Intro to Course

- Learn ruby
- Learn Object Oriented Programming
- Give homework
- Do some logic problems
- Ask questions
- Give me feedback

Intros

— Who are you
— What programming experience do you have
— How do you know me/someone in the group?

What is Ruby

— The greatest language ever
— Made by a guy we call Matz
— Miniswan

Who uses Ruby

-NASA
- Motorola
- Google
- GoPro
- Pinterest
- LivingSocial
- Groupon
- Hulu
- Twitter

```
puts "Hello World"
```

or Why is ruby so great?

- Human-oriented language
- Principle of Least Surprise
- Can overwrite anything
- Ruby Gems - Existing code that you can use

Set a variable to equal anything!

— Weakly typed
— Dynamically typed
— Duck typing

More cool stuff

- Interpreted language
  - MRI: Matz's Ruby Interpreter
  - Exception: JRuby & Rubinius (compiled)
- Everything is an object
- blocks and lambdas

Our first program!

Use a text editor and make a file named `ex1.rb`

Type `puts "Hello World!"`

In your console run `ruby ex1.rb`

Congratulations!

Using irb

— Interactive ruby shell
— a REPL for programming

Open terminal and type `irb`

Type `puts "Hello World!"`

???

Profit!

More irb

`ctrl + d` **or type** `exit` **to quit**

```
>> 100 + 32
=> 132
```

Variables

- **Local** `first_name`
- **Instance** `@first_name`
  - **storing information for individual objects**
- **Class** `@@first_name`
  - **store information per class hierarchy**
- **Global** `$FIRST_NAME`

How many days in a year?

```
>> 365 * 24 * 60
=> 525600
```

```
>> days = 365
=> 365
>> hours = 24
=> 24
>> minutes = 60
=> 60
>> days * hours * minutes
=> 525600
```

When you type days = 365, irb responds by printing 365. Why?

you’re assigning the value 365 to a variable called days

When irb sees any expression, it prints out the value of that expression

it’s the same behavior that lets you type 2 + 2 into irb and see the result without having to use an explicit print statement

Everything is an object

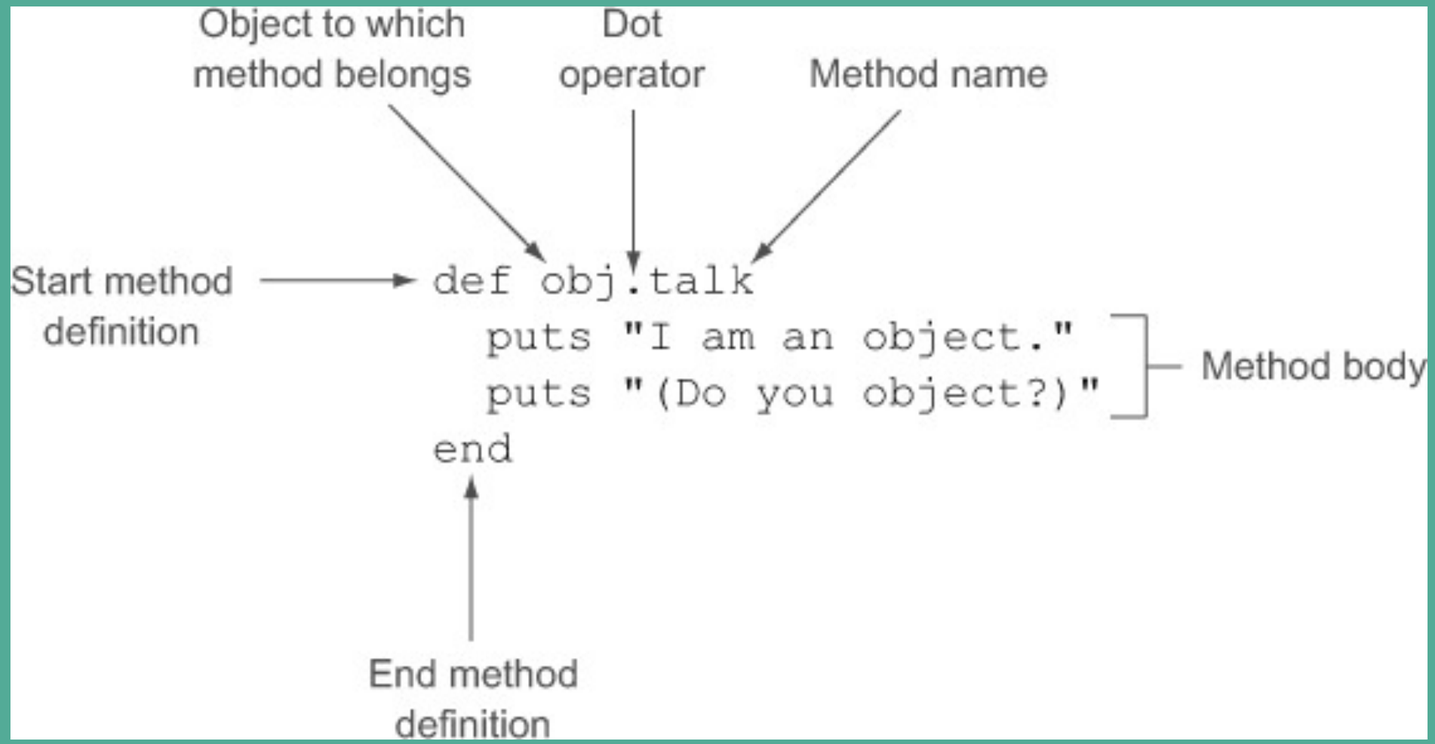
— "Rather than ask in the abstract whether a equals b, you ask a whether it considers itself equal to b"
— every object is an example or instance of a particular class

Create a new object

```
obj = Object.new
```

Let's make the object talk

```
def obj.talk
   puts "I am an object."
   puts "(Do you object?)"
end
```


Object to which method belongs
Dot operator
Method name
Start method definition
def obj.talk
puts "I am an object."
puts "(Do you object?)"
end
Method body
End method definition

```
>> obj.talk
=> I am an object.
=> Do you object?
```

The object obj understands, or responds to, the message talk

The dot (.) is the message-sending operator

Methods with arguments

Methods in Ruby are much like mathematical functions: input goes in, the wheels turn, and a result comes out

Celsius-to-Fahrenheit converter

```
def obj.c2f(c)
   c * 9.0 / 5 + 32
end

>> puts obj.c2f(100)
=> 212.0
```

parentheses are optional (but preferred)

```
#valid
def obj.c2f c
  c * 9.0 / 5 + 32
end

#valid
obj.c2f 100
```

Implicit return

- every method call returns a value
- The return value of any method is the same as the value of the last expression evaluated during execution of the method

Explicit return is optional

```
def obj.c2f(c)
   return c * 9.0 / 5 + 32
end
```

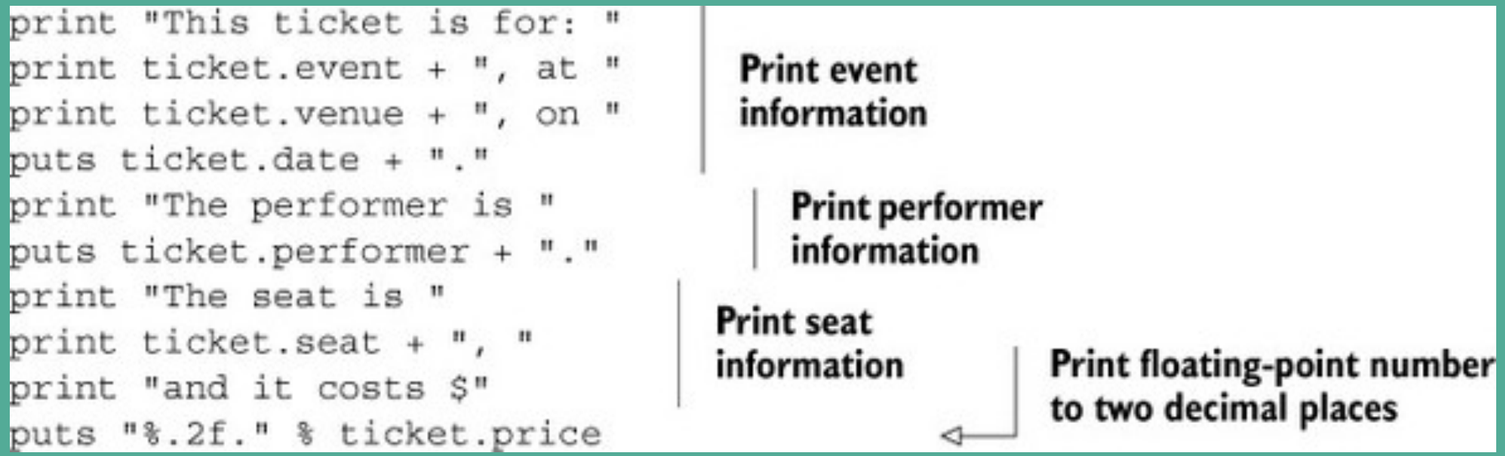
The use of keyword `return` is usually optional

have to use `return` if you want to return from somewhere in the middle of a method

empty method body returns nil.

Creating an object

```
ticket = Object.new

def ticket.date
  "01/02/03"
end

def ticket.venue
    "Town Hall"
end

def ticket.venue
    "Author's reading"
end

def ticket.performer
    "Mark Twain"
end

def ticket.seat
    "Second Balcony, row J, seat 12"
end

def ticket.price
    5.50
end
```


print "This ticket is for: "
print ticket.event + ", at "
print ticket.venue + ", on "
puts ticket.date + "."
print "The performer is "
puts ticket.performer + "."
print "The seat is "
print ticket.seat + ", "
print "and it costs $"
puts "%.2f." % ticket.price
Print event information
Print performer information
Print seat information
Print floating-point number to two decimal places

This ticket is for: Author's reading, at Town Hall, on 01/02/03.
The performer is Mark Twain.
The seat is Second Balcony, row J, seat 12, and it costs $5.50.

Too wordy :(

Using string interpolation

gives you a way to drop anything into a string

Whatever's inside the interpolation operator #{...} gets calculated separately, and the results of the calculation are inserted into the string

```
puts "This ticket is for: #{ticket.event}, at #{ticket.venue}." +
  "The performer is #{ticket.performer}." +
  "The seat is #{ticket.seat}, " +
  "and it costs $#{"%.2f." % ticket.price}"
```

You try!

- Make a file named ex2.rb
- Make an new Object
- Set that new object to a variable named `me`
- Define some methods that return your age, job, name, address
- Use `puts` to print out all the info
- `$ ruby ex2.rb`

Booleans

variables can also store something as `true` or `false`

Instead of this

```
def ticket.availability_status
  "sold"
end
```

We can use this

```
def ticket.available?
  false
end
```

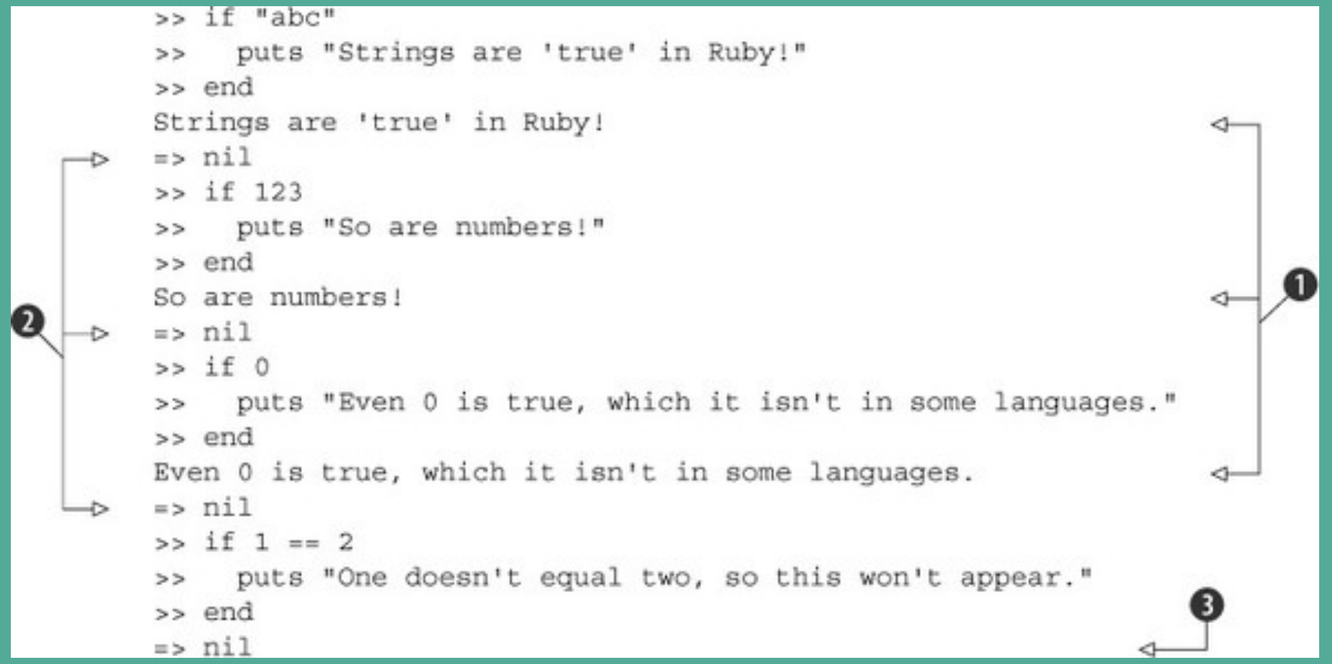
Ruby lets you write methods that evaluate to true or false and make the method calls look like questions:

```
if ticket.available?
  puts "You're in luck!"
else
  puts "Sorry--that seat has been sold."
end
```

Everything in Ruby has a Boolean value

true and false are objects

false and nil are indicators of a negative outcome


>> if "abc"
>>   puts "Strings are 'true' in Ruby!"
>> end
Strings are 'true' in Ruby!
=> nil
>> if 123
>>   puts "So are numbers!"
>> end
So are numbers!
=> nil
>> if 0
>>   puts "Even 0 is true, which it isn't in some languages."
>> end
Even 0 is true, which it isn't in some languages.
=> nil
>> if 1 == 2
>>   puts "One doesn't equal two, so this won't appear."
>> end
=> nil
1
2
3

Objects

Having a unique ID number for every object can come in handy when you're trying to determine whether two objects are the same as each other

```
>> puts 100 == 100
#> true

obj = Object.new
puts obj.object_id
str = "Strings are objects too!"
puts str.object_id
puts 100.object_id
```

```
a = Object.new
b = a
puts "a's id is #{a.object_id} and b's id is #{b.object_id}."
```

```
obj = Object.new
obj.quack
```

Did you get an error? What is it?

Duck typing

```
obj = Object.new
if obj.respond_to?("quack")
  obj.quack
else
  puts "Sorry, the object doesn't understand the 'quack' message."
end
```

`respond_to?` **method exists for all objects; you can ask any object whether it responds to any message Example of reflection: examining the state of a program while it’s running**

Make a file named ex3.rb

```
# from ex2.rb
me = Obj.new
...

print "Information desired: "
request = gets.chomp

if request == "name"
  me.name
elsif request == "job"
  me.job
...
else
  puts "Error, did not understand"
end

$ ruby ex3.rb
```

Too wordy :(

ex4.rb

```
# from ex2.rb
me = Obj.new
...

print "Information desired: "
request = gets.chomp

if request.respond_to?(request)
  puts me.send(request)
else
  puts "Error, did not understand"
end

$ ruby ex4.rb
```

We can `send` methods on objects
Like calling them with . notation

Commonly known as meta-programming

Methods and arguments

- The difference between required and optional arguments
- Methods you write in Ruby can take zero or more arguments
- Or a variable number of arguments
- How to assign default values to arguments
- The rules governing the order in which you have to arrange the parameters in the method signature so that Ruby can make sense of argument lists in method calls and bind the parameters correctly

Required and optional arguments

```ruby
obj = Object.new

def obj.compliment(name)
  puts "#{name} is the nicest person I know"
end

>> obj.compliment("Chris")
#> "Chris is the nicest person I know"

>> obj.compliment("Chris", "Paul")
#> ArgumentError: wrong number of arguments (2 for 1)
```

```ruby
def two_or_more(a,b,*c)
  puts "I require two or more arguments!"
  puts "And sure enough, I got: "
  p a, b, c
end
```

```
I require two or more arguments!
And sure enough, I got:
1
2
[3, 4, 5]
```

Multi args

putting a `*` in front of an argument will just assign the rest of the args to an array

```
[3, 4, 5]
```

An array is a list of things, we'll talk about it later.

Default values for arguments

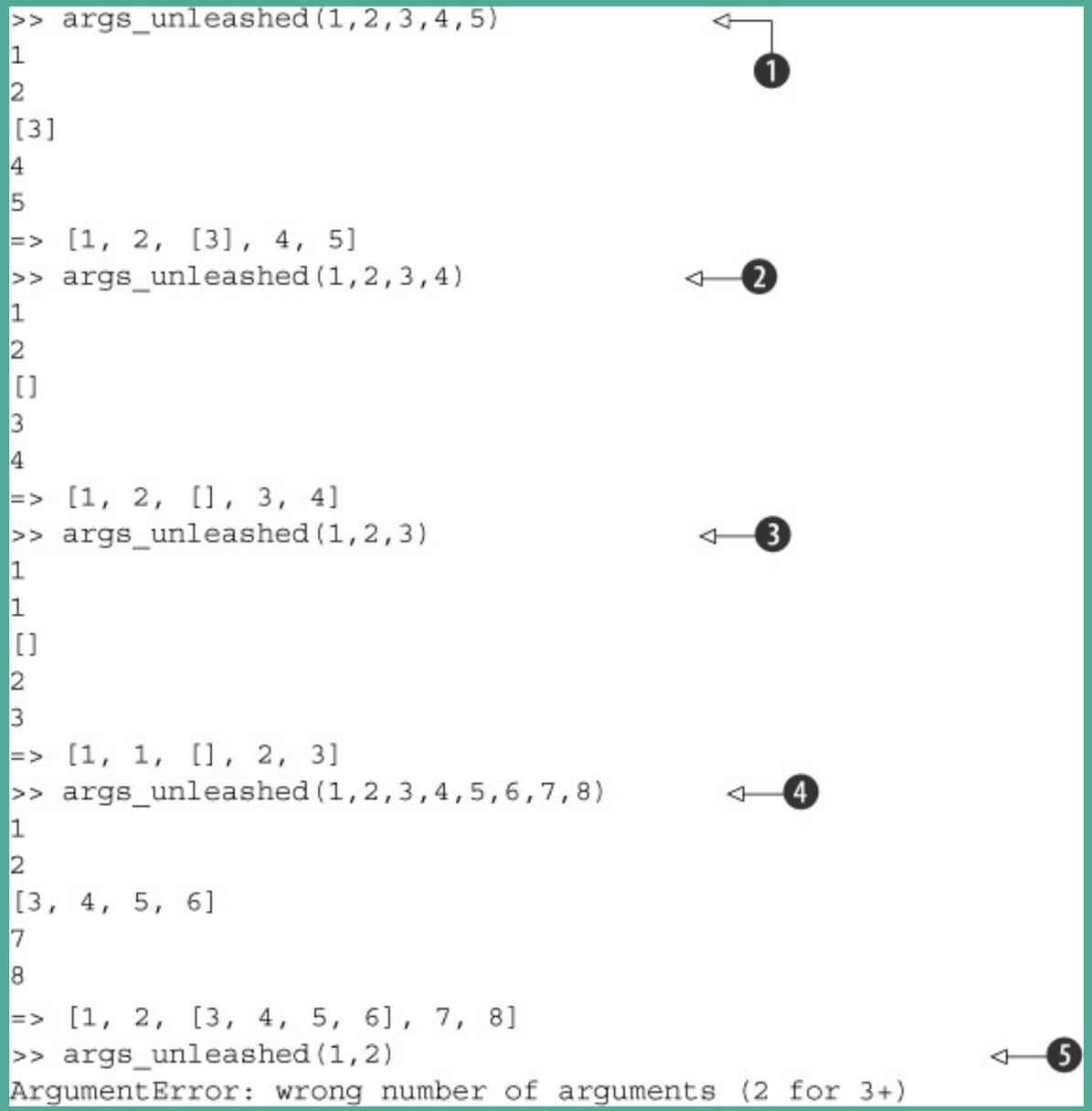
When you supply a default value for an argument, the result is that if that argument isn't supplied, the variable corresponding to the argument receives the default value.

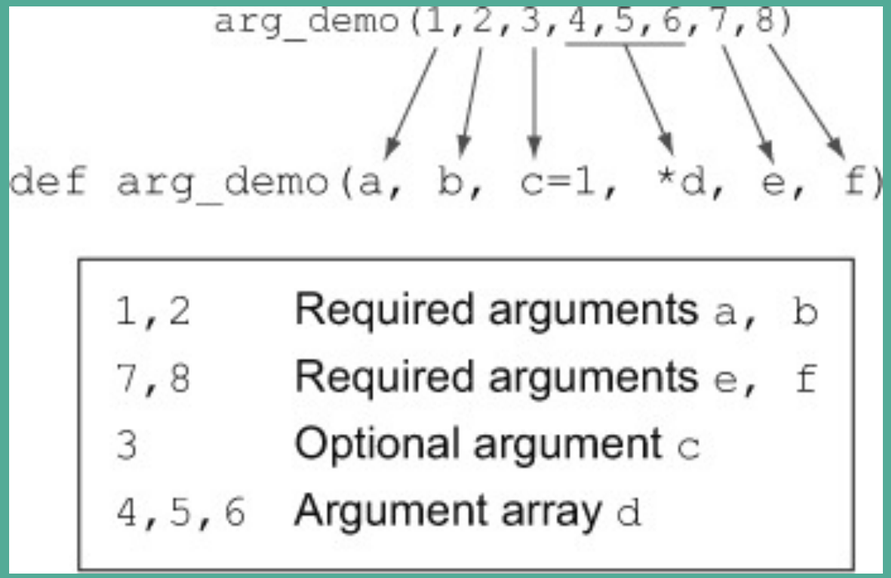
```
def best_friends(a,b,c= "Kindred")
  puts "My best friends are: #{a}, #{b}, and #{c}"
end

>> best_friends("Rob", "Anu")
#> My best friends are: Rob, Anu, and Kindred

>> best_friends("Rob", "Anu", "Jake")
#> My best friends are: Rob, Anu, and Jake
```

Let's go crazy

```
def args_unleashed(a,b=1,*c,d,e)
   puts "Arguments:"
   p a,b,c,d,e
end
```

Note can’t argument sponge (`*c`) to the left of any default-valued arguments


>> args_unleashed(1,2,3,4,5)
1
2
[3]
4
5
=> [1, 2, [3], 4, 5]
>> args_unleashed(1,2,3,4)
1
2
[]
3
4
=> [1, 2, [], 3, 4]
>> args_unleashed(1,2,3)
1
1
[]
2
3
=> [1, 1, [], 2, 3]
>> args_unleashed(1,2,3,4,5,6,7,8)
1
2
[3, 4, 5, 6]
7
8
=> [1, 2, [3, 4, 5, 6], 7, 8]
>> args_unleashed(1,2)
ArgumentError: wrong number of arguments (2 for 3+)
1
2
3
4
5

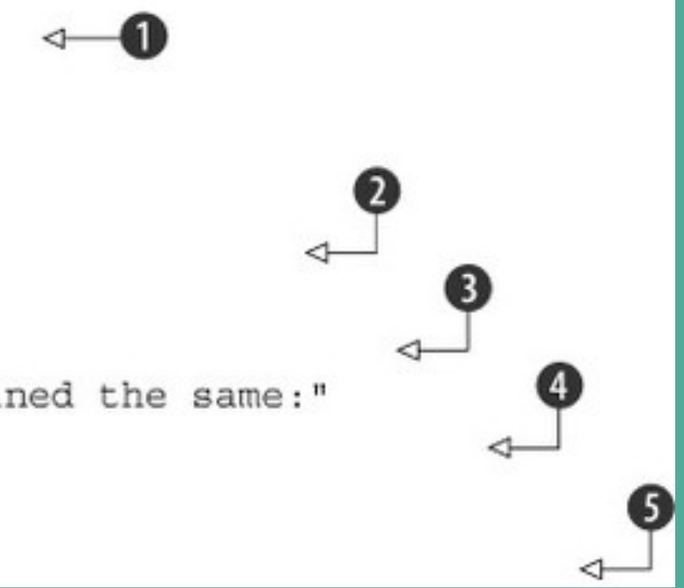

arg_demo(1,2,3,4,5,6,7,8)
def arg_demo(a, b, c=1, *d, e, f)
1,2 Required arguments a, b
7,8 Required arguments e, f
3 Optional argument c
4,5,6 Argument array d

Local variables

local in local variables pertains to the fact that they have limited scope: a local variable is only visible in a limited part of a program, such as a method definition

Local variable names can be reused in different scopes

```
def say_goodbye
  x = "Goodbye"
  puts x
end

def start_here
  x = "Hello"
  puts x
  say_goodbye
  puts "Let's check whether x remained the same:"
  puts x
end

start_here
```




```
Hello
Goodbye
Let's check whether x remained the same:
Hello
```

```
def say_goodbye
  str = "Hello"
  abc = str
  str = "Goodbye"
  puts str
  puts abc
end

>> say_goodbye
#> Hello
#> Goodbye


def say_goodbye
  str = "Hello"
  abc = str
  str.replace("Goodbye")
  puts str
  puts abc
end

>> say_goodbye
#> Goodbye
#> Goodbye
```

variables in Ruby (with a few exceptions, most notably variables bound to integers) don’t hold object values

str doesn’t contain "Hello"

str contains a reference to a string object

Exceptions to what I just said

— integers
— symbols (which look like :this)
— true, false, and nil

variables will hold the value of those objects itself

Mess with objects

```
s1 = "String version 1"
s2 = s1.dup

s1 == s2
#> true

s1.equal? s2
#> false

s3 = "Frozen string"
s3.freeze
s3.replace("New string")

#> RuntimeError: can't modify frozen String
```

if you `clone` a frozen object, the clone is also frozen

Classes

— Classes are important in Ruby; they're a way to bundle and label behaviors (you can have a Person class, a Task class, and so on)