Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

5. Hashes

Hashes and arrays are the two basic "aggregate" data types supported by most modern programming lagnguages. The basic interface of a hash is similar to that of an array. The difference is that while an array stores items according to a numeric index, the index of a hash can be any object at all.

Arrays and strings have been built into programming languages for decades, but built-in hashes are a relatively recent development. Now that they're around, it's hard to live without them: they're at least as useful as arrays.

You can create a Hash by calling `Hash.new` or by using one of the special sytaxes `Hash[]` or `{}`. With the `Hash[]` syntax, you pass in the initial elements as comma-separated object references. With the `{}` syntax, you pass in the initial contents as comma-separated key-value pairs.

```
empty = Hash.new                            # => {}
empty ={}                                   # => {}
numbers = { 'two' => 2, 'eight' => 8} # => {"two"=>2, "eight"=>8}
numbers = Hash['two', 2, 'eight', 8]        # => {"two"=>2, "eight"=>8}
```

Once the hash is created, you can do hash lookups and element assignments using the same syntax you would use to view and modify array elements:

```
numbers["two"]                              # => 2
numbers["ten"] = 10                         # => 10
numbers                                     # => {"two"=>2, "eight"=>8, "ten"=>10}
```

You can get an array containing the keys or values of a hash with `Hash#keys` or `Hash#values`. You can get the entire hash as an array with `Hash#to_a`:

```
numbers.keys                                # => ["two", "eight", "ten"]
numbers.values                              # => [2, 8, 10]
numbers.to_a                                 # => [["two", 2], ["eight", 8], ["ten", 10]]
```

Like an array, a hash contains references to objects, not copies of them. Modifications to the original objects will affect all references to them:

```
motto = "Don't tread on me"
flag = { :motto => motto,
         :picture => "rattlesnake.png"}
motto.upcase!
flag[:motto]                              # => "DON'T TREAD ON ME"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The defining feature of an array is its ordering. Each element of an array is assigned a `Fixnum` object as its key. The keys start from zero and there can never be gaps. In contrast, a hash has no natural ordering, since its keys can be any objects at all. This feature make hashes useful for storing lightly structured data or key-value pairs.

Consider some simple data for a person in an address book. For a side-by-side comparison I'll represent identical data as an array, then as a hash:

```
a = ["Maury", "Momento", "123 Elm St.", "West Covina", "CA"]
h = { :first_name => "Maury",
      :last_name => "Momento",
      :address => "123 Elm St."
      :city => "West Covina",
      :state => "CA" }
```

The array version is more concise, and if you know the numeric index, you can retrieve any element from it in constant time. The problem is knowing the index, and knowing what it means. Other than inspecting the records, there's no way to know whether the element at index `1` is a last name or a first name. Worse, if the array format changes to add an apartment number between the street address and city, all code that uses `a[3]` or `a[4]` will need to have its index changed.

The hash version doesn't have these problems. The last name will always be at `:last_name`, and it's easy (for a human, anyway) to know what `:last_name` means. Most of the time, hash lookups take no longer than array lookups.

The main advantage of a hash is that it's often easier to find what you're looking for. Checking whether an array contains a certain value might require scanning the entire array. To see whether a hash contains a value for a certain key, you only need to look up that key. The `set` library (as seen in the previous chapter) exploits this behavior to implement a class that looks like an array, but has the performance characteristics of a hash.

The downside of using a hash is that since it has no natural ordering, it can't be sorted except by turning it into an array first. There's also no guarantee of order when you iterate over a hash. Here's a contrasting case, in which an array is obviously the right choice:

```
a = [1, 4, 9, 16]
h = { :one_squared => 1, two_squared => 4, three_squared => 9,
      :four_squared => 16 }
```

In this case, there's a numeric order to the entries, and giving them additional labels distracts more than it helps.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

A hash in Ruby is actually implemented as an array. When you look up a key in a hash (either to see what's associated with that key, or to associate a value with the key), Ruby calculates the *hash code* of the key by calling its `hash` method. The result is used as a numeric index in the array. Recipe 5.5 will help you with the most common problem related to hash codes.

The performance of a hash depends a lot on the fact that it's very rare for two objects to have the same hash code. If all objects in a hash had the same hash code, a hash would be much slower than an array. Code like this would be a very bad idea:

```
class BadIdea
  def hash
    100
  end
end
```

Except for strings and other built-in objects, most objects have a hash code equivalent to their internal object ID. As seen above, you can override `Object#hash` to change this, but the only time you should need to do this is if your class also overrides `Object#==`. If two objects are considered equal, they should also have the same hash code; otherwise, they will behave strangely when you put them into hashes. Code like the fragment below is a very good idea:

```
class StringHolder
  attr_reader :string
  def initialize(s)
    @string = s
  end

  def ==(other)
    @string == other.string
  end

  def hash
    @string.hash
  end
end
a = StringHolder.new("The same string.")
b = StringHolder.new("The same string.")
a == b                                                  # => true
a.hash                                                  # => -1007666862
b.hash                                                  # => -1007666862
```

Recipe 5.1. Using Symbols as Hash Keys

Credit: Ben Giddings

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

When using a hash, you want the slight optimization you can get by using symbols as keys instead of strings.

Solution

Whenever you would otherwise use a quoted string, use a symbol instead. A symbol can be created by either using a colon in front of a word, like `:keyname`, or by transforming a string to a symbol using `String#intern`.

```
people = Hash.new
people[:nickname] = 'Matz'
people[:language] = 'Japanese'
people['last name'.intern] = 'Matsumoto'
people[:nickname]                                  # => "Matz"
people['nickname'.intern]                          # => "Matz"
```

Discussion

While '`name`' and '`name`' appear exactly identical, they're actually different. Each time you create a quoted string in Ruby, you create a unique object. You can see this by looking at the `object_id` method.

```
'name'.object_id                                   # => -605973716
'name'.object_id                                   # => -605976356
'name'.object_id                                   # => -605978996
```

By comparison, each instance of a symbol refers to a single object.

```
:name.object_id                                    # => 878862
:name.object_id                                    # => 878862
'name'.intern.object_id                            # => 878862
'name'.intern.object_id                            # => 878862
```

Using symbols instead of strings saves memory and time. It saves memory because there's only one symbol instance, instead of many string instances. If you have many hashes that contain the same keys, the memory savings adds up.

Using symbols as hash keys is faster because the hash value of a symbol is simply its object ID. If you use strings in a hash, Ruby must calculate the hash value of a string each time it's used as a hash key.

See Also

- Recipe 1.7, "Converting Between Strings and Symbols"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 5.2. Creating a Hash with a Default Value

Credit: Ben Giddings

Problem

You're using a hash, and you don't want to get `nil` as a value when you look up a key that isn't present in the hash. You want to get some more convenient value instead, possibly one calculated dynamically.

Solution

A normal hash has a default value of `nil`:

```
h = Hash.new
h[1]                                        # => nil
h['do you have this string?']               # => nil
```

There are two ways of creating default values for hashes. If you want the default value to be the same object for every hash key, pass that value into the `Hash` constructor.

```
h = Hash.new("nope")
h[1]                                      # => "nope"
h['do you have this string?']             # => "nope"
```

If you want the default value for a missing key to depend on the key or the current state of the hash, pass a code block into the hash constructor. The block will be called each time someone requests a missing key.

```
h = Hash.new { |hash, key| (key.respond_to? :to_str) ? "nope" : nil }
h[1]                                        # => nil
h['do you have this string']                # => "nope"
```

Discussion

The first type of custom default value is most useful when you want a default value of zero. For example, this form can be used to calculate the frequency of certain words in a paragraph of text:

```
text = 'The rain in Spain falls mainly in the plain.'
word_count_hash = Hash.new 0                    # => {}
text.split(/\W+/).each { |word| word_count_hash[word.downcase] += 1 }
word_count_hash
# => {"rain"=>1, "plain"=>1, "in"=>2, "mainly"=>1, "falls"=>1,
#         "the"=>2, "spain"=>1}
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

What if you wanted to make lists of the words starting with a given character? Your first attempt might look like this:

```
first_letter_hash = Hash.new []
text.split(/\W+/).each { |word| first_letter_hash[word[0,1].downcase] << word }
first_letter_hash
                                          # => {}
first_letter_hash["m"]
# => ["The", "rain", "in", "Spain", "falls", "mainly", "in", "the", "plain"]
```

What's going on here? All those words don't start with "m"....

What happened is that the array you passed into the `Hash` constructor is being used for *every* default value. `first_letter_hash["m"]` is now a reference to that array, as is `first_letter_hash["f"]` and even `first_letter_hash[1006]`.

This is a case where you need to pass in a block to the `Hash` constructor. The block is run every time the `Hash` can't find a key. This way you can create a different array each time.

```
first_letter_hash = Hash.new { |hash, key| hash[key] = [] }
text.split(/\W+/).each { |word| first_letter_hash[word[0,1].downcase] << word }
first_letter_hash
# => {"m"=>["mainly"], "p"=>["plain"], "f"=>["falls"], "r"=>["rain"],
#       "s"=>["Spain"], "i"=>["in", "in"], "t"=>["The", "the"]}
first_letter_hash["m"]
# => ["mainly"]
```

When a letter can't be found in the hash, Ruby calls the block passed into the `Hash` constructor. That block puts a new array into the hash, using the missing letter as the key. Now the letter is bound to a unique array, and words can be added to that array normally.

Note that if you want to add the array to the hash so it can be used later, you must assign it within the block of the `Hash` constructor. Otherwise you'll get a new, empty array every time you access `first_letter_hash["m"]`. The words you want to append to the array will be lost.

See Also

- This technique is used in recipes like Recipe 5.6, "Keeping Multiple Values for the Same Hash Key," and Recipe 5.12, "Building a Histogram"

Recipe 5.3. Adding Elements to a Hash

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You have some items, loose or in some other data structure, which you want to put into an existing hash.

Solution

To add a single key-value pair, assign the value to the element lookup expression for the key: that is, call `hash[key]=value`. Assignment will override any previous value for that key.

```
h = {}
h["Greensleeves"] = "all my joy"
h                                  # => {"Greensleeves"=>"all my joy"}
h["Greensleeves"] = "my delight"
h                                  # => {"Greensleeves"=>"my delight"}
```

Discussion

When you use a string as a hash key, the string is transparently copied and the copy is frozen. This is to avoid confusion should you modify the string in place, then try to use its original form to do a hash lookup:

```
key = "Modify me if you can"
h = { key => 1 }
key.upcase!                        # => "MODIFY ME IF YOU CAN"
h[key]                             # => nil
h["Modify me if you can"]          # => 1

h.keys                             # => ["Modify me if you can"]
h.keys[0].upcase!
# TypeError: can't modify frozen string
```

To add an array of key-value pairs to a hash, either iterate over the array with `Array#each`, or pass the hash into `Array#inject`. Using `inject` is slower but the code is more concise.

```
squares = [[1,1], [2,4], [3,9]]

results = {}
squares.each { |k,v| results[k] = v }
results                            # => {1=>1, 2=>4, 3=>9}

squares.inject({}) { |h, kv| h[kv[0]] = kv[1]; h }
# => {1=>1, 2=>4, 3=>9}
```

To turn a flat array into the key-value pairs of a hash, iterate over the array elements two at a time:

```
class Array
  def into_hash(h)
    unless size % 2 == 0
      raise StandardError, "Expected array with even number of elements"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    end
    0.step(size-1, 2) { |x| h[self[x]] = self[x+1] }
    h
  end
end

squares = [1,1,2,3,4,9]
results = {}
squares.into_hash(results)            # => {1=>1, 2=>3, 4=>9}

[1,1,2].into_hash(results)
# StandardError: Expected array with even number of elements
```

To insert into a hash every key-value from another hash, use `Hash#merge!`. If a key is present in both hashes when `a.merge!(b)` is called, the value in `b` takes precedence over the value in `a`.

```
squares = { 1 => 1, 2 => 4, 3 => 9}
cubes = { 3 => 27, 4 => 256, 5 => 3125}
squares.merge!(cubes)
squares                                    # =>{5=>3125, 1=>1, 2=>4, 3=>27, 4=>256}
cubes                                      # =>{5=>3125, 3=>27, 4=>256}
```

`Hash#merge!` also has a nondestructive version, `Hash#merge`, which creates a new `Hash` with elements from both parent hashes. Again, the hash passed in as an argument takes precedence.

To completely replace the entire contents of one hash with the contents of another, use `Hash#replace`.

```
squares = { 1 => 1, 2 => 4, 3 => 9}
cubes = { 1 => 1, 2 => 8, 3 => 27}
squares.replace(cubes)
squares                                        # => {1=>1, 2=>8, 3=>27}
```

This is different from simply assigning the `cubes` hash to the `squares` variable name, because `cubes` and `squares` are still separate hashes: they just happen to contain the same elements right now. Changing `cubes` won't affect `squares`:

```
cubes[4] = 64
squares                                        # => {1=>1, 2=>8, 3=>27}
```

`Hash#replace` is useful for reverting a `Hash` to known default values.

```
defaults = {:verbose => true, :help_level => :beginner }
args = {}
requests.each do |request|
  args.replace(defaults)
  request.process(args) #The process method might modify the args Hash.
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 4.12, "Building Up a Hash Using Injection," has more about the `inject` method
- Recipe 5.1, "Using Symbols as Hash Keys," for a way to save memory when constructing certain types of hashes
- Recipe 5.5, "Using an Array or Other Modifiable Object as a Hash Key," talks about how to avoid another common case of confusion when a hash key is modified

Recipe 5.4. Removing Elements from a Hash

Problem

Certain elements of your hash have got to go!

Solution

Most of the time you want to remove a specific element of a hash. To do that, pass the key into `Hash#delete`.

```
h = {}
h[1] = 10
h                                       # => {1=>10}
h.delete(1)
h                                       # => {}
```

Discussion

Don't try to delete an element from a hash by mapping it to `nil`. It's true that, by default, you get `nil` when you look up a key that's not in the hash, but there's a difference between a key that's missing from the hash and a key that's present but mapped to `nil`. `Hash#has_key?` will see a key mapped to `nil`, as will `Hash#each` and all other methods except for a simple fetch:

```
h = {}
h[5]                                    # => nil
h[5] = 10
h[5]                                    # => 10
h[5] = nil
h[5]                                    # => nil
h.keys                                  # => [5]
h.delete(5)
h.keys                                  # => []
```

`Hash#delete` works well when you need to remove elements on an ad hoc basis, but sometimes you need to go through the whole hash looking for things to remove. Use the

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`Hash#delete_if` iterator to delete key-value pairs for which a certain code block returns true (`Hash#reject` works the same way, but it works on a copy of the `Hash`). The following code deletes all key-value pairs with a certain value:

```
class Hash
  def delete_value(value)
    delete_if { |k,v| v == value }
  end
end

h = {'apple' => 'green', 'potato' => 'red', 'sun' => 'yellow',
     'katydid' => 'green' }
h.delete_value('green')
h                                   # => {"sun"=>"yellow", "potato"=>"red"}
```

This code implements the opposite of `Hash#merge`; it extracts one hash from another:

```
class Hash
  def remove_hash(other_hash)
    delete_if { |k,v| other_hash[k] == v }
  end
end

squares = { 1 => 1, 2 => 4, 3 => 9 }
doubles = { 1 => 2, 2 => 4, 3 => 6 }
squares.remove_hash(doubles)
squares                                 # => {1=>1, 3=>9}
```

Finally, to wipe out the entire contents of a `Hash`, use `Hash#clear`:

```
h = {}
1.upto(1000) { |x| h[x] = x }
h.keys.size                             # => 1000
h.clear
h                                       # => {}
```

See Also

- Recipe 5.3, "Adding Elements to a Hash"
- Recipe 5.7, "Iterating Over a Hash"

Recipe 5.5. Using an Array or Other Modifiable Object as a Hash Key

Problem

You want to use a modifiable built-in object (an array or a hash, but not a string) as a key in a hash, even while you modify the object in place. A naive solution tends to lose hash values once the keys are modified:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
coordinates = [10, 5]
treasure_map = { coordinates => 'jewels' }
treasure_map[coordinates]                    # => "jewels"

# Add a z-coordinate to indicate how deep the treasure is buried.
coordinates << -5

coordinates                                  # => [10, 5, -5]
treasure_map[coordinates]                    # => nil
                                             # Oh no!
```

Solution

The easiest solution is to call the `Hash#rehash` method every time you modify one of the hash's keys. `Hash#rehash` will repair the broken treasure map defined above:

```
treasure_map.rehash
treasure_map[coordinates]                    # => "jewels"
```

If this is too much code, you might consider changing the definition of the object you use as a hash key, so that modifications don't affect the way the hash treats it.

Suppose you want a reliably hashable `Array` class. If you want this behavior universally, you can reopen the `Array` class and redefine `hash` to give you the new behavior. But it's safer to define a subclass of `Array` that implements a reliable-hashing mixin, and to use that subclass only for the `Arrays` you want to use as hash keys:

```
module ReliablyHashable
 def hash
    return object_id
  end
end

class ReliablyHashableArray < Array
  include ReliablyHashable
end
```

It's now possible to keep track of the jewels:

```
coordinates = ReliablyHashableArray.new([10,5])
treasure_map = { coordinates => 'jewels' }
treasure_map[coordinates]                    # => "jewels"

# Add a z-coordinate to indicate how deep the treasure is buried.
coordinates.push(-5)

treasure_map[coordinates]                    # => "jewels"
```

Discussion

Ruby performs hash lookups using not the key object itself but the object's *hash code* (an integer obtained from the key by calling its `hash` method). The default implementation of

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`hash`, in `Object`, uses an object's internal ID as its hash code. `Array`, `Hash`, and `String` override this method to provide different behavior.

In the initial example, the hash code of [10,5] is 41 and the hash code of [10,5,–5] is–83. The mapping of the coordinate list to 'jewels' is still present (it'll still show up in an iteration over `each_pair`), but once you change the coordinate list, you can no longer use that variable as a key.

You may also run into this problem when you use a hash or a string as a hash key, and then modify the key in place. This happens because the hash implementations of many built-in classes try to make sure that two objects that are "the same" (for instance, two distinct arrays with the same contents, or two distinct but identical strings) get the same hash value. When `coordinates` is [10,5], it has a hash code of 41, like any other `Array` containing [10,5]. When `coordinates` is [10,5,–5] it has a hash code of–83, like any other `Array` with those contents.

Because of the potential for confusion, some languages don't let you use arrays or hashes as hash keys at all. Ruby lets you do it, but you have to face the consequences if the key changes. Fortunately, you can dodge the consequences by overriding hash to work the way you want.

Since an object's internal ID never changes, the `Object` implementation is what you want to get reliable hashing. To get it back, you'll have to override or subclass the `hash` method of `Array` or `Hash` (depending on what type of key you're having trouble with).

The implementations of `hash` given in the solution violate the principle that different representations of the same data should have the same hash code. This means that two `ReliablyHashableArray` objects will have different hash codes even if they have the same contents. For instance:

```
a = [1,2]
b = a.clone
a.hash                                  # => 11
b.hash                                  # => 11
a = ReliablyHashableArray.new([1,2])
b = a.clone
a.hash                                  # => -606031406
b.hash                                  # => -606034266
```

If you want a particular value in a hash to be accessible by two different arrays with the same contents, then you must key it to a regular array instead of a `ReliablyHashableArray`. You can't have it both ways. If an object is to have the same hash key as its earlier self, it can't also have the same hash key as another representation of its current state.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Another solution is to freeze your hash keys. Any frozen object can be reliably used as a hash key, since you can't do anything to a frozen object that would cause its hash code to change. Ruby uses this solution: when you use a string as a hash key, Ruby copies the string, freezes the copy, and uses *that* as the actual hash key.

See Also

- Recipe 8.15, "Freezing an Object to Prevent Changes"

Recipe 5.6. Keeping Multiple Values for the Same Hash Key

Problem

You want to build a hash that might have duplicate values for some keys.

Solution

The simplest way is to create a hash that initializes missing values to empty arrays. You can then append items onto the automatically created arrays:

```
hash = Hash.new { |hash, key| hash[key] = [] }

raw_data = [ [1, 'a'], [1, 'b'], [1, 'c'],
             [2, 'a'], [2, ['b', 'c']],
             [3, 'c'] ]
raw_data.each { |x,y| hash[x] << y }
hash
# => {1=>["a", "b", "c"], 2=>["a", ["b", "c"]], 3=>["c"]}
```

Discussion

A hash maps any given key to only one value, but that value can be an array. This is a common phenomenon when reading data structures from the outside world. For instance, a list of tasks with associated priorities may contain multiple items with the same priority. Simply reading the tasks into a hash keyed on priority would create key collisions, and obliterate all but one task with any given priority.

It's possible to subclass `Hash` to act like a normal hash until a key collision occurs, and then start keeping an array of values for the key that suffered the collision:

```
class MultiValuedHash < Hash
  def []=(key, value)
    if has_key?(key)
      super(key, [value, self[key]].flatten)
    else
      super
    end
   end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
hash = MultiValuedHash.new
raw_data.each { |x,y| hash[x] = y }
hash
# => {1=>["c", "b", "a"], 2=>["b", "c", "a"], 3=>"c"}
```

This saves a little bit of memory, but it's harder to write code for this class than for one that always keeps values in an array. There's also no way of knowing whether a value `[1,2,3]` is a single array value or three numeric values.

See Also

- Recipe 5.2, "Creating a Hash with a Default Value," explains the technique of the dynamic default value in more detail, and explains why you must initalize the empty list within a code block—never within the arguments to `Hash.new`

Recipe 5.7. Iterating Over a Hash

Problem

You want to iterate over a hash's key-value pairs as though it were an array.

Solution

Most likely, the iterator you want is `Hash#each_pair` or `Hash#each`. These methods yield every key-value pair in the hash:

```
hash = { 1 => 'one', [1,2] => 'two', 'three' => 'three' }

hash.each_pair { |key, value| puts "#{key.inspect} maps to #{value}"}
# [1, 2] maps to two
# "three" maps to three
# 1 maps to one
```

Note that `each` and `each_pair` return the key-value pairs in an apparently random order.

Discussion

`Hash#each_pair` and `Hash#each` let you iterate over a hash as though it were an array full of key-value pairs. `Hash#each_pair` is more commonly used and slightly more efficient, but `Hash#each` is more array-like. `Hash` also provides several other iteration methods that can be more efficient than `each`.

Use `Hash#each_key` if you only need the keys of a hash. In this example, a list has been stored as a hash to allow for quick lookups (this is how the `Set` class works). The values are irrelevant, but `each_key` can be used to iterate over the keys:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
active_toggles = { 'super' => true, 'meta' => true, 'hyper' => true }
active_toggles.each_key { |active| puts active }
# hyper
# meta
# super
```

Use `Hash#each_value` if you only need the values of a hash. In this example, `each_value` is used to summarize the results of a survey. Here it's the keys that are irrelevant:

```
favorite_colors = { 'Alice' => :red, 'Bob' => :violet, 'Mallory' => :blue,
                    'Carol' => :blue, 'Dave' => :violet }

summary = Hash.new 0
favorite_colors.each_value { |x| summary[x] += 1 }
summary
# => {:red=>1, :violet=>2, :blue=>2}
```

Don't iterate over `Hash#each_value` looking for a particular value: it's simpler and faster to use `has_value?` instead.

```
hash = {}
1.upto(10) { |x| hash[x] = x * x }
hash.has_value? 49                    # => true
hash.has_value? 81                    # => true
hash.has_value? 50                    # => false
```

Removing unprocessed elements from a hash during an iteration prevents those items from being part of the iteration. However, adding elements to a hash during an iteration will not make them part of the iteration.

Don't modify the keyset of a hash during an iteration, or you'll get undefined results and possibly a `RuntimeError`:

```
1.upto(100) { |x| hash[x] = true }
hash.keys { |k| hash[k * 2] = true }
# RuntimeError: hash modified during iteration
```

Using an array as intermediary

An alternative to using the hash iterators is to get an array of the keys, values, or key-value pairs in the hash, and then work on the array. You can do this with the `keys, values`, and `to_a` methods, respectively:

```
hash = {1 => 2, 2 => 2, 3 => 10}
hash.keys                                 # => [1, 2, 3]
hash.values                               # => [2, 2, 10]
hash.to_a                                 # => [[1, 2], [2, 2], [3, 10]]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The most common use of `keys` and `values` is to iterate over a hash in a specific order. All of `Hash`'s iterators return items in a seemingly random order. If you want to iterate over a hash in a certain order, the best strategy is usually to create an array from some portion of the hash, sort the array, then iterate over it.

The most common case is to iterate over a hash according to some property of the keys. To do this, sort the result of `Hash#keys`. Use the original hash to look up the value for a key, if necessary.

```
extensions = { 'Alice' => '104', 'Carol' => '210', 'Bob' => '110' }
extensions.keys.sort.each do |k|
  puts "#{k} can be reached at extension ##{extensions[k]}"
end
# Alice can be reached at extension #104
# Bob can be reached at extension #110
# Carol can be reached at extension #210
```

`Hash#values` gives you the values of a hash, but that's not useful for iterating because it's so expensive to find the key for a corresponding value (and if you *only* wanted the values, you'd use `each_value`).

`Hash#sort` and `Hash#sort_by` turn a hash into an array of two-element subarrays (one for each key-value pair), then sort the array of arrays however you like. Your custom sort method can sort on the values, on the values and the keys, or on some relationship between key and value. You can then iterate over the sorted array the same as you would with the `Hash.each` iterator.

This code sorts a to-do list by priority, then alphabetically:

```
to_do = { 'Clean car' => 5, 'Take kangaroo to vet' => 3,
          'Realign plasma conduit' => 3 }
to_do.sort_by { |task, priority| [priority, task] }.each { |k,v| puts k }
# Realign plasma conduit
# Take kangaroo to vet
# Clean car
```

This code sorts a hash full of number pairs according to the magnitude of the difference between the key and the value:

```
transform_results = { 4 => 8, 9 => 9, 10 => 6, 2 => 7, 6 => 5 }
by_size_of_difference = transform_results.sort_by { |x, y| (x-y).abs }
by_size_of_difference.each { |x, y| puts "f(#{x})=#{y}: difference #{y-x}" }
# f(9)=9: difference 0
# f(6)=5: difference -1
# f(10)=6: difference -4
# f(4)=8: difference 4
# f(2)=7: difference 5
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- See Recipe 5.8, "Iterating Over a Hash in Insertion Order," for a more complex iterator
- Recipe 5.12, "Building a Histogram"
- Recipe 5.13, "Remapping the Keys and Values of a Hash"

Recipe 5.8. Iterating Over a Hash in Insertion Order

Problem

Iterations over a hash happen in a seemingly random order. Sorting the keys or values only works if the keys or values are all mutually comparable. You'd like to iterate over a hash in the order in which the elements were added to the hash.

Solution

Use the `orderedhash` library (see below for how to get it). Its `OrderedHash` class acts like a hash, but it keeps the elements of the hash in insertion order.

```
require 'orderedhash'
h = OrderedHash.new
h[1] = 1
h["second"] = 2
h[:third] = 3

h.keys                                          # => [1, "second", :third]
h.values                                        # => [1, 2, 3]
h.each { |k,v| puts "The #{k} counting number is #{v}" }
# The 1 counting number is 1
# The second counting number is 2
# The third counting number is 3
```

Discussion

`OrderedHash` is a subclass of `Hash` that also keeps an array of the keys in insertion order. When you add a key-value pair to the hash, `OrderedHash` modifies both the underlying hash and the array. When you ask for a specific hash element, you're using the hash. When you ask for the `keys` or the `values`, the data comes from the array, and you get it in insertion order.

Since `OrderedHash` is a real hash, it supports all the normal hash operations. But any operation that modifies an `OrderedHash` may also modify the internal array, so it's slower than just using a hash. `OrderedHash#delete` is especially slow, since it must perform a linear search of the internal array to find the key being deleted. `Hash#delete` runs in constant time, but `OrderedHash#delete` takes time proportionate to the size of the hash.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- You can get `OrderedHash` from the RAA at http://raa.ruby-lang.org/project/orderedhash/; it's not available as a gem, and it has no `setup.rb` script, so you'll need to distribute `orderedhash.rb` with your project, or copy it into your Ruby library path
- There is a `queuehash` gem that provides much the same functionality, but it has worse performance than `OrderedHash`

Recipe 5.9. Printing a Hash

Credit: Ben Giddings

Problem

You want to print out the contents of a Hash, but `Kernel#puts` doesn't give very useful results.

```
h = {}
h[:name] = "Robert"
h[:nickname] = "Bob"
h[:age] = 43
h[:email_addresses] = {:home => "bob@example.com",
                       :work => "robert@example.com"}
h
# => {:email_addresses=>["bob@example.com", "robert@example.com"],
#         :nickname=>"Bob", :name=>"Robert", :age=>43}
puts h
# nicknameBobage43nameRobertemail_addresseshomebob@example.comworkrobert@example.com
puts h[:email_addresses]
# homebob@example.comworkrobert@example.com
```

Solution

In other recipes, we sometimes reformat the results or output of Ruby statements so they'll look better on the printed page. In this recipe, you'll see raw, unretouched output, so you can compare different ways of printing hashes.

The easiest way to print a hash is to use `Kernel#p`. `Kernel#p` prints out the "inspected" version of its arguments: the string you get by calling `inspect` on the hash. The

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

"inspected" version of an object often looks like Ruby source code for creating the object, so it's usually readable:

```
p h[:email_addresses]
# {:home=>"bob@example.com", :work=>"robert@example.com"}
```

For small hashes intended for manual inspection, this may be all you need. However, there are two difficulties. One is that `Kernel#p` only prints to `stdout`. The second is that the printed version contains no newlines, making it difficult to read large hashes.

```
p h
# {:nickname=>"Bob", :age=>43, :name=>"Robert", :email_addresses=>{:home=>
# "bob@example.com", :work=>"robert@example.com"}}
```

When the hash you're trying to print is too large, the `pp` ("pretty-print") module produces very readable results:

```
require 'pp'
pp h[:email_addresses]
# {:home=>"bob@example.com", :work=>"robert@example.com"}

pp h
# {:email_addresses=>{:home=>"bob@example.com", :work=>"robert@example.com"}
# :nickname=>"Bob",
# :name=>"Robert",
# :age=>43}
```

Discussion

There are a number of ways of printing hash contents. The solution you choose depends on the complexity of the hash you're trying to print, where you're trying to print the hash, and your personal preferences. The best general-purpose solution is the `pp` library.

When a given hash element is too big to fit on one line, `pp` knows to put it on multiple lines. Not only that, but (as with `Hash#inspect`), the output is valid Ruby syntax for creating the hash: you can copy and paste it directly into a Ruby program to recreate the hash.

The `pp` library can also pretty-print to I/O streams besides standard output, and can print to shorter lines (the default line length is 79). This example prints the hash to `$stderr` and wraps at column 50:

```
PP::pp(h, $stderr, 50)
# {:nickname=>"Bob",
# :email_addresses=>
# {:home=>"bob@example.com",
# :work=>"robert@example.com"},
# :age=>43,
# :name=>"Robert"}
# => #<IO:0x2c8cc>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

You can also print hashes by converting them into YAML with the `yaml` library. YAML is a human-readable markup language for describing data structures:

```
require 'yaml'
puts h.to_yaml
# --
# :nickname: Bob
# :age: 43
# :name: Robert
# :email_addresses:
#   :home: bob@example.com
#   :work: robert@example.com
```

If none of these is suitable, you can print the hash out yourself by using `Hash#each_pair` to iterate over the hash elements:

```
h[:email_addresses].each_pair do |key, val|
  puts "#{key} => #{val}"
end
# home => bob@example.com
# work => robert@example.com
```

See Also

- Recipe 8.10, "Getting a Human-Readable Printout of Any Object," covers the general case of this problem
- Recipe 13.1, "Serializing Data with YAML"

Recipe 5.10. Inverting a Hash

Problem

Given a hash, you want to switch the keys and values. That is, you want to create a new hash whose keys are the values of the old hash, and whose values are the keys of the old hash. If the old hash mapped "human" to "wolf;" you want the new hash to map "wolf" to "human."

Solution

The simplest technique is to use the `Hash#invert` method:

```
phone_directory = {     'Alice' => '555-1212',
                        'Bob' => '555-1313',
                        'Mallory' => '111-1111' }
phone_directory.invert
# => {"111-1111"=>"Mallory", "555-1212"=>"Alice", "555-1313"=>"Bob"}
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

`Hash#invert` probably won't do what you want if your hash maps more than one key to the same value. Only one of the keys for that value will show up as a value in the inverted hash:

```
phone_directory = {     'Alice' => '555-1212',
                        'Bob' => '555-1313',
                        'Carol' => '555-1313',
                        'Mallory' => '111-1111',
                        'Ted' => '555-1212' }
phone_directory.invert
# => {"111-1111"=>"Mallory", "555-1212"=>"Ted", "555-1313"=>"Bob"}
```

To preserve all the data from the original hash, borrow the idea behind Recipe 5.6, and write a version of `invert` that keeps an array of values for each key. The following is based on code by Tilo Sloboda:

```
class Hash
  def safe_invert
    new_hash = {}
    self.each do |k,v|
      if v.is_a? Array
        v.each { |x| new_hash.add_or_append(x, k) }
      else
        new_hash.add_or_append(v, k)
      end
    end
    return new_hash
  end
```

The `add_or_append` method a lot like the method `MultivaluedHash#[]=` defined in Recipe 5.6:

```
  def add_or_append(key, value)
    if has_key?(key)
      self[key] = [value, self[key]].flatten
    else
      self[key] = value
    end
  end
end
```

Here's `safe_invert` in action:

```
phone_directory.safe_invert
# => {"111-1111"=>"Mallory", "555-1212"=>["Ted", "Alice"],
# "555-1313"=>["Bob", "Carol"]}

phone_directory.safe_invert.safe_invert
# => {"Alice"=>"555-1212", "Mallory"=>"111-1111", "Ted"=>"555-1212",
# => "Carol"=>"555-1313", "Bob"=>"555-1313"}
```

Ideally, if you called an inversion method twice you'd always get the same data you started with. The `safe_invert` method does better than `invert` on this score, but it's not

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

perfect. If your original hash used arrays as hash keys, `safe_invert` will act as if you'd individually mapped each element in the array to the same value. Call `safe_invert` twice, and the arrays will be gone.

See Also

- Recipe 5.5, "Using an Array or Other Modifiable Object as a Hash Key"
- "True Inversion of a Hash in Ruby," by Tilo Sloboda (http://www.unixgods.org/~tilo/Ruby/invert_hash.html)
- The Facets library defines a `Hash#inverse` method much like `safe_invert`

Recipe 5.11. Choosing Randomly from a Weighted List

Problem

You want to pick a random element from a collection, where each element in the collection has a different probability of being chosen.

Solution

Store the elements in a hash, mapped to their relative probabilities. The following code will work with a hash whose keys are mapped to relative integer probabilities:

```
def choose_weighted(weighted)
  sum = weighted.inject(0) do |sum, item_and_weight|
    sum += item_and_weight[1]
  end
  target = rand(sum)
  weighted.each do |item, weight|
    return item if target <= weight
    target -= weight
  end
end
```

For instance, if all the keys in the hash map to 1, the keys will be chosen with equal probability. If all the keys map to 1, except for one which maps to 10, that key will be picked 10 times more often than any single other key. This algorithm lets you simulate those probability problems that begin like, "You have a box containing 51 black marbles and 17 white marbles...":

```
marbles = { :black => 51, :white => 17 }
3.times { puts choose_weighted(marbles) }
# black
# white
# black
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

I'll use it to simulate a lottery in which the results have different probabilities of showing up:

```
lottery_probabilities = { "You've wasted your money!" => 1000,
                          "You've won back the cost of your ticket!" => 50,
                          "You've won two shiny zorkmids!" => 20,
                          "You've won five zorkmids!" => 10,
                          "You've won ten zorkmids!" => 5,
                          "You've won a hundred zorkmids!" => 1 }

# Let's buy some lottery tickets.
5.times { puts choose_weighted(lottery_probabilities) }
# You've wasted your money!
# You've wasted your money!
# You've wasted your money!
# You've wasted your money!
# You've won five zorkmids!
```

Discussion

An extremely naive solution would put the elements in a list and choose one at random. This doesn't solve the problem because it ignores weights altogether: low-weight elements will show up exactly as often as high-weight ones. A less naive solution would be to repeat each element in the list a number of times proportional to its weight. Under this implementation, our simulation of the marble box would contain `:black` 51 times and `:white` 17 times, just like a real marble box. This is a common quick-and-dirty solution, but it's hard to maintain, and it uses lots of memory.

The algorithm given above actually works the same way as the less naive solution: the numeric weights stand in for multiple copies of the same object. Instead of picking one of the 68 marbles, we pick a number between 0 and 67 inclusive. Since we know there are 51 black marbles, we simply decide that the numbers from 0 to 50 will represent black marbles.

For the implementation given above to work, all the weights in the hash must be integers. This isn't a big problem the first time you create a hash, but suppose that after the lottery has been running for a while, you decide to add a new jackpot that's 10 times less common than the 100-zorkmid jackpot. You'd like to give this new possibility a weight of 0.1, but that won't work with the `choose_weighted` implementation. You'll need to give it a weight of 1, and multiply all the existing weights by 10.

There is an alternative, though: normalize the weights so that they add up to 1. You can then generate a random floating-point number between 0 and 1, and use a similar algorithm to the one above. This approach lets you weight the hash keys using any numeric objects you like, since normalization turns them all into small floating-point numbers anyway.

```
def normalize!(weighted)
  sum = weighted.inject(0) do |sum, item_and_weight|
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    sum += item_and_weight[1]
  end
  sum = sum.to_f
  weighted.each { |item, weight| weighted[item] = weight/sum }
end

lottery_probabilities["You've won five hundred zorkmids!"] = 0.1
normalize!(lottery_probabilities)
# => { "You've wasted your money!" => 0.920725531718995,
#          "You've won back the cost of your ticket!" => 0.0460362765859497,
#          "You've won two shiny zorkmids!" => 0.0184145106343799,
#          "You've won five zorkmids!" => 0.00920725531718995,
#          "You've won ten zorkmids!" => 0.00460362765859497,
#          "You've won a hundred zorkmids!" => 0.000920725531718995,
#          "You've won five hundred zorkmids!" => 9.20725531718995e-05 }
```

Once the weights have been normalized, we know that they sum to one (within the limits of floating-point arithmetic). This simplifies the code that picks an element at random, since we don't have to sum up the weights every time:

```
def choose_weighted_assuming_unity(weighted)
  target = rand
  weighted.each do |item, weight|
    return item if target <= weight
    target -= weight
  end
end

5.times { puts choose_weighted_assuming_unity(lottery_probabilities) }
# You've wasted your money!
# You've wasted your money!
# You've wasted your money!
# You've wasted your money!
# You've won back the cost of your ticket!
```

See Also

- Recipe 2.5, "Generating Random Numbers"
- Recipe 6.9, "Picking a Random Line from a File"

Recipe 5.12. Building a Histogram

Problem

You have an array that contains a lot of references to relatively few objects. You want to create a histogram, or frequency map: something you can use to see how often a given object shows up in the array.

Solution

Build the histogram in a hash, mapping each object found to the number of times it appears.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
module Enumerable
  def to_histogram
    inject(Hash.new(0)) { |h, x| h[x] += 1; h}
  end
end

[1, 2, 2, 2, 3, 3].to_histogram
# => {1=>1, 2=>3, 3=>2}

["a", "b", nil, "c", "b", nil, "a"].to_histogram
# => {"a"=>2, "b"=>2, "c"=>1, nil=>2}

"Aye\nNay\nNay\nAbstaining\nAye\nNay\nNot Present\n".to_histogram
# => {"Abstaining\n"=>1, "Nay\n"=>3, "Not Present\n"=>1, "Aye\n"=>2}

survey_results = { "Alice" => :red, "Bob" => :green, "Carol" => :green,
                   "Mallory" => :blue }
survey_results.values.to_histogram
# => {:red=>1, :green=>2, :blue=>1}
```

Discussion

Making a histogram is an easy and fast (linear-time) way to summarize a dataset. Histograms expose the relative popularity of the items in a dataset, so they're useful for visualizing optimization problems and dividing the "head" from the "long tail."

Once you have a histogram, you can find the most or least common elements in the list, sort the list by frequency of appearance, or see whether the distribution of items matches your expectations. Many of the other recipes in this book build a histogram as a first step towards a more complex algorithm.

Here's a quick way of visualizing a histogram as an ASCII chart. First, we convert the histogram keys to their string representations so they can be sorted and printed. We also store the histogram value for the key, since we can't do a histogram lookup later based on the string value we'll be using.

```
def draw_graph(histogram, char="#")
  pairs = histogram.keys.collect { |x| [x.to_s, histogram[x]] }.sort
```

Then we find the key with the longest string representation. We'll pad the rest of the histogram rows to this length, so that the graph bars will line up correctly.

```
  largest_key_size = pairs.max { |x,y| x[0].size <=> y[0].size }[0].size
```

Then we print each key-value pair, padding with spaces as necessary.

```
  pairs.inject("") do |s,kv|
    s << "#{kv[0].ljust(largest_key_size)} |#{char*kv[1]}\n"
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's a histogram of the color survey results from the Solution:

```
puts draw_graph(survey_results.values.to_histogram)
# blue  |#
# green |##
# red   |#
```

This code generates a bunch of random numbers, then graphs the random distribution:

```
random = []
100.times { random << rand(10) }
puts draw_graph(random.to_histogram)
# 0 |############
# 1 |########
# 2 |#######
# 3 |#########
# 4 |##########
# 5 |#############
# 6 |###############
# 7 |########
# 8 |#######
# 9 |###########
```

See Also

- Recipe 2.8, "Finding Mean, Median, and Mode"
- Recipe 4.9, "Sorting an Array by Frequency of Appearance"

Recipe 5.13. Remapping the Keys and Values of a Hash

Problem

You have two hashes with common keys but differing values. You want to create a new hash that maps the values of one hash to the values of another.

Solution

```
class Hash
  def tied_with(hash)
    remap do |h,key,value|
      h[hash[key]] = value
    end.delete_if { |key,value| key.nil? || value.nil? }
  end
```

Here is the `Hash#remap` method:

```
  def remap(hash={})
    each { |k,v| yield hash, k, v }
    hash
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's how to use `Hash#tied_with` to merge two hashes:

```
a = {1 => 2, 3 => 4}
b = {1 => 'foo', 3 => 'bar'}
a.tied_with(b)                          # => {"foo"=>2, "bar"=>4}
b.tied_with(a)                          # => {2=>"foo", 4=>"bar"}
```

Discussion

This `remap` method can be handy when you want to make a similar change to every item in a hash. It is also a good example of using the `yield` method.

`Hash#remap` is conceptually similar to `Hash#collect`, but `Hash#collect` builds up a nested array of key-value pairs, not a new hash.

See Also

- The Facets library defines methods `Hash#update_each` and `Hash#replace_each!` for remapping the keys and values of a hash

Recipe 5.14. Extracting Portions of Hashes

Problem

You have a hash that contains a lot of values, but only a few of them are interesting. You want to select the interesting values and ignore the rest.

Solution

You can use the `Hash#select` method to extract part of a hash that follows a certain rule. Suppose you had a hash where the keys were `Time` objects representing a certain date, and the values were the number of web site clicks for that given day. We'll simulate such as hash with random data:

```
require 'time'
click_counts = {}
1.upto(30) { |i| click_counts[Time.parse("2006-09-#{i}")] = 400 + rand(700) }
p click_counts
# {Sat Sep 23 00:00:00 EDT 2006=>803, Tue Sep 12 00:00:00 EDT 2006=>829,
# Fri Sep 01 00:00:00 EDT 2006=>995, Mon Sep 25 00:00:00 EDT 2006=>587,
# …
```

You might want to know the days when your click counts were low, to see if you could spot a trend. `Hash#select` can do that for you:

```
low_click_days = click_counts.select {|key, value| value < 450 }
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# [[Thu Sep 14 00:00:00 EDT 2006, 449], [Mon Sep 11 00:00:00 EDT 2006, 406],
#  [Sat Sep 02 00:00:00 EDT 2006, 440], [Mon Sep 04 00:00:00 EDT 2006, 431],
# …
```

Discussion

The array returned by `Hash#select` contains a number of key-value pairs as two-element arrays. The first element of one of these inner arrays is a key into the hash, and the second element is the corresponding value. This is similar to how `Hash#each` yields a succession of two-element arrays.

If you want another hash instead of an array of key-value pairs, you can use `Hash#inject` instead of `Hash#select`. In the code below, `kv` is a two-element array containing a key-value pair. `kv[0]` is a key from `click_counts`, and `kv[1]` is the corresponding value.

```
low_click_days_hash = click_counts.inject({}) do |h, kv|
  k, v = kv
  h[k] = v if v < 450
  h
end
# => {Mon Sep 25 00:00:00 EDT 2006=>403,
#         Wed Sep 06 00:00:00 EDT 2006=>443,
#         Thu Sep 28 00:00:00 EDT 2006=>419}
```

You can also use the `Hash.[]` constructor to create a hash from the array result of `Hash#select`:

```
low_click_days_hash = Hash[*low_click_days.flatten]
# => {Thu Sep 14 00:00:00 EDT 2006=>449, Mon Sep 11 00:00:00 EDT 2006=>406,
#         Sat Sep 02 00:00:00 EDT 2006=>440, Mon Sep 04 00:00:00 EDT 2006=>431,
#         …
```

See Also

- Recipe 4.13, "Extracting Portions of Arrays"

Recipe 5.15. Searching a Hash with Regular Expressions

Credit: Ben Giddings

Problem

You want to grep a hash: that is, find all keys and/or values in the hash that match a regular expression.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

The fastest way to grep the keys of a hash is to get the keys as an array, and grep that:

```
h = { "apple tree" => "plant", "ficus" => "plant",
      "shrew" => "animal", "plesiosaur" => "animal" }
h.keys.grep /p/
# => ["apple tree", "plesiosaur"]
```

The solution for grepping the values of a hash is similar (substitute `Hash#values` for `Hash#keys`), unless you need to map the values back to the keys of the hash. If that's what you need, the fastest way is to use `Hash#each` to get key-value pairs, and match the regular expression against each value.

```
h.inject([]) { |res, kv| res << kv if kv[1] =~ /p/; res }
# => [["ficus", "plant"], ["apple tree", "plant"]]
```

The code is similar if you need to find key-value pairs where either the key or the value matches a regular expression:

```
class Hash
  def grep(pattern)
    inject([]) do |res, kv|
      res << kv if kv[0] =~ pattern or kv[1] =~ pattern
      res
    end
  end
end

h.grep(/pl/)
# => [["ficus", "plant"], ["apple tree", "plant"], ["plesiosaur", "animal"]]
h.grep(/plant/)           # => [["ficus", "plant"], ["apple tree", "plant"]]
h.grep(/i.*u/)            # => [["ficus", "plant"], ["plesiosaur", "animal"]]
```

Discussion

`Hash` defines its own `grep` method, but it will never give you any results. `Hash#grep` is inherited from `Enumerable#grep`, which tries to match the output of `each` against the given regular expression. `Hash#each` returns a series of two-item arrays containing key-value pairs, and an array will never match a regular expression. The `Hash#grep` implementation above is more useful.

`Hash#keys.grep` and `Hash#values.grep` are more efficient than matching a regular expression against each key or value in a `Hash`, but those methods create a new array containing all the keys in the Hash. If memory usage is your primary concern, iterate over `each_key` or `each_value` instead:

```
res = []
h.each_key { |k| res << k if k =~ /p/ }
res                   # => ["apple tree", "plesiosaur"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01