

IT 資格 道 場

Ruby技術者認定試験 Silver 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

Rubyアソシエーション「Ruby技術者認定試験」Silver 出題範囲 (Ruby 3.1 対象) 準拠
2026年10月版

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

目次

出題範囲と章立て [overview]

第1章 リテラル・変数・定数・演算子 [1.1]

コメントと数値・文字列のリテラル

シンボル・配列・ハッシュのリテラルと同一性

and・or・not と &&・||・! の優先順位

nil と false だけが偽

変数の種類とスコープ

変数名の規則と予約語

定数の再代入と参照

多重代入

サンプルはここまでです

出題範囲と章立て [overview]

Ruby技術者認定試験 Silver は、Ruby で書かれたプログラムを正しく読み、書くための基礎を問う試験です。運営は一般財団法人 Rubyアソシエーションで、対象は Ruby 3.1 です[^1]。試験の形は次の表のとおりです。

項目	内容
問題数	50問(選択式)
試験時間	90分
合格基準	正答率75%
受験方式	コンピュータ試験(CBT)
対象バージョン	Ruby 3.1

出題範囲は3つの分野です。この教材も同じ順に章を立てます。

分野	章	内容
文法	第1～6章	リテラルと演算子、条件分岐とループ、例外処理、メソッド、ブロック、クラスとモジュールの定義
組み込みライブラリ	第7～15章	Object と Kernel、数値、String、Array、Hash、Enumerable と Comparable、File と Dir、Range、正規表現の基本
オブジェクト指向	第16～18章	ポリモルフィズム、継承、mix-in

公式は分野ごとの出題数を公表していません。

この教材は、コードを読んで頭の中で動かす力をつけるための本です。例のコードは手元の Ruby 3.1 で一度実行して、出力を確かめてください。バージョンが違くと結果が変わる書き方は、Ruby 3.1 の結果だけを載せ、ほかのバージョンの結果は載せません。

SAMPLE

第1章 リテラル・変数・定数・演算子 [1.1]

コメントと数値・文字列のリテラル

リテラルは、ソースコードに直接書いた値の書き方です。`10` や `"abc"` のように、そのまま値になる表記を指します。[^2] Ruby のコメントは `#` から行末までで、複数行は `=begin` と `=end` で囲みます。行の途中に書いた `=begin` は、コメントの開始になりません。

```
# 行全体のコメント
puts "ok" # 行末のコメント
=begin
この間は何行でもコメントです
=end
```

実行すると、コメントは無視されて `ok` だけが表示されます。

整数リテラルには、位取りの区切りと進数の接頭辞があります。`_` は数字の間に自由に入れられ、値には影響しません。`0x` は 16 進数、`0b` は 2 進数、`0o` と先頭の `0` は 8 進数です。小数点や `e` を含むと `Float` (浮動小数点数) になります。

```
p 1_000, 0x1f, 0b101, 0o17, 017, 3.14, 1e3, 1.5e-2
```

```
1000
31
5
15
15
3.14
1000.0
0.015
```

`1_000 + 0x10 + 0b11` は、`1000 + 16 + 3` と同じで `1019` です。`0x10` を 10 進数の 10 と読む人は、接頭辞の意味を取り違えています。`0x` や `0b` は、後ろの数字をその進数で読む合図なので、無視してはいけません。

文字列リテラルは、一重引用符 `'...'` と二重引用符 `"..."` で意味が変わります。二重引用符の中だけで、`\n` (改行) などのエスケープと、`{式}` の式展開が働きます。一重引用符の中では、エスケープも式展開も働きません。例外は `\'` と `\\` の 2 つだけで、前者は引用符 1 文字に、後者は `\\` 1 文字になります。

```
x = 3
puts '#{x}' + '#{x}'
p 'a\nb', "a\nb"
```

```
#{x}3
"a\nb"
"a\nb"
```

一重引用符の `'#{x}'` は展開されず、文字として `#{x}` が残ります。二重引用符の `"#{x}"` は 3 になるので、つなげると `#{x}3` です。p の出力に `\\` が 2 つ並ぶのは `inspect` の形のためで、実際の文字列に `\` は 1 つしかありません。式展開の中には任意の式を書けて、`"#{1 + 2}"` は `"3"` になります。展開は二重引用符の中に限られ、一重引用符では起きません。

`%w[a b c]` は文字列の配列 `["a", "b", "c"]` を作る書き方です。`%i[a b]` はシンボルの配列を作ります。`%q()` は一重引用符、`%Q()` は二重引用符と同じ規則の文字列です。

文字列リテラルは、評価のたびに新しい `String` オブジェクトを作ります。この性質があるので、次の節の同一性の比較は、同じ内容でも `true` になるとは限りません。ファイルの先頭に `# frozen_string_literal: true` と書くと、性質が変わります。このコメントはマジックコメントと呼ばれ、そのファイルの文字列リテラルを凍結(変更不可)にします。

```
# frozen_string_literal: true
p "ruby".equal?("ruby")
p "ruby".frozen?
begin
  "abc" << "d"
rescue => e
  p e.class
end
```

```
true
true
FrozenError
```

マジックコメントがあると、同じ内容の文字列リテラルは同じオブジェクトになり、`<<` で書き換えようとするとき `FrozenError` が起きます。マジックコメントが無ければ、文字列リテラルは凍結されず、評価のたびに別のオブジェクトです。式展開を含む `"a#{1}"` は、マジックコメントがあっても凍結されません。

シンボル・配列・ハッシュのリテラルと同一性

シンボルは `:ruby` のように、コロンに名前を続けて書く値です。ひとこと言くと、名前そのものを表す軽い値です。文字列に似ていますが、**同じ名前のシンボルは、プログラム中で常に同じ 1 つのオブジェクト**です。文字列は、同じ内容でも書くたびに別のオブジェクトになり、同一にはなりません。

オブジェクトが同じかどうかは `equal?` で調べます。内容が等しいかを調べる `==` とは、見ている対象が違います。

```
p :ruby.equal?(:ruby)
p "ruby".equal?("ruby")
p "ruby" == "ruby"
```

```
true
false
true
```

1 行目はシンボルなので `true` です。2 行目は別々に作られた文字列どうしで、同じオブジェクトではありません。3 行目は内容を比べる `==` で、同じ内容だから `true` です。`false` と `false` の組は、シンボルまで毎回別のオブジェクトと考えた誤りです。`true` と `true` の組は、文字列も同じオブジェクトと考えた誤りで、`frozen_string_literal: true` のあるファイルでしか成り立ちません。

シンボルは `:"hello world"` のように引用符で囲むと、空白を含む名前も作れます。`"ruby".to_sym` は `:ruby` を返し、`:ruby.to_s` は `"ruby"` を返します。変換しても元のオブジェクトは変わりません。シンボルと文字列の使い分けは、第9章で詳しく見ます。

配列リテラルは `[1, "two", :three]` のように、角括弧にコンマ区切りで書きます。中身の型は混ぜられ、配列の中に配列も置けます。添字は 0 から始まり、範囲外の添字を指定しても、エラーにはなりません。`nil` が返るだけです。

ハッシュリテラルは `{ キー => 値 }` の形です。キーがシンボルのときは `{ name: "Taro" }` と略記できて、`{ :name => "Taro" }` と同じ意味になります。

```
h = { name: "Taro", "age" => 20, 1 => :one }  
p h  
p h[:name], h["age"], h[:age]  
p({ "a": 1 })
```

```
{:name=>"Taro", "age"=>20, 1=>:one}
"Taro"
20
nil
{:a=>1}
```

文字列キーの `"age"` とシンボルキーの `:age` は別物なので、`h[:age]` は `nil` です。存在しないキーを引いても、エラーにはなりません。`{ "a": 1 }` のように引用符付きで書いたキーは、文字列ではなくシンボルの `:a` です。

and・or・not と &&・||・! の優先順位

`and`・`or`・`not` は、`&&`・`||`・`!` と同じ働きをしますが、優先順位がずっと低い演算子です。`=` よりも低いので、代入の右側に `or` を書くと、`or` の手前までしか代入されません。

```
a = false or true
b = (false or true)
c = false || true
p [a, b, c]
```

```
[false, true, true]
```

`a = false or true` は `(a = false) or true` と読まれるので、`a` は `false` です。括弧で囲んだ `b` と、`||` を使った `c` は `true` になります。`or` が代入より先に評価されることはありません。

`and`・`or` は、`x or puts "none"` のように、「左が偽なら右を実行する」という流れの制御にも使います。真偽の値を求める式には、優先順位の高い `&&` と `||` を使うのが安全で、代入と `or` を混ぜてはいけません。

nil と false だけが偽

条件式で偽として扱われる値は、`nil` と `false` の 2 つだけです。それ以外はすべて真です。`0` も空文字列 `""` も空配列 `[]` も真になります。C 言語や JavaScript のように `0` や空の値を偽にする言語の感覚は、Ruby に通用しません。

```
p [nil, false, 0, "", []].map { |v| v ? "真" : "偽" }  
values = [nil, false, 0, "", []]  
p values.select { |v| !v }.size
```

```
["偽", "偽", "真", "真", "真"]  
2
```

`!v` は `v` が偽のときだけ真になるので、`select` が残すのは `nil` と `false` の 2 つで、出力は `2` です。`3` と数えるのは `0` も偽と考えた誤りで、`4` は `""` まで偽と考えた誤りです。`5` は `[]` まで偽と数えた結果ですが、空の配列も真なので成り立ちません。

`nil` は「値が無い」ことを表す特別なオブジェクトで、`NilClass` のただ 1 つのインスタンスです。`nil.to_a` は `[]`、`nil.to_s` は空文字列 `""`、`nil.inspect` は `"nil"` を返します。`nil?` メソッドは `nil` のときだけ `true` です。`false.nil?` は `false` を返します。`nil` と `false` はどちらも偽ですが、同じ値ではありません。

`&&` と `||` は、真偽値ではなく**最後に評価した値そのもの**を返します。左辺で結果が決まると、右辺は評価されません。

```
p (nil && 1), (false || :x), (1 && 2), (nil || false)

s = nil
s ||= "dflt"
s ||= "other"

p s
```

```
nil
:x
2
false
"dflt"
```

`nil && 1` は左辺が偽なので `nil` です。`false || :x` は左辺が偽なので、右辺の `:x` になります。`s ||= "dflt"` は、`s` が偽のときだけ代入する書き方です。2 回目の `||=` では `s` がすでに真なので、代入は行われません。

変数の種類とスコープ

変数は名前の先頭の記号で種類が決まります。

種類	名前の形	例	参照できる範囲
ローカル変数	小文字か <code>_</code> で始まる	<code>count</code>	定義した範囲の中
インスタンス変数	<code>@</code> で始まる	<code>@name</code>	そのオブジェクトの中

種類	名前の形	例	参照できる範囲
グローバル変数	\$ で始まる	\$count	プログラム全体
定数	大文字で始まる	LIMIT	次の節で扱う

ローカル変数は、代入した場所から、囲んでいる範囲の終わりまで使えます。
def・class・module は、新しいローカル変数の範囲を作ります。 外側のローカル変数は、その内側から見えません。

```
count = 3
def show
  puts count
end
begin
  show
rescue NameError => e
  puts e.class
end
```

NameError

show 中の count は、外側の count と無関係です。count という名前のローカル変数もメソッドも見つからないので、NameError になります。メッセージは undefined local variable or method `count' for main:Object で始まります。3 が出力されるのは、メソッドの中から外側の変数が見える言語での動きです。空行が出力されることもありません。未定義の名前は nil にならず、エ

ラーになるからです。ArgumentError は引数の数が合わないときの例外ですが、show は引数なしで呼んでいます。

一方、ブロック (do ... end や { ... }) は、外側のローカル変数を見られます。ブロックの中で外側の変数に代入すれば、外側の値が変わります。ブロックの中で新しく作った変数は、ブロックを出ると使えません。

```
y = 1
[1].each { y += 1; w = 5 }
p y
p defined?(w)
```

```
2
nil
```

defined? は、式が定義済みかを調べる演算子で、未定義なら nil を返します。if や while や case は、新しい範囲を作りません。その中で作った変数は、後ろでも使えます。条件が偽で一度も実行されなくても、代入を含む行を Ruby が読んだ時点で、変数の名前は登録されます。

```
if false
  z = 1
end
p z
```

```
nil
```

登録されたが代入されていない変数は `nil` です。`for` もブロックではないので、`for` で使った変数はループの後でも残ります(第2章で扱います)。

インスタンス変数は、代入する前に読んでも `NameError` になりません。`nil` が返ります。ローカル変数やメソッド名のような未定義の名前とは、扱いが違います。グローバル変数も、代入前に読むと `nil` です。

```
p @undefined  
  
$g = 10  
  
def gg  
  $g += 1  
  
end  
  
gg  
  
p $g
```

```
nil  
  
11
```

グローバル変数は、`def` の内側でも遮られません。メソッドの中から書き換えることもできます。ただし、どこからでも書き換えられるので、使う場面は限られます。インスタンス変数とクラスの定義は、第6章で扱います。

変数名の規則と予約語

数字で始まる名前 (`3y`) やハイフンを含む名前 (`my-var`) は、変数名にできません。変数の種類は、名前の最初の記号で見分けます。ローカル変数は小文字かアンダースコアで始め、インスタンス変数は `@`、クラス変数は `@@`、グローバル変数は `$` で始め、定数は大文字で始めます。

種類	始まり	例
ローカル変数	小文字か <code>_</code>	<code>total</code> ・ <code>_tmp</code>
インスタンス変数	<code>@</code>	<code>@name</code>
クラス変数	<code>@@</code>	<code>@@count</code>
グローバル変数	<code>\$</code>	<code>\$stdout</code>
定数	大文字	<code>LIMIT</code>

インスタンス変数、クラス変数、グローバル変数の順に並べると、`@a`、`@@a`、`$a` です。`@` が 1 つか 2 つかで、インスタンス変数とクラス変数を取り違えないようにします。`$a` はグローバル変数であり、クラス変数ではありません。

予約語(キーワード)は、Ruby が文法のために取っておいている語です。`if`・`unless`・`end`・`self`・`nil`・`true`・`false`・`class`・`defined?` などがあり、変数名には使えません。`obj.if` のようにメソッド名だと分かる位置では使えますが、ふつうは避けます。`puts`・`p`・`print`・`rand`・`send`・`each` は予約語ではなく、メソッドです。`puts` という名前の変数を作ることできるので、予約語と同じ扱いはされません。

```
puts = 1
p puts
```

1

このように `puts` は予約語でないので、変数に使えます。`self` や `unless` を変数名にすると、構文エラーになります。見た目がメソッドの呼び出しに似て

いても、`unless` や `defined?` は予約語であり、`puts` を予約語と呼ぶことはありません。

定数の再代入と参照

定数は大文字で始まる名前で、`LIMIT = 10` のように代入して作ります。「変えない値」を表す慣習の名前ですが、Ruby は再代入を禁止していません。再代入しても例外にはならず、警告が出るだけです。

```
LIMIT = 10  
  
LIMIT = 20  
  
puts LIMIT
```

```
20
```

標準エラー出力には `already initialized constant LIMIT` という警告が出ます（`previous definition of LIMIT was here` の行も続きます）。値は新しい `20` に変わります。再代入が無視されて `10` のままになることはありません。`NameError` は未定義の名前を参照したときの例外で、`LIMIT` は定義済みだから起きません。`FrozenError` は凍結したオブジェクトを書き換えたときの例外であり、定数への代入そのものとは関係がありません。

定数を `freeze` すると、中身を変える操作が `FrozenError` になります。凍結していない配列を定数にしても、定数が指す配列の中身は変えられます。定数が固定しているのは「どのオブジェクトを指すか」だけで、オブジェクトの中身ではありません。定数で制限されるのは参照先の付け替えだけで、参照先の配列や文字列の中身を変えることは制限されません。

```
LIST = %w[a b].freeze
LIST2 = [1]
LIST2 << 2
p LIST2
begin
  LIST << "c"
rescue => e
  puts e.class
end
```

```
[1, 2]
FrozenError
```

定数は、メソッドの中からも参照できます。ローカル変数と違い、`def` の境界で遮られません。メソッドの中で定数に代入する書き方は、構文エラー（`dynamic constant assignment`）です。構文エラーはプログラムを読み込む段階で見つかるので、メソッドを呼ばなくても、同じファイルの `puts "done"` は実行されません。

モジュールやクラスの中で定義した定数は、外から `Outer::X` のように `::` でたどって参照します。`Outer.X` と `.` で書くと、メソッド呼び出しと解釈されるので、定数は取れません。実行すると `NoMethodError` が起きます。`Outer::` を付けずに `X` や `Inner` と書いても、最上位には定数が無いので見つかりません。実行すると `NameError` が起きます。クラスとモジュールの中での探索規則は第6章で扱い、この章では詳しく説明しません。

演算子の優先順位と代入の強さ

算術演算子の強さは、数学の約束と同じです。論理演算子と代入を含めた全体の順は、次の表のとおりです。[^3]

強さ	演算子	結合の向き
強い	!, ~、単項の +	右から
↓	**	右から
↓	単項の -	右から
↓	* / %	左から
↓	+ -	左から
↓	< > <= >=	左から
↓	== != <=>	並べて使えない
↓	&&	左から
↓		左から
↓		並べて使えない
↓	? : (三項演算子)	右から
↓	= += = など	右から
↓	not	右から
弱い	and or	左から

```
p 2 + 3 * 2 ** 2
p 2 ** 3 ** 2
p 10 - 3 - 2
p -2 ** 2
```

14

512

5

-4

$2 + 3 * 2 ** 2$ は、 $**$ が最も強いので $2 ** 2$ が 4、次に $3 * 4$ が 12、最後に $2 + 12$ で 14 です。20 は足し算を先に計算した読み方で、64 はべき乗を最後に回した読み方です。100 は左から順に計算した読み方で、どれも優先順位を守っていません。

$**$ は右から結合するので、 $2 ** 3 ** 2$ は $2 ** (3 ** 2)$ で 512 です。- は左から結合するので、 $10 - 3 - 2$ は $(10 - 3) - 2$ で 5 になります。 $-2 ** 2$ は $-(2 ** 2)$ と読まれるため、結果は -4 で、4 ではありません。整数どうしの $/$ は小数点以下を切り捨てるので、 $7 / 2$ は 3 です。

論理演算子には、記号の `&&` `||` `!` と、英語の `and` `or` `not` があります。意味は似ていますが、**英語版は代入より弱い**ので、同じ位置に書いても同じ結果にはなりません。記号版は代入より強い演算子です。

```
a = true && false
```

```
b = true and false
```

```
p [a, b]
```

```
[false, true]
```

$a = \text{true} \ \&\& \ \text{false}$ は `&&` が先に計算され、 $a = (\text{true} \ \&\& \ \text{false})$ と読まれるので、 a は `false` です。 $b = \text{true} \ \text{and} \ \text{false}$ は `and` が代入より弱いので、 $(b$

`= true`) and `false` と読まれ、`b` には `true` が入ります。式全体の値は捨てられ、`b` の値は変わりません。`d = false or true` と書いても、`d` は `false` のままです。`[false, false]` は `and` を `&&` と同じ強さと考えた誤りで、`[true, true]` は `&&` を代入より弱いと考えた誤りです。`and` と `or` は、`x or raise "error"` のような制御の流れに使い、値の計算には使いません。

多重代入

多重代入は、複数の変数へ 1 回の代入で値を割り当てる書き方です。右辺の値を左辺の変数へ、左から順に 1 つずつ割り当てます。

```
a, b = 1, 2
a, b = b, a
p [a, b]
a, b = [1, 2, 3]
p [a, b]
a, b, c = 1, 2
p [a, b, c]
```

```
[2, 1]
[1, 2]
[1, 2, nil]
```

右辺がすべて評価されてから割り当てるので、`a, b = b, a` で 2 つの値を入れ替えられます。右辺が配列 1 つのときは、中身が展開されます。変数より値が多いと、余った値は捨てられ、どの変数にも入りません。値が足りない変数は `nil` です。

変数に `*` を付けると、余った値を配列として受け取ります。`*` は 1 つの左辺に 1 回しか付けられません。

```
a, *b, c = 1, 2, 3, 4
p [a, b, c]
a, *b = 1
p [a, b]
*a, b = 1, 2, 3
p [a, b]
a, (b, c), d = 1, [2, 3], 4
p [a, b, c, d]
```

```
[1, [2, 3], 4]
[1, []]
[[1, 2], 3]
[1, 2, 3, 4]
```

`*b` が受け取るのは、前後の変数に割り当てた残りです。先頭の `a` が 1、後ろの `c` が最後の 4 になり、間の `[2, 3]` が `b` に入ります。`[1, [2, 3, 4], nil]` は `b` が余りを全部取ると考えた誤りで、`*` の後ろに変数があれば、最後の値はそちらに入ります。`[1, [2], 3]` は値の数を数え違えた形で、4 つの値からは作れません。`[1, [], 4]` が出るのは、値が 2 つしか無く、前後の変数に割り当てて余りが無いときです。`a, *b = 1` のように値が 1 つだけでも、`b` は `nil` でなく `[]` になります。

丸括弧を使うと、入れ子の配列も展開できます。右辺に値を 2 つ以上コンマで並べて 1 つの変数へ代入すると、配列にまとめ、展開はされません。`a = 1,`

2 の結果は [1, 2] です。

SAMPLE

サンプルはここまでです

続き(第2章～巻末)は、Ruby技術者認定試験 Silver 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

Ruby技術者認定試験 Silver 学習用テキスト(無料サンプル)

版

2026年10月版(Rubyアソシエーション「Ruby技術者認定試験」Silver 出題範囲 (Ruby 3.1 対象) 準拠)

発行

IT資格道場(<https://www.shikaku-doujou.com>)

お問い合わせ

info@shikaku-doujou.com

Ruby は、まつもとゆきひろ氏が開発したプログラミング言語です。Ruby技術者認定試験は一般財団法人 Rubyアソシエーションの試験名称です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式の出題範囲と Ruby 3.1 のリファレンスマニュアルの内容を基に作成したもので、一般財団法人 Rubyアソシエーションとの提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。