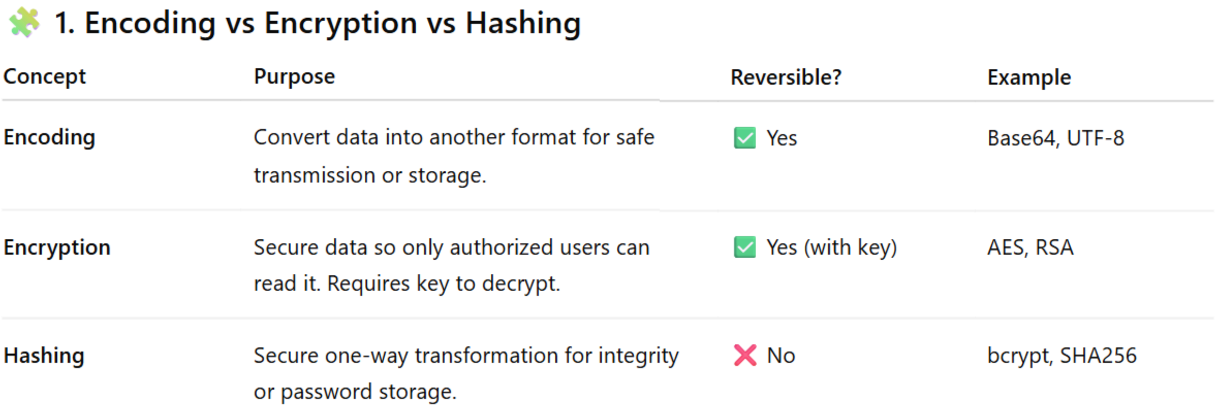
Security in Django: Encoding, Encryption, Hashing, and Password Management (with `bcrypt`)



🧩 1. Encoding vs Encryption vs Hashing

Concept	Purpose	Reversible?	Example
Encoding	Convert data into another format for safe transmission or storage.	✅ Yes	Base64, UTF-8
Encryption	Secure data so only authorized users can read it. Requires key to decrypt.	✅ Yes (with key)	AES, RSA
Hashing	Secure one-way transformation for integrity or password storage.	❌ No	bcrypt, SHA256

🧩 1. What Is Encoding?

Encoding means converting data from one format into another so that it can be:

- Safely transmitted, or
- Properly stored, or
- Understood by a particular system or program.

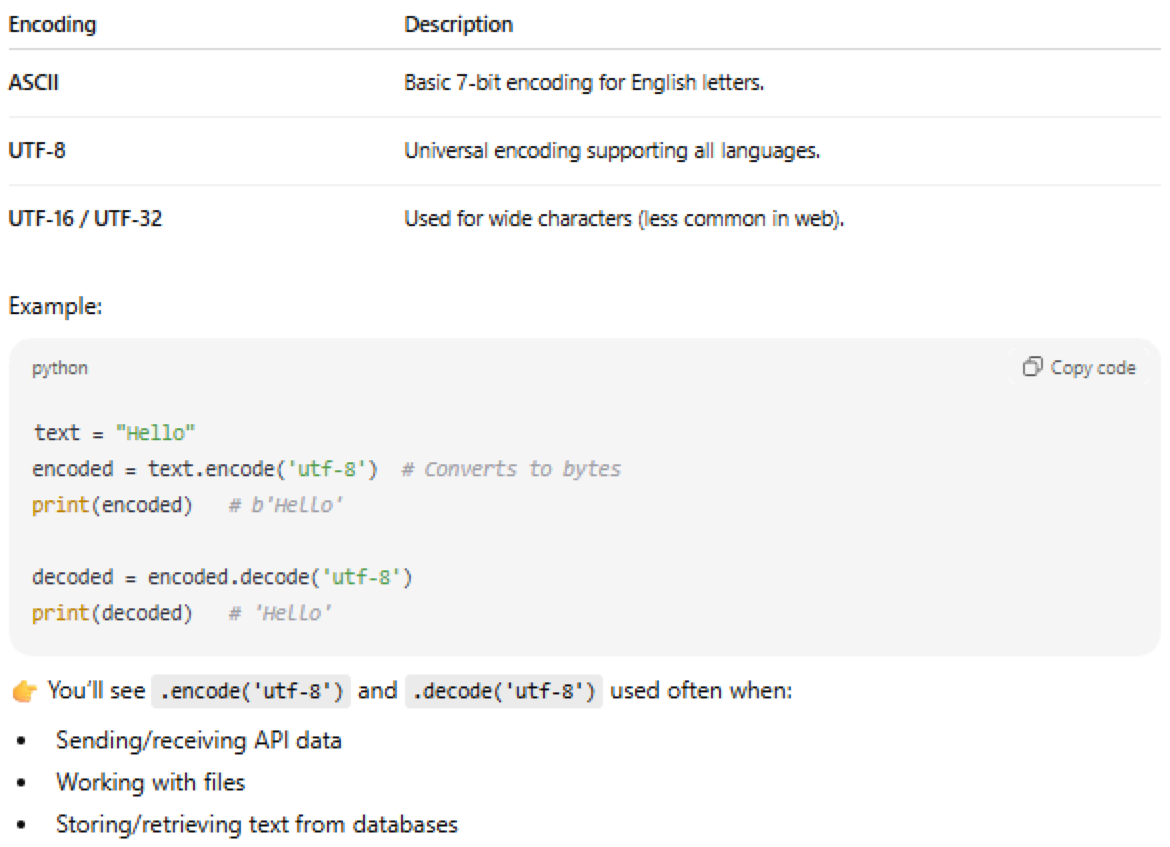
It is not about security — it's about compatibility and readability between systems.

👉 Think of it as translation, not protection.

⚙️ 2. Common Encoding Types

a) Character Encodings

Used to represent text characters as bytes.



Encoding	Description
ASCII	Basic 7-bit encoding for English letters.
UTF-8	Universal encoding supporting all languages.
UTF-16 / UTF-32	Used for wide characters (less common in web).

Example:

```python
text = "Hello"
encoded = text.encode('utf-8')  # Converts to bytes
print(encoded)   # b'Hello'

decoded = encoded.decode('utf-8')
print(decoded)   # 'Hello'
```

👉 You'll see `.encode('utf-8')` and `.decode('utf-8')` used often when:

- Sending/receiving API data
- Working with files
- Storing/retrieving text from databases

If UTF-8 didn’t exist, English, Hindi, Tamil, emojis, etc. would all break or show as ����.

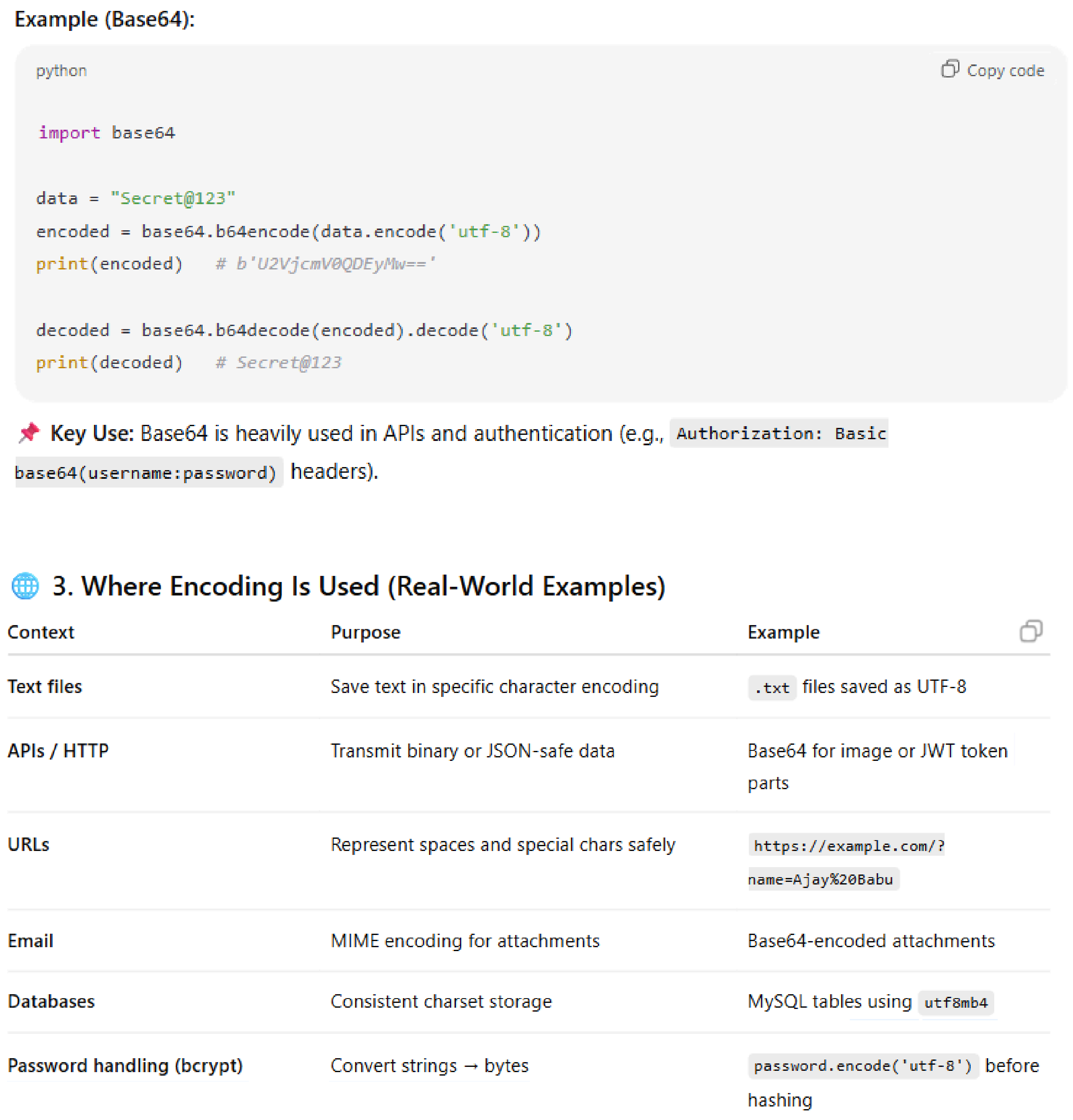
Binary-to-Text Encodings

Used to represent binary data (images, PDFs, passwords, etc.) in text format — usually for network transmission or JSON storage.

Common examples:

- Base64: To convert **any kind of data** (text, image, password hash, PDF, etc.) into safe **text format** that can be sent in:
  - JSON
  - HTTP headers
  - URLs

    - Email
    - Tokens (JWT)
- URL encoding (percent-encoding)
- Hexadecimal encoding



Example (Base64):

```python
import base64

data = "Secret@123"
encoded = base64.b64encode(data.encode('utf-8'))
print(encoded)   # b'U2VjcmV0QDEyMw=='

decoded = base64.b64decode(encoded).decode('utf-8')
print(decoded)   # Secret@123
```

📌 **Key Use:** Base64 is heavily used in APIs and authentication (e.g., `Authorization: Basic base64(username:password)` headers).

🌐 3. Where Encoding Is Used (Real-World Examples)

Context	Purpose	Example
Text files	Save text in specific character encoding	`.txt` files saved as UTF-8
APIs / HTTP	Transmit binary or JSON-safe data	Base64 for image or JWT token parts
URLs	Represent spaces and special chars safely	`https://example.com/?name=Ajay%20Babu`
Email	MIME encoding for attachments	Base64-encoded attachments
Databases	Consistent charset storage	MySQL tables using `utf8mb4`
Password handling (bcrypt)	Convert strings → bytes	`password.encode('utf-8')` before hashing

Hashing:

🔐 2. What is Hashing?

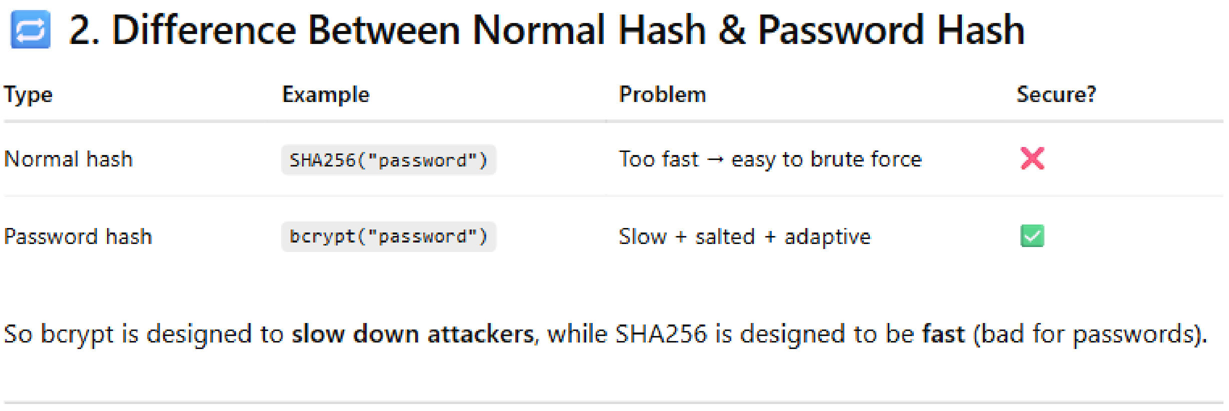
Hashing = one-way conversion of a string (like a password) → fixed-length scrambled form.

It’s used to:

- Store passwords securely.
- Verify data integrity.

Key traits:

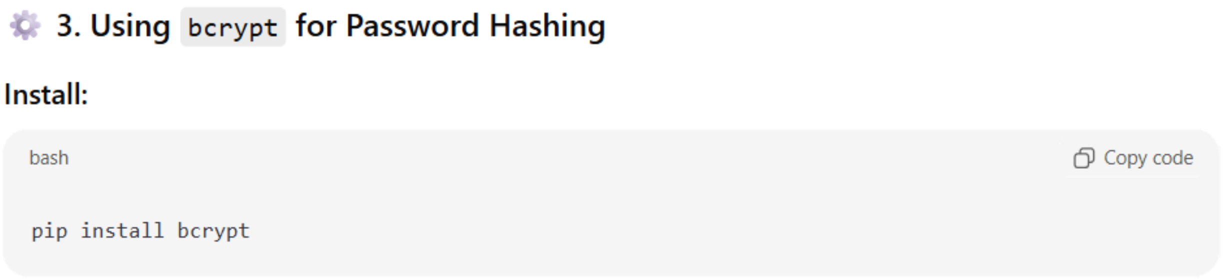
- **Irreversible**
- **Deterministic** (same input = same output)
- **Unique (ideally)** – small changes create big differences.
- **Slow intentionally** — to prevent brute-force attacks.



🔁 2. Difference Between Normal Hash & Password Hash

Type	Example	Problem	Secure?
Normal hash	`SHA256("password")`	Too fast → easy to brute force	❌
Password hash	`bcrypt("password")`	Slow + salted + adaptive	✅

So bcrypt is designed to **slow down attackers**, while SHA256 is designed to be **fast** (bad for passwords).

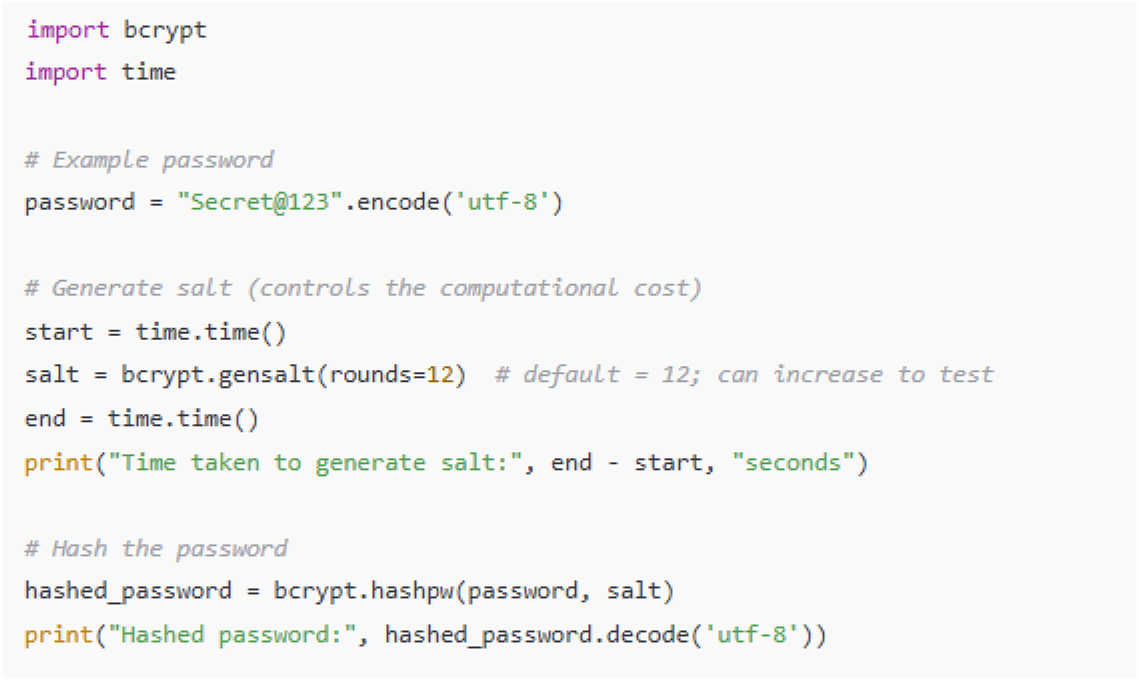


⚙️ 3. Using `bcrypt` for Password Hashing

Install:

```bash
pip install bcrypt
```

Working:

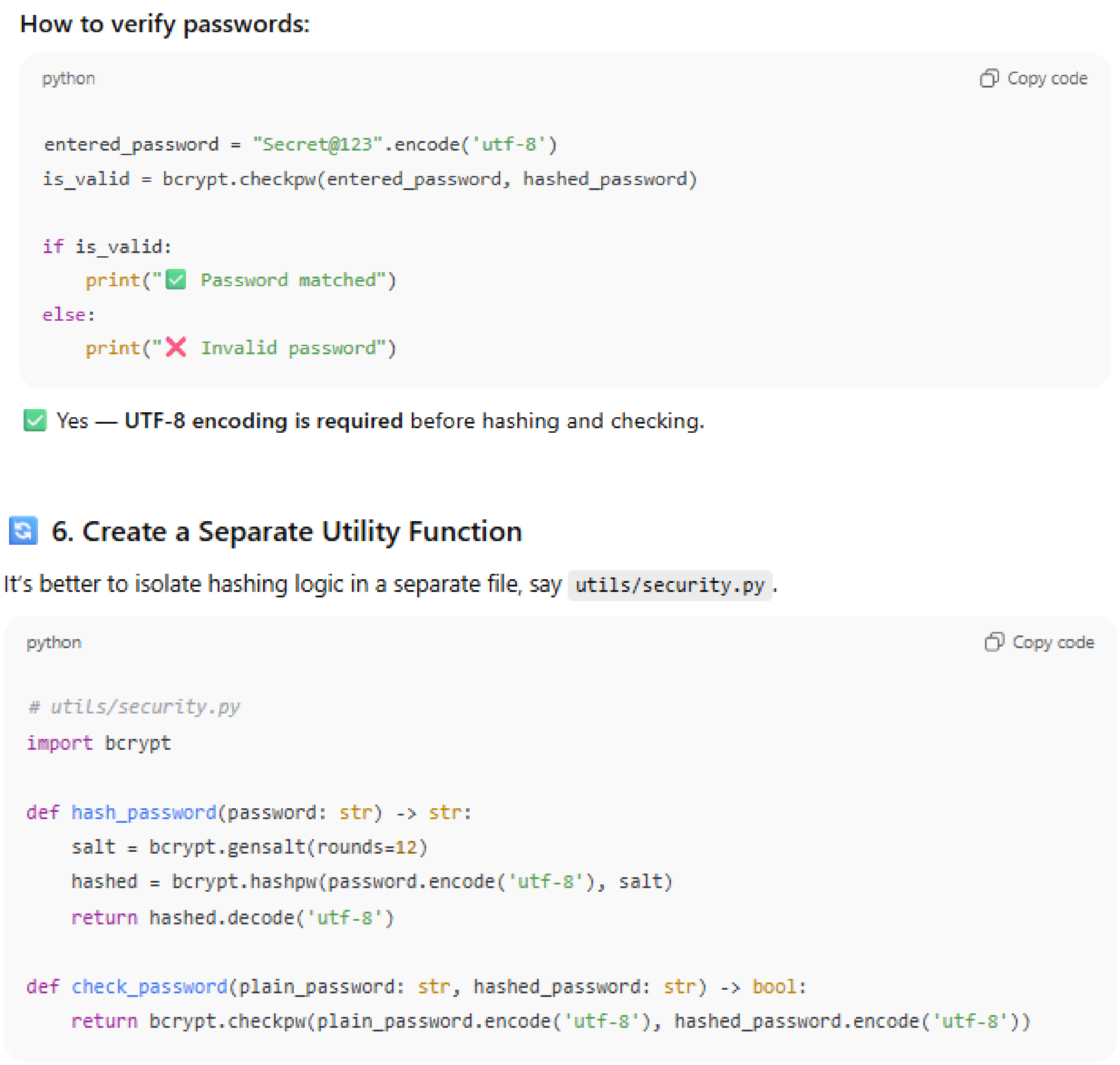


```python
import bcrypt
import time

# Example password
password = "Secret@123".encode('utf-8')

# Generate salt (controls the computational cost)
start = time.time()
salt = bcrypt.gensalt(rounds=12)  # default = 12; can increase to test
end = time.time()
print("Time taken to generate salt:", end - start, "seconds")

# Hash the password
hashed_password = bcrypt.hashpw(password, salt)
print("Hashed password:", hashed_password.decode('utf-8'))
```

Note: Increasing `rounds` (cost factor) makes hashing slower but more secure.
Example:

- rounds=10 → ~0.1s
- rounds=14 → ~2s
  Ideal range: 12–14 (depending on system performance).

We are encoding because **bcrypt cannot hash strings — it only accepts bytes**



How to verify passwords:

```python
entered_password = "Secret@123".encode('utf-8')
is_valid = bcrypt.checkpw(entered_password, hashed_password)

if is_valid:
    print("✅ Password matched")
else:
    print("❌ Invalid password")
```

✅ Yes — **UTF-8 encoding is required** before hashing and checking.

🔄 6. Create a Separate Utility Function

It's better to isolate hashing logic in a separate file, say `utils/security.py`.

```python
# utils/security.py
import bcrypt

def hash_password(password: str) -> str:
    salt = bcrypt.gensalt(rounds=12)
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')

def check_password(plain_password: str, hashed_password: str) -> bool:
    return bcrypt.checkpw(plain_password.encode('utf-8'), hashed_password.encode('utf-8'))
```

Usage:

```python
from utils.security import hash_password, check_password

# Hash during registration
hashed = hash_password('Secret@123')

# Verify during login
if check_password('Secret@123', hashed):
    print("✅ Login success")
else:
    print("❌ Wrong password")
```

🔥 3. How Bcrypt Works Internally (Step by Step)

Let's assume:

```python
salt = bcrypt.gensalt(rounds=12)
hashed = bcrypt.hashpw(password, salt)
```

📌 Step 1: Generate Random Salt

`gensalt()` creates a **16-byte random salt**, like:

```perl
$2b$12$Wm7qszM3yx8eWlJgW9hP5u
```

Salt prevents **rainbow table attacks** and makes every user's hash unique even if password is same.

Example:

User	Password	Hash
A	"123456"	`$2b$12$qj93...ss2`
B	"123456"	`$2b$12$u71y...pW8`

Same password, different hash ✅

📌 Step 2: Key Stretching (Expensive Computation)

bcrypt runs the Blowfish cipher algorithm **2^rounds times**.

So if rounds = 12:

yaml Copy code

```yaml
2^12 = 4096 iterations
```

If rounds = 15:

Copy code

```
2^15 = 32768 iterations (8x slower)
```

That's why `rounds` value is also called **work factor** (or cost factor).

✔️ Higher cost = more security
❌ Too high = slow login

📌 Step 3: Output Format of bcrypt Hash

Example hash:

perl Copy code

```perl
$2b$12$Wm7qszM3yx8eWlJgW9hP5uGBc90ywZxpx4SZ3dlC3kN6ZCiX0j2S
```

Breakdown:

Part	Meaning
`$2b$`	bcrypt version
`12`	cost factor (2^12 loops)
`Wm7qszM3yx8eWlJgW9hP5u`	salt (22 chars base64)
`GBc90ywZxpx4SZ3dlC3kN6ZCiX0j2S`	hashed password

So everything (algorithm + cost + salt + hash) is stored **in one string.**

That's why we don't have to store salt separately.