Go 101

image filters

09.10.2018

Uber Engineering Sofia

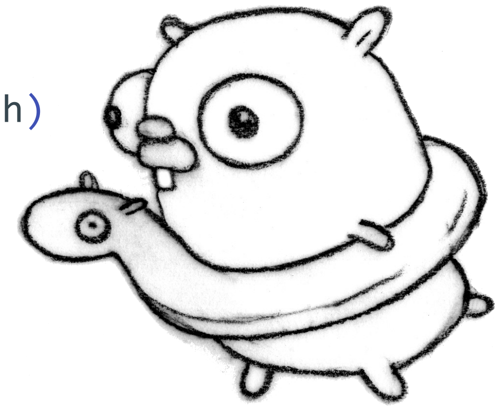
Creating our project

First create the directory for our project:

```
$ mkdir -p $GOPATH/src/go.uber.org/sofia-go101/image-kernels
```

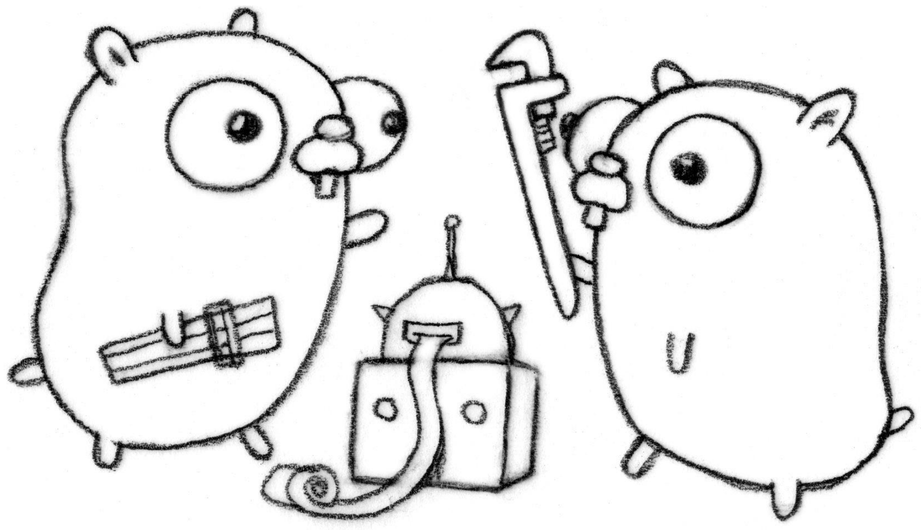
Then write our entry point in the `main` function in the file `$GOPATH/src/go.uber.org/sofia-go101/image-kernels/main.go`

```
func main() {
    filePath := os.Args[1]
    fmt.Printf("Attempting to read image from %s\n", filePath)
    return
}
```



Creating our project

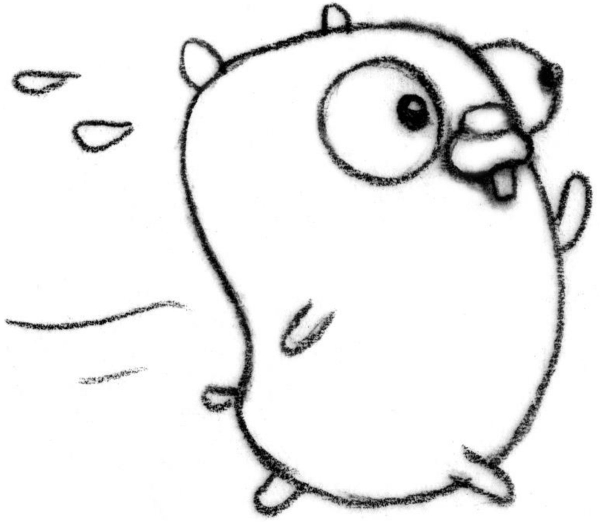
```go
package main

import (
    "fmt"
    "os"
)

func main() {
    filePath := os.Args[1]
    fmt.Printf("Attempting to read image from %s\n", filePath)
    return
}
```



*you can follow what we do in:
https://github.com/kunev/coding-girls-image-filters

Compiling and running our code

```
$ cd $GOPATH/go.uber.org/sofia-go101/image-kernels
$ go build
$ ./image-kernels not-an-actual-image-file.png
Attempting to read image from not-an-actual-image-file.png
```



It’s your turn from here on

We’ll give guidance, help out and make sure everyone manages to keep up with the same pace*. However from now on it’s all about you writing the code yourself.

*you can follow what we do in:
https://github.com/kunev/coding-girls-image-filters

Reading the image

We want to load the actual image data from a file. Use any image you want to play around with, or have some gophers.

Put the image file in the same directory as the `main.go` file and name it `input.jpg` or `input.png` depending on what the original format is (you can obviously name it whatever you want, but our examples onwards will assume it's called `input.jpg`/`input.png`).

Reading the image

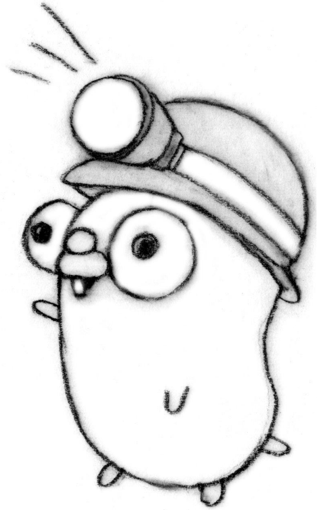
** you can import a package simply for its side effects using the following syntax:*

```
import (
        _ "some/package"
)
```

Using the built-in `os` and `image` packages load the image data in your code and print the *type** of the image that we're loading.

You can find documentations for the packages here:

- https://godoc.org/image
- https://godoc.org/os



You will need to use the `os.Open` and `image.Decode` functions.

You can run your code at any time by executing the following command from the shell:

```
$ go build && ./image-kernels input.jpg
```

Reading the image

The original Go gopher © 2009 Renée French. Used under Creative Commons Attribution 3.0 license. Illustration © 2016 Olga Shalakhina.

Here's how we can achieve this:

```
imageFile, err := os.Open(filePath)  // open the file
if err != nil {  // check if something went wrong opening the file
    log.Fatal(err)  // print the error and exit if anything broke
}

_, format, err := image.Decode(imageFile)  // try decoding the image data
if err != nil {  // check for error when decoding image data
    log.Fatal(err)  // print error and exit if anything went wrong
}

fmt.Printf("Read a %s image \n", format)  // print what the image format is
```

Reading the image

The original Go gopher © 2009 Renée French. Used under Creative Commons Attribution 3.0 license. Illustration © 2016 Olga Shalakhina.

Which makes our main function look like so:

```go
func main() {
    filePath := os.Args[1]
    fmt.Printf("Attempting to read image from %s\n", filePath)
    imageFile, err := os.Open(filePath)
    if err != nil {
        log.Fatal(err)
    }
    _, format, err := image.Decode(imageFile)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Read a %s image \n", format)
    return
}
```

Reading the image

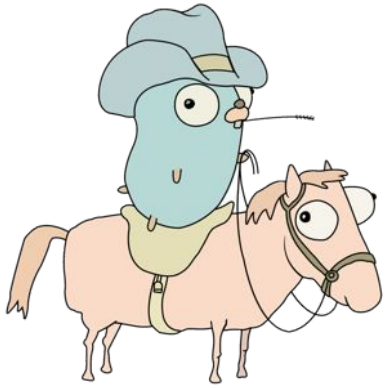
The import section at the top should now look like this:

```
import (
    "fmt"
    "image"
    "log"
    "os"

    _ "image/jpeg"  // this allows image.Decode to recognize jpeg files
    _ "image/png"  // this allows image.Decode to recognize png files
)
```

Print the image dimensions

Using whatever's available in the `Image` struct you get from the call to `image.Decode`, print the width and height of the image.

Remember, godoc.org is awesome and it's your best friend while writing go code!



hint: the `Size` of an image is something you can get from its `Bounds`

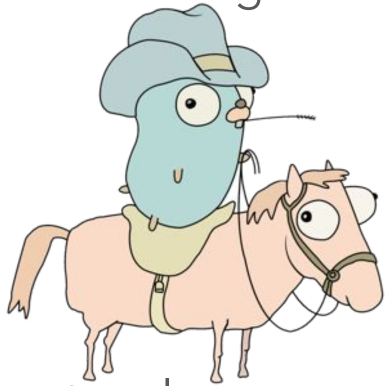
Print the image dimensions

First we'll actually use the image data we decode from the file, so we should give it a name. Change this line:

```
_, format, err := image.Decode(imageFile)
```

to

```
imageData, format, err := image.Decode(imageFile)
```

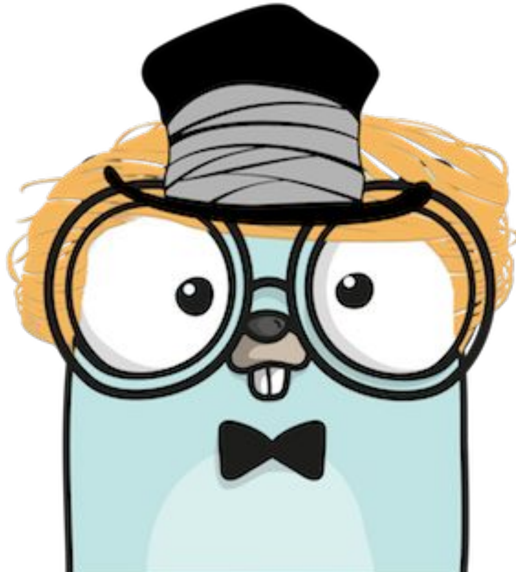


Then we can use the Bounds method of the Image struct to get a rectangle representing the image dimensions and we can call its Size method to get the width and height:

```
fmt.Printf("The size of the image is %s\n", imageData.Bounds().Size())
```

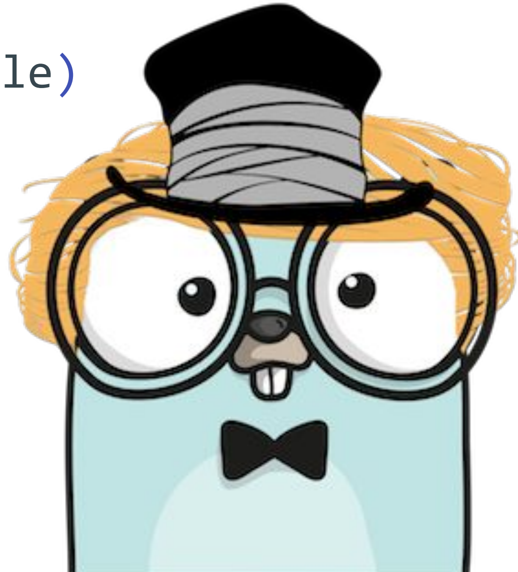
Extract reading in a function

At this point our main function is starting to get a bit big. We can make our lives easier by abstracting the opening of the file and reading of the image away in a function. Let’s call it `loadImage` and have it take the filename as an argument and return the image data, image format and an error.



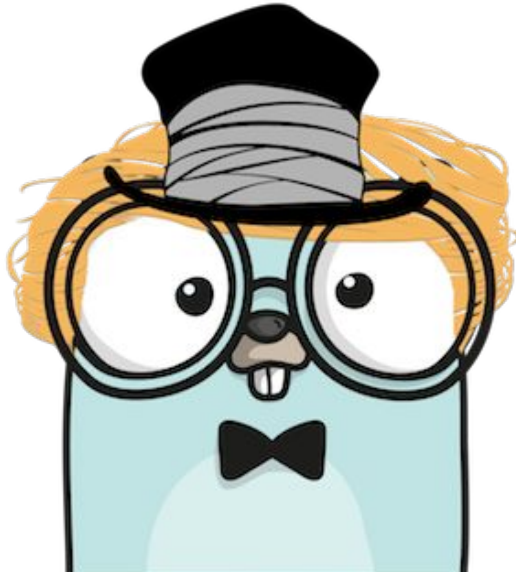
Extract the image reading in a function

```go
func loadImage(filePath string) (image.Image, string, error) {
    imageFile, err := os.Open(filePath)
    if err != nil {
        return nil, "", err
    }
    imageData, format, err := image.Decode(imageFile)
    if err != nil {
        return nil, "", err
    }
    return imageData, format, nil
}
```



Extract the image reading in a function

```go
func main() {
    filePath := os.Args[1]
    fmt.Printf("Attempting to read image from %s\n", filePath)
    imageData, format, err := loadImage(filePath)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Read a %s image \n", format)
    fmt.Printf("The size of the image is %s\n",
                imageData.Bounds().Size())
    return
}
```



Saving a copy of the image

As we will be making changes to the image we want to save it to another file. It's easier to have that set up before we start doing any actual processing, so we can see our results right away once we get there.

Let's just save a copy of our image. The `image/png` and `image/jpeg` packages both export an `Encode` function, which can be used to write image data to a file. We can use this to

We can use the `switch` construct to decide which one to use.

Now that we will use the `image/png` and `image/jpeg` packages we should remove the blank imports.

Saving a copy of the image

```go
writer, err := os.Create(fmt.Sprintf("output.%s", format))
if err != nil {
    log.Fatal(err)
}

switch format {
case "jpeg":
    jpeg.Encode(writer, imageData, nil)
case "png":
    png.Encode(writer, imageData)
}
```

Saving a copy of the image

It's a good idea to make sure we always handle errors though! We don't want things breaking silently and leaving us unaware of problems. Both Encode functions return errors we should handle.

```go
switch format {
case "jpeg":
    err := jpeg.Encode(writer, imageData, nil)
case "png":
    err := png.Encode(writer, imageData)
}

if err != nil {
    log.Fatal(err)
}
```

Saving a copy of the image

We can move the writing to a function as well:

```go
func writeImage(imageData image.Image, format string) error {
    writer, err := os.Create(fmt.Sprintf("output.%s", format))
    if err != nil {
        log.Fatal(err)
    }

    switch format {
    case "jpeg":
        return jpeg.Encode(writer, imageData, nil)
    case "png":
        return png.Encode(writer, imageData)
    default:
        return errors.New("Unknown format")
    }
}
```

Saving a copy of the image

So our `main` function becomes even easier to read:

```go
func main() {
    filePath := os.Args[1]
    fmt.Printf("Attempting to read image from %s\n", filePath)
    imageData, format, err := loadImage(filePath)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Read a %s image \n", format)
    fmt.Printf("The size of the image is %s\n", imageData.Bounds().Size())

    if err := writeImage(imageData, format); err != nil {
        log.Fatal(err)
    }
    return
}
```

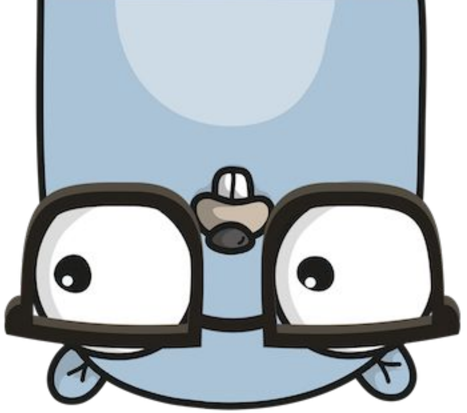


Image kernels

Image kernels are a simple way to do image processing. It a boils down to changing the value of a pixel based on the values of its neighbours and itself.

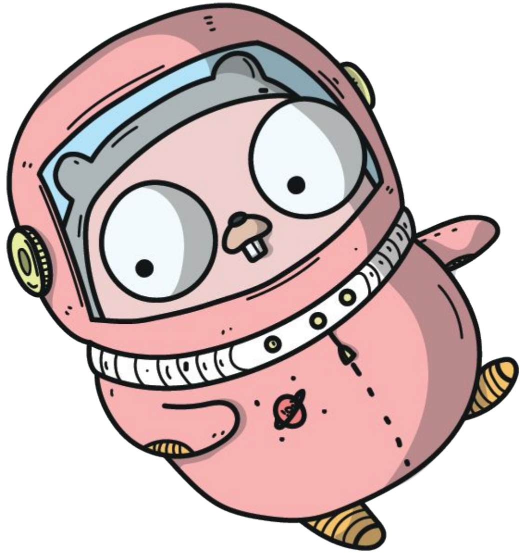
We'll use kernels to do the image processing. So let's start by creating a type for our kernels. We'll keep that in a separate `kernel` subpackage of our project. For that we need to create a directory called kernel and add a file to it (say `kernel.go`) which belongs to the `kernel` package.

source: https://twitter.com/noahhlo/status/437395572081688576

Image kernels

We’ll start off by creating just a Kernel type, then move on to implementing a `New` function for it and some methods.

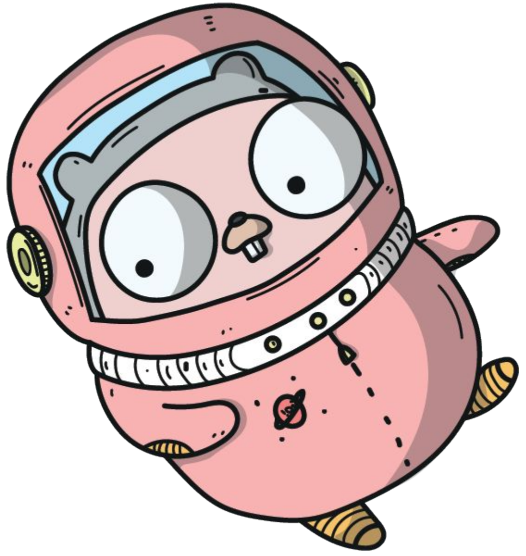
So our kernel/kernel.go file should look like this:



```
package kernel

// Kernel describes an image
kernel
type Kernel struct {
    Width        int
    Height       int
    Coefficients [][]float32
}
```

Image kernels

Now we want to have a New function that returns a kernel struct based on on `[][]float32` argument.

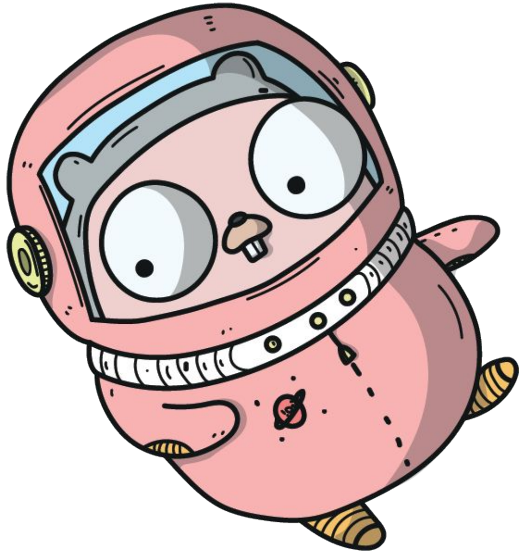


hint: you can get the length of a slice by calling the built-in `len` function

Image kernels

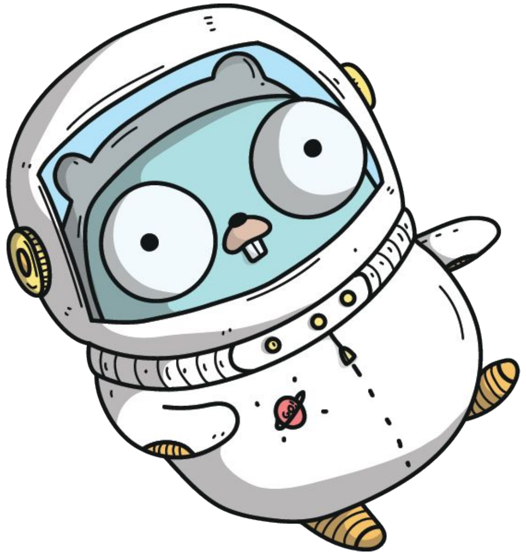
Here’s how that would look:

```
// New returns a Kernel wrapping the given
coefficients matrix
func New(coefficients [][]float32) Kernel {
    result := Kernel{}
    result.Height = len(coefficients)
    result.Width = len(coefficients[0])
    result.Coefficients = coefficients

    return result
}
```



Constructing an image kernel

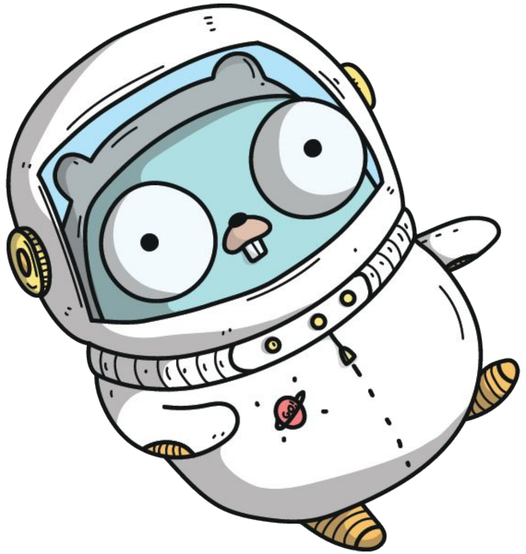
Let's try creating a kernel with our constructor to see what it looks like. We'll just instantiate it and pretty print it to see what our `New` function creates.



```go
k := kernel.New([][]float32{
    {0, 0, 0},
    {0, 1, 0},
    {0, 0, 0},
})
fmt.Printf("%#v", k)
```

Constructing an image kernel

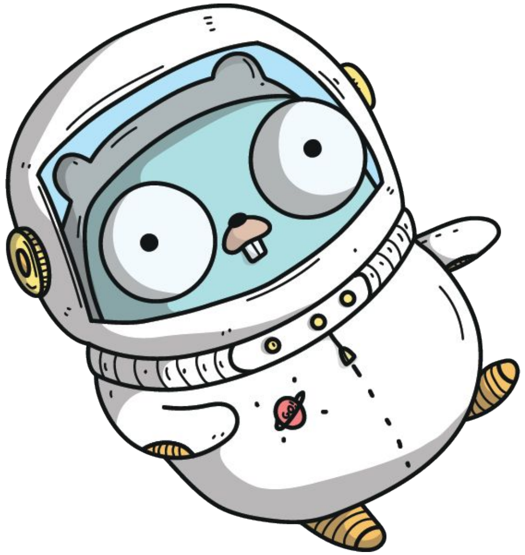
Then running our code should look like this:



```
$ go build && ./image-kernels input.jpg
Attempting to read image from input.jpg

Read a jpeg image

The size of the image is (1999,1324)

kernel.Kernel{Width:3, Height:3,
Coefficients:[][]float32{[]float32{0, 0, 0},
[]float32{0, 1, 0}, []float32{0, 0, 0}}}
```

Testing our image kernel

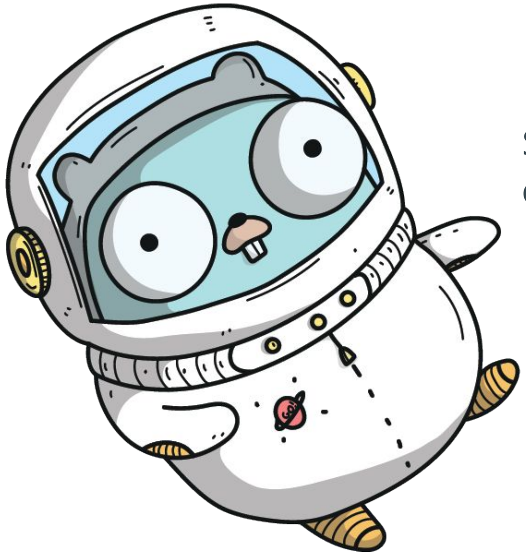
We can write a test for our kernel package in a kernel/kernel_test.go file that would look like this:



```go
package kernel

import "testing"

func TestNewKernel(t *testing.T) {
    kernel := New([][]float32{
        {0, 0, 0},
        {0, 1, 0},
        {0, 0, 0},
    })
    if kernel.Width != 3 {
        t.Fatal("Width is wrong")
    }
    if kernel.Height != 3 {
        t.Fatal("Height is wrong")
    }
}
```

Testing our image kernel

Running our tests is quite simple:



```
$ go test go.uber.org/sofia-go101/image-kernels/kernel
ok   go.uber.org/sofia-go101/image-kernels/kernel  0.001s
```

Applying the kernel

We're getting to the gist of it. We have a loaded image and we have created a kernel. Now we want to apply the kernel to the image.

What we want is to iterate over all the pixels of an image and do something for each of them. Similar to how we just copy and pasted the file using the image data we got, let's just initially make our kernel do nothing by giving it an `Apply` method that iterates over the pixels of the image data and simply copies them to another `Image` struct it returns.

You can use the `Bounds()` method again to get the parameters for iterating over the pixels of the image, then use `At()` and `Set()` to copy the pixels.

Applying the kernel

Here’s how we do this:

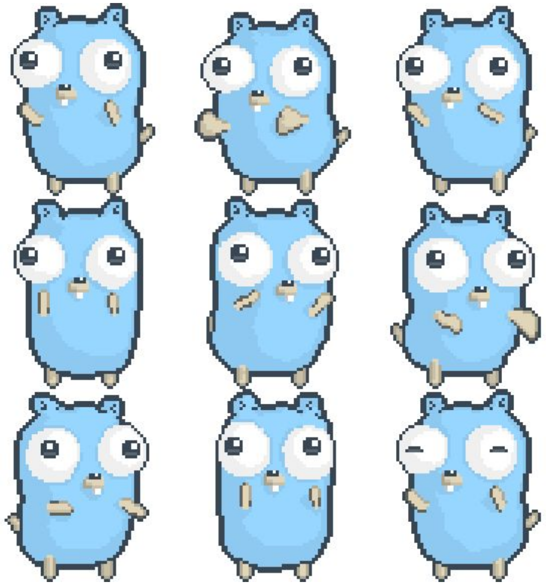
```go
// Apply applies a kernel to an image returning the resulting image
func (k Kernel) Apply(img image.Image) (image.Image, error) {
    imageBounds := img.Bounds()
    result := image.NewRGBA(imageBounds)

    for x := imageBounds.Min.X; x < imageBounds.Max.X; x++ {
        for y := imageBounds.Min.Y; y < imageBounds.Max.Y; y++ {
            result.Set(x, y, img.At(x, y))
        }
    }

    return result, nil
}
```

Getting a pixel’s neighbourhood

We compute the new value of a pixel with our kernel based on the values of its neighbouring pixels, so we need a way to get all of the valid neighbour pixels for a given pixel in the image.

We want a function that will give us the neighbourhood Width×Height pixels for a given pixel.

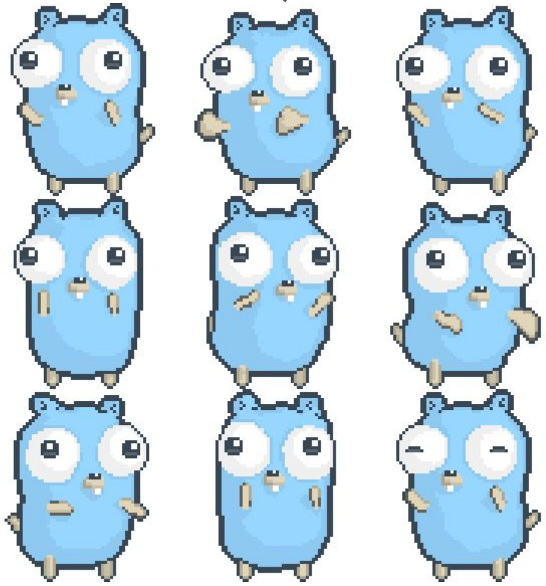
If each gopher on the right is a pixel, then the 3×3 neighbourhood of the middle one would be all nine gophers.



Getting a pixel's neighbourhood

We'll need to know exactly where a neighbour is in relation to the central pixel we are calculating a value for. For that purpose we'll create a unexported struct type that will hold a pixel value (a color.Color struct) and X and Y offsets.

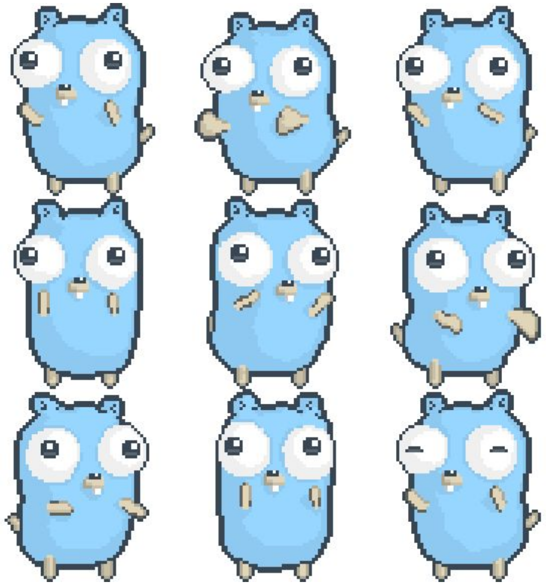
```go
type neighbour struct {
	xOffset int
	yOffset int
	clr     color.Color
}
```



Getting a pixel’s neighbours

Our function’s signature should be as follows:

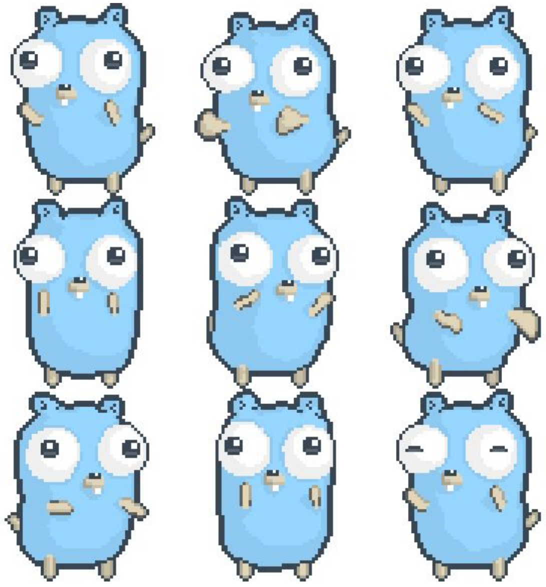
```
func (k Kernel) getNeighbourhood(x, y int, img image.Image) []neighbour
```



Getting a pixel’s neighbours

And this is how we can implement it:

```go
func (k Kernel) getNeighbourhood(x, y int, img image.Image) []neighbour {
    bounds := img.Bounds()
    neighbourhood := []neighbour{}
    for i := -k.Width / 2; i <= k.Width/2; i++ {
        if x+i < bounds.Min.X || x+i > bounds.Max.X {
            continue
        }
        for j := -k.Height / 2; j <= k.Height/2; j++ {
            if y+j < bounds.Min.Y || y+j > bounds.Max.Y {
                continue
            }
            neighbourhood = append(neighbourhood, neighbour{
                xOffset: i,
                yOffset: j,
                clr:     img.At(x+i, y+j),
            })
        }
    }
    return neighbourhood
}
```



Applying the kernel for a pixel

Every Color struct has four components to it: red, green, blue and alpha (the R, G, B, and A attributes of the struct). We want to apply the kernel on each color layer of our image and leave the alpha channel unchanged.

Knowing the offset of a neighbour we want to add the product of each of its color values to the respective color value of a new “empty” result image that we will return.

hint: all types in Go have default values, for numeric types that’s 0

Applying the kernel on a pixel

Our function signature should look something like this:

```go
func (k Kernel) pixelValueFromNeighbourhood(neighbourhood []neighbour) color.Color
```

hint: all types in Go have default values, for numeric types that’s 0

Applying the kernel on a pixel

The color.Color type has an RGBA method that returns four values. You can ignore one or several of the returned values by assigning them to the blank identifier _ (underscore).

You can cast a variable of one type to another by using the type you want to cast to as a function you call on the variable.

```go
var b float = 3.0
var a int = int(b)
```

Applying the kernel on a pixel

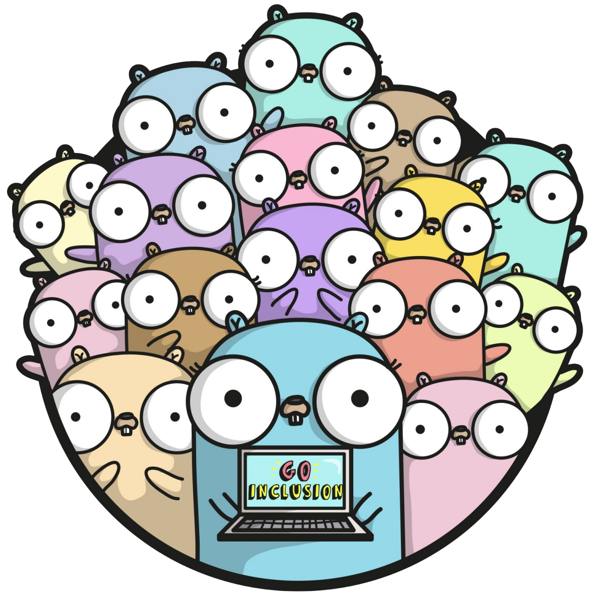
Here’s how all this looks:

```go
func (k Kernel) pixelValueFromNeighbourhood(neighbourhood []neighbour)
color.Color {
	result := color.RGBA64{}
	for _, n := range neighbourhood {
		coef := k.Coefficients[n.xOffset+k.Width/2][n.yOffset+k.Height/2]
		r, g, b, a := n.clr.RGBA()
		result.R += uint16(float32(r) * coef)
		result.G += uint16(float32(g) * coef)
		result.B += uint16(float32(b) * coef)
		result.A = uint16(a)
	}
	return result
}
```

Tying it together

Now that we've done pretty much everything we just need the final touch of actually making the computation for each pixel of the image.

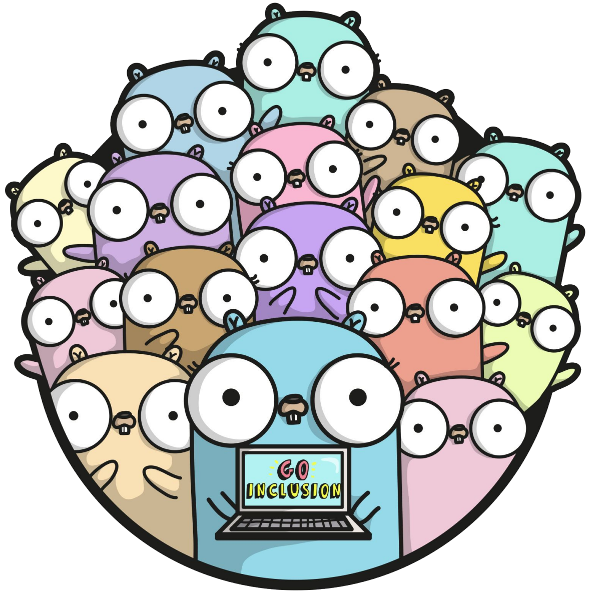
We have the loop set up in the `Apply` method already, we just need to use it to set the values of the pixels in our result image to be the output of the kernel application on the corresponding pixel's neighbourhood in the original image.



Tying it together

All that really is is simply:

```
neighbourhood := k.getNeighbourhood(x, y, img)
result.Set(x, y, k.pixelValueFromNeighbourhood(neighbourhood))
```



Tying it together

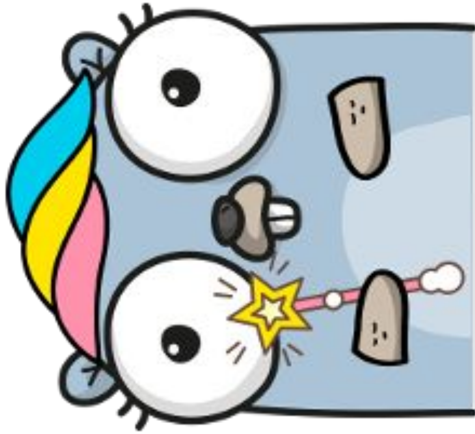
So now Apply looks like this:

```go
// Apply applies a kernel to an image returning the resulting image
func (k Kernel) Apply(img image.Image) (image.Image, error) {
    imageBounds := img.Bounds()
    result := image.NewRGBA(imageBounds)

    for x := imageBounds.Min.X; x < imageBounds.Max.X; x++ {
        for y := imageBounds.Min.Y; y < imageBounds.Max.Y; y++ {
            neighbourhood := k.getNeighbourhood(x, y, img)
            result.Set(x, y, k.pixelValueFromNeighbourhood(neighbourhood))
        }
    }

    return result, nil
}
```

Saving the result to the output file

All that's left is to call `Apply` from our `main` function and save the result:

```
k := kernel.New([][]float32{
    {0.1, 0.1, 0.1},
    {0.1, 0.1, 0.1},
    {0.1, 0.1, 0.1},
})
resultImage, _ := k.Apply(imageData)

if err := writeImage(resultImage, format); err != nil {
    log.Fatal(err)
}
```



Running our code

All that’s left is to call `Apply` from our `main` function and save the result:

```
$ cd $GOPATH/go.uber.org/sofia-go101/image-kernels
$ go build
$ ./image-kernels input.jpg
```