Corso di Laurea Magistrale in Design, Comunicazione Visiva e Multimediale - Sapienza Università di Roma

Interaction Design
A.A. 2017/2018

9 – Functions in Processing

Francesco Leotta, Andrea Marrella

Last update : 19/4/2018

Functions

- **Functions** are a means of **taking the parts** of a sketch and **separating them out into modular pieces**, making the code *easier to read and to revise*.
- When we write: `line(0,0,200,200)` we are calling the function `line(…)`, a **built-in function** of the Processing environment, which allows to draw a line…
- …but the ability to draw a line **does not magically exist**. Someone defined (hence, wrote the code for) how Processing should display a line!
  - Processing provides a **library** of available **built-in functions** called `processing.core`
- Programmers can define their **user-defined functions**. A **function definition** requires:
  - Return type
  - Function name
  - Arguments
- It looks like:

Remember the definition of the functions `setup()` and `draw()`. They follow the same schema as below.

```
returnType functionName (arguments) {
  // Block of code with the content of function
}
```

Defining and calling a function

- For now, let's focus solely on the **functionName** and code body, ignoring returnType and arguments. Here is a simple example:

```
void drawBlackCircle() {
 fill(0);
 ellipse(50,50,20,20);
}

void draw() {
 background(255);
 drawBlackCircle();
}
```

Function called `drawBlackCircle` that performs one task through two instructions, and consists of drawing an ellipse colored black at coordinate (50,50).

ATTENTION: *The code will never happen unless the function is actually called from a part of the program that is being executed*

This is accomplished by referencing the function name, that is, **calling the function.**

Divide the code with functions

```
int x = 0;
int speed = 1;
void setup() {
 size(200,200);
}
void draw() {
 background(255);
 x = x + speed; // Change x by speed
 // If we’ve reached an edge, reverse speed
 if ((x > width) || (x < 0)) {
  speed = speed *-1;
 }
 // Display circle at x location
 stroke(0);
 fill(175);
 ellipse(x,100,32,32);
}
```

Let’s examine a *bouncing ball* example and divide the code by using functions.

Move the ball!

Bounce the ball!

Display the ball!

Divide the code with functions

```
int x = 0;
int speed = 1;

void setup() {
 size(200,200);
}
void draw() {
 background(255);
 move();
 bounce();
 display();
}
```

Instead of writing out all the code about the ball in `draw()`, we simply call three functions.

Functions can be defined anywhere in the code outside of `setup()` and `draw()`

```
// A function to move the ball
void move() {
 // Change the x location by speed
 x = x + speed;
}

// A function to bounce the ball
void bounce() {
 // If we’ve reached an edge, reverse speed
 if ((x > width) || (x < 0)) {
  speed = speed * - 1;
 }
}

// A function to display the ball
void display() {
 stroke(0);
 fill(175);
 ellipse(x,100,32,32);
}
```

Arguments and Parameters

- **Arguments** are values that are “passed” into a function.
  - You can think of them as inputs that a function needs to operate.
- When we call the function `drawCircle(20,255)` we are calling the function `drawCircle` by passing it **two arguments**…
- …but we are required to give each argument a **name** and a **type** during the definition of the function. To this aim, we will use **parameters**!

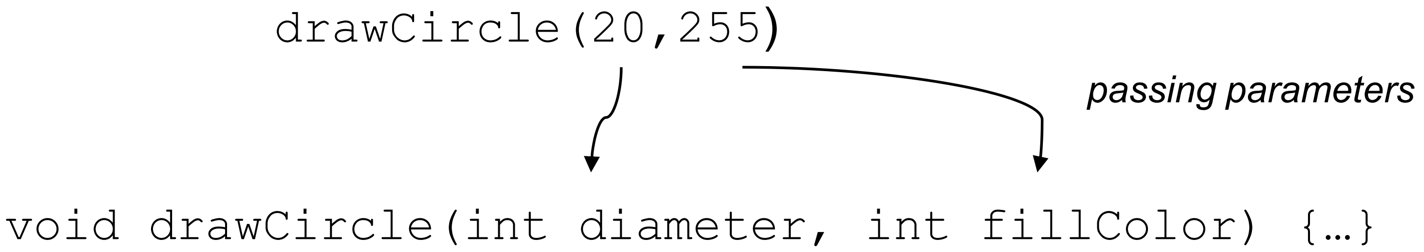
```
void drawCircle(int diameter, int fillColor) {
 fill(fillColor);
 ellipse(50,50,diameter,diameter);
}
```

`diameter` and `fillColor` are **parameters** of the function `drawCircle`.

- A parameter is a variable declaration inside the parentheses in the function definition. This variable is a **local variable** to be used only in that function.
- When we invoke the function `drawCircle(20,255)`, we are passing to it an integer representing the diameter of the circle (`20`) and another integer with the fill color (`255`).

Passing Parameters

- Technically speaking, *parameters* are the variables that live inside the parentheses in the function definition: `void drawCircle(int diameter, int fillColor) {…}`
- *Arguments* are the values passed into the function when it is called, that is, `drawCircle(20,255)`.



```
drawCircle(20,255)
```

passing parameters

```
void drawCircle(int diameter, int fillColor) {…}
```

- You must pass the **same number of parameters** as defined in the function.
- When a parameter is passed, it must be of the **same type** as declared within the arguments in the function definition.
  - An integer must be passed into an integer, a float into a float, and so on.
- The value you pass as a parameter to a function can be a literal value (20, 5, 4.3, etc.), a variable (x, y, etc.), or the result of an expression (8 + 3, 4 * x/2, random(0,10), etc.).

Return Type

- Finally we can answer to the question: «*What does void means?*»
- Let’s recall our function `drawCircle`

```
void drawCircle(int diameter, int fillColor) {
 fill(fillColor);
 ellipse(50,50,diameter,diameter);
}
```

- `drawCircle` is the **function name**, `diameter` and `fillColor` are the **parameters** of the function and `void` is the **return type**. Specifically, `void` means: **no return type**.
- The **return type** is the data type that the function returns.
- Let’s recall for a moment the `random(…)` function.

```
float w = random(1,100);
```

  - We asked the function for a random number between 1 and 100, and `random(…)` gave us back a random value within the appropriate range. Therefore, The `random(…)` function returned a value, specifically a **float** .

Return Type

- If we want to write our own function that returns a value, we have to specify the **return type** in the function definition. Let’s create a simple example:

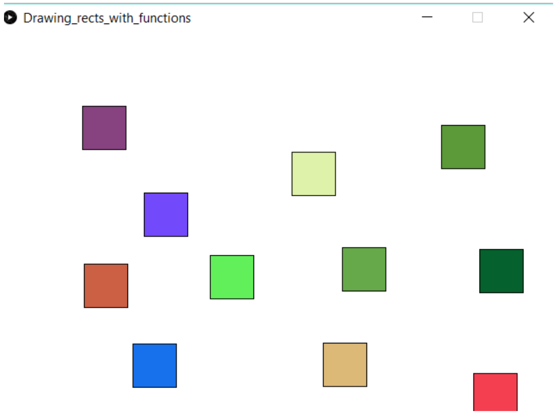
```
int sum(int a, int b, int c) {
 int total = a + b + c;
 return total;
}
```

- Instead of writing `void` as the return type as we have in previous examples, we now write `int`, hence, we want the functions returns an integer value.
- This specifies that the function **must return a value of type integer**. In order for a function to return a value, a `return` statement is required, followed by the return value.
- As soon as the `return` statement is executed, the program exits the function and sends the returned value back to the location in the code where the function was called.
  - That value can be used in an **assignment operation** (to give another variable a value) or in any appropriate expression.

```
int answer = sum(5,10,32);
```

Exercise 1 – *Drawing Rects with functions*

- Write a sketch that draws a new rectangle any time the user presses the left click of the mouse.
- Any rectangle:
  - Is centered around the <x,y> position of the mouse cursor
  - Has a fixed size
  - Is filled by random colors
- Accomplish the task by using a function `drawRect`



Solution of Exercise 1

```
int w;
int h;
void setup() {
  size(640, 480);
  background(255);
  w = 50;
  h = 50;
}
void draw() {}
void mouseClicked() {
 if(mouseButton == 37) {
      drawRect(mouseX, mouseY);
    }
  }
```

```
void drawRect(int xCoord, int yCoord) {

  float r = random(0,255);
  float g = random(0,255);
  float b = random(0,255);

  rectMode(CENTER);

  fill(r,g,b);
  rect(xCoord,yCoord,w,h);

}
```

What is an object?

- In Object-Oriented Programming languages, an **object** is a **thing** that *has properties* and *can do stuff*.
- For example, a human being:
  - has an height, a weight, etc.
  - performs some activities, as it can wake up (presumably you can also sleep), eat, or ride the subway, etc.
- In Programming languages, the *human being template* (to have height, hair, to sleep, to eat, and so on) is known as a ***class.***
- **A class is different from an object**.
- You are an object. I am an object. Albert Einstein is an object. Any person is an object of the class of human beings.
- **So how does this relate to programming?**
  - The **properties** of an object are **variables**.
  - The **stuff** an object can do are **functions**.

Using an object

```
Human human1;
Human human2;

void setup() {
 human1 = new Human();
 human2 = new Human();
}
void draw() {
 background(0);
 human1.move();
 human2.eat();
}
```

Step 1: **Declare an object**

It is like the declaration of a variable, but in this case the data type is **complex** and corresponds to a class name. The declared variables are **human1** and **human2**, two different variables thought to store two objects of kind Human (hence, two human beings).

Step 2: **Initialize an Object**

While with variables we simply assign primitive values, in this case we create a new instance object using the `new` operator followed by a special function called the ***constructor***. Any class provides at least a constructor (it is a function with the same name of the class, and it can provide arguments) that initializes all the object variables.

Step 2: **Using an Object**

Once an object has been successfully declared and initialized with a variable, we can finally use it calling the functions that are written into that object.

More on OOP

- **Any programmer can create its own class!**
- In this course, we do not go into details of classes and objects.
- Interested readers can find more details at the following URL:

  https://processing.org/tutorials/objects/