Homework 1 - *images and arrays*

18.S191, Fall 2023

This notebook contains *built-in, live answer checks*! In some exercises you will see a coloured box, which runs a test case on your code, and provides feedback based on the result. Simply edit the code, run it, and the check runs again.

Feel free to ask questions!

Initializing packages

When running this notebook for the first time, this could take up to 15 minutes. Hang in there!

```
begin
    import ImageMagick
    using Images
    using PlutoUI
    using HypertextLiteral: @htl, @htl_str
end
```

Exercise 1 - *Manipulating vectors (1D images)*

A `Vector` is a 1D array. We can think of that as a 1D image.

```
example_vector =  [0.5, 0.4, 0.3, 0.2, 0.1, 0.0, 0.7, 0.0, 0.7, 0.9]
```

```
colored_line(example_vector)
```

Exercise 1.1

👉 Make a random vector `random_vect` of length 10 using the `rand` function.

```
random_vect =
 [0.734343, 0.433268, 0.327499, 0.229958, 0.36828, 0.178648, 0.593449, 0.132584, 0.550443, 0.
```

```
random_vect = rand(10)
```

Got it!

Well done! You can run your code again to generate a new vector!

Hint

```
colored_line (generic function with 2 methods)
```

Exercise 1.2

👉 Make a function `my_sum` using a `for` loop, which computes the total of a vector of numbers.

```
my_sum (generic function with 1 method)
```

```
function my_sum(xs)
    # your code here!
    s=0
    for v in xs
        s += v
    end
    return s

end
```

```
6
```

```
my_sum([1,2,3])
```

> **Got it!**
>
> Great! 🎉

> **Hint**
>
> [blurred]

Exercise 1.3

👉 Use your `my_sum` function to write a function `mean`, which computes the mean/average of a vector of numbers.

```
mean (generic function with 1 method)
```

```
function mean(xs)
    # your code here!
    return sum(xs)/size(xs)[1]
end
```

```
Enter cell code...
```

```
2.0
```

```
mean([1, 2, 3])
```

```
Int64
```

```
typeof(size([1,2,3])[1])
```

> **Got it!**
>
> Let's move on to the next section.

👉 Define `m` to be the mean of `random_vect`.

```
m = 0.43283495875217187
```

```
1 m = mean(random_vect) # replace `missing` with your code!
```

> **Got it!**
>
> Keep it up!

Exercise 1.4

👉 Write a function `demean`, which takes a vector `xs` and subtracts the mean from each value in `xs`. Use your `mean` function!

Note about *mutation*

There are two ways to think about this exercise, you could *modify* the original vector, or you can *create a new vector*. We often prefer the second version, so that the original data is preserved. We generally only use code of the first variant in the most performance-sensitive parts of a program, as it requires more care to write and use correctly. ***Be careful not to get carried away in optimizing code**, especially when learning a new language!*

There is a convention among Julians that functions that modify their argument have a `!` in their name. For example, `sort(x)` returns a sorted *copy* of `x`, while `sort!(x)` *modifies* `x` to be sorted.

Tips for writing non-mutating code

1. *Rewriting* an existing mutating function to be non-mutating can feel like a 'tedious' and 'inefficient' process. Often, instead of trying to **rewrite** a mutating function, it's best to take a step back and try to think of your problem as *constructing something new*. Instead of a `for` loop, it might make more sense to use **descriptive** primitives like broadcasting with the dot syntax (also for math operators), and map and filter.

2. If a mutating algorithm makes the most sense for your problem, then you can first use `copy` to create a copy of an array, and then modify that copy.

We will cover this topic more in the later exercises!

```
demean (generic function with 1 method)
```

```
1 function demean(xs)
2     # your code here!
3     m= mean(xs)
4     return new_xs= xs .- m
5 end
```

```
test_vect =  [-1.0, -1.5, 8.5]
```

To verify our function, let's check that the mean of the `demean(test_vect)` is 0: (*Due to floating-point round-off error it may not be exactly 0.*)

```
0.0
```

```
mean(demeaned_test_vect)
```

```
demeaned_test_vect =  [-3.0, -3.5, 6.5]
```

```
demeaned_test_vect = demean(test_vect)
```

> **Got it!**
>
> Keep it up!

Exercise 1.5

👉 Generate a vector of 100 elements. Where:

- the center 20 elements are set to `1`, and
- all other elements are `0`.

```
create_bar (generic function with 1 method)
```

```
function create_bar()
    # your code here!
    vec=fill(0,100)
    vec[40:60] .= 1
    return vec
end
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    more ,0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

```
create_bar()
```

> **Hint**
>
> In [redacted], we drew a red square on top of the image Philip with a single command...

> **Got it!**
>
> Splendid!

Exercise 2 - *Manipulating images*

In this exercise we will get familiar with matrices (2D arrays) in Julia, by manipulating images. Recall that in Julia images are matrices of `RGB` color objects.

Let's load a picture of Philip again.

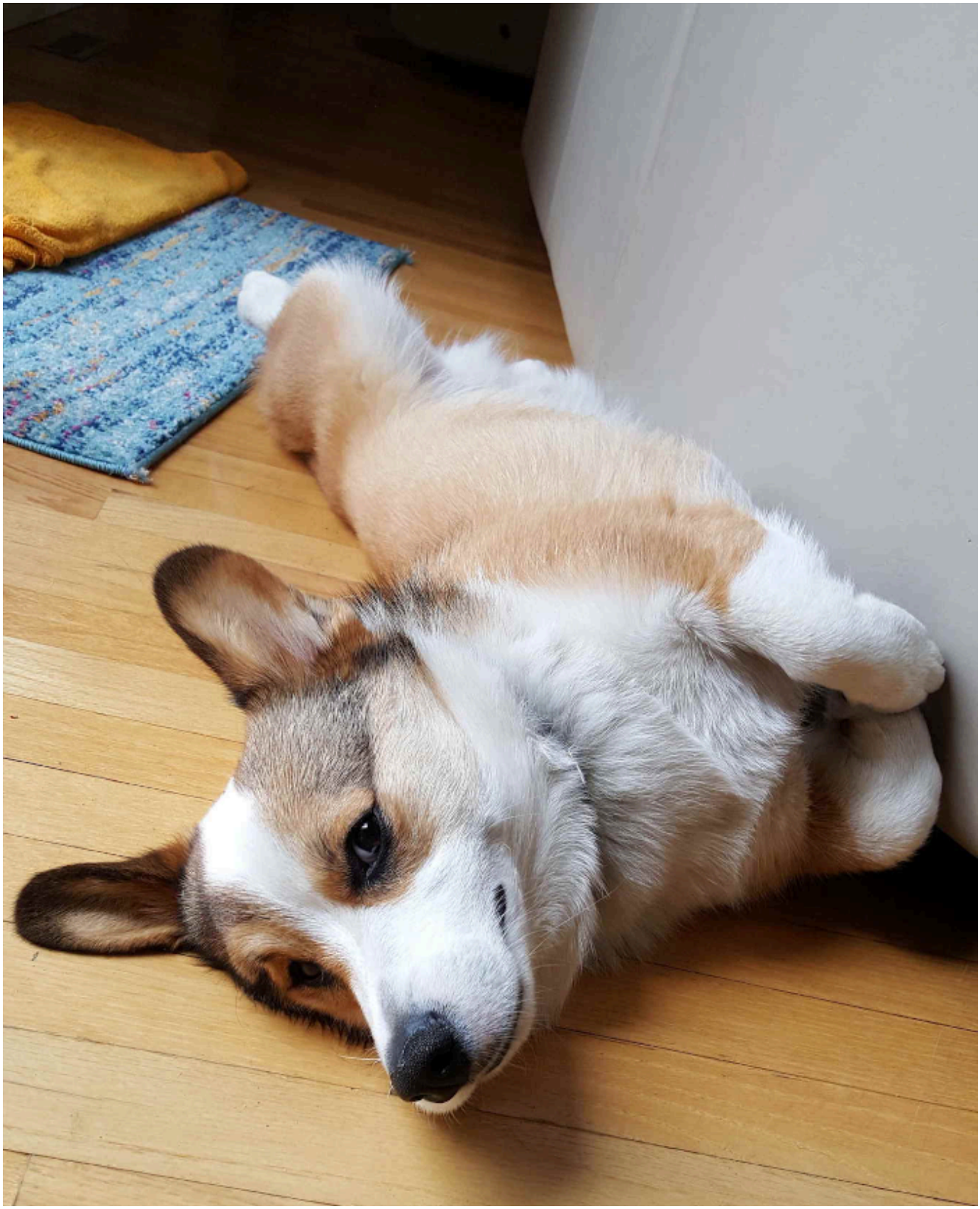
```
url =
"https://user-images.githubusercontent.com/6933510/107239146-dcc3fd00-6a28-11eb-8c7b-41aaf6
```

```
1 url = "https://user-images.githubusercontent.com/6933510/107239146-dcc3fd00-6a28-11eb-
  8c7b-41aaf6618935.png"
```

```
philip_filename = "C:\\Users\\Dell\\AppData\\Local\\Temp\\jl_iTIrudAPUv"
```

```
1 philip_filename = download(url) # download to a local file. The filename is returned
```

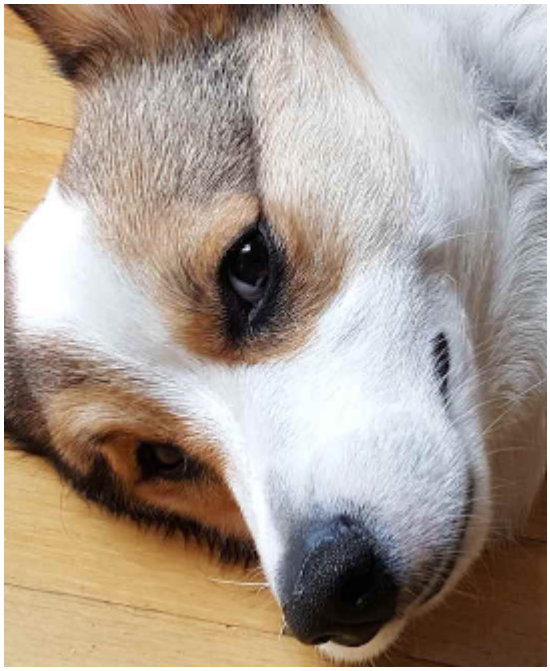
philip =



```
philip = load(philip_filename)
```

Hi there Philip

philip_head =



```
philip_head = philip[470:800, 140:410]
```

Recall from Section 1.1 that in Julia, an *image* is just a 2D array of color objects:

```
Matrix{RGB{N0f8}} (alias for Array{RGB{Normed{UInt8, 8}}, 2})
```

```
typeof(philip)
```

Every pixel (i.e. *element of the 2D array*) is of the `RGB` type:

philip_pixel =

```
philip_pixel = philip[100,100]
```

```
RGB{N0f8}
```

```
typeof(philip_pixel)
```

To get the values of its individual color channels, use the `r`, `g` and `b` *attributes*:

```
(0.455, 0.212, 0.078)
```

```
philip_pixel.r, philip_pixel.g, philip_pixel.b
```

And to create an `RGB` object yourself:

```
RGB(0.1, 0.4, 0.7)
```

Exercise 2.1

👉 Write a function `get_red` that takes a single pixel, and returns the value of its red channel.

```
get_red (generic function with 1 method)
```

```
function get_red(pixel::AbstractRGB)
    # your code here!
    return pixel.r
end
```

```
0.8
```

```
get_red(RGB(0.8, 0.1, 0.0))
```

> **Got it!**
>
> Great! 🥕

Exercise 2.2

👉 Write a function `get_reds` (note the extra `s`) that accepts a 2D color array called `image`, and returns a 2D array with the red channel value of each pixel. (The result should be a 2D array of *numbers*.) Use your function `get_red` from the previous exercise.

```
get_reds (generic function with 1 method)
```

```
function get_reds(image::AbstractMatrix)
    # your code here!
    return get_red.(image)
end
```

```
331×271 Array{N0f8,2} with eltype N0f8:
 0.647  0.482  0.325  0.22   0.243  0.263  …  0.863  0.82   0.863  0.918  0.875
 0.686  0.682  0.51   0.247  0.235  0.267     0.816  0.808  0.902  0.875  0.898
 0.894  0.765  0.612  0.467  0.376  0.243     0.824  0.902  0.824  0.894  0.882
 0.824  0.804  0.788  0.741  0.541  0.294     0.89   0.882  0.918  0.914  0.886
 0.867  0.851  0.824  0.765  0.651  0.518     0.878  0.878  0.875  0.843  0.843
 0.82   0.792  0.769  0.761  0.8    0.706  …  0.831  0.906  0.824  0.835  0.835
 0.824  0.816  0.824  0.847  0.859  0.804     0.855  0.855  0.855  0.835  0.875
 ⋮                                  ⋮     ⋱                             ⋮
 0.863  0.867  0.867  0.843  0.851  0.863  …  0.318  0.325  0.333  0.341  0.349
 0.855  0.863  0.867  0.875  0.871  0.867     0.325  0.329  0.337  0.345  0.349
 0.843  0.859  0.847  0.859  0.851  0.835     0.329  0.329  0.337  0.341  0.349
 0.863  0.867  0.843  0.859  0.863  0.863     0.341  0.325  0.325  0.341  0.353
 0.843  0.839  0.816  0.855  0.875  0.89      0.31   0.294  0.298  0.345  0.361
 0.867  0.875  0.851  0.855  0.867  0.882  …  0.298  0.306  0.325  0.357  0.365
```

```
get_reds(philip_head)
```

> **Got it!**
>
> Awesome!

> **Hint**

This exercise can be quite difficult if you use a `for` loop or list comprehension.

Instead, you should use the dot syntax to apply a function element-wise to an array. For example, this is how you get the square root of `3`:

```
sqrt.(3)
```

and this is how you get the square roots of `1`, `2` and `3`:

```
sqrt.([1, 2, 3])
```

Great! By extracting the red channel value of each pixel, we get a 2D array of numbers. We went from an image (2D array of RGB colors) to a matrix (2D array of numbers).

Exercise 2.3

Let's try to visualize this matrix. Right now, it is displayed in text form, but because the image is quite large, most rows and columns don't fit on the screen. Instead, a better way to visualize it is to **view a number matrix as an image**.

This is easier than you might think! We just want to map each number to an `RGB` object, and the result will be a 2D array of `RGB` objects, which Julia will display as an image.

First, let's define a function that turns a *number* into a *color*.

```
value_as_color (generic function with 1 method)
```

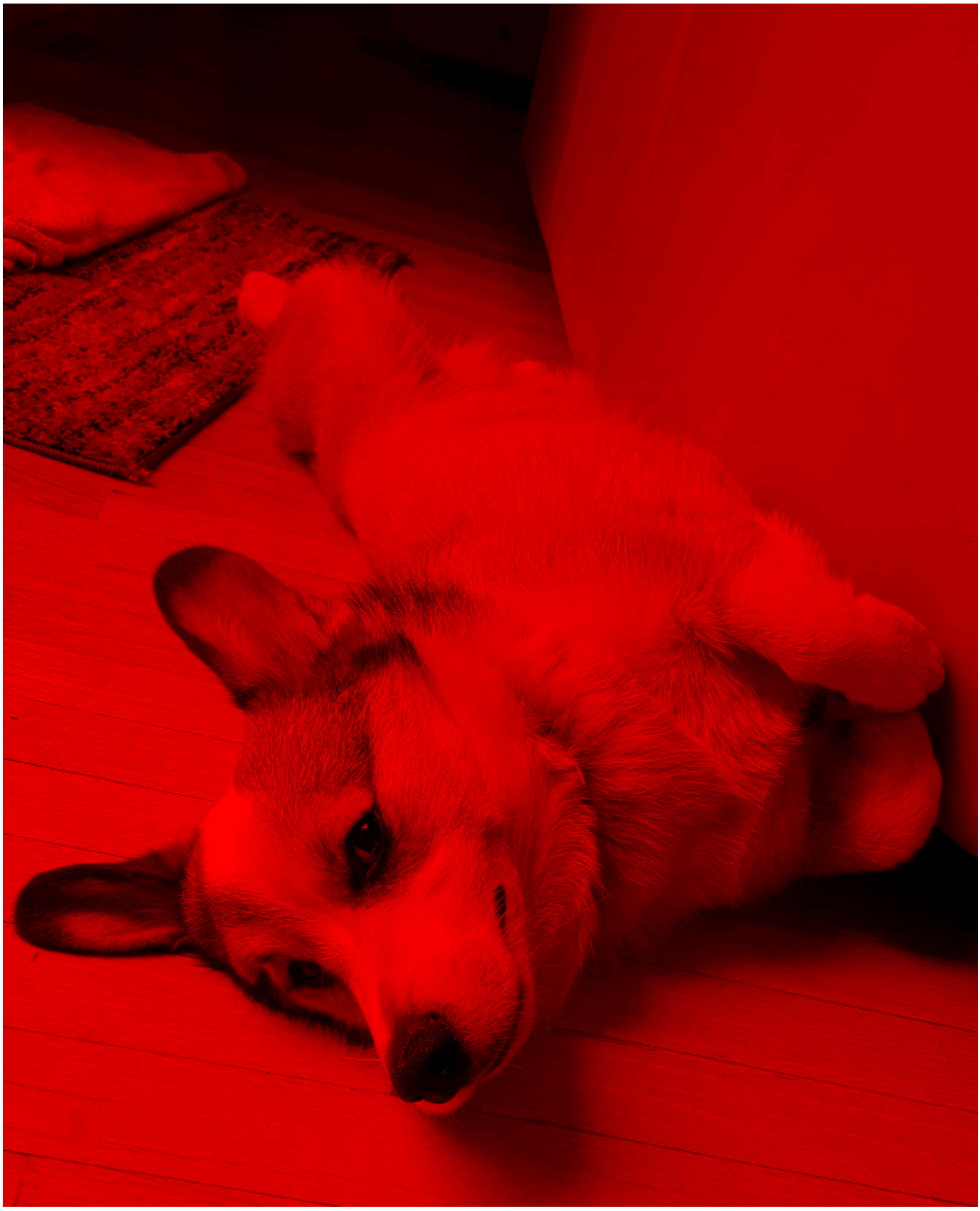
```
function value_as_color(x)

    return RGB(x, 0, 0)
end
```

```
value_as_color(0.0), value_as_color(0.6), value_as_color(1.0)
```

👉 Use the functions `get_reds` and `value_as_color` to visualize the red channel values of `philip_head`. Tip: Like in previous exercises, use broadcasting ('dot syntax') to apply a function *element-wise*.

Use the ➕ button at the bottom left of this cell to add more cells.



```
begin
    function get_red_values(image)
        return value_as_color.(get_reds(image))
    end
    get_red_values(philip)
end
```

Exercise 2.4

👉 Write four more functions, `get_green`, `get_greens`, `get_blue` and `get_blues`, to be the equivalents of `get_red` and `get_reds`. Use the ➕ button at the bottom left of this cell to add new cells.

`get_green (generic function with 1 method)`

```
function get_green(pixel)
    return pixel.g
end
```

`get_greens (generic function with 1 method)`

```
function get_greens(image)
    return get_green.(image)
end
```

`get_blue (generic function with 1 method)`

```
function get_blue(pixel)
    return pixel.b
end
```

`get_blues (generic function with 1 method)`

```
function get_blues(image)
    return get_blue.(image)
end
```

Exercise 2.5

👉 Write a function **`mean_color`** that accepts an object called `image`. It should calculate the mean amounts of red, green and blue in the image and return the average color. Be sure to use functions from previous exercises!

`mean_color (generic function with 1 method)`

```
function mean_color(image)
    # your code here!
    return RGB(mean(get_reds(image)),mean(get_greens(image)),mean(get_blues(image)))
end
```

```
mean_color(philip)
```

Got it!

You got the right answer!

At the end of this homework, you can see all of your filters applied to your webcam image!

Exercise 3 - *More filters*

In the previous exercises, we learned how to use Julia's *dot syntax* to apply a function *element-wise* to an array. In this exercise, we will use this to write more image filters, that you can then apply to your own webcam image!

Exercise 3.1

👉 Write a function `invert` that inverts a color, i.e. sends (r, g, b) to $(1-r, 1-g, 1-b)$.

```
invert (generic function with 1 method)
```

```
function invert(color::AbstractRGB)
    # your code here!
    return RGB(1-color.r,1-color.g,1-color.b)
end
```

> **Got it!**
>
> Well done!

Let's invert some colors:

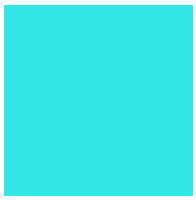
`color_black =`

```
color_black = RGB(0.0, 0.0, 0.0)
```

```
invert(color_black)
```

color_red =

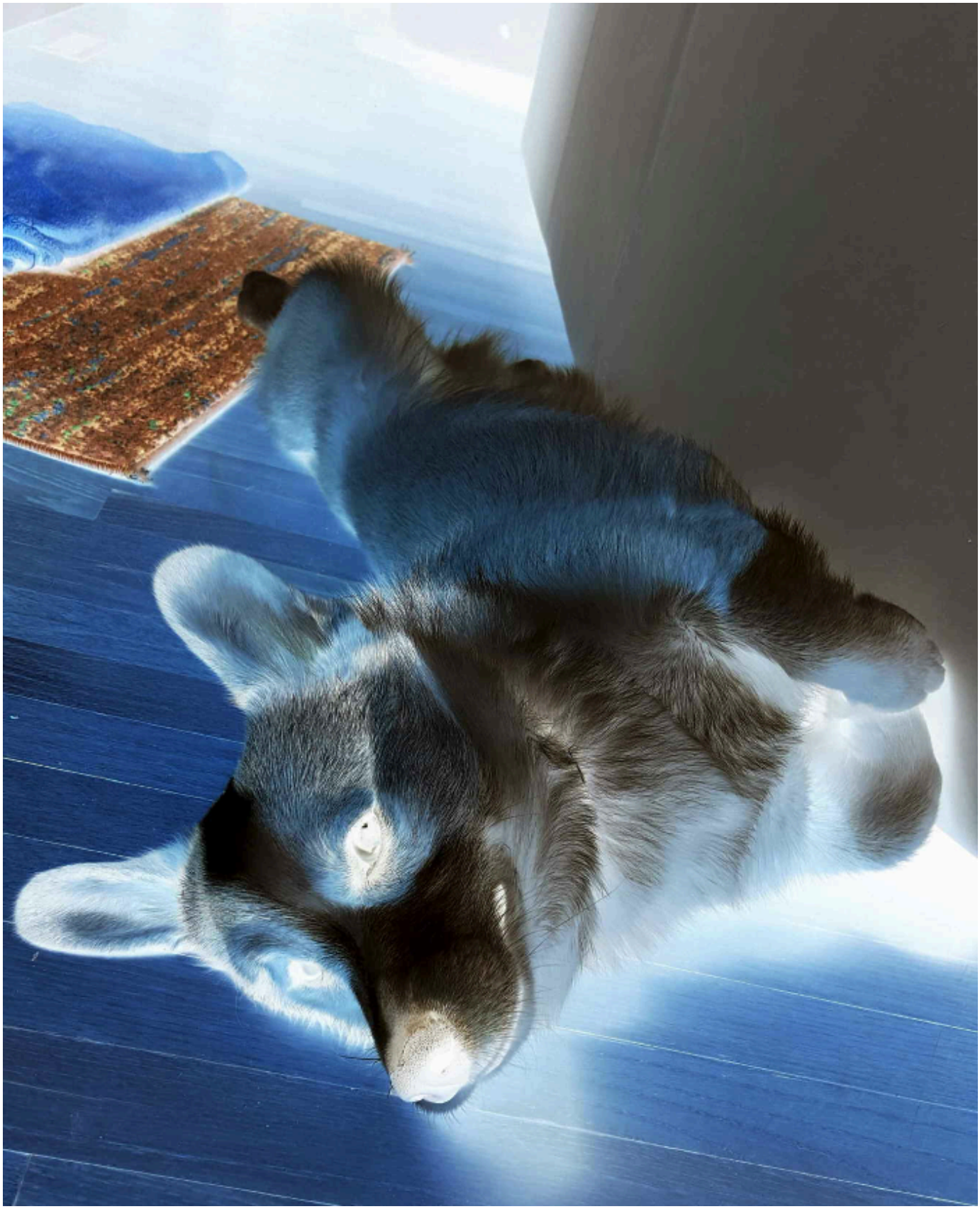
```
color_red = RGB(0.8, 0.1, 0.1)
```



```
invert(color_red)
```

👉 Can you invert the picture of Philip?

philip_inverted =



```
1 philip_inverted = invert.(philip) # replace `missing` with your code!
```

At the end of this homework, you can see all of your filters applied to your webcam image!

Exercise 3.2

👉 Look up the documentation on the `floor` function. Use it to write a function `quantize(x::Number)` that takes in a value x (which you can assume is between 0 and 1) and

"quantizes" it into bins of width 0.1. For example, check that 0.267 gets mapped to 0.2.

```
quantize (generic function with 3 methods)
```

```
function quantize(x::Number)
    # your code here!
    return (floor(10*x))/10
end
```

```
(0.2, 0.9)
```

```
quantize(0.267), quantize(0.91)
```

Got it!

Well done!

Intermezzo: *multiple methods*

In Julia, we often write multiple methods for the same function. When a function is called, the method is chosen based on the input arguments. Let's look at an example:

These are two *methods* to the same function, because they have

> **the same name, but different input types**

```
double (generic function with 2 methods)
```

```
function double(x::Number)

    return x * 2
end
```

```
double (generic function with 2 methods)
```

```
function double(x::Vector)

    return [x..., x...]
end
```

When we call the function `double`, Julia will decide which method to call based on the given input argument!

```
48
```

```
double(24)
```

```
[1, 2, 37, 1, 2, 37]
```

```
double([1,2,37])
```

We call this **multiple dispatch**, and it is one of Julia's key features. Throughout this course, you will see lots of real-world application, and learn to use multiple dispatch to create flexible and readable abstractions!

Exercise 3.3

👉 Write the second **method** of the function `quantize`, i.e. a new *version* of the function with the *same* name. This method will accept a color object called `color`, of the type `AbstractRGB`.

Here, `::AbstractRGB` is a **type annotation**. This ensures that this version of the function will be chosen when passing in an object whose type is a **subtype** of the `AbstractRGB` abstract type. For example, both the `RGB` and `RGBX` types satisfy this.

The method you write should return a new `RGB` object, in which each component (r, g and b) are quantized. Use your previous method for `quantize`!

```
quantize (generic function with 3 methods)
```

```
function quantize(color::AbstractRGB)
    # your code here!
    return RGB(quantize(color.r),quantize(color.g),quantize(color.b))
end
```

Got it!

You got the right answer!

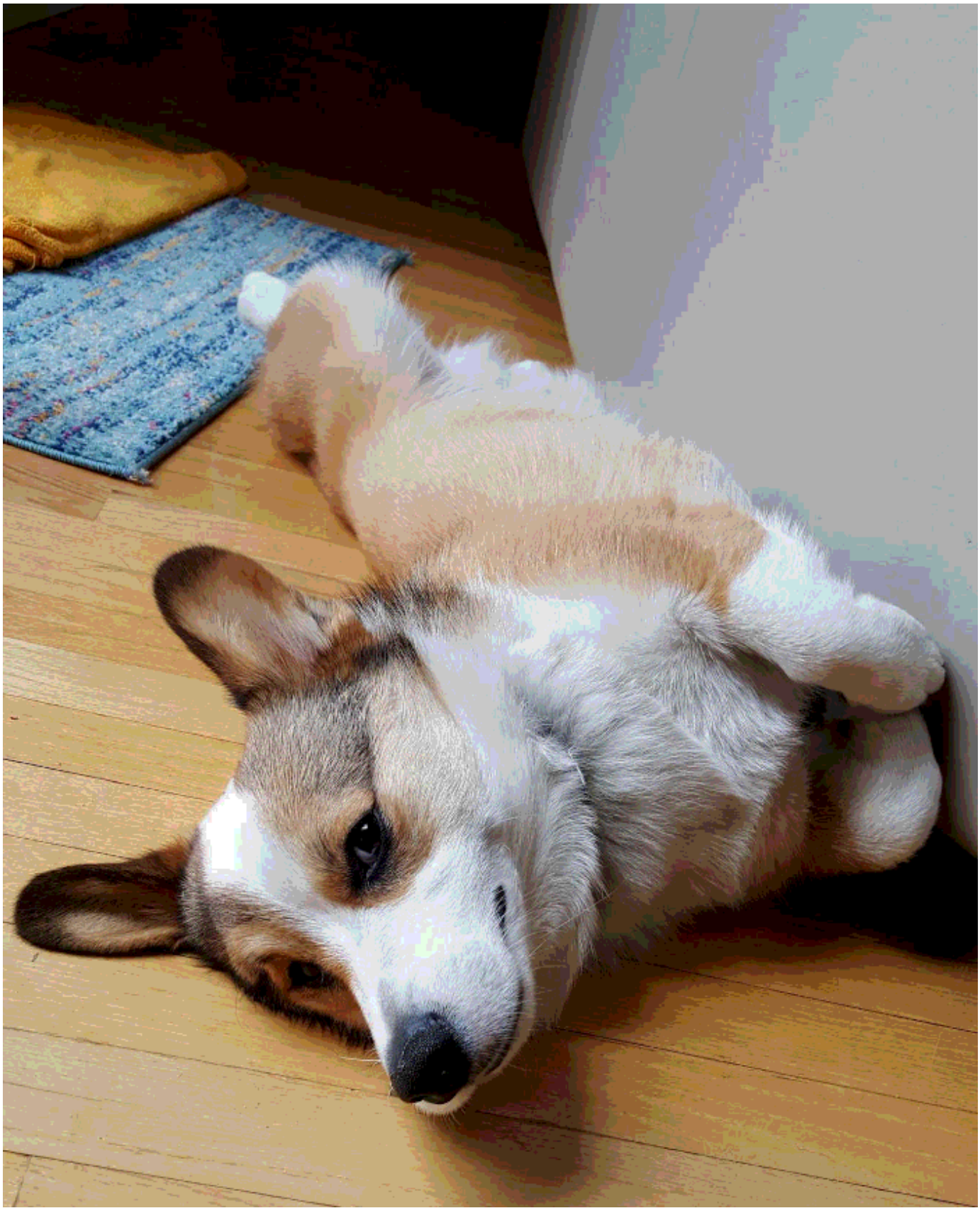
Exercise 3.4

👉 Write a method `quantize(image::AbstractMatrix)` that quantizes an image by quantizing each pixel in the image. (You may assume that the matrix is a matrix of color objects.)

```
quantize (generic function with 3 methods)
```

```
function quantize(image::AbstractMatrix)
    # your code here!
    return quantize.(image)
end
```

Let's apply your method!



```
1 quantize(philip)
```

Exercise 3.5

👉 Write a function `noisify(x::Number, s)` to add randomness of intensity s to a value x, i.e. to add a random value between $-s$ and $+s$ to x. If the result falls outside the range $[0, 1]$ you should "clamp" it to that range. (Julia has a built-in `clamp` function, or you can write your own function.)

```
noisify (generic function with 3 methods)
```

```
function noisify(x::Number, s)
    # your code here!
    noisy_value=x+(rand()-0.5)*2*s
    return clamp(noisy_value,0,1)
end
```

```
0.5525257332761524
```

```
noisify(0.5, 0.1) # edit this test case!
```

Got it!

Yay ❤

Hint

rand() generates a (uniformly) random floating point number between 0 and 1.

👉 Write the second method `noisify(c::AbstractRGB, s)` to add random noise of intensity s to each of the (r, g, b) values in a colour.

Use your previous method for `noisify`. *(Remember that Julia chooses which method to use based on the input arguments. So to call the method from the previous exercise, the first argument should be a number.)*

```
noisify (generic function with 3 methods)
```

```
function noisify(color::AbstractRGB, s)
    # your code here!
    return RGB(noisify(color.r,s),noisify(color.g,s),noisify(color.b,s))
end
```

Noise strength:

0.0

```
@bind color_noise Slider(0:0.01:1, show_value=true)
```

(original = , with_noise =)

```
(original=color_red, with_noise=noisify(color_red, color_noise))
```

Note about *array comprehension*

At this point, you already know of a few ways to make a new array based on one that already exists.

1. you can use a for loop to go through a array

2. you can use function broadcasting over a array
3. you can use ***array comprehension***!

The third option you are about to see demonstrated below and following the following syntax:

```
[function_to_apply(args) for args in some_iterable_of_your_choice]
```

This creates a new iterable that matches what you iterate through in the second part of the comprehension. Below is an example with `for` loops through two iterables that creates a 2-dimensional `Array`.

```
[
    noisify(color_red, strength)
    for
        strength in 0 : 0.05 : 1,
        row in 1:10
]'
```

👉 Write the third method `noisify(image::AbstractMatrix, s)` to noisify each pixel of an image. This function should be a single line!

```
noisify (generic function with 3 methods)
```

```
function noisify(image::AbstractMatrix, s)
    # your code here!
    return noisify.(image,s)
end
```

Exercise 3.6

Move the slider below to set the amount of noise applied to the image of Philip.

2.23

```
@bind philip_noise Slider(0:0.01:10, show_value=true)
```

```
noisify(philip_head, philip_noise)
```

👉 For which noise intensity does it become unrecognisable?

You may need noise intensities larger than 1. Why?

`answer_about_noise_intensity =`

The image is unrecognisable with intensity ...2.23

```
answer_about_noise_intensity = md"""
The image is unrecognisable with intensity ...2.23
"""
```

Camera input

Enable webcam

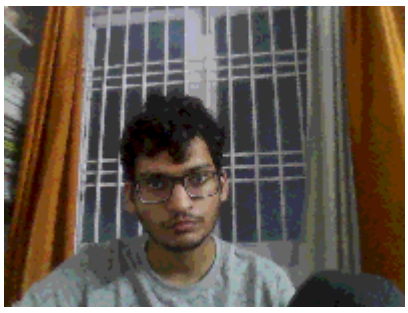
```
@bind cam_data camera_input()
```

cam_image =

Results

```
1 mean_color(cam_image)
```

```
1 invert.(cam_image)
```

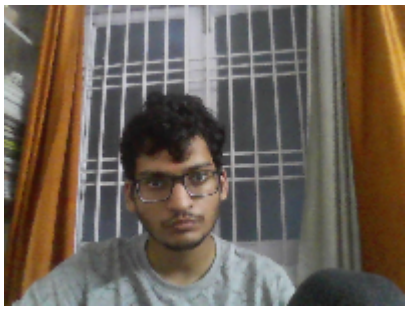


```
1 quantize(cam_image)
```

```
1 noisify(cam_image, .5)
```

```
[
    invert.(cam_image)       quantize(cam_image)
    noisify(cam_image, .5)   custom_filter(cam_image)
]
```

Write your own filter!



```
custom_filter(cam_image)
```

```
custom_filter (generic function with 1 method)
```

```
function custom_filter(pixel::AbstractRGB)

    # your code here!

    return pixel
end
```

```
custom_filter (generic function with 2 methods)
```

```
function custom_filter(image::AbstractMatrix)

    return custom_filter.(image)
end
```

Function library

Just some helper functions used in the notebook.

```
hint (generic function with 1 method)
```

```
almost (generic function with 1 method)
```

```
still_missing (generic function with 2 methods)
```

```
keep_working (generic function with 2 methods)
```

```
yays =
  [Fantastic!, Splendid!, Great!, Yay ♥, Great! 🥕, Well done!, Keep it up!, Good job!, Awesome!,
```

```
correct (generic function with 2 methods)
```

```
not_defined (generic function with 1 method)
```

```
todo (generic function with 1 method)
```

```
camera_input (generic function with 1 method)
```

```
process_raw_camera_data (generic function with 1 method)
```

homework 1, version 9