Operator Overloading: What is It?

by Kirby Urner
4D Solutions

for

Thunderbird Early College Charter
(TECC)

March 10, 2008

Early in our grade school training, we're exposed to these symbols for arithmetical operations, later called operators, but at first just called symbols or signs. For example, + is the “addition symbol” and – is the “subtraction symbol”.

As we walk through our early training, new symbols get introduced, or we get shifts in usage. For example, early elementary students see this division symbol a lot: ÷. However, in later years, a line between numerator and denominator, at a slant, or horizontal, starts taking over. Also, the little negative sign in the upper left, denoting a “negative number” tends to be replaced by a normal-sized negative sign, as in -3.

With the advent of computers, a lot of this cruft of the ages was done away with, plus given the multiplication symbol (x) looked too much like the letter x, the decision was made to use the asterisk (*) instead. This innovation permeated a large number of computer languages.

However, with the advent of *Mathematica* and other such computer algebra systems (CAS), there's been a lot of attention on getting the old typography back, including all the Greek letters, other fancy symbols. These luxuries weren't affordable in the early days of FORTRAN and COBOL.

But now, in the early 21st century, with Unicode replacing ASCII, we have all kinds of new glyphs available, including Elvyn and Klingon, should we wish to include them.

Returning to the topic at hand, operator overloading, there came a time in the evolution of computer languages, that language designers started letting programmers determine the meaning of +, *, / and – (among other symbols).

In other words, instead of just giving programmers a hard-wired fixed meaning for +, in association with a short list of types (integers, floating point, strings), the language would let programmers give + any meaning they liked, though one might hope within reason.

Lets take an example. In FORTRAN, 3 + 3 has a clear meaning to the compiler, as does x + y, if x and y are both of the same addable type. 2.0 + 5.0 is another good example. And here's a new twist: in many languages, character strings may be added like this: 'United St' + 'ates', giving 'United States'.

So those are a few meanings a FORTRAN-like language supports “out of the box” for the addition symbol. In Python, however, C++, or C#,

you can define a new type, say a Vector type, then write code for the __add__ method. Any method named __add__ (note the peculiar under bars), is going to control the behavior of +, when used with an object of this Vector type. Other objects will have a different meaning for __add__, like you can add Polynomials.

In sum, when a language offers operator overloading, you've been given the power to write instructions for some of the basic symbols. Instead of taking all their meanings for granted, you *define* their meanings, in the process of defining various *types*.

The sensitivity to different "types" is typical among programmers, but also among mathematicians, who organize their "types" into "sets". Some of the important sets they deal with are N, the natural or counting numbers, with or without zero (some texts use W for wholes, N + {0}), Z the integers, Q the rationals, R the reals, and C the complex numbers. Conveniently, each successive set contains all sets before it, like Russian dolls.

But then we have these other types, like Vectors, Quaternions, Matrices... the literature is thick with exotic creatures, their many behaviors. And because we have operator overloading, if Vectors permit being added (they do), or Matrices allow being multiplied (they do), then by defining behavior for __add__ and __mul__, we'll be able to express these capabilities very simply, using + and * respectively.

```
>>> v1 = Vector(( 0, 0, 4))
>>> v2 = Vector((-1,1,-5))
>>> v1 + v2
Vector (-1, 1, -1)
>>> v1 — v2
Vector (1, -1, 9)
>>> 2 * v1
Vector (0, 0, 8)
```

We like our TECC students to have mastery over fractions, or more specifically rational numbers in the set Q, in terms of their arithmetic operations. A traditional USA math curriculum got students adding, multiplying, subtracting and yes, dividing fractions. Our curriculum is no different in that respect.

Our Pythonic approach is to have each student gradually build up a class Q (or Rat or whatever), consisting of definitions for __add__, __mul__, __div__, __sub__ and __pow__, a few others. In studying Q, students need to go through the same mental gyrations, regarding "lowest terms," "simplification," "numerator," "denominator" and all the rest of it. We're not short-circuiting any of that wiring,

merely re-expressing it in a new, machine executable math notation (e.g. Python), one that accommodates both operator overloading, *and* a way of defining new kinds of “math objects” (beyond those built in).

My belief is that before long, having defined __add__ a few times, for different types of math object, the average student will start to feel comfortable with the idea of “family resemblance” i.e. there are commonalities between our uses of + across many sets, same with *.

I would call this an early receptivity to abstract algebra. And because working in Python over the years has heightened it, we'll be able to cap the high school experience with more college level abstract algebra, group theory, topics in number theory. This will be especially relevant to those bored with calculus, or certain they won't be needing it so much, which is true of many technical as well as non-technical careers.