A Major General’s Guide to Learning Python

For Robert Whittle

Merry Christmas Bob! You said you wanted to learn Python so I figured I'd put together this curriculum real quick for ya.

I didn't have time to test and 100% validate everything, so you may run into some technical hiccups. If that happens feel free to reach out.

This paper guide should be great for taking notes and referencing while you work, but I would highly recommend using the linked 'Google Doc' to help you as you'll need to Ctrl+f, copy paste, and click on links throughout.

https://docs.google.com/document/d/1b3Z1rkctEr84RWRrR188TOF2eHA7oCgcZZgMI3eWpcA/edit?usp=sharing

Getting Python Working

Instructions will probably not be exact as this stuff changes a lot. You can probably figure out most of the little differences, but if you get stuck you can always ask ChatGPT or me for help.

You'll need a few things to get started:

- Your Laptop
- Python 3
- Visual Studio Code

Installing Python 3

1. Open a browser and go to **python.org**

2. Click **Downloads**

3. Download the latest **Python 3 for macOS**

4. Open the .pkg installer

5. Click through the installer using **default options**

6. When finished, close the installer

Verify Installation

1. Open Terminal (⌘ + Space → "Terminal")
2. Type: python3 --version
    a. Expected output: Python 3.12.x (or something similar)

Install Visual Studio Code

This is the software you will be using to actually code.

1. Go to **code.visualstudio.com**
2. Download **VS Code for macOS**
3. Open the .zip
4. Drag **Visual Studio Code** into **Applications**
5. Launch VS Code from Applications

Setup VS Code for Python

1. Open VS Code
2. Click **Extensions** (left sidebar)
3. Search: **Python**
4. Install **Python** (publisher: Microsoft)
5. Restart VS Code if prompted

Establish your Workspace

1. Open **Finder**

Create a folder named: python_training

Open it in VS Code

1. VS Code → **File → Open Folder**

2. Select python_training

3. Click **Open**

First Code Test

1. Create file
    a. In VS Code -> New FIle
    b. Save as cadet_test.py
2. Enter this code

```
print("West Point Cadet reporting for duty.")
```

3. Save the file (⌘ + S)
4. Select python interpreter (this is basically telling VS Code which version of Python to use)
    a. In VS Code, Press ⌘ **+ Shift + P**
    b. Type: Python: Select Interpreter
    c. Choose **Python 3.x**
5. Run the File
    a. In VS Code -> Terminal -> New Terminal
    b. In the opened terminal type: python3 cadet_test.py

Expected output: West Point Cadet reporting for duty.

If you had an issue with setup let Max know and he can sort stuff out for you.

WEST POINT CADET

Module 1 — Basics

Status: Cadet Basic
Objective: Learn the absolute fundamentals of Python by building a real, useful running tool
End State: A working program that calculates pace and prints mile splits
Promotion To: Second Lieutenant (2LT)

As a West Point Cadet, your first responsibility is mastering **fundamentals**.
In Python, that means:

- Giving the computer information
- Performing calculations
- Reporting results clearly

Your first issued tool will calculate **running pace** and generate **mile splits** — something real runners actually use.

By the end of this module, you will understand:

- Variables
- Numbers (int, float)
- User input
- Basic coding math

Inside your python_training folder create: **cadet_pace_calc.py**.

Variables

A **variable** is a named container that holds information.

```
distance_miles = 3
time_minutes = 24
```

Python interprets this as

- "Distance equals 3 miles"
- "Time equals 24 minutes"

User Input

```
distance = input("Enter distance in miles: ")
```

- input() always returns text

If we want to meaningfully use this input as a number we need to convert it from text to a number! We can do this by wrapping distance in an int() or float() function. Example:

```
float(input("Enter distance in miles: "))
```

- Int is short for integer, think whole numbers
- Float is for numbers with decimals

Pace Formula

Pace = time / distance, so let's code that formula.

- Note: # In front of code makes it a **comment**, very important for staying organized and giving context for yourself or other programers
- Note: FIXME always means this code needs attention, useful for learning or reminding yourself to fix something later

Copy and paste the code below into your file on vscode. Fix the pace = line to properly use the variables as they should be utilized to calculate pace.

ISSUED CODE

```
# West Point Cadet - Pace Calculator

# Get user input
distance = float(input("Enter distance in miles: "))
time_minutes = float(input("Enter total time in minutes: "))

# Calculate pace
pace = ??? FIXME: Replace ‘???’ with the correct variables for finding pace

# Display result
print("\n--- PACE REPORT ---")
print("Distance:", distance, "miles")
print("Time:", time_minutes, "minutes")
print("Average pace:", round(pace, 2), "minutes per mile")
```

Once completed run the code by typing in terminal: *python3 cadet_pace_calc.py*

Expect output:

```
Enter distance in miles: 5
Enter total time in minutes: 40

--- PACE REPORT ---
Distance: 5.0 miles
Time: 40.0 minutes
Average pace: 8.0 minutes per mile
```

Loops

A **loop** repeats orders until complete.

- Note: Use <tab> or 4 x <spaces> for indentation

We’ll print **one split per mile.** Add this **while loop** to the bottom of your code:

```
print("\n--- MILE SPLITS ---")

mile = 1
while mile <= int(distance):
    print("Mile", mile, ":", round(pace * mile, 2), "minutes")
    mile = mile + 1
```

Python works a lot like English, where you can read it left to right to understand what is happening.

`while mile <= int(distance)` just means that **while** your **mile** count is less than or equal to the set distance (converted to an integer).

`print("Mile", mile, ":", round(pace * mile, 2), "minutes")` means the mile is printed along with the pace * mile (rounded to 2 decimal places).

`mile = mile + 1` is the most crucial step to making a functional loop. This increases the **mile** count by 1 every time the loop runs, that way it doesn't run forever.

You have just used:

- A loop
- A condition
- Incrementing logic

This is **core Python.**

Cadet PT Test (Required)

You **pass** if you can answer YES to all:

- I can explain what a variable is
- I understand why float() is needed

- I can modify distance/time and get correct output
- I understand how the loop stops

Promotion Board

1. What happens if distance = 0?
2. Why does `input()` return text?
3. What does `mile = mile + 1` do?

(No trick questions. If you can explain it, you understand it.)

Stretch Mission (Optional, but Encouraged)

Add:

- Seconds support (e.g., 25:30 → 25.5 minutes)
- Kilometer option (detect unit)
- Formatted pace like `8:00` instead of `8.0`

Promotion Orders

If your code:

- Runs without errors
- Prints pace correctly

- Prints mile splits

Then:

    🎖️ **PROMOTION ORDERS ISSUED**

SECOND LIEUTENANT (2LT)

Module 2 — Workout Library & Functions

Status: Junior Officer
Objective: Learn how to organize logic into **functions** and **lists**
End State: A reusable workout library the program can call on demand
Promotion To: First Lieutenant (1LT)

Mission Brief

As a 2LT, you stop doing everything yourself.

Instead, you:

- Define **standard procedures**
- Call them when needed
- Avoid rewriting the same code twice

In Python, those are called **functions**.

You will build a **Workout Library** that your future training-plan generator will rely on.

Learning Objectives

By the end of this module, you will understand:

- Functions (`def`)
- Function inputs (parameters)
- Return values

- Lists
- Selecting items from lists

In your python_training folder, create: workout_library.py

Why Functions Exist

Bad approach:

```
print("Easy Run: 4 miles at comfortable pace")
print("Easy Run: 4 miles at comfortable pace")
print("Easy Run: 4 miles at comfortable pace")
```

Lots of repeat lines of code is messy and unnecessary.

Good approach:

```
def easy_run():
    print("Easy Run: 4 miles at comfortable pace")
```

One definition with many uses.

Your first Function

Add this to your code and fix the FIXME:

```
def easy_run():
    print("Easy Run")
    FIXME: Add a print line to display ‘Distance: 4 miles’
    print("Pace: Comfortable, conversational")
```

To execute this function you run:

```
easy_run()
```

Just place that line anywhere below where you made the function. Python has chronological logic so you can’t use functions or variables that are specified after you to use them.

Functions with Inputs

Now we allow user input to make functions way more useful:

```python
def easy_run(distance):
    print("Easy Run")
    print("Distance:", distance, "miles")
    print("Pace: Comfortable")
```

You call this function with input's in the ():

```python
easy_run(5)
```

Let's say you want to use this function but for 3 miles, how would you do that? Play around with the code to figure it out!

Lists

A list stores multiple items in order.

```python
interval_workouts = [
    "6 x 400m fast, 200m jog",
    "4 x 800m at 5K pace",
    "3 x 1 mile at threshold"
]
```

You can store a lot of data in lists. In this example we are just storing 3 strings (text). You can access specific items from a list like this:

```python
print(interval_workouts[0])
```

Indexing starts at 0 in python.

Workout Library

Copy and paste this code into your python file, fixing the issues (FIXME).

```python
# 2LT - Workout Library

def easy_run(distance):
    return f"Easy Run: {distance} miles at comfortable pace"

def tempo_run(distance):
    return f"Tempo Run: {distance} miles at controlled hard pace"

def long_run(FIXME):
    FIXME f"Long Run: {FIXME} miles, slow and steady"

interval_workouts = [
    "6 x 400m fast, 200m jog",
    "4 x 800m at 5K pace",
    "3 x 1 mile at threshold"
]

def interval_workout(choice):
    return interval_workouts[choice]
```

To run this code you'll add this underneath. Change FIXME to select which interval workout you want to do from the list.

```python
print(easy_run(4))
print(tempo_run(5))
print(long_run(10))
print("Intervals:", interval_workout(FIXME))
```

Run the code with: python3 workout_library.py

Expected output:

```
Easy Run: 4 miles at comfortable pace
Tempo Run: 5 miles at controlled hard pace
Long Run: 10 miles, slow and steady
Intervals: 4 x 800m at 5K pace
```

Key Takeaways (Officer-Level)

- `def` defines a procedure
- Inputs make functions flexible
- `return` sends results back to the caller
- Lists store multiple options
- Indexing chooses specific options

This is the foundation of **clean architecture.**

2LT PT Test

You pass if you can:

- Explain what a function does
- Explain the difference between `print()` and `return`
- Change workout distances without breaking code
- Add a new interval workout to the list

Promotion Board Questions

1. Why is `return` better than printing inside a function?
2. What happens if `choice = 5` but the list has only 3 items?
3. Why do lists start at index 0?

Stretch Mission (Optional)

- Add a hill_workout() function
- Randomly select an interval workout
- Add a warm-up and cool-down string

Promotion Orders

If your workout library:

- Uses functions correctly
- Returns strings cleanly
- Runs without errors

Then:

🎖️ **PROMOTION ORDERS ISSUED**

FIRST LIEUTENANT (1LT)

Module 3 — Decision Rules Engine (if / elif / else)

Status: Platoon Leader
Objective: Teach the program how to **make decisions** instead of blindly issuing workouts
End State: A rules engine that adjusts training based on mileage, goals, and constraints
Promotion To: Captain (CPT)

Mission Brief

Up to now, your program **follows orders exactly**.
As a 1LT, your job is to ensure orders **make sense**.

In Python, decisions are made using:

- if
- elif
- else

You will build a **Decision Rules Engine** that prevents bad training decisions and adapts workouts intelligently.

Learning Objectives

By the end of this module, you will understand:

- Boolean logic
- if / elif / else
- Comparison operators

- Guardrails (preventing bad input)
- Simple decision trees

In your python_training folder, create: decision_rules.py

Comparison Operators

Operator	**Meaning**
`>`	greater than
`<`	less than
`>=`	greater or equal
`<=`	less or equal
`==`	equal
`!=`	not equal

```
weekly_miles > 30
```

This evaluates to **True** or **False**.

Basic Decision Structure

```
if weekly_miles < 20:
    print("Low mileage week")
elif weekly_miles < 40:
    print("Moderate mileage week")
else:
    print("High mileage week")
```

Only **one block** executes.

Training Guardrails

Training plans should follow rules:

- Low mileage → fewer hard workouts
- High mileage → more caution
- Different goals require different emphasis

Knowing this let’s make a decision rules engine:

```
# 1LT - Decision Rules Engine

def classify_mileage(weekly_miles):
    if weekly_miles < 20:
        return "low"
    elif weekly_miles < 40:
        return "moderate"
    else:
        return "high"

def speed_work_allowed(weekly_miles):
    if weekly_miles < 15:
        return False
    else:
        return True

def recommend_focus(goal_race):
    if goal_race == "5k":
        return "speed"
    elif goal_race == "10k":
        return "balanced"
    elif goal_race == "half":
        return "endurance"
    FIXME: add an elif statement for the ‘marathon’
    else:
        return "general_fitness"
```

To run your code, add this below the functions:

```
miles = FIXME: set how many miles you’re running
goal = "half"

print("Mileage class:", classify_mileage(miles))
print("Speed work allowed:", speed_work_allowed(miles))
print("Training focus:", recommend_focus(goal))
```

Expected output:

```
Mileage class: moderate
Speed work allowed: True
Training focus: endurance
```

Officer-Level Understanding

What you just built:

- A **decision layer**
- Clear, readable logic
- Guardrails that prevent bad plans

This is the difference between:

> "The program runs"
> and
> "The program thinks."

1LT PT Test

You pass if you can:

- Explain how if / elif / else works
- Modify mileage thresholds safely
- Add a new race type (e.g., 15K)
- Explain why speed_work_allowed() returns False sometimes

Promotion Board Questions

1. Why should low mileage runners limit speed work?
2. What happens if multiple if conditions are true?
3. Why is an else useful?

Stretch Mission (Optional)

- Add a rule for **age-based caution**
- Add logic for **days per week available**
- Print warnings for risky combinations

Promotion Orders

If your rules engine:

- Makes correct decisions
- Uses clean conditional logic
- Runs without errors

Then:

> 🎖️ **PROMOTION ORDERS ISSUED**

CAPTAIN (CPT)

Module 4 — Weekly Plan Generator (Loops + Integration)

Status: Company Commander
Objective: Combine workouts + decision rules into a **full 7-day training plan**
End State: A program that generates a realistic week of training
Promotion To: Major (MAJ)

Mission Brief

As a Captain, you are no longer issuing isolated orders.
You are responsible for **coordinated execution across time**.

In Python terms:

- You will **loop across days**
- Apply **decision rules**
- Assign the **right workout to the right day**
- Ensure balance: hard days, easy days, rest

This is where your program starts to look like a **real system**.

Learning Objectives

By the end of this module, you will understand:

- `for` loops
- Building and appending to lists
- Integrating multiple files (imports)

- Simple scheduling logic
- Avoiding consecutive hard days

In your python_training folder, create:weekly_plan.py

Import Your Existing Units

Python is really cool in that you don't have to have all your work in one file. Often it makes more sense to split up your code across different locations. At the top of weekly_plan.py, add:

```
from workout_library import easy_run, tempo_run, long_run, interval_workout
from decision_rules import speed_work_allowed, recommend_focus
```

These two lines grab all the functions you've made so far so we can use them here.

Controlled March

A for loop repeats a fixed number of times.

```
for day in range(1, 8):
    print("Day", day)
```

This will run **exactly 7 times**. We no longer run the script until something is done, the number of times looped is preset. I actually prefer 'for loops' as they feel more predictable.

The Weekly Plan

We will store each day's workout in an empty list:

```
week_plan = []
```

Each loop iteration will add **one day**.

Weekly Plan Generator

```
# CPT - Weekly Plan Generator
# Import these functions you made if you haven’t already
from workout_library import easy_run, tempo_run, long_run, interval_workout
from decision_rules import speed_work_allowed, recommend_focus

def generate_weekly_plan(weekly_miles, goal_race): # miles/goals as input
    plan = []
    focus = recommend_focus(goal_race)
    speed_ok = speed_work_allowed(weekly_miles)

    for day in range(1, 8):
        if day == 1:
            plan.append("Rest Day")

        elif day == 2:
            if speed_ok:
                plan.append(interval_workout(0))
            else:
                plan.append(easy_run(4))

        elif day == 3:
            plan.append(easy_run(5))

        elif day == 4:
            if focus == "speed":
                plan.append(tempo_run(4))
            else:
                plan.append(easy_run(4))

        elif day == 5:
            plan.append("Rest or Cross-Training")

        elif day == 6:
            plan.append(long_run(weekly_miles * 0.3))

        else:
            FIXME: what would you want to happen on day 7?
    return plan
```

This may seem like a lot but just look at it in sections and it is only what you've learned so far.

- `Plan.append` simply appends an item to the empty list you made

```
week = generate_weekly_plan(30, "half")

print("\n--- WEEKLY TRAINING PLAN ---")
for day, workout in enumerate(week, start=1):
    print("Day", day, ":", workout)
```

To run this function simply put in terminal: *python3 weekly_plan.py*

Expected output:

```
--- WEEKLY TRAINING PLAN ---
Day 1 : Rest Day
Day 2 : 6 x 400m fast, 200m jog
Day 3 : Easy Run: 5 miles at comfortable pace
Day 4 : Easy Run: 4 miles at comfortable pace
Day 5 : Rest or Cross-Training
Day 6 : Long Run: 9.0 miles, slow and steady
Day 7 : Easy Run: 3 miles at comfortable pace
```

Captain-Level Insight

What you accomplished:

- Coordinated **multiple components**
- Controlled flow with for + if
- Built a **coherent schedule**
- Returned structured data instead of printing chaos

This is **real programming**, not a toy example.

CPT Command Inspection (Required)

You pass if you can:

- Explain why range(1, 8) is used
- Change the long-run percentage safely
- Move rest days without breaking logic
- Adjust workouts for a 5K goal

Promotion Board Questions

1. Why do we avoid hard workouts on back-to-back days?
2. What happens if weekly_miles = 10?
3. Why return plan instead of printing inside the function?

Stretch Mission (Optional)

- Randomize which day speed work occurs
- Ensure **never** more than 2 hard days per week
- Add labels like “Quality Day” vs “Recovery”

Promotion Orders

If your weekly plan:

- Generates 7 days correctly
- Uses loops cleanly
- Integrates rules + workouts

Then:

> 🎖️ **PROMOTION ORDERS ISSUED**

MAJOR (MAJ)

Module 5 — Multi-Week Progression (Nested Loops + Cutback Weeks)

Status: Field Grade
Objective: Generate an **8–16 week plan** that progresses intelligently
End State: A multi-week plan with progression + cutbacks + a taper
Promotion To: Lieutenant Colonel (LTC

Mission Brief

A single week is tactics.
A full plan is operations.

Your program must now:

- Build week-by-week
- Increase workload safely
- Insert cutback weeks to reduce injury risk
- Taper near race day

This is where nested loops and planning logic matter.

I will say this is where the program starts to get quite a bit harder. Feel free to take your time with these as the amount of time needed to fully grasp each topic will increase.

Learning Objectives

By the end of this module, you will understand:

- Nested loops (weeks + days)

- Lists of weeks (2D lists)
- Progressive overload rules
- Cutback weeks (deload)
- Taper logic (reduce before race)

Create: **plan_builder.py**

Design Rules

We'll keep rules intentionally simple:

1. **Long run** starts at ~25–30% of weekly miles
2. Weekly miles increase by **~5–10%** most weeks
3. Every **4th week is a cutback** (reduce ~15–20%)
4. Final **2 weeks taper** (reduce volume)

The following can be a lot so don't be afraid to ask questions. ChatGPT is really really good at helping you understand code. Just copy and paste these instructions or a screenshot for help. Paste the following into plan_builder.py:

```
# MAJ - Multi-Week Plan Builder

from weekly_plan import generate_weekly_plan

def apply_cutback(weekly_miles):
    # reduce volume to recover
    return weekly_miles * 0.80

def apply_build(weekly_miles, build_rate=0.08):
    # increase volume for progression
    return weekly_miles * (1 + build_rate)

FIXME: Create an apply_taper -> input = weekly_miles, taper_factor
def ???(???, ???):
    return ??? * ???

def build_plan(start_weekly_miles, goal_race, weeks=12):
    all_weeks = []
    weekly_miles = float(start_weekly_miles)

    for week_num in range(1, weeks + 1):
        # Cutback every 4th week (except if it's in taper)
        in_taper = (week_num >= weeks - 1)

        if in_taper:
            # two-week taper
            if week_num == weeks - 1:
                weekly_miles = apply_taper(weekly_miles, 0.70)
            else:
                weekly_miles = apply_taper(weekly_miles, 0.50)

        elif week_num % 4 == 0:
            weekly_miles = apply_cutback(weekly_miles)

        else:
            # normal build week (except week 1)
            if week_num != 1:
                weekly_miles = apply_build(weekly_miles, build_rate=0.08)

        week_plan = generate_weekly_plan(weekly_miles, goal_race)
        all_weeks.append({
            "week": week_num,
            "weekly_miles": round(weekly_miles, 1),
            "days": week_plan
        })

    return all_weeks
```

After addressing the FIXME, add this below the function:

```
plan = build_plan(start_weekly_miles=25, goal_race="half", weeks=12)

for w in plan:
    print("\n==========================")
    print("WEEK", w["week"], "| Target Miles:", w["weekly_miles"])
    print("==========================")
    for day_num, workout in enumerate(w["days"], start=1):
        print("Day", day_num, ":", workout)
```

To execute run: **python3 plan_builder.py**

You should see 12 weeks, each with:

- A target weekly mileage
- A full 7-day schedule

MAJ-Level Insight: Nested Loops

- Outer loop = weeks
- Inner loop (inside generate_weekly_plan) = days
- Output becomes a “plan of plans”

That’s a major jump in capability.

MAJ Command Inspection

You pass if you can:

- Change weeks=12 to weeks=8 and it still works
- Explain what week_num % 4 == 0 means
- Identify which weeks are cutback weeks in your output
- Confirm taper happens in the final 2 weeks

Promotion Board Questions

1. Why do cutback weeks reduce injury risk?
2. Why do we taper near the end?
3. If progression is 8%, what happens over many weeks?

Stretch Missions (Optional)

1. Cap weekly increases

- If a build week would increase more than +5 miles, cap it.

2. Goal-specific taper

- Marathon taper longer than 5K taper.

3. Smarter long run

- Increase long run more slowly than total miles.

Promotion Orders

If your multi-week plan:

- Builds week-by-week
- Includes cutbacks
- Includes a taper

- Runs without errors

Then:

> 🎖️ **PROMOTION ORDERS ISSUED**

LIEUTENANT COLONEL (LTC)

Module 6 — Data Persistence & Validation (File I/O)

Status: Battalion Commander
Objective: Make plans persist, reload, and validate inputs like a real system
End State: Save/load training plans to files; prevent bad inputs from breaking the program
Promotion To: Colonel (COL)

Mission Brief

Up to now, your plan exists only while the program runs.
Real operations require:

- Records
- Accountability
- Reusability

As an LTC, you'll ensure:

- Plans are saved to disk
- Plans can be loaded later
- Inputs are checked before execution

This is the line between *script* and *software*.

Learning Objectives

By the end of this module, you will understand:

- Reading and writing files

- CSV format
- Basic input validation
- Defensive programming
- Separating “data” from “logic”

Create: **export_plan.py** and **validate_inputs.py**

Why CSV?

CSV (Comma-Separated Values):

- Human-readable
- Excel-compatible
- Easy for Python to handle

CSV Example:

```
Week,Day,Workout
1,1,Rest Day
1,2,Easy Run: 4 miles
```

Input Validation (Command Discipline)

Bad inputs destroy plans:

- Negative mileage
- Invalid race types
- Too many training days

We stop bad orders **before** execution.

Input Validation

Paste into validate_inputs.py:

```
# LTC - Input Validation

def validate_weekly_miles(miles):
    if miles <= 0:
        raise ValueError("Weekly mileage must be positive.")
    if miles > 120:
        raise ValueError("Weekly mileage exceeds safe limits.")
    return True


def validate_goal(goal):
    valid_goals = ["5k", "10k", "half", "marathon"]
    if goal not in valid_goals:
        raise ValueError(f"Goal must be one of {valid_goals}.")
    return True
```

Export & Load Plan

```
# LTC - Export and Load Training Plans

import csv


def export_plan_to_csv(plan, filename="training_plan.csv"):
    with open(filename, mode="w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["Week", "Day", "Workout"])

        for week in plan:
            week_num = week["week"]
            for day_num, workout in enumerate(week["days"], start=1):
                writer.writerow([week_num, day_num, workout])


def load_plan_from_csv(filename="training_plan.csv"):
    plan = []
    with open(filename, mode="r") as file:
        reader = csv.DictReader(file)
        for row in reader:
            plan.append(row)
    return plan
```

Mission

Mission (Correct Test Method)

Create a new file named: `test_export.py` (in the same folder as your old scripts *before* refactor, or in the parent folder of `pt_planner/` after refactor).

```python
from plan_builder import build_plan
from export_plan import export_plan_to_csv
from validate_inputs import validate_weekly_miles, validate_goal

def main():
    start_miles = 25
    goal = "half"

    validate_weekly_miles(start_miles)
    validate_goal(goal)

    plan = build_plan(start_miles, goal, weeks=8)
    export_plan_to_csv(plan, filename="training_plan.csv")

    print("Plan exported successfully to training_plan.csv")

if __name__ == "__main__":
    main()
```

Run **python3 test_export.py**

You should see a **training_plan.csv** was generated in your project folder. Open this and take a look to see if everything looks alright.

LTC-Level Insight

What you achieved:

- **Persistence**: plans survive program shutdown

- **Validation**: unsafe inputs stopped early
- **Interoperability:** CSV works outside Python

This is the core of **production readiness.**

LTC Command Inspection (Required)

You pass if you can:

- Open the CSV in Excel
- Reload the CSV without errors
- Trigger a validation error intentionally
- Explain why validation happens *before* plan generation

Promotion Board Questions

1. Why raise errors instead of printing warnings?
2. Why is CSV better than plain text here?
3. What could go wrong without validation?

Stretch Missions (Optional)

- Add date fields to the CSV

- Add a notes column
- Validate **days per week available**
- Write a simple summary report from the loaded CSV

Promotion Orders

If your system:

- Saves plans correctly
- Loads data safely
- Rejects bad inputs

Then:

🎖️ **PROMOTION ORDERS ISSUED**

COLONEL (COL)

Module 7 — Refactor, Standards, & Command Structure

Status: Senior Commander
Objective: Turn a working system into a **maintainable, professional codebase**
End State: Clean structure, documented functions, predictable behavior
Promotion To: Brigadier General (BG)

Mission Brief

As a Colonel, your responsibility is **not writing more code**.
It is ensuring:

- Clarity
- Discipline
- Sustainability

The system must:

- Be understandable months later
- Be usable by someone else
- Follow clear standards

This module is about **refactoring** and **command structure**.

Learning Objectives

By the end of this module, you will understand:

- Folder-based project structure
- Modules vs scripts
- Docstrings (documentation)
- `__main__` execution pattern
- Minimal self-testing

Final Command Structure

```
pt_planner/
├── __main__.py
├── workouts.py
├── rules.py
├── weekly.py
├── builder.py
├── export.py
├── validate.py
└── README.md
```

Create an empty file named `__init__.py` inside this `pt_planner/` folder.

File mapping (old → new)

Old file	New file
workout_library.py	workouts.py
decision_rules.py	rules.py

weekly_plan.py	weekly.py
plan_builder.py	builder.py
export_plan.py	export.py
validate_inputs.py	validate.py

Since we're renaming some files we have to adjust some imports in past files as well.

In weekly.py, change:

```
from workout_library import easy_run, tempo_run, long_run, interval_workout
from decision_rules import speed_work_allowed, recommend_focus
```

to

```
from .workouts import easy_run, tempo_run, long_run, interval_workout
from .rules import speed_work_allowed, recommend_focus
```

In builder.py, change:

```
from weekly_plan import generate_weekly_plan
```

to

```
from .weekly import generate_weekly_plan
```

In test_export.py, change:

```
from plan_builder import build_plan
from export_plan import export_plan_to_csv
from validate_inputs import validate_weekly_miles, validate_goal
```

to

```
from pt_planner.builder import build_plan
from pt_planner.export import export_plan_to_csv
from pt_planner.validate import validate_weekly_miles, validate_goal
```

If `test_export.py` is *inside* `pt_planner/` as `pt_planner/test_export.py`, use relative imports (i.e. from .builder import build_plan)

Modules vs Scripts

- **Module:** defines functions
- **Script:** runs the program

Only **one file** should “run” things:

```
__main__.py
```

Paste this into that file:

```
# COL - Main Command Entry

from .builder import build_plan
from .export import export_plan_to_csv
from .validate import validate_weekly_miles, validate_goal


def main():
    start_miles = 25
    goal = "half"
    weeks = 12

    validate_weekly_miles(start_miles)
    validate_goal(goal)

    plan = build_plan(start_miles, goal, weeks)
    export_plan_to_csv(plan)

    print("Training plan generated and exported successfully.")
```

```python


if __name__ == "__main__":
    main()
```

`if __name__ == "__main__":` is essentially a guaranteed if statement that will run the main() function.

Run the entire system with:

```
python3 -m pt_planner
```

Docstrings

When creating functions it's good practice (basically required) to add docstrings to functions. For example:

```python
def easy_run(distance):
    """
    Generates an easy run description.

    Args:
        distance (float): Distance in miles.

    Returns:
        str: Workout description.
    """
```

This is the common formatting used. Anything between the quotations acts as a comment explaining what the function does.

COL Command Inspection

You pass if:

- `python3 -m pt_planner` runs cleanly
- Every function has a docstring

- No file prints output on import
- Folder structure matches diagram

Promotion Board Questions

1. Why should only `__main__.py` execute code?
2. Why is refactoring safer than adding features?
3. What breaks if logic and execution mix?

Stretch Missions (Optional)

- Add a config file for user settings
- Add logging instead of printing
- Add a version number to the tool

Promotion Orders

If your system:

- Is cleanly structured
- Well-documented
- Easy to understand

Then:

🎖️ **PROMOTION ORDERS ISSUED**

BRIGADIER GENERAL (BG)

Module 8 — Smart Adjustments & Risk Control

Status: General Officer (1-Star)
Objective: Add **adaptive intelligence** so the system responds to reality
End State: The planner detects risk, adjusts weeks dynamically, and flags problems
Promotion To: Major General (MG) (Final)

Mission Brief

At this level, you no longer ask:

> “Does the plan run?”

You ask:

> “Does the plan make sense **when things go wrong**?”

Real runners:

- Miss days
- Accumulate fatigue
- Increase mileage too fast

Your system must now:

- **Detect risk**
- **Adjust intelligently**
- **Warn the user before failure**

This is where your planner becomes **decision support**, not just automation.

Learning Objectives

By the end of this module, you will understand:

- Simple scoring systems
- Rate-of-change logic
- Flags & warnings
- Adaptive adjustments
- Separating *recommendation* from *execution*

Create: **adaptive.py**

Ramp Rate (Injury Risk)

Ramp rate = how fast mileage increases week-to-week.

Rule of thumb:

- 10% increase → elevated risk

We encode this in adaptive.py:

```python
# BG - Adaptive Risk Analysis

def calculate_ramp_rate(prev_miles, current_miles):
    """
    Calculates week-to-week mileage increase percentage.
    """
    if prev_miles == 0:
        return 0
    return (current_miles - prev_miles) / prev_miles


def ramp_rate_warning(prev_miles, current_miles):
    rate = calculate_ramp_rate(prev_miles, current_miles)
    if rate > 0.10:
        return True, round(rate * 100, 1)
    else:
        return False, round(rate * 100, 1)
```

Fatigue Score

We assign **points:**

- Easy run = 1
- Tempo / Intervals = 3
- Long run = 4
- Rest = 0

This gives a **fatigue score per week.** Add this below ramp functions:

```
def fatigue_score(week_plan):
    score = 0

    for workout in week_plan:
        if workout is None:
            continue
        text = workout.lower()

        if "rest" in text:
            score += 0
        elif "interval" in text or "tempo" in text:
            score += 3
        elif "long run" in text:
            FIXME: the score should increase by 4
        else:
            score += 1

    return score
```

Adaptive Adjustment

If:

- Ramp rate too high **or**
- Fatigue score too high

→ Automatically adjust next week. Add this code to your file:

```python
def adaptive_adjustment(prev_week, current_week):
    warnings = []

    ramp_flag, ramp_pct = ramp_rate_warning(
        prev_week["weekly_miles"],
        current_week["weekly_miles"]
    )

    fatigue = fatigue_score(current_week["days"])

    if ramp_flag:
        warnings.append(f"Ramp rate high: {ramp_pct}%")

    if fatigue > 15:
        warnings.append(f"High fatigue score: {fatigue}")

    return warnings
```

Warnings are powerful, allowing for proper debugging and project management. The 'f' in those append statements allow for the use of variables in print statements a little easier (e.g. `f"High fatigue score: {fatigue}"`)

Integration Test

Add this to the end of your code:

```python
if __name__ == "__main__":
    from .builder import build_plan

    plan = build_plan(25, "half", weeks=6)
```

```python
for i in range(1, len(plan)):
    warnings = adaptive_adjustment(plan[i-1], plan[i])
    if warnings:
        print(f"\nWARNING - WEEK {plan[i]['week']} WARNINGS:")
        for w in warnings:
            print("-", w)
```

Run: **python3 -m pt_planner.adaptive**

You should see warnings only when justified, not constantly.

BG-Level Insight

What you've added:

- **Metrics** (ramp rate, fatigue)
- **Evaluation logic**
- **Human-readable warnings**

This is how real systems assist humans **without replacing judgment**.

BG Command Inspection (Required)

You pass if you can:

- Explain ramp rate in plain language
- Adjust fatigue thresholds safely
- Trigger and suppress warnings intentionally
- Explain why warnings ≠ automatic failure

Promotion Board Questions

1. Why not automatically cancel workouts when fatigue is high?
2. Why are heuristics better than rigid rules here?
3. What happens if a system over-warns?

Stretch Missions (Optional)

- Lower fatigue if a rest day follows
- Flag **two hard days in a row**
- Adjust next week's mileage automatically
- Print a "Commander's Risk Summary"

Promotion Orders

If your system:

- Detects risk intelligently
- Warns without spamming
- Integrates cleanly with the plan builder

Then:

⭐ **PROMOTION ORDERS AUTHORIZED**

MAJOR GENERAL (MG)

Capstone — Strategic Command Tool

Status: Senior General Officer
Mission Complete: You have built a **real, end-to-end software system** using Python fundamentals—no shortcuts, no magic.

This is the final rank. What follows is your **After Action Review**, what you've truly learned, and how this translates beyond running or Python.

WHAT YOU BUILT

You now command a system that can:

- Accept validated inputs
- Generate weekly and multi-week training plans
- Apply progression, cutbacks, and tapering
- Persist plans to disk (CSV)
- Reload data
- Detect injury risk via ramp rate
- Measure fatigue
- Issue warnings instead of blindly executing
- Run as a clean command-line tool

That is **not beginner Python**.
That is **applied systems thinking**.

WHAT YOU ACTUALLY LEARNED

Python Fundamentals (Fully Mastered)

- Variables & types
- Functions & returns
- Loops (simple + nested)
- Conditional logic
- Lists, dictionaries
- File I/O
- Imports & modules
- Error handling via validation
- Project structure
- Execution control (`__main__`)

Engineering Thinking

- Separation of concerns
- Guardrails before execution
- Adaptation instead of rigidity
- Metrics before decisions
- Warnings over automation
- Refactoring for clarity

This is **how professionals think**, not just how beginners code.

Rank	**Skill Acquired**
Cadet	Core syntax & math
2LT	Functions & reuse
1LT	Decision logic
CPT	Weekly scheduling
MAJ	Multi-week operations
LTC	Persistence & validation
COL	Structure & standards
BG	Adaptive intelligence
MG	Strategic system design

WHERE THIS GOES NEXT

If you ever want to extend this:

Option 1 — Visualization

- Weekly mileage charts
- Fatigue graphs

Option 2 — Real Data

- Import GPS or running logs
- Adjust plans based on performance

Option 3 — Broader Domains

This same structure applies to:

- Budget planning
- Logistics scheduling
- Maintenance cycles
- Training pipelines
- Risk assessment tools

FINAL ORDERS

You are no longer “learning Python.”

You now:

- Understand how programs are *designed*
- Can read and reason about real code
- Can modify systems safely
- Can explain *why* logic exists

That is the difference between:

> “I followed a tutorial”
> and
> “I can build tools.”

Nice job Bob :)