# Visual to text coding LESSON 12: **Functions that give back**

Please refer to the online lesson plan on the DT Hub to access all website links and additional resources.

This is the final in a series of lessons to transition from visual coding to text-based coding with a General Purpose Programming language. See next steps for suggested courses and learning sequences after this lesson.

Included videos can be used by a beginner teacher and/or students to see how to code each of the simple programs step-by-step in *all three languages*: Scratch, Python and JavaScript.

This lesson may take two to three 45-minute periods. It builds on the coding concept of functions (see [ACTDIP030](https://jsfiddle.net/nathanesa/Lkh5pqvf/?accContentId=ACTDIP030) in Australian Curriculum: Digital Technologies – Digital Technologies Processes and Production Skills), by introducing the concept of **return values**. Functions are often written to **return** a result, which can then be used in the main program.

# Curriculum links

Links with Digital Technologies Curriculum Area

| **Strand** | **Year** | **Content Description** |
| --- | --- | --- |
| Processes and Production Skills | Years 5–6 | Design algorithms involving multiple alternatives (branching) and iteration (AC9TDI6P02). |
| Years 7–8 | Design algorithms involving nested control structures and represent them using flowcharts and pseudocode (AC9TDI8P05).  Trace algorithms to predict output for a given input and to identify errors (AC9TDI8P06). |

# Assessment

Students can undertake a self-reflection of the programming task. The teacher can use the completed self-assessments to assist in summative assessment.

* Download the self-assessment sheet in Word or PDF format.

In assessing code in languages like Python or JavaScript, consider a rubric that brings in important skills for General Purpose Programming.

* Download a sample rubric in Word or PDF format.

# Learning hook



Image: CordMediaStuttgart/Pixabay

We can think of functions like tools that we delegate jobs to.

Here's how a toaster might look as a function:

|  |  |  |
| --- | --- | --- |
|  | What we provide (arguments) | What it gives back (return value) |
| Toaster | Slice of bread, toast setting, electricity | Toasted bread |

What's missing from the table above?

The table only describes how to use the toaster. It doesn't give the code *inside* the function.

Modern toasters are actually very complicated! Most of us don't really know how to make one, but we generally don't need to. We just need to know how to *use* it. (See this video on Thomas Thwaites' quest to try to build a modern toaster from raw materials.)

As a class, see if you can complete the table for other tools:

|  |  |  |
| --- | --- | --- |
|  | What we provide (arguments) | What it gives back (return value) |
| Toaster | Slice of bread, toast setting, electricity | Toasted bread |
| Kettle |  |  |
| Coffee machine |  |  |
| Hole punch |  |  |
| 3D printer |  |  |

In this lesson, we'll see how Python and JavaScript come with many built-in functions that we have already been using. These functions may have very complicated code inside, but luckily we don't need to know it.

Then we'll look at the last concept we need to write our own functions – the return value.

# Learning map and outcomes

In this lesson, students will:

1. access an online programming environment for visual code (**Scratch**) *and* for General Purpose Programming (**Python** or **JavaScript**)
2. identify and describe built-in functions already used
3. practise writing functions with return values
4. observe how functions are used in Graphical User Interfaces (GUIs), triggered by user inputs
5. write organised code for a small battle game by building a set of functions.

# Learning input

Did you know you have been using built-in Python functions or JavaScript functions since the beginning of this course? Functions like **print**, **alert** or **randint** are all available because they were once written by someone else.

Work through these three examples to learn more about the functions you've been using.

**Example 1**

Consider this Python code:

print("Happy New Year!")

Identify and write down:

1. the name of the function being called
2. the argument being supplied.

**Example 1: Solution:**

1. The function being called is **print**.
2. The argument is the text "Happy New Year!".

**Example 2**

Consider this JavaScript code:

alert("Welcome!");

Identify and write down:

1. the name of the function being called
2. the argument being supplied.

**Example 2: Solution:**

1. The function being called is **alert**.
2. The argument is the text "Welcome!".

Here's the entry for JavaScript's **alert** function from the JavaScript documentation. You can see that it shows the function name, the parameters and describes what the function does.



**Example 3**

Just as parameters allow us to supply a function with values when we call it, functions can also **return** a value, to give back a result rather than just display it.

Consider this Python code:

dice\_roll = randint(1, 6)

Identify and write down:

1. the name of the function being called
2. the arguments being supplied
3. the value returned by the function to be stored in dice\_roll.

**Example 3: Solution:**

1. The function being called is **randint**.
2. The arguments are numbers 1 and 6.
3. The value returned is a random number between 1 and 6.

Here's the entry for Python's **randint** function from the Python documentation. You can see that it shows the function name, the parameters and describes what the function returns.



**Example 4**

Consider this JavaScript code:

playerName = prompt("What is your name?");

Identify and write down:

1. the name of the function being called
2. the arguments being supplied
3. the value returned by the function to be stored in playerName.

**Example 4: Solution:**

1. The function being called is **prompt**.
2. The argument is the text "What is your name?".
3. The value returned is text that the user typed in response to the question, eg "Bob".

JavaScript's **prompt** function is described in the JavaScript documentation. There's even an optional parameter you may not have used before.

# Learning construction

**STEP 1: SET-UP**

Unlike lessons 10 and 11, this lesson does not use turtle graphics. The Python and JavaScript environments can be set up as in Lesson 1.

SIDEBAR – What happened to Scratch?

In this final lesson of the course, we're using a feature that is not available in Scratch. Scratch allows functions created with 'My Blocks' to have parameters, but they cannot return values.

Does this mean that a 'My Blocks' function can't change anything in the main program it was called from? Not entirely. Functions in Scratch have access to any variables in the same sprite script, so they can see and alter the values of these variables. They also have access to all variables that were created as "For all sprites".

But Scratch is quite lenient. In many languages, the code inside a function cannot see the variables that were declared and used outside of that function. Those variables are "out of scope" from the perspective of the function. That can be a good thing! It reduces bugs from reusing variable names.

Languages each have their own rules about variable scope. See this article about Python, and this article about JavaScript. By understanding the rules of their chosen language and properly managing variable scope, programmers can better structure their programs.

**STEP 2: WRITING A FUNCTION TO RETURN A VALUE**

This video demonstrates a simple function that calculates the square of a given number, then returns the result. Try it yourself!

*Completed code: Python, JavaScript*

**STEP 3: GETTING GOLD COINS**

This video shows a second example with a function that returns a random number between 2 and 20. Try it yourself!

(This is especially helpful for JavaScript! The messy code needed to get a random integer is now tidied away from the main program into a function.)

*Completed code: Python, JavaScript*

Next, let's say you're a bit more lucky than most. Change the code inside the function so that it always returns between 10 and 30 gold coins.

Finally, write a second function that returns a number of silver coins between 0 and 100. Add the call to your main program so that the silver coins are displayed after the gold coins.

*Solution code: Python, JavaScript*

**STEP 4: A FUNCTION TO ANALYSE AN ARRAY**

This final example shows a function that accepts an array, finds the lowest value in it, and returns that value. Try it yourself!

*Completed code: Python, JavaScript*

Next, add a second function to find the highest value in a given array. Test it with the same array from the main program.

*Solution code: Python, JavaScript*

**STEP 5: FUNCTION EXERCISE**

Carefully read the pseudocode below.

1 BEGIN

2 Function calculateFactorial(**number**)

3 **result** ← **1**

4 For **i** from 2 to **number**

5 **result** ← **result** × **i**

6 End For

7 Return **result**

8 EndFunction

9

10 **factorial** ← calculateFactorial(4)

11 Display 'The factorial of 4 is', **factorial**

12 Display 'The factorial of 5 is', calculateFactorial(5)

13 END

Predict the output of the program.

**Step 5: Solution:**

This function in this program calculates the factorial of a number, which is the product of all the positive whole numbers up to and including that number. For example, the factorial of 4 is found by 1 × 2 × 3 × 4 = 24.

Here's the expected output of the program:

The factorial of 4 is 24

The factorial of 5 is 120

Now, implement the code in Python *or* JavaScript.

*Solution code: Python, JavaScript*

Mathematically, the factorial of 0 is always 1. Will the function work correctly if the argument for **number** is 0? Try it and see!

**Step 5: Solution 2:**

Yes, the function has been designed to return 1 when **number** is 0. On Line 3 of the pseudocode, **result** is set to 1. The for loop on lines 4 through 6 will never take place, since **number** is less than 2. This means that the **result** will keep the value 1, and this will be returned on Line 7.

**STEP 6: PREVIEW OF GRAPHICAL USER INTERFACES**

So far, all our programs have relied on simple text input and output, sometimes called a Command Line Interface.

But both Python and JavaScript can be used to code applications with a **Graphical User Interface** (GUI) involving buttons, textboxes, images and other components. This is the kind of application we use every day on webpages, phone or desktop apps, and functions are critical to making them work.

The final videos online preview JavaScript programs with GUIs.

* Functions practical video 1
* Functions practical video 2

# Challenge

These challenges use the skills covered so far. By writing or modifying their own programs, students have an opportunity to demonstrate Application and Creation.

1. Write and test a function that accepts a person's name, then returns a fancy greeting by choosing one of three random adjectives.

For example, "Ladies and gentlemen, introducing the Illustrious Bob!"

Here is the pseudocode for the function itself:

Function produceWelcome(**name**)

**randomNumber** ← choose random between 1 and 3

If **randomNumber** = 1

**adjective** ← 'Amazing'

Else If randomNumber = 2

**adjective** ← 'Illustrious'

Else

**adjective** ← 'Glorious'

End If

**result** = 'Ladies and gentlemen, introducing the', **adjective**, **name**, '!'

Return **result**

EndFunction

In Python or JavaScript, implement the function and some test code in the main program.

*Solution code: Python, JavaScript*

1. Develop a simple battle game with two functions.

The function **getOgreAttack()** generates the amount of damage done every time the ogre attacks the player.

* First, roll a 6-sided die. If the value is 2 or less, the troll misses and the damage is 0. Otherwise, roll two 6-sided dice and get the sum. This damage value is returned.

**TASK:** Write the pseudocode for this function.

**First solution:**

Function getOgreAttack()

**firstRoll** ← choose random between 1 and 6

If firstRoll < 3

**damage** ← 0

Else

**damage** ← (random between 1 and 6) + (random between 1 and 6)

End If

Return **damage**

EndFunction

**TASK:** Now code and test the function in Python or JavaScript.

*Sample solution: Python, JavaScript*

The function **getPlayerAttack(lunge)** is called every time the player takes a swing at the ogre. It calculates and returns the damage value by the following process:

* + If **lunge** is False, this is a normal attack. First, roll an 8-sided die. If the value is 2 or less, the player misses and the damage is 0. If the value is 8, this is a critical hit and the damage is 20. Otherwise, roll three 5-sided dice and get the sum.
  + If **lunge** is True, this is a lunge attack. First, roll an 8-sided die. If the value is 3 or less, the player misses and the damage is 0. Otherwise, roll three 10-sided dice and get the sum.

**TASK:** Write the pseudocode for this function.

**Second Solution***:*

Function getPlayerAttack(**lunge**)

If **lunge** = false

**firstRoll** ← choose random between 1 and 8

If **firstRoll** < 3

**damage** ← 0

Else If **firstRoll** = 8

**damage** ← 20

Else

**damage** = (random between 1 and 8) + (random between 1 and 8) + (random between 1 and 8)

End If

Else

**firstRoll** ← choose random between 1 and 8

If **firstRoll** < 4

**damage** ← 0

Else

**damage** = (random between 1 and 10) + (random between 1 and 10) + (random between 1 and 10)

End If

End If

Return **damage**

EndFunction

**TASK:** Now code and test the function in Python or JavaScript.

*Sample solution: Python, JavaScript*

The main program keeps track of the player's health (start with 40) and ogre's health (start with 60).

It contains a loop for the game, with the following steps inside:

* The ogre goes first. Get the ogre's damage using the function you made, then subtract it from the player's health. Display as "The ogre hits you for ?? damage."
* If the player is still alive, ask the player whether to lunge or not, then get the player's damage using the function you made and subtract it from the ogre's health. Display as "You hit the ogre for ?? damage."
* Display current health of player and ogre.

The loop ends if the player's health *or* the ogre's health reaches 0 or below.

**TASK:** Write the pseudocode for the main program.

**Third Solution**

BEGIN

**playerHealth** ← 40

**ogreHealth** ← 60

While **playerHealth** > 0 and **ogreHealth** > 0

// Do ogre attack.

**ogreDamage** ← getOgreAttack()

Display 'Ogre hits you for', **ogreDamage**, 'points.'

**playerHealth** = **playerHealth** - **ogreDamage**

// Do player attack.

If **playerHealth** > 0:

**response** ← Input 'Do you want to lunge? (Y/N)'

If response == 'Y'

**playerDamage** ← getPlayerAttack(true)

Else

**playerDamage** ← getPlayerAttack(false)

End If

Display 'You hit Ogre for', playerDamage, 'points.'

**ogreHealth** ← ogreHealth – playerDamage

End If

// Display health.

Display 'Ogre health points:', ogreHealth

Display 'Your health points:', playerHealth

End While

// Finally, declare the winner.

If ogreHealth <= 0

Display 'You defeated the ogre!'

Else

Display 'The ogre defeated you!'

End If

END

**TASK:** Finally, code and test the main program along with the functions you already coded.

*Solutions: Python, JavaScript* Next steps

Congratulations on working through the 12 lessons in this series.

Click here for suggestions for continuing the journey with JavaScript or Python.

# Resources

* Setting up online environments
* Online environments for coding in each language
  + Scratch
  + repl.it : an online environment suited to Python
  + JSFiddle : an online environment suited to JavaScript
* Cheat sheets listing basic commands for coding:
  + Python Cheatsheet (from Grok Learning)
  + JavaScript CheatSheet (Tip: Press the little blue tabs to move Variables, Basics, Strings and Data Types to the top.)

# Next steps

View our next lesson sequence that provides step-by-step video tutorials and challenges to incorporate Graphical User Interfaces (GUIs) into your General Purpose Programming.

Here are some other recommended options to continue with General Purpose Programming:

* Coding a sentimental chatbot in Python

A series of video tutorials to build a chatbot in Python that incorporates AI (Natural Language Processing). Coding difficulty is appropriate for students who have worked through the Visual to Text Coding lesson sequence.

* App Lab at code.org

Use JavaScript\* via blocks or text to create apps. The App Lab is also incorporated into free courses at code.org.

* Code for schools (csinschools) Python Editor

Code for Schools manages to break down complex content into manageable and accessible interactive lessons that engage students and progress their learning in alignment with the Australian curriculum.

* The MakeCode platform is a set of online environments supporting both block coding and general purpose programming.

MakeCode for micro:bit is an online environment for writing code for the popular micro:bit device. It supports both JavaScript\* and Python. A simulator allows testing of code without the physical device, though the device is recommended. Lessons and project ideas are available on the official site. Physical Tech from Go to WHOA! is also suggested.

MakeCode Arcade is an online environment for writing code for MakeCode Arcade devices, which function like small handheld game consoles. It supports JavaScript\*. A simulator allows testing of code without the physical device. Lessons and project ideas are available on the official site.

MakeCode for Minecraft is an online environment for writing code to run in Minecraft. It supports both JavaScript\* and Python.

\* This is actually Microsoft's variant TypeScript.

It is linguistically very similar to JavaScript.