[Index | Exercise 2.3 | Exercise 2.5]

Exercise 2.4

Objectives:

- Make a new primitive type

In most programs, you use the primitive types such as `int`, `float`, and `str` to represent data. However, you're not limited to just those types. The standard library has modules such as the `decimal` and `fractions` module that implement new primitive types. You can also make your own types as long as you understand the underlying protocols which make Python objects work. In this exercise, we'll make a new primitive type. There are a lot of little details to worry about, but this will give you a general sense for what's required.

(a) Mutable Integers

Python integers are normally immutable. However, suppose you wanted to make a mutable integer object. Start off by making a class like this:

```
# mutint.py

class MutInt:
    __slots__ = ['value']

    def __init__(self, value):
        self.value = value
```

Try it out:

```
>>> a = MutInt(3)
>>> a
<__main__.MutInt object at 0x10e79d408>
>>> a.value
3
>>> a.value = 42
>>> a.value
42
>>> a + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'MutInt' and 'int'
>>>
```

That's all very exciting except that nothing really works with this new `MutInt` object. Printing is horrible, none of the math operators work, and it's basically rather useless. Well, except for the fact that its value is mutable--it does have that.

(b) Fixing output

You can fix output by giving the object methods such as `__str__()`, `__repr__()`, and `__format__()`. For example:

```
# mint.py

class MutInt:
    __slots__ = ['value']

    def __init__(self, value):
        self.value = value

    def __str__(self):
        return str(self.value)

    def __repr__(self):
        return f'MutInt({self.value!r})'

    def __format__(self, fmt):
        return format(self.value, fmt)
```

Try it out:

```
>>> a = MutInt(3)
>>> print(a)
3
>>> a
MutInt(3)
>>> f'The value is {a:*^10d}'
The value is ****3*****
>>> a.value = 42
>>> a
MutInt(42)
>>>
```

(c) Math Operators

You can make an object work with various math operators if you implement the appropriate methods for it. However, it's your responsibility to recognize other types of data and implement the appropriate conversion code. Modify the `MutInt` class by giving it an `__add__()` method as follows:

```
class MutInt:
    __slots__ = ['value']
    def __init__(self, value):
        self.value = value

    ...

    def __add__(self, other):
        if isinstance(other, MutInt):
            return MutInt(self.value + other.value)
        elif isinstance(other, int):
            return MutInt(self.value + other)
        else:
            return NotImplemented
```

With this change, you should find that you can add both integers and mutable integers. The result is a `MutInt` instance. Adding other kinds of numbers results in an error:

```
>>> a = MutInt(3)
>>> b = a + 10
>>> b
MutInt(13)
>>> b.value = 23
>>> c = a + b
>>> c
MutInt(26)
>>> a + 3.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'MutInt' and 'float'
>>>
```

One problem with the code is that it doesn't work when the order of operands is reversed. Consider:

```
>>> a + 10
MutInt(13)
>>> 10 + a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'MutInt'
>>>
```

This is occurring because the `int` type has no knowledge of `MutInt` and it's confused. This can be fixed by adding an `__radd__()` method. This method is called if the first attempt to call `__add__()` didn't work with the provided object.

```
class MutInt:
    __slots__ = ['value']
    def __init__(self, value):
        self.value = value

    ...

    def __add__(self, other):
        if isinstance(other, MutInt):
            return MutInt(self.value + other.value)
        elif isinstance(other, int):
            return MutInt(self.value + other)
        else:
            return NotImplemented

    __radd__ = __add__    # Reversed operands
```

With this change, you'll find that addition works:

```
>>> a = MutInt(3)
>>> a + 10
MutInt(13)
>>> 10 + a
MutInt(13)
>>>
```

Since our integer is mutable, you can also make it recognize the in-place add-update operator `+=` by implementing the `__iadd__()` method:

```
class MutInt:
    __slots__ = ['value']

    def __init__(self, value):
        self.value = value

    ...

    def __iadd__(self, other):
        if isinstance(other, MutInt):
            self.value += other.value
            return self
        elif isinstance(other, int):
            self.value += other
            return self
        else:
            return NotImplemented
```

This allows for interesting uses like this:

```
>>> a = MutInt(3)
>>> b = a
>>> a += 10
>>> a
MutInt(13)
>>> b                     # Notice that b also changes
MutInt(13)
>>>
```

That might seem kind of strange that `b` also changes, but there are subtle features like this with built-in Python objects. For example:

```
>>> a = [1,2,3]
>>> b = a
>>> a += [4,5]
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]

>>> c = (1,2,3)
>>> d = c
>>> c += (4,5)
>>> c
(1, 2, 3, 4, 5)
>>> d                      # Explain difference from lists
(1, 2, 3)
>>>
```

(d) Comparisons

One problem is that comparisons still don't work. For example:

```
>>> a = MutInt(3)
>>> b = MutInt(3)
>>> a == b
False
>>> a == 3
False
>>>
```

You can fix this by adding an `__eq__()` method. Further methods such as `__lt__()`, `__le__()`, `__gt__()`, `__ge__()` can be used to implement other comparisons. For example:

```
class MutInt:
    __slots__ = ['value']
```

```python
    def __init__(self, value):
        self.value = value

    ...
    def __eq__(self, other):
        if isinstance(other, MutInt):
            return self.value == other.value
        elif isinstance(other, int):
            return self.value == other
        else:
            return NotImplemented

    def __lt__(self, other):
        if isinstance(other, MutInt):
            return self.value < other.value
        elif isinstance(other, int):
            return self.value < other
        else:
            return NotImplemented
```

Try it:

```python
>>> a = MutInt(3)
>>> b = MutInt(3)
>>> a == b
True
>>> c = MutInt(4)
>>> a < c
True
>>> a <= c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<=' not supported between instances of 'MutInt' and 'MutInt'
>>>
```

The reason the `<=` operator is failing is that no `__le__()` method was provided. You could code it separately, but an easier way to get it is to use the `@total_ordering` decorator:

```python
from functools import total_ordering

@total_ordering
class MutInt:
    __slots__ = ['value']

    def __init__(self, value):
        self.value = value

    ...
    def __eq__(self, other):
        if isinstance(other, MutInt):
            return self.value == other.value
        elif isinstance(other, int):
            return self.value == other
        else:
            return NotImplemented

    def __lt__(self, other):
        if isinstance(other, MutInt):
            return self.value < other.value
        elif isinstance(other, int):
            return self.value < other
        else:
            return NotImplemented
```

`@total_ordering` fills in the missing comparison methods for you as long as you minimally provide an equality operator and one of the other relations.

(e) Conversions

Your new primitive type is almost complete. You might want to give it the ability to work with some common conversions. For example:

```
>>> a = MutInt(3)
>>> int(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string, a bytes-like object or a
number, not 'MutInt'
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float() argument must be a string, a bytes-like object or a
number, not 'MutInt'
>>>
```

You can give your class an `__int__()` and `__float__()` method to fix this:

```
from functools import total_ordering

@total_ordering
class MutInt:
    __slots__ = ['value']

    def __init__(self, value):
        self.value = value
    ...

    def __int__(self):
        return self.value

    def __float__(self):
        return float(self.value)
```

Now, you can properly convert:

```
>>> a = MutInt(3)
>>> int(a)
3
>>> float(a)
3.0
>>>
```

As a general rule, Python never automatically converts data though. Thus, even though you gave the class an `__int__()` method, `MutInt` is still not going to work in all situations when an integer might be expected. For example, indexing:

```
>>> names = ['Dave', 'Guido', 'Paula', 'Thomas', 'Lewis']
>>> a = MutInt(1)
>>> names[a]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not MutInt
>>>
```

This can be fixed by giving `MutInt` an `__index__()` method that produces an integer. Modify the class like this:

```
from functools import total_ordering

@total_ordering
class MutInt:
    __slots__ = ['value']

    def __init__(self, value):
        self.value = value

    ...

    def __int__(self):
        return self.value

    __index__ = __int__     # Make indexing work
```

```
```

Discussion

Making a new primitive datatype is actually one of the most complicated programming tasks in Python. There are a lot of edge cases and low-level issues to worry about--especially with regard to how your type interacts with other Python types. Probably the key thing to keep in mind is that you can customize almost every aspect of how an object interacts with the rest of Python if you know the underlying protocols. If you're going to do this, it's advisable to look at the existing code for something similar to what you're trying to make.

[Solution | Index | Exercise 2.3 | Exercise 2.5]
