Underrated Ruby & Rails Features

Unlock hidden gems in Ruby and Rails! This cheat sheet explores powerful, yet often overlooked, methods and features that can elevate your code quality, performance, and developer productivity, especially for experienced Rails developers.

Core Ruby Underrated Methods

Enumerable & Array Power-Ups

`Enumerable#each_with_object(memo)`	Iterates over an enumerable, yielding each element and a memo object. Useful for building complex data structures.
	`[1, 2, 3].each_with_object({}) { |i, h| h[i] = i * 2 }`
	`# => {1=>2, 2=>4, 3=>6}`

`Enumerable#chunk { |element| ... }`	Splits the enumerable into chunks based on the return value of the block. Adjacent elements returning the same value are grouped.
	`[1, 2, 4, 5, 6, 8, 9].chunk { |x| x.even? }.to_a`
	`# => [[false, [1]], [true, [2, 4, 6, 8]], [false, [9]]]`

`Array#bsearch { |x| ... }`	Binary search on a sorted array. Returns an element or its index based on the block's boolean or numeric result. Fast for large arrays.
	`a = [1, 4, 8, 10, 11, 15]`
	`a.bsearch { |x| x >= 8 } # => 8 (finds first element >= 8)`
	`a.bsearch { |x| x > 8 }  # => 10 (finds first element > 8)`

`Enumerable#slice_before { |e| ... }`	Creates an enumerator that iterates over slices of the original enumerable. A new slice starts right before an element for which the block returns true.
	`(1..10).slice_before { |i| i % 3 == 0 }.to_a`
	`# => [[1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]`

`Enumerable#slice_after { |e| ... }`	Creates an enumerator over slices. A new slice starts right after an element for which the block returns true.
	`(1..10).slice_after { |i| i % 3 == 0 }.to_a`
	`# => [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10]]`

`Enumerable#tally` (Ruby 2.7+)	Counts the occurrences of each element and returns a hash. Simpler than `group_by.transform_values(&:count)`.
	`['a', 'b', 'a', 'c', 'b', 'a'].tally`
	`# => {"a"=>3, "b"=>2, "c"=>1}`

Object & Module Tricks

`Object#tap { |obj| ... }`	Yields the object to the block and returns the object itself. Useful for chaining methods while performing side effects or debugging.
	`[1, 2, 3].tap { |a| puts "Original: #{a.inspect}" }.map { |x| x * 2 }`
	`# Original: [1, 2, 3]`
	`# => [2, 4, 6]`

`Object#yield_self { |obj| ... }` (or `#then`)	Yields the object to the block and returns the *result* of the block. Useful for breaking down method chains or passing the result of one operation as an argument to another.
	`"hello".yield_self { |s| s.capitalize }.yield_self { |s| "Greeting: #{s}" }`
	`# => "Greeting: Hello"`

`Object#public_send(method_name, *args, &block)`	Invokes the method identified by `method_name` on the object, but *only* if it's a public method. Safer than `send` when method names come from external sources.
	`class MyClass`
	`  def public_method; 'public'; end`
	`  private def private_method; 'private'; end`
	`end`
	
	`obj = MyClass.new`
	`obj.public_send(:public_method) # => "public"`
	`# obj.public_send(:private_method) # => NoMethodError`

`Module#delegate(*methods, to:)`	Provides a shortcut to define methods that forward to another object. Reduces boilerplate.
	`class User`
	`  def profile; @profile ||= Profile.new; end`
	`  delegate :address, :phone, to: :profile`
	`end`
	
	`user = User.new`
	`# user.address now calls user.profile.address`

`Kernel#binding`	Returns a `Binding` object representing the current execution context (variables, methods in scope). Essential for debugging (e.g., `binding.pry`) or templating.
	`x = 10`
	`b = binding`
	`eval("x * 2", b) # => 20`

`Kernel#define_method(name, &block)`	Dynamically defines a method with the given name and block. Useful for metaprogramming.
	`class MyClass`
	`  define_method(:my_dynamic_method) { |arg| "Hello, #{arg}!" }`
	`end`
	
	`MyClass.new.my_dynamic_method("World") # => "Hello, World!"`

String & Numeric Helpers

`String#undump`	Reverses the effect of `String#dump`. Converts escape sequences back into their characters. Useful for parsing Ruby strings.
	`"\"hello\nworld\"".undump # => ""hello\nworld""`
	`"\u{20AC}".undump # => "€"`

`String#chomp!` (with argument)	Removes a record separator (like newline) from the end of the string, *in place*. Can take an argument to specify the separator.
	`s = "hello\n"`
	`s.chomp!  # s is now "hello"`
	
	`s = "hello\r\n"`
	`s.chomp!("\r\n") # s is now "hello"`

`Numeric#with_indefinite_article`	Adds 'a' or 'an' prefix based on the number. Part of Active Support's `Inflector`.
	`5.with_indefinite_article # => "a 5"`
	`8.with_indefinite_article # => "an 8"`

`String#indent(amount, indent_string = ' ')`	Indents the string by the specified amount using the given indent string. Preserves existing newlines.
	`"hello\nworld".indent(2)`
	`# => "  hello\n  world"`
	
	`"- item 1\n- item 2".indent(4, '*')`
	`# => "****- item 1\n****- item 2"`

`String#squish`	Removes leading and trailing whitespace and replaces internal sequences of whitespace with a single space. Part of Active Support. `"  hello   world\n".squish # => "hello world"`

`String#truncate_words(words_count, options = {})`	Truncates a string after a certain number of words. More semantically useful than truncating characters.
	`"Long sentence to be truncated".truncate_words(3, separator: '...')`
	`# => "Long sentence to..."`

ActiveRecord Deep Dive

Relation & Query Magic

`ActiveRecord::Relation#merge(other_relation)`	Merges two relations. Useful for combining dynamic scopes.
	`# In a controller or service:`
	`users = User.active`
	`users = users.admin if current_user.admin?`
	`users = users.where(country: params[:country]) if params[:country].present?`
	
	`# This is equivalent to:`
	`users = User.active.merge(User.admin if`
	`current_user.admin?).merge(User.where(country: params[:country]) if`
	`params[:country].present?).compact`

`ActiveRecord::Relation#annotate`	Adds comments to the SQL query, useful for debugging and profiling tools to identify where queries originate.
	`User.where(active: true).annotate("Fetching active users for report").to_a`
	`# SELECT /* Fetching active users for report */ users.* FROM users WHERE`
	`users.active = TRUE`

`ActiveRecord::Relation#find_each` / `#find_in_batches`	Iterates over records in batches, processing them one by one (`find_each`) or in groups (`find_in_batches`). Prevents loading the entire dataset into memory for large tables.
	`User.find_each { |user| user.process! }`
	`User.find_in_batches(batch_size: 1000) do |batch|`
	`  batch.each(&:process!)`
	`end`

`ActiveRecord::Relation#pluck(*column_names)` / `#pick(*column_names)`	`pluck` returns an array of values for the specified column(s). `pick` returns the value of the first column for the first record *only*. Both bypass loading full objects.
	`User.where(active: true).pluck(:email)`
	`# => ["a@b.com", "c@d.com", ...]`
	`User.where(active: true).limit(1).pick(:email)`
	`# => "a@b.com"`

`ActiveRecord::Relation#except(*skips)` / `#only(*takes)`	Allows removing or keeping specific query clauses (`where`, `order`, `limit`, `offset`, `select`, etc.) from a relation. Useful when modifying a predefined scope.
	`User.active.order(:created_at).limit(10).except(:order, :limit).to_a`
	`# Removes order and limit from the 'active' scope`

`ActiveRecord::Relation#unscope(*args)`	Removes conditions or scopes applied earlier in the chain, including default scopes. `# Assuming a default scope `default_scope { where(deleted: false) }` `User.unscope(where: :deleted).where(deleted: true).to_a` `# Finds deleted users, ignoring the default scope`

Model & Class Methods

`ActiveRecord::Base.suppress(*exceptions)`	Suppresses specified exceptions within the block. Returns nil if an exception is suppressed, otherwise the result of the block.
	`ActiveRecord::Base.suppress(ActiveRecord::RecordNotFound) do`
	`  User.find(params[:id])`
	`end`
	`# Returns nil if record not found, otherwise the user object`

`ActiveRecord::Base.transaction(options = {}) { ... }` (with isolation)	Executes block within a database transaction. Can specify isolation level (`:read_committed`, `:repeatable_read`, `:serializable`) for stronger data consistency guarantees.
	`ActiveRecord::Base.transaction(isolation: :serializable) do`
	`  # Complex operations requiring strict consistency`
	`end`

`ActiveRecord::Base.upsert_all(hashes, options = {})`	Inserts multiple records, updating them if a conflict arises (based on unique index). Highly performant for bulk insert/update.
	`User.upsert_all([`
	`  { email: 'a@b.com', name: 'Alice' },`
	`  { email: 'c@d.com', name: 'Bob' }`
	`], unique_by: :email)`

`ActiveRecord::Base.insert_all(hashes, options = {})`	Inserts multiple records in a single SQL statement. Much faster than creating individual records. Bypasses validations and callbacks.
	`User.insert_all([`
	`  { email: 'a@b.com', name: 'Alice' },`
	`  { email: 'c@d.com', name: 'Bob' }`
	`])`

`ActiveRecord::Base.silence_sql { ... }`	Suppresses SQL logging within the block. Useful for noisy operations during tests or specific rake tasks.
	`ActiveRecord::Base.silence_sql do`
	`  User.count # No SQL printed to console/logs for this line`
	`end`

`ActiveRecord::Type::Value`	Base class for defining custom attribute types. Allows casting, serializing, and deserializing complex data to/from database columns.
	`# app/types/my_array_type.rb`
	`class MyArrayType < ActiveRecord::Type::Value`
	`  def cast(value); Array(value).compact; end`
	`  def serialize(value); Array(value).compact.to_json; end`
	`  def deserialize(value); JSON.parse(value.to_s); rescue; []; end`
	`end`
	
	`# In model:`
	`attribute :my_column, MyArrayType.new`

Data Handling & Efficiency

`ActiveRecord::Relation#strict_loading!`	Enforces N+1 prevention. Any attempt to lazy-load an association on records from this relation will raise an error.
	`# In a scope or controller action:`
	`User.includes(:profile).strict_loading!`
	`# User.first.posts # => Would raise ActiveRecord::StrictLoadingViolationError`

`ActiveRecord::Relation#in_batches(of: batch_size) { |relation| ... }`	Iterates over batches of records. Provides a relation object for each batch, allowing further filtering/operations on the batch.
	`User.active.in_batches(of: 500).each do |user_batch|`
	`  user_batch.update_all(status: 'processed')`
	`end`

`ActiveRecord::Base.cache_key(timestamp, ids, column)`	Generates a cache key based on the latest update timestamp, a list of IDs, and an optional column. Useful for caching collections.
	`# In a view fragment cache:`
	`cache cache_key([@project, @users], :updated_at) do`
	`  # Render user list`
	`end`

`attribute :name, :jsonb, default: {}`	Directly defines attributes backed by database types like `jsonb` or `hstore` as standard model attributes. Allows easy access and modification.
	`# In a model:`
	`attribute :settings, :jsonb, default: {}`
	
	`user = User.new`
	`user.settings['theme'] = 'dark'`
	`user.save`
	`# SELECT ... '{"theme":"dark"}' ...`

`ActiveRecord::Base#becomes(klass)`	Returns a new instance of `klass` with the same ID and attributes as the original object. Useful for single-table inheritance or dynamic object types.
	`# Assuming Post and Article are subclasses of Content (STI)`
	`content = Content.find(1)`
	`article = content.becomes(Article)`
	`article.is_a?(Article) # => true`

Rails Beyond MVC Basics

Controller & View Rendering

`ActionController::Renderer`	Allows rendering views or templates outside of a controller context (e.g., in background jobs, mailers, console). Great for generating HTML snippets asynchronously.
	`# In a job or service object:`
	`html = ApplicationController.render(template: 'users/show', assigns: { user: @user }, layout: false)`
	`# Send 'html' via email or websocket`

`ActionView::Helpers::TagHelper#tag.div`	A helper to build HTML tags more programmatically and safely than concatenating strings. Avoids raw HTML in helpers.
	`# Instead of: "<div class=\"#{css_class}\">#{content}</div>".html_safe`
	`tag.div(content, class: css_class)`

`render plain: '...', status: :ok`	Allows rendering plain text responses easily, useful for simple API endpoints or health checks, bypassing template lookup.
	`def health_check`
	`  render plain: 'OK', status: :ok`
	`end`

`before_action :require_login!, unless: :skip_login?`	Using `unless` and `if` options on callbacks for conditional execution. Allows centralizing logic.
	`# In ApplicationController:`
	`class ApplicationController < ActionController::Base`
	`  before_action :authorize!`
	
	`  def authorize!`
	`  # ... authorization logic ...`
	`  end`
	
	`  def skip_authorization? # Can be overridden in subclasses`
	`  false`
	`  end`
	`end`
	
	`# In specific controller:`
	`class PublicController < ApplicationController`
	`  skip_before_action :authorize!, only: [:index]`
	`  def skip_authorization?; true; end # Override to skip for the whole controller`
	`end`

`ActionView::Helpers::FormHelper#form_with(model: @user, local: true)` / `remote: true`	`form_with` defaults to AJAX (`remote: true`). Explicitly setting `local: true` forces a standard browser form submission. Understanding this default is key.
	`<%= form_with model: @user, local: true do |form| %>`
	`  <%# Standard form submission %>`
	`<% end %>`
	
	`<%= form_with model: @user, remote: true do |form| %>`
	`  <%# AJAX submission via Rails UJS %>`
	`<% end %>`

`respond_to do |format| ... end`	Though not strictly 'underrated', mastering its use for multiple formats (HTML, JSON, XML, JS, PDF, etc.) in complex actions is crucial for robust APIs and flexible controllers.
	`def index`
	`  @users = User.all`
	`  respond_to do |format|`
	`  format.html # Renders index.html.erb`
	`  format.json { render json: @users }`
	`  format.csv  { send_data @users.to_csv, filename: "users-#{Date.today}.csv" }`
	`  end`
	`end`

Configuration & Utility Helpers

`Rails.configuration.x`	A namespace for application-specific configuration settings. Defined in environment files (`config/environments/*.rb`). Preferable to global constants.
	`# config/environments/production.rb`
	`Rails.application.configure do`
	`  config.x.payment_gateway_url = "https://prod.gateway.com"`
	`  config.x.feature_flags = config_for(:features)`
	`end`
	
	`# Anywhere in app:`
	`Rails.configuration.x.payment_gateway_url`
	`Rails.configuration.x.feature_flags[:new_dashboard]`

`config_for(:filename)`	Loads a YAML file from `config/`, rendering ERB and selecting the configuration for the current Rails environment. Useful for service credentials, feature flags, etc.
	`# config/services.yml`
	`default:`
	`  api_key: <%= ENV['DEFAULT_API_KEY'] %>`
	`production:`
	`  endpoint: https://api.prod.com`
	
	`# In code:`
	`services_config = Rails.application.config_for(:services)`
	`services_config['api_key'] # => value from ENV or default`

`Rails.cache.fetch(key, options = {}) { ... }` (with options)	`fetch` is common, but explore options like `expires_in`, `race_ttl`, `compress`, `version`. `race_ttl` prevents stampeding herd problem.
	`Rails.cache.fetch("all_users", expires_in: 12.hours, race_ttl: 10.seconds) do`
	`  User.all.to_a # Recompute if cache is stale, but serve stale data for a bit`
	`end`

`delegate_missing_to(target)`	Defines `method_missing` to delegate any undefined methods to the specified target object. Use with caution, but can be powerful for wrappers/decorators.
	`class UserPresenter`
	`  delegate_missing_to :@user`
	
	`  def initialize(user); @user = user; end`
	
	`  def full_name`
	`  "#{@user.first_name} #{@user.last_name}"`
	`  end`
	`end`
	
	`presenter = UserPresenter.new(user)`
	`presenter.email # Delegates to @user.email`
	`presenter.full_name # Calls presenter's method`

`ActiveSupport::Concern`

Standard way to package shared functionality (methods, includes, callbacks) to be included in multiple classes. Improves code organization and avoids polluting the global namespace.

```ruby
# app/models/concerns/activable.rb
module Activable
  extend ActiveSupport::Concern

  included do
    scope :active, -> { where(active: true) }
    before_create :set_active_status
  end

  def activate!
    update!(active: true)
  end

  private
  def set_active_status
    self.active = true if self.active.nil?
  end
end

# In model:
class User < ApplicationRecord
  include Activable
end
```

`rake notes`

Searches code for comments like `TODO`, `FIXME`, `OPTIMIZE`, `DEPRECATED` (and custom ones via `config.annotations.register_tags`). Useful for tracking technical debt.

```ruby
# Add a custom tag in config/application.rb
config.annotations.register_tags("REVIEW")

# In code:
# REVIEW: This logic seems complex, simplify if possible

# Run:
rake notes
rake notes:review
```

`ActionMailer::MailDeliveryJob`

The default job class used when you call `.deliver_later` on a mailer. You can customize it or subclass it to add specific behaviors (e.g., retry logic, error handling, recipient filtering).

```ruby
# In config/application.rb
config.action_mailer.delivery_job = CustomMailDeliveryJob

# app/jobs/custom_mail_delivery_job.rb
class CustomMailDeliveryJob < ActionMailer::MailDeliveryJob
  queue_as :critical_mail

  def perform(mailer, method, format, delivery_method, *, **)
    # Add pre-delivery checks or logging
    super # Call original perform
    # Add post-delivery logging
  end
end
```

`ActiveJob::Base.discard_on` / `retry_on`

Built-in mechanisms for handling job failures. `discard_on` immediately discards the job on specific exceptions; `retry_on` retries the job with configurable delays.

```ruby
class MyJob < ApplicationJob
  # Discard job if a specific error occurs
  discard_on ActiveJob::DeserializationError

  # Retry job up to 3 times on network errors with escalating delays
  retry_on Net::ReadTimeout, wait: :exponentially_longer, attempts: 3

  def perform(record)
    # ... job logic ...
  end
end
```

`perform_later(record)` vs `perform_later(record.id)`

Passing ActiveRecord objects directly to `perform_later` automatically handles serialization/deserialization. Pass the ID if the record might be deleted or changed significantly before the job runs.

```ruby
# Safer if record might be deleted:
MyJob.perform_later(user.id)

# Simpler if record is expected to exist and not change drastically:
MyJob.perform_later(user)
# Inside job, user will be deserialized automatically
```

`queue_as :priority`

Specifies the queue name for a job. Allows prioritizing or segregating jobs based on importance or resource requirements.

```ruby
class CriticalJob < ApplicationJob
  queue_as :critical
  # ...
end

class LowPriorityJob < ApplicationJob
  queue_as :low
  # ...
end
```

`ActionMailer::Preview`	Allows developers to preview emails in the browser without sending them. Configure previews in `test/mailers/previews`.
	`# test/mailers/previews/user_mailer_preview.rb`
	`class UserMailerPreview < ActionMailer::Preview`
	`  def welcome`
	`  UserMailer.welcome(User.first)`
	`  end`
	`  def password_reset`
	`  UserMailer.password_reset(User.last, "faketoken")`
	`  end`
	`end`
	`# Access at http://localhost:3000/rails/mailers`

`config.action_mailer.delivery_method = :test`	Configures mailers to store emails in `ActionMailer::Base.deliveries` array instead of sending them. Indispensable for testing email functionality.
	`# In test environment config:`
	`config.action_mailer.delivery_method = :test`
	`config.action_mailer.perform_deliveries = true # Ensure delivery is attempted`
	`# In test:`
	`email = UserMailer.welcome(@user).deliver_now`
	`assert_equal 1, ActionMailer::Base.deliveries.size`
	`assert_equal "Welcome!", email.subject`