Ruby - Feature #18690

Allow `Kernel#then` to take arguments

04/12/2022 09:20 AM - sawa (Tsuyoshi Sawada)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Kernel#then passes the receiver to the block as its first positional block parameter.

```
1.5.then{|x| Math.atan(x)}
```

I would like to propose to let then take arguments, which would be passed to the block as the other block parameters.

```
3.then(4){|x, y| Math.hypot(x, y)}
```

There are two uses. First, to separate bulky or repeated parameters from the routine. Instead of writing:

```
honyarara.then{|x|
  foo(x)
  bar(fugafugafuga)
  baz(hogehogehoge)
  qux(x, fugafugafuga, hogehogehoge)
}
```

we can then write:

```
honyarara.then(fugafugafuga, hogehogehoge){|x, y, z|
  foo(x)
  bar(y)
  baz(x)
  qux(x, y, z)
}
```

Second, to use a proc with multiple parameters when, for some reason, you do not want to define a method to do it:

```
p = ->(x, y, z){
  foo(x)
  bar(y)
  baz(x)
  qux(x, y, z)
}

honyarara.then(fugafugafuga, hogehogehoge, &p)
```

History

#1 - 04/12/2022 09:21 AM - sawa (Tsuyoshi Sawada)

- Description updated

#2 - 04/12/2022 09:52 AM - Eregon (Benoit Daloze)

The last example is just:

```
p.call(honyarara, fugafugafuga, hogehogehoge)
```

isn't it? And that's a lot more readable IMHO.

I'm against this proposal, IMHO having multiple local variables for the same thing only increases confusion and hurt readability.
If fugafugafuga etc are long method calls/expressions, then they could be saved in local variables outside the then block and that would again be more readable.

Also then is an alias of yield_self which is literally yield self, so I think those semantics would be weird for yield self.

I think then/yield_safe makes sense in a method chain (to not need to break it in multiple lines/break the reading flow).
Extra variables/arguments as you show can just be declared before/outside-the-block as local variables. By definition they are independent of the method chain and so that seems always a better solution.

#3 - 05/10/2022 06:02 PM - nevans (Nicholas Evans)

For your scenarios, as written, I agree with Benoit's #note-2 suggestions. ;) I also agree that core/stdlib #then should only ever yield a single value to its block. *However*, it's worth noting that multi-parameter blocks will automatically deconstruct a single array arg. E.g:

```
irb(main):001:0> %i[hello world this_is_the_third].then {|x, y, z| puts "first: #{x}"; puts "second: #{y}";
puts "third: #{z}"; [x.to_s, y.to_s.upcase, z.to_s[-5..]] }
first: hello
second: world
third: this_is_the_third
=> ["hello", "WORLD", "third"]



irb(main):023:0> def foo(x) = puts "foo(%p)" % x
=> :foo
irb(main):024:0> def bar(y) = puts "bar(%p)" % y
=> :bar
irb(main):025:0> def baz(x) = puts "baz(%p)" % x
=> :baz
irb(main):026:0> def qux(x, y, z) = puts("qux(%p, %p, %p)" % [x, y, z]).then { :done_qux }
=> :qux
irb(main):027:0> honyarara, fugafugafuga, hogehogehoge = :honyarara, :fugafugafuga, :hogehogehoge
=> [:honyarara, :fugafugafuga, :hogehogehoge]
irb(main):028:1* [honyarara, fugafugafuga, hogehogehoge].then {|x,y,z|
# blocks automatically destructure array args
irb(main):029:1*     foo(x); bar(y); baz(x); qux(x, y, z)
irb(main):030:0> }
foo(:honyarara)
bar(:fugafugafuga)
baz(:honyarara)
qux(:honyarara, :fugafugafuga, :hogehogehoge)
=> :done_qux
irb(main):031:0> p = proc {|x,y,z| foo(x); bar(y); baz(x); qux(x, y, z) } # procs handle args like blocks
=> #<Proc:0x00007f418f800ed0 (irb):31>
irb(main):032:0> [honyarara, fugafugafuga, hogehogehoge].then(&p)
foo(:honyarara)
bar(:fugafugafuga)
baz(:honyarara)
qux(:honyarara, :fugafugafuga, :hogehogehoge)
=> :done_qux
irb(main):033:0> lunary = -> a { x,y,z = a; foo(x); bar(y); baz(x); qux(x, y, z) }
# lambdas handle args like methods
=> #<Proc:0x00007f41900ff130 (irb):33 (lambda)>
irb(main):034:0> [honyarara, fugafugafuga, hogehogehoge].then(&lunary)
foo(:honyarara)
bar(:fugafugafuga)
baz(:honyarara)
qux(:honyarara, :fugafugafuga, :hogehogehoge)
=> :done_qux
irb(main):035:0> l3args = -> x, y, z { foo(x); bar(y); baz(x); qux(x, y, z) }
=> #<Proc:0x00007fb5a9025c40 (irb):35 (lambda)>
irb(main):036:0> [honyarara, fugafugafuga, hogehogehoge].then{l3args.(*_1)}
foo(:honyarara)
bar(:fugafugafuga)
baz(:honyarara)
qux(:honyarara, :fugafugafuga, :hogehogehoge)
=> :done_qux
irb(main):037:0> lwrapped = -> ((x, y, z)) { foo(x); bar(y); baz(x); qux(x, y, z) }
=> #<Proc:0x00007fb5a88402c8 (irb):37 (lambda)>
irb(main):038:0> [honyarara, fugafugafuga, hogehogehoge].then(&lwrapped)
foo(:honyarara)
bar(:fugafugafuga)
baz(:honyarara)
qux(:honyarara, :fugafugafuga, :hogehogehoge)
=> :done_qux
```

I most commonly do this when I'm iteratively constructing a one-shot data-munging query in irb/pry. Pipeline segments can output arrays which can be deconstructed by the next segment into multiple args. IMO, it's a great technique for the REPL, but positional parameters can get unwieldy fast. Except for when it's *very* simple and self-documenting, I prefer to refactor to something that's easier to read before committing or merging.

#4 - 05/10/2022 06:16 PM - zverok (Victor Shepelev)

As a (maybe useful) sidenote, if we'll try to think in "useful atomic constructs" instead of "making existing multi-purpose", Enumerator#with_object is *almost* what might help here:

```
3.then.with_object(4) { |x, y| p [x, y] }
# prints [3, 4] -- as we need, it passes both to the block!
#=> 4 -- but returns the object, not the block's result
```

So, to "return the result", we need to go a long way:

```
3.then.with_object(4).map { |x, y| x + y }.first
# => 7
# ...and with several objects, it becomes even more cumbersome:
3.then.with_object(4).with_object(5).map { |(x, y), z| x + y + z }.first
```

I *feel* like some good atomic solution might be here somewhere, though :)

#5 - 02/17/2023 08:42 AM - rubyFeedback (robert heiler)

I do not have any strong opinions either way, but Benoit wrote:

> The last example is just:
>
> ```
> p.call(honyarara, fugafugafuga, hogehogehoge)
> ```
>
> isn't it? And that's a lot more readable IMHO.

Versus:

```
honyarara.then(fugafugafuga, hogehogehoge, &p)
```

And I am not sure the .call() is per se more readable.

I agree about the trailing &p part; that one looks a
bit weird. I guess it is just the block. But ignoring
this, if it is merely between .call versus .then,
then I think .then may be quite explicit and perhaps
"more readable", whatever that means. So I am not sure
that .call() is implicitely more readable than .then().

Although, I think one problem is that some ruby authors
write code like:

```
if condition then
  do_something
end
```

Or something like that. I never used that style, but some
folks used that style in the past. So perhaps it's not
quite so readable if we include the totality of the syntax
out there.

zverok wrote:

```
3.then.with_object(4).with_object(5).map { |(x, y), z| x + y + z }.first
```

Is this still ruby though? :P