**Хеширование**

**Что такое хеширование**

Процесс трансформации данных любого типа и длины в битовую строку. Это необратимый процесс. Например, вы хотите захешировать некоторую сущность, скажем, пароль. Но вы не сможете потом восстановить исходный пароль из его хеша.

Благодаря хешированию исходный объект трансформируется в строку фиксированной длины, которую можно рассматривать как «отпечаток пальца» этого объекта. Он единственный и неповторимый и принадлежит только этому объекту. Идеальная защита для паролей. Если злоумышленник взломает БД с паролями, то сможет получить только хеши, от которых мало толку.

Типичный реальный пример использования хешей:

1. Вы создали аккаунт в приложении.
2. Ваш пароль хешируется за счет хеш-функции и сохраняется в БД.
3. Если вы потом пытаетесь залогиниться, то введенный вами пароль пропускается через хеш-функцию и сравнивается с хешем правильного пароля, сохраненным в БД.
4. При совпадении хешей пользователь получает доступ к ресурсу, иначе пароль будет запрошен повторно.
5. Шаги 3 и 4 проходят каждый раз, когда вы будете проходить авторизацию.

Еще один яркий пример вы встретите, когда при регистрации в приложении вам высылают код активации. Для него тоже создается хеш и записывается в БД. Потом вы вводите полученный код и сравнивается хеш для него и записанный в БД хеш. С этим примером вы встретитесь (наверно) на курсе «Django Framework».

**Подробнее про хеш-функции**

Под хеш-функцией в Python понимается функция, принимающая объект-данные (байтовый тип) и возвращающая последовательность фиксированной длины. То, что возвращает функция, называют хешем (а также значением хеша, контрольной суммой). Хеш-функция устроена таким образом, что, если мы в исходном объекте изменим его хотя бы на один символ, это приведет и к полному изменению хеша. Отметим, что эта функция односторонняя, т.е. ее легко рассчитать, но обратный процесс требует существенно больших затрат.

Хеш-функции применяются в рамках реализации криптографических алгоритмов, электронных подписях, для контроля целостности данных, для обнаружения несанкционированных манипуляций, при сканировании отпечатков пальцев, в хеш-таблицах, для хранения паролей, безопасного хранения паролей в БД.

Стоит отметить, что сами по себе эти ф-ции ничего не шифруют и не дешифруют, но применяются во многих криптографических протоколах и инструментах.

**Известные хеш-функции в Python**

**md5.** Применяется в рамках проверки целостности данных. В других областях применение достаточно редкое по причине доказанной уязвимости функции. Длина сообщения 128 битов.

**sha.** Применяется более широко, длина сообщения составляет от 160 до 512 битов.

Как же в Python воспользоваться данными функциями?

Модуль hashlib («из коробки») предоставляет необходимые возможности.

**Листинг 1. task\_1.py**

|  |
| --- |
| **import** hashlib  print(hashlib.algorithms\_available) print(hashlib.algorithms\_guaranteed)  **""" Результат: {'sha3\_384', 'shake\_128', 'sha224', 'blake2s', 'sha512', 'shake\_256',  'sha3\_512', 'sha256', 'md5', 'sha3\_256', 'blake2b', 'sha384', 'sha3\_224', 'sha1'}  {'sha3\_384', 'shake\_128', 'sha224', 'blake2s', 'sha512', 'shake\_256', 'sha3\_512', 'sha256', 'md5', 'sha3\_256', 'blake2b', 'sha384', 'sha3\_224', 'sha1'} """** |

Функция algorithms\_available возвращает список все алгоритмов, доступных в системе. Здесь есть и дубликаты названий.

Функция algorithms\_guaranteed возвращает только существующие алгоритмы модуля. В этом списке всегда в наличии md5, sha1, sha224, sha256, sha384, sha512.

**Рассмотрим возможности md5.**

**Листинг 2. task\_2.py**

|  |
| --- |
| *"""Примеры с md5"""* **import** hashlib  hash\_obj = hashlib.md5(**b'Testing md5 func'**) print(hash\_obj) *# -> <md5 HASH object @ 0x0000021C4B589A20>* print(type(hash\_obj)) *# -> <class '\_hashlib.HASH'>* res = hash\_obj.hexdigest() print(type(res)) *# -> <class 'str'>* print(res) *# -> b631e4f1254574b9c386fcbc9145d0c3* print() hash\_obj\_2 = hashlib.md5((**"Тестируем функцию md5"**).encode(**'utf-8'**)) print(hash\_obj\_2) *# -> <md5 HASH object @ 0x0000021C4D53ED50>* print(type(hash\_obj\_2)) *# -> <class '\_hashlib.HASH'>* res\_2 = hash\_obj\_2.hexdigest() print(type(res\_2)) *# -> <class 'str'>* print(res\_2) *# -> cb63de18e7c52d17e3b5e9743210ab74* |

Результат работы скрипта показывает, что мы вначале получаем хеш-объект, а затем его трансформируем в строку с шестнадцатеричным представлением. Функция md5 должна принимать байты в качестве входного параметра.

**Рассмотрим возможности sha.**

**Листинг 3. task\_3.py**

|  |
| --- |
| *"""Примеры с sha"""* **import** hashlib   *#-------------------------------------sha1--------------------------------------#* hash\_obj = hashlib.sha1(**b'Testing sha1 func'**) hex\_dig\_res = hash\_obj.hexdigest()  print(hex\_dig\_res) *# -> d9536c477c646977dce73445a656a9c5e1c19d59* print()  *#-------------------------------------sha224--------------------------------------#* hash\_obj = hashlib.sha224(**b'Testing sha1 func'**) hex\_dig\_res = hash\_obj.hexdigest() *# -> 5a39dff4807dc145be2cc85efa7b4c165bed383e69e0691546b2589f* print(hex\_dig\_res)  print()  *#-------------------------------------sha256--------------------------------------#* hash\_obj = hashlib.sha256(**b'Testing sha1 func'**) hex\_dig\_res = hash\_obj.hexdigest() *# -> c54034d262f7c0b9b82ce4988f115925ee684dd39e399c9ddea0c776d27d7521* print(hex\_dig\_res)  print()  *#-------------------------------------sha384--------------------------------------#* hash\_obj = hashlib.sha384(**b'Testing sha1 func'**) hex\_dig\_res = hash\_obj.hexdigest() *# -> 4080aa6d42e2a67c1f6307771ecbe11a23ec8283fd775d720381844cb3b9e4d038f5446f9db3123bbb4bba588c436f3f* print(hex\_dig\_res)  print()  *#-------------------------------------sha512--------------------------------------#* hash\_obj = hashlib.sha512(**b'Testing sha1 func'**) hex\_dig\_res = hash\_obj.hexdigest() *# -> d4ea479e7b84b3c71416311af2e79f2919233775f86a8273eaf7e14440a306df6ad9587a1d6fe624529118efa2b55740a138276a0630dc0b059066ddaec7a60f* print(hex\_dig\_res) |

**Рассмотрим возможности какого-то из алгоритмов, доступного на вашем ПК.**

**Листинг 4. task\_4.py**

|  |
| --- |
| *"""Пример с алгоритмом, доступным на вашем ПК"""* **import** hashlib  hash\_obj = hashlib.new(**'blake2b'**) hash\_obj.update(**b'blake2b example'**)  print(hash\_obj.hexdigest()) *# -> 6c81637f462af6927bc2eafcbb04eb02a9df72371c8f10813ccd9c782cdb9861f4078e9c # 4441eaf1cc364dddfaad6fe9e63436699d0bc0a653e54e2587e409be* |

**Рассмотрим пример с использованием криптографической «соли»**

Теперь поговорим о «соленых» хешах.

Под солью понимается строка данных, передаваемая хеш-функции для вычисления хеша (для некоторого объекта) в целях его (хеша) усложнения.

Представьте, что два пользователя ввели одинаковый пароль. Как быть? Ведь в этом случае и хеши будут одинаковыми.

Для решения этой проблемы к паролю можно прибавить некоторую уникальную строку – «соль».

Она может храниться рядом с хешем пароля или даже стать частью хеша. У каждого пользователя должна быть уникальная соль и тогда проблема отпадает.

Соль дает возможность защититься от хакерских атак типа «брутфорс атаки», радужные таблицы, таблицы поиска, словарные атаки и др.

**Листинг 5. task\_5.py**

|  |
| --- |
| *"""Хеширование и соль"""  # Модуль uuid применяется для генерации случайного числа* **from** uuid **import** uuid4 **import** hashlib  salt = uuid4().hex *# -> 80740ba2a1584aa7bf96d32bbe774e54* print(type(salt))  passwd = **"programmer"**  *# соль-часть хеша*res = hashlib.sha256(salt.encode() + passwd.encode()).hexdigest() *# -> efbb20c297f52672a5211f1358ad8d # 72907f56e1ff24cd67a6e8b4683a6a18d2* print(res) |

**Листинг 6. task\_6.py**

|  |
| --- |
| *"""Хеширование и соль"""  # Модуль uuid применяется для генерации случайного числа* **from** uuid **import** uuid4 **import** hashlib  salt = uuid4().hex *# -> 6a1e300f48c54b77a7ef4af58376a95c* print(salt)  passwd = **"programmer"** *# соль идет дополнительно к хешу* res = hashlib.sha256(salt.encode() + passwd.encode()).hexdigest() + **':'** + salt *# -> 022cf2d005a201ab60 # 5658a3345dcfaef7a8bfe050efa7c671b3754e91f1e9bb:6a1e300f48c54b77a7ef4af58376a95c* print(res) |

**Листинг 7. task\_7.py**

|  |
| --- |
| *"""Для создания хешей паролей эффективное решение Так называемый парольный хеш """* **from** hashlib **import** pbkdf2\_hmac **from** binascii **import** hexlify  *# Здесь мы создаем хеш sha256 в пароле при помощи соли со 100,000 итераций.* obj = pbkdf2\_hmac(hash\_name=**'sha256'**,  password=**b'any\_password'**,  salt=**b'any\_salt'**,  iterations=100000)  print(obj) *# -> b'n\x97\xba\xd2\x1fb\x00\xf9\x08p6\xa7\x1e| # \xa9\xfa\x01\xa5\x9e\x1di\x7f~\x02\x84\xcd\x7f\x9b\x89}|\x02'  # это значение можно записать в БД* result = hexlify(obj)  print(result) *# -> b'6e97bad21f6200f9087036a71e7ca9fa01a59e1d697f7e0284cd7f9b897d7c02'* |

Чтобы вычислить хэш используется специальная конструкция pbkdf2\_hmac, в которой указаны параметры — имя алгоритма шифрования (sha256), пароль в виде строки байтов (b’pswd’, наличие соли (b’salt’) и длина расширенного ключа (100000). Плюс такого подхода в том, что недопустимо изменение даже одного символа, т.е. принцип целостности информации тут сохраняется.

**Функция hash**

Мы помним, что есть и такая функция. Как быть с ней?

**Листинг 8. task\_8.py**

|  |
| --- |
| *"""Встроенная функция hash()"""* print(hash(**'http://stackoverflow.com'**)) *# -> -8823307079175196, а потом -8860150030631583233 и т.д.  # а такие результаты у других разработчиков # hash('http://stackoverflow.com') Result: 1934711907  # hash('http://stackoverflow.com') Result: -5768830964305142685  # сделаем выводы!* |

Эта функция возвращает нам целые числа, которые вроде подходят…но так ли это? Оказывается нет. Если один и тот же код запустить на разных платформах нет уверенности, что функция вернет тот же результат. Будут различия на различных ОС и на различных версиях Python (32-х и 64-х разрядные). Вывод один – всегда и везде используем hashlib.

**Хеш-таблицы**

Давайте вспомним, как на первом уроке мы анализировали сложности различных алгоритмов. Например, мы говорили о сложности задачи поиска элемента в массиве.

В обычном случае, это будет линейное время, которое определяется сложностью O(n).

Если мы отсортируем массив, то применив бинарный поиск, сможет найти элемент массива за логарифмическое время - O(logn). Но и в том случае нам придется поискать и затратить на это время. Как было бы хорошо, чтобы мы сразу нашли нужный элемент, независимо от размера исходного массива.

Давайте сделаем еще лучше. Реализуем такую структуру данных, в которой поиск элемента мы выполним за постоянное время O(1). Здесь нам помогут хэш-таблицы. Но пока поговорим о хэш-функциях.

Это функция, которая принимает некоторый объект (строку, под которой понимается, как правило, последовательность байтов) и возвращает число.

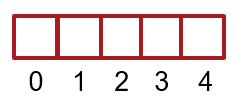
Говорят, что ф-ция переводит строки на числа.

Как же найти закономерность, по которой по которой формируются числа для строк? Возможно ли это? Возможно, но нужно помнить, что такая ф-ция должна соответствовать двум требованиям:

1. Последовательность ф-ции. Вы передаете ей, например, строку «программа» и получаете значение 2. Это значит, что теперь, передавая ей эту строку, вы всегда должны получать это число. Ф-ция должна это обеспечивать. Иначе смысла в итоговой хэш-таблице нет.
2. Разным строкам должны соответствовать разные числа. Если хэш-функция для всех строк (последовательностей байтов) возвращает одно и то же число, то такая ф-ция никуда не годится. В идеале, каждой строке должно соответствовать строго одно число.

Перед тем, как подобраться к хэш-таблицам, начнем издалека – с массивов.

У нас есть пустой массив:

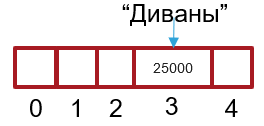


Пусть в нем хранятся цены товаров. Мы будем передавать хэш-функции название очередного товара, а хэш-функция будет возвращать некоторое число.

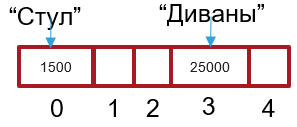


Например, ф-ция вернула число 3. Что нам с ним делать?

Сохранить название товара в массиве в элементе этого массива с индексом 3.



Далее мы передаем ф-ции имена все новых и новых товаров. И записывать их цены в нужные позиции массива.



Постепенно весь массив будет заполнен ценами наших товаров.

Теперь смотрите мы хотим узнать цену некоторого товара! Как это сделать?? Если бы мы работали по старому подходу поиска, то вероятно затратили бы некоторое время на поиск. Но благодаря новому подходу какой-то новый поиск, перебор осуществлять не нужно. Мы просто передаем имя нужного нам товара хэш-функции, а она возвращает нам позицию нужного элемента с ценой товара.



По этой позиции мы без труда находим цену нужного товара.

В итоге хэш-функция сообщает, где находится цена нужного товара и нам вообще не требуется ничего искать.

Почему такой подход работает:

1. Хэш-функция всегда связывает цену товара с одним индексом. Каждый раз, когда эта хэш-функция вызывается для названия товара, мы получаем одно и то же число. На первом вызове это функции мы узнаем, где необходимо сохранить цену товара, а при последующих вызовах эта ф-ция сообщит откуда извлечь нужную цену.
2. Хэш-функция связывает разные объекты (строки) с разными индексами. Например, «Иванов» связывается с индексом 3, а «Петров» - с индексом 5. Таким образом, для каждой строки определяется позиция массива, куда записывается цена товара.
3. Хэш-функция точно знает размер массива и возвращает только существующие индексы. Например, для массива длиной в 10 элементов хэш-функция не вернет значение 50, т.к. это значение не является существующим индексом массива.

Теперь, если мы свяжем хэш-функцию и массив, то получим структуру данных под названием хэш-таблица. Получается хэш-таблицы напоминают нам массивы? Но это не совсем так. Отличия есть. Массивы отображаются на адреса памяти напрямую, а хэш-таблицы устроены более высокоуровнево. ОНИ ОПРЕДЕЛЯЮТ МЕСТО ХРАНЕНИЯ ЭЛЕМЕНТОВ ПРИ ПОМОЩИ ХЭШ-ФУНКЦИЙ.

Хэш-таблицы имеют и другие названия: «словари», «отображения», «ассоциативные массивы». Хэш-таблицы работаю исключительно быстро. Хэш-таблицы для хранения данных используют массивы, поэтому по скорости обращения к элементам они не уступают массивам.

Но стоит заметить, что Питонисту никогда не придется программировать логику реализации хэш-таблиц самостоятельно. В любом полноценном языке программирования существует реализация хэш-таблиц. В Python они тоже есть и называются они словарями.

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Простейшая хеш-таблица в Python"""* goods = dict()  goods[**'Диван'**] = 25000 goods[**'Кровать'**] = 7000 goods[**'Стул'**] = 1500  print(goods) *# -> {'Диван': 25000, 'Кровать': 7000, 'Стул': 1500}* |

Хеш-таблица состоит из ключей и значений. Рассмотрим примеры использования хеш-таблиц: найти номер телефона компании в справочнике, исключить дубли, закэшировать данные.

Почему эти задачи идеально решаются благодаря использованию хеш-таблиц?

Потому что, если бы для хранения элементов применялись списки, то перемещение по ним потребовало бы гораздо больше времени, чем при извлечении значения из хеш-таблицы.

Хеш-таблица дает ответ мгновенно, благодаря константной сложности.

Например, кеширование лежит в основе мемоизации при решении задачи на определение чисел Фибоначчи.

**Листинг 10. task\_10.py**

|  |
| --- |
| *"""Мемоизация, как инструмент борьбы с проблемами рекурсии"""* **def** memorize(func):  **def** g(n, memory={}):  r = memory.get(n)  **if** r **is None**:  r = func(n)  memory[n] = r  **return** r  **return** g   @memorize **def** f(n):  **if** n < 2:  **return** n  **return** f(n - 1) + f(n - 2)   n = 8  print(f(n)) |

**Коллизии**

Подведем промежуточные итоги. Теперь мы знаем, что словари в Python опираются на структуру данных «хеш-таблица».

Хеш-таблицы хранение ключей в соответствии с хеш-значением каждого ключа. Хеш-значение определяется из ключа в виде числового значения фиксированной длины, однозначно идентифицирующего ключ.

Благодаря этому операции поиска выполняются очень быстро. Ведь намного быстрее найти числовое хеш-значение ключа в таблице, чем сравнивать полный объект-ключ с остальными ключами и выполнять проверку на эквивалентность.

Но есть одно НО! Алгоритмы вычисления хешей не идеальны, поэтому существует вероятность того, что два или более ключа будут иметь одинаковый производный хеш. Если два ключа имеют одинаковое хеш-значение, то мы имеем дело с хеш-конфликтом. И как с ним должны разбираться алгоритмы вставки и поиска элементов в хеш-таблице?

Конечно нам, Python-разработчикам, эти проблемы решать не требуется. Ведь функционал хеш-таблиц полностью реализован в словарях. Но что же там, «под капотом»? В Python для разрешения коллизий применяется метод открытой адресации. Поэтому мы этих проблем не замечаем.

**Листинг 11. task\_11.py**

|  |
| --- |
| *"""Пример столкновения хешей"""* print(hash(1.1)) *# -> 2040142438* print(hash(214748749.8)) *# -> 2040142438  # но проблем с использованием слвоаря не возникает* d = {1.1: **'a'**, 4504.1: **'b'**} print(d[1.1]) print(d[4504.1]) |

В Python хеш-столкновения решаются путем использования \_\_eq\_\_ методов проверки на равенство по значению.

Некоторые, хеш-функции md5 и sha1 уже доказано, что имеют уязвимости. Поэтому используйте их совместно с солью и даст высокий уровень безопасности.

**Не злоупотребляйте хешированием!**

Некоторые разработчики, освоив хеш-функции, начинают их комбинировать, что не то, что не увеличивает защиту, а даже ослабляет ее. Но есть также мнение, что такой подход, наоборот, полезен и замедляет скорость взлома. Что можно сказать? Если вы используете соль, то в нагромождении ф-ций нет необходимости.

**Листинг 12. task\_12.py**

|  |
| --- |
| *"""Так делать плохо"""* md5(sha1(password)) md5(md5(salt) + md5(password)) sha1(sha1(password)) sha1(str\_rot13(password + salt)) md5(sha1(md5(md5(password) + sha1(password)) + md5(password))) |

**ПОДВЕДЕМ ИТОГИ**

* Словарь в Python – фундаментальный тип данных, реализованный в виде хеш-таблицы, с открытой адресацией и встроенным методом разрешения коллизий. (применяется метод квадратичного пробинга, когда таблица расширяется при заполнении более чем на 2/3).
* Ключ – обязательно хешируемый объект, т.е. у него должен существовать метод \_\_hash\_\_.
* Словари, благодаря константной сложности, обеспечивает быстрый поиск по ключу.
* Словари требуют больше памяти, т.к. хеш-таблица должна быть достаточно большой для эффективного ее использования.