```
talk = {
    'title':  'What is a Python dictionary?',
    'person': 'Iain Watts (Fusionbox)',
    'where':  'Denver Python Meetup!',
    'with':   ['Pizza', 'Beer'],
}
```

```
talk = {
    'title':  'What is a Python dictionary?',
    'person': 'Iain Watts (Fusionbox)',
    'where':  'Denver Python Meetup!',
    'with':   ['Pizza', 'Beer'],
}
```

Keys

Values

```
>>> talk['title']
'What is a Python dictionary?'
>>> talk['alternative_title'] = 'Hashmaps are awesome!'
>>> talk['alternative_title']
'Hashmaps are awesome!'
>>> 'where' in talk
True
>>> 'foobar' in talk
False
>>> talk['foobar']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'foobar'
```

Dictionaries are in Python's soul

Python:

```
places = {
    (1, 30): 'House',
    (-2, 6): 'Tree',
    (13, 9): 'Rock',
}
```

Java:

```
HashMap<List<Integer>, String> places;
places.put(Arrays.asList(1, 30), "House");
places.put(Arrays.asList(-2, 6), "Tree");
places.put(Arrays.asList(13, 9), "Rock");`
```

C++: Censored for reasons of sanity

Some features of dictionaries that will be explained when we look under the hood:

Fast lookup, insertion, and deletion:

```
elem in my_dict              # O(1) - constant time
my_dict['foo'] = 'bar'       # O(1) - constant time
del my_dict['foo']           # O(1) - constant time

elem in my_list              # O(N) - linear in size of list
my_list.insert(42, 'bar')    # O(N) - linear in size of list
del my_list[42]              # O(N) - linear in size of list
```

Arbitrary ordering:

```
for key, val in my_dict.items():
    # items will be iterated over in arbitrary order
```

Some features of dictionaries that will be explained when we look under the hood:

Some things can't be keys:

```
>>> my_dict = {[1, 2]: 'foobar'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Don't try to add or remove items while iterating over a dictionary!

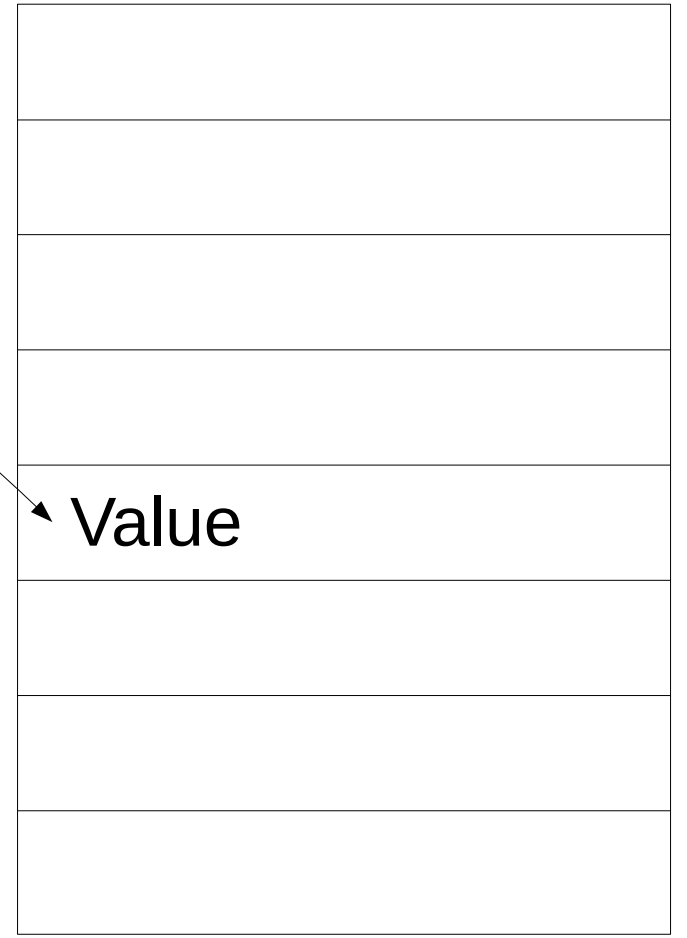
```
>>> my_dict = {1: 'pizza', 2: 'beer', 3: 'broccoli'}
>>> for key, val in my_dict.items():
...     if val == 'broccoli':
...         del my_dict[key]
...     else:
...         print(val)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

The Hash Map (simplified):

hash(key)

e.g. -4099108754737663647

key
(e.g. ‘apple’)



hash(key) translates to some memory location in the Hash Table, where the value is stored

Hence the fast O(1) lookup – knowing the key, the computer can go straight to the memory address to get the value. And that takes constant time.

So what is the hash() function?

An algorithm for translating an object to an integer

An object is *hashable* if:

1) hash() returns the same value over the lifetime of the object

2) if a == b then hash(a) == hash(b)

hash() is a Python ‘builtin’ - we can call it on any hashable object to get the hash:

```
>>> hash('apple')
-4099108754737663647
```

What's hashable?

```
>>> # Integers:
... hash(42)
42

>>> # Strings:
... hash('apple')
9170153690556920463

>>> # Tuples:
... hash((9, -5))
-3713072971715812669

>>> # Frozensets:
... hash(frozenset([5, 9, 2]))
8280055867996868246

>>> # Functions:
... hash(lambda x: x**2)
-9223632973068613 43
>>>
>>> def foo(a):
...     return a + 'bar'
...
>>> hash(foo)
8739547914457
```

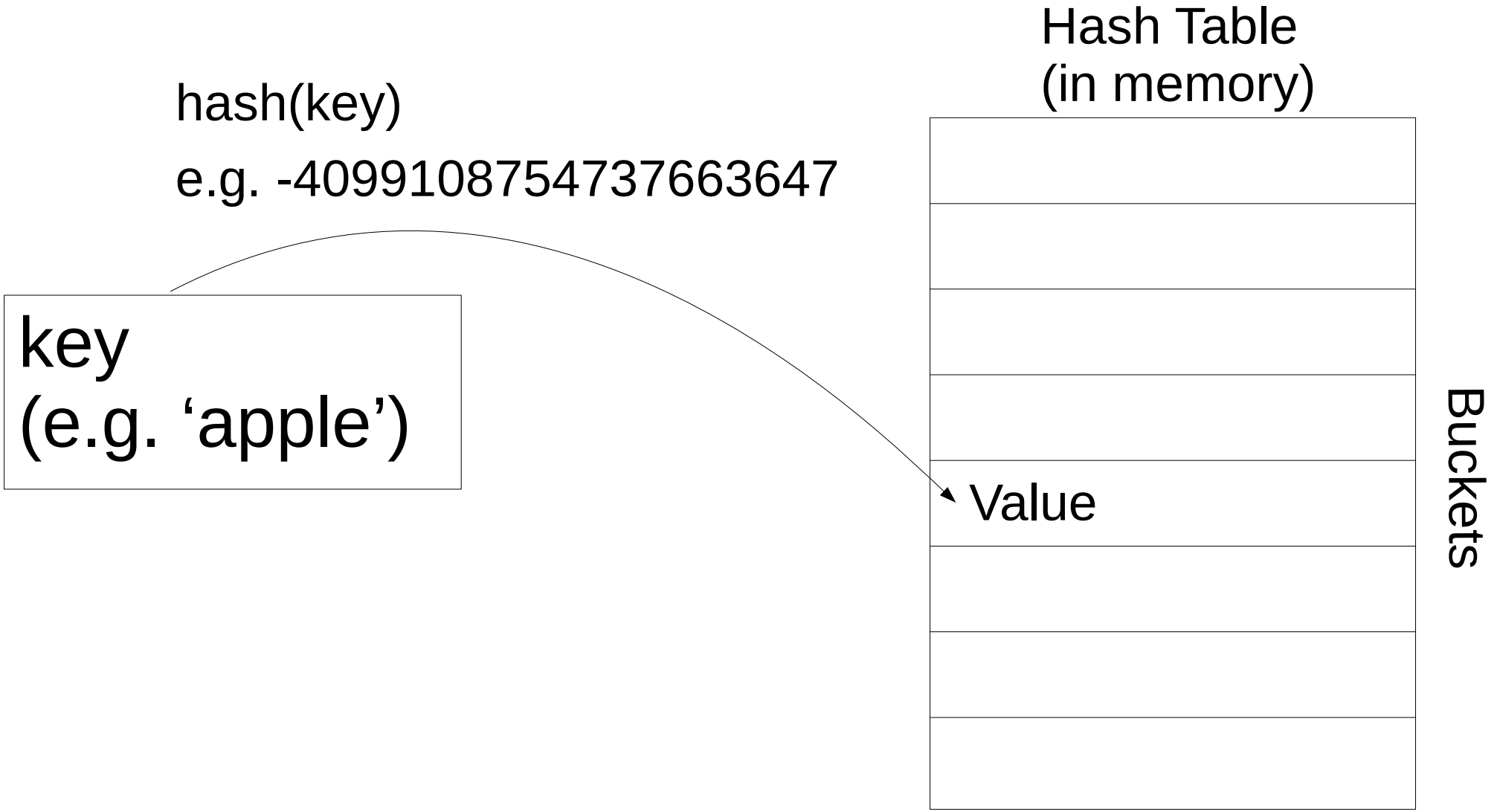
What's not hashable (i.e. can't be a key)?

Mutable objects*

```
>>> # Lists:
... hash([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unhashable type: 'list'
>>>
>>> # Dictionaries and sets:
... hash({1: 'foo', 2: 'bar'})
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unhashable type: 'dict'
>>>
>>> # Tuples containing mutable objects:
... hash((1, [2, 3]))
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unhashable type: 'list'
```

* or immutable objects that contain mutable objects
(looking at you, tuples!)

A closer look at CPython's dictionary implementation



A closer look at CPython's dictionary implementation

```
my_dict['pizza']
```

Key: 'pizza'

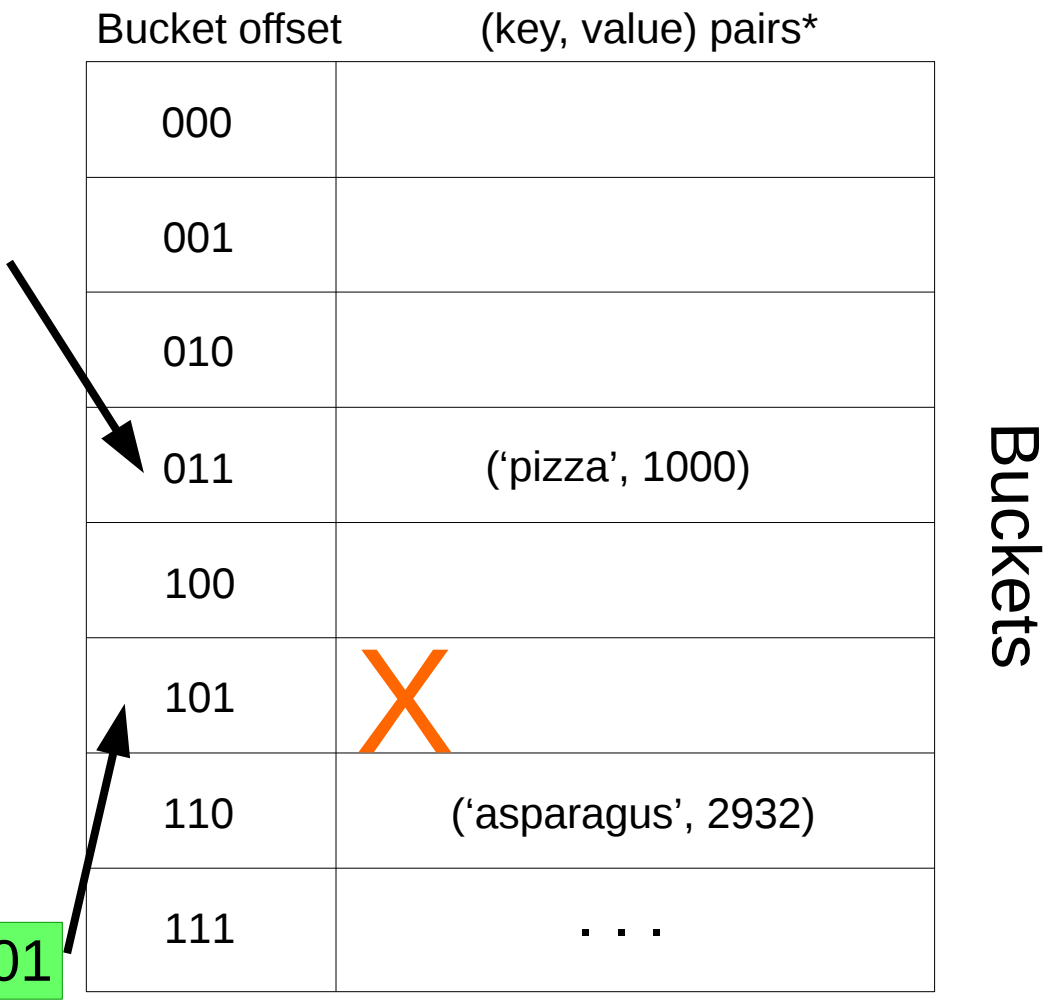
hash('pizza'): 0101110101011011

Size of table is 8, so we get the bucket by taking the 3 least significant bits of the hash (=> arbitrary ordering!)

```
my_dict['spinach']
```

Key: 'spinach'

hash('spinach'): 0100001101101101



* actually they're references

A closer look at CPython's dictionary implementation

But what about collisions??
(hashes have same last three bits)?

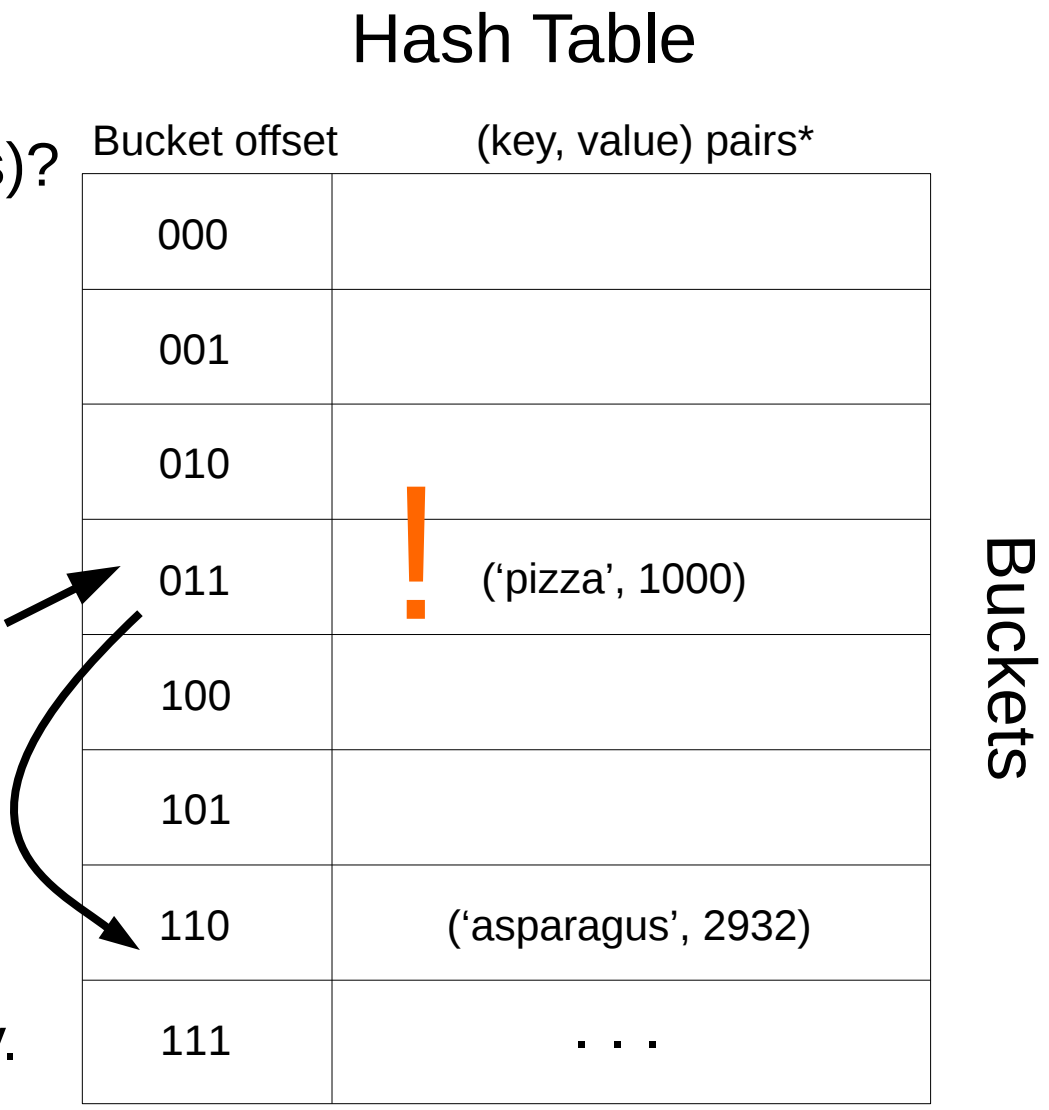
```
my_dict['asparagus']
```

Key: 'asparagus'

hash('asparagus'):
0111010110010011

Collision resolution algorithm:
('Open addressing')

Feed the hash into a recurrence relation that gets a new row to try. If there's a collision there, repeat and continue.



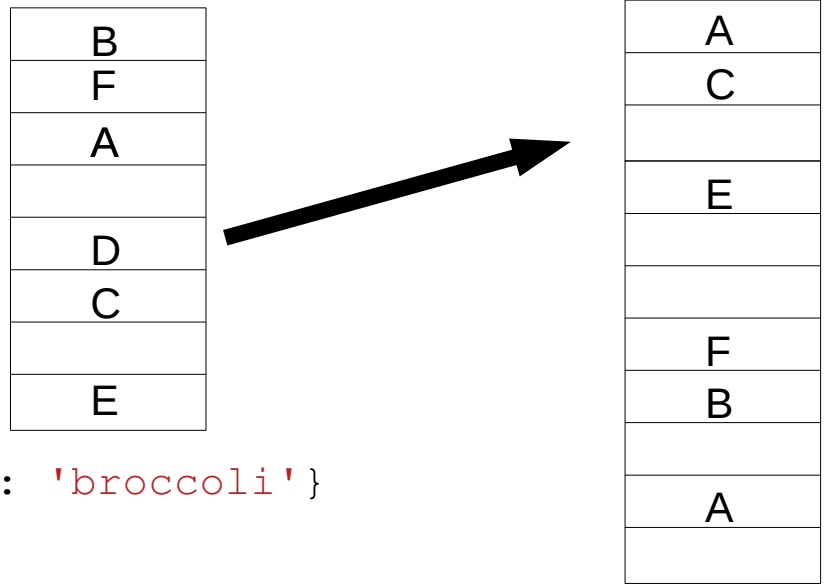
* actually they're references

What happens if the table gets too full?

When it's about 2/3 full, Python will resize the table by copying the data to a new memory location. Similar to how lists grow in 'O(1) amortized time'.

That's why we can't add or delete items while iterating!

When we add or delete an item, it could result in a table resizing.



And that would shuffle the order of the items, breaking the iteration!

```
>>> my_dict = {1: 'pizza', 2: 'beer', 3: 'broccoli'}
>>> for key, val in my_dict.items():
...     if val == 'broccoli':
...         del my_dict[key]
...     else:
...         print(val)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

 # Thanks for listening!

```
talk = {
    'title':  'What is a Python dictionary?',
    'person': 'Iain Watts (Fusionbox)',
    'where':  'Denver Python Meetup!',
    'with':   ['Pizza', 'Beer'],
}
```