

Урок 3. Хеширование

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc88654242)

[ЧТО ТАКОЕ ХЕШИРОВАНИЕ И ДЛЯ ЧЕГО ОНО ИСПОЛЬЗУЕТСЯ 3](#_Toc88654243)

[ПРО ХЕШ-ФУНКЦИИ 5](#_Toc88654244)

[ИЗВЕСТНЫЕ ХЕШ-ФУНКЦИИ 6](#_Toc88654245)

[ПРИМЕР ИСПОЛЬЗОВАНИЯ ХЕШ-ФУНКЦИИ 7](#_Toc88654246)

[ПОЧЕМУ НЕ ВСЕ АЛГОРИТМЫ ХЕШИРОВАНИЯ НАДЕЖНЫ 8](#_Toc88654247)

[ЕЩЕ ПРИМЕРЫ РАБОТЫ ХЕШ-ФУНКЦИЙ 9](#_Toc88654248)

[АЛГОРИТМ PBKDF2\_HMAC 10](#_Toc88654249)

[ЧТО ТАКОЕ «СОЛЬ» 12](#_Toc88654250)

[ХЕШ-ТАБЛИЦЫ 13](#_Toc88654251)

[КОЛЛИЗИИ 15](#_Toc88654252)

[НЕ ЗЛОУПОТРЕБЛЯЙТЕ ХЕШИРОВАНИЕМ! 16](#_Toc88654253)

[ПОДВЕДЕМ ИТОГИ ПО ХЕШ-ТАБЛИЦАМ 16](#_Toc88654254)



# ЧТО ТАКОЕ ХЕШИРОВАНИЕ И ДЛЯ ЧЕГО ОНО ИСПОЛЬЗУЕТСЯ

Если взять техническое определение термина, то это процесс трансформации данных любого типа и длины в состоящую из букв и цифр строку фиксированной длины.

Представьте, что вы проходите процедуру регистрации в социальных сетях. При этом вы указываете свой пароль. Если его сохранить в базе данных в явном виде, злоумышленник может получить доступ к данным, похитить пароль и использовать в своих целях. Перспектива не самая лучшая, не так ли?

Здесь нам поможет хеширование и в базу данных мы будем записывать не сам пароль, а его хеш. Да, злоумышленник может получить доступ к хешу, но обратить его в исходный пароль станет уже трудно выполнимой задачей.

Таким образом, процесс хеширования является однонаправленным, т.е. существуют встроенные функции, обеспечивающие получение хеша объекта, но нет функций, которые бы обратили хеш объекта в исходный объект.

Продолжим пример с паролем.

Получается, вы, как пользователь знаете свой пароль. Вы его указали при регистрации, но в базу данных запишется его хеш. О самом пароле будете знать только вы. Когда вы потом попытаетесь снова войти в систему, вы вновь введете свой пароль. Для него будет рассчитан также хеш и, если он совпадет с хешем из базы данных, система поймет, что это на самом деле вы, а не злоумышленник и допустит вас к ресурсу.

Аналогия из жизни: вы приходите на работу и для того, чтобы пройти через проходную, прикладываете палец руки, система понимает, что это вы и впускает вас. Т.е. где-то в базе данных хранится образ вашего отпечатка пальца, т.е. его хеш и при прохождении проверки проверяется сходство реального отпечатка и образа.

Некоторые, считают, что процессы хеширования и шифрования – это одно и то же. На самом деле это не так. Алгоритмы хеширования – однонаправленные, а шифрования – двунаправленные. Т.е. мы можем что-то зашифровать, а затем расшифровать. Шифрование может применяться, например, в мессенджерах при отправке сообщений.

**Обратите внимание:** с шифрованием мы обязательно познакомимся на курсе «Разработка мессенджера». Что же касается процесса хеширования, то с ним мы будем иметь дело опять же на курсе «Разработка мессенджера», когда клиенты-собеседники будут регистрироваться в нашем мессенджере и мы будем вычислять хеши паролей клиентов и записывать их в базу данных. Кроме того, хеши будем вычислять для кодов активации пользователей, зарегистрировавшихся в нашем веб-приложении в рамках курса «Разработка на Django».

Давайте еще раз для закрепления!

Хеширование – это необратимый процесс. Например, вы хотите захешировать некоторую сущность. Но вы не сможете потом восстановить ее исходный вид из 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.

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.

# ПРО ХЕШ-ФУНКЦИИ

Кто же отвечает за создание хеша объекта? Это специализированные хеш-функции. В Python мы будем применять функции из модуля **hashlib**.

Приведем еще одну аналогию.

Представьте, вы хотите превратить куски мяса в фарш. Для этого вы используете мясорубку. После получения фарша, уже будет невозможно узнать о размере исходных кусков и сложно распознать вид мяса.

Очень напоминает хеширование, когда мы берем объект и с помощью хеш-функции получаем для него хеш. И узнать каков был хешируемый объект, становится сложно выполнимой задачей.

Стоит обратить внимание, что любая хеш-функция ВСЕГДА! дает один и тот же хеш для одного и то же объекта. Но эти хеши различаются для различных функций.

Например, функции хеширования **md5()** и **sha1()** дадут следующие хеши для пароля «Testing»: fa6a5a3224d7da66d9e0bdec25f62cf0 и 0820b32b206b7352858e8903a838ed14319acdfd.

И эти хеши всегда будут такими у этих функций для этого пароля!

Еще раз о применимости!

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 функции ничего не шифруют и не дешифруют, но применяются во многих криптографических протоколах и инструментах.

# ИЗВЕСТНЫЕ ХЕШ-ФУНКЦИИ

Как было сказано выше, при работе с хешами в Python мы будем опираться на модуль **hashlib**. В данном модуле реализован программный интерфейс (встроенные функции) для множества алгоритмов безопасного хеширования. Понятия «безопасный хеш» и «дайджест сообщения» являются взаимозаменяемыми. Ранее алгоритмы хеширования назывались дайджестами сообщений, теперь – безопасными хешами.

Каждому типу хеша соответствует один метод конструктора. При этом все конструкторы обеспечивают создание хеш-объектов. Пример такой функции-конструктора – **sha256()**. Эта функция отвечает за создание хеш-объекта SHA-256, который мы затем заполняем байтоподобными объектами.

После этого нам остается запросить у созданного объекта дайджест конкатенации данных, которые мы только что передали созданному объекту. Для решения задачи применяются функции **digest()** и **hexdigest()**.

**hash.digest () –** возвращает дайджест данных, переданных в объект-хеш. Дайджест возвращается в байтовом формате.

**hash.hexdigest() –** тот же принцип, но дайджест возвращается в строковом формате, где строка содержит шестнадцатеричные цифры.

Какими функциями-алгоритмами хеширования мы можем воспользоваться?

**Листинг 1. task\_1.py**

|  |
| --- |
| **import** hashlib  *# список все алгоритмов, доступных в системе* print(hashlib.algorithms\_available) *# существующие алгоритмы модуля* 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'} """** |

Параметр algorithms\_guaranteed ссылается на перечень хеш-алгоритмов, гарантированно поддерживаемых модулем **hashlib** на всех возможных платформах.

Параметр algorithms\_available ссылается на перечень хеш-алгоритмов, доступных в текущем Python-интерпретаторе.

# ПРИМЕР ИСПОЛЬЗОВАНИЯ ХЕШ-ФУНКЦИИ

Рассмотрим реализацию хеширования на примере одной из известных функций – **md5()**.

Эта функция возвращает 128-битное хеш-значение. Применяется исключительно для контроля целостности данных. В других областях применение достаточно редкое по причине доказанной уязвимости функции.

**Листинг 2. task\_2.py**

|  |
| --- |
| *"""Пример с md5"""* **import** hashlib  hash\_obj = hashlib.md5(**b'Testings'**) 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) *# -> 04f957651ea90b4a03bff2cc6ffc6be6* print() hash\_obj\_2 = hashlib.md5(**'Testings'**.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) *# -> 04f957651ea90b4a03bff2cc6ffc6be6* |

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.

Здесь стоит отметить, что функции **md5()** и **sha1()** признаны ненадежными по причине того, что есть случаи распознавания сгенерированных с их помощью хешей. Как так? Мы ведь сказали, что функции однонаправленные и не существует стандартных способов узнать из хеша исходный объект. Это так, но возможности злоумышленников совершенствуются и если они располагают мощными вычислительными ресурсами, то они имеют и возможность распознать хешируемый объект.

# ПОЧЕМУ НЕ ВСЕ АЛГОРИТМЫ ХЕШИРОВАНИЯ НАДЕЖНЫ

Известные алгоритмы хеширования, такие как **sha1**, **sha256**, **md5** характеризуются быстротой вычисления, а это означает, что можно быстро рассчитать различные комбинации строк. Рассчитать и сопоставить с хешем.

При этом применяется подход грубой силы или **«брутфорс»**. В этом случае задача злоумышленника – получить хеши, сгенерировать максимальное количество паролей, выполнить их хеширование и найти соответствие. Подход считается устаревшим.

Но поиск все равно займет значительное время, поэтому эффективнее будет методика **«перебор по словарю»**. Когда у злоумышленника в распоряжении уже есть файл или база данных с потенциально подходящими паролями. В этом случае пароль именно угадывается. Генерации любых возможных комбинаций символов не происходит. Более актуальный и эффективный подход.

Еще одной тактикой является применение **«радужных таблиц»**. Если обратиться к википедии, то это *специальный вариант*[*таблиц поиска*](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0)*(*[*англ.*](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA)*lookup table) для обращения*[*криптографических хеш-функций*](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F)*, использующий механизм*[*разумного компромисса между временем поиска по таблице и занимаемой памятью*](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%80%D0%BE%D0%BC%D0%B8%D1%81%D1%81_%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%B8_%D0%B8_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8)*(*[*англ.*](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA)*time-memory tradeoff)*. Предполагает групповой подход к случайной генерации паролей.

Как же сделать хеш объекта более стойким уловкам злоумышленника? Ответ – «посолить». И это не шутка. Что такое «соль» мы узнаем позднее.

# ЕЩЕ ПРИМЕРЫ РАБОТЫ ХЕШ-ФУНКЦИЙ

Хеш-функции различаются длиной строки (дайджеста).

**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() *# -> 4080aa6d42e2a67c1f6307771ecbe11a23ec8283fd775d72 # 0381844cb3b9e4d038f5446f9db3123bbb4bba588c436f3f* print(hex\_dig\_res)  print()  *# ----------------------------sha512------------------------------------#* hash\_obj = hashlib.sha512(**b'Testing sha1 func'**) hex\_dig\_res = hash\_obj.hexdigest() *# -> d4ea479e7b84b3c71416311af2e79f2919233775f86a8273eaf7e14440a # 306df6ad9587a1d6fe624529118efa2b55740a138276a0630dc0b059066ddaec7a60f* print(type(hex\_dig\_res)) |

# АЛГОРИТМ PBKDF2\_HMAC

Резюмируя вышесказанное, если у злоумышленника находятся в распоряжении значительные вычислительные ресурсы, а также необходимые знания и навыки, то разгадать захешированный объект он точно сможет.

Исходя из этого решением проблемы могло бы стать замедление расчета хешей. Значит для вычисления большого количества хешей будет необходимо больше времени. Время поиска существенно вырастет и обнаружение совпадений может занять десятилетия. Поэтому алгоритм и соответствующая ему функция **pbkdf2\_hmac()** считаются весьма надежными и применяются в production-версиях программных продуктов.

В случае с **pbkdf2\_hmac()** разработчик может самостоятельно установить вычислительные затраты, тем самым замедлив вычисления. Можно увеличить количество итераций в процессе хеширования, т.е. итоговый результат будет вычисляться дольше. Для различного рода атак, особенно для брутфорс, это станет непреодолимым препятствием.

Для одной-двух попыток приведенный механизм проблем не создаст, но для миллионов и миллиардов превратится в сложно разрешимую задачу. Важно, что функция **pbkdf2\_hmac()** не остановит возможности проведения атак со стороны злоумышленника, но сделает их более трудоемкими, а значит сложнее реализуемыми.

**Листинг 4. task\_4.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\_1'**,  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) *# result = type(obj)* print(result) *# -> b'6e97bad21f6200f9087036a71e7ca9fa # 01a59e1d697f7e0284cd7f9b897d7c02'* |

В приведенном примере:

hash\_name=**'sha256'** –это алгоритм получения хеш дайджеста для функции pbkdf2\_hmac()

password=**b'any\_password'** –это пароль, для которого проводится вычисление.

salt=**b'any\_salt\_1'** –соль

iterations=100000 –количество итераций в вычислении (с ростом этого числа растет время вычислений)

При работе с представленным алгоритмом правильно говорить, что генерируется ключ (*b'6e97bad21f6200f9087036a71e7ca9fa01a59e1d697f7e0284cd7f9b897d7c02'*), а не хеш.

# ЧТО ТАКОЕ «СОЛЬ»

Вспомним про пример с фаршем. Если мы прокрутим мясо и добавим соль, то фарш еще изменит свои характеристики и узнать из чего он был сделан, станет еще сложнее. Аналогично в случае с хешами. Соль увеличит затраты злоумышленника при применении методики «грубой силы» и усложнит работу для радужных таблиц.

Соль не требуется как-то скрывать, шифровать. Она хранится в открытом виде. Даже зная ее, злоумышленник никак не сможет облегчить себе задачу. Солью может выступать любой байтовый объект. Можно взять логин пользователя, перевести его в байты и получится готовая соль.

Важно, чтобы соль была уникальной для каждого пользователя. При этом сам пароль не обязательно должен быть уникальным.

Для генерации соли также допустимо пользоваться встроенными средствами, например, функцией **os.urandom()**. Эта функция возвращает случайные байты. Или функцией **uuid4.uuid4()**. Она обеспечит генерацию случайного числа в формате шестнадцатеричной строки.

**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* salt = **'my\_salt'**   passwd = **"programmer"** *# соль-часть хеша* res = hashlib.sha256(salt.encode() + passwd.encode()).hexdigest() *# -> efbb20c297f52672a5211f1358ad8d72907f56e1ff24cd67a6e8b4683a6a18d2* print(res) |

Как мы уже говорили выше для хранения хеша можно воспользоваться привычными редяционными базами данных или файлами формата JSON, CSV или даже простым текстовым файлом. Аналогичным образом можно хранить и соль.

Пример с вычислением парольного хеша и солью.

**Листинг 6. task\_6.py**

|  |
| --- |
| **import** hashlib  *# соль сгенерирована и сохранена для пользовтаеля* salt = **b''** *# ключ мы уже ранее расчитали и "записали" в базу данных* key = **b'\xeb\xe3\xb1\xcc\xc8\x16d\x84\xe6\xa0\xde@'** \  **b'\xdc\xa8\xc18k\n\x8bI[C.ww\x9d\xc2L \xda^\x96'** *# пароль, указанный пользователем* password\_to\_check = **'my\_pass'** new\_key = hashlib.pbkdf2\_hmac(  **'sha256'**,  password\_to\_check.encode(**'utf-8'**),  salt,  100000 )  **if** new\_key == key:  print(**'Пароль верный'**) **else**:  print(**'Пароль неверный'**) |

# ХЕШ-ТАБЛИЦЫ

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

При обычном переборе элементов массива поиск занимает линейное время, которое определяется сложностью O(n).

Если мы отсортируем массив, то применив бинарный поиск, сможем найти элемент массива за логарифмическое время – O (log n). Но и в том случае нам придется затратить время.

Было бы эффективнее сразу находить нужный элемент, независимо от размера исходного массива, т.е. за постоянное время O(1). Здесь нам помогут хеш-таблицы.

Хеш-таблицы имеют и другие названия: «словари», «отображения», «ассоциативные массивы». Операции с хеш-таблицами выполняются исключительно быстро.

Но стоит заметить, что Питонисту никогда не придется программировать логику реализации хеш-таблиц самостоятельно. В любом полноценном языке программирования существует реализация хеш-таблиц.

**Листинг 7. task\_7.py**

|  |
| --- |
| *"""Простейшая хеш-таблица в Python"""* goods = dict()  goods[**'Диван'**] = 25000 goods[**'Кровать'**] = 7000 goods[**'Стул'**] = 1500 goods[1] = 1500  print(goods) *# -> {'Диван': 25000, 'Кровать': 7000, 'Стул': 1500}* |

Хеш-таблицы как средство применяются в мемоизации (кэширование). При мемоизации результаты выполнения функций записываются в хеш-таблицу, где ключом является аргумент функции, а значением – результат выполнения функции. Если функция будет вычисляться от того же аргумента повторно, значение извлекается из кеша, а не получается путем повторного вычисления функции.

**Листинг 8. task\_8.py**

|  |
| --- |
| *"""Мемоизация, как инструмент борьбы с проблемами рекурсии"""  # кэширование - это механизм # хеширование - это средство* **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-значение определяется из ключа в виде числового значения фиксированной длины, однозначно идентифицирующего ключ.

Благодаря этому операции поиска выполняются очень быстро.

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 9. task\_9.py**

|  |
| --- |
| *"""Пример неправильного ключа"""* d = {[1, 2]: **'a'**, 4504.1: **'b'**}  print(d) |

Не забудьте, что ключами словаря (хеш-таблицы) могут быть только хешируемые объекты, например, списки не подойдут и будет сгенерировано исключение.

# 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 хеш-функций нет необходимости.

**Листинг 10. task\_10.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)))  *# поэтому лучше просто "солить" хеши # sha256 + соль* |

# ПОДВЕДЕМ ИТОГИ ПО ХЕШ-ТАБЛИЦАМ

* 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.

**Листинг 11. task\_11.py**

|  |
| --- |
| **from** sys **import** getsizeof **from** pympler.asizeof **import** asizeof   d = {1: **'1'**, 2: **'2'**, 3: **'3'**} print(getsizeof(d)) *# -> 240* print(asizeof(d)) *# -> 504* t = (1, 2, 3) print(getsizeof(t)) *# -> 72* print(asizeof(t)) *# -> 168* |