Hashing algorithms

There are many types of hashing algorithms, like cryptography, checksums, data structures, similarity detection, compression and encoding, analytics, monitoring, load balancing, etc.

I’m going to talk about the following segregation:

1. Cryptography hash functions
2. Password hashing algorithms
3. Checksums hashing
4. Data Structures?
5. Load balancing?
6. Similarity Detection?

Cryptography hash functions (CHF) PREREQUISITES

- Transforms input data into a fixed-length sequence of characters
- designed to be fast.
- deterministic—the hash function consistently computes the same hash for the same input.
- one-way—the algorithm is made irreversible in the sense that it is computationally impossible to recover the original input from its hash value.
- collision —The function minimizes the chance that two distinct inputs will produce identical hash values

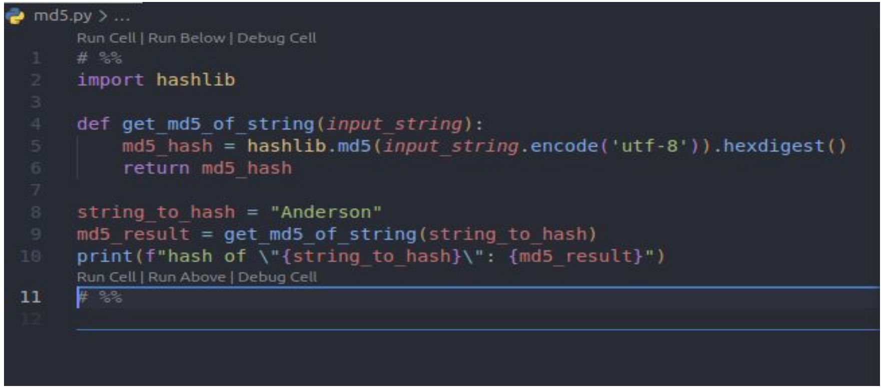
Cryptography hash functions (CHF)

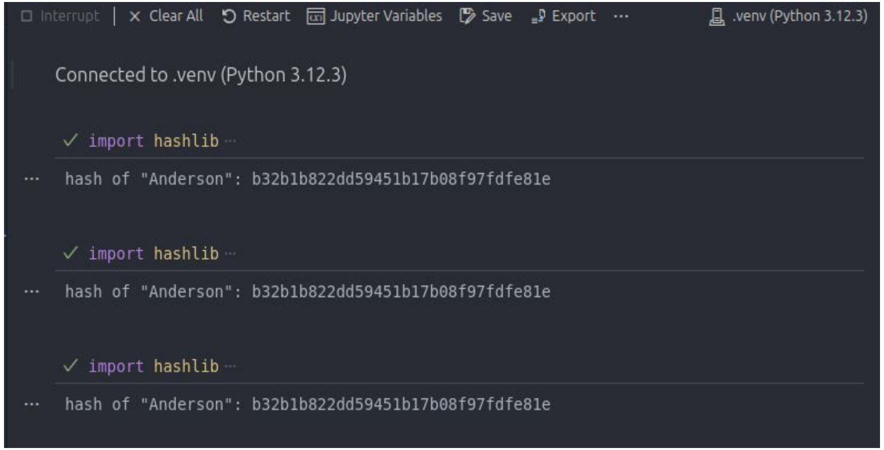
MOST USED

1. MD5
2. SHA-1
3. SHA-2
4. SHA-3
5. BLAKE2
6. BLAKE3

Cryptography hash functions (CHF)

MD5 (MESSAGE DIGEST ALGORITHM 5)





Cryptography hash functions (CHF)

MD5 COLLISION

```
Run Cell | Run Above | Debug Cell
# %%
## simulating a md5 colission
import hashlib
# Source: https://www.johndcook.com/blog/2024/03/20/md5-hash-collision/

string1 = b"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"
string2 = b"TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"

if(string1 != string2):
    print("Input strings are different\n")

hash1 = hashlib.md5(string1).hexdigest()
hash2 = hashlib.md5(string2).hexdigest()

print(f"MD5 Hash 1: {hash1}\n")
print(f"MD5 Hash 2: {hash2}")

if hash1 == hash2:
    print("\nMD5 Collision Detected")
```

```
Interrupt | × Clear All  Restart  Jupyter Variables ...  .venv (Python 3.12.3)

Connected to .venv (Python 3.12.3)

✓ ## simulating a md5 colission ...
... Input strings are different

MD5 Hash 1: faad49866e9498fc1719f5289e7a0269

MD5 Hash 2: faad49866e9498fc1719f5289e7a0269

MD5 Collision Detected
```

Cryptography hash functions (CHF)

SHA-1 (SECURE HASH ALGORITHM 2)

sha-1.py > ...

Run Cell | Run Below | Debug Cell

```python
# %%
## creating a sha-1
import hashlib
string_to_hash = "Anderson"
hash = hashlib.sha1(string_to_hash.encode()).hexdigest()
print(hash)
```

Run Cell | Run Above | Debug Cell

```python
# %%
```

Interrupt | Clear All | Restart | Jupyter Variables | Save | Export ... | .venv (Python 3.12.3)

Connected to .venv (Python 3.12.3)

```
## creating a sha-1 ...
59355083dfe3bdbf05328a45e6fd9f21e1d28814

## creating a sha-1 ...
59355083dfe3bdbf05328a45e6fd9f21e1d28814

## creating a sha-1 ...
59355083dfe3bdbf05328a45e6fd9f21e1d28814
```

Cryptography hash functions (CHF)

SHA-1 COLLISION

```
Run Cell | Run Above | Debug Cell
# %%
import hashlib

def check_if_files_is_different():
    with open("./files/shattered-1.pdf", "rb") as f1, open("./files/shattered-2.pdf", "rb") as f2:
        if f1.read() != f2.read():
            print("The files are different\n")

def sha1_hash(filename):
    with open(filename, 'rb') as f:
        return hashlib.sha1(f.read()).hexdigest()

check_if_files_is_different()
hash1 = sha1_hash("./files/shattered-1.pdf")
hash2 = sha1_hash("./files/shattered-2.pdf")

print("hash1: ", hash1)
print("hash2: ", hash2)

if hash1 == hash2:
    print("\nCollision detected")
```

```
Interrupt | Clear All  Restart  Jupyter Variables  Save  Export  Expand  Collapse    .venv (Python 3.12.3)

Connected to .venv (Python 3.12.3)

✓ import hashlib ...

... The files are different

hash1:  38762cf7f55934b34d179ae6a4c80cadccbb7f0a
hash2:  38762cf7f55934b34d179ae6a4c80cadccbb7f0a

Collision detected
```

Cryptography hash functions (CHF)

SHA-2 (SECURE HASH ALGORITHM 2)

Run Cell | Run Below | Debug Cell | You, yesterday | 1 author (You)

```python
# %%
## sha-2 hash-224
import hashlib

def generate_sha_hash(input_string):
    return hashlib.sha224(input_string.encode('utf-8')).hexdigest()

string_to_hash = "Anderson"
hashed_value = generate_sha_hash(string_to_hash)

print(f"hash224: {hashed_value}")
```

```
✓ ## sha-2 hash-224 ···
hash224: 6bca1d6c60885d0f5f79907ccd2c2af99f32fcc20daeee9f9acfd194

✓ ## sha-2 hash-224 ···
hash224: 6bca1d6c60885d0f5f79907ccd2c2af99f32fcc20daeee9f9acfd194

✓ ## sha-2 hash-224 ···
hash224: 6bca1d6c60885d0f5f79907ccd2c2af99f32fcc20daeee9f9acfd194
```

Cryptography hash functions (CHF)

SHA-3 (SECURE HASH ALGORITHM 3)

```
Run Cell | Run Below | Debug Cell | You, yesterday | 1 author (You)
# %%
import hashlib

def generate_sha3_hash(input_string):
    return hashlib.sha3_224(input_string.encode('utf-8')).hexdigest()

string_to_hash = "Anderson"
hash_result = generate_sha3_hash(string_to_hash)
print(f"hash256: {hash_result}")
```

```
✓ import hashlib ···
··· hash224: ef9c6260bcd3c2e0c5b9c34d2dbbf55a71882f2e3d32b5b7720651c8

✓ import hashlib ···
··· hash224: ef9c6260bcd3c2e0c5b9c34d2dbbf55a71882f2e3d32b5b7720651c8

✓ import hashlib ···
··· hash224: ef9c6260bcd3c2e0c5b9c34d2dbbf55a71882f2e3d32b5b7720651c8
```

Cryptography hash functions (CHF)

BLAKE2

blake2.py > ...

Run Cell | Run Below | Debug Cell | You, 3 seconds ago | 1 author (You)

```python
# %%
## Can produce a hash digest of up to 64 bytes (512 bits).
import hashlib

string_to_hash = b"Anderson"
blake_object = hashlib.blake2b(digest_size=64)
blake_object.update(string_to_hash)
hash_result = blake_object.hexdigest()

print(f"Hash: {hash_result}")

```

Run Cell | Run Above | Debug Cell

Interactive - blake2.py

Interrupt | Clear All | Restart | Jupyter Variables | Save | Export | ... | .venv (Python 3.12.3)

Connected to .venv (Python 3.12.3)

```
✓ ## Can produce a hash digest of up to 64 bytes (512 bits). ...
... Hash: ff3d180526da68d6c5dccb6e9648eaeddbd6644f2abb056e7e442f16de7b9f4e

✓ ## Can produce a hash digest of up to 64 bytes (512 bits). ...
... Hash: ff3d180526da68d6c5dccb6e9648eaeddbd6644f2abb056e7e442f16de7b9f4e

✓ ## Can produce a hash digest of up to 64 bytes (512 bits). ...
... Hash: ff3d180526da68d6c5dccb6e9648eaeddbd6644f2abb056e7e442f16de7b9f4e
```

Cryptography hash functions (CHF)

BLAKE3

cryptography_hash_functions > blake_3.py > ...

Run Cell | Run Below | Debug Cell

```python
# %%
# basic usage
import blake3

data = b"Anderson"
hash_object = blake3.blake3(data)
hash_hex = hash_object.hexdigest()
print(f"Hash: {hash_hex}")
```

Run Cell | Run Above | Debug Cell

```python
# %%
# Keyed hashing
import blake3

my_key = b"my_key"
keyed_hash = blake3.blake3(key=my_key)
keyed_hash.update(b"Sensitive information.")
print(f"Keyed Hash: {keyed_hash.hexdigest()}")
```

Run Cell | Run Above | Debug Cell

```python
# %%
import blake3

extended_output = blake3.blake3(b"input_for_xof").digest(length=64)
print(f"Extended Output: {extended_output.hex()}")

seeked_output = blake3.blake3(b"input_for_xof").digest(length=10, seek=10)
print(f"Seeked Output (bytes 10-19): {seeked_output.hex()}")
```

Run Cell | Run Above | Debug Cell

```python
# %%
import blake3

large_input = bytearray(10 * 1024 * 1024)  # 10 MB of data

hash_auto_threads = blake3.blake3(large_input, max_threads=2).hexdigest()
print(f"Hash: {hash_auto_threads}")
```

Cryptographic Hash Algorithms

TRADEOFFS

CRITERION	MD5	SHA-1	SHA-2	SHA-3	BLAKE2	BLAKE3
Security	Broken	Broken	Secure	Secure	Secure	Very Secure
Collision	Weak	Weak	Strong	Strong	Strong	Very Strong
Speed	Fast	Medium	Medium	Slower	Very Fast	Extremely Fast
Length Extension	Vulnerable	Vulnerable	Vulnerable	Resistant	Resistant	Resistant

Password hashing algorithms PREREQUISITES

- one-way process
- deterministic nature
- resistance to collisions
- computational cost
- salting—Salting is a technique where a random value (the salt) is added to the password before hashing.

Password hashing algorithms

MOST USED

1. Bcrypt
2. Argon2
3. PBKDF2

Password hashing algorithms

BCRYPT

password_hashing_algorithms > bcrypt_.py > ...

```python
# %%
import bcrypt

password = b"mysecretpassword123"

# The 'rounds' parameter in gensalt() determines the computational cost.
# Higher rounds mean more secure but slower hashing. Default is 12.
salt = bcrypt.gensalt(rounds=12)

# 4. Hash the password using the salt
hashed_password = bcrypt.hashpw(password, salt)

print(f"Original password: {password.decode('utf-8')}")
print(f"Hashed password: {hashed_password.decode('utf-8')}")

# Verify a password against the stored hash
# Simulate a user entering their password for login
entered_password = "mysecretpassword123"
entered_byte_password = entered_password.encode('utf-8')

print(f"\nTrying to authenticate with password: {entered_password}")
if bcrypt.checkpw(entered_byte_password, hashed_password):
    print("Password match! User authenticated.\n")

# Example with an incorrect password
incorrect_password = "wrongpassword"
print(f"Trying to authenticate with incorrect password: {incorrect_password}")
incorrect_byte_password = incorrect_password.encode('utf-8')

if bcrypt.checkpw(incorrect_byte_password, hashed_password):
    print("Password match! (This should not happen for incorrect password)")
else:
    print("Incorrect password. Authentication failed (as expected).")
# %%
```

Password hashing algorithms

ARGON2

password_hashing_algorithms > argon2_.py > ...

Run Cell | Run Below | Debug Cell

```python
# %%
import argon2

time_cost = 16
memory_cost = 2**15  # 32768 KiB (32MB)
parallelism = 2
hash_len = 32
salt = b'some salt'
password = b'password'

argon2_hasher = argon2.PasswordHasher(
    time_cost=time_cost,
    memory_cost=memory_cost,
    parallelism=parallelism,
    hash_len=hash_len,
    salt_len=16 # defaults to 16
)

hashed_password = argon2_hasher.hash(password.decode('utf-8'))
print("Argon2 hash", hashed_password)

# Verify the correct password
password_is_right = argon2_hasher.verify(hashed_password, "password")
print("Password is right ", password_is_right)

# incorrect password
try:
    argon2_hasher.verify(hashed_password, "wrong password")
except argon2.exceptions.VerifyMismatchError:
    print("Password is wrong")
```

Run Cell | Run Above | Debug Cell

Password hashing algorithms

PBKDF2 (PASSWORD-BASED KEY DERIVATION FUNCTION 2)

```
Run Cell | Run Below | Debug Cell | You, 6 seconds ago | 1 author (You)
# %%
import hashlib
import os

def hash_password(password: str, salt: bytes = None, iterations: int = 260000) -> bytes:
    if salt is None:
        salt = os.urandom(16)

    derived_key = hashlib.pbkdf2_hmac(
        'sha256',              # Hashing algorithm (e.g., 'sha256', 'sha512')
        password.encode('utf-8'),  # Password must be bytes
        salt,                  # Salt must be bytes
        iterations,            # Number of iterations
        dklen=32               # Desired length of the derived key in bytes (e.g., 32 for SHA256)
    )
    return salt + derived_key

def verify_password(stored_hash: bytes, input_password: str, iterations: int = 260000) -> bool:
    salt = stored_hash[:16]  # Extract the salt (first 16 bytes)
    stored_derived_key = stored_hash[16:]  # Extract the derived key

    # Re-derive the key using the input password and extracted salt
    re_derived_key = hashlib.pbkdf2_hmac(
        'sha256',
        input_password.encode('utf-8'),
        salt,
        iterations,
        dklen=32
    )
    return re_derived_key == stored_derived_key

user_password = "mySecretPassword123!"
      You, yesterday • pbkdf2 …
hashed_password_data = hash_password(user_password)
print(f"Hashed password (salt + derived key in hex): {hashed_password_data.hex()}")

if verify_password(hashed_password_data, user_password):
    print("Password verification successful!")

if not verify_password(hashed_password_data, "wrongPassword"):
    print("Incorrect password verification failed (as expected).")
Run Cell | Run Above | Debug Cell
# %%
```

Password hashing algorithms

TRADEOFFS

CRITERIA	PBKDF2	ARGON2	BCRYPT
Resistance to Brute Force	Medium (CPU-bound)	High (CPU + memory-bound)	Good (CPU-bound)
Resistance to GPU/ASIC	Low to Medium	High	Medium
Configurable Parameters	Iterations	Iterations, memory, parallelism	Cost factor (log rounds)
Speed	Fast	Slow (intentionally)	Reasonably fast
Memory Usage	Low	High (configurable)	Very low

Checksum hashing algorithms PREREQUISITES

- fixed-size output
- one-way function
- sensitivity to input changes
- collision resistance
- deterministic

Checksum hashing algorithms

MOST USED

1. CRC32
2. Adler-32
3. MD5
4. SHA-1 (variants)

Checksum hashing algorithms

CRC32 (CYCLIC REDUNDANCY CHECK 32)

Run Cell | Run Below | Debug Cell | You, 6 hours ago | 1 author (You)

```python
# %%
import zlib

string_to_hash = "Anderson"
str_hash = zlib.crc32(string_to_hash.encode('utf-8'))
print(f"hash of '{string_to_hash}': {str_hash}")

```

Run Cell | Run Above | Debug Cell

```python
# %%      You, 7 hours ago • crc32
import zlib

file_hash = 0
with open("../files/test_file.txt",'rb') as f:
    while True:
        chunk = f.read(4096)  # Read 4KB at a time
        if not chunk:
            break
        file_hash = zlib.crc32(chunk)

print(f"hash of 'test_file.txt': {file_hash}")
```

Run Cell | Run Above | Debug Cell

```python
# %%
```

```
import zlib ···
hash of 'Anderson': 1187322618

import zlib ···
hash of 'test_file.txt': 2473840611
```

Checksum hashing algorithms

ADLER-32 (MARK ADLER'S 32-BIT)

```
checksums > adler-32.py > ...
Run Cell | Run Below | Debug Cell
# %%
import zlib

string_to_hash = "Anderson"
hash = zlib.adler32(string_to_hash.encode('utf-8'))
print(f"hash: {hash}")

```

```
Run Cell | Run Above | Debug Cell
# %%
import zlib

file_checksum = 0
with open("../files/test_file.txt",'rb') as f:
    while True:
        chunk = f.read(4096)  # Read 4KB at a time
        if not chunk:
            break
        file_checksum = zlib.adler32(chunk)

print(f"hash of 'test_file.txt': {file_checksum}")
Run Cell | Run Above | Debug Cell
# %%
```

Password hashing algorithms

TRADEOFFS

CRITERIA	CRC32	Adler-32
Speed	Slower	Faster
Checksum Size	32 bits	32 bits
Collision Resistance	Better	Worse
Data Size Sensitivity	Performs well on all sizes	Performs poorly on small inputs

thanks, best regards, **Anderson Babinski**

github.com/andeerlb

linkedin.com/andersonbabinski

This is a repo linked this slide
github.com/andeerlb/hashing-algorithms