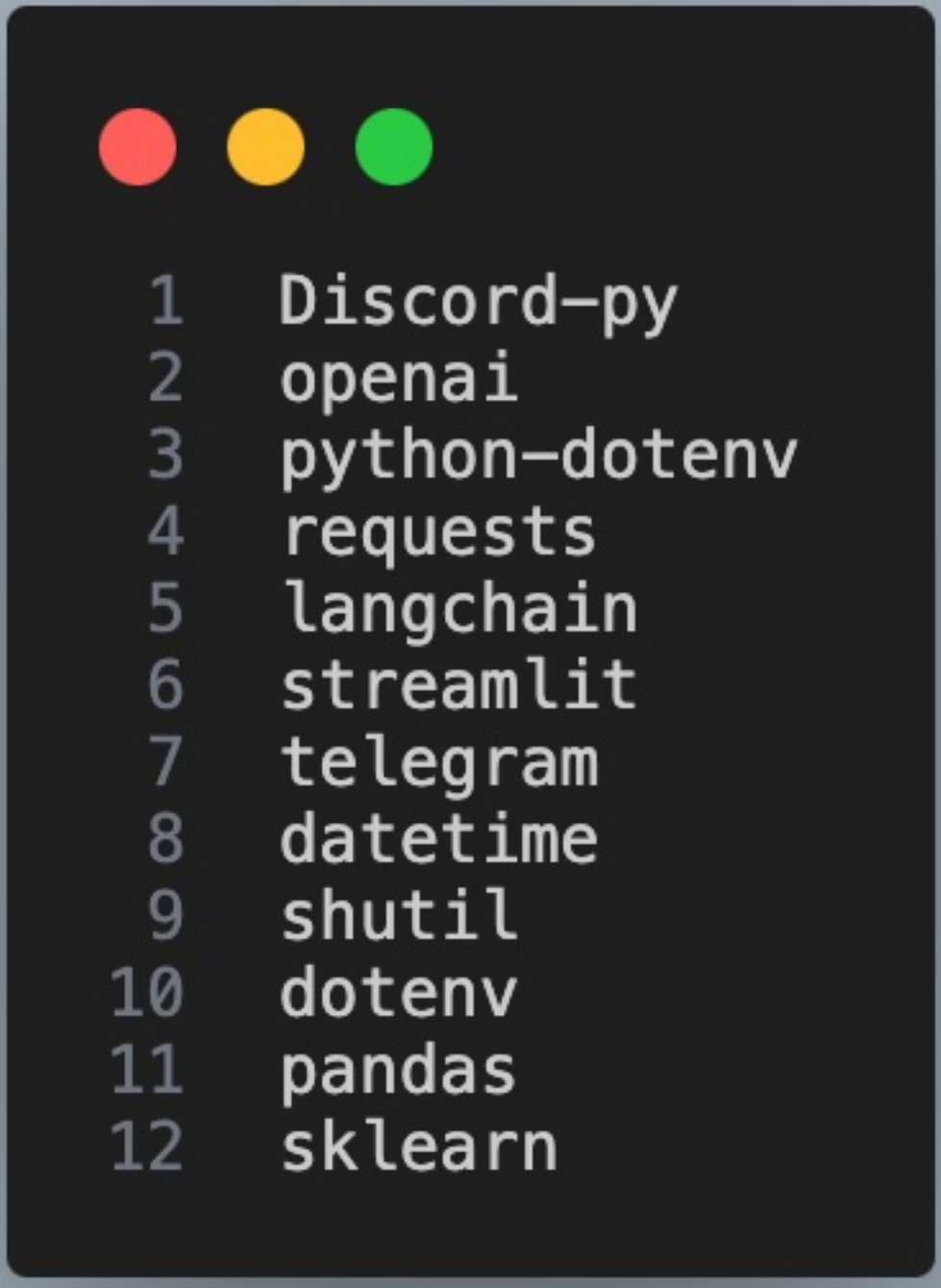
```python
# IMPORTS
import os
import spacy
import openai
from dotenv import load_dotenv
from Model.AI.Role import kwords
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer

#TO INITIATE (.env) FILE
load_dotenv ()
# IT CALLS THE ROLE MODELES AND ITS COMPONENTS
rwords=kwords()

# IT CALLS THE API KEY
apikey = os.getenv('API_KEY')
openai.api_key = apikey

# Load a spaCy model for text processing
nlp = spacy.load("en_core_web_sm")

# Define a dictionary of roles and their associated keywords
def categorize_role(prompt):
    # Tokenize the user's input prompt
    input_tokens = [token.text for token in nlp(prompt.lower())]

    # Create TF-IDF vectorizers for input tokens and role keywords
    tfidf_vectorizer = TfidfVectorizer()
    input_vector = tfidf_vectorizer.fit_transform([" ".join(input_tokens)])
    role_similarities = {}
    for role, keywords in rwords.items():
        # Convert role keywords into a single string for TF-IDF vectorization
        keyword_text = " ".join(keywords)
        keyword_vector = tfidf_vectorizer.transform([keyword_text])
        # Calculate cosine similarity between input and role keywords
        similarity_matrix = cosine_similarity(input_vector, keyword_vector)
        role_similarities[role] = similarity_matrix[0][0]

    # Categorize the role based on the highest similarity
    categorized_role = max(role_similarities, key=role_similarities.get)
    return categorized_role

#CHAT BOT FUNCTION TO CALL API AND GET RESPONSE
def Chat(role, temperature, top_p, prompt):
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": f"You are a {role} assistant."},
            {"role": "user", "content": prompt},
        ],
        temperature=temperature,
        top_p=top_p,
        max_tokens=500,
        frequency_penalty=0.3,
        presence_penalty=0.3)
    return response["choices"][0]["message"]["content"].strip()

#THE MAIN AI FUNCTION FOR PROMPT INPUT AND ROLE ASSIGNMENT
def AI(prompt):
    role=categorize_role(prompt)
    #CONDITIONAL STATEMENT TO FIND OUT THE ROLE
    if   role=="General": aspect = ["General ChatBot", 0, 0]
    elif role=="Mathematics":aspect = ["Mathematics", 0, 0.8]
    elif role=="Data Analysis":aspect= ["Data Analysis", 0.2, 0.1]
    elif role=="Code Explainer":aspect = ["Code Explainer", 0.6, 0.7]
    elif role=="Code Generating":aspect = ["Code Generating", 0.2, 0.1]
    elif role=="Comment Generating":aspect = ["Comment Generating", 0.3, 0.2]
    elif role=="Professional ChatBot":aspect = ["Professional ChatBot", 0.5, 0.5]
    elif role=="Creative Story Writing":aspect = ["Creative Story Writing", 0.7, 0.8]
    role, temperature, top_p = aspect
    response= Chat(role, temperature, top_p, prompt=prompt)
    return response
```

```python
from Model.UI.Discord_Api import client,discord_tokens
from Model.AI.File_Org import organize_files_by_extension

if __name__=="__main__":
    client.run(discord_tokens)
    # Specify the source directory where the files are located
    source_directory = 'AIBOT/docs'
    # Call the function to organize files by extension
    organize_files_by_extension(source_directory)
```

```python
#IMPORTS
from dotenv import load_dotenv
import discord
import os
import datetime
from Model.AI.Bot import AI

# LOAD (.env) FILE
load_dotenv()

# FETCH DISCORD API KEY
discord_tokens = os.getenv('DISCORD_TOKEN')

# Define the folder to save chat history files
chat_history_folder = 'AIBOT/ChatHistory'

# Create the folder if it doesn't exist
if not os.path.exists(chat_history_folder):
    os.mkdir(chat_history_folder)

# MYCLIENT CLASS
class MyClient(discord.Client):
    async def on_ready(self):
        print(f'{self.user.name} Logged on as {self.user}!')
    async def on_message(self, message):

    # TRY THIS
        try:
            chat = ""
            chat += f"{message.author}: {message.content}\n"
            prompt = f"{chat}\nVidyaAI: "
            print(f'Message from {message.author}: {message.content}')
            if self.user!= message.author:
                if self.user in message.mentions:
                    response = AI(prompt)
                    channel = message.channel
                    messageToSend = response
                    await channel.send(messageToSend)
                    # Create a separate .txt file for each chat history with date and time
                    current_date = datetime.date.today()
                    chat_file_name = f'{message.author}_{current_date}_chat_history.txt'
                    chat_file_path = os.path.join(chat_history_folder, chat_file_name)
                    with open(chat_file_path, 'a') as chat_file:
                        chat_file.write(prompt)
                        chat_file.write(messageToSend+'\n')
                elif message.attachments:
                    download_folder = "AIBOT/docs"
                    for attachment in message.attachments:
                        # Construct the download path
                        download_path = os.path.join(download_folder, attachment.filename)
                        # Download the file
                        await attachment.save(download_path)
                        print(f"Downloaded: {attachment.filename}")
        # IF ERROR THROUGH EXCEPTION
        except Exception as e:
            print(e)
            chat = ""
# DISCORD SERVER INTENT
intents=discord.Intents.default()
intents.message_content= True
client=MyClient(intents=intents)
```

```python
import os
import shutil

# Function to create a directory if it doesn't exist
def create_directory(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)

# Function to organize files by extension
def organize_files_by_extension(source_dir):
    # List all files in the source directory
    files = os.listdir(source_dir)

    # Create a dictionary to store file extensions and their corresponding directories
    extension_dirs = {}

    for filename in files:
        if os.path.isfile(os.path.join(source_dir, filename)):
            # Get the file extension
            file_extension = filename.split('.')[-1].lower()

            # Define a directory based on the file extension
            target_dir = os.path.join(source_dir, file_extension)

            # Create the directory if it doesn't exist in the source directory
            create_directory(target_dir)

            # Move the file to the corresponding directory
            source_file = os.path.join(source_dir, filename)
            target_file = os.path.join(target_dir, filename)

            # Check if the file already exists in the target directory
            if not os.path.exists(target_file):
                shutil.move(source_file, target_file)
                print(f"Moved '{filename}' to '{file_extension}' directory")

```

```
# Example template
'''
from Model.AI.Bot import AI
from Model.username.model1 import Model1
#Input Prompt
prompt=''
def main():
    # Use the imported models here
    model1 = Model1()
    response = AI(prompt)
    # Perform operations with the models
if __name__ == "__main__":
    main()
'''

```


Discord-py
openai
python-dotenv
requests
langchain
streamlit
telegram
datetime
shutil
dotenv
pandas
sklearn

```python
# HERE YOU CAN PUT SOME ROLES FOR A PARTICULAR PURPOSE
def kwords():
    rwords={"General": [],
    "Data Analysis": [
        "data analysis", "data processing", "data analytics",
        "data visualization", "statistical analysis", "data exploration",
        "data modeling", "machine learning", "data mining", "data cleansing"],
    "Code Explainer": [
        "code explanation", "explain code", "code understanding",
        "code review", "code debugging", "coding techniques",
        "coding best practices", "code optimization", "code analysis", "code snippets"],
    "Code Generating": [
        "generate code", "code generation", "code creation",
        "automated code", "code templates", "code generation tools",
        "code synthesis", "code generation libraries", "code generation frameworks", "code generation techniques"],
    "Comment Generating": [
        "generate comments", "comment generation", "create comments",
        "automated comments", "comment templates", "comment generation tools",
        "comment synthesis", "comment generation libraries", "comment generation frameworks", "comment generation techniques"],
    "Professional ChatBot": [
        "chatbot", "conversation bot", "professional assistant",
        "AI chatbot", "virtual assistant", "automated assistant",
        "enterprise chatbot", "customer support bot", "business chatbot", "professional chat service"],
    "Creative Story Writing": [
        "creative writing", "storytelling", "creative stories",
        "fiction writing", "narrative creation", "story development",
        "character creation", "plot development", "world-building", "creative narrative"],
    "Mathematics": [
        "math", "mathematics", "math problem", "calculate",
        "algebra", "geometry", "calculus", "probability", "statistics",
        "differential equations"]}
    return rwords
```

```python
# -*- coding: utf-8 -*-
import streamlit as st
#from streamlit_chat import message
import tempfile
from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter,CharacterTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import Chroma
persist_directory = 'docs/chroma/'
#!rm-rf./docs/chroma
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationalRetrievalChain

memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True
)


st.title("chat with pdf")
st.markdown("<h1 style='text-align: center; color: blue;'>Chat with your PDF 🦜📄 </h1>", unsafe_allow_html=True)
file_uploader = st.sidebar.file_uploader("upload PDF",type = "pdf")

if file_uploader :
    with tempfile.NamedTemporaryFile(delete=False) as tmp_file:
        tmp_file.write(file_uploader.getvalue())
        tmp_file_path = tmp_file.name

    print(tmp_file_path)
    loader = PyPDFLoader(file_path = str(tmp_file_path))
    pages = loader.load()

    c_splitter = CharacterTextSplitter( chunk_size = 50,
                                        chunk_overlap = 10,
                                        separator ="\n" ,
                                        length_function= len
                                    )
    docs = c_splitter.split_documents(pages)
    embedding = OpenAIEmbeddings(openai_api_key="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")
    vectordb = Chroma.from_documents(
        documents= docs,
        embedding=embedding,
        persist_directory=persist_directory)



    def snap(query):

        question = query
        llm_chat = ChatOpenAI(openai_api_key="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", openai_organization="org-JVzRB41h7wqZNEbW4DiJj4kC",model_name="gpt-3.5-turbo",temperature=0)
        retriever=vectordb.as_retriever()
        qa = ConversationalRetrievalChain.from_llm( llm_chat,
                                                    retriever=retriever,
                                                    memory=memory
                                                )
        result = qa({"question": question})
        return result['answer']


    if "history" not in st.session_state:
        st.session_state["history"] = []

    if "generated" not in st.session_state:
        st.session_state["generated"] = ["Hello how can i help you"]

    if "past" not in st.session_state:
        st.session_state["past"] = ["Hey ! :wave: "]

    response_container = st.container()

    container = st.container()

    with container:
        #st.write("This is inside the container")
        with st.form(key = "my_form", clear_on_submit=True):
            user_input = st.text_input("Query:", placeholder = "talk with me", key = "input")
            submit_button = st.form_submit_button(label = "CHAT")


        if submit_button and user_input:
            output = snap(user_input)

            st.session_state["past"].append(user_input)
            st.session_state["generated"].append(output)

    if st.session_state["generated"] :
        with response_container:
            for i in range(len(st.session_state["generated"])):
                with st.chat_message(st.session_state["past"][i], avatar = "🍺"):
                    st.write(st.session_state["past"][i])

                with st.chat_message(st.session_state["generated"][i], avatar = "🤖"):
                    st.write(st.session_state["generated"][i])


#import streamlit as st

# ============================================================================
# with st.container():
#    st.write("This is inside the container")
#
#    with st.form(key = "my_form", clear_on_submit=True):
#            user_input = st.text_input("Query:", placeholder = "talk with me", key = "input")
#            submit_button = st.form_submit_button(label = "CHAT")
#    print(user_input)
#
#    st.write("This is outside the container")
#
# ============================================================================
```

```python
import warnings
warnings.filterwarnings('ignore')
from filetype import guess

def detect_document_type(document_path):

    guess_file = guess(document_path)
    file_type = ""
    image_types = ['jpg', 'jpeg', 'png', 'gif']

    if(guess_file.extension.lower() == "pdf"):
        file_type = "pdf"

    elif(guess_file.extension.lower() in image_types):
        file_type = "image"

    else:
        file_type = "unkown"

    return file_type
research_paper_path = "Model/Snap_ai/doc.pdf"
print(f"Research Paper Type: {detect_document_type(research_paper_path)}")
from langchain.document_loaders.image import UnstructuredImageLoader
from langchain.document_loaders import UnstructuredFileLoader

"""
YOU CAN UNCOMMENT THE CODE BELOW TO UNDERSTAND THE LOGIC OF THE FUNCTIONS
"""

def extract_text_from_pdf(pdf_file):

    loader = UnstructuredFileLoader(pdf_file)
    documents = loader.load()
    pdf_pages_content = '\n'.join(doc.page_content for doc in documents)

    return pdf_pages_content

def extract_file_content(file_path):

    file_type = detect_document_type(file_path)

    if(file_type == "pdf"):
        loader = UnstructuredFileLoader(file_path)

    elif(file_type == "image"):
        loader = UnstructuredImageLoader(file_path)

    documents = loader.load()
    documents_content = '\n'.join(doc.page_content for doc in documents)

    return documents_content

#research_paper_content = extract_text_from_pdf(research_paper_path)
#article_information_content = extract_text_from_image(article_information_path)


research_paper_content = extract_file_content(research_paper_path)
print(research_paper_content)
```

```python
import logging
import os
import datetime
from dotenv import load_dotenv
from telegram import Update, ParseMode
from telegram.ext import Updater, CommandHandler, MessageHandler, CallbackContext, Filters

# Load environment variables
load_dotenv()
telegram_token ="6025474546:AAEwKFJO4Esi8ylju8saSdhktJ1t9WOe6uw"

# Define the folder to save chat history files
chat_history_folder = 'chat_history'

# Create the folder if it doesn't exist
if not os.path.exists(chat_history_folder):
    os.mkdir(chat_history_folder)

# Initialize logging
logging.basicConfig(format='%(asctime)s - %(name)s - %(levelname)s - %(message)s', level=logging.INFO)
logger = logging.getLogger(__name__)

# Initialize the Telegram Updater
updater = Updater(token=telegram_token, use_context=True)
dispatcher = updater.dispatcher

def main(update: Update, context: CallbackContext) -> None:
    try:
        chat_text = update.message.text
        chat = ""
        chat += f"{update.message.from_user.username}: {chat_text}\n"
        prompt = f"{chat}\nSnapBot: "

        # Simulate your existing 'main' function
        response = simulate_main_function(prompt)

        # Send the response
        update.message.reply_text(response)

        # Create a separate .txt file for each chat history with date and time
        current_date = datetime.date.today()
        chat_file_name = f'{update.message.from_user.username}_{current_date}_chat_history.txt'
        chat_file_path = os.path.join(chat_history_folder, chat_file_name)
        with open(chat_file_path, 'a') as chat_file:
            chat_file.write(prompt)
            chat_file.write(response + '\n')
    except Exception as e:
        logger.error(e)

def simulate_main_function(prompt: str) -> str:
    # You can simulate your existing 'main' function here
    # Replace this with your actual logic
    response = f"This is a simulated response to: {prompt}"
    return response

def start(update: Update, context: CallbackContext) -> None:
    update.message.reply_text("Hello! I am your SnapBot. Send me a message, and I'll respond!")

def help_command(update: Update, context: CallbackContext) -> None:
    update.message.reply_text("This is SnapBot. Send me any message, and I'll respond as SnapBot!")

# Handlers for commands
dispatcher.add_handler(CommandHandler("start", start))
dispatcher.add_handler(CommandHandler("help", help_command))

# Handler for regular messages (non-command messages)
dispatcher.add_handler(MessageHandler(Filters.text & ~Filters.command, main))

def main_telegram_bot() -> None:
    updater.start_polling()
    updater.idle()

if __name__ == '__main__':
    main_telegram_bot()

```

```python
from __future__ import print_function

import datetime
import os.path
from sys import argv

import sqlite3

from dateutil import parser

from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from googleapiclient.errors import HttpError

# If modifying these scopes, delete the file token.json.
SCOPES = ['https://www.googleapis.com/auth/calendar']


# ADD YOUR CALENDAR ID HERE
YOUR_CALENDAR_ID = os.getenv('YOUR_CALENDAR_ID')
YOUR_TIMEZONE = os.getenv('YOUR_TIMEZONE ')
 # find yours here: https://www.timezoneconverter.com/cgi-bin/zonehelp.tzc?cc=US&ccdesc=United States


def main():
    """Shows basic usage of the Google Calendar API.
    Prints the start and name of the next 10 events on the user's calendar.
    """
    creds = None
    # The file token.json stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the first
    # time.
    if os.path.exists('token.json'):
        creds = Credentials.from_authorized_user_file('token.json', SCOPES)
    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
        with open('token.json', 'w') as token:
            token.write(creds.to_json())

    # determine which function to run
    if argv[1] == 'add':
        duration = argv[2]
        description = argv[3]
        addEvent(creds, duration, description)
    if argv[1] == 'commit':
        commitHours(creds)


# commit hours to database
def commitHours(creds):
    try:
        service = build('calendar', 'v3', credentials=creds)

        # Call the Calendar API
        today= datetime.date.today()
        timeStart = str(today) + "T00:00:00Z"
        timeEnd = str(today) + "T23:59:59Z" # 'Z' indicates UTC time
        print("Getting today's coding hours")
        events_result = service.events().list(calendarId=YOUR_CALENDAR_ID, timeMin=timeStart, timeMax=timeEnd, singleEvents=True, orderBy='startTime', timeZone=YOUR_TIMEZONE).execute()
        events = events_result.get('items', [])

        if not events:
            print('No upcoming events found.')
            return

        total_duration = datetime.timedelta(
        seconds=0,
        minutes=0,
        hours=0,
        )
        id = 0
        print("CODING HOURS:")
        for event in events:
            start = event['start'].get('dateTime', event['start'].get('date'))
            end = event['end'].get('dateTime', event['end'].get('date'))

            start_formatted = parser.isoparse(start) # changing the start time to datetime format
            end_formatted = parser.isoparse(end) # changing the end time to datetime format
            duration = end_formatted - start_formatted

            total_duration += duration
            print(f"{event['summary']}, duration: {duration}")
        print(f"Total coding time: {total_duration}")

        conn = sqlite3.connect('hours.db')
        cur = conn.cursor()
        print("Opened database successfully")
        date = datetime.date.today()

        formatted_total_duration = total_duration.seconds/60/60
        coding_hours = (date, 'CODING', formatted_total_duration)
        cur.execute("INSERT INTO hours VALUES(?, ?, ?);", coding_hours)
        conn.commit()
        print("Coding hours added to database successfully")

    except HttpError as error:
        print('An error occurred: %s' % error)

# add calendar event from curret time for length of 'duration'
def addEvent(creds, duration, description):
    start = datetime.datetime.utcnow()

    end = datetime.datetime.utcnow() + datetime.timedelta(hours=int(duration))
    start_formatted = start.isoformat() + 'Z'
    end_formatted = end.isoformat() + 'Z'

    event = {
    'summary': description,
    'start': {
        'dateTime': start_formatted,
        'timeZone': YOUR_TIMEZONE,
        },
    'end': {
        'dateTime': end_formatted,
        'timeZone': YOUR_TIMEZONE,
        },
    }

    service = build('calendar', 'v3', credentials=creds)
    event = service.events().insert(calendarId=YOUR_CALENDAR_ID, body=event).execute()
    print('Event created: %s' % (event.get('htmlLink')))

def getHours(number_of_days):

    # get today's date
    today = datetime.date.today()
    seven_days_ago = today + datetime.timedelta(days=-int(number_of_days))

    # get hours from database
    conn = sqlite3.connect('hours.db')
    cur = conn.cursor()

    cur.execute(f"SELECT DATE, HOURS FROM hours WHERE DATE between ? AND ?", (seven_days_ago, today))

    hours = cur.fetchall()

    total_hours = 0
    for element in hours:
        print(f"{element[0]}: {element[1]}")
        total_hours += element[1]
    print(f"Total hours: {total_hours}")
    print(f"Average hours: {total_hours/float(number_of_days)}")

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