Functions

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to “call” the function to perform its task when needed.

We haven’t defined precisely what a function is before, but we’ve used them plenty. For example, rect, ellipse, and strokeColor are functions we’ve used for drawing. Also, the update function, where we’ve written the code for our first projects.

Defining a function

A function has the following components.

- **Name** - describes the task that the function performs. To use a function, you “call” that function with its name and pass it input values (known as arguments) that match the types of the function’s parameters.
- **Parameters** - optionally, one or more named values that the function takes as input.
- **Return type** - optionally, a type of value that the function will pass back as output when it is done.
- **Body** - the actual instructions that implement the function. (ie. the block of code inside curly braces.)

Defining a function example

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

Defining a function example

keyword func ***name*** ***parameter*** ***return type*** ***body***

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

Calling a function example

```
let message = greet(person: “Jane”)
```

Calling a function example

calling the function named greet

```
let message = greet(person: “Jane”)
```

Calling a function will cause the instructions in the body of the function to be performed.

Calling a function example

pass the string value "Jane" after the person label.

```
let message = greet(person: "Jane")
```

Values passed into the function as input in the call are called arguments. The argument values are "passed" to the function parameters. The argument types must match the parameter types.

Parameters

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

pass the string value “Jane” after the person label.

```
let message = greet(person: “Jane”)
```

A parameter is really a local variable of the function. Its initial value is set by the argument that is passed to function from the call.

Calling a function example

greet returns a String value as output.

```
let message = greet(person: “Jane”)
```

In this example, the returned value is assigned to a new let constant named message.

After the function has performed all the instructions in its body, the program continues back at the location from which it was called.

Variable scope

The **scope** of an item represents the period of time for which the item is accessible.

Essentially, whenever you see a set of curly braces, a new nested scope is created.

Any declared item is accessible from code that is at the same level, or at a lower level of scope than the level where the item was declared.

Why write functions?

Encapsulation

Modularity

Reusability

Safety

Testability

Why write functions?

Encapsulation. Functions let you combine a group of instructions into one thing, which you can now think of, and use as one unit.

Modularity. Functions are one way to break down a larger problem into smaller parts, making code more manageable, readable, and easier to understand. Functions make it easier to think of our programs in smaller sets of instructions.

Reusability. Functions allow us to reuse code without having to retype it.

Safety. Local variables of a function exist only while the function is executing. These local variables have limited side effects. When global variables are used, you need to be aware of how different functions might make use of those variables.

Testability. Its easier to test small bits of code with limited focus. Creating functions that are small and of limited purpose makes it easier to test code, and verify that it works correctly.

Functions inside functions

Be aware that Swift allows you to define a function INSIDE another function. However, if you do this, that new function is only usable inside the enclosing function.

A common early mistake - accidentally declare a function inside another function, and then wonder why you get an error trying to use the function in another part of your program.

DRY

Don’t

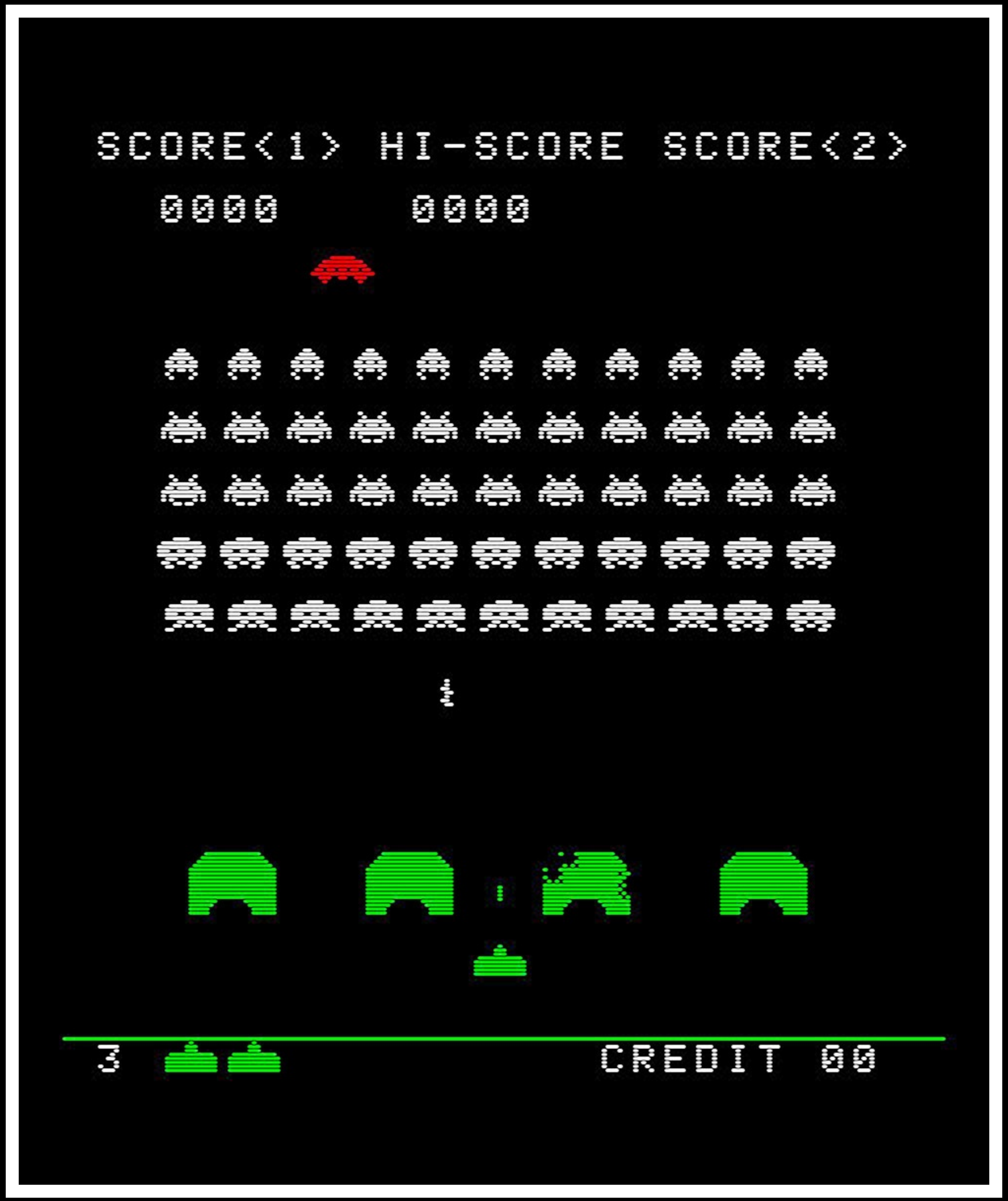
Repeat

Yourself

A favorite programmer’s aphorism.

When programming, you shouldn’t be writing the same code, over and over.

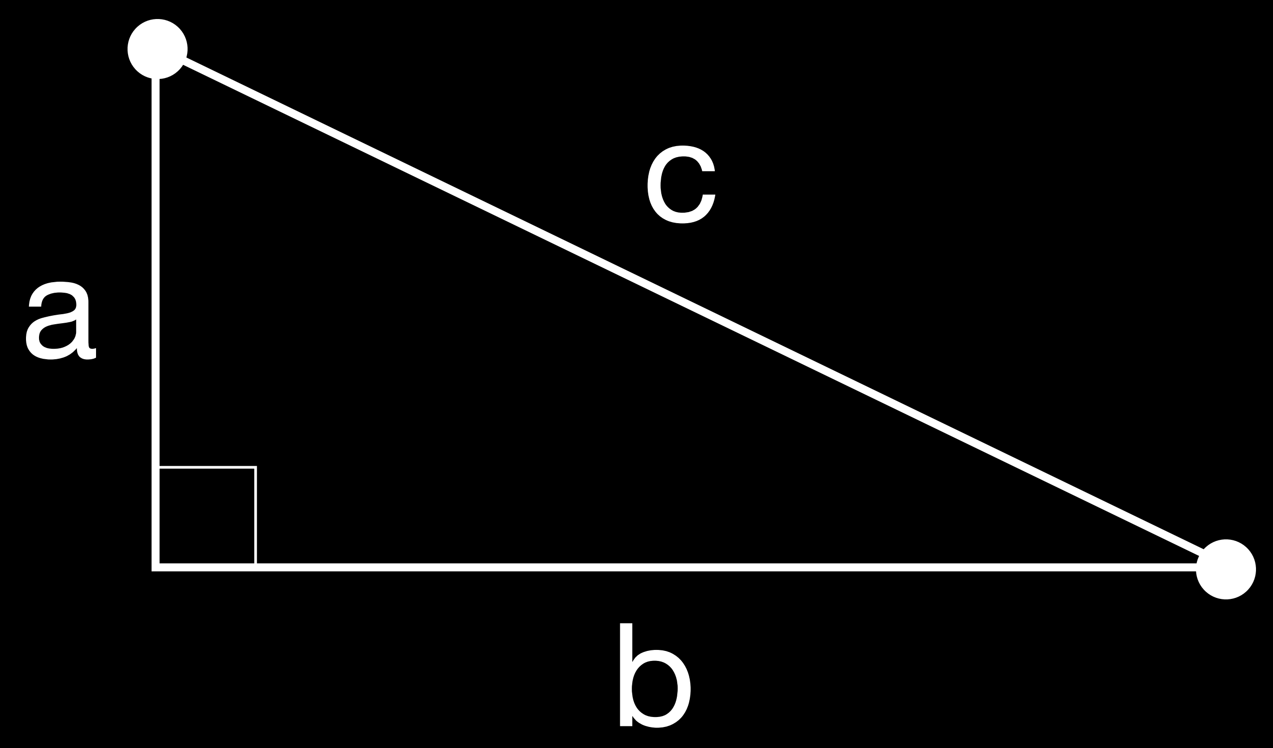
- Clear background.
- Move the aliens.
- Move the hero.
- Draw the aliens.
- Draw the hero.
- Draw the barriers.
- Draw the score.
- Draw other info.



```
background(gray: 0)
moveAliens()
moveHero()
drawAliens()
drawHero()
drawBarriers()
drawScores()
drawInfo()
```

Example

What if sometimes you need to calculate the distance between two points.



Pythagorean Theorem

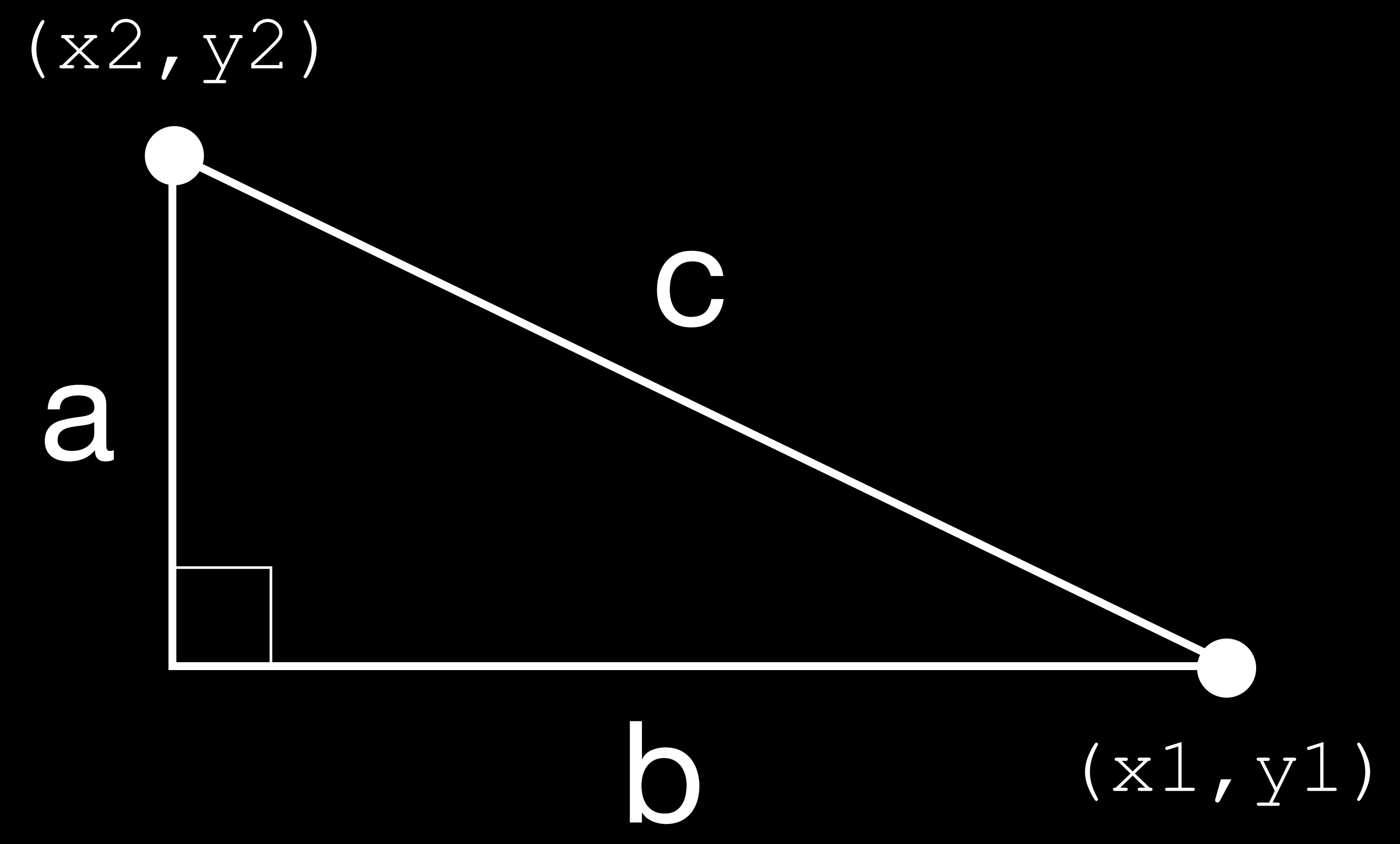
$$a^2 + b^2 = c^2$$

$$c = \sqrt{a^2 + b^2}$$

```
c = sqrt(a*a + b*b)
```

Example

What if sometimes you need to calculate the distance between two points.



```
func distance(x1: Double, y1: Double, x2: Double, y2: Double) -> Double {
    let a = y2 - y1
    let b = x2 - x1
    let c = sqrt(a * a + b * b)
    return c
}
```

Exercise

Convert from fahrenheit to celsius.

Write a function to convert a temperature value in degrees fahrenheit to degrees celsius. The function should have one parameter - for a temperature value (floating point number) in degrees fahrenheit - and return a floating point value - the output temperature in degrees celsius.

```
celsius = (fahrenheit - 32.0) * (5.0/9.0)
```

Exercise

Convert from fahrenheit to celsius.

Write a function to convert a temperature value in degrees fahrenheit to degrees celsius. The function should have one parameter - for a temperature value (floating point number) in degrees fahrenheit - and return a floating point value - the output temperature in degrees celsius.

```
func toCelsius(fromFahrenheit: Double) -> Double {
    return (fromFahrenheit - 32.0) * (5.0 / 9.0)
}
```