COURSE NOTES

SELF EDUCATION EDITION

PYTHON

WRITTEN BY

ANDRII VOZNESENSKYI

Python Course Notes

Course Instructor:

Associate Engineer, Andrii Voznesenskyi

Mathematics and Information Systems Faculty
Warsaw University of Technologies

May 20, 2023

License:

Purpose:

The primary purpose of this document is to serve as an educational guide and reference for students or individuals who are eager to expand their knowledge and understanding about Python programming language, specifically with regards to variables, types, lists, tuples, and dictionaries. The problems and solutions herein are intended to help individuals improve their coding skills, analytical thinking and problem-solving abilities. It is not intended for any commercial use.

This document is intended for the exclusive use of colleagues or students seeking to enhance their individual understanding of the course and engage in self-education. It is of utmost importance to strictly adhere to the restrictions and copyright information provided within this document.

"Education is the kindling of a flame, not the filling of a vessel."
Socrates, Ancient Greek philosopher

Contents

Chapter 1

Introduction

Python is a high-level, interpreted dynamically-typed programming language. It was created by Guido van Rossum and first released in 1991. Python is designed to be highly readable, with a simple and consistent syntax that promotes readability and therefore reduces the cost of program maintenance. Now let's take a deeper dive into it.

If you've chosen to read this book, you might already have some understanding of Python and its significance as a valuable tool for learning. If not, you will likely discover the value of Python as you progress through this book and engage in a few projects. Before delving into the details, let's briefly explore why Python has gained popularity. In this chapter, we will adopt a question-and-answer format to address common queries posed by beginners.

1.1 What Makes Python a Preferred Choice?

Given the abundance of programming languages available today, this question naturally arises for newcomers. With approximately 1 million Python users at present, it is impossible to provide a definitive answer. The choice of programming tools often depends on unique constraints or personal preferences.

However, after teaching Python to numerous groups comprising over 3,000 students in the past 12 years, certain common themes have emerged. The primary factors that attract Python users can be summarized as follows:

1.1.1 Software quality

Many individuals appreciate Python's emphasis on readability, coherence, and overall software quality, setting it apart from other scripting languages. Python code is designed to be easily readable, promoting reusability and maintainability. Its consistent structure enables easy comprehension, even for those who didn't author it. Additionally, Python offers robust support for advanced software reuse mechanisms, such as object-oriented programming (OOP).

1.1.2 Developer productivity

Python significantly enhances developer productivity compared to compiled or statically typed languages like C, C++, and Java. Python code typically requires one-third to one-fifth the amount of code compared to equivalent C++ or Java code. This means less

typing, reduced debugging, and easier maintenance. Python programs also execute instantly, without the lengthy compile and link steps required by some other tools, thereby accelerating programmer speed.

1.1.3 Program portability

Most Python programs run flawlessly across major computer platforms. Transferring Python code between Linux and Windows, for instance, usually involves a simple copy-paste operation. Moreover, Python offers multiple options for developing portable graphical user interfaces, database access programs, web-based systems, and more. Even operating system interfaces, including program launches and directory processing, are highly portable in Python.

1.1.4 Support libraries

Python comes with an extensive standard library, which provides a wide range of prebuilt and portable functionality. This library facilitates various application-level programming tasks, from text pattern matching to network scripting. Additionally, Python can be extended through both custom libraries and a vast collection of third-party application support software. The Python ecosystem offers tools for website development, numerical programming, serial port access, game development, and much more. The NumPy extension, for example, is often regarded as a free and more powerful alternative to the Matlab numeric programming system.

1.1.5 Component integration

Python scripts seamlessly integrate with other components of an application, thanks to a variety of integration mechanisms. This integration capability enables Python to serve as a tool for customizing and extending products. Presently, Python code can invoke C and C++ libraries, be called from C and C++ programs, integrate with Java and .NET components, communicate through frameworks like COM, interface with devices via serial ports, and interact over networks using interfaces such as SOAP, XML-RPC, and CORBA. Python is not a standalone tool but an interconnected part of the software landscape.

1.1.6 Enjoyment

Python's user-friendly nature and built-in toolset make programming a more enjoyable experience than a tedious chore. Although this aspect may be intangible, its positive impact on productivity is a significant advantage.

Among these factors, the most compelling benefits for most Python users are software quality and increased productivity.

1.1.7 Okay, but What's the Downside?

After using Python for 17 years and teaching it for 12, the only downside I've found is that, as currently implemented, its execution speed may not always match that of compiled languages like C and C++.

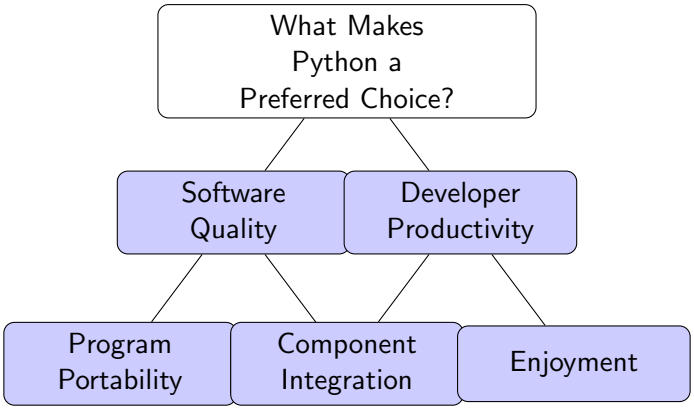

Figure 1.1: Factors that make Python a preferred choice

We will discuss implementation concepts in detail later in this book. In short, the standard implementations of Python today compile source code statements to an intermediate format called byte code and then interpret the byte code. Byte code provides portability as a platform-independent format. However, since Python is not compiled all the way down to binary machine code, some programs may run more slowly in Python than in a fully compiled language like C.

Whether you will ever be concerned about the execution speed difference depends on the types of programs you write. Python has been optimized numerous times, and Python code runs fast enough in most application domains. Furthermore, when performing "real" tasks in a Python script, such as processing a file or constructing a graphical user interface (GUI), your program will run at C speed, as such tasks are immediately dispatched to compiled C code inside the Python interpreter. Moreover, Python's speed of development often outweighs any loss in speed of execution, particularly given modern computer speeds.

However, there are some domains that require optimal execution speeds, such as numeric programming and animation. In such cases, you can still use Python by splitting off the parts of the application that require optimal speed into compiled extensions and linking them into your system for use in Python scripts.

We won't delve into extensions much in this text, but this is essentially an instance of Python serving as a control language, as discussed earlier. A prime example of this dual-language strategy is the NumPy numeric programming extension for Python. By combining compiled and optimized numeric extension libraries with the Python language, NumPy turns Python into an efficient and easy-to-use numeric programming tool. You may not need to code such extensions in your own Python work, but they provide a powerful optimization mechanism if the need arises.

1.2 Who Uses Python Today?

At present, the best estimate of the size of the Python user base is that there are roughly 1 million Python users worldwide. This estimate is based on various statistics, such as download rates and developer surveys. Due to its open-source nature, obtaining an exact count is challenging, as there are no license registrations to tally. Additionally, Python is automatically included with Linux distributions, Macintosh computers, and some products and hardware, further complicating the user-base picture.

However, it is evident that Python enjoys a large user base and a very active developer community. Python has been around for approximately 19 years, widely used, and proven to be stable and robust. Besides individual users, Python is being applied in revenue-generating products by real companies. Some notable examples include:

- Google, which extensively uses Python in its web search systems and employs Python's creator.
- The YouTube video sharing service, which is largely written in Python.
- The popular BitTorrent peer-to-peer file sharing system, which is a Python program.
- Google's App Engine web development framework, which uses Python as its application language.
- EVE Online, a Massively Multiplayer Online Game (MMOG), which makes extensive use of Python.
- Maya, a powerful integrated 3D modeling and animation system, which provides a Python scripting API.
- Various companies like Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM, which use Python for hardware testing.
- Industrial Light & Magic, Pixar, and others, which use Python in the production of animated movies.
- Financial institutions like JPMorgan Chase, UBS, Getco, and Citadel, which apply Python for financial market forecasting.
- NASA, Los Alamos, Fermilab, JPL, and others, which use Python for scientific programming tasks.
- iRobot, which uses Python to develop commercial robotic devices.
- ESRI, which uses Python as an end-user customization tool for its popular GIS mapping products.
- The NSA, which uses Python for cryptography and intelligence analysis.
- The IronPort email server product, which uses more than 1 million lines of Python code.
- The One Laptop Per Child (OLPC) project, which builds its user interface and activity model in Python.

This list is far from exhaustive, but it serves to illustrate Python's wide range of application domains. Python's general-purpose nature makes it applicable to almost all fields, not just one. In fact, it's safe to say that virtually every substantial organization writing software is using Python, whether for short-term tactical tasks, such as testing and administration, or for long-term strategic product development.

For further details on companies using Python today, you can visit Python's website at `http://www.python.org`.

1.3 What Can You Do with Python?

In addition to being a well-designed programming language, Python is useful for accomplishing real-world tasks?tasks that developers face every day. It is commonly used in various domains as a tool for scripting, integrating components, and implementing standalone programs. Python's applications are virtually unlimited, from website development and gaming to robotics and spacecraft control.

However, the most common Python roles today can be classified into a few broad categories. The following sections provide a brief overview of some of Python's most popular applications, along with the tools used in each domain. Keep in mind that we won't explore these tools in depth here; if you are interested in any of these topics, you can refer to the Python website or other resources for more details.

1.3.1 Systems Programming

Python's built-in interfaces to operating system services make it ideal for writing portable and maintainable system administration tools and utilities, often referred to as shell tools. Python programs can search files and directory trees, launch other programs, perform parallel processing with processes and threads, and more.

Python's standard library includes POSIX bindings and supports various OS tools, such as environment variables, files, sockets, pipes, processes, threads, regular expression pattern matching, command-line arguments, standard stream interfaces, shell-command launchers, filename expansion, and more. Additionally, most of Python's system interfaces are designed to be portable. For example, a script that copies directory trees usually runs unchanged on all major Python platforms. The Stackless Python system, used by EVE Online, offers advanced solutions for multiprocessing requirements.

1.3.2 GUIs

Python's simplicity and rapid development cycle make it well-suited for graphical user interface (GUI) programming. Python includes a standard object-oriented interface to the Tk GUI API called tkinter (Tkinter in Python 2.6). This interface allows Python programs to create portable GUIs with a native look and feel. Python/tkinter GUIs run on Microsoft Windows, X Windows (Unix and Linux), and Mac OS (both Classic and OS X). The PMW extension package adds advanced widgets to the tkinter toolkit. Additionally, the wxPython GUI API, based on a C++ library, provides an alternative toolkit for creating portable GUIs in Python.

Higher-level toolkits like PythonCard and Dabo are built on top of base APIs such as wxPython and tkinter. With the appropriate libraries, you can also utilize GUI support from other toolkits in Python, such as PyQt for Qt, PyGTK for GTK, PyWin32 for MFC, IronPython for .NET, and Jython (the Java version of Python, described in Chapter 2) or JPype for Swing. For web-based applications or simple interface requirements, Jython and Python web frameworks, along with server-side CGI scripts, offer additional options for user interface development.

1.3.3 Internet Scripting

Python provides standard internet modules that enable programs to perform a wide range of networking tasks in client and server modes. Scripts can communicate over

sockets, extract form information from server-side CGI scripts, transfer files via FTP, parse and generate XML files, send, receive, compose, and parse emails, fetch web pages via URLs, parse HTML and XML from web pages, communicate over XML-RPC, SOAP, and Telnet, and much more. Python's libraries simplify these tasks significantly.

Additionally, numerous third-party tools are available on the web for internet programming in Python. For example, the HTMLGen system generates HTML files from Python class-based descriptions, the mod_python package efficiently runs Python within the Apache web server and supports serverside templating with its Python Server Pages, and the Jython system enables seamless Python/Java integration and server-side applet coding for clients.

Moreover, full-fledged web development framework packages for Python, such as Django, TurboGears, web2py, Pylons, Zope, and WebWare, allow rapid construction of feature-rich, production-quality websites. Many of these frameworks include object-relational mappers, a Model/View/Controller architecture, serverside scripting and templating, and AJAX support, providing complete and enterpriselevel web development solutions.

1.3.4 Component Integration

Python's ability to be extended by and embedded in C and C++ systems makes it useful as a flexible glue language for scripting the behavior of other systems and components. For instance, integrating a C library into Python allows Python to test and launch the library's components. Similarly, embedding Python in a product facilitates on site customizations without recompiling the entire product or shipping its source code.

Tools like SWIG and SIP can automate much of the work required to link compiled components into Python for use in scripts, while the Cython system enables developers to mix Python and Clike code. Larger frameworks, such as Python's COM support on Windows, the Jython Java-based implementation, the IronPython .NETbased implementation, and various CORBA toolkits for Python, provide alternative ways to script components. On Windows, for instance, Python scripts can use frameworks to script Word and Excel.

1.3.5 Database Programming

Python provides interfaces to commonly used relational database systems, including Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, and SQLite, among others. The Python community has defined a portable database API that allows Python scripts to access SQL database systems using a consistent interface. This means that a script developed for the free MySQL system will often work unchanged on other systems, such as Oracle, by simply replacing the underlying vendor interface.

Python's standard pickle module offers a simple object persistence system, allowing programs to easily save and restore Python objects to files and file-like objects. Additionally, third-party open-source systems like ZODB provide complete object-oriented database systems for Python scripts, and others, such as SQLObject and SQLAlchemy, map relational tables onto Python's class model. Furthermore, starting from Python 2.5, the in-process SQLite embedded SQL database engine is included as a standard part of Python.

1.3.6 Rapid Prototyping

Python programs can interact with components written in both Python and C, treating them the same way. This allows developers to prototype systems using Python and subsequently move selected components to a compiled language like C or C++ for delivery. Unlike some prototyping tools, Python doesn't necessitate a complete rewrite once the prototype solidifies. Python can coexist seamlessly with compiled languages, enabling developers to optimize performance-critical sections while retaining other parts of the system in Python for ease of maintenance and use.

1.3.7 Numeric and Scientific Programming

The NumPy numeric programming extension for Python is one of the most notable tools in this domain. It includes an array object, interfaces to standard mathematical libraries, and more. By integrating Python with numeric routines coded in a compiled language for speed, NumPy transforms Python into a sophisticated yet easy-to-use numeric programming tool that can often replace code written in traditional compiled languages such as FORTRAN or C++. Additional numeric tools for Python support animation, 3D visualization, parallel processing, and other tasks. The SciPy and ScientificPython extensions, for example, provide additional libraries for scientific programming and leverage NumPy functionality.

1.3.8 Gaming, Images, Serial Ports, XML, Robots, and More

Python finds applications in numerous other domains as well. For instance, you can use Python for game programming and multimedia with the pygame system, serial port communication on various platforms with the PySerial extension, image processing with libraries like PIL, PyOpenGL, Blender, and Maya, robot control programming with the PyRo toolkit, XML parsing with the xml library package, the xmlrpclib module, and third-party extensions, artificial intelligence programming with neural network simulators and expert system shells, and natural language analysis with the NLTK package. There are even programs like PySol that allow you to play solitaire. Support for many of these fields can be found on the Python Package Index (PyPI) website and through web searches.

This list is by no means exhaustive, but it demonstrates the wide range of applications Python can handle. Python's ability to integrate components and its general-purpose nature make it applicable in a wide variety of domains.

1.4 How Is Python Supported?

Python is an open-source system with a large and active development community that responds quickly to issues and develops enhancements at an impressive pace. Python developers coordinate their work online using a source-control system. Changes to the language follow a formal Python Enhancement Proposal (PEP) protocol and undergo scrutiny from other developers and the BDFL (Benevolent Dictator For Life), Guido van Rossum, the creator of Python.

The Python community is known for its rapid response to user queries and issues, providing support that many commercial software help desks would find challenging to

match. Python's complete source code is available, empowering developers to study or modify the language's implementation as needed. Unlike commercial software, Python is not subject to the whims of a single vendor, and its ultimate documentation source is accessible to all.

The Python Software Foundation (PSF), a formal nonprofit organization, plays a key role in organizing conferences and addressing intellectual property issues. Python conferences, such as O'Reilly's OSCON and the PSF's PyCon, provide platforms for developers to share knowledge and experiences. PyCon, in particular, has experienced significant growth in recent years, attracting a large number of attendees. The Python community remains highly active and engaged in the ongoing development and support of the language.

1.5 What Are Python's Technical Strengths?

Python boasts several technical strengths that make it a popular choice among developers. Here are some of its key features:

Object-Oriented: Python is an object-oriented language that supports advanced concepts like polymorphism, operator overloading, and multiple inheritance. Despite these powerful features, Python's simple syntax and typing make it easy to apply object-oriented programming (OOP) principles. Python allows developers to use OOP when necessary but also supports procedural programming. This flexibility makes Python an ideal scripting tool for object-oriented systems languages like C++ and Java.

Free and Open Source: Python is completely free to use and distribute. It is an open-source language, meaning its complete source code is available for free on the internet. There are no restrictions on copying, embedding, or shipping Python with your products. Python's open-source nature also means that it benefits from a large and active development community, which continuously enhances and supports the language.

Portability: The standard implementation of Python is written in portable ANSI C and runs on virtually every major platform. Python programs can be executed on a wide range of systems, from Linux and Windows to Mac OS, BeOS, and even gaming consoles and cell phones. Python's standard library modules are designed to be portable across platforms, enabling developers to write code that runs consistently on different systems.

It's worth noting that Python's portability extends to its byte code, which is a platform-independent format. Python programs are compiled to byte code, which can be executed by any Python interpreter on any platform. This allows for easy distribution and execution of Python programs across different systems.

Expressive and Readable Syntax: Python's syntax is designed to be highly readable and expressive. It emphasizes code readability, making it easier for developers to write and maintain Python programs. Python code is often referred to as "executable pseudo-code" due to its natural language-like syntax. The use of whitespace indentation for code blocks, rather than curly braces or keywords, further enhances the readability of Python code.

Large Standard Library: Python comes with a large standard library that provides a wide range of functionality. The standard library includes modules for file I/O, network programming, database access, GUI development, regular expressions, unit testing, web scraping, and much more. This extensive collection of modules allows developers to leverage prebuilt functionality and accelerate the development process.

Third-Party Libraries and Ecosystem: In addition to its standard library, Python has a vibrant ecosystem of third-party libraries and frameworks. The Python Package Index (PyPI) hosts thousands of open-source packages that extend Python's capabilities in various domains. These packages cover areas such as web development, data analysis, machine learning, scientific computing, game development, and more. The availability of a rich set of third-party libraries allows developers to find and integrate solutions quickly, saving time and effort.

1.6 Conclusion

Python is a powerful and versatile programming language that has gained popularity due to its simplicity, readability, and extensive ecosystem. It offers numerous benefits, such as high software quality, developer productivity, program portability, support libraries, component integration, and an enjoyable programming experience. Python is widely used in various domains, including web development, scientific programming, system administration, GUI programming, and more. It is supported by a large and active community, providing rapid responses and continuous development. Python's technical strengths, such as being object-oriented, free and open source, portable, and having an expressive syntax, contribute to its widespread adoption and success in the software development industry.

We hope this introduction has piqued your interest in Python and its capabilities. Whether you're a beginner or an experienced programmer, Python offers a powerful and enjoyable environment for developing a wide range of applications. Happy coding!

Chapter 2

Installation

Python is a versatile programming language that can be installed on various operating systems, including Windows, macOS, and Linux. In this chapter, we will explore the installation process for each of these platforms.

To install Python, you can download the latest version from the official Python website. Python can run on a variety of platforms including Windows, Mac, and various distributions of Linux. There are also several distributions of Python for specific purposes like Anaconda, which is focused on data analysis and scientific computing.

2.1 Python on Linux

Many Linux distributions come with Python pre-installed, but the versions might vary. To install the latest version or manage different Python versions, you can use package managers like `apt` (for Debian-based distributions) or `dnf` (for Fedora-based distributions). Here's an example of installing Python on Ubuntu using `apt`:

1. Open a terminal: `Ctrl + Alt + T` key shortcut was originally the one to open a terminal window in the Arch distributions.

2. Update package lists: Enter the following command to update the package lists:

```
sudo apt update
```

3. Install Python: Enter the following command to install Python 3:

```
sudo apt install python3
```

4. Verify the installation: Once the installation is complete, you can check the installed version by entering the command `python3 --version`.

Note that the package manager commands may differ depending on your Linux distribution. Refer to your distribution's documentation for the specific instructions.

2.2 Python on macOS

macOS usually comes with a pre-installed version of Python. However, it is recommended to install the latest version to ensure compatibility with the latest features and libraries. Follow these steps to install Python on macOS:

1. Check the installed version: Open the Terminal application by searching for it in Spotlight. Then, enter the command `python --version` and press Enter. If a Python version is displayed (e.g., Python 3.9.6), it means Python is already installed. You can skip the installation if the version is up to date.

2. Download the Python installer: Visit the official Python website at `python.org/downloads` using a web browser. Look for the macOS installer section and click on the Python 3 download button.

3. Run the installer: Once the installer is downloaded, double-click on it to start the installation process. Follow the on-screen instructions, and make sure to select the option to install Python for all users and to add Python to the system's PATH.

4. Verify the installation: After the installation is complete, open a new Terminal window and enter the command `python --version` again. It should display the newly installed Python version, confirming a successful installation.

2.3 Python on Windows

Windows systems do not come with Python pre-installed, so we need to download and install it manually. Here's a step-by-step guide to installing Python on Windows:

1. Check if Python is already installed: Open a command prompt by pressing the Windows key and typing "Command Prompt." Then, enter the command `python` and press Enter. If Python is installed, you will see a Python prompt (`>>>`). Otherwise, you will see an error message indicating that Python is not recognized as a command.

2. Download the Python installer: Go to the official Python website at `python.org/downloads` using a web browser. You will find two buttons labeled "Download Python 3" and "Download Python 2". It is recommended to choose the Python 3 version unless you have specific reasons to use Python 2. Click on the Python 3 button to download the installer.

3. Run the installer: Once the installer is downloaded, double-click on it to run the installation wizard. Make sure to select the option "Add Python to PATH" during the installation process. This option ensures that Python is added to the system's environment variables, allowing you to run Python from any location in the command prompt.

4. Verify the installation: After the installation is complete, open a new command prompt and enter the command `python` again. This time, it should launch the Python prompt without any errors, indicating a successful installation.

2.4 IDE (Integrated Development Environment)

An IDE is a software application that provides a comprehensive environment for writing, debugging, and running code. While Python can be written using any text editor, using an IDE can enhance your productivity and provide useful features like code completion, debugging tools, and project management.

One popular IDE for Python development is Visual Studio Code (VSCode). It is a free and open-source IDE developed by Microsoft. Here are some reasons why you might consider using VSCode for your Python programming:

- **Cross-platform**: VSCode is available for Windows, macOS, and Linux, making it a versatile choice regardless of your operating system.

- **Extensible**: VSCode has a rich extension ecosystem, allowing you to customize and enhance your development experience. There are numerous Python-related extensions available that provide additional features and integrations.

- **Intelligent coding assistance**: VSCode offers features like code completion, syntax highlighting, linting, and code formatting, which can help you write cleaner and error-free code.
- **Integrated terminal**: VSCode provides an integrated terminal within the IDE, allowing you to run Python code and execute commands without switching to an external terminal.
- **Version control integration**: VSCode has built-in support for version control systems like Git, making it easy to manage and track changes in your code.

To install Visual Studio Code (VSCode) as your IDE for Python programming, you can follow these steps:

1. Visit the official VSCode website at `https://code.visualstudio.com/` using a web browser.
2. Download the appropriate installer for your operating system (Windows, macOS, or Linux).
3. Run the installer executable and follow the instructions to complete the installation.
4. Once the installation is complete, launch VSCode.

To start coding in Python using VSCode, you can install the Python extension by Microsoft. Here's how:

1. Open VSCode and click on the Extensions icon on the sidebar (or use the shortcut Ctrl+Shift+X).
2. Search for "Python" in the Extensions Marketplace.
3. Click on the "Python" extension by Microsoft and click on the "Install" button.
4. After the installation is complete, you can use VSCode to write, debug, and run Python code.

Remember that choosing an IDE is a personal preference, and there are many other IDEs available for Python development. You can explore different options and select the one that suits your needs and workflow the best.

Chapter 3

Your First Python Program

Starting out in programming can be challenging. The syntax, concepts, and logic involved may seem overwhelming at first. The Python language, with its unique features and flexibility, can sometimes add to the confusion. However, like any other skill, programming can be mastered with dedication and hard work.

In this chapter, we will guide you through writing your first Python program and introduce you to some key points of the language. Remember, even the most experienced programmers were once beginners, facing similar obstacles. With determination and perseverance, you can overcome any initial difficulties and unlock the vast world of programming.

Here's how you can write your first Python program. This simple program outputs the string "Hello, World!" to the console:

Example 3.0: For performing the example below with a full correctness, create a new file with the name `my_first_program_in_python.py` and inside the file created write the following content of the example:

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

You may easily save the written program=, as most of IDE support `Ctrl + S` shortcut and you may than run the program from the terminal by using the next script: `python3 my_first_program_in_python.py`. At all next chapters the information about the program execution will not be mentioned in cate the program is believed to be quite simple.

3.1 The print() function

The print() function is a built-in function in Python that is used to output or display information to the console or terminal. It takes one or more arguments (values or expressions) and displays them as text.

In the example above, the print() function is used to display the string "Hello, World!" to the console. The string is enclosed in double quotation marks, indicating that it is a string literal.

You can also use single quotation marks to enclose strings. Both double and single quotation marks are valid ways to define strings in Python. In the example 3.0 you might note that the only one way of the quotation marks for the string argument was shown. Below you may also see the other ways of the usage:

Example 3.1: For performing the example below with a full correctness, create a new file with the name `my_first_program_in_python.py` and inside the file created write the following content of the example:

```
1 print('Hello, World!')
2 print('''Hello, World!''')
3 print("""Hello, World!""")
```

In the updated code of example 3.0 expansion, three different ways of using quotation marks for defining strings are shown:

- The first example uses single quotation marks `('Hello, World!')` to enclose the string.
- The second example uses triple single quotation marks `('''Hello, World!''')` to enclose the string. This allows the string to span multiple lines.
- The third example uses triple double quotation marks `("""Hello, World!""")` to enclose the string. Similar to triple single quotation marks, this also allows the string to span multiple lines.

All three ways are valid in Python, and you can choose the one that suits your preference or the requirements of your code.

3.2 Using f-strings with the print() function

In addition to printing simple strings, you can use f-strings (formatted string literals) with the print() function to dynamically format and display values within a string.

To use an f-string, you prefix the string with the letter f or F and enclose the expression you want to evaluate within curly braces .

Here's an example that demonstrates the usage of an f-string with the print() function:

Example 3.2:

```
1 name = "Alice"
2 age = 25
3 print(f"My name is {name} and I am {age} years old.")
```

When this program is executed, it will output the following text to the console:

```
My name is Alice and I am 25 years old.
```

In the f-string, the expressions name and age are evaluated and their values are inserted into the string at the respective positions.

F-strings are a powerful feature in Python that allow you to easily format strings with variables or expressions. They provide a concise and readable way to combine text and values in output statements.

3.3 Using the print() function's properties

The print() function in Python has some useful properties that can be used to modify its behavior. Here are a few commonly used properties:

- **sep**: This property allows you to specify the separator between multiple arguments passed to the print() function. By default, the separator is a space character. You can change it by assigning a different value to print()'s sep property.
- **end**: This property allows you to specify the string that should be printed at the end of the print() function's output. By default, the end property is set to a newline character (), which causes the next output to appear on a new line. You can change it by assigning a different value to print()'s end property.
- **file**: This property allows you to redirect the output of the print() function to a file or a different output stream instead of the default standard output (console). By default, the file property is set to sys.stdout, which represents the standard output. You can change it by assigning a different file object or output stream to print()'s file property.

Here's an example that demonstrates the usage of these properties:

Example 3.3:

```
1 name = "Alice"
2 age = 25
3 output_file = open("output.txt", "w")
4
5 print(f"My name is {name}", end=", ")
6 print(f"and I am {age} years old.", file=output_file, sep="|"
    )
7
8 output_file.close()
```

In this example, we have set the end property of the first print() statement to ", " to print a comma followed by a space instead of the default newline character. We have also set the file property of the second print() statement to output_file, which is a file object representing a file named "output.txt". Additionally, we have set the sep property of the second print() statement to "|" to separate the two arguments with a pipe character. Actually, the reading and writing to the files operation will be covered with more precision a bit later. The author would like you also to note the the words "`name`" and "`age`" are some specific words called variables, the whole sense of which will be covered in the next chapter.

When this program is executed, it will print the following text to the console:

```
My name is Alice, and I am 25 years old.
```

It will also create a file named "output.txt" with the following content:

```
My name is Alice|and I am 25 years old.
```

These properties provide flexibility in controlling the output format and destination when using the print() function in Python. The example 3.3. is provided here to make the reader's understandance quite a bit wider for a while, in this place the feeling of disorientation and unfavorable desire to learn is expected, however wait for a short while and you will know that there is nothing to be like the elephant out of the room.

3.4 Using the print().format() function in Python

The `print().format()` function in Python is a versatile tool for formatting strings. It allows you to dynamically insert values into a string and control the formatting of those values. This can be incredibly useful when you need to create well-formatted and customizable output. Let's explore the capabilities of this function with some examples:

1. Basic usage:

```
print("Hello, {}. You are {} years old.".format("Alice", 25))
```

Output: "Hello, Alice. You are 25 years old."

In this example, the curly braces act as placeholders, which will be replaced by the values provided in the `format()` method, in the order they appear. This allows you to dynamically insert values into a string.

2. Positional arguments:

```
print("{1}, {0}".format("world", "Hello"))
```

Output: "Hello, world". The numbers inside the curly braces represent the index of the arguments in the `format()` method. By specifying the index of each argument, you can control their order in the resulting string. This is especially useful when you want to rearrange the values or reuse them multiple times.

3. Named arguments:

```
print("{greeting}, {name}".format(greeting="Hello", name="world"))
```

Output: "Hello, world". The strings inside the curly braces represent the names of the arguments in the `format()` method. Using named arguments allows you to specify values based on their corresponding names, rather than their positions. This provides clarity and flexibility when dealing with complex formatting scenarios.

The general syntax for the `print().format()` function is as follows:

```
print(<format_string>.format(<arguments>))
```

Here, `<format_string>` is the string containing the placeholders and desired formatting, and ¡arguments¿ are the values or variables that will be inserted into the placeholders. The placeholders can be positioned by index or named, giving you fine-grained control over the arrangement of values within the output string.

By utilizing the `print().format()` function, you can achieve precise and customizable string formatting, making it a powerful tool for string manipulation and display in Python. This flexibility empowers you to create visually appealing and informative outputs, enhancing the overall quality and professionalism of your Python programs.

3.5 Problems

1. Write a Python program that prints your name.
2. Write a Python program that prints the sum of two numbers.
3. Write a Python program that prints the product of three numbers.
4. Write a Python program that prints the result of dividing two numbers.
5. Write a Python program that prints the remainder of dividing two numbers.
6. Write a Python program that prints the square root of a number.
7. Write a Python program that prints the absolute value of a number.
8. Write a Python program that prints a sentence using the f-string format.
9. Write a Python program that prints the result of an arithmetic expression using f-strings.
10. Write a Python program that prints a message using the .format() method.
11. Write a Python program that prints the result of an arithmetic expression using the .format() method.
12. Write a Python program that prints a sentence with multiple placeholders using f-strings.
13. Write a Python program that prints a sentence with named placeholders using the .format() method.
14. Write a Python program that prints a formatted date using f-strings.
15. Write a Python program that prints a formatted time using the .format() method.
16. Write a Python program that prints a formatted floating-point number using f-strings.
17. Write a Python program that prints a formatted integer using the .format() method.
18. Write a Python program that prints a sentence with multiple formatted values using f-strings.

19. Write a Python program that prints a sentence with multiple formatted values using the .format() method.

20. Write a Python program that prints a complex sentence with multiple placeholders and formatted values using f-strings and the .format() method.

Chapter 4

Variables and Types

Those acquainted with more basic languages like C or C++, are well aware that a significant part of their work involves creating objects or data structures, representing components in their applications. This involves the painstaking work of structuring memory, administering memory allocation, crafting search and access routines, amongst other things. These tasks are indeed as monotonous and error-prone as they sound, and they tend to divert focus from the actual objectives of your program.

However, in Python, much of this laborious work becomes unnecessary. This is owing to Python's in-built powerful object types that are integral to the language. This means that there is seldom a need to code object implementations prior to solving problems. Indeed, unless you require special processing that in-built types do not offer, it is often better to use an in-built object rather than creating your own. Here are some reasons:

- In-built objects simplify programming. For basic tasks, in-built types are frequently all you require to represent the structure of problem domains. Since collections such as lists and search tables such as dictionaries are immediately available, they can be employed straight away. A great deal of work can be accomplished using Python's in-built object types alone.

- In-built objects serve as components of extensions. For more intricate tasks, you might need to provide your own objects using Python classes or C language interfaces. However, as we will see later, objects implemented manually are often built on top of in-built types such as lists and dictionaries. For instance, a stack data structure may be implemented as a class that manages or customises a built-in list.

- In-built objects tend to be more efficient than custom data structures. Python's in-built types utilise optimised data structure algorithms which are implemented in C for speed. Even though you can write similar object types on your own, achieving the performance level that in-built object types provide can be quite challenging.

- In-built objects are a standard part of the language. Python to some extent borrows from languages that rely on built-in tools (e.g., LISP) and languages that depend on the programmer to provide tool implementations or frameworks of their own (e.g., C++). Even though you can implement unique object types in Python, you do not need to do so just to get started. Furthermore, as Python's built-ins are standard, they are always the same; proprietary frameworks, conversely, tend to vary from site to site.

To sum up, not only do built-in object types simplify programming, but they are also more potent and efficient than most that can be created from scratch. Regardless of whether you implement new object types, built-in objects form the backbone of every Python program.

In Python, variables are created when you assign a value to them. Python is dynamically typed, which means that you don't need to specify the type of a variable when you declare it. Here's an example of variable assignment in Python:

Example 4.0:

```
x = 5
y = "Hello, World!"
```

In this example, x is an integer and y is a string.

Variables in Python can store values of different types, such as integers, floats, strings, booleans, lists, tuples, and dictionaries. The type of a variable can change dynamically based on the value assigned to it. Python provides built-in functions to determine the type of a variable, such as the `type()` function.

For example, consider the following code snippet:

Example 4.0:

```
x = 5
print(type(x)) # Output: <class 'int'>

x = "Hello, World!"
print(type(x)) # Output: <class 'str'>
```

In this code, the `type()` function is used to determine the type of the variable `x` before and after reassignment. The output shows that `x` changes from an integer (`int`) to a string (`str`).

Python also supports type hints, which allow you to specify the expected type of a variable. Although these hints are not enforced by the Python interpreter, they can be useful for documentation and code readability. Type hints can be added using the colon (`:`) followed by the type after the variable name.

Example 4.0: Here's an example:

```
x: int = 5
y: str = "Hello, World!"
```

In this code, the type hints indicate that `x` should be an integer and `y` should be a string. While these hints are optional, they can help make your code more understandable and catch potential type-related errors early on.

Table 4.1: Built-in objects preview

Object type	**Example literals/creation**
Numbers	1234, 3.1415, 3+4j, Decimal, Fraction
Strings	'spam', "guido's", b'a01c'
Lists	[1, [2, 'three'], 4]
Dictionaries	{'food': 'spam', 'taste': 'yum'}
Tuples	(1, 'spam', 4, 'U')
Files	myfile = open('eggs', 'r')
Sets	set('abc'), 'a', 'b', 'c'
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

4.1 Python's Core Data Types

Table 4.1 previews Python's built-in object types and some of the syntax used to code their literals—that is, the expressions that generate these objects. Some of these types will probably seem familiar if you've used other languages; for instance, numbers and strings represent numeric and textual values, respectively, and files provide an interface for processing files stored on your computer.

Table 4.1 isn't really complete because everything we process in Python programs is a kind of object. For instance, when we perform text pattern matching in Python, we create pattern objects, and when we perform network scripting, we use socket objects. These other kinds of objects are generally created by importing and using modules and have behavior all their own.

As we'll see in later parts of the book, program units such as functions, modules, and classes are objects in Python too—they are created with statements and expressions such as `def`, `class`, `import`, and `lambda` and may be passed around scripts freely, stored within other objects, and so on. Python also provides a set of implementation-related types such as compiled code objects, which are generally of interest to tool builders more than application developers; these are also discussed in later parts of this text.

We usually refer to the other object types in Table 4.1 as core data types because they are effectively built into the Python language—that is, there is specific expression syntax for generating most of them. For instance, when you run the following code:

```
1    'spam'
```

you are, technically speaking, running a literal expression that generates and returns a new string object. There is specific Python language syntax to make this object. Similarly, an expression wrapped in square brackets makes a list, one in curly braces makes a dictionary, and so on. Even though, as we'll see, there are no type declarations in Python, the syntax of the expressions you run determines the types of objects you create and use. In fact, object-generation expressions like those in Table 4.1 are generally where types originate in the Python language.

Just as importantly, once you create an object, you bind its operation set for all time—you can perform only string operations on a string and list operations on a list. As you'll learn, Python is dynamically typed (it keeps track of types for you automatically instead of requiring declaration code), but it is also strongly typed (you can perform on

an object only operations that are valid for its type).

Functionally, the object types in Table 4.1 are more general and powerful than what you may be accustomed to. For instance, you'll find that lists and dictionaries alone are powerful data representation tools that obviate most of the work you do to support collections and searching in lower-level languages. In short, lists provide ordered collections of other objects, while dictionaries store objects by key; both lists and dictionaries may be nested, can grow and shrink on demand, and may contain objects of any type.

However, it's important to note that the table above represents only a subset of the object types available in Python. In the chapters that follow, we will delve into more general types and explore additional object types and concepts beyond those shown in Table 4.2. This will give us a comprehensive understanding of the rich set of data structures and types that Python provides.

So, let's continue our journey and explore more advanced and powerful object types in the upcoming chapters.

Table 4.2: Python Object Types

Type	Description	Example
int	Integers	`x = 5`
float	Floating-point numbers	`y = 3.14`
complex	Complex numbers	`z = 2 + 3j`
str	Strings	`s = "Hello, World!"`
list	Lists	`lst = [1, 2, 3]`
tuple	Tuples	`tup = (4, 5, 6)`
dict	Dictionaries	`person = {'name': 'John', 'age': 25}`
set	Sets	`s = {1, 2, 3}`
bool	Booleans	`a = True` `b = False`
bytes	Bytes	`bt = b'Hello'`
bytearray	Bytearrays	`ba = bytearray(b'World')`

Python's flexibility in handling different variable types and its support for dynamic typing make it a versatile language for various programming tasks.

Python supports several data types, including:

4.2 Numeric Types

Numeric Types:

- **int** - integers, such as 1, 2, -5, etc.

Example 4.2: The `int` type represents integers, which are whole numbers without a fractional part. For example:

```
x = 5
print(x) # Output: 5
```

- **float** - floating-point numbers, such as 3.14, -2.5, etc.

Example 4.2: The `float` type represents floating-point numbers, which have a decimal point. For example:

```
y = 3.14
print(y) # Output: 3.14
```

- **complex** - complex numbers, represented as *real + imaginary*, e.g., 2 + 3j.

Example 4.2: The `complex` type represents complex numbers, which consist of a real part and an imaginary part. For example:

```
z = 2 + 3j
print(z) # Output: (2+3j)
```

If you've done any programming or scripting in the past, some of the object types in Table 4-1 will probably seem familiar. Even if you haven't, numbers are fairly straightforward. Python's core objects set includes the usual suspects: integers (numbers without a fractional part), floating-point numbers (roughly, numbers with a decimal point in them), and more exotic numeric types (complex numbers with imaginary parts, fixed-precision decimals, rational fractions with a numerator and denominator, and full-featured sets).

Although Python offers some fancier options, its basic number types are fundamental. Numbers in Python support the normal mathematical operations. For instance, the plus sign (+) performs addition, the star (*) is used for multiplication, and two stars (**) are used for exponentiation:

Example 4.2:

```
print(123 + 222) # Integer addition
# Output: 345

print(1.5 * 4) # Floating-point multiplication
# Output: 6.0

print(2 ** 100) # 2 to the power 100
# Output: 1267650600228229401496703205376
```

Python's integer type automatically provides extra precision for large numbers when needed. You can compute even extremely large numbers as integers in Python, but be cautious when working with very large results as they may have a significant number of digits:

Example 4.2:

```
print(len(str(2 ** 1000000))) # How many digits in a really
   BIG number?
# Output: 301030
```

When working with floating-point numbers, you may encounter a display issue due to the limited precision of floating-point representations:

Example 4.2:

```
print(3.1415 * 2) # str: user-friendly
# Output: 6.283
```

Python provides two ways to print an object: with full precision (as in the first result shown here) and in a user-friendly form (as in the second). The difference can matter when dealing with classes, but for now, if something looks odd, try displaying it with the `print` statement.

Python also includes various numeric modules that offer additional tools for numerical operations. For example, the `math` module provides advanced mathematical functions, and the `random` module allows random number generation:

Example 4.2:

```
import math

print(math.pi)
# Output: 3.1415926535897931

print(math.sqrt(85))
# Output: 9.2195444572928871

import random

print(random.random())
# Output: 0.59268735266273953

print(random.choice([1, 2, 3, 4]))
# Output: 1
```

Python also includes more exotic numeric objects such as complex numbers, fixed-precision decimals, rational numbers, sets, and Booleans.

4.3 Sequence Types

Sequence Types:

- **str** - strings, a sequence of characters enclosed in single or double quotes, e.g., "Hello", 'World'.

Example 4.3:

```
1 s = "Hello, World!"
2 print(s) # Output: Hello, World!
```

Strings are immutable, meaning they cannot be modified after they are created. However, you can perform various operations on strings such as searching for substrings, replacing parts of the string, splitting the string into substrings, and more. String methods like `find()`, `replace()`, `split()`, and `upper()` can be used to manipulate and transform strings. In the Table 4.3 all the methods usage are clearly shown.

Table 4.3: String Methods

Method	Example	Explanation
`len()`	`len("Hello")`	Returns the length of the string.
`str.lower()`	`"Hello".lower()`	Converts all characters in the string to lowercase.
`str.upper()`	`"Hello".upper()`	Converts all characters in the string to uppercase.
`str.strip()`	`" Hello ".strip()`	Removes leading and trailing whitespace from the string.
`str.split()`	`"Hello,World".split(",")`	Splits the string into a list of substrings using the specified delimiter.
`str.find()`	`"Hello".find("l")`	Returns the index of the first occurrence of a substring in the string.
`str.replace()`	`"Hello".replace("H", "J")`	Replaces all occurrences of a substring with another substring.

- **list** - ordered, mutable sequences of objects, enclosed in square brackets, e.g., [1, 2, 3]. If you have ever had a pleasure to meet the other definition, the curiosity you might have may be satisfied by looking at the chapter where we will be talking about the data structures more deeply. In the Table 4.4 all the methods implemented in the list object are clearly shown. In Python, the simplicity of a language specification allow us not to remember about the precision of this specific data structure definition as it is for example implemented in the C++.

Example 4.3:

```
1 lst = [1, 2, 3]
2 print(lst) # Output: [1, 2, 3]
```

Lists in Python are versatile and can contain objects of different types. They support operations like indexing, slicing, concatenation, and more. Additionally, lists have various methods such as `append()`, `pop()`, `sort()`, and `reverse()` that allow you to modify the list in-place. In the Table 4.4 all the methods usage are clearly shown.

Table 4.4: List Methods

Method	Example	Explanation
`len()`	`len([1, 2, 3])`	Returns the number of elements in the list.
`list.append()`	`[1, 2].append(3)`	Appends an element to the end of the list.
`list.extend()`	`[1, 2].extend([3, 4])`	Extends the list by appending elements from another list.
`list.insert()`	`[1, 3].insert(1, 2)`	Inserts an element at the specified index.
`list.remove()`	`[1, 2, 3].remove(2)`	Removes the first occurrence of an element from the list.
`list.pop()`	`[1, 2, 3].pop()`	Removes and returns the last element of the list.
`list.index()`	`[1, 2, 3].index(2)`	Returns the index of the first occurrence of an element in the list.
`list.sort()`	`[3, 1, 2].sort()`	Sorts the elements of the list in ascending order.
`list.reverse()`	`[1, 2, 3].reverse()`	Reverses the order of the elements in the list.

- **tuple** - ordered, immutable sequences of objects, enclosed in parentheses, e.g., (4, 5, 6).

  Example 4.3:

```
1 tup = (4, 5, 6)
2 print(tup) # Output: (4, 5, 6)
```

  Tuples are similar to lists, but they are immutable, meaning their elements cannot be changed after creation. This makes tuples useful for representing collections of related values that should not be modified. In contrast, the tuple type in C# is mutable and allows elements to be modified.

  Tuples in Python support indexing and slicing, just like lists and strings. You can access individual elements of a tuple using square brackets and the element's index. Additionally, tuples have several methods that allow you to manipulate them, such as `len()`, `index()`, and `count()`.

  In Table 4.5, you can find examples and explanations of some common methods used for tuple manipulation.

4.4 Mapping Type

Mapping Type:

- **dict** - dictionaries, unordered collections of key-value pairs, enclosed in curly braces, e.g., 'name': 'John', 'age': 25.

Table 4.5: Tuple Methods

Method	Example	Explanation
`len()`	`len((4, 5, 6))`	Returns the number of elements in the tuple.
`tuple.index()`	`(4, 5, 6).index(5)`	Returns the index of the first occurrence of an element in the tuple.
`tuple.count()`	`(4, 4, 5, 6).count(4)`	Returns the number of occurrences of an element in the tuple.

Example 4.4:

```
person = {'name': 'John', 'age': 25}
print(person) # Output: {'name': 'John', 'age': 25}
```

Dictionaries are useful for storing and retrieving data by using unique keys as identifiers. Each key in a dictionary is associated with a value, and you can access values by their corresponding keys. Dictionary keys must be immutable types such as strings or numbers. Dictionaries support operations like adding new key-value pairs, accessing values by keys, modifying values, and more.

In addition to the basic sequence operations, such as indexing and slicing, each sequence type has specific methods associated with it. For example, string methods like `find()` and `replace()` allow you to search for substrings or replace parts of a string. Here are a few examples of string methods:

- `find(substring)` - Returns the index of the first occurrence of the substring in the string, or -1 if the substring is not found.

Example 4.4:

```
s = "Hello, World!"
index = s.find("World")
print(index) # Output: 7
```

- `replace(old, new)` - Returns a new string where all occurrences of the old substring are replaced with the new substring.

Example 4.4:

```
s = "Hello, World!"
new_s = s.replace("World", "Python")
print(new_s) # Output: Hello, Python!
```

- `split(separator)` - Splits the string into a list of substrings at each occurrence of the separator.

Example 4.4:

```
s = "Hello, World!"
words = s.split(", ")
print(words) # Output: ['Hello', 'World!']
```

- `upper()` - Returns a new string with all characters converted to uppercase.

Example 4.4:

```
s = "Hello, World!"
upper_s = s.upper()
print(upper_s) # Output: HELLO, WORLD!
```

Table 4.6: Dictionary Methods

Method	Example	Explanation
`len()`	`len({'name': 'John', 'age': 25})`	Returns the number of key-value pairs in the dictionary.
`dict.keys()`	`{'name': 'John', 'age': 25}.keys()`	Returns a list of all keys in the dictionary.
`dict.values()`	`{'name': 'John', 'age': 25}.values()`	Returns a list of all values in the dictionary.
`dict.get()`	`{'name': 'John', 'age': 25}.get('name')`	Returns the value associated with the specified key, or a default value if the key is not found.
`dict.pop()`	`{'name': 'John', 'age': 25}.pop('age')`	Removes and returns the value associated with the specified key.
`dict.update()`	`{'name': 'John'}.update({'age': 25})`	Updates the dictionary with the key-value pairs from another dictionary.

Similarly, lists have methods like `append()`, `pop()`, `sort()`, and `reverse()`. Dictionaries have methods for adding, accessing, and modifying key-value pairs, such as `keys()`, `values()`, and `update()`.

These methods provide additional functionality and flexibility when working with different sequence types in Python. For more details on available methods and their usage, you can refer to Python's standard library documentation or use the `dir()` and `help()` functions to explore the available attributes and get help on specific methods of sequence types.

Table 4.6, you can find the usage and explanations of some common methods used for dictionaries manipulation implemented in Python.

4.5 Set Types

Set Types:

- **set** - unordered collections of unique elements, enclosed in curly braces, e.g., 1, 2, 3.

  Example 4.5:

```
s = {1, 2, 3}
print(s) # Output: {1, 2, 3}
```

- **frozenset** - immutable sets, enclosed in parentheses, e.g., frozenset(4, 5, 6).

  Example 4.5:

```
fs = frozenset({4, 5, 6})
print(fs) # Output: frozenset({4, 5, 6})
```

Sets in Python are unordered collections of unique elements. They are represented by curly braces () and do not allow duplicate values. Sets are useful for operations like membership testing, removing duplicates from a sequence, and mathematical set operations such as union, intersection, and difference.

Here's a table 4.7 showcasing some common methods used for set manipulation:

Table 4.7: Set Methods

Method	Example	Explanation
`len()`	`len({1, 2, 3})`	Returns the number of elements in the set.
`set.add()`	`{1, 2}.add(3)`	Adds an element to the set.
`set.remove()`	`{1, 2, 3}.remove(2)`	Removes an element from the set.
`set.discard()`	`{1, 2, 3}.discard(4)`	Removes an element from the set if it exists, without raising an error if it doesn't.
`set.pop()`	`{1, 2, 3}.pop()`	Removes and returns an arbitrary element from the set.
`set.union()`	`{1, 2}.union({2, 3})`	Returns a new set with all unique elements from both sets.
`set.intersection()`	`{1, 2}.intersection({2, 3})`	Returns a new set with common elements of both sets.
`set.difference()`	`{1, 2, 3}.difference({2, 3, 4})`	Returns a new set with elements in the set that are not in the given set.
`set.clear()`	`{1, 2, 3}.clear()`	Removes all elements from the set.

4.6 Boolean Type

Boolean Type:

- **bool** - boolean values, either *True* or *False*.

Example 4.6:

```
a = True
b = False
print(a) # Output: True
print(b) # Output: False
```

The boolean type in Python represents logical values, which can be either *True* or *False*. Booleans are often used in decision-making and control flow statements to determine the execution path based on conditions.

In Python, boolean values can be assigned to variables, used in expressions, and compared using logical operators such as `and`, `or`, and `not`. The boolean type is also the result of comparison operations, such as equality (`==`), inequality (`!=`), greater than (`>`), less than (`<`), etc.

Here's an example that demonstrates the usage of boolean values in Python:

```
x = 10
y = 5
is_greater = x > y
print(is_greater)  # Output: True
```

In this example, the boolean variable `is_greater` is assigned the value `True` because the condition `x > y` is true.

Boolean values play a crucial role in control flow statements like `if`, `while`, and `for` loops, as they determine the execution of specific code blocks based on the evaluation of conditions.

Understanding boolean values and their usage is fundamental in programming, as it allows you to create conditional logic and make decisions based on different conditions. Now it is a hight time to tale a brief look how the boolean type may be explained using the mathematical logic basics and the elements of the set theory.

Mathematical logic, as the foundation of mathematics, is about symbolic abstraction. One way to analyze and understand a proposition is by utilizing mathematical symbols and the rules of logic.

Consider two propositions, A and B. We can combine these propositions using logical operators to form complex propositions. The logical operators often used are 'AND' ($\wedge$), 'OR' ($\vee$), and 'NOT' ($\neg$).

The 'AND' operator, denoted as $A \wedge B$, results in true only if both A and B are true. The 'OR' operator, denoted as $A \vee B$, results in true if either A or B (or both) are true. The 'NOT' operator is a unary operator that negates the truth value of its operand. If A is true, then $\neg A$ is false, and vice versa.

The truth values of these logical operations can be represented by truth tables. A truth table for a logical operation lists the possible truth values of the operation for each possible combination of truth values of its operands.

For example, for the logical 'AND' operator, we have the following truth table:

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

For the logical 'OR' operator, we have the following truth table:

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

Understanding these basic operations and truth tables is essential for the study of mathematical logic, and it serves as the basis for more advanced concepts in the field, such as implication, biconditional, contradiction, and equivalence.

4.7 Binary Types

Binary Types:

- **bytes** - sequences of integers in the range 0-255, enclosed in parentheses with a *b* prefix, e.g., b'Hello'.

  Example 4.7:

```
1 bt = b'Hello'
2 print(bt) # Output: b'Hello'
```

- **bytearray** - mutable sequences of integers in the range 0-255, enclosed in parentheses with a *bytearray* prefix, e.g., bytearray(b'World').

  Example 4.7:

```
1 ba = bytearray(b'World')
2 print(ba) # Output: bytearray(b'World')
```

Binary types in Python, such as `bytes` and `bytearray`, are used to represent sequences of integers in the range 0-255. These types are often used to handle binary data, such as file I/O, network communication, and cryptographic operations.

`bytes` objects are immutable, meaning they cannot be modified after creation. They are represented by a sequence of integers enclosed in parentheses with a *b* prefix, e.g., `b'Hello'`. You can access individual bytes within a `bytes` object using indexing.

`bytearray` objects, on the other hand, are mutable and allow in-place modifications. They are represented by a sequence of integers enclosed in parentheses with a *bytearray* prefix, e.g., `bytearray(b'World')`. You can modify individual bytes within a `bytearray` object using indexing.

Here's a table 4.8 showcasing some common operations and functions used with binary types:

Each data type in Python has its own characteristics and methods associated with it, allowing you to perform various operations and manipulations. Python also provides built-in functions to convert between different data types.

It's important to note that variables in Python are dynamically typed, meaning that you can assign values of different types to the same variable. Python automatically adjusts the type of the variable based on the assigned value. This flexibility allows for easy and convenient programming.

Table 4.8: Binary Operations and Functions (Continued)

Operation/Function	Example	Explanation
Indexing	`bt[0]`	Accesses the byte at the specified index in the `bytes` object.
Slicing	`ba[1:3]`	Returns a new `bytearray` object containing a slice of bytes from the original object.
Length	`len(bt)`	Returns the number of bytes in the `bytes` object.
Conversion	`bytes([65, 66, 67])`	Creates a `bytes` object from a list of integers representing ASCII values.
`hex()`	`ba.hex()`	Returns a hexadecimal representation of the `bytearray` object.
`decode()`	`bt.decode()`	Decodes the `bytes` object into a string using a specified encoding (e.g., UTF-8).

4.8 Problems and Solutions

1. Write a function to check if a given variable is of a certain type.

   Solution:

```
    def check_type(var, type):
        return isinstance(var, type)
```

2. Convert a float to an integer without using built-in functions.

   Solution:

```
    def float_to_int(f):
        return int(f // 1)
```

3. Swap two variables without using a temporary variable.

   Solution:

```
    def swap_vars(a, b):
        a, b = b, a
        return a, b
```

4. Write a function to compute the factorial of a number.

   Solution:

```
    def factorial(n):
        return 1 if (n==1 or n==0) else n * factorial(n -
            1)
```

5. Convert a decimal number to binary without using built-in functions.

   Solution:

```
    def decimal_to_binary(n):
        return bin(n).replace("0b", "")
```

6. Write a function to reverse a string.

   Solution:

```
    def reverse_string(s):
        return s[::-1]
```

7. Write a function to count the occurrences of a character in a string.

   Solution:

```
    def count_char(s, c):
        return s.count(c)
```

8. Implement a function to convert a list of integers into a single integer.

   Solution:

```
    def list_to_int(l):
        return int("".join(map(str, l)))
```

9. Write a function to check if a string is a palindrome.

   Solution:

```
    def is_palindrome(s):
        return s == s[::-1]
```

10. Implement a function to calculate the nth Fibonacci number.

    Solution:

```
    def fibonacci(n):
        if n<=0:
            return "Input should be positive integer"
        elif n==1:
            return 0
        elif n==2:
            return 1
        else:
            a, b = 0, 1
            for i in range(2, n):
                a, b = b, a + b
            return b
```

11. Write a function to check if a list is empty or not.

    Solution:

```
    def is_empty(lst):
        return not lst
```

12. Write a function to clone or copy a list.

    Solution:

```
    def copy_list(lst):
        return lst[:]
```

13. Write a function to get the frequency of a particular element in a list.

    Solution:

```
    def count_element(lst, elem):
        return lst.count(elem)
```

14. Write a function to find the largest number in a list.

    Solution:

```
    def max_of_list(lst):
        return max(lst)
```

15. Write a function to print all unique values in a list.

    Solution:

```
    def unique_elements(lst):
        return list(set(lst))
```

16. Write a function to check if a list is sorted or not.

    Solution:

```
    def is_sorted(lst):
        return lst == sorted(lst)
```

17. Write a function to remove duplicates from a list.

    Solution:

```
    def remove_duplicates(lst):
        return list(set(lst))
```

18. Write a function to find the second largest number in a list.

    Solution:

```
    def second_largest(lst):
        return sorted(lst)[-2]
```

19. Write a function to find the intersection of two lists.

    Solution:

```
    def intersect_lists(lst1, lst2):
        return list(set(lst1) & set(lst2))
```

20. Write a function to concatenate two lists in the following manner: if ‘list1 = [’a’, ’b’]‘ and ‘list2 = [’1’, ’2’]‘, the result should be ‘[’a1’, ’b2’]‘.

    Solution:

```
    def concatenate_lists(list1, list2):
        return [i + j for i, j in zip(list1, list2)]
```

21. Write a function to reverse a list.

    Solution:

```
    def reverse_list(lst):
        return lst[::-1]
```

22. Write a function that flattens a nested list.

    Solution:

```
    def flatten_list(nested_lst):
        return [elem for sublist in nested_lst for elem
           in sublist]
```

23. Write a function that takes two lists and returns True if they have at least one common member.

    Solution:

```
    def common_member(lst1, lst2):
        return any(elem in lst1 for elem in lst2)
```

24. Write a function to convert a tuple to a string.

    Solution:

```
    def tuple_to_string(tup):
        return ''.join(tup)
```

25. Write a function to print a tuple with string formatting.

    Solution:

```
    def print_tuple(tup):
        print("This is your tuple: %s" % (tup,))
```

26. Write a function to replace the last value of tuples in a list.

    Solution:

```
    def replace_last_value_in_tuples(lst, value):
        return [(tup[:-1] + (value,)) for tup in lst]
```

27. Write a function to sort a list of tuples using the second element.

    Solution:

```
    def sort_tuples(lst):
        return sorted(lst, key=lambda x: x[1])
```

28. Write a function to remove an empty tuple(s) from a list of tuples.

    Solution:

```
    def remove_empty_tuples(lst):
        return [tup for tup in lst if tup]
```

29. Write a function to find the index of an item of a tuple.

    Solution:

```
    def index_of_tuple(tup, item):
        return tup.index(item)
```

30. Write a function to find the length of a tuple.

    Solution:

```
    def length_of_tuple(tup):
        return len(tup)
```

31. Write a function to add a key-value pair to a dictionary.

    Solution:

```
    def add_key_value(dictionary, key, value):
        dictionary[key] = value
        return dictionary
```

32. Write a function to concatenate two dictionaries.

    Solution:

```
    def concat_dictionaries(dict1, dict2):
        dict1.update(dict2)
        return dict1
```

33. Write a function to check if a given key exists in a dictionary.

    Solution:

```
    def check_key(dictionary, key):
        return key in dictionary
```

34. Write a function to generate and print a dictionary that contains numbers (between 1 and n) in the form (x, x*x).

    Solution:

```
    def square_dictionary(n):
        return {i: i*i for i in range(1, n+1)}
```

35. Write a function to sum all the items in a dictionary.

Solution:

```
    def sum_dictionary(dictionary):
        return sum(dictionary.values())
```

36. Write a function to multiply all the items in a dictionary.

Solution:

```
    def multiply_dictionary(dictionary):
        result = 1
        for value in dictionary.values():
            result *= value
        return result
```

37. Write a function to remove a key from a dictionary.

Solution:

```
    def remove_key(dictionary, key):
        if key in dictionary:
            del dictionary[key]
        return dictionary
```

38. Write a function to map two lists into a dictionary.

Solution:

```
    def map_lists_to_dict(keys, values):
        return dict(zip(keys, values))
```

39. Write a function to sort a dictionary by key.

Solution:

```
    def sort_dict_by_key(dictionary):
        return {k: dictionary[k] for k in sorted(
           dictionary)}
```

40. Write a function to get the maximum and minimum value in a dictionary.

Solution:

```
    def max_and_min(dictionary):
        max_value = max(dictionary.values())
        min_value = min(dictionary.values())
        return max_value, min_value
```

Chapter 5

Operators

Python includes a variety of operators for performing operations on values. These include arithmetic operators, comparison operators, assignment operators, logical operators, and more.

5.1 Arithmetic Operators

Python includes arithmetic operators for performing mathematical operations on values. Here are some commonly used arithmetic operators:

- $+$ **(Addition)**: Adds two values.
- $-$ **(Subtraction)**: Subtracts one value from another.
- $*$ **(Multiplication)**: Multiplies two values.
- / **(Division)**: Divides one value by another.
- // **(Floor Division)**: Performs division and rounds the result down to the nearest whole number.
- % **(Modulus)**: Returns the remainder after division.
- $**$ **(Exponentiation)**: Raises a value to the power of another value.

Here are some examples of using arithmetic operators:

Example 5.1:

```
x = 5 + 3 # Addition: x = 8
y = 10 - 4 # Subtraction: y = 6
z = 3 * 2 # Multiplication: z = 6
w = 12 / 4 # Division: w = 3.0
v = 13 // 5 # Floor Division: v = 2
r = 15 % 4 # Modulus: r = 3
s = 2 \** 4 # Exponentiation: s = 16
```

In these examples, the arithmetic operators perform the specified operations on the values and assign the results to the respective variables.

5.2 Mathematical Functions

In addition to the basic arithmetic operators, Python also provides built-in mathematical functions that allow you to perform more complex mathematical calculations. These functions are part of the math module, which needs to be imported before use.

Here are some commonly used mathematical functions in Python:

- `abs(x)`: Returns the absolute value of `x`.
- `round(x, n)`: Rounds `x` to `n` decimal places. If `n` is not specified, it rounds to the nearest integer.
- `max(iterable)`: Returns the largest value in the `iterable`.
- `min(iterable)`: Returns the smallest value in the `iterable`.
- `pow(x, y)`: Returns `x` raised to the power of `y`.
- `sqrt(x)`: Returns the square root of `x`.
- `sin(x)`: Returns the sine of `x` (in radians).
- `cos(x)`: Returns the cosine of `x` (in radians).
- `tan(x)`: Returns the tangent of `x` (in radians).
- `log(x)`: Returns the natural logarithm (base e) of `x`.
- `log10(x)`: Returns the base-10 logarithm of `x`.

Here are some examples of using mathematical functions:

Example 5.2:

```
import math

abs_value = abs(-5) # abs_value = 5
rounded = round(3.14159, 2) # rounded = 3.14
largest = max([4, 9, 2, 7]) # largest = 9
smallest = min([4, 9, 2, 7]) # smallest = 2
power = pow(2, 3) # power = 8
sqrt_value = math.sqrt(16) # sqrt_value = 4.0
sine = math.sin(math.pi/2) # sine = 1.0
cosine = math.cos(math.pi) # cosine = -1.0
tangent = math.tan(math.pi/4) # tangent = 1.0
natural_log = math.log(10) # natural_log = 2.302585092994046
log_10 = math.log10(100) # log_10 = 2.0
```

In these examples, the mathematical functions are used to perform various calculations, such as finding the absolute value, rounding to a specific decimal place, finding the maximum and minimum values, raising a number to a power, calculating trigonometric functions, and logarithmic functions.

Note that the math module is imported using the `import` statement to access these mathematical functions.

5.3 Comparison Operators

Comparison operators are used to compare two values and return a boolean result (True or False) based on the comparison. Here are some commonly used comparison operators:

- $>$ **(Greater than)**: Returns True if the left value is greater than the right value.
- $<$ **(Less than)**: Returns True if the left value is less than the right value.
- $>=$ **(Greater than or equal to)**: Returns True if the left value is greater than or equal to the right value.
- $<=$ **(Less than or equal to)**: Returns True if the left value is less than or equal to the right value.
- $==$ **(Equal to)**: Returns True if the left value is equal to the right value.
- $!=$ **(Not equal to)**: Returns True if the left value is not equal to the right value.

Here are some examples of using comparison operators:

Example 5.3:

```
x = 5 > 3 # Greater than: x = True
y = 10 < 4 # Less than: y = False
z = 3 >= 2 # Greater than or equal to: z = True
w = 12 <= 4 # Less than or equal to: w = False
v = 5 == 5 # Equal to: v = True
r = 15 != 4 # Not equal to: r = True
```

In these examples, the comparison operators compare the values and return the boolean results based on the comparison.

5.4 Assignment Operators

Assignment operators are used to assign values to variables. The basic assignment operator in Python is $=$. It assigns the value on the right side to the variable on the left side.

Here are some examples of using assignment operators:

Example 5.4:

```
x = 5 # Assigns the value 5 to the variable x
y = 2 # Assigns the value 2 to the variable y
```

Python also provides compound assignment operators that combine an arithmetic operation with assignment. These include $+=$, $-=$, $*=$, $/=$, and

5.5 Set Operations

Python provides several set operations that allow you to manipulate and perform computations on sets. Here are some commonly used set operations:

- **Union** ($\cup$): Returns a new set containing all the unique elements from both sets.
- **Intersection** ($\cap$): Returns a new set containing the common elements between both sets.
- **Difference** ($-$): Returns a new set containing the elements that are in the first set but not in the second set.
- **Symmetric Difference** ($\oplus$): Returns a new set containing the elements that are in either of the sets, but not in both.
- **Subset** ($\subset$): Checks if one set is a subset of another set.
- **Superset** ($\supset$): Checks if one set is a superset of another set.

Here's an example of using set operations in Python:

Example 5.5:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

union_set = set1 | set2 # Union: union_set = {1, 2, 3, 4, 5, 6}
intersection_set = set1 & set2 # Intersection: intersection_set = {3, 4}
difference_set = set1 - set2 # Difference: difference_set = {1, 2}
symmetric_difference_set = set1 ^ set2 # Symmetric Difference: symmetric_difference_set = {1, 2, 5, 6}
is_subset = set1.issubset(set2) # Subset: is_subset = False
is_superset = set1.issuperset(set2) # Superset: is_superset = False
```

5.6 Logical Operators

Logical operators are used to combine and manipulate boolean values (`True` and `False`). Python includes three logical operators: `and`, `or`, and `not`.

Here's an example that demonstrates the usage of logical operators:

Example 5.6:

```
x = 5
y = 10

print(x > 0 and y < 100) # Output: True
```

```
print(x > 0 or y < 100) # Output: True
print(not x > 0) # Output: False
```

In this example, the `and` operator returns `True` if both the expressions on its left and right side are `True`. The `or` operator returns `True` if at least one of the expressions on its left or right side is `True`. The `not` operator negates the boolean value.

5.7 Bitwise Operators

Bitwise operators in Python are used to perform bitwise operations on individual bits of integer values. These operators treat the operands as sequences of bits and perform operations on corresponding bits.

Here are the bitwise operators available in Python:

- **AND** (): Performs a bitwise AND operation on the corresponding bits of two operands.
- **OR** (|): Performs a bitwise OR operation on the corresponding bits of two operands.
- **XOR** (): Performs a bitwise XOR (exclusive OR) operation on the corresponding bits of two operands.
- **NOT** (∼): Performs a bitwise NOT operation, which flips the bits of the operand.
- **Left Shift** (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
- **Right Shift** (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

Here's an example of using bitwise operators in Python:

Example 5.7:

```
a = 10 # Binary: 1010
b = 6 # Binary: 0110

bitwise_and = a & b # Bitwise AND: 2 (Binary: 0010)
bitwise_or = a | b # Bitwise OR: 14 (Binary: 1110)
bitwise_xor = a ^ b # Bitwise XOR: 12 (Binary: 1100)
bitwise_not_a = ~a # Bitwise NOT of a: -11 (Binary:
   11111111111111111111111111110101)
left_shift = a << 2 # Left shift a by 2 positions: 40 (Binary
   : 101000)
right_shift = a >> 1 # Right shift a by 1 position: 5 (Binary
   : 101)
```

In this example, the bitwise operators are applied to the variables `a` and `b`, resulting in new values that represent the bitwise operations performed on the individual bits.

Bitwise operators are useful in scenarios where you need to manipulate or extract specific bits from integers, perform binary flag operations, or optimize certain algorithms by taking advantage of bitwise operations at the binary level.

5.8 Number Base Conversion Functions

In Python, you can convert numbers between different bases, such as binary, octal, decimal, and hexadecimal. The built-in functions bin(), oct(), hex(), and int() allow you to perform number base conversions. Here's how you can use these functions:

- **Binary to Decimal**: Use the int() function with the base parameter set to 2 to convert a binary number to decimal.

  Example 5.8:

```
binary_num = '10101'
decimal_num = int(binary_num, 2)
print(decimal_num) # Output: 21
```

- **Decimal to Binary**: Use the bin() function to convert a decimal number to binary.

  Example 5.8:

```
decimal_num = 21
binary_num = bin(decimal_num)
print(binary_num) # Output: '0b10101'
```

- **Decimal to Octal**: Use the oct() function to convert a decimal number to octal.

  Example 5.8:

```
decimal_num = 21
octal_num = oct(decimal_num)
print(octal_num) # Output: '0o25'
```

- **Decimal to Hexadecimal**: Use the hex() function to convert a decimal number to hexadecimal.

  Example 5.8:

```
decimal_num = 21
hexadecimal_num = hex(decimal_num)
print(hexadecimal_num) # Output: '0x15'
```

- **Hexadecimal to Decimal**: Use the int() function with the base parameter set to 16 to convert a hexadecimal number to decimal.

  Example 5.8:

```
hexadecimal_num = '15'
decimal_num = int(hexadecimal_num, 16)
print(decimal_num) # Output: 21
```

These functions provide convenient ways to convert numbers between different bases in Python. By specifying the appropriate base parameter, you can convert numbers to and from binary, octal, decimal, and hexadecimal representations. In additional references of the book the systems conversion is covered with a bit more understandable example with no special functions usage.

5.9 Problems and Solutions

1. Write a Python program to calculate the area of a rectangle. The length and width of the rectangle are provided as inputs from the user.

```
length = float(input("Enter the length of the rectangle: "))
width = float(input("Enter the width of the rectangle: "))
area = length * width
print("The area of the rectangle is:", area)
```

   To calculate the area of a rectangle, we need to multiply its length by its width. The program prompts the user to enter the length and width as floating-point numbers, performs the calculation, and displays the result as the area of the rectangle.

2. Write a Python program to convert Celsius to Fahrenheit. The temperature in Celsius is provided as an input from the user.

```
celsius = float(input("Enter the temperature in Celsius: "))
fahrenheit = (celsius * 9/5) + 32
print("The temperature in Fahrenheit is:", fahrenheit)
```

   To convert Celsius to Fahrenheit, we can use the formula `F = (C * 9/5) + 32`, where `F` represents Fahrenheit and `C` represents Celsius. The program prompts the user to enter the temperature in Celsius, performs the conversion using the formula, and displays the result as the temperature in Fahrenheit.

3. Write a Python program to check if a number is even or odd. The number is provided as an input from the user.

```
    number = int(input("Enter a number: "))
    if number % 2 == 0:
        print("The number is even.")
    else:
        print("The number is odd.")
```

   To check if a number is even or odd, we use the modulus operator (%) to find the remainder when the number is divided by 2. If the remainder is 0, the number is even; otherwise, it is odd.

4. Write a Python program to calculate the square of a number. The number is provided as an input from the user.

```
    number = float(input("Enter a number: "))
    square = number ** 2
    print("The square of the number is:", square)
```

   To calculate the square of a number, we use the exponentiation operator (**). We raise the number to the power of 2 to obtain its square.

5. Write a Python program to find the maximum of three numbers. The numbers are provided as inputs from the user.

```
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    num3 = float(input("Enter the third number: "))
    maximum = max(num1, num2, num3)
    print("The maximum number is:", maximum)
```

To find the maximum of three numbers, we can use the built-in `max()` function. We provide the three numbers as arguments to the `max()` function, and it returns the maximum value among them.

6. Write a Python program to check if a year is a leap year. The year is provided as an input from the user.

```
    year = int(input("Enter a year: "))
    if year % 4 == 0 and (year % 100 != 0 or year % 400
       == 0):
        print("The year is a leap year.")
    else:
        print("The year is not a leap year.")
```

To check if a year is a leap year, we use the following conditions: - The year must be divisible by 4. - If the year is divisible by 100, it must also be divisible by 400.

If both conditions are satisfied, the year is a leap year; otherwise, it is not.

7. Write a Python program to calculate the sum of all even numbers from 1 to 100.

```
    sum_even = 0
    for number in range(2, 101, 2):
        sum_even += number
    print("The sum of all even numbers from 1 to 100 is:"
       , sum_even)
```

To calculate the sum of all even numbers from 1 to 100, we use a `for` loop to iterate over the range of numbers from 2 to 100 with a step of 2. We add each even number to the variable `sum_even` to accumulate the sum.

8. Write a Python program to reverse a string. The string is provided as an input from the user.

```
    string = input("Enter a string: ")
    reversed_string = string[::-1]
    print("The reversed string is:", reversed_string)
```

To reverse a string, we can use string slicing with a step of -1 (`[::-1]`). This will create a new string with the characters in reverse order.

9. Write a Python program to check if a string is a palindrome. The string is provided as an input from the user.

```
string = input("Enter a string: ")
reversed_string = string[::-1]
if string == reversed_string:
    print("The string is a palindrome.")
else:
    print("The string is not a palindrome.")
```

To check if a string is a palindrome, we compare the original string with its reverse. If they are equal, the string is a palindrome; otherwise, it is not.

10. Write a Python program to calculate the factorial of a number. The number is provided as an input from the user.

```
number = int(input("Enter a number: "))
factorial = 1
for i in range(1, number + 1):
    factorial *= i
print("The factorial of the number is:", factorial)
```

To calculate the factorial of a number, we use a `for` loop to iterate from 1 to the given number. In each iteration, we multiply the current factorial by the loop variable `i`.

11. Write a Python program to calculate the area of a triangle. The base and height of the triangle are provided as inputs from the user.

```
base = float(input("Enter the base of the triangle: "))
height = float(input("Enter the height of the triangle: "))
area = 0.5 * base * height
print("The area of the triangle is:", area)
```

To calculate the area of a triangle, we use the formula `A = 0.5 * base * height`, where `A` represents the area, `base` is the length of the base, and `height` is the height of the triangle.

12. Write a Python program to check if a string is an anagram of another string. The two strings are provided as inputs from the user.

```
string1 = input("Enter the first string: ")
string2 = input("Enter the second string: ")
if sorted(string1.lower()) == sorted(string2.lower()):
    print("The strings are anagrams.")
else:
    print("The strings are not anagrams.")
```

To check if two strings are anagrams, we convert both strings to lowercase, sort the characters, and compare the sorted strings. If they are equal, the strings are anagrams; otherwise, they are not.

13. Write a Python program to calculate the Fibonacci sequence up to a specified number of terms. The number of terms is provided as an input from the user.

```
num_terms = int(input("Enter the number of terms: "))
fibonacci = [0, 1]
for i in range(2, num_terms):
    next_term = fibonacci[i-1] + fibonacci[i-2]
    fibonacci.append(next_term)
print("The Fibonacci sequence is:", fibonacci)
```

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. To calculate the Fibonacci sequence, we use a `for` loop to iterate from the third term up to the specified number of terms. We calculate each term by adding the two previous terms and append it to the Fibonacci sequence list.

14. Write a Python program to find the prime factors of a number. The number is provided as an input from the user.

```
number = int(input("Enter a number: "))
factors = []
divisor = 2
while number > 1:
    if number % divisor == 0:
        factors.append(divisor)
        number /= divisor
    else:
        divisor += 1
print("The prime factors are:", factors)
```

Prime factors are the prime numbers that divide a given number without leaving a remainder. We start with the smallest prime factor (2) and continuously divide the number by the factor until it becomes 1. If the number is divisible by the factor, it is a prime factor, and we add it to the factors list.

15. Write a Python program to sort a list of numbers in ascending order using the bubble sort algorithm. The list of numbers is provided as an input from the user.

```
numbers = input("Enter a list of numbers separated by
    spaces: ").split()
numbers = [int(num) for num in numbers]

# Bubble sort algorithm
for i in range(len(numbers)):
    for j in range(len(numbers) - i - 1):
        if numbers[j] > numbers[j+1]:
            numbers[j], numbers[j+1] = numbers[j+1],
               numbers[j]

print("The sorted list is:", numbers)
```

The bubble sort algorithm repeatedly swaps adjacent elements if they are in the wrong order. We iterate through the list multiple times, comparing each pair of adjacent numbers and swapping them if necessary. This process is repeated until the list is sorted in ascending order.

16. Write a Python program to calculate the sum of digits in a positive integer. The integer is provided as an input from the user.

```
number = int(input("Enter a positive integer: "))
sum_of_digits = 0
while number > 0:
    digit = number % 10
    sum_of_digits += digit
    number //= 10
print("The sum of digits is:", sum_of_digits)
```

To calculate the sum of digits in an integer, we continuously extract the rightmost digit (remainder of division by 10) and add it to the sum. We then divide the number by 10 to remove the rightmost digit. This process is repeated until the number becomes 0.

17. Write a Python program to count the occurrences of each word in a given string. The string is provided as an input from the user.

```
string = input("Enter a string: ")
words = string.lower().split()
word_count = {}
for word in words:
    if word in word_count:
        word_count[word] += 1
    else:
        word_count[word] = 1
print("Word counts:", word_count)
```

We convert the string to lowercase and split it into individual words. We then use a dictionary (`word_count`) to keep track of the count of each word. For each word in the list, if it already exists in the dictionary, we increment its count; otherwise, we add it to the dictionary with an initial count of 1.

18. Write a Python program to generate a random password of a specified length. The length of the password is provided as an input from the user.

```
import random
import string

length = int(input("Enter the length of the password: "))
characters = string.ascii_letters + string.digits + string.punctuation
password = ''.join(random.choice(characters) for _ in range(length))
```

```
7     print("The generated password is:", password)
```

We import the `random` and `string` modules. The `characters` variable contains all possible characters for the password (letters, digits, and punctuation). We use a `for` loop and `random.choice()` function to randomly select characters from the `characters` string and join them together to form the password of the specified length.

19. Write a Python program to remove duplicate elements from a list. The list is provided as an input from the user.

```
1  numbers = input("Enter a list of numbers separated by
       spaces: ").split()
2  numbers = [int(num) for num in numbers]
3  unique_numbers = list(set(numbers))
4  print("List with duplicates removed:", unique_numbers
       )
```

We first split the user input into a list of numbers and convert them to integers. Then, we convert the list to a set, which automatically removes duplicate elements due to the nature of sets. Finally, we convert the set back to a list to preserve the original order of the elements.

20. Write a Python program to find the intersection of two lists. The two lists are provided as inputs from the user.

```
1  list1 = input("Enter the first list of numbers
       separated by spaces: ").split()
2  list2 = input("Enter the second list of numbers
       separated by spaces: ").split()
3  list1 = [int(num) for num in list1]
4  list2 = [int(num) for num in list2]
5  intersection = list(set(list1) & set(list2))
6  print("The intersection of the two lists is:",
       intersection)
```

We split the user inputs into two lists of numbers and convert them to integers. We then use the `set()` function to convert each list to a set. The ampersand operator (&) is used to find the intersection of the two sets, which contains the common elements. Finally, we convert the intersection set back to a list.

21. Write a Python program to perform a bitwise AND operation on two integers. The integers are provided as inputs from the user.

```
1 num1 = int(input("Enter the first integer: "))
2 num2 = int(input("Enter the second integer: "))
3 result = num1 & num2
4 print("The result of the bitwise AND operation is:",
     result)
```

The bitwise AND operator () performs a bitwise logical AND operation between the corresponding bits of two integers. The program prompts the user to enter two integers, performs the bitwise AND operation, and displays the result.

22. Write a Python program to perform a bitwise OR operation on two integers. The integers are provided as inputs from the user.

```
num1 = int(input("Enter the first integer: "))
num2 = int(input("Enter the second integer: "))
result = num1 | num2
print("The result of the bitwise OR operation is:",
   result)
```

The bitwise OR operator (|) performs a bitwise logical OR operation between the corresponding bits of two integers. The program prompts the user to enter two integers, performs the bitwise OR operation, and displays the result.

23. Write a Python program to perform a bitwise XOR operation on two integers. The integers are provided as inputs from the user.

```
num1 = int(input("Enter the first integer: "))
num2 = int(input("Enter the second integer: "))
result = num1 ^ num2
print("The result of the bitwise XOR operation is:",
   result)
```

The bitwise XOR operator (^) performs a bitwise logical XOR (exclusive OR) operation between the corresponding bits of two integers. The program prompts the user to enter two integers, performs the bitwise XOR operation, and displays the result.

24. Write a Python program to perform a bitwise NOT operation on an integer. The integer is provided as an input from the user.

```
num = int(input("Enter an integer: "))
result = ~num
print("The result of the bitwise NOT operation is:",
   result)
```

The bitwise NOT operator ( ) performs a bitwise logical NOT operation on each bit of an integer, inverting its value. The program prompts the user to enter an integer, performs the bitwise NOT operation, and displays the result.

25. Write a Python program to perform a left shift operation on an integer. The integer and the number of positions to shift are provided as inputs from the user.

```
num = int(input("Enter an integer: "))
shift = int(input("Enter the number of positions to shift
   : "))
result = num << shift
print("The result of the left shift operation is:",
   result)
```

The left shift operator (<<) shifts the bits of an integer to the left by the specified number of positions. The program prompts the user to enter an integer and the number of positions to shift, performs the left shift operation, and displays the result.

26. Write a Python program to perform a right shift operation on an integer. The integer and the number of positions to shift are provided as inputs from the user.

```
num = int(input("Enter an integer: "))
shift = int(input("Enter the number of positions to shift
   : "))
result = num >> shift
print("The result of the right shift operation is:",
   result)
```

The right shift operator (>>) shifts the bits of an integer to the right by the specified number of positions. The program prompts the user to enter an integer and the number of positions to shift, performs the right shift operation, and displays the result.

27. Write a Python program to check if a specific bit is set in an integer. The integer and the position of the bit are provided as inputs from the user.

```
num = int(input("Enter an integer: "))
position = int(input("Enter the position of the bit: "))
mask = 1 << position
result = (num & mask) != 0
print("The bit is set:", result)
```

The program prompts the user to enter an integer and the position of the bit to check. It creates a mask with the bit set at the specified position. The bitwise AND operation is performed between the integer and the mask to extract the value of the bit at that position. The result is then checked if it is not equal to 0, indicating that the bit is set.

28. Write a Python program to set a specific bit in an integer. The integer, the position of the bit, and the value (0 or 1) are provided as inputs from the user.

```
num = int(input("Enter an integer: "))
position = int(input("Enter the position of the bit: "))
value = int(input("Enter the value (0 or 1): "))
mask = 1 << position
if value == 1:
result = num | mask
else:
result = num & (~mask)
print("The result after setting the bit is:", result)
```

The program prompts the user to enter an integer, the position of the bit to set, and the value (0 or 1) to set. It creates a mask with the bit set at the specified position. If the value is 1, the bitwise OR operation is performed between the integer and

the mask to set the bit. If the value is 0, the bitwise AND operation is performed between the integer and the complement of the mask to clear the bit. The result is then displayed.

29. Write a Python program to check if any bit is set in an integer. The integer is provided as an input from the user.

```
num = int(input("Enter an integer: "))
result = num != 0
print("Any bit is set:", result)
```

The program prompts the user to enter an integer. It checks if the integer is not equal to 0, indicating that at least one bit is set. The result is then displayed.

30. Write a Python program to perform logical AND, OR, and NOT operations on two Boolean values. The Boolean values are provided as inputs from the user.

```
value1 = bool(input("Enter the first Boolean value: "))
value2 = bool(input("Enter the second Boolean value: "))
logical_and = value1 and value2
logical_or = value1 or value2
logical_not1 = not value1
logical_not2 = not value2
print("Logical AND:", logical_and)
print("Logical OR:", logical_or)
print("Logical NOT (value1):", logical_not1)
print("Logical NOT (value2):", logical_not2)
```

The program prompts the user to enter two Boolean values. It performs the logical AND (`and`), logical OR (`or`), and logical NOT (`not`) operations on the values and displays the results.

Chapter 6

Control Flow

Control flow refers to the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

6.1 If Statements

The ‘if‘ statement is used for conditional execution in Python. It executes a block of code if a specified condition is true:

Example 6.1:

```
1 if 5 > 2:
2     print("Five is greater than two!")
```

In this example, the condition is ‘5 ¿ 2‘ which is true, so the print statement is executed.

The else keyword

The ‘else‘ keyword in Python is used to define a block of code to be executed if the condition in the ‘if‘ statement is false:

Below the function which will tell if the 5 is less than 2 or else if the 5 is greater or equal than 5

Example 6.1:

```
1 if 5 < 2:
2     print("Five is less than two!")
3 else:
4     print("Five is not less than two!")
```

In this example, because the condition ‘5 ¡ 2‘ is false, the print statement under the ‘else‘ keyword is executed.

The elif keyword

The ‘elif‘ keyword in Python is short for ”else if”. It allows you to specify a new condition to be checked if the first condition is false:

Example 6.1: Lets make the consideration if the x is less than defined numbers:

```
x = 20
if x < 10:
    print("x is less than 10")
elif x < 30:
    print("x is less than 30")
else:
    print("x is 30 or more")
```

In this example, because ‘x‘ is ‘20‘, the condition ‘x ¡ 10‘ is false, so Python moves on to the ‘elif‘ statement. The condition ‘x ¡ 30‘ is true, so the corresponding print statement is executed.

6.2 While Loops

A ‘while‘ loop in Python executes a block of code as long as a specified condition is true:

Example 6.2:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

In this example, as long as ‘i‘ is less than ‘6‘, the loop will continue to execute and print the value of ‘i‘. The ‘i += 1‘ statement increments ‘i‘ by ‘1‘ after each loop iteration.

Here are a few more complex examples and constructions using while loops:

Example 6.2:

```
% Sum of numbers from 1 to 10
total = 0
i = 1
while i <= 10:
    total += i
    i += 1
print("Sum:", total)

% Countdown from 10 to 1
num = 10
while num >= 1:
    print(num)
    num -= 1
```

```
14 print("Blastoff!")
15
16 % Finding the first Fibonacci number greater than 1000
17 a, b = 0, 1
18 while b <= 1000:
19     a, b = b, a + b
20 print("First Fibonacci number greater than 1000:", b)
```

In Example 1, the loop calculates the sum of numbers from 1 to 10 using the `total` variable. Example 2 demonstrates a countdown from 10 to 1, and the loop prints the current value of `num` in each iteration. Finally, Example 3 finds the first Fibonacci number greater than 1000 using the variables `a` and `b` to keep track of the current and previous Fibonacci numbers.

Feel free to customize and experiment with these examples to further explore the capabilities of while loops in Python.

6.3 For Loops

A 'for' loop in Python is used to iterate over a sequence (like a list, tuple, set, or string) or other iterable objects. Iterating over a sequence is called traversal.

Iterating over lists

Here's an example of a 'for' loop in Python that iterates over a list:

Example 6.3:

```
1 fruits = ["apple", "banana", "cherry"]
2 for fruit in fruits:
3     print(fruit)
```

In this example, 'fruit' is a variable that takes the value of the next item in 'fruits' each time through the loop. So, this 'for' loop will print each fruit in the 'fruits' list.

Using the range() function

The range() function in Python is used to generate a sequence of numbers which might be used to iterating through using for loop. Here are some examples:

1. With only one argument:

   Example 6.3:

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

   This will output the numbers 0 through 4. Note that 'range(5)' generates numbers from 0 to 4, not up to 5. This is because 'range()' generates numbers up to, but not including, the end value you specify.

2. With two arguments:

Example 6.3:

```
    for i in range(2, 5):
        print(i)
```

This will output the numbers 2 through 4. Here, ‘range(2, 5)‘ generates numbers from 2 up to, but not including, 5.

3. With three arguments:

Example 6.3:

```
    for i in range(0, 10, 2):
        print(i)
```

This will output the numbers 0, 2, 4, 6, and 8. Here, ‘range(0, 10, 2)‘ generates numbers from 0 up to, but not including, 10, with a step of 2.

Nested for loops

You can use a ‘for‘ loop inside another ‘for‘ loop to iterate over multiple dimensions of data. This is known as a nested loop. For example:

Example 6.3:

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

In this example, the outer loop runs three times, and each time it runs, the inner loop also runs three times. This results in a total of nine iterations of the inner loop, once for each combination of ‘i‘ and ‘j‘ values.

The enumerate() function

When iterating over a sequence, you may sometimes want to know the index of the current item in the sequence. You can use the ‘enumerate()‘ function for this:

Example 6.3:

```
for i, fruit in enumerate(fruits):
    print(f"The fruit at index {i} is {fruit}.\n")
```

In this example, ‘enumerate(fruits)‘ generates a sequence of pairs, where the first element of each pair is the index and the second element is the corresponding item from ‘fruits‘. The ‘for‘ loop then unpacks each pair into ‘i‘ and ‘fruit‘. At the end of the f-stting constructed there is a new line symbol `\n`.

6.4 Break, Continue, and Pass

Control flow isn't just about determining the sequence of execution; it's also about having control over that sequence when necessary. Python provides three essential keywords that can influence loops in various ways: 'break', 'continue', and 'pass'.

Break

The **break** statement is used to exit the loop prematurely. Once a **break** statement is encountered, Python will terminate the loop and move on to the next line of code after the loop. This is particularly useful when you're searching for something or when you need to stop a loop based on a specific condition.

Example 6.4:

```
for num in range(10):
    if num == 5:
        break
    print(num)
```

Here, numbers from 0 to 4 are printed. When the loop encounters the number '5', the **break** statement is executed, and the loop terminates.

Continue

While **break** stops the loop, the **continue** statement stops the current iteration and continues with the next one. It can be used when you want to skip a specific iteration based on a condition.

Example 6.4:

```
for num in range(10):
    if num % 2 == 0:  # checks if the number is even
        continue
    print(num)
```

In this example, only odd numbers between 0 and 9 are printed. If an even number is detected, the 'continue' statement is executed, and the loop skips the 'print' statement for that iteration.

Pass

The **pass** statement is a no-operation placeholder. It's a command that does nothing and can be used when a statement is syntactically required but you don't want to execute any command or code. It's often used as a placeholder with the intention of filling it in later.

Example 6.4:

```
for num in range(10):
    if num % 2 == 0:
        pass  # we'll handle this later
    else:
        print(f"{num} is odd")
```

Here, every even number will encounter the **pass** statement, essentially doing nothing for that iteration. The odd numbers will have their value printed.

6.5 Conclusion

Mastering control flow in Python is crucial for creating efficient and effective algorithms. In the previous chapter some examples of the practical usage have already been mentioned. In the next we will also has a grate to chance to participate in the control flow father discussion. With a combination of conditional statements, loops, and control keywords, you can create complex logic for your programs. The examples and advanced constructions provided here should offer a deeper understanding, but the best way to become proficient is to practice, experiment, and apply these concepts in real-world scenarios.

6.6 Problems and solutions

1. **Basic If Statement**: Check if a given number is even or odd.

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

2. **Elif Construction**: Determine if a number is negative, positive, or zero.

```
num = int(input("Enter a number: "))
if num > 0:
    print("Positive")
elif num < 0:
    print("Negative")
else:
    print("Zero")
```

3. **Simple While Loop**: Print numbers from 1 to 5 using a 'while' loop.

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

4. **For Loop with Strings**: Print each character of a given string.

```
for char in "hello":
    print(char)
```

5. **Range Function**: Print all odd numbers between 1 and 10.
   Solution:

```
for num in range(1, 11, 2):
    print(num)
```

6. **Break in For Loop**: Search for the first vowel in a given string.

```
string = "pythonista"
for char in string:
    if char in 'aeiou':
        print(char)
        break
```

7. **Continue in For Loop**: Print all numbers in a range except those divisible by 3.

```
for num in range(10):
    if num % 3 == 0:
        continue
    print(num)
```

8. **Nested Loops**: Using two loops, print pairs of numbers (i, j) where i and j both range from 1 to 3.

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

9. **Enumerate Function**: Given a list of colors, print each color with its index.

```
colors = ["red", "green", "blue"]
for index, color in enumerate(colors):
    print(index, color)
```

10. **While Loop with Break**: Create a loop that continuously asks the user for input and breaks when the user enters "stop".

```
while True:
    text = input("Enter text (or 'stop' to exit): ")
    if text == "stop":
        break
```

11. **For Loop with Else**: Use a for loop to search for a number in a list. If the number is not found, print a message using the else clause of the for loop.

```
numbers = [1, 3, 5, 7, 9]
for num in numbers:
    if num == 4:
        break
else:
    print("Number 4 is not in the list.")
```

12. **Nested Conditional Statements**: Check if a given number is positive even, positive odd, negative even, negative odd, or zero.

```
num = int(input("Enter a number: "))
if num > 0:
    if num % 2 == 0:
```

```
        print("Positive even")
    else:
        print("Positive odd")
elif num < 0:
    if num % 2 == 0:
        print("Negative even")
    else:
        print("Negative odd")
else:
    print("Zero")
```

13. **Using pass in a Loop**: Iterate over a range of numbers. If the number is even, do nothing using the ‘pass‘ statement, otherwise, print the number.

```
for i in range(1, 6):
    if i % 2 == 0:
        pass
    else:
        print(i)
```

14. **Infinite Loop with Break**: Create a loop that will run indefinitely until a certain user input is given, then use ‘break‘ to exit the loop.

```
while True:
    response = input("Enter 'q' to quit: ")
    if response == 'q':
        break
```

15. **Skip Even Numbers Using Continue**: Iterate over numbers from 1 to 10, but skip printing even numbers using the ‘continue‘ statement.

```
for num in range(1, 11):
    if num % 2 == 0:
        continue
    print(num)
```

16. **Nested Loops to Generate Patterns**: Use nested loops to generate a pattern of stars, where each line contains 1 more star than the previous line, for 5 lines.

```
for i in range(1, 6):
    for j in range(i):
        print("*", end="")
    print()
```

Chapter 7

Functions

Functions in Python are defined using the ‘def‘ keyword. Functions allow for code reuse and can make programs more modular and easier to understand. Here’s how you can define a function in Python:

Example 7.0:

```
1 def my_function():
2     print("Hello from a function")
```

In this example, ‘my_function‘ is a function that prints ”Hello from a function” when called.

7.1 Calling a Function

To call a function in Python, you simply need to write the function’s name followed by parentheses. Here’s how you can call the function we defined earlier:

```
1 my_function()
```

7.2 Parameters and Arguments

Functions can also take parameters, which are values that you can pass into the function. The parameters are defined in the function definition, and the values you pass in are called arguments. Here’s an example of a function with parameters:

Lets imagine we have to write the function to greet the user after they input their name:

Example 7.2:

```
1 def greet(name):
2     print(f"Hello, {name}!")
3
4 greet("Alice")
```

In this example, ‘name‘ is a parameter. You can call this function with an argument like this: This will output: ”Hello, Alice!”.

7.3 Return Values

Functions in Python can also return values. This is done using the ‘return‘ statement. Here’s an example of a function that returns a value:

Example 7.3:

```
def square(number):
    return number ** 2
```

In this example, the ‘square‘ function takes a number as a parameter and returns the square of that number. You can call this function and store its return value in a variable like this:

```
squared = square(5)
print(squared)  # Output: 25
```

7.4 Default Parameter Value

In Python, you can also provide a default value for a function’s parameters. This value will be used if the function is called without an argument for that parameter. Here’s an example:

Lets create a function that takes the parameter of the name of the user but also has its default value set as the Stranger string:

Example 7.4:

```
def greet(name="Stranger"):
    print(f"Hello, {name}!")
```

In this example, if the ‘greet‘ function is called without an argument, it will use ”Stranger” as the default value for the ‘name‘ parameter:

```
greet()  # Output: Hello, Stranger!
```

7.5 Multiple Parameters and Arguments

Python functions can take multiple parameters. You just need to separate them with a comma in the function definition. Here’s an example:

A function to compute the sum of two numbers:

Example 7.5:

```
def add_numbers(num1, num2):
        return num1 + num2
```

You can call this function with two arguments like this:

```
sum = add_numbers(5, 7)
print(sum) # Output: 12
```

7.6 Variable-length Arguments

Sometimes you may want to define a function that can take any number of arguments. You can do this in Python using *args and **kwargs. Here's an example:

A function that accepts any number of arguments and returns their sum:

Example 7.6:

```
1 def add_numbers(*args):
2         return sum(args)
```

You can call this function with any number of arguments like this:

```
1 sum = add_numbers(1, 2, 3, 4, 5)
2 print(sum) # Output: 15
```

7.7 Function Annotations

Function annotations provide a way of associating various parts of a function with arbitrary python expressions at compile time. The annotations can be useful for type-checking and documentation:

```
1 def multiply(a: int, b: int) -> int:
2     return a * b
```

7.8 Docstrings

Documentation strings, or docstrings, provide a way to associate documentation with Python program entities:

```
1 def add(a, b):
2     """This function adds two numbers and returns the result.
         """
3     return a + b
```

You can access the docstring using the `__doc__` attribute of the function.

7.9 First-Class Functions

In Python, functions are first-class citizens, meaning they can be used as values:

```
1 def greet():
2     return "Hello"
3
4 # Assign function to a variable
5 say_hello = greet
6 print(say_hello())
```

7.10 Closures

Closures allow a nested function to capture and remember values from its containing scope:

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

closure = outer_function(10)
print(closure(5))
```

7.11 Decorators

Decorators are a powerful and expressive tool in Python, allowing programmers to modify the behavior of functions or methods without changing their actual code. They are a form of metaprogramming and are often used for logging, enforcing access control, instrumentation, and more.

In essence, a decorator is a higher-order function, meaning it takes one or more functions as arguments and returns a new function. The new function usually extends or alters the behavior of the original function in some way.

The '@' syntax before a function definition is a syntactic sugar and is just an easier way of writing '$say_hello = my_decorator(say_hello)$'.

Here's an example of a simple decorator:

```
def my_decorator(func):
    def wrapper():
        print("Something before the function.")
        func()
        print("Something after the function.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

When you call the function `say_hello()`, as we did in the last line of the previous example, you'll get the following output:

```
Something before the function.
Hello!
Something after the function.
```

Decorators can also take arguments, allowing for more customizable behavior. They can be stacked to apply multiple decorators to a single function, and can also be used to decorate class methods.

7.12 Recursive Functions

A recursive function is a function that calls itself during its execution. This enables the function to be repeated several times, as it can call itself during its execution. Here's an example of a recursive function:

A function to compute the factorial of a number using recursion:

Example 7.12:

```
def factorial(n):
        if n == 1:
                return 1
        else:
                return n * factorial(n-1)
```

You can call this function like this:

```
fact = factorial(5)
print(fact) # Output: 120
```

7.13 Lambda Functions

In Python, anonymous function means that a function is without a name. The lambda keyword is used to create anonymous functions. Here's an example:

A lambda function to compute the square of a number:

Example 7.13:

```
square = lambda x : x ** 2
```

You can call this function like this:

```
sq = square(5)
print(sq) # Output: 25
```

7.14 Problems on Functions

1. **Basic Function** Write a function named ‘welcome‘ that prints ”Welcome to Python!”.

```
def welcome():
    print("Welcome to Python!")
```

2. **Function with Parameter** Create a function ‘display‘ that takes a string as a parameter and prints it.

```
def display(message):
    print(message)
```

3. **Function Returning Value** Define a function ‘add‘ that takes two numbers as parameters and returns their sum.

```
def add(a, b):
    return a + b
```

4. **Default Parameter** Write a function **display_name** that takes a name as a parameter and prints it. If no name is provided, it should print ”Guest”.

```
def display_name(name="Guest"):
    print(name)
```

5. **Multiple Parameters** Design a function **multiply** that takes two numbers and returns their product.

```
def multiply(a, b):
    return a * b
```

6. **Variable-length Arguments** Create a function **find_max** that finds and returns the maximum number from any given numbers.

```
def find_max(*args):
    return max(args)
```

7. **Function Annotations** Design a function with annotations that takes two floats as parameters and returns their division.

```
def divide(a: float, b: float) -> float:
    return a / b
```

8. **Using Docstrings** Write a function to compute the area of a circle given its radius. Include a docstring to explain its purpose.

```
def area_circle(radius):
    """Compute the area of a circle given its radius."""
    return 3.14159 * radius * radius
```

9. **First-Class Functions** Assign the **add** function you wrote in Problem 3 to a variable **sum_func** and demonstrate its usage.

```
sum_func = add
result = sum_func(3, 4)  # result will be 7
```

10. **Closure** Write a closure that returns a function to compute the power of a given number.

```
def power(n):
    def compute(x):
        return x ** n
    return compute
```

11. **Simple Decorator** Create a decorator **bold_print** that prints the decorated function's output surrounded by asterisks.

```
def bold_print(func):
    def wrapper(*args, **kwargs):
        print("*****")
        func(*args, **kwargs)
        print("*****")
    return wrapper

@bold_print
def message():
    print("Hello World!")
```

12. **Recursive Function** Write a recursive function to compute the Fibonacci series up to the nth term.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

13. **Lambda Function** Use a lambda function to compute the cube of a number.

```
cube = lambda x: x ** 3
```

14. **Function Returning Function** Write a function ‘operation‘ that takes a character (either ’+’ or ’-’) and returns the appropriate function (either add or subtract).

```
def add(x, y):
    return x + y

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

```

```
def operation(op):
    if op == '+':
        return add
    else:
        return subtract
```

15. **Function with Both *args and **kwargs** Design a function **display_data** that can accept any number of positional and keyword arguments and prints them.

```
def display_data(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

16. **Nested Functions** Write a function that contains three nested functions. Each nested function should print a message.

    Solution:

```
def outer():
    print("This is the outer function.")

    def first():
        print("This is the first nested function.")

    def second():
        print("This is the second nested function.")

    def third():
        print("This is the third nested function.")

    first()
    second()
    third()
```

17. **Recursive Power Function** Write a recursive function to compute the power of a number raised to an integer.

```
def power(base, exp):
    if exp == 0:
        return 1
    else:
        return base * power(base, exp-1)
```

18. **Decorator with Arguments** Write a decorator ‘repeat(n)‘ that repeats the execution of the decorated function n times.

```
def repeat(n):
    def decorator(func):
```

```
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def message():
    print("Repeated Message!")
```

19. **Lambda with Multiple Inputs** Use a lambda function to add three numbers.

```
add_three = lambda x, y, z: x + y + z
```

20. **Function that Returns Lambda** Write a function that returns a lambda function to compute the nth power of a number.

```
def nth_power(n):
    return lambda x: x ** n
```

Chapter 8

Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that focuses on the concept of objects, which are instances of classes. It allows for the organization of code into reusable and modular components. Python is an object-oriented programming language that fully supports OOP concepts. In this section, we will explore the key principles and features of OOP in Python.

8.1 Classes and Objects

At the core of OOP is the concept of classes and objects. A class is a blueprint for creating objects, which are instances of that class. It defines the attributes (data) and methods (functions) that the objects will have. Here's an example of a simple class in Python:

Example 8.1:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius**2

# Create an instance of the Circle class
my_circle = Circle(5)

# Accessing attributes and calling methods
print(my_circle.radius)  # Output: 5
print(my_circle.calculate_area())  # Output: 78.5
```

In the above example, we define a `Circle` class with an attribute `radius` and a method `calculate_area()`. We then create an instance of the `Circle` class called `my_circle` and access its attributes and methods using the dot notation.

8.2 Inheritance

Inheritance is a powerful feature in OOP that allows a class to inherit attributes and methods from another class. The class that is being inherited from is called the base class or superclass, and the class that inherits from it is called the derived class or subclass. The derived class can add its own attributes and methods or override the ones inherited from the base class.

Example 8.2:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement
           this method")


class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

my_dog = Dog("Buddy")
my_cat = Cat("Whiskers")

print(my_dog.speak())  # Output: Woof!
print(my_cat.speak())  # Output: Meow!
```

In the above example, we define an `Animal` base class with an abstract method `speak()`. We then create two derived classes, `Dog` and `Cat`, that inherit from the `Animal` class and provide their own implementation of the `speak()` method.

8.3 Encapsulation and Data Hiding

Encapsulation is the practice of bundling data and methods together within a class, hiding the internal details of how the data is stored and processed. In Python, we can achieve encapsulation by using private and protected attributes and methods.

Private attributes and methods are indicated by prefixing them with two underscores (__). They can only be accessed within the class itself and not from outside the class or its subclasses. Protected attributes and methods are indicated by prefixing them with a single underscore (_). They can be accessed within the class, its subclasses, and even from outside the class, but it is considered best practice not to do so. Here's an example:

Example 8.3:

```
class Car:
    def __init__(self, brand, model, year):
        self.__brand = brand
        self._model = model
        self.year = year

    def start_engine(self):
        print("Engine started.")

    def __private_method(self):
        print("This is a private method.")

my_car = Car("Toyota", "Camry", 2022)
print(my_car.__brand)  # Raises an AttributeError
print(my_car._model)  # Accessible, but conventionally
   considered protected
print(my_car.year)  # Accessible
my_car.start_engine()  # Outputs: Engine started.
my_car.__private_method()  # Raises an AttributeError
```

In the above example, the `Car` class has a private attribute `__brand`, a protected attribute `_model`, and a public attribute `year`. It also has a public method `start_engine()` and a private method `__private_method()`.

When accessing the attributes and methods, we see that `__brand` raises an `AttributeError` because it is private and cannot be accessed from outside the class. `_model` is accessible, but it is conventionally considered protected and should be accessed with caution. `year` can be accessed freely as it is a public attribute. Similarly, the public method `start_engine()` can be called, but the private method `__private_method()` raises an `AttributeError`.

Encapsulation helps in creating robust and maintainable code by controlling the access to class attributes and methods, preventing unintended modifications, and encapsulating the internal implementation details.

8.4 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common super class. It's an important principle that fosters flexibility and extensibility.

```
def animal_speak(animal):
    return animal.speak()

a = Dog("Buddy")
b = Cat("Mittens")

print(animal_speak(a))  # Outputs: Woof!
print(animal_speak(b))  # Outputs: Meow!
```

8.5 Composition

Composition is a design principle where a class is composed of one or more objects from other classes, rather than inheriting from them.

```
class Engine:
    def start(self):
        return "Engine starting!"

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return self.engine.start()

car = Car()
print(car.start())  # Outputs: Engine starting!
```

These are just some of the key features of object-oriented programming in Python. There are many more concepts and techniques that can be explored to leverage the power of OOP in your Python programs.

8.6 General Case of Usage and Simple Example

In addition to the concepts discussed above, we can implement classes for geometric figures such as triangles, rectangles, and circles, with methods to calculate their areas and perimeters. Here's an example:

Example 8.6:

```
import math

class GeometricFigure:
    def area(self):
        raise NotImplementedError("Subclass must implement
            this method")

    def perimeter(self):
        raise NotImplementedError("Subclass must implement
            this method")

class Triangle(GeometricFigure):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

```

```
18      def perimeter(self):
19          return 3 * self.base
20
21  class Rectangle(GeometricFigure):
22      def __init__(self, length, width):
23          self.length = length
24          self.width = width
25
26      def area(self):
27          return self.length * self.width
28
29      def perimeter(self):
30          return 2 * (self.length + self.width)
31
32  class Circle(GeometricFigure):
33      def __init__(self, radius):
34          self.radius = radius
35
36      def area(self):
37          return math.pi * self.radius**2
38
39      def perimeter(self):
40          return 2 * math.pi * self.radius
41
42  # Create instances of geometric figures
43  triangle = Triangle(5, 3)
44  rectangle = Rectangle(4, 6)
45  circle = Circle(2)
46
47  # Calculate and display areas and perimeters
48  print("Triangle: Area =", triangle.area(), "Perimeter =",
        triangle.perimeter())
49  print("Rectangle: Area =", rectangle.area(), "Perimeter =",
        rectangle.perimeter())
50  print("Circle: Area =", circle.area(), "Circumference =",
        circle.perimeter())
```

In the above example, we explored the concepts of inheritance, polymorphism, and encapsulation in object-oriented programming.

Inheritance allows a class to inherit attributes and methods from a base class. In this case, the Triangle, Rectangle, and Circle classes inherit from the GeometricFigure class. By doing so, they gain access to the area() and perimeter() methods defined in the base class. This promotes code reuse and allows for a hierarchical organization of related classes.

Polymorphism allows objects of different classes to be treated as objects of a common base class. In our example, we can treat instances of Triangle, Rectangle, and Circle as objects of the GeometricFigure class. This means we can use them interchangeably when calling the area() and perimeter() methods. The appropriate implementation for each

class will be automatically invoked at runtime, based on the actual type of the object.

Encapsulation is the practice of bundling data and methods together within a class, hiding the internal details of how the data is stored and processed. In our example, the internal workings of each geometric figure class are encapsulated within their respective classes. The data (such as base, height, length, width, and radius) is stored as instance variables and can only be accessed or modified through the defined methods (area() and perimeter()). This helps in creating modular and maintainable code, as the internal implementation can be changed without affecting the code that uses the objects.

By employing inheritance, polymorphism, and encapsulation, we can design flexible and modular object-oriented programs that are easier to understand, maintain, and extend.

8.7 Problems on OOP

1. **Singleton Pattern**: Ensure that a class has only one instance, and provide a global point of access to it.

   Solution:

```
class Singleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__
               (cls)
        return cls._instance
```

2. **Factory Pattern**: Create a class that is responsible for creating and returning instances of classes based on input arguments.

   Solution:

```
class AnimalFactory:
    def create_animal(self, animal_type):
        if animal_type == "Dog":
            return Dog()
        elif animal_type == "Cat":
            return Cat()
        else:
            return None
```

3. **Mixin Usage**: Create a mixin for JSON serialization of a Python object.

   Solution:

```
import json

class JSONMixin:
    def to_json(self):
        return json.dumps(self.__dict__)
```

4. **Diamond Inheritance Problem**: Create a diamond inheritance structure and solve the method resolution order problem using ‘super()‘.

   Solution:

```
class A:
    def method(self):
        print("A method")

class B(A):
    def method(self):
        print("B method")
```

```
8
9  class C(A):
10     def method(self):
11         print("C method")
12
13 class D(B, C):
14     def method(self):
15         super().method()
16
17 d = D()
18 d.method()  # Outputs: B method
```

5. **Observer Pattern**: Create a system where an object maintains a list of its dependents and notifies them of any state changes.

   Solution:

```
1  class Observer:
2      def update(self):
3          pass
4
5  class Subject:
6      def __init__(self):
7          self._observers = []
8
9      def add_observer(self, observer):
10         self._observers.append(observer)
11
12     def remove_observer(self, observer):
13         self._observers.remove(observer)
14
15     def notify(self):
16         for observer in self._observers:
17             observer.update()
```

6. **Decorator Pattern**: Implement a decorator pattern to add additional responsibilities to an object dynamically.

   Solution:

```
1  class Coffee:
2      def cost(self):
3          return 5
4
5  class MilkDecorator:
6      def __init__(self, coffee):
7          self._coffee = coffee
8
9      def cost(self):
10         return self._coffee.cost() + 2
```

7. **Strategy Pattern**: Implement a strategy pattern where the algorithm can be selected at runtime.

   Solution:

```
class Strategy:
    def execute(self):
        pass

class ConcreteStrategyA(Strategy):
    def execute(self):
        print("Strategy A")

class ConcreteStrategyB(Strategy):
    def execute(self):
        print("Strategy B")

class Context:
    def __init__(self, strategy):
        self._strategy = strategy

    def execute(self):
        self._strategy.execute()
```

8. **State Pattern**: Implement a pattern to allow an object to change its behavior when its internal state changes.

   Solution:

```
class State:
    def handle(self):
        pass

class StateA(State):
    def handle(self):
        print("State A handled")

class StateB(State):
    def handle(self):
        print("State B handled")

class Context:
    def __init__(self, state):
        self._state = state

    def request(self):
        self._state.handle()
```

9. **Prototype Pattern**: Implement the prototype pattern to clone objects.

   Solution:

```
import copy

class Prototype:
    def clone(self):
        return copy.deepcopy(self)
```

10. **Chain of Responsibility Pattern**: Implement a pattern to pass a request along a chain of potential handlers until one of them handles the request.

    Solution:

```
class Handler:
    def __init__(self, successor=None):
        self._successor = successor

    def handle(self, request):
        if self._successor:
            self._successor.handle(request)
```

11. **Multiple Interfaces**: Create a class that implements two distinct interfaces.

    Solution:

```
class Walker:
    def walk(self):
        pass

class Swimmer:
    def swim(self):
        pass

class Amphibian(Walker, Swimmer):
    def walk(self):
        print("Walking")

    def swim(self):
        print("Swimming")
```

12. **Adapter Pattern**: Create a pattern that allows two incompatible interfaces to work together.

    Solution:

```
class OldSystem:
    def old_request(self):
        print("Old request")

class Adapter:
    def __init__(self, old_system):
        self._old_system = old_system

```

```
9       def request(self):
10          self._old_system.old_request()
```

13. **Proxy Pattern**: Implement a pattern to provide a surrogate for another object to control access to it.

    Solution:

```
1   class RealSubject:
2       def request(self):
3           print("Real request")
4
5   class Proxy:
6       def __init__(self, real_subject):
7           self._real_subject = real_subject
8
9       def request(self):
10          self._real_subject.request()
```

14. **Composite Pattern**: Implement a pattern to treat individual objects and compositions uniformly.

    Solution:

```
1   class Component:
2       def operation(self):
3           pass
4
5   class Leaf(Component):
6       def operation(self):
7           print("Leaf operation")
8
9   class Composite(Component):
10      def __init__(self):
11          self._children = []
12
13      def operation(self):
14          for child in self._children:
15              child.operation()
16
17      def add(self, component):
18          self._children.append(component)
19
20      def remove(self, component):
21          self._children.remove(component)
```

15. **Mediator Pattern**: Implement a pattern to centralize complex communications and control between related objects.

    Solution:

```
class Mediator:
    def notify(self, sender, event):
        pass

class ConcreteMediator(Mediator):
    def __init__(self, component1, component2):
        self._component1 = component1
        self._component1.mediator = self
        self._component2 = component2
        self._component2.mediator = self

    def notify(self, sender, event):
        if event == "A":
            self._component2.react_on_a()
        elif event == "B":
            self._component1.react_on_b()
```

16. **Momento Pattern**: Implement a pattern to capture an object's internal state so that it can be restored later.

    Solution:

```
class Momento:
    def __init__(self, state):
        self._state = state

    def get_state(self):
        return self._state

class Originator:
    _state = ""

    def set(self, state):
        self._state = state

    def save_to_momento(self):
        return Momento(self._state)

    def restore_from_momento(self, momento):
        self._state = momento.get_state()
```

17. **Flyweight Pattern**: Implement a pattern to minimize memory usage by sharing data between similar objects.

    Solution:

```
class Flyweight:
    def operation(self, extrinsic_state):
        pass

```

```
5  class ConcreteFlyweight(Flyweight):
6      _intrinsic_state = ""
7
8      def operation(self, extrinsic_state):
9          print(extrinsic_state + self._intrinsic_state)
10
11 class FlyweightFactory:
12     _flyweights = {}
13
14     def get_flyweight(self, key):
15         if key not in self._flyweights:
16             self._flyweights[key] = ConcreteFlyweight()
17         return self._flyweights[key]
```

18. **Template Pattern**: Implement a pattern that defines the structure of an algorithm, but allows subclasses to override certain steps.

    Solution:

```
1  class Algorithm:
2      def step1(self):
3          pass
4
5      def step2(self):
6          pass
7
8      def template_method(self):
9          self.step1()
10         self.step2()
11
12 class ConcreteAlgorithm(Algorithm):
13     def step1(self):
14         print("Step 1")
15
16     def step2(self):
17         print("Step 2")
```

19. **Visitor Pattern**: Implement a pattern that lets you add further operations to objects without having to modify them.

    Solution:

```
1  class Visitor:
2      def visit_concrete_element_a(self, element):
3          pass
4
5      def visit_concrete_element_b(self, element):
6          pass
7
8  class ConcreteVisitor1(Visitor):
```

```
9       def visit_concrete_element_a(self, element):
10          print("Concrete visitor 1 for element A")
11
12      def visit_concrete_element_b(self, element):
13          print("Concrete visitor 1 for element B")
14
15  class Element:
16      def accept(self, visitor):
17          pass
18
19  class ConcreteElementA(Element):
20      def accept(self, visitor):
21          visitor.visit_concrete_element_a(self)
22
23  class ConcreteElementB(Element):
24      def accept(self, visitor):
25          visitor.visit_concrete_element_b(self)
```

Chapter 9

Turtle library

9.1 General operations on the Turtle pseudo object

The Turtle library is a popular tool in Python used for creating simple graphics. It is especially suitable for beginner programmers because of its easy-to-understand syntax and interactive nature. The library creates a panel and pen that you can move around to create intricate designs.

The following is an example of how to use the Turtle library to draw a square:

Example 9.1:

```
from turtle import *

# Move turtle
for _ in range(4):
    forward(100)
    right(90)

# Keep the window open
exitonclick()
```

In the above code, we first import the turtle module. Then, we create a turtle screen with a white background. Next, we create a turtle object **my_turtle**. In the for loop, the turtle moves forward by 100 units and then turns right by 90 degrees. This is repeated four times to complete a square. The 'turtle.done()' line is used to keep the turtle graphics window open.

The Turtle library provides numerous other functionalities, such as changing the pen color, filling shapes with color, and much more, allowing users to create a wide range of graphics.

9.2 Changing the pen color and filling shapes

You can change the color of the pen in the Turtle library using the 'color()' function. You can also fill shapes using the **begin_fill()** and **end_fill()** functions. Here is an example:

Example 9.2:

```
from turtle import *

color("red")

begin_fill()
for _ in range(4):
    forward(100)
    right(90)
end_fill()

exitonclick()
```

In the above code, we set the pen color to red using the **color()** function. Then, we use the **begin_fill()** function before drawing the square and the **end_fill()** function after drawing the square. This fills the square with the current pen color.

9.3 Changing the pen size

You can change the pen size in the Turtle library using the **pensize()** function. Here is an example:

Example 9.3:

```
from turtle import *

pensize(10)

for _ in range(4):
    forward(100)
    right(90)

exitonclick()
```

In the above code, we set the pen size to 10 using the **pensize()** function. This makes the lines drawn by the turtle thicker.

Chapter 10

Complex Python Programs

In this section, we will present some more complex Python programs to demonstrate the power and flexibility of the language.

10.1 Factorial Function

Factorial of a non-negative integer n is the product of all positive integers less than or equal to n. It is denoted by n!.

Here's a Python function that calculates the factorial of a number using recursion:

Example 10.1:

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         return n * factorial(n-1)
```

10.2 Fibonacci Sequence

The Fibonacci sequence is a sequence of numbers where the next number in the sequence is found by adding up the two numbers before it.

Here's a Python function that returns the n-th number in the Fibonacci sequence using recursion:

Example 10.2:

```
1 def fibonacci(n):
2     if n <= 1:
3        return n
4     else:
5        return fibonacci(n-1) + fibonacci(n-2)
```

10.3 Factorial Function with Dynamic Programming

By using dynamic programming techniques, we can optimize the calculation of the factorial by storing previously calculated values in a lookup table:

```
def factorial(n, lookup={}):
    if n == 0 or n == 1:
        lookup[n] = 1

    # check if value already exists in lookup table
    if n not in lookup:
        lookup[n] = n * factorial(n-1)

    return lookup[n]
```

This approach ensures that each factorial is only calculated once and then stored in a dictionary for quick access.

10.4 Fibonacci Sequence with Dynamic Programming

Similarly, we can optimize the Fibonacci sequence calculation using dynamic programming to avoid recalculating the same values:

```
def fibonacci(n, lookup={}):
    # Base case
    if n == 0 or n == 1:
        lookup[n] = n

    # If the value is not calculated previously, calculate it
    if n not in lookup:
        lookup[n] = fibonacci(n-1, lookup) + fibonacci(n-2,
           lookup)

    return lookup[n]
```

With dynamic programming, the above Fibonacci function will run significantly faster when called repeatedly or with large inputs, because it avoids the expensive computation of recalculating the Fibonacci sequence from scratch each time.

10.5 Sorting Algorithm: Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.

Here's a Python implementation of Bubble Sort:

Example 10.5:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
```

```
4           for j in range(0, n-i-1):
5               if arr[j] > arr[j+1] :
6                   arr[j], arr[j+1] = arr[j+1], arr[j]
7       return arr
```

10.6 Prime Numbers

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Here's a Python function that checks if a number is prime:

Example 10.6:

```
1 def is_prime(n):
2     if n <= 1:
3         return False
4     for i in range(2, n):
5         if n % i == 0:
6             return False
7     return True
```

Chapter 11

Data Structures in Python

Python incorporates several built-in data structures for the efficient storage and manipulation of data, which include lists, tuples, sets, and dictionaries.

11.1 Lists

In Python, a list is an ordered collection of items which are not restricted to a specific type. Lists are defined by enclosing a comma-separated sequence of items within square brackets '[]'. Here is an illustrative example of a list in Python:

Example 11.1:

```
fruits = ["apple", "banana", "cherry"]
```

You can add an item to the list using the append() method, and remove an item using the remove() method:

Example 11.1:

```
fruits.append("date")
fruits.remove("banana")
```

11.2 Tuples

A tuple is akin to a list in terms of being an ordered collection of items, however, it is defined with parentheses '()' as opposed to square brackets. The principal difference lies in the fact that tuples are immutable. Here's an example:

Example 11.2:

```
fruits = ("apple", "banana", "cherry")
```

Accessing elements in a tuple is similar to accessing elements in a list:

Example 11.2:

```
print(fruits[1])  # Output: "banana"
```

11.3 Sets

A set in Python is characterized as an unordered collection of unique items. Sets are defined by enclosing a comma-separated list of items within curly braces ‘‘. Here’s an example of a set:

Example 11.3:

```
fruits = {"apple", "banana", "cherry"}
```

You can add an item to a set using the add() method, and remove an item using the remove() method:

Example 11.3:

```
fruits.add("date")
fruits.remove("banana")
```

11.4 Dictionaries

A dictionary in Python constitutes an unordered collection of items. Each item stored in a dictionary has a key and a corresponding value. The key can be utilized to access the related value. Dictionaries are defined by enclosing a comma-separated list of key-value pairs within curly braces ‘‘. The key and value are separated by a colon ‘:‘. Here’s an example of a dictionary:

Example 11.4:

```
person = {"name": "Alice", "age": 25}
```

You can change the value of an item in a dictionary like this:

Example 11.4:

```
person["age"] = 30
```

You can add a new item to a dictionary like this:

Example 11.4:

```
person["profession"] = "Engineer"
```

This page was left empty with accordance to author's hidden intention...

Chapter 12

Projects and training with Tasks

12.1 Project 1

Project Description

The task of this program is to create an interactive drawing application using the turtle module in Python. The program allows the user to control a turtle object on the screen and perform various actions.

Specification(may be modified by user as the task is creative one)

Here is a breakdown of the tasks performed by the program:

[label=0.]Set up the Turtle:

1.
   - Create a turtle object.
   - Set the turtle's speed to 100.
   - Set the turtle's initial color to red.
   - Set the turtle's width to 1.
   - Set the turtle's shape to "turtle".
   - Put the turtle's pen down to start drawing.
2. Define Color Changing Functions:
   - Implement `turtle_color_red()` to change the turtle's color to red.
   - Implement `turtle_color_green()` to change the turtle's color to green.
3. Define Mouse Event Function:
   - Implement `fxn(x, y)` to handle mouse drag events.
   - Stop backtracking of the turtle.
   - Adjust the turtle's angle and direction towards the new coordinates (x, y).
   - Move the turtle to the new coordinates (x, y).

- Enable the function to be called again for further dragging.

4. Define Keyboard Event Functions:
   - Implement `move_forward()` to move the turtle forward by 50 units.
   - Implement `move_backward()` to move the turtle backward by 50 units.
   - Implement `turn_left()` to rotate the turtle left by 45 degrees.
   - Implement `turn_right()` to rotate the turtle right by 45 degrees.
   - Implement `fill_screen()` to fill the entire screen with color.
5. Set up Event Listeners:
   - Get the turtle's screen object.
   - Enable listening for key and mouse events.
   - Register event handlers for specific keys and mouse clicks.
   - When events occur, the corresponding functions are called to perform the desired actions.
6. Enter the Main Event Loop:
   - Start the turtle's event loop.
   - The program continuously listens for events and responds accordingly.
   - The program remains interactive until the window is closed.

The main goal of this program is to provide an interactive drawing experience where the user can control the turtle's movement, change its color, and fill the screen with color. The program utilizes various event-driven functions to respond to user inputs and update the turtle's behavior on the screen.

The one of possible solutions

Example 12.1:

```
from turtle import *
from random import randint
from time import sleep

t = Turtle()  # Create a turtle object
t.speed(100)  # Set the turtle's speed
t.color("red")  # Set the turtle's color
t.width(1)  # Set the turtle's width
t.shape("turtle")  # Set the turtle's shape
t.pendown()  # Put the turtle's pen down to start drawing
```

```
11
12 def turtle_color_red():
13     t.color("red")  # Change the turtle's color to red
14
15 def turtle_color_green():
16     t.color("green")  # Change the turtle's color to green
17
18 def fxn(x, y):
19     t.ondrag(None)  # Stop backtracking
20     t.setheading(t.towards(x, y))  # Move the turtle's angle
          and direction towards x and y
21     t.goto(x, y)  # Go to x, y
22     t.ondrag(fxn)  # Call the function again for further
          dragging
23
24 def move_forward():
25     t.forward(50)  # Move the turtle forward by 50 units
26
27 def move_backward():
28     t.backward(50)  # Move the turtle backward by 50 units
29
30 def turn_left():
31     t.left(45)  # Turn the turtle left by 45 degrees
32
33 def turn_right():
34     t.right(45)  # Turn the turtle right by 45 degrees
35
36 def fill_screen():
37     t.begin_fill()  # Start filling the shape with color
38     t.goto(-scr.window_width() / 2, -scr.window_height() / 2)
          # Go to the bottom-left corner of the screen
39     t.goto(scr.window_width() / 2, -scr.window_height() / 2)
          # Go to the bottom-right corner of the screen
40     t.goto(scr.window_width() / 2, scr.window_height() / 2)
          # Go to the top-right corner of the screen
41     t.goto(-scr.window_width() / 2, scr.window_height() / 2)
          # Go to the top-left corner of the screen
42     t.goto(-scr.window_width() / 2, -scr.window_height() / 2)
          # Go back to the bottom-left corner of the screen
43     t.end_fill()  # Stop filling the shape with color
44
45 t.speed(10)  # Set the turtle's speed
46 scr = t.getscreen()  # Get the turtle's screen
47 scr.listen()  # Enable listening for key and mouse events
48 scr.onkey(turtle_color_red, "r")  # Call turtle_color_red
     function when 'r' key is pressed
49 scr.onkey(turtle_color_green, "g")  # Call turtle_color_green
     function when 'g' key is pressed
```

```
50 scr.onkey(move_forward, "Up")  # Call move_forward function
     when 'Up' arrow key is pressed
51 scr.onkey(move_backward, "Down")  # Call move_backward
     function when 'Down' arrow key is pressed
52 scr.onkey(turn_left, "Left")  # Call turn_left function when
     'Left' arrow key is pressed
53 scr.onkey(turn_right, "Right")  # Call turn_right function
     when 'Right' arrow key is pressed
54 scr.onclick(fxn)  # Call fxn function when the screen is
     clicked
55 scr.onkey(fill_screen, "f")  # Call fill_screen function when
      'f' key is pressed
56 scr.mainloop()  # Start the turtle's event loop
```

12.2 Project 2 specification

Task 1 (10 points)

Construct a list of MIT classes in the following format (in this context, the list is not a data structure, but a simple enumeration of elements):

- Course 1 - Civil and Environmental Engineering
- Course 2 - Mechanical Engineering
- Course 3 - Materials Science and Engineering
- Course 4 - Architecture
- Course 5 - Chemistry
- Course 6 - Electrical Engineering and Computer Science
- Course 7 - Biology
- Course 8 - Physics
- Course 9 - Brain and Cognitive Sciences
- Course 10 - Chemical Engineering

Attention! Employing the `.format` function will guarantee acquiring 5 points of task performance.

Task 2 (10 points)

Request a user to input a number of an integer in the range $[1, \ldots, i, \ldots, 10]$, where $i \in \mathbb{N}$, and $1 \leq i \leq 10$. Once the program receives the number from the user, it returns the name of the course corresponding to the given number.
Example:

```
Program:  Enter the number...
User:  1
Program:  Course 1 - Civil and Environmental Engineering
```

Task 3 (10 points)

Consider the fraction $\frac{a}{b} = \frac{nominator}{denominator}$. In this task, ask the user to input the numerator and denominator values. The program should return the result and (if possible) the remainder of this division operation. For this task, create a variable used in a while loop called `i = a` and another variable called `answer`; while `i > 0`, let `i = i - b` and `answer = answer + 1`. If `i == 0`, the program should print the result as the value of the `str` function type in the following format: `"Result" + str(answer)`. If `i != 0`, the program prints `"Result"` as the `answer - 1` string value and then the program prints: `"Reminder:" + str(i+b)`. The names of any variables here in the task may be chosen by the student. Following this example for naming the variables may make the solution more readable for the grader.

12.3 Project 3 specification

Task 1 (10 points)

Write a function `calculate_the_sum_of_n_numbers(n)` which calculates the sum of n numbers, where n is a natural finite number.
Example:

```
calculate_the_sum_of_n_numbers(100)
```

This will be used to calculate the sum $1 + 2 + 3 + \cdots + 100 = \frac{100*101}{2} = 50 * 101 = 5050$.

Task 2 (10 points)

Write a function `nums_to_n(n)` to print all the odd numbers from 1 to n, where n is a natural finite number, and if the number is divisible by 5, the function should draw the user's attention to it. The function does not return anything, it prints all odd numbers from 1 to n.
Example:

```
nums_to_n(10)
```

This code should print:

```
1;
3;
5 is divisible by 5;
7;
9;
```

Remember to include a semicolon at the end of each line!

Task 3 (10 points)

Write a function named `analyse_the_number(x, a_less, b_greater)` which will:

- Inform the user if the number is odd or even;
- Inform the user if the number is greater than or equal to `"b_greater"`;
- Inform the user if the number is less than or equal to `"a_less"`;
- Tell the user what is the factorial of x: if $x = 6$, it will print $6 * 5 * 4 * 3 * 2 * 1$;
- Print $(x^{b\_greater})^{a\_less}$;
- If the number is from the set $[-2, 2]$ (the number x may be 2, -1, 0, 1, 2), the function will print the number as follows: if the number x is 2, the program will print: "two".

12.4 Specification of the Project 4

Task 1 (10 points)

Write a function `calculate_factorial(n)` that calculates and returns the factorial of a natural number n. This function should raise an exception if n is negative or if it's not an integer.
Example:

```
1 calculate_factorial(5)
```

This will return $5 * 4 * 3 * 2 * 1 = 120$.

Task 2 (10 points)

Write a function `print_fibonacci(n)` that prints the first n numbers in the Fibonacci sequence. The function should print each number on a new line. The function does not return anything, it only prints the Fibonacci numbers.
Example:

```
1 print_fibonacci(7)
```

This code should print:

```
0
1
1
2
3
5
8
```

Task 3 (10 points)

Write a function `is_prime(n)` that returns `True` if a number is prime and `False` otherwise. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.
Example:

```
1 is_prime(7)
```

This will return `True` because 7 is a prime number.

12.5 Specification of the Project 5

Task 1 (10 points)

Write a function `read_file(file_name)` that reads a text file and returns a list where each element is a line in the file. This function should handle possible exceptions due to file handling issues.
Example:

```
read_file('example.txt')
```

This function will read the text file 'example.txt' and return its contents as a list of strings.

Task 2 (10 points)

Write a function `find_pattern(file_name, pattern)` that uses regular expressions to find all occurrences of a specific pattern in a text file. This function should return a list of all matches.
Example:

```
find_pattern('example.txt', r'\d+')
```

This function will find all sequences of digits in 'example.txt' and return them as a list.

Task 3 (20 points)

Write a function `web_scrape(url, tag)` that uses BeautifulSoup to scrape a specific tag from a webpage. This function should return a list of strings, each of which corresponds to the inner content of one instance of the tag in the HTML of the webpage.
Example:

```
web_scrape('https://example.com', 'h1')
```

This function will return the inner content of all 'h1' tags in the HTML of 'https://example.com'.

12.6 Specification of the Project 6

Task 1 (10 points)

Write a function `is_prime(n)` that checks whether a given integer n is a prime number. The function should return `True` if the number is prime and `False` otherwise.
Example:

```
1 is_prime(7)
```

This function will return `True`, as 7 is a prime number.

Task 2 (15 points)

Write a function `calculate_factorial(n)` that calculates and returns the factorial of a number n. The function should raise an exception if n is negative or not an integer.
Example:

```
1 calculate_factorial(5)
```

This function will return $5 * 4 * 3 * 2 * 1 = 120$.

Task 3 (15 points)

Write a function `calculate_combination(n, r)` that calculates and returns the combination of n items taken r at a time, where n and r are natural numbers and $r \leq n$.
Example:

```
1 calculate_combination(5, 2)
```

This function will return 10, as there are 10 ways to choose 2 items from 5.

Task 4 (20 points)

Write a function `calculate_mean_median(file_name)` that reads a file containing a list of numbers, one per line, and returns a tuple containing the mean and median of these numbers. The function should handle possible exceptions due to file handling issues.
Example:

```
1 calculate_mean_median('numbers.txt')
```

This function will return a tuple (mean, median) calculated from the numbers in the file 'numbers.txt'.

Task 5 (20 points)

Write a function `plot_distribution(file_name)` that reads a file containing a list of numbers and uses matplotlib to plot a histogram of the data. The function should handle possible exceptions due to file handling issues.
Example:

```
1 plot_distribution('numbers.txt')
```

This function will read the numbers from the file 'numbers.txt' and display a histogram of the data.

12.7 Specification of the Project 7

Task 1 (20 points)

Define a Python class `Matrix` to represent a mathematical matrix. The class should include methods for matrix addition, matrix subtraction, matrix multiplication, and transpose of a matrix.
Example:

```
matrix1 = Matrix([[1, 2], [3, 4]])
matrix2 = Matrix([[5, 6], [7, 8]])
print(matrix1.add(matrix2))
print(matrix1.subtract(matrix2))
print(matrix1.multiply(matrix2))
print(matrix1.transpose())
```

This should print the results of the matrix operations.

Task 2 (20 points)

Write a Python function `fibonacci(n)` that calculates the nth Fibonacci number using recursion.
Example:

```
print(fibonacci(10))
```

This should print the 10th Fibonacci number.

Task 3 (20 points)

Write a Python function `simulate_random_walk(steps)` that simulates a one-dimensional random walk of a specified number of steps using NumPy and returns the final position.
Example:

```
print(simulate_random_walk(1000))
```

This should print the final position of the random walk after 1000 steps.

Task 4 (20 points)

Write a Python function `plot_random_walk(steps)` that simulates a one-dimensional random walk of a specified number of steps and creates a plot of the walk using Matplotlib.
Example:

```
plot_random_walk(1000)
```

This should display a plot of the random walk after 1000 steps.

Task 5 (20 points)

Write a Python function `simulate_random_walks(number_of_walks, steps)` that simulates a specified number of one-dimensional random walks of a specified number of steps each. The function should return the average final position and the standard

deviation of the final positions of the walks.
Example:

```
average, std_dev = simulate_random_walks(100, 1000)
print('Average:', average)
print('Standard Deviation:', std_dev)
```

This should print the average and standard deviation of the final positions of 100 random walks of 1000 steps each.

12.8 Specification of the Project 8

Task 1 (25 points)

Write a Python function `solve_quadratic(a, b, c)` that solves a quadratic equation of the form $ax^2 + bx + c = 0$. The function should return the roots of the equation.
Example:

```
print(solve_quadratic(1, -3, 2))
```

This should print the roots of the equation $x^2 - 3x + 2 = 0$.

Task 2 (25 points)

Write a Python function `function_analysis(f, x_range)` that plots a given function $f(x)$ over a specified range of x values, calculates the maximum and minimum of the function over that range, and integrates the function over that range. The function should use the matplotlib library for plotting and the scipy library for integration.
Example:

```
import numpy as np
def f(x):
    return x**2
function_analysis(f, np.linspace(0, 1, 100))
```

This should plot the function $f(x) = x^2$ over the range from 0 to 1, print the maximum and minimum values over that range, and print the integral of $f(x)$ over that range.

Task 3 (25 points)

Write a Python function `solve_system(A, b)` that solves a system of linear equations given by the matrix equation $Ax = b$. The function should use the numpy library to solve the equation.
Example:

```
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
print(solve_system(A, b))
```

This should print the solution to the system of equations $3x_1 + x_2 = 9$ and $x_1 + 2x_2 = 8$.

Task 4 (25 points)

Write a Python function `calculate_derivative(f, x)` that calculates the derivative of a function at a given point. The function should use the scipy library to calculate the derivative.
Example:

```
def f(x):
    return x**3
print(calculate_derivative(f, 2))
```

This should print the derivative of the function $f(x) = x^3$ at $x = 2$.

12.9 Specification of the Project 9

Task 1 (25 points)

Write a Python function `polynomial_coefficients(n)` that generates the coefficients of a polynomial of degree n. The coefficients should be random integers in the range from -10 to 10. The function should return a list of the coefficients.
Example:

```
print(polynomial_coefficients(3))
```

This might print the list `[-2, 5, -10, 1]`, which corresponds to the polynomial $-2x^3 + 5x^2 - 10x + 1$.

Task 2 (25 points)

Write a Python function `polynomial_value(coefs, x)` that calculates the value of a polynomial at a given point x. The coefficients of the polynomial are given as a list of numbers.
Example:

```
coefs = [1, -3, 2]
x = 2
print(polynomial_value(coefs, x))
```

This should print the value of the polynomial $x^2 - 3x + 2$ at $x = 2$.

Task 3 (25 points)

Write a Python function `solve_quadratic(coefs)` that solves a quadratic equation given by its coefficients. The coefficients are given as a list of three numbers. The function should return the roots of the equation.
Example:

```
coefs = [1, -3, 2]
print(solve_quadratic(coefs))
```

This should print the roots of the quadratic equation $x^2 - 3x + 2 = 0$.

Task 4 (25 points)

Write a Python function `plot_polynomial(coefs, x_range)` that plots a polynomial given by its coefficients over a specified range of x values. The coefficients are given as a list of numbers. The function should use the matplotlib library for plotting.
Example:

```
coefs = [1, -3, 2]
x_range = np.linspace(-10, 10, 200)
plot_polynomial(coefs, x_range)
```

This should plot the polynomial $x^2 - 3x + 2$ over the range from -10 to 10.

12.10 Specification of the Project 10

Task 1 (20 points)

Write a Python function `scrape_webpage(url)` that takes in a URL of a webpage and scrapes the text content of that webpage using the BeautifulSoup library. The function should return a string of the webpage content. You must handle exceptions properly in your function to account for potential errors like network issues or invalid URLs.

Task 2 (20 points)

Extend the `scrape_webpage(url)` function to now save the scraped webpage content into a .txt file. The filename should be based on the webpage's title.

Task 3 (30 points)

Write a Python function `count_word_frequency(text)` that takes in a string of text and counts the frequency of each word in that text. The function should ignore common stopwords (like 'a', 'the', 'and', etc.) and return a dictionary where the keys are words and the values are their respective frequencies.

Task 4 (30 points)

Write a Python function `plot_word_frequency(word_frequency)` that takes in a dictionary of word frequencies (like the one produced by the `count_word_frequency(text)` function) and generates a bar chart of the top 10 most frequent words using matplotlib. The chart should have words on the x-axis and their frequencies on the y-axis.

Chapter 13

Specification tasks ”S”

For this parf of tasks the user is supposed to use the testing functions included in the content of the book. All tasks are structured in the way that reaced should implement the function described in the specification for each task of the separate S training tasks.

13.1 Task S1

Task 1 (10 points)

Write a function `construct_course_list()` that constructs and returns a list of MIT classes in the following format:

- Course 1 - Civil and Environmental Engineering
- Course 2 - Mechanical Engineering
- Course 3 - Materials Science and Engineering
- Course 4 - Architecture
- Course 5 - Chemistry
- Course 6 - Electrical Engineering and Computer Science
- Course 7 - Biology
- Course 8 - Physics
- Course 9 - Brain and Cognitive Sciences
- Course 10 - Chemical Engineering

Returns:

- `course_list` (list): A list of strings representing the MIT course names.

Example:

```
1 construct_course_list()
```

This will return the following list:

```
[
"Course 1 - Civil and Environmental Engineering",
"Course 2 - Mechanical Engineering",
"Course 3 - Materials Science and Engineering",
"Course 4 - Architecture",
"Course 5 - Chemistry",
"Course 6 - Electrical Engineering and Computer Science",
"Course 7 - Biology",
"Course 8 - Physics",
"Course 9 - Brain and Cognitive Sciences",
"Course 10 - Chemical Engineering"
]
```

Task 2 (10 points)

Write a function `get_course_name(number)` that takes a number as input and returns the name of the MIT course corresponding to the given number.

Function Input:

- `number` (int): The course number to retrieve the name for.

Returns:

- `course_name` (str): The name of the MIT course corresponding to the given number.

Example:

```
get_course_name(1)
```

This will return the following string:

```
"Course 1 - Civil and Environmental Engineering"
```

Task 3 (10 points)

Create a function that takes a day of the year (an integer from 1 to 365) as input and returns the corresponding month. You can assume a non-leap year. For simplicity, you can consider each month to have a fixed number of days as it has in the callendar (January: 31 days, February: 28 days(you has to consider this number in your solution) and 29 in every leap year, March: 31 days, April: 30 days, May: 31 days, June: 30 days, July: 31 days, August: 31 days, September: 30 days, October: 31 days, November: 30 days, December: 31 days,). The function should return the month as a string. You can choose the month names as per your preference. **Example:**

```
print(get_month(75))  # Output: March
```

Task 4 (10 points)

Create a function `perform_operation(a, b, c=0, d=0, e=0, f=0, g=0, h=0, i=0, operator='+')` that takes two numbers (`a` and `b`) and up to eight additional numbers (`c` to `i`) as arguments. The function should also accept an operator (`+`, `-`, `*`, `/`) as a keyword argument. The function should perform the corresponding operation on all the provided

numbers and return the result. If the operator is division (/) and the second number is zero, the function should return an error message.

Function Inputs:

- `a` (numeric): The first number.
- `b` (numeric): The second number.
- `c` to `i` (numeric, optional): Up to eight additional numbers.
- `operator` (str, optional): The operator to perform the operation. Default is `'+'`.

Returns:

- `result` (numeric or str): The result of the operation. If the operator is division (/) and the second number is zero, return the error message: `"Error: Division by zero"`.

Example:

```
print(perform_operation(5, 3, 2, 4, operator='+')) # Output:
    14
print(perform_operation(10, 2, 3, operator='*')) # Output: 60
print(perform_operation(5, 0, operator='/')) # Output: "Error
    : Division by zero"
```

Note: The specific implementation details and variable names may vary.

Task 5 (10 points)

Consider the fraction $\frac{a}{b} = \frac{nominator}{denominator}$, where a represents the numerator and b represents the denominator. In this task, ask the user to input the numerator and denominator values. The program should calculate the result of the division and (if possible) the remainder of the division operation.

Create a function `divide_fraction(numerator, denominator)` that takes the numerator and denominator as input and returns the division result and remainder (if applicable).

Within the function, create a variable `i` and set it equal to `numerator`, and create another variable `answer` and set it to 0.

Use a while loop to iterate while `i` is greater than 0. In each iteration, subtract the `denominator` from `i` and increment `answer` by 1.

If `i` becomes 0 after the loop, return a string in the format: "Result: `answer`".

If `i` is not 0, return a string in the format: "Result: `answer-1` \n Reminder: `i+denominator`".

Feel free to choose your own variable names for this task.

Function Inputs:

- `numerator` (int): The numerator of the fraction.
- `denominator` (int): The denominator of the fraction.

Returns:

- `result` (str): A string containing the division result and remainder (if applicable) in the format mentioned above.

Example:

```
1 divide_fraction(7, 3)
```

This will return the following string:

```
"Result: 2
 Reminder: 1"
```

Maybe the usage of the \n will help.

Note: The specific implementation details and variable names may vary.

13.2 Task S2

Task 1 (10 points)

Write a function `calculate_the_sum_of_n_numbers(n)` which calculates the sum of n numbers, where n is a natural finite number.
Example:

```
calculate_the_sum_of_n_numbers(100)
```

This will be used to calculate the sum $1 + 2 + 3 + \cdots + 100 = \frac{100*101}{2} = 50 * 101 = 5050$.

Task 2 (10 points)

Write a function `nums_to_n(n)` to print all the odd numbers from 1 to n, where n is a natural finite number, and if the number is divisible by 5, the function should draw the user's attention to it. The function does not return anything, it prints all odd numbers from 1 to n.
Example:

```
nums_to_n(10)
```

This code should print:

```
1;
3;
5 is divisible by 5;
7;
9;
```

Remember to include a semicolon at the end of each line!

Task 3 (10 points)

Write a function named `analyse_the_number(x, a_less, b_greater)` which will:

- Inform the user if the number is odd or even;
- Inform the user if the number is greater than or equal to `"b_greater"`;
- Inform the user if the number is less than or equal to `"a_less"`;
- Tell the user what is the factorial of x: if $x = 6$, it will print $6 * 5 * 4 * 3 * 2 * 1$;
- Print the number$(x^{b_greater})^{a_less}$;
- If the number is from the set $[-2, 2]$ (the number x may be 2, -1, 0, 1, 2), the function will print the number as follows: if the number x is 2, the program will print: "two".

Task 4 (10 points)

Solve the following system of linear equations:

$$2x + 3y = 7$$
$$4x - 2y = 10$$

Write a function `solve_system(a1, b1, c1, a2, b2, c2)` that takes the coefficients of two linear equations of the form $a_1x + b_1y = c_1$ and $a_2x + b_2y = c_2$ as inputs and solves the system of equations. The function should return the solution as a tuple (x, y) representing the values of x and y that satisfy both equations. If the system of equations has no solution or infinite solutions, return "No unique solution".

Example:

```
solve_system(2, 3, 7, 4, -2, 10) # Output: (2.0, 1.0)
```

```
solve_system(1, -2, 3, 2, -4, 6) # Output: "No unique
    solution"
```

```
solve_system(1, 2, 3, 2, 4, 6) # Output: "No unique solution"
```

```
solve_system(0, 0, 0, 0, 0, 0) # Output: "No unique solution"
```

Note: Show all the necessary steps and explanations to justify your solutions. You may use any appropriate methods or techniques taught in secondary school mathematics.

Task 5 (10 points)

Write a Python function `is_palindrome(s)` that takes a string `s` as input and returns `True` if the string is a palindrome and `False` otherwise. A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward, ignoring spaces, punctuation, and capitalization.

Example:

```
is_palindrome("racecar") # Output: True
```

```
is_palindrome("Hello, World!") # Output: False
```

Write the function `is_palindrome` to solve the task and test it with different strings.

13.3 Task S3

Task 1 (10 points)

Write a function `calculate_factorial(n)` that calculates and returns the factorial of a natural number n. This function should raise an exception if n is negative or if it's not an integer.

Example:

```
calculate_factorial(5) # Output: 120
```

This will return $5 * 4 * 3 * 2 * 1 = 120$.

Task 2 (10 points)

Write a function `print_fibonacci(n)` that prints the first n numbers in the Fibonacci sequence. The function should print each number on a new line. The function does not return anything, it only prints the Fibonacci numbers.

Example:

```
print_fibonacci(7)
```

This code should print:

```
0
1
1
2
3
5
8
```

Task 3 (10 points)

Write a function `is_prime(n)` that returns `True` if a number is prime and `False` otherwise. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Example:

```
is_prime(7)
```

This will return `True` because 7 is a prime number.

Task 4 (10 points)

Write a function `find_palindromes(words)` that takes a list of words and returns a new list containing only the palindromes from the original list. A palindrome is a word that reads the same forwards and backwards. The function should ignore case sensitivity, meaning "Mom" and "mOm" should be considered palindromes.

Example:

```
words = ["level", "deed", "hello", "Madam", "world"]
find_palindromes(words)
```

This will return `["level", "deed", "Madam"]`, as these words are palindromes.

Task 5 (10 points)

Write a function `calculate_mean(numbers)` that takes a list of numbers and returns the mean (average) value. The function should return the mean as a floating-point number.

Example:

```
numbers = [1, 2, 3, 4, 5]
calculate_mean(numbers) # Output: 3.0
```

This will return the mean of the numbers in the list, which is 3.0.

13.4 Task S4

Task 1 (10 points)

Write a function `calculate_poly_function_val(a, b, c, d, e, k, g)` that calculates the value of a function $f(a, b, c, d, e, k, g) = 10a^3 + 11b^{10} + 12c^3 + 3d^2 + 6e^{18} + 67k^{12} + 22g^4 + \frac{127}{168}a^4 + 4e + \frac{6}{7}g^2 + \sqrt[3]{\frac{3}{2}a^2} - \frac{2}{5}d + 10e^2 - \pi k + \frac{(c+d+k+g)a^2}{b}$ at the given point (a, b, c, d, e, k, g).

Function Inputs:

- `a` (float): The value of the variable a.
- `b` (float): The value of the variable b.
- `c` (float): The value of the variable c.
- `d` (float): The value of the variable d.
- `e` (float): The value of the variable e.
- `f` (float): The value of the variable k.
- `g` (float): The value of the variable g.

Returns:

- `result` (float): The calculated value of the function $f(a, b, c, d, e, k, g)$ at the given point.

Example:

```
calculate_poly_function_val(1.5, 2.3, -0.7, 4.2, 0.8, -1.1,
    3.6)
```

This will return the calculated value of the function at the given point.

Note: Make sure to handle any necessary mathematical operations correctly according to the specified function.

Task 2 (10 points)

Write a function `calculate_factorial(n)` that calculates and returns the factorial of a number n. The function should raise an exception if n is negative or not an integer.
Example:

```
calculate_factorial(5)
```

This function will return $5 * 4 * 3 * 2 * 1 = 120$.

Task 3 (10 points)

Write a function `calculate_combination(n, r)` that calculates and returns the combination of n items taken r at a time, where n and r are natural numbers and $r \leq n$. More presize information about the Newtonian binomial symbol defined to tel how many ways exists to choose r different items from n is below

Example:

```
calculate_combination(5, 2)
```

This function will return 10, as there are 10 ways to choose 2 items from 5.

The combination, denoted as $\binom{n}{r}$, represents the number of ways to choose r items from a set of n distinct items, without regard to their order. It can be calculated using the formula:

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Task 4 (10 points)

Write a function `analyze_apple_weights(apple_weights)` that takes a list of apple weights as input and performs the following analysis:

Calculate the average weight of the apples. Determine the maximum weight among the apples. Determine the minimum weight among the apples.

The function should return a dictionary containing the analysis results.

Function Input:

- `apple_weights` (list): A list of floating-point numbers representing the weights of the apples.

Returns:

- `analysis_results` (dictionary): A dictionary containing the following analysis results: - `"average_weight"`: The average weight of the apples. - `"max_weight"`: The maximum weight among the apples. - `"min_weight"`: The minimum weight among the apples.

Example:

```
apple_weights = [0.2, 0.5, 0.3, 0.6, 0.4]
analyze_apple_weights(apple_weights)
```

This will return the following dictionary:

```
{
"average_weight": 0.4,
"max_weight": 0.6,
"min_weight": 0.2
}
```

Note: This simplified task focuses on calculating the average, maximum, and minimum weights of the apples in the dataset. It omits the additional analysis requirements mentioned in the original task.

Task 5 (10 points)

Consider the list of MIT classes in the following format:

- Course 1 - Civil and Environmental Engineering
- Course 2 - Mechanical Engineering
- Course 3 - Materials Science and Engineering
- Course 4 - Architecture
- Course 5 - Chemistry
- Course 6 - Electrical Engineering and Computer Science
- Course 7 - Biology
- Course 8 - Physics
- Course 9 - Brain and Cognitive Sciences
- Course 10 - Chemical Engineering

Write a function `get_course_name(number)` that takes a number as input and returns the name of the MIT course corresponding to the given number.

Function Input:

- `number` (int): The course number to retrieve the name for.

Returns:

- `course_name` (str): The name of the MIT course corresponding to the given number.

Example:

```
1 get_course_name(1)
```

This will return the following string:

```
"Course 1 - Civil and Environmental Engineering"
```

13.5 Task S5

Task 1 (10 points)

Create a function `perform_operation(a, b, c=0, d=0, e=0, f=0, g=0, h=0, i=0, operator='+')` that takes two numbers (`a` and `b`) and up to eight additional numbers (`c` to `i`) as arguments. The function should also accept an operator (`+`, `-`, `*`, `/`) as a keyword argument. The function should perform the corresponding operation on all the provided numbers and return the result. If the operator is division (`/`) and the second number is zero, the function should return an error message.

Function Inputs:

- `a` (numeric): The first number.
- `b` (numeric): The second number.
- `c` to `i` (numeric, optional): Up to eight additional numbers.
- `operator` (str, optional): The operator to perform the operation. Default is `'+'`.

Returns:

- `result` (numeric or str): The result of the operation. If the operator is division (`/`) and the second number is zero, return the error message: `"Error: Division by zero"`.

Example:

```
print(perform_operation(5, 3, 2, 4, operator='+')) # Output:
    14
print(perform_operation(10, 2, 3, operator='*')) # Output: 60
print(perform_operation(5, 0, operator='/')) # Output: "Error
    : Division by zero"
```

Note: The specific implementation details and variable names may vary.

Task 2 (10 points)

Write a Python function `fibonacci(n)` that calculates the nth Fibonacci number using recursion.

Example:

```
print(fibonacci(10))
```

This should print the 10th Fibonacci number.

Task 3 (10 points)

Let's calculate the value of the iterated integral of a differentiable and continuous function $f(x, y, x) = 1$:

$$\int_a^b \int_c^d \int_e^g 1 \, dx \, dy \, dz = (b-a)(d-c)(g-e)$$

Write a function `calculate_iterated_integral(a, b, c, d, e, g)` that calculates the value of the iterated integral of a differentiable and continuous function $f(x, y, z) = 1$. The function should take the limits of integration aaa, bbb, ccc, ddd, eee, and ggg as inputs and return the calculated value of the iterated integral.

Function Inputs:

- `a` (numeric): The lower limit of integration for the variable xxx.
- `b` (numeric): The upper limit of integration for the variable xxx.
- `c` (numeric): The lower limit of integration for the variable yyy.
- `d` (numeric): The upper limit of integration for the variable yyy.
- `e` (numeric): The lower limit of integration for the variable zzz.
- `g` (numeric): The upper limit of integration for the variable zzz.

Returns:

- `result` (numeric): The calculated value of the iterated integral, which is equal to $(b - a)(d - c)(g - e)$.

Example:

```
result = calculate_iterated_integral(1, 3, 2, 4, 0, 5)
print(result) # Output: 36
```

Note: The specific implementation details, function name, and variable names may vary.

Task 4 (10 points)

Write a Python function `is_palindrome(s)` that takes a string `s` as input and returns `True` if the string is a palindrome and `False` otherwise. A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward, ignoring spaces, punctuation, and capitalization.

Example:

```
is_palindrome("racecar") # Output: True
```

```
is_palindrome("Hello, World!") # Output: False
```

Write the function `is_palindrome` to solve the task and test it with different strings.

Task 5 (10 points)

Create a function that takes a day of the year (an integer from 1 to 365) as input and returns the corresponding month. You can assume a non-leap year. For simplicity, you can consider each month to have a fixed number of days. The function should return the month as a string. You can choose the month names as per your preference. **Example:**

```
print(get_month(75))  # Output: March
```

Chapter 14

Additional ”P” tasks for mini-projects

This section is more like the projects' tasks before the separate section with the specification tasks driven with testing functions.

14.1 Project 1: Turtle drawing

Description

The task of this program is to create an interactive drawing application using the turtle module in Python. The program allows the user to control a turtle object on the screen and perform various actions.

Specification

Here is a breakdown of the tasks performed by the program:

[label=0.]Set up the Turtle:

1.
   - Create a turtle object.
   - Set the turtle's speed to 100.
   - Set the turtle's initial color to red.
   - Set the turtle's width to 1.
   - Set the turtle's shape to ”turtle”.
   - Put the turtle's pen down to start drawing.
2. Define Color Changing Functions:
   - Implement `turtle_color_red()` to change the turtle's color to red.
   - Implement `turtle_color_green()` to change the turtle's color to green.
3. Define Mouse Event Function:
   - Implement `fxn(x, y)` to handle mouse drag events.

- Stop backtracking of the turtle.
- Adjust the turtle's angle and direction towards the new coordinates (x, y).
- Move the turtle to the new coordinates (x, y).
- Enable the function to be called again for further dragging.

4. Define Keyboard Event Functions:
   - Implement `move_forward()` to move the turtle forward by 50 units.
   - Implement `move_backward()` to move the turtle backward by 50 units.
   - Implement `turn_left()` to rotate the turtle left by 45 degrees.
   - Implement `turn_right()` to rotate the turtle right by 45 degrees.
   - Implement `fill_screen()` to fill the entire screen with color.
5. Set up Event Listeners:
   - Get the turtle's screen object.
   - Enable listening for key and mouse events.
   - Register event handlers for specific keys and mouse clicks.
   - When events occur, the corresponding functions are called to perform the desired actions.
6. Enter the Main Event Loop:
   - Start the turtle's event loop.
   - The program continuously listens for events and responds accordingly.
   - The program remains interactive until the window is closed.

The main goal of this program is to provide an interactive drawing experience where the user can control the turtle's movement, change its color, and fill the screen with color. The program utilizes various event-driven functions to respond to user inputs and update the turtle's behavior on the screen.

14.2 Project 2: Number Guessing Game

Description

In this project, you will create a number guessing game. The program will generate a random number between a specified range, and the user will have to guess the number within a certain number of attempts. After each guess, the program will provide feedback to the user if the guess is too high or too low. The game will continue until the user guesses the correct number or runs out of attempts.

Specifications

- The program should generate a random number between a specified range.
- The user should be prompted to enter their guess.
- The program should provide feedback to the user if the guess is too high or too low.
- The program should keep track of the number of attempts.
- The game should continue until the user guesses the correct number or runs out of attempts.
- The program should display a message indicating whether the user won or lost the game.

14.3 Project 3: To-Do List

Description

In this project, you will create a simple to-do list application. The program will allow the user to add tasks, mark tasks as completed, and view the list of tasks. The tasks will be stored in memory while the program is running, and they will be lost once the program is closed.

Specifications

- The program should provide a menu with options to add a task, mark a task as completed, and view the list of tasks.
- The user should be able to enter the details of a task (e.g., task name, due date) when adding a task.
- The program should store the tasks in a list or data structure.
- The program should display the list of tasks with their details.
- The user should be able to mark a task as completed, which will update its status in the list.
- The program should handle invalid inputs and provide appropriate error messages.

14.4 Project 4: Simple Calculator

Description

In this project, you will create a simple calculator program. The program will prompt the user to enter two numbers and an operation (+, -, *, /), and it will perform the corresponding calculation and display the result.

Specifications

- The program should prompt the user to enter the first number.
- The program should prompt the user to enter the second number.
- The program should prompt the user to enter the operation (+, -, *, /).
- The program should perform the corresponding calculation based on the entered numbers and operation.
- The program should display the result of the calculation.
- The program should handle invalid inputs and provide appropriate error messages.

14.5 Project 5: Hangman Game

Description

In this project, you will create a Hangman game. The program will select a random word from a predefined list, and the user will have to guess the letters of the word one by one. The user will have a limited number of attempts, and the program will provide feedback on the correctness of each guess.

Specifications

- The program should select a random word from a predefined list of words.
- The program should display the initial state of the word with underscores for the unknown letters.
- The program should prompt the user to enter a letter guess.
- The program should check the correctness of the guess and update the state of the word accordingly.
- The program should display the updated state of the word with the correctly guessed letters.
- The program should keep track of the number of attempts and limit the guesses to a certain number.
- The program should display a message indicating whether the user won or lost the game.

Chapter 15

Additional references

15.1 Manual Number Base Conversion

If you don't have access to a computer or programming language, you can still manually convert numbers between different bases using pen and paper. Here's a step-by-step guide to manually convert numbers between binary, decimal, and hexadecimal bases:

15.1.1 Binary to Decimal Conversion

To convert a binary number to decimal manually, follow these steps:

1. Write down the binary number.
2. Start from the rightmost digit and assign powers of 2 to each digit, starting from 0 for the rightmost digit.
3. Multiply each digit by its corresponding power of 2.
4. Sum up the results to obtain the decimal equivalent.

For example, let's convert the binary number 10101 to decimal:

Binary number:		10	10	1
Powers of 2:		$2^4 2^3$	$2^2 2^1$	2^0
Multiplication:		160	40	1
Sum:	$2^4 * 1 + 2^3 * 0 + 2^2 * 1 + 2^1 * 0 + 2^0 * 1 = 21$			

Therefore, the binary number 10101 is equivalent to the decimal number 21.

15.1.2 Decimal to Binary Conversion

To convert a decimal number to binary manually, follow these steps:

1. Write down the decimal number.
2. Divide the number by 2 and write down the quotient and remainder.

3. Repeat the division process with the quotient until the quotient becomes 0.

4. The binary representation is obtained by reading the remainders from the last division in reverse order.

For example, let's convert the decimal number 21 to binary:

Table 15.1: Decimal to Binary Conversion

Step	**Division**	**Result**
Step 1	21	
Step 2	$21 \div 2 = 10$	Quotient: 10, Remainder: 1
Step 3	$10 \div 2 = 5$	Quotient: 5, Remainder: 0
Step 4	$5 \div 2 = 2$	Quotient: 2, Remainder: 1
Step 5	$2 \div 2 = 1$	Quotient: 1, Remainder: 0
Step 6	$1 \div 2 = 0$	Quotient: 0, Remainder: 1

Binary representation (reading remainders in reverse order): 10101
Therefore, the decimal number 21 is equivalent to the binary number 10101.

15.1.3 Decimal to Hexadecimal Conversion

To convert a decimal number to hexadecimal manually, follow these steps:

1. Write down the decimal number.

2. Divide the number by 16 and write down the quotient and remainder.

3. Repeat the division process with the quotient until the quotient becomes 0.

4. The hexadecimal representation is obtained by reading the remainders from the last division in reverse order. Replace any remainders greater than 9 with the corresponding hexadecimal letters: A for 10, B for 11, C for 12, D for 13, E for 14, and F for 15.

For example, let's convert the decimal number 21 to hexadecimal:

Table 15.2: Decimal to Hexadecimal Conversion

Step	**Division**	**Result**
Step 1	21	
Step 2	$21 \div 16 = 1$	Quotient: 1, Remainder: 5
Step 3	$1 \div 16 = 0$	Quotient: 0, Remainder: 1

Hexadecimal representation (reading remainders in reverse order): 15
Therefore, the decimal number 21 is equivalent to the hexadecimal number 15.
By following these manual conversion steps, you can convert numbers between binary, decimal, and hexadecimal bases without the need for a computer or programming language.

15.2 A Deep Dive into the Quadratic Equation

In the discipline of algebra, a preeminent form of a polynomial equation is the second-order polynomial equation, more commonly known as the quadratic equation. A polynomial function of a single variable x is typically expressed in the following general form:

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n, \tag{15.1}$$

where the $f : \mathbb{R} \to \mathbb{R}$ or generally $f : \mathbb{C} \to \mathbb{C}$, $a_i,\ i \in \mathbb{N}, i \in \{0, 1, ..., n\}$ are constants and they are known as the coefficients of the polynomial. The power n is a nonnegative integer and is called the degree of the polynomial. The coefficient a_n of the highest power is called the leading coefficient.

The standard form of second order equation is typically expressed as $ax^2 + bx + c = 0$, where a, b, and c are constants that are defined such that $a \neq 0$. The coefficients a, b, and c are often referred to as the quadratic, linear, and constant terms, respectively.

The problem is: find the values of the f domain, such that f(x) = 0, f defined in (1). An essential concept in the examination of these quadratic equations is the quadratic formula, which is universally used to calculate the roots of the equation. The quadratic formula is stated as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{15.2}$$

An interesting feature of the quadratic formula is the term under the square root, $b^2 - 4ac$. This term is known as the discriminant. It plays a pivotal role in determining the nature of the solutions to the quadratic equation.

- If the discriminant is positive, the equation possesses two distinct real roots.
- If the discriminant equals zero, the equation has one real root, often referred to as a repeated or double root.
- If the discriminant is negative, the equation gives rise to two complex roots.

Consider, for instance, the quadratic equation $x^2 - 5x + 6 = 0$. The coefficients of this equation are $a = 1$, $b = -5$, and $c = 6$. By calculating the discriminant, we find $(-5)^2 - 4 * 1 * 6 = 25 - 24 = 1$. As this is a positive value, we infer that the equation has two distinct real roots. By substituting the coefficients into the quadratic formula, we find the solutions to be:

$$x = \frac{-(-5) \pm \sqrt{(-5)^2 - 4 * 1 * 6}}{2 * 1} = \frac{5 \pm 1}{2}$$

Therefore, the roots of the equation are $x = \frac{5+1}{2} = 3$ and $x = \frac{5-1}{2} = 2$. The study of quadratic equations and their solutions is a fundamental part of algebra, and it underlies many of the more advanced topics in mathematics.

15.3 Solving Second Order Equations

A second order equation, also known as a quadratic equation, is an equation of the form $ax^2 + bx + c = 0$, where a, b, and c are constants, and $a \neq 0$.

The solutions to a quadratic equation are given by the quadratic formula (2):

The term under the square root, $b^2 - 4ac$, is known as the discriminant. It determines the nature of the roots of the quadratic equation.

- If the discriminant is positive, the equation has two distinct real roots.
- If the discriminant is zero, the equation has exactly one real root (or a repeated real root).
- If the discriminant is negative, the equation has two complex roots.

For example, let's solve the equation $x^2 - 5x + 6 = 0$. Here $a = 1$, $b = -5$, and $c = 6$. The discriminant is $(-5)^2 - 4 * 1 * 6 = 25 - 24 = 1$, which is positive. Thus the equation has two distinct real roots. Applying the quadratic formula gives:

$$x = \frac{-(-5) \pm \sqrt{(-5)^2 - 4 * 1 * 6}}{2 * 1} = \frac{5 \pm 1}{2}$$

So the solutions are $x = \frac{5+1}{2} = 3$ and $x = \frac{5-1}{2} = 2$.

Recommended Literature

Bibliography

[1] Matthes, E. (2019). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. No Starch Press.

[2] Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.

[3] Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media.

[4] Lutz, M. (2013). *Learning Python*. O'Reilly Media.

[5] VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media.

[6] Slatkin, B. (2015). *Effective Python: 59 Specific Ways to Write Better Python*. Addison-Wesley Professional.

[7] Downey, A. B. (2012). *Think Python: How to Think Like a Computer Scientist*. Green Tea Press.

[8] Python Software Foundation. *Python Documentation*. Retrieved from `https://docs.python.org`

[9] Python Software Foundation. *Python Tutorial*. Retrieved from `https://docs.python.org/3/tutorial/index.html`

[10] Real Python. *Python Tutorials and Articles*. Retrieved from `https://realpython.com`