**Hashing implementations in python**

Python offers several types of hashing implementations, mainly divided into two categories:

**1. Built-in hash() Function**

* Python's built-in hash() function returns an integer hash value for any hashable object.
* It is mainly used internally for data structures like dictionaries and sets.
* The hash values are consistent within a Python session but can vary between sessions due to randomization for security.
* Usage example:

hash\_value = hash("hello world")

* This function is fast but not designed for cryptographic or secure hashing needs.

**2. hashlib Module for Secure Hashing**

* Python’s hashlib module provides a common interface to many secure hash algorithms, suitable for cryptographic and data integrity purposes.
* Supported algorithms include:
  + MD5 (128-bit)
  + SHA-1 (160-bit)
  + SHA-2 family: SHA-224, SHA-256, SHA-384, SHA-512
  + SHA-3 family: SHA3-224, SHA3-256, SHA3-384, SHA3-512
  + BLAKE2 (BLAKE2b and BLAKE2s)
* Example to generate a SHA-256 hash:

import hashlib  
text = "Hello World"  
hash\_object = hashlib.sha256(text.encode())  
print(hash\_object.hexdigest())

**Summary Table of Python Hashing Types**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Usage | Output Type | Security Level | Notes |
| hash() built-in | Hashing objects for dict/set keys | Integer | Non-cryptographic | Fast but varies between sessions |
| hashlib.md5 | Checksum, legacy cryptography | Hexadecimal string | Low (cryptographically broken) | Useful for non-secure checks |
| hashlib.sha1 | Older secure hashing | Hexadecimal string | Moderate | Being phased out, less preferred for security |
| hashlib.sha256 | Modern secure hashing | Hexadecimal string | High | Widely used for cryptography and integrity checks |
| hashlib.sha3\_xxx | Newer SHA-3 family algorithms | Hexadecimal string | High | Alternative SHA family |
| hashlib.blake2 | Fast and secure hashing | Hexadecimal string | High | Supports keyed and salted hashing |

Thus, Python offers a simple built-in hash for general object hashing and a powerful module with multiple algorithms for cryptographic and secure hashing needs.

If required, I can provide usage examples or deeper explanations for any specific hashing type.

**Why is hash() randomized between Python runs**

The reason Python's built-in hash() function returns different values for the same input across different runs is due to a security feature called **hash randomization**, introduced in Python 3.3.

**Why is hash() Randomized?**

* The primary goal is to **prevent denial-of-service (DoS) attacks** that exploit hash collision vulnerabilities.
* Attackers can craft many inputs that produce the same hash value (collisions), causing the underlying hash table data structures (like dictionaries) to degrade in performance to O(n) and potentially crash or slow down applications.
* To mitigate this, Python seeds the hash function with a **random value at interpreter startup** (called a "salt"), making hash results unpredictable and different on each run.
* As a result, the same string or object will have different hash values across Python sessions, improving security by preventing attackers from predicting hash collisions.
* The randomized seed is consistent within a single run, so dictionary and set operations remain stable during execution.

**Control Over Hash Randomization**

* Hash randomization can be disabled or controlled by setting the PYTHONHASHSEED environment variable:
  + PYTHONHASHSEED=0 disables hash randomization (not recommended).
  + Setting it to a fixed integer produces deterministic hashes for debugging or testing but reduces security.

**Summary**

Python randomizes hash() outputs across runs as a deliberate security measure to make the hash function less predictable and protect applications from certain types of attacks exploiting hash collisions.

If consistent hashing is required across runs, it is recommended to use cryptographic hash functions from the hashlib module instead.

This behavior improves security while keeping hash usage effective and safe for built-in collections like dictionaries and sets.[[1]](file:///C:\\Users\\latha\\Downloads\\Why%20is%20hash()%20randomized%20between%20Python%20runs.docx" \l "fn1)[[2]](file:///C:\\Users\\latha\\Downloads\\Why%20is%20hash()%20randomized%20between%20Python%20runs.docx" \l "fn2)[[3]](file:///C:\\Users\\latha\\Downloads\\Why%20is%20hash()%20randomized%20between%20Python%20runs.docx" \l "fn3)

**closed or open type during collision, in python?**

In Python, dictionary collisions are handled using an **open addressing** method with a form of **probing**, specifically a variant called **quadratic probing** or a related probing sequence.

**Open Addressing in Python Dictionaries**

* Python dictionaries use a **hash table** where each slot in the table can hold only one entry (key-value pair).
* When a collision occurs (two different keys hash to the same slot), Python uses open addressing to find the next available empty slot for the new entry.
* The probing sequence is not a simple linear probing but a slightly more complex function to reduce clustering and minimize consecutive collisions.
* The sequence tries slots based on a formula involving quadratic increments and some perturbation to scatter probes more randomly but deterministically.

**Why Open Addressing?**

* It keeps all entries within the main hash table array rather than using secondary structures like linked lists.
* This leads to better cache locality and memory efficiency.
* When a key’s initial hash slot is occupied, Python keeps probing new positions until an empty slot is found.

**Summary**

* Python dictionaries use **open addressing** with a specialized probing sequence to resolve collisions.
* This is an **open type** collision handling scheme, not closed/addressing with chaining (which uses linked lists or sublists).
* Each dictionary slot holds either a single key-value entry or is empty.
* On collision, the algorithm probes the hash table in a predictable but dispersed sequence until it finds a free slot for insertion or the matching key for lookup.

This open addressing strategy enhances performance and memory usage in Python’s implementation of dictionaries.[[1]](#fn1)[[2]](#fn2)[[3]](#fn3)[[4]](#fn4)

⁂

1. <https://www.designgurus.io/answers/detail/how-are-pythons-built-in-dictionaries-implemented>

1. <https://stackoverflow.com/questions/21595048/how-python-dict-stores-key-value-when-collision-occurs>

1. <https://www.freecodecamp.org/news/exploring-python-internals-the-dictionary-a32c14e73efa/>

1. <https://www.playfulpython.com/deep-dive-python-dictionaries/>

1. <https://achieversnoida.hashnode.dev/how-does-python-handle-hash-collisions-in-dictionaries>

1. <https://sqlpad.io/tutorial/python-hash-table/>

1. <https://www.geeksforgeeks.org/dsa/separate-chaining-collision-handling-technique-in-hashing/>