

DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON <https://guides.rubyonrails.org>.

## Active Support Core Extensions

Active Support is the Ruby on Rails component responsible for providing Ruby language extensions and utilities.

It offers a richer bottom-line at the language level, targeted both at the development of Rails applications, and at the development of Ruby on Rails itself.

After reading this guide, you will know:

- What Core Extensions are.
- How to load all extensions.
- How to cherry-pick just the extensions you want.
- What extensions Active Support provides.

---

## How to Load Core Extensions

### Stand-Alone Active Support

In order to have the smallest default footprint possible, Active Support loads the minimum dependencies by default. It is broken in small pieces so that only the desired extensions can be loaded. It also has some convenience entry points to load related extensions in one shot, even everything.

Thus, after a simple require like:

```
require "active_support"
```

only the extensions required by the Active Support framework are loaded.

### Cherry-picking a Definition

This example shows how to load [Hash#with\\_indifferent\\_access](#). This extension enables the conversion of a Hash into an [ActiveSupport::HashWithIndifferentAccess](#) which permits access to the keys as either strings or symbols.

```
{a: 1}.with_indifferent_access["a"] # => 1
```

For every single method defined as a core extension this guide has a note that says where such a method is defined. In the case of `with_indifferent_access` the note reads:

NOTE: Defined in `active_support/core_ext/hash/indifferent_access.rb`.

That means that you can require it like this:

```
require "active_support"
require "active_support/core_ext/hash/indifferent_access"
```

Active Support has been carefully revised so that cherry-picking a file loads only strictly needed dependencies, if any.

### Loading Grouped Core Extensions

The next level is to simply load all extensions to `Hash`. As a rule of thumb, extensions to `SomeClass` are available in one shot by loading `active_support/core_ext/some_class`.

Thus, to load all extensions to `Hash` (including `with_indifferent_access`):

```
require "active_support"
require "active_support/core_ext/hash"
```

### Loading All Core Extensions

You may prefer just to load all core extensions, there is a file for that:

```
require "active_support"
require "active_support/core_ext"
```

### Loading All Active Support

And finally, if you want to have all Active Support available just issue:

```
require "active_support/all"
```

That does not even put the entire Active Support in memory upfront indeed, some stuff is configured via `autoload`, so it is only loaded if used.

## Active Support Within a Ruby on Rails Application

A Ruby on Rails application loads all Active Support unless `config.active_support.bare` is true. In that case, the application will only load what the framework itself cherry-picks for its own needs, and can still cherry-pick itself at any granularity level, as explained in the previous section.

## Extensions to All Objects

### `blank?` and `present?`

The following values are considered to be blank in a Rails application:

- `nil` and `false`,
- strings composed only of whitespace (see note below),
- empty arrays and hashes, and
- any other object that responds to `empty?` and is empty.

INFO: The predicate for strings uses the Unicode-aware character class `[:space:]`, so for example U+2029 (paragraph separator) is considered to be whitespace.

WARNING: Note that numbers are not mentioned. In particular, 0 and 0.0 are **not** blank.

For example, this method from `ActionController::HttpAuthentication::Token::ControllerMethods` uses `blank?` for checking whether a token is present:

```
def authenticate(controller, &login_procedure)
  token, options = token_and_options(controller.request)
  unless token.blank?
    login_procedure.call(token, options)
  end
end
```

The method `present?` is equivalent to `!blank?`. This example is taken from `ActionDispatch::Http::Cache::Response`:

```
def set_conditional_cache_control!
  return if self["Cache-Control"].present?
  # ...
end
```

NOTE: Defined in `active_support/core_ext/object/blank.rb`.

### presence

The `presence` method returns its receiver if `present?`, and `nil` otherwise. It is useful for idioms like this:

```
host = config[:host].presence || 'localhost'
```

NOTE: Defined in `active_support/core_ext/object/blank.rb`.

### duplicable?

As of Ruby 2.5, most objects can be duplicated via `dup` or `clone`:

```
"foo".dup          # => "foo"
"".dup             # => ""
Rational(1).dup    # => (1/1)
Complex(0).dup     # => (0+0i)
1.method(:+).dup   # => TypeError (allocator undefined for Method)
```

Active Support provides `duplicable?` to query an object about this:

```
"foo".duplicable?  # => true
"".duplicable?     # => true
Rational(1).duplicable? # => true
Complex(1).duplicable? # => true
1.method(:+).duplicable? # => false
```

WARNING: Any class can disallow duplication by removing `dup` and `clone` or raising exceptions from them. Thus only `rescue` can tell whether a given arbitrary object is duplicable. `duplicable?` depends on the hard-coded list above, but it is much faster than `rescue`. Use it only if you know the hard-coded list is enough in your use case.

NOTE: Defined in `active_support/core_ext/object/duplicable.rb`.

## deep\_dup

The `deep_dup` method returns a deep copy of a given object. Normally, when you `dup` an object that contains other objects, Ruby does not `dup` them, so it creates a shallow copy of the object. If you have an array with a string, for example, it will look like this:

```
array      = ['string']
duplicate = array.dup

duplicate.push 'another-string'

# the object was duplicated, so the element was added only to the duplicate
array      # => ['string']
duplicate  # => ['string', 'another-string']

duplicate.first.gsub!('string', 'foo')

# first element was not duplicated, it will be changed in both arrays
array      # => ['foo']
duplicate  # => ['foo', 'another-string']
```

As you can see, after duplicating the `Array` instance, we got another object, therefore we can modify it and the original object will stay unchanged. This is not true for array's elements, however. Since `dup` does not make a deep copy, the string inside the array is still the same object.

If you need a deep copy of an object, you should use `deep_dup`. Here is an example:

```
array      = ['string']
duplicate = array.deep_dup

duplicate.first.gsub!('string', 'foo')

array      # => ['string']
duplicate  # => ['foo']
```

If the object is not duplicable, `deep_dup` will just return it:

```
number = 1
duplicate = number.deep_dup
number.object_id == duplicate.object_id  # => true
```

NOTE: Defined in `active_support/core_ext/object/deep_dup.rb`.

## try

When you want to call a method on an object only if it is not `nil`, the simplest way to achieve it is with conditional statements, adding unnecessary clutter. The alternative is to use `try`. `try` is like `Object#public_send` except that it returns `nil` if sent to `nil`.

Here is an example:

```
# without try
unless @number.nil?
  @number.next
end

# with try
@number.try(:next)
```

Another example is this code from `ActiveRecord::ConnectionAdapters::AbstractAdapter` where `@logger` could be `nil`. You can see that the code uses `try` and avoids an unnecessary check.

```
def log_info(sql, name, ms)
  if @logger.try(:debug?)
    name = '%s (%.1fms)' % [name || 'SQL', ms]
    @logger.debug(format_log_entry(name, sql.squeeze(' ')))
  end
end
```

`try` can also be called without arguments but a block, which will only be executed if the object is not nil:

```
@person.try { |p| "#{p.first_name} #{p.last_name}" }
```

Note that `try` will swallow no-method errors, returning nil instead. If you want to protect against typos, use `try!` instead:

```
@number.try(:nest) # => nil
@number.try!(:nest) # NoMethodError: undefined method `nest' for 1:Integer
```

NOTE: Defined in `active_support/core_ext/object/try.rb`.

### `class_eval(*args, &block)`

You can evaluate code in the context of any object's singleton class using `class_eval`:

```
class Proc
  def bind(object)
    block, time = self, Time.current
    object.class_eval do
      method_name = "__bind_#{time.to_i}_#{time.usec}"
      define_method(method_name, &block)
      method = instance_method(method_name)
      remove_method(method_name)
      method
    end.bind(object)
  end
end
```

NOTE: Defined in `active_support/core_ext/kernel/singleton_class.rb`.

### `acts_like?(duck)`

The method `acts_like?` provides a way to check whether some class acts like some other class based on a simple convention: a class that provides the same interface as `String` defines

```
def acts_like_string?  
end
```

which is only a marker, its body or return value are irrelevant. Then, client code can query for duck-type-safeness this way:

```
some_class.acts_like?(:string)
```

Rails has classes that act like `Date` or `Time` and follow this contract.

NOTE: Defined in `active_support/core_ext/object/acts_like.rb`.

### `to_param`

All objects in Rails respond to the method `to_param`, which is meant to return something that represents them as values in a query string, or as URL fragments.

By default `to_param` just calls `to_s`:

```
7.to_param # => "7"
```

The return value of `to_param` should **not** be escaped:

```
"Tom & Jerry".to_param # => "Tom & Jerry"
```

Several classes in Rails overwrite this method.

For example `nil`, `true`, and `false` return themselves. `Array#to_param` calls `to_param` on the elements and joins the result with `"/"`:

```
[0, true, String].to_param # => "0/true/String"
```

Notably, the Rails routing system calls `to_param` on models to get a value for the `:id` placeholder.

`ActiveRecord::Base#to_param` returns the `id` of a model, but you can redefine that method in your models.

For example, given

```
class User  
  def to_param  
    "#{id}-#{name.parameterize}"  
  end  
end
```

we get:

```
user_path(@user) # => "/users/357-john-smith"
```

WARNING. Controllers need to be aware of any redefinition of `to_param` because when a request like that comes in "357-john-smith" is the value of `params[:id]` .

NOTE: Defined in `active_support/core_ext/object/to_param.rb` .

### **to\_query**

The `to_query` method constructs a query string that associates a given `key` with the return value of `to_param` . For example, with the following `to_param` definition:

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

we get:

```
current_user.to_query('user') # => "user=357-john-smith"
```

This method escapes whatever is needed, both for the key and the value:

```
account.to_query('company[name]')
# => "company%5Bname%5D=Johnson+%26+Johnson"
```

so its output is ready to be used in a query string.

Arrays return the result of applying `to_query` to each element with `key[]` as key, and join the result with "&":

```
[3.4, -45.6].to_query('sample')
# => "sample%5B%5D=3.4&sample%5B%5D=-45.6"
```

Hashes also respond to `to_query` but with a different signature. If no argument is passed a call generates a sorted series of key/value assignments calling `to_query(key)` on its values. Then it joins the result with "&":

```
{c: 3, b: 2, a: 1}.to_query # => "a=1&b=2&c=3"
```

The method `Hash#to_query` accepts an optional namespace for the keys:

```
{id: 89, name: "John Smith".to_query('user')
# => "user%5Bid%5D=89&user%5Bname%5D=John+Smith"
```

NOTE: Defined in `active_support/core_ext/object/to_query.rb` .

### **with\_options**

The method `with_options` provides a way to factor out common options in a series of method calls.

Given a default options hash, `with_options` yields a proxy object to a block. Within the block, methods called on the proxy are forwarded to the receiver with their options merged. For example, you get rid of the duplication in:

```
class Account < ApplicationRecord
  has_many :customers, dependent: :destroy
  has_many :products,  dependent: :destroy
  has_many :invoices,  dependent: :destroy
  has_many :expenses,  dependent: :destroy
end
```

this way:

```
class Account < ApplicationRecord
  with_options dependent: :destroy do |assoc|
    assoc.has_many :customers
    assoc.has_many :products
    assoc.has_many :invoices
    assoc.has_many :expenses
  end
end
```

That idiom may convey *grouping* to the reader as well. For example, say you want to send a newsletter whose language depends on the user. Somewhere in the mailer you could group locale-dependent bits like this:

```
I18n.with_options locale: user.locale, scope: "newsletter" do |i18n|
  subject i18n.t :subject
  body    i18n.t :body, user_name: user.name
end
```

TIP: Since `with_options` forwards calls to its receiver they can be nested. Each nesting level will merge inherited defaults in addition to their own.

NOTE: Defined in `active_support/core_ext/object/with_options.rb`.

## JSON support

Active Support provides a better implementation of `to_json` than the `json` gem ordinarily provides for Ruby objects. This is because some classes, like `Hash` and `Process::Status` need special handling in order to provide a proper JSON representation.

NOTE: Defined in `active_support/core_ext/object/json.rb`.

## Instance Variables

Active Support provides several methods to ease access to instance variables.

### `instance_values`

The method `instance_values` returns a hash that maps instance variable names without "@" to their corresponding values. Keys are strings:



```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_values # => {"x" => 0, "y" => 1}
```

NOTE: Defined in `active_support/core_ext/object/instance_variables.rb`.

### `instance_variable_names`

The method `instance_variable_names` returns an array. Each name includes the "@" sign.

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_variable_names # => ["@x", "@y"]
```

NOTE: Defined in `active_support/core_ext/object/instance_variables.rb`.

## Silencing Warnings and Exceptions

The methods `silence_warnings` and `enable_warnings` change the value of `$VERBOSE` accordingly for the duration of their block, and reset it afterwards:

```
silence_warnings { Object.const_set "RAILS_DEFAULT_LOGGER", logger }
```

Silencing exceptions is also possible with `suppress`. This method receives an arbitrary number of exception classes. If an exception is raised during the execution of the block and is `kind_of?` any of the arguments, `suppress` captures it and returns silently. Otherwise the exception is not captured:

```
# If the user is locked, the increment is lost, no big deal.
suppress(ActiveRecord::StaleObjectError) do
  current_user.increment! :visits
end
```

NOTE: Defined in `active_support/core_ext/kernel/reporting.rb`.

### `in?`

The predicate `in?` tests if an object is included in another object. An `ArgumentError` exception will be raised if the argument passed does not respond to `include?`.

Examples of `in?` :

```
1.in?([1,2])      # => true
"lo".in?("hello") # => true
25.in?(30..50)    # => false
1.in?(1)          # => ArgumentError
```

NOTE: Defined in `active_support/core_ext/object/inclusion.rb`.

## Extensions to Module

### Attributes

#### `alias_attribute`

Model attributes have a reader, a writer, and a predicate. You can alias a model attribute having the corresponding three methods all defined for you by using `alias_attribute`. As in other aliasing methods, the new name is the first argument, and the old name is the second (one mnemonic is that they go in the same order as if you did an assignment):

```
class User < ApplicationRecord
  # You can refer to the email column as "login".
  # This can be meaningful for authentication code.
  alias_attribute :login, :email
end
```

NOTE: Defined in `active_support/core_ext/module/aliasing.rb`.

### Internal Attributes

When you are defining an attribute in a class that is meant to be subclassed, name collisions are a risk. That's remarkably important for libraries.

Active Support defines the macros `attr_internal_reader`, `attr_internal_writer`, and `attr_internal_accessor`. They behave like their Ruby built-in `attr_*` counterparts, except they name the underlying instance variable in a way that makes collisions less likely.

The macro `attr_internal` is a synonym for `attr_internal_accessor`:

```
# library
class ThirdPartyLibrary::Crawler
  attr_internal :log_level
end

# client code
class MyCrawler < ThirdPartyLibrary::Crawler
  attr_accessor :log_level
end
```

In the previous example it could be the case that `:log_level` does not belong to the public interface of the library and it is only used for development. The client code, unaware of the potential conflict, subclasses and defines its own `:log_level`. Thanks to `attr_internal` there's no collision.

By default the internal instance variable is named with a leading underscore, `@_log_level` in the example above. That's configurable via `Module.attr_internal_naming_format` though, you can pass any `sprintf`-like format string with a leading `@` and a `%s` somewhere, which is where the name will be placed. The default is `"@_%s"`.

Rails uses internal attributes in a few spots, for examples for views:

```
module ActionView
  class Base
    attr_internal :captures
    attr_internal :request, :layout
    attr_internal :controller, :template
  end
end
```

NOTE: Defined in `active_support/core_ext/module/attr_internal.rb`.

### Module Attributes

The macros `mattr_reader`, `mattr_writer`, and `mattr_accessor` are the same as the `cattr_*` macros defined for class. In fact, the `cattr_*` macros are just aliases for the `mattr_*` macros. Check [Class Attributes](#).

For example, the API for the logger of Active Storage is generated with `mattr_accessor`:

```
module ActiveSupport
  mattr_accessor :logger
end
```

NOTE: Defined in `active_support/core_ext/module/attribute_accessors.rb`.

### Parents

`module_parent`

The `module_parent` method on a nested named module returns the module that contains its corresponding constant:

```
module X
  module Y
    module Z
    end
  end
end

M = X::Y::Z

X::Y::Z.module_parent # => X::Y
M.module_parent       # => X::Y
```

If the module is anonymous or belongs to the top-level, `module_parent` returns `Object`.

WARNING: Note that in that case `module_parent_name` returns `nil`.

NOTE: Defined in `active_support/core_ext/module/introspection.rb`.

### `module_parent_name`

The `module_parent_name` method on a nested named module returns the fully qualified name of the module that contains its corresponding constant:

```
module X
  module Y
    module Z
    end
  end
end

M = X::Y::Z

X::Y::Z.module_parent_name # => "X::Y"
M.module_parent_name       # => "X::Y"
```

For top-level or anonymous modules `module_parent_name` returns `nil`.

WARNING: Note that in that case `module_parent` returns `Object`.

NOTE: Defined in `active_support/core_ext/module/introspection.rb`.

### `module_parents`

The method `module_parents` calls `module_parent` on the receiver and upwards until `Object` is reached. The chain is returned in an array, from bottom to top:

```
module X
  module Y
    module Z
    end
  end
end

M = X::Y::Z

X::Y::Z.module_parents # => [X::Y, X, Object]
M.module_parents       # => [X::Y, X, Object]
```

NOTE: Defined in `active_support/core_ext/module/introspection.rb`.

## Anonymous

A module may or may not have a name:

```
module M
end

M.name # => "M"

N = Module.new
N.name # => "N"
```

```
Module.new.name # => nil
```

You can check whether a module has a name with the predicate [anonymous?](#) :

```
module M
end

M.anonymous? # => false

Module.new.anonymous? # => true
```

Note that being unreachable does not imply being anonymous:

```
module M
end

m = Object.send(:remove_const, :M)

m.anonymous? # => false
```

though an anonymous module is unreachable by definition.

NOTE: Defined in `active_support/core_ext/module/anonymous.rb`.

## Method Delegation

### `delegate`

The macro [delegate](#) offers an easy way to forward methods.

Let's imagine that users in some application have login information in the `User` model but name and other data in a separate `Profile` model:

```
class User < ApplicationRecord
  has_one :profile
end
```

With that configuration you get a user's name via their profile, `user.profile.name`, but it could be handy to still be able to access such attribute directly:

```
class User < ApplicationRecord
  has_one :profile

  def name
    profile.name
  end
end
```

That is what `delegate` does for you:

```
class User < ApplicationRecord
  has_one :profile

  delegate :name, to: :profile
end
```

It is shorter, and the intention more obvious.

The method must be public in the target.

The `delegate` macro accepts several methods:

```
delegate :name, :age, :address, :twitter, to: :profile
```

When interpolated into a string, the `:to` option should become an expression that evaluates to the object the method is delegated to. Typically a string or symbol. Such an expression is evaluated in the context of the receiver:

```
# delegates to the Rails constant
delegate :logger, to: :Rails

# delegates to the receiver's class
delegate :table_name, to: :class
```

WARNING: If the `:prefix` option is `true` this is less generic, see below.

By default, if the delegation raises `NoMethodError` and the target is `nil` the exception is propagated. You can ask that `nil` is returned instead with the `:allow_nil` option:

```
delegate :name, to: :profile, allow_nil: true
```

With `:allow_nil` the call `user.name` returns `nil` if the user has no profile.

The option `:prefix` adds a prefix to the name of the generated method. This may be handy for example to get a better name:

```
delegate :street, to: :address, prefix: true
```

The previous example generates `address_street` rather than `street`.

WARNING: Since in this case the name of the generated method is composed of the target object and target method names, the `:to` option must be a method name.

A custom prefix may also be configured:

```
delegate :size, to: :attachment, prefix: :avatar
```

In the previous example the macro generates `avatar_size` rather than `size`.

The option `:private` changes methods scope:

```
delegate :date_of_birth, to: :profile, private: true
```

The delegated methods are public by default. Pass `private: true` to change that.

NOTE: Defined in `active_support/core_ext/module/delegation.rb`

### `delegate_missing_to`

Imagine you would like to delegate everything missing from the `User` object, to the `Profile` one. The [delegate\\_missing\\_to](#) macro lets you implement this in a breeze:

```
class User < ApplicationRecord
  has_one :profile

  delegate_missing_to :profile
end
```

The target can be anything callable within the object, e.g. instance variables, methods, constants, etc. Only the public methods of the target are delegated.

NOTE: Defined in `active_support/core_ext/module/delegation.rb`.

## Redefining Methods

There are cases where you need to define a method with `define_method`, but don't know whether a method with that name already exists. If it does, a warning is issued if they are enabled. No big deal, but not clean either.

The method [redefine\\_method](#) prevents such a potential warning, removing the existing method before if needed.

You can also use [silence\\_redefinition\\_of\\_method](#) if you need to define the replacement method yourself (because you're using `delegate`, for example).

NOTE: Defined in `active_support/core_ext/module/redefine_method.rb`.

## Extensions to `Class`

### Class Attributes

#### `class_attribute`

The method [class\\_attribute](#) declares one or more inheritable class attributes that can be overridden at any level down the hierarchy.

```
class A
  class_attribute :x
end

class B < A; end

class C < B; end
```

```

A.x = :a
B.x # => :a
C.x # => :a

B.x = :b
A.x # => :a
C.x # => :b

C.x = :c
A.x # => :a
B.x # => :b

```

For example `ActionMailer::Base` defines:

```

class_attribute :default_params
self.default_params = {
  mime_version: "1.0",
  charset: "UTF-8",
  content_type: "text/plain",
  parts_order: [ "text/plain", "text/enriched", "text/html" ]
}.freeze

```

They can also be accessed and overridden at the instance level.

```

A.x = 1

a1 = A.new
a2 = A.new
a2.x = 2

a1.x # => 1, comes from A
a2.x # => 2, overridden in a2

```

The generation of the writer instance method can be prevented by setting the option `:instance_writer` to `false`.

```

module ActiveRecord
  class Base
    class_attribute :table_name_prefix, instance_writer: false, default: "my"
  end
end

```

A model may find that option useful as a way to prevent mass-assignment from setting the attribute.

The generation of the reader instance method can be prevented by setting the option `:instance_reader` to `false`.

```

class A
  class_attribute :x, instance_reader: false

```



```

end

A.new.x = 1
A.new.x # NoMethodError

```

For convenience `class_attribute` also defines an instance predicate which is the double negation of what the instance reader returns. In the examples above it would be called `x?` .

When `:instance_reader` is `false` , the instance predicate returns a `NoMethodError` just like the reader method.

If you do not want the instance predicate, pass `instance_predicate: false` and it will not be defined.

NOTE: Defined in `active_support/core_ext/class/attribute.rb` .

### **`cattr_reader` , `cattr_writer` , and `cattr_accessor`**

The macros `cattr_reader` , `cattr_writer` , and `cattr_accessor` are analogous to their `attr_*` counterparts but for classes. They initialize a class variable to `nil` unless it already exists, and generate the corresponding class methods to access it:

```

class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans.
  cattr_accessor :emulate_booleans
end

```

Also, you can pass a block to `cattr_*` to set up the attribute with a default value:

```

class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans with default value of true.
  cattr_accessor :emulate_booleans, default: true
end

```

Instance methods are created as well for convenience, they are just proxies to the class attribute. So, instances can change the class attribute, but cannot override it as it happens with `class_attribute` (see above). For example given

```

module ActionView
  class Base
    cattr_accessor :field_error_proc, default: Proc.new { ... }
  end
end

```

we can access `field_error_proc` in views.

The generation of the reader instance method can be prevented by setting `:instance_reader` to `false` and the generation of the writer instance method can be prevented by setting `:instance_writer` to `false` . Generation of both methods can be prevented by setting `:instance_accessor` to `false` . In all cases, the value must be exactly `false` and not any false value.

```

module A
  class B
    # No first_name instance reader is generated.
    attr_accessor :first_name, instance_reader: false
    # No last_name= instance writer is generated.
    attr_accessor :last_name, instance_writer: false
    # No surname instance reader or surname= writer is generated.
    attr_accessor :surname, instance_accessor: false
  end
end

```

A model may find it useful to set `:instance_accessor` to `false` as a way to prevent mass-assignment from setting the attribute.

NOTE: Defined in `active_support/core_ext/module/attribute_accessors.rb`.

## Subclasses and Descendants

### subclasses

The [subclasses](#) method returns the subclasses of the receiver:

```

class C; end
C.subclasses # => []

class B < C; end
C.subclasses # => [B]

class A < B; end
C.subclasses # => [B]

class D < C; end
C.subclasses # => [B, D]

```

The order in which these classes are returned is unspecified.

NOTE: Defined in `active_support/core_ext/class/subclasses.rb`.

### descendants

The [descendants](#) method returns all classes that are `<` than its receiver:

```

class C; end
C.descendants # => []

class B < C; end
C.descendants # => [B]

class A < B; end
C.descendants # => [B, A]

```

```
class D < C; end
C.descendants # => [B, A, D]
```

The order in which these classes are returned is unspecified.

NOTE: Defined in `active_support/core_ext/class/subclasses.rb`.

## Extensions to `String`

### Output Safety

#### Motivation

Inserting data into HTML templates needs extra care. For example, you can't just interpolate `@review.title` verbatim into an HTML page. For one thing, if the review title is "Flanagan & Matz rules!" the output won't be well-formed because an ampersand has to be escaped as "&amp;". What's more, depending on the application, that may be a big security hole because users can inject malicious HTML setting a hand-crafted review title. Check out the section about cross-site scripting in the [Security guide](#) for further information about the risks.

#### Safe Strings

Active Support has the concept of (*html*) *safe* strings. A safe string is one that is marked as being insertable into HTML as is. It is trusted, no matter whether it has been escaped or not.

Strings are considered to be *unsafe* by default:

```
"".html_safe? # => false
```

You can obtain a safe string from a given one with the [html\\_safe](#) method:

```
s = "".html_safe
s.html_safe? # => true
```

It is important to understand that `html_safe` performs no escaping whatsoever, it is just an assertion:

```
s = "<script>...</script>".html_safe
s.html_safe? # => true
s           # => "<script>...</script>"
```

It is your responsibility to ensure calling `html_safe` on a particular string is fine.

If you append onto a safe string, either in-place with `concat` / `<<`, or with `+`, the result is a safe string. Unsafe arguments are escaped:

```
"".html_safe + "<" # => "&lt;"
```

Safe arguments are directly appended:

```
"".html_safe + "<".html_safe # => "<"
```

These methods should not be used in ordinary views. Unsafe values are automatically escaped:

```
<%= @review.title %> <%# fine, escaped if needed %>
```

To insert something verbatim use the `raw` helper rather than calling `html_safe` :

```
<%= raw @cms.current_template %> <%# inserts @cms.current_template as is %>
```

or, equivalently, use `<%=` :

```
<%= @cms.current_template %> <%# inserts @cms.current_template as is %>
```

The `raw` helper calls `html_safe` for you:

```
def raw(stringish)
  stringish.to_s.html_safe
end
```

NOTE: Defined in `active_support/core_ext/string/output_safety.rb` .

### Transformation

As a rule of thumb, except perhaps for concatenation as explained above, any method that may change a string gives you an unsafe string. These are `downcase` , `gsub` , `strip` , `chomp` , `underscore` , etc.

In the case of in-place transformations like `gsub!` the receiver itself becomes unsafe.

INFO: The safety bit is lost always, no matter whether the transformation actually changed something.

### Conversion and Coercion

Calling `to_s` on a safe string returns a safe string, but coercion with `to_str` returns an unsafe string.

### Copying

Calling `dup` or `clone` on safe strings yields safe strings.

#### `remove`

The method `remove` will remove all occurrences of the pattern:

```
"Hello World".remove(/Hello /) # => "World"
```

There's also the destructive version `String#remove!` .

NOTE: Defined in `active_support/core_ext/string/filters.rb` .

#### `squish`

The method `squish` strips leading and trailing whitespace, and substitutes runs of whitespace with a single space each:

```
" \n  foo\n\r \t bar \n".squish # => "foo bar"
```

There's also the destructive version `String#squish!` .

Note that it handles both ASCII and Unicode whitespace.

NOTE: Defined in `active_support/core_ext/string/filters.rb` .

## **truncate**

The method [truncate](#) returns a copy of its receiver truncated after a given `length` :

```
"Oh dear! Oh dear! I shall be late!".truncate(20)
# => "Oh dear! Oh dear!..."
```

Ellipsis can be customized with the `:omission` option:

```
"Oh dear! Oh dear! I shall be late!".truncate(20, omission: '&hellip;')
# => "Oh dear! Oh &hellip;"
```

Note in particular that truncation takes into account the length of the omission string.

Pass a `:separator` to truncate the string at a natural break:

```
"Oh dear! Oh dear! I shall be late!".truncate(18)
# => "Oh dear! Oh dea..."
" Oh dear! Oh dear! I shall be late!".truncate(18, separator: ' ')
# => "Oh dear! Oh..."
```

The option `:separator` can be a regexp:

```
"Oh dear! Oh dear! I shall be late!".truncate(18, separator: /\s/)
# => "Oh dear! Oh..."
```

In above examples "dear" gets cut first, but then `:separator` prevents it.

NOTE: Defined in `active_support/core_ext/string/filters.rb` .

## **truncate\_bytes**

The method [truncate\\_bytes](#) returns a copy of its receiver truncated to at most `bytesize` bytes:

```
"👍👍👍👍".truncate_bytes(15)
# => "👍👍👍👍..."
```

Ellipsis can be customized with the `:omission` option:

```
"👍👍👍👍".truncate_bytes(15, omission: "👉")
# => "👍👍👍👍👉"
```

NOTE: Defined in `active_support/core_ext/string/filters.rb`.

### **truncate\_words**

The method `truncate_words` returns a copy of its receiver truncated after a given number of words:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4)
# => "Oh dear! Oh dear!..."
```

Ellipsis can be customized with the `:omission` option:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, omission: '&hellip;')
# => "Oh dear! Oh dear!&hellip;"
```

Pass a `:separator` to truncate the string at a natural break:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(3, separator: '!')
# => "Oh dear! Oh dear! I shall be late..."
```

The option `:separator` can be a regexp:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, separator: /\s/)
# => "Oh dear! Oh dear!..."
```

NOTE: Defined in `active_support/core_ext/string/filters.rb`.

### **inquiry**

The `inquiry` method converts a string into a `StringInquirer` object making equality checks prettier.

```
"production".inquiry.production? # => true
"active".inquiry.inactive?        # => false
```

NOTE: Defined in `active_support/core_ext/string/inquiry.rb`.

### **starts\_with? and ends\_with?**

Active Support defines 3rd person aliases of `String#start_with?` and `String#end_with?`:

```
"foo".starts_with?("f") # => true
"foo".ends_with?("o")   # => true
```

NOTE: Defined in `active_support/core_ext/string/starts_ends_with.rb`.

### **strip\_heredoc**

The method `strip_heredoc` strips indentation in heredocs.

For example in

```
if options[:usage]
  puts <<-USAGE.strip_heredoc
    This command does such and such.

    Supported options are:
      -h      This message
    ...
  USAGE
end
```

the user would see the usage message aligned against the left margin.

Technically, it looks for the least indented line in the whole string, and removes that amount of leading whitespace.

NOTE: Defined in `active_support/core_ext/string/strip.rb`.

## indent

The `indent` method indents the lines in the receiver:

```
<<EOS.indent(2)
def some_method
  some_code
end
EOS
# =>
def some_method
  some_code
end
```

The second argument, `indent_string`, specifies which indent string to use. The default is `nil`, which tells the method to make an educated guess peeking at the first indented line, and fallback to a space if there is none.

```
"foo".indent(2) # => "  foo"
"foo\n\t\tbar".indent(2) # => "\t\tfoo\n\t\t\tbar"
"foo".indent(2, "\t") # => "\t\tfoo"
```

While `indent_string` is typically one space or tab, it may be any string.

The third argument, `indent_empty_lines`, is a flag that says whether empty lines should be indented. Default is false.

```
"foo\n\nbar".indent(2) # => "  foo\n\n  bar"
"foo\n\nbar".indent(2, nil, true) # => "  foo\n  \n  bar"
```

The `indent!` method performs indentation in-place.

NOTE: Defined in `active_support/core_ext/string/indent.rb`.

## Access

### `at(position)`

The `at` method returns the character of the string at position `position` :

```
"hello".at(0) # => "h"
"hello".at(4) # => "o"
"hello".at(-1) # => "o"
"hello".at(10) # => nil
```

NOTE: Defined in `active_support/core_ext/string/access.rb` .

### `from(position)`

The `from` method returns the substring of the string starting at position `position` :

```
"hello".from(0) # => "hello"
"hello".from(2) # => "llo"
"hello".from(-2) # => "lo"
"hello".from(10) # => nil
```

NOTE: Defined in `active_support/core_ext/string/access.rb` .

### `to(position)`

The `to` method returns the substring of the string up to position `position` :

```
"hello".to(0) # => "h"
"hello".to(2) # => "hel"
"hello".to(-2) # => "hell"
"hello".to(10) # => "hello"
```

NOTE: Defined in `active_support/core_ext/string/access.rb` .

### `first(limit = 1)`

The `first` method returns a substring containing the first `limit` characters of the string.

The call `str.first(n)` is equivalent to `str.to(n-1)` if `n > 0`, and returns an empty string for `n == 0`.

NOTE: Defined in `active_support/core_ext/string/access.rb` .

### `last(limit = 1)`

The `last` method returns a substring containing the last `limit` characters of the string.

The call `str.last(n)` is equivalent to `str.from(-n)` if `n > 0`, and returns an empty string for `n == 0`.

NOTE: Defined in `active_support/core_ext/string/access.rb` .

## Inflections



## pluralize

The method [pluralize](#) returns the plural of its receiver:

```
"table".pluralize      # => "tables"
"ruby".pluralize       # => "rubies"
"equipment".pluralize  # => "equipment"
```

As the previous example shows, Active Support knows some irregular plurals and uncountable nouns. Built-in rules can be extended in `config/initializers/inflections.rb`. This file is generated by default, by the `rails new` command and has instructions in comments.

`pluralize` can also take an optional `count` parameter. If `count == 1` the singular form will be returned. For any other value of `count` the plural form will be returned:

```
"dude".pluralize(0) # => "dudes"
"dude".pluralize(1) # => "dude"
"dude".pluralize(2) # => "dudes"
```

Active Record uses this method to compute the default table name that corresponds to a model:

```
# active_record/model_schema.rb
def undecorated_table_name(model_name)
  table_name = model_name.to_s.demodulize.underscore
  pluralize_table_names ? table_name.pluralize : table_name
end
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb`.

## singularize

The [singularize](#) method is the inverse of `pluralize`:

```
"tables".singularize  # => "table"
"rubies".singularize   # => "ruby"
"equipment".singularize # => "equipment"
```

Associations compute the name of the corresponding default associated class using this method:

```
# active_record/reflection.rb
def derive_class_name
  class_name = name.to_s.camelize
  class_name = class_name.singularize if collection?
  class_name
end
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb`.

## camelize

The method `camelize` returns its receiver in camel case:

```
"product".camelize    # => "Product"
"admin_user".camelize # => "AdminUser"
```

As a rule of thumb you can think of this method as the one that transforms paths into Ruby class or module names, where slashes separate namespaces:

```
"backoffice/session".camelize # => "Backoffice::Session"
```

For example, Action Pack uses this method to load the class that provides a certain session store:

```
# action_controller/metal/session_management.rb
def session_store=(store)
  @@session_store = store.is_a?(Symbol) ?
    ActionDispatch::Session.const_get(store.to_s.camelize) :
    store
end
```

`camelize` accepts an optional argument, it can be `:upper` (default), or `:lower`. With the latter the first letter becomes lowercase:

```
"visual_effect".camelize(:lower) # => "visualEffect"
```

That may be handy to compute method names in a language that follows that convention, for example JavaScript.

INFO: As a rule of thumb you can think of `camelize` as the inverse of `underscore`, though there are cases where that does not hold: `"SSLerror".underscore.camelize` gives back `"SslError"`. To support cases such as this, ActiveSupport allows you to specify acronyms in `config/initializers/inflections.rb`:

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.acronym 'SSL'
end

"SSLerror".underscore.camelize # => "SSLerror"
```

`camelize` is aliased to `camelcase`.

NOTE: Defined in `active_support/core_ext/string/inflections.rb`.

### **underscore**

The method `underscore` goes the other way around, from camel case to paths:

```
"Product".underscore    # => "product"
"AdminUser".underscore  # => "admin_user"
```

Also converts `::` back to `/`:

```
"Backoffice::Session".underscore # => "backoffice/session"
```

and understands strings that start with lowercase:

```
"visualEffect".underscore # => "visual_effect"
```

`underscore` accepts no argument though.

Rails uses `underscore` to get a lowercased name for controller classes:

```
# actionpack/lib/abstract_controller/base.rb
def controller_path
  @controller_path ||= name.delete_suffix("Controller").underscore
end
```

For example, that value is the one you get in `params[:controller]` .

INFO: As a rule of thumb you can think of `underscore` as the inverse of `camelize` , though there are cases where that does not hold. For example, `"SSLError".underscore.camelize` gives back `"SslError"` .

NOTE: Defined in `active_support/core_ext/string/inflexions.rb` .

### **`titleize`**

The method `titleize` capitalizes the words in the receiver:

```
"alice in wonderland".titleize # => "Alice In Wonderland"
"fermat's enigma".titleize      # => "Fermat's Enigma"
```

`titleize` is aliased to `titlecase` .

NOTE: Defined in `active_support/core_ext/string/inflexions.rb` .

### **`dasherize`**

The method `dasherize` replaces the underscores in the receiver with dashes:

```
"name".dasherize      # => "name"
"contact_data".dasherize # => "contact-data"
```

The XML serializer of models uses this method to dasherize node names:

```
# active_model/serializers/xml.rb
def reformat_name(name)
  name = name.camelize if camelize?
  dasherize? ? name.dasherize : name
end
```

NOTE: Defined in `active_support/core_ext/string/inflexions.rb` .

## demodulize

Given a string with a qualified constant name, [demodulize](#) returns the very constant name, that is, the rightmost part of it:

```
"Product".demodulize           # => "Product"
"Backoffice::UsersController".demodulize # => "UsersController"
"Admin::Hotel::ReservationUtils".demodulize # => "ReservationUtils"
"::Inflections".demodulize      # => "Inflections"
"".demodulize                   # => ""
```

Active Record for example uses this method to compute the name of a counter cache column:

```
# active_record/reflection.rb
def counter_cache_column
  if options[:counter_cache] == true
    "#{active_record.name.demodulize.underscore.pluralize}_count"
  elsif options[:counter_cache]
    options[:counter_cache]
  end
end
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb`.

## deconstantize

Given a string with a qualified constant reference expression, [deconstantize](#) removes the rightmost segment, generally leaving the name of the constant's container:

```
"Product".deconstantize           # => ""
"Backoffice::UsersController".deconstantize # => "Backoffice"
"Admin::Hotel::ReservationUtils".deconstantize # => "Admin::Hotel"
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb`.

## parameterize

The method [parameterize](#) normalizes its receiver in a way that can be used in pretty URLs.

```
"John Smith".parameterize # => "john-smith"
"Kurt Gödel".parameterize # => "kurt-godel"
```

To preserve the case of the string, set the `preserve_case` argument to true. By default, `preserve_case` is set to false.

```
"John Smith".parameterize(preserve_case: true) # => "John-Smith"
"Kurt Gödel".parameterize(preserve_case: true) # => "Kurt-Godel"
```

To use a custom separator, override the `separator` argument.

```
"John Smith".parameterize(separator: "_") # => "john\_smith"
"Kurt Gödel".parameterize(separator: "_") # => "kurt\_godel"
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb` .

### **tableize**

The method `tableize` is `underscore` followed by `pluralize` .

```
"Person".tableize      # => "people"
"Invoice".tableize     # => "invoices"
"InvoiceLine".tableize # => "invoice_lines"
```

As a rule of thumb, `tableize` returns the table name that corresponds to a given model for simple cases. The actual implementation in Active Record is not straight `tableize` indeed, because it also demodulizes the class name and checks a few options that may affect the returned string.

NOTE: Defined in `active_support/core_ext/string/inflections.rb` .

### **classify**

The method `classify` is the inverse of `tableize` . It gives you the class name corresponding to a table name:

```
"people".classify      # => "Person"
"invoices".classify    # => "Invoice"
"invoice_lines".classify # => "InvoiceLine"
```

The method understands qualified table names:

```
"highrise_production.companies".classify # => "Company"
```

Note that `classify` returns a class name as a string. You can get the actual class object by invoking `constantize` on it, explained next.

NOTE: Defined in `active_support/core_ext/string/inflections.rb` .

### **constantize**

The method `constantize` resolves the constant reference expression in its receiver:

```
"Integer".constantize # => Integer

module M
  X = 1
end
"M::X".constantize # => 1
```

If the string evaluates to no known constant, or its content is not even a valid constant name, `constantize` raises `NameError` .

Constant name resolution by `constantize` starts always at the top-level `Object` even if there is no leading "::".

```
X = :in_Object
module M
  X = :in_M

  X # => :in_M
  "::X".constantize # => :in_Object
  "X".constantize # => :in_Object (!)
end
```

So, it is in general not equivalent to what Ruby would do in the same spot, had a real constant be evaluated.

Mailer test cases obtain the mailer being tested from the name of the test class using `constantize` :

```
# action_mailer/test_case.rb
def determine_default_mailer(name)
  name.delete_suffix("Test").constantize
rescue NameError => e
  raise NonInferableMailerError.new(name)
end
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb` .

### **humanize**

The method [humanize](#) tweaks an attribute name for display to end users.

Specifically, it performs these transformations:

- Applies human inflection rules to the argument.
- Deletes leading underscores, if any.
- Removes a "\_id" suffix if present.
- Replaces underscores with spaces, if any.
- Downcases all words except acronyms.
- Capitalizes the first word.

The capitalization of the first word can be turned off by setting the `:capitalize` option to false (default is true).

```
"name".humanize # => "Name"
"author_id".humanize # => "Author"
"author_id".humanize(capitalize: false) # => "author"
"comments_count".humanize # => "Comments count"
"_id".humanize # => "Id"
```

If "SSL" was defined to be an acronym:

```
'ssl_error'.humanize # => "SSL error"
```

The helper method `full_messages` uses `humanize` as a fallback to include attribute names:

```
def full_messages
  map { |attribute, message| full_message(attribute, message) }
end

def full_message
  # ...
  attr_name = attribute.to_s.tr('.', '_').humanize
  attr_name = @base.class.human_attribute_name(attribute, default: attr_name)
  # ...
end
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb`.

### `foreign_key`

The method `foreign_key` gives a foreign key column name from a class name. To do so it demodulizes, underscores, and adds "\_id":

```
"User".foreign_key      # => "user_id"
"InvoiceLine".foreign_key # => "invoice_line_id"
"Admin::Session".foreign_key # => "session_id"
```

Pass a false argument if you do not want the underscore in "\_id":

```
"User".foreign_key(false) # => "userid"
```

Associations use this method to infer foreign keys, for example `has_one` and `has_many` do this:

```
# active_record/associations.rb
foreign_key = options[:foreign_key] || reflection.active_record.name.foreign_key
```

NOTE: Defined in `active_support/core_ext/string/inflections.rb`.

## Conversions

### `to_date`, `to_time`, `to_datetime`

The methods `to_date`, `to_time`, and `to_datetime` are basically convenience wrappers around

`Date._parse`:

```
"2010-07-27".to_date      # => Tue, 27 Jul 2010
"2010-07-27 23:37:00".to_time # => 2010-07-27 23:37:00 +0200
"2010-07-27 23:37:00".to_datetime # => Tue, 27 Jul 2010 23:37:00 +0000
```

`to_time` receives an optional argument `:utc` or `:local`, to indicate which time zone you want the time in:

```
"2010-07-27 23:42:00".to_time(:utc) # => 2010-07-27 23:42:00 UTC
"2010-07-27 23:42:00".to_time(:local) # => 2010-07-27 23:42:00 +0200
```

Default is `:local` .

Please refer to the documentation of `Date._parse` for further details.

INFO: The three of them return `nil` for blank receivers.

NOTE: Defined in `active_support/core_ext/string/conversions.rb` .

## Extensions to `Symbol`

### `starts_with?` and `ends_with?`

Active Support defines 3rd person aliases of `Symbol#start_with?` and `Symbol#end_with?` :

```
:foo.starts_with?("f") # => true
:foo.ends_with?("o")  # => true
```

NOTE: Defined in `active_support/core_ext/symbol/starts_ends_with.rb` .

## Extensions to `Numeric`

### Bytes

All numbers respond to these methods:

- [bytes](#)
- [kilobytes](#)
- [megabytes](#)
- [gigabytes](#)
- [terabytes](#)
- [petabytes](#)
- [exabytes](#)

They return the corresponding amount of bytes, using a conversion factor of 1024:

```
2.kilobytes    # => 2048
3.megabytes    # => 3145728
3.5.gigabytes  # => 3758096384
-4.exabytes    # => -4611686018427387904
```

Singular forms are aliased so you are able to say:

```
1.megabyte # => 1048576
```

NOTE: Defined in `active_support/core_ext/numeric/bytes.rb` .

### Time

The following methods:

- [seconds](#)



- [minutes](#)
- [hours](#)
- [days](#)
- [weeks](#)
- [fortnights](#)

enable time declarations and calculations, like `45.minutes + 2.hours + 4.weeks`. Their return values can also be added to or subtracted from Time objects.

These methods can be combined with [from\\_now](#), [ago](#), etc, for precise date calculations. For example:

```
# equivalent to Time.current.advance(days: 1)
1.day.from_now

# equivalent to Time.current.advance(weeks: 2)
2.weeks.from_now

# equivalent to Time.current.advance(days: 4, weeks: 5)
(4.days + 5.weeks).from_now
```

WARNING. For other durations please refer to the time extensions to `Integer`.

NOTE: Defined in `active_support/core_ext/numeric/time.rb`.

## Formatting

Enables the formatting of numbers in a variety of ways.

Produce a string representation of a number as a telephone number:

```
5551234.to_fs(:phone)
# => 555-1234
1235551234.to_fs(:phone)
# => 123-555-1234
1235551234.to_fs(:phone, area_code: true)
# => (123) 555-1234
1235551234.to_fs(:phone, delimiter: " ")
# => 123 555 1234
1235551234.to_fs(:phone, area_code: true, extension: 555)
# => (123) 555-1234 x 555
1235551234.to_fs(:phone, country_code: 1)
# => +1-123-555-1234
```

Produce a string representation of a number as currency:

```
1234567890.50.to_fs(:currency) # => $1,234,567,890.50
1234567890.506.to_fs(:currency) # => $1,234,567,890.51
1234567890.506.to_fs(:currency, precision: 3) # => $1,234,567,890.506
```

Produce a string representation of a number as a percentage:

```

100.to_fs(:percentage)
# => 100.000%
100.to_fs(:percentage, precision: 0)
# => 100%
1000.to_fs(:percentage, delimiter: '.', separator: ',')
# => 1.000,000%
302.24398923423.to_fs(:percentage, precision: 5)
# => 302.24399%

```

Produce a string representation of a number in delimited form:

```

12345678.to_fs(:delimited) # => 12,345,678
12345678.05.to_fs(:delimited) # => 12,345,678.05
12345678.to_fs(:delimited, delimiter: ".") # => 12.345.678
12345678.to_fs(:delimited, delimiter: ",") # => 12,345,678
12345678.05.to_fs(:delimited, separator: " ") # => 12,345,678 05

```

Produce a string representation of a number rounded to a precision:

```

111.2345.to_fs(:rounded) # => 111.235
111.2345.to_fs(:rounded, precision: 2) # => 111.23
13.to_fs(:rounded, precision: 5) # => 13.00000
389.32314.to_fs(:rounded, precision: 0) # => 389
111.2345.to_fs(:rounded, significant: true) # => 111

```

Produce a string representation of a number as a human-readable number of bytes:

```

123.to_fs(:human_size) # => 123 Bytes
1234.to_fs(:human_size) # => 1.21 KB
12345.to_fs(:human_size) # => 12.1 KB
1234567.to_fs(:human_size) # => 1.18 MB
1234567890.to_fs(:human_size) # => 1.15 GB
1234567890123.to_fs(:human_size) # => 1.12 TB
1234567890123456.to_fs(:human_size) # => 1.1 PB
1234567890123456789.to_fs(:human_size) # => 1.07 EB

```

Produce a string representation of a number in human-readable words:

```

123.to_fs(:human) # => "123"
1234.to_fs(:human) # => "1.23 Thousand"
12345.to_fs(:human) # => "12.3 Thousand"
1234567.to_fs(:human) # => "1.23 Million"
1234567890.to_fs(:human) # => "1.23 Billion"
1234567890123.to_fs(:human) # => "1.23 Trillion"
1234567890123456.to_fs(:human) # => "1.23 Quadrillion"

```

NOTE: Defined in `active_support/core_ext/numeric/conversions.rb`.

## Extensions to Integer

### multiple\_of?

The method [multiple\\_of?](#) tests whether an integer is multiple of the argument:

```
2.multiple_of?(1) # => true
1.multiple_of?(2) # => false
```

NOTE: Defined in `active_support/core_ext/integer/multiple.rb`.

### ordinal

The method [ordinal](#) returns the ordinal suffix string corresponding to the receiver integer:

```
1.ordinal # => "st"
2.ordinal # => "nd"
53.ordinal # => "rd"
2009.ordinal # => "th"
-21.ordinal # => "st"
-134.ordinal # => "th"
```

NOTE: Defined in `active_support/core_ext/integer/inflections.rb`.

### ordinalize

The method [ordinalize](#) returns the ordinal string corresponding to the receiver integer. In comparison, note that the `ordinal` method returns **only** the suffix string.

```
1.ordinalize # => "1st"
2.ordinalize # => "2nd"
53.ordinalize # => "53rd"
2009.ordinalize # => "2009th"
-21.ordinalize # => "-21st"
-134.ordinalize # => "-134th"
```

NOTE: Defined in `active_support/core_ext/integer/inflections.rb`.

## Time

The following methods:

- [months](#)
- [years](#)

enable time declarations and calculations, like `4.months + 5.years`. Their return values can also be added to or subtracted from Time objects.

These methods can be combined with [from\\_now](#), [ago](#), etc, for precise date calculations. For example:

```
# equivalent to Time.current.advance(months: 1)
1.month.from_now

# equivalent to Time.current.advance(years: 2)
2.years.from_now

# equivalent to Time.current.advance(months: 4, years: 5)
(4.months + 5.years).from_now
```

WARNING. For other durations please refer to the time extensions to `Numeric`.

NOTE: Defined in `active_support/core_ext/integer/time.rb`.

## Extensions to `BigDecimal`

### `to_s`

The method `to_s` provides a default specifier of "F". This means that a simple call to `to_s` will result in floating-point representation instead of engineering notation:

```
BigDecimal(5.00, 6).to_s # => "5.0"
```

Engineering notation is still supported:

```
BigDecimal(5.00, 6).to_s("e") # => "0.5E1"
```

## Extensions to `Enumerable`

### `sum`

The method `sum` adds the elements of an enumerable:

```
[1, 2, 3].sum # => 6
(1..100).sum # => 5050
```

Addition only assumes the elements respond to `+`:

```
[[1, 2], [2, 3], [3, 4]].sum # => [1, 2, 2, 3, 3, 4]
%w(foo bar baz).sum # => "foobarbaz"
{a: 1, b: 2, c: 3}.sum # => [:a, 1, :b, 2, :c, 3]
```

The sum of an empty collection is zero by default, but this is customizable:

```
[] .sum # => 0
[] .sum(1) # => 1
```

If a block is given, `sum` becomes an iterator that yields the elements of the collection and sums the returned values:

```
(1..5).sum {|n| n * 2 } # => 30
[2, 4, 6, 8, 10].sum    # => 30
```

The sum of an empty receiver can be customized in this form as well:

```
[].sum(1) {|n| n**3} # => 1
```

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **index\_by**

The method `index_by` generates a hash with the elements of an enumerable indexed by some key.

It iterates through the collection and passes each element to a block. The element will be keyed by the value returned by the block:

```
invoices.index_by(&:number)
# => {'2009-032' => <Invoice ...>, '2009-008' => <Invoice ...>, ...}
```

WARNING. Keys should normally be unique. If the block returns the same value for different elements no collection is built for that key. The last item will win.

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **index\_with**

The method `index_with` generates a hash with the elements of an enumerable as keys. The value is either a passed default or returned in a block.

```
post = Post.new(title: "hey there", body: "what's up?")

%i( title body ).index_with { |attr_name| post.public_send(attr_name) }
# => { title: "hey there", body: "what's up?" }

WEEKDAYS.index_with(Interval.all_day)
# => { monday: [ 0, 1440 ], ... }
```

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **many?**

The method `many?` is shorthand for `collection.size > 1`:

```
<% if pages.many? %>
  <%= pagination_links %>
<% end %>
```

If an optional block is given, `many?` only takes into account those elements that return true:

```
@see_more = videos.many? {|video| video.category == params[:category]}
```

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **exclude?**

The predicate `exclude?` tests whether a given object does **not** belong to the collection. It is the negation of the built-in `include?`:

```
to_visit << node if visited.exclude?(node)
```

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **including**

The method `including` returns a new enumerable that includes the passed elements:

```
[ 1, 2, 3 ].including(4, 5) # => [ 1, 2, 3, 4, 5 ]
["David", "Rafael"].including %w[ Aaron Todd ] # => ["David", "Rafael", "Aaron", "Todd"]
```

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **excluding**

The method `excluding` returns a copy of an enumerable with the specified elements removed:

```
["David", "Rafael", "Aaron", "Todd"].excluding("Aaron", "Todd") # => ["David", "Rafael"]
```

`excluding` is aliased to `without`.

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **pluck**

The method `pluck` extracts the given key from each element:

```
[{ name: "David" }, { name: "Rafael" }, { name: "Aaron" }].pluck(:name) # => ["David", "Rafael", "Aaron"]
[{ id: 1, name: "David" }, { id: 2, name: "Rafael" }].pluck(:id, :name) # => [[1, "David"], [2, "Rafael"]]
```

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

### **pick**

The method `pick` extracts the given key from the first element:

```
[{ name: "David" }, { name: "Rafael" }, { name: "Aaron" }].pick(:name) # => "David"
[{ id: 1, name: "David" }, { id: 2, name: "Rafael" }].pick(:id, :name) # => [1, "David"]
```

NOTE: Defined in `active_support/core_ext/enumerable.rb`.

## Extensions to `Array`

### Accessing

Active Support augments the API of arrays to ease certain ways of accessing them. For example, `to` returns the subarray of elements up to the one at the passed index:

```
%w(a b c d).to(2) # => ["a", "b", "c"]
[].to(7)          # => []
```

Similarly, `from` returns the tail from the element at the passed index to the end. If the index is greater than the length of the array, it returns an empty array.

```
%w(a b c d).from(2) # => ["c", "d"]
%w(a b c d).from(10) # => []
[].from(0)          # => []
```

The method `including` returns a new array that includes the passed elements:

```
[ 1, 2, 3 ].including(4, 5)          # => [ 1, 2, 3, 4, 5 ]
[ [ 0, 1 ] ].including([ [ 1, 0 ] ]) # => [ [ 0, 1 ], [ 1, 0 ] ]
```

The method `excluding` returns a copy of the Array excluding the specified elements. This is an optimization of `Enumerable#excluding` that uses `Array#-` instead of `Array#reject` for performance reasons.

```
["David", "Rafael", "Aaron", "Todd"].excluding("Aaron", "Todd") # => ["David", "Rafael"]
[ [ 0, 1 ], [ 1, 0 ] ].excluding([ [ 1, 0 ] ])                  # => [ [ 0, 1 ] ]
```

The methods `second`, `third`, `fourth`, and `fifth` return the corresponding element, as do `second_to_last` and `third_to_last` (`first` and `last` are built-in). Thanks to social wisdom and positive constructiveness all around, `forty_two` is also available.

```
%w(a b c d).third # => "c"
%w(a b c d).fifth # => nil
```

NOTE: Defined in `active_support/core_ext/array/access.rb`.

### Extracting

The method `extract!` removes and returns the elements for which the block returns a true value. If no block is given, an Enumerator is returned instead.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
odd_numbers = numbers.extract! { |number| number.odd? } # => [1, 3, 5, 7, 9]
numbers # => [0, 2, 4, 6, 8]
```

NOTE: Defined in `active_support/core_ext/array/extract.rb`.

## Options Extraction

When the last argument in a method call is a hash, except perhaps for a `&block` argument, Ruby allows you to omit the brackets:

```
User.exists?(email: params[:email])
```

That syntactic sugar is used a lot in Rails to avoid positional arguments where there would be too many, offering instead interfaces that emulate named parameters. In particular it is very idiomatic to use a trailing hash for options.

If a method expects a variable number of arguments and uses `*` in its declaration, however, such an options hash ends up being an item of the array of arguments, where it loses its role.

In those cases, you may give an options hash a distinguished treatment with `extract_options!`. This method checks the type of the last item of an array. If it is a hash it pops it and returns it, otherwise it returns an empty hash.

Let's see for example the definition of the `caches_action` controller macro:

```
def caches_action(*actions)
  return unless cache_configured?
  options = actions.extract_options!
  # ...
end
```

This method receives an arbitrary number of action names, and an optional hash of options as last argument. With the call to `extract_options!` you obtain the options hash and remove it from `actions` in a simple and explicit way.

NOTE: Defined in `active_support/core_ext/array/extract_options.rb`.

## Conversions

### `to_sentence`

The method `to_sentence` turns an array into a string containing a sentence that enumerates its items:

```
%w().to_sentence           # => ""
%w(Earth).to_sentence       # => "Earth"
%w(Earth Wind).to_sentence  # => "Earth and Wind"
%w(Earth Wind Fire).to_sentence # => "Earth, Wind, and Fire"
```

This method accepts three options:



- `:two_words_connector` : What is used for arrays of length 2. Default is " and ".
- `:words_connector` : What is used to join the elements of arrays with 3 or more elements, except for the last two. Default is ", ".
- `:last_word_connector` : What is used to join the last items of an array with 3 or more elements. Default is ", and ".

The defaults for these options can be localized, their keys are:

Option	I18n key
<code>:two_words_connector</code>	<code>support.array.two_words_connector</code>
<code>:words_connector</code>	<code>support.array.words_connector</code>
<code>:last_word_connector</code>	<code>support.array.last_word_connector</code>

NOTE: Defined in `active_support/core_ext/array/conversions.rb`.

### `to_fs`

The method `to_fs` acts like `to_s` by default.

If the array contains items that respond to `id`, however, the symbol `:db` may be passed as argument. That's typically used with collections of Active Record objects. Returned strings are:

```
[].to_fs(:db)           # => "null"
[user].to_fs(:db)       # => "8456"
invoice.lines.to_fs(:db) # => "23,567,556,12"
```

Integers in the example above are supposed to come from the respective calls to `id`.

NOTE: Defined in `active_support/core_ext/array/conversions.rb`.

### `to_xml`

The method `to_xml` returns a string containing an XML representation of its receiver:

```
Contributor.limit(2).order(:rank).to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors type="array">
#   <contributor>
#     <id type="integer">4356</id>
#     <name>Jeremy Kemper</name>
#     <rank type="integer">1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id type="integer">4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank type="integer">2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
```

```
#   </contributor>
# </contributors>
```

To do so it sends `to_xml` to every item in turn, and collects the results under a root node. All items must respond to `to_xml`, an exception is raised otherwise.

By default, the name of the root element is the underscored and dasherized plural of the name of the class of the first item, provided the rest of elements belong to that type (checked with `is_a?`) and they are not hashes. In the example above that's "contributors".

If there's any element that does not belong to the type of the first one the root node becomes "objects":

```
[Contributor.first, Commit.first].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <id type="integer">4583</id>
#     <name>Aaron Batalion</name>
#     <rank type="integer">53</rank>
#     <url-id>aaron-batalion</url-id>
#   </object>
#   <object>
#     <author>Joshua Peek</author>
#     <authored-timestamp type="datetime">2009-09-02T16:44:36Z</authored-timestamp>
#     <branch>origin/master</branch>
#     <committed-timestamp type="datetime">2009-09-02T16:44:36Z</committed-
timestamp>
#     <committer>Joshua Peek</committer>
#     <git-show nil="true"></git-show>
#     <id type="integer">190316</id>
#     <imported-from-svn type="boolean">false</imported-from-svn>
#     <message>Kill AMo observing wrap_with_notifications since ARes was only using
it</message>
#     <sha1>723a47bfb3708f968821bc969a9a3fc873a3ed58</sha1>
#   </object>
# </objects>
```

If the receiver is an array of hashes the root element is by default also "objects":

```
[{a: 1, b: 2}, {c: 3}].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <b type="integer">2</b>
#     <a type="integer">1</a>
#   </object>
#   <object>
#     <c type="integer">3</c>
```

```
#   </object>
# </objects>
```

WARNING. If the collection is empty the root element is by default "nil-classes". That's a gotcha, for example the root element of the list of contributors above would not be "contributors" if the collection was empty, but "nil-classes". You may use the `:root` option to ensure a consistent root element.

The name of children nodes is by default the name of the root node singularized. In the examples above we've seen "contributor" and "object". The option `:children` allows you to set these node names.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder via the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder:

```
Contributor.limit(2).order(:rank).to_xml(skip_types: true)
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors>
#   <contributor>
#     <id>4356</id>
#     <name>Jeremy Kemper</name>
#     <rank>1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id>4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank>2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

NOTE: Defined in `active_support/core_ext/array/conversions.rb`.

## Wrapping

The method `Array.wrap` wraps its argument in an array unless it is already an array (or array-like).

Specifically:

- If the argument is `nil` an empty array is returned.
- Otherwise, if the argument responds to `to_ary` it is invoked, and if the value of `to_ary` is not `nil`, it is returned.
- Otherwise, an array with the argument as its single element is returned.

```
Array.wrap(nil)      # => []
Array.wrap([1, 2, 3]) # => [1, 2, 3]
Array.wrap(0)        # => [0]
```

This method is similar in purpose to `Kernel#Array`, but there are some differences:

- If the argument responds to `to_ary` the method is invoked. `Kernel#Array` moves on to try `to_a` if the returned value is `nil`, but `Array.wrap` returns an array with the argument as its single element right away.
- If the returned value from `to_ary` is neither `nil` nor an `Array` object, `Kernel#Array` raises an exception, while `Array.wrap` does not, it just returns the value.
- It does not call `to_a` on the argument, if the argument does not respond to `to_ary` it returns an array with the argument as its single element.

The last point is particularly worth comparing for some enumerables:

```
Array.wrap(foo: :bar) # => [{:foo=>:bar}]
Array(foo: :bar)      # => [[:foo, :bar]]
```

There's also a related idiom that uses the splat operator:

```
[*object]
```

NOTE: Defined in `active_support/core_ext/array/wrap.rb`.

## Duplicating

The method `Array#deep_dup` duplicates itself and all objects inside recursively with the Active Support method `Object#deep_dup`. It works like `Array#map`, sending `deep_dup` method to each object inside.

```
array = [1, [2, 3]]
dup = array.deep_dup
dup[1][2] = 4
array[1][2] == nil # => true
```

NOTE: Defined in `active_support/core_ext/object/deep_dup.rb`.

## Grouping

```
in_groups_of(number, fill_with = nil)
```

The method `in_groups_of` splits an array into consecutive groups of a certain size. It returns an array with the groups:

```
[1, 2, 3].in_groups_of(2) # => [[1, 2], [3, nil]]
```

or yields them in turn if a block is passed:

```
<% sample.in_groups_of(3) do |a, b, c| %>
  <tr>
    <td><%= a %></td>
    <td><%= b %></td>
    <td><%= c %></td>
  </tr>
<% end %>
```

The first example shows how `in_groups_of` fills the last group with as many `nil` elements as needed to have the requested size. You can change this padding value using the second optional argument:

```
[1, 2, 3].in_groups_of(2, 0) # => [[1, 2], [3, 0]]
```

And you can tell the method not to fill the last group by passing `false`:

```
[1, 2, 3].in_groups_of(2, false) # => [[1, 2], [3]]
```

As a consequence `false` can't be used as a padding value.

NOTE: Defined in `active_support/core_ext/array/grouping.rb`.

**`in_groups(number, fill_with = nil)`**

The method `in_groups` splits an array into a certain number of groups. The method returns an array with the groups:

```
%w(1 2 3 4 5 6 7).in_groups(3)
# => [["1", "2", "3"], ["4", "5", nil], ["6", "7", nil]]
```

or yields them in turn if a block is passed:

```
%w(1 2 3 4 5 6 7).in_groups(3) {|group| p group}
["1", "2", "3"]
["4", "5", nil]
["6", "7", nil]
```

The examples above show that `in_groups` fills some groups with a trailing `nil` element as needed. A group can get at most one of these extra elements, the rightmost one if any. And the groups that have them are always the last ones.

You can change this padding value using the second optional argument:

```
%w(1 2 3 4 5 6 7).in_groups(3, "0")
# => [["1", "2", "3"], ["4", "5", "0"], ["6", "7", "0"]]
```

And you can tell the method not to fill the smaller groups by passing `false`:

```
%w(1 2 3 4 5 6 7).in_groups(3, false)
# => [["1", "2", "3"], ["4", "5"], ["6", "7"]]
```

As a consequence `false` can't be used as a padding value.

NOTE: Defined in `active_support/core_ext/array/grouping.rb`.

**`split(value = nil)`**

The method `split` divides an array by a separator and returns the resulting chunks.

If a block is passed the separators are those elements of the array for which the block returns true:

```
(-5..5).to_a.split { |i| i.multiple_of?(4) }  
# => [[-5], [-3, -2, -1], [1, 2, 3], [5]]
```

Otherwise, the value received as argument, which defaults to `nil`, is the separator:

```
[0, 1, -5, 1, 1, "foo", "bar"].split(1)  
# => [[0], [-5], [], ["foo", "bar"]]
```

TIP: Observe in the previous example that consecutive separators result in empty arrays.

NOTE: Defined in `active_support/core_ext/array/grouping.rb`.

## Extensions to Hash

### Conversions

#### `to_xml`

The method `to_xml` returns a string containing an XML representation of its receiver:

```
{"foo" => 1, "bar" => 2}.to_xml  
# =>  
# <?xml version="1.0" encoding="UTF-8"?>  
# <hash>  
#   <foo type="integer">1</foo>  
#   <bar type="integer">2</bar>  
# </hash>
```

To do so, the method loops over the pairs and builds nodes that depend on the *values*. Given a pair `key`, `value`:

- If `value` is a hash there's a recursive call with `key` as `:root`.
- If `value` is an array there's a recursive call with `key` as `:root`, and `key` singularized as `:children`.
- If `value` is a callable object it must expect one or two arguments. Depending on the arity, the callable is invoked with the `options` hash as first argument with `key` as `:root`, and `key` singularized as second argument. Its return value becomes a new node.
- If `value` responds to `to_xml` the method is invoked with `key` as `:root`.
- Otherwise, a node with `key` as tag is created with a string representation of `value` as text node. If `value` is `nil` an attribute "nil" set to "true" is added. Unless the option `:skip_types` exists and is true, an attribute "type" is added as well according to the following mapping:

```
XML_TYPE_NAMES = {  
  "Symbol"    => "symbol",  
  "Integer"   => "integer",
```

```

    "BigDecimal" => "decimal",
    "Float"       => "float",
    "TrueClass"   => "boolean",
    "FalseClass"  => "boolean",
    "Date"        => "date",
    "DateTime"    => "datetime",
    "Time"        => "datetime"
  }

```

By default the root node is "hash", but that's configurable via the `:root` option.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder with the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder.

NOTE: Defined in `active_support/core_ext/hash/conversions.rb`.

## Merging

Ruby has a built-in method `Hash#merge` that merges two hashes:

```

{a: 1, b: 1}.merge(a: 0, c: 2)
# => {:a=>0, :b=>1, :c=>2}

```

Active Support defines a few more ways of merging hashes that may be convenient.

### `reverse_merge` and `reverse_merge!`

In case of collision the key in the hash of the argument wins in `merge`. You can support option hashes with default values in a compact way with this idiom:

```

options = {length: 30, omission: "..."}.merge(options)

```

Active Support defines `reverse_merge` in case you prefer this alternative notation:

```

options = options.reverse_merge(length: 30, omission: "...")

```

And a bang version `reverse_merge!` that performs the merge in place:

```

options.reverse_merge!(length: 30, omission: "...")

```

WARNING. Take into account that `reverse_merge!` may change the hash in the caller, which may or may not be a good idea.

NOTE: Defined in `active_support/core_ext/hash/reverse_merge.rb`.

### `reverse_update`

The method `reverse_update` is an alias for `reverse_merge!`, explained above.

WARNING. Note that `reverse_update` has no bang.

NOTE: Defined in `active_support/core_ext/hash/reverse_merge.rb`.

### `deep_merge` and `deep_merge!`

As you can see in the previous example if a key is found in both hashes the value in the one in the argument wins.

Active Support defines `Hash#deep_merge`. In a deep merge, if a key is found in both hashes and their values are hashes in turn, then their *merge* becomes the value in the resulting hash:

```
{a: {b: 1}}.deep_merge(a: {c: 2})  
# => {:a=>{:b=>1, :c=>2}}
```

The method `deep_merge!` performs a deep merge in place.

NOTE: Defined in `active_support/core_ext/hash/deep_merge.rb`.

## Deep duplicating

The method `Hash#deep_dup` duplicates itself and all keys and values inside recursively with Active Support method `Object#deep_dup`. It works like `Enumerator#each_with_object` with sending `deep_dup` method to each pair inside.

```
hash = { a: 1, b: { c: 2, d: [3, 4] } }  
  
dup = hash.deep_dup  
dup[:b][:e] = 5  
dup[:b][:d] << 5  
  
hash[:b][:e] == nil      # => true  
hash[:b][:d] == [3, 4]  # => true
```

NOTE: Defined in `active_support/core_ext/object/deep_dup.rb`.

## Working with Keys

### `except` and `except!`

The method `except` returns a hash with the keys in the argument list removed, if present:

```
{a: 1, b: 2}.except(:a) # => {:b=>2}
```

If the receiver responds to `convert_key`, the method is called on each of the arguments. This allows `except` to play nice with hashes with indifferent access for instance:

```
{a: 1}.with_indifferent_access.except(:a) # => {}  
{a: 1}.with_indifferent_access.except("a") # => {}
```

There's also the bang variant `except!` that removes keys in place.

NOTE: Defined in `active_support/core_ext/hash/except.rb`.



### **stringify\_keys and stringify\_keys!**

The method [stringify\\_keys](#) returns a hash that has a stringified version of the keys in the receiver. It does so by sending `to_s` to them:

```
{nil => nil, 1 => 1, a: :a}.stringify_keys
# => {"" => nil, "1" => 1, "a" => :a}
```

In case of key collision, the value will be the one most recently inserted into the hash:

```
{"a" => 1, a: 2}.stringify_keys
# The result will be
# => {"a"=>2}
```

This method may be useful for example to easily accept both symbols and strings as options. For instance

`ActionView::Helpers::FormHelper` defines:

```
def to_check_box_tag(options = {}, checked_value = "1", unchecked_value = "0")
  options = options.stringify_keys
  options["type"] = "checkbox"
  # ...
end
```

The second line can safely access the "type" key, and let the user to pass either `:type` or "type".

There's also the bang variant [stringify\\_keys!](#) that stringifies keys in place.

Besides that, one can use [deep\\_stringify\\_keys](#) and [deep\\_stringify\\_keys!](#) to stringify all the keys in the given hash and all the hashes nested in it. An example of the result is:

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_stringify_keys
# => {""=>nil, "1"=>1, "nested"=>{"a"=>3, "5"=>5}}
```

NOTE: Defined in `active_support/core_ext/hash/keys.rb`.

### **symbolize\_keys and symbolize\_keys!**

The method [symbolize\\_keys](#) returns a hash that has a symbolized version of the keys in the receiver, where possible. It does so by sending `to_sym` to them:

```
{nil => nil, 1 => 1, "a" => "a"}.symbolize_keys
# => {nil=>nil, 1=>1, :a=>"a"}
```

WARNING. Note in the previous example only one key was symbolized.

In case of key collision, the value will be the one most recently inserted into the hash:

```
{"a" => 1, a: 2}.symbolize_keys
# => {:a=>2}
```

This method may be useful for example to easily accept both symbols and strings as options. For instance

`ActionText::TagHelper` defines

```
def rich_text_area_tag(name, value = nil, options = {})
  options = options.symbolize_keys

  options[:input] ||= "trix_input_#{ActionText::TagHelper.id += 1}"
  # ...
end
```

The third line can safely access the `:input` key, and let the user to pass either `:input` or `"input"`.

There's also the bang variant [symbolize\\_keys!](#) that symbolizes keys in place.

Besides that, one can use [deep\\_symbolize\\_keys](#) and [deep\\_symbolize\\_keys!](#) to symbolize all the keys in the given hash and all the hashes nested in it. An example of the result is:

```
{nil => nil, 1 => 1, "nested" => {"a" => 3, 5 => 5}}.deep_symbolize_keys
# => {nil=>nil, 1=>1, nested:{a:3, 5=>5}}
```

NOTE: Defined in `active_support/core_ext/hash/keys.rb`.

#### **`to_options` and `to_options!`**

The methods [to\\_options](#) and [to\\_options!](#) are aliases of `symbolize_keys` and `symbolize_keys!`, respectively.

NOTE: Defined in `active_support/core_ext/hash/keys.rb`.

#### **`assert_valid_keys`**

The method [assert\\_valid\\_keys](#) receives an arbitrary number of arguments, and checks whether the receiver has any key outside that list. If it does `ArgumentError` is raised.

```
{a: 1}.assert_valid_keys(:a) # passes
{a: 1}.assert_valid_keys("a") # ArgumentError
```

Active Record does not accept unknown options when building associations, for example. It implements that control via `assert_valid_keys`.

NOTE: Defined in `active_support/core_ext/hash/keys.rb`.

## Working with Values

#### **`deep_transform_values` and `deep_transform_values!`**

The method [deep\\_transform\\_values](#) returns a new hash with all values converted by the block operation. This includes the values from the root hash and from all nested hashes and arrays.

```
hash = { person: { name: 'Rob', age: '28' } }
```

```
hash.deep_transform_values{ |value| value.to_s.upcase }  
# => {person: {name: "ROB", age: "28"}}
```

There's also the bang variant `deep_transform_values!` that destructively converts all values by using the block operation.

NOTE: Defined in `active_support/core_ext/hash/deep_transform_values.rb`.

## Slicing

The method `slice!` replaces the hash with only the given keys and returns a hash containing the removed key/value pairs.

```
hash = {a: 1, b: 2}  
rest = hash.slice!(:a) # => {:b=>2}  
hash                  # => {:a=>1}
```

NOTE: Defined in `active_support/core_ext/hash/slice.rb`.

## Extracting

The method `extract!` removes and returns the key/value pairs matching the given keys.

```
hash = {a: 1, b: 2}  
rest = hash.extract!(:a) # => {:a=>1}  
hash                  # => {:b=>2}
```

The method `extract!` returns the same subclass of Hash that the receiver is.

```
hash = {a: 1, b: 2}.with_indifferent_access  
rest = hash.extract!(:a).class  
# => ActiveSupport::HashWithIndifferentAccess
```

NOTE: Defined in `active_support/core_ext/hash/slice.rb`.

## Indifferent Access

The method `with_indifferent_access` returns an `ActiveSupport::HashWithIndifferentAccess` out of its receiver:

```
{a: 1}.with_indifferent_access["a"] # => 1
```

NOTE: Defined in `active_support/core_ext/hash/indifferent_access.rb`.

## Extensions to Regexp

### multiline?

The method `multiline?` says whether a regexp has the `/m` flag set, that is, whether the dot matches newlines.

```
%r{.}.multiline? # => false
%r{.}m.multiline? # => true

Regexp.new('.').multiline? # => false
Regexp.new('.', Regexp::MULTILINE).multiline? # => true
```

Rails uses this method in a single place, also in the routing code. Multiline regexps are disallowed for route requirements and this flag eases enforcing that constraint.

```
def verify_regexp_requirements(requirements)
  # ...
  if requirement.multiline?
    raise ArgumentError, "Regexp multiline option is not allowed in routing requirements: #{requirement.inspect}"
  end
  # ...
end
```

NOTE: Defined in `active_support/core_ext/regexp.rb`.

## Extensions to Range

### to\_s

Active Support extends the method `Range#to_s` so that it understands an optional format argument. As of this writing the only supported non-default format is `:db`:

```
(Date.today..Date.tomorrow).to_s
# => "2009-10-25..2009-10-26"

(Date.today..Date.tomorrow).to_s(:db)
# => "BETWEEN '2009-10-25' AND '2009-10-26'"
```

As the example depicts, the `:db` format generates a `BETWEEN` SQL clause. That is used by Active Record in its support for range values in conditions.

NOTE: Defined in `active_support/core_ext/range/conversions.rb`.

### === and include?

The methods `Range#===` and `Range#include?` say whether some value falls between the ends of a given instance:

```
(2..3).include?(Math::E) # => true
```

Active Support extends these methods so that the argument may be another range in turn. In that case we test whether the ends of the argument range belong to the receiver themselves:

```

(1..10) === (3..7) # => true
(1..10) === (0..7) # => false
(1..10) === (3..11) # => false
(1...9) === (3..9) # => false

(1..10).include?(3..7) # => true
(1..10).include?(0..7) # => false
(1..10).include?(3..11) # => false
(1...9).include?(3..9) # => false

```

NOTE: Defined in `active_support/core_ext/range/compare_range.rb` .

### overlaps?

The method [Range#overlaps?](#) says whether any two given ranges have non-void intersection:

```

(1..10).overlaps?(7..11) # => true
(1..10).overlaps?(0..7) # => true
(1..10).overlaps?(11..27) # => false

```

NOTE: Defined in `active_support/core_ext/range/overlaps.rb` .

## Extensions to Date

### Calculations

INFO: The following calculation methods have edge cases in October 1582, since days 5..14 just do not exist. This guide does not document their behavior around those days for brevity, but it is enough to say that they do what you would expect. That is, `Date.new(1582, 10, 4).tomorrow` returns `Date.new(1582, 10, 15)` and so on. Please check `test/core_ext/date_ext_test.rb` in the Active Support test suite for expected behavior.

#### Date.current

Active Support defines [Date.current](#) to be today in the current time zone. That's like `Date.today` , except that it honors the user time zone, if defined. It also defines [Date.yesterday](#) and [Date.tomorrow](#) , and the instance predicates [past?](#) , [today?](#) , [tomorrow?](#) , [next\\_day?](#) , [yesterday?](#) , [prev\\_day?](#) , [future?](#) , [on\\_weekday?](#) and [on\\_weekend?](#) , all of them relative to `Date.current` .

When making Date comparisons using methods which honor the user time zone, make sure to use `Date.current` and not `Date.today` . There are cases where the user time zone might be in the future compared to the system time zone, which `Date.today` uses by default. This means `Date.today` may equal `Date.yesterday` .

NOTE: Defined in `active_support/core_ext/date/calculations.rb` .

### Named dates

#### beginning\_of\_week , end\_of\_week

The methods [beginning\\_of\\_week](#) and [end\\_of\\_week](#) return the dates for the beginning and end of the week, respectively. Weeks are assumed to start on Monday, but that can be changed passing an argument, setting thread local `Date.beginning_of_week` or [config.beginning\\_of\\_week](#) .

```
d = Date.new(2010, 5, 8)      # => Sat, 08 May 2010
d.beginning_of_week          # => Mon, 03 May 2010
d.beginning_of_week(:sunday) # => Sun, 02 May 2010
d.end_of_week                # => Sun, 09 May 2010
d.end_of_week(:sunday)      # => Sat, 08 May 2010
```

`beginning_of_week` is aliased to [at\\_beginning\\_of\\_week](#) and `end_of_week` is aliased to [at\\_end\\_of\\_week](#).

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

#### **`monday` , `sunday`**

The methods [monday](#) and [sunday](#) return the dates for the previous Monday and next Sunday, respectively.

```
d = Date.new(2010, 5, 8)      # => Sat, 08 May 2010
d.monday                      # => Mon, 03 May 2010
d.sunday                      # => Sun, 09 May 2010

d = Date.new(2012, 9, 10)     # => Mon, 10 Sep 2012
d.monday                      # => Mon, 10 Sep 2012

d = Date.new(2012, 9, 16)     # => Sun, 16 Sep 2012
d.sunday                      # => Sun, 16 Sep 2012
```

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

#### **`prev_week` , `next_week`**

The method [next\\_week](#) receives a symbol with a day name in English (default is the thread local [Date.beginning\\_of\\_week](#), or [config.beginning\\_of\\_week](#), or `:monday`) and it returns the date corresponding to that day.

```
d = Date.new(2010, 5, 9)      # => Sun, 09 May 2010
d.next_week                   # => Mon, 10 May 2010
d.next_week(:saturday)       # => Sat, 15 May 2010
```

The method [prev\\_week](#) is analogous:

```
d.prev_week                   # => Mon, 26 Apr 2010
d.prev_week(:saturday)       # => Sat, 01 May 2010
d.prev_week(:friday)         # => Fri, 30 Apr 2010
```

`prev_week` is aliased to [last\\_week](#).

Both `next_week` and `prev_week` work as expected when `Date.beginning_of_week` or `config.beginning_of_week` are set.

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

#### **`beginning_of_month` , `end_of_month`**

The methods `beginning_of_month` and `end_of_month` return the dates for the beginning and end of the month:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_month    # => Sat, 01 May 2010
d.end_of_month          # => Mon, 31 May 2010
```

`beginning_of_month` is aliased to `at_beginning_of_month`, and `end_of_month` is aliased to `at_end_of_month`.

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

#### **`beginning_of_quarter`, `end_of_quarter`**

The methods `beginning_of_quarter` and `end_of_quarter` return the dates for the beginning and end of the quarter of the receiver's calendar year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_quarter   # => Thu, 01 Apr 2010
d.end_of_quarter         # => Wed, 30 Jun 2010
```

`beginning_of_quarter` is aliased to `at_beginning_of_quarter`, and `end_of_quarter` is aliased to `at_end_of_quarter`.

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

#### **`beginning_of_year`, `end_of_year`**

The methods `beginning_of_year` and `end_of_year` return the dates for the beginning and end of the year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_year      # => Fri, 01 Jan 2010
d.end_of_year            # => Fri, 31 Dec 2010
```

`beginning_of_year` is aliased to `at_beginning_of_year`, and `end_of_year` is aliased to `at_end_of_year`.

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

### **Other Date Computations**

#### **`years_ago`, `years_since`**

The method `years_ago` receives a number of years and returns the same date those many years ago:

```
date = Date.new(2010, 6, 7)
date.years_ago(10) # => Wed, 07 Jun 2000
```

`years_since` moves forward in time:

```
date = Date.new(2010, 6, 7)
date.years_since(10) # => Sun, 07 Jun 2020
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2012, 2, 29).years_ago(3)    # => Sat, 28 Feb 2009
Date.new(2012, 2, 29).years_since(3)  # => Sat, 28 Feb 2015
```

`last_year` is short-hand for `#years_ago(1)` .

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb` .

#### **months\_ago , months\_since**

The methods `months_ago` and `months_since` work analogously for months:

```
Date.new(2010, 4, 30).months_ago(2)   # => Sun, 28 Feb 2010
Date.new(2010, 4, 30).months_since(2)  # => Wed, 30 Jun 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2010, 4, 30).months_ago(2)   # => Sun, 28 Feb 2010
Date.new(2009, 12, 31).months_since(2) # => Sun, 28 Feb 2010
```

`last_month` is short-hand for `#months_ago(1)` .

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb` .

#### **weeks\_ago**

The method `weeks_ago` works analogously for weeks:

```
Date.new(2010, 5, 24).weeks_ago(1)    # => Mon, 17 May 2010
Date.new(2010, 5, 24).weeks_ago(2)    # => Mon, 10 May 2010
```

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb` .

#### **advance**

The most generic way to jump to other days is `advance` . This method receives a hash with keys `:years` , `:months` , `:weeks` , `:days` , and returns a date advanced as much as the present keys indicate:

```
date = Date.new(2010, 6, 6)
date.advance(years: 1, weeks: 2) # => Mon, 20 Jun 2011
date.advance(months: 2, days: -2) # => Wed, 04 Aug 2010
```

Note in the previous example that increments may be negative.

To perform the computation the method first increments years, then months, then weeks, and finally days. This order is important towards the end of months. Say for example we are at the end of February of 2010, and we want to move one month and one day forward.

The method `advance` advances first one month, and then one day, the result is:



```
Date.new(2010, 2, 28).advance(months: 1, days: 1)
# => Sun, 29 Mar 2010
```

While if it did it the other way around the result would be different:

```
Date.new(2010, 2, 28).advance(days: 1).advance(months: 1)
# => Thu, 01 Apr 2010
```

NOTE: Defined in `active_support/core_ext/date/calculations.rb`.

## Changing Components

The method `change` allows you to get a new date which is the same as the receiver except for the given year, month, or day:

```
Date.new(2010, 12, 23).change(year: 2011, month: 11)
# => Wed, 23 Nov 2011
```

This method is not tolerant to non-existing dates, if the change is invalid `ArgumentError` is raised:

```
Date.new(2010, 1, 31).change(month: 2)
# => ArgumentError: invalid date
```

NOTE: Defined in `active_support/core_ext/date/calculations.rb`.

## Durations

`Duration` objects can be added to and subtracted from dates:

```
d = Date.current
# => Mon, 09 Aug 2010
d + 1.year
# => Tue, 09 Aug 2011
d - 3.hours
# => Sun, 08 Aug 2010 21:00:00 UTC +00:00
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
Date.new(1582, 10, 4) + 1.day
# => Fri, 15 Oct 1582
```

## Timestamps

INFO: The following methods return a `Time` object if possible, otherwise a `DateTime`. If set, they honor the user time zone.

`beginning_of_day`, `end_of_day`

The method `beginning_of_day` returns a timestamp at the beginning of the day (00:00:00):

```
date = Date.new(2010, 6, 7)
date.beginning_of_day # => Mon Jun 07 00:00:00 +0200 2010
```

The method `end_of_day` returns a timestamp at the end of the day (23:59:59):

```
date = Date.new(2010, 6, 7)
date.end_of_day # => Mon Jun 07 23:59:59 +0200 2010
```

`beginning_of_day` is aliased to `at_beginning_of_day`, `midnight`, `at_midnight`.

NOTE: Defined in `active_support/core_ext/date/calculations.rb`.

**`beginning_of_hour`, `end_of_hour`**

The method `beginning_of_hour` returns a timestamp at the beginning of the hour (hh:00:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_hour # => Mon Jun 07 19:00:00 +0200 2010
```

The method `end_of_hour` returns a timestamp at the end of the hour (hh:59:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_hour # => Mon Jun 07 19:59:59 +0200 2010
```

`beginning_of_hour` is aliased to `at_beginning_of_hour`.

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

**`beginning_of_minute`, `end_of_minute`**

The method `beginning_of_minute` returns a timestamp at the beginning of the minute (hh:mm:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_minute # => Mon Jun 07 19:55:00 +0200 2010
```

The method `end_of_minute` returns a timestamp at the end of the minute (hh:mm:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_minute # => Mon Jun 07 19:55:59 +0200 2010
```

`beginning_of_minute` is aliased to `at_beginning_of_minute`.

INFO: `beginning_of_hour`, `end_of_hour`, `beginning_of_minute`, and `end_of_minute` are implemented for `Time` and `DateTime` but **not** `Date` as it does not make sense to request the beginning or end of an hour or minute on a `Date` instance.

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

**`ago`, `since`**

The method `ago` receives a number of seconds as argument and returns a timestamp those many seconds ago from midnight:

```
date = Date.current # => Fri, 11 Jun 2010
date.ago(1)         # => Thu, 10 Jun 2010 23:59:59 EDT -04:00
```

Similarly, `since` moves forward:

```
date = Date.current # => Fri, 11 Jun 2010
date.since(1)       # => Fri, 11 Jun 2010 00:00:01 EDT -04:00
```

NOTE: Defined in `active_support/core_ext/date/calculations.rb`.

## Extensions to `DateTime`

WARNING: `DateTime` is not aware of DST rules and so some of these methods have edge cases when a DST change is going on. For example `seconds_since_midnight` might not return the real amount in such a day.

### Calculations

The class `DateTime` is a subclass of `Date` so by loading

`active_support/core_ext/date/calculations.rb` you inherit these methods and their aliases, except that they will always return datetimes.

The following methods are reimplemented so you do **not** need to load

`active_support/core_ext/date/calculations.rb` for these ones:

- `beginning_of_day` / `midnight` / `at_midnight` / `at_beginning_of_day`
- `end_of_day`
- `ago`
- `since` / `in`

On the other hand, `advance` and `change` are also defined and support more options, they are documented below.

The following methods are only implemented in `active_support/core_ext/date_time/calculations.rb` as they only make sense when used with a `DateTime` instance:

- `beginning_of_hour` / `at_beginning_of_hour`
- `end_of_hour`

### Named Datetimes

`DateTime.current`

Active Support defines `DateTime.current` to be like `Time.now.to_datetime`, except that it honors the user time zone, if defined. The instance predicates `past?` and `future?` are defined relative to

`DateTime.current`.

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

### Other Extensions

### `seconds_since_midnight`

The method `seconds_since_midnight` returns the number of seconds since midnight:

```
now = DateTime.current      # => Mon, 07 Jun 2010 20:26:36 +0000
now.seconds_since_midnight # => 73596
```

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

### `utc`

The method `utc` gives you the same datetime in the receiver expressed in UTC.

```
now = DateTime.current # => Mon, 07 Jun 2010 19:27:52 -0400
now.utc                # => Mon, 07 Jun 2010 23:27:52 +0000
```

This method is also aliased as `getutc`.

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

### `utc?`

The predicate `utc?` says whether the receiver has UTC as its time zone:

```
now = DateTime.now # => Mon, 07 Jun 2010 19:30:47 -0400
now.utc?           # => false
now.utc.utc?       # => true
```

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

### `advance`

The most generic way to jump to another datetime is `advance`. This method receives a hash with keys `:years`, `:months`, `:weeks`, `:days`, `:hours`, `:minutes`, and `:seconds`, and returns a datetime advanced as much as the present keys indicate.

```
d = DateTime.current
# => Thu, 05 Aug 2010 11:33:31 +0000
d.advance(years: 1, months: 1, days: 1, hours: 1, minutes: 1, seconds: 1)
# => Tue, 06 Sep 2011 12:34:32 +0000
```

This method first computes the destination date passing `:years`, `:months`, `:weeks`, and `:days` to `Date#advance` documented above. After that, it adjusts the time calling `since` with the number of seconds to advance. This order is relevant, a different ordering would give different datetimes in some edge-cases. The example in `Date#advance` applies, and we can extend it to show order relevance related to the time bits.

If we first move the date bits (that have also a relative order of processing, as documented before), and then the time bits we get for example the following computation:

```
d = DateTime.new(2010, 2, 28, 23, 59, 59)
# => Sun, 28 Feb 2010 23:59:59 +0000
d.advance(months: 1, seconds: 1)
# => Mon, 29 Mar 2010 00:00:00 +0000
```

but if we computed them the other way around, the result would be different:

```
d.advance(seconds: 1).advance(months: 1)
# => Thu, 01 Apr 2010 00:00:00 +0000
```

WARNING: Since `DateTime` is not DST-aware you can end up in a non-existing point in time with no warning or error telling you so.

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

## Changing Components

The method `change` allows you to get a new datetime which is the same as the receiver except for the given options, which may include `:year`, `:month`, `:day`, `:hour`, `:min`, `:sec`, `:offset`, `:start`:

```
now = DateTime.current
# => Tue, 08 Jun 2010 01:56:22 +0000
now.change(year: 2011, offset: Rational(-6, 24))
# => Wed, 08 Jun 2011 01:56:22 -0600
```

If hours are zeroed, then minutes and seconds are too (unless they have given values):

```
now.change(hour: 0)
# => Tue, 08 Jun 2010 00:00:00 +0000
```

Similarly, if minutes are zeroed, then seconds are too (unless it has given a value):

```
now.change(min: 0)
# => Tue, 08 Jun 2010 01:00:00 +0000
```

This method is not tolerant to non-existing dates, if the change is invalid `ArgumentError` is raised:

```
DateTime.current.change(month: 2, day: 30)
# => ArgumentError: invalid date
```

NOTE: Defined in `active_support/core_ext/date_time/calculations.rb`.

## Durations

[Duration](#) objects can be added to and subtracted from datetimes:

```
now = DateTime.current
# => Mon, 09 Aug 2010 23:15:17 +0000
now + 1.year
# => Tue, 09 Aug 2011 23:15:17 +0000
now - 1.week
# => Mon, 02 Aug 2010 23:15:17 +0000
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
DateTime.new(1582, 10, 4, 23) + 1.hour
# => Fri, 15 Oct 1582 00:00:00 +0000
```

## Extensions to `Time`

### Calculations

They are analogous. Please refer to their documentation above and take into account the following differences:

- `change` accepts an additional `:usec` option.
- `Time` understands DST, so you get correct DST calculations as in

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>

# In Barcelona, 2010/03/28 02:00 +0100 becomes 2010/03/28 03:00 +0200 due to DST.
t = Time.local(2010, 3, 28, 1, 59, 59)
# => Sun Mar 28 01:59:59 +0100 2010
t.advance(seconds: 1)
# => Sun Mar 28 03:00:00 +0200 2010
```

- If `since` or `ago` jumps to a time that can't be expressed with `Time` a `DateTime` object is returned instead.

#### `Time.current`

Active Support defines `Time.current` to be today in the current time zone. That's like `Time.now`, except that it honors the user time zone, if defined. It also defines the instance predicates `past?`, `today?`, `tomorrow?`, `next_day?`, `yesterday?`, `prev_day?` and `future?`, all of them relative to `Time.current`.

When making `Time` comparisons using methods which honor the user time zone, make sure to use `Time.current` instead of `Time.now`. There are cases where the user time zone might be in the future compared to the system time zone, which `Time.now` uses by default. This means `Time.now.to_date` may equal `Date.yesterday`.

NOTE: Defined in `active_support/core_ext/time/calculations.rb`.

#### `all_day`, `all_week`, `all_month`, `all_quarter`, and `all_year`

The method `all_day` returns a range representing the whole day of the current time.

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_day
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Mon, 09 Aug 2010 23:59:59 UTC +00:00
```

Analogously, `all_week`, `all_month`, `all_quarter` and `all_year` all serve the purpose of generating time ranges.

```

now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_week
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Sun, 15 Aug 2010 23:59:59 UTC +00:00
now.all_week(:sunday)
# => Sun, 16 Sep 2012 00:00:00 UTC +00:00..Sat, 22 Sep 2012 23:59:59 UTC +00:00
now.all_month
# => Sat, 01 Aug 2010 00:00:00 UTC +00:00..Tue, 31 Aug 2010 23:59:59 UTC +00:00
now.all_quarter
# => Thu, 01 Jul 2010 00:00:00 UTC +00:00..Thu, 30 Sep 2010 23:59:59 UTC +00:00
now.all_year
# => Fri, 01 Jan 2010 00:00:00 UTC +00:00..Fri, 31 Dec 2010 23:59:59 UTC +00:00

```

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

#### **prev\_day , next\_day**

[prev\\_day](#) and [next\\_day](#) return the time in the last or next day:

```

t = Time.new(2010, 5, 8) # => 2010-05-08 00:00:00 +0900
t.prev_day               # => 2010-05-07 00:00:00 +0900
t.next_day               # => 2010-05-09 00:00:00 +0900

```

NOTE: Defined in `active_support/core_ext/time/calculations.rb`.

#### **prev\_month , next\_month**

[prev\\_month](#) and [next\\_month](#) return the time with the same day in the last or next month:

```

t = Time.new(2010, 5, 8) # => 2010-05-08 00:00:00 +0900
t.prev_month             # => 2010-04-08 00:00:00 +0900
t.next_month             # => 2010-06-08 00:00:00 +0900

```

If such a day does not exist, the last day of the corresponding month is returned:

```

Time.new(2000, 5, 31).prev_month # => 2000-04-30 00:00:00 +0900
Time.new(2000, 3, 31).prev_month # => 2000-02-29 00:00:00 +0900
Time.new(2000, 5, 31).next_month # => 2000-06-30 00:00:00 +0900
Time.new(2000, 1, 31).next_month # => 2000-02-29 00:00:00 +0900

```

NOTE: Defined in `active_support/core_ext/time/calculations.rb`.

#### **prev\_year , next\_year**

[prev\\_year](#) and [next\\_year](#) return a time with the same day/month in the last or next year:

```

t = Time.new(2010, 5, 8) # => 2010-05-08 00:00:00 +0900
t.prev_year              # => 2009-05-08 00:00:00 +0900
t.next_year              # => 2011-05-08 00:00:00 +0900

```

If date is the 29th of February of a leap year, you obtain the 28th:

```
t = Time.new(2000, 2, 29) # => 2000-02-29 00:00:00 +0900
t.prev_year               # => 1999-02-28 00:00:00 +0900
t.next_year               # => 2001-02-28 00:00:00 +0900
```

NOTE: Defined in `active_support/core_ext/time/calculations.rb`.

#### `prev_quarter` , `next_quarter`

`prev_quarter` and `next_quarter` return the date with the same day in the previous or next quarter:

```
t = Time.local(2010, 5, 8) # => 2010-05-08 00:00:00 +0300
t.prev_quarter             # => 2010-02-08 00:00:00 +0200
t.next_quarter             # => 2010-08-08 00:00:00 +0300
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Time.local(2000, 7, 31).prev_quarter # => 2000-04-30 00:00:00 +0300
Time.local(2000, 5, 31).prev_quarter # => 2000-02-29 00:00:00 +0200
Time.local(2000, 10, 31).prev_quarter # => 2000-07-31 00:00:00 +0300
Time.local(2000, 11, 31).next_quarter # => 2001-03-01 00:00:00 +0200
```

`prev_quarter` is aliased to `last_quarter`.

NOTE: Defined in `active_support/core_ext/date_and_time/calculations.rb`.

## Time Constructors

Active Support defines `Time.current` to be `Time.zone.now` if there's a user time zone defined, with fallback to `Time.now`:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.current
# => Fri, 06 Aug 2010 17:11:58 CEST +02:00
```

Analogously to `DateTime`, the predicates `past?`, and `future?` are relative to `Time.current`.

If the time to be constructed lies beyond the range supported by `Time` in the runtime platform, usecs are discarded and a `DateTime` object is returned instead.

## Durations

`Duration` objects can be added to and subtracted from time objects:

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now + 1.year
# => Tue, 09 Aug 2011 23:21:11 UTC +00:00
```



```
now - 1.week
# => Mon, 02 Aug 2010 23:21:11 UTC +00:00
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
Time.utc(1582, 10, 3) + 5.days
# => Mon Oct 18 00:00:00 UTC 1582
```

## Extensions to `File`

### `atomic_write`

With the class method `File.atomic_write` you can write to a file in a way that will prevent any reader from seeing half-written content.

The name of the file is passed as an argument, and the method yields a file handle opened for writing. Once the block is done `atomic_write` closes the file handle and completes its job.

For example, Action Pack uses this method to write asset cache files like `all.css`:

```
File.atomic_write(joined_asset_path) do |cache|
  cache.write(join_asset_file_contents(asset_paths))
end
```

To accomplish this `atomic_write` creates a temporary file. That's the file the code in the block actually writes to. On completion, the temporary file is renamed, which is an atomic operation on POSIX systems. If the target file exists `atomic_write` overwrites it and keeps owners and permissions. However there are a few cases where `atomic_write` cannot change the file ownership or permissions, this error is caught and skipped over trusting in the user/filesystem to ensure the file is accessible to the processes that need it.

NOTE. Due to the `chmod` operation `atomic_write` performs, if the target file has an ACL set on it this ACL will be recalculated/modified.

WARNING. Note you can't append with `atomic_write`.

The auxiliary file is written in a standard directory for temporary files, but you can pass a directory of your choice as second argument.

NOTE: Defined in `active_support/core_ext/file/atomic.rb`.

## Extensions to `NameError`

Active Support adds `missing_name?` to `NameError`, which tests whether the exception was raised because of the name passed as argument.

The name may be given as a symbol or string. A symbol is tested against the bare constant name, a string is against the fully qualified constant name.

TIP: A symbol can represent a fully qualified constant name as in `: "ActiveRecord::Base"`, so the behavior for symbols is defined for convenience, not because it has to be that way technically.

For example, when an action of `ArticlesController` is called Rails tries optimistically to use `ArticlesHelper`. It is OK that the helper module does not exist, so if an exception for that constant name is raised it should be silenced. But it could be the case that `articles_helper.rb` raises a `NameError` due to an actual unknown constant. That should be reraised. The method `missing_name?` provides a way to distinguish both cases:

```
def default_helper_module!  
  module_name = name.delete_suffix("Controller")  
  module_path = module_name.underscore  
  helper module_path  
rescue LoadError => e  
  raise e unless e.is_missing? "helpers/#{module_path}_helper"  
rescue NameError => e  
  raise e unless e.missing_name? "#{module_name}Helper"  
end
```

NOTE: Defined in `active_support/core_ext/name_error.rb`.

## Extensions to `LoadError`

Active Support adds `is_missing?` to `LoadError`.

Given a path name `is_missing?` tests whether the exception was raised due to that particular file (except perhaps for the ".rb" extension).

For example, when an action of `ArticlesController` is called Rails tries to load `articles_helper.rb`, but that file may not exist. That's fine, the helper module is not mandatory so Rails silences a load error. But it could be the case that the helper module does exist and in turn requires another library that is missing. In that case Rails must reraise the exception. The method `is_missing?` provides a way to distinguish both cases:

```
def default_helper_module!  
  module_name = name.delete_suffix("Controller")  
  module_path = module_name.underscore  
  helper module_path  
rescue LoadError => e  
  raise e unless e.is_missing? "helpers/#{module_path}_helper"  
rescue NameError => e  
  raise e unless e.missing_name? "#{module_name}Helper"  
end
```

NOTE: Defined in `active_support/core_ext/load_error.rb`.

## Extensions to `Pathname`

### `existence`

The `existence` method returns the receiver if the named file exists otherwise returns `+nil+`. It is useful for idioms like this:

```
content = Pathname.new("file").existence&.read
```

NOTE: Defined in `active_support/core_ext/pathname/existence.rb` .