Assessment - Object-oriented programming

In this exercise, we'll create a few classes to simulate a server that's taking connections from the outside and then a load balancer that ensures that there are enough servers to serve those connections.

To represent the servers that are taking care of the connections, we'll use a Server class. Each connection is represented by an id, that could, for example, be the IP address of the computer connecting to the server. For our simulation, each connection creates a random amount of load in the server, between 1 and 10.

Run the following code that defines this Server class.

In [1]:
```
#Begin Portion 1#
import random

class Server:
    def __init__(self):
        """Creates a new server instance, with no active connections."""
        self.connections = {}

    def add_connection(self, connection_id):
        """Adds a new connection to this server."""
        connection_load = random.random()*10+1
        # Add the connection to the dictionary with the calculated load
        self.connections[connection_id] = connection_load

    def close_connection(self, connection_id):
        """Closes a connection on this server."""
        # Remove the connection from the dictionary
        if connection_id in self.connections:
            del self.connections[connection_id]

    def load(self):
        """Calculates the current load for all connections."""
        total = 0
        # Add up the load for each of the connections
        for load in self.connections.values():
            total += load
        return total

    def __str__(self):
        """Returns a string with the current load of the server"""
        return "{:.2f}%".format(self.load())

#End Portion 1#
```

Now run the following cell to create a Server instance and add a connection to it, then check the load:

In [2]:
```
server = Server()
server.add_connection("192.168.1.1")

print(server.load())
```

```
10.421441070930666
```

After running the above code cell, if you get a **NameError** message, be sure to run the Server class definition code block first.

The output should be 0. This is because some things are missing from the Server class. So, you'll need to go back and fill in the blanks to make it behave properly.

Go back to the Server class definition and fill in the missing parts for the `add_connection` and `load` methods to make the cell above print a number different than zero. As the load is calculated randomly, this number should be different each time the code is executed.

Hint: Recall that you can iterate through the values of your connections dictionary just as you would any sequence.

Great! If your output is a random number between 1 and 10, you have successfully coded the `add_connection` and `load` methods of the Server class. Well done!

What about closing a connection? Right now the `close_connection` method doesn't do anything. Go back to the Server class definition and fill in the missing code for the `close_connection` method to make the following code work correctly:

In [4]:
```
server.close_connection("192.168.1.1")
print(server.load())
```

```
0
```

You have successfully coded the `close_connection` method if the cell above prints 0.

Hint: Remember that `del` dictionary[key] removes the item with key *key* from the dictionary.

Alright, we now have a basic implementation of the server class. Let's look at the basic LoadBalancing class. This class will start with only one server available. When a connection gets added, it will randomly select a server to serve that connection, and then pass on the connection to the server. The LoadBalancing class also needs to keep track of the ongoing connections to be able to close them. This is the basic structure:

In [4]:
```
#Begin Portion 2#
class LoadBalancing:
    def __init__(self):
        """Initialize the load balancing system with one server"""
        self.connections = {}
        self.servers = [Server()]

    def add_connection(self, connection_id):
        """Randomly selects a server and adds a connection to it."""
        server = random.choice(self.servers)
        # Add the connection to the dictionary with the selected server
        self.connections[connection_id] = connection_load

        # Add the connection to the server

    def close_connection(self, connection_id):
        """Closes the connection on the the server corresponding to connection_id."""
        # Find out the right server
        # Close the connection on the server
        # Remove the connection from the load balancer

    def avg_load(self):
        """Calculates the average load of all servers"""
        # Sum the load of each server and divide by the amount of servers
        return 0

    def ensure_availability(self):
        """If the average load is higher than 50, spin up a new server"""
        pass

    def __str__(self):
        """Returns a string with the load for each server."""
        loads = [str(server) for server in self.servers]
        return "[{}]".format(",".join(loads))
#End Portion 2#
```

As with the Server class, this class is currently incomplete. You need to fill in the gaps to make it work correctly. For example, this snippet should create a connection in the load balancer, assign it to a running server and then the load should be more than zero:

In [5]:
```
l = LoadBalancing()
l.add_connection("fdca:83d2::f20d")
print(l.avg_load())
```

```
0
```

After running the above code, the output is 0. Fill in the missing parts for the `add_connection` and `avg_load` methods of the LoadBalancing class to make this print the right load. Be sure that the load balancer now has an average load more than 0 before proceeding.

What if we add a new server?

In [6]:
```
l.servers.append(Server())
print(l.avg_load())
```

```
0
```

The average load should now be half of what it was before. If it's not, make sure you correctly fill in the missing gaps for the `add_connection` and `avg_load` methods so that this code works correctly.

Hint: You can iterate through the all servers in the *self.servers* list to get the total server load amount and then divide by the length of the *self.servers* list to compute the average load amount.

Fantastic! Now what about closing the connection?

In [7]:
```
l.close_connection("fdca:83d2::f20d")
print(l.avg_load())
```

```
0
```

Fill in the code of the LoadBalancing class to make the load go back to zero once the connection is closed.

Great job! Before, we added a server manually. But we want this to happen automatically when the average load is more than 50%. To make this possible, fill in the missing code for the `ensure_availability` method and call it from the `add_connection` method after a connection has been added. You can test it with the following code:

In [8]:
```
for connection in range(20):
    l.add_connection(connection)
print(l)
```

```
[0.00%,0.00%]
```

The code above adds 20 new connections and then prints the loads for each server in the load balancer. If you coded correctly, new servers should have been added automatically to ensure that the average load of all servers is not more than 50%.

Run the following code to verify that the average load of the load balancer is not more than 50%.

In [9]:
```
print(l.avg_load())
```

```
0
```

Awesome! If the average load is indeed less than 50%, you are all done with this assessment.