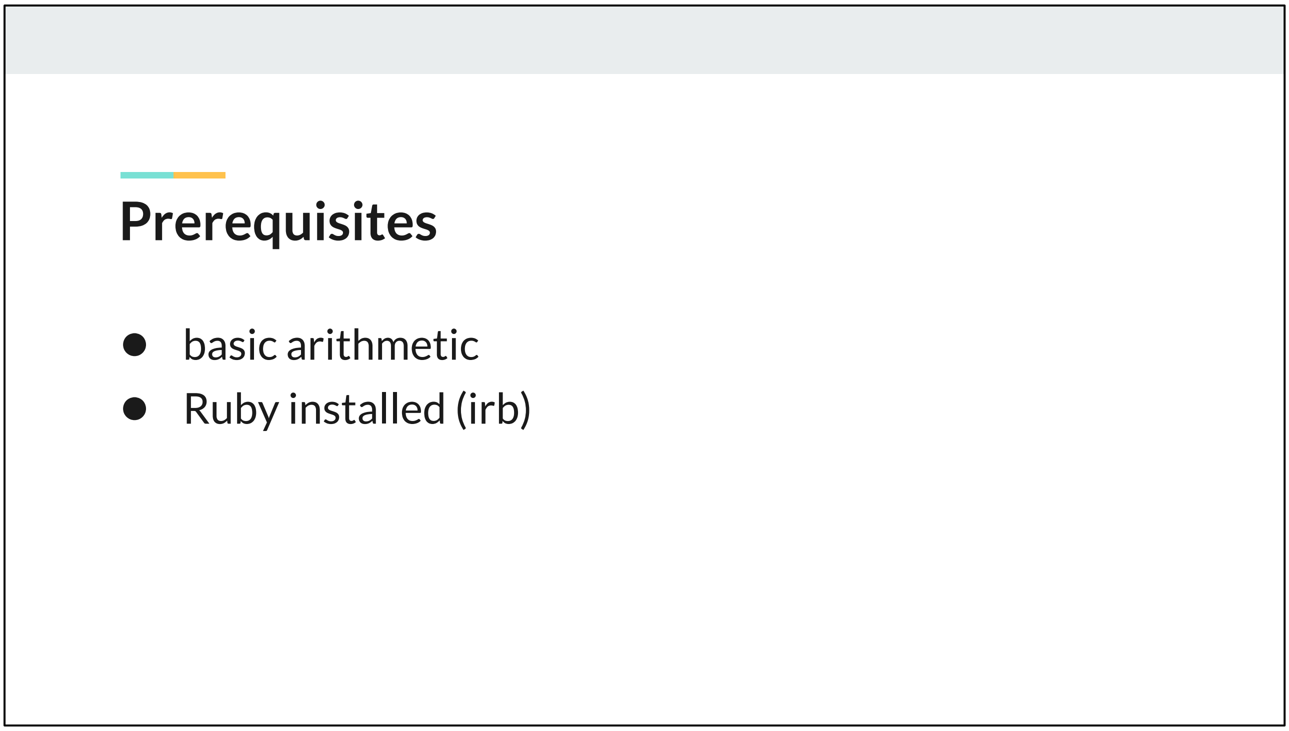
Ruby Syntax

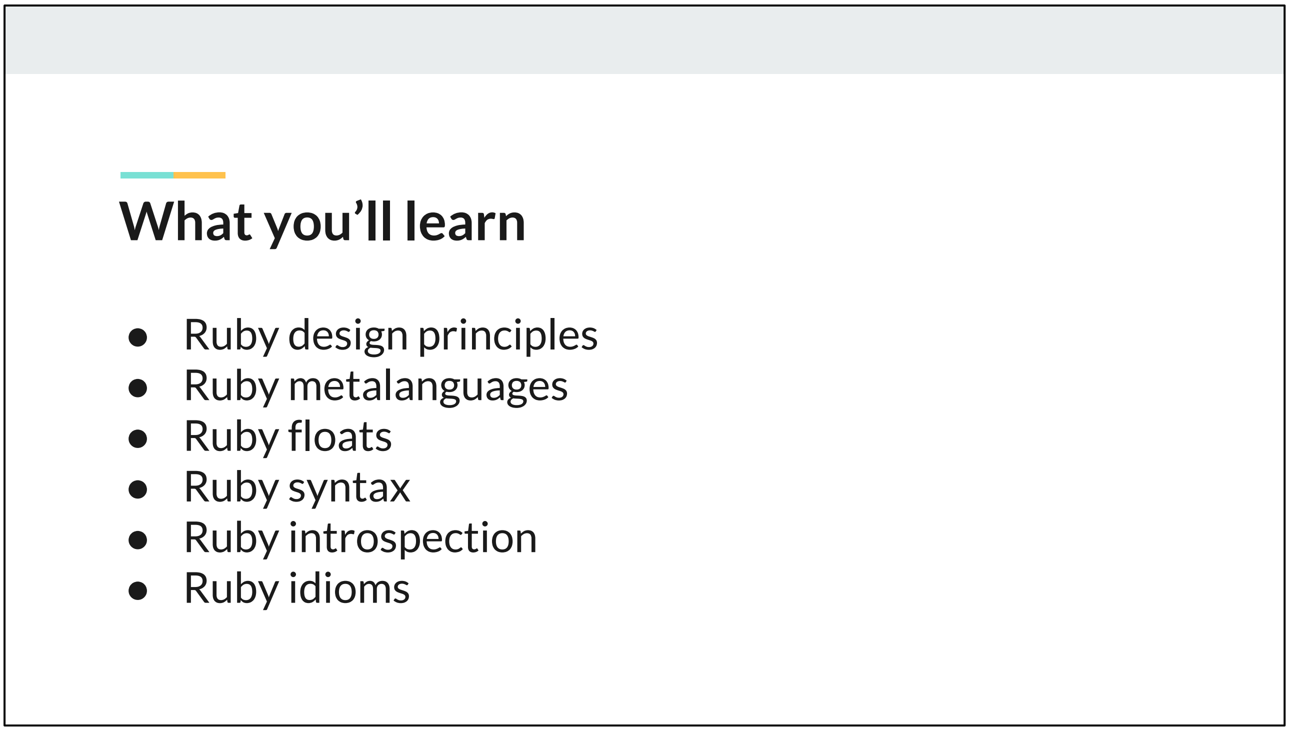


Prerequisites

- basic arithmetic
- Ruby installed (irb)

By “basic”, I’m talking about elementary and middle school math.

Finally, you should have Ruby installed. You should be able to type i-r-b in the Terminal and get something interesting to happen.

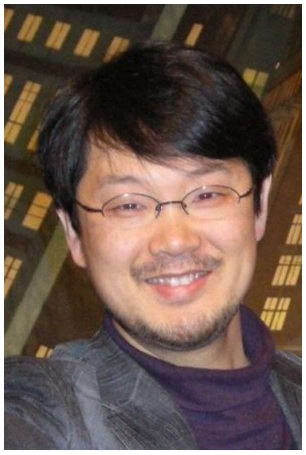


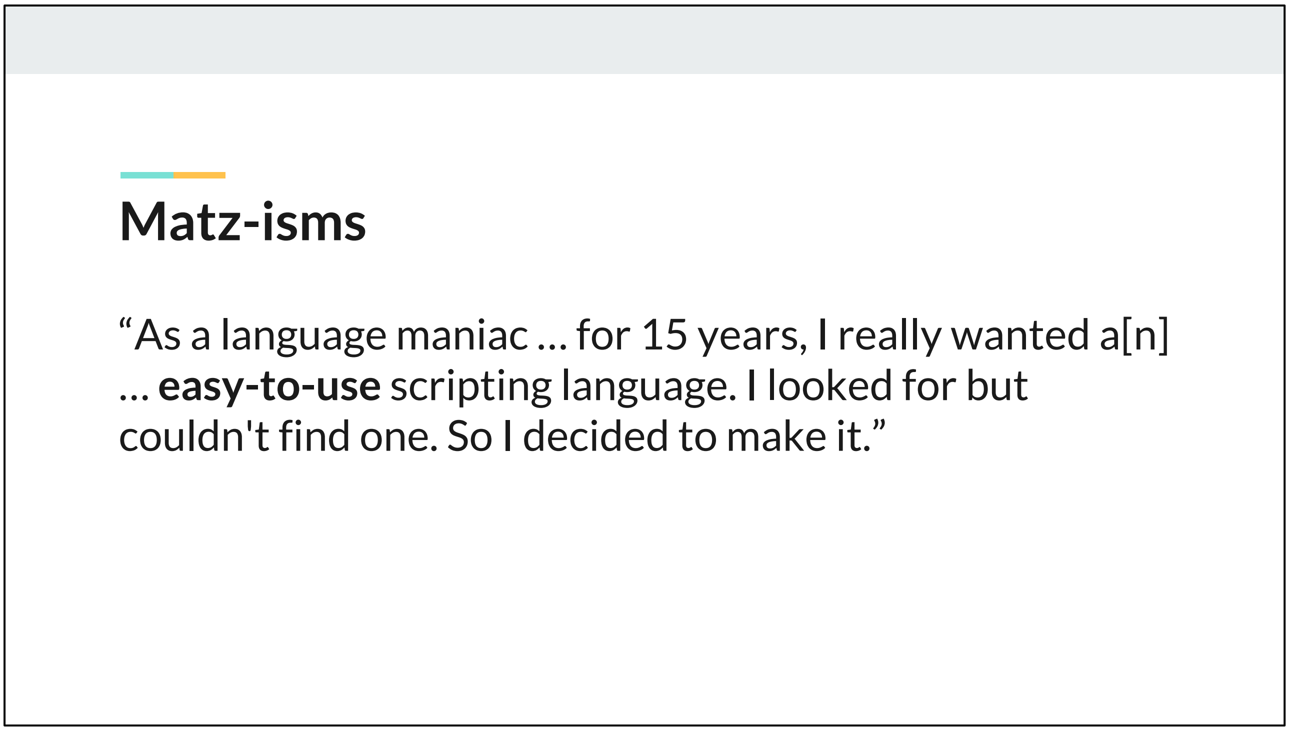
What you'll learn

- Ruby design principles
- Ruby metalanguages
- Ruby floats
- Ruby syntax
- Ruby introspection
- Ruby idioms

What is Ruby?

Ruby was created in the mid-90s by Japanese computer scientist and software programmer Yukihiro "Matz" Matsumoto.

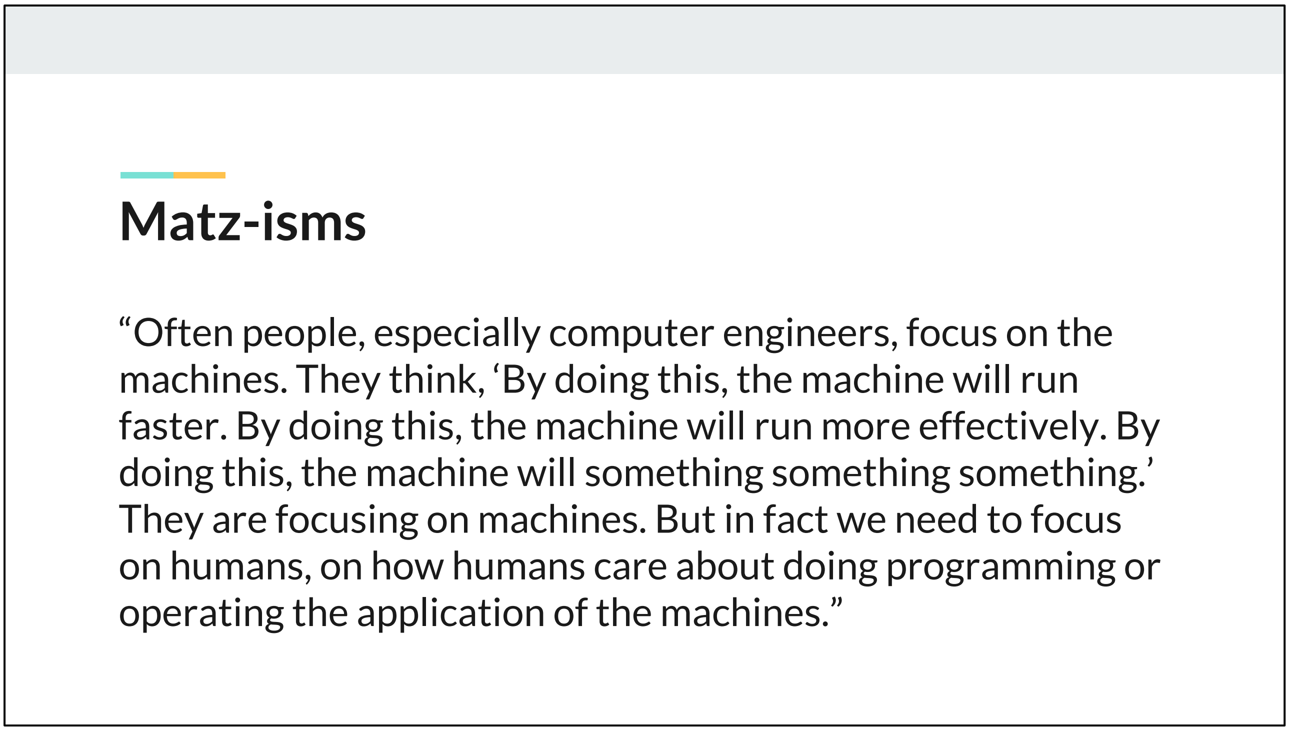




Matz-isms

“As a language maniac … for 15 years, I really wanted a[n] … **easy-to-use** scripting language. I looked for but couldn't find one. So I decided to make it.”

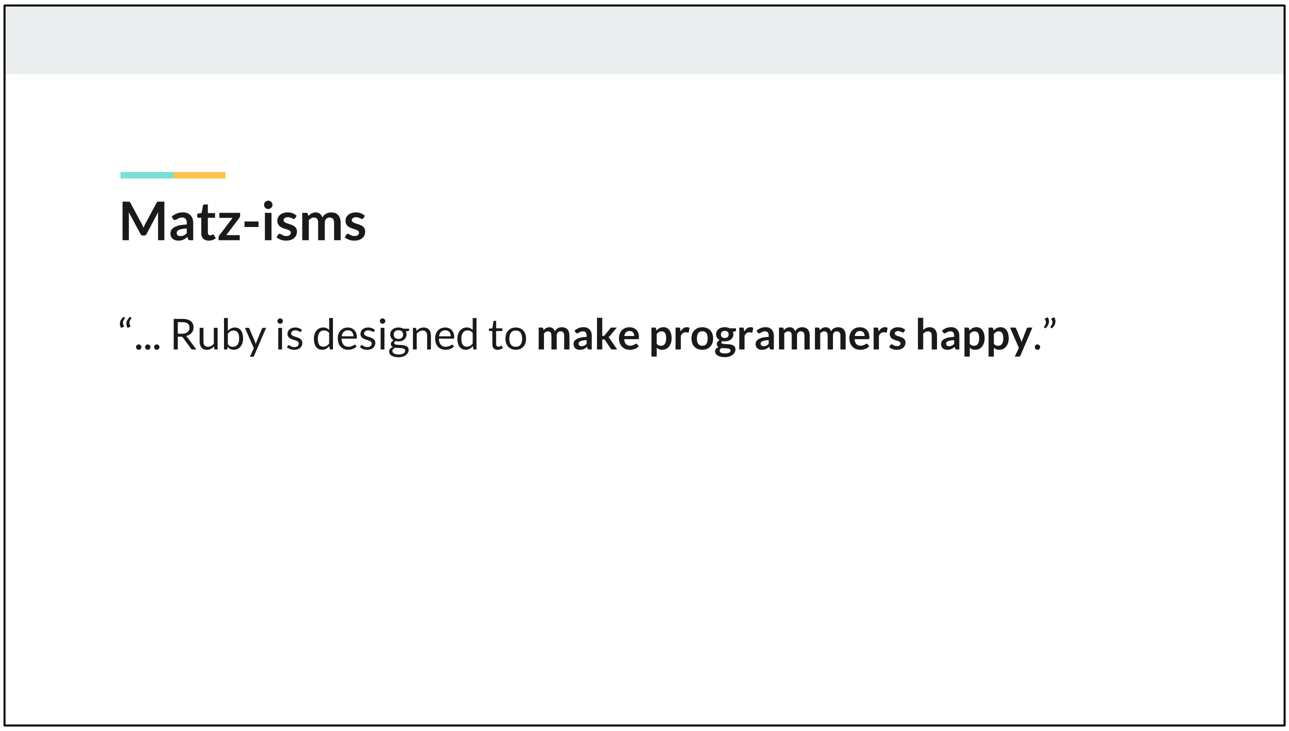
From this quote, you can see how Matz designed Ruby to be easy-to-use.



Matz-isms

“Often people, especially computer engineers, focus on the machines. They think, ‘By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something.’ They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines.”

This quote demonstrates how Matz designed Ruby to be human-focused rather than machine-focused.

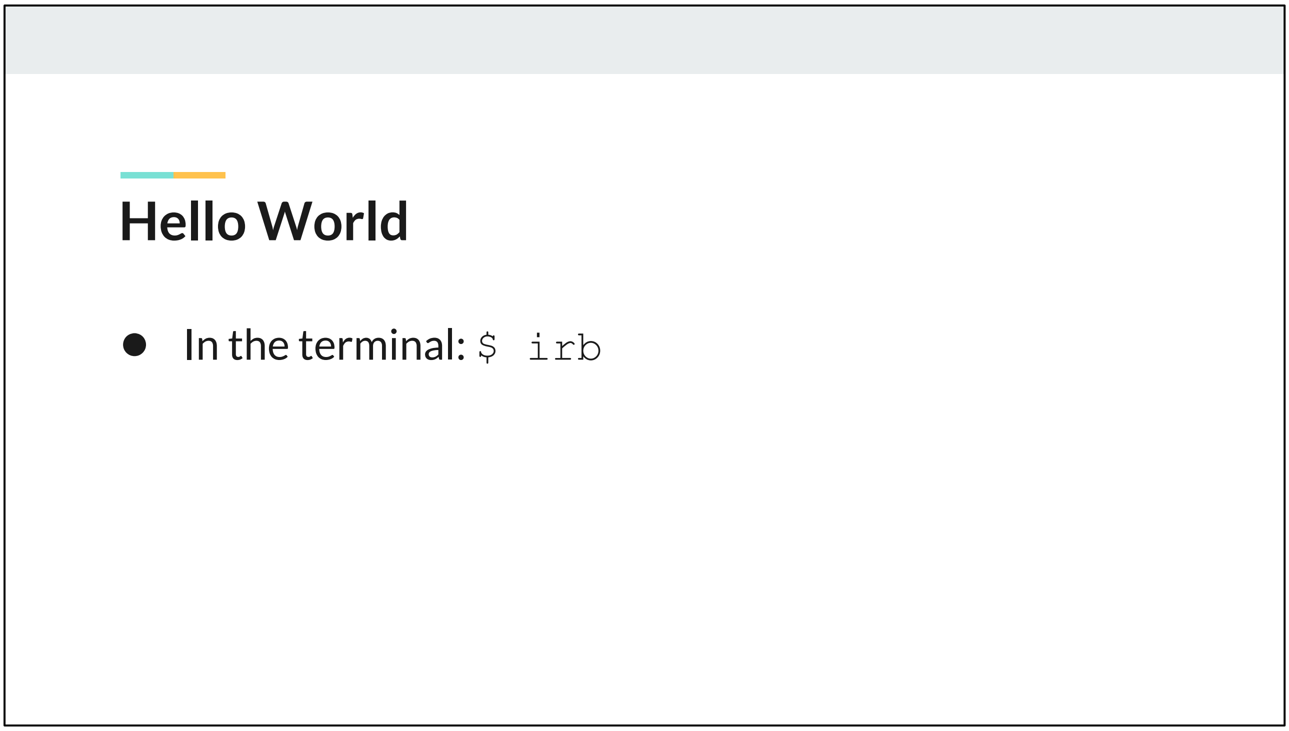


Matz-isms

"... Ruby is designed to **make programmers happy**."

Overall, Matz designed Ruby to make programmers happy.

This is a revolutionary design principle. Not very often has a technology been designed to make *your* job - as developers - more enjoyable. Sure there have been things built to make you more efficient, more reliable, more whatever. But very few go so far as to focus on developer *happiness*



Hello World

- In the terminal: `$ irb`

Type “irb” into your Terminal now and press “Enter”.

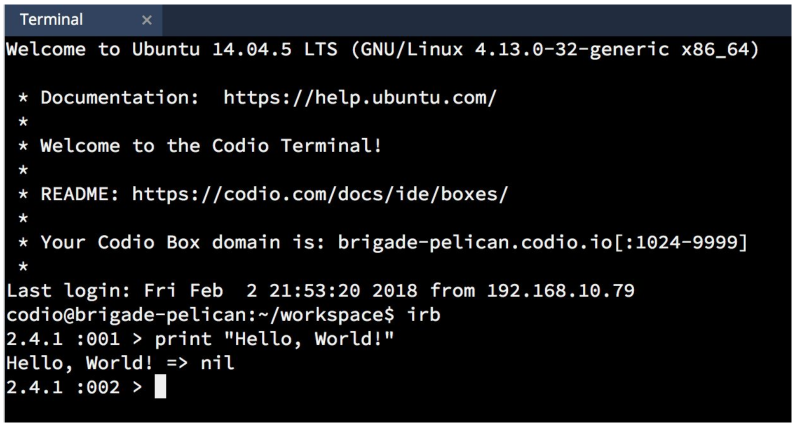
IRB (interactive ruby shell) run a Ruby REPL. REPL is an acronym for Read-Eval-Print-Loop. This program will read a line of Ruby code, evaluate it, print the result, and repeat (or loop).

Hello World

```
print "Hello world!"
```

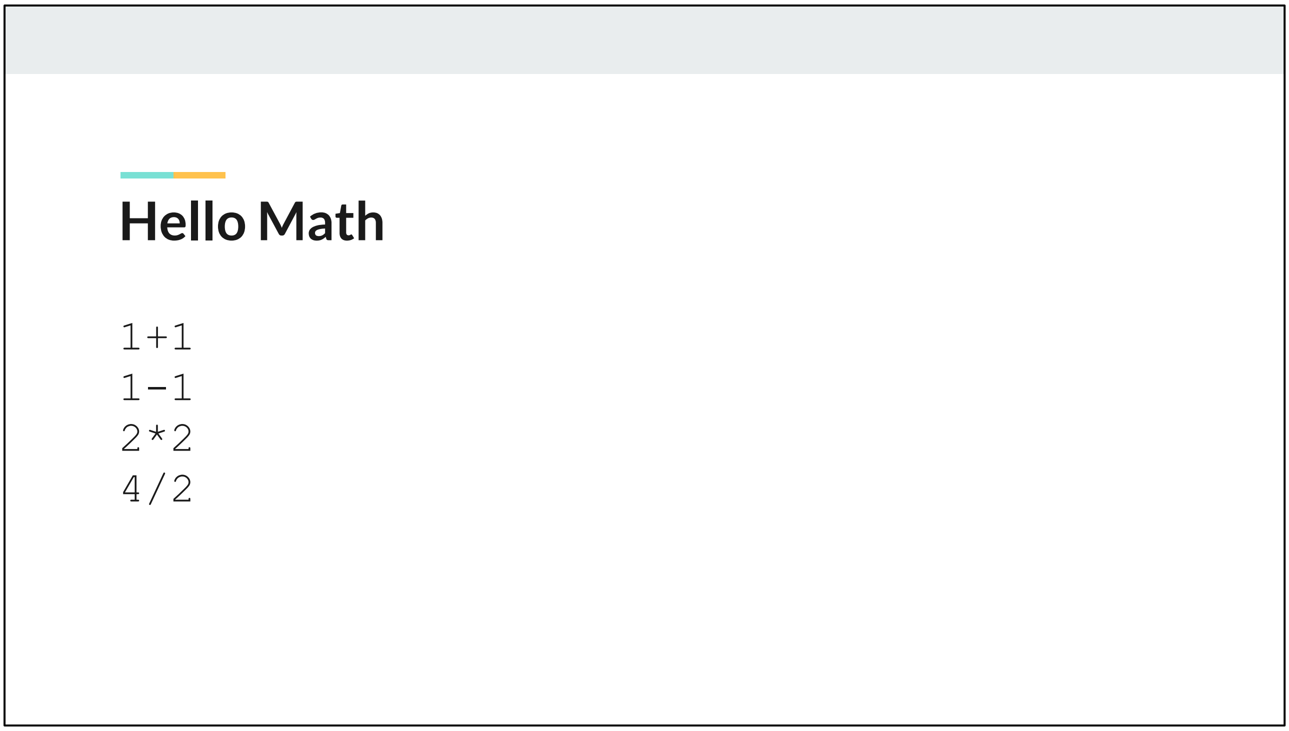
Type this into the Ruby REPL and press Enter.

Hello World



Terminal ×

```
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.13.0-32-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
 *
 * Welcome to the Codio Terminal!
 *
 * README: https://codio.com/docs/ide/boxes/
 *
 * Your Codio Box domain is: brigade-pelican.codio.io[:1024-9999]
 *
Last login: Fri Feb  2 21:53:20 2018 from 192.168.10.79
codio@brigade-pelican:~/workspace$ irb
2.4.1 :001 > print "Hello, World!"
Hello, World! => nil
2.4.1 :002 >
```

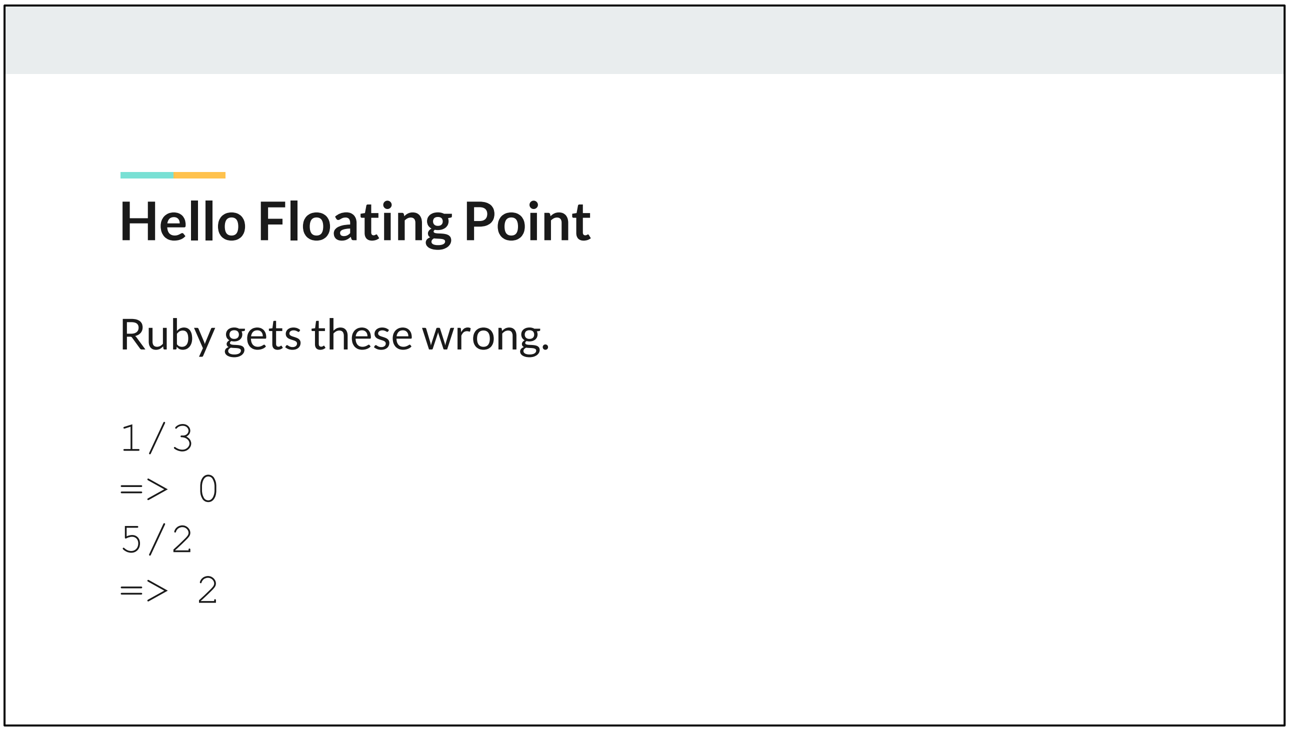


Hello Math

```
1+1
1-1
2*2
4/2
```

Ruby can do simple math problems as well. + is addition. - is subtraction. And the asterisk (*) is multiplication.

The forward-slash is division. It’s called a forward slash because, if someone is walking left-to-right (like we read), this is what it looks like when that person leans forward.



Hello Floating Point

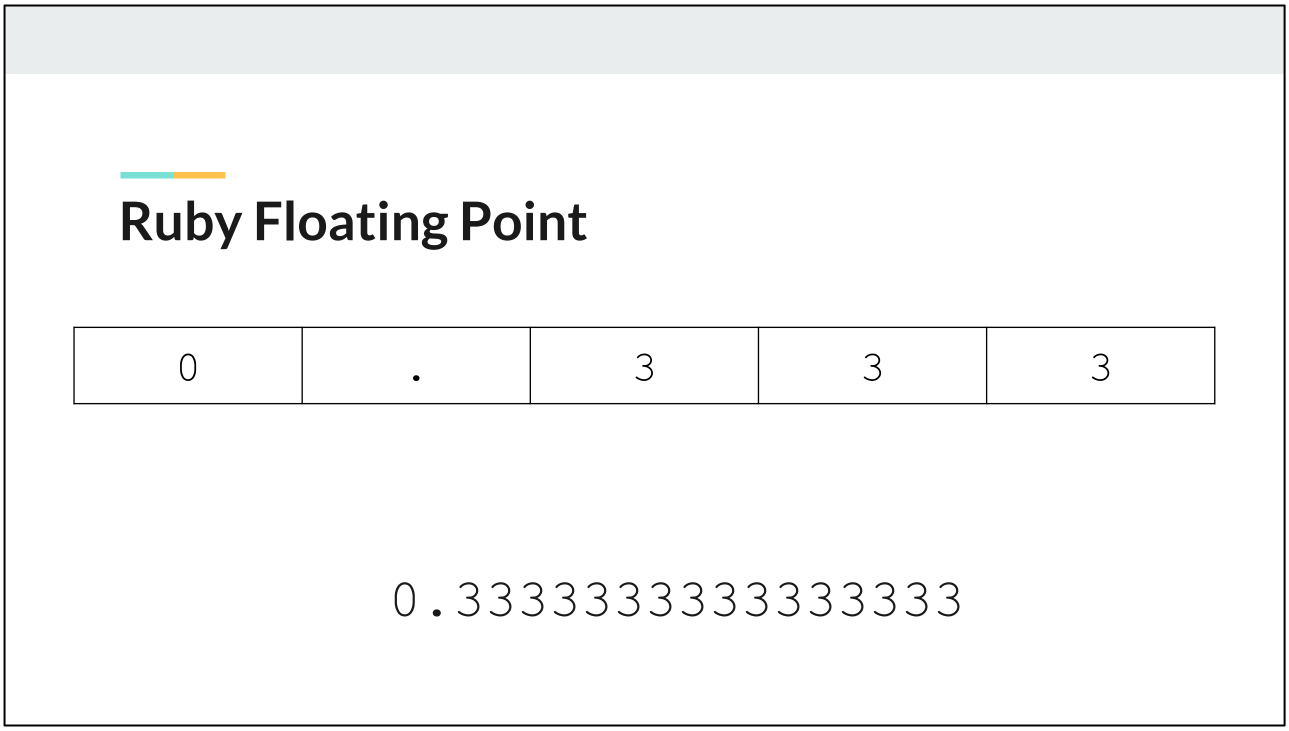
Ruby gets these wrong.

```
1/3
=> 0
5/2
=> 2
```

Ruby actually gets these math problems wrong.

⅓ returns 0 instead of 0.333333…
5/2 returns 2 instead of 2.5.

Why is that?



Ruby Floating Point

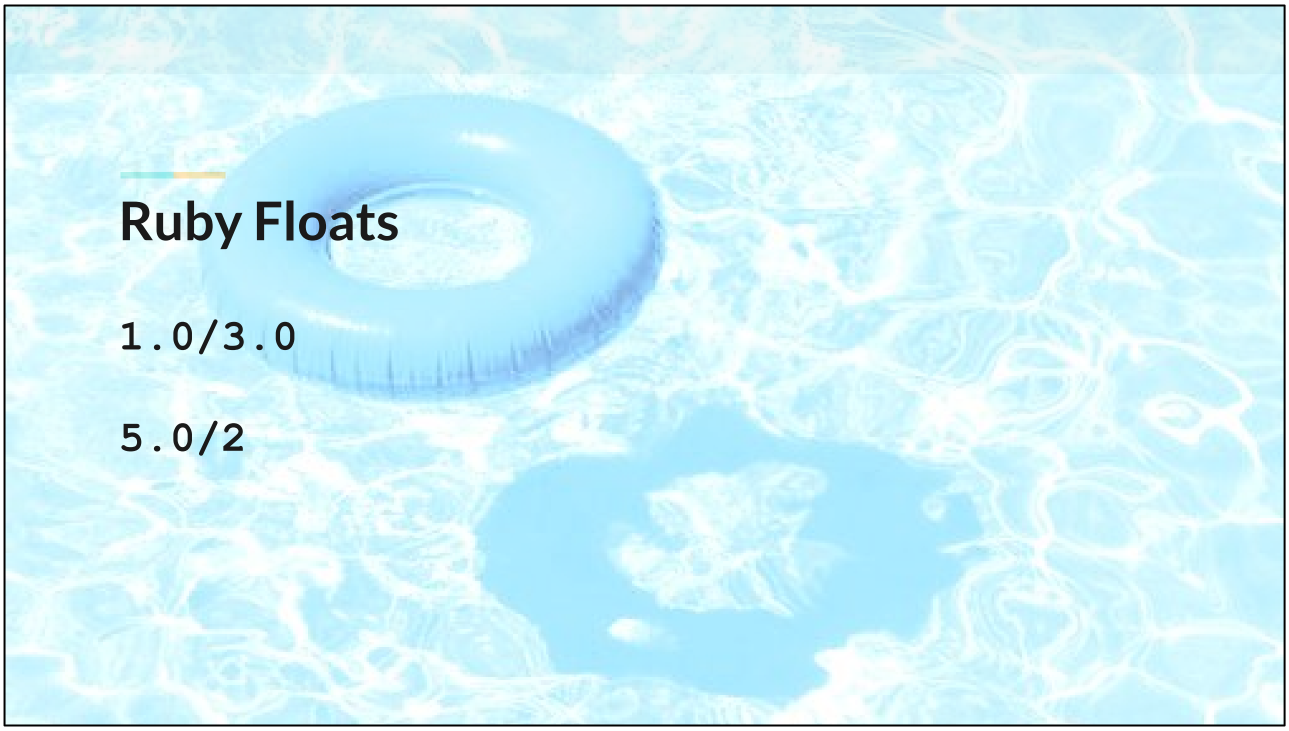
0	.	3	3	3

```
0.3333333333333333
```

It turns out that numbers with decimal points are little different than numbers without decimal points.

If the answer to ⅓ is a 0 followed by an infinite number of 3’s, how do you print that result? How does your laptop not run out of space running this calculation?

The answer is actually quite simple. Ruby stops producing output after 16 digits passed the decimal point. This partial number is called a “floating-point” number or a “float”. Due to physical limitations, a computer can only *approximate* the real answer to these kinds of problems.



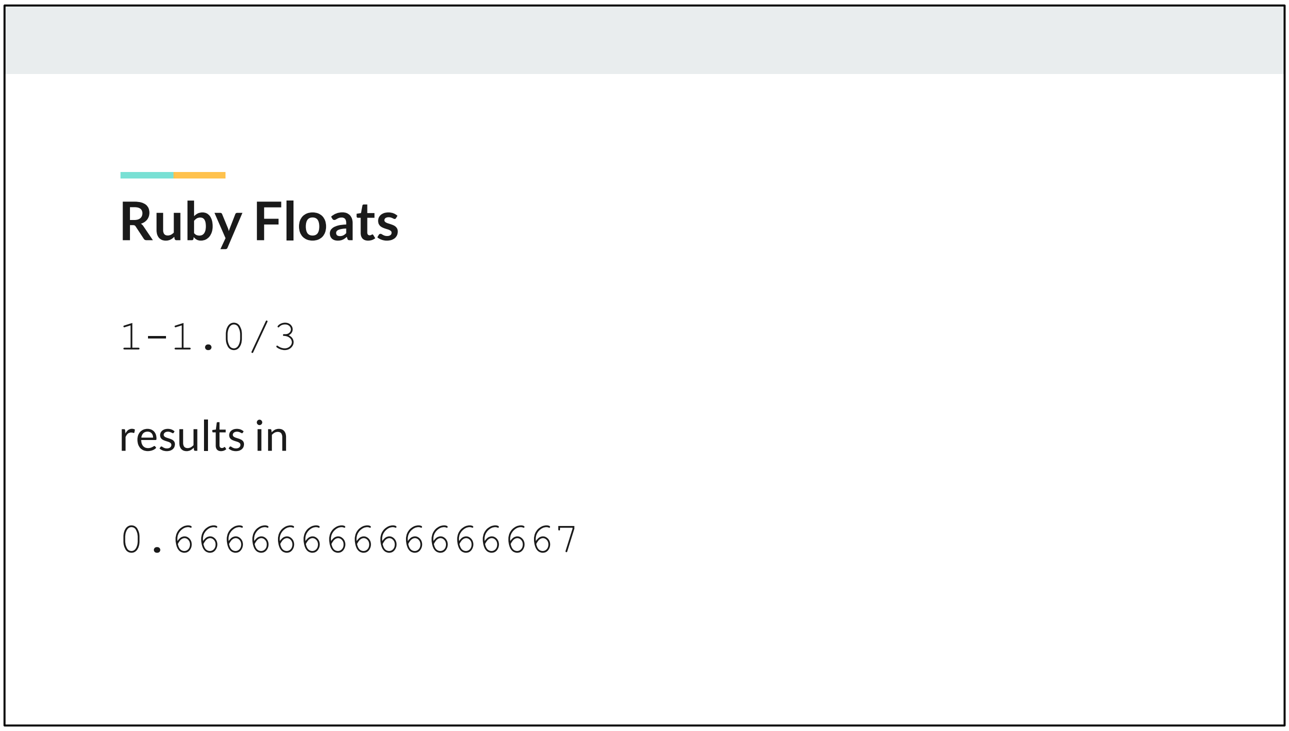
Ruby Floats

```
1.0/3.0

5.0/2
```

Because doing math with these kinds of numbers is “close, but not quite right”, Ruby forces you to be explicit. If both of your numbers use a decimal point, then Ruby assumes it’s ok to return an estimated answer with a decimal point.

Even if only one of your numbers has a decimal point, that signals to Ruby that, yes, it’s ok to output a “Float”.

In other words, Ruby is protecting you from unintentional mistakes by making you explicitly ask for a Float.



Ruby Floats

```
1-1.0/3
```

results in

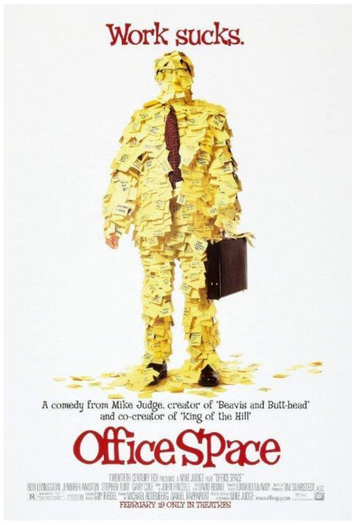
```
0.6666666666666667
```

So what’s the big deal? Why do we have to be so careful with Floats?

Since Floats are just approximations, sometimes your math is going to be off by a little. If that little bit doesn’t matter to you, then go nuts. In this case, the result should be 1 followed by an infinite number of 6’s, but instead the 16th digit after the decimal was rounded to 7.

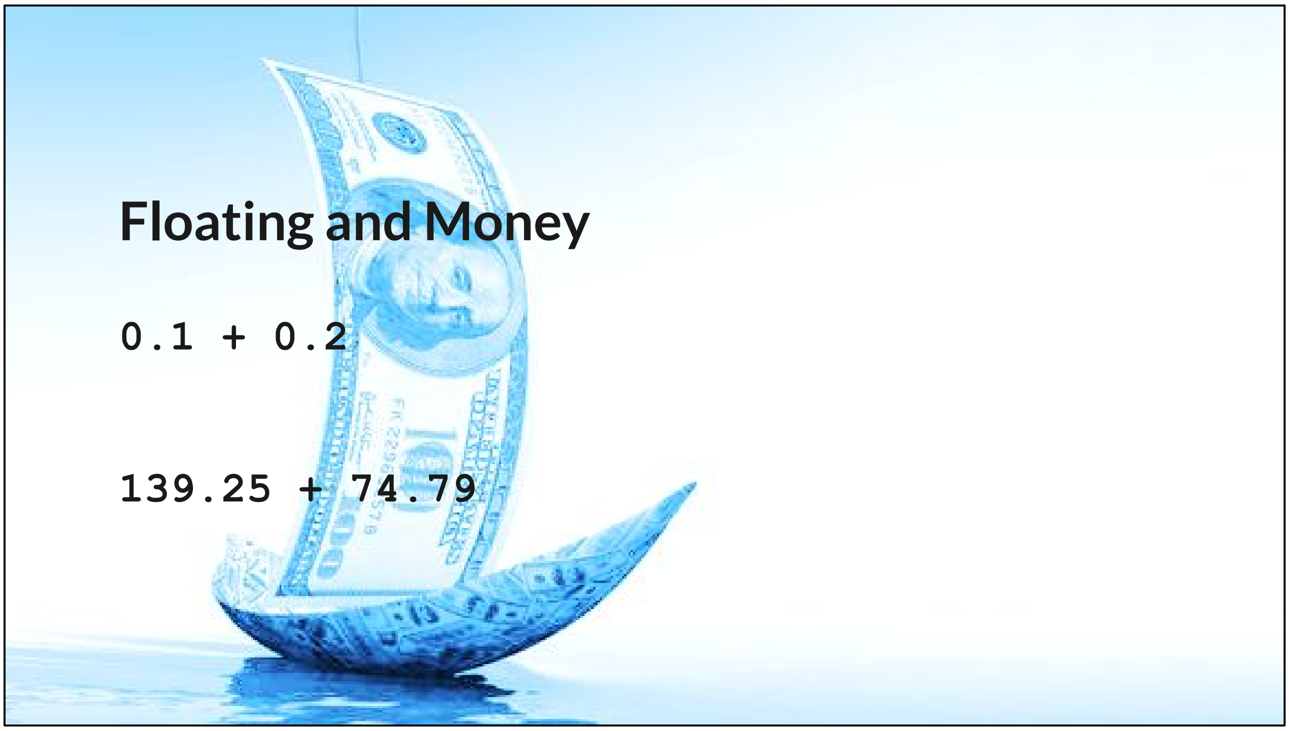
But if that error *does* matter to you, then it’s a big deal. For example, money. If you’re calculating interest payments on millions of accounts, these little errors, little fractions of a penny, after a while, can add up to big problems.

Rounding and Money


Work sucks.
A comedy from Mike Judge, creator of 'Beavis and Butt-head' and co-creator of 'King of the Hill'
Office Space
FEBRUARY 19 ONLY IN THEATRES

Did anyone see the movie Office Space? Spoiler alert…

Peter, Michael, and Samir hatch a plan to take advantage of these little money rounding errors to steal fractions of a penny from every transaction processed by their employer. They expect that, over time, the hardly noticeable rounding errors will turn into real money. But the errors actually amount to over $300,000 in a few days.



Floating and Money

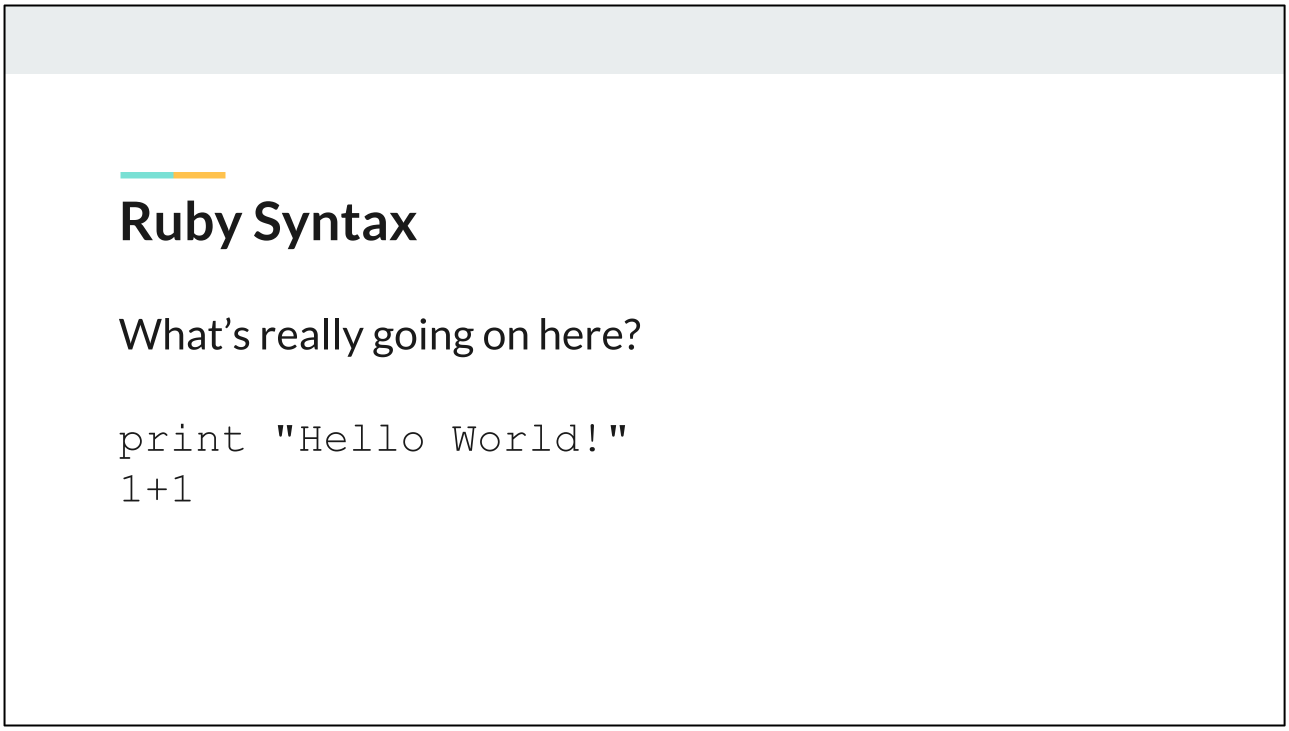
```
0.1 + 0.2


139.25 + 74.79
```

You might think to yourself, well, if I only add and subtract with money, and never multiply or divide (e.g. calculate taxes), then I'll be ok. How else would I generate half-pennies?

You'd be wrong. Both of these simple Float additions produce incorrect results.

http://0.30000000000000004.com/

If you find yourself having to do precise math with money (or, say, you're trying to calculate the appropriate trajectory to land a robot on Mars), then you need to be more careful. For example, with money, there are accounting rules about what to do with fractional currency.
http://www.daviddarling.info/encyclopedia/B/bankers_rounding.html



Ruby Syntax

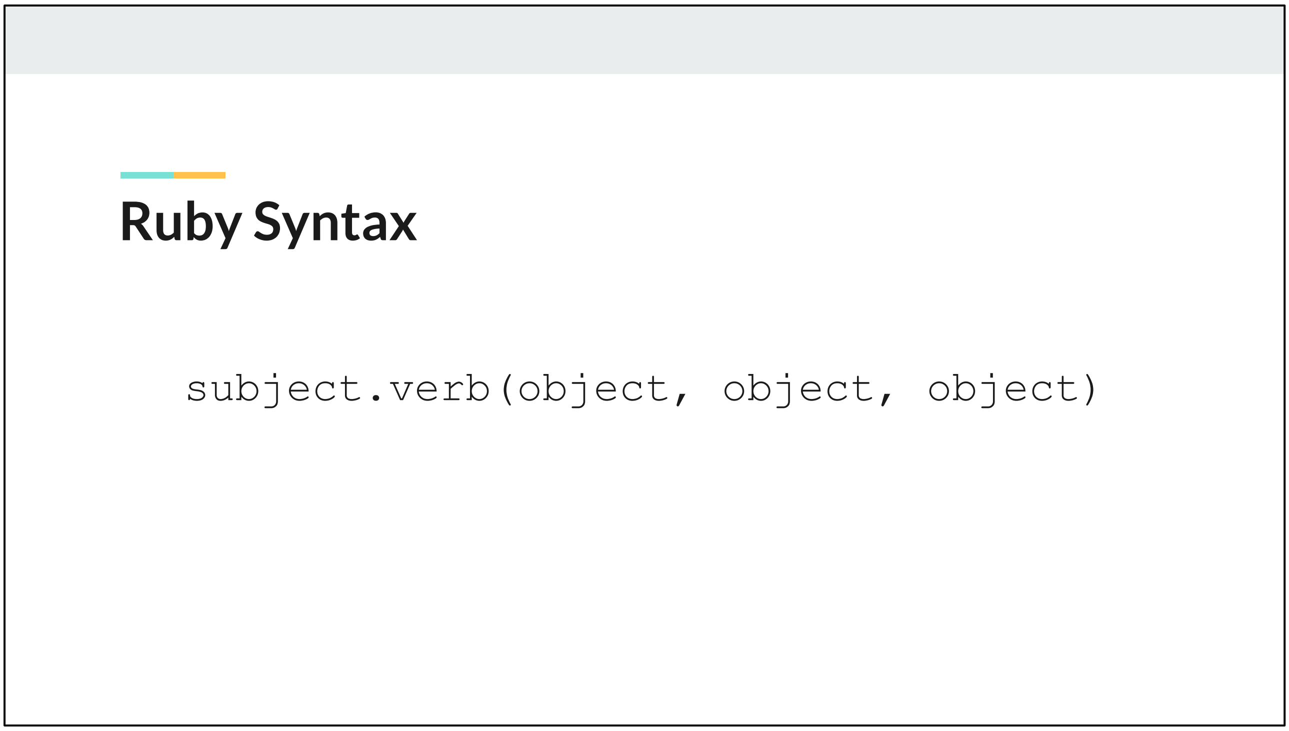
What’s really going on here?

```
print "Hello World!"
1+1
```

Back to the Ruby:

These are the only two types of Ruby expressions we’ve seen so far.

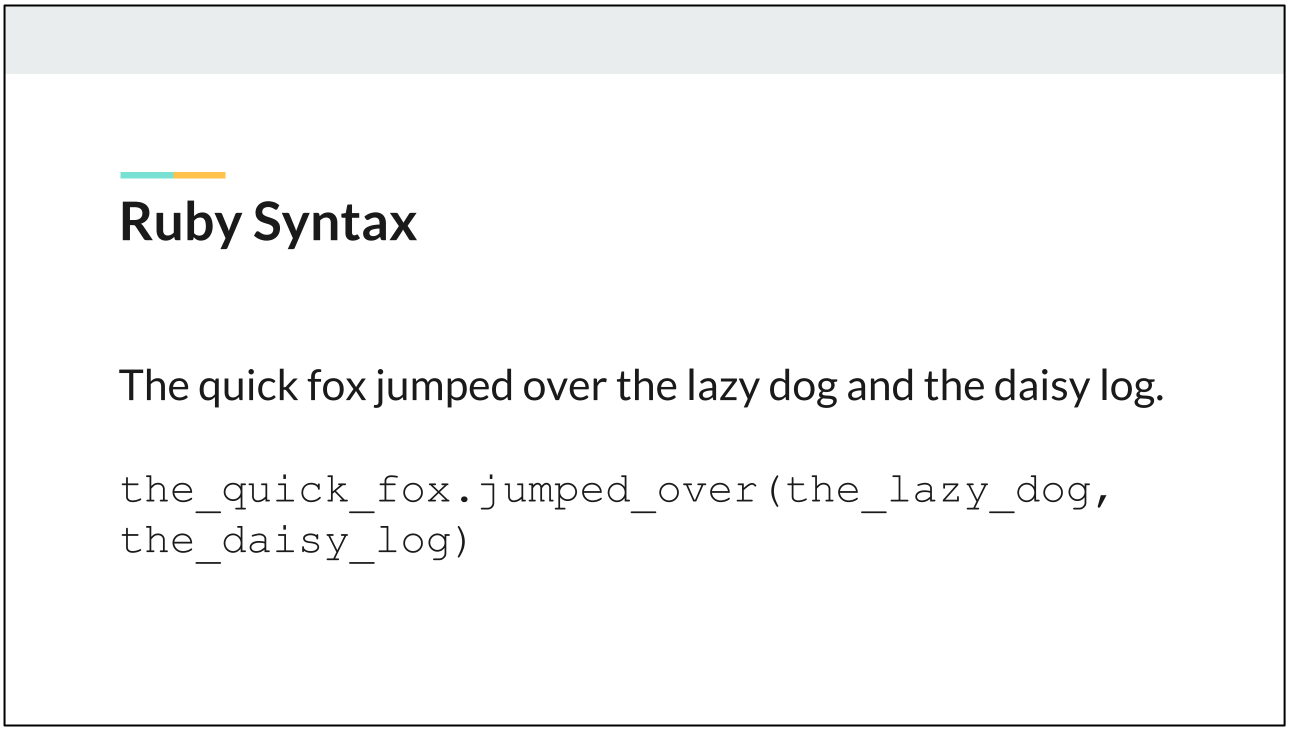
What’s *really* going on here? Let’s dive deeper.



Ruby Syntax

```
subject.verb(object, object, object)
```

This is the standard Ruby syntax. Just like an English sentence, there’s a subject, a verb, and a list of objects that the verb is being applied to.



Ruby Syntax

The quick fox jumped over the lazy dog and the daisy log.

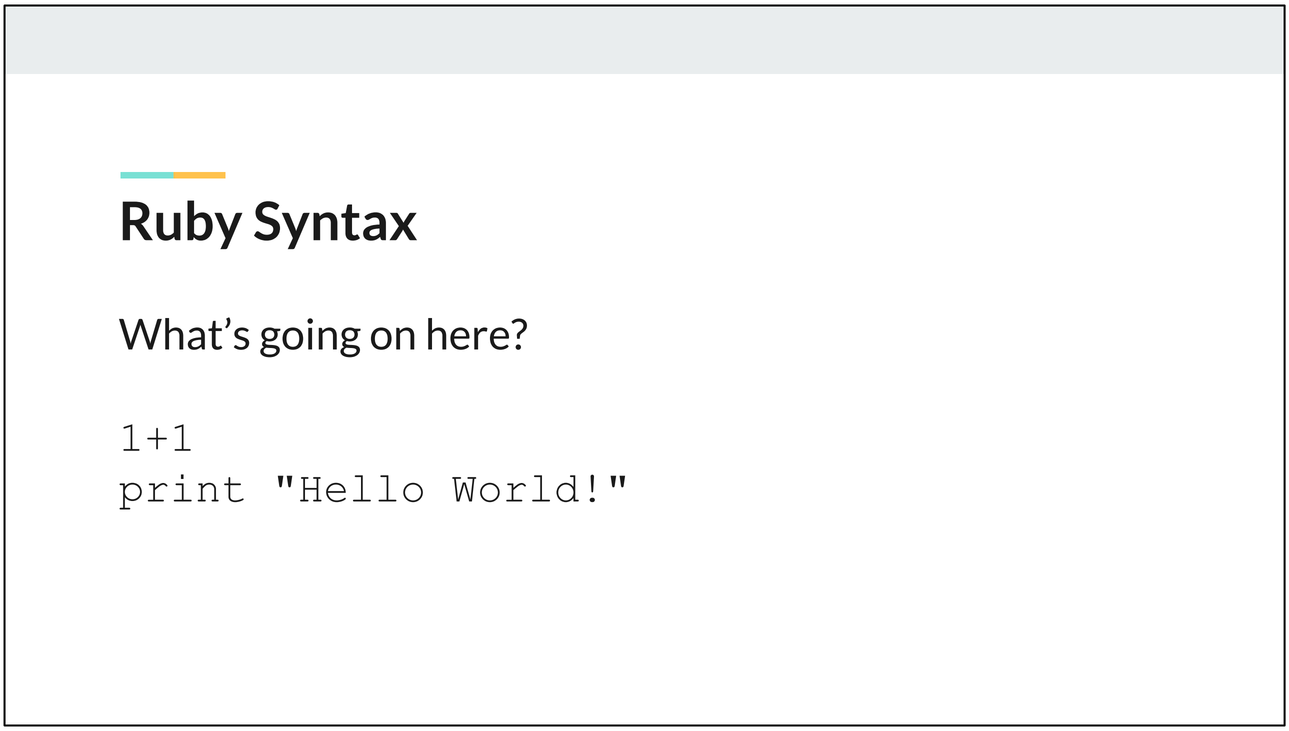
```
the_quick_fox.jumped_over(the_lazy_dog,
the_daisy_log)
```

For example, [read the sentence].

If you were to try to express this sentence in the “Ruby way”, you’d do it like this.

The subject of the sentence, “the_quick_fox”, then a period, then the verb phrase “jumped over”, then a list of the objects “the lazy dog” and “the daisy log”, separated by a comma, and wrapped in parentheses.

This line is legal Ruby syntax, but doesn’t actually have any meaning. If you try to enter it into the REPL, Ruby will complain that it doesn’t know what “the_quick_fox” is.

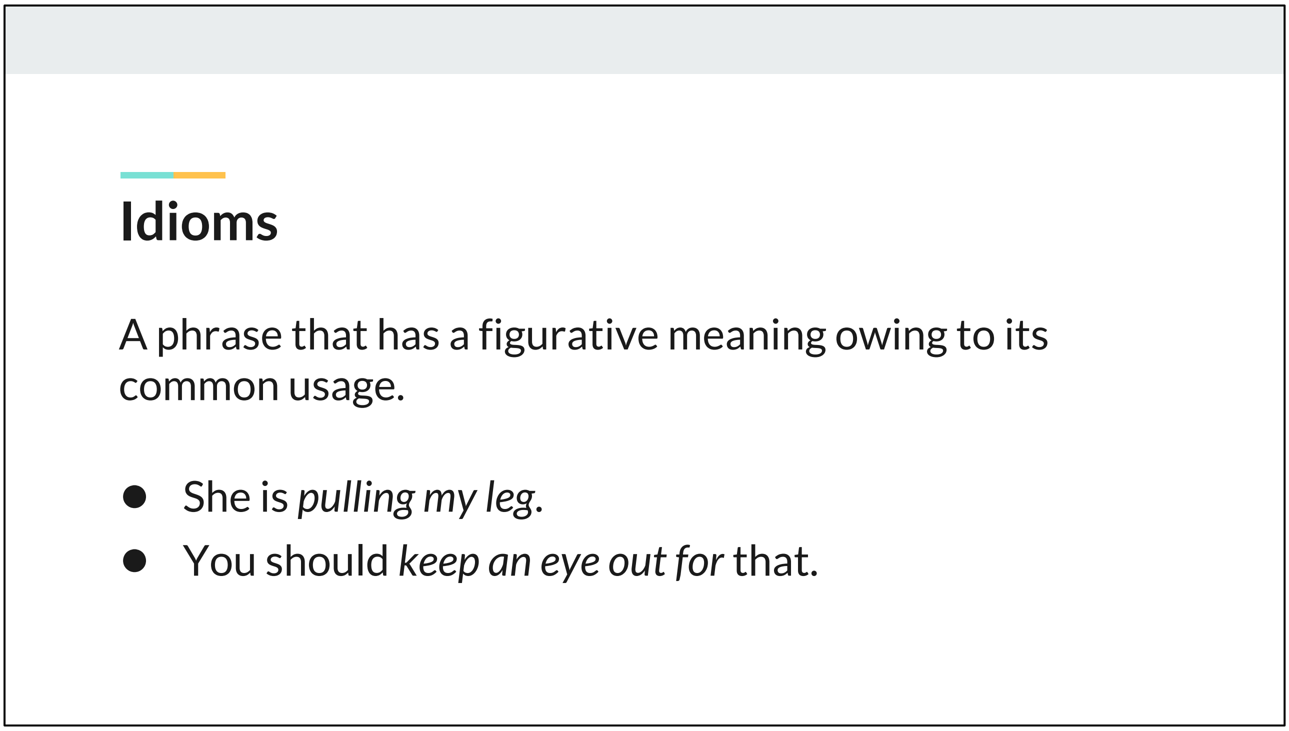


Ruby Syntax

What's going on here?

```
1+1
print "Hello World!"
```

We've only seen two different lines of valid Ruby so far - simple arithmetic and printing words - and neither of them look like the standard Ruby syntax we just introduced.


Idioms

A phrase that has a figurative meaning owing to its common usage.

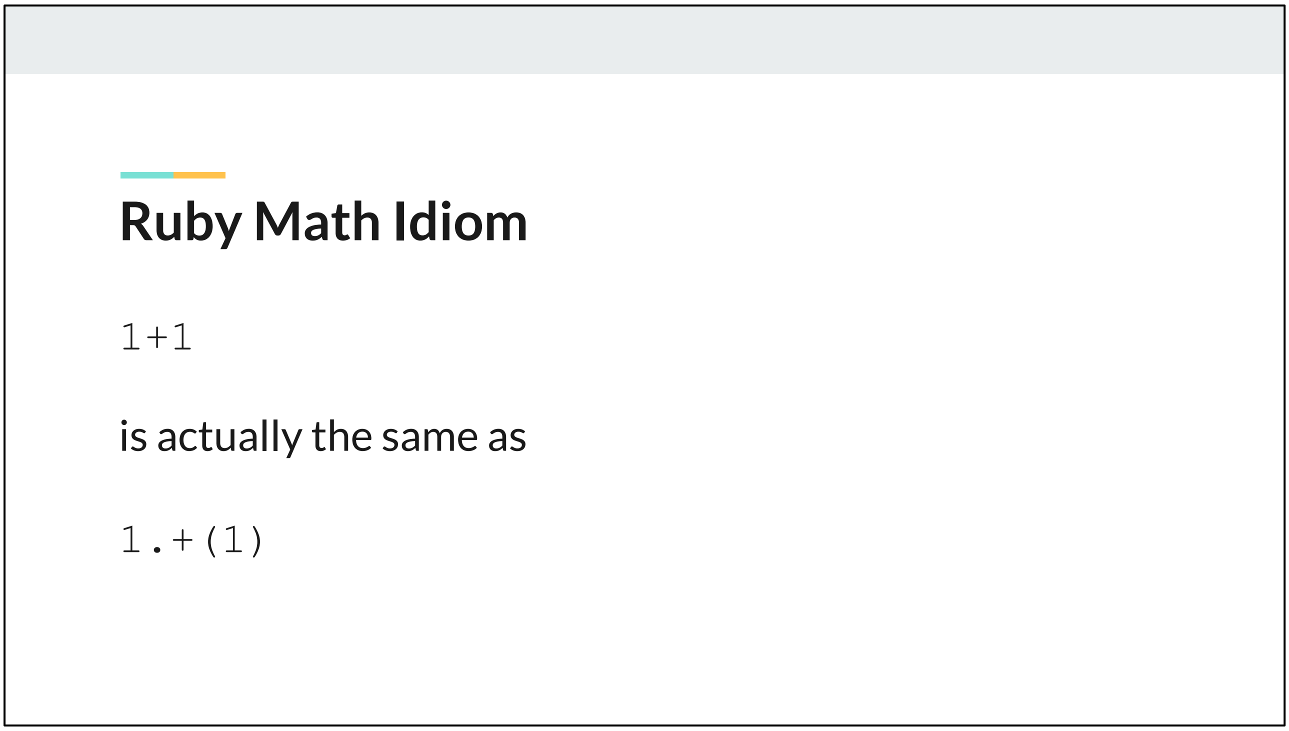
- She is *pulling my leg*.
- You should *keep an eye out for* that.

Who is familiar with the concept of an idiom in the English language? It's a combination of words with a figurative meaning owing to its common usage. When someone says that a phrase is "just an expression", they're probably referring to an idiom. Here are two examples.

If we took these sentences literally, they'd have a different meaning (or no meaning at all).

I suppose a woman could be literally pulling my leg, but that's not what I mean when I say that.

Also, how exactly do I keep my eye... out… for something?



Ruby Math Idiom

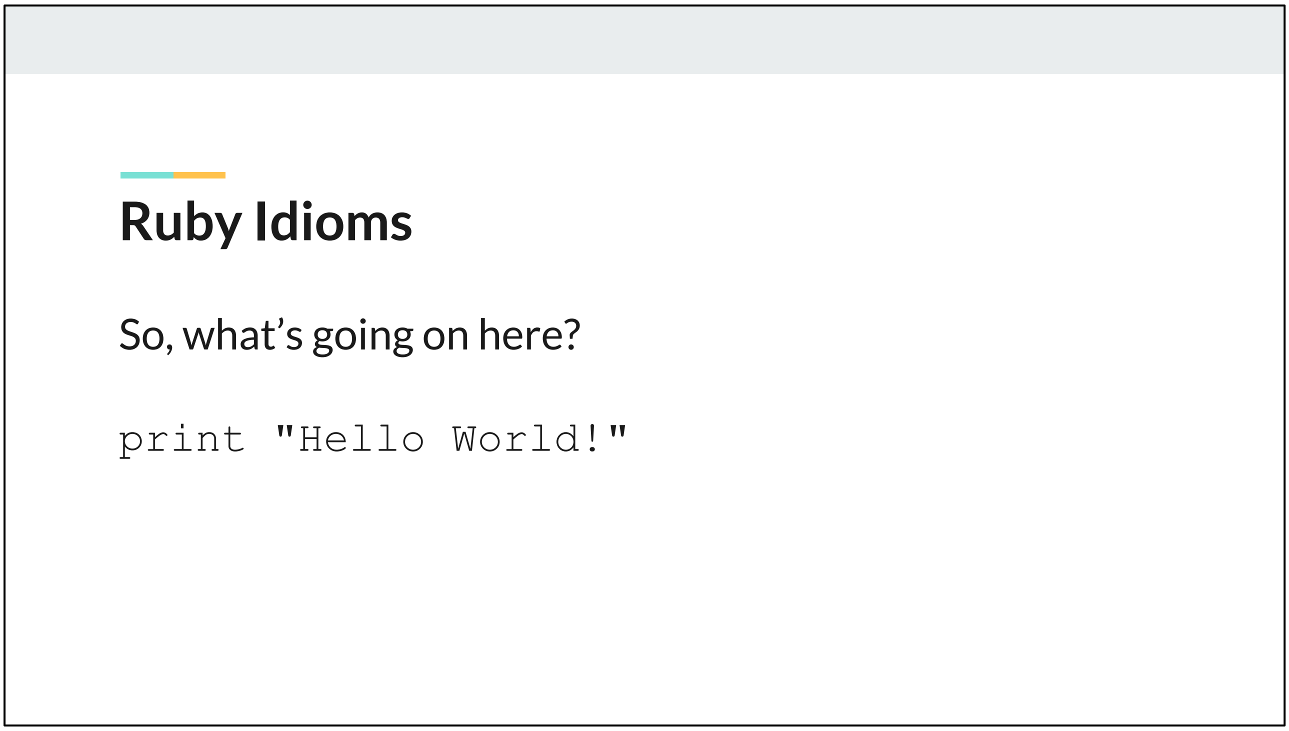
```
1+1
```

is actually the same as

```
1.+(1)
```

Ruby has idioms as well. Technically 1+1 is not valid Ruby syntax. It’s missing the period and the parentheses. But we humans are *so* used to seeing mathematical expressions written in a particular way, that the standard Ruby syntax just looks... weird. So Ruby, to improve human readability (and happiness), allows you express the same concept, in this case 1+1, in different ways.

This actually isn’t very special. Most programming languages allow this as well for simple math.

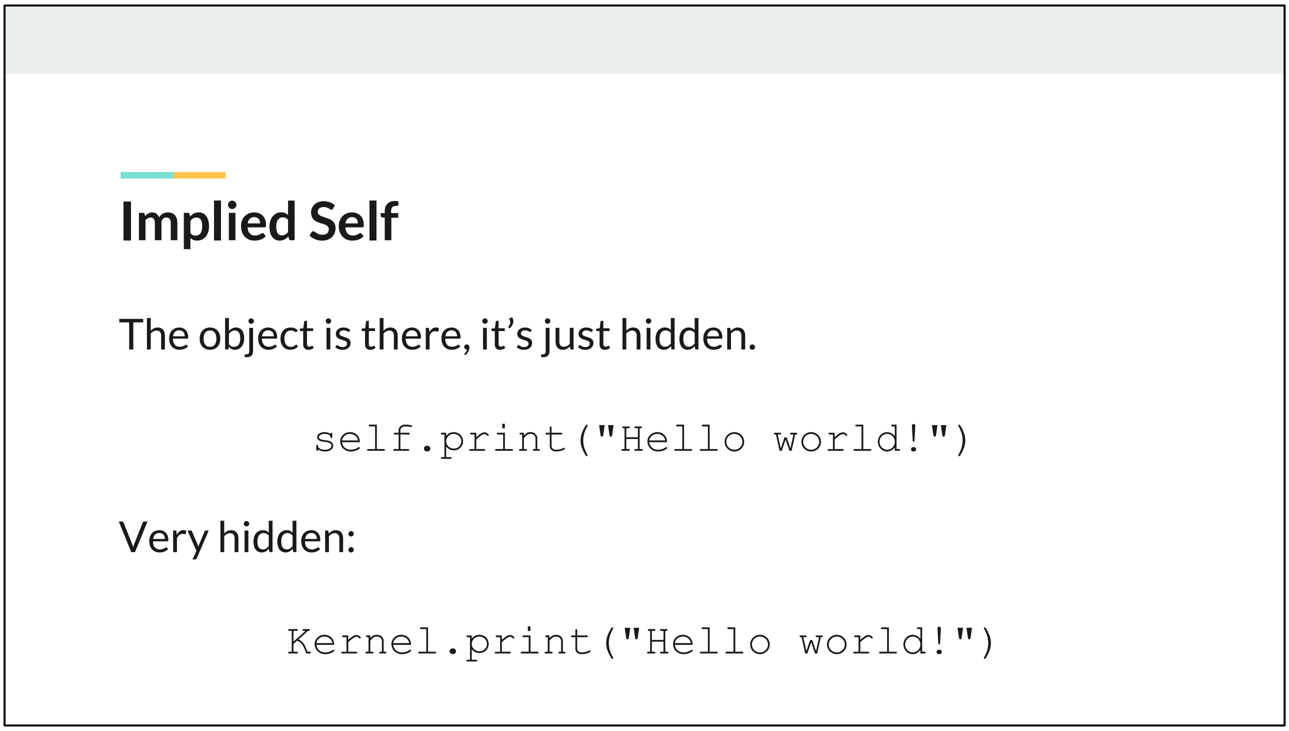


Ruby Idioms

So, what’s going on here?

```
print "Hello World!"
```

Where’s the subject of this Ruby sentence? If print is the verb, *what* is printing?



Implied Self

The object is there, it’s just hidden.

```
self.print("Hello world!")
```

Very hidden:

```
Kernel.print("Hello world!")
```

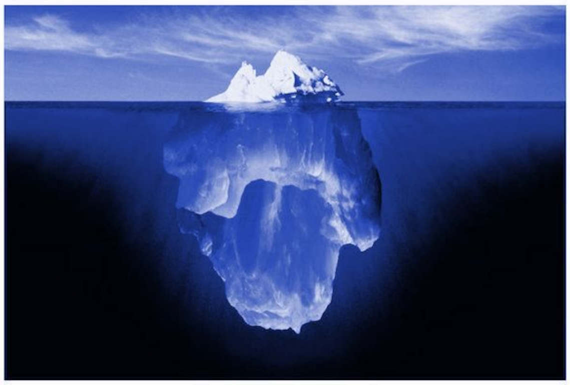
It turns out there is a subject in this sentence, it’s just hidden. When you leave the subject blank, Ruby inserts one for you. In Ruby, the subject is *usually* implied to be the word “self”.

Unfortunately, *this* is not one of those cases. For `print` (and a few other verbs we’ll learn about later), the subject of the sentence is `Kernel`. `Kernel` is a special library that adds some features to the Ruby language for interacting with the “kernel”, which is part of the computer’s operating system.

Speaking of libraries...

Ruby Standard Library

`methods`



Ruby, like most other programming languages, comes with a bunch of stuff for free. These freebies are stored in “libraries” (for lack of a better term right now).

Think of “libraries” as a collection of new “verbs” that can be added to the the Ruby language. In Ruby, they’re actually called “methods”, so we’re going to start using that term from now on. (Because the analogy with English can’t go on forever.)

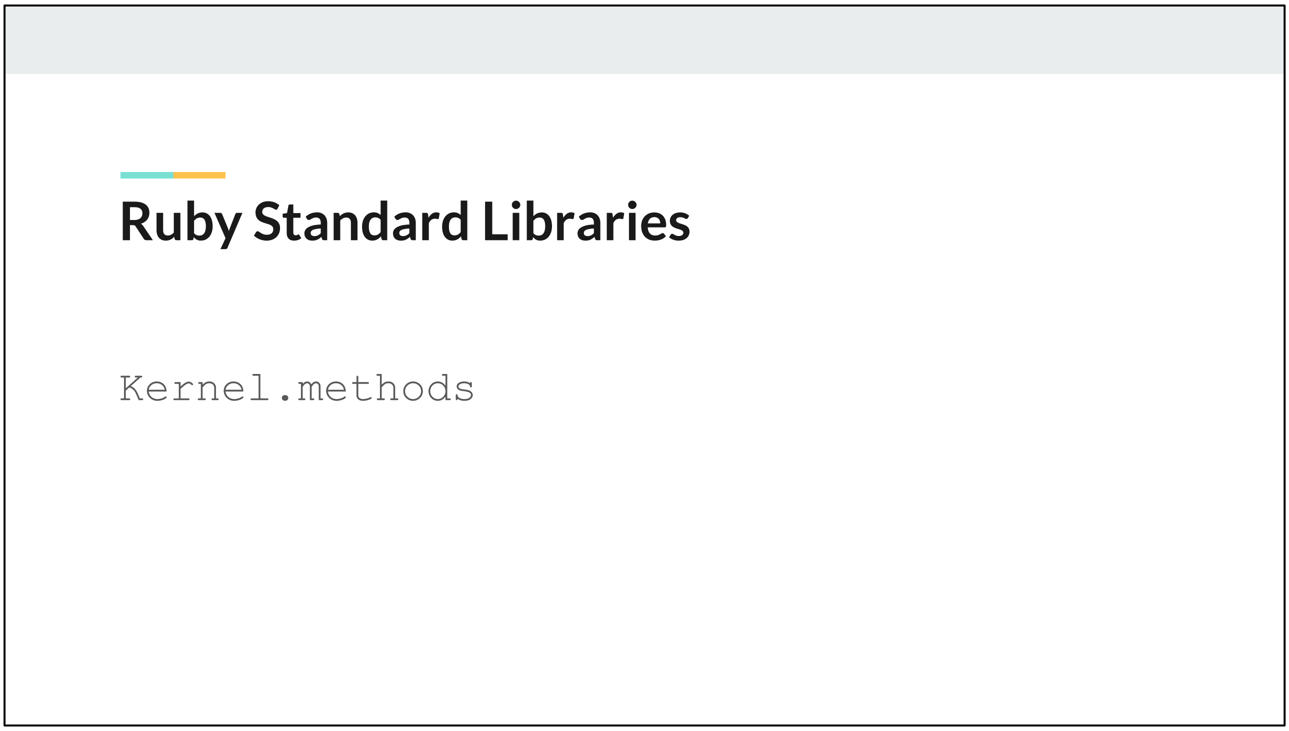
We’ve seen some methods for doing basic math and printing stuff to the screen, but that’s just the tip of the iceberg. Check out what happens when you type the word “methods” into the REPL. That’s just *some* of the rest of the iceberg.

Thanks to the Ruby Standard Library, you never have to write in machine code. Everything you need to be “Turing complete” is available in the standard library.

You may have also noticed that the word “methods”, which *does* something in the Ruby language, isn’t an English verb. You can’t “methods” something. Not all Ruby methods are verbs. That’s why now is a good time to start using the correct terminology.

So `methods` is a Ruby method that returns a list of available methods.

Got it?

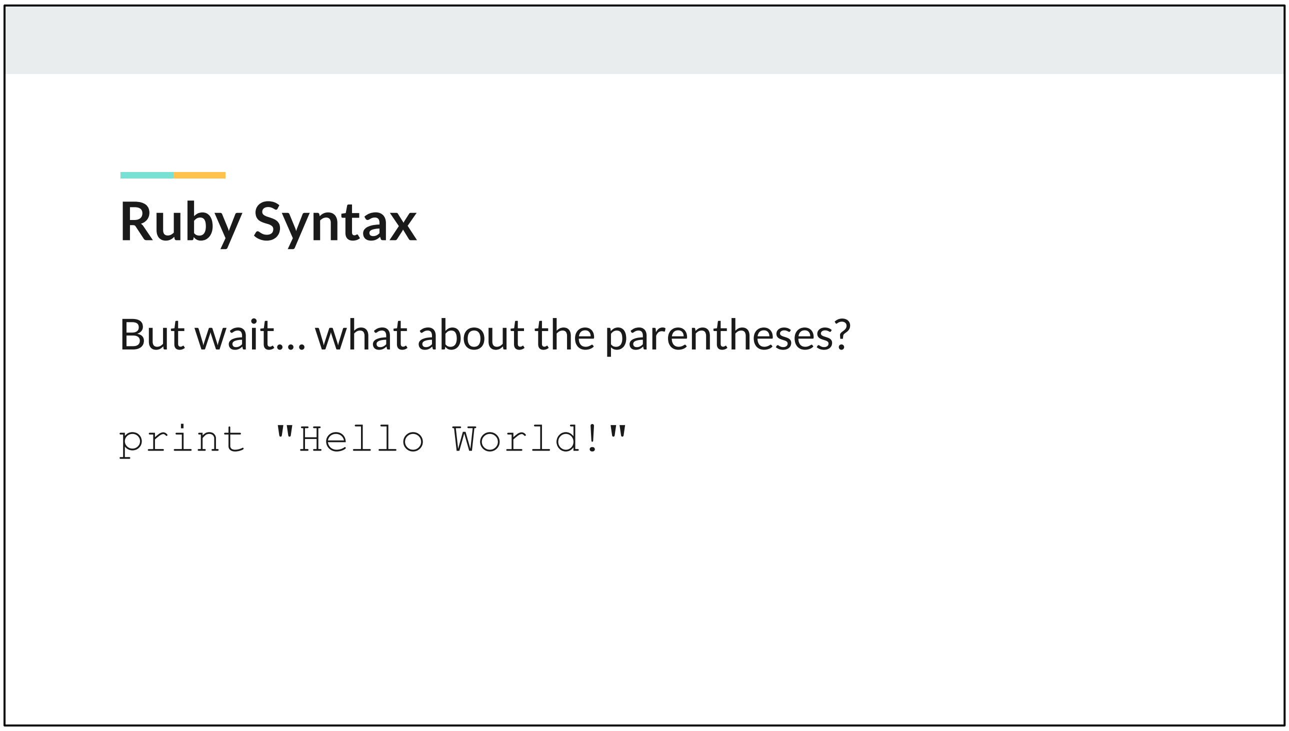


Ruby Standard Libraries

```
Kernel.methods
```

You may have noticed that our friend the `print` method wasn't in the list of methods we got when we typed "methods". That's because `print` was added to Ruby by a library named "Kernel". To see the methods the `Kernel` library adds to Ruby, use this line of code. That's where `print` is hiding.

So the Ruby Standard Library is actually a collection of libraries.

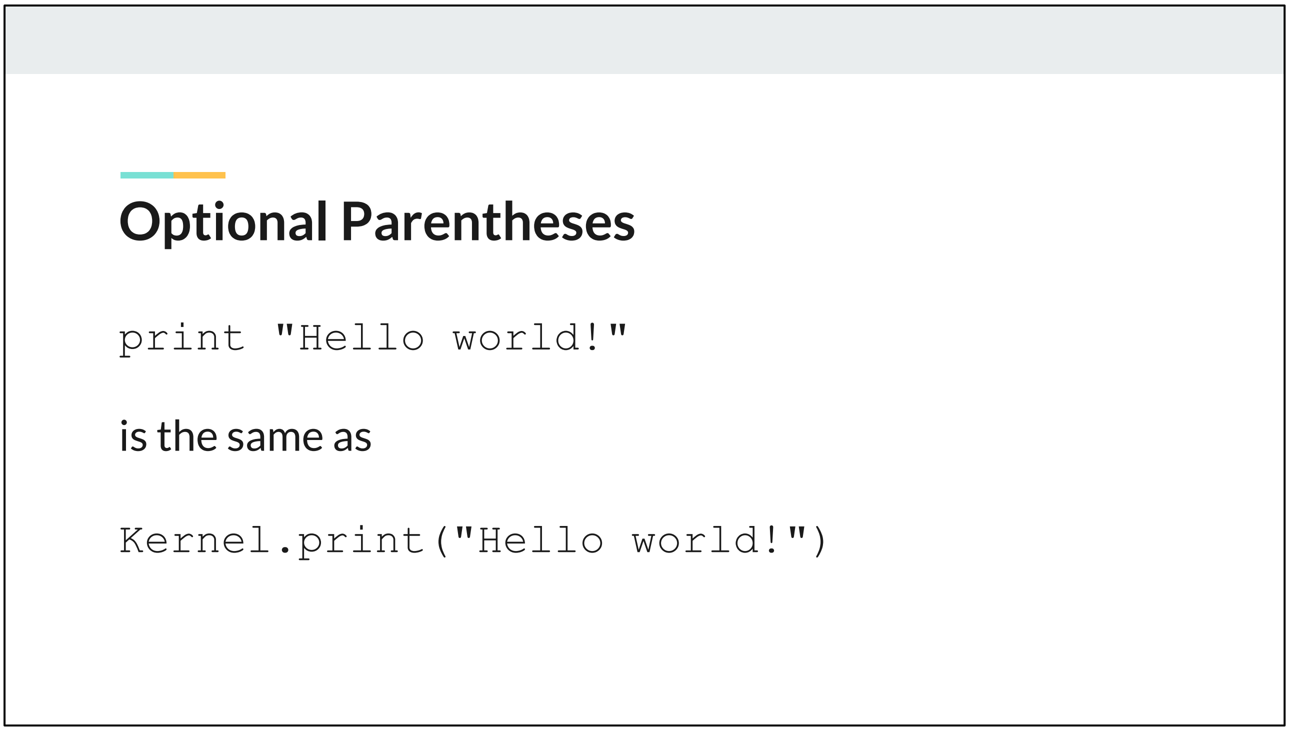


Ruby Syntax

But wait… what about the parentheses?

```
print "Hello World!"
```

It looks like there one more idiom I forgot to mention. Our “Hello world” program is missing parentheses. What’s up with that?



Optional Parentheses

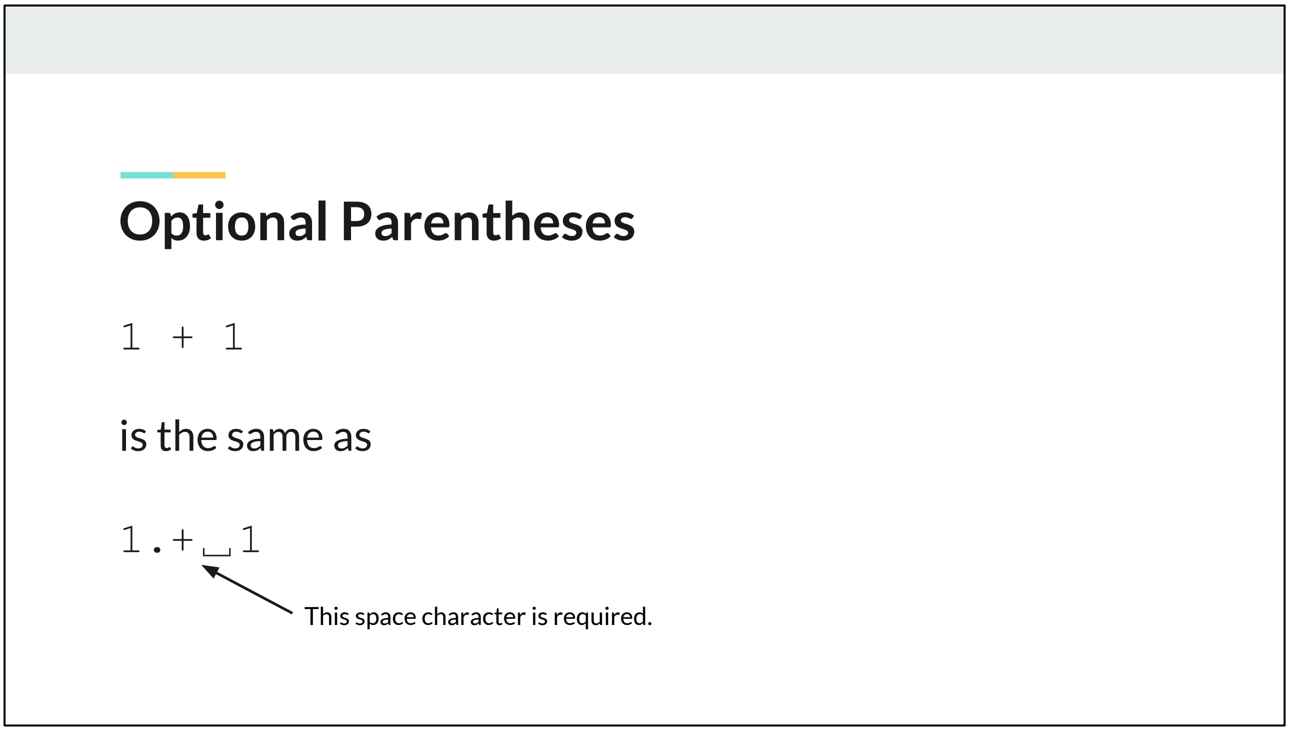
```
print "Hello world!"
```

is the same as

```
Kernel.print("Hello world!")
```

The last idiom I’m going to talk about today is that, in Ruby, parentheses are optional. In English, we use a space to separate words. So Ruby allows that as well. Because it’s all about developer happiness.

So the line at the top is easier to read (for most people) than the standard syntax equivalent on the bottom.



Optional Parentheses

```
1 + 1
```

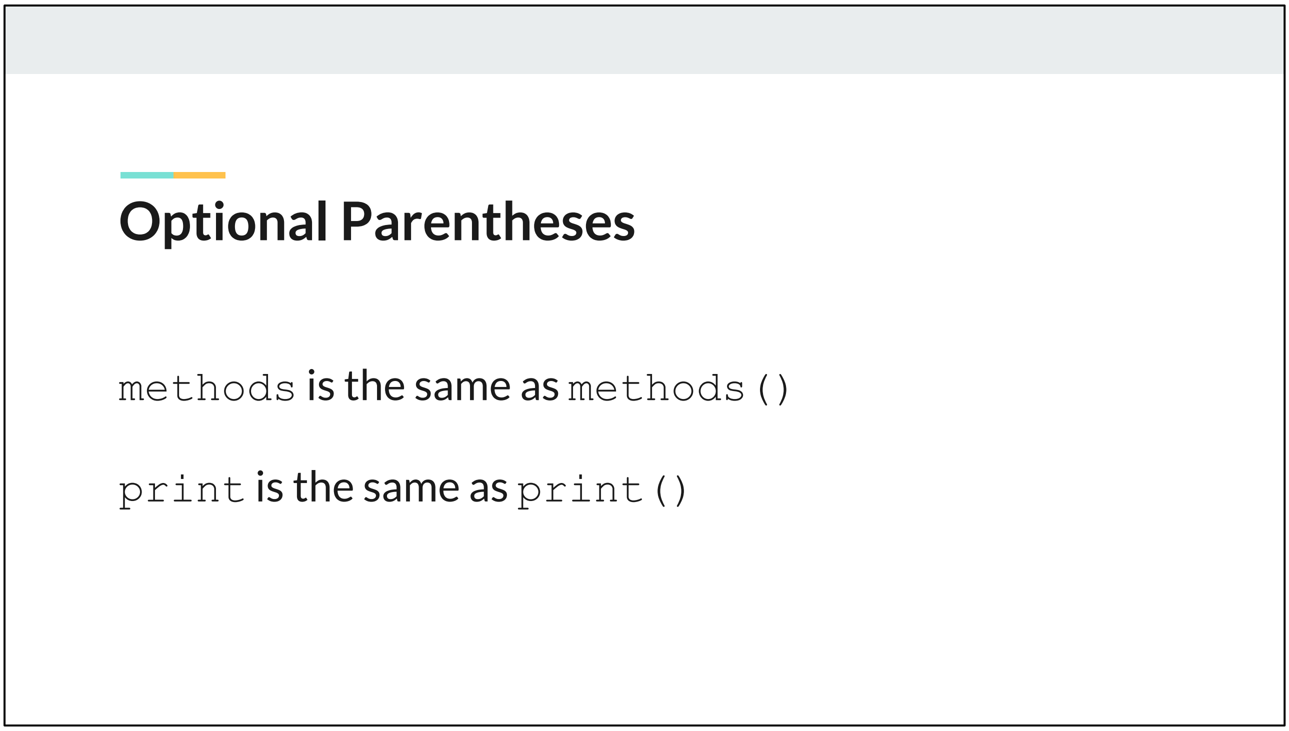
is the same as

```
1.+␣1
```

This space character is required.

This idiom works with basic math as well, but you shouldn’t use it. Don’t mix idioms that make code read better as English with idioms that make code read better as math.

In this case, the English-y syntax on the bottom is both weird *and* misleading. It looks like you’re using a Float, but you’re not. You’re actually using the + method without parentheses. A space is required.



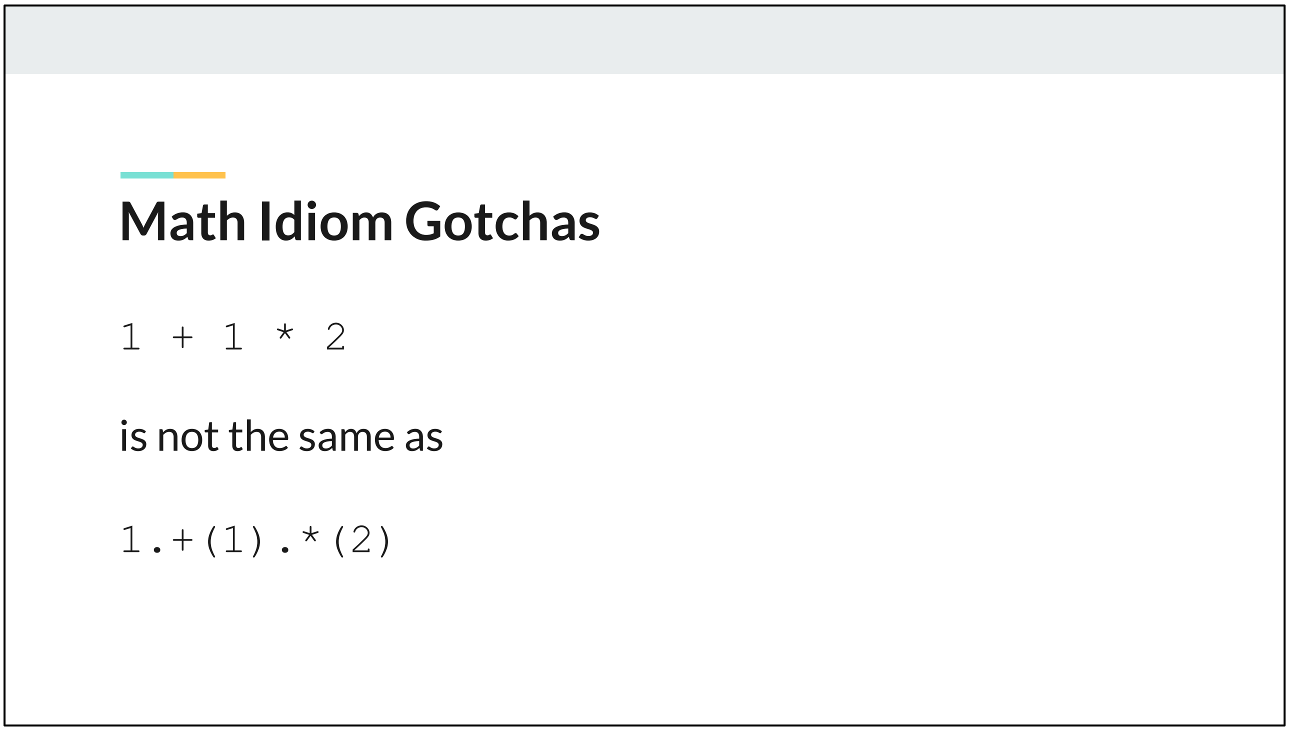
Optional Parentheses

`methods` is the same as `methods()`

`print` is the same as `print()`

As a result of the “optional parentheses” rule, if you leave off the parentheses *and* you don’t have any objects to list, you can just type the method name and hit enter.

We’ve already seen an example of this with the `methods` method, which outputs a list of available methods. The `print` method can also be used without “objects” and parentheses. It will just print nothing.



Math Idiom Gotchas

```
1 + 1 * 2
```

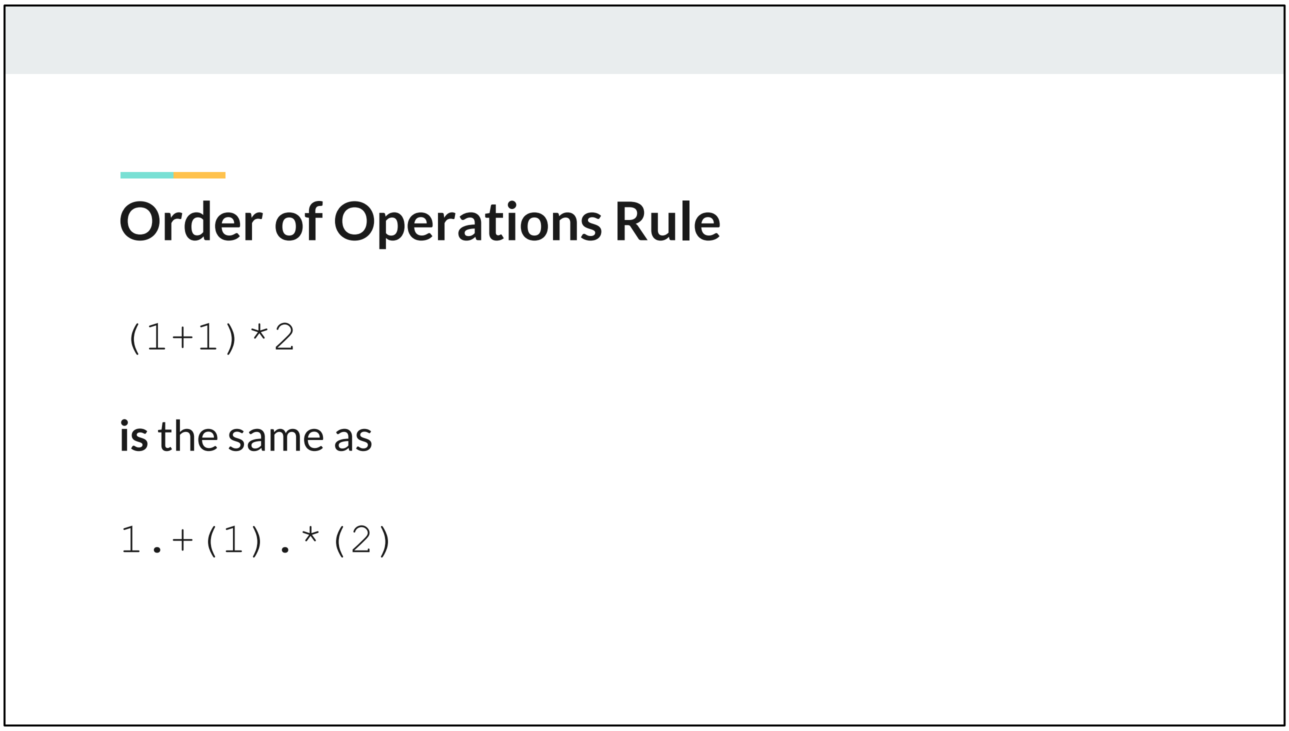
is not the same as

```
1.+(1).*(2)
```

You have to be careful with idiomatic Ruby expressions. For example, these two lines are not equal. The first line is 3. The second is 4. Does anyone know why?

The mathematical idiom that Ruby allows you to use, the one we all learned in school, also happens to follow the “order of operations” rule, which states that multiplication and division always come before addition and subtraction, regardless of the order in the expression.

“Order of operations” doesn’t apply in the standard Ruby syntax. The standard syntax is simply evaluated from left to right.



Order of Operations Rule

```
(1+1)*2
```

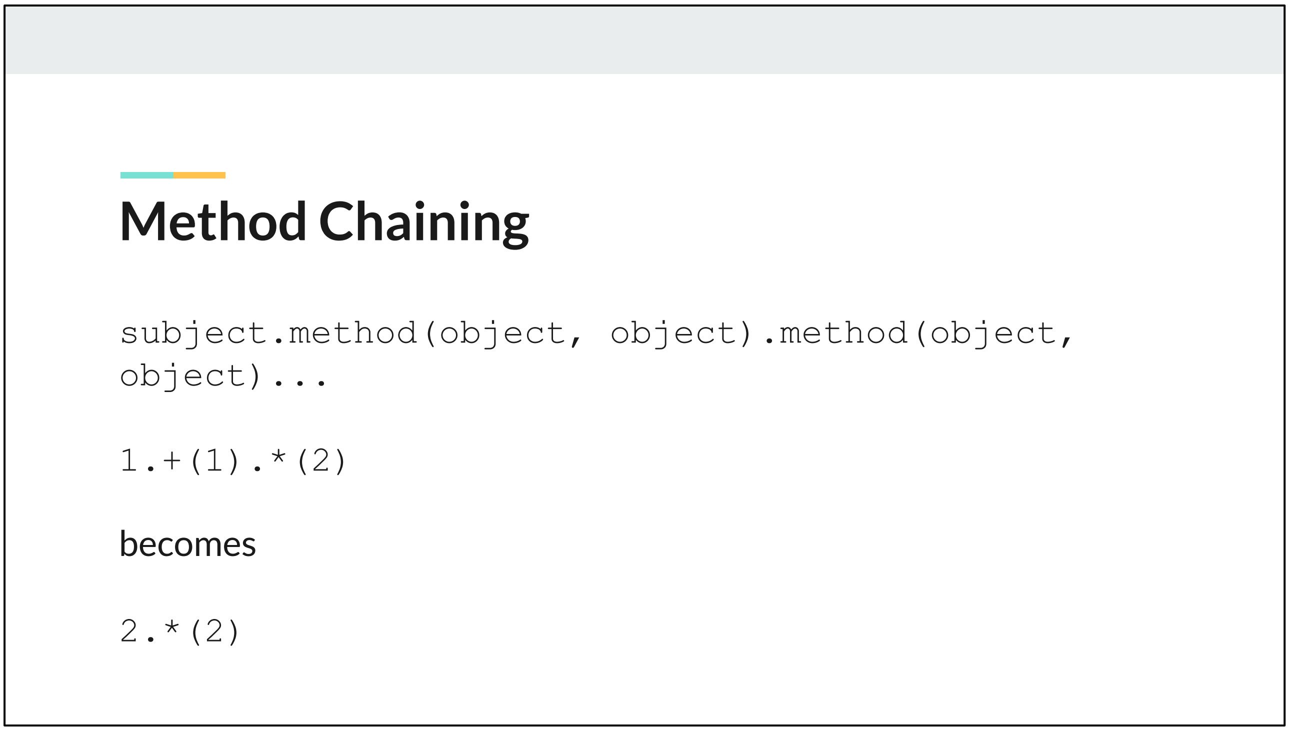
is the same as

```
1.+(1).*(2)
```

So whenever you combine multiple mathematical operations idiomatically, if you can’t remember the “order of operations” rule, then add parentheses around the expression you want to evaluate first. Be explicit about the ordering with a generous use of parentheses.

In fact, since another dev reading your code might not know the “order of operations” rule (yes, it can happen), it’s good practice to add the parentheses even if you *do* know the order of operations rule. Write code that plays nicely with other people. That’s going to be an important lesson for later.

By the way, you may have noticed that I just snuck in a new feature of Ruby syntax here that I haven’t talked about yet.



Method Chaining

```
subject.method(object, object).method(object,
object)...

1.+(1).*(2)
```

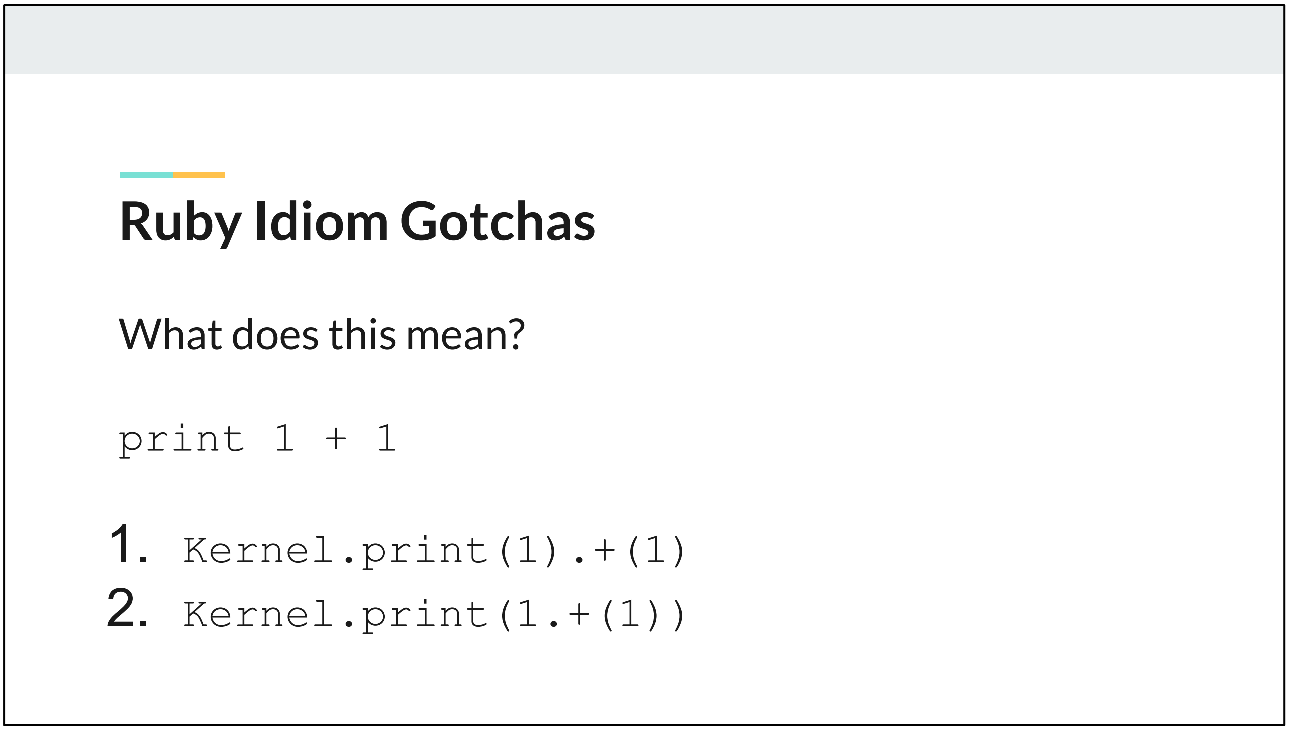
becomes

```
2.*(2)
```

You can smush together two different Ruby expressions into a single line. Just add a period to the end, then another method, then another list of objects.

This is called “method chaining”. The result of the first expression becomes the subject of the next expression. So, in the example at the bottom here, 1+1 is 2, which then becomes the subject for the next expression 2*2.

You can chain as many expressions together as you’d like, but try to keep it short. If the resulting line of code wraps over two lines (like the example at the top of this slide), then break the chain into multiple lines.



Ruby Idiom Gotchas

What does this mean?

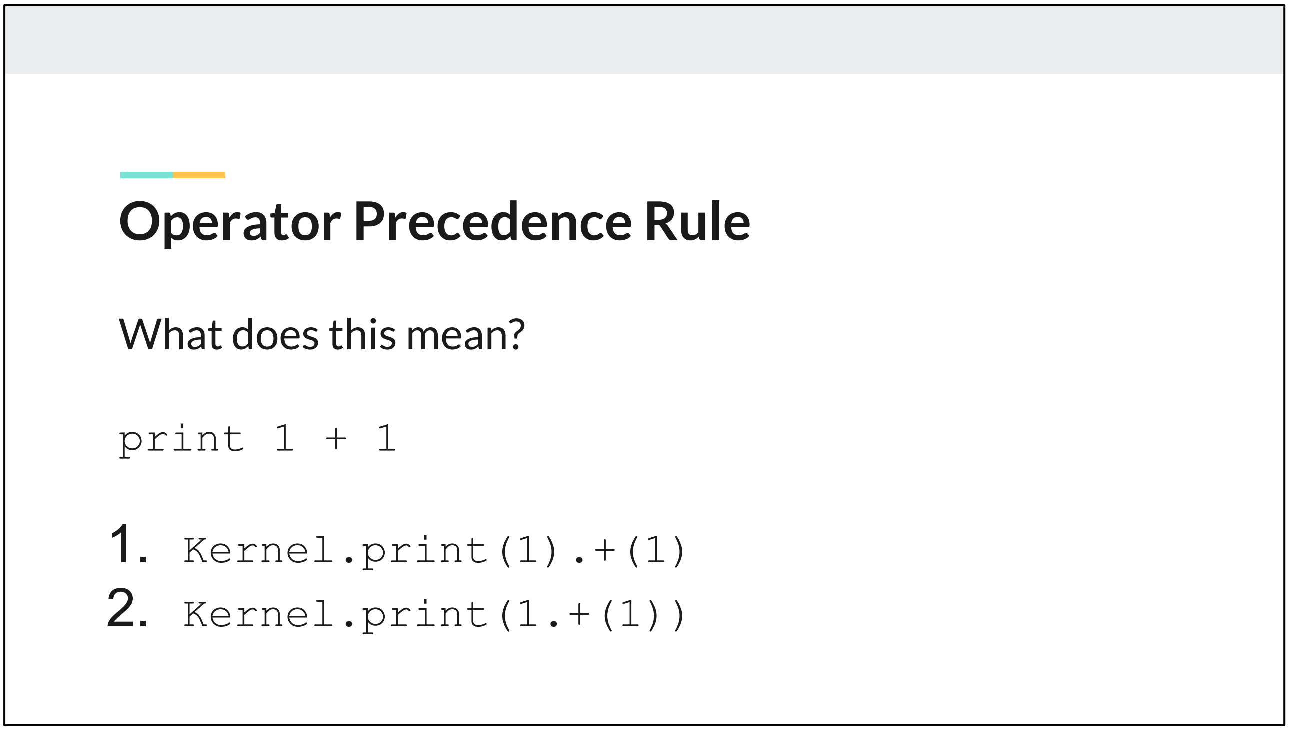
```
print 1 + 1
```

1. `Kernel.print(1).+(1)`
2. `Kernel.print(1.+(1))`

Method chaining actually introduces another, much more sinister version of the “order of operations” gotcha we just talked about.

Which of the following expressions, 1 or 2, is equivalent to the idiomatic expression “print 1 + 1”?

Who thinks 1? Who thinks 2?



Operator Precedence Rule

What does this mean?

```
print 1 + 1
```

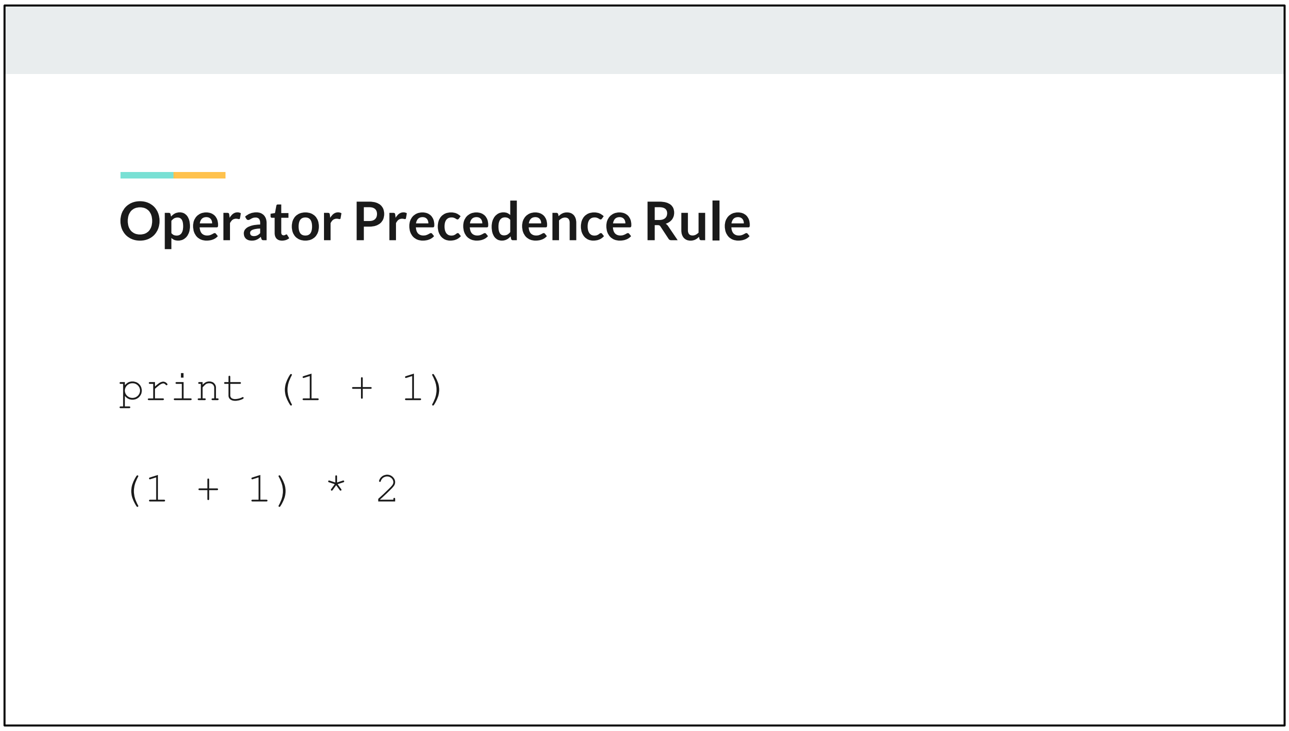
1. `Kernel.print(1).+(1)`
2. `Kernel.print(1.+(1))`

The answer is 2.

print 1 + 1 doesn't cause an error. It prints 2.
#1 is an error
#2 does the same thing as the expression. It just prints 2.

Leaving mathematics and entering the realm of computer science, the "order of operations" rule becomes the "operator precedence" rule - which is like the "order of operations" rule on steroids. In Ruby (and any other language that supports method chaining), there's not only a preference for multiplication over addition, but there's also a preference for one method over another.

So in this case, Ruby prefers to do math *before* executing a regular method. The '`+`' method is evaluated *before* the `print` method.

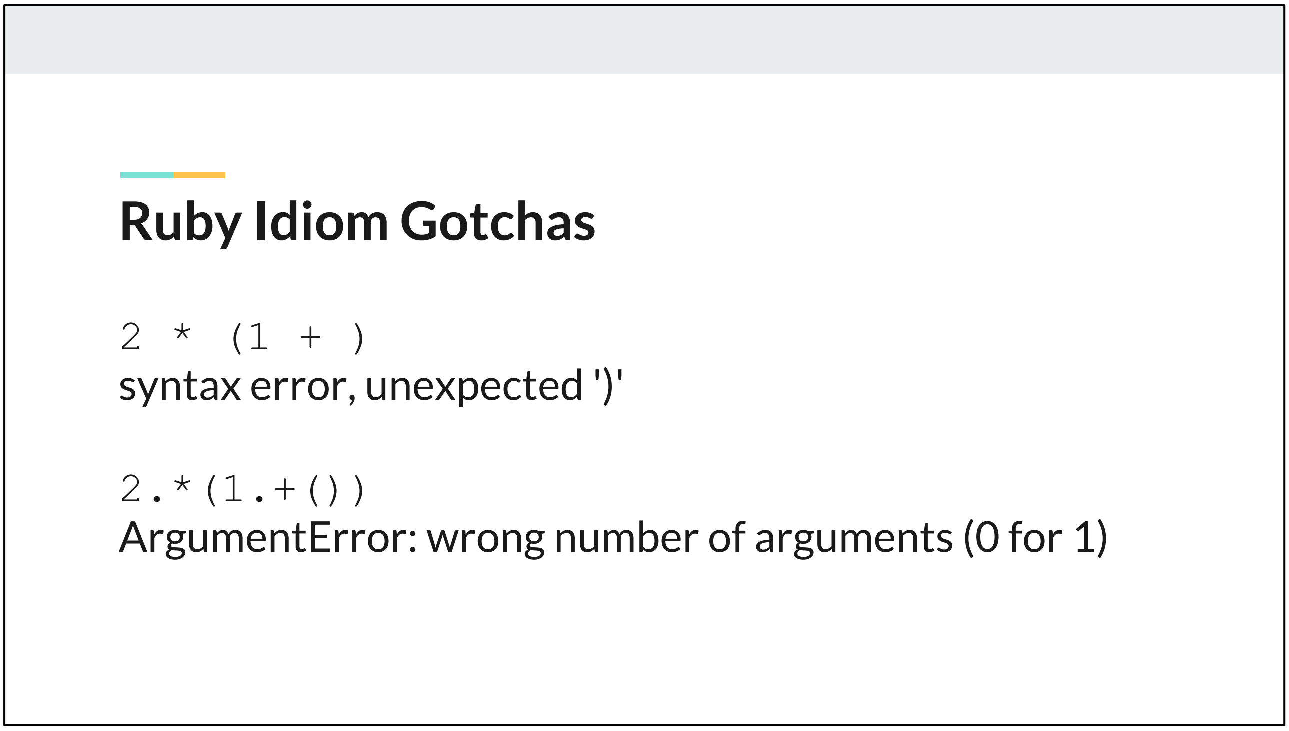


Operator Precedence Rule

```
print (1 + 1)

(1 + 1) * 2
```

“Operator precedence”, unfortunately, is not always obvious, even to experienced programmers. I still mess it up sometimes.

As a result, whenever you’re method chaining, it’s *always* going to be best to use parentheses to make your assumptions explicit. Don’t leave *any* room for misunderstanding (either by yourself or others).



Ruby Idiom Gotchas

```
2 * (1 + )
```
syntax error, unexpected ')'

```
2.*(1.+())
```
ArgumentError: wrong number of arguments (0 for 1)

The final warning I want to give you about idiomatic Ruby: error messages.

These two lines of code produce different error messages.

The first complains about invalid syntax.

The second is better. The error tells you *why* it’s broken. It says that you have an error because you’re missing an “argument” (which is what we’ve been calling the “objects”, the stuff in parentheses). There should be 1 “argument” listed, but you’ve provided 0 (none).

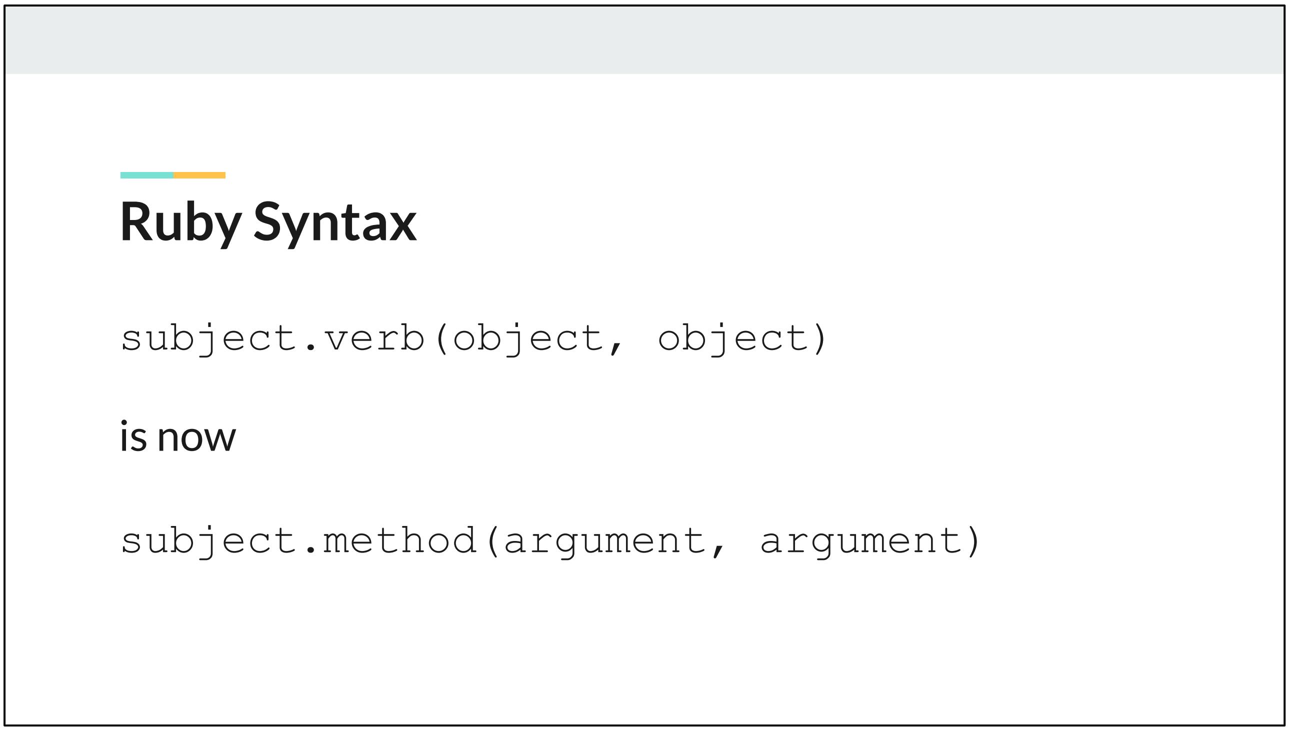
Ruby Idioms

Great fun, until they're not.

The moral of the story is that idiomatic Ruby expressions are fun, as long as they're simple.

When they get complicated (like with method chaining) and especially when they break (by displaying errors), it's time to switch to the standard Ruby syntax. That way both you *and* Ruby don't have to work so hard trying to figure out what you mean.

Sometimes Ruby idioms are more readable. Sometimes they aren't. And sometimes they just break stuff.



Ruby Syntax

```
subject.verb(object, object)
```

is now

```
subject.method(argument, argument)
```

Note at this point that we’ve made some replacements in our standard syntax terminology. What we used to call the “verb of the Ruby sentence” is now the “method of the Ruby expression”. What we used the call the “objects of the verb” are now the “arguments of the method”.

Instead of saying “a verb is being applied to objects”, we’re going to say that “a method is being applied to arguments” (or, in reverse, that “arguments are being *passed* to a method”).

We need to let the English fall away, leaving only the Ruby behind.

Next on the chopping block: the “subject”. What is a Ruby “subject” really? You’ll soon find out!