
CODE WITHOUT BARS:
ADVANCED PYTHON ADVENTURES
Breaking Cycles, Building Futures with Python
By Encrypter15

Copyright © 2025 by Encrypter15

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

For permission requests, contact the publisher at:
Encrypter15@gmail.com

**First Edition, 2025**

**Published by**:
Encrypter15 Press

**Printed in the United States of America**

**Disclaimer**:
This book is designed for educational purposes and to inspire personal transformation through coding. The author and publisher are not responsible for any misuse of the information provided. The examples and exercises are intended for paper-based learning and may require adaptation for use on a computer.

**Dedication**:
To all aspiring coders breaking barriers and building futures—your journey proves that “what one man can do, another can do.”

**Acknowledgments**:
Special thanks to programs like The Last Mile, whose work in reducing recidivism through coding education inspired this book, and to the trailblazers like Edmund Hillary and Roger Bannister, whose stories remind us to push beyond limits.

**Cover Design**: Encrypter15

**Note on Accessibility**:
This book is designed for learners in any environment, including those without access to computers. All exercises are paper-based to ensure inclusivity and empowerment for all readers.

# Chapter 1: Welcome to Advanced Python

Welcome to *Code Without Bars: Advanced Python Adventures*! You've already mastered the basics of Python—variables, loops, functions, lists, dictionaries, and more—through your journey in *Code Without Bars: Learning Python from the Ground Up*. Now, it's time to take your skills to the next level. This book will guide you through advanced Python concepts like object-oriented programming, complex data structures, file handling, and APIs, all while keeping the focus on practical, paper-based learning. No computer is needed—just your brain, a pen, and some paper. In this chapter, we'll review your foundational skills, set the stage for advanced coding, and introduce tools like pseudocode and flowcharts to plan complex programs. Let's dive into your next adventure and explore how coding can transform your life and inspire others!

---

## Reflecting on Your Python Foundation

Before we leap into advanced topics, let's take a moment to celebrate what you've already learned. You can:
- **Store Data**: Use variables for numbers and strings, lists for collections, and dictionaries for key-value pairs.
- **Control Flow**: Write loops (`for` and `while`) to repeat tasks and conditions (`if`/`elif`/`else`) to make decisions.
- **Organize Code**: Create functions to reuse code and handle user input/output for interactivity.
- **Build Projects**: Design programs like a calculator, quiz game, journal app, and to-do list.
- **Debug Errors**: Spot and fix syntax, runtime, and logic errors, making your programs robust.

These skills are the bedrock of coding. Advanced Python builds on them, enabling you to create more efficient, modular, and real-world programs—like apps that track habits, analyze data, or even power games. As you learn, you're not just coding; you're building a path to opportunity, reducing recidivism, and showing the next generation that a "change gang" mindset—focused on growth and contribution—beats a "chain gang" label every time.

---

## What Makes a Program "Advanced"?

An advanced program does more with less: it's efficient, reusable, and solves complex problems. Here's what sets advanced Python apart:
- **Modularity**: Using classes and functions to organize code into reusable pieces, like building blocks.

- **Complex Data**: Handling intricate data with sets, tuples, or nested dictionaries, like organizing a whole library instead of a single book.
- **Real-World Applications**: Connecting to files, APIs, or databases to save data or fetch information, like a weather app pulling live data.
- **Robustness**: Anticipating and handling errors gracefully, ensuring your program doesn't crash.

For example, your to-do list app from the first book used a dictionary and a loop to manage tasks. An advanced version might use a *class* to represent tasks, save them to a file, and validate inputs to prevent errors—making it more like a professional app.

---

## Planning Complex Programs

Advanced programs require careful planning to avoid getting lost in the code. Two tools will help: **pseudocode** and **flowcharts**.

- **Pseudocode**: A plain-English description of what your program does, like a recipe. It focuses on logic, not syntax. For example:

```
Program: Track study hours
Create a dictionary to store subjects and hours
Show a menu: add hours, view totals, quit
If user chooses add:
  Get subject and hours
  Update dictionary
If user chooses view:
  Show all subjects and hours
If user chooses quit:
  Stop
```

- **Flowcharts**: Diagrams that show the program's flow using shapes (ovals for start/end, rectangles for actions, diamonds for decisions). For the study tracker:
  - Oval: Start
  - Rectangle: Show menu
  - Diamond: User chooses option?
  - Rectangle (if add): Get subject/hours, update dictionary
  - Rectangle (if view): Show dictionary
  - Oval (if quit): End

These tools help you think through the program before writing Python, catching logic errors early.

Let's practice planning.

**Exercise: Write Pseudocode**
On paper, write pseudocode for a program that tracks exercise goals (e.g., push-ups per day).
Example:
- Pseudocode:

```
Program: Exercise Goal Tracker
Create a dictionary for days and push-up counts
Loop forever:
  Show menu: add push-ups, view goals, quit
  If add:
    Ask for day and number of push-ups
    Add to dictionary
    Say "Saved!"
  If view:
    Show all days and push-up counts
  If quit:
    Stop
```

Write pseudocode for your own program, like a habit tracker or budget planner. Include at least three actions (e.g., add, view, quit). Check: Is the logic clear? Does it cover all steps? Refine if needed.

---

## Coding for Transformation

Learning advanced Python isn't just about writing better code—it's about rewriting your future. Programs like The Last Mile show that coding reduces recidivism by up to 95% for participants, offering skills for jobs that pay $60,000+ annually. By mastering Python, you're not just learning syntax; you're building a bridge to opportunity. More than that, you're setting an example for your children and their children, showing that a "change gang" coder can break the cycle of incarceration. As Edmund Hillary proved by summiting Everest and Roger Bannister showed by breaking the four-minute mile, "what one man can do, another can do." Your coding journey is proof that transformation is possible.

---

## Why This Chapter Matters

This chapter sets the foundation for advanced Python by connecting your existing skills to new possibilities. You'll learn to plan complex programs, think modularly, and see coding as a tool for

personal and generational change. In the coming chapters, you'll dive into object-oriented programming, advanced data structures, and real-world projects—all while practicing on paper. You're not just coding; you're coding your way to a new life.

**Reflection Prompt**
How can advanced Python skills help you achieve your goals, like finding a job or inspiring others? What's one program you're excited to build in this book?

**Paper Coding Exercise**
On paper, plan a program using pseudocode and a rough flowchart. Example:
- Program: Budget Tracker
- Pseudocode:

```
Create dictionary for categories (e.g., food, rent) and amounts
Loop:
  Show menu: add expense, view budget, quit
  If add:
    Ask for category and amount
    Update dictionary
  If view:
    Show total and each category
  If quit:
    Stop
```

- Flowchart: Draw an oval (start), rectangle (menu), diamond (choice), and paths for add/view/quit.

Write your own pseudocode and sketch a flowchart for a program (e.g., a goal tracker or book log). Test the logic in your head and fix any gaps. Save this plan—it's the start of your advanced coding adventure!

# Chapter 2: Object-Oriented Programming (OOP) Basics

Welcome to the next step in your Python journey! In **Object-Oriented Programming (OOP)**, a way to structure code that models real-world objects, like a car or a person, making your programs more modular and reusable. You'll learn how to create **classes** and **objects**, define attributes and methods, and apply them to practical examples. This is a big leap toward professional coding, and it's all doable on paper—no computer needed, just your brain, a pen, and some paper. Let's get started and see how OOP can transform your coding and your future!*Code Without Bars: Advanced Python Adventures*, you're building on your foundational skills to create more powerful and organized programs. In this chapter, we'll dive into

---

## What Are Classes and Objects?

In OOP, a **class** is like a blueprint for an object, defining its properties (data) and behaviors (actions). An **object** is an instance of a class, like a specific house built from a blueprint. For example, imagine a "Car" class:
- **Properties** (attributes): color, speed, model.
- **Behaviors** (methods): drive, stop, honk.

Here's how you'd write a simple class in Python:

```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model

    def drive(self):
        return f"The {self.color} {self.model} is driving!"
```

- `class Car`: Defines the class named "Car."
- `__init__(self, color, model)`: The special method to initialize objects, setting attributes like `color` and `model`. The `self` parameter refers to the object being created.
- `def drive(self)`: A method that describes an action, using `self` to access the object's attributes.

To create objects and use them:

```
my_car = Car("red", "Toyota")  # Create an object
print(my_car.color)  # Prints: red
print(my_car.drive())  # Prints: The red Toyota is driving!
```

Objects let you create multiple instances with different data:

```
your_car = Car("blue", "Honda")
print(your_car.drive())  # Prints: The blue Honda is driving!
```

OOP makes code reusable and organized, like having a template for all cars instead of writing separate code for each one.

Let's practice creating a class.

**Exercise: Create a Simple Class**
Write a Python program on paper for a class representing a **Book**. Example:
- Code:

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def read(self):
        return f"Reading {self.title} with {self.pages} pages."

my_book = Book("Python Guide", 200)
print(my_book.read())
```

- Result: Shows `Reading Python Guide with 200 pages.`

Write your own class for something simple, like a `Pet` with attributes (name, type) and a method (e.g., `play`). Create one object and call its method. Check: Did you use `__init__` and `self` correctly? Is the method indented? Fix any errors.

---

## Adding More Methods

Classes can have multiple methods to define different behaviors. For example, let's expand the `Car` class:

```
class Car:
    def __init__(self, color, model, speed):
        self.color = color
        self.model = model
```

```
        self.speed = speed

    def drive(self):
        return f"The {self.color} {self.model} is driving at {self.speed} mph."

    def stop(self):
        return f"The {self.color} {self.model} has stopped."
```

You can use it like this:

```
car = Car("green", "Ford", 60)
print(car.drive())  # Prints: The green Ford is driving at 60 mph.
print(car.stop())   # Prints: The green Ford has stopped.
```

Methods can also modify attributes. For example:

```
class Car:
    def __init__(self, color, model, speed):
        self.color = color
        self.model = model
        self.speed = speed

    def accelerate(self, increase):
        self.speed = self.speed + increase
        return f"{self.model} now at {self.speed} mph!"
```

Using it:

```
car = Car("black", "Tesla", 50)
print(car.accelerate(20))  # Prints: Tesla now at 70 mph!
```

Let's practice adding methods.

**Exercise: Class with Multiple Methods**
Write a Python program on paper for a class with at least two methods. Example:
- Code:

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

      def study(self):
```

```
            return f"{self.name} is studying."

        def improve_grade(self, points):
            self.grade = self.grade + points
            return f"{self.name}'s grade is now {self.grade}."

    student = Student("Alex", 80)
    print(student.study())
    print(student.improve_grade(5))
```

- Result: Shows:

```
    Alex is studying.
    Alex's grade is now 85.
```

Write your own class (e.g., `Goal` with name and progress) with two methods, one that modifies an attribute. Create an object and call both methods. Check: Do the methods use `self`? Does the attribute change correctly? Fix any errors.

---

## Using Classes in a Program

Classes shine when you combine them with loops, conditions, or user input. Let's create a small program that manages multiple objects. Here's an example for a **Goal Tracker** using a `Goal` class:

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

    def update_progress(self, amount):
        self.progress = self.progress + amount
        return f"{self.name} progress: {self.progress}%"

    def is_complete(self):
        if self.progress >= 100:
            return f"{self.name} is complete!"
        return f"{self.name} is {self.progress}% done."

goals = []
while True:
    print("1. Add goal")
    print("2. Update progress")
```

```
    print("3. View goals")
    print("4. Quit")
    choice = input("Choose 1-4: ")

    if choice == "1":
        name = input("Goal name: ")
        goals.append(Goal(name, 0))
        print(f"Added {name}")
    elif choice == "2":
        name = input("Goal name to update: ")
        for goal in goals:
            if goal.name == name:
                amount = int(input("Progress to add: "))
                print(goal.update_progress(amount))
                break
        else:
            print("Goal not found!")
    elif choice == "3":
        if len(goals) == 0:
            print("No goals yet.")
        else:
            for i, goal in enumerate(goals, 1):
                print(f"{i}. {goal.is_complete()}")
    elif choice == "4":
        print("Keep pushing forward!")
        break
    else:
        print("Choose 1-4!")
```

This program:
- Defines a `Goal` class with attributes (`name`, `progress`) and methods (`update_progress`, `is_complete`).
- Uses a list to store `Goal` objects.
- Runs a menu loop to add goals, update progress, view statuses, or quit.
- Checks for errors like missing goals.

Let's test it in your head:
- Choose `1`, enter "Learn Python" → "Added Learn Python"
- Choose `2`, enter "Learn Python", add 50 → "Learn Python progress: 50%"
- Choose `3` → Shows "1. Learn Python is 50% done."
- Choose `4` → "Keep pushing forward!"

Let's practice building a class-based program.

**Exercise: Class-Based Program**
Write a Python program on paper that uses a class and a menu loop. Example:

- Code:

```
class Task:
    def __init__(self, name):
        self.name = name
        self.done = False

    def mark_done(self):
        self.done = True
        return f"{self.name} is done!"

    def status(self):
        return f"{self.name}: {'Done' if self.done else 'Pending'}"

tasks = []
while True:
    print("1. Add task  2. Mark done  3. View tasks")
    choice = input("Choose 1-3: ")
    if choice == "1":
        name = input("Task: ")
        tasks.append(Task(name))
        print("Task added!")
    elif choice == "2":
        name = input("Task to mark done: ")
        for task in tasks:
            if task.name == name:
                print(task.mark_done())
                break
        else:
            print("Task not found!")
    elif choice == "3":
        for i, task in enumerate(tasks, 1):
            print(f"{i}. {task.status()}")
    else:
        print("Choose 1-3!")
```

- Result: Adds tasks, marks them done, and shows statuses.

Write your own program with a class (e.g., `Pet` with name and energy) and a menu to add objects, update them, and view them. Check: Does the class work? Is the loop correct? Fix any errors.

---

## Why OOP Matters

Object-Oriented Programming makes your code:
- **Reusable**: Classes let you create templates for similar objects.
- **Organized**: Group data and actions together, like a toolbox.
- **Scalable**: Build complex programs, like games or trackers, with clear structure.

OOP also mirrors real-world transformation. Just as you're breaking the "chain gang" label to become a "change gang" coder, classes let you model new possibilities—like a `Student` class for your learning journey or a `Future` class for your goals. As Edmund Hillary and Roger Bannister showed, one person's breakthrough inspires others. Your coding can inspire your community, showing that "what one man can do, another can do."

In the next chapter, we'll explore **inheritance** and **encapsulation** to make classes even more powerful. For now, keep practicing OOP on paper—you're building the foundation for advanced Python programs!

---

**Reflection Prompt**
How could a class model something in your life, like a goal or a skill? How does learning OOP make you feel about your coding journey?

**Paper Coding Exercise**
Write a Python program on paper that:
- Defines a class with at least two attributes and two methods.
- Creates at least two objects.
- Uses a loop or menu to interact with objects (e.g., update or view).
Example: A `Goal` class with name, progress, and methods to update and check completion. Run it in your head with sample inputs and fix any errors.

# Chapter 3: Advanced OOP: Inheritance and Encapsulation

You've taken your first steps into Object-Oriented Programming (OOP), creating classes and objects to model real-world things like pets or goals. Now, let's level up with **inheritance** and **encapsulation**, two powerful OOP concepts that make your code more flexible and secure. Inheritance lets you build new classes based on existing ones, like a family tree of code. Encapsulation protects your data by controlling how it's accessed or changed, like locking a safe. These tools will help you write cleaner, more professional programs while reinforcing your journey from "chain gang" to "change gang." You can practice on paper—no computer needed, just your brain, a pen, and some paper. Let's dive in and make your Python skills even stronger!

---

## Understanding Inheritance

**Inheritance** allows a new class (called a **subclass** or **child class**) to inherit attributes and methods from an existing class (called a **parent class**). This saves you from rewriting code and lets you extend or specialize behavior. For example, a `Dog` class can inherit from an `Animal` class, adding dog-specific features while keeping shared ones.

Here's a simple example:

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

    def eat(self):
        return f"{self.name} is eating."

class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        return f"{self.name} says woof!"

my_dog = Dog("Buddy")
print(my_dog.eat())  # Prints: Buddy is eating.
print(my_dog.bark())  # Prints: Buddy says woof!
```

- `Dog(Animal)`: Means `Dog` inherits from `Animal`, getting its `__init__` and `eat` method.
- `Dog` adds its own method (`bark`) for dog-specific behavior.
- Objects of `Dog` can use both `Animal` and `Dog` methods.

You can also override parent methods in the subclass:

```
class Cat(Animal):
    def eat(self):  # Override the parent's eat method
        return f"{self.name} is eating fish."

my_cat = Cat("Whiskers")
print(my_cat.eat())  # Prints: Whiskers is eating fish.
```

Let's practice inheritance.

**Exercise: Create a Subclass**
Write a Python program on paper for a parent class and a subclass. Example:
- Code:

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def move(self):
        return f"{self.brand} is moving."

class Bike(Vehicle):
    def pedal(self):
        return f"{self.brand} bike is pedaling fast!"

my_bike = Bike("Schwinn")
print(my_bike.move())
print(my_bike.pedal())
```

- Result: Shows:

```
Schwinn is moving.
Schwinn bike is pedaling fast!
```

Write your own program with a parent class (e.g., `Person`) and a subclass (e.g., `Student`) with at least one new method. Create an object and call both parent and subclass methods. Check: Does the subclass inherit correctly? Are methods indented? Fix any errors.

---

## Exploring Encapsulation

**Encapsulation** protects a class's data by making attributes private, so they can only be accessed or changed through specific methods. In Python, private attributes are marked with an

underscore (e.g., `_balance`), signaling they shouldn't be accessed directly. You use **getter** and **setter** methods to control access.

Here's an example:

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance  # Private attribute

    def get_balance(self):  # Getter
        return f"{self.owner}'s balance: ${self._balance}"

    def deposit(self, amount):  # Setter
        if amount > 0:
            self._balance += amount
            return f"Deposited ${amount}. New balance: ${self._balance}"
        return "Invalid deposit amount."
```

Using it:

```
account = BankAccount("Alex", 100)
print(account.get_balance())  # Prints: Alex's balance: $100
print(account.deposit(50))    # Prints: Deposited $50. New balance: $150
# print(account._balance)     # Bad practice! Use get_balance instead.
```

- `_balance` is private, so you should use `get_balance` or `deposit` to access or change it.
- This protects the data (e.g., preventing negative deposits).

Let's practice encapsulation.

**Exercise: Encapsulated Class**
Write a Python program on paper for a class with a private attribute and getter/setter methods.
Example:
- Code:

```
  class Goal:
      def __init__(self, name, progress):
          self.name = name
          self._progress = progress

      def get_progress(self):
          return f"{self.name} progress: {self._progress}%"

      def add_progress(self, amount):
```

```
            if amount >= 0:
                self._progress += amount
                return f"Added {amount}%. Now at {self._progress}%"
            return "Invalid progress!"

    goal = Goal("Study Python", 20)
    print(goal.get_progress())
    print(goal.add_progress(10))
```

- Result: Shows:

```
    Study Python progress: 20%
    Added 10%. Now at 30%
```

Write your own class (e.g., `Task` with name and private priority) with a getter and setter. Create an object and use the methods. Check: Is the attribute private with `_`? Do the methods control access? Fix any errors.

---

## Combining Inheritance and Encapsulation

Let’s combine these concepts in a program that tracks fitness goals. We’ll have a parent class `Exercise` and a subclass `Run`, with private attributes for safety.

```
class Exercise:
    def __init__(self, name):
        self.name = name
        self._calories = 0  # Private

    def get_calories(self):
        return f"{self.name} burned {self._calories} calories."

    def add_calories(self, amount):
        if amount > 0:
            self._calories += amount
            return f"Added {amount} calories."
        return "Invalid amount."

class Run(Exercise):
    def __init__(self, name, distance):
        super().__init__(name)  # Call parent's __init__
        self.distance = distance

    def track_run(self):
```

```
        calories = self.distance * 100  # Simple calorie estimate
        return self.add_calories(calories)

runner = Run("Morning Run", 2)
print(runner.track_run())       # Prints: Added 200 calories.
print(runner.get_calories())    # Prints: Morning Run burned 200 calories.
```

- `super().__init__(name)`: Calls the parent's `__init__` to set `name` and `_calories`.
- `_calories` is private, accessed only through `get_calories` and `add_calories`.
- `Run` inherits from `Exercise` and adds a `distance` attribute and `track_run` method.

Let's practice combining these.

**Exercise: Inheritance and Encapsulation Program**
Write a Python program on paper with a parent class, a subclass, and encapsulation. Example:
- Code:

```
class Item:
    def __init__(self, name):
        self.name = name
        self._weight = 0

    def get_weight(self):
        return f"{self.name} weighs {self._weight} lbs."

    def set_weight(self, weight):
        if weight >= 0:
            self._weight = weight
            return f"Set {self.name} to {weight} lbs."
        return "Invalid weight!"

class Book(Item):
    def __init__(self, name, pages):
        super().__init__(name)
        self.pages = pages

    def read(self):
        return f"Reading {self.name} with {self.pages} pages."

book = Book("Python Guide", 300)
print(book.set_weight(2))
print(book.get_weight())
print(book.read())
```

- Result: Shows:

```
Set Python Guide to 2 lbs.
```

```
Python Guide weighs 2 lbs.
Reading Python Guide with 300 pages.
```

Write your own program with a parent class (e.g., `Goal`), a subclass (e.g., `StudyGoal`), and a private attribute with getter/setter methods. Create an object and use its methods. Check: Does `super()` work? Are private attributes protected? Fix any errors.

---

## Why Inheritance and Encapsulation Matter

- **Inheritance**: Saves time by reusing code and creates hierarchies, like a family of related classes. It's like inheriting resilience from trailblazers like Edmund Hillary, who summited Everest, or Roger Bannister, who broke the four-minute mile—showing that "what one man can do, another can do."
- **Encapsulation**: Keeps your data safe and your code predictable, like locking away old habits to protect your new "change gang" mindset.
- **Real-World Impact**: These concepts power complex apps, like task managers or games, and mirror your journey of building a secure, structured future.

By mastering OOP, you're not just coding—you're crafting a path to opportunity, reducing recidivism, and inspiring the next generation to see themselves as coders, not convicts.

In the next chapter, we'll explore **sets** and **tuples** to handle specialized data, adding more tools to your Python toolbox. Keep practicing on paper—you're building skills for a transformed future!

---

**Reflection Prompt**
How could inheritance or encapsulation model something in your life, like goals or skills? How does OOP make you feel about breaking cycles and creating change?

**Paper Coding Exercise**
Write a Python program on paper that:
- Defines a parent class with a private attribute and getter/setter methods.
- Creates a subclass that inherits and adds a new method.
- Creates an object and calls methods from both classes.
Example: A `Task` parent class with private priority and a `DailyTask` subclass with a time attribute. Run it in your head with sample inputs and fix any errors.

# Chapter 4: Advanced Data Structures: Sets and Tuples

You've mastered classes, inheritance, and encapsulation, making your Python programs more modular and robust. Now, let's expand your data-handling skills with two new data structures: **sets** and **tuples**. These complement lists and dictionaries, offering unique ways to store and process information. Sets are great for handling unique items, like removing duplicates, while tuples are perfect for fixed, unchangeable data. In this chapter, you'll learn how to use sets and tuples, combine them with loops and conditions, and apply them to practical programs—all on paper, with just your brain, a pen, and some paper. These tools will help you build more efficient programs and reinforce your journey as a "change gang" coder, breaking free from cycles and inspiring others. Let's dive in!

---

## Understanding Sets

A **set** in Python is a collection of **unique** items, stored in curly braces `{}` or created with `set()`. Unlike lists, sets don't allow duplicates and don't maintain order, making them ideal for tasks like finding unique values or comparing collections. For example:

```
unique_numbers = {1, 2, 2, 3}  # Duplicates are removed
print(unique_numbers)  # Prints: {1, 2, 3}
```

You can create a set from a list to remove duplicates:

```
numbers = [1, 1, 2, 3, 3]
unique = set(numbers)
print(unique)  # Prints: {1, 2, 3}
```

Sets support operations like:
- **Union (`|`)**: Combines items from two sets.
- **Intersection (`&`)**: Finds items in both sets.
- **Difference (`-`)**: Finds items in one set but not another.

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}
print(set1 | set2)  # Prints: {1, 2, 3, 4} (union)
print(set1 & set2)  # Prints: {2, 3} (intersection)
print(set1 - set2)  # Prints: {1} (difference)
```

Let’s practice sets.

**Exercise: Create a Set Program**
Write a Python program on paper that uses a set to remove duplicates. Example:
- Code:

```
  # Remove duplicate tasks
  tasks = ["study", "exercise", "study", "read"]
  unique_tasks = set(tasks)
  print("Unique tasks:", unique_tasks)
```

- Result: Shows `Unique tasks: {'study', 'exercise', 'read'}`

Write your own program that creates a list with duplicates (e.g., goals or items) and converts it to a set to show unique items. Check: Does the set remove duplicates? Is the syntax correct? Fix any errors.

---

## Set Operations in Action

Sets are powerful for comparing or combining data. Here’s an example that checks common tasks between two days:

```
day1_tasks = {"study", "exercise", "read"}
day2_tasks = {"exercise", "code", "read"}
common_tasks = day1_tasks & day2_tasks
all_tasks = day1_tasks | day2_tasks
print("Common tasks:", common_tasks)  # Prints: {'exercise', 'read'}
print("All tasks:", all_tasks)  # Prints: {'study', 'exercise', 'read', 'code'}
```

You can also add or remove items:

```
tasks = {"study", "exercise"}
tasks.add("code")  # Add a single item
tasks.remove("study")  # Remove an item (raises error if not found)
print(tasks)  # Prints: {'exercise', 'code'}
```

Let’s practice set operations.

**Exercise: Set Operations Program**
Write a Python program on paper that uses set operations. Example:

- Code:

```
  # Compare goal lists
  week1_goals = {"learn Python", "exercise", "read"}
  week2_goals = {"exercise", "code app", "read"}
  shared = week1_goals & week2_goals
  print("Shared goals:", shared)
```

- Result: Shows `Shared goals: {'exercise', 'read'}`

Write your own program with two sets (e.g., skills or habits for two weeks). Use at least one operation (`|`, `&`, or `-`) to compare them. Check: Does the operation work as expected? Are sets defined correctly? Fix any errors.

---

## Understanding Tuples

A **tuple** is an immutable sequence, meaning it can't be changed after creation. Tuples are defined with parentheses `()` or created with `tuple()`. They're great for fixed data, like coordinates or settings. For example:

```
point = (10, 20)  # A tuple for x, y coordinates
print(point[0])  # Prints: 10 (access first item)
```

Tuples can hold mixed data types:

```
settings = ("dark mode", 12, True)  # Theme, font size, notifications on
print(settings)  # Prints: ('dark mode', 12, True)
```

You can't modify tuples (e.g., `point[0] = 5` raises an error), but you can loop through them or convert them to lists:

```
numbers = (1, 2, 3)
for num in numbers:
    print(num)  # Prints: 1, 2, 3
as_list = list(numbers)  # Convert to list: [1, 2, 3]
```

Let's practice tuples.

**Exercise: Create a Tuple Program**

Write a Python program on paper that uses a tuple. Example:
- Code:

```
# Store fixed schedule
schedule = ("study", "exercise", "rest")
print("Today's plan:", schedule)
for task in schedule:
    print("Do:", task)
```

- Result: Shows:

```
Today's plan: ('study', 'exercise', 'rest')
Do: study
Do: exercise
Do: rest
```

Write your own program with a tuple (e.g., fixed goals or coordinates). Loop through it or print one item. Check: Is the tuple immutable? Is the syntax correct? Fix any errors.

---

## Combining Sets and Tuples in a Program

Let's build a program that uses sets and tuples to track unique tasks and a fixed schedule. This could model your journey toward change, like tracking skills while sticking to a consistent plan.

```
tasks_done = set()  # Track unique tasks
daily_schedule = ("study", "exercise", "code")  # Fixed schedule

while True:
    print("1. Add completed task")
    print("2. View tasks and schedule")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    if choice == "1":
        task = input("Enter task completed: ")
        tasks_done.add(task)
        print(f"Added {task} to completed tasks!")
    elif choice == "2":
        print("Daily schedule:", daily_schedule)
        if len(tasks_done) == 0:
            print("No tasks completed yet.")
        else:
            print("Completed tasks:", tasks_done)
```

```
            common = tasks_done & set(daily_schedule)
            print("Tasks in schedule:", common)
    elif choice == "3":
        print("Keep building your future!")
        break
    else:
        print("Choose 1, 2, or 3!")
```

This program:
- Uses a **set** (`tasks_done`) to store unique completed tasks.
- Uses a **tuple** (`daily_schedule`) for a fixed daily plan.
- Allows adding tasks, viewing them, and checking which completed tasks match the schedule.
- Uses set intersection (`&`) to find common tasks.

Let's test it in your head:
- Choose `1`, enter "study" → "Added study to completed tasks!"
- Choose `1`, enter "code" → "Added code to completed tasks!"
- Choose `2` → Shows:

```
Daily schedule: ('study', 'exercise', 'code')
Completed tasks: {'study', 'code'}
Tasks in schedule: {'study', 'code'}
```

- Choose `3` → "Keep building your future!"

Let's practice combining sets and tuples.

**Exercise: Sets and Tuples Program**
Write a Python program on paper that uses a set and a tuple. Example:
- Code:

```
# Track habits
habits = set()
fixed_goals = ("learn code", "exercise", "read")
while True:
    print("1. Add habit  2. View habits  3. Quit")
    choice = input("Choose 1-3: ")
    if choice == "1":
        habit = input("Habit done: ")
        habits.add(habit)
        print("Habit added!")
    elif choice == "2":
        print("Fixed goals:", fixed_goals)
        print("Habits done:", habits)
    elif choice == "3":
        print("Done!")
```

```
            break
        else:
            print("Choose 1-3!")
```

- Result: Adds habits to a set, shows them with a fixed tuple of goals.

Write your own program with a set (e.g., completed tasks) and a tuple (e.g., daily priorities). Include a menu to add and view. Check: Does the set ensure uniqueness? Is the tuple fixed? Fix any errors.

---

## Why Sets and Tuples Matter

- **Sets**: Perfect for tracking unique items, like completed goals or skills, ensuring no duplicates. They're efficient for comparisons, like finding common tasks across days.
- **Tuples**: Ideal for data that shouldn't change, like a fixed schedule or settings, keeping your program predictable.
- **Real-World Impact**: These structures help manage complex data, like tracking progress in a job-training program. They mirror your journey of building a unique, unchangeable commitment to change—breaking the "chain gang" cycle and becoming a "change gang" coder.

By mastering sets and tuples, you're adding tools to build programs that are efficient and reliable, just like the transformation you're creating in your life. As Edmund Hillary and Roger Bannister showed, breaking barriers opens doors for others. Your coding journey does the same for your community.

In the next chapter, we'll explore **list comprehensions** and **lambda functions** to write concise, powerful code. Keep practicing on paper—you're crafting a future one line at a time!

---

**Reflection Prompt**
How could sets or tuples help you organize something in your life, like goals or habits? How does learning these structures make you feel about your potential to inspire others?

**Paper Coding Exercise**
Write a Python program on paper that:
- Uses a set to track unique items (e.g., skills learned).
- Uses a tuple for fixed data (e.g., a schedule).
- Includes a menu to add items, view both, and quit.
- Uses a set operation (e.g., intersection) to compare data.

Example: A program to track unique exercises and compare them to a fixed workout plan. Run it in your head with sample inputs and fix any errors.

# Chapter 5: List Comprehensions and Lambda Functions

You've expanded your Python toolbox with sets and tuples, making your data handling more efficient and flexible. Now, let's take your coding to the next level with **list comprehensions** and **lambda functions**, two advanced techniques that let you write concise, powerful code. List comprehensions offer a compact way to create and process lists, while lambda functions provide quick, anonymous functions for tasks like sorting or filtering. These tools will streamline your programs and reinforce your journey as a "change gang" coder, breaking cycles and inspiring others. As with previous chapters, you'll practice on paper—no computer needed, just your brain, a pen, and some paper. Let's dive in and make your code sharper and smarter!

---

## Understanding List Comprehensions

A **list comprehension** is a concise way to create or transform a list in a single line, combining a loop and optional conditions. Instead of writing a full `for` loop, you can pack the logic into square brackets. Here's a basic example:

```
numbers = [1, 2, 3, 4]
doubles = [num * 2 for num in numbers]
print(doubles)  # Prints: [2, 4, 6, 8]
```

This is equivalent to:

```
doubles = []
for num in numbers:
    doubles.append(num * 2)
```

List comprehensions can include conditions:

```
evens = [num for num in numbers if num % 2 == 0]
print(evens)  # Prints: [2, 4]
```

- Syntax: `[expression for item in iterable if condition]`
- Use cases: Filtering data, transforming lists, or creating new lists from existing ones.

Here's a practical example:

```
tasks = ["study", "exercise", "read"]
```

```
long_tasks = [task for task in tasks if len(task) > 5]
print(long_tasks)  # Prints: ['exercise']
```

Let's practice list comprehensions.

**Exercise: Create a List Comprehension**
Write a Python program on paper that uses a list comprehension. Example:
- Code:

```
  # Filter high scores
  scores = [85, 60, 90, 45]
  high_scores = [score for score in scores if score >= 80]
  print("High scores:", high_scores)
```

- Result: Shows `High scores: [85, 90]`

Write your own program with a list comprehension to filter or transform a list (e.g., tasks longer than 4 letters or doubled goals). Check: Is the syntax correct (`[expression for item in list if condition]`)? Does it produce the expected list? Fix any errors.

---

## Exploring Lambda Functions

A **lambda function** is a small, anonymous function defined with the `lambda` keyword. It's perfect for quick tasks, like sorting or applying a rule, without writing a full function. The syntax is:

```
lambda arguments: expression
```

For example:

```
double = lambda x: x * 2
print(double(5))  # Prints: 10
```

Lambda functions are often used with built-in functions like `sorted()` or `map()`. Here's an example with sorting a list of tuples:

```
goals = [("study", 50), ("exercise", 80), ("read", 20)]
sorted_goals = sorted(goals, key=lambda x: x[1])
print(sorted_goals)  # Prints: [('read', 20), ('study', 50), ('exercise', 80)]
```

- `key=lambda x: x[1]`: Sorts by the second item (progress) in each tuple.
- Lambda functions are short and don't need a `def` or name.

You can also use `map()` to apply a lambda to every item in a list:

```
numbers = [1, 2, 3]
squares = list(map(lambda x: x ** 2, numbers))
print(squares)  # Prints: [1, 4, 9]
```

Let's practice lambda functions.

**Exercise: Lambda Function Program**
Write a Python program on paper that uses a lambda function. Example:
- Code:

```
  # Triple numbers
  numbers = [2, 4, 6]
  triples = list(map(lambda x: x * 3, numbers))
  print("Triples:", triples)
```

- Result: Shows `Triples: [6, 12, 18]`

Write your own program using a lambda function to transform a list (e.g., add 10 to scores) or sort it (e.g., by length of strings). Check: Is the lambda syntax correct (`lambda x: expression`)? Does it work as expected? Fix any errors.

---

## Combining List Comprehensions and Lambda Functions

List comprehensions and lambda functions are powerful together, letting you process data concisely. Here's a program that uses both to manage tasks:

```
tasks = [("study", 2), ("exercise", 4), ("read", 1)]
# Filter tasks with hours >= 2 using list comprehension
long_tasks = [task[0] for task in tasks if task[1] >= 2]
print("Long tasks:", long_tasks)

# Sort tasks by hours using lambda
sorted_tasks = sorted(tasks, key=lambda x: x[1], reverse=True)
print("Sorted by hours:", sorted_tasks)
```

Output:

```
Long tasks: ['study', 'exercise']
Sorted by hours: [('exercise', 4), ('study', 2), ('read', 1)]
```

Let's build a program that tracks progress toward goals, using both tools:

```
goals = [("learn Python", 50), ("exercise", 80), ("read book", 30)]
while True:
    print("1. View high-progress goals")
    print("2. View goals sorted by progress")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    if choice == "1":
        high_goals = [goal[0] for goal in goals if goal[1] >= 50]
        print("High progress goals:", high_goals)
    elif choice == "2":
        sorted_goals = sorted(goals, key=lambda x: x[1])
        print("Goals by progress:", sorted_goals)
    elif choice == "3":
        print("Keep pushing forward!")
        break
    else:
        print("Choose 1, 2, or 3!")
```

This program:
- Uses a **list comprehension** to filter goals with 50% or higher progress.
- Uses a **lambda function** to sort goals by progress.
- Runs a menu loop for interactivity.

Let's test it in your head:
- Choose `1` → Shows `High progress goals: ['learn Python', 'exercise']`
- Choose `2` → Shows `Goals by progress: [('read book', 30), ('learn Python', 50), ('exercise', 80)]`
- Choose `3` → "Keep pushing forward!"

Let's practice combining these.

**Exercise: List Comprehension and Lambda Program**
Write a Python program on paper that uses both a list comprehension and a lambda function.
Example:

- Code:

```
# Track habits
habits = [("study", 3), ("code", 5), ("rest", 2)]
while True:
    print("1. View habits > 2 hours")
    print("2. Sort habits by hours")
    choice = input("Choose 1-2: ")
    if choice == "1":
        long_habits = [h[0] for h in habits if h[1] > 2]
        print("Long habits:", long_habits)
    elif choice == "2":
        sorted_habits = sorted(habits, key=lambda x: x[1])
        print("Sorted habits:", sorted_habits)
    else:
        print("Choose 1 or 2!")
```

- Result: Shows:

```
Long habits: ['study', 'code']
Sorted habits: [('rest', 2), ('study', 3), ('code', 5)]
```

Write your own program with a list of tuples (e.g., tasks and priorities), a list comprehension to filter, and a lambda to sort. Include a menu. Check: Do the comprehension and lambda work? Is the menu loop correct? Fix any errors.

---

## Why List Comprehensions and Lambda Functions Matter

- **List Comprehensions**: Make your code concise and readable, reducing loops to one line. They're like streamlining your goals—efficient and clear.
- **Lambda Functions**: Offer quick solutions for small tasks, like sorting or transforming data, without cluttering your code. They're like quick decisions that keep you moving forward.
- **Real-World Impact**: These tools are used in data analysis, web apps, and automation—skills that reduce recidivism by opening doors to tech jobs. Just as Edmund Hillary and Roger Bannister broke barriers, your concise code shows others that "what one man can do, another can do."

By mastering these techniques, you're not just coding smarter—you're building a mindset of efficiency and possibility, inspiring your community to see a path beyond the "chain gang" to the "change gang."

In the next chapter, we'll dive into **file handling** to save and retrieve data, making your programs persistent like your commitment to change. Keep practicing on paper—you're crafting a future of opportunity!

---

**Reflection Prompt**
How could list comprehensions or lambda functions simplify a program you've written, like a to-do list or quiz? How does writing concise code make you feel about your growth as a coder?

**Paper Coding Exercise**
Write a Python program on paper that:
- Uses a list comprehension to filter a list of tuples (e.g., goals with high progress).
- Uses a lambda function to sort or transform data.
- Includes a menu loop with at least two options (e.g., filter, sort).
Example: A program to filter tasks by time spent and sort by priority. Run it in your head with sample inputs and fix any errors.

# Chapter 6: File Handling and Persistence

You've sharpened your Python skills with list comprehensions and lambda functions, making your code concise and powerful. Now, let's make your programs last beyond a single run with **file handling**. This chapter teaches you how to save data to files and read it back, ensuring your programs can store information like tasks, goals, or progress—key to building real-world applications. File handling adds **persistence**, meaning your data sticks around, just like your commitment to becoming a "change gang" coder. You'll practice on paper—no computer needed, just your brain, a pen, and some paper. Let's dive in and learn how to make your programs endure, inspiring yourself and others to break cycles and build lasting futures!

---

## Understanding File Handling

File handling in Python lets you save data to a file (like a text or CSV file) and read it later. This is crucial for programs like journals or trackers that need to remember data. Python uses the `open()` function with modes:
- `"w"`: Write (creates or overwrites a file).
- `"r"`: Read (reads an existing file).
- `"a"`: Append (adds to the end of a file).

Here's a simple example of writing to a file:

```
with open("goals.txt", "w") as file:
    file.write("Learn Python: 50%\n")
```

And reading it back:

```
with open("goals.txt", "r") as file:
    content = file.read()
    print(content)  # Prints: Learn Python: 50%
```

- `with` ensures the file closes automatically, preventing errors.
- `write()` adds text; `read()` retrieves it.
- `\n` adds a new line for readability.

Since you're practicing on paper, you'll **simulate** file handling by imagining the file's content as a string or list. For example, instead of writing to "goals.txt," you might store data in a list called `file_content = ["Learn Python: 50%"]`.

Let’s practice basic file handling.

**Exercise: Simulate Writing to a File**
Write a Python program on paper that simulates writing to a file. Example:
- Code:

```
  # Simulate saving tasks
  file_content = []
  task = input("Enter task: ")
  file_content.append(task + "\n")
  print("Saved to file:", file_content)
```

- Result: If user enters “Study Python,” shows `Saved to file: ['Study Python\n']`

Write your own program that simulates saving a goal or note to a “file” (a list). Check: Does the list act like a file? Is the data stored correctly? Fix any errors.

---

## Reading and Writing with Loops

File handling often involves loops to process multiple lines. Here’s an example that saves multiple tasks and reads them:

```
# Simulate file as a list
file_content = []

while True:
    print("1. Add task")
    print("2. View tasks")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    if choice == "1":
        task = input("Task: ")
        file_content.append(task + "\n")  # Simulate writing
        print("Task saved!")
    elif choice == "2":
        if len(file_content) == 0:
            print("No tasks in file.")
        else:
            print("Tasks:")
            for i, line in enumerate(file_content, 1):
                print(f"{i}. {line.strip()}")  # strip() removes \n
    elif choice == "3":
        print("File saved. Goodbye!")
```

```
        break
    else:
        print("Choose 1, 2, or 3!")
```

- `file_content` simulates the file, storing each task as a line.
- `strip()` removes newlines when displaying.
- The loop mimics a real program's interactivity.

In a real program, you'd use:

```
with open("tasks.txt", "a") as file:
    file.write(task + "\n")  # Append to file
with open("tasks.txt", "r") as file:
    lines = file.readlines()  # Read all lines
```

Let's practice reading and writing.

**Exercise: Simulate File Read/Write Program**
Write a Python program on paper that simulates reading and writing to a file. Example:
- Code:

```
  # Simulate task tracker
  file_content = []
  while True:
      print("1. Add task  2. View tasks  3. Quit")
      choice = input("Choose 1-3: ")
      if choice == "1":
          task = input("Task: ")
          file_content.append(task)
          print("Saved!")
      elif choice == "2":
          if len(file_content) == 0:
              print("Empty file.")
          else:
              for task in file_content:
                  print(task)
      elif choice == "3":
          print("Done!")
          break
      else:
          print("Choose 1-3!")
```

- Result: Saves tasks to `file_content` and displays them.

Write your own program that simulates saving and reading data (e.g., goals or notes) using a list as the “file.” Include a menu. Check: Does the list store data correctly? Does the read option display it? Fix any errors.

---

## Working with CSV Files (Simulated)

**CSV files** (Comma-Separated Values) are great for structured data, like a table of tasks and statuses. A CSV might look like:

```
task,status
"Study Python","pending"
"Exercise","done"
```

In Python, you’d use the `csv` module, but on paper, you can simulate it with a list of lists or dictionaries. Here’s an example:

```
# Simulate CSV as a list of dictionaries
csv_file = []

while True:
    print("1. Add goal")
    print("2. View goals")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    if choice == "1":
        goal = input("Goal: ")
        status = input("Status (pending/done): ")
        csv_file.append({"goal": goal, "status": status})
        print("Goal saved!")
    elif choice == "2":
        if len(csv_file) == 0:
            print("No goals yet.")
        else:
            print("Goals:")
            for i, row in enumerate(csv_file, 1):
                print(f"{i}. {row['goal']} - {row['status']}")
    elif choice == "3":
        print("Goals saved. Keep going!")
        break
    else:
        print("Choose 1, 2, or 3!")
```

- `csv_file` simulates a CSV as a list of dictionaries.
- Each “row” has a goal and status, like a spreadsheet.
- In a real program, you’d use:

```
import csv
with open("goals.csv", "a", newline="") as file:
    writer = csv.DictWriter(file, fieldnames=["goal", "status"])
    writer.writerow({"goal": goal, "status": status})
```

Let’s practice CSV simulation.

**Exercise: Simulate CSV Program**
Write a Python program on paper that simulates a CSV file. Example:
- Code:

```
# Simulate goal tracker CSV
csv_file = []
while True:
    print("1. Add goal  2. View goals  3. Quit")
    choice = input("Choose 1-3: ")
    if choice == "1":
        goal = input("Goal: ")
        csv_file.append({"goal": goal, "progress": 0})
        print("Saved!")
    elif choice == "2":
        if len(csv_file) == 0:
            print("No goals.")
        else:
            for i, row in enumerate(csv_file, 1):
                print(f"{i}. {row['goal']} - {row['progress']}%")
    elif choice == "3":
        print("Done!")
        break
    else:
        print("Choose 1-3!")
```

- Result: Saves goals with 0% progress and displays them.

Write your own program that simulates a CSV (e.g., tasks with priorities) using a list of dictionaries. Include a menu. Check: Does each “row” have the right fields? Does the display work? Fix any errors.

---

## Why File Handling Matters

- **Persistence**: Files let your programs save data, like goals or progress, making them practical for real-world use, such as tracking skills for a job.
- **Organization**: CSVs structure data like a spreadsheet, perfect for complex apps like finance trackers.
- **Real-World Impact**: File handling is used in apps, databases, and automation—skills that open doors to tech jobs, reducing recidivism by up to 95% in programs like The Last Mile. Saving data is like saving your progress toward a “change gang” future, showing your community that “what one man can do, another can do,” just like Edmund Hillary and Roger Bannister.

By mastering file handling, you’re building programs that last, mirroring your commitment to lasting change for yourself and future generations.

In the next chapter, we’ll explore **error handling and exceptions** to make your programs even more robust. Keep practicing on paper—you’re coding a path to opportunity!

---

**Reflection Prompt**
How could file handling help you track something important, like goals or skills? How does making your programs persistent make you feel about your coding journey?

**Paper Coding Exercise**
Write a Python program on paper that:
- Simulates a file (list or list of dictionaries) to store data (e.g., tasks, goals).
- Includes a menu to add data, view it, and quit.
- Uses a loop to process multiple entries.
Example: A program to save and view study notes in a simulated CSV. Run it in your head with sample inputs and fix any errors.

# Chapter 7: Error Handling and Exceptions

You've learned to save data with file handling, making your Python programs persistent and practical. Now, let's make them **bulletproof** with advanced **error handling and exceptions**. Errors are inevitable—users might enter invalid data, files might not exist, or calculations might fail—but good programmers anticipate and manage these issues. In this chapter, you'll learn to use `try`/`except` blocks effectively, raise custom exceptions, and create robust programs that don't crash. These skills will make your code reliable, just like your commitment to becoming a "change gang" coder, breaking cycles and inspiring others. You'll practice on paper—no computer needed, just your brain, a pen, and some paper. Let's dive in and make your programs resilient!

---

## Understanding Error Handling

Python's `try`/`except` blocks catch errors (exceptions) to prevent your program from crashing. You've used them before for basic errors, like non-numeric input with `ValueError`. Now, let's deepen that knowledge. Common exceptions include:
- `ValueError`: Invalid data, like converting "abc" to an integer.
- `KeyError`: Accessing a nonexistent dictionary key.
- `FileNotFoundError`: Trying to open a missing file.

Here's a basic example:

```
try:
    number = int(input("Enter a number: "))
    print(f"You entered {number}")
except ValueError:
    print("Please enter a valid number!")
```

If the user enters "abc," the program catches the `ValueError` and prints an error message instead of crashing.

You can handle multiple exceptions:

```
try:
    data = {"score": 90}
    key = input("Enter key: ")
    print(data[key])
except KeyError:
    print("That key doesn't exist!")
except ValueError:
```

```
    print("Invalid input!")
```

Let's practice basic error handling.

**Exercise: Basic Try/Except Program**
Write a Python program on paper that uses `try`/`except` to handle an error. Example:
- Code:

```
  # Safe division
  try:
      num = int(input("Enter a number: "))
      result = 100 / num
      print(f"100 divided by {num} is {result}")
  except ValueError:
      print("Enter a number, not text!")
  except ZeroDivisionError:
      print("Cannot divide by zero!")
```

- Result: Handles non-numeric input or division by zero.

Write your own program that catches at least one exception (e.g., `ValueError` for invalid input in a goal tracker). Check: Does the `try`/`except` catch the right error? Is the message clear? Fix any errors.

---

## Raising Custom Exceptions

Sometimes, you want to enforce your own rules by **raising** exceptions with the `raise` keyword. For example, you might reject negative numbers in a program tracking progress:

```
progress = int(input("Enter progress (0-100): "))
if progress < 0 or progress > 100:
    raise ValueError("Progress must be between 0 and 100!")
```

You can combine `raise` with `try`/`except`:

```
try:
    progress = int(input("Enter progress (0-100): "))
    if progress < 0 or progress > 100:
        raise ValueError("Progress must be 0-100!")
    print(f"Progress set to {progress}%")
except ValueError as e:
```

```
    print(f"Error: {e}")
```

- `raise ValueError`: Triggers an exception with a custom message.
- `except ValueError as e`: Captures the error message for display.

Let's practice raising exceptions.

**Exercise: Raise Exception Program**
Write a Python program on paper that raises a custom exception. Example:
- Code:

```
  # Validate task length
  try:
      task = input("Enter task name: ")
      if len(task) < 3:
          raise ValueError("Task name must be at least 3 characters!")
      print(f"Task added: {task}")
  except ValueError as e:
      print(f"Error: {e}")
```

- Result: Rejects tasks shorter than 3 characters.

Write your own program that raises an exception for invalid input (e.g., negative goal progress or empty task names). Check: Does the `raise` condition make sense? Is the error caught? Fix any errors.

---

## Creating Custom Exception Classes

For more control, you can define your own exception classes by inheriting from `Exception`. This is useful for specific errors in complex programs. Here's an example:

```
class InvalidGoalError(Exception):
    pass

try:
    goal = input("Enter goal: ")
    if goal.strip() == "":
        raise InvalidGoalError("Goal cannot be empty!")
    print(f"Goal set: {goal}")
except InvalidGoalError as e:
    print(f"Error: {e}")
```

- `class InvalidGoalError(Exception)`: Creates a custom exception.
- `raise InvalidGoalError`: Triggers it with a specific message.

Let's practice custom exceptions.

**Exercise: Custom Exception Program**
Write a Python program on paper with a custom exception class. Example:
- Code:

```
  class ShortTaskError(Exception):
      pass

  try:
      task = input("Task name: ")
      if len(task) < 2:
          raise ShortTaskError("Task name too short!")
      print(f"Task saved: {task}")
  except ShortTaskError as e:
      print(f"Error: {e}")
```

- Result: Rejects tasks with fewer than 2 characters.

Write your own program with a custom exception (e.g., `InvalidProgressError` for negative progress). Check: Is the custom exception defined correctly? Does the program catch it? Fix any errors.

---

## Building a Robust Program

Let's combine error handling with a goal tracker that uses custom exceptions and file handling (simulated as a list). This program ensures robust input validation:

```
class InvalidGoalError(Exception):
    pass

file_content = []  # Simulate file
goals = {}

while True:
    print("1. Add goal")
    print("2. View goals")
    print("3. Quit")
    choice = input("Choose 1-3: ")
```

```
    try:
        if choice == "1":
            goal = input("Goal name: ")
            if goal.strip() == "":
                raise InvalidGoalError("Goal cannot be empty!")
            progress = int(input("Progress (0-100): "))
            if progress < 0 or progress > 100:
                raise ValueError("Progress must be 0-100!")
            goals[goal] = progress
            file_content.append(f"{goal},{progress}\n")  # Simulate CSV write
            print(f"Goal saved: {goal}")
        elif choice == "2":
            if len(goals) == 0:
                print("No goals yet.")
            else:
                print("Goals:")
                for i, (goal, prog) in enumerate(goals.items(), 1):
                    print(f"{i}. {goal}: {prog}%")
                print("File content:", file_content)
        elif choice == "3":
            print("Goals saved. Keep breaking cycles!")
            break
        else:
            raise ValueError("Choose 1, 2, or 3!")
    except InvalidGoalError as e:
        print(f"Goal Error: {e}")
    except ValueError as e:
        print(f"Input Error: {e}")
    except Exception:
        print("Unexpected error. Try again!")
```

This program:
- Uses a custom `InvalidGoalError` for empty goals.
- Validates progress and choices with `ValueError`.
- Simulates saving to a CSV file with `file_content`.
- Catches multiple exceptions for robustness.

Let's test it in your head:
- Choose `1`, enter "" (empty), 50 → "Goal Error: Goal cannot be empty!"
- Choose `1`, enter "Study", "abc" → "Input Error: invalid literal for int()…"
- Choose `1`, enter "Study", 50 → "Goal saved: Study"
- Choose `2` → Shows "1. Study: 50%" and file content.
- Choose `3` → "Goals saved. Keep breaking cycles!"

Let's practice a robust program.

**Exercise: Robust Goal Tracker**
Write a Python program on paper with custom exceptions and error handling. Example:
- Code:

```
class EmptyTaskError(Exception):
    pass

tasks = []
while True:
    print("1. Add task  2. View tasks  3. Quit")
    choice = input("Choose 1-3: ")
    try:
        if choice == "1":
            task = input("Task: ")
            if task.strip() == "":
                raise EmptyTaskError("Task cannot be empty!")
            tasks.append(task)
            print("Task saved!")
        elif choice == "2":
            if len(tasks) == 0:
                print("No tasks.")
            else:
                for i, task in enumerate(tasks, 1):
                    print(f"{i}. {task}")
        elif choice == "3":
            print("Done!")
            break
        else:
            raise ValueError("Invalid choice!")
    except EmptyTaskError as e:
        print(f"Error: {e}")
    except ValueError as e:
        print(f"Error: {e}")
```

- Result: Handles empty tasks and invalid choices.

Write your own program with a custom exception (e.g., `InvalidInputError`), a menu, and error handling for at least two cases. Check: Do all exceptions get caught? Is the program robust? Fix any errors.

---

## Why Error Handling Matters

- **Robustness**: Proper error handling prevents crashes, making your programs reliable, like a well-planned path to a tech job.
- **User Experience**: Clear error messages guide users, just as clear goals guide your journey from “chain gang” to “change gang.”
- **Real-World Impact**: Robust code is essential for professional apps, like those built by graduates of The Last Mile, who achieve near-zero recidivism through coding skills. By handling errors, you’re building programs—and a future—that can withstand challenges, inspiring others as Edmund Hillary and Roger Bannister did: “What one man can do, another can do.”

In the next chapter, we’ll explore **APIs** (simulated) to fetch and process external data, expanding your programs’ reach. Keep practicing on paper—you’re coding a resilient future!

---

**Reflection Prompt**
How could error handling make a program you’ve written (e.g., to-do list) more reliable? How does building robust code reflect your own journey of breaking cycles?

**Paper Coding Exercise**
Write a Python program on paper that:
- Defines a custom exception class (e.g., `InvalidDataError`).
- Uses a menu to add and view data (e.g., goals or tasks).
- Handles at least two types of errors (e.g., `ValueError`, custom exception).
Example: A task tracker that rejects empty names or invalid priorities. Run it in your head with sample inputs and fix any errors.

# Chapter 8: Working with APIs (Simulated)

You've built robust programs with advanced error handling, ensuring they can withstand mistakes and keep running smoothly. Now, let's expand your Python skills to connect with the outside world through **APIs** (Application Programming Interfaces). APIs let your programs fetch data from external sources, like weather updates, news, or task lists, making them dynamic and powerful. Since you're practicing on paper, we'll **simulate** API responses using dictionaries to mimic real data, allowing you to learn the logic without a computer. This chapter will teach you how to process API-like data, combine it with your existing skills, and apply it to practical programs—all with just your brain, a pen, and some paper. By mastering APIs, you're not just coding; you're opening doors to tech opportunities and showing your community that a "change gang" mindset can break cycles and inspire generations. Let's dive in!

---

## What Is an API?

An **API** is like a messenger that lets your program request and receive data from another system, such as a website or database. For example:
- A weather API might return `{"city": "New York", "temp": 75, "condition": "sunny"}`.
- A task API might return `[{"task": "Study Python", "status": "pending"}, {"task": "Exercise", "status": "done"}]`.

In a real program, you'd use the `requests` module to fetch data:

```
import requests
response = requests.get("https://api.example.com/weather")
data = response.json()  # Converts response to a Python dictionary
print(data["temp"])  # Prints: 75
```

Since you're on paper, we'll simulate API responses with predefined dictionaries or lists, mimicking the structure of real API data. This lets you practice processing data as if it came from an external source.

Here's a simple simulated API example:

```
# Simulate weather API response
weather_data = {"city": "Chicago", "temp": 68, "condition": "cloudy"}
print(f"Today in {weather_data['city']}: {weather_data['temp']}°F,
{weather_data['condition']}")
```

Output: `Today in Chicago: 68°F, cloudy`

Let’s practice simulating an API.

**Exercise: Simulate API Response**
Write a Python program on paper that processes a simulated API response. Example:
- Code:

```
# Simulate task API
api_response = [
    {"task": "Learn Python", "priority": 1},
    {"task": "Exercise", "priority": 2}
]
print("Tasks from API:")
for task in api_response:
    print(f"{task['task']} (Priority: {task['priority']})")
```

- Result: Shows:

```
Tasks from API:
Learn Python (Priority: 1)
Exercise (Priority: 2)
```

Write your own program with a simulated API response (e.g., a dictionary or list for goals or weather). Process and display the data. Check: Does the program access the data correctly? Is the output clear? Fix any errors.

---

## Processing API Data with Loops and Conditions

API responses often contain lists or dictionaries, so you’ll use loops, conditions, and list comprehensions to process them. Here’s an example that filters high-priority tasks from a simulated API:

```
api_response = [
    {"task": "Study", "priority": 3},
    {"task": "Rest", "priority": 1},
    {"task": "Code", "priority": 2}
]
high_priority = [task["task"] for task in api_response if task["priority"] >= 2]
print("High-priority tasks:", high_priority)
```

Output: `High-priority tasks: ['Study', 'Code']`

You can also use conditions to process nested data:

```
weather_data = {"city": "Miami", "forecast": [{"day": "Monday", "temp": 80},
{"day": "Tuesday", "temp": 82}]}
for day in weather_data["forecast"]:
    if day["temp"] > 80:
        print(f"Hot day: {day['day']} at {day['temp']}°F")
```

Output: `Hot day: Tuesday at 82°F`

Let's practice processing API data.

**Exercise: Process Simulated API Data**
Write a Python program on paper that processes a simulated API response with a loop or comprehension. Example:
- Code:

```
  # Filter completed goals
  api_response = [
      {"goal": "Learn Python", "status": "done"},
      {"goal": "Exercise", "status": "pending"}
  ]
  completed = [g["goal"] for g in api_response if g["status"] == "done"]
  print("Completed goals:", completed)
```

- Result: Shows `Completed goals: ['Learn Python']`

Write your own program with a simulated API response (e.g., goals or events). Use a loop or comprehension to filter or process data (e.g., show events for today). Check: Does the filtering work? Is the data structure correct? Fix any errors.

---

## Building a Program with Simulated APIs

Let's create a program that simulates an API for a goal tracker, combining API processing with your skills in loops, classes, and error handling. The "API" returns a list of goals, and you can view or filter them.

```
class InvalidFilterError(Exception):
```

```
    pass

# Simulated API response
api_response = [
    {"goal": "Study Python", "progress": 70, "category": "education"},
    {"goal": "Run 5K", "progress": 40, "category": "fitness"},
    {"goal": "Read Book", "progress": 90, "category": "education"}
]

while True:
    print("1. View all goals")
    print("2. View goals by category")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    try:
        if choice == "1":
            print("All goals from API:")
            for i, goal in enumerate(api_response, 1):
                print(f"{i}. {goal['goal']}: {goal['progress']}%
({goal['category']})")
        elif choice == "2":
            category = input("Enter category (education/fitness): ")
            if category not in ["education", "fitness"]:
                raise InvalidFilterError("Invalid category!")
            filtered = [g for g in api_response if g["category"] == category]
            if len(filtered) == 0:
                print(f"No goals in {category}.")
            else:
                print(f"{category} goals:")
                for i, goal in enumerate(filtered, 1):
                    print(f"{i}. {goal['goal']}: {goal['progress']}%")
        elif choice == "3":
            print("Keep coding your future!")
            break
        else:
            raise ValueError("Choose 1, 2, or 3!")
    except InvalidFilterError as e:
        print(f"Filter Error: {e}")
    except ValueError as e:
        print(f"Input Error: {e}")
    except Exception:
        print("Unexpected error. Try again!")
```

This program:
- Simulates an API with a list of goal dictionaries.
- Uses a custom `InvalidFilterError` for invalid categories.

- Processes data with loops and list comprehensions.
- Handles errors for robust interaction.

Let's test it in your head:
- Choose `1` → Shows all goals with progress and categories.
- Choose `2`, enter "education" → Shows "Study Python: 70%" and "Read Book: 90%".
- Choose `2`, enter "health" → "Filter Error: Invalid category!"
- Choose `3` → "Keep coding your future!"

Let's practice a full program.

**Exercise: Simulated API Program**
Write a Python program on paper that uses a simulated API response. Example:
- Code:

```
class InvalidStatusError(Exception):
    pass

api_response = [
    {"task": "Code", "status": "done"},
    {"task": "Rest", "status": "pending"}
]
while True:
    print("1. View all tasks  2. View done tasks  3. Quit")
    choice = input("Choose 1-3: ")
    try:
        if choice == "1":
            for i, task in enumerate(api_response, 1):
                print(f"{i}. {task['task']}: {task['status']}")
        elif choice == "2":
            done = [t["task"] for t in api_response if t["status"] == "done"]
            if not done:
                raise InvalidStatusError("No tasks done!")
            print("Done tasks:", done)
        elif choice == "3":
            print("Done!")
            break
        else:
            raise ValueError("Invalid choice!")
    except InvalidStatusError as e:
        print(f"Error: {e}")
    except ValueError as e:
        print(f"Error: {e}")
```

- Result: Shows all tasks or filters done ones, handles errors.

Write your own program with a simulated API (e.g., weather or tasks), a custom exception, and a menu to view or filter data. Check: Does the API data process correctly? Are errors handled? Fix any errors.

---

## Why APIs Matter

- **Dynamic Data**: APIs let your programs fetch real-time information, like weather or job listings, making them more useful.
- **Real-World Skills**: API handling is critical for web development and data analysis—fields with high-paying jobs that reduce recidivism, as seen in programs like The Last Mile, where graduates achieve near-zero reoffending rates.
- **Inspiration**: Processing external data mirrors your journey of pulling in new opportunities, breaking the “chain gang” cycle, and showing others, as Edmund Hillary and Roger Bannister did, that “what one man can do, another can do.”

By mastering API-like data processing, you’re building skills for dynamic programs and a dynamic future, inspiring your community to see coding as a path to change.

In the next chapter, we’ll explore **regular expressions** for advanced text processing, adding another tool to your Python arsenal. Keep practicing on paper—you’re coding a transformative path!

---

**Reflection Prompt**
How could an API help you build a program for your life, like tracking job skills or goals? How does learning to process data make you feel about your coding potential?

**Paper Coding Exercise**
Write a Python program on paper that:
- Simulates an API response (list or dictionary, e.g., tasks or events).
- Uses a menu to view all data or filter it (e.g., by status or type).
- Includes a custom exception for invalid inputs.
Example: A program to process a simulated job skills API, filtering high-demand skills. Run it in your head with sample inputs and fix any errors.

# Chapter 9: Regular Expressions for Text Processing

You've learned to handle dynamic data with simulated APIs, making your Python programs more connected and versatile. Now, let's dive into **regular expressions** (regex), a powerful tool for searching, validating, and manipulating text. Regex lets you find patterns—like emails, dates, or specific words—in strings, which is essential for tasks like data cleaning or input validation. In this chapter, you'll learn regex basics, use Python's `re` module (simulated on paper), and apply them to practical programs. This skill will make your code more precise, just like your commitment to becoming a "change gang" coder, breaking cycles and inspiring others. You'll practice on paper—no computer needed, just your brain, a pen, and some paper. Let's get started and unlock the power of text processing!

---

## Understanding Regular Expressions

A **regular expression** is a pattern that describes a set of strings. Python's `re` module lets you use regex to search, match, or replace text. For example, you can find all numbers in a string or check if an input is a valid email.

Here are common regex patterns:
- `\d`: Matches any digit (0-9).
- `\w`: Matches any word character (a-z, A-Z, 0-9, underscore).
- `.`: Matches any character (except newline).
- ``*: Matches zero or more occurrences.*
- `+`: Matches one or more occurrences.
- `\s`: Matches whitespace.

Example using `re` (simulated):

```
import re
text = "My number is 123 and my email is user@example.com"
numbers = re.findall(r"\d+", text)  # Find all numbers
print(numbers)  # Prints: ['123']
```

Key `re` functions:
- `re.findall(pattern, string)`: Returns a list of all matches.
- `re.search(pattern, string)`: Returns the first match or None.
- `re.match(pattern, string)`: Checks if the string starts with the pattern.

Since you're on paper, you'll simulate `re` by manually applying patterns to strings and noting matches.

Here's a simple example:

```
# Simulate finding dates (e.g., 2025-08-31)
text = "Meeting on 2025-08-31 and 2025-09-01"
dates = []  # Simulate re.findall(r"\d{4}-\d{2}-\d{2}", text)
dates.append("2025-08-31")
dates.append("2025-09-01")
print("Dates found:", dates)
```

Let's practice basic regex.

**Exercise: Simulate Regex Search**
Write a Python program on paper that simulates finding a pattern in a string. Example:
- Code:

```
  # Simulate finding numbers
  text = "Scores: 85, 90, 72"
  numbers = []  # Simulate re.findall(r"\d+", text)
  numbers.append("85")
  numbers.append("90")
  numbers.append("72")
  print("Numbers found:", numbers)
```

- Result: Shows `Numbers found: ['85', '90', '72']`

Write your own program that simulates finding a pattern (e.g., words with 3+ letters) in a string. List the matches manually. Check: Does the pattern make sense? Are the matches correct? Fix any errors.

---

## Validating Input with Regex

Regex is great for validating inputs, like ensuring an email follows the pattern `username@domain.com`. A simple email pattern might be `r"\w+@\w+\.\w+"` (word characters, @, word characters, dot, word characters).

Here's an example:

```
# Simulate email validation
email = input("Enter email: ")
if email in ["user@example.com", "test@domain.org"]:  # Simulate
re.match(r"\w+@\w+\.\w+", email)
```

```
    print("Valid email!")
else:
    print("Invalid email!")
```

In a real program:

```
import re
email = input("Enter email: ")
if re.match(r"\w+@\w+\.\w+", email):
    print("Valid email!")
else:
    print("Invalid email!")
```

Let's practice validation.

**Exercise: Simulate Regex Validation**
Write a Python program on paper that simulates validating input with regex. Example:
- Code:

```
  # Simulate date validation (YYYY-MM-DD)
  date = input("Enter date (YYYY-MM-DD): ")
  valid_dates = ["2025-08-31", "2025-09-01"]  # Simulate
re.match(r"\d{4}-\d{2}-\d{2}", date)
  if date in valid_dates:
      print("Valid date!")
  else:
      print("Invalid date!")
```

- Result: Validates dates like "2025-08-31."

Write your own program that simulates validating input (e.g., phone numbers like "123-456-7890"). Check: Is the pattern clear? Does it reject invalid inputs? Fix any errors.

---

## Combining Regex with a Program

Let's build a program that simulates processing a journal entry, extracting specific patterns (like dates or tasks) and validating inputs. We'll use a custom exception for robustness.

```
class InvalidEntryError(Exception):
    pass
```

```
# Simulate journal file
journal = ["2025-08-31: Study Python", "2025-09-01: Exercise", "Task: Code
app"]

while True:
    print("1. Add journal entry")
    print("2. View dates")
    print("3. View tasks")
    print("4. Quit")
    choice = input("Choose 1-4: ")

    try:
        if choice == "1":
            entry = input("Enter entry (e.g., YYYY-MM-DD: Task or Task: Name):
")
            if not entry:
                raise InvalidEntryError("Entry cannot be empty!")
            journal.append(entry)
            print("Entry saved!")
        elif choice == "2":
            dates = [line[:10] for line in journal if line[:4] in ["2025",
"2026"]]  # Simulate re.findall(r"\d{4}-\d{2}-\d{2}", line)
            if not dates:
                print("No dates found.")
            else:
                print("Dates:", dates)
        elif choice == "3":
            tasks = [line[6:] for line in journal if line.startswith("Task:")]
# Simulate re.findall(r"Task: (.+)", line)
            if not tasks:
                print("No tasks found.")
            else:
                print("Tasks:", tasks)
        elif choice == "4":
            print("Journal saved. Keep transforming!")
            break
        else:
            raise ValueError("Choose 1, 2, 3, or 4!")
    except InvalidEntryError as e:
        print(f"Entry Error: {e}")
    except ValueError as e:
        print(f"Input Error: {e}")
    except Exception:
        print("Unexpected error. Try again!")
```

This program:
- Simulates a journal as a list of strings.

- Uses list comprehensions to mimic regex for finding dates (`YYYY-MM-DD`) or tasks (`Task: ...`).
- Includes a custom `InvalidEntryError` for empty entries.
- Processes and displays data with a menu.

Let's test it in your head:
- Choose `1`, enter "2025-09-02: Read" → "Entry saved!"
- Choose `1`, enter "Task: Debug code" → "Entry saved!"
- Choose `2` → Shows `Dates: ['2025-08-31', '2025-09-01', '2025-09-02']`
- Choose `3` → Shows `Tasks: ['Code app', 'Debug code']`
- Choose `4` → "Journal saved. Keep transforming!"

Let's practice a regex program.

**Exercise: Regex Journal Program**
Write a Python program on paper that simulates regex processing. Example:
- Code:

```
class EmptyNoteError(Exception):
    pass

notes = ["Task: Study", "2025-08-31: Plan", "Task: Exercise"]
while True:
    print("1. Add note  2. View tasks  3. Quit")
    choice = input("Choose 1-3: ")
    try:
        if choice == "1":
            note = input("Note (e.g., Task: Name): ")
            if not note:
                raise EmptyNoteError("Note cannot be empty!")
            notes.append(note)
            print("Note saved!")
        elif choice == "2":
            tasks = [n[6:] for n in notes if n.startswith("Task:")]
            if not tasks:
                print("No tasks.")
            else:
                print("Tasks:", tasks)
        elif choice == "3":
            print("Done!")
            break
        else:
            raise ValueError("Invalid choice!")
    except EmptyNoteError as e:
        print(f"Error: {e}")
    except ValueError as e:
        print(f"Error: {e}")
```

- Result: Saves notes and extracts tasks starting with “Task:”.

Write your own program with a simulated journal or log, extracting a pattern (e.g., dates or priorities) and including a custom exception. Check: Does the pattern extraction work? Are errors handled? Fix any errors.

---

## Why Regular Expressions Matter

- **Precision**: Regex lets you find and validate specific patterns, like emails or dates, making your programs accurate.
- **Versatility**: Used in data cleaning, web scraping, and input validation—skills critical for tech jobs that reduce recidivism, as seen in programs like The Last Mile with near-zero reoffending rates.
- **Inspiration**: Just as regex extracts meaning from text, you’re extracting new opportunities from your journey, showing your community, like Edmund Hillary and Roger Bannister, that “what one man can do, another can do.”

By mastering regex, you’re adding precision to your coding arsenal, building programs—and a future—that are reliable and impactful.

In the next chapter, we’ll build a **personal finance manager** project, combining all your skills to create a practical tool. Keep practicing on paper—you’re coding a transformative path!

---

**Reflection Prompt**
How could regex help you process text in your life, like notes or goals? How does mastering text processing make you feel about your potential to inspire others?

**Paper Coding Exercise**
Write a Python program on paper that:
- Simulates a journal or log as a list of strings.
- Uses a menu to add entries and extract a pattern (e.g., tasks or dates).
- Includes a custom exception for invalid inputs.
Example: A log that extracts priorities (e.g., “Priority: High”) and rejects empty entries. Run it in your head with sample inputs and fix any errors.

# Chapter 10: Project: Build a Personal Finance Manager

You've mastered advanced Python skills—classes, sets, tuples, list comprehensions, lambda functions, file handling, error handling, APIs, and regular expressions. Now, let's put them all together in a practical project: a **Personal Finance Manager**. This program will track income, expenses, and savings goals, helping you manage money and plan for the future. It's a real-world application that combines your skills into a tool you could use post-release, reinforcing your journey as a "change gang" coder breaking cycles and inspiring others. You'll design it on paper—no computer needed, just your brain, a pen, and some paper. Let's build a program that empowers financial freedom and shows your community that "what one man can do, another can do"!

---

## Planning the Personal Finance Manager

Before coding, let's plan the program:
- **Purpose**: Track income, expenses, and savings goals to monitor financial progress.
- **Features**:
  - Add income or expenses with categories (e.g., food, rent).
  - View balances and summaries (e.g., total income, expenses, savings).
  - Save transactions to a "file" (simulated as a list or CSV).
  - Validate inputs (e.g., positive amounts, valid categories).
- **Python Tools**:
  - **Classes**: Use a `Transaction` class to store income/expense details.
  - **Dictionaries/Lists**: Store transactions in a list or dictionary.
  - **File Handling**: Simulate saving to a CSV file.
  - **Error Handling**: Use custom exceptions for invalid inputs.
  - **Regex**: Validate input formats (e.g., amounts like "10.50").
- **Output**: Clear summaries with totals and categorized transactions.

This project mirrors your journey: just as you're managing resources to build a new future, this program helps manage finances to break cycles of hardship.

---

## Designing the Program

Let's create a **Personal Finance Manager** that uses a class, error handling, and simulated file storage. Here's the complete program:

```
import re

class InvalidAmountError(Exception):
    pass

class InvalidCategoryError(Exception):
    pass

class Transaction:
    def __init__(self, type, amount, category):
        self.type = type  # "income" or "expense"
        self._amount = amount  # Private
        self.category = category

    def get_details(self):
        return f"{self.type.capitalize()}: ${self._amount} ({self.category})"

    def get_amount(self):
        return self._amount if self.type == "income" else -self._amount

# Simulated CSV file
csv_file = []
transactions = []
categories = ["salary", "food", "rent", "savings"]

while True:
    print("1. Add income")
    print("2. Add expense")
    print("3. View summary")
    print("4. Quit")
    choice = input("Choose 1-4: ")

    try:
        if choice in ["1", "2"]:
            type = "income" if choice == "1" else "expense"
            amount = input("Enter amount (e.g., 10.50): ")
            if not re.match(r"^\d+\.\d{2}$", amount):  # Simulate regex for
decimal
                raise InvalidAmountError("Amount must be like 10.50!")
            amount = float(amount)
            if amount <= 0:
                raise InvalidAmountError("Amount must be positive!")
            category = input(f"Category ({', '.join(categories)}): ")
            if category not in categories:
                raise InvalidCategoryError(f"Category must be one of
{categories}!")

            trans = Transaction(type, amount, category)
            transactions.append(trans)
```

```
            csv_file.append({"type": type, "amount": amount, "category":
category})
            print(f"{type.capitalize()} added: ${amount} ({category})")

        elif choice == "3":
            if not transactions:
                print("No transactions yet.")
            else:
                total_income = sum([t.get_amount() for t in transactions if
t.type == "income"])
                total_expense = sum([-t.get_amount() for t in transactions if
t.type == "expense"])
                balance = total_income - total_expense
                print(f"Summary: Income=${total_income:.2f},
Expenses=${total_expense:.2f}, Balance=${balance:.2f}")
                print("Transactions:")
                for i, trans in enumerate(transactions, 1):
                    print(f"{i}. {trans.get_details()}")
                print("File content:", csv_file)

        elif choice == "4":
            print("Finances saved. Keep building your future!")
            break
        else:
            raise ValueError("Choose 1, 2, 3, or 4!")
    except InvalidAmountError as e:
        print(f"Amount Error: {e}")
    except InvalidCategoryError as e:
        print(f"Category Error: {e}")
    except ValueError as e:
        print(f"Input Error: {e}")
    except Exception:
        print("Unexpected error. Try again!")
```

This program:
- Uses a `Transaction` class with private `_amount` and methods to access details.
- Validates amounts with a simulated regex (`r"^\d+\.\d{2}$"` for decimals like “10.50”).
- Uses custom exceptions (`InvalidAmountError`, `InvalidCategoryError`) for invalid inputs.
- Stores transactions in `transactions` (list) and `csv_file` (simulated CSV).
- Calculates totals with list comprehensions and shows a summary.
- Simulates saving to a CSV with a list of dictionaries.

Let’s test it in your head:
- Choose `1`, enter “100.50”, “salary” → “Income added: $100.50 (salary)”
- Choose `2`, enter “20.00”, “food” → “Expense added: $20.00 (food)”
- Choose `3` → Shows:

```
  Summary: Income=$100.50, Expenses=$20.00, Balance=$80.50
  Transactions:
  1. Income: $100.50 (salary)
  2. Expense: $20.00 (food)
  File content: [{'type': 'income', 'amount': 100.50, 'category': 'salary'},
{'type': 'expense', 'amount': 20.00, 'category': 'food'}]
```

- Choose `2`, enter “-5.00”, “rent” → “Amount Error: Amount must be positive!”
- Choose `4` → “Finances saved. Keep building your future!”

---

## Breaking Down the Program

- **Class (`Transaction`)**: Encapsulates transaction data (`type`, `_amount`, `category`) with methods to access or compute amounts (positive for income, negative for expenses).
- **Error Handling**: Catches invalid amounts (non-decimal or negative) and categories with custom exceptions.
- **File Handling**: Simulates a CSV file with `csv_file`, storing each transaction as a dictionary.
- **Regex**: Validates decimal amounts (simulated as a condition for paper practice).
- **List Comprehensions**: Calculates totals efficiently (`sum([t.get_amount() for t in transactions if t.type == "income"])`).
- **Menu Loop**: Provides interactivity with options to add, view, or quit.

This program combines nearly all your skills, creating a practical tool for financial planning.

---

## Practice Building the Project

Let’s practice a simplified version of the finance manager.

**Exercise: Simplified Finance Manager**
Write a Python program on paper for a basic finance manager. Example:
- Code:

```
  class InvalidInputError(Exception):
      pass

  transactions = []
  while True:
      print("1. Add transaction  2. View balance  3. Quit")
      choice = input("Choose 1-3: ")
```

```
try:
    if choice == "1":
        amount = int(input("Amount: "))
        if amount <= 0:
            raise InvalidInputError("Amount must be positive!")
        transactions.append(amount)
        print("Transaction saved!")
    elif choice == "2":
        if not transactions:
            print("No transactions.")
        else:
            total = sum(transactions)
            print(f"Balance: ${total}")
    elif choice == "3":
        print("Done!")
        break
    else:
        raise InvalidInputError("Invalid choice!")
except InvalidInputError as e:
    print(f"Error: {e}")
except ValueError:
    print("Error: Enter a number!")
```

- Result: Tracks positive amounts and shows the total balance.

Write your own program with:
- A class for transactions (at least one attribute and method).
- A menu to add transactions and view a summary.
- A custom exception for invalid inputs.
Check: Does the class work? Are errors caught? Does the summary calculate correctly? Fix any errors.

---

## Why This Project Matters

- **Practicality**: A finance manager helps you plan for stability, a key factor in reducing recidivism, as seen in programs like The Last Mile, where coding skills lead to jobs and near-zero reoffending rates.
- **Skill Integration**: This project uses classes, error handling, file simulation, and more, showcasing your advanced Python mastery.
- **Inspiration**: Managing finances mirrors managing your future—both require discipline and vision. By building this program, you're showing your community, like Edmund Hillary and Roger Bannister, that "what one man can do, another can do," inspiring generational change beyond the "chain gang" mindset.

In the next chapter, we'll build a **text adventure game**, combining your skills into a fun, interactive project. Keep practicing on paper—you're coding a path to financial and personal freedom!

---

**Reflection Prompt**
How could a finance manager help you plan for your future, like saving for education or a job? How does building this program make you feel about your ability to break cycles?

**Paper Coding Exercise**
Write a Python program on paper for a finance manager that:
- Uses a class for transactions with at least two attributes (e.g., amount, category).
- Includes a menu to add transactions, view a summary, and quit.
- Uses a custom exception and validates inputs (e.g., positive amounts).
- Simulates saving to a file (list or dictionary).
Example: A program to track income/expenses with categories. Run it in your head with sample inputs and fix any errors.

# Chapter 11: Project: Create a Text Adventure Game

You've built a powerful Personal Finance Manager, integrating classes, error handling, file handling, and more to create a practical tool. Now, let's combine your advanced Python skills into a fun and engaging project: a **Text Adventure Game**. This program will let users explore rooms, make choices, and manage an inventory, simulating a journey through a virtual world. It's a creative way to apply your knowledge of classes, dictionaries, lists, loops, and error handling, while mirroring your own journey of navigating challenges and breaking cycles as a "change gang" coder. You'll design it on paper—no computer needed, just your brain, a pen, and some paper. Let's craft a game that inspires you and others to see that "what one man can do, another can do"!

---

## Planning the Text Adventure Game

Before coding, let's plan the game:
- **Purpose**: Create an interactive text-based adventure where players explore rooms, interact with objects, and collect items.
- **Features**:
  - Rooms with descriptions and exits (e.g., "kitchen" leads to "hall").
  - A player with an inventory to collect items.
  - Choices to move between rooms, pick up items, or quit.
  - Error handling for invalid inputs (e.g., nonexistent rooms).
- **Python Tools**:
  - **Classes**: A `Player` class for inventory and location; a `Room` class for details.
  - **Dictionaries**: Store rooms and their connections (e.g., `{"kitchen": {"exits": ["hall"]}}`).
  - **Lists**: Track the player's inventory.
  - **Error Handling**: Use custom exceptions for invalid moves or actions.
  - **Loops/Conditions**: Handle user choices and game flow.
- **Output**: Describe the current room, show inventory, and confirm actions.

This game mirrors your journey: just as you navigate obstacles to build a new future, the player navigates rooms to achieve their goal, inspiring others to break free from the "chain gang" mindset.

---

## Designing the Program

Let's create a **Text Adventure Game** that uses classes, dictionaries, and error handling to guide a player through a simple world. Here's the complete program:

```
class InvalidActionError(Exception):
    pass

class Room:
    def __init__(self, name, description, exits, items):
        self.name = name
        self.description = description
        self.exits = exits  # List of room names
        self.items = items  # List of items in room

    def get_description(self):
        return f"{self.name}: {self.description}\nExits: {self.exits}\nItems:
{self.items}"

class Player:
    def __init__(self, location):
        self.location = location  # Current room name
        self.inventory = []

    def move(self, direction, rooms):
        if direction in rooms[self.location].exits:
            self.location = direction
            return f"Moved to {self.location}."
        raise InvalidActionError(f"Cannot go to {direction} from
{self.location}!")

    def pick_item(self, item, rooms):
        if item in rooms[self.location].items:
            rooms[self.location].items.remove(item)
            self.inventory.append(item)
            return f"Picked up {item}."
        raise InvalidActionError(f"No {item} in {self.location}!")

# Game setup
rooms = {
    "kitchen": Room("Kitchen", "A cozy room with a stove.", ["hall"],
["knife"]),
    "hall": Room("Hall", "A long corridor.", ["kitchen", "bedroom"], ["key"]),
    "bedroom": Room("Bedroom", "A quiet room with a bed.", ["hall"], ["book"])
}
player = Player("kitchen")

while True:
    print("\n" + rooms[player.location].get_description())
    print("Inventory:", player.inventory)
    print("1. Move to another room")
```

```
    print("2. Pick up an item")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    try:
        if choice == "1":
            direction = input(f"Enter direction ({',
'.join(rooms[player.location].exits)}): ")
            print(player.move(direction, rooms))
        elif choice == "2":
            item = input(f"Enter item ({',
'.join(rooms[player.location].items)}): ")
            print(player.pick_item(item, rooms))
        elif choice == "3":
            print("Thanks for playing! Keep breaking barriers!")
            break
        else:
            raise InvalidActionError("Choose 1, 2, or 3!")
    except InvalidActionError as e:
        print(f"Error: {e}")
    except Exception:
        print("Unexpected error. Try again!")
```

This program:
- Uses a `Room` class to store room details (name, description, exits, items).
- Uses a `Player` class to track location and inventory.
- Stores rooms in a dictionary with names as keys and `Room` objects as values.
- Handles actions (move, pick item) with error checking via `InvalidActionError`.
- Runs a menu loop to interact with the game world.

Let’s test it in your head:
- Start in kitchen → Shows “Kitchen: A cozy room with a stove. Exits: ['hall'] Items: ['knife']”
- Choose `1`, enter “hall” → “Moved to hall.”
- Choose `2`, enter “key” → “Picked up key.”
- Choose `2`, enter “book” → “Error: No book in hall!”
- Choose `3` → “Thanks for playing! Keep breaking barriers!”

---

## Breaking Down the Program

- **Classes**:
  - `Room`: Encapsulates room data with a method to display details.
  - `Player`: Manages location and inventory, with methods for moving and picking items.

- **Dictionary**: `rooms` stores the game world, linking rooms via exits.
- **Error Handling**: `InvalidActionError` catches invalid moves or item pickups.
- **Menu Loop**: Drives interactivity, letting players explore and act.
- **Data Structures**: Lists (`exits`, `items`, `inventory`) and dictionaries (`rooms`) organize the game.

This program combines your skills into a fun, interactive experience, showcasing your ability to build complex systems.

---

## Practice Building the Project

Let's practice a simplified version of the text adventure game.

**Exercise: Simplified Text Adventure**
Write a Python program on paper for a basic text adventure. Example:
- Code:

```
class InvalidMoveError(Exception):
    pass

rooms = {
    "cave": {"desc": "A dark cave.", "exits": ["forest"]},
    "forest": {"desc": "A dense forest.", "exits": ["cave"]}
}
player_location = "cave"

while True:
    print(f"{player_location}: {rooms[player_location]['desc']}")
    print("Exits:", rooms[player_location]['exits'])
    print("1. Move  2. Quit")
    choice = input("Choose 1-2: ")
    try:
        if choice == "1":
            destination = input("Where to? ")
            if destination in rooms[player_location]["exits"]:
                player_location = destination
                print(f"Moved to {player_location}")
            else:
                raise InvalidMoveError(f"Cannot go to {destination}!")
        elif choice == "2":
            print("Game over!")
            break
        else:
            raise InvalidMoveError("Invalid choice!")
```

```
        except InvalidMoveError as e:
            print(f"Error: {e}")
```

- Result: Lets players move between rooms and handles invalid moves.

Write your own program with:
- A dictionary for at least two rooms with descriptions and exits.
- A menu to move or quit.
- A custom exception for invalid actions.
Check: Do the exits work? Are errors caught? Does the loop update the location? Fix any errors.

---

## Why This Project Matters

- **Creativity**: A text adventure lets you build a world, sparking imagination and problem-solving, just like crafting a new future.
- **Skill Integration**: Combines classes, dictionaries, error handling, and loops, showcasing your advanced Python mastery.
- **Real-World Impact**: Game development is a viable career path, with skills transferable to web or app development—fields that reduce recidivism, as seen in programs like The Last Mile, where coders achieve near-zero reoffending rates. By building this game, you're showing your community, like Edmund Hillary and Roger Bannister, that "what one man can do, another can do," inspiring generational change.

In the next chapter, we'll reflect on your coding journey and plan your future as a coder, tying together all you've learned. Keep practicing on paper—you're coding a path to a transformative future!

---

**Reflection Prompt**
How could a text adventure reflect your own journey, like overcoming obstacles? How does building a game make you feel about your potential to inspire others?

**Paper Coding Exercise**
Write a Python program on paper for a text adventure that:
- Uses a class for rooms or the player (with at least one method).
- Includes a dictionary for rooms with descriptions and exits.
- Has a menu to move, interact (e.g., pick items), and quit.
- Uses a custom exception for invalid actions.

Example: A game with two rooms and an inventory system. Run it in your head with sample inputs and fix any errors.

# Chapter 12: Your Future in Coding

You've reached the final chapter of *Code Without Bars: Advanced Python Adventures*! You've built a powerful toolkit—classes, inheritance, encapsulation, sets, tuples, list comprehensions, lambda functions, file handling, error handling, APIs, regular expressions, and two major projects: a Personal Finance Manager and a Text Adventure Game. These skills aren't just about writing code; they're about rewriting your future, breaking cycles of recidivism, and inspiring generations to see beyond the "chain gang" to a "change gang" mindset. In this chapter, we'll reflect on your progress, plan your next steps in coding, and design a final paper-based program to solidify your skills. No computer is needed—just your brain, a pen, and some paper. Let's chart your path forward and celebrate your journey as a coder who proves, like Edmund Hillary and Roger Bannister, that "what one man can do, another can do"!

---

## Reflecting on Your Progress

You've come a long way from basic variables and loops. Let's review what you've mastered:
- **Object-Oriented Programming**: Created classes like `Transaction` and `Room`, using inheritance and encapsulation to model real-world systems.
- **Advanced Data Structures**: Used sets for unique items, tuples for fixed data, and dictionaries for structured storage.
- **Efficient Coding**: Wrote concise code with list comprehensions and lambda functions.
- **Robustness**: Handled errors with custom exceptions, ensuring programs don't crash.
- **Real-World Applications**: Built a finance manager to plan budgets and a text adventure game for creative problem-solving.
- **Persistence and Processing**: Simulated file handling and APIs to save and fetch data, plus regex for precise text manipulation.

These skills are the foundation for professional programming, opening doors to careers in web development, data analysis, or automation—fields where programs like The Last Mile show coders achieving near-zero recidivism rates and jobs paying $60,000+ annually. Your journey mirrors those of trailblazers like Hillary, who summited Everest, and Bannister, who broke the four-minute mile, showing that barriers are meant to be overcome.

**Exercise: Reflect on Your Journey**
On paper, answer these questions:
1. Which advanced concept (e.g., classes, regex) are you most proud of mastering? Why?
2. How has a project (e.g., finance manager, adventure game) changed how you see coding?
3. How can coding help you break cycles of recidivism and inspire others in your community?
Check: Are your answers specific? For example, instead of "classes," say "using a `Player` class to manage inventory." Refine for clarity.

---

## Applying Your Skills to Break Cycles

Your Python skills are more than technical—they're transformative. Coding reduces recidivism by providing marketable skills for stable jobs, as seen in programs like The Last Mile, where graduates achieve 0-5% recidivism compared to the national 70-75%. Here's how your skills can make an impact:
- **Personal Growth**: Build tools like trackers or planners to organize your goals, like budgeting for education or tracking job skills.
- **Career Opportunities**: Use your knowledge for freelance work, web development, or data analysis—fields with high demand and second-chance employers.
- **Generational Change**: By becoming a coder, you show your children and community that a "change gang" path is possible, breaking cycles of incarceration. As one Last Mile graduate said, "Coding didn't just change my life; it changed my kids' future."

Your programs can inspire others, just as Hillary and Bannister inspired climbers and runners by proving the impossible was possible.

**Exercise: Plan a Real-World Application**
On paper, describe a program you could build to support your future or community. Example:
- Program: Job Skill Tracker
- Features: Add skills (e.g., Python, HTML), track progress, save to a "file."
- Python Tools: Class for skills, dictionary for storage, file handling, error handling.
- Purpose: Prepare for tech jobs and show progress to employers.
Write your own program idea with:
- What it does.
- At least two Python features (e.g., classes, regex).
- How it could reduce recidivism or inspire others.
Check: Is the plan clear and achievable with your skills? Refine if needed.

---

## Designing a Final Program

Let's cap your journey with a final project: a **Skill Builder App**. This program tracks skills you're learning (e.g., coding, communication), their progress, and categories, simulating a tool for job readiness. It combines classes, dictionaries, file handling, and error handling, reflecting your full skill set.

Here's the program:

```
import re

class InvalidSkillError(Exception):
    pass

class Skill:
    def __init__(self, name, progress, category):
        self.name = name
        self._progress = progress  # Private
        self.category = category

    def update_progress(self, amount):
        if amount < 0:
            raise InvalidSkillError("Progress cannot be negative!")
        self._progress += amount
        return f"{self.name} progress updated to {self._progress}%"

    def get_details(self):
        return f"{self.name} ({self.category}): {self._progress}%"

# Simulated CSV file
csv_file = []
skills = []
categories = ["coding", "communication", "leadership"]

while True:
    print("1. Add skill")
    print("2. Update skill progress")
    print("3. View skills by category")
    print("4. Quit")
    choice = input("Choose 1-4: ")

    try:
        if choice == "1":
            name = input("Skill name: ")
            if not name.strip():
                raise InvalidSkillError("Skill name cannot be empty!")
            category = input(f"Category ({', '.join(categories)}): ")
            if category not in categories:
                raise InvalidSkillError(f"Category must be one of
{categories}!")
            skill = Skill(name, 0, category)
            skills.append(skill)
            csv_file.append({"name": name, "progress": 0, "category":
category})
            print(f"Added {name}")
        elif choice == "2":
            name = input("Skill to update: ")
```

```
            for skill in skills:
                if skill.name == name:
                    amount = input("Progress to add (e.g., 10): ")
                    if not re.match(r"^\d+$", amount):  # Simulate regex for
integer
                        raise InvalidSkillError("Progress must be a number!")
                    amount = int(amount)
                    print(skill.update_progress(amount))
                    # Update CSV simulation
                    for row in csv_file:
                        if row["name"] == name:
                            row["progress"] = skill._progress
                    break
            else:
                raise InvalidSkillError("Skill not found!")
        elif choice == "3":
            category = input(f"Category ({', '.join(categories)}): ")
            if category not in categories:
                raise InvalidSkillError(f"Category must be one of
{categories}!")
            filtered = [s for s in skills if s.category == category]
            if not filtered:
                print(f"No skills in {category}.")
            else:
                print(f"{category} skills:")
                for i, skill in enumerate(filtered, 1):
                    print(f"{i}. {skill.get_details()}")
                print("File content:", csv_file)
        elif choice == "4":
            print("Skills saved. Keep building your future!")
            break
        else:
            raise ValueError("Choose 1, 2, 3, or 4!")
    except InvalidSkillError as e:
        print(f"Skill Error: {e}")
    except ValueError as e:
        print(f"Input Error: {e}")
    except Exception:
        print("Unexpected error. Try again!")
```

This program:
- Uses a `Skill` class with private `_progress`, methods for updating and displaying details.
- Validates inputs with regex (simulated for integers) and custom `InvalidSkillError`.
- Stores skills in a list and simulates a CSV file with a list of dictionaries.
- Filters skills by category using list comprehensions.
- Runs a menu loop for interactivity.

Let’s test it in your head:
- Choose `1`, enter “Python”, “coding” → “Added Python”
- Choose `2`, enter “Python”, “20” → “Python progress updated to 20%”
- Choose `3`, enter “coding” → Shows “1. Python (coding): 20%” and file content.
- Choose `2`, enter “Java”, “10” → “Skill Error: Skill not found!”
- Choose `4` → “Skills saved. Keep building your future!”

**Exercise: Final Program Design**
Write a Python program on paper for a skill or goal tracker (or similar app, e.g., learning log).
Example:
- Code:

```
class InvalidGoalError(Exception):
    pass

goals = []
while True:
    print("1. Add goal  2. View goals  3. Quit")
    choice = input("Choose 1-3: ")
    try:
        if choice == "1":
            name = input("Goal: ")
            if not name:
                raise InvalidGoalError("Goal cannot be empty!")
            goals.append({"name": name, "progress": 0})
            print("Goal added!")
        elif choice == "2":
            if not goals:
                print("No goals.")
            else:
                for i, goal in enumerate(goals, 1):
                    print(f"{i}. {goal['name']}: {goal['progress']}%")
        elif choice == "3":
            print("Done!")
            break
        else:
            raise ValueError("Invalid choice!")
    except InvalidGoalError as e:
        print(f"Error: {e}")
    except ValueError as e:
        print(f"Error: {e}")
```

- Result: Tracks goals and handles errors.

Write your own program with:
- A class for data (e.g., skill or goal) with at least one method.
- A dictionary or list to store data.

- A menu with at least three options (e.g., add, update, view).
- A custom exception for invalid inputs.
Run it in your head with sample inputs and fix any errors.

---

## Planning Your Coding Future

Your skills are ready for the real world. Here's how to keep growing:
- **Practice Regularly**: Write one small program daily on paper, like a task filter or regex validator.
- **Build Projects**: Plan apps for your needs, like a resume builder or habit tracker, to prepare for jobs.
- **Learn More**: Study advanced topics like databases or web frameworks (e.g., Flask) on paper to prep for computer access.
- **Join Communities**: When possible, connect with coders on platforms like X to share ideas and get feedback.
- **Inspire Others**: Share your journey to show your community that coding breaks cycles, as seen in programs with 0-5% recidivism rates.

**Exercise: Set a Coding Goal**
On paper, write a coding goal for the next month. Example:
- Goal: Design 5 paper-based programs, like a job tracker or quiz app.
- Plan: Spend 15 minutes daily planning and writing one feature.
- Why: Build confidence and prepare for tech opportunities.
Write your own goal with:
- What you'll do (e.g., design programs, learn a new concept).
- How you'll do it (e.g., daily practice, study a topic).
- Why it matters (e.g., job readiness, inspiring others).
Check: Is the goal specific and achievable? Adjust if needed.

---

## Why This Matters

Your coding journey is a testament to transformation. By mastering advanced Python, you're building skills for careers that reduce recidivism and create stability, as seen in programs like The Last Mile. You're also setting an example for your children and community, showing that a "change gang" coder can break generational cycles, just as Hillary and Bannister broke barriers. Every program you write is a step toward a future where you're not defined by your past but by your potential.

**Final Reflection Prompt**
How has learning advanced Python changed how you see yourself? How will you use coding to build a future for yourself and inspire others?

**Final Paper Coding Exercise**
Write a Python program on paper for an app of your choice (e.g., skill tracker, journal, or mini-game). Include:
- A class with at least two methods.
- A dictionary or list to store data.
- A menu loop with at least three options.
- A custom exception for invalid inputs.
- Simulated file handling (list or dictionary).
Run it in your head with sample inputs and fix any errors. Save this program—it's your blueprint for a transformative future!

---

*Thank you for completing Code Without Bars: Advanced Python Adventures! You're not just a coder—you're a change-maker. Keep coding, keep inspiring, and keep proving that "what one man can do, another can do!"*