


Hashing algorithms

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

A large, dark blue, abstract shape that starts from the bottom left and extends diagonally upwards towards the right, filling the lower half of the slide.

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)

```
md5.py > ...
Run Cell | Run Below | Debug Cell
1 # %%
2 import hashlib
3
4 def get_md5_of_string(input_string):
5     md5_hash = hashlib.md5(input_string.encode('utf-8')).hexdigest()
6     return md5_hash
7
8 string_to_hash = "Anderson"
9 md5_result = get_md5_of_string(string_to_hash)
10 print(f"hash of \"{string_to_hash}\": {md5_result}")
Run Cell | Run Above | Debug Cell
11 # %%
12
```

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

Connected to .venv (Python 3.12.3)

```
✓ import hashlib ...
... hash of "Anderson": b32b1b822dd59451b17b08f97fdfe81e

✓ import hashlib ...
... hash of "Anderson": b32b1b822dd59451b17b08f97fdfe81e

✓ import hashlib ...
... hash of "Anderson": b32b1b822dd59451b17b08f97fdfe81e
```

Cryptography hash functions (CHF)

MD5 COLLISION

```
Run Cell | Run Above | Debug Cell
12 # %%
13 ## simulating a md5 collision
14 import hashlib
15 # Source: https://www.johndcook.com/blog/2024/03/20/md5-hash-collision/
16
17 string1 = b"TEXTCOLLBYf6iJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"
18 string2 = b"TEXTCOLLBYf6iJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"
19
20 if(string1 != string2):
21     print("Input strings are different\n")
22
23 hash1 = hashlib.md5(string1).hexdigest()
24 hash2 = hashlib.md5(string2).hexdigest()
25
26 print(f"MD5 Hash 1: {hash1}\n")
27 print(f"MD5 Hash 2: {hash2}")
28
29 if hash1 == hash2:
30     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 collision ...
... 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
1  # %%
2  ## creating a sha-1
3  import hashlib
4  string_to_hash = "Anderson"
5  hash = hashlib.shal(string_to_hash.encode()).hexdigest()
6  print(hash)
Run Cell | Run Above | Debug Cell
7  # %%
8
```

```
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
7  # %%
8  import hashlib
9
10 def check_if_files_is_different():
11     with open("./files/shattered-1.pdf", "rb") as f1, open("./files/shattered-2.pdf", "rb") as f2:
12         if f1.read() != f2.read():
13             print("The files are different\n")
14
15 def sha1_hash(filename):
16     with open(filename, 'rb') as f:
17         return hashlib.shal(f.read()).hexdigest()
18
19 check_if_files_is_different()
20 hash1 = sha1_hash("./files/shattered-1.pdf")
21 hash2 = sha1_hash("./files/shattered-2.pdf")
22
23 print("hash1: ", hash1)
24 print("hash2: ", hash2)
25
26 if hash1 == hash2:
27     print("\nCollision detected")
```

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

Connected to .env (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)

```
sha-1.py > ...
Run Cell | Run Below | Debug Cell
1 # %%
2 ## creating a sha-1
3 import hashlib
4 string_to_hash = "Anderson"
5 hash = hashlib.shal(string_to_hash.encode()).hexdigest()
6 print(hash)
Run Cell | Run Above | Debug Cell
7 # %%
8
```

```
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)

BLAKE2

```
blake2.py > ...
Run Cell | Run Below | Debug Cell | You, 3 seconds ago | 1 author (You)
1 # %%
2 ## Can produce a hash digest of up to 64 bytes (512 bits).
3 import hashlib
4
5 string_to_hash = b"Anderson"
6 blake_object = hashlib.blake2b(digest_size=64)
7 blake_object.update(string_to_hash)
8 hash_result = blake_object.hexdigest()
9
10 print(f"Hash: {hash_result}")
11
```

Interactive - blake2.py X

Interrupt | X 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: ff3d180526da68d6c5dcc6e9648eaeddbd6644f2abb056e7e442f16de7b9f4e

✓ ## Can produce a hash digest of up to 64 bytes (512 bits). ...
... Hash: ff3d180526da68d6c5dcc6e9648eaeddbd6644f2abb056e7e442f16de7b9f4e

✓ ## Can produce a hash digest of up to 64 bytes (512 bits). ...
... Hash: ff3d180526da68d6c5dcc6e9648eaeddbd6644f2abb056e7e442f16de7b9f4e
```

Cryptography hash functions (CHF)

BLAKE3

cryptography_hash_functions > blake_3.py > ...

Run Cell | Run Below | Debug Cell

```
1 # %%  
2 # basic usage  
3 import blake3  
4  
5 data = b"Anderson"  
6 hash_object = blake3.blake3(data)  
7 hash_hex = hash_object.hexdigest()  
8 print(f"Hash: {hash_hex}")
```

Run Cell | Run Above | Debug Cell

```
9 # %%  
10 # Keyed hashing  
11 import blake3  
12  
13 my_key = b"my_key"  
14 keyed_hash = blake3.blake3(key=my_key)  
15 keyed_hash.update(b"Sensitive information.")  
16 print(f"Keyed Hash: {keyed_hash.hexdigest()}")  
Run Cell | Run Above | Debug Cell
```

```
17 # %%  
18 import blake3  
19  
20 extended_output = blake3.blake3(b"input for xof").digest(length=64)  
21 print(f"Extended Output: {extended_output.hex()}")  
22  
23 seeked_output = blake3.blake3(b"input for xof").digest(length=10, seek=10)  
24 print(f"Seeked Output (bytes 10-19): {seeked_output.hex()}")  
Run Cell | Run Above | Debug Cell  
25 # %%  
26 import blake3  
27  
28 large_input = bytearray(10 * 1024 * 1024) # 10 MB of data  
29  
30 hash_auto_threads = blake3.blake3(large_input, max_threads=2).hexdigest()  
31 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 > ...
Run Cell | Run Below | Debug Cell
1 # %%
2 import bcrypt
3
4 password = b"mysecretpassword123"
5
6 # The 'rounds' parameter in gensalt() determines the computational cost.
7 # Higher rounds mean more secure but slower hashing. Default is 12.
8 salt = bcrypt.gensalt(rounds=12)
9
10 # 4. Hash the password using the salt
11 hashed_password = bcrypt.hashpw(password, salt)
12
13 print(f"Original password: {password.decode('utf-8')}")
14 print(f"Hashed password: {hashed_password.decode('utf-8')}")
15
16 # Verify a password against the stored hash
17 # Simulate a user entering their password for login
18 entered_password = "mysecretpassword123"
19 entered_byte_password = entered_password.encode('utf-8')
20
21 print(f"\nTrying to authenticate with password: {entered_password}")
22 if bcrypt.checkpw(entered_byte_password, hashed_password):
23     print("Password match! User authenticated.\n")
24
25 # Example with an incorrect password
26 incorrect_password = "wrongpassword"
27 print(f"Trying to authenticate with incorrect password: {incorrect_password}")
28 incorrect_byte_password = incorrect_password.encode('utf-8')
29
30 if bcrypt.checkpw(incorrect_byte_password, hashed_password):
31     print("Password match! (This should not happen for incorrect password)")
32 else:
33     print("Incorrect password. Authentication failed (as expected).")
34 # %%
Run Cell | Run Above | Debug Cell
```

Password hashing algorithms

ARGON2

```
password_hashing_algorithms > argon2_py > ...  
Run Cell | Run Below | Debug Cell  
1 # %%  
2 import argon2  
3  
4 time_cost = 16  
5 memory_cost = 2**15 # 32768 KiB (32MB)  
6 parallelism = 2  
7 hash_len = 32  
8 salt = b'some salt'  
9 password = b'password'  
10  
11 argon2_hasher = argon2.PasswordHasher(  
12     time_cost=time_cost,  
13     memory_cost=memory_cost,  
14     parallelism=parallelism,  
15     hash_len=hash_len,  
16     salt_len=16 # defaults to 16  
17 )  
18  
19 hashed_password = argon2_hasher.hash(password.decode('utf-8'))  
20 print("Argon2 hash", hashed_password)  
21  
22 # Verify the correct password  
23 password_is_right = argon2_hasher.verify(hashed_password, "password")  
24 print("Password is right ", password_is_right)  
25  
26 # incorrect password  
27 try:  
28     argon2_hasher.verify(hashed_password, "wrong password")  
29 except argon2.exceptions.VerifyMismatchError:  
30     print("Password is wrong")  
Run Cell | Run Above | Debug Cell
```


Password hashing algorithms

PBKDF2 (PASSWORD-BASED KEY DERIVATION FUNCTION 2)

```
password_hashing_algorithms > argon2_py > ...  
Run Cell | Run Below | Debug Cell  
1 # %%  
2 import argon2  
3  
4 time_cost = 16  
5 memory_cost = 2**15 # 32768 KiB (32MB)  
6 parallelism = 2  
7 hash_len = 32  
8 salt = b'some salt'  
9 password = b'password'  
10  
11 argon2_hasher = argon2.PasswordHasher(  
12     time_cost=time_cost,  
13     memory_cost=memory_cost,  
14     parallelism=parallelism,  
15     hash_len=hash_len,  
16     salt_len=16 # defaults to 16  
17 )  
18  
19 hashed_password = argon2_hasher.hash(password.decode('utf-8'))  
20 print("Argon2 hash", hashed_password)  
21  
22 # Verify the correct password  
23 password_is_right = argon2_hasher.verify(hashed_password, "password")  
24 print("Password is right ", password_is_right)  
25  
26 # incorrect password  
27 try:  
28     argon2_hasher.verify(hashed_password, "wrong password")  
29 except argon2.exceptions.VerifyMismatchError:  
30     print("Password is wrong")  
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)

```
password_hashing_algorithms > argon2_py > ...  
Run Cell | Run Below | Debug Cell  
1 # %%  
2 import argon2  
3  
4 time_cost = 16  
5 memory_cost = 2**15 # 32768 KiB (32MB)  
6 parallelism = 2  
7 hash_len = 32  
8 salt = b'some salt'  
9 password = b'password'  
10  
11 argon2_hasher = argon2.PasswordHasher(  
12     time_cost=time_cost,  
13     memory_cost=memory_cost,  
14     parallelism=parallelism,  
15     hash_len=hash_len,  
16     salt_len=16 # defaults to 16  
17 )  
18  
19 hashed_password = argon2_hasher.hash(password.decode('utf-8'))  
20 print("Argon2 hash", hashed_password)  
21  
22 # Verify the correct password  
23 password_is_right = argon2_hasher.verify(hashed_password, "password")  
24 print("Password is right ", password_is_right)  
25  
26 # incorrect password  
27 try:  
28     argon2_hasher.verify(hashed_password, "wrong password")  
29 except argon2.exceptions.VerifyMismatchError:  
30     print("Password is wrong")  
Run Cell | Run Above | Debug Cell
```

Checksum hashing algorithms

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

checksums >  adler-32.py > ...

Run Cell | Run Below | Debug Cell

```
1 # %%  
2 import zlib  
3  
4 string_to_hash = "Anderson"  
5 hash = zlib.adler32(string_to_hash.encode('utf-8'))  
6 print(f"hash: {hash}")  
7  
8
```

Run Cell | Run Above | Debug Cell

```
9 # %%  
10 import zlib  
11  
12 file_checksum = 0  
13 with open("../files/test_file.txt", 'rb') as f:  
14     while True:  
15         chunk = f.read(4096) # Read 4KB at a time  
16         if not chunk:  
17             break  
18         file_checksum = zlib.adler32(chunk)  
19  
20 print(f"hash of 'test_file.txt': {file_checksum}")  
Run Cell | Run Above | Debug Cell  
21 # %%
```

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