Ruby Basics

History

- Ruby is a server-side language. It was designed and developed in 1995 by Yukihiro "Matz" Matsumoto in Japan.

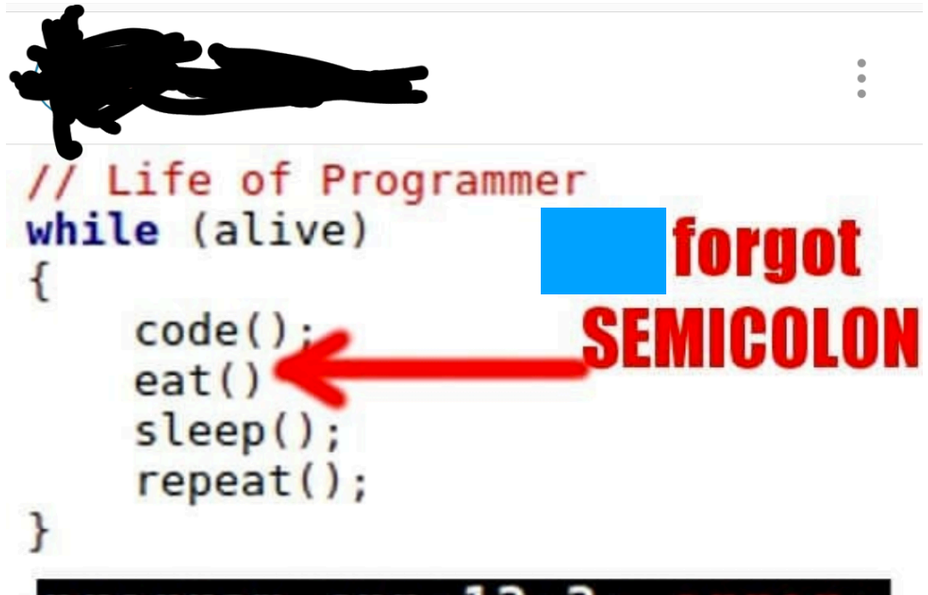
- His philosophy in building Ruby was not about simplicity, but making programming in Ruby feel natural in a way that mirrors life.

Learning a new Language

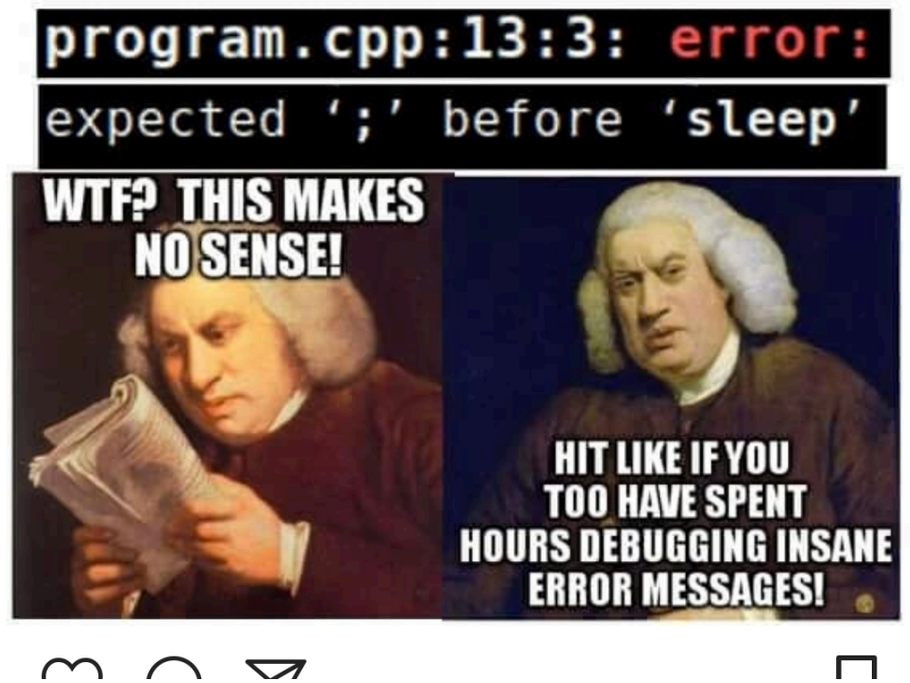
- Why do I need this?
- How do define variables?
- What are the available data types?
- How can manipulate data?
- How can I print to screen?
- Where can I read about it?
- Is it different from JavaScript?
- Does it have a well written documentation like JS?
- How do I define a function?
- What can it do?

Ruby

- Flexibility
  Ruby is flexible making it ideal for coding. What this means is that there are no* blackboxes in Ruby. You can open up anything and change it. This could be a powerful tool in the hands of a skilled Rubyist.



- Readability
  Ruby is hyper readable, the lack of `;`, `()`, keywords
  like `var`, `return`, `function` mean that your semantic naming of methods and variables combined with the semantic nature of the core library methods result in code that feels more like English than a programming language.



Simple Code in Ruby

```ruby
5.times { print “Hello Ebere, How are you today?” }

exit unless "restaurant".include? "aura"

['toast', 'cheese', 'cola'].each { |food| print food.capitalize  }
  number = 3
  # => 3

  if( number == 3 ) # with parens
    puts( "It's a 3!" )
  end
  # It's a 3!
  # => nil

  if number == 3 # without parens
    puts "It's a 3!"
  end
  # It's a 3!
  # => nil
```

Variables

Variables

We no longer need to precede new variables with var. Just use the name of the variable!

- Variables are instantiated as they are used
- Written in snake_case. That means all lower case with words separated by underscores.
- Variable names should still be semantic
- Variables are still assigned using a single equals sign (=)
-

```
my_favorite_animal = "flying squirrel"
# => "flying squirrel"
```

```
foo Local variable
```

```
$foo Global variable
```

```
@foo Instance variable in
object
```

```
@@foo Class variable
```

```
MAX_USERS “Constant” (by
convention)
```

Inputs and Outputs

puts

puts (short for "put string") is the equivalent of Javascript's console.log()

```ruby
puts "Hello, Ruby!"
# Hello, Ruby!
# => nil
```

gets

Ruby also allows us to easily accept inputs from the command line using gets

```ruby
user_input = gets
# => "My input\n" (Note that this line was typed by the user in the terminal)

user_input
# => "My input\n"
```

Usually followed by .chomp

Data Types

Everything Is An Object!

Everything in Ruby is an **object**.

- By "object" we mean that everything has its own set of properties and methods
- Not a new concept. Some data types in Javascript had their own properties and methods (e.g., `string.length`)
- You will learn more about this when you dive into Ruby OOP

Numbers

```
1 + 2 # Addition
# => 3

6 - 5 # Subtraction
# => 1

5 * 2 # Multiplication
# => 10

30 / 5 # Division
# => 6
```

-

```
31 / 5 # Note: integer division
# => 6

30 % 5 # Modulo (remainder)
# => 0

31 % 5
# => 1

3 ** 2 # Exponentiation
# => 9
```

Strings

Words, just like in Javascript.

- Surrounded by single or double-quotes
- Ruby uses similar escape characters
  - Here is a list of them
  - Must instantiate string with double-quotes for escape characters to work
-

```ruby
name = "John"
# => "John"

full_name = "John\nDoe"
# => "John\nDoe"

single_quote = 'John\nDoe'
# => "John\nDoe"

puts full_name
# John
# Doe
# => nil

puts single_quote
# John\nDoe
# => nil
```

-

Strings(JS vs RUBY)

```
'foo' + 'bar' # =>
'foobar'
'foo' + 2 # =>
TypeError: no implicit
conversion of Integer
into String
'foo' + 2.to_s # =>
'foo2'
```

```
# Concatenation
"Hello " + "there!"
# => "Hello there!"

# Multiplication
"Hello there! " * 3
# => "Hello there! Hello
there! Hello there! "
```

Booleans

Still `true` and `false`.

- We'll be using them in conditionals and comparisons just like in Javascript

Comparison operators in Ruby are nearly identical to Javascript. However, the check for equality is always for both value and data type.

- `<`, `>`, `<=`, `>=`, `==`, `!=`

Logical operators are also similar.

- `!`, `&&`, `||`

- The only falsey values in Ruby are `nil` and `false`

Nil

Ruby's "nothing".

- The equivalent of Javascript's `null`
- You will usually see it when something does not have a return value (e.g., a `puts` statement)
- Like in Javascript, `nil` is falsey

Need to check if something is `nil`?
Use `.nil?`

```
something = "A thing"
# => "A thing"

something.nil?
# => false

something = nil
# => nil

something.nil?
# => true
```

Operators

You'll use the following list of operators to do math in Ruby or to compare things. Scan over the list, recognise a few. You know, addition + and subtraction - and so on.

```
  ** !  ~  *  /  %  +  -  &
<< >> |  ^  >  >= <  <= <=>
|| != =~ !~ && += -= == ===
      .. ... not and or
```

- **Combined combination (<=>)** operator return 0 when first operand equal to second, return 1 when first operand is greater than second operand, and return -1 when first operator is less than second operand.
- Append (<<)

Conditionals

Pretty similar to Javascript, with some differences.

- No parentheses or curly brackets required
- Begin blocks using if, elsif (no second "e"!) and else
- We close the whole loop using end
  - This will be used throughout Ruby when dealing with code blocks (e.g., method/function) Here's an example where we check for height at a roller coaster...
-

```
puts "Welcome to the Iron Rattler!
How tall are you (in feet)?"
height = gets.chomp.to_i

if height < 4
  puts "Sorry, you'll fly out of
your seat if we let you on."

elsif height < 7
  puts "All aboard!"

else
  puts "If you value your head, you
should not get on this ride."
end
```

Conditionals

`if` / `unless`

We also have single-line ifs

```
puts 'you are old!' if age >= 100
```

You may even see `unless`

```
puts 'you are old!' unless age < 100
```

When you see an `unless foo`, read it as `if !foo`

`if !foo` can always be written as `unless foo` which creates a more readable line

Ternary operator

A ternary operator looks just like we have seen in JS

```
num.even? ? "#{num} is even!" :
"#{num} is odd!"
```

Bang

The Bang Symbol (!)

All of the Ruby data types we have discussed so far are mutable.

We can not only change what variables are pointing to in memory, but we can directly modify those values stored in memory as well. Methods with an ! attached to the end of them usually mean that they will modify the value in memory they are being called on.

```
a = "cheeseburger"
# => "cheeseburger"

a.upcase = "cheeseburger"
# => "CHEESEBURGER"

a
# => "cheeseburger"
```

```
a.upcase!
# => "CHEESEBURGER"
```

Things can get tricky when you have multiple variables pointing at the same value. For example...

```
a = "cheeseburger"
# => "cheeseburger"

b = a
# => "cheeseburger"

b.upcase!
# => "CHEESEBURGER"

a
# => "CHEESEBURGER"
```

Symbols

Symbols

Symbols are immutable, constant values. That means they contain the same value through the entirety of a program and cannot be changed.

- Kind of like a string that never changes
- Syntax: variable_name = :symbol_name
- No Javascript equivalent (until ES6 came along!))

Symbols

```
favorite_animal = :dog
# => :dog
```

```
favorite_animal.upcase!
# NoMethodError: undefined method `upcase!' for :dog:Symbol
# Did you mean?  upcase
```

Symbols

When/why would you use symbols?

- Most common use is as keys in hashes, the Ruby equivalent of objects (more on that later)
- Make sure values that need to be constant stay constant
- Enhance performance, use less memory

Every string you create is unique and takes up space on your computer, even if they're the same value! When we're busy looking up key/value pairs, we don't want to be wasting memory - we want it to be fast!

Symbols

```
"Your Name".object_id
    #=> a number

"Your Name".object_id
#=> a different number

:your_name.object_id
    #=> a number

:your_name.object_id
#=> the same number!
```

Data Types Exercise

Data Collections

An ordered collection of related values. Same syntax as Javascript arrays.

- Square brackets
- Values separated by commas
- Zero-indexed
-

```
numbers = [1, 2, 3]
# => [1, 2, 3]

animals = ["dog", "cat", "horse"]
# => ["dog", "cat", "horse"]

animals[0]
# => "dog"

animals[1] = "elephant"
# => "elephant"

animals
# => ["dog", "elephant", "horse"]
```

Data Collections

```ruby
numbers = [1, 2, 3]
# => [1, 2, 3]

more_numbers = [4, 5, 6]
# => [4, 5, 6]

lots_of_numbers = numbers + more_numbers
# => [1, 2, 3, 4, 5, 6]

lots_of_numbers - [4, 5, 6]
# => [1, 2, 3]

numbers * 3
# => [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Data Collections

Array Methods

Ruby is very nice. It provides us with an extensive library of array methods we can use to traverse and manipulate arrays.

- The Ruby documentation for `Array` is a great resource for learning more about these methods.
- Can't go over them all, but chances are if you could do it in Javascript then you can do it in Ruby.

IMPORTANT: You DO NOT need to memorize these. The following is just a sample of array methods available to you. You'll come to be more familiar with these as you need them and look them up in documentation.

```ruby
numbers = [1, 2, 3, 4, 5]
# => [1, 2, 3, 4, 5]

numbers.push(6)
# => [1, 2, 3, 4, 5, 6]

numbers.push(7, 8, 9)
# => [1, 2, 3, 4, 5, 6, 7, 8, 9]

numbers.pop
# => 9

numbers
# => [1, 2, 3, 4, 5, 6, 7, 8]
]
```