

Creative Coding 4

David Lynch

Last Semester

- Basic I/O using `print()` and `input()`
- Boolean logic and conditional statements
- Conditional looping using `while()` and `for()`
- An intro to *Processing*
- 2D primitives in *Processing* (Shapes, lines, etc.)
- Basic maths functions in *Processing*.

This Semester

- Functions
- Classes
- Functional Programming
- Object-Oriented Programming
- Intro to Algorithms and Data Structures
- How Python Works
- Code Optimisation Tips and Tricks
- Further exploring *Processing*
- 3D Graphics in Processing
- File I/O
- Good Coding Practices
- Other Tips and Tricks

Today's Class

- What a function is
- How to call a function
- How to define functions
- The *Call Stack*
- Lambda Expressions
- Implementing *map* and *filter*

A Recap Exercise

Implement an Analogue Clock in *Processing*

Steps:

- Set up a *Processing* environment
- Print out current time (HH:MM:SS)
- Dynamically represent this time
- Make it look like an analogue clock
- Style it as you want!

What are functions?

What are functions?

Variables are symbolic representations of data. We can store data in variables, and modify it without having to manually type it out each time.

Functions are like variables, but it stores instructions so we don't have to constantly re-write segments of code.

What are functions?

Functions are 'first class' data types in Python. This means they share a lot of properties with variables like:

- Being written to
- Being read into a single variable
- Stored in lists and classes
- Passed to functions

This allows us to draw a lot of utility from Python's own functions, and our own.

Calling Functions in Python

Python follows a C-like convention for calling functions.
You can call functions like this:

```
funcname(arg1, arg2, arg3, ...)
```

e.g. `print("Hello CS++")`, `input("How are you?")`

Functions also return data which replaces the function call at that location with some data.

e.g. `x = sum(4)` \rightarrow `x = 10`

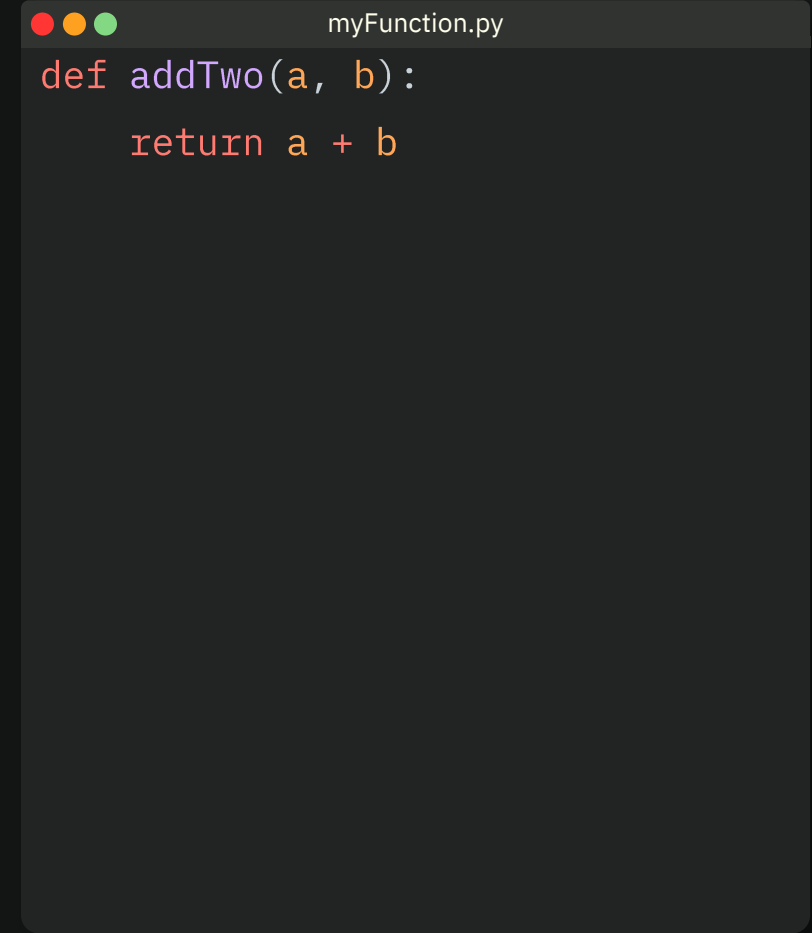
Function Definition

We define functions with the `def` keyword

- In *Processing*, we define the `settings`, `setup`, and `draw` functions using this same syntax.

For example, on the right is a function which adds two numbers together.

Notice that the body of the function is indented relative to the definition

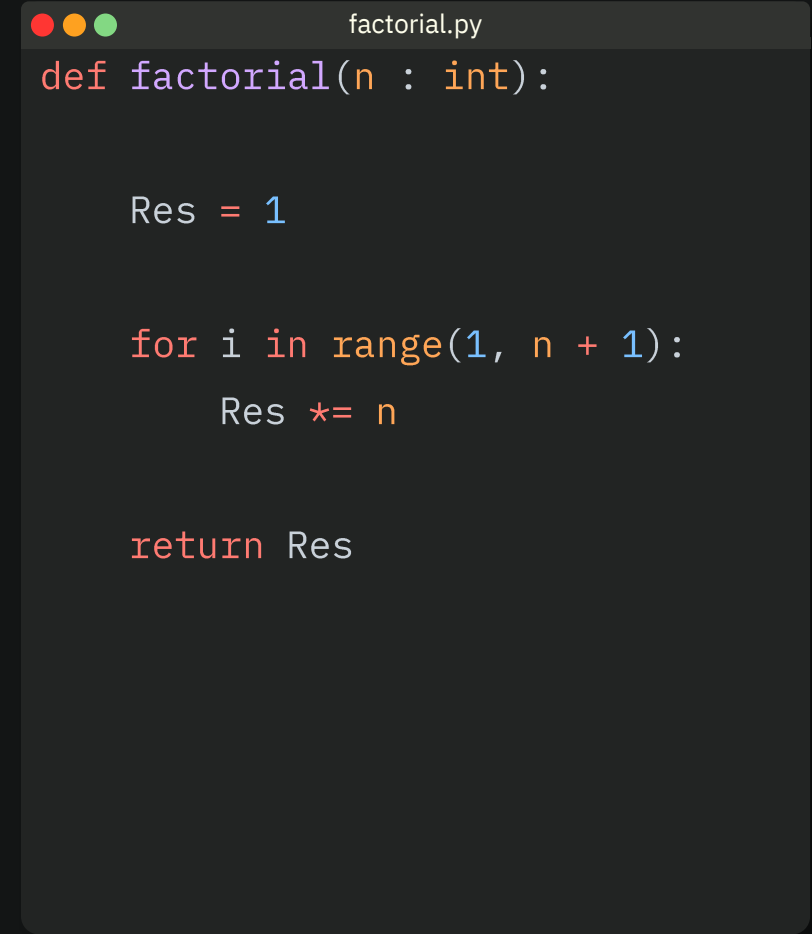


```
def addTwo(a, b):  
    return a + b
```

Type Hinting

Since Python is a dynamic language, it only checks types at runtime.

To help document our programs, we can give the function arguments a type. Some Python implementations enforce type hinting.



```
def factorial(n : int):  
  
    Res = 1  
  
    for i in range(1, n + 1):  
        Res *= i  
  
    return Res
```

The Call Stack

The Call Stack

The Call Stack is a data structure that computers use to keep track of the control flow of their programs.

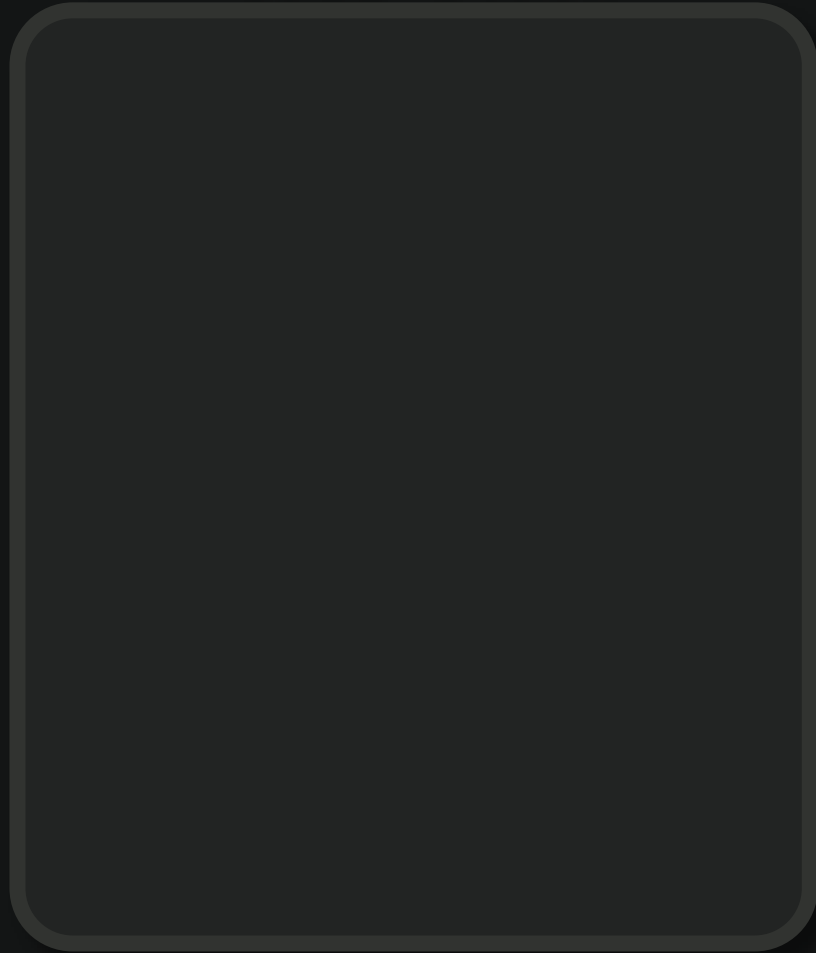
When you call a function in Python, certain things are ‘pushed’ onto the Call Stack.

- The function’s arguments
- The address of the next instruction after the function call
- The variables local to the function

We can see the Call Stack in action when we write code in Python

Call Stack Example

Call Stack



```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```



Call Stack Example

Call Stack

__main__

```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

subTwo(1, 3)

__main__

Terminal

>

```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```


Call Stack Example

Call Stack

subTwo(1, 3)

__main__

Terminal

>

```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

print("In sub function", a, b)

subTwo(1, 3)

__main__

Terminal

> In sub function 1 3

```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

mulTwo(a, b)

subTwo(1, 3)

__main__

Terminal

> In sub function 1 3

sayHello.py

```
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

mulTwo(a, b)

subTwo(1, 3)

__main__

Terminal

> In sub function 1 3

```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

print("In mul function", a, b)

mulTwo(a, b)

subTwo(1, 3)

__main__



Terminal

```
In sub function 1 3
> In mul function 1 3
```

sayHello.py

```
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

addTwo(a, b)

mulTwo(a, b)

subTwo(1, 3)

__main__

Terminal

```
In sub function 1 3  
> In mul function 1 3
```

```
sayHello.py  
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

addTwo(a, b)

mulTwo(a, b)

subTwo(1, 3)

__main__

Terminal

```
In sub function 1 3  
> In mul function 1 3
```

sayHello.py

```
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

print(In add function a, b)

addTwo(a, b)

mulTwo(a, b)

subTwo(1, 3)

__main__

Terminal

```
In sub function 1 3
In mul function 1 3
> In add function 1 3
```

```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```


Call Stack Example

Call Stack

addTwo(a, b)

mulTwo(a, b)

subTwo(1, 3)

__main__



Terminal

```
In sub function 1 3
In mul function 1 3
> In add function 1 3
```

sayHello.py

```
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

mulTwo(a, b)

subTwo(1, 3)

__main__

Terminal

```
In sub function 1 3
In mul function 1 3
> In add function 1 3
```

```
sayHello.py
def addTwo(a, b):
    print("In add function", a, b)
    return a + b

def mulTwo(a, b):
    print("In mul function", a, b)
    return a * b * addTwo(a, b)

def subTwo(a, b):
    print("In sub function", a, b)
    return a - b - mulTwo(a, b)

if __name__ == "__main__":
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

subTwo(1, 3)

__main__

Terminal

```
In sub function 1 3  
In mul function 1 3  
> In add function 1 3
```

sayHello.py

```
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

subTwo(1, 3)

__main__

Terminal

```
In sub function 1 3  
In mul function 1 3  
> In add function 1 3
```

```
sayHello.py  
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

print(subTwo(1, 3))

__main__

Terminal

```
In sub function 1 3  
In mul function 1 3  
In add function 1 3  
> -14
```

```
sayHello.py  
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

__main__

Terminal

```
In sub function 1 3  
In mul function 1 3  
In add function 1 3  
> -14
```

```
sayHello.py  
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Call Stack Example

Call Stack

Terminal

```
In sub function 1 3  
In mul function 1 3  
In add function 1 3  
> -14
```

sayHello.py

```
def addTwo(a, b):  
    print("In add function", a, b)  
    return a + b  
  
def mulTwo(a, b):  
    print("In mul function", a, b)  
    return a * b * addTwo(a, b)  
  
def subTwo(a, b):  
    print("In sub function", a, b)  
    return a - b - mulTwo(a, b)  
  
if __name__ == "__main__":  
    print(subTwo(1, 3))
```

Anonymous Functions

AKA Lambda Expressions

Anonymous Functions

Anonymous functions have **NO NAME**.

Called 'Lambda Functions' after *Lambda Calculus*.

Lambda Calculus is a model of computing which only uses Anonymous functions.

In Python, we define these like this:

```
lambda arg1, arg2, etc : [logic]
```

Lambda Function Examples

```
lambdaFunctions.py  
  
lambda x, y : x + y  
lambda a : a ** 2  
lambda x : lambda y : lambda z : x + y + z  
  
x = lambda a, b: a - b  
x(1,2)
```

Map and Filter

Map and Filter

Map and Filter are two functions built into Python. They take functions and lists as arguments.

`map(function, list): returns newList`

Map takes a function as its first argument, and applies it to every element of a list, returning a new list.

`filter(function, list): returns newList`

Filter takes a function as its first argument and applies it to every element of a list, and returns a new list which satisfies the conditions of the function.

Map and Filter

Map and Filter are two functions built into Python. They take functions and lists as arguments.

map()

Map
element

IMPORTANT NOTE!

Map and Filter return 'map' and 'filter' objects respectively. These need to be converted to a list to actually see the data.

Processing also has it's own map function, which is different to the one in Python. Make sure you are using the correct one.

Filter takes a function as its first argument and applies it to every element of a list, and returns a new list which satisfies the conditions of the function.

Problems

Problems

1. Define a function which prints “Hello, CS++”
2. Make it so you can pass a string to the function made in #1, and the function prints that string after “Hello, CS++”
3. Define a function which lets you add two numbers together
4. Add an extra argument to the function so that the programmer can chose the operation.
 - a. The operations should be passed as a string, e.g. ‘+’
5. Write a function where you take a list and a number as an argument. Return ‘True’ or ‘False’ if the number is in the list.
6. Write a function for each maths operation (+, -, *, /, **, //, %) where it will perform these operations on two numbers. Add all these functions to a list, and then use a for loop to call them all. You can use Lambda Expressions for this problem.

Processing Maths Functions

Processing Maths Functions

`dist(x1, y1, x2, y2)`: Returns the distance between the points (x_1, y_1) and (x_2, y_2)

`sin(theta)`: Takes *theta* (in Radians) and returns the sine of that angle.

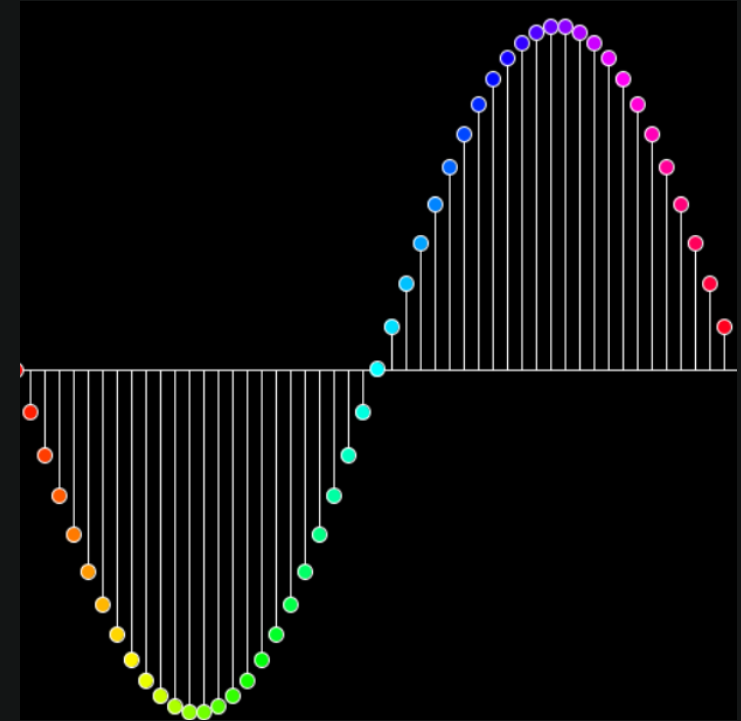
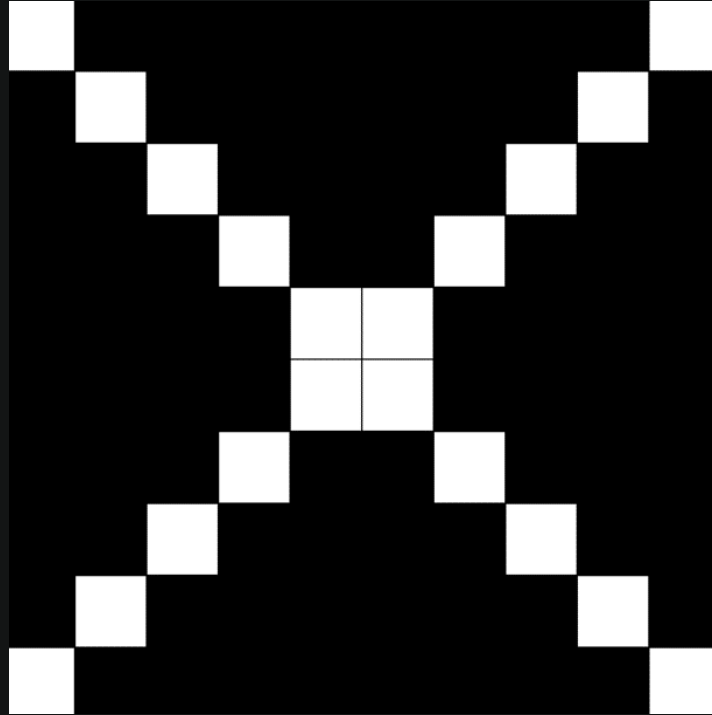
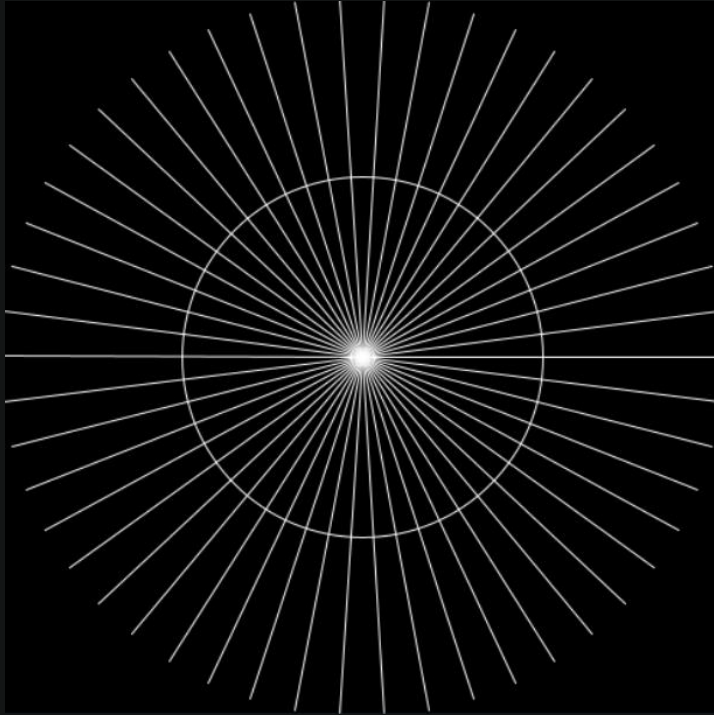
`cos(theta)`: Takes *theta* (in Radians) and returns the cosine of that angle.

`radians(theta)`: Converts *theta* from radians to degrees.

`map(value, start1, stop1, start2, stop2)`: Maps the value from range (start1, stop1) to range (start2, stop2)

Processing Images

Processing Images



The End

Thanks for coming!

Notes uploaded on the Discord.

See you next week!