

PRACTICE SHEET

What you will learn:

AI and LLM-Driven Programming(prompt design, Text-to-Speech using gTTS, OpenAI's ChatCompletion, DALL·E image generation), Math & Scientific Computation, GUI & Visualization(Tkinter and matplotlib)

Use of libraries (NumPy, SymPy, etc.) allowed where noted.

For OpenAI API Key details, see Question 14.

[EASY] Q1) Which of the following prompts is **least likely** to generate accurate code using an LLM like GPT or Copilot?

- A. "Write Python code to simulate an RLC circuit using numerical integration."
 - B. "Implement a basic stack using list."
 - C. "Make me something cool."
 - D. "Find eigenvalues of a matrix using NumPy."
-

[EASY] Q2) Consider a 3x3 matrix with non-zero determinant. What's the **time complexity** of solving the system using naive Gaussian Elimination?

- A. $O(n^2)$
 - B. $O(n^3)$
 - C. $O(\log n)$
 - D. $O(n)$
-

[EASY] Q3) Which of the following tasks is **currently least accurate** when solved using GitHub Copilot or LLMs **without human review**?

- A. Sorting a list
 - B. Implementing Newton-Raphson for root finding
 - C. Writing parallel code for stress analysis in a bridge
 - D. Reading a CSV file in pandas
-

[MEDIUM] Q4) Implement a function using **sets and dictionaries** that maps a list of symbolic expressions to their derivatives using **sympy**.

Input: ["x**2", "sin(x)", "e**x"]

Output: { "x**2": "2*x", "sin(x)": "cos(x)", ... }

Hint :

```
from sympy import symbols, diff, sin, exp
```

```
from sympy.parsing.sympy_parser import parse_expr
```

[MEDIUM] Q5) In a cutting-edge **chemical engineering lab**, you're working on a **continuous stirred-tank reactor (CSTR)**. Inside this reactor, three reactants X, Y and Z are interacting under a set of carefully controlled reactions.

After hours of experimenting, you've discovered that the system follows a **linear set of balance equations** based on reaction stoichiometry, flow rates, and conversions. Your job is to **calculate the steady-state concentrations** of the reactants in the reactor.

The equations governing the concentrations are:

$$2x + y - z = 1 \quad (\text{Equation from Reaction A})$$

$$-x + 3y + 2z = 12 \quad (\text{Equation from Reaction B})$$

$$3x - y + z = 3 \quad (\text{Equation from Reaction C})$$

Where:

- x = concentration of Reactant X
- y = concentration of Reactant Y
- z = concentration of Reactant Z

To make your lab results reproducible and efficient, you decide to solve this system using **NumPy in Python**.

[EASY]Q6) You are given a grayscale image as a 2D NumPy array. What will `image[::-1]` do?

- A. Flip the image left to right
- B. Flip the image top to bottom

- C. Rotate the image 180 degrees
 - D. Do nothing
-

[MEDIUM] Q7) You are helping design a **smart bridge** that automatically adjusts the placement of **3 heavy loads** (boxes) on a **beam to maintain balance**. The beam is modeled as a 1D line from 0 to 10 meters. You have **three loads** (say, 30kg, 20kg, and 10kg).

The system finds the **center of mass (COM)** of these weights and tells you if the beam will **tip left**, **stay balanced**, or **tip right**. Your task is to **enter load positions** and **check stability** using a GUI app.

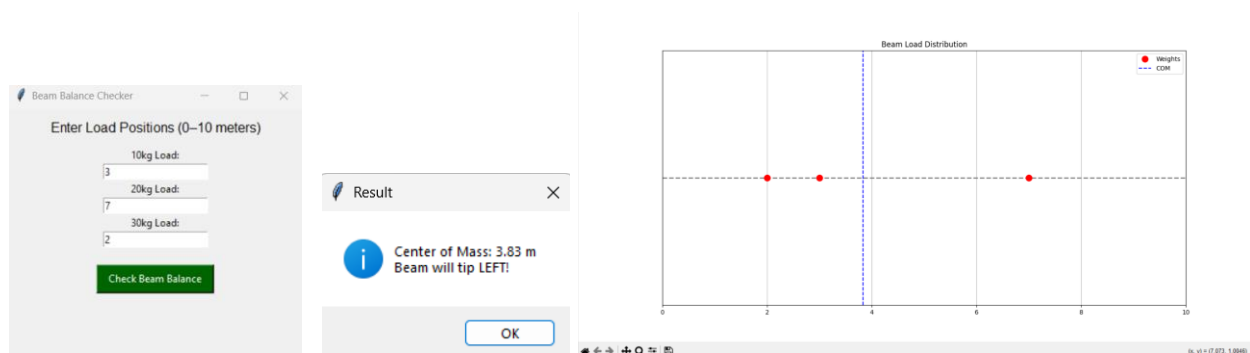
If the COM is:

- $< 5 \rightarrow$ Beam tips left
- $= 5 \rightarrow$ Balanced
- $5 \rightarrow$ Beam tips right

Use the **center of mass** formula: $COM = \frac{\sum m_i x_i}{\sum m_i}$

Where:

- m_i = mass of object i
- x_i = position of object i on beam



Hints:

This question uses Tkinter, which is Python's standard library for building GUI (Graphical User Interface) apps. The goal is to let the user input the positions of 3 weights on a beam and check if the beam stays balanced.

You'll use basic physics (Center of Mass) and visualize the setup using Matplotlib.

import tkinter as tk

```

from tkinter import _____ # Fill up: Show result messages (Hint: messagebox)

import numpy as np

import matplotlib.pyplot as plt

def check_balance():

    try:

        # Get user input from Entry fields and convert to float

        pos1 = float(_____.get()) # Fill: Get position of first load

        pos2 = float(_____.get()) # Fill: Get position of second load

        pos3 = float(_____.get()) # Fill: Get position of third load

        # Define weights

        masses = np.array([10, 20, 30])

        positions = np.array([pos1, pos2, pos3])

        # Calculate center of mass

        com = np.sum(_____ * _____) / np.sum(_____) # Fill: Use masses and positions

        # Determine balance status

        if com < 5:

            status = "Beam will tip LEFT!"

        elif com > 5:

            status = "Beam will tip RIGHT!"

        else:

            status = "Beam is perfectly BALANCED!"

        _____.showinfo("Result", f"Center of Mass: {com:.2f} m\n{status}") # Fill: show result
box

        # Plot the beam and weights

        plt.figure(figsize=(8, 1))

        plt.hlines(1, 0, 10, colors='gray', linestyle='dashed', linewidth=2)

```

```

plt.plot(positions, [1]*3, 'ro', markersize=10, label="Weights")
plt.axvline(com, color='blue', linestyle='--', label='Center of Mass')
plt.xlim(0, 10)
plt.yticks([])
plt.grid(True, axis='x')
plt.legend()
plt.title("Beam Load Distribution")
plt.show()
except ValueError:
    _____.showerror("Input Error", "Please enter valid numbers.") # Fill: error message box

# --- GUI SETUP ---
root = tk.Tk()
root.title("Beam Balance Checker")
# Create input fields
tk.Label(root, text="Enter Load Positions (0–10 meters)").pack()
tk.Label(root, text="10kg Load:").pack()
entry1 = tk.Entry(root)
entry1.pack()
tk.Label(root, text="20kg Load:").pack()
entry2 = tk.Entry(root)
entry2.pack()
tk.Label(root, text="30kg Load:").pack()
entry3 = tk.Entry(root)
entry3.pack()
# Button to trigger solution
tk.Button(root, text="Check Beam Balance", command=check_balance).pack(pady=20)

```

```
root.mainloop()
```

[EASY] Q8) Imagine you are building a **smart lab assistant** that can **speak instructions or announcements** aloud. Your first goal is to write a Python program that takes **any text input** and converts it to **human-like speech** using AI (Text-to-Speech).

Later, this could be integrated into systems like:

- Smart notice boards
- Lab safety voice guides
- Voice-enabled IoT systems

Let's start by building the **base module** that converts text to voice.

Hint:

- Use the **gTTS (Google Text-to-Speech)** library
- It can **convert any text** to an MP3 voice file in any supported language
- Use `os.system()` to **play the audio**

Starter Code:

```
from gtts import _____ # Fill: Import the text-to-speech function
import os

# Step 1: Input sentence from user
text = input("Enter the sentence to speak: ")

# Step 2: Convert text to speech (language: English)
tts = gTTS(_____, lang='en') # Fill: text input

# Step 3: Save the voice output to a file
tts._____("voice_output.mp3") # Fill: Save as MP3 file

# Step 4: Play the audio (works on Windows)
os.system("_____ voice_output.mp3") # Fill: command to play
```

[MEDIUM] Q9) As an engineering student, you often need quick explanations, formulas, or examples. Wouldn't it be cool to have a **personal AI assistant** that can explain topics in plain English?

Your goal is to build a **simple chatbot using OpenAI's GPT model**, which takes a question from the user and responds intelligently. Build a smart Q&A assistant, research helper, or even a subject-specific bot.

Hint:

- Use `openai.ChatCompletion.create()` to generate a response
- Set a **system role** like "You're a helpful engineer" to shape the chatbot's behavior
- This can be expanded into a **Tkinter GUI chatbot** or **voice-interactive bot**

Starter Code:

```
import openai

import os

from dotenv import load_dotenv

load_dotenv()

openai.api_key = os.getenv("_____") # Fill: Load API key

# Step 1: Get user question

user_prompt = input("Ask your AI assistant a question: ")

# Step 2: Create chat with system and user messages

response = openai.ChatCompletion.create(

    model="_____", # Fill: Model name

    messages=[

        {"role": "system", "content": "You're a helpful engineering assistant."},

        {"role": "user", "content": user_prompt}

    ]

)

# Step 3: Print AI reply
```

```
print("AI:", response['choices'][0]['message']['_____']) # Fill: where to find response
```

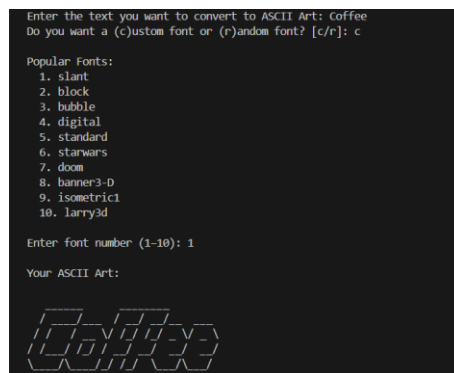
[EASY] Q10) You're participating in a college tech fest, and your club needs to generate eye-catching ASCII banners for its terminal-based event booth.

Instead of boring plain text like "Welcome" or "ROBOTICS CLUB", you decide to build a **Text-to-ASCII Art generator** in Python that turns text into stylish ASCII logos, using the magic of the pyfiglet library.

You'll let the user choose from popular banner styles, like block, bubble, slant, starwars, doom, etc, or even better: let the system **randomly pick a cool font** each time for surprise.

What is pyfiglet?

- pyfiglet is a Python port of the classic **Figlet** tool.
- It lets you turn **simple strings into ASCII art banners** using different pre-designed font styles.



```
Enter the text you want to convert to ASCII Art: coffee
Do you want a (c)ustom font or (r)andom font? [c/r]: c

Popular Fonts:
1. slant
2. block
3. bubble
4. digital
5. standard
6. starwars
7. doom
8. banner3-D
9. isometric1
10. larry3d

Enter font number (1-10): 1

Your ASCII Art:
  COFFEE
```

Challenge Goal:

Build a Python program that:

1. Takes user text input (e.g., Welcome to AI)
2. Lets the user pick a font from a menu (or go random!)
3. Displays the ASCII art banner in the terminal

Starter code:

```
from pyfiglet import _____
```

```
import random
```

```
# Step 1: Take text input from the user
```



```

text = input("Enter text to convert into ASCII art: ")

# Step 2: Ask for font mode

mode = input("Choose font mode - custom (c) or random (r): ").strip().lower()

figlet = Figlet()

# Step 3: Define popular font choices

fonts = {
    "1": "slant",
    "2": "block",
    "3": "bubble",
    "4": "digital",
    "5": "standard",
}

if mode == 'c':
    print("\nAvailable Fonts:")
    for num, name in fonts.items():
        print(f"{num}. {name}")
    choice = input("Pick a font number: ")
    font_name = fonts.get(choice)
    if font_name:
        figlet.setFont(font=font_name) # Fill: font_name
    else:
        print("Invalid choice. Using default.")
else:
    figlet.setFont(font=random.choice(figlet.getFonts()))

# Step 4: Render the text in selected font

ascii_output = figlet.renderText(text) # Fill: render method

```

```
print("\nHere is your ASCII banner:\n")  
  
print(ascii_output)
```

[MEDIUM] Q11) Imagine you're a game developer or a creative designer. You want to describe your idea **out loud** and have an AI **visualize it** in real-time — just like telling an artist what to draw.

Your job is to build an AI pipeline named "**Talk to Paint**", that:

1. Takes **your voice input**
2. Transcribes it using **Whisper**
3. Sends that transcription to **DALL·E** to create an image
4. Shows or prints the image URL

Use this to design: fantasy worlds, futuristic gadgets, surreal scenes - **just by speaking**.

Example Use Cases:

- Say "A robot watering a bonsai plant on Mars"
- Say "A futuristic classroom floating in space"
- Say "A cat surfing a rainbow wave"

Hint:

Step 1: Set Up Environment

- Use python-dotenv to load your API key securely.
- Create a .env file like:

```
OPENAI_API_KEY=your-key-here
```

Step 2: Transcribe Audio (Whisper)

- Use `openai.Audio.transcribe()`
- Pass your audio file (e.g., `my_voice.mp3`) in binary mode
- Choose model: "whisper-1"
- Extract the "text" from the response

Step 3: Generate Image (DALL·E)

- Use `openai.Image.create()`
- Pass the transcribed text as the prompt
- Choose size: "512x512"
- Extract the image URL from `response["data"][0]["url"]`

Final Assembly:

- Define two functions:
`transcribe_audio(path) → returns text`
`generate_image(text) → returns image link`
- Call them in sequence in `if __name__ == "__main__":` block

Libraries to Import:

- `openai` for API
 - `os` for environment variables
 - `dotenv` to load `.env`
-

[MEDIUM] Q12) Implement merge sort. Merge sort has two phases: the dividing phase and the merge phase. You could write this code in one line by using Python's `sorted()` function:

But this would defeat the purpose of the question, so don't use the `sorted()` function or `sort()` method as part of your solution.

Input: `arr = [38, 27, 43, 3, 9, 82, 10]`

Output: `[3, 9, 10, 27, 38, 43, 82]`

[HARD] Q13) You have a perfectly square box with side length **50 meters**. Inside the box, a small ball is moving with an initial **position (x, y)**, **velocity v**, and direction given by angle **θ (in degrees)** measured from the x-axis.

The box has **perfectly elastic walls**, meaning whenever the ball hits a wall, it **bounces off with the same speed and angle of incidence equals angle of reflection**.

Write a Python function that takes the following as input:

- Initial position: `x, y` (in meters)

- Angle: theta (in degrees)
- Velocity: v (in m/s)
- Time: t (in seconds)

And returns the **final position (xf, yf)** of the ball after t seconds.

Hint:

- Use theta to calculate x and y components of motion.
 - For reflection, **fold the motion** like wrapping beyond boundaries (similar to modulo with direction flip).
 - You can ignore acceleration (assume constant velocity).
-

[MEDIUM] Q14) Design a CLI(Command Line Interface) that takes a natural language coding prompt and returns Python code using OpenAI API for basic numerical operations (e.g., "integrate x^2 from 0 to 5").

Features:

1. Takes user input (natural language).
2. Uses OpenAI API (gpt-3.5-turbo or gpt-4) to generate Python code.
3. Executes and displays the result (optional).
4. Works for basic numerical operations (integration, differentiation, solving equations, etc.).

Step 1: Install Required Libraries

bash - pip install openai python-dotenv

Step 2: Set Up OpenAI API Key

1. Get an API key from [OpenAI](#).
2. Store it in .env:

```
echo "OPENAI_API_KEY=your_api_key_here" > .env
```

Step 3: CLI Code (nl_to_code.py)

STEP 1: Import required modules

Import the 'openai' module and others like 'os' and 'dotenv' for managing environment variables

import _____ # Fill here (Hint: OpenAI's Python SDK)

import _____ # For environment variable access

from dotenv import _____ # Load variables from .env file

STEP 2: Load your OpenAI API key from the .env file

_____() # Hint: load .env

openai.api_key = os._____("OPENAI_API_KEY") # Fetch the key

STEP 3: Function to call OpenAI API and generate code

def generate_python_code(prompt: str) -> str:

"""

Uses OpenAI API to generate Python code from a natural language prompt.

"""

response = openai._____.create(# Fill: Which API endpoint are we calling?

model="_____", # Model to use (Hint: 'gpt-3.5-turbo')

messages=[

 {"role": "system", "content": "You are a Python coding assistant. Convert the user's request into executable Python code. Only return the code, no explanations."},

 {"role": "user", "content": prompt}

],

temperature=_____, # Lower value = more consistent results (Hint: 0.3)

)

return response._____[0].message._____.strip() # Extract and return code only

STEP 4: Main CLI loop

def main():

```

print("Natural Language to Python Code CLI")

print("Example prompts:")

print("- 'integrate x^2 from 0 to 5'")

print("- 'solve 2x + 5 = 15'")

print("- 'find the derivative of sin(x)'\n")

while True:

    user_input = input("\nEnter your request (or 'quit' to exit): ")._____() # Fill: strip or
lower?

    if user_input.lower() in ["quit", "exit"]:

        break

    try:

        code = _____(user_input) # Call your function

        print("\nGenerated Python Code:\n")

        print(code)

        # Optional: Execute the code

        execute = input("\nExecute code? (y/n): ").strip().lower()

        if execute == "y":

            try:

                _____(code) # Dangerous in real-world apps; OK for controlled environments

            except Exception as e:

                print(f"Error executing code: {e}")

        except Exception as e:

            print(f"Error generating code: {e}")

# STEP 5: Run the script

if __name__ == "__main__":

    _____() # Fill: What function starts the program?

```

Step 4: Run the CLI

```
bash - python nl_to_code.py
```

Example Input:

Enter your request: "integrate x^2 from 0 to 5"

Output:

```
from scipy.integrate import quad
result, _ = quad(lambda x: x**2, 0, 5)
print(result) # Output: 41.666...
```

[HARD] Q15) You are provided with three algorithms:

1. Linear Search (expected time complexity: $O(n)$)
2. Nested Loops (expected time complexity: $O(n^2)$)
3. Recursive Fibonacci (expected time complexity: $O(2^n)$)

Use Python and NumPy to:

- Measure how the execution time grows with input size.
- Store and analyze runtime data in NumPy arrays.
- Plot the growth curves using matplotlib.

Your goal is to validate their theoretical time complexities empirically.

Hints:

1. Use NumPy arrays to hold input sizes and measured runtimes.
2. For runtime measurement, use `time.time()` or `time.perf_counter()` for higher resolution.
3. In Linear Search, make sure the target element is not in the list to simulate worst-case time.

4. In Recursive Fibonacci, limit the input size to below 30 to avoid excessive computation time.
5. Use matplotlib.pyplot to plot three subplots side by side.
6. Label axes and include legends for clarity.

Starter Code Template:

```
import time

import numpy as np

import matplotlib.pyplot as plt

# --- Algorithms ---

# 1. Linear Search ( $O(n)$ ) - Write Linear complexity function definition

# 2. Nested Loop ( $O(n^2)$ ) - Write Quadratic complexity function definition

# 3. Recursive Fibonacci ( $O(2^n)$ ) - Write Exponential complexity function definition

# --- Runtime Measurement Helper ---

def measure_runtime(func, *args):

    start = time.time()

    func(*args)

    end = time.time()

    return end - start

# --- Input Size Arrays ---

linear_input = np.linspace(1, 1000000, 40, dtype=int)

nested_input = np.linspace(1, 500, 20, dtype=int)

fibonacci_input = np.arange(1, 25, 1)

# --- Runtime Storage Arrays ---

linear_runtimes = np.zeros_like(linear_input, dtype=float)

nested_runtimes = np.zeros_like(nested_input, dtype=float)
```



```

fibonacci_runtimes = np.zeros_like(fibonacci_input, dtype=float)

# --- Benchmarking ---

# Linear Search

for i, n in enumerate(linear_input):

    arr = np.arange(n)

    linear_runtimes[i] = measure_runtime(linear_search, arr, -1) # Worst case

# Nested Loop

for i, n in enumerate(nested_input):

    nested_runtimes[i] = measure_runtime(nested_loop, n)

# Fibonacci

for i, n in enumerate(fibonacci_input):

    fibonacci_runtimes[i] = measure_runtime(fibonacci, n)

# --- Plotting ---

plt.figure(figsize=(15, 5))

# Linear Search Plot

# Nested Loop Plot

# Fibonacci Plot

plt.tight_layout()

plt.show()

```

