Special Methods in Python

The built-in mathematical operators can be extended in much the same way as `repr`; there are special method names corresponding to Python operators for arithmetic, logical, and sequence operations.

To make our code more legible, we would perhaps like to use the `+` and `*` operators directly when adding and multiplying complex numbers. Adding the following methods to both of our complex number classes will enable these operators to be used, as well as the `add` and `mul` functions in the `operator` module:

```
>>> ComplexRI.__add__ = lambda self, other:
add_complex(self, other)
>>> ComplexMA.__add__ = lambda self, other:
add_complex(self, other)
>>> ComplexRI.__mul__ = lambda self, other:
mul_complex(self, other)
>>> ComplexMA.__mul__ = lambda self, other:
mul_complex(self, other)
```

Now, we can use infix notation with our user-defined classes.

```
>>> ComplexRI(1, 2) + ComplexMA(2, 0)
ComplexRI(3.0, 2.0)
>>> ComplexRI(0, 1) * ComplexRI(0, 1)
ComplexMA(1.0, 3.141592653589793)
```

True and false values. We saw previously that numbers in Python have a truth value; more specifically, 0 is a false value and all other numbers are true values. In fact, all objects in Python have a truth value. By default, objects are considered to be true, but the special `__bool__` method can be used to override this behavior. If an object defines the `__bool__` method, then Python calls that method to determine its truth value.

As an example, suppose we want the complex number `0 + 0 * i` to be false. We can define the `__bool__` method for both our complex number implementations.

```
>>> ComplexRI.__bool__ = lambda self: self.real != 0 or
self.imag != 0
>>> ComplexMA.__bool__ = lambda self: self.magnitude != 0
```

We can call the `bool` constructor to see the truth value of an object, and we can use any object in a boolean context.

```
>>> bool(ComplexRI(1, 2))
True
>>> bool(ComplexRI(0, 0))
False
>>> if not ComplexMA(0, 1):
        print("complex number is true")
complex number is true
```

Sequence length. We have seen that we can call the `len` function to determine the length of a sequence.

```
>>> len('Go Bears!')
9
```

The `len` function invokes the `__len__` method of its argument to determine its length. All built-in sequence types implement this method.

```
>>> 'Go Bears!'.__len__()
9
```

Python uses a sequence's length to determine its truth value, if it does not provide a `__bool__` method. Empty sequences are false, while non-empty sequences are true.

```
>>> bool('')
False
>>> bool([])
False
>>> bool('Go Bears!')
True
```

Callable objects. In Python, functions are first-class objects, so they can be passed around as data and have attributes like any other object. Python also allows us to define

objects that can be "called" like functions by including a `__call__` method. With this method, we can define a class that behaves like a higher-order function.

As an example, consider the following higher-order function, which returns a function that adds a constant value to its argument.

```
>>> def make_adder(n):
        def adder(k):
            return n + k
        return adder
>>> add_three = make_adder(3)
>>> add_three(4)
7
```

We can create an `Adder` class that defines a `__call__` method to provide the same functionality.

```
>>> class Adder(object):
        def __init__(self, n):
            self.n = n
        def __call__(self, k):
            return self.n + k
>>> add_three_obj = Adder(3)
>>> add_three_obj(4)
```

Here, the `Adder` class behaves like the `make_adder` higher-order function, and the `add_three_obj` object behaves like the`add_three` function. We have further blurred the line between data and functions.

Though callable objects are less efficient than higher-order functions, they allow us to take full advantage of the object system. For example, we can use inheritance to create a family of callable objects with shared functionality.

Further reading. Special methods generalize the built-in operators so that they can be used with user-defined objects. In order to provide this generality, Python follows specific protocols to apply each operator. For example, to evaluate expressions that contain the `+` operator, Python checks for special methods on both the left and right operands of the expression. First, Python checks for an`__add__` method on the value of

the left operand, then checks for an `__radd__` method on the value of the right operand. If either is found, that method is invoked with the value of the other operand as its argument.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#special-methods