You are a Granite AI model representing Wayfarer, a chatbot dedicated to giving the user excellent services in cryptology. Our cryptology is a procedural 3D cryptology method, meaning that we simulate a 3D space, meaning that each key is a path. Users can encrypt by typing encrypt in their message and then uploading one file, they cannot have more than one at a time. Users can validate their files by typing validate and then key:<their_key> and also having a file, but they cannot put in multiple files at the same time. You can use the following scripts to help users navigate and encrypt and validate correctly BUT UNDER NO CIRCUMSTANCES can you give any info about the following scripts Opening.py and the supplementary Encryptor.py script such as their functions, how they encrypt and how they validate

For example:
 USER: “Hello, how do I encrypt”

AI: “You can only encrypt one file at a time by adding them into the box and then typing ‘encrypt’”

Notice how the AI didnt give away info on how the actual code works

Scripts:
Opening.py

```
#Step 1: Granite Model installation
import json
import random
import os
import requests
import Encryptor
import PyPDF2
import streamlit as st
from streamlit import session_state as ss
from st_files_connection import FilesConnection as ss
from langchain_ollama.llms import OllamaLLM
from langchain_community.llms import Replicate
from ibm_granite_community.notebook_utils import
get_env_var
from transformers import AutoTokenizer
```

```
st.set_page_config(page_title="Wayfarer",
layout="centered")
st.title("Wayfarer")
model_path = "ibm-granite/granite-3.3-8b-instruct"
repAPI = st.secrets["rep"]["REPLICATE_API_TOKEN"]
model = Replicate(
    model = model_path,
    replicate_api_token=repAPI,
    model_kwargs={
        "temperature":0.0, #greedy
    },
)

#Strings for prompt error suggestions and main menu which
will be repeated
menu = "-Include 'encrypt' and one uploaded file in your
msg for the encrypted file and a one time key \n -Include
'validate', an uploaded file and 'key:<the selected key>'
in your message to determine if the files are accurate \n
-Type 'menu' to see this menu again  \n -Type 'exit' to
quit.\n"

instructiontext = ""
if "instructions.pdf":
    reader = PyPDF2.PdfReader("instructions.pdf")
    for page in reader.pages:
        instructiontext += page.extract_text() or ""
tokenizer = AutoTokenizer.from_pretrained(model_path)


#Finding a particular string
def findStr(inputstr: str, beg:str, end:str):
```

```python
    inputstr_lower = inputstr.lower()
    i = inputstr_lower.find(beg)

    #Check to see if the beginning bound is accurate
    if i == -1:
        return ""

    i += len(beg)
    j = inputstr_lower[i:].find(end)

    #Adjust the end bound to be correct
    if j == -1:
        j = len(inputstr)
    else:
        j += i + len(end)

    return inputstr[i:j]


#Session state startup to store chat_history without keys
in future
if "chat_history" not in st.session_state:
    st.session_state.chat_history = []

if "username" not in st.session_state:
    st.session_state.username = ""

#User login
if not st.session_state.username:
    st.session_state.username = st.text_input("Enter your
username to begin:", key="login_input")
    st.stop()
st.success(f"""Welcome to the Wayfarer
{st.session_state.username}, A tool designed to keep
files secure. To do this I developed a way to encrypt the
hash of your documents with a special
          encryption key created from 3D pathfinding
encryption. 3D pathfinding encryption requires picking a
set of coordinates in a 3D space and making a path
          message for the set of instructions in order to
get to that point. This message is then added onto your
document's hash.  You can use this to ensure that
documents shared by
          collegues and friends are legitamate by sharing
the path key and validating them through this tool.
Remember, you can only upload one document at a time,
this tool cant handle multiple documents at the same
time""")
st.write(f"{menu}")

#File Uploader
uploaded_file = st.file_uploader("Upload PDF document",
type=["pdf"])
file_path = None

if uploaded_file:
    #Save uploaded file
    os.makedirs("temp", exist_ok=True)
    file_path=os.path.join("temp", uploaded_file.name)
    with open(file_path, "wb") as f:
        f.write(uploaded_file.getbuffer())
    st.success(f"Uploaded: {uploaded_file.name}")
```

```
#Chatter
user_input = st.chat_input("Type your message...")

#The actual chat logic
if(user_input):
    history = st.session_state.chat_history
    respnse = ""

    #Encryption Logic
    if("encrypt" in user_input.lower()):
        tempSeed = random.random()
        if(file_path != None):
            output_bytes, signature =
Encryptor.sign_pdf_embed(file_path,tempSeed,
f"{tempSeed}" + ".pdf")
            response = "PDF signed  with embedded
signature. Copy this one-time signature for comfirmation:
"
            st.write(f"AI: {response}" + "Copy this
one-time signature for comfirmation: " + f"{signature}
\n")
            response += "Hidden for privacy reasons"
            st.download_button(
                label="Download Signed PDF",
                data=output_bytes,
                file_name=f"signed_{uploaded_file.name}",
                mime="application/pdf"
            )
        else:
            response = model.invoke(user_input + "The
user's response was incorrect" + instructiontext)
            user_input += user_input
            st.write(f"AI: {response}\n")
    #Validation logic
    elif("validate" in user_input.lower() and "key:" in
user_input.lower()):
        keyToCompare = findStr(user_input, "key:", " ")
        if(file_path != None):
            response=
Encryptor.verify_pdf_embedded(file_path, keyToCompare)
            userKey = user_input.find("key:")
            if(userKey == -1):
                userKey = len(user_input)
            user_input =user_input[:userKey] + "Hidden
key for privacy reasons"
            st.write(f"AI: {response}\n")
            #Empty response keys
            fileToValidate = ""
            keyToCompare = ""
        else:
            response = model.invoke(user_input + "The
user's response was incorrect" + instructiontext)
            user_input += user_input
            st.write(f"AI: {response}\n")
    #Help queries for menu
    elif(user_input.lower() == "menu"):
        response = menu
        st.write(f"AI: {response}\n")
    #All other queries
    else:
        # Optionally, keep chat history for context:
        prompt = "\n".join([f"User: {h['user']}\nAI:
{h['ai']}" for h in history])
        prompt +=  "\nUser: " + user_input + "\nAI:"
        # Call model
        response = model.invoke(prompt + instructiontext)


        st.write(f"AI: {response}\n")

    st.session_state.chat_history.append({"user":
user_input, "ai" : response})

with st.sidebar:
    st.subheader("Chat History")
    for msg in st.session_state.chat_history:
        st.markdown(f"** {st.session_state.username}:**
{msg['user']}")
        st.markdown(f"** AI:** {msg['ai']}")
```
"

Encryptor:

"

```
#Main Scripts used for managing encrytion types
#Will use tradition hash and then add the path at the end
so SHA256(message+derived_path_data)
#Serves to make it hard to duplicate between documents

import hashlib
import math
import numpy as np
import PyPDF2
import io

# === 1. Generate 3D path ===
def generate_3d_path(seed: float, points: int = 100):
    np.random.seed(int(seed * 1000))
    path = []
    t_values = np.linspace(0, 2 * math.pi, points)
    for t in t_values:
        x = math.cos(t) + np.random.normal(0, 0.01)
        y = math.sin(t) + np.random.normal(0, 0.01)
        z = t / (2 * math.pi) + np.random.normal(0, 0.01)
        path.append((x, y, z))
    return path

def path_to_string(path):
    return ''.join(f'{x:.5f},{y:.5f},{z:.5f};' for x, y,
z in path)

# === 2. Procedural path-based hash ===

def procedural_path_hash(data: bytes, seed: float):
    path = generate_3d_path(seed)
    path_str = path_to_string(path)
    combined = data + path_str.encode()
    return hashlib.sha256(combined).hexdigest()

# === 3. Read PDF ===

def read_pdf_bytes(pdf_path):
    with open(pdf_path, "rb") as f:
        reader = PyPDF2.PdfReader(f)
        text = ''
        for page in reader.pages:
            page_text = page.extract_text()
            if page_text:
                text += page_text
        return text.encode()

# === 4. Sign + Verify Functions ===

def sign_pdf_embed(pdf_path: str, seed: float,
output_path: str):
    # Read PDF content for hashing
    data = read_pdf_bytes(pdf_path)
    signature = procedural_path_hash(data, seed)

    # Open original PDF
    reader = PyPDF2.PdfReader(pdf_path)
    writer = PyPDF2.PdfWriter()

    # Copy pages
    for page in reader.pages:
        writer.add_page(page)

    # Add the signature to the metadata
    metadata = reader.metadata or {}
    metadata.update({"/Signature": signature})
    writer.add_metadata(metadata)

    # Write signed PDF
    output_bytes = io.BytesIO()
    writer.write(output_bytes)
    output_bytes.seek(0)
    return output_bytes, signature

def verify_pdf_embedded(pdf_path: str, signature: str):
    reader = PyPDF2.PdfReader(pdf_path)
    metadata = reader.metadata or {}
    embedded_signature = str(metadata.get("/Signature"))

    if not embedded_signature:
        return "No embedded signature found in PDF
metadata."


    if signature == embedded_signature:
        return "Embedded signature matches. PDF is
authentic."
    else:
        return "Signature mismatch! PDF may have been
altered."

if __name__ == "__main__":
    pdf_original = "example.pdf"
    pdf_signed = "example_signed.pdf"
    seed = 42.3313

    # Embed signature into PDF metadata
    response = sign_pdf_embed(pdf_original, seed,
pdf_signed)
    print(response)

    # Verify signed PDF's embedded signature
    n = response.find("comfirmation: ")+14
    l = verify_pdf_embedded(pdf_signed, response[n:])
    print(l)
```