Programming for Visual Artists

Aalto University
School of Arts, Design and Architecture

2024/2025
Department of Art and Media

Some things to check

Processing reference:
https://processing.org/reference

Recap

IN CASE YOU MISSED IT

- Control Structures
- Animation
- Algorithm Patterns
- Nested Loops
- Boolean Logic
- Decimals/Binary conversion

Welcome

- Recap
- Today’s Goals

Functions

- Modularity
- Reusable Code Components
- How to define a function?
- Parameters
- Return Values

Examples

BREAK (10:30–10:45)

Coding tasks

- Generative Grid
- Animated Waves

Q&A

- For tomorrow…

For today...

Functions

In Processing, a function works much like it does in other programming languages. It's a block of code designed to perform a specific task, making your program more **modular** and easier to **understand** and **maintain**.

Key takeaways

- **Reusable Blocks:** Functions allow you to encapsulate tasks that can be executed whenever needed without rewriting the code.
- **Modularity:** They help organize your code by separating it into logical parts.
- **Parameters and Return Values:** You can pass values (parameters) to functions, and functions can return results after performing their operations.
- **Abstraction:** By using functions, you can hide the complexity of operations, allowing you to use the function without knowing its inner workings.

Functions - example

Function Definition:
The function drawCircle is defined with three parameters: centerX, centerY, and radius. It calculates the width and height of the ellipse by doubling the radius.

Calling the Function:
Inside the setup() function, drawCircle is called with specific arguments to draw a circle in the middle of the window.

Benefits:
This approach keeps the code clean and organized. If you need to change how circles are drawn, you only have to update the drawCircle function, and all calls to it will use the new logic.

```
void setup() {
  size(400, 400);
  background(255);
  // Call our custom function to draw a circle at the center with a radius of 50
  drawCircle(width/2, height/2, 50);
}

void draw() {
  // draw() can be empty if no continuous drawing is needed
}

// This function takes the center coordinates and the radius of the circle
void drawCircle(float centerX, float centerY, float radius) {
  ellipse(centerX, centerY, radius * 2, radius * 2);
}
```

Functions - modularity

Modularity means breaking your program into smaller, independent parts so each part handles a specific task. This makes the code easier to read and maintain.

In this example, drawCircle and drawSquare are modular functions that each perform a specific task. This keeps the main setup() function clear and organized while allowing you to easily reuse or modify the drawing logic.

```
void setup() {
  size(400, 400);
  background(255);

  // Calling modular functions to draw shapes
  drawCircle(100, 100, 50);
  drawSquare(200, 200, 80);
}

// Function to draw a circle at (x, y) with a given radius
void drawCircle(float x, float y, float radius) {
  ellipse(x, y, radius * 2, radius * 2);
}

// Function to draw a square with the top-left corner at (x, y) and a given side l
void drawSquare(float x, float y, float sideLength) {
  rect(x, y, sideLength, sideLength);
}
```

Functions – reusability of code

Reusability of code refers to writing pieces of code, such as functions or modules, that can be used in different parts of a program or even in different projects without rewriting them. This saves time, reduces errors, and makes maintenance easier.

- **Efficiency:** Write once, use many times.
- **Maintainability:** Updates to the code need to be made only once.
- **Consistency:** Reusing code ensures consistent behaviour across different parts of an application.

In this example, the drawShape function can be reused to draw both circles and squares by simply changing the parameter, demonstrating how reusability of code improves efficiency and consistency.

```
void setup() {
  size(400, 400);
  background(255);

  // Reusing the drawShape function to draw different shapes
  drawShape("circle", 100, 100, 50);
  drawShape("square", 250, 100, 50);
}

void draw() {
  // draw() remains empty as shapes are drawn in setup()
}

// A reusable function that draws either a circle or a square ba
void drawShape(String shape, float x, float y, float size) {
  if (shape.equals("circle")) {
    ellipse(x, y, size * 2, size * 2);
  } else if (shape.equals("square")) {
    rect(x - size, y - size, size * 2, size * 2);
  }
}
```

Functions – how to define?

```
// Function with no return value (void)
void myFunction() {
  // Code to execute when the function is called
}

// Function with a return value, for example an integer
int add(int a, int b) {
  return a + b;
}
```

In Processing, defining a function is like other programming languages. You specify a return type (like **void for no return value** or **a data type if you expect a value**), provide a function name, and list any parameters within parentheses. The function's code block is enclosed in curly braces { }.

- **Defining the Function:**
  - displayMessage() is a void function that doesn't return any value; it just prints a message.
  - add(int a, int b) takes two integer parameters and returns their sum.
- **Calling the Function:**
  - In the setup() function, displayMessage() is called to execute its code.
  - The result of add(5, 7) is stored in the variable sum and then printed using println().

```
void setup() {
  size(400, 400);
  background(255);

  // Calling a void function
  displayMessage();

  // Calling a function with a return value and printing the result
  int sum = add(5, 7);
  println("Sum: " + sum);
}

// A simple void function that prints a message
void displayMessage() {
  println("Hello from a function!");
}

// A function that takes two integers, adds them, and returns the result
int add(int a, int b) {
  return a + b;
}
```

In Processing, both the setup() and draw() functions are defined as **void** because they don't return any values. Instead, they're designed to perform actions (like **setting up the canvas** or **continuously drawing on it**) **without needing to send any data back** to the part of the program that calls them.

Processing is built around a specific structure. The environment expects setup() and draw() to run at designated times:

- **setup() runs once when the program starts.**
- **draw() runs in a loop after setup().**

For **static sketches**, you might only use setup().

For **animated or interactive sketches**, you generally need draw() (in addition to setup()) to continuously update the screen.

Void?

Functions - parameters

Parameters are variables that you include in a function's definition to allow you to pass values into the function when it is called. They enable the function to operate on different data each time, making the function more flexible and reusable.

Placeholders: Parameters act as placeholders for the values (arguments) that you provide when calling the function.

Data Types: In Processing, each parameter must have a specified data type (e.g., int, float, String).

Order Matters: The order in which parameters are defined in the function must match the order of the arguments passed during the function call.

Function Definition:
The function drawColoredCircle is defined with four parameters:

x and y (type float): Determine the position of the circle.

radius (type float): Determines the size of the circle.

c (type color): Determines the fill colour of the circle.

Function Call:
In the setup() function, drawColoredCircle is called with specific values for each parameter. This allows the same function to be used to draw circles at different positions, sizes, or colours by simply changing the arguments.

```
void setup() {
  size(400, 400);
  background(255);

  // Calling the function with parameters
  drawColoredCircle(200, 200, 50, color(255, 0, 0));
}

// Function with parameters: x, y coordinates, radius, and a color
void drawColoredCircle(float x, float y, float radius, color c) {
  fill(c);
  ellipse(x, y, radius * 2, radius * 2);
}
```

Functions – return values

Return values are the output that a function sends back to the part of the program that called it. When a function is designed to perform a calculation or process data, it often returns a result so that you can use that value elsewhere in your code.

Purpose:
Return values allow functions to produce and communicate a result after performing operations.

Specifying Return Type:
In Processing (and similar languages), **the function's return type must be specified before its name**. For instance, use int if the function returns an integer, float for a floating-point number, or void if the function doesn't return anything.

Return Statement:
Within the function, the return statement is used to send the output back to where the function was called.

Function Definition:
The multiply function is defined with a return type of int. It takes two integer parameters (**a and b**), calculates their **product**, and returns that value.

Using the Return Value:
In the setup() function, the result of calling **multiply(4, 5)** is stored in the variable **result** and then printed out.

```
void setup() {
  size(400, 400);
  background(255);

  // Calling the function and using its return value
  int result = multiply(4, 5);
  println("The product is: " + result);  // Outputs: The product is: 20
}

// Function that multiplies two numbers and returns the result
int multiply(int a, int b) {
  int product = a * b;
  return product;
}
```

Functions - recursiveness

```
void setup() {
  printCountdown(5); // Start counting down from 5
}

// A recursive function that prints a countdown
void printCountdown(int n) {
  if (n <= 0) {
    println("Done!");
  } else {
    println(n);
    printCountdown(n - 1); // The function calls itself with a smaller number
  }
}
```

Imagine a function saying:
"I call myself because I'm awesome—and because I just can't stop calling myself!"

Recursion can be understood as a process **where a function solves a problem by calling itself with simpler or smaller pieces of the problem until it reaches a simple case it can solve immediately**.
Think of it like a set of nested Russian dolls: each doll contains a smaller one until you reach the tiniest doll that doesn't open further.

Imagine you want to count down from a number until you reach zero. Instead of writing out a loop manually, you can have a function call itself to handle the countdown.

Base Case:
The condition **if (n <= 0)** stops the recursion. When the function reaches 0 or less, it prints "Done!" and stops calling itself.

Recursive Call:
If n is greater than 0, the function prints the number and then calls itself with n - 1. This gradually reduces the number until the base case is met.

Loops vs Recursion

Loops:
Some problems are easier to understand using a loop, especially if you're just iterating over a range of numbers.

Recursion:
May lead to clearer and more elegant solutions for problems that involve breaking down a task into similar sub-tasks.

Use **loops** when efficiency and memory usage are critical, and the problem is naturally iterative.
For simple, repetitive tasks, loops are usually more efficient because they don't create additional function calls and use less memory. They're straightforward when you know exactly how many times you need to repeat an action.

Use **recursion** when it simplifies your code and makes it easier to understand for problems that involve repetitive subproblems or branching structures.
Recursion can be more intuitive and elegant for problems that have a natural hierarchical or nested structure (like tree traversal or solving puzzles). However, it uses additional memory for each recursive call and can lead to stack overflow with too many levels.

```
void setup() {
  int n = 5;
  while (n > 0) {
    println(n);
    n--;  // Decrement n
  }
  println("Done!");
}
```

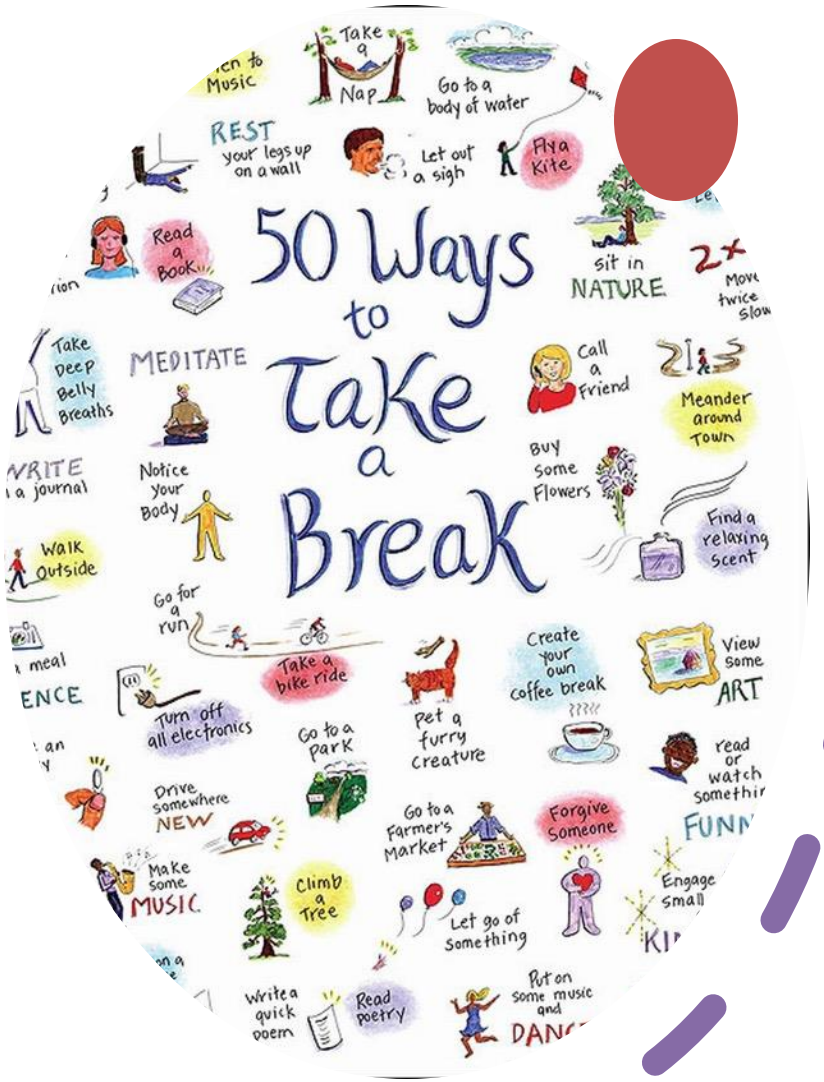
```
void setup() {
  printCountdown(5);
}

void printCountdown(int n) {
  if (n <= 0) {
    println("Done!");
  } else {
    println(n);
    printCountdown(n - 1);
  }
}
```

Break

15 min.!

Please don’t be late!



Hands-On Exercise!

Objectives:

- Still a grid, but a generative one.
- Things happen when hovering.
- Experiment! Try out things!

Get an example from here: https://github.com/ptiagomp/aalto-programming-visual-artists-24-25/tree/main/Session-05_10032025

(if bored, check for the “extra” files!)

Discussion & Q&A

Share your feedback!

Am I going too fast or too slow? Is this too easy or too hard?

Tomorrow's topics:

- Objects and arrays.

Don't forget the assignments!