Lab 3

This lab requires the implementation of functions that use boolean logic and functions that use conditional (`if`) statements.

Download lab3 files and place it in your `cpe101` directory. This includes three subdirectories (corresponding to the different parts of the lab).

Object Equality

As experienced in the previous lab, checking object equality by comparing each field individually is tedious. Though individual attribute comparisons are necessary to properly test the `__init__` function, once object creation is known to work we would prefer to compare objects for equality in a simpler manner. This can be done by defining the `__eq__` function within a class. We will use a simple definition of `__eq__` to reduce the tedium of writing test cases (more sophisticated checks may be introduced in CPE 102).

When an object is compared to another value using `==`, the default behavior is check if the two values are actually the same object (i.e., this is typically referred to as reference equality because both operands must refer to the same object for the check to return `True`). Instead, if you define the `__eq__` function in the object's class, then that function will be called when `==` is used. As such, the `__eq__` can define what it means for the object to be equal to some other value (e.g., they may be considered equal only when each of their attributes is equal).

`__eq__`

In the `object_equality` directory you will modify the `point.py` and `object_equality_tests.py` files.

Modify the definition of the `Point` class to add (a simplified) `__eq__` function. This function will take two arguments: `self` (the target of the call, much like with `__init__`) and `other` (the other operand). The function must return `True` when the `x` and `y` attributes in `self` are equal to the `x` and `y` attributes in `other`. Your function will assume that `other` has these attributes.

The attributes for a `Point` are typically of a floating point type. As such, you should use the `epsilon_equal` defined in `utility.py` when writing your `__eq__` function.

Tests

Modify `object_equality_tests.py` to test that your implementation of `__eq__` behaves as expected. You can do so by using `assertEqual` to compare two `Point` objects.

`__ne__`

Note that defining `__eq__` does not change the behavior of the `!=` operator. To do this you must define the `__ne__` function for the class. Doing so is not required for this lab, but you might try defining `__ne__` once you have completed the required portions of the lab.

Boolean Logic

In the `logic` directory create a file named `logic.py`. This part of the lab requires that you implement and test multiple functions that compute boolean values (similar to `is_positive` from the previous lab). You should develop these functions and their tests one at a time. The function implementations must be placed in `logic.py`. The test cases will, of course, be placed in the provided `logic_tests.py`.

You **must** write each of the following functions **without** using any sort of conditional (`if`) statement.

You must provide at least two test cases for each of these functions though you should really provide enough test cases to ensure that the function works even for edge cases.

This part will be executed with: **python logic_tests.py**

is_even

Write a function, named `is_even`, that takes a single argument assumed to be an integer and that returns `True` when the integer is even.

There are many ways that one can implement this function; as one example, you should explore the remainder/modulus operator (`%`).

in_an_interval

Write a function, named `in_an_interval`, that takes a single number argument and returns `True` when the argument falls in one of the following intervals (recall that a square bracket indicates inclusivity whereas a parenthesis indicates exclusivity; e.g., the interval [2,9) includes 2 but not 9). The intervals are [2,9), (47,92), (12, 19], and [101,103].

This function is meant as an exercise without any clear real-world analog. Think carefully about the test cases that you should write to verify that this function works (even though only two are required).

Functions with If Statements

This part of the lab is similar to the previous part, but the functions for this part will use conditional (`if`) statements.

In the `conditional` directory create a file named `conditional.py`. Place your test cases in the provided `conditional_tests.py` file.

You must provide at least two test cases for each of these functions, but you should really provide enough test cases to ensure that every path through the function is properly tested.

This part will be executed with: **python conditional_tests.py**

max_101

Write a function, named `max_101`, that takes two numbers as arguments and returns the largest of the two values. (Note, this function is actually already provided in Python as `max`. Do not use the provided function; reason through the logic yourself).

max_of_three

Write a function, named `max_of_three`, that takes three arguments of type `float` and returns the largest of the three values. You should write this using `if` statements for the practice, but give consideration to how you might write this using the `max_101` function above. (Again, do not use the built-in `max` function.)

rental_late_fee

Write a function, named `rental_late_fee`, that takes a single argument representing the number of days late a rental item is returned. This function will return the number of dollars (represented as an integer in this example) for the assessed late fee as defined by the following table.

Days	Fee
≤ 0	0
≤ 9	5
≤ 15	7
≤ 24	19
> 24	100

Demonstration

Demonstrate the test cases from the each part of the lab to your instructor to have this lab recorded as completed. In addition, be prepared to show your instructor the source code for functions in `logic.py`, `point.py`, and `conditional.py`.