Table of Contents

Hash Data Syntax
Common Uses
Creating a Hash
Hash Value Basics
Entry Order
Hash Keys
Hash Key Equivalence
Modifying an Active Hash Key
User-Defined Hash Keys
Default Values
Default Proc
What's Here
Methods for Creating a Hash
Methods for Setting Hash State
Methods for Querying
Methods for Comparing
Methods for Fetching
Methods for Assigning
Methods for Deleting
Methods for Iterating
Methods for Converting
Methods for Transforming Keys and Values
Other Methods

Parent

Object

Included Modules

Enumerable

Methods

::[]
::new
::ruby2_keywords_hash
::ruby2_keywords_hash?
::try_convert
#<
#<=
#==
#>
#>=
#[]
#[]=
#any?
#assoc
#clear

#compact
#compact!
#compare_by_identity
#compare_by_identity?
#deconstruct_keys
#default
#default=
#default_proc
#default_proc=
#delete
#delete_if
#dig
#each
#each_key
#each_pair
#each_value
#empty?
#eql?
#except
#fetch
#fetch_values
#filter
#filter!
#flatten
#has_key?
#has_value?
#hash
#include?
#initialize_copy
#inspect
#invert
#keep_if
#key
#key?
#keys
#length
#member?
#merge
#merge!
#rassoc
#rehash
#reject
#reject!
#replace
#select
#select!
#shift
#size
#slice
#store
#to_a
#to_h
#to_hash
#to_proc
#to_s
#transform_keys
#transform_keys!
#transform_values
#transform_values!

#update
#value?
#values
#values_at

class Hash

A Hash maps each of its unique keys to a specific value.

A Hash has certain similarities to an `Array`, but:

- An `Array` index is always an `Integer`.
- A Hash key can be (almost) any object.

Hash Data Syntax

The older syntax for Hash data uses the “hash rocket,” `=>`:

```
h = {:foo => 0, :bar => 1, :baz => 2}
h # => {:foo=>0, :bar=>1, :baz=>2}
```

Alternatively, but only for a Hash key that’s a `Symbol`, you can use a newer JSON-style syntax, where each bareword becomes a Symbol:

```
h = {foo: 0, bar: 1, baz: 2}
h # => {:foo=>0, :bar=>1, :baz=>2}
```

You can also use a `String` in place of a bareword:

```
h = {'foo': 0, 'bar': 1, 'baz': 2}
h # => {:foo=>0, :bar=>1, :baz=>2}
```

And you can mix the styles:

```
h = {foo: 0, :bar => 1, 'baz': 2}
h # => {:foo=>0, :bar=>1, :baz=>2}
```

But it's an error to try the JSON-style syntax for a key that's not a bareword or a String:

```
# Raises SyntaxError (syntax error, unexpected ':', expecting =>):
h = {0: 'zero'}
```

Hash value can be omitted, meaning that value will be fetched from the context by the name of the key:

```
x = 0
y = 100
h = {x:, y:}
h # => {:x=>0, :y=>100}
```

Common Uses

You can use a Hash to give names to objects:

```
person = {name: 'Matz', language: 'Ruby'}
person # => {:name=>"Matz", :language=>"Ruby"}
```

You can use a Hash to give names to method arguments:

```
def some_method(hash)
  p hash
end
some_method({foo: 0, bar: 1, baz: 2}) # => {:foo=>0, :bar=>1, :baz=>2}
```

Note: when the last argument in a method call is a Hash, the curly braces may be omitted:

```
some_method(foo: 0, bar: 1, baz: 2) # => {:foo=>0, :bar=>1, :baz=>2}
```

You can use a Hash to initialize an object:

```
class Dev
  attr_accessor :name, :language
  def initialize(hash)
    self.name = hash[:name]
    self.language = hash[:language]
```

```
  end
end
matz = Dev.new(name: 'Matz', language: 'Ruby')
matz # => #<Dev: @name="Matz", @language="Ruby">
```

Creating a Hash

You can create a Hash object explicitly with:

- A hash literal.

You can convert certain objects to Hashes with:

- Method `Hash`.

You can create a Hash by calling method `Hash.new`.

Create an empty Hash:

```
h = Hash.new
h # => {}
h.class # => Hash
```

You can create a Hash by calling method `Hash.[]`.

Create an empty Hash:

```
h = Hash[]
h # => {}
```

Create a Hash with initial entries:

```
h = Hash[foo: 0, bar: 1, baz: 2]
h # => {:foo=>0, :bar=>1, :baz=>2}
```

You can create a Hash by using its literal form (curly braces).

Create an empty Hash:

```
h = {}
h # => {}
```

Create a Hash with initial entries:

```
h = {foo: 0, bar: 1, baz: 2}
h # => {:foo=>0, :bar=>1, :baz=>2}
```

Hash Value Basics

The simplest way to retrieve a Hash value (instance method `[]`):

```
h = {foo: 0, bar: 1, baz: 2}
h[:foo] # => 0
```

The simplest way to create or update a Hash value (instance method `[]=`):

```
h = {foo: 0, bar: 1, baz: 2}
h[:bat] = 3 # => 3
h # => {:foo=>0, :bar=>1, :baz=>2, :bat=>3}
h[:foo] = 4 # => 4
h # => {:foo=>4, :bar=>1, :baz=>2, :bat=>3}
```

The simplest way to delete a Hash entry (instance method `delete`):

```
h = {foo: 0, bar: 1, baz: 2}
h.delete(:bar) # => 1
h # => {:foo=>0, :baz=>2}
```

Entry Order

A Hash object presents its entries in the order of their creation. This is seen in:

- Iterative methods such as `each`, `each_key`, `each_pair`, `each_value`.
- Other order-sensitive methods such as `shift`, `keys`, `values`.
- The `String` returned by method `inspect`.

A new Hash has its initial ordering per the given entries:

```
h = Hash[foo: 0, bar: 1]
h # => {:foo=>0, :bar=>1}
```

New entries are added at the end:

```
h[:baz] = 2
h # => {:foo=>0, :bar=>1, :baz=>2}
```

Updating a value does not affect the order:

```
h[:baz] = 3
h # => {:foo=>0, :bar=>1, :baz=>3}
```

But re-creating a deleted entry can affect the order:

```
h.delete(:foo)
h[:foo] = 5
```

```
h # => {:bar=>1, :baz=>3, :foo=>5}
```

Hash Keys

Hash Key Equivalence

Two objects are treated as the same hash key when their `hash` value is identical and the two objects are `eql?` to each other.

Modifying an Active Hash Key

Modifying a Hash key while it is in use damages the hash's index.

This Hash has keys that are Arrays:

```
a0 = [ :foo, :bar ]
a1 = [ :baz, :bat ]
h = {a0 => 0, a1 => 1}
h.include?(a0) # => true
h[a0] # => 0
a0.hash # => 110002110
```

Modifying array element `a0[0]` changes its hash value:

```
a0[0] = :bam
a0.hash # => 1069447059
```

And damages the Hash index:

```
h.include?(a0) # => false
h[a0] # => nil
```

You can repair the hash index using method `rehash`:

```
h.rehash # => {[:bam, :bar]=>0, [:baz, :bat]=>1}
h.include?(a0) # => true
h[a0] # => 0
```

A `String` key is always safe. That's because an unfrozen `String` passed as a key will be replaced by a duplicated and frozen String:

```
s = 'foo'
s.frozen? # => false
h = {s => 0}
first_key = h.keys.first
first_key.frozen? # => true
```

User-Defined Hash Keys

To be useable as a Hash key, objects must implement the methods `hash` and `eql?`. Note: this requirement does not apply if the Hash uses `compare_by_identity` since comparison will then rely on the keys' object id instead of `hash` and `eql?`.

`Object` defines basic implementation for `hash` and `eq?` that makes each object a distinct key. Typically, user-defined classes will want to override these methods to provide meaningful behavior, or for example inherit `Struct` that has useful definitions for these.

A typical implementation of `hash` is based on the object's data while `eql?` is usually aliased to the overridden `==` method:

```
class Book
  attr_reader :author, :title

  def initialize(author, title)
    @author = author
    @title = title
  end

  def ==(other)
    self.class === other &&
      other.author == @author &&
      other.title == @title
  end

  alias eql? ==

  def hash
    @author.hash ^ @title.hash # XOR
  end
end

book1 = Book.new 'matz', 'Ruby in a Nutshell'
book2 = Book.new 'matz', 'Ruby in a Nutshell'

reviews = {}

reviews[book1] = 'Great reference!'
reviews[book2] = 'Nice and compact!'

reviews.length #=> 1
```

Default Values

The methods `[]`, `values_at` and `dig` need to return the value associated to a certain key. When that key is not found, that value will be determined by its default proc (if any) or else its default (initially 'nil`).

You can retrieve the default value with method `default`:

```
h = Hash.new
h.default # => nil
```

You can set the default value by passing an argument to method `Hash.new` or with method `default=`

```
h = Hash.new(-1)
h.default # => -1
h.default = 0
h.default # => 0
```

This default value is returned for `[]`, `values_at` and `dig` when a key is not found:

```
counts = {foo: 42}
counts.default # => nil (default)
counts[:foo] = 42
counts[:bar] # => nil
counts.default = 0
counts[:bar] # => 0
counts.values_at(:foo, :bar, :baz) # => [42, 0, 0]
counts.dig(:bar) # => 0
```

Note that the default value is used without being duplicated. It is not advised to set the default value to a mutable object:

```
synonyms = Hash.new([])
synonyms[:hello] # => []
synonyms[:hello] << :hi # => [:hi], but this mutates the default!
synonyms.default # => [:hi]
synonyms[:world] << :universe
synonyms[:world] # => [:hi, :universe], oops
synonyms.keys # => [], oops
```

To use a mutable object as default, it is recommended to use a default proc

Default `Proc`

When the default proc for a Hash is set (i.e., not `nil`), the default value returned by method `[]` is determined by the default proc alone.

You can retrieve the default proc with method `default_proc`:

```
h = Hash.new
h.default_proc # => nil
```

You can set the default proc by calling `Hash.new` with a block or calling the method `default_proc=`

```
h = Hash.new { |hash, key| "Default value for #{key}" }
h.default_proc.class # => Proc
h.default_proc = proc { |hash, key| "Default value for #{key.inspect}" }
h.default_proc.class # => Proc
```

When the default proc is set (i.e., not `nil`) and method `[]` is called with with a non-existent key, `[]` calls the default proc with both the Hash object itself and the missing key, then returns the proc's return value:

```
h = Hash.new { |hash, key| "Default value for #{key}" }
h[:nosuch] # => "Default value for nosuch"
```

Note that in the example above no entry for key `:nosuch` is created:

```
h.include?(:nosuch) # => false
```

However, the proc itself can add a new entry:

```
synonyms = Hash.new { |hash, key| hash[key] = [] }
synonyms.include?(:hello) # => false
synonyms[:hello] << :hi # => [:hi]
synonyms[:world] << :universe # => [:universe]
synonyms.keys # => [:hello, :world]
```

Note that setting the default proc will clear the default value and vice versa.

Be aware that a default proc that modifies the hash is not thread-safe in the sense that multiple threads can call into the default proc concurrently for the same key.

What's Here

First, what's elsewhere. Class Hash:

- Inherits from class Object.
- Includes module Enumerable, which provides dozens of additional methods.

Here, class Hash provides methods that are useful for:

- Creating a Hash
- Setting Hash State
- Querying
- Comparing
- Fetching
- Assigning
- Deleting
- Iterating
- Converting
- Transforming Keys and Values
- And more....

Class Hash also includes methods from module `Enumerable`.

Methods for Creating a Hash

- `::[]`: Returns a new hash populated with given objects.
- `::new`: Returns a new empty hash.
- `::try_convert`: Returns a new hash created from a given object.

Methods for Setting Hash State

- `compare_by_identity`: Sets `self` to consider only identity in comparing keys.
- `default=`: Sets the default to a given value.
- `default_proc=`: Sets the default proc to a given proc.
- `rehash`: Rebuilds the hash table by recomputing the hash index for each key.

Methods for Querying

- `any?`: Returns whether any element satisfies a given criterion.
- `compare_by_identity?`: Returns whether the hash considers only identity when comparing keys.
- `default`: Returns the default value, or the default value for a given key.
- `default_proc`: Returns the default proc.
- `empty?`: Returns whether there are no entries.
- `eql?`: Returns whether a given object is equal to `self`.
- `hash`: Returns the integer hash code.
- `has_value?`: Returns whether a given object is a value in `self`.
- `include?`, `has_key?`, `member?`, `key?`: Returns whether a given object is a key in `self`.
- `length`, `size`: Returns the count of entries.
- `value?`: Returns whether a given object is a value in `self`.

Methods for Comparing

- #<: Returns whether `self` is a proper subset of a given object.
- #<=: Returns whether `self` is a subset of a given object.
- `==`: Returns whether a given object is equal to `self`.
- #>: Returns whether `self` is a proper superset of a given object

- #>=: Returns whether `self` is a superset of a given object.

Methods for Fetching

- `[]` : Returns the value associated with a given key.
- `assoc` : Returns a 2-element array containing a given key and its value.
- `dig` : Returns the object in nested objects that is specified by a given key and additional arguments.
- `fetch` : Returns the value for a given key.
- `fetch_values` : Returns array containing the values associated with given keys.
- `key` : Returns the key for the first-found entry with a given value.
- `keys` : Returns an array containing all keys in `self` .
- `rassoc` : Returns a 2-element array consisting of the key and value of the first-found entry having a given value.
- `values` : Returns an array containing all values in `self` /
- `values_at` : Returns an array containing values for given keys.

Methods for Assigning

- `[]=` , `store` : Associates a given key with a given value.
- `merge` : Returns the hash formed by merging each given hash into a copy of `self` .
- `merge!` , `update` : Merges each given hash into `self` .
- `replace` : Replaces the entire contents of `self` with the contents of a given hash.

Methods for Deleting

These methods remove entries from `self` :

- `clear` : Removes all entries from `self` .
- `compact!` : Removes all `nil` -valued entries from `self` .
- `delete` : Removes the entry for a given key.
- `delete_if` : Removes entries selected by a given block.
- `filter!` , `select!` : Keep only those entries selected by a given block.
- `keep_if` : Keep only those entries selected by a given block.
- `reject!` : Removes entries selected by a given block.
- `shift` : Removes and returns the first entry.

These methods return a copy of `self` with some entries removed:

- `compact`: Returns a copy of `self` with all `nil`-valued entries removed.
- `except`: Returns a copy of `self` with entries removed for specified keys.
- `filter`, `select`: Returns a copy of `self` with only those entries selected by a given block.
- `reject`: Returns a copy of `self` with entries removed as specified by a given block.
- `slice`: Returns a hash containing the entries for given keys.

Methods for Iterating

- `each`, `each_pair`: Calls a given block with each key-value pair.
- `each_key`: Calls a given block with each key.
- `each_value`: Calls a given block with each value.

Methods for Converting

- `inspect`, `to_s`: Returns a new `String` containing the hash entries.
- `to_a`: Returns a new array of 2-element arrays; each nested array contains a key-value pair from `self`.
- `to_h`: Returns `self` if a Hash; if a subclass of Hash, returns a Hash containing the entries from `self`.
- `to_hash`: Returns `self`.
- `to_proc`: Returns a proc that maps a given key to its value.

Methods for Transforming Keys and Values

- `transform_keys`: Returns a copy of `self` with modified keys.
- `transform_keys!`: Modifies keys in `self`
- `transform_values`: Returns a copy of `self` with modified values.
- `transform_values!`: Modifies values in `self`.

Other Methods

- `flatten`: Returns an array that is a 1-dimensional flattening of `self`.
- `invert`: Returns a hash with the each key-value pair inverted.

Public Class Methods

Hash[] → new_empty_hash
Hash[hash] → new_hash
Hash[[*2_element_arrays]] → new_hash
Hash[*objects] → new_hash

Returns a new Hash object populated with the given objects, if any. See `Hash::new`.

With no argument, returns a new empty Hash.

When the single given argument is a Hash, returns a new Hash populated with the entries from the given Hash, excluding the default value or proc.

```
h = {foo: 0, bar: 1, baz: 2}
Hash[h] # => {:foo=>0, :bar=>1, :baz=>2}
```

When the single given argument is an `Array` of 2-element Arrays, returns a new Hash object wherein each 2-element array forms a key-value entry:

```
Hash[ [ [:foo, 0], [:bar, 1] ] ] # => {:foo=>0, :bar=>1}
```

When the argument count is an even number; returns a new Hash object wherein each successive pair of arguments has become a key-value entry:

```
Hash[:foo, 0, :bar, 1] # => {:foo=>0, :bar=>1}
```

Raises an exception if the argument list does not conform to any of the above.

new(default_value = nil) → new_hash
new {|hash, key| ... } → new_hash

Returns a new empty Hash object.

The initial default value and initial default proc for the new hash depend on which form above was used. See Default Values.

If neither an argument nor a block given, initializes both the default value and the default proc to `nil`:

```
h = Hash.new
h.default # => nil
h.default_proc # => nil
```

If argument `default_value` given but no block given, initializes the default value to the given `default_value` and the default proc to `nil`:

```
h = Hash.new(false)
h.default # => false
h.default_proc # => nil
```

If a block given but no argument, stores the block as the default proc and sets the default value to `nil`:

```
h = Hash.new {|hash, key| "Default value for #{key}" }
h.default # => nil
h.default_proc.class # => Proc
h[:nosuch] # => "Default value for nosuch"
```

ruby2_keywords_hash(hash) → hash

Duplicates a given hash and adds a ruby2_keywords flag. This method is not for casual use; debugging, researching, and some truly necessary cases like deserialization of arguments.

```
h = {k: 1}
h = Hash.ruby2_keywords_hash(h)
def foo(k: 42)
  k
end
foo(*[h]) #=> 1 with neither a warning or an error
```

ruby2_keywords_hash?(hash) → true or false

Checks if a given hash is flagged by `Module#ruby2_keywords` (or `Proc#ruby2_keywords`). This method is not for casual use; debugging, researching, and some truly necessary cases like serialization of arguments.

```
ruby2_keywords def foo(*args)
  Hash.ruby2_keywords_hash?(args.last)
end
foo(k: 1)   #=> true
foo({k: 1}) #=> false
```

try_convert(obj) → obj, new_hash, or nil

If `obj` is a Hash object, returns `obj`.

Otherwise if `obj` responds to `:to_hash`, calls `obj.to_hash` and returns the result.

Returns `nil` if `obj` does not respond to `:to_hash`

Raises an exception unless `obj.to_hash` returns a Hash object.

Public Instance Methods

hash < other_hash → true or false

Returns `true` if `hash` is a proper subset of `other_hash`, `false` otherwise:

```
h1 = {foo: 0, bar: 1}
h2 = {foo: 0, bar: 1, baz: 2}
h1 < h2 # => true
h2 < h1 # => false
h1 < h1 # => false
```

hash <= other_hash → true or false

Returns `true` if `hash` is a subset of `other_hash`, `false` otherwise:

```
h1 = {foo: 0, bar: 1}
h2 = {foo: 0, bar: 1, baz: 2}
h1 <= h2 # => true
h2 <= h1 # => false
h1 <= h1 # => true
```

hash == object → true or false

Returns `true` if all of the following are true:

- `object` is a Hash object.
- `hash` and `object` have the same keys (regardless of order).
- For each key `key`, `hash[key] == object[key]`.

Otherwise, returns `false`.

Equal:

```
h1 = {foo: 0, bar: 1, baz: 2}
h2 = {foo: 0, bar: 1, baz: 2}
h1 == h2 # => true
h3 = {baz: 2, bar: 1, foo: 0}
h1 == h3 # => true
```

hash > other_hash → true or false

Returns `true` if `hash` is a proper superset of `other_hash`, `false` otherwise:

```
h1 = {foo: 0, bar: 1, baz: 2}
h2 = {foo: 0, bar: 1}
h1 > h2 # => true
```

```
h2 > h1 # => false
h1 > h1 # => false
```

hash >= other_hash → true or false

Returns `true` if `hash` is a superset of `other_hash`, `false` otherwise:

```
h1 = {foo: 0, bar: 1, baz: 2}
h2 = {foo: 0, bar: 1}
h1 >= h2 # => true
h2 >= h1 # => false
h1 >= h1 # => true
```

hash[key] → value

Returns the value associated with the given `key`, if found:

```
h = {foo: 0, bar: 1, baz: 2}
h[:foo] # => 0
```

If `key` is not found, returns a default value (see Default Values):

```
h = {foo: 0, bar: 1, baz: 2}
h[:nosuch] # => nil
```

hash[key] = value → value

Associates the given `value` with the given `key`; returns `value`.

If the given `key` exists, replaces its value with the given `value`; the ordering is not affected (see Entry Order):

```
h = {foo: 0, bar: 1}
h[:foo] = 2 # => 2
h.store(:bar, 3) # => 3
h # => {:foo=>2, :bar=>3}
```

If `key` does not exist, adds the `key` and `value`; the new entry is last in the order (see Entry Order):

```
h = {foo: 0, bar: 1}
h[:baz] = 2 # => 2
h.store(:bat, 3) # => 3
h # => {:foo=>0, :bar=>1, :baz=>2, :bat=>3}
```

Also aliased as: store

any? → true or false
any?(object) → true or false
any? {|key, value| ... } → true or false

Returns `true` if any element satisfies a given criterion; `false` otherwise.

If `self` has no element, returns `false` and argument or block are not used.

With no argument and no block, returns `true` if `self` is non-empty; `false` if empty.

With argument `object` and no block, returns `true` if for any key `key` `h.assoc(key) == object`:

```
h = {foo: 0, bar: 1, baz: 2}
h.any?([:bar, 1]) # => true
h.any?([:bar, 0]) # => false
h.any?([:baz, 1]) # => false
```

With no argument and a block, calls the block with each key-value pair; returns `true` if the block returns any truthy value, `false` otherwise:

```
h = {foo: 0, bar: 1, baz: 2}
h.any? {|key, value| value < 3 } # => true
h.any? {|key, value| value > 3 } # => false
```

Related: Enumerable#any?

assoc(key) → new_array or nil

If the given `key` is found, returns a 2-element Array containing that key and its value:

```
h = {foo: 0, bar: 1, baz: 2}
h.assoc(:bar) # => [:bar, 1]
```

Returns `nil` if key `key` is not found.

clear → self

Removes all hash entries; returns `self`.

compact → new_hash

Returns a copy of `self` with all `nil`-valued entries removed:

```
h = {foo: 0, bar: nil, baz: 2, bat: nil}
h1 = h.compact
h1 # => {:foo=>0, :baz=>2}
```

compact! → self or nil

Returns `self` with all its `nil`-valued entries removed (in place):

```
h = {foo: 0, bar: nil, baz: 2, bat: nil}
h.compact! # => {:foo=>0, :baz=>2}
```

Returns `nil` if no entries were removed.

compare_by_identity → self

Sets `self` to consider only identity in comparing keys; two keys are considered the same only if they are the same object; returns `self`.

By default, these two object are considered to be the same key, so `s1` will overwrite `s0`:

```
s0 = 'x'
s1 = 'x'
h = {}
h.compare_by_identity? # => false
h[s0] = 0
h[s1] = 1
h # => {"x"=>1}
```

After calling #compare_by_identity, the keys are considered to be different, and therefore do not overwrite each other:

```
h = {}
h.compare_by_identity # => {}
h.compare_by_identity? # => true
h[s0] = 0
h[s1] = 1
h # => {"x"=>0, "x"=>1}
```

compare_by_identity? → true or false

Returns `true` if compare_by_identity has been called, `false` otherwise.

deconstruct_keys(p1)

default → object
default(key) → object

Returns the default value for the given `key`. The returned value will be determined either by the default proc or by the default value. See Default Values.

With no argument, returns the current default value:

```
h = {}
h.default # => nil
```

If `key` is given, returns the default value for `key`, regardless of whether that key exists:

```
h = Hash.new { |hash, key| hash[key] = "No key #{key}"}
h[:foo] = "Hello"
h.default(:foo) # => "No key foo"
```

default = value → object

Sets the default value to `value`; returns `value`:

```
h = {}
h.default # => nil
h.default = false # => false
h.default # => false
```

See Default Values.

default_proc → proc or nil

Returns the default proc for `self` (see Default Values):

```
h = {}
h.default_proc # => nil
h.default_proc = proc {|hash, key| "Default value for #{key}" }
h.default_proc.class # => Proc
```

default_proc = proc → proc

Sets the default proc for `self` to `proc`: (see Default Values):

```
h = {}
h.default_proc # => nil
h.default_proc = proc { |hash, key| "Default value for #{key}" }
h.default_proc.class # => Proc
h.default_proc = nil
h.default_proc # => nil
```

delete(key) → value or nil
delete(key) {|key| ... } → object

Deletes the entry for the given `key` and returns its associated value.

If no block is given and `key` is found, deletes the entry and returns the associated value:

```
h = {foo: 0, bar: 1, baz: 2}
h.delete(:bar) # => 1
h # => {:foo=>0, :baz=>2}
```

If no block given and `key` is not found, returns `nil`.

If a block is given and `key` is found, ignores the block, deletes the entry, and returns the associated value:

```
h = {foo: 0, bar: 1, baz: 2}
h.delete(:baz) { |key| raise 'Will never happen'} # => 2
h # => {:foo=>0, :bar=>1}
```

If a block is given and `key` is not found, calls the block and returns the block's return value:

```
h = {foo: 0, bar: 1, baz: 2}
h.delete(:nosuch) { |key| "Key #{key} not found" } # => "Key nosuch not found"
h # => {:foo=>0, :bar=>1, :baz=>2}
```

delete_if {|key, value| ... } → self
delete_if → new_enumerator

If a block given, calls the block with each key-value pair; deletes each entry for which the block returns a truthy value; returns `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.delete_if {|key, value| value > 0 } # => {:foo=>0}
```

If no block given, returns a new Enumerator:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.delete_if # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:delete_if>
e.each { |key, value| value > 0 } # => {:foo=>0}
```

dig(key, *identifiers) → object

Finds and returns the object in nested objects that is specified by `key` and `identifiers`. The nested objects may be instances of various classes. See Dig Methods.

Nested Hashes:

```
h = {foo: {bar: {baz: 2}}}
h.dig(:foo) # => {:bar=>{:baz=>2}}
h.dig(:foo, :bar) # => {:baz=>2}
h.dig(:foo, :bar, :baz) # => 2
h.dig(:foo, :bar, :BAZ) # => nil
```

Nested Hashes and Arrays:

```
h = {foo: {bar: [:a, :b, :c]}}
h.dig(:foo, :bar, 2) # => :c
```

This method will use the default values for keys that are not present:

```
h = {foo: {bar: [:a, :b, :c]}}
h.dig(:hello) # => nil
h.default_proc = -> (hash, _key) { hash }
h.dig(:hello, :world) # => h
h.dig(:hello, :world, :foo, :bar, 2) # => :c
```

each {|key, value| ... } → self
each → new_enumerator

Calls the given block with each key-value pair; returns `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.each_pair {|key, value| puts "#{key}: #{value}"} # => {:foo=>0, :bar=>1, :ba
```

Output:

```
foo: 0
bar: 1
baz: 2
```

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.each_pair # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:each_pair>
h1 = e.each {|key, value| puts "#{key}: #{value}"}
h1 # => {:foo=>0, :bar=>1, :baz=>2}
```

Output:

```
foo: 0
bar: 1
baz: 2
```

Alias for: each_pair

each_key {|key| ... } → self
each_key → new_enumerator

Calls the given block with each key; returns `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.each_key {|key| puts key }  # => {:foo=>0, :bar=>1, :baz=>2}
```

Output:

```
foo
bar
baz
```

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.each_key # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:each_key>
h1 = e.each {|key| puts key }
h1 # => {:foo=>0, :bar=>1, :baz=>2}
```

Output:

```
foo
bar
baz
```

each_pair -> new_enumerator

Calls the given block with each key-value pair; returns `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.each_pair {|key, value| puts "#{key}: #{value}"} # => {:foo=>0, :bar=>1, :ba
```

Output:

```
foo: 0
bar: 1
baz: 2
```

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.each_pair # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:each_pair>
h1 = e.each {|key, value| puts "#{key}: #{value}"}
h1 # => {:foo=>0, :bar=>1, :baz=>2}
```

Output:

```
foo: 0
bar: 1
baz: 2
```

Also aliased as: each

`each_value {|value| ... } → self`
`each_value → new_enumerator`

Calls the given block with each value; returns `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.each_value {|value| puts value } # => {:foo=>0, :bar=>1, :baz=>2}
```

Output:

```
0
1
2
```

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.each_value # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:each_value>
h1 = e.each {|value| puts value }
h1 # => {:foo=>0, :bar=>1, :baz=>2}
```

Output:

```
0
1
2
```

empty? → true or false

Returns `true` if there are no hash entries, `false` otherwise:

```
{}.empty? # => true
{foo: 0, bar: 1, baz: 2}.empty? # => false
```

eql? object → true or false

Returns `true` if all of the following are true:

- `object` is a Hash object.
- `hash` and `object` have the same keys (regardless of order).
- For each key `key`, `h[key] eql? object[key]`.

Otherwise, returns `false`.

Equal:

```
h1 = {foo: 0, bar: 1, baz: 2}
h2 = {foo: 0, bar: 1, baz: 2}
h1.eql? h2 # => true
h3 = {baz: 2, bar: 1, foo: 0}
h1.eql? h3 # => true
```

except(*keys) → a_hash

Returns a new Hash excluding entries for the given `keys`:

```
h = { a: 100, b: 200, c: 300 }
h.except(:a)          #=> {:b=>200, :c=>300}
```

Any given `keys` that are not found are ignored.

fetch(key) → object
fetch(key, default_value) → object
fetch(key) {|key| ... } → object

Returns the value for the given `key`, if found.

```
h = {foo: 0, bar: 1, baz: 2}
h.fetch(:bar) # => 1
```

If `key` is not found and no block was given, returns `default_value`:

```
{}.fetch(:nosuch, :default) # => :default
```

If `key` is not found and a block was given, yields `key` to the block and returns the block's return value:

```
{}.fetch(:nosuch) {|key| "No key #{key}"} # => "No key nosuch"
```

Raises KeyError if neither `default_value` nor a block was given.

Note that this method does not use the values of either default or default_proc.

fetch_values(*keys) → new_array
fetch_values(*keys) {|key| ... } → new_array

Returns a new Array containing the values associated with the given keys *keys:

```
h = {foo: 0, bar: 1, baz: 2}
h.fetch_values(:baz, :foo) # => [2, 0]
```

Returns a new empty Array if no arguments given.

When a block is given, calls the block with each missing key, treating the block's return value as the value for that key:

```
h = {foo: 0, bar: 1, baz: 2}
values = h.fetch_values(:bar, :foo, :bad, :bam) {|key| key.to_s}
values # => [1, 0, "bad", "bam"]
```

When no block is given, raises an exception if any given key is not found.

filter()

Returns a new Hash object whose entries are those for which the block returns a truthy value:

```
h = {foo: 0, bar: 1, baz: 2}
h.select {|key, value| value < 2 } # => {:foo=>0, :bar=>1}
```

Returns a new Enumerator if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.select # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:select>
e.each {|key, value| value < 2 } # => {:foo=>0, :bar=>1}
```

Alias for: select

filter!()

Returns `self`, whose entries are those for which the block returns a truthy value:

```
h = {foo: 0, bar: 1, baz: 2}
h.select! {|key, value| value < 2 }  => {:foo=>0, :bar=>1}
```

Returns `nil` if no entries were removed.

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.select!  # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:select!>
e.each { |key, value| value < 2 } # => {:foo=>0, :bar=>1}
```

Alias for: select!

flatten → new_array
flatten(level) → new_array

Returns a new `Array` object that is a 1-dimensional flattening of `self`.

By default, nested Arrays are not flattened:

```
h = {foo: 0, bar: [:bat, 3], baz: 2}
h.flatten # => [:foo, 0, :bar, [:bat, 3], :baz, 2]
```

Takes the depth of recursive flattening from `Integer` argument `level`:

```
h = {foo: 0, bar: [:bat, [:baz, [:bat, ]]]}
h.flatten(1) # => [:foo, 0, :bar, [:bat, [:baz, [:bat]]]]
h.flatten(2) # => [:foo, 0, :bar, :bat, [:baz, [:bat]]]
h.flatten(3) # => [:foo, 0, :bar, :bat, :baz, [:bat]]
h.flatten(4) # => [:foo, 0, :bar, :bat, :baz, :bat]
```

When `level` is negative, flattens all nested Arrays:

```
h = {foo: 0, bar: [:bat, [:baz, [:bat, ]]]}
h.flatten(-1) # => [:foo, 0, :bar, :bat, :baz, :bat]
h.flatten(-2) # => [:foo, 0, :bar, :bat, :baz, :bat]
```

When `level` is zero, returns the equivalent of `to_a` :

```
h = {foo: 0, bar: [:bat, 3], baz: 2}
h.flatten(0) # => [[:foo, 0], [:bar, [:bat, 3]], [:baz, 2]]
h.flatten(0) == h.to_a # => true
```

has_key?(key) → true or false

Returns `true` if `key` is a key in `self`, otherwise `false`.

Alias for: include?

has_value?(value) → true or false

Returns `true` if `value` is a value in `self`, otherwise `false`.

Also aliased as: value?

hash → an_integer

Returns the `Integer` hash-code for the hash.

Two Hash objects have the same hash-code if their content is the same (regardless of order):

```
h1 = {foo: 0, bar: 1, baz: 2}
h2 = {baz: 2, bar: 1, foo: 0}
h2.hash == h1.hash # => true
h2.eql? h1 # => true
```

include?(key) → true or false

Returns `true` if `key` is a key in `self`, otherwise `false`.

Also aliased as: member?, has_key?, key?

initialize_copy(other_hash) -> self

Replaces the entire contents of `self` with the contents of `other_hash`; returns `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.replace({bat: 3, bam: 4}) # => {:bat=>3, :bam=>4}
```

Also aliased as: replace

inspect → new_string

Returns a new `String` containing the hash entries:

```
h = {foo: 0, bar: 1, baz: 2}
h.inspect # => "{:foo=>0, :bar=>1, :baz=>2}"
```

Also aliased as: to_s

invert → new_hash

Returns a new Hash object with the each key-value pair inverted:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = h.invert
h1 # => {0=>:foo, 1=>:bar, 2=>:baz}
```

Overwrites any repeated new keys: (see Entry Order):

```
h = {foo: 0, bar: 0, baz: 0}
h.invert # => {0=>:baz}
```

keep_if {|key, value| ... } → self
keep_if → new_enumerator

Calls the block for each key-value pair; retains the entry if the block returns a truthy value; otherwise deletes the entry; returns `self`.

```
h = {foo: 0, bar: 1, baz: 2}
h.keep_if { |key, value| key.start_with?('b') } # => {:bar=>1, :baz=>2}
```

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.keep_if # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:keep_if>
e.each { |key, value| key.start_with?('b') } # => {:bar=>1, :baz=>2}
```

key(value) → key or nil

Returns the key for the first-found entry with the given `value` (see Entry Order):

```
h = {foo: 0, bar: 2, baz: 2}
h.key(0) # => :foo
h.key(2) # => :bar
```

Returns `nil` if no such value is found.

key?(key) → true or false

Returns `true` if `key` is a key in `self`, otherwise `false`.

Alias for: include?

keys → new_array

Returns a new `Array` containing all keys in `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.keys # => [:foo, :bar, :baz]
```

length → integer

Returns the count of entries in `self`:

```
{foo: 0, bar: 1, baz: 2}.length # => 3
```

Alias for: size

member?(key) → true or false

Returns `true` if `key` is a key in `self`, otherwise `false`.

Alias for: include?

merge → copy_of_self
merge(*other_hashes) → new_hash
merge(*other_hashes) { |key, old_value, new_value| ... } → new_hash

Returns the new Hash formed by merging each of `other_hashes` into a copy of `self`.

Each argument in `other_hashes` must be a Hash.

With arguments and no block:

- Returns the new Hash object formed by merging each successive Hash in `other_hashes` into `self`.
- Each new-key entry is added at the end.
- Each duplicate-key entry's value overwrites the previous value.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = {bat: 3, bar: 4}
h2 = {bam: 5, bat:6}
h.merge(h1, h2) # => {:foo=>0, :bar=>4, :baz=>2, :bat=>6, :bam=>5}
```

With arguments and a block:

- Returns a new Hash object that is the merge of `self` and each given hash.
- The given hashes are merged left to right.
- Each new-key entry is added at the end.
- For each duplicate key:
  - Calls the block with the key and the old and new values.
  - The block's return value becomes the new value for the entry.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = {bat: 3, bar: 4}
h2 = {bam: 5, bat:6}
h3 = h.merge(h1, h2) { |key, old_value, new_value| old_value + new_value }
h3 # => {:foo=>0, :bar=>5, :baz=>2, :bat=>9, :bam=>5}
```

With no arguments:

- Returns a copy of `self`.
- The block, if given, is ignored.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h.merge # => {:foo=>0, :bar=>1, :baz=>2}
h1 = h.merge { |key, old_value, new_value| raise 'Cannot happen' }
h1 # => {:foo=>0, :bar=>1, :baz=>2}
```

merge! → self
merge!(*other_hashes) → self

`merge!(*other_hashes) { |key, old_value, new_value| ... } → self`

Merges each of `other_hashes` into `self`; returns `self`.

Each argument in `other_hashes` must be a Hash.

With arguments and no block:

- Returns `self`, after the given hashes are merged into it.
- The given hashes are merged left to right.
- Each new entry is added at the end.
- Each duplicate-key entry's value overwrites the previous value.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = {bat: 3, bar: 4}
h2 = {bam: 5, bat:6}
h.merge!(h1, h2) # => {:foo=>0, :bar=>4, :baz=>2, :bat=>6, :bam=>5}
```

With arguments and a block:

- Returns `self`, after the given hashes are merged.
- The given hashes are merged left to right.
- Each new-key entry is added at the end.
- For each duplicate key:
  - Calls the block with the key and the old and new values.
  - The block's return value becomes the new value for the entry.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = {bat: 3, bar: 4}
h2 = {bam: 5, bat:6}
h3 = h.merge!(h1, h2) { |key, old_value, new_value| old_value + new_value }
h3 # => {:foo=>0, :bar=>5, :baz=>2, :bat=>9, :bam=>5}
```

With no arguments:

- Returns `self`, unmodified.
- The block, if given, is ignored.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h.merge # => {:foo=>0, :bar=>1, :baz=>2}
```

```
h1 = h.merge! { |key, old_value, new_value| raise 'Cannot happen' }
h1 # => {:foo=>0, :bar=>1, :baz=>2}
```

Alias for: update

rassoc(value) → new_array or nil

Returns a new 2-element Array consisting of the key and value of the first-found entry whose value is == to value (see Entry Order):

```
h = {foo: 0, bar: 1, baz: 1}
h.rassoc(1) # => [:bar, 1]
```

Returns nil if no such value found.

rehash → self

Rebuilds the hash table by recomputing the hash index for each key; returns self.

The hash table becomes invalid if the hash value of a key has changed after the entry was created. See Modifying an Active Hash Key.

reject {|key, value| ... } → new_hash
reject → new_enumerator

Returns a new Hash object whose entries are all those from self for which the block returns false or nil:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = h.reject {|key, value| key.start_with?('b') }
h1 # => {:foo=>0}
```

Returns a new Enumerator if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.reject # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:reject>
h1 = e.each {|key, value| key.start_with?('b') }
h1 # => {:foo=>0}
```

reject! {|key, value| ... } → self or nil
reject! → new_enumerator

Returns self, whose remaining entries are those for which the block returns false or nil:

```
h = {foo: 0, bar: 1, baz: 2}
h.reject! {|key, value| value < 2 } # => {:baz=>2}
```

Returns `nil` if no entries are removed.

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.reject! # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:reject!>
e.each {|key, value| key.start_with?('b') } # => {:foo=>0}
```

replace(other_hash) → self

Replaces the entire contents of `self` with the contents of `other_hash`; returns `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.replace({bat: 3, bam: 4}) # => {:bat=>3, :bam=>4}
```

Alias for: initialize_copy

select {|key, value| ... } → new_hash
select → new_enumerator

Returns a new Hash object whose entries are those for which the block returns a truthy value:

```
h = {foo: 0, bar: 1, baz: 2}
h.select {|key, value| value < 2 } # => {:foo=>0, :bar=>1}
```

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.select # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:select>
e.each {|key, value| value < 2 } # => {:foo=>0, :bar=>1}
```

Also aliased as: filter

select! {|key, value| ... } → self or nil
select! → new_enumerator

Returns `self`, whose entries are those for which the block returns a truthy value:

```
h = {foo: 0, bar: 1, baz: 2}
h.select! {|key, value| value < 2 }  => {:foo=>0, :bar=>1}
```

Returns `nil` if no entries were removed.

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.select!  # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:select!>
e.each { |key, value| value < 2 } # => {:foo=>0, :bar=>1}
```

Also aliased as: filter!

shift → [key, value] or nil

Removes the first hash entry (see Entry Order); returns a 2-element `Array` containing the removed key and value:

```
h = {foo: 0, bar: 1, baz: 2}
h.shift # => [:foo, 0]
h # => {:bar=>1, :baz=>2}
```

Returns nil if the hash is empty.

size → integer

Returns the count of entries in `self`:

```
{foo: 0, bar: 1, baz: 2}.length # => 3
```

Also aliased as: length

slice(*keys) → new_hash

Returns a new Hash object containing the entries for the given `keys`:

```
h = {foo: 0, bar: 1, baz: 2}
h.slice(:baz, :foo) # => {:baz=>2, :foo=>0}
```

Any given `keys` that are not found are ignored.

store(key, value)

Associates the given `value` with the given `key`; returns `value`.

If the given `key` exists, replaces its value with the given `value`; the ordering is not affected (see Entry Order):

```
h = {foo: 0, bar: 1}
h[:foo] = 2 # => 2
h.store(:bar, 3) # => 3
h # => {:foo=>2, :bar=>3}
```

If `key` does not exist, adds the `key` and `value`; the new entry is last in the order (see Entry Order):

```
h = {foo: 0, bar: 1}
h[:baz] = 2 # => 2
h.store(:bat, 3) # => 3
h # => {:foo=>0, :bar=>1, :baz=>2, :bat=>3}
```

Alias for: []=

to_a → new_array

Returns a new `Array` of 2-element `Array` objects; each nested `Array` contains a key-value pair from `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.to_a # => [[:foo, 0], [:bar, 1], [:baz, 2]]
```

to_h → self or new_hash
to_h {|key, value| ... } → new_hash

For an instance of Hash, returns `self`.

For a subclass of Hash, returns a new Hash containing the content of `self`.

When a block is given, returns a new Hash object whose content is based on the block; the block should return a 2-element `Array` object specifying the key-value pair to be included in the returned Array:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = h.to_h {|key, value| [value, key] }
h1 # => {0=>:foo, 1=>:bar, 2=>:baz}
```

to_hash → self

Returns `self`.

to_proc → proc

Returns a `Proc` object that maps a key to its value:

```
h = {foo: 0, bar: 1, baz: 2}
proc = h.to_proc
proc.class # => Proc
proc.call(:foo) # => 0
proc.call(:bar) # => 1
proc.call(:nosuch) # => nil
```

to_s()

Returns a new `String` containing the hash entries:

```
h = {foo: 0, bar: 1, baz: 2}
h.inspect # => "{:foo=>0, :bar=>1, :baz=>2}"
```

Alias for: inspect

transform_keys {|key| ... } → new_hash
transform_keys(hash2) → new_hash
transform_keys(hash2) {|other_key| ...} → new_hash
transform_keys → new_enumerator

Returns a new Hash object; each entry has:

- A key provided by the block.
- The value from `self`.

An optional hash argument can be provided to map keys to new keys. Any key not given will be mapped using the provided block, or remain the same if no block is given.

Transform keys:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = h.transform_keys {|key| key.to_s }
h1 # => {"foo"=>0, "bar"=>1, "baz"=>2}

h.transform_keys(foo: :bar, bar: :foo)
#=> {bar: 0, foo: 1, baz: 2}

h.transform_keys(foo: :hello, &:to_s)
#=> {:hello=>0, "bar"=>1, "baz"=>2}
```

Overwrites values for duplicate keys:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = h.transform_keys {|key| :bat }
h1 # => {:bat=>2}
```

Returns a new Enumerator if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.transform_keys # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:transform
h1 = e.each { |key| key.to_s }
h1 # => {"foo"=>0, "bar"=>1, "baz"=>2}
```

transform_keys! {|key| ... } → self
transform_keys!(hash2) → self
transform_keys!(hash2) {|other_key| ...} → self
transform_keys! → new_enumerator

Same as Hash#transform_keys but modifies the receiver in place instead of returning a new hash.

transform_values {|value| ... } → new_hash
transform_values → new_enumerator

Returns a new Hash object; each entry has:

- A key from `self`.
- A value provided by the block.

Transform values:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = h.transform_values {|value| value * 100}
h1 # => {:foo=>0, :bar=>100, :baz=>200}
```

Returns a new Enumerator if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.transform_values # => #<Enumerator: {:foo=>0, :bar=>1, :baz=>2}:transfo
h1 = e.each { |value| value * 100}
h1 # => {:foo=>0, :bar=>100, :baz=>200}
```

transform_values! {|value| ... } → self
transform_values! → new_enumerator

Returns `self`, whose keys are unchanged, and whose values are determined by the given block.

```
h = {foo: 0, bar: 1, baz: 2}
h.transform_values! {|value| value * 100} # => {:foo=>0, :bar=>100, :baz=>200
```

Returns a new `Enumerator` if no block given:

```
h = {foo: 0, bar: 1, baz: 2}
e = h.transform_values! # => #<Enumerator: {:foo=>0, :bar=>100, :baz=>200}:tr
h1 = e.each {|value| value * 100}
h1 # => {:foo=>0, :bar=>100, :baz=>200}
```

update(*other_hashes) { |key, old_value, new_value| } -> self

Merges each of `other_hashes` into `self`; returns `self`.

Each argument in `other_hashes` must be a Hash.

With arguments and no block:

- Returns `self`, after the given hashes are merged into it.
- The given hashes are merged left to right.
- Each new entry is added at the end.
- Each duplicate-key entry's value overwrites the previous value.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = {bat: 3, bar: 4}
h2 = {bam: 5, bat:6}
h.merge!(h1, h2) # => {:foo=>0, :bar=>4, :baz=>2, :bat=>6, :bam=>5}
```

With arguments and a block:

- Returns `self`, after the given hashes are merged.
- The given hashes are merged left to right.
- Each new-key entry is added at the end.
- For each duplicate key:
  - Calls the block with the key and the old and new values.
  - The block's return value becomes the new value for the entry.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h1 = {bat: 3, bar: 4}
h2 = {bam: 5, bat:6}
h3 = h.merge!(h1, h2) { |key, old_value, new_value| old_value + new_value }
h3 # => {:foo=>0, :bar=>5, :baz=>2, :bat=>9, :bam=>5}
```

With no arguments:

- Returns `self`, unmodified.
- The block, if given, is ignored.

Example:

```
h = {foo: 0, bar: 1, baz: 2}
h.merge # => {:foo=>0, :bar=>1, :baz=>2}
h1 = h.merge! { |key, old_value, new_value| raise 'Cannot happen' }
h1 # => {:foo=>0, :bar=>1, :baz=>2}
```

Also aliased as: merge!

value?(value) → true or false

Returns `true` if `value` is a value in `self`, otherwise `false`.

Alias for: has_value?

values → new_array

Returns a new `Array` containing all values in `self`:

```
h = {foo: 0, bar: 1, baz: 2}
h.values # => [0, 1, 2]
```

values_at(*keys) → new_array

Returns a new `Array` containing values for the given `keys`:

```
h = {foo: 0, bar: 1, baz: 2}
h.values_at(:baz, :foo) # => [2, 0]
```

The default values are returned for any keys that are not found:

```
h.values_at(:hello, :foo) # => [nil, 0]
```

Validate

Generated by RDoc 6.4.0.

Based on Darkfish by Michael Granger.

Ruby-doc.org is provided by James Britt and Neurogami.

Maximum R+D.