

Proposal to add Data#[]

05/31/2025 06:10 AM - ybiquitous (Masafumi Koba)

<div><div>Status:Open</div><div>Priority:Normal</div><div>Assignee:</div><div>Target version:</div></div>	
<div><div>Description</div><div><div>Proposal</div><div>I propose to add a new instance method #[] to the Data class, similar to Struct#[].</div><div>If writing the method signature in RBS, it would be like this:</div><div><pre>class Data def []: (name: String Symbol) -> untyped end</pre></div><div>Requirements:</div><div><ul style="list-style-type: none">• Data#[] accepts a member name as a Symbol or String, e.g., data[:id] == data["id"].• Data#[] returns a value associated with the given name, e.g., data[:id] == data.id.• Data#[] raises NameError if the given name is not one of the members, e.g., data[:invalid].</div><div>Note: Please assume that data = Data.define(:id).new(id: 100) is given in the examples above.</div><div><div>Motivation</div><div>In Active Support Core Extensions of Rails, I found a use case that Data#[] would be helpful with Enumerable#pluck.</div><div>Please look at this example:</div><div><pre># data_test.rb require "bundler/inline" gemfile do source "https://rubygems.org" gem "activesupport", "8.0.2" end require "active_support/core_ext/enumerable" StructPerson = Struct.new(:name, :age) DataPerson = Data.define(:name, :age) struct_people = [StructPerson.new("Bob", 25),] puts "Struct:" puts "=> #{struct_people.pluck(:name)}" data_people = [DataPerson.new("Charlie", 35),] puts "Data:" begin puts "=> #{data_people.pluck(:name)}" rescue => e puts e.detailed_message(syntax_suggest: true) end</pre></div></div></div></div>	

Running this script outputs below:

```
$ ruby data_test.rb
Struct:
=> ["Bob"]
Data:
undefined method '[]' for an instance of DataPerson (NoMethodError)
```

```
map { |element| element[key] }
      ^^^^^
```

Note: This output resulted on ruby 3.4.4 (2025-05-14 revision a38531fd3f) +PRISM [arm64-darwin24].

The error reason is that the Enumerable#pluck extension expects all the elements in the array to respond to [].

See also the pluck code here:

https://github.com/rails/rails/blob/v8.0.2/activerecord/lib/active_record/core_ext/enumerable.rb#L145-L152

If Data#[] was implemented as follows,

```
# data_patch.rb
class Data
  def [] (name)
    unless members.include?(name.to_sym)
      raise NameError, "no member '#{name}' in data"
    end
    public_send(name)
  end
end
```

The script is successful:

```
$ ruby -r ./data_patch.rb data_test.rb
Struct:
=> ["Bob"]
Data:
=> ["Charlie"]
```

Ref:

- https://guides.rubyonrails.org/v8.0.2/active_support_core_extensions.html#pluck
- <https://api.rubyonrails.org/v8.0.2/classes/Enumerable.html#method-i-pluck>

Although Enumerable#pluck is just an example, I guess that there would be other cases where Data objects should respond to [].

Reasoning

From the long discussion in [#16122](#) that introduced Data, I understand that Data#[] was rejected in [#16122#note-28](#) because [] was like Enumerable.

I also agree with rejecting Data#each since Data is not an Enumerable, but [] seems acceptable to me.

Reasons:

- [] is sometimes used for non-container objects, such as [ActiveRecord::AttributeMethods#\[\]](#).
- Considering Data#to_h is provided, it seems reasonable that we could access a Data member's value through [].
- From the similarity between Struct and Data, some people might expect Hash-like accessing via [].
- Unless [] is provided, we have to call public_send or convert to a hash via to_h when we want to get member values with names. It'd be inefficient and easy to make mistakes.

Let me show an example:

```
Person = Data.define(:name, :age)
charlie = Person.new("Charlie", 35)

member_names_from_user_input = ["age", "hash"] # "hash" is a malicious input.
```

```

# Using #public_send
member_names_from_user_input.map { charlie.public_send(it) }
#=> [35, -1363726049241242821]

# Using #to_h and Hash#fetch with string keys
member_names_from_user_input.map { charlie.to_h.fetch(it) }
#=> key not found: "age" (KeyError)

# Using #to_h and Hash#fetch with symbol keys
member_names_from_user_input.map { charlie.to_h.fetch(it.to_sym) }
#=> key not found: :hash (KeyError)

# Using #[]
member_names_from_user_input.map { charlie[it] }
#=> no member 'hash' in data (NameError)

```

The last example is most elegant for the same goal. That's why I propose to add `Data#[]`.

However, there might be downsides that I have overlooked, so I'd appreciate it if you could let me know.

Thanks.

History

#1 - 05/31/2025 07:16 AM - zverok (Victor Shepelev)

The design goal of `Data` was to be as close to "just a simple atomic object" as possible and convenient. It is a "value object," not a "container". `#[]` is closer to a container protocol and starts to erode the design.

For a particular justification examples:

For the first one, instead of `data_people.pluck(:name)` you can just use `data_people.map(&:name)`.

For the second one (user's input as data members)--I don't think it is the language's goal to support particular cases like this. If it is realistic, then implementing it yourself is a couple of lines--and gives the program's author freedom to handle this "security problem" any way they find suitable:

```

Person = Data.define(:name, :age) do
  def [](key)
    members.include?(key) or raise SecurityError, "They tried to still the #{key}!"
    public_send(key)
  end
end

```

Introducing `#[]` is like opening a can of worms on the "why it is not more like Hash/Struct" question. Like, if we support dynamic key passing (and generally treating *fields* as *container keys*), why can't we have "enumerate everything dynamically"? Why don't we have `#[]=?` Etc. etc.

#2 - 05/31/2025 01:41 PM - ybiquitous (Masafumi Koba)

[@zverok \(Victor Shepelev\)](#) Thank you for the feedback!

My example of `data_people.pluck(:name)` was not so good, sorry. A strength of `pluck` is to accept multiple keys like `data_people.pluck(:name, :age)`, so `.map` isn't a proper alternative in this case.

For the second example of `member_names_from_user_input = ["age", "hash"]`, the use case may not be so typical, as you pointed out, although it's not so smart for every `Data.define` in an app or library to implement `#[]` repeatedly.

It is a "value object," not a "container".

I understand your intention in the words and that you're very careful to extend the current scope of `Data` more than necessary.

However, I'm still curious why I've come up with this `#[]` idea.

- Data objects can be initialized with keyword arguments, like Hash objects.
- Data objects can be converted to Hash objects via `#to_h`.

I think that these `Data` features (or interfaces) would easily make people recognize the similarity between `Data` and `Hash`. Especially when `Data` objects are serialized as JSON and used to communicate with external systems, I bet that the similarity stands out to people.

I also don't think `Data` should mimic `Hash` mostly, but unless this similarity essentially exists, providing a few methods helping `Data` programming seems acceptable to many folks, and I don't think it violates the `Data` design policy.

Also, from the perspective of Duck Typing, it seems to make sense that people expect Data to respond to #[] like Hash.

That's why I still think Data#[] is a good compromise to maximize its usability and keep its design policy as-is.

#3 - 05/31/2025 04:42 PM - zverok (Victor Shepelev)

However, I'm still curious why I've come up with this #[] idea.

- Data objects can be initialized with keyword arguments, like Hash objects.
- Data objects can be converted to Hash objects via #to_h.

I think that these Data features (or interfaces) would easily make people recognize the similarity between Data and Hash.

I am not sure about this argument. The "can be initialized with keyword arguments" and "can be converted to Hash" are probably the most common features among all kinds of data-representing/data-modelling objects.

The logical leap "so they are like Hash, and should have more Hash-like behavior" doesn't work for me personally.

But others might have other opinions.

Nowadays, I don't have the capacity to be involved in Ruby's development; I just explained the initial design idea. Let the core team decide.

#4 - 06/01/2025 07:48 PM - Eregon (Benoit Daloze)

I think it's easy enough to add [] for your Data subclass where you want it.

I think it's best to avoid adding extra methods to Data as it seems not unlikely some might want [] to not be defined or to be defined differently.

[@zverok \(Victor Shepelev\)](#) makes a good point that Data shouldn't be used like a Hash with dynamic keys (especially not become OpenStruct-like). It's a data structure with well-known fields and these should be accessed directly, the case of accessing them generically is the exception and can already be done through to_h.

#5 - 06/02/2025 10:17 AM - ybiquitous (Masafumi Koba)

[@zverok \(Victor Shepelev\)](#) [@Eregon \(Benoit Daloze\)](#) Thanks for the feedback.

Reconsidering my motivation, I may have expected Data to be something like an *immutable* Struct (or even an *immutable* Hash).

I'm still attracted to this idea of an immutable Hash-like data structure, but I must agree with the possibility of interface pollution that Data would suffer like OpenStruct, as you worry. As long as there is no fascinating use case, we should probably stop enhancing the Data interface.

Also, I can work around the issue by extending Data only in my app, not monkey patching, like this:

```
module DynamicallyAccessibleData
  def [](name)
    unless members.include?(name.to_sym)
      raise NameError, "no member '#{name}' in data"
    end
    public_send(name)
  end
end

Person = Data.define(:name, :age).include(DynamicallyAccessibleData)
person = Person.new(name: "Charlie", age: 35)
person[:name] #=> "Charlie"
person[:age] #=> 35
person[:foo] #=> NameError: no member 'foo' in data
```

I'll leave the ticket open for a while since anyone else might be interested, but if no one reacts, I'll close it.

Otherwise, please close it if you find it clear that nobody does. Thanks.