Diploma in Computer Studies
SEP 2021
Week-4

Welcome to Creative Computing

DCR2284

Learning Objectives

- At the end of the course, students will be able to:
  - CO1: Describe the creative concepts in mathematics and computing.
  - CO2: Explain the importance origins of geometry to develop motion, images and sound.
  - CO3: Build the Processing application to construct shapes and objects.
  - CO4: Write the coordinate transformations for motions using Processing..

Assignment -1

- Progress
- Surgery Session on Friday

`functions();`

Programs get big

- As soon as we want to do something even slightly impressive our programs start to get pretty complicated
- We have to write a lot of code to achieve what we want
- We need some way to deal with this level of complexity

Yaa..h Abstraction!

- Consider `rect(0,0,100,100);`
- We understand this as "draw a rectangle with its top left corner at 0,0 and a width and height of 100"
- But of course there's a *lot* going on behind the scenes to transform that one line of code into an actual rectangle in our window...

`rect(0,0,100,100);`

- We call `rect(0,0,100,100);`
- In the Processing library it calls *another* kind of `rect()` function
- That calls `rectImpl()`
- That one calls `quad()`
- Which calls `beginShape()` and `vertex()`
- And `vertex()` sets elements in an array called `vertices`
- ... and on and on it goes

`rect(0,0,100,100);`

- We call `rect(0,0,100,100);`
- In the Processing library it calls *another* kind of `rect()` function
- That calls `rectImpl()`
- That one calls `quad()`
- Which calls `beginShape()` and `vertex()`
- And `vertex()` sets elements in an array called `vertices`
- ... and on and on it goes
- Thank god we don't need to know all that and can just say "draw a rectangle"

We're on a need-to-know basis

- In programming we only want to know as much as we *need* to know to get our work done
- Computation is all about *hiding* the details when they're irrelevant
- This ability to ignore those details frees us up to do more, better, and more creative work
- Now, of course, we know more than we used to - we know about the code level

It would be nice to hide things from ourselves!

- Given how helpful it is to have all the details of `rect()` hidden...
- ... it would be nice if we could use this trick of hiding stuff ourselves
- We already do this with variables in some sense, hiding changing numbers inside names int a =10;
- But we could think more clearly about our code if we could tidy it up based on what it does

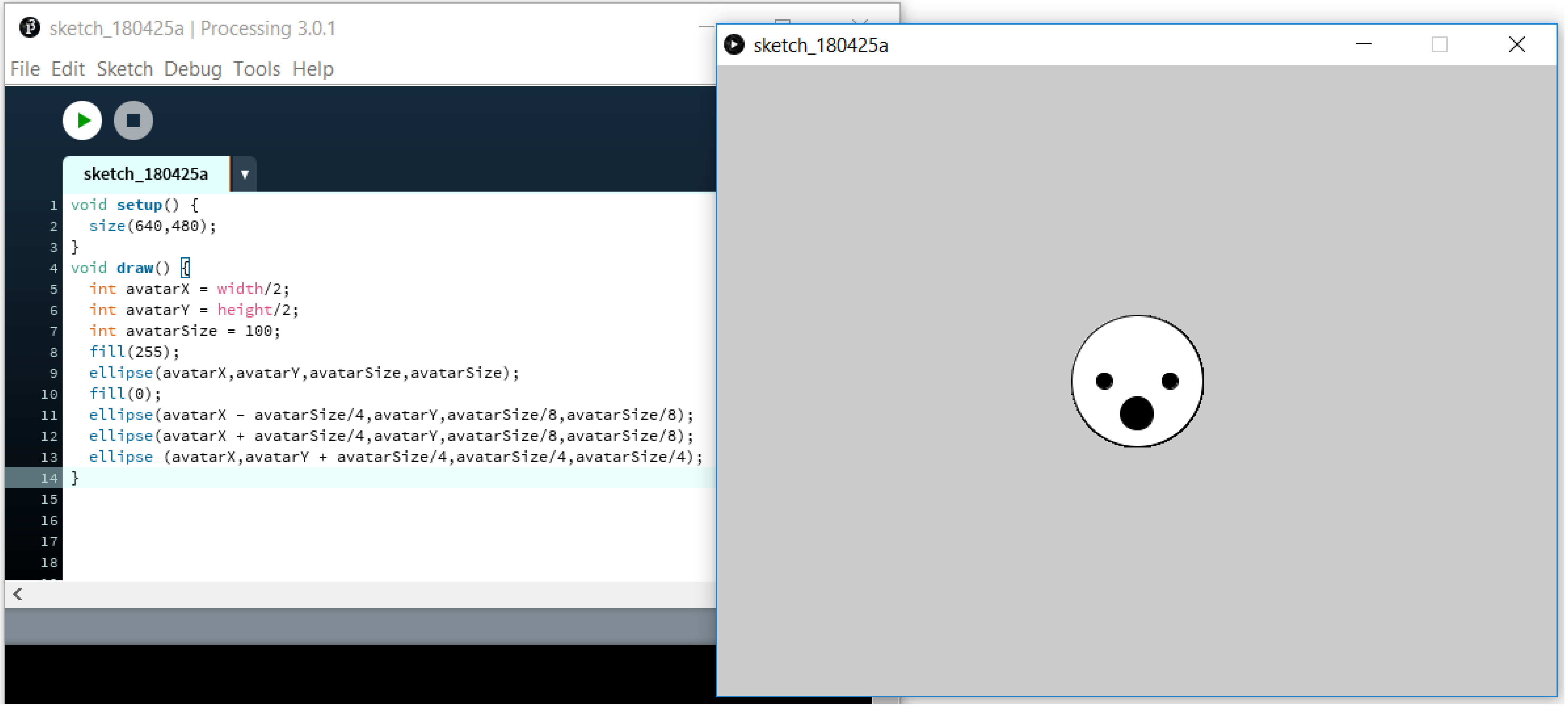
doThatThingYouDo();

- It will not surprise you to learn that we *can* hide things from ourselves
- Just like we use the `rect()` function to draw a rectangle without know how it works...
- ... we can define our *own* functions to organise our code

So you want to draw an avatar

```
void setup() {
  size(640,480);
}

void draw() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```


sketch_180425a | Processing 3.0.1
File Edit Sketch Debug Tools Help
sketch_180425a
sketch_180425a

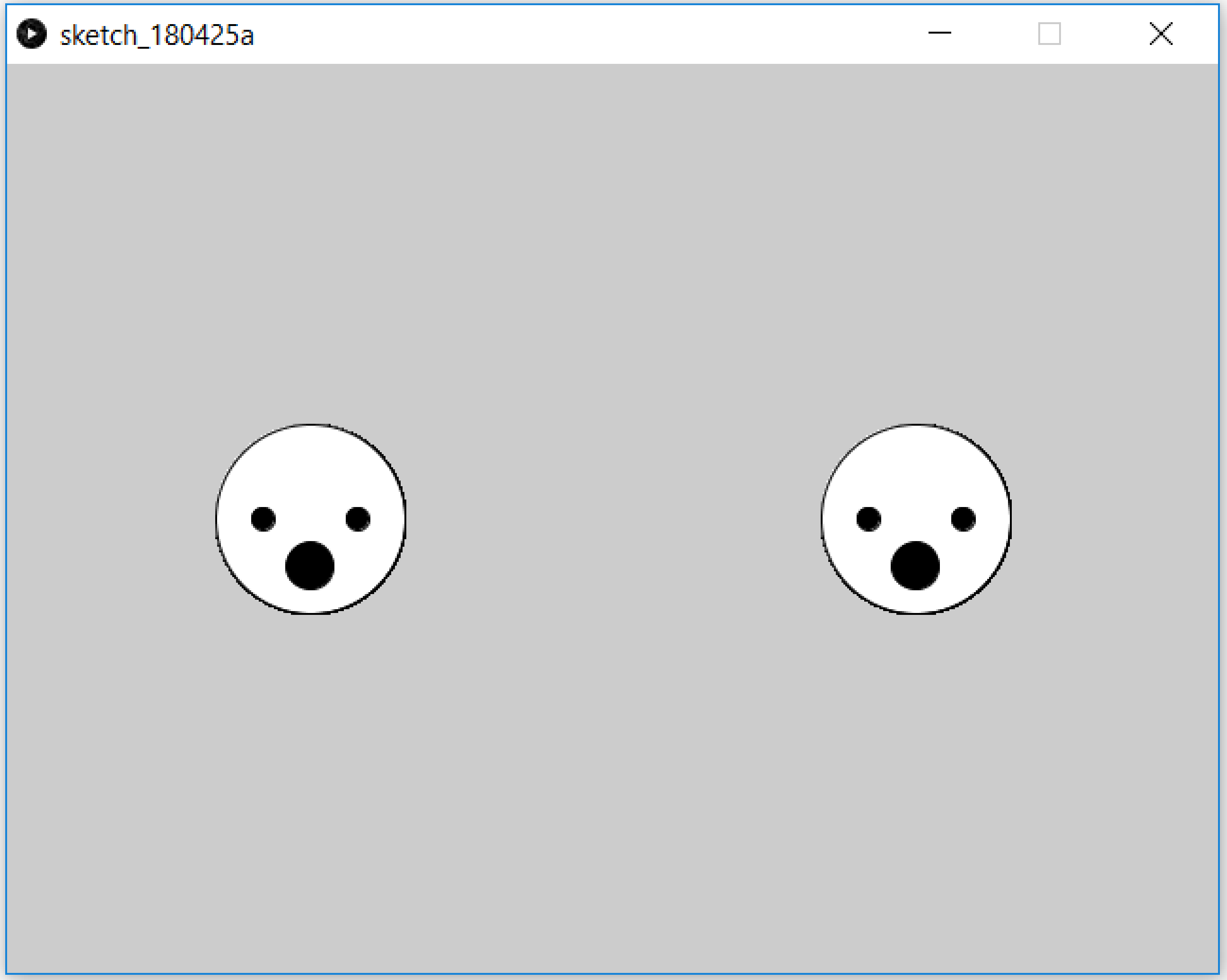
```
void setup() {
  size(640,480);
}
void draw() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

So you want to draw two avatars...

```
void setup() {
  size(640,480);
}

void draw() {
  int avatarX = width/4;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);

  int avatar2X = 3*width/4;
  int avatar2Y = height/2;
  int avatar2Size = 100;
  fill(255);
  ellipse(avatar2X,avatar2Y,avatar2Size,avatar2Size);
  fill(0);
  ellipse(avatar2X - avatar2Size/4,avatar2Y,avatar2Size/8,avatar2Size/8);
  ellipse(avatar2X + avatar2Size/4,avatar2Y,avatar2Size/8,avatar2Size/8);
  ellipse (avatar2X,avatar2Y + avatar2Size/4,avatar2Size/4,avatar2Size/4);
}
```


sketch_180425a

Well, that worked, but...

- As soon as we wanted to basically do the same thing twice our code started looking pretty stupid
- We're so obviously doing *almost* the same thing twice, shouldn't there just be a way to call `drawAvatar()`?
- Well yes there is, **obviously**.

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}
```

Okay, fine.

- Apparently our `drawAvatar()` doesn't exist as a function in Processing

Okay, fine.

- Apparently our `drawAvatar()` doesn't exist as a function in Processing
- **Yet!**

Okay, fine.

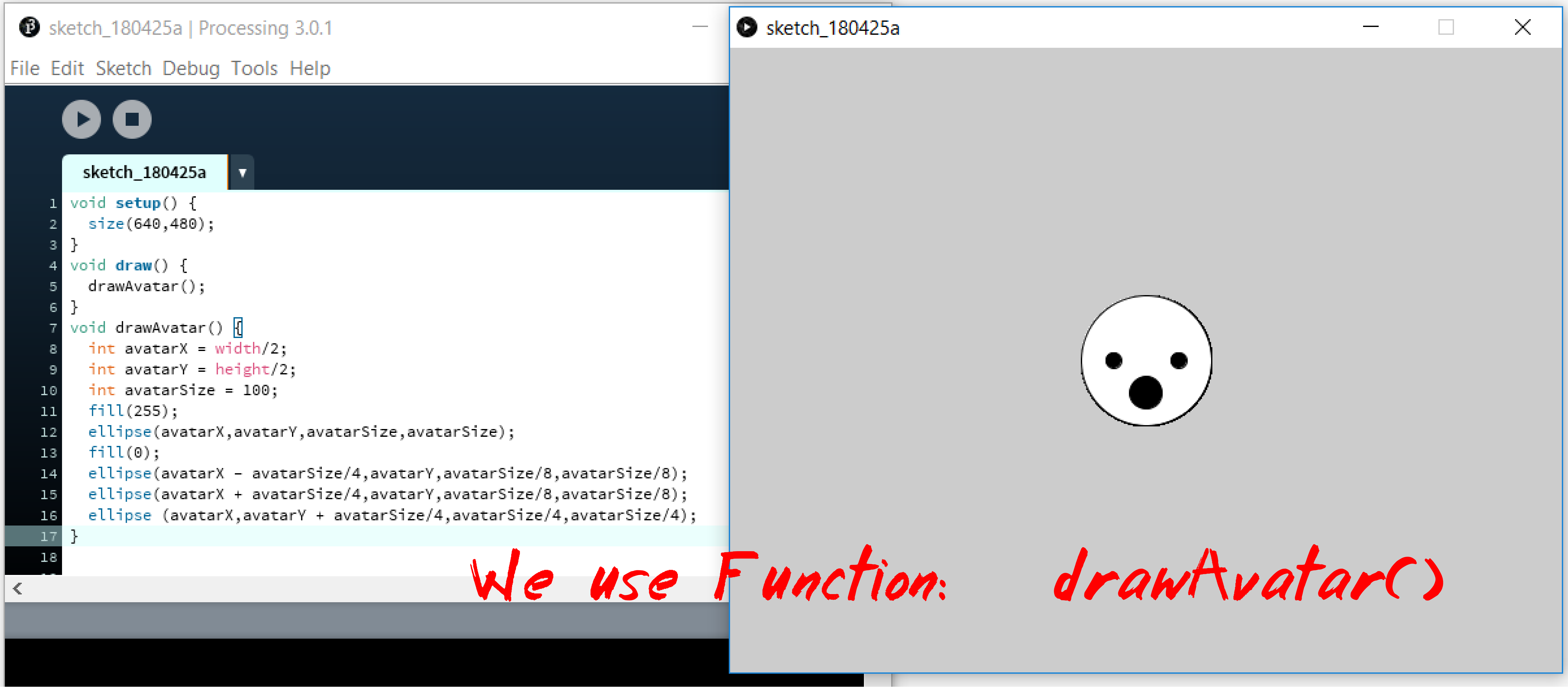
- Apparently our `drawAvatar()` doesn't exist as a function in Processing
- **Yet!**
- We're going to have to define it ourselves so we can use it

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```


sketch_180425a | Processing 3.0.1
File Edit Sketch Debug Tools Help
sketch_180425a
sketch_180425a

```
void setup() {
  size(640,480);
}
void draw() {
  drawAvatar();
}
void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

We use Function: drawAvatar()

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

- It comes *after* our `draw()` function

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

- First we write `void`. We'll explain that soon!

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

- Next we have the *name* of the function, `drawAvatar`

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

- Next is `()` - empty parentheses. This function needs no extra information.

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

- Then we have `{`, meaning "I'm about to tell you what code this function runs"

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

- Then we have the 9 lines of code that execute the function! (Could be more! Could be less!)

Defining a function

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

- And we finish off with our } to say the function is now *defined*

It works!

- We have now *abstracted* the idea of "draw an avatar" into our function
- Notice how our `draw()` now looks *even more clear than before*
- It literally says what it is going to do: draw an avatar
- This idea of moving blocks of related code into functions to make your programs clearer is a huge win

Flow...

- The program starts with `setup()` and runs the code there
- Then it jumps to `draw()` and starts running that code
- It gets to `drawAvatar()`, our function, and jumps to *that* function
- It runs the code inside `drawAvatar()` then jumps *back* to where it was in `draw()`
- Then it hits the end of `draw()` and jumps back to the top of `draw()` for the next frame
- And on it goes...

All neat and tidy!

```
void setup() {
  size(640,480);
  setupAvatar();
  setupWorld();
}

void draw() {
  updatePhysics();
  handleInput();
  drawWorld();
  drawAvatar();
  checkWinState();
}

// Actual definitions of those functions would be down here...
```

- We can imagine programs where everything is in functions!
- `draw()` becomes a nice story of what happens in the program

Okay, but I still want those two avatars...

```
void setup() {
  size(640,480);
}

void draw() {
  drawAvatar();
  drawAvatar();
}

void drawAvatar() {
  int avatarX = width/2;
  int avatarY = height/2;
  int avatarSize = 100;
  fill(255);
  ellipse(avatarX,avatarY,avatarSize,avatarSize);
  fill(0);
  ellipse(avatarX - avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse(avatarX + avatarSize/4,avatarY,avatarSize/8,avatarSize/8);
  ellipse (avatarX,avatarY + avatarSize/4,avatarSize/4,avatarSize/4);
}
```

Oh, right.

- We can call `drawAvatar()` twice, and it works
- But it draws the avatar in the *same place* both times

Information

- Lots of functions only make sense if you can give them *information*
- We don't get a rectangle if we just call `rect();` because it doesn't make sense
- We call `rect(0,0,100,100);` and specify *where* the rectangle should be and what *dimensions* it should have
- We want something like that for `drawAvatar()`

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- Here is `drawAvatar()` again, this time with *arguments*

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- It's exactly the same, but now we have something *inside the parentheses*
- And the code in the function has changed a bit too

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- First we have `int x`
- This means the function wants to be given or *passed* an `int`
- That `int` will be called `x` inside the function
- It specifies the x position of the avatar being drawn
- It's just like a variable

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- Then we have a `,` - a comma
- As you might expect, we use the comma to *separate arguments*
- Because `drawAvatar()` takes multiple arguments

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- Then we have a `int y`
- This is the argument that will specify the y position of the avatar being drawn by the function
- Again, it's like a variable inside the function

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- Another comma before we get our final argument

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- Then we have a `int size`
- This is the argument that specifes the *size* of our avatar

Defining functions with arguments

```
void drawAvatar(int x, int y, int size) {
  fill(255);
  ellipse(x,y,size,size);
  fill(0);
  ellipse(x - size/4,y,size/8,size/8);
  ellipse(x + size/4,y,size/8,size/8);
  ellipse (x,y + size/4,size/4,size/4);
}
```

- Inside the *code* of the function we can see that we are using the arguments just like variables
- But it's different now because the function doesn't know what values they have
- It has to wait until it is called and is given the information required

Calling a function with arguments

```
void draw() {
  drawAvatar();
}
```

- If we try to call our function like this now, what will happen?

Calling a function with arguments

```
void draw() {
  drawAvatar();
}
```

- If we try to call our function like this now, what will happen?
- Yep. Doesn't work.
- Because now that function needs *information*
- So we need to put *parameters* into our function call

Calling a function with arguments

```
void draw() {
  drawAvatar(width/2,height/2,100);
}
```

Now we can draw an avatar!

Calling a function with arguments

```
void draw() {
  drawAvatar(width/4,height/2,100);
  drawAvatar(3*width/4,height/2,200);
}
```

- Even better! We can draw *two* avatars in different places using the parameters!
- Notice, too, how we don't need to be able to *see* the `drawAvatar()` function definition itself
- So long as we *know how it works*
- This is a strong case for *good documentation* like sensible comments that explain your functions!

Functions with *results*

```
void setup() {
  size(640,480);
}

void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  tripleTheWidth(w);
  ellipse(3*width/4,height/2,w,w);
}

void tripleTheWidth(int w) {
  w = w * 3;
}
```

- Sometimes we want functions that *change* something
- What will this do?

Functions with *results*

```
void setup() {
  size(640,480);
}

void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  tripleTheWidth(w);
  ellipse(3*width/4,height/2,w,w);
}

void tripleTheWidth(int w) {
  w = w * 3;
}
```

- The `int w` inside `draw()` is **not the same** as the `int w` in `tripleTheWidth()`!
- The function *does* triple a `w`, just *not the one we wanted*

Functions with *results*

```
void setup() {
  size(640,480);
}

void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  tripleTheWidth(w);
  ellipse(3*width/4,height/2,w,w);
}

void tripleTheWidth(int w) {
  w = w * 3;
}
```

- That's because when we call `tripleTheWidth(w);` Processing passes through the *value inside* `w`
- Not the variable itself

Many happy returns...

- If we can send things into a function (with parameters/arguments), surely we can get things out?
- Well, yes, **obviously** we can, geez. Surprise!
- This is particularly helpful if we have a function that *calculates* something
- Or perhaps a function that can *check* something for us and report back

`tripleTheWidth`

```
void tripleTheWidth(int w) {
  w = w * 3;
}
```

- It *does* triple the value passed in as the argument
- But it doesn't *give it back* after its tripled
- Pointless!

tripleTheWidth

```
void tripleTheWidth(int w) {
  w = w * 3;
}
```

- Finally we're going to talk about that `void` at the start of the function definition
- That `void` means "this function doesn't give anything back"
- And if we can write `void` to mean that, maybe we can write something else to *give something back*...
- Like... what?

`tripleTheWidth` still

```
int tripleTheWidth(int w) {
  w = w * 3;
}
```

- If, instead of `void` we write `int` we're saying "this function gives you back an integer"
- So that bit in front of the function definition is the *return type*
- It tells us the *kind of thing* this function gives back
- But this doesn't work... why?

`tripleTheWidth` doesn't

```
int tripleTheWidth(int w) {
  w = w * 3;
  return w;
}
```

- In order to give something back we need to `return` it inside the function
- We do this by writing `return` and then the thing we want to return, like the resulting argument `w`
- The thing we `return` has to match the *type* we said we would return at the front of the function definition (an `int` in this case)
- Now Processing doesn't complain

Damn you tripleTheWidth!!!

```
void setup() {
  size(640,480);
}

void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  tripleTheWidth(w);
  ellipse(3*width/4,height/2,w,w);
}

int tripleTheWidth(int w) {
  w = w * 3;
  return w;
}
```

- THIS STILL DOESN'T WORK??? WHYYYYY???

Damn you `tripleTheWidth`!!!

```
void setup() {
  size(640,480);
}

void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  tripleTheWidth(w);
  ellipse(3*width/4,height/2,w,w);
}

int tripleTheWidth(int w) {
  w = w * 3;
  return w;
}
```

- THIS STILL DOESN'T WORK??? WHYYYYY???
- Yeah, because we don't actually *use* the value `tripleTheWidth` is trying to give back

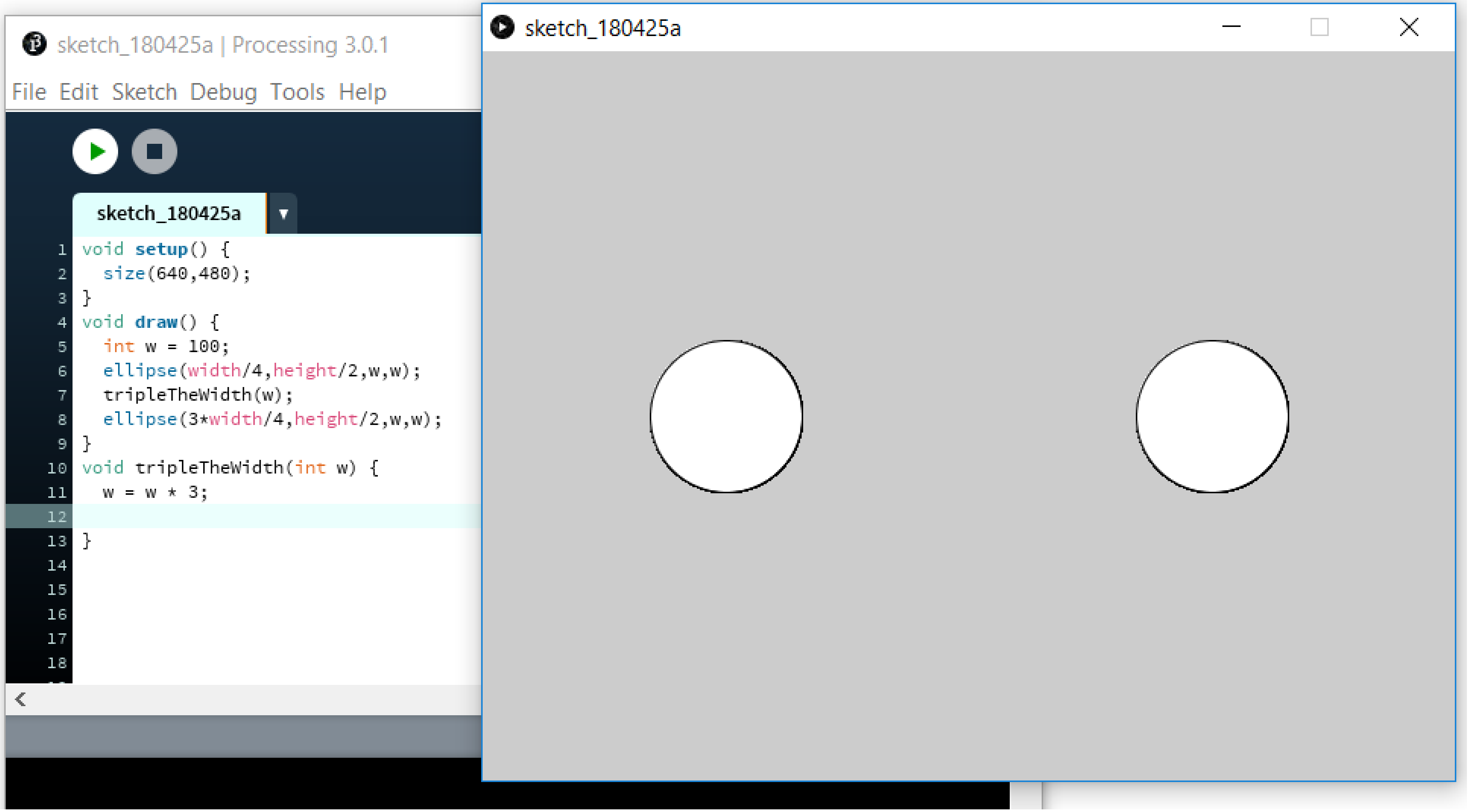
`tripleTheWidth` you beautiful function you!

```
void setup() {
  size(640,480);
}

void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  w = tripleTheWidth(w);
  ellipse(3*width/4,height/2,w,w);
}

int tripleTheWidth(int w) {
  w = w * 3;
  return w;
}
```

- We need to *receive* the value calculated by `tripleTheWidth`
- We can put it back into the `w` variable for instance


sketch_180425a | Processing 3.0.1
File Edit Sketch Debug Tools Help
sketch_180425a
sketch_180425a

```
void setup() {
  size(640,480);
}
void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  tripleTheWidth(w);
  ellipse(3*width/4,height/2,w,w);
}
void tripleTheWidth(int w) {
  w = w * 3;

}
```

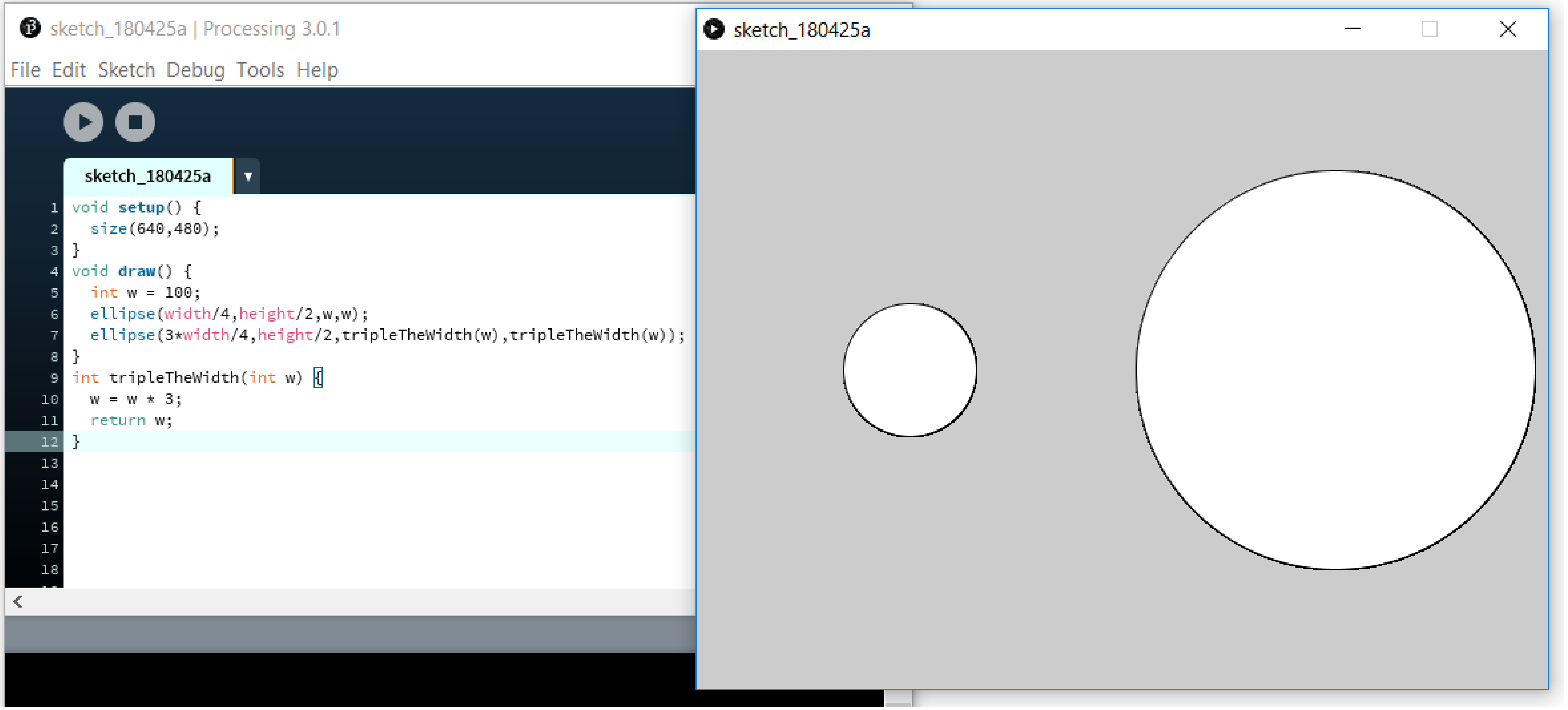
tripleTheWidth you beautiful function you!

```
void setup() {
  size(640,480);
}

void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  ellipse(3*width/4,height/2,tripleTheWidth(w),tripleTheWidth(w));
}

int tripleTheWidth(int w) {
  w = w * 3;
  return w;
}
```

- Or we can use it directly wherever we want to use the *value it calculates*
- So we can imagine that function call being substituted with the *value* that is `returned`


sketch_180425a | Processing 3.0.1
File Edit Sketch Debug Tools Help
sketch_180425a
sketch_180425a

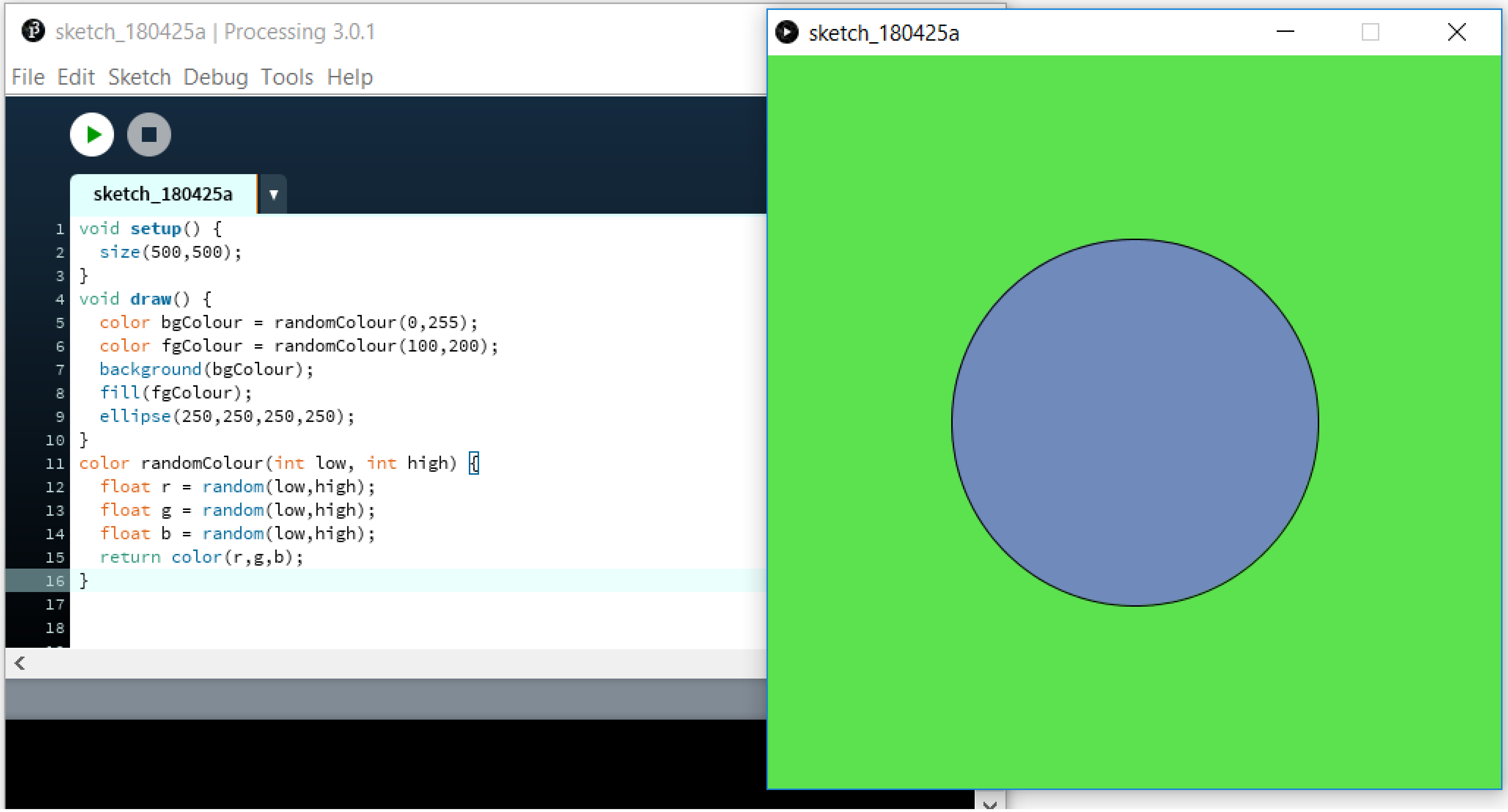
```
void setup() {
  size(640,480);
}
void draw() {
  int w = 100;
  ellipse(width/4,height/2,w,w);
  ellipse(3*width/4,height/2,tripleTheWidth(w),tripleTheWidth(w));
}
int tripleTheWidth(int w) {
  w = w * 3;
  return w;
}
```

More useful? Class- Activity

```
void setup() {
  size(500,500);
}

void draw() {
  color bgColour = randomColour(0,255);
  color fgColour = randomColour(100,200);
  background(bgColour);
  fill(fgColour);
  ellipse(250,250,250,250);
}

color randomColour(int low, int high) {
  float r = random(low,high);
  float g = random(low,high);
  float b = random(low,high);
  return color(r,g,b);
}
```


sketch_180425a | Processing 3.0.1
File Edit Sketch Debug Tools Help
sketch_180425a
sketch_180425a

```
void setup() {
  size(500,500);
}
void draw() {
  color bgColour = randomColour(0,255);
  color fgColour = randomColour(100,200);
  background(bgColour);
  fill(fgColour);
  ellipse(250,250,250,250);
}
color randomColour(int low, int high) {
  float r = random(low,high);
  float g = random(low,high);
  float b = random(low,high);
  return color(r,g,b);
}
```

Modularity and reuse!

There are two main reasons why functions are so great, and they have special names!

Functions are **modular**. We can tidy our code into separate, self-contained blocks that make sense as a unit. Our code becomes more organised, more readable, easier to fix.

Functions are **reusable**. We can use a function over and over again without writing out all the code in it. This makes our programming more efficient and less lengthy. It's like free code!

Food for thought

- With functions it's like we suddenly have this team of different workers who we can ask to do specific things for us whenever we want
- Sometimes we give them some information so they can do their job (parameters / arguments)
- Sometimes they come back and give us some information that they worked out (return values)

Points to Ponder!

The weird thing is that these workers are all also... *us*.

Thank you