

The
Pragmatic
Programmers

Programming Ruby 3.3



The Pragmatic Programmers' Guide

Noel Rappin
with Dave Thomas
Edited by Katharine Dvorak

The Facets  of Ruby Series

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit <https://www.pragprog.com>.

Copyright © The Pragmatic Programmers, LLC.

When you see a variable in your code, it's useful to know what values can be assigned to that variable without the code breaking. For example, you might have the following Ruby method:

```
def mystery_method(x)
  x * 3
end
```

You'd likely expect that `x` should be a number. But it's also completely valid Ruby for `x` to be a string ("`a`" * 3 resolves to "`aaa`") or an array (`[:a]` * 3 resolves to `[:a, :a, :a]`).

Let's say that this method is in your code, and over time people call `mystery_method` with strings, integers, floating-point numbers, and so on, until somebody changes the method and inadvertently changes what variables it'll accept. Here's an example:

```
def mystery_method(x)
  x.abs * 3
end
```

Now all of those string and array uses break because `abs` isn't defined for strings and arrays. If the original developer had been able to specify that `x` must be numeric, then the string and array uses would've found some other place to multiply by three and wouldn't have broken when the method changed.

Historically, Ruby has gotten along just fine without requiring or allowing developers to augment code with this kind of information about the expected values of a variable, which is often called the *type* of the variable.

Spurred by the increasing complexity of large Ruby projects and the possibility of improved performance, Ruby 3.0 added RBS, a mechanism for allowing developers to specify type information about classes and methods. In addition, a third-party tool called Sorbet provides a separate mechanism for type control in Ruby. In this chapter, we take a look at both RBS and Sorbet. We'll also briefly look at a gem called Literal, which provides some type checking with less complexity than RBS or Sorbet.

What's a Type?

The terminology around types in programming languages can be confusing because each language community uses the terms slightly differently.

Most generally, a variable, attribute, or method argument can be declared as belonging to a type. This *type declaration* limits the set of values that can be assigned to that variable, attribute, or method argument. The type also determines the behavior of the variable within the program and often also determines how the value is stored internally by the programming language. For example, the result of the expression `x / y` depends on the type of `x` and `y`. In many languages, the result will be different if the numbers are integers than if they are floating-point types.

Many programming languages have a set of "basic types" that can be used, often including strings, boolean values, different kinds of numerical values, and so on. Ruby doesn't have basic types. Every variable in Ruby is an instance of a class, and that class determines the behavior of the variable. In Ruby, `x / y` is equivalent to the method call `x./(y)`, and the behavior depends on what class `x` is.

In many programming languages, you must declare the type of a variable before it's used. This is called *explicit typing*. Some programming languages can infer the type of a variable from its first use, so if I write 'let x = 3' in TypeScript, TypeScript knows that 'x' is a number. This is called *type inference*.

In either case, there is usually a tool, often part of the compiler, that evaluates every variable interaction to see if type information is followed. If, later in the TypeScript code, I try to say x = "foo", TypeScript will give a compilation error because "foo" is a string and TypeScript thinks x needs to be an integer. This is called *static typing*.

Without type information, Ruby doesn't do static typing. In Ruby, the type of a variable is determined while the code is running by the values that are assigned to it, and Ruby determines if the variable can receive a method only at runtime. This is called *dynamic typing*, and the process of determining the behavior of the method at the last possible moment is called *late binding*.

There's another distinction around typing that isn't as useful to us. In some languages, the type barriers are more permeable, and if you type 3 + "3", the language will automatically coerce the string to an integer and allow the addition to continue. This is called *weak typing*, and languages that don't do this have *strong typing*. You'll sometimes see "strong typing" incorrectly used as a synonym for "static typing," but these are two different concepts and it's useful to keep them separate.

These are some of the benefits of static typing and a compilation step that validates all assignments:

- If the compiler and runtime know information about what type to expect, they can often optimize internal behavior and improve performance.
- A person reading the code can get more information about the intent and behavior of the code if there is type information.
- A developer tool like an IDE or editor can use type information to provide information to the developer as the code is being written.

These are some of the drawbacks:

- Statically typed code is usually more verbose than dynamic code. Though type inferencing has improved this situation.
- Sometimes a developer has to spend time convincing the type system that the code that has been written is correct.
- Statically typed code is often less flexible than dynamic code and harder to change. (To be fair, lots of people would see this as an advantage.)

The goal of the type systems in Ruby is to allow for as many of the benefits of typed languages as we can get without giving up the flexibility that makes Ruby, Ruby.

Official Ruby Typing with RBS

The official Ruby typing system is called RBS (short for Ruby Signature). With RBS, you create a separate file that contains type signature information for all or part of your code.

Writing RBS

To take a look at how RBS works, we'll use the gem we created in [Writing and Packaging Your Own Code into Gems, on page 7](#), and augment it with RBS typing. If you look at the `Aaagmnr` gem code, you'll see that it contains a directory named `sig` that we didn't talk much about. That directory is where you're supposed to put the type information, and right now it contains one file:

```
gems/aaagmnr/sig/aaagmnr.rbs
```

```
module Aaagmnr
  VERSION: String
  # See the writing guide of rbs: https://github.com/ruby/rbs#guides
end
```

The only thing this file tells us is that the module `Aaagmnr` has a `VERSION` constant, which is a `String`. True enough, but not useful.

Here's what an RBS file for the entire gem looks like:

```
typed_ruby/aaagmnr/sig/aaagmnr.rbs
```

```
module Aaagmnr
  class Error
  end

  class Finder
    @signatures: Hash[String, Array[String]]
    def self.from_file: (String file_name) -> Finder
    def initialize: (Array[String] dictionary_words) -> void
    def lookup: (String word) -> Array[String]
    def signature_of: (String word) -> String
  end

  class Options
    attr_reader dictionary: String
    attr_reader words_to_find: Array[String]

    def initialize: (Array[String] argv) -> void
    private def parse: (Array[String] argv) -> void
  end

  class Runner
    @options: Options
    def initialize: (Array[String] argv) -> void
    def run: () -> void
  end

  VERSION: String
end
```

The goal here is to describe the expected types of all the modules, constants, attributes, and methods in the gem. The syntax is meant to be similar enough to Ruby to be readable, while still providing for type information.

A full description of the syntax may be found at <https://github.com/ruby/rbs/blob/master/docs/syntax.md>. We'll talk about the most common usages here.

The `.rbs` file typically combines one or more entire modules into a single file, but, as with typical Ruby, there's nothing preventing you from splitting the file up as you please.

The basic structure of module and class declarations has the same syntax as regular Ruby and the side effect of allowing those constant names to be used as type names. In other words, the same way we use `String` as a type name, after declaring class `Finder`, we could also use `Finder` as a type in a method argument or return value or wherever.

Inside each class, this gem has two different kinds of declarations.

We declare attributes and constants, including `'VERSION: String'`. The `'Finder'` class declares `'@signatures: Hash[String, Array[String]]'`, meaning it expects to have an instance variable called `'@signatures'` and the type of that variable is a `'Hash'` whose keys are of type `'String'` and whose values are of type `'Array[String]'`. The general syntax is the name of the instance variable, followed by a colon and then by the type. The square bracket syntax here is called a `_generic_` (more on this in [Advanced RBS Syntax, on page ?](#)), and it allows us to define both the type of a container and the type of objects in the container, so `'Array[String]'` is an `'Array'` container where each element is a `'String'`.

In general, any time you see a type in RBS, you can add a `?` to the end of it to indicate that the type is *optional*, meaning that the value can be `nil`. So, `@name: String` means the name has to be a string, but `@name: String?` means the name can be a string or can be `nil`.

The lines `attr_reader dictionary: Array[String]` and `attr_reader words_to_find: Array[String]` are also attribute declarations. Similar to how Ruby code works, `attr_reader` is both a shortcut for declaring the type of an instance variable and a getter method. The related declaration `attr_writer` declares the type of the instance variable and the setter method, and `attr_accessor` declares all three. The syntax is: the kind of declaration, the name of the attribute, a colon, and the type.

The rest of the lines in the file are type signatures of methods. For example, `def lookup: (String word) -> Array[String]` tells us that the `lookup` method takes a positional argument named `word` of type `String` and returns an array of strings.

The general syntax here is `def` followed by the name of the method, a colon, the attributes inside parentheses, the `->` arrow, and the return type.

The attribute listing has a few variants. As that declaration shows, positional arguments have the type first followed by the variable name—the name is optional and isn't checked against the name in the actual code. The keyword arguments are in a different order: name, colon, and then type. So `def lookup: (word: String) -> Array[String]` would indicate that `word` is a keyword argument. Keyword arguments are checked against the actual Ruby method signature.

An optional argument is denoted with a `?` prefix, so `def lookup: (?String)` is a method with an optional positional argument, but if the argument is specified, it can't be `nil`. The two kinds of optional can be combined: `def lookup: (?String?)` is a method that takes an argument that's both optional and can take a `nil` value.

Before we talk about more complex RBS syntax, let's take a look at how you can use RBS.

Using RBS

Having taken the effort to create these type annotations, what can we do with them? Well, there are two answers:

- There are some command-line tools that will do static analysis of your Ruby code. For example, based on the RBS files, a tool might find cases where the code doesn't match the type information, indicating a potential bug.
- Depending on the editor or development environment you're using, the tool may be able to use the RBS files to provide hints or real-time error analysis as you type. RubyMine provides significant support for RBS files.

The Ruby interpreter could also use RBS information to optimize code generation, it seems as though more on that line of work is yet to come.

RBS offers its own command-line tools. You may need to gem install rbs to get access to them.

The RBS command line are generally proof-of-concept tools or tools that might be used by an editor or something other than directly by the developer.

One exception is the rbs prototype GENERATOR FILE_PATTERNS tool, which generates basic RBS files for any file that matches the pattern. The generator is one of rb, for Ruby files, rbi for Sorbet RBI files (see [Ruby Typing with Sorbet, on page ?](#)), or runtime for the runtime API.

For example, this gives us most of our Ruby files:

```
$ rbs prototype rb lib/aaagmnr/**/*.rb
module Aaagmnr
  class Finder
    @signatures: untyped

    def self.from_file: (untyped file_name) -> untyped
    def initialize: (untyped dictionary_words) -> void
    def lookup: (untyped word) -> untyped
    def signature_of: (untyped word) -> untyped
  end
end

module Aaagmnr
  class Options
    @dictionary: untyped

    @words_to_find: untyped
    DEFAULT_DICTIONARY: "/usr/share/dict/words"
    attr_reader dictionary: untyped
    attr_reader words_to_find: untyped

    def initialize: (untyped argv) -> void
  private
    def parse: (untyped argv) -> untyped
  end
end

module Aaagmnr
  class Runner
    @options: untyped

    def initialize: (untyped argv) -> void
```

```

    def run: () -> untyped
  end
end

module Aaagmnr
  VERSION: "0.1.0"
end

```

This is similar to the RBS file that we created by hand, with a few changes:

- We forgot a couple of constants, like `DEFAULT_DICTIONARY`
- In quite a few cases, RBS can't determine a type and puts in `untyped`. We know that `Finder#lookup` takes a string and returns an array of strings, but `TypeProf` returns `def lookup: (untyped word) -> untyped`, meaning that it can't infer a type for the parameter.

What you get from the boilerplate, then, is a mix of items that we know about the code that RBS can't figure out and occasionally items that RBS can figure out but that we, the developers, didn't necessarily see. This makes 'rbs prototype' a useful way to start with RBS, but not necessarily the completed goal.

The `rbs list` tool gives you a list of classes and modules used by the application. The `rbs ancestors` and `rbs methods` tools both take the name of a class and provide what RBS knows about the ancestors or methods of that class:

```

$ rbs ancestors String
::String
::Comparable
::Object
::Kernel
::BasicObject

```

And the `rbs method` call takes a class and a method name and provides what RBS knows about that method:

```

$rbs method String length
::String#length
  defined_in: ::String
  implementation: ::String
  accessibility: public
  types:
    () -> ::Integer
  at <FULL PATH ELIDED>/string.rbs:2267:14...2267:27

```

In this case, `String#length` takes no arguments and returns an `Integer`.

There are other RBS commands that aren't documented, and which presumably are either not expected to be in use or are not yet complete.

The entire Ruby standard library has RBS files, so you can get type information about any method in that library.

Ruby also provides a tool called `TypeProf`, which can help you generate RBS files. To use this tool, first add `gem "typeprof"` to the `gemfile` of the application you're working on and then `bundle install`.

TypeProf takes a Ruby file, which should be the top-level file of your gem or application, and optionally takes an RBS file, and it then spits out an RBS file. We don't want to spit out a third version of the same RBS file, the TypeProf output should be similar to rbs prototype

As an alternative to running TypeProf from the command line, an experimental plugin for Visual Studio Code (<https://marketplace.visualstudio.com/items?itemName=mame.ruby-typeprof>) will generate method type signatures as you write code.

To generate these type signatures, TypeProf executes your Ruby code...kind of. The phrase the documentation uses is “abstractly executes,” which means that it walks through the code paths knowing the types of variables but not their values.

In other words, TypeProf tracks type information through the code, using variable assignments and what it knows about method calls.

As an example, run the following code through TypeProf:

```
def approximate_word_count(sentence)
  sentence.split(/\W+/).size
end

approximate_word_count("This is a sample word count")
```

TypeProf will infer that `sentence` is a string from the literal assignment, and then it'll walk through the known type signatures in the method. The `split` method takes a string and returns an array of strings, and the `size` method takes an array and returns an integer. So TypeProf deduces that `approximate_word_count` takes a string argument and returns an integer.

TypeProf has some limitations. If there's no call to a method or no assignment in the code, then TypeProf is limited in how much information it has and will only provide limited and probably overly general results. Metaprogramming will often confuse TypeProf, especially if a lot of the data is unknown at load time. For example, TypeProf might be able to manage a `define_method` over a known array, but using `send` where the argument is a variable will confound it.

TypeProf continues to be under active development. An up-to-date description of changes can be found at <https://github.com/ruby/typeprof/blob/master/doc/doc.md>.