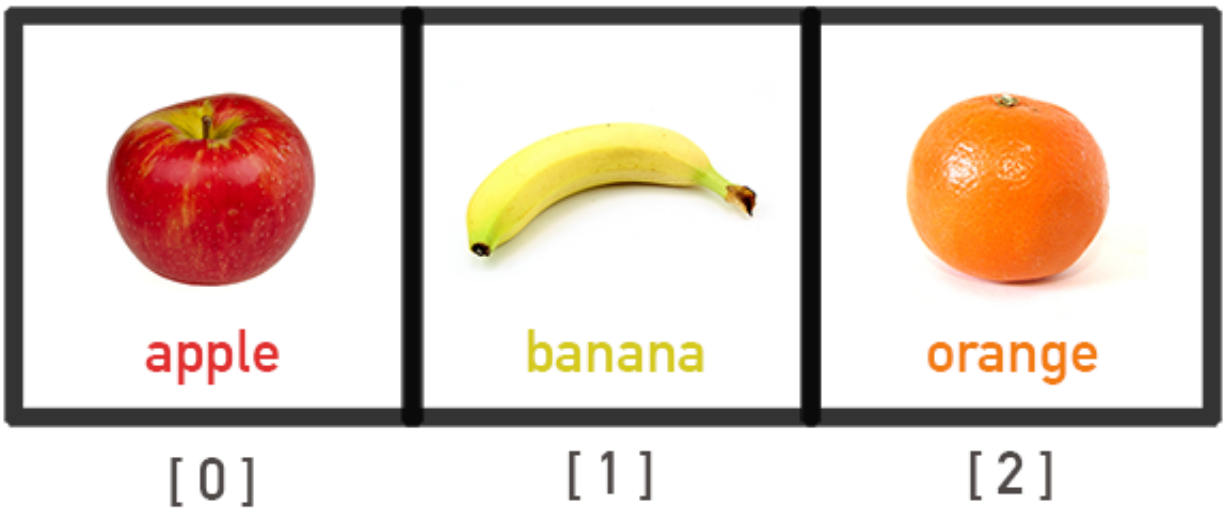
6 Ruby: Arrays

A Collection of Objects

Strings and numbers represent *singular* objects, but you'll often work with *collections* of objects in programming. Ruby, like most programming languages, provides an object called `Array`. An array is a collection of objects (like `String`s and `Integer`s) in a specific order.



Each element in an array has a position (or "index") within it. To retreive this element, use the same syntax that specifies a character in a string: `[]`. Like string indexes, the index count for an array starts at `0`. Consider the following example:

```ruby
fruits = ["apple", "banana", "orange"]

fruits[0]
#=> "apple"

fruits[2]
#=> "orange"
```

> Everything after a `#` in Ruby is a comment. Here, we're writing `#=> "apple"` to show what that line above the comment should return.

To get the last element in the `fruits` array, we could just type `fruits[2]`. But what if we didn't know the length of the array? Ruby provides some simple methods for accessing specific elements in an array. We can use the `last` method to solve this problem:

```ruby
fruits = ["apple", "banana", "orange"]

fruits.last
#=> "orange"

fruits.first
#=> "apple"
```

In the above example, we declared a complete array:

```ruby
fruits = ["apple", "banana", "orange"]
```

But we can also assign elements to an array one at a time:

```ruby
fruits = ["apple"]
fruits[1] = "banana"
fruits[2] = "orange"

fruits
#=> ["apple", "banana", "orange"]
```

> What happens if we add an element to a location greater than the length of the array? Try it using

> **repl.it**.

Use the "shovel" operator (`<<`) to add an element to the end of an array.

```
numbers = [1]
numbers << 2

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

> Adding an element to the end of an array is called *appending*.

Arrays can mix object types. For example, an array can hold a combination of strings, numbers and even other arrays:

```
multi_object_collection = []
multi_object_collection[0] = "apple"
multi_object_collection[1] = 17
multi_object_collection << ["hello", 3]

multi_object_collection
#=> ["apple", 17, ["hello", 3]]
```

`%w` syntax

Ruby offers many shortcuts. As a developer, you're likely to see arrays created using Ruby's **Percent Strings** syntax. Percent Strings like `%w` are followed with opening and closing symbols (usually `{}`, but others like `[]` or `!!` work). Strings inside are delimited with a space:

```
fruits = %w{apple banana orange}
#=> ["apple", "banana", "orange"]
```

This syntax creates only strings (not numbers or other types) inside the array:

```ruby
integers = [1, 2, 3]
#=> [1, 2, 3]
strings = ["1", "2", "3"]
#=> ["1", "2", "3"]
integers_or_strings = %w{1 2 3}
#=> ["1", "2", "3"]
```

> The quotes around the numbers in the output indicate the values are strings.

Capitalization matters. Remember *string interpolation* from the Ruby Syntax checkpoint? When capitalized, `%W` processes interpolation:

```ruby
%w{10 20 #{10+20}}
#=> ["10", "20", "\#{10+20}"]
%W{10 20 #{10+20}}
#=> ["10", "20", "30"]
```

Negative Indexes

Indexes, like `0` and `1` in the examples above, retrieve elements from an array going left to right. Ruby also supports negative indexes, which retrieve elements from right to left. For example:

```ruby
fruits = ["apple", "banana", "orange"]

fruits[-1]
#=> "orange"

fruits[-2]
#=> "banana"

fruits[-3]
#=> "apple"
```

> When using negative indexes, counting starts at `-1`, rather than `0`.

What are two other ways to access the element in location `[-1]`?

Ranges

A `Range` specifies a sequential set of `String`s or `Integer`s. A range looks like this:

```
(1..5)
("a".."e")
```

We can create an `Array` from a `Range` using the `to_a` ("to array") method:

```
(1..5).to_a
#=> [1, 2, 3, 4, 5]

("a".."e").to_a
#=> ["a", "b", "c", "d", "e"]
```

> The parentheses around the range are optional but important for order of operations. What would happen if we removed the parentheses from `(1..5).to_a`? Use **repl.it**, and see if you can understand the error message.

We can also index multiple `Array` elements with a `Range`:

```
fictional_cats = ['Cheshire Cat', 'Garfield', 'Catwoman', 'The Cat in the Hat']

fictional_cats[1..3]
#=> ["Garfield", "Catwoman", "The Cat in the Hat"]
```

We can also use negative indexes in ranges:

```
fictional_cats = ['Cheshire Cat', 'Garfield', 'Catwoman', 'The Cat in the Hat']

fictional_cats[-2..-1]
#=> ["Catwoman", "The Cat in the Hat"]
```

Exercises

- [x] ~~Array Definition~~
- [x] ~~Array Methods~~
- [x] ~~Grocery List~~

Solution Walk-Throughs

Do not watch these videos until after you've attempted your own solutions. If you struggle to complete these exercises, submit your best effort to your mentor *before watching a solution walk-through*. Submit your answers even if some tests don't pass, as it's important for your mentor to understand your thought process. Discuss difficult exercises with your mentor.

There are many possible solutions to the exercises. A walk-through is meant to provide insight into the thought process of *one possible solution* per exercise. If your solution is different yet passes all the tests, do not change your solution to match what is seen in the video. Instead, discuss your thought process with your mentor.

1. **Array Definition Solution**
2. **Array Methods Solution**
3. **Grocery List Solution**

assignment completed

?