Python Files

-- **FileName:**
10CC.py

Content:

```
import hashlib

text = input("Enter text: ").strip().encode()

hashes = {
"MD5": hashlib.md5(text).hexdigest(),
"SHA1": hashlib.sha1(text).hexdigest(),
"SHA256": hashlib.sha256(text).hexdigest()
}

for algo, h in hashes.items():
print(f"{algo}: {h}")
```

-- **FileName:**
11CC.py

Content:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import os

print("Installing libary...")

os.system("pip install pycryptodome")
# Key must be 16, 24, or 32 bytes
key = b'ThisIsA16ByteKey'

message = "Hello, this is a secret message!"
data = message.encode()

# Pad the data to be multiple of 16 bytes
padded_data = pad(data, AES.block_size)

# Encrypt
cipher = AES.new(key, AES.MODE_ECB)
ciphertext = cipher.encrypt(padded_data)
print("Encrypted (hex):", ciphertext.hex())

# Decrypt
decipher = AES.new(key, AES.MODE_ECB)
decrypted_padded = decipher.decrypt(ciphertext)
decrypted = unpad(decrypted_padded, AES.block_size)
print("Decrypted:", decrypted.decode())
```

-- **FileName:** 12CC.py

Content:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

key = b'ThisIsA16ByteKey' # 16-byte key
message = "Hello, this is a secret message!"
data = message.encode()

# Generate a random IV
iv = get_random_bytes(16)

# Encrypt
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(data, AES.block_size))
print("Encrypted (hex):", ciphertext.hex())

# Decrypt
decipher = AES.new(key, AES.MODE_CBC, iv)
decrypted_padded = decipher.decrypt(ciphertext)
decrypted = unpad(decrypted_padded, AES.block_size)
print("Decrypted:", decrypted.decode())
```

-- **FileName:** 13CC.py

Content:

```
import secrets

# 128-bit key (16 bytes)
key_128 = secrets.token_bytes(16)
print("AES-128 key (hex):", key_128.hex())

# 256-bit key (32 bytes)
key_256 = secrets.token_bytes(32)
print("AES-256 key (hex):", key_256.hex())
```

-- **FileName:** 14CC.py

Content:

```
from Crypto.Util.Padding import pad, unpad

block_size = 16 # AES block size in bytes

def pkcs7_pad(plaintext: bytes) -> bytes:
return pad(plaintext, block_size)
```

```
def pkcs7_unpad(padded: bytes) -> bytes:
return unpad(padded, block_size)

# Example usage
plaintext = b"Hello, this is a test!"
print("Original:", plaintext)

padded = pkcs7_pad(plaintext)
print("Padded (hex):", padded.hex())

unpadded = pkcs7_unpad(padded)
print("Unpadded:", unpadded)
```

-- **FileName:**
15CC.py

Content:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes

key = get_random_bytes(16) # 128-bit AES key
input_file = "example.txt"
output_file = "example.enc"

# Read plaintext
with open(input_file, "rb") as f:
plaintext = f.read()

# Generate random IV
iv = get_random_bytes(16)

# Encrypt
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))

# Write IV + ciphertext to file
with open(output_file, "wb") as f:
f.write(iv + ciphertext)

print(f"File encrypted as {output_file}")
print(f"AES key (hex) for decryption: {key.hex()}")
```

-- **FileName:**
16CC.py

Content:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
```

```
key_hex = input("Enter AES key (hex): ").strip()
key = bytes.fromhex(key_hex)

input_file = "example.enc"
output_file = "example_decrypted.txt"

# Read IV + ciphertext from file
with open(input_file, "rb") as f:
iv = f.read(16) # first 16 bytes are IV
ciphertext = f.read()

# Decrypt
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext_padded = cipher.decrypt(ciphertext)
plaintext = unpad(plaintext_padded, AES.block_size)

# Write decrypted file
with open(output_file, "wb") as f:
f.write(plaintext)

print(f"File decrypted as {output_file}")
```

-- **FileName:**
17CC.py

Content:

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad

key = b'8ByteKey' # DES key must be exactly 8 bytes
message = "Hello DES encryption!"
data = message.encode()

# Pad data to 8-byte blocks
padded_data = pad(data, DES.block_size)

# Encrypt (ECB mode)
cipher = DES.new(key, DES.MODE_ECB)
ciphertext = cipher.encrypt(padded_data)
print("Encrypted (hex):", ciphertext.hex())

# Decrypt
decipher = DES.new(key, DES.MODE_ECB)
decrypted_padded = decipher.decrypt(ciphertext)
decrypted = unpad(decrypted_padded, DES.block_size)
print("Decrypted:", decrypted.decode())
```

-- **FileName:**
18CC.py

Content:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes

key = b'ThisIsA16ByteKey' # 16-byte AES key
plaintext = b"This is a secret message. " * 4 # repeated to show pattern

# Pad plaintext
padded = pad(plaintext, AES.block_size)

# Encrypt with ECB
cipher_ecb = AES.new(key, AES.MODE_ECB)
ciphertext_ecb = cipher_ecb.encrypt(padded)

# Encrypt with CBC
iv = get_random_bytes(16)
cipher_cbc = AES.new(key, AES.MODE_CBC, iv)
ciphertext_cbc = cipher_cbc.encrypt(padded)

# Show results
print("ECB ciphertext (hex):", ciphertext_ecb.hex())
print("CBC ciphertext (hex):", ciphertext_cbc.hex())
```

-- **FileName:** 19CC.py

Content:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes

key_input = input("Enter a key (16, 24, or 32 chars): ").strip()
key = key_input.encode()
if len(key) not in (16, 24, 32):
print("Key must be 16, 24, or 32 bytes!")
exit()

text = input("Enter text to encrypt: ").strip().encode()
output_file = input("Enter output filename: ").strip()

# Generate random IV
iv = get_random_bytes(16)

# Encrypt
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(text, AES.block_size))

# Save IV + ciphertext to file
with open(output_file, "wb") as f:
f.write(iv + ciphertext)

print(f"Encrypted data saved to {output_file}")
```

-- **FileName:**
1CC.py

Content:

```
import base64

original_bytes = b"Hello, World!"
encoded_bytes = base64.b64encode(original_bytes)
print(f"Base64 Encoded: {encoded_bytes}")

decoded_bytes = base64.b64decode(encoded_bytes)
print(f"Base64 Decoded: {decoded_bytes}")
```

-- **FileName:**
20CC.py

Content:

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util import Counter

key = b'ThisIsA16ByteKey' # 16-byte AES key
plaintext = b"Hello, this is a secret message!"

# Generate a random 64-bit nonce
nonce = get_random_bytes(8)

# Create counter starting from nonce
ctr = Counter.new(64, prefix=nonce, initial_value=0)

# Encrypt
cipher = AES.new(key, AES.MODE_CTR, counter=ctr)
ciphertext = cipher.encrypt(plaintext)
print("Encrypted (hex):", ciphertext.hex())

# Decrypt (need the same nonce)
ctr_dec = Counter.new(64, prefix=nonce, initial_value=0)
decipher = AES.new(key, AES.MODE_CTR, counter=ctr_dec)
decrypted = decipher.decrypt(ciphertext)
print("Decrypted:", decrypted.decode())
```

-- **FileName:**
21CC.py

Content:

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
```

```
# Generate a private key
private_key = rsa.generate_private_key(
public_exponent=65537,
key_size=2048 # Can be 2048, 3072, or 4096 (higher = more secure, slower)
)

# Generate the public key
public_key = private_key.public_key()

# Serialize private key to PEM format
pem_private = private_key.private_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PrivateFormat.PKCS8,
encryption_algorithm=serialization.NoEncryption() # Or use BestAvailableEncryption(b"password")
)

# Serialize public key to PEM format
pem_public = public_key.public_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo
)

# Save to files
with open("private_key.pem", "wb") as f:
f.write(pem_private)

with open("public_key.pem", "wb") as f:
f.write(pem_public)

print("RSA key pair generated successfully!")
```

-- **FileName:** 22CC.py

Content:

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes

# 1. Generate RSA private & public keys
private_key = rsa.generate_private_key(
public_exponent=65537,
key_size=2048
)
public_key = private_key.public_key()

# 2. Message to encrypt
message = b"Hello, World!"

# 3. Encrypt with public key
ciphertext = public_key.encrypt(
message,
padding.OAEP( # Use OAEP padding for security
mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(),
label=None
)
)

print("Encrypted message (hex):", ciphertext.hex())

# 4. Decrypt with private key
decrypted = private_key.decrypt(
ciphertext,
padding.OAEP(
mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(),
label=None
)
)

print("Decrypted message:", decrypted.decode())
```

-- **FileName:** 23CC.py

Content:

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes

# 1. Generate RSA key pair
private_key = rsa.generate_private_key(
public_exponent=65537,
key_size=2048
)
public_key = private_key.public_key()

# 2. Message to sign
message = b"Hello, World!"

# 3. Sign with private key
signature = private_key.sign(
message,
padding.PSS( # Probabilistic Signature Scheme
mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH
),
hashes.SHA256()
)

print("Signature (hex):", signature.hex())

# 4. Verify with public key
try:
public_key.verify(
signature,
message,
padding.PSS(
mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH
),
hashes.SHA256()
)
print("■ Signature is valid")
except Exception:
print("■ Signature is invalid")
```

--- **FileName:** 24CC.py

Content:

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes

# 1. Generate RSA key pair
private_key = rsa.generate_private_key(
public_exponent=65537,
key_size=2048
)
public_key = private_key.public_key()

# 2. Message
message = b"Hello, World!"

# 3. Sign with private key
signature = private_key.sign(
message,
padding.PSS(
mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH
),
hashes.SHA256()
)

# 4. Verify with public key
try:
public_key.verify(
signature,
message,
padding.PSS(
mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH
),
hashes.SHA256()
)
print("■ Signature is valid")
except Exception:
print("■ Signature is invalid")
```

-- **FileName:**
25CC.py

Content:

```
import os
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.exceptions import InvalidSignature

PRIVATE_KEY_PATH = "private_key.pem"
PUBLIC_KEY_PATH = "public_key.pem"


def generate_and_save_keys(private_path: str, public_path: str, key_size: int = 2048):
private_key = rsa.generate_private_key(public_exponent=65537, key_size=key_size)
public_key = private_key.public_key()

with open(private_path, "wb") as f:
f.write(private_key.private_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PrivateFormat.PKCS8,
encryption_algorithm=serialization.NoEncryption()
))
try:
os.chmod(private_path, 0o600)
except Exception:
pass

with open(public_path, "wb") as f:
f.write(public_key.public_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo
))

return private_key, public_key


def load_keys(private_path: str, public_path: str):
with open(private_path, "rb") as f:
private_key = serialization.load_pem_private_key(f.read(), password=None)

with open(public_path, "rb") as f:
public_key = serialization.load_pem_public_key(f.read())

return private_key, public_key


def sign_message(private_key, message: bytes) -> bytes:
signature = private_key.sign(
message,
padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH),
hashes.SHA256()
)
return signature
```

```
def verify_signature(public_key, signature: bytes, message: bytes) -> bool:
try:
public_key.verify(
signature,
message,
padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH),
hashes.SHA256()
)
return True
except InvalidSignature:
return False


def main():
if not (os.path.exists(PRIVATE_KEY_PATH) and os.path.exists(PUBLIC_KEY_PATH)):
private_key, public_key = generate_and_save_keys(PRIVATE_KEY_PATH, PUBLIC_KEY_PATH)
else:
private_key, public_key = load_keys(PRIVATE_KEY_PATH, PUBLIC_KEY_PATH)

message = b"Hello, World!"
signature = sign_message(private_key, message)

print(signature.hex())

if verify_signature(public_key, signature, message):
print("Signature is valid.")
else:
print("Signature is invalid.")


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

-- **FileName:** 26CC.py

Content:

```
import os
import json
import base64

from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.exceptions import InvalidTag
os.system("pip install cryptography")

PRIVATE_KEY_PATH = "private_key.pem"
PUBLIC_KEY_PATH = "public_key.pem"


def generate_rsa_keypair(private_path: str, public_path: str, key_size: int = 2048):
```

```
private_key = rsa.generate_private_key(public_exponent=65537, key_size=key_size)
public_key = private_key.public_key()

with open(private_path, "wb") as f:
f.write(private_key.private_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PrivateFormat.PKCS8,
encryption_algorithm=serialization.NoEncryption()
))
try:
os.chmod(private_path, 0o600)
except Exception:
pass

with open(public_path, "wb") as f:
f.write(public_key.public_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo
))

return private_key, public_key


def load_private_key(path: str):
with open(path, "rb") as f:
return serialization.load_pem_private_key(f.read(), password=None)


def load_public_key(path: str):
with open(path, "rb") as f:
return serialization.load_pem_public_key(f.read())


def hybrid_encrypt(public_key, plaintext: bytes) -> bytes:
# 1) Generate random AES-256 key and nonce
aes_key = AESGCM.generate_key(bit_length=256)
aesgcm = AESGCM(aes_key)
nonce = os.urandom(12) # recommended size for GCM

# 2) Encrypt plaintext with AES-GCM
ciphertext = aesgcm.encrypt(nonce, plaintext, associated_data=None)

# 3) Encrypt AES key with RSA-OAEP
enc_key = public_key.encrypt(
aes_key,
padding.OAEP(
mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(),
label=None
)
)

# 4) Package as JSON with base64 fields
package = {
"enc_key": base64.b64encode(enc_key).decode("ascii"),
"nonce": base64.b64encode(nonce).decode("ascii"),
"ciphertext": base64.b64encode(ciphertext).decode("ascii")
}
return json.dumps(package).encode("utf-8")


def hybrid_decrypt(private_key, package_bytes: bytes) -> bytes:
# 1) Parse package
package = json.loads(package_bytes.decode("utf-8"))
enc_key = base64.b64decode(package["enc_key"])
nonce = base64.b64decode(package["nonce"])
ciphertext = base64.b64decode(package["ciphertext"])

# 2) Decrypt AES key using RSA-OAEP
aes_key = private_key.decrypt(
enc_key,
padding.OAEP(
mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(),
label=None
)
)

# 3) Decrypt ciphertext with AES-GCM
aesgcm = AESGCM(aes_key)
try:
plaintext = aesgcm.decrypt(nonce, ciphertext, associated_data=None)
except InvalidTag as e:
raise ValueError("Decryption failed or authentication tag invalid") from e

return plaintext


def ensure_keys():
if not (os.path.exists(PRIVATE_KEY_PATH) and os.path.exists(PUBLIC_KEY_PATH)):
generate_rsa_keypair(PRIVATE_KEY_PATH, PUBLIC_KEY_PATH)


def example_usage():
ensure_keys()
public_key = load_public_key(PUBLIC_KEY_PATH)
private_key = load_private_key(PRIVATE_KEY_PATH)

message = b"Hello, World!"
packaged = hybrid_encrypt(public_key, message)
print("Encrypted package (JSON):")
print(packaged.decode("utf-8"))

decrypted = hybrid_decrypt(private_key, packaged)
print("Decrypted message:")
print(decrypted.decode("utf-8"))


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

-- **FileName:**
27CC.py

Content:

```
import os
from typing import Tuple
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives.asymmetric.rsa import RSAPrivateKey, RSAPublicKey
from cryptography.hazmat.primitives import serialization

PRIVATE_KEY_PATH = "private_key.pem"
PUBLIC_KEY_PATH = "public_key.pem"

def ensure_rsa_keys(private_path: str = PRIVATE_KEY_PATH, public_path: str =
PUBLIC_KEY_PATH, key_size: int = 2048) -> Tuple[RSAPrivateKey, RSAPublicKey]:
if not (os.path.exists(private_path) and os.path.exists(public_path)):
private_key: RSAPrivateKey = rsa.generate_private_key(public_exponent=65537,
key_size=key_size)
public_key: RSAPublicKey = private_key.public_key()

with open(private_path, "wb") as f:
f.write(private_key.private_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PrivateFormat.PKCS8,
encryption_algorithm=serialization.NoEncryption()
))
try:
os.chmod(private_path, 0o600)
except Exception:
pass

with open(public_path, "wb") as f:
f.write(public_key.public_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo
))
return private_key, public_key

with open(private_path, "rb") as f:
loaded_private = serialization.load_pem_private_key(f.read(), password=None)
with open(public_path, "rb") as f:
loaded_public = serialization.load_pem_public_key(f.read())

if not isinstance(loaded_private, RSAPrivateKey) or not isinstance(loaded_public, RSAPublicKey):
raise TypeError("Loaded keys are not RSA keys")

return loaded_private, loaded_public
```

-- **FileName:**
28CC.py

Content:

```
import math

def factor_n(n):
for i in range(2, int(math.isqrt(n)) + 1):
if n % i == 0:
return i, n // i
raise ValueError("Failed to factor n")

def extended_gcd(a, b):
if a == 0:
return b, 0, 1
g, y, x = extended_gcd(b % a, a)
return g, x - (b // a) * y, y

def modinv(a, m):
g, x, _ = extended_gcd(a, m)
if g != 1:
raise ValueError("No modular inverse exists")
return x % m

n = 3233
e = 17
ciphertext = 855

p, q = factor_n(n)
phi = (p - 1) * (q - 1)
d = modinv(e, phi)

plaintext = pow(ciphertext, d, n)
print(f"p = {p}, q = {q}")
print(f"d = {d}")
print(f"Decrypted message: {plaintext}")
```

-- **FileName:** 29CC.py

Content:

```
import time
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend

key_sizes = [512, 1024, 2048]
message = b"Test message for RSA performance"

def test_rsa_speed():
for size in key_sizes:
start_gen = time.time()
private_key = rsa.generate_private_key(public_exponent=65537, key_size=size,
backend=default_backend())
public_key = private_key.public_key()
```

```
end_gen = time.time()
gen_time = end_gen - start_gen

start_enc = time.time()
ciphertext = public_key.encrypt(
message,
padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(),
label=None)
)
end_enc = time.time()
enc_time = end_enc - start_enc

start_dec = time.time()
plaintext = private_key.decrypt(
ciphertext,
padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(),
label=None)
)
end_dec = time.time()
dec_time = end_dec - start_dec

print(f"Key size: {size} bits")
print(f"Key generation time: {gen_time:.6f} sec")
print(f"Encryption time: {enc_time:.6f} sec")
print(f"Decryption time: {dec_time:.6f} sec")
print(f"Decrypted message matches original: {plaintext == message}")
print("-"*50)

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

-- **FileName:** 2CC.py

Content:

```
text = "Hello, World!"

# Convert each character to its hex code
hex_string = text.encode("utf-8").hex()

print(hex_string)
```

-- **FileName:** 3CC.py

Content:

```
import hashlib

Result = hashlib.md5(b"Hello, World!").hexdigest()
print(Result)
```

-- **FileName:** 4CC.py

Content:

```
import hashlib

Input = input("Enter text to hash: ")
Result = hashlib.sha1(Input.encode()).hexdigest()
print(Result)
```

-- **FileName:** 5CC.py

Content:

```
import hashlib

Input = input("Enter a password to hash: ")


Result = hashlib.sha256(Input.encode()).hexdigest()
print(Result)
```

-- **FileName:** 6CC.py

Content:

```
import hashlib

def sha256sum(filename):
h = hashlib.sha256()
with open(filename, "rb") as f:
for chunk in iter(lambda: f.read(4096), b""):
h.update(chunk)
return h.hexdigest()

file1 = input("Enter first file path: ").strip()
file2 = input("Enter second file path: ").strip()

try:
hash1 = sha256sum(file1)
hash2 = sha256sum(file2)

print(f"{file1} -> {hash1}")
print(f"{file2} -> {hash2}")

if hash1 == hash2:
print("■ Files are identical (SHA-256 match).")
else:
print("■ Files differ (SHA-256 mismatch).")
except FileNotFoundError as e:
```

```
print(f"Error: {e}")
```

-- **FileName:** 7CC.py

Content:

```
import hashlib

target_hash = input("Enter MD5 hash: ").strip()

for pin in range(10000):
guess = f"{pin:04d}" # ensures 4 digits with leading zeros
h = hashlib.md5(guess.encode()).hexdigest()
if h == target_hash:
print(f"PIN found: {guess}")
break
else:
print("No match found in range 0000-9999")
```

-- **FileName:** 8CC.py

Content:

```
import base64

file_in = input("Enter file path: ").strip()
file_out = file_in + ".b64"

with open(file_in, "rb") as f_in, open(file_out, "wb") as f_out:
base64.encode(f_in, f_out)

print(f"Encoded file saved as {file_out}")
```

-- **FileName:** 9CC.py

Content:

```
import base64
import binascii
import re

def detect_encoding(s: str) -> str:
# Check Hex (only hex chars, even length)
if re.fullmatch(r"[0-9a-fA-F]+", s) and len(s) % 2 == 0:
return "Hex"

# Check Base64 (try decoding safely)
try:
```

```
base64.b64decode(s, validate=True)
return "Base64"
except (binascii.Error, ValueError):
pass

return "Unknown"

input_str = input("Enter a string: ").strip()
encoding = detect_encoding(input_str)
print(f"Detected encoding: {encoding}")
```

-- **FileName:** make_repo_pdf.py

Content:

```
from reportlab.lib.pagesizes import A4
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Table, TableStyle
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib import colors
import os

def read_file_content(filename):
"""Safely read file content as text."""
try:
with open(filename, "r", encoding="utf-8") as f:
return f.read()
except Exception as e:
return f"Error reading file: {e}"

def add_file_section(story, styles, filename, content):
"""Add one file’s section to the PDF."""
section = f"""
------------------------------------------------------------------------------------------------------------
FileName: {filename}


Content:
{content.replace('\n', '
')}

------------------------------------------------------------------------------------------------------------
"""
story.append(Paragraph(section, styles["Normal"]))
story.append(Spacer(1, 20))

def make_pdf():
doc = SimpleDocTemplate("result.pdf", pagesize=A4)
styles = getSampleStyleSheet()
story = []

include_all = input("Do you want to include all files in this folder? (yes/no): ").strip().lower()

if include_all == "yes":
py_files = [f for f in os.listdir() if f.endswith(".py")]
```

```
txt_files = [f for f in os.listdir() if f.endswith(".txt")]

# ---- Add .py files sections ----
if py_files:
story.append(Paragraph("Python Files", styles["Heading2"]))
story.append(Spacer(1, 10))
for f in py_files:
content = read_file_content(f)
add_file_section(story, styles, f, content)

# ---- Add .txt files sections ----
if txt_files:
story.append(Paragraph("Text Files", styles["Heading2"]))
story.append(Spacer(1, 10))
for f in txt_files:
content = read_file_content(f)
add_file_section(story, styles, f, content)

# ---- Python Files Table ----
if py_files:
data = [["Python File Name", "Preview (first 80 chars)"]]
for f in py_files:
code = read_file_content(f)
preview = (code[:80] + "...") if len(code) > 80 else code
data.append([f, preview])
table = Table(data, colWidths=[200, 300])
table.setStyle(TableStyle([
("BACKGROUND", (0, 0), (-1, 0), colors.lightblue),
("GRID", (0, 0), (-1, -1), 1, colors.black),
("ALIGN", (0, 0), (-1, -1), "LEFT"),
]))
story.append(Paragraph("Python Files Summary", styles["Heading3"]))
story.append(table)
story.append(Spacer(1, 20))

# ---- Text Files Table ----
if txt_files:
data = [["Text File Name", "Preview (first 80 chars)"]]
for f in txt_files:
text = read_file_content(f)
preview = (text[:80] + "...") if len(text) > 80 else text
data.append([f, preview])
table = Table(data, colWidths=[200, 300])
table.setStyle(TableStyle([
("BACKGROUND", (0, 0), (-1, 0), colors.lightgreen),
("GRID", (0, 0), (-1, -1), 1, colors.black),
("ALIGN", (0, 0), (-1, -1), "LEFT"),
]))
story.append(Paragraph("Text Files Summary", styles["Heading3"]))
story.append(table)

print("■ Added all .py and .txt files to result.pdf")

else:
# ---- Single file mode ----
filename = input("Enter the file name (.py or .txt): ").strip()
```

```
if not os.path.exists(filename):
print("■ File not found.")
return
if not (filename.endswith(".py") or filename.endswith(".txt")):
print("■ Only .py and .txt files are allowed.")
return

content = read_file_content(filename)
add_file_section(story, styles, filename, content)
print(f"■ Added {filename} to result.pdf")

# ---- Build PDF ----
doc.build(story)
print("■ PDF created successfully: result.pdf")

if __name__ == "__main__":
make_pdf()

--------------------------------------------------------------------------------------------------------------
```

Python Files Summary

Python File Name	Preview (first 80 chars)

10CC.py	import hashlib
	
	text = input("Enter text: ").strip().encode()
	
	hashes = {
	    "M...

11CC.py	from Crypto.Cipher import AES
	from Crypto.Util.Padding import pad, unpad
	import ...

12CC.py	from Crypto.Cipher import AES
	from Crypto.Util.Padding import pad, unpad
	from Cr...

13CC.py	import secrets
	
	# 128-bit key (16 bytes)
	key_128 = secrets.token_bytes(16)
	print...

14CC.py	from Crypto.Util.Padding import pad, unpad block_size = 16 # AES block size in...

15CC.py	from Crypto.Cipher import AES
	from Crypto.Util.Padding import pad
	from Crypto.Ra...

16CC.py	from Crypto.Cipher import AES
	from Crypto.Util.Padding import unpad
	
	key_hex = i...

17CC.py	from Crypto.Cipher import DES
	from Crypto.Util.Padding import pad, unpad
	
	key = ...

18CC.py	from Crypto.Cipher import AES
	from Crypto.Util.Padding import pad
	from Crypto.Ra...

19CC.py	from Crypto.Cipher import AES
	from Crypto.Util.Padding import pad
	from Crypto.Ra...

1CC.py	import base64
	
	original_bytes = b"Hello, World!"
	encoded_bytes = base64.b64encod...

20CC.py	from Crypto.Cipher import AES
	from Crypto.Random import get_random_bytes
	from Cr...

21CC.py	from cryptography.hazmat.primitives.asymmetric import rsa
	from cryptography.hazm...

22CC.py	from cryptography.hazmat.primitives.asymmetric import rsa, padding from cryptogr...
23CC.py	from cryptography.hazmat.primitives.asymmetric import rsa, padding from cryptogr...
24CC.py	from cryptography.hazmat.primitives.asymmetric import rsa, padding from cryptogr...
25CC.py	import os from cryptography.hazmat.primitives.asymmetric import rsa, padding fro...

26CC.py	import os
	import json
	import base64
	
	from cryptography.hazmat.primitives.asymmet...

27CC.py	import os
	from typing import Tuple
	from cryptography.hazmat.primitives.asymmetri...

28CC.py	import math
	
	def factor_n(n):
	    for i in range(2, int(math.isqrt(n)) + 1):
	   ...

29CC.py	import time from cryptography.hazmat.primitives.asymmetric import rsa, padding f...

2CC.py	text = "Hello, World!"
	
	# Convert each character to its hex code
	hex_string = te...

3CC.py	import hashlib
	
	Result = hashlib.md5(b"Hello, World!").hexdigest()
	print(Result)

4CC.py	import hashlib
	
	Input = input("Enter text to hash: ")
	Result = hashlib.sha1(Inp...

5CC.py	import hashlib
	
	Input = input("Enter a password to hash: ")
	
	Result = hashlib.s...

6CC.py	import hashlib
	
	def sha256sum(filename):
	    h = hashlib.sha256()
	    with open(...

7CC.py	import hashlib
	
	target_hash = input("Enter MD5 hash: ").strip()
	
	for pin in rang...

8CC.py	import base64
	
	file_in = input("Enter file path: ").strip()
	file_out = file_in +...

9CC.py	import base64
	import binascii
	import re
	
	def detect_encoding(s: str) -> str:
	   ...

make_repo_pdf.py	from reportlab.lib.pagesizes import A4
	from reportlab.platypus import SimpleDocT...