`hashlib` --- Algorithmes de hachage sécurisés et synthèse de messages

Code source : Lib/hashlib.py

Ce module implémente une interface commune à différents algorithmes de hachage sécurisés et de synthèse de messages. Sont inclus les algorithmes standards FIPS de hachage SHA1, SHA224, SHA256, SHA384, et SHA512 (définis dans FIPS 180-2) ainsi que l'algorithme MD5 de RSA (défini par la **RFC 1321**). Les termes "algorithmes de hachage sécurisé" et "algorithme de synthèse de message" sont interchangeables. Les anciens algorithmes étaient appelés" algorithmes de synthèse de messages". Le terme moderne est "algorithme de hachage sécurisé".

Note: Si vous préférez utiliser les fonctions de hachage *adler32* ou *crc32*, elles sont disponibles dans le module `zlib`.

Avertissement: Certains algorithmes ont des faiblesses connues relatives à la collision, se référer à la section "Voir aussi" à la fin.

Algorithmes de hachage

Il y a un constructeur nommé selon chaque type de *hash*. Tous retournent un objet haché avec la même interface. Par exemple : utilisez `sha256()` pour créer un objet haché de type SHA-256. Vous pouvez maintenant utiliser cet objet bytes-like objects (normalement des `bytes`) en utilisant la méthode `update()`. À tout moment vous pouvez demander le *digest* de la concaténation des données fournies en utilisant les méthodes `digest()` ou `hexdigest()`.

Note: Pour de meilleures performances avec de multiples fils d'exécution, le GIL Python est relâché pour des données dont la taille est supérieure à 2047 octets lors de leur création ou leur mise à jour.

Note: Fournir des objets chaînes de caractères à la méthode `update()` n'est pas implémenté, comme les fonctions de hachages travaillent sur des *bytes* et pas sur des caractères.

Constructors for hash algorithms that are always present in this module are `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `blake2b()`, and `blake2s()`. `md5()` is normally available as well, though it may be missing or blocked if you are using a rare "FIPS compliant" build of Python. Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform. On most platforms the `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`

are also available.

Nouveau dans la version 3.6: Les constructeurs SHA3 (Keccak) et SHAKE `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`.

Nouveau dans la version 3.6: Les fonctions `blake2b()` et `blake2s()` ont été ajoutées.

Modifié dans la version 3.9: All hashlib constructors take a keyword-only argument *usedforsecurity* with default value `True`. A false value allows the use of insecure and blocked hashing algorithms in restricted environments. `False` indicates that the hashing algorithm is not used in a security context, e.g. as a non-cryptographic one-way compression function.

Hashlib now uses SHA3 and SHAKE from OpenSSL 1.1.1 and newer.

Par exemple, pour obtenir l'empreinte de la chaîne `b'Nobody inspects the spammish repetition'` :

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:
>>> m.digest_size
32
>>> m.block_size
64
```

En plus condensé :

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdig
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

`hashlib.`**new**(*name*, [*data*,]*, *usedforsecurity=True*)

Est un constructeur générique qui prend comme premier paramètre le nom de l'algorithme désiré (*name*) . Il existe pour permettre l'accès aux algorithmes listés ci-dessus ainsi qu'aux autres algorithmes que votre librairie OpenSSL peut offrir. Les constructeurs nommés sont beaucoup plus rapides que `new()` et doivent être privilégiés.

En utilisant `new()` avec un algorithme fourni par OpenSSL :

```
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Hashlib fournit les constantes suivantes :

`hashlib.`**algorithms_guaranteed**

Un ensemble contenant les noms des algorithmes de hachage garantis d'être implémentés par ce module sur toutes les plate-formes. Notez que *md5* est dans cette liste malgré certains éditeurs qui offrent une implémentation Python de la librairie compatible FIPS l'excluant.

Nouveau dans la version 3.2.

`hashlib.`**`algorithms_available`**

Un ensemble contenant les noms des algorithmes de hachage disponibles dans l'interpréteur Python. Ces noms sont reconnus lorsqu'ils sont passés à la fonction `new()`. `algorithms_guaranteed` est toujours un sous-ensemble. Le même algorithme peut apparaître plusieurs fois dans cet ensemble sous un nom différent (grâce à OpenSSL).

Nouveau dans la version 3.2.

Les valeurs suivantes sont fournis en tant qu'attributs constants des objets hachés retournés par les constructeurs :

`hash.`**`digest_size`**

La taille du *hash* résultant en octets.

`hash.`**`block_size`**

La taille interne d'un bloc de l'algorithme de hachage en octets.

L'objet haché possède les attributs suivants :

`hash.`**`name`**

Le nom canonique de cet objet haché, toujours en minuscule et toujours transmissible à la fonction `new()` pour créer un autre objet haché de ce type.

Modifié dans la version 3.4: L'attribut *name* est présent dans CPython depuis sa création, mais n'était pas spécifié formellement jusqu'à Python 3.4, il peut ne pas exister sur certaines plate-formes.

L'objet haché possède les méthodes suivantes :

`hash.`**`update`**(*`data`*)

Met à jour l'objet haché avec bytes-like object. Les appels répétés sont équivalent à la concaténation de tous les arguments : `m.update(a); m.update(b)` est équivalent à `m.update(a+b)`.

Modifié dans la version 3.1: Le GIL Python est relâché pour permettre aux autres fils d'exécution de tourner pendant que la fonction de hachage met à jour des données plus larges que 2047 octets, lorsque les algorithmes fournis par OpenSSL sont utilisés.

`hash.`**`digest`**()

Renvoie le *digest* des données passées à la méthode `update()`. C'est un objet de type *bytes* de taille `digest_size` qui contient des octets dans l'intervalle 0 à 255.

`hash.`**`hexdigest`**()

Comme la méthode `digest()` sauf que le *digest* renvoyé est une chaîne de caractères de longueur double, contenant seulement des chiffres hexadécimaux. Cela peut être utilisé pour échanger sans risque des valeurs dans les *e-mails* ou dans les environnements non binaires.

`hash.`**`copy`**()

Renvoie une copie ("clone") de l'objet haché. Cela peut être utilisé pour calculer efficacement les *digests* de données partageant des sous-chaînes communes.

Synthèse de messages de taille variable SHAKE

Les algorithmes `shake_128()` et `shake_256()` fournissent des messages de longueur variable avec des `longueurs_en_bits // 2` jusqu'à 128 ou 256 bits de sécurité. Leurs méthodes *digests* requièrent une longueur. Les longueurs maximales ne sont pas limitées par l'algorithme SHAKE.

`shake.`**`digest`**(*`length`*)

Renvoie le *digest* des données passées à la méthode `update()`. C'est un objet de type *bytes* de taille *length* qui contient des octets dans l'intervalle 0 à 255.

`shake.`**`hexdigest`**(*`length`*)

Comme la méthode `digest()` sauf que le *digest* renvoyé est une chaîne de caractères de longueur double, contenant seulement des chiffres hexadécimaux. Cela peut être utilisé pour échanger sans risque des valeurs dans les *e-mails* ou dans les environnements non binaires.

Dérivation de clé

Les algorithmes de dérivation de clés et d'étirement de clés sont conçus pour le hachage sécurisé de mots de passe. Des algorithmes naïfs comme `sha1(password)` ne sont pas résistants aux attaques par force brute. Une bonne fonction de hachage doit être paramétrable, lente, et inclure un sel.

`hashlib.`**`pbkdf2_hmac`**(*`hash_name`*, *`password`*, *`salt`*, *`iterations`*, *`dklen=None`*)

La fonction fournit une fonction de dérivation PKCS#5 (*Public Key Cryptographic Standards #5 v2.0*). Elle utilise HMAC comme fonction de pseudo-aléatoire.

La chaîne de caractères *hash_name* est le nom de l'algorithme de hachage désiré pour le HMAC, par exemple `"sha1"` ou `"sha256"`. *password* et *salt* sont interprétés comme des tampons d'octets. Les applications et bibliothèques doivent limiter *password* à une longueur raisonnable (comme `1024`). *salt* doit être de 16 octets ou plus provenant d'une source correcte, e.g. `os.urandom()`.

Le nombre d'*iterations* doit être choisi sur la base de l'algorithme de hachage et de la puissance de calcul. En 2013, au moins 100000 itérations de SHA-256 sont recommandées.

dklen est la longueur de la clé dérivée. Si *dklen* vaut `None` alors la taille du message de l'algorithme de hachage *hash_name* est utilisé, e.g. 64 pour SHA-512.

```
>>> import hashlib
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 10000
>>> dk.hex()
'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

Nouveau dans la version 3.4.

Note: Une implémentation rapide de *pbkdf2_hmac* est disponible avec OpenSSL. L'implémentation Python utilise une version anonyme de `hmac`. Elle est trois fois plus lente et ne libère pas le GIL.

`hashlib.`**`scrypt`**(*`password`*, *, *`salt`*, *`n`*, *`r`*, *`p`*, *`maxmem=0`*, *`dklen=64`*)

La fonction fournit la fonction de dérivation de clé *scrypt* comme définie dans **RFC 7914**.

password et *salt* doivent être des bytes-like objects. Les applications et bibliothèques doivent limiter *password* à une longueur raisonnable (e.g. 1024). *salt* doit être de 16 octets ou plus provenant d'une source correcte, e.g. `os.urandom()`.

n est le facteur de coût CPU/Mémoire, *r* la taille de bloc, *p* le facteur de parallélisation et *maxmem* limite la mémoire (OpenSSL 1.1.0 limite à 32 MB par défaut). *dklen* est la longueur de la clé dérivée.

Disponibilité : OpenSSL 1.1+.

Nouveau dans la version 3.6.

BLAKE2

BLAKE2 est une fonction de hachage cryptographique définie dans la **RFC 7693** et disponible en deux versions :

- **BLAKE2b**, optimisée pour les plates-formes 64-bit et produisant des messages de toutes tailles entre 1 et 64 octets,
- **BLAKE2s**, optimisée pour les plates-formes de 8 à 32-bit et produisant des messages de toutes tailles entre 1 et 32 octets.

BLAKE2 gère diverses fonctionnalités **keyed mode** (un remplacement plus rapide et plus simple pour HMAC), **salted hashing**, **personalization**, et **tree hashing**.

Les objets hachés de ce module suivent l'API des objets du module `hashlib` de la librairie

standard.

Création d'objets hachés

Les nouveaux objets hachés sont créés en appelant les constructeurs :

hashlib.**blake2b**(*data=b''*, *, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

hashlib.**blake2s**(*data=b''*, *, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

Ces fonctions produisent l'objet haché correspondant aux calculs de BLAKE2b ou BLAKE2s. Elles prennent ces paramètres optionnels :

- *data*: morceau initial de données à hacher, qui doit être un objet de type bytes-like object. Il peut être passé comme argument positionnel.
- *digest_size*: taille en sortie du message en octets.
- *key*: clé pour les code d'authentification de message *keyed hashing* (jusqu'à 64 octets pour BLAKE2b, jusqu'à 32 octets pour BLAKE2s).
- *salt*: sel pour le hachage randomisé *randomized hashing* (jusqu'à 16 octets pour BLAKE2b, jusqu'à 8 octets pour BLAKE2s).
- *person*: chaîne de personnalisation (jusqu'à 16 octets pour BLAKE2b, jusqu'à 8 octets pour BLAKE2s).

Le tableau suivant présente les limites des paramètres généraux (en octets) :

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

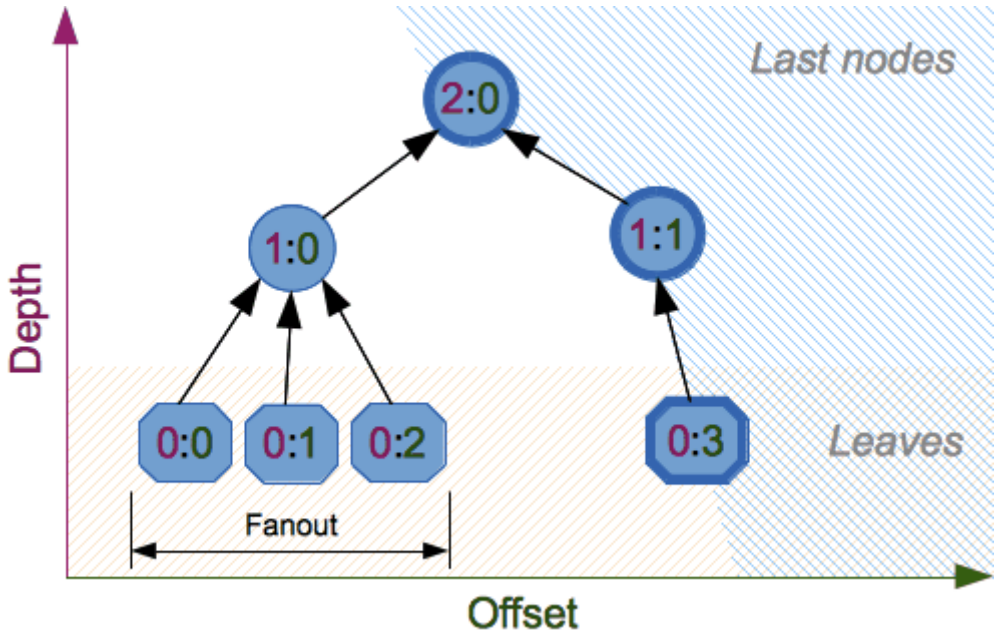
Note: Les spécifications de BLAKE2 définissent des longueurs constantes pour les sel et chaînes de personnalisation, toutefois, par commodité, cette implémentation accepte des chaînes *byte* de n'importe quelle taille jusqu'à la longueur spécifiée. Si la longueur du paramètre est moindre par rapport à celle spécifiée, il est complété par des zéros, ainsi, par exemple, `b'salt'` et `b'salt\x00'` sont la même valeur (Ce n'est pas le cas pour *key*.)

Ces tailles sont disponibles comme constants du module et décrites ci-dessous.

Les fonctions constructeur acceptent aussi les paramètres suivants pour le *tree hashing* :

- *fanout*: *fanout* (0 à 255, 0 si illimité, 1 en mode séquentiel).
- *depth*: profondeur maximale de l'arbre (1 à 255, 255 si illimité, 1 en mode séquentiel).

- *leaf_size*: taille maximale en octets d'une feuille (0 à 2**32-1, 0 si illimité ou en mode séquentiel).
- *node_offset*: décalage de nœud (0 à 2**64-1 pour BLAKE2b, 0 à 2**48-1 pour BLAKE2s, 0 pour la première feuille la plus à gauche, ou en mode séquentiel).
- *node_depth*: profondeur de nœuds (0 à 255, 0 pour les feuilles, ou en mode séquentiel).
- *inner_size*: taille interne du message (0 à 64 pour BLAKE2b, 0 à 32 pour BLAKE2s, 0 en mode séquentiel).
- *last_node*: booléen indiquant si le nœud traité est le dernier (*False* pour le mode séquentiel).



Voir section 2.10 dans BLAKE2 specification pour une approche compréhensive du *tree hashing*.

Constantes

`blake2b.`**`SALT_SIZE`**

`blake2s.`**`SALT_SIZE`**

Longueur du sel (longueur maximale acceptée par les constructeurs).

`blake2b.`**`PERSON_SIZE`**

`blake2s.`**`PERSON_SIZE`**

Longueur de la chaîne de personnalisation (longueur maximale acceptée par les constructeurs).

`blake2b.`**`MAX_KEY_SIZE`**

`blake2s.`**`MAX_KEY_SIZE`**

Taille maximale de clé.

`blake2b.`**`MAX_DIGEST_SIZE`**

`blake2s.`**`MAX_DIGEST_SIZE`**

Taille maximale du message que peut fournir la fonction de hachage.

Exemples

Hachage simple

Pour calculer les *hash* de certaines données, vous devez d'abord construire un objet haché en appelant la fonction constructeur appropriée (`blake2b()` or `blake2s()`), ensuite le mettre à jour avec les données en appelant la méthode `update()` sur l'objet, et, pour finir, récupérer l'empreinte du message en appelant la méthode `digest()` (ou `hexdigest()` pour les chaînes hexadécimales).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a
```

Pour raccourcir, vous pouvez passer directement au constructeur, comme argument positionnel, le premier morceau du message à mettre à jour :

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a
```

Vous pouvez appeler la méthode `hash.update()` autant de fois que nécessaire pour mettre à jour le *hash* de manière itérative :

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a
```

Usage de tailles d'empreintes différentes

BLAKE2 permet de configurer la taille des empreintes jusqu'à 64 octets pour BLAKE2b et jusqu'à 32 octets pour BLAKE2s. Par exemple, pour remplacer SHA-1 par BLAKE2b sans changer la taille de la sortie, nous pouvons dire à BLAKE2b de produire une empreinte de

20 octets :

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Les objets hachés avec différentes tailles d'empreintes ont des sorties complètement différentes (les *hash* plus courts *ne sont pas* des préfixes de *hash* plus longs); BLAKE2b et BLAKE2s produisent des sorties différentes même si les longueurs des sorties sont les mêmes :

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Code d'authentification de message

Le hachage avec clé (*keyed hashing* en anglais) est une alternative plus simple et plus rapide à un code d'authentification d'une empreinte cryptographique de message avec clé (HMAC). BLAKE2 peut être utilisé de manière sécurisée dans le mode préfixe MAC grâce à la propriété d'indifférentiabilité héritée de BLAKE.

Cet exemple montre comment obtenir un code d'authentification de message de 128-bit (en hexadécimal) pour un message `b'message data'` avec la clé `b'pseudorandom key'` :

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

Comme exemple pratique, une application web peut chiffrer symétriquement les *cookies* envoyés aux utilisateurs et les vérifier plus tard pour être certaine qu'ils n'aient pas été altérés :

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
```

```
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Même s'il possède en natif la création de code d'authentification de message (MAC), BLAKE2 peut, bien sûr, être utilisé pour construire un HMAC en combinaison du module `hmac` :

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Hachage randomisé

En définissant le paramètre *salt* les utilisateurs peuvent introduire de l'aléatoire dans la fonction de hachage. Le hachage randomisé est utile pour se protéger des attaques par collisions sur les fonctions de hachage utilisées dans les signatures numériques.

> Le hachage aléatoire est conçu pour les situations où une partie, le préparateur du message, génère tout ou partie d'un message à signer par une seconde partie, le signataire du message. Si le préparateur du message est capable de trouver des collisions sur la fonction cryptographique de hachage (c.-à-d. deux messages produisant la même valeur une fois hachés), alors ils peuvent préparer des versions significatives du message qui produiront les mêmes *hachs* et même signature mais avec des résultats différents (e.g. transférer 1000000$ sur un compte plutôt que 10$). Les fonctions cryptographiques de hachage ont été conçues dans le but de résister aux collisions, mais la concentration actuelle d'attaques sur les fonctions de hachage peut avoir pour conséquence

> qu'une fonction de hachage donnée soit moins résistante qu'attendu. Le hachage aléatoire offre au signataire une protection supplémentaire en réduisant la probabilité que le préparateur puisse générer deux messages ou plus qui renverront la même valeur haché lors du processus de génération de la signature --- même s'il est pratique de trouver des collisions sur la fonction de hachage. Toutefois, l'utilisation du hachage aléatoire peut réduire le niveau de sécurité fourni par une signature numérique lorsque tous les morceaux du message sont préparés par le signataire.
>
> (NIST SP-800-106 "Randomized Hashing for Digital Signatures", article en anglais)

Dans BLAKE2, le sel est passé une seule fois lors de l'initialisation de la fonction de hachage, plutôt qu'à chaque appel d'une fonction de compression.

Avertissement: *Salted hashing* (ou juste hachage) avec BLAKE2 ou toute autre fonction de hachage générique, comme SHA-256, ne convient pas pour le chiffrement des mots de passe. Voir BLAKE2 FAQ pour plus d'informations.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personnalisation

Parfois il est utile de forcer une fonction de hachage à produire différentes empreintes de message d'une même entrée pour différentes utilisations. Pour citer les auteurs de la fonction de hachage Skein :

> Nous recommandons que tous les développeurs d'application considèrent sérieusement de faire cela ; nous avons vu de nombreux protocoles où un *hash* était calculé à un endroit du protocole pour être utilisé à un autre endroit car deux calculs de *hash* étaient réalisés sur des données similaires ou liées, et qu'un attaquant peut forcer une application à prendre en entrée le même *hash*. Personnaliser chaque fonction de hachage utilisée dans le protocole stoppe immédiatement ce genre d'attaque.
>
> (The Skein Hash Function Family, p. 21, article en anglais)

BLAKE2 peut être personnalisé en passant des *bytes* à l'argument *person* :

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

La personnalisation et le *keyed mode* peuvent être utilisés ensemble pour dériver différentes clés à partir d'une seule.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21K
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

Mode Arbre

L'exemple ci-dessous présente comment hacher un arbre minimal avec deux nœuds terminaux :

```
  10
 /  \
00  01
```

Cet exemple utilise en interne des empreintes de 64 octets, et produit finalement des empreintes 32 octets :

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
```

```
...                 node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...                 leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                 node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...                 leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                 node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

Crédits

BLAKE2 a été conçu par *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn*, et *Christian Winnerlein* basé sur SHA-3 finaliste BLAKE créé par *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier*, et *Raphael C.-W. Phan*.

Il utilise le cœur de l'algorithme de chiffrement de ChaCha conçu par *Daniel J. Bernstein*.

L'implémentation dans la librairie standard est basée sur le module pyblake2. Il a été écrit par *Dmitry Chestnykh* et basé sur l'implémentation C écrite par *Samuel Neves*. La documentation a été copiée depuis pyblake2 et écrite par *Dmitry Chestnykh*.

Le code C a été partiellement réécrit pour Python par *Christian Heimes*.

Le transfert dans le domaine publique s'applique pour l'implémentation C de la fonction de hachage, ses extensions et cette documentation :

> Tout en restant dans les limites de la loi, le(s) auteur(s) a (ont) consacré tous les droits d'auteur et droits connexes et voisins de ce logiciel au domaine public dans le monde entier. Ce logiciel est distribué sans aucune garantie.
>
> Vous devriez recevoir avec ce logiciel une copie de la licence *CC0 Public Domain Dedication*. Sinon, voir https://creativecommons.org/publicdomain/zero/1.0/.

Les personnes suivantes ont aidé au développement ou contribué aux modification du projet et au domaine public selon la licence Creative Commons Public Domain Dedication 1.0 Universal :

- *Alexandr Sokolovskiy*

Voir aussi:

Module `hmac`
: Un module pour générer des codes d'authentification utilisant des *hash*.

Module `base64`

Un autre moyen d'encoder des *hash* binaires dans des environnements non binaires.

https://blake2.net

Site officiel de BLAKE2.

https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf

La publication FIPS 180-2 sur les algorithmes de hachage sécurisés.

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms

Article Wikipedia contenant les informations relatives aux algorithmes ayant des problèmes et leur interprétation au regard de leur utilisation.

https://www.ietf.org/rfc/rfc2898.txt

PKCS #5: Password-Based Cryptography Specification Version 2.0