PYTHON

Projects

Number Guessing Game · Simple Calculator · Password Generator

To-Do List CLI App · Quiz Game · Contact Book

Simple Web Scraper · Capstone: Expense Tracker

Philip Osztromok

Generated with Claude

Table of Contents

Python Projects (Beginner) — 8 Chapters

🎯 Number Guessing Game

Welcome to Python Projects — a different format from Courses 1-3. Instead of a topic followed by challenges, each chapter here builds one complete, playable project end to end, scoped to Fundamentals-level skills. First up: a number guessing game, using nothing beyond loops, conditionals, and the `random` module — all from Course 1.

🎮 What We're Building

A simple, genuinely fun command-line game: the program secretly picks a random number between 1 and 100, the player guesses repeatedly, and after each guess the program says "too high," "too low," or "correct" — ending with how many attempts it took.

Step 1: Picking a Random Number

```
import random

secret_number = random.randint(1, 100)
```

`random.randint(1, 100)` reuses the exact function from Course 1, Chapter 9's own dice-roller challenge — inclusive on both ends, so it can return anything from 1 through 100.

Step 2: The Guessing Loop

```
while True:
    guess = int(input("Guess a number between 1 and 100: "))
```

An infinite `while True` loop (Course 1, Chapter 4) — the game keeps asking for guesses until something inside the loop explicitly `break`s out of it. `input()` always returns a string, so `int(...)` wraps it immediately to get a real number to compare.

Step 3: Giving Feedback

```
if guess < secret_number:
    print("Too low!")
elif guess > secret_number:
    print("Too high!")
else:
    print("Correct! 🎉")
    break
```

Plain `if`/`elif`/`else` comparisons (Course 1, Chapter 3) — the `break` only fires on a correct guess, exiting the `while True` loop from Step 2 at exactly that point.

Step 4: Counting Attempts & Ending the Game

```
attempts = 0
while True:
    guess = int(input("Guess a number between 1 and 100: "))
    attempts += 1
if guess < secret_number:
        print("Too low!")
    elif guess > secret_number:
        print("Too high!")
    else:
        print(f"Correct! You got it in {attempts} attempts. 🎉")
        break
```

`attempts` is the same running-total accumulator pattern from Course 1, Chapter 4 — initialized to 0 outside the loop, incremented by 1 on every pass through it, then read once the loop finally ends.

🏁 The Complete Game

```
import random

secret_number = random.randint(1, 100)
attempts = 0
print("I'm thinking of a number between 1 and 100.")

while True:
    guess = int(input("Your guess: "))
    attempts += 1
if guess < secret_number:
        print("Too low!")
    elif guess > secret_number:
        print("Too high!")
    else:
        print(f"Correct! You got it in {attempts} attempts. 🎉")
        break
```

That's a complete, playable game in 15 lines — every piece is something already covered in Course 1, combined into something genuinely fun to run rather than a standalone exercise.

⚠ A Non-Numeric Guess Crashes This Version

Typing `"forty"` instead of a number raises a `ValueError` from `int(...)`, crashing the whole program — this version doesn't validate input yet. Course 2, Chapter 3's `try`/`except ValueError` is exactly the fix, and it's one of this chapter's own suggested extensions below.

random.randint()

Picks the secret number, inclusive on both ends.

while True + break

Loops until the correct guess is made, then exits.

if/elif/else

Compares the guess to the secret number and gives a hint.

Accumulator pattern

`attempts` counts guesses across every pass through the loop.

🚀 Extend This Project

Try these on your own — no solutions provided, just like a real side project:

- Wrap the `int(input(...))` line in a `try`/`except ValueError` (Course 2, Chapter 3) so a non-numeric guess prints a friendly message and asks again, instead of crashing.
- Add a difficulty selector at the start (easy = 1-50, hard = 1-500), changing the `random.randint()` range based on the player's choice.
- After the player wins, ask if they want to play again — wrap the entire game in an outer loop, reusing the loop-inside-a-loop pattern from Course 2, Chapter 5's nested comprehensions (conceptually, not literally a comprehension here).
- Give up after 10 attempts instead of looping forever — track `attempts` against a limit and `break` with a "you lose" message if it's exceeded.

🎯 What's Next

Next chapter: **Simple Calculator** — functions and basic error handling.

🧮 Simple Calculator

A command-line calculator: four operations, a repeating menu, and just enough error handling that a bad input doesn't crash the whole program. Built entirely from functions and conditionals — Course 1 material, put to real use.

🎮 What We're Building

A menu-driven calculator: pick an operation, enter two numbers, get a result — and keep going until you choose to quit, without the program ever crashing on a mistake like dividing by zero.

Step 1: The Four Operations as Functions

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

def subtract(a, b):
    return a - b

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

def divide(a, b):
    return a / b
```

Four small, single-purpose functions (Course 1, Chapter 8) — each takes two numbers and returns one result. Keeping each operation in its own function is what makes Step 3's menu logic clean: it just calls the right one, rather than repeating the arithmetic inline four separate times.

Step 2: Basic Error Handling — Division by Zero

```
def divide(a, b):
    if b == 0:
        print("Error: can't divide by zero")
        return None
return a / b
```

The simplest form of error handling is a guard clause — a plain `if` check (Course 1, Chapter 3) before the risky operation, rather than letting it crash. Returning `None` signals "this didn't produce a real result," the same convention Course 2, Chapter 3's `safe_divide()` challenge used.

Step 3: A Menu Loop

```
while True:
    print("\n1. Add  2. Subtract  3. Multiply  4. Divide  5. Quit")
    choice = input("Choose an option: ")

    if choice == "5":
        print("Goodbye!")
        break

    a = float(input("First number: "))
    b = float(input("Second number: "))

    if choice == "1":
        print("Result:", add(a, b))
    elif choice == "2":
        print("Result:", subtract(a, b))
    elif choice == "3":
        print("Result:", multiply(a, b))
    elif choice == "4":
        print("Result:", divide(a, b))
    else:
        print("Not a valid option, try again.")
```

The same `while True` + `break` shape from Chapter 1's guessing game — the menu keeps redisplaying until `"5"` is chosen. `float(input(...))` allows decimal numbers, not just whole ones, wider than the `int()` conversion Chapter 1 used.

🏁 The Complete Calculator

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

def subtract(a, b):
    return a - b

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

def divide(a, b):
    if b == 0:
        print("Error: can't divide by zero")
        return None
return a / b

while True:
    print("\n1. Add  2. Subtract  3. Multiply  4. Divide  5. Quit")
    choice = input("Choose an option: ")

    if choice == "5":
        print("Goodbye!")
        break

    a = float(input("First number: "))
    b = float(input("Second number: "))

    if choice == "1":
        print("Result:", add(a, b))
    elif choice == "2":
        print("Result:", subtract(a, b))
    elif choice == "3":
        print("Result:", multiply(a, b))
    elif choice == "4":
        print("Result:", divide(a, b))
    else:
        print("Not a valid option, try again.")
```

⚠ Non-Numeric Input Still Crashes This Version

Typing `"five"` for a number raises a `ValueError` from `float(...)`, the exact same crash risk flagged in Chapter 1's guessing game. This version's "basic error handling" is specifically the division-by-zero guard — handling malformed number input too would mean reaching for `try`/`except` (Course 2, Chapter 3), left here as an extension rather than baked in, to keep this chapter's core build within Course 1-level tools.

Functions

Each operation is its own small, reusable function.

Guard clause

An `if` check before the risky operation — the simplest error handling.

Menu loop

`while True` + `break`, dispatching on the user's choice.

float() conversion

Allows decimal input, wider than `int()`.

🚀 Extend This Project

Try these on your own:

- Wrap the two `float(input(...))` lines in `try`/`except ValueError` (Course 2, Chapter 3), printing a friendly message and returning to the menu instead of crashing.
- Add an exponent operation (`**`) and a modulo operation (`%`) as menu options 6 and 7.
- Keep a running history of every calculation in a list, and add a menu option to print it all out.
- Instead of a numbered menu, accept an operator symbol directly (`+`, `-`, `*`, `/`) as the choice.

🎯 What's Next

Next chapter: **Password Generator** — strings and the `random`/`secrets` modules.

🔐 Password Generator

A customizable password generator — but the real lesson in this chapter isn't string handling, it's a genuinely important distinction most beginners never hear explicitly: `random` and `secrets` look similar but are not interchangeable for anything security-related.

🎮 What We're Building

A tool that asks how long the password should be and which character types to include (lowercase, uppercase, digits, symbols), then generates a genuinely random password matching those choices.

Step 1: Character Pools with the string Module

```
import string

print(string.ascii_lowercase)   # abcdefghijklmnopqrstuvwxyz
print(string.ascii_uppercase)   # ABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.digits)            # 0123456789
print(string.punctuation)       # !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

The `string` module provides these character sets as ready-made constants — no need to type them out by hand. Combining the ones you want (with `+`, Course 1, Chapter 5's string concatenation) builds the pool a password's characters get picked from.

Step 2: random vs secrets — Choosing the Right Tool

```
import random
import secrets

random.choice("abc")    # fine for games, simulations, non-security randomness
secrets.choice("abc")   # the correct choice for anything security-sensitive
```

⚠ random Is Not Cryptographically Secure

Course 1, Chapter 9 and the earlier projects in this course used `random` for a dice roll and a secret number — perfectly fine there, since nothing security-sensitive depended on it. `random`'s numbers are generated by a pseudo-random algorithm that is **predictable** if an attacker can determine its internal state — completely unacceptable for passwords, tokens, or anything else where guessability matters. The `secrets` module exists specifically for this: it's built on a cryptographically secure random source, and `secrets.choice()` is the correct replacement for `random.choice()` anywhere security is actually at stake.

Step 3: Building the Password

```
pool = string.ascii_lowercase + string.ascii_uppercase + string.digits

password = "".join(secrets.choice(pool) for _ in range(12))
print(password)
```

A generator expression (Course 2, Chapter 5) calls `secrets.choice(pool)` once per character, 12 times; `"".join(...)` (Course 1, Chapter 5) combines the individual characters into one final string. `_` is a common convention for a loop variable whose actual value is never used — only the number of repetitions matters here.

Step 4: Letting the User Customize It

```
length = int(input("Password length: "))

use_upper = input("Include uppercase? (y/n): ").lower() == "y"
use_digits = input("Include digits? (y/n): ").lower() == "y"
use_symbols = input("Include symbols? (y/n): ").lower() == "y"

pool = string.ascii_lowercase
if use_upper:
    pool += string.ascii_uppercase
if use_digits:
    pool += string.digits
if use_symbols:
    pool += string.punctuation
```

Each `if` check (Course 1, Chapter 3) conditionally grows the character pool — `pool +=` is the same string-concatenation-as-accumulation pattern from Course 1, Chapter 4's loop-accumulator idea, just applied once per option instead of once per loop iteration.

🏁 The Complete Password Generator

```
import string
import secrets

length = int(input("Password length: "))
use_upper = input("Include uppercase? (y/n): ").lower() == "y"
use_digits = input("Include digits? (y/n): ").lower() == "y"
use_symbols = input("Include symbols? (y/n): ").lower() == "y"

pool = string.ascii_lowercase
if use_upper:
    pool += string.ascii_uppercase
if use_digits:
    pool += string.digits
if use_symbols:
    pool += string.punctuation

password = "".join(secrets.choice(pool) for _ in range(length))
print(f"Your password: {password}")
```

string module

`ascii_lowercase`, `ascii_uppercase`, `digits`, `punctuation` — ready-made character sets.

secrets, not random

Cryptographically secure — the correct choice for anything security-sensitive.

Conditional pool building

Each `if` optionally adds a character set based on user choice.

Generator expression + join

Builds the final password one character at a time, joined into a string.

🚀 Extend This Project

Try these on your own:

- Guarantee at least one character from each selected type appears (right now, a short password could randomly end up all lowercase even with other options enabled) — build one guaranteed character per selected type first, then fill the rest randomly.
- Add input validation with `try`/`except` (Course 2, Chapter 3) around the length input.
- Generate several passwords at once and let the user pick their favorite from a numbered list.
- Add a simple strength estimate — print "weak"/"medium"/"strong" based on length and how many character types were included.

🎯 What's Next

Next chapter: **To-Do List CLI App** — functions and basic persistence to a text/JSON file.

📝 To-Do List CLI App

Python Advanced's capstone (`py3-8`) built a task tracker using classes, `@dataclass`, and a dedicated storage abstraction. This chapter builds something genuinely similar — a to-do list that persists across runs — using nothing but plain functions and a list of dicts, proving real persistence doesn't require OOP at all.

🎮 What We're Building

A command-line to-do list: add tasks, list them, mark them done, and have the list still be there the next time you run the program — all stored in a plain JSON file.

Step 1: Representing a Task as a Dict

```
task = {"title": "Buy milk", "done": False}
tasks = [task]
```

No class needed — a task is just a dict (Course 1, Chapter 7) with two keys, and the whole to-do list is a plain list of these dicts. This is the direct "no heavy OOP required" alternative to `py3-8`'s `Task` dataclass — less structure, but perfectly workable for a project this size.

Step 2: Loading and Saving with JSON

```
import json
from pathlib import Path

FILE = Path("tasks.json")

def load_tasks():
    if not FILE.exists():
        return []
    return json.loads(FILE.read_text())

def save_tasks(tasks):
    FILE.write_text(json.dumps(tasks, indent=2))
```

`pathlib` and `json` here work exactly like Course 2, Chapters 4 and 7 — but note there's no separate `Task` class to reconstruct on load, unlike `py3-8`'s `Task(**item)` step. `json.loads()` already returns plain dicts, which is exactly what this version's `tasks` list is made of — one less layer, at the cost of losing the type-checking and auto-generated `__eq__` a dataclass would give.

Step 3: Adding, Listing, Completing Tasks

```
def add_task(tasks, title):
    tasks.append({"title": title, "done": False})

def list_tasks(tasks):
    if not tasks:
        print("No tasks yet!")
        return
for i, task in enumerate(tasks, start=1):
        status = "x" if task["done"] else " "
print(f"{i}. [{status}] {task['title']}")

def complete_task(tasks, index):
    if 0 <= index < len(tasks):
        tasks[index]["done"] = True
else:
        print("Invalid task number.")
```

`enumerate(tasks, start=1)` reuses Course 1, Chapter 4's numbered-list pattern for a human-friendly 1-based display. `complete_task` mutates the dict directly through its list position — `tasks[index]["done"] = True` — since `tasks.append()` is passed the same list object every function receives (Course 1's mutable-list-aliasing behavior, here working in the app's favor).

⚠ Using List Position as an ID Is Fragile

`complete_task` identifies a task purely by its position in the list. If tasks are ever deleted (see this chapter's extension ideas) or reordered, a previously-noted "task #3" can silently become a different task. `py3-8`'s dataclass version sidestepped this entirely with a persistent `id` field that never changes — worth adding here too if the list ever needs deletion or reordering support.

Step 4: The Menu Loop

```
tasks = load_tasks()

while True:
    print("\n1. Add  2. List  3. Complete  4. Quit")
    choice = input("Choose an option: ")

    if choice == "1":
        title = input("Task title: ")
        add_task(tasks, title)
        save_tasks(tasks)
    elif choice == "2":
        list_tasks(tasks)
    elif choice == "3":
        num = int(input("Task number to complete: "))
        complete_task(tasks, num - 1)
        save_tasks(tasks)
    elif choice == "4":
        print("Goodbye!")
        break
else:
        print("Not a valid option, try again.")
```

Same menu-loop shape as Chapters 1 and 2. `save_tasks(tasks)` is called after every mutation (add, complete) — skip it and the change would only exist in memory for this run, lost the moment the program exits, undermining the whole point of persistence.

🏁 The Complete To-Do App

```
import json
from pathlib import Path

FILE = Path("tasks.json")

def load_tasks():
    if not FILE.exists():
        return []
    return json.loads(FILE.read_text())

def save_tasks(tasks):
    FILE.write_text(json.dumps(tasks, indent=2))

def add_task(tasks, title):
    tasks.append({"title": title, "done": False})

def list_tasks(tasks):
    if not tasks:
        print("No tasks yet!")
        return
for i, task in enumerate(tasks, start=1):
        status = "x" if task["done"] else " "
print(f"{i}. [{status}] {task['title']}")

def complete_task(tasks, index):
    if 0 <= index < len(tasks):
        tasks[index]["done"] = True
else:
        print("Invalid task number.")

tasks = load_tasks()

while True:
    print("\n1. Add  2. List  3. Complete  4. Quit")
    choice = input("Choose an option: ")

    if choice == "1":
        title = input("Task title: ")
        add_task(tasks, title)
        save_tasks(tasks)
    elif choice == "2":
        list_tasks(tasks)
    elif choice == "3":
        num = int(input("Task number to complete: "))
```

```
            complete_task(tasks, num - 1)
            save_tasks(tasks)
        elif choice == "4":
            print("Goodbye!")
            break
    else:
            print("Not a valid option, try again.")
```

Task as a dict

No class needed — `{"title": ..., "done": ...}` is enough.

load_tasks / save_tasks

JSON + pathlib persistence, functions instead of a storage class.

Mutating through a shared reference

Functions modify the same `tasks` list every caller shares.

Save after every mutation

Skipping `save_tasks()` loses the change when the program exits.

🚀 Extend This Project

Try these on your own:

- Add a `delete_task(tasks, index)` function and menu option, using a list comprehension (Course 1/2) to build the filtered list.
- Fix this chapter's own warn-box: give each task a persistent `id` (an incrementing counter, like `py3-8`'s `next_id` logic) instead of relying on list position.
- Add input validation with `try`/`except` (Course 2, Chapter 3) around the task-number input.
- Add a due date field and sort `list_tasks()`'s output by it.

🎯 What's Next

Next chapter: **Quiz Game** — dictionaries, lists, and scoring logic.

🧠 Quiz Game

A multiple-choice quiz that scores itself — questions stored as a list of dictionaries, presented one at a time, with a running score tallied and reported as a percentage at the end.

🎮 What We're Building

A quiz that presents a series of multiple-choice questions, checks each answer as it's given, and prints a final score out of the total, plus a percentage.

Step 1: Representing Questions

```
questions = [
    {
        "question": "What does CPU stand for?",
        "choices": ["Central Processing Unit", "Computer Personal Unit", "Central Program Utility"],
        "answer": "A"
    },
    {
        "question": "Which keyword defines a function in Python?",
        "choices": ["func", "def", "function"],
        "answer": "B"
    },
]
```

Each question is a dict (Course 1, Chapter 7) with three keys; the whole quiz is a list of these dicts, reusing exactly the "list of dicts" shape Chapter 4's to-do app used for tasks. `"answer"` stores the correct choice's letter, not its full text — comparing letters is simpler than comparing whole sentences.

Step 2: Presenting a Question

```
letters = "ABC"
def ask_question(q):
    print(q["question"])
    for i, choice in enumerate(q["choices"]):
        print(f"  {letters[i]}. {choice}")
```

`enumerate` (Course 1, Chapter 4) pairs each choice with its position; `letters[i]` reuses plain string indexing (Course 1, Chapter 5) to turn that position into the matching letter — `letters[0]` is `"A"`, `letters[1]` is `"B"`, and so on.

Step 3: Checking Answers & Scoring

```
def check_answer(q, user_answer):
    return user_answer.strip().upper() == q["answer"]
```

⚠ Normalize User Input Before Comparing

A player typing `"b"`, `" B"`, or `"B "` should all count as correct — but `"b" == "B"` is `False` (Course 1's string comparisons are case-sensitive), and stray whitespace from pressing Enter carelessly breaks an exact match too. `.strip()` (Course 1, Chapter 5) removes surrounding whitespace, `.upper()` normalizes case — chaining both before comparing is standard practice for any free-text-ish user input.

Step 4: The Quiz Loop & Final Score

```
score = 0
for q in questions:
    ask_question(q)
    user_answer = input("Your answer: ")
    if check_answer(q, user_answer):
        print("Correct!\n")
        score += 1
else:
        print(f"Nope — the answer was {q['answer']}.\n")

percentage = (score / len(questions)) * 100
print(f"Final score: {score}/{len(questions)} ({percentage:.0f}%)")
```

`score` is the accumulator pattern one more time (Course 1, Chapter 4) — incremented once per correct answer, read only after the loop finishes. `score / len(questions)` reuses the average-style division from earlier chapters, scaled to a percentage and formatted to a whole number with `:.0f` (Course 1, Chapter 5).

🏁 The Complete Quiz Game

```
questions = [
    {
        "question": "What does CPU stand for?",
        "choices": ["Central Processing Unit", "Computer Personal Unit", "Central Program Utility"],
        "answer": "A"
    },
    {
        "question": "Which keyword defines a function in Python?",
        "choices": ["func", "def", "function"],
        "answer": "B"
    },
]

letters = "ABC"
def ask_question(q):
    print(q["question"])
    for i, choice in enumerate(q["choices"]):
        print(f"  {letters[i]}. {choice}")

def check_answer(q, user_answer):
    return user_answer.strip().upper() == q["answer"]

score = 0
for q in questions:
    ask_question(q)
    user_answer = input("Your answer: ")
    if check_answer(q, user_answer):
        print("Correct!\n")
        score += 1
else:
        print(f"Nope — the answer was {q['answer']}.\n")

percentage = (score / len(questions)) * 100
print(f"Final score: {score}/{len(questions)} ({percentage:.0f}%)")
```

List of dicts

Each question is a dict; the quiz is a list of them — same shape as Chapter 4's tasks.

enumerate + string indexing

Turns each choice's position into a letter label (A, B, C...).

Normalized comparison

`.strip().upper()` before comparing user input to the stored answer.

Scoring

An accumulator, converted to a percentage at the end.

🚀 Extend This Project

Try these on your own:

- Use `random.shuffle(questions)` so the question order is different each playthrough.
- Load `questions` from an external JSON file (Course 2, Chapter 7) instead of hardcoding them, so new quizzes don't require editing the code.
- Add a category field to each question, and let the player choose a category before starting.
- Track and display which specific questions were missed at the end, not just the final score.

🎯 What's Next

Next chapter: **Contact Book** — file I/O and CRUD operations against a JSON file.

📇 Contact Book

Chapter 4's to-do app only ever added and completed tasks. This chapter builds the full set: **C**reate, **R**ead, **U**pdate, and **D**elete — the four operations almost every real data-backed app needs — applied to a simple JSON-backed contact book.

🎮 What We're Building

A contact book storing name, phone, and email per contact, supporting adding new contacts, searching/listing them, editing an existing contact's details, and deleting one — all persisted to a JSON file between runs.

Step 1: Representing a Contact

```
contact = {"name": "Ada Lovelace", "phone": "555-0100", "email": "ada@example.com"}
```

Same "plain dict, no class" approach as Chapter 4's tasks — a contact is just three related pieces of string data (Course 1, Chapters 5 and 7).

Step 2: Create — Adding a Contact

```
def add_contact(contacts, name, phone, email):
    contacts.append({"name": name, "phone": phone, "email": email})
```

Step 3: Read — Listing and Searching

```
def list_contacts(contacts):
    for i, c in enumerate(contacts, start=1):
        print(f"{i}. {c['name']} — {c['phone']} — {c['email']}")

def search_contacts(contacts, query):
    query = query.lower()
    return [c for c in contacts if query in c["name"].lower()]
```

`search_contacts` reuses a list comprehension (Course 1/2) with a substring `in` check — `.lower()` on both sides makes the search case-insensitive, so searching `"ada"` still finds `"Ada Lovelace"`.

Step 4: Update — Editing a Contact

```
def update_contact(contacts, index, field, new_value):
    if 0 <= index < len(contacts):
        contacts[index][field] = new_value
    else:
        print("Invalid contact number.")
```

The same bounds-checked guard clause from Chapter 4's `complete_task`. `contacts[index][field] = new_value` reassigns one key on the target contact's dict directly — `field` can be `"name"`, `"phone"`, or `"email"`, letting one function handle updating any of the three.

Step 5: Delete — Removing a Contact

```
def delete_contact(contacts, index):
    if 0 <= index < len(contacts):
        del contacts[index]
    else:
        print("Invalid contact number.")
```

`del contacts[index]` reuses `del` from Course 1, Chapter 7's dictionary coverage — it works on list items too, removing exactly the one element at that position and shifting the rest down.

⚠ Deleting Shifts Every Later Index Down by One

After `delete_contact(contacts, 2)`, whatever was previously contact `#4` is now contact `#3` — every index after the deleted one shifts down. If the menu loop (Step 6) ever displayed a list and then asked for a number to act on *before* re-listing, a stale index could now point at the wrong contact. Always re-list before acting on an index, or (as Chapter 4's own extension suggested) give each contact a persistent ID that never changes regardless of deletions elsewhere in the list.

Step 6: Tying It Together — Load, Save, and the Menu Loop

```
import json
from pathlib import Path

FILE = Path("contacts.json")

def load_contacts():
    if not FILE.exists():
        return []
    return json.loads(FILE.read_text())

def save_contacts(contacts):
    FILE.write_text(json.dumps(contacts, indent=2))
```

Identical `load`/`save` shape to Chapter 4's to-do app — this pattern (a `Path`, a `load_` function returning `[]` if the file doesn't exist yet, a `save_` function writing the whole list back) is worth recognizing as reusable across almost any small JSON-backed CLI tool, not something unique to contacts or tasks specifically.

🏁 The Complete Contact Book

```
import json
from pathlib import Path

FILE = Path("contacts.json")

def load_contacts():
    if not FILE.exists():
        return []
    return json.loads(FILE.read_text())

def save_contacts(contacts):
    FILE.write_text(json.dumps(contacts, indent=2))

def add_contact(contacts, name, phone, email):
    contacts.append({"name": name, "phone": phone, "email": email})

def list_contacts(contacts):
    for i, c in enumerate(contacts, start=1):
        print(f"{i}. {c['name']} — {c['phone']} — {c['email']}")

def search_contacts(contacts, query):
    query = query.lower()
    return [c for c in contacts if query in c["name"].lower()]

def update_contact(contacts, index, field, new_value):
    if 0 <= index < len(contacts):
        contacts[index][field] = new_value
    else:
        print("Invalid contact number.")

def delete_contact(contacts, index):
    if 0 <= index < len(contacts):
        del contacts[index]
    else:
        print("Invalid contact number.")

contacts = load_contacts()

while True:
    print("\n1. Add  2. List  3. Search  4. Update  5. Delete  6. Quit")
    choice = input("Choose an option: ")

    if choice == "1":
        name = input("Name: ")
```

```
        phone = input("Phone: ")
        email = input("Email: ")
        add_contact(contacts, name, phone, email)
        save_contacts(contacts)
    elif choice == "2":
        list_contacts(contacts)
    elif choice == "3":
        query = input("Search for: ")
        for c in search_contacts(contacts, query):
            print(f"{c['name']} — {c['phone']} — {c['email']}")
    elif choice == "4":
        list_contacts(contacts)
        num = int(input("Contact number to update: "))
        field = input("Field to update (name/phone/email): ")
        new_value = input("New value: ")
        update_contact(contacts, num - 1, field, new_value)
        save_contacts(contacts)
    elif choice == "5":
        list_contacts(contacts)
        num = int(input("Contact number to delete: "))
        delete_contact(contacts, num - 1)
        save_contacts(contacts)
    elif choice == "6":
        print("Goodbye!")
        break
else:
        print("Not a valid option, try again.")
```

Note that options 4 and 5 both call `list_contacts(contacts)` first, before asking for a number — directly applying this chapter's own warn-box advice, so the number the user types always matches what they're currently looking at.

Create

`add_contact()` — appends a new dict to the list.

Read

`list_contacts()` / `search_contacts()` — display or filter.

Update

`update_contact()` — reassigns one field on an existing dict.

Delete

`delete_contact()` — `del` by index, shifting later items down.

🚀 Extend This Project

Try these on your own:

- Give each contact a persistent `id` instead of relying on list position, fixing this chapter's own warn-box concern properly.
- Add a CSV export option using the `csv` module (Course 2, Chapter 7) so contacts can be opened in a spreadsheet.
- Validate that `email` contains an `@` symbol before accepting it, printing an error and re-asking otherwise.
- Prevent exact duplicate names from being added, warning the user and asking for confirmation first.

🎯 What's Next

Next chapter: **Simple Web Scraper** — introducing an external library, `requests` and BeautifulSoup.

Simple Web Scraper

Every project so far has used only the standard library. This one introduces two genuinely popular third-party packages — `requests` for fetching a web page, and `BeautifulSoup` for parsing its HTML — putting Course 1, Chapter 9's `pip` and `venv` coverage to real use for the first time in this course.

🎮 What We're Building

A scraper for quotes.toscrape.com — a site built specifically as safe, legal scraping practice — that fetches the page, pulls out every quote's text and author, and saves the results to a JSON file.

Step 1: Installing the Libraries

```
# with a venv active (Course 1, Chapter 9):
pip install requests beautifulsoup4
```

Two packages installed in one `pip install` call — `requests` handles the network request, `beautifulsoup4` (imported as `bs4`) handles parsing the returned HTML into something searchable.

Step 2: Fetching a Page with requests

```
import requests

response = requests.get("https://quotes.toscrape.com")
print(response.status_code)   # 200 — success
print(response.text[:200])    # the raw HTML, as a string
```

`requests.get(url)` sends an HTTP request and returns a `Response` object. `status_code` follows the same HTTP status conventions from the site's own API-focused courses (200 = success, 404 = not found); `.text` is the full HTML the server sent back, as a plain string.

Step 3: Parsing HTML with BeautifulSoup

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(response.text, "html.parser")
quote_divs = soup.find_all("div", class_="quote")
print(len(quote_divs))  # 10 — one per quote on the page
```

`BeautifulSoup(html, "html.parser")` turns the raw HTML string into a navigable object. `.find_all("div", class_="quote")` searches for every `<div class="quote">` element — the trailing underscore in `class_` avoids colliding with Python's own `class` keyword (Course 2, Chapter 1).

Step 4: Extracting the Data

```
quotes = []

for div in quote_divs:
    text = div.find("span", class_="text").get_text()
    author = div.find("small", class_="author").get_text()
    quotes.append({"text": text, "author": author})

print(quotes[0])
# {'text': '"The world as we have created it...', 'author': 'Albert Einstein'}
```

Within each quote's `div`, `.find()` (singular — one match) locates the specific `<span>`/`<small>` tags holding the text and author; `.get_text()` extracts just the readable text, stripping the surrounding HTML tags. `quotes.append({...})` reuses the exact list-of-dicts pattern from Chapters 4-6's to-do app and contact book.

Step 5: Saving the Results

```
import json
from pathlib import Path

Path("quotes.json").write_text(json.dumps(quotes, indent=2))
```

The same `pathlib` + `json` save pattern from every earlier project in this course — scraped data is just data, and it persists the same way hand-entered data does.

🏁 The Complete Web Scraper

```
import json
import requests
from bs4 import BeautifulSoup
from pathlib import Path

response = requests.get("https://quotes.toscrape.com")
soup = BeautifulSoup(response.text, "html.parser")
quote_divs = soup.find_all("div", class_="quote")

quotes = []
for div in quote_divs:
    text = div.find("span", class_="text").get_text()
    author = div.find("small", class_="author").get_text()
    quotes.append({"text": text, "author": author})

Path("quotes.json").write_text(json.dumps(quotes, indent=2))
print(f"Saved {len(quotes)} quotes.")
```

⚠ Scraping Has Real Legal and Ethical Limits

Not every site allows scraping — many explicitly forbid it in their Terms of Service, and even where it's technically allowed, hammering a server with rapid repeated requests can degrade it for real users or get your IP address blocked. Before scraping any real site: check its `/robots.txt` file (a standard, machine-readable statement of what's allowed), read its Terms of Service, and add a delay between requests (`time.sleep()`, Course 3, Chapter 3) if fetching multiple pages. `quotes.toscrape.com` is used here specifically because it's built and maintained *for* scraping practice — that permission doesn't transfer to other sites.

requests.get()

Fetches a page — `.status_code` and `.text` on the response.

BeautifulSoup

Parses HTML into a searchable structure.

find_all() / find()

Locate matching elements by tag name and class.

Scraping ethics

Check `robots.txt` and Terms of Service; don't hammer a server.

🚀 Extend This Project

Try these on your own:

- Follow the "Next →" link on each page to scrape all 10 pages of quotes, not just the first — adding a `time.sleep(1)` between requests, per this chapter's warn-box.
- Wrap `requests.get()` in `try`/`except` (Course 2, Chapter 3) to handle a network failure gracefully instead of crashing.
- Also scrape each quote's tags (`class_="tags"`), adding a `"tags"` key to each quote's dict.
- Count and print how many quotes each unique author has, using a dict as a counter (Course 1, Chapter 7's word-frequency pattern).

🎯 What's Next

Final chapter: **Capstone: Expense Tracker** — a small CLI app tying together file I/O, functions, and error handling into one complete, slightly larger project.

💰 Capstone: Expense Tracker

The final project — and the final chapter of the entire Python track. An expense tracker ties together everything this course has built: the load/save JSON pattern from Chapters 4-6, dict-based records, dict-based summarization (Course 1, Chapter 7's word-frequency idea, here counting money instead of words), and — for the first time in this course — `try/except` built directly into the app itself, rather than left as an extension.

🎮 What We're Building

An expense tracker: log expenses with an amount, category, and description; list them all; see totals by category and overall; delete a mistaken entry — all persisted to JSON, and all resilient to bad input.

Step 1: Representing an Expense

```
expense = {"amount": 12.50, "category": "Food", "description": "Lunch"}
```

Step 2: Load & Save

```
import json
from pathlib import Path

FILE = Path("expenses.json")

def load_expenses():
    if not FILE.exists():
        return []
    return json.loads(FILE.read_text())

def save_expenses(expenses):
    FILE.write_text(json.dumps(expenses, indent=2))
```

By this point in the course, this pattern should feel completely familiar — the fourth time this exact `load_`/`save_` shape has appeared, after the to-do app, contact book, and scraper.

Step 3: Adding an Expense — With Real Error Handling

```
def add_expense(expenses):
    try:
        amount = float(input("Amount: "))
    except ValueError:
        print("That's not a valid number — expense not added.")
        return
if amount <= 0:
        print("Amount must be positive — expense not added.")
        return

    category = input("Category: ")
    description = input("Description: ")
    expenses.append({"amount": amount, "category": category, "description": description})
    print("Expense added.")
```

Chapters 1-7 flagged crash-prone `float(input(...))` calls in warn-boxes and left the fix as an extension. Here, it's built in directly: `try`/`except ValueError` (Course 2, Chapter 3) catches non-numeric input cleanly, and a guard clause (Chapter 2's own technique) rejects a zero or negative amount as a separate, sensible business rule — two different kinds of "bad input," handled with the two different tools each one actually calls for.

Step 4: Viewing and Summarizing

```
def list_expenses(expenses):
    if not expenses:
        print("No expenses recorded yet.")
        return
for i, e in enumerate(expenses, start=1):
        print(f"{i}. ${e['amount']:.2f} — {e['category']} — {e['description']}")

def total_by_category(expenses):
    totals = {}
    for e in expenses:
        totals[e["category"]] = totals.get(e["category"], 0) + e["amount"]
    return totals

def total_spending(expenses):
    return sum(e["amount"] for e in expenses)
```

`total_by_category` reuses Course 1, Chapter 7's word-frequency-counter pattern exactly — `totals.get(category, 0) + amount` in place of `counts.get(word, 0) + 1`, summing dollar amounts per category instead of counting word occurrences. `total_spending` reuses the generator-expression-plus-`sum()` pattern from Course 2, Chapter 5 and Course 3's own performance chapter.

Step 5: Deleting an Expense

```
def delete_expense(expenses, index):
    if 0 <= index < len(expenses):
        del expenses[index]
        print("Deleted.")
    else:
        print("Invalid expense number.")
```

Same bounds-checked `del` as Chapter 6's contact book — and same rule for the menu loop below: always `list_expenses()` immediately before asking for a number to delete, exactly the fix Chapter 6's warn-box called for.

🏁 The Complete Expense Tracker

```
import json
from pathlib import Path

FILE = Path("expenses.json")

def load_expenses():
    if not FILE.exists():
        return []
    return json.loads(FILE.read_text())

def save_expenses(expenses):
    FILE.write_text(json.dumps(expenses, indent=2))

def add_expense(expenses):
    try:
        amount = float(input("Amount: "))
    except ValueError:
        print("That's not a valid number — expense not added.")
        return
if amount <= 0:
        print("Amount must be positive — expense not added.")
        return

    category = input("Category: ")
    description = input("Description: ")
    expenses.append({"amount": amount, "category": category, "description": description})
    print("Expense added.")

def list_expenses(expenses):
    if not expenses:
        print("No expenses recorded yet.")
        return
for i, e in enumerate(expenses, start=1):
        print(f"{i}. ${e['amount']:.2f} — {e['category']} — {e['description']}")

def total_by_category(expenses):
    totals = {}
    for e in expenses:
        totals[e["category"]] = totals.get(e["category"], 0) + e["amount"]
    return totals

def total_spending(expenses):
    return sum(e["amount"] for e in expenses)
```

```
def delete_expense(expenses, index):
    if 0 <= index < len(expenses):
        del expenses[index]
        print("Deleted.")
    else:
        print("Invalid expense number.")

expenses = load_expenses()

while True:
    print("\n1. Add  2. List  3. Totals by Category  4. Total Spending  5. Delete  6. Quit")
    choice = input("Choose an option: ")

    if choice == "1":
        add_expense(expenses)
        save_expenses(expenses)
    elif choice == "2":
        list_expenses(expenses)
    elif choice == "3":
        for category, total in total_by_category(expenses).items():
            print(f"{category}: ${total:.2f}")
    elif choice == "4":
        print(f"Total spending: ${total_spending(expenses):.2f}")
    elif choice == "5":
        list_expenses(expenses)
        try:
            num = int(input("Expense number to delete: "))
        except ValueError:
            print("That's not a valid number.")
            continue
        delete_expense(expenses, num - 1)
        save_expenses(expenses)
    elif choice == "6":
        print("Goodbye!")
        break
else:
        print("Not a valid option, try again.")
```

Option 5's own `try`/`except ValueError` around `int(input(...))` reuses `continue` from Course 1, Chapter 4 — skipping straight back to the top of the menu loop rather than letting a bad number crash the whole program, the same protective instinct `add_expense` applied to the amount field.

File I/O

`load_expenses`/`save_expenses` — the same JSON pattern from every chapter since 4.

Functions

Each responsibility — add, list, total, delete — is its own small function.

Dict-based summarization

`total_by_category` reuses Course 1's word-frequency-counter pattern.

Error handling, built in

`try`/`except` around every risky input — not left as an exercise this time.

🎓 Course Complete!

That's all 8 projects of **Python Projects** — a guessing game, calculator, password generator, to-do app, quiz, contact book, web scraper, and this expense tracker capstone — and with it, the complete Python track: Fundamentals → Intermediate → Advanced → Projects, 33 chapters across 4 courses. Every tool from every course ended up here, put to real, practical use.