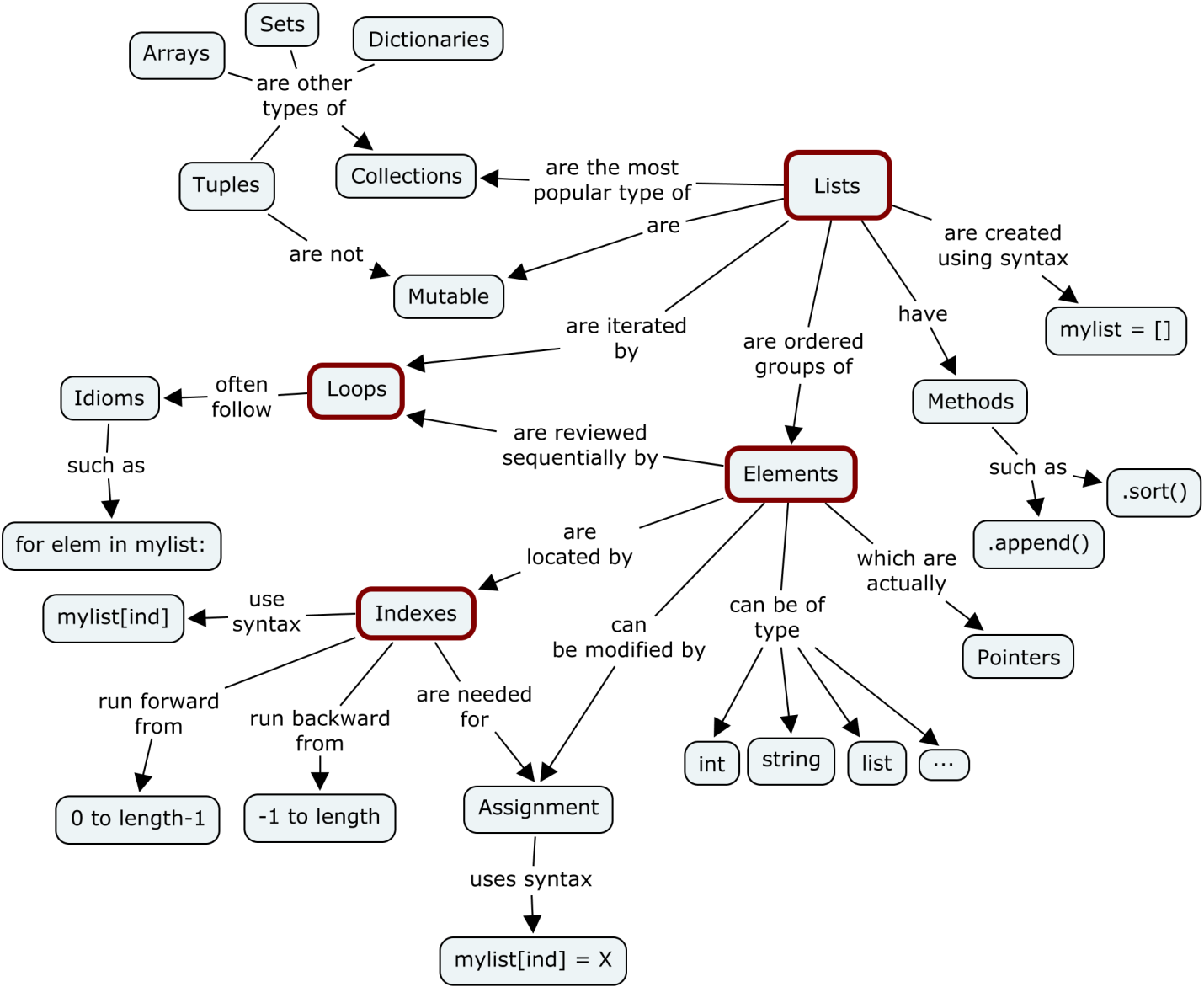
Mutable and Immutable Objects

Everything in Python is an object. And what every newcomer to Python should quickly learn is that all objects in Python can be either mutable or immutable.



Objects of built-in types like (int, float, bool, str, tuple, unicode) are immutable. Objects of built-in types like (list, set, dict) are mutable. Custom classes are generally mutable. To simulate immutability in a class, one should override attribute setting and deletion to raise exceptions.

Now comes the question, how do we find out if our variable is a mutable or immutable object. For this we should understand what ‘ID’ and ‘TYPE’ functions are for.

**ID and TYPE**

The built-in function **id**() returns the identity of an object as an integer. This integer usually corresponds to the object’s location in memory, although this is specific to the Python implementation and the platform being used. The **is** operator compares the identity of two objects.

The built-in function type() returns the type of an object. Let’s look at a simple example

''' Example 1 '''  
>>> x = "Holberton"  
>>> y = "Holberton"  
>>> id(x)  
140135852055856  
>>> id(y)  
140135852055856  
>>> print(x is y) '''comparing the types'''  
True''' Example 2 '''  
>>> a = 50  
>>> type(a)  
<class: ‘int’>  
>>> b = "Holberton"  
>>> type(b)  
<class: 'string'>

We have now seen how to compare two simple string variables to find out the types and id’s .So using these two functions, we can check to see how different types of objects are associated with variables and how objects can be changed .

# ****Mutable and Immutable Objects****

So as we discussed earlier, a mutable object can change its state or contents and immutable objects cannot.

***Mutable objects****:*

list, dict, set, byte array

***Immutable objects*:**

int, float, complex, string, tuple, frozen set [note: immutable version of set], bytes

**Example**

A practical example to find out the mutability of object types

x = 10x = y

We are creating an object of type int. identifiers x and y points to the same object.

id(x) == id(y)id(y) == id(10)

if we do a simple operation.

x = x + 1

Now

id(x) != id(y)id(x) != id(10)

The object in which x was tagged is changed. object 10 was never modified. **Immutable objects doesn’t allow modification after creation**

In the case of **mutable objects**

m = list([1, 2, 3])n = m

We are creating an object of type list. identifiers m and m tagged to the same list object, which is a collection of 3 immutable int objects.

id(m) == id(n)

Now poping an item from list object does change the object,

m.pop()

object id will not be changed

id(m) == id(n)

m and n will be pointing to the same list object after the modification. The list object will now contain [1, 2].