Photo by Vincentiu Solomon on Unsplash

Hashing is a key part of most programming languages. Large amounts of data can be represented in a fixed buffer. Key-value structures use hashes to store references.

Hashes are used to secure. Hashes can be deterministic or non-deterministic. Hashes can be significantly different with small changes to data or very similar.

This article will review the most common ways to hash data in Python.

Python provides the built-in `.hash()` function as shown below.

```
>>> hash("test")
2314058222102390712
```

The above was run in Python 2.7, let's try Python 3.7.

```
>>> hash("test")
5946494221830395164
```

The result is different and will be different for each new Python invocation. Python has never guaranteed that `.hash()` is deterministic.

In Python 2.x, it would be deterministic most of the time but not always. Python 3.x added randomness to `.hash()` to improve security. The default sort order of dictionaries, sets, and lists is backed by built-in hashing.

I have a whole project covering Python 2.x hashing in Python 3.x. Generally, `.hash()` shouldn't be relied on for anything across Python invocations.

Checksums are used to validate data in a file. ZIP files use checksums to ensure a file is not corrupt when decompressing. Unlike Python's built-in hashing, it's deterministic. The same data will return the same result each time.

Below is an example using zlib's alder32 and crc32. Alder32 is usually a better choice as it's much faster and almost as reliable as crc32.

```
>>> import zlib
>>> zlib.adler32(b"test")
73204161
>>> zlib.crc32(b"test")
3632233996
```

For a small database, adler32 could be used as a simple ID hash. But collisions will quickly become a concern as data grows.

Secure hashes and message digests have evolved over the years. From MD5 to SHA1 to SHA256 to SHA512.

Each method grows in size, improving security and reducing the risk of hash collisions. A collision is when two different arrays of data resolve to the same hash.

Hashing can take a large amount of arbitrary data and build a digest of the content. Open-source software builds digests of their packages to help users know that they can trust that files haven't been tampered with. Small changes to the file will result in a much different hash.

Look at how different two MD5 hashes are after changing one character.

```
>>> import hashlib
>>> hashlib.md5(b"test1").hexdigest()
'5a105e8b9d40e1329780d62ea2265d8a'
>>> hashlib.md5(b"test2").hexdigest()
'ad0234829205b9033196ba818f7a872b'
```

Let's look at some common secure hash algorithms.

MD5– 16 bytes/128 bit

MD5 hashes are 16 bytes or 128 bits long. See the example below, note that a hex digest is representing each byte as a hex string (i.e. the leading 09 is one byte). MD5 hashes are no longer commonly used.

```
>>> import hashlib
>>> hashlib.md5(b"test").hexdigest()
'098f6bcd4621d373cade4e832627b4f6'
>>> len(hashlib.md5(b"test").digest())
16
```

SHA1–20 bytes/160 bit

SHA1 hashes are 20 bytes or 160 bits long. SHA1 hashes are also no longer commonly used.

```
>>> import hashlib
>>> hashlib.sha1(b"test").hexdigest()
'a94a8fe5ccb19ba61c4c0873d391e987982fbbd3'
>>> len(hashlib.sha1(b"test").digest())
20
```