Python Programming Exercises

Chapter 1: Introduction to Python

1. Research and list five popular applications or systems developed using Python.
2. Compare Python with another programming language you're familiar with. List three similarities and three differences.
3. Explain the difference between an interpreted and a compiled language. Where does Python fit in this categorization?
4. Investigate the latest Python version. What are three new features introduced in this version?
5. Write a short essay (200-300 words) on why Python is popular in data science and machine learning.

Chapter 2: Setting Up Your Python Environment

1. Install Python on your system. Write a simple "Hello, World!" program and run it from the command line.
2. Create a virtual environment and activate it. Install a third-party package (e.g., requests) in this environment.
3. Write a Python script that prints the version of Python you're using and lists all installed packages.
4. Set up an Integrated Development Environment (IDE) for Python development (e.g., PyCharm, VS Code). Configure it to use the virtual environment you created.
5. Create a Python script that uses a feature from a recent Python version (e.g., f-strings from Python 3.6+). Run it in your IDE and from the command line.

Chapter 3: Variables and Data Types

1. Write a program that declares variables of each basic data type in Python (int, float, string, boolean). Print the type and value of each variable.
2. Create a list, tuple, and dictionary containing at least three elements each. Print these data structures and their types.
3. Write a program that demonstrates type conversion between different data types (e.g., string to int, float to int, int to string).
4. Create a program that uses complex numbers to perform basic arithmetic operations.
5. Write a script that demonstrates the difference between mutable and immutable data types in Python.

Chapter 4: Logging to Console and String Interpolation

1. Write a program that uses different print statements to output various data types (string, integer, float, boolean).
2. Create a program that demonstrates the use of escape characters in strings (e.g., newline, tab, quote).
3. Write a script that uses string formatting with the .format() method to print a sentence with at least three variables.
4. Create a program that uses f-strings to format a complex sentence including expressions and function calls.
5. Write a script that compares the performance of string concatenation, .format() method, and f-strings for a large number of operations. (Hint: Use the `timeit` module)

Chapter 5: User Input

1. Write a program that asks the user for their name and age, then prints a greeting message.
2. Create a simple calculator that takes two numbers and an operation (+, -, *, /) as input from the user, performs the operation, and prints the result.
3. Write a program that converts temperatures between Fahrenheit and Celsius based on user input.
4. Create a program that generates a simple quiz. Ask the user multiple-choice questions, keep track of their score, and display the final result.
5. Write a script that validates user input (e.g., checking if an entered value is a valid integer within a specific range).

Chapter 6: Operators and Control Flow Statements

1. Write a program that determines if a year entered by the user is a leap year.
2. Create a simple number guessing game where the computer generates a random number and the user tries to guess it.
3. Write a program that prints the first 20 numbers in the Fibonacci sequence using a loop and conditional statements.
4. Implement a simple calculator using if-elif-else statements to perform different operations based on user input.
5. Create a program that simulates a simple rock-paper-scissors game against the computer.

Chapter 7: Loops

1. Write a program that prints a multiplication table for numbers 1 through 10 using nested loops.
2. Create a program that finds and prints all prime numbers between 1 and 100 using the Sieve of Eratosthenes algorithm.
3. Implement a program that prints Pascal's triangle up to n rows, where n is entered by the user.
4. Write a program that simulates a simple ATM machine using a while loop for the main interaction and a nested if-elif-else for different operations.

5. Create a program that plays the "FizzBuzz" game for numbers from 1 to 100. (Print "Fizz" for multiples of 3, "Buzz" for multiples of 5, and "FizzBuzz" for multiples of both.)

Chapter 8: Lists and Tuples

1. Write a program that creates a list of numbers and performs various operations (append, insert, remove, pop, index, count, sort, reverse).
2. Implement a function that takes two lists and returns a new list containing only the common elements (without using sets).
3. Create a program that simulates a simple to-do list using a list. Allow the user to add, remove, and view tasks.
4. Write a function that takes a list of numbers and returns a tuple containing the minimum, maximum, and average values.
5. Implement a simple deck of cards using a list of tuples, where each tuple represents a card (suit, value). Include functions to shuffle the deck and deal cards.

Chapter 9: Functions

1. Write a function that calculates the factorial of a number using recursion. Compare its performance with an iterative version.
2. Implement a function that takes a variable number of arguments and returns their sum.
3. Create a function that generates a Fibonacci sequence up to n terms using a generator.
4. Write a decorator function that measures and prints the execution time of any function it decorates.
5. Implement a higher-order function that takes a list of numbers and a function as arguments, applying the function to each element in the list and returning the result.

Chapter 10: Dictionaries and Sets

1. Create a program that simulates a simple phonebook using a dictionary. Allow users to add, delete, and look up entries.
2. Write a function that takes a string and returns a dictionary with each unique character as a key and its frequency as the value.
3. Implement a program that uses sets to find the unique words in a given text file.

4. Create a simple cache mechanism using a dictionary, where computed values are stored and retrieved to avoid redundant calculations.
5. Write a program that simulates a basic inventory system for a store using dictionaries and sets. Include functions to add items, update quantities, and check for item availability.

Chapter 11: Classes and Objects

1. Design a `BankAccount` class with methods for deposit, withdrawal, and checking balance. Include appropriate error handling.
2. Create a `Shape` class hierarchy with a base `Shape` class and derived classes for different shapes (e.g., `Circle`, `Rectangle`, `Triangle`). Include methods to calculate area and perimeter for each shape.
3. Implement a simple `Library` class that manages a collection of `Book` objects. Include methods to add books, remove books, and search for books by title or author.
4. Design a `Vehicle` class hierarchy with a base `Vehicle` class and derived classes for different types of vehicles (e.g., `Car`, `Motorcycle`, `Truck`). Include appropriate attributes and methods for each class.
5. Create a simple game using classes, where you have different character types (e.g., `Warrior`, `Mage`, `Archer`) that can battle each other. Include methods for attacking, taking damage, and checking if a character is defeated.

Chapter 12: String Methods

1. Write a program that takes a sentence as input and capitalizes the first letter of each word.
2. Create a function that checks if a given string is a palindrome, ignoring spaces, punctuation, and letter casing.
3. Implement a simple text-based search engine that finds occurrences of words in a given text, accounting for partial matches and case insensitivity.
4. Write a program that extracts all email addresses from a given text using regular expressions.
5. Create a function that takes a string and returns the most frequent character(s) in it. If there are multiple characters with the same highest frequency, return all of them.

Chapter 13: Error Handling, Data Validation, and Type Conversion

1. Write a function that performs division and uses a try-except block to handle potential ZeroDivisionError and TypeError exceptions.

2. Create a program that reads a CSV file containing numeric data. Use exception handling to skip rows with invalid data and report the line numbers of skipped rows.
3. Implement a function that validates an email address using regular expressions. Raise a custom exception if the email is invalid.
4. Write a program that prompts the user to enter a date in the format MM/DD/YYYY. Validate the input and convert it to a datetime object. Handle potential ValueError exceptions.
5. Create a decorator that can be used to validate function arguments. For example, it should check if arguments are of the correct type and within a specified range.

Chapter 14: Modules and Packages

1. Create a simple module with functions for basic mathematical operations (add, subtract, multiply, divide). Import and use this module in another script.
2. Write a script that uses the `os` and `sys` modules to print information about the current operating system and Python environment.
3. Create a package with modules for different geometric shapes (e.g., circle, rectangle, triangle). Each module should contain functions to calculate area and perimeter. Use this package in a main script.
4. Write a program that uses the `requests` module to fetch data from a public API and process the response.
5. Create a module that implements a simple logging system. Use this module in another script to log messages at different severity levels.

Chapter 15: File I/O

1. Write a program that reads a text file, counts the occurrences of each word, and writes the results to a new file in descending order of frequency.
2. Create a script that merges multiple CSV files into a single CSV file. Allow the user to specify which columns to include from each file.
3. Implement a program that reads a JSON file containing nested data structures, modifies some values, and writes the updated data back to a new JSON file.
4. Write a script that creates a backup of a specified directory, copying all files and subdirectories to a new location. Use the `os` and `shutil` modules.
5. Create a program that simulates a simple database using file I/O. Implement functions to add records, search for records, update records, and delete records. Store the data in a CSV or JSON format.

Chapter 16: Basic Data Structures and Algorithms

1. Implement a Stack class using a Python list. Include methods for push, pop, peek, and is_empty.
2. Create a Queue class using two stacks. Implement enqueue and dequeue operations.
3. Write a function that checks if a given string has balanced parentheses using a stack.
4. Implement the binary search algorithm for a sorted list of integers. Compare its performance with the linear search for large lists.
5. Create a program that implements the merge sort algorithm to sort a list of integers. Compare its performance with Python's built-in sort() method for large lists.

Chapter 17: Multithreading and Multiprocessing

1. Write a program that downloads multiple files concurrently using threads. Compare its performance with a sequential download.
2. Implement a producer-consumer problem using threading and a queue. The producer should generate random numbers, and the consumer should calculate their factorial.
3. Create a program that uses multiprocessing to calculate the sum of large lists of numbers. Compare its performance with a single-process version.
4. Write a script that uses threading to simulate a simple chat server and multiple clients.
5. Implement a parallel version of the merge sort algorithm using multiprocessing. Compare its performance with the single-process version for large lists.

Chapter 18: Asynchronous Programming

1. Write an asynchronous program that fetches data from multiple APIs concurrently using `aiohttp`.
2. Create an asynchronous web scraper that extracts information from multiple web pages concurrently.
3. Implement an asynchronous producer-consumer pattern using `asyncio.Queue`.
4. Write a program that uses asynchronous file I/O to read and process multiple large files concurrently.
5. Create an asynchronous chat server and client using `asyncio` and websockets.

Chapter 19: Network Programming

1. Implement a simple HTTP server using the `http.server` module. Serve static files and handle basic GET and POST requests.
2. Create a UDP-based chat application that allows multiple clients to communicate.
3. Write a program that uses sockets to implement a basic client-server application. The server should perform a specific task (e.g., mathematical operations) based on client requests.
4. Implement a simple port scanner that checks for open ports on a given IP address or hostname.
5. Create a program that downloads a webpage, extracts all the links, and then downloads the pages those links point to (up to a specified depth).

These exercises cover a wide range of Python programming topics and progressively increase in difficulty. They should provide comprehensive practice for learners working through the Python Programming Guide.