```
###
    
    The programs in this document cover:
    
        • Objects / Complex Data
        • Object Literals and Function Calls
        • References

###
```

```
########################################
######################################## Objects
########################################

######################################## P1
# In addition to primitive datatypes there are complex datatypes or 'data structures'.
# The most common and generic complex type is an 'Object'.
# You can think of it as a collection of variables.

pet = {}          # The curly brackets are literally an Object or an 'Object Literal'.
pet.name = "Sam"  # Create a variable, name, 'on the object'.
pet.type = "Cat"  # A variable on an object is called a 'property'.
                  # The Object.property notation is called 'Dot Notation' or 'Dot Syntax'.
                  # It is common in many languages.

# Properties work EXACTLY like variables. They are just variables grouped with an object.
print(pet.name)   # "Sam"
print(pet.type)   # "Cat"
pet.type = "Old cat"   # Overwrite the data in the type property
print(pet.type)   # "Old cat"
```

```python
######################################## P2
# Objects allow us to create structured data.

pet1 = {}
pet1.type = "Cat"
pet1.name = "Sam"

pet2 = {}
pet2.type = "Dog"
pet2.name = "Ralph"

print(pet1.type)    # "Cat"
print(pet1.name)    # "Sam"
print(pet2.type)    # "Dog"
print(pet2.name)    # "Ralph"


```

```
50  ######################################## P2a
51  # Layers are themselves complex objects.
52
53  box = new Layer()    # We'll discuss the 'new' keyword later.
54  box.x = 100
55  box.y = 50
56
57  print(box.x)    # 100
58  print(box.y)    # 50
59
60  # We'll come back to them later.
61
62
```

```
######################################## P3
# You can define an object's properties in the Object Literal.
# However, instead of using '=' we use ':'.
# This is called 'Object Literal Notation' opposed to 'Dot Notation' or 'Dot Syntax'.

pet = {name:"Sam",type:"Cat"}    # Each 'property:value' is called a 'property value pair'.
                                 # There are two 'property value pairs' in this object literal.
print(pet)  # {name:"Sam",type:"Cat"}

# You might expect to use '=' instead of ':' for object literals.
# The reasons for the using ':' are partly historical and partly because of
# small distinctions between -assigning- data and -creating- data.
```

```
77  ######################################## P4
78  # You can create nested object structures or hierarchies.
79
80  pet = {name:"Sam Jr."}
81  pet.parent = {name:"Sam Sr."}
82  pet.parent.parent = {name:"Ol papa Sam"}
83
84  print(pet.parent.name)          # "Sam Sr."
85  print(pet.parent.parent.name)   # "Ol papa Sam."
86
87
```

```python
######################################## P5
# You can also create objects with nested structures on one line using 'Object Literal Notation'

pet = {name:"Sam",type:"Cat",age:3,parent:{name: "Sam Sr."}}

print(pet.name)      # "Sam"
print(pet.type)      # "Cat"
print(pet.age)       # 3
print(pet.parent.name) # "Sam Sr."

# Above, pet refers to an object with a property parent that refers
# to ANOTHER object with a property name.

```

```
101 ######################################## P6
102 # Here's an example of a deeply nested complex data structure.
103
104 arm = {lowerArm:{hand:{fingers:5}}}
105
106 print(arm.lowerArm.hand.fingers) # 5
107
108
```

```
109 ########################################
110 ######################################## Object Literals and Function Calls
111 ########################################
112
113 ######################################## P7
114 # Some functions accept objects as arguments
115
116 pet = {name:"Mike Hat",type:"Dog"}
117 print(pet) # {name:"Mike Hat",type:"Dog"}
118
119
```

```
120 ######################################## P8
121 # We can also use an Object Literal directly as an argument
122
123 print({name:"Mike Hat",type:"Dog"}) # {name:"Mike Hat",type:"Dog"}
124
125
126
127
```

```
128 ########################################
129 ######################################## Built in variables
130 ########################################
131
132 ######################################## P9
133 # Framer and JavaScript come with many built in variables.
134 # They are often organized into objects.
135 # In most cases, you can not put data in them, you can only read their values.
136 # Here are two useful ones.
137
138 print(Screen.width)     # depends on your output window width
139 print(Screen.height)    # depends on your output window height
140
141
```

```coffeescript
########################################
######################################## [Optional] Object literal shorthand
########################################

######################################## P10
# Nested object structures are not uncommon in CoffeeScript.
# Putting them on one line is messy though.
# So CoffeeScript has a shorthand.
# It looks like this:

course =
    students:18
    room:5221
    instructor:
        name:"Bob"
        age:81

print(course) # {students:18, room:5221, instructor:{name:"Bob", age:81}}

print(course.students)      # 18
print(course.room)          # 5221
print(course.instructor)    # {name:"Bob", age:81}
print(course.instructor.name)   # "Bob"

# Instead of curly brackets we start the object on a new line and indent.
# property:value pairs at the same level of indentation are part of the same object.
# Indenting further creates a new nested object.
# This is another example of 'significant whitespace'
```

```
######################################## P11
# Here's another example:

# This statement...
pet = {name:"Sam",type:"Cat",age:3}

# is the same as this statement...
pet =                # variable declaration and assignment, the line return indicates 'start new object'
    name:"Sam"       # a property value pair, and the beginning of the object
    type:"Cat"       # another property value pair. It belongs to the same object
    age:3            # another
    parent:          # another property, but the value is...
        name: "Sam Sr."    # A NEW object

print(pet.name)          # "Sam"
print(pet.type)          # "Cat"
print(pet.age)           # 3
print(pet.parent.name)   # "Sam Sr."
```

```
193 ######################################## P12
194 # We can also use an Object Literal Shorthand when calling functions.
195
196 print(
197     name:"Mike Hat"     # The line return AND the tab together indicate 'new object literal'
198     type:"Dog")
199                         # {name:"Mike Hat", type:"Dog"}
200
201 # This program is EXACTLY the same as P8.
202
```

```
######################################## P13
# We can combine object literal shorthand with function shorthand.
# This is very common.

print                   # The line return and tab indicate a new object literal.
    name:"Mike Hat"     # Because there is an argument, the print function needs no parenthesis.
    type:"Dog"
                        # {name:"Mike Hat", type:"Dog"}

# This program is EXACTLY the same as P8 and P12.
# We call the function print and pass it an object with two properties.

```

```
215 ########################################
216 ######################################## [Optional] References (Tricky!)
217 ########################################
218
219 ######################################## P14
220 # Assigning Objects to variables is DIFFERENT from assigning primitive data to variables.
221 # Object data is always given it's OWN UNIQUE memory area.
222 # When an Object is assigned to a variable, the variable 'refers to' this memory.
223 # The variable has a 'reference'. A reference is NOT the data. It only 'refers to' the data.
224 # This means multiple variables can 'refer to' the SAME Object Data.
225
226
227 clark = {power:"flight"}    # Create an object literal with property 'power' that holds data "flight" and assign a REFERENCE to this object to the
    variable 'clark'.
228 superman = clark            # Copy the clark's REFERENCE to the above object into the new variable 'superman'
229 superman.weakness = "kryptonite" # Create new property 'weakness' on the object data above.
230
231 print(superman.power)       # "flight"
232 print(clark.power)          # "flight"
233 print(superman.weakness)    # "kryptonite"
234 print(clark.weakness)       # "kryptonite"
235                             # Clark and Superman 'refer to' the SAME object!
236
237
```

```
238 ######################################## P14a
239 # Here's an example using Layers.
240
241 box = new Layer()          # make a Layer
242 theOnlyBox = box           # theOnlyBox now refers to our original layer.
243 myFavoriteBox = box        # myFavoriteBox now refers to our original layer.
244                            # There is still only one layer, just three variables refering to it.
245
246 myFavoriteBox.backgroundColor = "red" # We modify layer data.
247 box = "nothing"            # the Layer data still exists. box just doesn't refer to it anymore.
248
249
```

```
250 ######################################## P15
251 # Note the difference when using primitive data.
252
253 clark = "kryptonian"    # Assign data to variable 'clark'
254 superman = clark        # COPY data 'kryptonian' to variable 'superman'
255 clark = "Kent"          # Assign new data to variable 'clark'.
256                         # superman variable unaffected.
257
258 print(clark)            # "Kent"
259 print(superman)         # "kryptonian"
260
261
```

```
######################################## P16
# If we assign primitive data to a variable that has a REFERENCE, the REFERENCE is overwritten,
# but the original object data will persist in memory

clark = {power:"flight"}
superman = clark
clark = "Kent"  # This overwrites the -reference- NOT the -object data-.
                # The object still exists.
                # The variable superman still refers to it.

print(clark)    # "Kent"
print(superman) # {power:"flight"}
```

```
276 ######################################## P16b
277 # An example of the above using Layers
278
279 box = new Layer()
280 box = "nothing"     # No variables refer to the layer data, but it still exists!
281                     # The layer data is still in memory and in our document even though we can't refer to it!
282
283
284 ########################################
285 ######################################## End
286 ########################################
287
```