>>> # Множества и Хеш
>>>

Хеш - Отличная штука для идентификации объектов: такой унифицированный способ определять объекты

>>> # id - вполне себе коротенький хеш
>>> a = "qwertyhbvcdsertyhnbvcd"
>>> id(a)
139682395472080
>>> hex(id(a))
'0x7f0a579040d0'
>>>

Основные требования к хешу:
- Хеш - функция, переводящая одно перечислимое множество в другое пречислимое множество (в дискретное вообще здорово)
- Для одинаковых объектов должен быть одинаковый хеш (однозначная)
- Область значения должна быть меньше области определения (иначе нафига мы вообще делаем хеш, если количество идентификаций не уменьшится или, вообще круто, увеличится? Смысла в хеше тогда нет)

Исходя из этой логики, понимаем, что нам не подходит ***id()***,как хеш: для одинаковых объектов их идентификатор может отличаться (это же разные объекты. То, что они одинаковые по значению, не важно.)

>>> a = (1, 2, 3,)
>>> b = (1, 2, 3,)
>>> id(a)
139682385148864
>>> id(b)
139682385249984
>>> id(a) == id(b)
False
>>>

В Питончике для этих прелестей хеширования есть функция ***hash()***

>>>
>>> hash(a)
5293440672954974 51
>>> hash(b)
5293440672954974 51
>>>
>>>
>>> a = (1, 2, 4,)
>>> hex(id(b))
'0x7f0a56f446c0'
>>> hex(id(a))
'0x7f0a56e165c0'
>>> hex(hash(b))
'0x7589b9fe71bcceb'
>>> hex(hash(a))
'-0x3c8f1547b57f9643'
>>>
>>>
>>> hash(12)
12
>>> hash(12.1)
230584300921368588
>>>

>>> # hash переводит любой КОНСТАНТНЫЙ объект в чиселко
Объект обязательно должен быть константным, иначе мы могли бы вычислить хеш, потом поменять объект и получить равный другому, у которого был свой хеш => не выполняется второй пункт правил

>>> # в Питоне сравнение любых константных объектов начинается со сравнения их хешей. Только если совпадают, то поэлементное.
Это следует из того, что у нас есть правило только на однозначность, но не на взаимную однозначность: у одинаковых объектов обязан быть одинаковый хеш, но у разных не обязан быть разный (Собственно, если бы он был у всех разный, то нарушаем правило про уменьшение области значения)

Работа с хешом дёшевая, тк хеш создаётся вместе с объектом, сразу запоминается и при вызове функции просто выводится
Хеш соответствует методу ***__hash__()***

>>> a = (1, 2, 4,)
>>> a.__hash__()
-4363729961677198915
>>> hash(a)
-4363729961677198915
>>>

>>> **#Множество** - хеш-таблица; способ хеширование - открытая адресация(перехеш); по сути, дин.массив, который удваивается при заполнении где-то на 2/3, тк там хеш уже плохо работает

Множество может хранить любые хешируемые объекты. Реализовано множество, как динамический массив, где индексы это и есть хеши (мб хеши % размер_массива; главное, что до элемента можно добраться через его хеш)

Про открытую адресацию: Как раз таки из-за того, что хеши могут совпадать, на несколько объектов может выпадать один хеш. В некоторых системах указатель просто пробегает элементы после позиции, рассчитанной хешом, проверяя, есть ли объект в множестве(или ищет свободное место для записи нового). При Открытой Адресации вычисляется хеш от хеша (не в чистую, с какими-то там доп.параметрами типа размера массива, текущего положения и ещё чего-то), соответственно получаем новую позицию, проверяем её, потом следующую и так далее.

Про 2/3: чтобы сохранять преимущества по быстрому поиску элементов, удваивание размера дин.массива множества происходит не при полном заполнении таблицы, а при наполнении где-то на 2/3 объёма

```
>>> s = {1, 2, 3, "qwer"}
>>> t = {1, 2, 3, "qwer", [1, 2]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```
вот, список - нехешируемый, идёт нафиг

```
>>> s
{1, 2, 3, 'qwer'}
>>> {'qwe', 'rty', 'asd'}
{'qwe', 'rty', 'asd'}
```

>>> # Сборка
```
>>> {f"/{i*2 + 1}/" for i in range(10, 100, 9)}
{'/57/', '/147/', '/129/', '/165/', '/183/', '/21/', '/111/', '/93/', '/39/', '/75/'}
```

>>> # Константный поиск в среднем(+ линейное масштабирование)
>>>
>>> # Искать можно в множестве(если мы среди какой-то константной конструкции ищем)

Если есть какой-то набор дискретный набор константных параметров, то их можно запихнуть в множество и проверить соответствие с помощью поиска в множестве

Например, чтобы проверить, что s это строка или байтовый массив, можно сделать это через множество

>>> s = "qwert"
>>> if type(s) in {str, bytes, bytearray}:
... print(str(s))
...
qwert
>>> if type(s) in {str, bytes, bytearray}:
... print(str(s))
...
qwert
>>> s = 123
>>> if type(s) in {str, bytes, bytearray}:
... print(str(s))
...
>>>

чиселко 123 это не строка, и не байтовый массив, поэтому не вывелось

>>> # Теоретико-множественные операции

Множества можно преобразовывать. При этом создаётся новое множество, так что чтобы сохранить изменения, не забываем приравнять

```
>>> s, t = set(range(1, 5)), set(range(3, 8))
>>> s
{1, 2, 3, 4}
>>> t
{3, 4, 5, 6, 7}

>>> s | t # объединение
{1, 2, 3, 4, 5, 6, 7}

>>> s & t # пересечение
{3, 4}

>>> s ^ t # объединение без пересечения
{1, 2, 5, 6, 7}

>>> s - t # Разность - первое без второго # Порядок важен!
{1, 2}

>>> t - s
{5, 6, 7}

>>>
>>>
>>> s |= t  # вот и приравняли, сохранили в s результат объединения
>>> s
{1, 2, 3, 4, 5, 6, 7}
>>> s -= {2, 4, 6}
>>> s
{1, 3, 5, 7}

>>> id(s)
139682385008832
>>> s |= {77, 46}
>>> id(s)
139682385008832
```

При применении операций объект не меняется, а модифицируется, поэтому ***id()*** не поменялся

>>> # множество - модифицируемый (значит, нехешируемы) объект, не может быть элементом множества

>>> {1, {2, 3}}
Traceback (most recent call last):
*** File "<stdin>", line 1, in <module>***
TypeError: unhashable type: 'set'

>>> # но можно сделать константное множество
>>>
>>> {1, frozenset((2, 3))}
{1, frozenset({2, 3})}

У Множества есть метод **pop()**, который выбрасывает из множества элементы в порядке сортировки хешей

Пример1
>>> s = {1, 2, 3, 4, 5}
>>> s.pop()
1
>>> s.pop()
2
>>> s.pop()
3
>>> s.pop()
4
>>> s.pop()
5
>>> s.pop()
Traceback (most recent call last):
*** File "<stdin>", line 1, in <module>***
KeyError: 'pop from an empty set'
>>>

Пример2

```
>>> s = {5, 4, 3, 2, 1}
>>> s.pop()
1
>>> s.pop()
2
>>> s.pop()
3
>>> s.pop()
4
>>> s.pop()
5
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'

>>> s = {5, 4, 3, 2, 1}
>>> s
{1, 2, 3, 4, 5}
```

Пример3

```
>>> s = {(12, 13), (14, 15), (3, ), 8}
>>>
>>> s
{8, (12, 13), (3,), (14, 15)} # { обрабатывает элементы в порядке сортировки ключей, поэтому такой порядок}
>>> s.pop()
8
>>> s.pop()
(12, 13)
>>> s.pop()
(3,)
>>> s.pop()
(14, 15)
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
>>>
```

>>> # **Словари:** хранение произвольного объекта, определяемого по хешируемому объекту однозначно
>>>
>>> # ключ - хешируемый объект
>>> # Значение - Любой объект

Вот теперь мы можем хранить в множестве и ехешируемые объекты)) Потому что они определяются хешируемыми

>>>
>>> # Хранение в виде отдельно индексов и отдельно списка добавленных объектов -> элементы можно получить в порядке добавления

Раньше (до py3.7) Словарь хранился в виде большой таблицы с множеством ключей и ссылок на объекты - тяжело, долго, неприятно, внутри куча пустого места, тк всё же разбросано внутри хеш-таблицы

Теперь это выглядит примерно так:

Отдельно журнал индексов: [None, 0, None, None, 2, 1, None, None]

И отдельно хеш список: (Условно, не помню, как выглядела картинка точно; на Вики у Курячего ссылочка есть)
[Hash0, Obj0, obj0_ref,
Hash1, Obj1, obj1_ref,
Hash2, Obj2, obj2_ref]

Ссылочка, чтобы не искать:
https://mail.python.org/pipermail/python-dev/2012-December/123028.html

Главное: Элементы хранятся в хеш-списке по порядку добавления, без дыр, а позиция в хеш-таблице смотрится по журналу

Пример:
>>> d = {f'/{i}/': 2*i + 1 for i in range(24)}
>>> d
{'/0/': 1, '/1/': 3, '/2/': 5, '/3/': 7, '/4/': 9, '/5/': 11, '/6/': 13, '/7/': 15, '/8/': 17, '/9/': 19, '/10/': 21, '/11/': 23, '/12/': 25, '/13/': 27, '/14/': 29, '/15/': 31, '/16/': 33, '/17/': 35, '/18/': 37, '/19/': 39, '/20/': 41, '/21/': 43, '/22/': 45, '/23/': 47} # выведены в порядке добавления
>>> del d['/12/']
>>> d
{'/0/': 1, '/1/': 3, '/2/': 5, '/3/': 7, '/4/': 9, '/5/': 11, '/6/': 13, '/7/': 15, '/8/': 17, '/9/': 19, '/10/': 21, '/11/': 23, '/13/': 27, '/14/': 29, '/15/': 31, '/16/': 33, '/17/': 35, '/18/': 37, '/19/': 39, '/20/': 41, '/21/': 43, '/22/': 45, '/23/': 47}
>>> d['/12/'] = "QQ"
>>> d
{'/0/': 1, '/1/': 3, '/2/': 5, '/3/': 7, '/4/': 9, '/5/': 11, '/6/': 13, '/7/': 15, '/8/': 17, '/9/': 19, '/10/': 21, '/11/': 23, '/13/': 27, '/14/': 29, '/15/': 31, '/16/': 33, '/17/': 35, '/18/': 37, '/19/': 39, '/20/': 41, '/21/': 43, '/22/': 45, '/23/': 47, '/12/': 'QQ'} # ключ '/12/' только добавился, поэтому в конце
>>> d['/15/'] = "QKRQ"
>>> d
{'/0/': 1, '/1/': 3, '/2/': 5, '/3/': 7, '/4/': 9, '/5/': 11, '/6/': 13, '/7/': 15, '/8/': 17, '/9/': 19, '/10/': 21, '/11/': 23, '/13/': 27, '/14/': 29, '/15/': 'QKRQ', '/16/': 33, '/17/': 35, '/18/': 37, '/19/': 39, '/20/': 41, '/21/': 43, '/22/': 45, '/23/': 47, '/12/': 'QQ'} # Мы изменили значение, но не добавляли ничего нового, позиция не поменялась

<u>Важный вывод:</u>
>>> # добавили - порядок добавления поменялся; изменили -не поменялся
>>>
>>>

К составляющим Словаря можно обращаться отдельно с помощью встроенных методов. Всё это на ходу генерируемые последовательности. Они отдельно нигде не хранятся, а просто создаются в момент запроса

>>> d
{'/0/': 1, '/1/': 3, '/2/': 5, '/3/': 7, '/4/': 9, '/5/': 11, '/6/': 13, '/7/': 15, '/8/': 17, '/9/': 19, '/10/': 21, '/11/': 23, '/13/': 27, '/14/': 29, '/15/': 'QKRQ', '/16/': 33, '/17/': 35, '/18/': 37, '/19/': 39, '/20/': 41, '/21/': 43, '/22/': 45, '/23/': 47, '/12/': 'QQ'}

>>> d.keys() # Отдельно список ключей (в порядке добавления)
dict_keys(['/0/', '/1/', '/2/', '/3/', '/4/', '/5/', '/6/', '/7/', '/8/', '/9/', '/10/', '/11/', '/13/', '/14/', '/15/', '/16/', '/17/', '/18/', '/19/', '/20/', '/21/', '/22/', '/23/', '/12/'])

>>> d.values() # Отдельно список значений (в порядке добавления)
dict_values([1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 27, 29, 'QKRQ', 33, 35, 37, 39, 41, 43, 45, 47, 'QQ'])

>>> d.items() # Список пар Ключ-Значение
dict_items([('/0/', 1), ('/1/', 3), ('/2/', 5), ('/3/', 7), ('/4/', 9), ('/5/', 11), ('/6/', 13), ('/7/', 15), ('/8/', 17), ('/9/', 19), ('/10/', 21), ('/11/', 23), ('/13/', 27), ('/14/', 29), ('/15/', 'QKRQ'), ('/16/', 33), ('/17/', 35), ('/18/', 37), ('/19/', 39), ('/20/', 41), ('/21/', 43), ('/22/', 45), ('/23/', 47), ('/12/', 'QQ')])

>>> dd = d.items()
>>> list(dd)
[('/0/', 1), ('/1/', 3), ('/2/', 5), ('/3/', 7), ('/4/', 9), ('/5/', 11), ('/6/', 13), ('/7/', 15), ('/8/', 17), ('/9/', 19), ('/10/', 21), ('/11/', 23), ('/13/', 27), ('/14/', 29), ('/15/', 'QKRQ'), ('/16/', 33), ('/17/', 35), ('/18/', 37), ('/19/', 39), ('/20/', 41), ('/21/', 43), ('/22/', 45), ('/23/', 47), ('/12/', 'QQ')]

последний метод очень удобен для доступа к парных значениям в итерациаонных контрукциях

```
>>> for key, value in d.items():
...     print(key, value)
...
/0/ 1
/1/ 3
/2/ 5
/3/ 7
/4/ 9
/5/ 11
/6/ 13
/7/ 15
/8/ 17
/9/ 19
/10/ 21
/11/ 23
/13/ 27
/14/ 29
/15/ QKRQ
/16/ 33
/17/ 35
/18/ 37
/19/ 39
/20/ 41
/21/ 43
/22/ 45
/23/ 47
/12/ QQ
>>>
```

Можно сделать то же самое через индексацию (но никто в здравом уме так не делает)

```
>>> d
{'/0/': 1, '/1/': 3, '/2/': 5, '/3/': 7, '/4/': 9, '/5/': 11, '/6/': 13, '/7/': 15, '/8/': 17, '/9/': 19, '/10/': 21, '/11/': 23, '/13/': 27, '/14/': 29, '/15/': 'QKRQ', '/16/': 33, '/17/': 35, '/18/': 37, '/19/': 39, '/20/': 41, '/21/': 43, '/22/': 45, '/23/': 47, '/12/': 'QQ'}
>>> list(d)
['/0/', '/1/', '/2/', '/3/', '/4/', '/5/', '/6/', '/7/', '/8/', '/9/', '/10/', '/11/', '/13/', '/14/', '/15/', '/16/', '/17/', '/18/', '/19/', '/20/', '/21/', '/22/', '/23/', '/12/']
>>> for k in d:
...     print(k, d[k])
...
/0/ 1
/1/ 3
/2/ 5
/3/ 7
/4/ 9
/5/ 11
/6/ 13
/7/ 15
/8/ 17
/9/ 19
/10/ 21
/11/ 23
/13/ 27
/14/ 29
/15/ QKRQ
/16/ 33
/17/ 35
/18/ 37
/19/ 39
/20/ 41
/21/ 43
/22/ 45
/23/ 47
/12/ QQ
>>> ###  Но так мы не делаем: можно поломать словарь; да и это тупо дольше
>>>
```

Вот-вот, что я и говорил)))

Касательно доступа к данным по ключу: У нас есть парочка методов, которые также выдаёт нам значение по ключу, но лучше.

```
>>> d
{'/0/': 1, '/1/': 3, '/2/': 5, '/3/': 7, '/4/': 9, '/5/': 11, '/6/': 13, '/7/': 15, '/8/': 17, '/9/': 19, '/10/': 21, '/11/': 23, '/13/': 27, '/14/': 29, '/15/': 'QKRQ', '/16/': 33, '/17/': 35, '/18/': 37, '/19/': 39, '/20/': 41, '/21/': 43, '/22/': 45, '/23/': 47, '/12/': 'QQ'}
>>> d["/3/"]
7
>>> d['/33/']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '/33/'
```

Обычная индексация при попадании в несуществующий ключ выдаёт ошибку

```
>>> d.get('/3/')
7
>>> d.get('/33/')
>>> print(d.get('/33/'))
None
>>> print(d.get('/33/', 'QQ'))
QQ
```

метод get проверяет, есть ли значение по ключу и возвращает None, если значения не существует. Если подать второй параметр, то вместо None выведется он

Поэтому можно писать вот такую дичь:

```
>>> d = {1: 1, 2: 2}
>>> d
{1: 1, 2: 2}
>>> d.get(1)
1
>>> d.get(123, "Nope")
'Nope'
>>> d.get(123, d.get(2, "Nope"))
2
>>> d.get(123, d.get(23, "Nope")) # Больше вложенностей богу вложенностей
'Nope'
```

Ещё одна крутая штука - ***setdefault()***. В него подаются строго два параметра - ключ и дефолт. Если ключа в словаре не существует, то он заводится со значением дефолта

```
>>> d = {}
>>> type(d)
<class 'dict'>
>>>
>>> d[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>> d.setdefault(3, "QQ")
'QQ'
>>> d
{3: 'QQ'}
>>>
```

Просто все методы словарика

```
>>>
>>> dir(d)
['__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__ior__', '__iter__',
'__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__ror__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
>>>
```

Тк словарь это тоже множество(просто для парных элементов как бы), к ним применимы **теоретико-множественные операции**

>>>
>>> d = dict(zip(range(1, 10), range(21, 30)))
>>> c = dict(zip(range(6, 16), range(126, 136)))

>>> d
{1: 21, 2: 22, 3: 23, 4: 24, 5: 25, 6: 26, 7: 27, 8: 28, 9: 29}
>>> c
{6: 126, 7: 127, 8: 128, 9: 129, 10: 130, 11: 131, 12: 132, 13: 133, 14: 134, 15: 135}

>>> c | d
{6: 26, 7: 27, 8: 28, 9: 29, 10: 130, 11: 131, 12: 132, 13: 133, 14: 134, 15: 135, 1: 21, 2: 22, 3: 23, 4: 24, 5: 25}

>>> # значения старого обновляются по новым, добавляются недостаток из второго. # Обновление первого вторым
Это тоже новый объект, чтобы его сохранить, надо приравнять

>>> ## Где используют словари - ВЕЗДЕ
>>>
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'a': (1, 2, 4), 'b': (1, 2, 3), 's': set(), 't': {3, 4, 5, 6, 7}, 'd': {1: 21, 2: 22, 3: 23, 4: 24, 5: 25, 6: 26, 7: 27, 8: 28, 9: 29}, 'dd': dict_items([('/0/', 1), ('/1/', 3), ('/2/', 5), ('/3/', 7), ('/4/', 9), ('/5/', 11), ('/6/', 13), ('/7/', 15), ('/8/', 17), ('/9/', 19), ('/10/', 21), ('/11/', 23), ('/13/', 27), ('/14/', 29), ('/15/', 'QKRQ'), ('/16/', 33), ('/17/', 35), ('/18/', 37), ('/19/', 39), ('/20/', 41), ('/21/', 43), ('/22/', 45), ('/23/', 47), ('/12/', 'QQ')]), 'key': '/12/', 'value': 'QQ', 'k': '/12/', 'c': {6: 126, 7: 127, 8: 128, 9: 129, 10: 130, 11: 131, 12: 132, 13: 133, 14: 134, 15: 135}}

>>> res = globals()
>>> type(res)
<class 'dict'>
>>> res['c'][11] = "QKRQ"
>>> c
{6: 126, 7: 127, 8: 128, 9: 129, 10: 130, 11: 'QKRQ', 12: 132, 13: 133, 14: 134, 15: 135}
>>> res["aaa"] = 123 # Да, мы прямо через словарик создали переменную в ***globals()***
>>> aaa
123

>>> **# Именный параметры функции**

Как мы помним (или делаем вид { :-) } Сейчас можно сделать серьёзное лицо и сказать мне: "Ну и дурак ты Стёпа, я всё это, конечно, знаю", а я тебе отвечу: "Слушайте дальше"), в функциях параметры делятся на Позиционные, Смешанные и Именные. В параметрах функции у каждого блока есть свои определённые отсеки. Внутри отсека: Позиционные подаются без названий в строго определённом порядке, Смешанные могут либо подаваться без имени, и тогда они встают на соответствующую им позицию, или с указанием имени, Именные подаются в любом порядке с явным указанием имени.

Так вот, именные параметры в функцию передаются как раз в виде словарика:

```
>>>
>>> def fun(*args, **kwargs):
...     print(args, kwargs)
...
>>> fun(1, 2, 3, a = "@", b = 'e')
(1, 2, 3) {'a': '@', 'b': 'e'}

>>> fun(**{'-=-=': 123, 'gnu': 'is not unix'})
() {'-=-=': 123, 'gnu': 'is not unix'}
```

>>> # **передача пространства имён, как параметра**

>>> help(eval)

>>> #eval(source, globals=None, locals=None, /)
>>> # Evaluate the given source in the context of globals and locals.
>>>

В Евале (выше как раз его описание из Хелпа) есть доп.параметры: его собственные globals и locals. И их можно задавать. Тогда евал сначала посмотрит в свой Локалс за значениями переменных, потом в свой глобалс, потом только в общие локалс и глобалс

>>> eval("a*b")
Traceback (most recent call last):
*** File "<stdin>", line 1, in <module>***
*** File "<string>", line 1, in <module>***
NameError: name 'a' is not defined # Ругается, тк нет у нас переменной a

>>> eval("a*b", {'a': 123, 'b': 234, 'c': 345}) # Подаём ему наш глобалс, где она есть
28782

>>> # сначала в локалс, потом в глобалс
>>> eval("a + b", {'a': 1, 'b': 2}, {'a': 3, 'b': 4})
7

>>> help(exec) # Работает тоже со своими локалс и глобалс