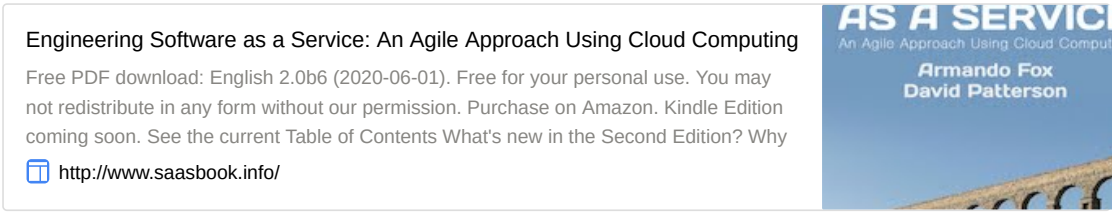
Ruby

A PROGRAMMER'S BEST FRIEND

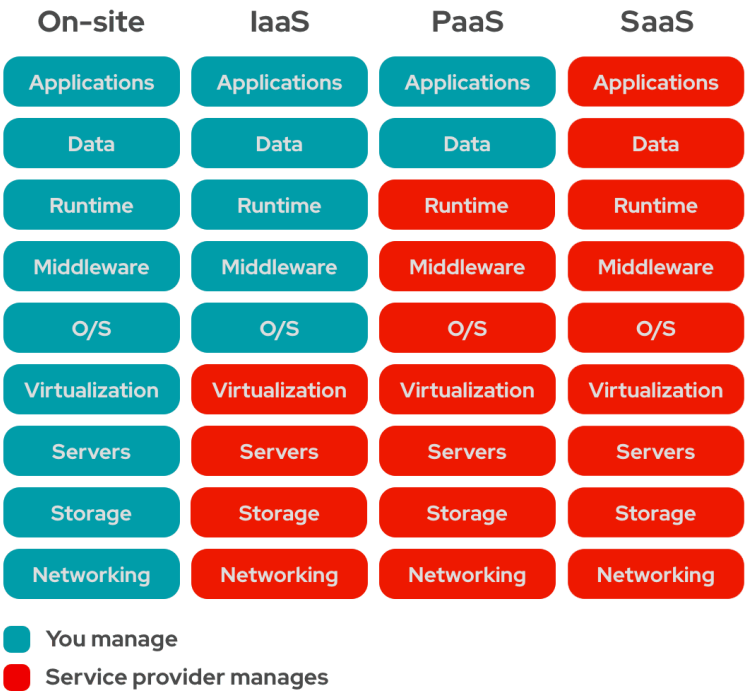
Part 1: The Basics

▼ Preface

- Reference Book

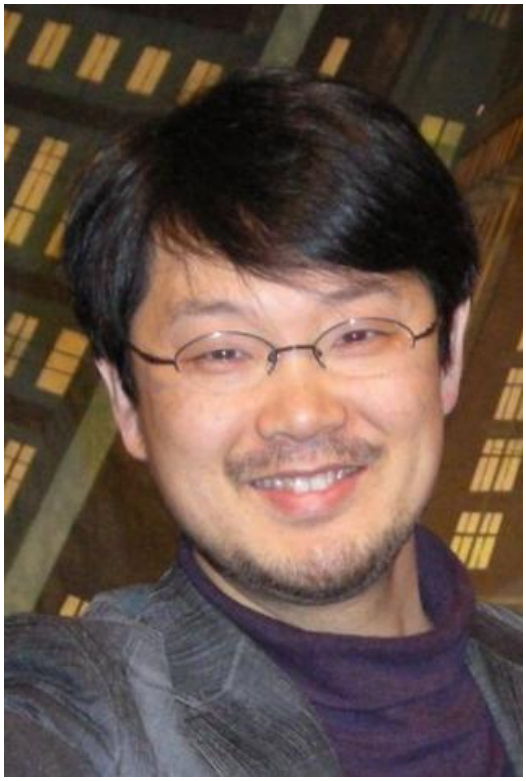


- Software as a service (SaaS)



▼ Introduction to Ruby

- Created in 1995 by Yukihiro “Matz” Matsumoto



Yukihiro Matsumoto

- Syntax like Perl, Python, and Smalltalk
- Principle of least astonishment (POLA)
  - Readable, like English
  - Unsurprising syntax, naming, and behavior
- Interpreted language, not compiled
  - Requires Ruby interpreter
- Dynamic Type
- Object-oriented
  - Most everything is an object
  - No "primitives"
- Whitespace independent
- No semi-colons
- Ruby vs Ruby on Rails
  - Ruby on Rails
    - Web framework written in Ruby
    - MVC Design Pattern and focuses on DRY
  - Ruby
    - Multipurpose language

        - Not just for the web
    - Community

▼ Installation (Ruby on Rails)

- The majority of Ruby on Rails developers use macOS and Linux
- Windows has always been possible, but a rockier path
- Some libraries and Ruby Gems require Unix

▼ Possible solutions for windows users:

- Unix-like OS
    - Dual Boot ⇒ Fedora
    - Virtual machine
- **WSL** (Windows Subsystem for Linux) or **Multipass**
- RubyInstaller

> RubyInstaller for Windows
>
> RubyInstaller versions 3.0.2-1, 2.7.4-1 and 2.6.8-1 are released. These are maintenance releases with bug and security fixes. Read full article RubyInstaller versions 2.7.2-1, 2.6.6-2 and 2.5.8-2 are released. These
>
> https://rubyinstaller.org/

▼ Using Ruby

```
# Execute single command
ruby -e 'puts 123'
# Execute ruby file
ruby simple_file.rb
```

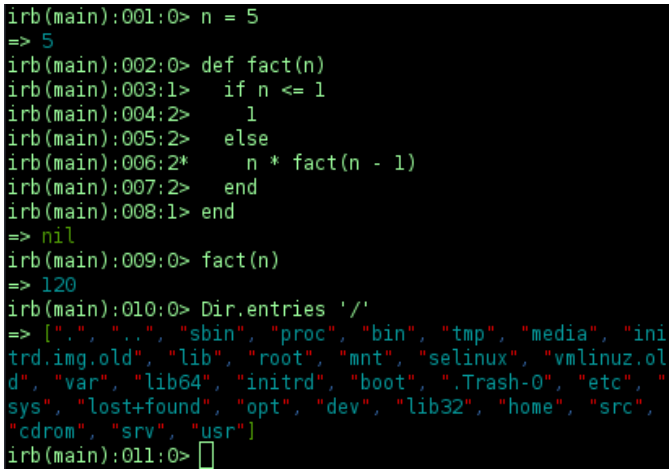
▼ Documentation

> Ruby-Doc.org
>
> Fast, searchable Ruby documentation for core and standard libraries. Plus, links to tutorials, guides, books, and related sites.
>
> https://ruby-doc.org/

```
ri String
ri String.new
ri String#upcase
```

▼ Interactive Ruby Shell (IRB)

```
irb
```



▼ Object Types

Objects:

- Similar to physical objects
- Objects can represent abstract ideas
- Object have attributes
- Objects are instances of a class
- Objects have behaviors

▼ Variables

- Not objects
- Store a reference to an object
- Ruby naming convention
  - Lowercase with underscore ⇒ **snake_case**
- Scope

```
# Global
$variable
# Class
@@variable
# Instance
@variable
# Local
variable
# Block
variable
```

▼ Numbers

- Integers
- Floating-point numbers (“floats”)
- Math Order of Operation
  - Standard math rules: PEMDAS

▼ Basic Math Operation

```
4+2
4-2
4/2
4*2
4**2
```

▼ Math Assignment Operators

```
x = 4
x += 2 # same as" x = x + 2
x -= 1
x *= 2
x /= 5
```

▼ Number Methods

```
100.class
100.next
temp = -50
temp.abs
10.0.class
3.0.to_i
10.to_f
12.6.round
12.9.floor
12.3.ceil


# 10 / 3
# 10.0 / 3
```

▼ Strings

A sequence of characters

- Letters, numbers, symbols
- Spaces, tabs, line returns

▼ Code Example

```
greeting = "Hello"
target = "world"
# Concatenation
greeting + ' ' + target
# Append
greeting << ' '
greeting << 'world'
# Multiplication
"test " * 3
#Methods
greeting.reverse
greeting.capitalize
greeting.upcase
greeting.reverse.capitalize
greeting.downcase
greeting.length
5.to_s
"1-2-3-4".split('-')
```

```
a = "hello"
"hello".count "lo"
"hello".delete "lo"
"hello".delete_prefix("hel")
"hello".delete_suffix("llo")
"".empty?
" ".empty?
a.gsub(/[eo]/, 'e' => 3, 'o' => '*')
a.gsub(/[eo]/, '*')
"hello".include? "ol"
"hello".index('e')
"hello".index('lo')
"abcd".insert(0, 'X')
```

```
a = "hello"
"hello".replace "hi"
a["el"] = "--"
a[1..2] = 'el'
"  hello  ".lstrip
"  hello  ".rstrip
"  hello  ".strip
"yellow    moon".squeeze
"hello".start_with?("hell")
"hello".end_with?("ello")
"abcd".succ
"THX1138".succ
"Hello".swapcase
"9".upto("11").to_a
"hello"[1]
```

```
# Escaping
"Let’s escape!"
'Let’s escape!'
'Let\’s escape!'
# Control Characters
puts "\ta\tb\nc"
# Interpolation
msg = 'a test message'
"message: #{msg}."
"1+1 = #{1+1}."
```

▼ Arrays

An ordered, integer-indexed collection of objects

- Like an expandable file folder
- Put objects into “pockets”

- Pockets can be empty
- Unlimited number of pockets
- Pocket count starts at 0

▼ Code Example

```
empty_array = []
Array.new(3)
Array.new(3, true)
Array.new(4) { 'a' }
Array.new(4) {|i| i.to_s }

my_array = ['a', 'b', 'c', 'd', 'e', 6, 7]
my_array[0]
my_array[10]
my_array[1] = 'q'
my_array[2] = nil
my_array << 'e'
my_array << ['f', 'g']

my_array[-1]
my_array[2,3]
my_array[-2,2]
my_array[2..3]
my_array[-4..-1]
```

```
# Array Methods
array = [2,4,['a','b'],nil,4,'c']
array.first
array.last
array.size
array.length
array.reverse
array.shuffle
array.uniq
array.compact
array.flatten
array.include?(2)
array.delete_at(1)
array.delete('c')
[1,2,3,4].join
[1,2,3,4].join('-')

# Array Addition and Subtraction
[1,2,3] + [3,4,5]
[1,2,3] – [2]
```

```
# Destructive method with exclamation mark
array = Array.new(5) {|i| i+1 }
array.shuffle
array
array.shuffle!
array
```

```
array = 'a'.upto('h').to_a
# push
array.push(1)
array.push(2,3)
# pop
array.pop
```

```
array.pop(2)
# shift
array.shift
array.shift(2)
# unshift
array.unshift("a")
array.unshift(1, 2)
```

▼ Hashes

An unordered, object-indexed collection of objects

- key-value pairs
- Order not Important
- Find items by key, not position
- Hash keys must be unique

▼ Code Example

```
car = {
   'brand' => 'Ford',
   'model' => 'Fiesta',
   'color' => 'blue',
}
car['brand']
car['color'] = 'green'
car['doors'] = 3
car.keys
car.values
car.size
car.length
car.to_a
```

▼ Symbols

Like strings but cannot be edited

- Begin with a colon
- Not delimited by quotes
- Ruby naming convention
  - Lowercase, underscore, no spaces

▼ Code Example

```
"test".object_id
"test".object_id
:test.object_id
:test.object_id
person =  {
  :first_name => 'Benjamin',
  :last_name => 'Franklin'
}
person[:first_name ]
person['first_name']
# Hash symbol shorthand
score = {:low=>2, :high=>8}
score = {low:2, high:8}
```

▼ Booleans

An object that is either true or false

Comparison and Logic Operators

Aa Operators	.
==	Equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal to
!	Not
!=	Not equal
&&	And
||	Or

▼ Code Example

```
x = 1
x == 1
x != 1
true.class
false.class
x < 3
x > 3
x > 0 && x < 100
x >= 100 || x <= 50
!x
x.nil?
2.between?(1,5)
[1,2,3].include?(2)
[1,2,3].empty?
[].empty?
hash = {'a' => 1, 'b' => 2}
hash.has_key?('a')
hash.has_key?(:a)
hash.has_value?(1)
hash.has_value?(5)
```

▼ Ranges

Ranges are a range of sequential objects

```
# Inclusive range
1..10
# Exclusive range
1...10

inclusive = 1..10
exclusive = 1...10
inclusive.class
inclusive.begin
inclusive.first
inclusive.end
inclusive.last
a1 = [*inclusive]
a2 = [*exclusive]
```

```
alpha = 'a'..'m'
alpha.include?('g')
```

▼ Constants

- Similar to variables
- Use for storing values that are constant
- Named using all uppercase

▼ Code Example

```
MAX_SCORE = 100
MAX_SCORE = 50
# warning: already initialized constant
```

▼ Nil

Nil is and object in Ruby, and it just is nothing.

▼ Code Example

```
# Nil
nil.class
nil == false
nil.nil?
# Booleans and Nil
product.nil?
product == nil
!product
```

▼ Control Structures

- Add dynamism to code
- Determine circumstances when code executes
- Conditional
- Loops
- Iterators

▼ Conditional

```
# Conditional: if else elsif
if boolean
   # ...
elsif boolean
   # ...
else
   # ...
end

# Example
x = 15

if x <= 10
  puts "10 or below"
```

```
elsif x >= 20
  puts "20 or above"
else
  puts "Between 10 and 20"
end
```

```
# Conditional: unless
unless boolean
   # ...
end
# same as
if !boolean
   # ...
end

# Example
unless array.empty?
   # ...
end

unless search_result.nil?
   # ...
end

# Example
cart = ['apple', 'banana', 'carrot']

unless cart.empty?
  puts "The first item is: #{cart[0]}"
else
  puts "The cart is empty."
end
```

```
# Conditional: case
case
when boolean
   # ...
when boolean
   # ...
else
   # ...
end
```

```
# Conditional: case
case test_value
when value
   # ...
when value
   # ...
else
   # ...
end
```

```
# case: Example
count = 7
case
when count == 0
  puts "nobody"
when count == 1
  puts "1 person"
when (2..5).include?(count)
  puts "several people"
```

```
else
  puts "many people"
end

case count
when 0
  puts "nobody"
when 1
  puts "1 person"
when 2..5
  puts "several people"
else
  puts "many people"
end
```

```
# Conditional: Shorthand operators

# Ternary Operator
boolean ? result : result2
puts count == 1 ? "person" : "people"
```

```
# Conditional: Shorthand operators

# Or Operator
x = y || z
# same as
if y
   x = y
else
   x = z
end
```

```
# Conditional: Shorthand operators

# Or-Equals Operator
x ||= y
# same as
unless x
   x = y
end
```

```
# Conditional: Shorthand operators

# Statement Modifiers
x = y unless x
puts "Hello" if greeting_enabled
```

▼ Loops

Loops allow us to repeat a section of code over again. We can also do that conditionally so it'll execute it until a certain condition is met, or we can use conditions inside our loops so that every time as conditions change, the loop's behavior changes.

```
loop do
  # ...
end
# Control Methods:
# break = Terminate the whole loop
```

```
# next = Jump to next loop
# redo = Redo this loop
# retry = Start the whole loop ever

# Example
i = 5
loop do
  break if i <= 0
  puts "Countdown: #{i}"
  i -= 1
end
puts "Blast off!"
```

```
while boolean
   # ...
end

until boolean
   # ...
end

# Example
i = 5
while i > 0
  puts "Countdown: #{i}"
  i -= 1
end
puts "Blast off!"

cart = ['apple', 'banana', 'carrot']
until cart.empty?
  first = cart.shift
  puts first.upcase
end
```

▼ Iterators

Iterators are a lot like loops, but instead of performing code a certain number of times or until a condition is met, an iterator uses a set of objects and executes the code once each time for those objects.

- To say or do again
- To apply a procedure repeatedly
- To perform code on each item in a set

▼ Code Example

```
5.times {puts "Hello"}
1.upto(5) {puts "Hello"}
5.downto(1) {puts "Hello"}
(1..5).each {puts "Hello"}
```

```
# Block Variable
5.times do |i|
  puts "Countdown #{5-i}"
end
puts "Blast off!"
```

```
fruits = ['banana', 'apple', 'pear']

fruits.each do |fruit|
  puts fruit.capitalize
end

# for
for fruit in fruits
  puts fruit.capitalize
end
```

▼ Iterators: By Class

- **Numbers**: *times, upto, downto, step*

▼ Code Example

```
12.step(6, -2) do |i|
  puts i
end
```

- **Range**: *each, step*
- **String**: *each_line, each_char, each_byte*
- **Array**: *each, each_index, each_with_index*
- **Hash**: *each, each_key, each_value, each_pair*

▼ Scripting

▼ Best practices

- Give Ruby files an ***.rb*** extension
- Put a shebang line at the top

```
#!/usr/bin/env ruby
```

▼ Exit a running script

```
# Exit a running script
fruits = ['banana', 'apple', 'pear']
fruits.each do |fruit|
  if fruit == 'apple’
    # exit
    # exit!
    # abort
    abort("Exit on apple")
  end
  puts fruit.capitalize
end
```

▼ Input and output

```
#!/usr/bin/env ruby
print "What is your name? "
```

```
response = gets.chomp
puts "Hello, #{response}!"
```

- Enumerable and Code Blocks
  - Enumerable
    - Countable items
      - Array
      - Ranges
      - Hashes
      - String (sort of)
    - In the Ruby programming language, there's actually a module called Enumerable and it is a module that's included in each of those classes. What we say in Ruby is that it mixes in this module called Enumerable.

      > Array
      >
      > An Array is an ordered, integer-indexed collection of objects, called elements. Any object may be an Array element. Array indexing starts at 0, as in C or Java. A positive index is an offset from the first element: A negative index is an offset, backwards, from the end of the array: A non-negative index is in range if it is smaller than the size of the array.
      >
      > 📥 https://ruby-doc.org/core-3.0.2/Array.html

    - Methods
      - count
      - each, each_with_index
      - first, last
      - include?
      - max, min (if comparable)
  - Code Block
    - Many ruby methods will accept an optional code block
    - The code blocks usually modifies default behavior
    - Code blocks format
      - Curly-brace format
        - Single-line blocks
        - Blocks that return data, no changes

        ```
        5.times {|i| puts i}
        ```

      - Do-end format
        - Multi-line blocks
        - Blocks that perform actions, make changes

```
scores = {"low" => 2, "high" => 8}
scores.each do |k, v|
  puts  "#{k.capitalize}: #{v}"
end
```

▼ Code Block: Common Usage

- find
- map
- inject
- sort
- merge

▼ **Find** Methods

- find / detect
- find_all / select
- any?, none?
- all?, one?
- delete_if

▼ Code Example

```
(1...10).find {|n| n % 3 == 0}

numbers = [*1..10]
numbers.delete_if {|n| n <= 5}

(1..10).any? {|n| n <= 5}
```

▼ **Map** Methods

- map / collect
- Iterate through an enumerable
- Execute a code block on each item
- Add the result of the block to a new array
- Number of items in = Number of items out

▼ Code Example

```
x = [1,2,3,4,5]
y = x.map {|n| n+1}
```

```
scores = {"low" => 2, "high" => 8}
adjusted_scores = scores.map do |k,v|
   "#{k.capitalize}: #{v*100}"
end
```

```
fruits = ['apple', 'banana', 'pear']
y = fruits.map do |fruit|
  if fruit == 'apple'
    fruit.capitalize
  end
end

y2 = fruits.map do |fruit|
  fruit == 'apple' ? fruit.capitalize : fruit
end

cap_fruits = fruits.map do |fruit|
  puts fruit.capitalize
end

cap_fruits
# [nil, nil, nil]
```

▼ **Inject** Methods

- inject / reduce
- “Accumulator”
- Block variable to use for accumulation
  - Ruby convention: memo

▼ Code Example

```
(1..5).inject {|memo, n| memo + n}
# memo = 0 + 1
# memo = memo + 2
# memo = memo + 3
# memo = memo + 4
# memo = memo + 5
# 15

[2,4,6].inject {|memo, n| memo ** n}
# (2 ** 4) ** 6
```

```
# Return Values Matter
(1..5).inject do |memo, n|
  memo + n
  x = 0
end
# 0
(1..5).inject do |memo, n|
  if n % 2 == 0
    memo + n
  end
end
# undefined method ‘+’ for nil
```

```
# Not Just for Math
fruits = ['apple', 'banana', 'pear']
longest = fruits.inject do |memo, fruit|
  if fruit.length > memo.length
    fruit
```

```
  else
    memo
  end
end
```

▼ **Sort** Methods

- Sort methods use the comparison operator
- <=>
  - “Spaceship operator”

```
1 <=> 2
# -1
2 <=> 1
# 1
2 <=> 2
# 0
```

value1 <=> value2

-1	Less than	Moves “left”
0	Equal	Stays
1	More than	Moves “right”

▼ Code Example

```
array = [5,8,2,6,1,3]
x = array.sort {|v1,v2| v1 <=> v2}

fruits = ['apple', 'banana', 'pear']
x1 = fruits.sort

x2 = fruits.sort do |f1, f2|
  f1.length <=> f2.length
end

x3 = fruits.sort_by {|fruit| fruit.length}
```

```
hash = {a: 4, c: 5, b: 3}
hash.sort {|p1, p2| p1[0] <=> p2[0]}
```

```
hash.sort {|p1, p2| p1[1] <=> p2[1]}
```

```
# Sort Methods: Without <=>
fruits = ['apple', 'banana', 'pear']
x = fruits.sort do |fruit1, fruit2|
  case fruit1
  when 'apple'; 1
  when 'banana'; -1
  when 'pear'; 0
  end
end
# ["banana", "pear", "apple"]
```

▼ **Merge** Methods

- Used for hashes only
- Merges two hashes together

▼ Code Example

```
h1 = {:a => 2, :b => 4, :c => 6}
h2 = {:a => 3, :b => 4, :d => 8}

h1.merge(h2)
h2.merge(h1)
```

```
h1 = {:a => 2, :b => 4, :c => 6}
h2 = {:a => 3, :b => 4, :d => 8}

h1.merge(h2) {|key,old,new| new}
h1.merge(h2) {|key,old,new| old}
h1.merge(h2) {|key,old,new| old < new ? old : new}
h1.merge(h2) {|key,old,new| old * new}
```

▼ Custom Methods

- Instructions to perform a specific task, packaged as a unit
- Can be defined once and called multiple times
  - Don’t repeat yourself (DRY)
- Must be defined before they can be called
- Can be redefined without error
- Call by Value
- Method Names
  - Generally, lowercase with underscores
  - First character – lowercase letter or underscore
  - Contain letters, digits, underscores
  - Last character can also be ? ! =

    - Avoid using same names for variables and methods
- Variable Scope
    - Local variables inside methods only have scope inside methods
    - Local variables outside methods do not have scope inside methods
    - Global, class, and instance variables have scope both outside and inside methods

---

▼ Code Example

```
def some_name
  # ...
end

def welcome
  puts 'Hello'
end

def subtract(n1, n2)
  n1 - n2
end

def add_and_subtract(n1, n2)
  add = n1 + n2
  sub = n1 - n2
  [add, sub]
end

def welcome(greeting, name='friend', punct='!')
  greeting + ' ' + name + punct
end

def longest_word(words=[])
  longest = words.inject do |memo, word|
    memo.length > word.length ? memo : word
  end
end
```