Ruby Data Types

Prerequisites

- Ruby Syntax
- A Ruby REPL (irb)

Get your IRB prompt ready. We're going to write some Ruby today.

What you'll learn

~~All~~ Most of the Ruby primitive types

- Integer & Float
- String
- Boolean
- Nil
- Array
- Hash & Symbol
- Object

Ruby Syntax

```
subject.method(argument, argument)

subject.methods
```

Recall the Ruby syntax from a previous lecture. Like an English sentence, it has a "subject". Every Ruby "subject" has a list of allowed methods. We can inspect this list by using the `methods` method.

Each Ruby method can be applied to a list of arguments. The list may contain 0 or more arguments.

What kind of things can be the "subject" of a Ruby sentence? We've seen a some examples already.

Integer

```
1+1

1.methods
1.class
```

Recall the simple arithmetic we've seen before. The subject of this Ruby sentence is 1.

What is 1, exactly? Let's introspect it.

Recall that we've already seen one way to introspect, or find out more about something, in Ruby. If we type `1.methods` into IRB, we'll see a list of the methods that 1 supports. We should see all of our basic arithmetic methods there: +, -, *, and /.

So 1 can be described by what it does.

If you look hard, you may be able to find a method named `class`. That's another introspective method. That tell us what Ruby thinks 1 *is*. 1 *is* an `Integer`.

1 is just an example. You could use whatever whole number you'd like: -1, 0, 100, 1million, whatever you'd like. They all *do* the same things (meaning they have the same list of methods) and they all *are* the same things (`Integer`s).

Integer

```
-1.abs
1.even? & 1.odd?
1.zero?
1.next
```

Here are some examples of methods you're likely to use.

The `abs` method returns the absolute value, which is just the positive version of the number. So -1 becomes 1 and 1 just stays as one.

The "`even?`" and "`odd?`" methods let you know whether or not the number is even or odd. They'll return a `true` or `false`. By convention, methods that return `true` or `false` end with a question mark. That's not a requirement, but it's a way Ruby developers try to help other coders quickly understand what their code does after, for example, inspecting the output of the `methods` method.

"`zero?`" is one of those question mark methods. It returns `true` if the number is 0. (There is no `one?` method.)

`next` gives you the next number, which is the same as adding 1. It's useful for counting.

Integer

```
0 < 1
1 > 0
1 <= 1
1 >= 1
1 == 1
1 <=> 1
```

Numbers can also be compared to each other. You can check if an `Integer` is less than, greater than, less-than-or-equal-to, greater-than-or-equal-to, and equal-to any other number. They all look like and work the way you expect, except **equal-to is 2 equals signs (==), not one.**

The bottom method is special. It's not something you would normally see in a math class. It's actually a combination of all of the above comparisons. It's called a spaceship. If the number on the left is less-than the number on the right, it returns -1. If the number on the left is greater than the number on the right, it returns 1. If both numbers are equal, it returns 0.

Spaceships are useful because there are technically three answers to a comparison of any two numbers. Either the numbers are equal, one is greater, or the other is greater. So `true` and `false` aren't enough to represent all the possible results. You need three values: -1, 0, and 1.

Integer

```
10.modulo 3

10 % 3
```

These two lines of code do the same thing. They both take the modulo of two numbers.

In this case, that means they return the remainder when 10 is divided by 3. So 3 goes into 10 3 times with a remainder of 1. So the answer is 1.

Float

```
1.0.class

1.5.round
1.5.truncate

1.5.round.class
```

Numbers with decimal points, even if the digits after the decimal point are all zero, are treated as `Float`s. They support most of the same methods that `Integer`s do, like comparisons (greater-than, less-than, spaceship). They don't support some methods, like "`even?`" and "`odd?`".

The only thing I usually do with floats, other than compare them, is `round` or `truncate` them. Rounding a `Float` outputs the closest `Integer` value. For example, everything up to, but not including, 1.5 rounds to 1. 1.5 and up rounds to 2.

Truncating just cuts off everything after the decimal. So 1-point-anything truncates to 1.

Note that rounding and truncating result in an `Integer`, which you can confirm by chaining the `class` method onto the end of the expression. So rounding and truncating can both be considered ways to *convert* a `Float` into an `Integer`.

Float

```
(1+1).class
(1.0 + 1).class
```

Note that any time you do math combining `Float`s and `Integer`s, the answer is always going to be a `Float`.

Once you start using any imprecise numbers, you'll always get back imprecise numbers.

String

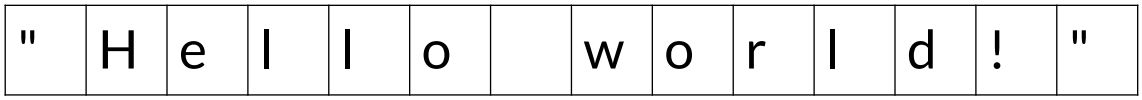
```
"Hello world!"

"Hello world!".class
"Hello world!".methods
```

Here's our friend "Hello world!" again. Note that, in IRB, you can just type `"Hello world!"` to print "Hello world!" to the screen. You don't even need to use `print`. But we'll need `print` later when we break out of IRB and do some real coding.

Ruby tells us "Hello world!" is a `String`. What's that?

String




```
"a".class
"".class
```

"String" is a strange name for what is essentially a collection of letters.

A "String" is a sequence of characters with quotes around it. It's called a String because the characters are strung together, like clothes on a clothesline. Imagine the double-quotes are clothespins.

So a `String`, when presented this way, looks like a list of things. It's a list of characters. And it behaves like a list, which we'll see later.

But what's in the list? More `String`s. A `String` is made of `String`s. The single character "a" is still a `String`. If you cut a real piece of string into smaller pieces, all those pieces are still strings.

Even the empty string is a `String`.

I'm displaying this `String` using a series of blocks for a reason. It should look familiar. It's Turing tape.

Ruby reads a `String` the way a computer would read Turing tape. Going from left to right, Ruby sees the quotes and transitions into the state "I'm going to begin reading the characters in a string". Then it proceeds along the tape: "H" is a character in this String, "e" is a character in this String, and so on, until it reaches another quote and enters the state "I'm done reading all the characters in this String".

This may seem like an unnecessarily complicated way of thinking about Strings, but thinking like this will be useful later.

String

```
"a" <=> "b"
"hi".capitalize, "hi".upcase,
"Hi".downcase
"".empty?
"hello".end_with? "lo"
"hello".include? "hell"
```

Strings have lots of interesting methods. And displaying and manipulating Strings is a big part of building apps for the web. So let's review some methods.

Strings can be compared. They're compared alphabetically, so "a" is less than "b".

You can change the capitalization of Strings all sorts of ways. The "`capitalize`" method just uppercases the first letter. `upcase` will uppercase all the letters. `downcase` will do the opposite, lowercasing all the letters.

You can test whether or not a `String` is empty, whether or not it ends with another `String`, and whether or not it includes another `String`. When a `String` appears inside another `String`, it's called a substring. So "hell" is a substring of "hello".

String

```
"hello".gsub "l", "y"
"hello".length
" hello ".strip
" world".prepend "hello"
"hello".concat "world"
"hello".reverse
```

You can find-and-replace one or more characters with `gsub`, which stands for "global substitution". Global, in this example, means `gsub` will replace both L's in `"hello"`, not just one. The "`sub`" method will only replace the first occurrence. That's not as common.

You can get the `length` of a string, which is the number of characters in it.

You can remove all spaces from the beginning and end of a string with "`strip`". There's also an `lstrip` and `rstrip` if you just want to remove the spaces from the left-side (beginning or leading side) or right-side (end or trailing side) of a `String`.

You can add a `String` to the beginning of a `String` with `prepend`. You can add a `String` to the end of a `String` with `concat`.

You can also `reverse` the characters in a String. That's fun. Not very useful, but fun.

Converting Strings

```
"1" + "2"

"1".to_i
"1.0".to_f

"1".to_i + "1".to_i
```

On the web, everything is a string. Even numbers are strings. When you enter your age into a website, when that number gets into Ruby, it's going to be a `String`.

If you don't realize that a number is actually a `String`, you may try to do math with it. And, lucky for you, `String`s support addition. So you won't get an error.

Unfortunately, it's not mathematical addition. Adding two `String`s together just mashes them together. So String-1 plus String-2 is String-12, not the number 3.

To convert a `String` to an `Integer`, use `to_i`, as in "convert this string in*to* an *integer*. Similarly for a `Float`, use `to_f`, meaning "convert this string in*to* a *float*. Then you can do math with actual numbers.

Converting into Strings

```
1.to_s

1.to_s + 2.to_s
```

You can convert your numbers back into Strings as well. `to_s` means "convert this thing in*to* a String".

Most things in Ruby support `to_s`.

The "to_" pattern shows up a lot in Ruby. It's a convention for converting between different types of things. Like all conventions, it's not a requirement, but it's there to help.

Character Encodings

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
32	20	[SPACE]	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u

Computers don't actually understand letters. Computers can solve any math problem, but math problems only deal with numbers. Sure, there are variables in mathematical equations, which we'll get to, but those are just placeholders for numbers.

So, in the 1960s, what later became known as the *American* National Standards Institute (or ANSI) decided how numbers correspond to letters. (I'm emphasizing *American* because that's going to be important later)

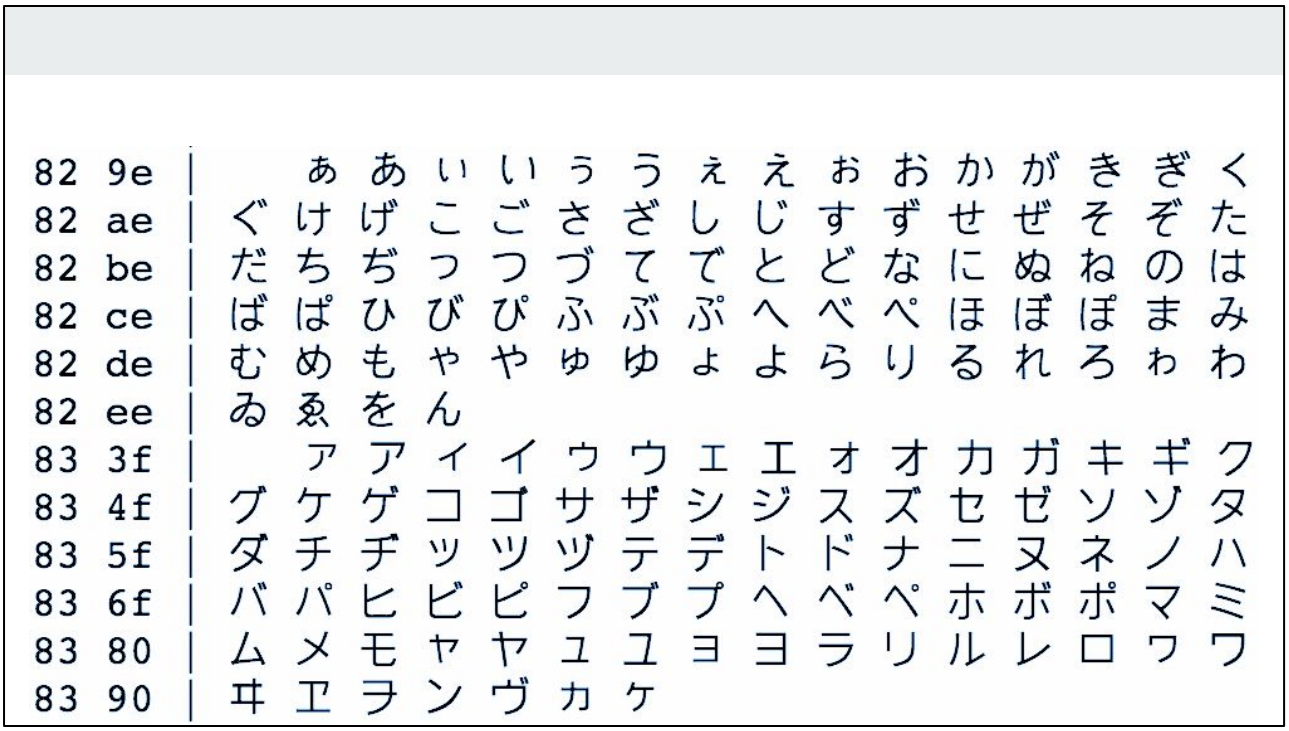
My computer only knows how to create numbers. If my computer wants to send your computer the letter "lowercase a", it sends you a 97. If you're using the same table I'm using (like this one here), your computer knows that 97 means "lowercase-a" (3rd column).

These tables represent *character encodings*. There's a numeric code for each character. (Ignore the Hex column. Focus on the Decimal column instead. We'll talk about Hex later.)

The U.S. character encoding, the *American* Standard Code for Information Interchange (or ASCII), has codes for all the letters and punctuation in the *American* English language. So if I wanted to send you the string "A", I'd send you a 65 (2nd column in this table).

Confusingly, if I wanted to send you the String "1" (rather than the number 1) I'd send you a 49.

This, for obvious reasons, doesn't work with other languages. There are no encodings for foreign characters. This doesn't even work for the British because, for example, this table doesn't have an encoding for the British pound symbol.



So, as expected, other countries created their own character encodings. Japan had one for Japanese characters. China had one. Russia had one.

The character encodings for India, Vietnam, and Yugoslavia even borrowed the name ASCII, calling themselves ISCII, VISCII, and YUSCII respectively.

This was a problem. A computer in Japan couldn't output Strings that could be understood by a computer in the U.S.

Character Encodings

- ANSI
  - ASCII
- Unicode
  - UTF-8

```
print "£"
```

So in the 1980s a group of programmers got together to create a "universal character encoding". They called it Unicode. It would try to make enough room for all of humanity's written characters, including Asian languages (like Chinese, Japanese, and Korean) and even historic languages (like Egyptian Hieroglyphics).

The Unicode group produced a few different encodings, adjusting over time as individual countries chimed in. For example, rarely used Asian characters that were part of the names of people or places petitioned to be added to the standard.

The current predominant standard is UTF-8 (the 8-bit version of the UCS Transformation Format). In typical American style, UTF-8 matches precisely code-for-code all the code mappings in ASCII. So no computers using ASCII needed to change. For example, they could continue using the same code 97 to represent lowercase-a. U-S-A! U-S-A! [85, 83, 65]!

All the new characters were just added to the end of the table, growing ASCII's original 128 character set to 1,112,064 characters.

But other character encodings, including older versions of UTF, are still out there. So Ruby, and most other programming languages, have libraries to translate between those different encodings. We won't get into that, but you will see UTF-8 appear later when we start building webpages. UTF-8 is the most common character encoding on the web.

The Future of Unicode

```
print "💩"
```

https://en.wikipedia.org/wiki/Emoji

Starting in 2010, emoji characters were introduced into the Unicode table.

So, after years of political fighting to petition for the addition of certain characters to the Unicode standard, we've decided rather casually to add things like smileys (in multiple ethnicities), food, Zodiac signs, symbols…

and, of course, the pile of poop - the true height of technological progress.

Protip: ctrl + command + space opens an emoji keyboard on Mac, which you can use to enter emoji's into IRB and, later, into your Ruby programs.

String Escapes

```
""Ruby would be easy", they said"
```

One final note about strings. Since we use double-quote to delineate the beginning and end of a String, how do you use double-quote *inside* a String?

This won’t work. Ruby says it’s a syntax error.

String Escapes

```
print '"Ruby would be easy", they said'

print "\"Ruby would be easy\", they said"

print '\'Hi\''
```

There are two ways to fix this problem. The first is that you can use single-quotes instead of double-quotes around a String. That's good enough for most people.

But what if you want to use single and double-quotes in a String?

The better answer is that you can use "escape characters". If you use a backslash in a String, the character after the backslash is treated as special. So if you use backslash-double-quote, Ruby treats it as an actual double-quote. The same happens if you use backslash-single-quote.

This is certainly more confusing to read. Don't be afraid to think back to your String state machine to understand what's going on here. Going left to right, when it sees the first double-quote, it enters the "I'm reading characters in a String" state. When it sees the backslash, it enters the state, "The next character, and only the next character, is special.". Then it reads the special character. Then it goes back to the "I'm reading a String again" state. And it continues until the first, non-special double-quote.

Note here that I'm using the `print` method again. When you just enter a String into IRB, it shows you the escape characters. But when you "print" the String, it shows you the result *after* escaping the characters.

String Escapes

```
print "Windows uses the backslash (\)
character to separate directory names."
```

But doesn't this just introduce another problem?

First we introduced single-quotes around a String as a way to fix double-quotes inside a String. Then we introduced backslashes before a character as a way to fix both single- and double-quotes inside a String. But what if I need to use a backslash in a String? We broke the backslash. This line of code ignores the backslash completely.

Are we just going in circles?

String Escapes

```
print "Windows uses the backslash (\\)
character to separate directory names."
```

Escape codes are a common pattern in computer programming. You're going to see them again when we talk about the web.

Using escape codes requires you to pick a single character to indicate the beginning of something special. But once you pick your escape character, how do you use it for anything other than escape codes?

The answer is you need to *escape the escape character*. You have to use a double-backslash to represent a single backslash.

The backslash character isn't particularly special. It's just a character, ASCII 92. But Ruby (and many other languages in the C-family) have arbitrarily promoted it, so now it's "special".

Boolean

```
true

false

(1 == 1).class
```

Just typing the words "true" and "false", without quotes, outputs a boolean value, which is just a fancy word for something that's either true or false.

Ruby says true's class is "TrueClass" while false's class is "FalseClass", but you don't have to worry about those right now. Just call them Booleans.

Booleans are going to be important when we talk about programming with logic, but they're not all that useful on their own. You'll usually only see it in your output when you do comparisons. In fact, you'll rarely have to write the words true or false in your program. We'll find out why later.

Nil

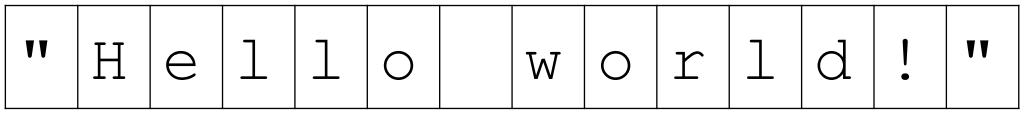
```
"" <=> 1
nil?
nil.nil?
("" <=> 1).nil?
(1 <=> "") + 1
NoMethodError: undefined method `+' for nil:NilClass
```

Nil, like Boolean, is more useful as the result of something else you write. So, for example, if you try to compare a String and an Integer using a spaceship, the result is nil. (Spaceship prefers to return nil rather than display an error, which is what greater-than and less-than do.)

The "nil?" method exists everywhere. It's a way for you to check whether or not an expression evaluates to nil.

The nil? method is useful because nils have a tendency to appear in unexpected places, leading to some of the most common errors in programming. For example, in this penultimate line, you might expect the spaceship to return -1, 0, or 1. But you made a mistake, so it's returning nil. Since you're using method chaining, that nil ends up being added to 1, which returns an error. You'll probably see errors like this a lot.

Array

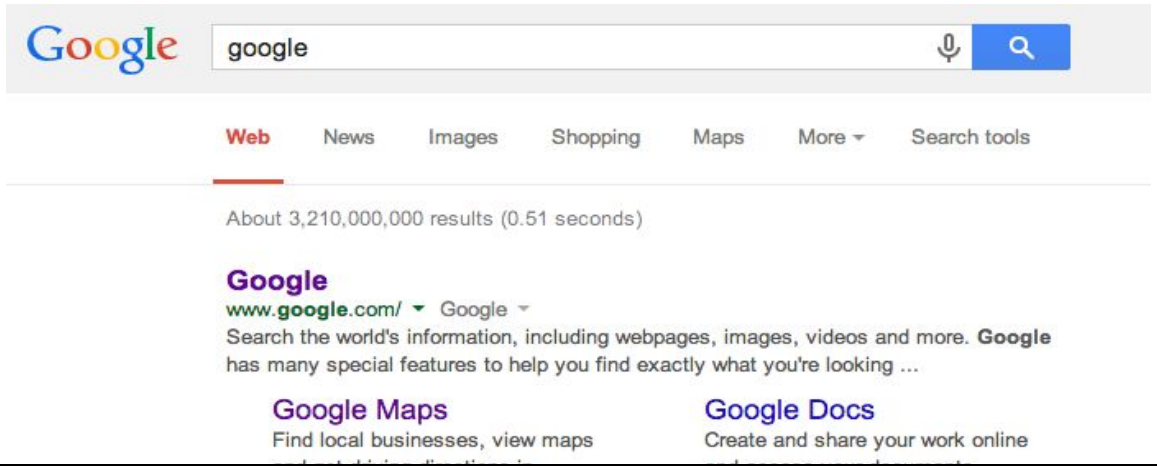


Now let's get to the good stuff. Up to now we've been talking about building blocks. Now it's time to start building stuff with those blocks.

An Array is just a list, but programmers call them Arrays. An Array is a data structure, meaning that it contains other pieces of data.

The Array represents, in a very literal sense, the Turing Tape at the foundation of computer programming. It's the one aspect of software that most closely echoes the underlying hardware.

Array



Arrays are fundamental and everywhere. For example, they are the most common data structure on the web.

What's Google? It's an Array of search results.

Array

What's Amazon? It's an Array of products that you can buy.

Array

```
[1,2,3]

["chunky", "bacon"]

[true, 0, 1.0, nil, "hello"]
```

Arrays are everywhere online. And creating them is pretty easy. Just take any of the basic types of data we've seen before, separate them with commas, and wrap square brackets around them.

You can even combine different types of data into a single array. Although this is possible, I don't recommend mixing data types in one array.

Array

```
[]

[[1,2,3], ["a", "b", "c"]]
```

Arrays can be empty.

Arrays can even contain other arrays.

Array

```
[] << 1
[].push(1, 2, 3)
[1,2,3].length
[1,2,3].first
[1,2,3].last
```

You can add items to an array with the double bracket. If you want to add multiple items at once, use push instead.

You can get the length of an Array, just like you did with a String.

You can get back the first or last item in the array.

Array

```
[1,2,3][0]
[1,2,3].[](0)
[1,2,3].at(0)
```

Arrays start counting at 0!

What if you want to get back some item that isn't first or last?

Getting random items back out of an array is interesting. It turns out there a new piece of Ruby syntax that just applies to Arrays, the square bracket.

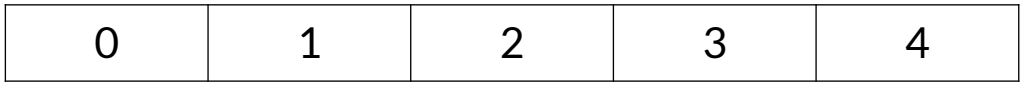
After an Array, if you attach a number in square brackets, **called an index**, you'll get back the item at that position in the Array.

The standard syntax equivalent of the square bracket is pretty weird, but it does obey the subject-dot-method notation.

If you really want to use the standard syntax, I recommend the "at" method. But no one does that. Everyone uses the square bracket notation, [1, 2, 3][0]. All the historical array algorithms are written using this bracket notation.

Big red flag warning: Arrays start counting at 0. Arrays start counting at 0. Arrays start counting at 0.

Arrays start at 0




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

I know, it's confusing, but it's an old decision that programming languages have inherited from their forefathers. (Arrays are as old as they are important). The first item in the Array is at position 0. The second is at position 1. And so on. We just have to deal with it.

One of the most common errors in programming is the "off-by-1" error when someone forgets that... *Arrays start counting at 0*. So, for example, in this expression at the bottom, if you try to get the item at position 3, and you expect to get the number 3, you won't. You'll get nil.

Array

-5	-4	-3	-2	-1

Try:

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

Now it’s about to get even more confusing.

Arrays allow negative indices (plural of index). But, unlike positive indices, negative indices *do* start a 1. So the last item is at index -1. The 2nd to last item is at index -2. And so on.

So if you want the third from the end, you can use -3. If you try to negative-index too far into the Array, you get nil.

I know this is terrible. This is why even experienced developers make mistakes. And since doing the wrong thing sometimes results in nil, then off-by-1 errors turn into nil errors. It’s a conspiracy.

Strings are Arrays

```
"hello"[0]
"hello"[-1]
"hello" << " world"
```

Strings are Arrays of characters, so some, but not all, of the same methods can be used.

Symbol

```
:a

:a.class

:a.methods
"a".methods - :a.methods
```

What's a Symbol?

A Symbol is a cheap String. To create a symbol, you precede some letters with a colon.

What do I mean by a cheap String? Symbols are "smaller" than Strings. For example, let's compare their supported methods.

Did you know you can subtract Arrays? When you subtract Arrays, you get back items in the first Array that aren't in the second Array. Subtracting Arrays is like deleting multiple items from an Array at one time.

If we take the list of methods the String "a" has and subtract the list of methods the Symbol "a" has, we're left with the list of methods that apply to Strings, but not to Symbols. And there are a bunch of 'em. So the Symbol-a does much less than the String-a, which makes it cheaper and faster for Ruby to build.

Symbol

```
:a.object_id == :a.object_id

"a".object_id == "a".object_id
```

Besides being smaller, Symbols are also faster in two more ways.

First, symbols are recyclable. Every time you tell Ruby to make a String, it's makes one fresh-to-order. And you can tell it's fresh because it has a new object_id, which is like the serial number Ruby uses to keep track of the String (don't forget, computers are all numbers behind the scenes).

But every time you use a symbol, Ruby checks if it can find one to recycle. If it can find one, it uses it. Otherwise it generates a new one.

So the first time you type colon-a, Ruby digs through the symbol Recycle Bin for a Symbol-a. If it doesn't find one, it creates a new one. The next time you type colon-a, Ruby sees that there's already one around, so it returns that one rather than make a new one. How do you know it's a recycled symbol? Because Ruby didn't generate a new serial number for it. Every symbol-a object-id is the same.

The second way Symbols are faster is that they're just faster to type. colon-a versus double-quote, a, double-quote. It's one less button for you to press. So it's faster to type out.

In our next Ruby data type, we'll see a common use of Symbols.

NOTE: Symbols are not Strings. Sometimes Strings are the right tool for the job.

You can’t concatenate two symbols together. You can’t use spaces or control characters in Symbols.

Hash

Good (Strings as KEYS): KEY VALUE

```
{ "first" => "John", "last" => "Doe",
"age" => 32 }
```

Better (Symbols as KEYS): KEY VALUE

```
{ :first => "John", :last => "Doe",
:age => 32 }
```

A hash is similar to an Array that it stores other data types. However, the way values in hashes are stored and accessed is different from Arrays.

As you may recall, you can access an array using an index, e.g. list[2].

Instead of indices, hashes use KEYS to store and access VALUES. KEYS can either be `String`s or `Symbol`s. VALUES can be other data types.

Hash

Better (Symbols as KEYS):

KEY VALUE

```
{ :first => "John", :last => "Doe",
:age => 32 }
```

BEST (Symbols as KEYS):

KEY VALUE

```
{ first: "John", last: "Doe", age: 32 }
```

In fact, the new leaner Hash syntax looks even better. Keep in mind that the BEST way uses a shorthand notation. We are using Symbols but aren't using the `:name =>` construct.

But regardless of which one you use, almost everyone prefers to use Symbols rather than Strings. Faster is better.

Hash - Accessing Values

```
{ first: "John", last: "Doe", age: 32 }[:age]

{ first: "John", last: "Doe", age: 32 }[age]
```

DOES NOT WORK. WHY?

```
{ "first" => "John", "last" => "Doe", "age" => 32
}["age"]

{ "first" => "John", "last" => "Doe", "age" => 32
}["address"]
```

Returns nil

To access a value in a Hash, use the `[KEY]` syntax.

NOTE: If you use a KEY that does not exist in the hash, it will return nil.

Arrays vs Hashes

Array:

```
["John", "Doe", 32][2]
```

Hash:

```
{ first: "John", last: "Doe", age: 32 }[:age]
```

Q1: Which data structure would be better to store contact info?
Q2: What if I wanted to store contact info of many people, what data structure could I use?

Summary

Arrays store values that can be accessed using an Index (Integer).

Hashes store values that can be accessed using a Key (Symbol or String).

Each of these data structures can store other data types. Because they are stored and accessed differently, they each are useful in their own way. **Think about scenarios in which Arrays are better suited that Hashes, and vice versa.**

Object

```
Object.new
Object.new.methods
Object.new.class

self.class
```

The last Ruby data type is the catch all. When none of the other types apply, what's left is a plain old Object.

You can't do anything interesting with an Object. An Object is just a template. It's the foundation upon which all the other types are built.

The REPL itself is just an Object.

Object

```
Integer:    1.class.ancestors
Float:      1.0.class.ancestors
String:     "1".class.ancestors
Boolean:    true.class.ancestors
Nil:        nil.class.ancestors
Array:        [].class.ancestors
```

All the types we've seen so far:

```
Integer
Float
String
Boolean
Nil
Array
Hash
Symbol
```

All of them share the same 3 components.

Object

`Object`: the foundation of everything

`Kernel`: where methods like `print` are hiding

`BasicObject`: even more useless than `Object`

`Object`: the foundation of everything, like the atoms of data types

`Kernel`: the library where methods like `print` are hiding

and `BasicObject`: something even more useless than `Object`s

It's actually not fair to call `Object` useless. A plain `Object`, created by `Object.new`, is pretty useless. But the concept of an Object is incredibly important. It's a template that allows you to create your own, brand new data types.