Literals

Literals create objects you can use in your program. Literals include:

- Booleans and nil
- Numbers
- Strings
- Symbols
- Arrays
- Hashes
- Ranges
- Regular Expressions
- Procs

Booleans and nil

nil and false are both false values. nil is sometimes used to indicate "no value" or "unknown" but evaluates to false in conditional expressions.

true is a true value. All objects except nil and false evaluate to a true value in conditional expressions.

(There are also the constants TRUE, FALSE and NIL, but the lowercase literal forms are preferred.)

Numbers

You can write integers of any size as follows:

```
1234
1_234
```

These numbers have the same value, 1,234. The underscore may be used to enhance readability for humans. You may place an underscore anywhere in the number.

Floating point numbers may be written as follows:

```
12.34
1234e-2
1.234E1
```

These numbers have the same value, 12.34. You may use underscores in floating point numbers as well.

You can use a special prefix to write numbers in decimal, hexadecimal, octal or binary formats. For decimal numbers use a prefix of 0d, for hexadecimal numbers use a prefix of 0x, for octal numbers use a prefix of 0 or 0o, for binary numbers use a prefix of 0b. The alphabetic component of the number is not case-sensitive.

Examples:

```
0d170
0D170

0xaa
0xAa
0xAA
0Xaa
0XAa
0XaA

0252
0o252
0O252

0b10101010
0B10101010
```

All these numbers have the same decimal value, 170. Like integers and floats you may use an underscore for readability.

Strings

The most common way of writing strings is using ":

```
"This is a string."
```

The string may be many lines long.

Any internal " must be escaped:

```
"This string has a quote: \".  As you can see, it is escaped"
```

Double-quote strings allow escaped characters such as \n for newline, \t for tab, etc.

Double-quote strings allow interpolation of other values using #{...}:

```
"One plus one is two: #{1 + 1}"
```

Any expression may be placed inside the interpolated section, but it's best to keep the expression small for readability.

Interpolation may be disabled by escaping the "#" character or using single-quote strings:

```
'#{1 + 1}' #=> "\#{1 + 1}"
```

In addition to disabling interpolation, single-quoted strings also disable all escape sequences except for the single-quote (') and backslash (\).

You may also create strings using %:

```
%Q(1 + 1 is #{1 + 1}) #=> "1 + 1 is 2"
```

There are two different types of % strings %q(...) behaves like a single-quote string (no interpolation or character escaping) while %Q behaves as a double-quote string. See Percent Strings below for more discussion of the syntax of percent strings.

Adjacent string literals are automatically concatenated by the interpreter:

```
"con" "cat" "en" "at" "ion" #=> "concatenation"
"This string contains " "no newlines."              #=> "This string contains
no newlines."
```

Any combination of adjacent single-quote, double-quote, percent strings will be concatenated as long as a percent-string is not last.

```
%q{a} 'b' "c" #=> "abc"
"a" 'b' %q{c} #=> NameError: uninitialized constant q
```

Here Documents

If you are writing a large block of text you may use a "here document" or "heredoc":

```
expected_result = <<HEREDOC
This would contain specially formatted text.

That might span many lines
HEREDOC
```

The heredoc starts on the line following <<HEREDOC and ends with the next line that starts with HEREDOC. The result includes the ending newline.

You may use any identifier with a heredoc, but all-uppercase identifiers are typically used.

You may indent the ending identifier if you place a "-" after <<:

```
  expected_result = <<-INDENTED_HEREDOC
This would contain specially formatted text.

That might span many lines
  INDENTED_HEREDOC
```

Note that the while the closing identifier may be indented, the content is always treated as if it is flush left. If you indent the content those spaces will appear in the output.

A heredoc allows interpolation and escaped characters. You may disable interpolation and escaping by surrounding the opening identifier with single quotes:

```
expected_result = <<-'EXPECTED'
One plus one is #{1 + 1}
EXPECTED

p expected_result # prints: "One plus one is \#{1 + 1}\n"
```

The identifier may also be surrounded with double quotes (which is the same as no quotes) or with backticks. When surrounded by backticks the HEREDOC behaves like Kernel#`:

```
puts <<-`HEREDOC`
cat #{__FILE__}
HEREDOC
```

To call a method on a heredoc place it after the opening identifier:

```
expected_result = "One plus one is #{1 + 1}
".chomp
```

You may open multiple heredocs on the same line, but this can be difficult to read:

```
puts("content for heredoc one
", "content for heredoc two
")
```

Symbols

A Symbol represents a name inside the ruby interpreter. See Symbol for more details on what symbols are and when ruby creates them internally.

You may reference a symbol using a colon: :my_symbol.

You may also create symbols by interpolation:

```
:"my_symbol1"
:"my_symbol#{1 + 1}"
```

Note that symbols are never garbage collected so be careful when referencing symbols using interpolation.

Like strings, a single-quote may be used to disable interpolation:

```
:'my_symbol#{1 + 1}' #=> :"my_symbol\#{1 + 1}"
```

When creating a Hash there is a special syntax for referencing a Symbol as well.

Arrays

An array is created using the objects between [and]:

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

You may place expressions inside the array:

```
[1, 1 + 1, 1 + 2]
[1, [1 + 1, [1 + 2]]]
```

See Array for the methods you may use with an array.

Hashes

A hash is created using key-value pairs between { and }:

```
{ "a" => 1, "b" => 2 }
```

Both the key and value may be any object.

You can create a hash using symbol keys with the following syntax:

```
{ a: 1, b: 2 }
```

This same syntax is used for keyword arguments for a method.

See Hash for the methods you may use with a hash.

Ranges

A range represents an interval of values. The range may include or exclude its ending value.

```
(1..2)  # includes its ending value
(1...2) # excludes its ending value
```

You may create a range of any object. See the Range documentation for details on the methods you need to implement.

Regular Expressions

A regular expression is created using "/":

```
/my regular expression/
```

The regular expression may be followed by flags which adjust the matching behavior of the regular expression. The "i" flag makes the regular expression case-insensitive:

```
/my regular expression/i
```

Interpolation may be used inside regular expressions along with escaped characters. Note that a regular expression may require additional escaped characters than a string.

See Regexp for a description of the syntax of regular expressions.

Procs

A proc can be created with ->:

```
-> { 1 + 1 }
```

Calling the above proc will give a result of 2.

You can require arguments for the proc as follows:

```
->(v) { 1 + v }
```

This proc will add one to its argument.

Percent Strings

Besides %(...) which creates a String, The % may create other types of object. As with strings, an uppercase letter allows interpolation and escaped characters while a lowercase letter disables them.

These are the types of percent strings in ruby:

```
%i
```

Array of Symbols

```
%q
```

String

```
%r
```

Regular Expression

```
%s
```

Symbol

```
%w
```

Array of Strings

```
%x
```

Backtick (capture subshell result)

For the two array forms of percent string, if you wish to include a space in one of the array entries you must escape it with a "\" character:

```
%w[one one-hundred\ one]
#=> ["one", "one-hundred one"]
```

If you are using "(", "[", "{", "<" you must close it with ")", "]", "}", ">" respectively. You may use most other non-alphanumeric characters for percent string delimiters such as "%", "|", "^", etc.