Given Ruby already supports destructuring assignment with Array (a, b = [1, 2]), I propose destructuring assignments for Hash.

Basic example

```ruby
params = {name: "John Smith", age: 42}
{name: name, age: age} = params

# name == "John Smith"
# age == 42
```

This would replace a common pattern of assigning hash values to local variables to work with.

General syntax

```ruby
{ <key-exp> => <variable_name>, ... } = <object that responds to #[]>
```

```ruby
# Symbols
{ foo: bar } = { foo: "bar" }
bbar == "bar"
```

```ruby
# Potential shorthand
{ foo } = { foo: "bar" }
foo == "bar"
```

Use cases

```ruby
# MatchData
{ username: username, age: age } = "user:jsmith age:42".match(/user:(?<username>\w+) age:(?<age>\d+)/)
username == "jsmith"
age == "42"
```

Edge cases

```ruby
# Variable being assigned to more than once should use the last one
{ foo: var, bar: var } = {foo: 1, bar: 2}
var == 2
```

Thoughts?

Related issues:

- Related to Ruby master - Bug #10028: nested rest keyword argument (Rejected)
- Is duplicate of Ruby master - Feature #6414: Destructuring Assignment (Closed)

History

#1 - 09/11/2013 06:16 PM - whitequark (whitequark *)

This is an awesome idea! However, the parser bit is really evil. I tried implementing it myself (quite a bit of time ago) and completely gave up. It's not above my comprehension, but the amount of work even for my Ruby parser port is huge and daunting. Doing it in C is a nightmare.

That being said, I'm willing to discuss and/or provide guidance to any interested parties.
Marcandre (Marc-Andre Lafortune) wrote:

I suggested something similar in [ruby-core:41772].
Here is a summary from my similar suggestion made in [ruby-core:41772]:

```ruby
{:
key: 'default',
other_key: **other_options} = {other_key: 42, foo: 'bar'}
key # => 'default'
other_key # => 42
other_options # => {foo: 'bar'}
```

You'll note that it doesn't give the possibility to map the key to a different variable. Indeed, I don't think that it would be useful and I would rather encourage rubists to use meaningful option and variable names. It also makes very similar to the way we declare keyword arguments for methods, so no additional learning curve. Your proposal is quite different.

Sawa (Tsuyoshi Sawada) wrote:

Given that destructive assignments with array prohibits the [] on the left side of the assignment, that is:

```
a, b = [1, 2]
```

instead of:

```
[a, b] = [1, 2]
```

it would be more consistent if your proposal were:

```
n = name, age: age = {name: "John Smith", age: 42}
```

rather than:

```
{name: name, age: age} = {name: "John Smith", age: 42}
```

Chendo (Jack Chen) wrote:

Marcandre (Marc-Andre Lafortune) wrote:

I suggested something similar in [ruby-core:41772].
Here is a summary from my similar suggestion made in [ruby-core:41772]:

```ruby
{:
key: 'default',
other_key: **other_options} = {other_key: 42, foo: 'bar'}
key # => 'default'
other_key # => 42
other_options # => {foo: 'bar'}
```

You'll note that it doesn't give the possibility to map the key to a different variable. Indeed, I don't think that it would be useful and I would rather encourage rubists to use meaningful option and variable names. It also makes very similar to the way we declare keyword arguments for methods, so no additional learning curve. Your proposal is quite different.

I considered the case of default options, but I couldn't figure out a way to make it read well, and there are many cases where the keys in the hash are not symbols. No value variable after other_key: feels a bit off to me, too.

I'm all for a way to figure out how to get the use case of default options in somehow but I feel that needs more consideration where as this is useful by itself.

Sawa (Tsuyoshi Sawada) wrote:

Given that destructive assignments with array prohibits the [] on the left side of the assignment, that is:

```
a, b = [1, 2]
```

instead of:

```
[a, b] = [1, 2]
```

it would be more consistent if your proposal were:

```
n = name, age: age = {name: "John Smith", age: 42}
```

rather than:

```
{name: name, age: age} = {name: "John Smith", age: 42}
```
I left the braces in because I felt it would be easier to parse, however if without braces is doable as well, that would work also. Will update the proposal.

#6 - 09/12/2013 11:58 AM - marcandre (Marc-Andre Lafortune)
chendo (Jack Chen) wrote:

No value variable after other_key: feels a bit off to me, too.

Not surprising it feels off today, but you better get used to it because it's coming in 2.1.0: https://bugs.ruby-lang.org/issues/7701

#7 - 09/15/2013 03:59 PM - Anonymous
whitequark (whitequark *): Hi whitequark, you here? Let me raise my commendations to you for your parser gem!

As for the issue at hand, why not just say:

```ruby
{ name: "JohnSmith", age: 42 }!
```

and have the assignment done:

```ruby
name = "JohnSmith"
age = 42
```

If you want the assignment done to different variables, why not take Rails's Hash#slice one step further:

```ruby
{ n: "JohnSmith", a: 42, foo: "bar" }.slice( name: :n, age: :a ) # produces { name: "JohnSmith", age: 42 }
```

and then

```ruby
{ n: "JohnSmith", a: 42, foo: "bar" }.slice( name: :n, age: :a ) !
# foo: "bar" is ignored away by the #slice method
```

produces the desired assignment:

```ruby
name = "JohnSmith"
age = 42
```

I hope that .! syntax proposal doesn't suck too hard! It might be a general way of making objects perform assignments to local variables. I'm concerned about feature creep, though.

#8 - 09/15/2013 04:06 PM - charliesome (Charlie Somerville)
boris_stitnicky: This sort of feature would be close to impossible to implement in CRuby. I can't speak for JRuby or Rubinius (although I would imagine they're in the same position here) but CRuby relies on being able to statically determine all local variables for a scope ahead of time.

#9 - 09/15/2013 04:21 PM - Anonymous
charliesome (Charlie Somerville): I thought myself chendo was stretching it, thanks for making me realize why I felt so. It's all about those famous

```ruby
a = a #=> nil
```

cases :-) But... somehow... sorry for a quiche eater like me to say this... I thought that maybe being able to statically determine local variables is itself a design smell that might need to be removed from the language... Sorry again for raising issues.

#10 - 09/20/2013 09:35 PM - alexeymuranov (Alexey Muranov)
How about this:

```ruby
(x, y, *rest, :a => v1, :b => v2, **options) = 1, 2, 3, 4, :a => :foo, :b => :bar, :c => false, :d => true
x      #=> 1
y      #=> 2
rest   #=> [3, 4]
v1     #=> :foo
v2     #=> :bar
options #=> {:c=>false, :d=>true}
```

#11 - 04/17/2014 11:25 PM - seanlinsley (Sean Linsley)
This is what I'm imagining:

```ruby
a, *b, c:, d: 'd', **e = [1, {c: 2}]
```

a == 1
Where an error would be thrown if the hash didn't have the given key, and no default was provided.

**#12 - 07/10/2014 07:00 AM - ko1 (Koichi Sasada)**

- Description updated

**#13 - 07/10/2014 07:11 AM - ko1 (Koichi Sasada)**

+1 for this proposal.

I feel it is fine for me:

```ruby
k1: 1, k2: 2 = h
kr1:, kr2: = h
 #=> same as
k1 = h.fetch(k1, 1)
k2 = h.fetch(k2, 2)
kr1 = h.fetch(k1)
kr2 = h.fetch(k2)
```

Problem is what happen when 'h' is not a hash object (and doesn't have to_hash method).
Just ignore is one option (what ary assignment do, like "a, b = 1 #=> 1, nil").

```ruby
a, *b, c:, d: 'd', **e = [1, {c: 2}]
```

It should be:

```ruby
a, (c, d: 'd', **e) = [1, {c: 2}]
```

**#14 - 07/11/2014 08:03 AM - nobu (Nobuyoshi Nakada)**

- Related to Bug #10028: nested rest keyword argument added

**#15 - 07/13/2014 11:05 PM - seanlinsley (Sean Linsley)**

Koichi Sasada wrote:

Problem is what happen when 'h' is not a hash object (and doesn't have to_hash method).
Just ignore is one option (what any assignment do, like "a, b = 1 #=> 1, nil").

```ruby
a, *b, c:, d: 'd', **e = [1, {c: 2}]
```

It should be:

```ruby
a, (c, d: 'd', **e) = [1, {c: 2}]
```

I don't follow. Can't this assignment behave the same way that method argument destructuring does? This currently works:

```ruby
def foo(a, *b, c:, d: 'd', **e)
  [a, b, c, d, e]
end

foo 1, c: 2
 #=> [1, [], 2, "d", {}]
```

**#16 - 07/14/2014 10:02 AM - marcandre (Marc-Andre Lafortune)**
Sean Linsley wrote:
I don't follow. Can't this assignment behave the same way that method argument destructuring does?

I agree. Destructuring should work as method & block passing works (apart from block passing)

#17 - 05/17/2016 08:18 AM - matz (Yukihiro Matsumoto)
The proposed syntax is much harder to implement than it looks. It conflicts with Hash literals. As a result, humans can be confused as well.

Probably this kind of problems should be addressed by pattern matching, which we are considering to add to Ruby in the future.

Matz.

#18 - 08/09/2016 05:42 PM - bughit (bug hit)
the closest you can get to hash destructuring is via block params:

```
{a: 1, b: 2}.tap do |a:, b:|  
end
```

but unfortunately this has its own issues (#11048), it's too strict about missing/extra keys, which doesn't make sense since blocks are intended to be looser with parameter binding.

#19 - 10/06/2021 08:32 PM - jeremyevans0 (Jeremy Evans)
- Is duplicate of Feature #6414: Destructuring Assignment added

#20 - 10/06/2021 08:32 PM - jeremyevans0 (Jeremy Evans)
- Status changed from Open to Closed