Ruby master - Feature #15485

Refactor String#split

12/29/2018 11:48 AM - zverok (Victor Shepelev)

Status:	Rejected
Priority:	Normal
Assignee:	
Target version:	

Description

In #4780, new "block form" of #split was introduced. It behaves this way:

```
"several\nlong\nlines".split("\n") { |part| puts part if part.start_with?('l') }
# prints:
#   long
#   lines
# => "several\nlong\nlines"
```

Justification is stated as: "If the string is very long, and I only need to play with the split string one by one, this will not create a useless expensive array."

I understand the justification, but strongly believe that **implementation is unfortunate**. In the current implementation, the only way to "play with the split string one by one" is side-effect-full, like this:

```
result = []
lines.split("\n") { |ln| result << ln if ln.match?(PATTERN) }
```

This is very unidiomatic and unlike most of other methods that accept both block and no-block forms (it is understandable as original ticket is 7 years old, community practices were pretty different back then).

Our typical modern solution of the same problem is **enumerators**.

I propose redefining method as following:

```
lines.split("\n") # => Array, calculated immediately
lines.split("\n", enumerator: true) # => Enumerator, yielding split results one by one
```

It will allow all kind of idiomatic processing without any intermediate Array creation, like:

```
lines.split("\n", enumerator: true).take_while { |ln| ln == '__END__' }
lines.split("\n", enumerator: true).grep(PATTERN)
# ...and so on...
```

One more thing to note, that this call-sequence underlines "just an optimization" nature of the change: When you have "too large string" to process, you just add enumerator: true to your code without changing anything else.

PS: We can't change split to return enumerator **always**, because it would break a lot of sane code like lines.split("\n").join("\r\n")

History

#1 - 12/29/2018 12:36 PM - shevegen (Robert A. Heiler)

I think it is not good to assume that it is unidiomatic per se, at the least not the code that was shown, e. g.:

```
lines.split("\n") { |ln| result << ln if ln.match?(PATTERN) }
```

I myself would possibly not write code like this, but I am not sure if we can say that it is unidiomatic.
I more feel as if several other proposals in the last some months or so are unidiomatic. ;)

But I think styles are different so I would suggest to not assume one style to be less idiomatic than another style.

I am not a big fan of the proposal such as:

```
lines.split("\n", enumerator: true).take_while { |ln| ln == '__END__' }
```

```
lines.split("\n", enumerator: true).grep(PATTERN)
```

But I also do not feel that strongly against it either. It's just weird to me that the other feature would be called unidiomatic, but this suggestion be ... more idiomatic. Seems strange to me.

By the way, I agree about the comment that you made with "One more thing to note, that this call-sequence underlines "just an optimization" nature of the change", which I think was the original impetus for the change (and thus I am not a huge fan of it either, since a lot of optimization-related changes have other problems, such as the syntax being more clumsy in order to express an idea via code), but I disagree with "enumerator: true" too, just to avoid using the block form.

I think the best variant is **.split("\n")** by far, everything else is worse. :) Granted, this may obviously not be the fastest, but if I look at the syntax used both here and in suggestion #4780 then it really beats both alternatives.

I guess one big difference is the **block form**. In your examples you avoid the block for .split (you use it for .take_while). If we ignore the syntax part, and just focus on the block argument to .split, then I think it is ok to have the block form of split be used as suggested in #4780, even if I think #4780 is not extremely elegant (I prefer the term elegant to idiomatic, since ruby allows for a lot of different code style e. g. method calls with or without (), omitting return or using it, etc..).

It should also be noted down that ruby has no problem with "side-effects" anywhere. Ruby is not Haskell; several suggestions seem to come from a "functional" focus on programming and I am not sure if Ruby is the ideal functional-programming style language, even if it is multi-paradigm. :) (Style is only one matter though; another is syntax, and when it comes to syntax, I feel that there have been several proposals in the last few years that are not good, many of which were inspired by "functional aspects".)

#2 - 12/30/2018 10:52 AM - mame (Yusuke Endoh)

zverok (Victor Shepelev) wrote:

```
result = []
lines.split("\n") { |ln| result << ln if ln.match?(PATTERN) }
```

> This is very unidiomatic and unlike most of other methods that accept both block and no-block forms

(Personally I like this style because it is very clear and explicit. Anyway.)

There is Object#to_enum for your use case.

```
lines.to_enum(:split, "\n").select {|ln| ln.match?(PATTERN) }
```

Object#to_enum is applicable to all methods that accept a block. This is a more general solution than adding enumerator: keyword to individual methods.

#3 - 12/30/2018 12:21 PM - zverok (Victor Shepelev)

> Personally I like this style because it is very clear and explicit. Anyway.

Well, on its own as just 2 code lines it probably is.
But any realistic use-case I can see of is like, for example "take all lines before empty line (extract header)", and it will became like:

```
# old split:
def extract_header(body)
  body.split("\n")
      .take_while { |ln| !ln.empty? }
      .map { |ln| ln.split(': ', 2) } # ....and so on
end

# new split:
def extract_header(body)
  header = []
  body.split("\n") { |ln|
    break if ln.empty?
    header << ln
  }
  header.map { |ln| ln.split(': ', 2) }
end
```

```
# my proposal (or, of course, #to_enum)
def extract_header(body)
  body.split("\n", enumertor: true)
      .take_while { |ln| !ln.empty? }
      .map { |ln| ln.split(': ', 2) }
end
```

I understand not everybody is fond of "functional-first"/"chaining-first" approach, but it seems that Ruby's evolution is clearly heading this way, so recent introduction of the method that "just yields" seems a bit off-the-track to me.

> There is Object#to_enum for your use case.

Yeah, of course, that's nice! Though seems a bit like a "fix" for unfortunate API.
Though looking from another angle, it could be treated also as a natural thing to do: implement minimal "yielding" in a method, and use to_enum for more complicated cases...

#4 - 12/30/2018 05:52 PM - marcandre (Marc-Andre Lafortune)

As mame (Yusuke Endoh) said, that's what to_enum is for.

There is no way we are going to modify all forms not returning Enumerators, when there's already a good solution (that's shorter than the proposition!).

This request should be closed.

#5 - 12/30/2018 06:16 PM - zverok (Victor Shepelev)

OK, after a bit of thinking, this makes perfect sense to me.
Please close, and sorry for misreporting.

#6 - 12/31/2018 03:48 AM - marcandre (Marc-Andre Lafortune)

- Status changed from Open to Rejected