A Better Box

The previous exercise demonstrated how to define Classes, and create and operate on class instances. Each new class is considered a new type: the class instances are its values and the methods are the operations that can be used to manipulate these values. For consistency, we would like to use standard Python functions and operators with values of a new type. This exercise shows how this is done in Python using the concepts of overloading and introspection.

**Part (a):** Python uses mapping rules to rewrite calls of standard functions (e.g., `str`) and operators (e.g., `+` and `*`) as calls on methods with special names.\(^1\) If a class contains definitions for methods with these names, then Python uses the definitions to overload the corresponding functions and operators.

In this part of this exercise, you will make the `Box` class from the last exercise easier to use by overloading some standard functions and operators.

1. Download `box.py` to a new directory for this exercise. Load it into Spyder and run it. Discuss with your partner what statement in the program is producing each line displayed in the shell. Run it a second time. What differences do you notice in the output produced by the two runs? What do you think could account for these differences?

2. For Python to know how you want it to print instances of a new type, you need to overload the string conversion function, `str`.\(^2\)

The mapping rule for the `str` function is as follows: *If the type of `exp` is not one of the standard Python types, then Python maps `str(exp)` to `exp.__str__()`.*

For example, if `b` is an instance of `Box`, then Python maps `str(b)` to `b.__str__()`. Because of this rule, you can overload the `str` function by defining a method named `__str__` in the `Box` class. This method must have a single parameter (`self`) and it must return a string (the string you want Python to display when printing a `Box` value).

Your `Box` class already defines a method that returns such a string: `getStr`. To overload the `str` function so it returns this string when called with a `Box` value, change the name of the method to be `__str__`. Do that now. Then rerun the program. Check that the program displays the expected strings.

3. Evaluate `b1` in the shell. From this experiment you can see that Python uses different conventions for printing an object and for displaying an object in the shell.

---

\(^1\) The names all start and end with two underscores (e.g., `'__'`).

\(^2\) Python calls the string conversion function, `str`, on each expression passed to the `print` function and displays the string returned.
In the shell, Python calls `repr(exp)` to know how to display `exp`. By convention, it displays the string returned by this call *without* the string delimiters. For example:

```
In [24]: repr(5)
Out[24]: '5'
In [25]: 5
Out[25]: 5
In [26]: repr('hi')
Out[26]: "'hi'
In [27]: 'hi'
Out[27]: 'hi'
```

So to affect how instances of a new type are displayed when evaluated in the shell, you need to overload the `repr` function.

The mapping rule for the `repr` function is as follows: *If the type of `exp` is not one of the standard Python types, then Python maps `repr(exp)` to `exp.__repr__()`.*

Because of this rule, you can overload the `repr` function by defining a method named `__repr__` in the `Box` class. This method must have a single parameter (`self`) and it must return a string.

Add a `__repr__` method to the `Box` class. Have it return the same string as `__str__`. But be smart about it: *don't copy the body of the `__str__` method*; instead, call it! (Why should you call it instead of copying its body?)

Test it out by re-running your program and then again evaluating `b1` in the shell. Also evaluate both `str(b1)` and `repr(b1)` in the shell. Why do `str(b1)` and `repr(b1)` both display with quotes around them, but `b1` does not?

4. Next you will overload the `+` operator. But first, evaluate `b1 + b2` in the shell to see that it produces a `TypeError`.

`Overloading a binary operator can require two mappings: a “normal” mapping and a “reverse” mapping. Python uses the normal mapping for a binary operator unless the first argument is a standard Python type or doing so produces an exception, in which case it uses the reverse mapping.

The normal mapping of `exp1 + exp2` is `exp1.__add__(exp2)`.

For example, the normal mapping of `b1 + b2` is `b1.__add__(b2)`.* Because of this mapping, you can overload the `+` operator to work with `Box` values by defining a method called `__add__` in the `Box` class. This method must have exactly two parameters. It
should not modify either of its parameters, and it should return a **Box** value (the sum of the arguments).

We already have a method that meets these requirements: **add**. To overload the `+` operator, change the method name to be `__add__`. Do that now. Rerun the program.

Why does the third assignment statement now produce an **AttributeError**? Change this assignment statement to: **b3 = b1 + b2**. Then rerun the program and check that it produces the expected output.

5. Next, you will overload the `*` operator using the normal mapping for multiplication:

   \[
   \text{The normal mapping of } \exp1 \ast \exp2 \text{ is } \exp1.\_\_mul\_\_ (\exp2).
   \]

   The `\_\_mul\_\_` method must have two parameters. For consistency, it should not modify either of its arguments and it should return a value (the product of the arguments).

   We have a method that multiplies a **Box** by a scalar: **mult**. You can overload `*` by renaming this function to `\_\_mul\_\_`. Do that now. Then rerun the program.

   Why does the fourth assignment statement now produce an **AttributeError**? Rewrite the expression in this assignment statement to use the `*` operator, rerun the program, and check that the program displays the correct output.

6. There is a problem with our overloaded `*` operator: Multiplication should be commutative. That is, **1.5 * b2** should return the same value as **b2 * 1.5**. Evaluate **1.5 * b2** in the shell. Why doesn’t Python use the normal mapping with this expression?

   Reverse mappings provide a way to solve this problem. Python uses a reverse mapping if the normal mapping produces an exception or the type of the second argument is not a standard Python type.

   \[
   \text{The reverse mapping of } \exp1 \ast \exp2 \text{ is } \exp2.\_\_rmul\_\_ (\exp1).
   \]

   For example, the reverse mapping of **1.5 * b2** is **b2.\_\_rmul\_\_ (1.5)**. Add an `\_\_rmul\_\_` method to the **Box** class. Define it so that scalar multiplication is commutative—that is, so that evaluating **exp1 * exp2** returns the same **Box** value as evaluating **exp2 * exp3**. Be smart about it: **don't copy the body of the \_\_mul\_\_ method**; instead, call it! (Why should you call it instead of copying its body?)

   To test it, un-comment the assignment and print statement under the comment
   ```python
   # For use in Step 6 of Part (a), run the program, and check that b4 and b5 have the same coordinates and dimensions.
   ```
7. Un-comment the assignment and three print statements under the comment

    # For use in Step 7 of Part (a). Rerun the program. What does this tell you
    about the default behavior of the == operator when the types of its operands are not
    standard Python types?

To make your * operator truly commutative, you need to override ==. The mapping rule
for == is as follows: If the type of exp1 is not one of the standard Python types, then
Python maps exp1 == exp2 to exp1.__eq__(exp2).

Because of this rule, you can overload the == operator by defining a method named
__eq__ in the Box class. This method must have two parameters and it must return a
bool value (i.e., True or False).

Add a definition for a __eq__ method. The method should return True if called with
Box values that have equal x-coordinates, equal y-coordinates, equal heights, and equal
lengths; otherwise, it should return False.

Run the program and check that b4 == b5 now returns True.

Overloading the == operator also implicitly overloads the != operator. Test this out in
the shell. For example, what is returned by b4 != b5? How about b2 != b5? How
do you think Python determines the value to return for these Boolean expressions?
Part (b): In this part, you will see how to use introspection to make your Box class more consistent with other Python types. But first, you need to become familiar with two Python functions that support introspection: `type` and `isinstance`.

1. **`type(exp)`** returns the type of `exp`. Your program can use it to check if an expression is of an expected type.

   Consider, for example, the following function definition:

   ```python
   def what_type(param):
       if type(param) == int:
           print('got an int')
       elif type(param) == float:
           print('got a float')
       elif type(param) == Box:
           print('got a box')
       else:
           print('did not get an int, float, or Box')
   ```

   Assume this function definition is added to the program and the program is rerun. Predict what will be displayed by the following function calls:

   ```
   what_type(5)
   what_type(b1)
   what_type('hi')
   ```

2. **`isinstance(exp, type-or-tuple_of_types)`** is called with an expression as the 1st argument and either a type or a tuple of types as the 2nd; and returns `True`, if the 1st argument is an instance of a type in the 2nd argument, or `False`, otherwise.

   Consider, for example, the following function definition:

   ```python
   def is_num(param):
       return isinstance(param, (int, float))
   ```

   Assume this function definition is added to the program and the program is rerun. Predict what will be returned by the following function calls:

   ```
   is_num(5)
   is_num(b1)
   is_num('hi')
   ```
3. Armed with introspection, you will now revisit the overloaded `==` operator which you created in part (a). In the shell, evaluate `'hi' == 1`. Now evaluate `b1 == 1`. Why does the latter raise an exception?

For consistency with Python’s `==` operator, what should be returned by evaluating `b1 == 1`?

Your `__eq__` method can use introspection to make the overloaded `==` operator consistent with Python’s `==` operator. Python’s mapping rules guarantee that, in the body of `__eq__`, the first parameter, `self`, references a `Box` value. So the `__eq__` method should check if the other parameter is an instance of `Box`. If not, the method should return `False`; on the other hand, if it is, the method should return `True`, if the two `Box` values have equal x-coordinates, equal y-coordinates, equal heights, and equal lengths, and `False`, if not.

Modify your `__eq__` method accordingly. Test that both `==` and `!=` now allow you to compare a `Box` value with values of other types. For example:

```
In [24]: b1 == 3
Out[24]: False
In [25]: 3.1 == b1
Out[25]: False
In [26]: b1 + b2 == 'hi'
Out[26]: False
In [27]: b1 != 0.0
Out[27]: True
In [28]: 0 != 2 * b1
Out[28]: True
```

4. In the shell, evaluate the following:
   ```python
   b7 = Box('hi')
   print( b7 )
   ```

   Explain why the exception occurs.

   What would be more consistent behavior with how other types work? (Consider, for example, what happens if you evaluate `int('hi')`)

   Read the documentation for the `Box` constructor. Software engineers use “Requires” comments to document expectations of valid arguments. In the case of a `Box`, all arguments should be numeric (`int` or `float`) and `length` and `height` should be positive. The constructor should check that these conditions are met; to be consistent with other type constructors, the constructor should raise a `ValueError` if it is called with illegal arguments.

   Using introspection, modify the constructor so that it raises a `ValueError` when called with arguments that do not satisfy the documented requirements.
Run some tests in the console to verify that the modified constructor raises a `ValueError` if the user attempts to create an illegal `Box` instance. For example:

```
In [35]: Box('hi')
Traceback (most recent call last):

   File "<ipython-input-35-7a8bda6b997b>" , line 1 , in <module>
     Box('hi')

   File "/Users/lauradillon/Dropbox/291-PLTL/Exercises/Classes/Advanced/boxSolution.py" , line 13 , in __init__
     raise ValueError ( err.format(x_coord, y_coord, length, height))

ValueError: invalid arguments for Box(): hi, 0, 100, 100

In [36]: Box(length=0)
Traceback (most recent call last):

   File "<ipython-input-36-5679330f5b92>" , line 1 , in <module>
     Box(length=0)

   File "/Users/lauradillon/Dropbox/291-PLTL/Exercises/Classes/Advanced/boxSolution.py" , line 17 , in __init__
     raise ValueError ( err.format(length))

ValueError: invalid argument for Box(): 0 (length must be positive)
```
5. In the console, evaluate `b1 + 1`.

Explain why the exception occurs.

What would be more consistent behavior with how other types work? (Consider, for example, what happens if you evaluate `1 + 'hi'`.)

Using introspection, modify `__add__` so that it raises a `TypeError` if the second argument is not a `Box` instance.

Run some tests in the console to check that the operator behaves as intended. For example:

```
In [48]: b1 + 0
Traceback (most recent call last):
  File "<ipython-input-48-48292014ab0d>" , line 1, in <module>
    b1 + 0
  File "~/Users/lauradillon/Dropbox/PLTL/Exercises/Classes/Advanced/boxSolution.py" , line 37, in __add__
    raise TypeError (err.format("Box", type(other)))
TypeError: unsupported operand type(s) for +: 'Box' and '<class 'int'>'
```

```
In [49]: b1 + b2 + 'hi'
Traceback (most recent call last):
  File "<ipython-input-49-678a65d73a28>" , line 1, in <module>
    b1 + b2 + 'hi'
  File "~/Users/lauradillon/Dropbox/PLTL/Exercises/Classes/Advanced/boxSolution.py" , line 37, in __add__
    raise TypeError (err.format("Box", type(other)))
TypeError: unsupported operand type(s) for +: 'Box' and '<class 'str'>'
```

6. As time permits, modify the other operator definitions so that their behavior is consistent with the other standard operators.