The Magic of `+""` and `-""` in Ruby

Demystifying Unary String Operators for Performance and Safety

---

Ruby is renowned for its developer happiness and elegant syntax. It's a language where common tasks often read like natural English. However, beneath this friendly surface lie powerful, slightly esoteric features designed for fine-grained control and performance optimization. One such feature—often puzzling to newcomers and occasionally overlooked by seasoned developers—is the use of unary operators on strings: specifically, `+""` and `-""`.

If you've ever dug into the source code of popular Ruby gems like Rails or sidekiq, you might have stumbled across a line like this and paused:

```
buffer = +""
```

What exactly is happening here? Why not just write `buffer = ""`? Let's dive into the mechanics, the advantages, and why this tiny symbol makes a significant difference.

The Problem: The Frozen String Literal Pragma

To understand `+""`, we first have to understand a major shift in Ruby's approach to memory management.

Historically, every time you declared a string literal in Ruby, a new object was created in memory. If you had a loop that printed `"hello"` 1,000 times, Ruby instantiated 1,000 distinct string objects, creating work for the garbage collector.

To combat this, Ruby 2.3 introduced the frozen string literal pragma:

```
# frozen_string_literal: true
```

When placed at the top of a file, this magic comment instructs Ruby to freeze all string literals in that file. A frozen string cannot be modified. They become constants in memory, drastically reducing object allocations. This is considered a best practice in modern Ruby development.

However, this introduces a new problem. What if you *want* to build a string dynamically using append operations (`<<`)?

```
# frozen_string_literal: true

buffer = ""
buffer << "Hello" # => FrozenError (can't modify frozen String)
```

The Solution: The Unary Plus (`+""`)

Enter the unary `+` operator. Introduced in Ruby 2.3 alongside the frozen string pragma, `+` explicitly unfreezes a string literal, returning a mutable copy.

```
# frozen_string_literal: true

buffer = +""
buffer << "Hello "
buffer << "World"
puts buffer # => "Hello World"
```

> **In short:** `+""` says to Ruby, "I know frozen strings are enabled here, but I specifically need this particular string to be mutable because I plan to change it."

Why is this better than `String.new`?

You could achieve the same result using `String.new`.

```
buffer = String.new
```

Functionally, `+""` and `String.new` achieve the same goal. However, `+""` is generally preferred in the Ruby community for a few reasons:

- **Brevity:** It's significantly shorter and reads more like a literal assignment.
- **Idiomatic:** It has become the recognized standard idiom in modern Ruby libraries.
- **Performance (Micro-optimization):** Historically, evaluating the literal `+""` was marginally faster than the method dispatch required for `String.new`, although modern Ruby versions have largely leveled this playing field.

The Counterpart: The Unary Minus (`-""`)

If `+` unfreezes a string, what does `-` do? The unary minus does the opposite: it guarantees a string is frozen and deduplicated.

```
# frozen_string_literal: false (or omitted)

str1 = -"immutable"
str2 = -"immutable"

puts str1.object_id == str2.object_id # => true
str1 << " change" # => FrozenError (can't modify frozen String)
```

When you use `-`, Ruby checks an internal "frozestring" table. If a frozen string with the identical content already exists, it returns a reference to that existing object rather than creating a new one. This is equivalent to calling `"immutable".freeze`, but it is syntactically cleaner when used inline.

Why Do Developers Miss This?

If these operators are so useful, why aren't they universally understood?

1. **It's visually subtle:** The difference between `""` and `+""` is a single character. It's easy for the eyes to glide over it during code review or while casually reading a library's source code.
2. **It relies on file-level pragmas:** If you aren't in the habit of using `# frozen_string_literal: true` in your projects, you rarely encounter the `FrozenError` that necessitates `+""`. Many smaller scripts or older legacy applications run without the pragma, meaning a regular `""` works fine as a mutable buffer.
3. **It feels "un-Ruby-like":** Ruby is usually explicit and readable (e.g., `[1, 2].empty?`). Using arithmetic operators like `+` and `-` on strings to control memory allocation feels a bit like C-style pointer manipulation, which breaks the mental model some developers have of the language.

Best Practices & Takeaways

To write modern, performant, and safe Ruby code, adopt these habits:

- **Always freeze by default:** Add `# frozen_string_literal: true` to the top of all new Ruby files. It's an easy win for memory efficiency.
- **Use `+""` for buffers:** When you need to incrementally build a string using `<<`, initialize it with `+""`.
- **Avoid `+=` in loops:** Building strings with `+=` creates a new object on every iteration, regardless of pragmas. Always prefer appending to a mutable buffer with `<<`.

```
# BAD (Creates 1001 string objects)
# frozen_string_literal: true
result = +""
1000.times { result += "a" }

# GOOD (Creates 1 mutable string object and modifies it in place)
# frozen_string_literal: true
result = +""
1000.times { result << "a" }
```

The unary operators `+` and `-` on strings are small, esoteric features that pack a significant punch. Understanding them not only helps you write better code but also enables you to read and understand the source code of the Ruby ecosystem's most robust libraries.