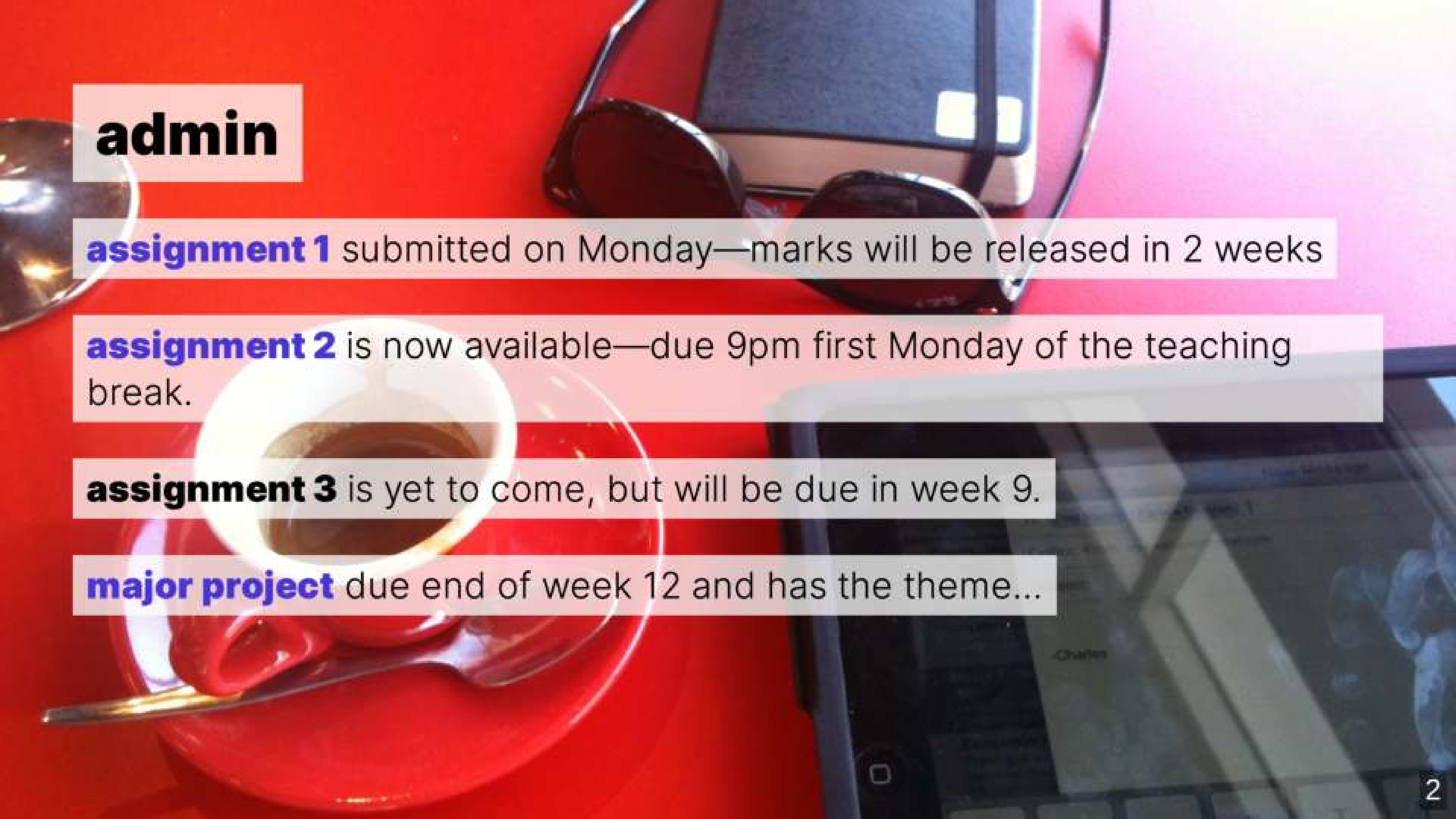
COMP1720

Art & Interaction in New Media

Week 4: functions & arrays

Dr Charles Martin

Semester 2, 2020



admin

assignment 1 submitted on Monday—marks will be released in 2 weeks

assignment 2 is now available—due 9pm first Monday of the teaching break.

assignment 3 is yet to come, but will be due in week 9.

major project due end of week 12 and has the theme...

fearful symmetry

recap

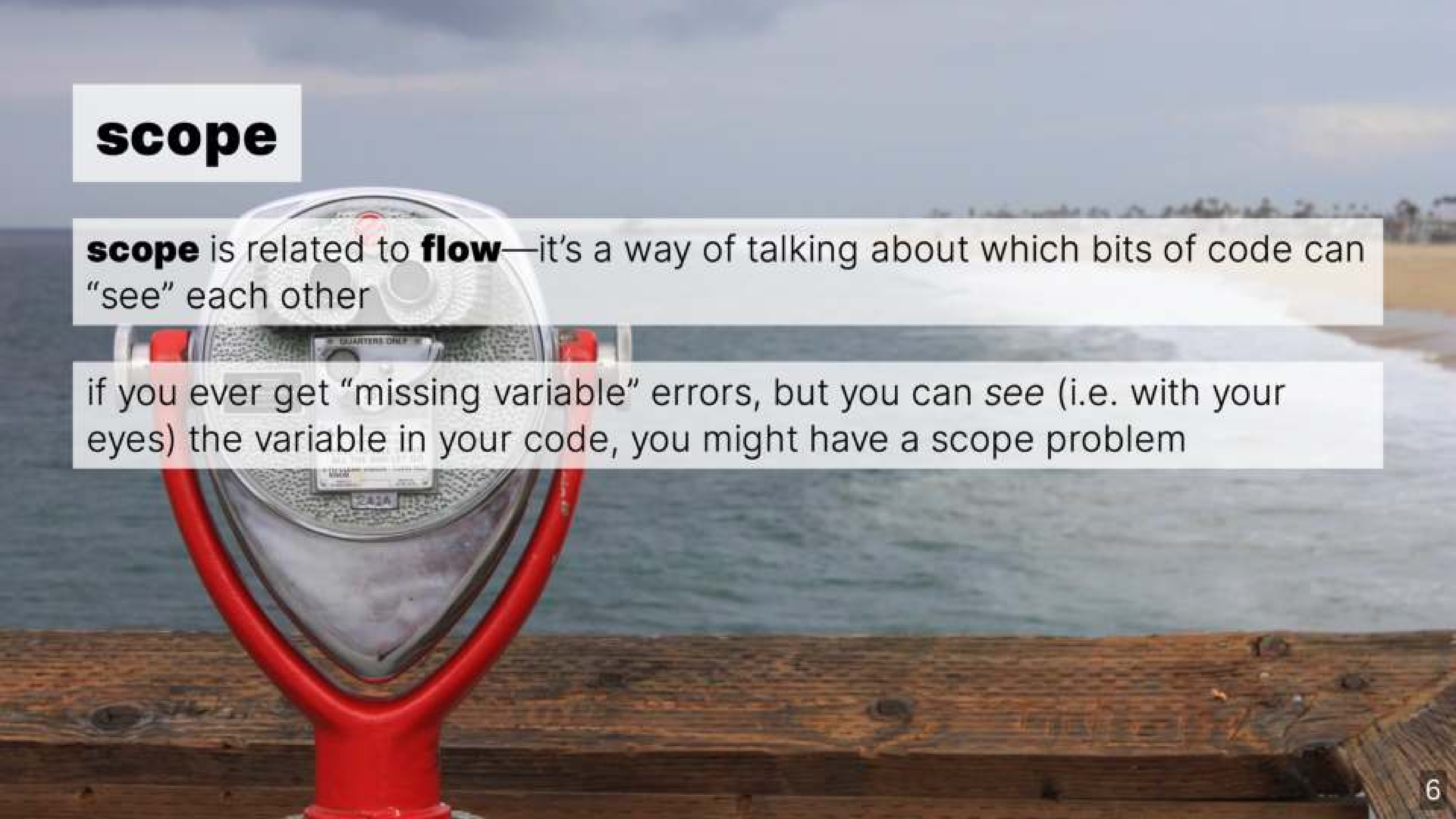
- specifying positions & colours with numbers
- doing stuff with the p5 library (drawing `rect()`s and `ellipse()`s, setting `fill()` and `stroke()` colours)
- using variables (and maybe even declaring your own)
- doing simple maths with `+`, `-`, `*`, `/`
- doing things conditionally with `if` statements and **Boolean** expressions like `mouseX < 100`
- looping with `while` and `for`

code theory

variable scope (a new word for something you’ve seen before)

functions (including making our own functions)

arrays (collections of things)



scope

scope is related to **flow**—it's a way of talking about which bits of code can "see" each other

if you ever get "missing variable" errors, but you can *see* (i.e. with your eyes) the variable in your code, you might have a scope problem

global scope

variable declarations “outside” of all the functions (e.g. `setup()` and `draw()`) are said to be in the **global** scope, and they’re visible from anywhere in the program

```
// x is a "global" variable
var x = 200;

function setup() {
  createCanvas(800, 600);
}

function draw() {
  ellipse(x, x, x, x);
}
```

but what about this?

```
function setup() {
  createCanvas(800, 600);
  var x = 200;
}

function draw() {
  ellipse(x, x, x, x);
}
```

if something’s not working

check the console

how about this?

```
function setup() {
  createCanvas(800, 600);
}

function draw() {
  var x = 200;
  ellipse(x, x, x, x);
}
```

this works because `x` is in the `draw` function's scope—it's visible inside `draw`'s curly brackets, but not outside

scoping tips

scoping might cause frustration at first, but it's actually a good thing—isolation makes our code **clearer** & **more robust**

in general, having lots of **global variables** is bad coding style—variables should only be visible (in scope) where they'll be used

the brackets matter!

`{ } [ ] ( )`

concept 2: functions

first, some definitions...

> **function** (*noun*): the act of executing or performing any duty, office, or calling; performance **https://www.wordnik.com/words/function**
>
> **function** (*noun*): a sequence of program instructions that perform a specific task, packaged as a unit **https://en.wikipedia.org/wiki/Subroutine**

anatomy of a function (recap)

```
name(parameter1, parameter2, ...);
```

```
rect(100, 100, 100, 100);
```

the **name** (in this case `rect`) specifies *what* to do

the **parameters** (in the brackets) tell the function *how* to do it, e.g. where to draw the rect and how big

together, they allow us to tell the computer do some basic thing, repeatedly, and with slight differences each time

speak the lingo

we say we "call" a function because we're telling it to do its' job (like the staff at a restaurant)

how many jobs should a function have?

the "**you had one job**" principle is important for functions!

talk

```
rect(100, 100, 100, 100);
```

how do you find out what the parameters mean?

check the `rect()` **reference**

writing your own functions

you've been using functions all along: `setup()` and `draw()`, as well as all the p5 functions like `ellipse()`

You can write your **own** functions where you get to pick the parameters & what they're called. Here's an example `polkadot()` function

```
function polkadot(x, y){
  fill(255,0,0);
  ellipse(x, y, 20, 20);
}
```

creating functions that give back values

parameters allow us to send values (parameters) *in* to a function, how do we get values back *out*?

the answer: we use a `return` statement in the body of the function

```
function double(x) {
  return x * 2;
}
```

now we can use our function like this

```
background(double(50));
```

p5 has a few other “special” functions

special from a *flow* perspective, anyway

- `mousePressed()`
- `keyPressed()`

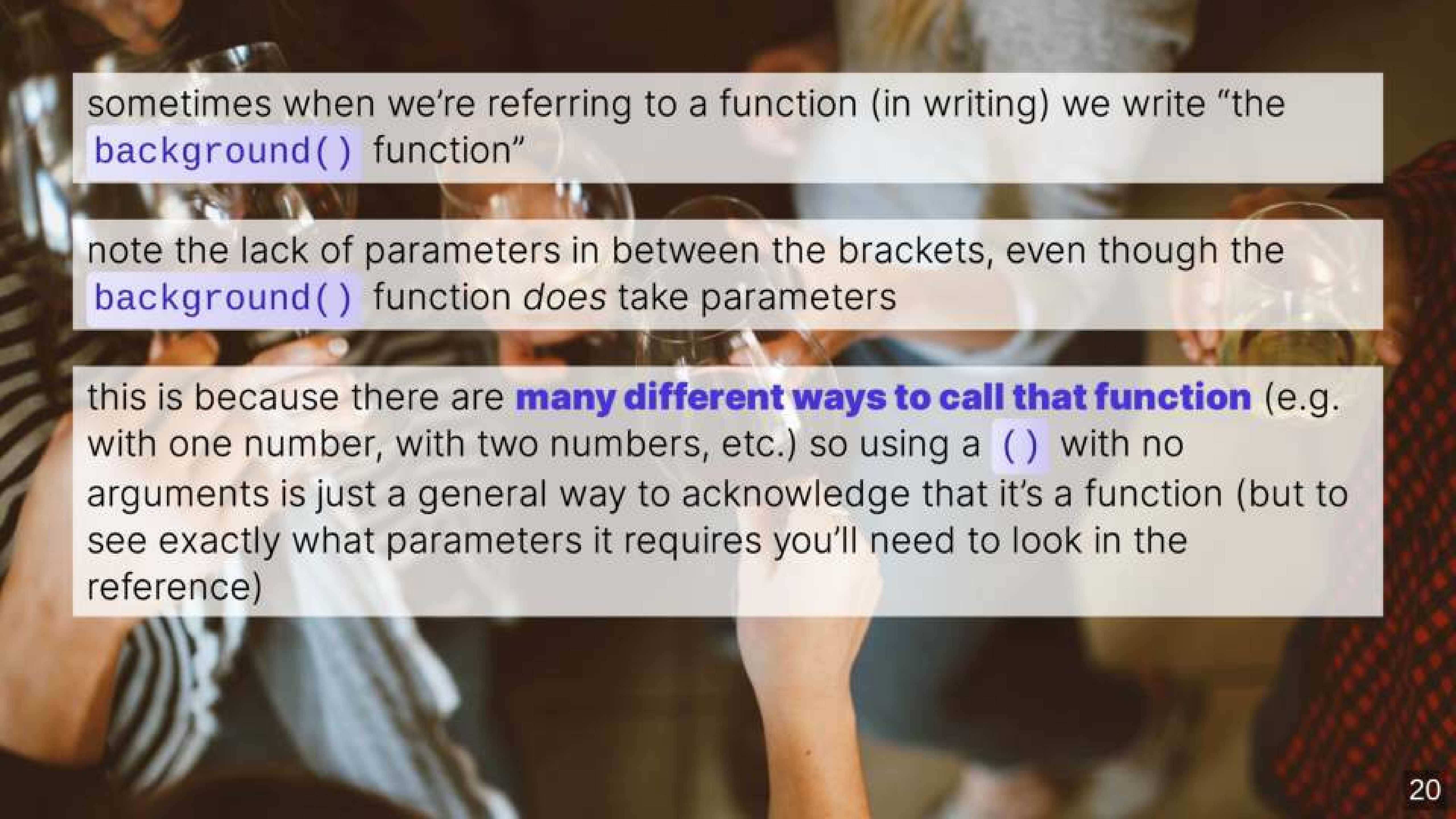
and a few more...

read the reference!

I really don't mean to harp on about this, but if you can't read the reference then you'll really have trouble

MDN has some great docs on **Functions**

we'll use functions **constantly** for the rest of the course, so it's really worth getting your head around them



sometimes when we're referring to a function (in writing) we write "the `background()` function"

note the lack of parameters in between the brackets, even though the `background()` function *does* take parameters

this is because there are **many different ways to call that function** (e.g. with one number, with two numbers, etc.) so using a `()` with no arguments is just a general way to acknowledge that it's a function (but to see exactly what parameters it requires you'll need to look in the reference)

example: making a button

from this week's labs: making a simple "button" has two sub-tasks:

1. drawing the button
2. figuring out whether the button is clicked

these sub-tasks require the same info, though: where and how big is the button?

let's combine them into a function which

1. draws a rectangle
2. returns a `Boolean` (either `true` or `false` depending on whether the button is being clicked)

concept 3: arrays

we've already met the **Number**, **String** and **Boolean** types in this course

```
var myNumber = 7;
var myString = "tennis is fun";
var myBoolean = true;
```

Can we collect some of these things together into a group?

what do you think these are?

```
var arrayOfNumbers = [100, 24, -2, 18, 106, 42, 1, 8];
var arrayOfStrings = ["hello", "darkness",
                      "my", "old", "friend"];
var arrayOfBooleans = [true, false, true, true, false];
```

```
var arrayOfWhatever = [100, 200, false, "Banana"];
var emptyArray = [];
```

they're **arrays**

again, look for the matching pairs

when you see a [, there will be a matching]

everything inside will be initialised as the elements of the array

some new vocabulary

the *array* is the whole collection

each member of the array is called an *element*

the “element position” is called the *index* (e.g. the first element is at index `0`, the second at index `1`, etc.)

the number of elements in the array is called the *length* of the array

what's going on here?

```
var allTheThings = [0, 120, 500];
```

we're **declaring** a variable called `allTheThings`, and **initialising** it to be an array with 3 elements: `0`, `120` and `500`

arrays as variables

```
// < variable part >  < array part >
   var allTheThings = [0, 120, 500];
```

we're combining something we're learning today (arrays) with something we learned in week 2 (**declaring & initialising variables**)

no magic here!

why use arrays?

arrays are a really useful part of javascript

as well as just declaring & initialising an array, there are a bunch of things you can do to it by default

- find out how big it is
- add/remove/modify elements
- join it to other arrays
- look for particular elements in the array
- etc.

the **Array reference is on MDN**, but several p5 functions use arrays as well

oddNumbers array

for the following slides, assume we've got an array `oddNumbers` with some stuff in it

```
var oddNumbers = [1, 2, 3, 5, 7];
```

using the elements in an array

use square brackets to “reference” (i.e. retrieve) an element from an array

```
var firstOddNumber = oddNumbers[0];
var secondOddNumber = oddNumbers[1];

background(oddNumbers[3]);
```

array referencing gotchas

the array index starts at `0`, so `oddNumbers[0]` is the first element, and `oddNumbers[n]` is the `n+1`th element for any index `n`

if you try and access an element that isn't there, the result is undefined

```
// this will break because there is no 40th element
background(oddNumbers[40]);
```

can we change the things in arrays?

putting elements into an array

to put an element onto the "end", use `push()`

```
oddNumbers.push(53);
// oddNumbers is now [1, 2, 3, 5, 7, 53]
```

to put an element onto the "front", use `unshift()` (**weird name**)

```
oddNumbers.unshift(-7);
// oddNumbers is now [-7, 1, 2, 3, 5, 7, 53]
```

removing elements from an array

to *remove* an element from the "end", use `pop()` (opposite of `push()`)

```
oddNumbers.pop();
// oddNumbers is now [-7, 1, 2, 3, 5, 7]
```

to *remove* an element from the "front", use `shift()` (oppposite of `unshift()`)

```
oddNumbers.shift();
// oddNumbers is now [1, 2, 3, 5, 7]
```

here’s a table

```
var things = [1, 2, 3];
```

	add	remove
front	`things.unshift(value)`	`things.shift()`
back	`things.push(value)`	`things.pop()`

modifying the elements in an array

similar to using the elements of the array (e.g. `oddNumbers[0]`) but this time we assign a new value to that element using the equals sign

```
// oddNumbers is [1, 2, 3, 5, 7];
oddNumbers[4] = 12;
// oddNumbers is now [1, 2, 3, 5, 12];
```

talk

suppose you see this:

```
var allTheThings = [0, 120, 500];

// some code here you can't see

allTheThings.push(50);
```

what are the elements of `allTheThings` at this point? how do you know?

how do I see what's in my array at any point?

use **print** and the console!

```
// put this somewhere in your code
print(oddNumbers);
```

then, open up the console and see! (`cmd ⌘`+`alt`+`J` or `ctrl`+`alt`+`J`)

arrays *in* arrays

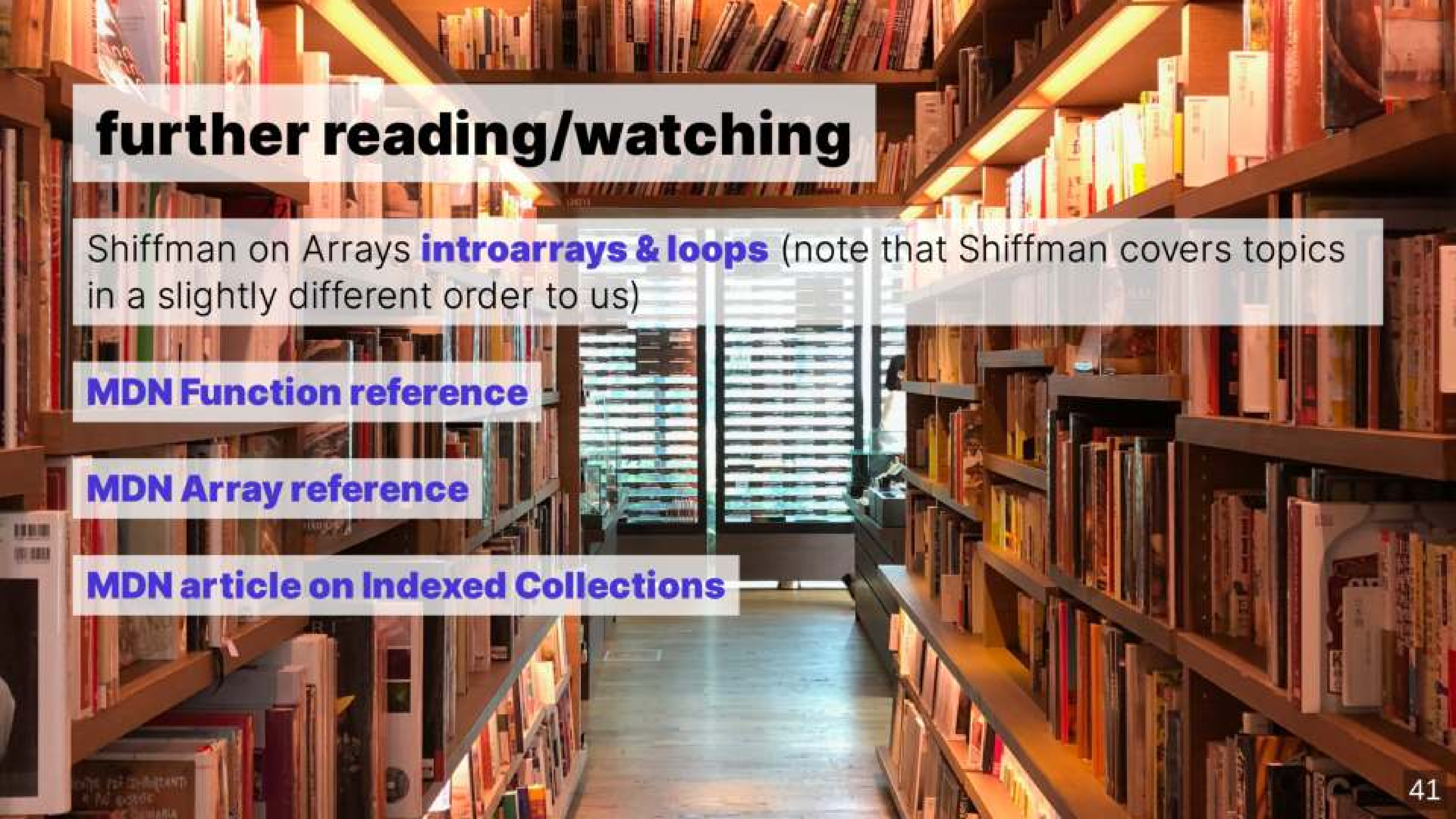
remember how we said you could put *any* type into an array? that includes more arrays!

```
var nestedArray = [[1, 3], [2, 4]];

nestedArray[0] // the value at index 0 is the Array [1, 3]

nestedArray[0][1] // what do you think this is?
```

“looping” over an array

the `for` loop from **last week’s lecture** can be used to generate the indexes

```
for(var i = 0; i < oddNumbers.length; i = i+1){
  doStuff(oddNumbers[i]);
}
```



further reading/watching

Shiffman on Arrays **introarrays & loops** (note that Shiffman covers topics in a slightly different order to us)

MDN Function reference

MDN Array reference

MDN article on Indexed Collections

questions?