntax*, to write instructions in Python.

By the end of this chapter, you'll be able to write and understand basic Python programs. It's like learning the ABCs of Python!

So, let's get started and have some fun with Python programming!

Objectives

In this chapter, you will:

- Learn how to write and run your first Python program.
- Learn how to name your script correctly.
- Understand the basic rules of Python syntax.
- Learn how to write comments in your code.
- Discover the importance of indentation in Python.
- Use the `print` function to display messages on the screen.

1.2 Concepts

1.2.1 Naming Conventions for Python Scripts

When writing Python scripts, it's important to follow naming conventions to ensure that your code is readable and maintainable. Proper naming conventions help make your scripts easier to understand and work with, both for yourself and for others who might read or use your code.

1. **Descriptive and Meaningful Names**: Choose names that clearly describe what the script does. Avoid vague or overly generic names.
2. **Use Lowercase Letters**: Python script names should be written in lowercase letters. This makes them easy to read and consistent with the general Python style guide.
3. **Use Underscores to Separate Words**: If a script name consists of multiple words, separate them with underscores (‘_‘) rather than spaces or camel case. This improves readability.
4. **Avoid Special Characters**: Stick to letters, numbers, and underscores. Avoid using special characters like hyphens (‘-‘) or spaces, as they can cause issues in some environments.
5. **Keep Names Short and Concise**: While names should be descriptive, they should also be short enough to be easily readable and not overly long.

Examples of Good and Bad Names

Good Script Names	Bad Script Names
data_analysis.py	DataAnalysis.py
calculate_sum.py	calculate-Sum.py
plot_graph.py	plotgraph.py
generate_report.py	generate report.py
web_scraper.py	webscraper_v2.py

Table 1.1: Examples of Good and Bad Python Script Names

Explanation

- **Good Script Names**:
  - `data_analysis.py`: Clear and descriptive, uses underscores to separate words.
  - `calculate_sum.py`: Descriptive of its function, uses lowercase and underscores.
  - `plot_graph.py`: Easy to understand and follows conventions.
  - `generate_report.py`: Concise and descriptive with underscores.
  - `web_scraper.py`: Clearly describes the purpose of the script.
- **Bad Script Names**:
  - `DataAnalysis.py`: Uses camel case and starts with an uppercase letter.
  - `calculate-Sum.py`: Uses a hyphen, which is not recommended.

- `plotgraph.py`: Lacks readability without underscores to separate words.
- `generate report.py`: Contains spaces, which can cause issues.
- `webscraper_v2.py`: Includes versioning in the name, which can be avoided.

1.2.2 The Importance of Indentation

Why Indentation Matters

In Python, indentation is not just a matter of style; it's a fundamental part of the language's syntax. Unlike many other programming languages that use braces or other symbols to define code blocks, Python relies on indentation to determine the grouping of statements. This means that consistent and correct indentation is crucial for writing functional and error-free Python code.

1. **Defines Code Blocks**: Indentation in Python is used to define blocks of code, such as those within functions, loops, and conditionals. Proper indentation is necessary to indicate which statements are part of which block.

2. **Prevents Syntax Errors**: Incorrect indentation can lead to syntax errors, causing your code to fail or behave unexpectedly. Python will raise an 'IndentationError' if the indentation is inconsistent or incorrect.

3. **Improves Readability**: Consistent indentation makes your code easier to read and understand. It visually separates different blocks of code, making it clear which statements belong together.

4. **Ensures Correct Execution**: Proper indentation ensures that code executes in the correct order. Misaligned code can lead to logical errors or unintended behavior.

Examples

Here are some examples that illustrate the importance of indentation in Python:

Correct Indentation

Listing 1.1: Correct Indentation Example

```python
def greet(name):
    # This is a correctly indented block
    if name:
        print(f"Hello, {name}!")
    else:
        print("Hello, world!")

greet("Alice")
```

In the example above, the 'if' and 'else' statements are correctly indented to show that they are part of the 'greet' function. The 'print' statements are properly indented to show their association with the 'if' and 'else' blocks.

Incorrect Indentation

Listing 1.2: Incorrect Indentation Example

```python
def greet(name):
    # This code will raise an IndentationError
    if name:
    print(f"Hello, {name}!")
    else:
        print("Hello, world!")

greet("Alice")
```

In this example, the ‘print‘ statement under the ‘if‘ condition is not correctly indented, leading to an ‘IndentationError‘. Python expects consistent indentation within code blocks.

Another Incorrect Indentation Example

Listing 1.3: Another Incorrect Indentation Example

```python
for i in range(5):
    print(i)
  print("Done!")
```

Here, the ‘print(”Done!”)‘ statement is incorrectly indented. It should be at the same level as the ‘for‘ loop if it’s meant to be outside of the loop block. The incorrect indentation may cause logical errors in the code.

Best Practices

- Always use a consistent number of spaces (typically 4) for each level of indentation.
- Avoid mixing tabs and spaces; choose one method and stick to it.
- Use an integrated development environment (IDE) or text editor that helps manage indentation automatically.

Proper indentation is essential for writing clear, correct, and maintainable Python code. By following these guidelines and examples, you can avoid common pitfalls and ensure your code functions as intended.

1.2.3 Comments

Comments are an essential part of writing clear and maintainable code. They provide explanations and context for the code, making it easier for others (and yourself) to understand what the code does. Comments are ignored by the Python interpreter when the code runs, so they don’t affect the execution of the program.

Why Comments Are Important

- **Enhance Readability**: Comments help explain the purpose of code sections, making it easier for others to read and understand your code.

- **Document Your Code**: Comments can describe the logic behind complex code, the purpose of functions, and the roles of different variables, which helps in documenting the code for future reference.
- **Assist in Debugging**: Comments can be used to temporarily disable parts of the code for debugging purposes without deleting the code.
- **Improve Collaboration**: When working in teams, comments ensure that everyone understands the code's functionality and how different parts of the code interact.

Single-Line Comments

Single-line comments are used to add brief explanations or notes on a single line. They begin with the hash symbol (‘#‘) and continue to the end of the line.

Listing 1.4: Single-Line Comment Example

```python
# This is a single-line comment
print("Hello, world!") # This prints a message to the screen
```

In the example above, `# This is a single-line comment` and `# This prints a message to the screen` are single-line comments. They provide brief descriptions of the code but do not span multiple lines.

MultiLine Comments

Multi-line comments are used for longer explanations that span multiple lines. In Python, multi-line comments are usually written using triple quotes (‘”’‘ or ‘”””‘). Although technically these are multi-line strings, they are often used as comments.

Listing 1.5: Multi-Line Comment Example

```python
"""
This is a multi-line comment.
It can span multiple lines and is useful for longer explanations.
Python treats these triple-quoted strings as comments if they are not assigned
    to any variable.
"""
print("Hello, world!")
```

In the example above, the triple quotes are used to enclose a multi-line comment that explains what the code does. This type of comment is useful for providing detailed descriptions.

Docstrings

Docstrings are a special kind of comment used to describe the purpose and usage of functions, classes, and modules. They are enclosed in triple quotes and appear immediately after the function, class, or module definition. Docstrings can be accessed programmatically and are often used for generating documentation.

Listing 1.6: Docstring Example

```python
def greet(name):
    """
    Greets the person with the provided name.

    Parameters:
    name (str): The name of the person to greet.

    Returns:
    None
    """
    print(f"Hello, {name}!")
```

In the example above, the docstring provides detailed information about the `greet` function, including its parameters and return value. This documentation helps users understand how to use the function.

Best Practices for Comments

- **Be Clear and Concise**: Ensure comments are easy to understand and provide clear explanations.
- **Update Comments**: Keep comments up-to-date with code changes to avoid confusion.
- **Avoid Redundancy**: Do not comment on obvious code. Use comments to explain why something is done, not what is done.
- **Use Docstrings for Documentation**: Use docstrings for documenting functions, classes, and modules.

Comments are a powerful tool for writing understandable and maintainable code. By using single-line comments, multi-line comments, and docstrings effectively, you can make your code more accessible and easier to work with.

1.2.4 The `print` Function and String Formatting

The `print` function is one of the most commonly used functions in Python. It outputs data to the console, which is useful for displaying results, debugging, and interacting with users. Python also provides various ways to format strings, allowing you to control how text and variables are presented.

Using the `print` Function

The `print` function writes text or variables to the console. It can handle multiple arguments and will automatically separate them with spaces by default. The syntax of the `print` function is as follows:

Listing 1.7: Basic Usage of `print`

```
print(object1, object2, ..., sep=' ', end='\n')
```

- **object1, object2, ...**: These are the objects to be printed. They can be strings, numbers, or other data types.
- **sep**: A string inserted between objects. The default is a space (“ ”).

- **end**: A string appended after the last object. The default is a newline character ('\n').

Basic Examples

Listing 1.8: Basic `print` Function Example

```python
print("Hello, world!") # Prints a simple message
print("Hello", "world!") # Prints multiple arguments separated by a space
print("Hello", "world!", sep='-') # Custom separator
print("Hello", end='') # Custom end character
print(" world!")
```

In these examples:

- `print("Hello, world!")` prints a simple message.
- `print("Hello", "world!")` prints multiple arguments separated by a space.
- `print("Hello", "world!", sep='-')` prints with a custom separator ('-').
- `print("Hello", end='')` prints without a newline at the end, followed by another `print` statement.

String Formatting

String formatting allows you to include variables and expressions inside strings, making it easier to create dynamic text. Python offers several methods for string formatting:

Old-Style Formatting

This method uses the '%' operator to format strings. It is similar to formatting in C.

Listing 1.9: Old-Style String Formatting Example

```python
name = "Alice"
age = 12
print("Name: %s, Age: %d" % (name, age))
```

In this example: - `%s` is a placeholder for a string. - `%d` is a placeholder for an integer.

str.format() Method

This method uses curly braces (") as placeholders and the `format` method to substitute values.

Listing 1.10: str.format() Method Example

```python
name = "Alice"
age = 12
print("Name: {}, Age: {}".format(name, age))
```

In this example: - are placeholders replaced by the values passed to `format`.

f-Strings (Formatted String Literals)

Introduced in Python 3.6, f-strings provide a concise and readable way to embed expressions inside string literals.

Listing 1.11: f-Strings Example

```python
name = "Alice"
age = 12
print(f"Name: {name}, Age: {age}")
```

In this example: - `f"Name:  name, Age:  age"` directly embeds variables into the string.

Best Practices for String Formatting

- **Choose the Right Method**: Use f-strings for readability and performance when using Python 3.6 or later. Use ‘str.format()‘ for compatibility with older Python versions.
- **Be Consistent**: Stick to one method for consistency throughout your codebase.
- **Use Descriptive Variable Names**: Ensure variables used in formatting are named clearly to improve code readability.

Understanding the `print` function and string formatting is crucial for effectively displaying and manipulating text in Python programs. By mastering these techniques, you can create dynamic and user-friendly output.

1.2.5 Errors and Debugging

Errors are inevitable in programming, but understanding them and knowing how to debug can make the process of writing and maintaining code much smoother. This section will cover common types of errors in Python and provide techniques for debugging.

Types of Errors

Python code can produce various types of errors. The most common ones include:

1- Syntax Errors

A syntax error occurs when Python cannot interpret the code due to incorrect syntax. This type of error usually happens when you misspell a keyword, omit a punctuation mark, or make a similar mistake.

Listing 1.12: Syntax Error Example

```python
print("Hello, world!" # Missing closing parenthesis
```

The above code will raise a `SyntaxError` because the closing parenthesis is missing.

2- Logical Errors

Logical errors occur when a program runs without errors but produces incorrect or unintended results. These errors are often caused by mistakes in the program's logic or algorithm. Unlike syntax or runtime errors, logical errors do not raise exceptions or cause the program to crash, making them harder to detect.

Examples of Logical Errors

Listing 1.13: Logical Error Example

```python
def calculate_area(radius):
    # Incorrect formula for area of a circle
    return radius + 3.14 * (radius ** 2)

area = calculate_area(5)
print(f"The area is: {area}")
```

In this example: - The formula used to calculate the area of a circle is incorrect. The correct formula is $\pi \times radius^2$, but the code mistakenly uses $radius + \pi \times radius^2$. - The program runs without errors but produces an incorrect result.

Identifying and Fixing Logical Errors

1. **Understand the Expected Output**: Clearly define what the correct output should be for given inputs.

2. **Review the Algorithm**: Check the logic and steps of the algorithm used in the code.

3. **Use Test Cases**: Test the code with various inputs, including edge cases, to verify that it behaves as expected.

4. **Check Intermediate Results**: Print or log intermediate values to ensure they match expected values at different stages of computation.

5. **Refactor Code**: Simplify or reorganize code to make it easier to understand and identify logical flaws.

Listing 1.14: Improved Version of the Logical Error Example

```python
import math

def calculate_area(radius):
    # Correct formula for area of a circle
    return math.pi * (radius ** 2)

area = calculate_area(5)
print(f"The area is: {area}")
```

In the corrected version: - The formula is updated to use the correct calculation for the area of a circle: $\pi \times radius^2$. - The program now produces the correct result.

Best Practices for Avoiding Logical Errors

- **Plan Before Coding**: Design and plan the algorithm before writing code to avoid logical errors.
- **Break Down Problems**: Divide complex problems into smaller, manageable parts to simplify logic and debugging.
- **Document Your Code**: Use comments to describe the logic and assumptions in your code to make it easier to review and debug.
- **Peer Review**: Have others review your code to catch potential logical errors that you might have missed.

Logical errors can be challenging to identify and fix because they don't always produce visible symptoms. By understanding and applying strategies for detecting and addressing logical errors, you can improve the correctness and reliability of your programs.

3- Indentation Errors

An indentation error occurs when the code blocks are not properly aligned. Python uses indentation to define the structure of the code, so incorrect indentation will lead to errors.

Listing 1.15: Indentation Error Example

```python
def greet(name):
    print(f"Hello, {name}!")
   print("Welcome!") # Incorrect indentation
```

In this example, the second `print` statement is misindented, causing an `IndentationError`.

4- Name Errors

A name error happens when the code tries to use a variable or function that has not been defined.

Listing 1.16: Name Error Example

```python
print(x) # x is not defined
```

Here, the variable `x` is not defined before it is used, resulting in a `NameError`.

5- Type Errors

A type error occurs when an operation or function is applied to an object of inappropriate type.

Listing 1.17: Type Error Example

```python
print("The number is: " + 5) # Attempting to concatenate a string and an
    integer
```

In this case, Python raises a `TypeError` because you cannot concatenate a string and an integer directly.

Debugging Techniques

Debugging is the process of identifying and fixing errors in your code. Here are some techniques to help with debugging:

Using Print Statements

One of the simplest debugging methods is to use `print` statements to output values and track the flow of execution.

Listing 1.18: Using `print` Statements for Debugging

```python
def add_numbers(a, b):
    print(f"a: {a}, b: {b}") # Print values of variables
    return a + b

result = add_numbers(5, '10') # This will raise a TypeError
```

In this example, print statements help in understanding the values of `a` and `b` before the function returns.

Using a Debugger

A debugger is a tool that allows you to pause execution, inspect variables, and step through code line by line. Integrated development environments (IDEs) like PyCharm, VSCode, and others come with built-in debuggers.

- **Breakpoints**: Set breakpoints to pause execution at specific lines of code.
- **Step Through Code**: Execute your code one line at a time to understand its behavior.
- **Inspect Variables**: Check the values of variables at different points in the code.

Handling Exceptions with `try` and `except`

You can handle runtime errors using `try` and `except` blocks to manage exceptions gracefully.

Listing 1.19: Exception Handling Example

```python
try:
    result = 10 / 0 # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("You can't divide by zero!")
```

In this example, the `try` block contains code that may cause an error. The `except` block catches the error and provides a user-friendly message.

Reading Tracebacks

Tracebacks provide valuable information about the sequence of function calls and the location of the error. Carefully read the traceback to identify where the error occurred and trace its origin.

Listing 1.20: Traceback Example

```python
def divide(x, y):
    return x / y

divide(1, 0) # This will raise a ZeroDivisionError
```

The traceback will indicate that the error occurred in the `divide` function when dividing by zero.

Best Practices for Debugging

- **Understand the Error Message**: Read and interpret error messages to understand what went wrong.
- **Isolate the Problem**: Simplify the code to isolate the issue and identify the root cause.
- **Test Frequently**: Test your code frequently to catch errors early in the development process.
- **Document Your Process**: Keep notes on debugging steps and solutions for future reference.

By learning how to identify and fix errors effectively, you can improve the reliability and quality of your code. Debugging is an essential skill that enhances your problem-solving abilities and helps you write better programs.

Key Terms

- **Syntax**: The set of rules that defines how to write code in Python (Grammar).
- **Indentation**: The use of 4 spaces or a tab at the beginning of a line to indicate a block of code.
- **Comment**: A line of text in the code that is not executed by the computer and is used to explain the code.
- **Print Function**: A function in Python used to display messages on the screen.
- **Statement**: A single line of code that performs an action.
- **Block of Code**: A group of statements that are grouped together, often by indentation.

1.3 Summary

1.3.1 Naming Conventions for Python Scripts

- **Descriptive Names**: Use names that clearly describe the script's purpose.

- **Lowercase Letters**: Write script names in lowercase.
- **Underscores**: Separate words with underscores for readability.
- **Avoid Special Characters**: Stick to letters, numbers, and underscores.
- **Conciseness**: Keep names short and clear.

1.3.2 The Importance of Indentation

- **Defines Code Blocks**: Indentation is crucial for grouping statements into blocks.
- **Prevents Errors**: Correct indentation avoids `IndentationError`.
- **Improves Readability**: Consistent indentation makes code more readable.
- **Ensures Correct Execution**: Proper indentation maintains the correct flow of execution.

1.3.3 Comments

- **Readability**: Comments explain and clarify code to improve understanding.
- **Documentation**: They provide explanations for code logic, functions, and variables.
- **Debugging**: Temporarily disabling code using comments can assist in debugging.
- **Collaboration**: Comments make code easier to understand for others working on it.

Types of Comments

- **Single-Line Comments**: Use `#` for comments on a single line.
- **Multi-Line Comments**: Use triple quotes `'''...'''` or `"""..."""` for longer explanations.

1.3.4 Errors and Debugging

Common Errors

- **Naming Conventions**
  - *Incorrect Characters*: Using spaces or special characters in script names can lead to file not found errors.
  - *Conflicting Names*: Naming scripts the same as existing Python modules can cause import errors or unexpected behavior.
- **Comments**

  - *Unintentional Code Execution*: Forgetting to comment out code may result in unintended execution or errors.
  - *Misleading Comments*: Comments that do not accurately reflect the code can cause confusion and misinterpretation.
  - *Unclosed Multi-Line Comments*: Forgetting to close multi-line comments with the correct ending delimiter can lead to syntax errors.

- **Types of Errors** Python code can produce various types of errors. The most common ones include:

  - **Syntax Errors**: Occur due to incorrect syntax such as missing punctuation or misspelled keywords.
  - **Logical Errors**: Occur when the code runs without crashing but produces incorrect results due to flawed logic.
  - **Indentation Errors**: Result from improper alignment of code blocks, which Python uses to define code structure.
  - **Name Errors**: Happen when the code tries to use a variable or function that has not been defined.
  - **Type Errors**: Occur when an operation or function is applied to an object of inappropriate type, such as concatenating a string with an integer.

Debugging Techniques

- **Print Statements**: Use print statements to check the values of variables at different stages of execution.
- **Python Debugger (pdb)**: Use `import pdb; pdb.set_trace()` to set breakpoints and step through the code.
- **Integrated Development Environment (IDE) Tools**: Utilize built-in debugging tools in IDEs like Visual Studio Code or PyCharm.
- **Error Messages**: Pay close attention to error messages and stack traces for clues on what went wrong.

Next Steps

Outline what to study next or how this chapter connects to upcoming topics.

1.4 Quiz

To test your understanding of the material covered in this chapter, please complete the following quiz:

- Click the link below to access the quiz.
- Answer the questions to the best of your ability.

- Submit your responses to receive feedback on your performance.

Quiz Link: Click here to take the quiz

1.5 Exercises and Homeworks

Research Topics

Assign topics iteratively. Each student must choose 3 topics they like, then iteratively, the student will be assigned only one topic out of the three, such that, no two student will have the same topic, if possible.

1. **The Future of Quantum Computing:** Quantum computing holds the promise of revolutionizing technology by solving complex problems beyond the capabilities of classical computers. As research progresses, quantum computers are expected to impact fields such as cryptography, optimization, and materials science, though widespread practical use may still be years away.

2. **The Evolution of the Internet:** The internet has evolved from a simple network of connected computers into a global system facilitating instant communication, information sharing, and commerce. Key stages include the development of web browsers, social media, and mobile internet, all of which have transformed how we interact and conduct business.

3. **Understanding Cybersecurity:** Cybersecurity involves protecting computer systems and networks from digital attacks, theft, and damage. It encompasses measures like encryption, firewalls, and threat detection to safeguard sensitive data and maintain privacy, ensuring the integrity and availability of information in a connected world.

4. **The Role of Robotics in Modern Industry:** Robotics is increasingly central to modern industry, automating tasks in manufacturing, logistics, and other sectors. Robots enhance efficiency, precision, and safety while reducing human labor and operational costs, driving innovation in production processes and service delivery.

5. **Introduction to Virtual Reality (VR):** Virtual Reality (VR) creates immersive digital environments that simulate real or imagined worlds. Using VR headsets and controllers, users can interact with these environments, making it valuable for applications in gaming, training, education, and virtual tours.

6. **Exploring Augmented Reality (AR):** Augmented Reality (AR) overlays digital information onto the real world, enhancing the user's perception of their surroundings. Through devices like smartphones or AR glasses, AR applications blend virtual elements with physical reality, useful in fields such as gaming, navigation, and training.

7. **The Impact of Social Media on Communication:** Social media has transformed communication by enabling instant, global interactions. Platforms like Facebook, Twitter, and Instagram facilitate social networking, information sharing, and personal expression, influencing relationships, news dissemination, and marketing strategies.

8. **The Basics of Game Design:** Game design involves creating interactive entertainment experiences, including storytelling, character development, and gameplay mechanics. Key aspects include designing game levels, developing engaging narratives, and balancing challenges to create enjoyable and immersive gaming experiences.

9. **Introduction to Cloud Computing:** Cloud computing delivers computing services over the internet, including storage, processing power, and software applications. By providing on-demand access to these resources, cloud computing enables businesses and individuals to scale operations, reduce costs, and access advanced technologies without owning physical infrastructure.

10. **Exploring Wearable Technology:** Wearable technology includes devices like smartwatches, fitness trackers, and augmented reality glasses that are worn on the body. These devices collect and provide data on health metrics, notifications, and environmental information, enhancing personal convenience and health monitoring.

11. **The Science of 3D Printing:** 3D printing, or additive manufacturing, creates physical objects from digital models by layering materials like plastic, metal, or resin. This technology enables rapid prototyping, customized production, and complex designs, revolutionizing fields such as manufacturing, healthcare, and engineering.

12. **Introduction to Smart Home Technology:** Smart home technology integrates devices and systems like thermostats, lighting, and security cameras into a connected network that can be controlled remotely. This technology enhances convenience, energy efficiency, and security, allowing users to automate and monitor their homes through smartphones or voice assistants.

13. **The Role of Augmented Reality in Education:** Augmented Reality (AR) in education enhances learning by overlaying digital content onto physical materials, such as textbooks or classroom environments. AR applications can visualize complex concepts, provide interactive experiences, and support immersive learning, making education more engaging and effective.

14. **Exploring Wearable Health Tech:** Wearable health technology includes devices that monitor and track health metrics such as heart rate, sleep patterns, and physical activity. These devices provide valuable insights for personal health management, support preventative care, and assist healthcare professionals in monitoring patients remotely.

15. **Introduction to Data Science:** Data science involves extracting insights from large volumes of data using techniques from statistics, machine learning, and data visualization. It encompasses data collection, cleaning, and analysis to support decision-making and uncover patterns, trends, and predictions in various domains.

16. **The Impact of Technology on the Environment:** Technology affects the environment through both positive and negative impacts. While advancements can lead to more efficient resource use and environmental monitoring, technology can also contribute to pollution, resource depletion, and electronic waste, necessitating sustainable practices and innovations.

17. **Introduction to Internet of Things (IoT):** The Internet of Things (IoT) connects everyday objects to the internet, allowing them to send and receive data. IoT applications range from smart home devices to industrial sensors, enabling automation, remote monitoring, and improved efficiency in various sectors by integrating physical and digital systems.

18. **Exploring the Ethics of Technology:** The ethics of technology involves examining the moral implications of technological advancements and their impact on society. Topics include privacy concerns, data security, artificial intelligence ethics, and the equitable distribution of technological benefits, ensuring responsible development and use of technology.

19. **Interperter vs Compiler:** Interpreters and compilers are tools used to execute programs written in high-level languages. An interpreter translates code line-by-line into machine code at runtime, while a compiler translates the entire code into machine code before execution. Each approach has different performance and debugging implications.

20. **The creator of python:** Python was created by Guido van Rossum and first released in 1991. Van Rossum designed Python with an emphasis on readability, simplicity, and ease of use, making it a popular language for beginners and professionals alike. Python has since evolved into one of the most widely used programming languages.

21. **Compare multiple Programming Languages:** Comparing programming languages involves evaluating their syntax, performance, use cases, and community support. Languages like Python, Java, and C++ offer different strengths: Python is known for ease of use, Java for portability, and C++ for performance. The choice of language often depends on the specific requirements of a project or application.

22. **Networking:** Networking involves the exchange of data between computers and devices through various protocols. Key protocols include TCP/IP for reliable data transmission, HTTP for web communication, and DNS for domain name resolution. Understanding networking is crucial for setting up and maintaining efficient and secure communication systems in various environments.

23. **RaDAR:** Radar (Radio Detection and Ranging) uses radio waves to detect and locate objects. It emits radio waves that bounce off objects and return to the radar receiver, allowing the determination of distance, speed, and direction. Limitations include reduced accuracy in adverse weather conditions and difficulties detecting small or low-reflectivity objects.

24. **LiDAR:** Lidar (Light Detection and Ranging) uses laser pulses to measure distances to objects, creating high-resolution 3D maps. It operates by sending laser beams and measuring the time it takes for the reflections to return. Limitations include reduced effectiveness in fog, rain, or snow, and potential high costs associated with the technology.

25. **camera (type, how it works, limitations, etc.):** Cameras capture visual information using sensors such as CCD or CMOS. They convert light into electronic signals to create images. Different types include digital, film, and smartphone cameras, each with specific strengths and limitations. Common issues include limited performance in low light and potential image distortion.

26. **Stereo Camera and Depth Camera:** Stereo cameras use two or more lenses to capture images from slightly different perspectives, enabling the calculation of depth information and creating 3D representations. Depth cameras, such as those using structured light or time-of-flight, measure distance by analyzing light patterns or time delays. Limitations include challenges in varying lighting conditions and potential inaccuracies in depth measurement.

Part II

Small Projects

Project 1

Working With Polynomials

1.1 What is a Polynomial?

A **polynomial** is a special type of math expression that involves numbers and variables. Here's what you need to know:

- **Variable:** A letter that stands for a number, like x or y.
- **Coefficient:** A number that multiplies the variable, like 3 in $3x$.
- **Exponent:** A number that shows how many times the variable is multiplied by itself, like 2 in x^2.

A polynomial is made up of terms, which are parts of the expression separated by plus (+) or minus (−) signs.

1.2 Examples of Polynomials

Here are some examples of polynomials:

- $3x^2 + 2x - 5$
- $4y^3 - 7y + 1$
- $-2x + 7$

In these examples:

- $3x^2 + 2x - 5$ has three terms: $3x^2$, $2x$, and -5.
- $4y^3 - 7y + 1$ has three terms: $4y^3$, $-7y$, and 1.
- $-2x + 7$ has two terms: $-2x$ and 7.

1.3 Polynomial and Calculations

For the polynomial:

$$y = 2x^3 - 3x^2 + x - 5$$

We calculate y for the given values of x:
Below is the plot of the polynomial and the given points:

x	y
-2	-35
-1	-11
0	-5
1	-5
2	1

Table 1.1: Values of y for the polynomial $y = 2x^3 - 3x^2 + x - 5$

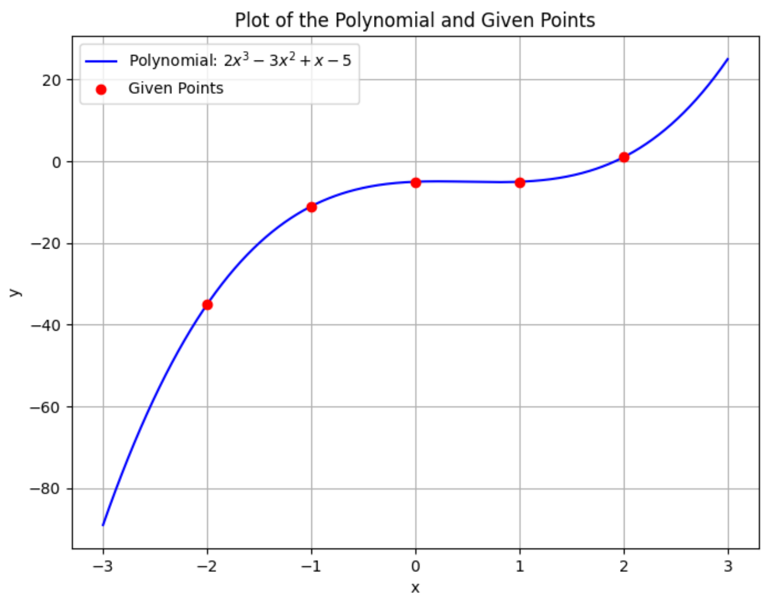

Figure 1.1: Plot of the polynomial $y = 2x^3 - 3x^2 + x - 5$ with given points.

1.4 What is a First-Order Polynomial?

A **first-order polynomial** is a polynomial of degree 1. It has the general form:

$$y = mx + b$$

where:

- m is the slope of the line, which indicates how steep the line is.
- b is the y-intercept, which is the point where the line crosses the y-axis.

The graph of a first-order polynomial is a straight line.

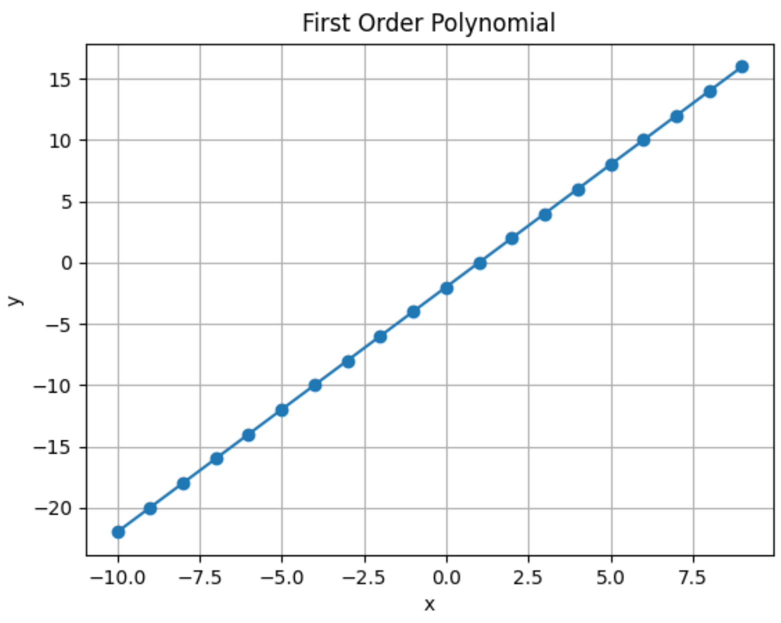

Figure 1.2: Plot of the first-order polynomial $y = 2x - 2$.

1.4.1 Python Code Example

Here is a Python code snippet to plot a first-order polynomial using Matplotlib:

Listing 1.1: Python code to plot a first-order polynomial

```python
import matplotlib.pyplot as plt

def first_order_polynomial(X: list[int], a: float, b: float) -> list[float]:
    Y: list[float] = []
    for x in X:
        y = a * x + b
        Y.append(y)
    return Y

x = list(range(-10, 10))
y = first_order_polynomial(x, 2, -2)

# plotting the points
plt.plot(x, y, "-o", label="Line")
plt.title("First Order Polynomial")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

1.5 Visualization

Below is the plot of the first-order polynomial generated by the Python code:

1.6 Second Order Polynomial (Parabola)