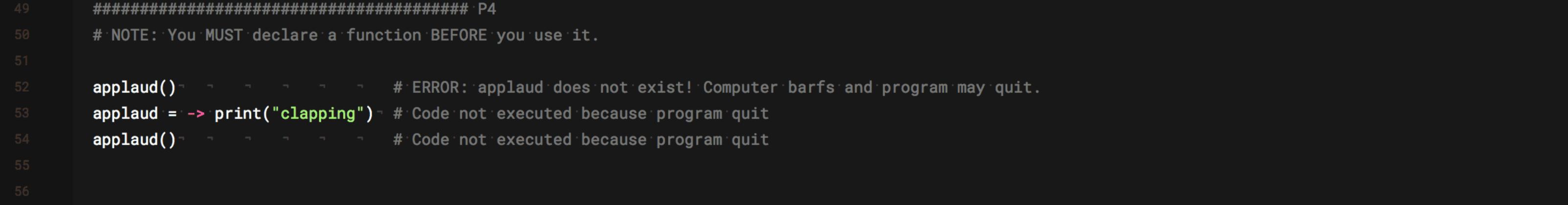


```
#'In CoffeeScropt, a function or "function literal" looks like this:
#
# -> FUNCTION CODE/BODY
# example:
-> print("I ran!") -
# This code will have no result right now.
# The "->" means "new function starting here"
# The code that comes after is the function body
# All together this is called a "function declaration"
```

28	######################################
29	# The function does not run automatically though.
30	#'It'is'set'aside'for'running'LATER.
31	#'It'is'as'if'you'-explained-'to'the'computer'how'to'do'something'instead'of'-telling-'it'to'do'something.
32	
33	print("step <sup>.</sup> 1")
34	-> print("I don't run")
35	print("step <sup>.</sup> 2")
36	
37	

```
#'If'we'want'to'USE our function, we must store it in a variable so we can refer to it.
# We can us the variable name to call the function.
applaud = ·-> ·print("clapping") ·# ·Store ·the ·function ·in ·the ·variable · 'applaud'
applaud() * # Call the custom function
applaud() *# Call the custom function again
applaud() # ...and again
#'In'a'sense, we have named our function.
```



```
# Again, a function declaration DESCRIBES the function, it does not run the function code.
# Pay careful attention to the order that the print commands run.
print(1)
sayThree = -> print(3)
                            # Tell the computer to remember this code under the name 'applaud'
print(2)
sayThree()
                            # Execute the applaud code.
                            #'You'can'think'of'a'function'call'as'meaning'"go'to'and'run'that'code"
print(4)
```



```
# You can write each line by itself using a tab-indent.
# This is the more common way of writing functions in CoffeeScript.
applaud = ->
   print("clapping")  # Code INSIDE the function body
   print("clapping") - # · Code · INSIDE · the · function · body
print("Hello") # Code OUTSIDE the function body 7 7 7
applaud() # Code OUTSIDE the function body 7 7 7 7
```

```
# You can write functions that accept arguments or input
echo = (sound)-> "#" sound is a variable that only exists inside of the function. The value is set when the function is called.
   print(sound) ' # The sound variable dissapears once the function is complete.
   print(sound) ' # You can think of a function as an itty bitty program. When it runs it may create some variables.
¬¬¬¬¬¬++Andonce the function is done running it cleans up it's variables, just like a full program does.
¬¬¬¬¬¬+"Tobe technical, a variable that is part of a function declaration is called a "parameter".
¬¬¬¬¬¬"#'The'-data-'that'is'passed'to'the'parameter'during'the'function'call'is'called'the'"argument".
echo("Hello!") - # The string "Hello!" will be assigned to the parameter 'sound' when the echo function runs.
" " " " #'But'sound'will'only'equal'"Hello"'this'one time. " '''
echo("Goodbye!") * When the echo funtion runs this time, 'sound' will be set eugal to "Goodbye!"
```

```
#·When·we·can·call·our·function·with·an·argument,·we·can·use·CoffeeScript's·shortcut·like·normal.
echo = (sound)->
   print(sound)
   print(sound)
echo "Hello!" · ¬ ¬ # · Exactly · the · same · as · echo("Hello!")
echo "Goodbye!" -- # Exactly the same as echo ("Goodbye!")
```

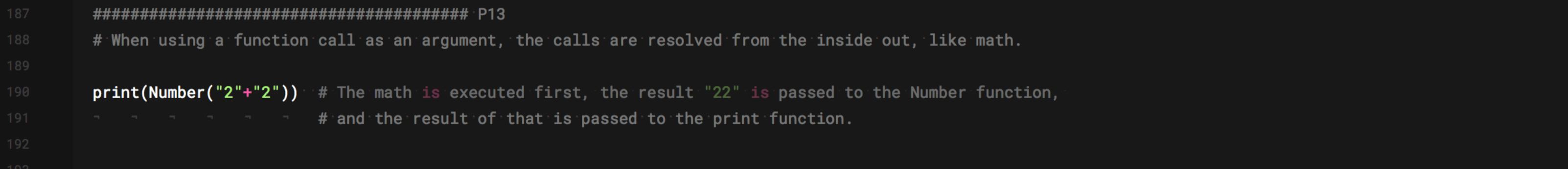
```
# Functions can be written to accept multiple arguments:
echo = (sound1, sound2) -> # 'sound1' and 'sound2' are both parameters.
   print(sound1) -
   print(sound2)
   print(sound1)
   print(sound2)
echo("Hello", "Sam")
```

```
# NOTE: You can pass more arguments to a function than it can accept
# The extra arguments will be ignored
echo = (sound1, sound2)->
   print(sound1)
   print(sound2)
   print(sound1)
   print(sound2)
echo("Hello","Sam","James","Bill") *# Only the first two arguments are used by the function. The second two are ignored
```

```
# Functions can accept objects as arguments.
# This is a super common activity when using Framer. We will talk about it more in the animation section
printArea = (obj)->
   print( obj.width * obj.height )
                              # "0,10,100" When printArea function runs
printArea({width:20,height:10})
                                          the 'obj' parameter in the function
                                          will refer to the object literal we create here.
```

```
# CoffeeScript shortcuts for objects and functions can be combined.
# This leads to very easy to write code, but the code may be ambiguous unless you
# understand the shortcuts.
printArea = (obj)->
   print( obj.width * obj.height )
printArea({width:20, height:10}) - # "0,10,100"
printArea { width: 20, height: 10} - # Exactly the same as above
printArea · ¬ ¬ ¬ ¬ ¬ # · Exactly · the · same · as · above
   width:1
   height:1
#'I'll avoid the shortcuts for a little longer.
```

174	#######################################
175	######################################
	#######################################
178	######################################
	#·Some·functions,·when·called,·turn·into,·or·"return",·data.
180	#·For·example, the built in function Number accepts primitive data, and returns a number.
181	
182	
183	<pre>print(Number("2")) ** # Number("2") * returns * the *number * data * '2'</pre>
184	
185	#·Function·calls·that·return·data·can·be·used·anywhere·you·use·data,·even·as·arguments.
186	



```
#'In'most'languages, you can return data from a custom function by using the "return" keyword.
bake = (ingredient)->
   return ingredient + " pie"
pie = bake("apple") # Try making different kinds of pies!
bake(pie)
print(pie) # "apple pie"
```

204	######################################
205	#'In'CoffeeScript, by default, a function returns whatever data is on the last line of its body.
206	#·This·can·save·us·some·typing:
207	
208	bake = (ingredient)->
209	ingredient:+:":pie":#:The:math:is:executed:and:the:results:are:automatically:returned
210	
	print(bake("apple"))
213	
214	
215	######################################
	######################################
	######################################