A Proposal to Pioneer a Role for Python Within an Alternative K-12 Mathematics Track

by
Kirby Urner
June, 2005

Oregon Curriculum Network
Portland, Oregon, USA

A Proposal to Europython 2005

A Proposal to Pioneer a Role for Python Within an Alternative K-12 Mathematics Track

By Kirby Urner
June 3, 2005
4D Solutions
Portland, Oregon, USA

Nomenclature

Given this paper was designed for an international readership, I should not assume a universal namespace when it comes to education matters. For example, when I transferred to the British system for a year in the mid-1960s, I became aware that what I called "third grade" my counterparts in Britain called "first form."

In the USA, the namespace features twelve grades pre-college, plus kindergarten and possibly pre-school. College is typically a four year experience, leading to a Bachelor of the Arts or Sciences degree. Further degrees require additional years of study.

These pre-college years are generally referred to as K-12 (kindergarten through twelfth grade). Or one might say K-16 to mean all formal schooling through the end of college. The term *high school* (or *secondary school*) embraces grades 9-12.

I will use the above terminology on the understanding that rough translations to other schooling systems, including home schooling, will be required. Given we all grow older according to various biological clocks, at least the aging variable may be taken as global, even while the steps to higher learning are differently defined and described.

CP4E

Since the early days of Guido's DARPA-funded "computer programming for everybody" proposal (CP4E), elements within the Python community have harbored the ambition to see their language integrated into various curricula, with a special focus on those who are choosing other walks of life besides computer science (CS). The CP4E strategy is to reach beyond the walls of CS to embrace, well, everybody with a desire to learn.[1]

This strategy is realistic even within the standard college computer science curriculum, as the introductory courses tend to be designed to accommodate non-CS majors, as well as those expecting to continue in CS. In other words, students in neighboring departments such as physics or biology, may fulfill various requirements, or simply amplify their skill sets in relevant ways, by diving into a CS sequence. Teachers are finding that if such students first encounter Python, versus say C++ or even Java, their satisfaction with the course material tends to increase, given Python's relative ease of use.

A parallel strategy is to reach into the pre-college years and find ways to leverage existing and/or traditional content in ways that foster a more mature approach to computer technology. At least in the USA, pre-college mathematics text books and teacher groups such as the NCTM give lip service to "technology in the classroom" but in practice this mostly translates into calculator, not computer use. The current status quo, as of 2005, is that programming of any kind is usually not discussed in math class. On the other hand, elective computer science and programming courses are beginning to appear at the high school level, and mathematics *is* discussed in them. In sum, pre-college CS is where math and programming currently come together.

My interest is in strengthening the pre-college CS curriculum to where it becomes less an exotic elective, and more a viable alternative to the standard mathematics sequence. As a student, if you find yourself attracted to programming, and to more computer and technology use in general, then you may seek to escape the non-technologically sophisticated mathematics curriculum of the past, and learn what you need in a new way.[2] Python may well be a part of this experience, and the mathematics content in this new track will be strong enough to bring you up to a college level of preparation, independently of the traditional math curriculum.

OO and Mathematics

The task of integrating mathematics in a contemporary computer language such as Python brings up the question of how to fit object oriented thinking into an integrating framework. We don't want our discussion of classes and objects to come across as some elaborate digression that has no relevance to mathematics as traditionally taught, since the whole point is to achieve a better synthesis between pre-college math and programming.

I believe the solution is to recast mathematics as an extensible type system, with OO languages providing concrete examples of what this means in practice.[3] Once you have some primitive types available, such as integers, floats and strings, your job as a programmer and mathematician is to develop new types by means of composition and inheritance. A type defines both properties (or state) and behavior i.e. nouns and verbs. An object of a particular type generally has its individual values, such as numerator and denominator in the case of a rational number, plus all the method-defined behaviors associated with this type, such as an ability to add or multiply with other objects of the same (or perhaps a different) type.

In other words, bringing traditional mathematics into an OO context involves making "dot notation" a more ordinary and intrinsic aspect of math notation generally. Students should develop an appreciation for how `object.method(arguments)` and `object.property` expressions give form to standard algorithms. Additionally, they need to understand constructors (e.g. Python's `__init__`) whereby new objects come into existence. The difference between a class definition, and objects as specific *instances* of a class, will be used to discuss and manipulate such mathematical entities as rational numbers, polynomials, vectors, matrices, and polyhedra.

Data Structures

Typically, the one data structure that has received the most attention, especially since the so-called “new math” (SMSG) in the 1960s, is the set.[4] Students learn that a set consists of unique elements, with membership determined either by roster (a simple listing) or by rule (criteria). Sets may contain finite or infinite members, the latter tending to be rule-defined (e.g. all even numbers). Once the set concept is established, operations such as union and intersection are introduced. In the USA, the “new math” started introducing these concepts as early as 2nd grade.

There’s a lot of history behind this push to introduce sets early in the curriculum -- and behind its fading momentum. Partly the idea was to get students used to new symbolism. The symbols for “is an element of” (membership), “is a subset of,” union and intersection, were becoming more common in college level reading, and the idea was to get students fluent in higher math more quickly.

In Python, we have a set type (a built-in as of Python 2.4), but it comes across as a specialized kind of dictionary, one with keys but no values. The unordered aspect of dictionaries is thereby preserved (no integer-indexing of the membership, unless you’ve used integers as keys – but even then, no slices), plus the uniqueness of dictionary keys is consistent with the set concept. However, the set is but one example of a collection type. Lists tend to be the more common workhorse of the Python language, thanks especially to list comprehension syntax.

I would propose a parallel shift in emphasis in K-12 mathematics, at least along this alternative CS-informed track, where programming in Python will be a popular option: keep the material on sets, including the specialized notation, but integrate it into a larger discussion of collection types. Use dot-notation to provide a more universal way to express static and dynamic aspects of collections (sets included), plus use this opportunity to introduce integer-based indexing. Mathematics is replete with indexing, including in two-dimensional tables such as matrices. N-dimensional structures follow by extension (a quick foray into some array-based language, such as J, might make sense at this point).[5]

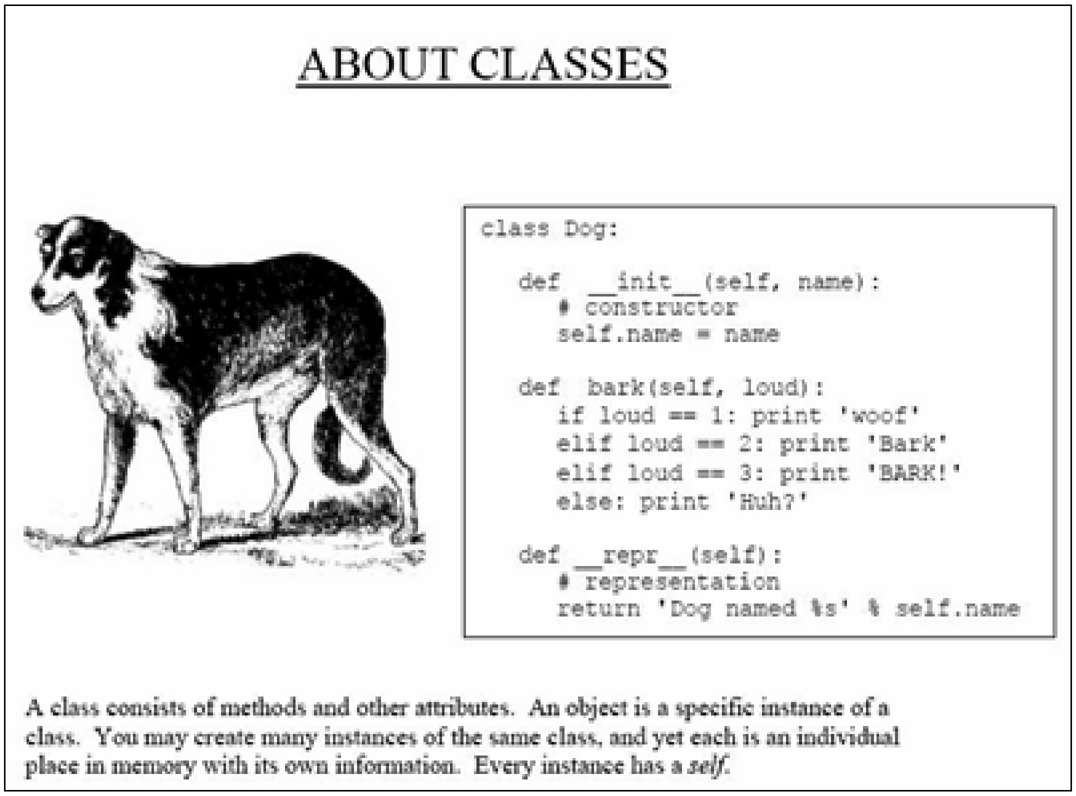
The rule-defined infinite set becomes a useful jumping-off point for getting into Python generators, i.e. some rule keeps giving a next member of the set, indefinitely and *ad infinitum*. It’s not that the members “already exist” in computer memory, but that their successive generation has been algorithmically defined. This shift in our approach to infinite sets has philosophical implications which I consider both demystifying and healthy.

Sequences

The common wisdom in CS pedagogy is that the class/object discussion should come sometime after practice has been gained with the easier concepts of assignment to variables (name binding), data types, and simple procedural programming, which includes the concepts of scope, conditional execution, and looping control structures (with recursion considered an advanced way of looping, unless we're doing Scheme, in which case recursive looping is the norm).

I agree with this common wisdom, but would say that in the case of Python, we're able to get into the class/object discussion earlier, rather than later. This is in part because the primitive types have introspective abilities, such as uniform responsiveness to the dir command, as in dir(1). The fact that 1 .__add__(1) returns 2 is something to mention early, as a first step towards appreciation for operator overloading.[6]

I would say dot-notation, and the concept of objects as containers for state and behaviors, deriving from shared class definitions, is accessible well before it becomes necessary to write one's own types into existence. Whereas procedural programming is a prerequisite for writing methods internal to class definitions, dot-notation is a prerequisite to even using the data structures (e.g. set, list, dictionary, tuple, stack, queue, tree, network). And dot-notation makes the most sense when you unpack it in terms of a hierarchy of classes. Fortunately, thinking in terms of class hierarchies is nothing new. Aristotle did it. We're talking taxonomies, already deeply engrained in childhood experience, through visits to the zoo, and obvious commonalities among mammals versus insects or plants.



ABOUT CLASSES

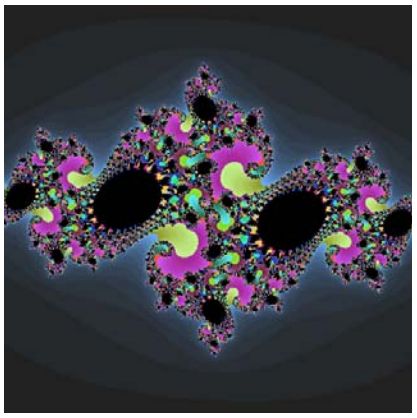
```
class Dog:

   def  __init__(self, name):
      # constructor
      self.name = name

   def  bark(self, loud):
      if loud == 1: print 'woof'
      elif loud == 2: print 'Bark'
      elif loud == 3: print 'BARK!'
      else: print 'Huh?'

   def __repr__(self):
      # representation
      return 'Dog named %s' % self.name
```

A class consists of methods and other attributes. An object is a specific instance of a class. You may create many instances of the same class, and yet each is an individual place in memory with its own information. Every instance has a *self*.

Excerpt from a hand-out, class for early teen home scholars,
Free Geek, Portland, Oregon

In the mode of procedural programming, or defining functions in Python, I recommend following the logic in *The Book of Numbers* by Conway and Guy, and/or in Sloane's *On-Line Encyclopedia of Integer Sequences*: use geometric packings, gnomons, sharply angular shapes, to reinforce algebraic and visual cortex connections.[7] For example, start with the square and triangular numbers. Move to Gauss and the formula for the sums of N consecutive integers, starting with 1 (= triangular numbers). Talk about tetrahedral numbers as a growing stack of ever larger triangular numbers. Program these sequences in Python, including as generators.[8]

Once we have some Python-generated sequences handy, a curriculum might turn to their broad characteristics: sequences may be convergent/divergent, oscillatory, chaotic. Dwelling on these differences sets the stage for later forays into calculus, trigonometry, fractal geometry, and dynamical systems theory.



A Julia Set, generated using Python's complex numbers type, and PIL

The Big Picture

My overall expectation is that many of the trends I'm discussing here are, in broad outline, very likely to pan out. However, as individuals with architectural and aesthetic sensibilities, we're inclined to add details and specifics, which is where my particular castings come in. I don't set education policy for the USA, obviously. I follow in a tradition pioneered by many earlier writers: I sketch possibilities and encourage discussion.

In the realm of megatrends, I see a general rise in digital versus analog technologies, the replacement of the latter by the former. Within this, we see a resurgence of interest in discrete mathematics, because at bottom computer memory is finite and binary, yet we still need to represent the smoothness we find in nature. The various file format and encoding strategies, used to preserve information (e.g. JPEG and PNG formats) use digital or on-off logic, but in terms of gradation, they have to remain attuned to subtle (fuzzy, analog) differences, otherwise we'll write them off as too unsophisticated.

As a consequence of this megatrend, I think we're seeing where colleges will open doors to discrete mathematicians, so if that's what you studied in high school, we have good career options for you. In recent generations, this hasn't been so true. Television, radio and electromagnetism are based in a continuum calculus, an analog conception. A physics major needed lots of calculus – and still does. But now we have all these doors open in CS and IT which, although wholly dependent on electromagnetism, use a more Boolean approach. The foundation is logic gates: AND OR XOR NOR and so on.

Given industry is opening doors to discrete mathematicians in large numbers, CS has a strong hand in terms of curriculum, even if not in terms of absolute numbers (there will always be more English majors). CS is in great demand within the biology department, as a sponsor of bioinformatics, a discipline requiring new algorithms as well as new implementations of pre-computer techniques. Biology is where the action is these days, even more than in physics.[9] In sum, the ascendancy of biology in part explains the new appeal of discrete mathematics. Putting it this way sounds paradoxical, as we tend to regard our biological selves as analog (but this too may be changing).

Consequent to all of the above, I believe Python is well positioned to bolster discrete mathematics with concrete command line experience, even pre-college, in K-12.

Operations Modulo N

Whereas the "new math" sucked in a lot of elementary set theory and Boolean algebra (in the form of truth tables), it didn't do much with number or group theory. And yet these latter contain many easy and accessible parts, group theory especially, which is a logic of permutations, which we encounter routinely in K-12 in the form of probability studies. So I envision that if CP4E takes off, we'll be incorporating more of both. I would commend a conceptual appreciation of RSA, the encryption algorithm, as a worthy goal for high school students. As evidence that this is doable, I offer the autobiographical *In Code: A Mathematical Journey* (ISBN 1-56512-377-8) by teen author Sarah Flannery, about her forays into cryptography. The book was a bestseller.

In order to build up number and group theory in the early grades, I would use a Python class with operator overloading, designed to imitate integers (which doesn't mean I'd use inheritance), but modulo some class variable. For example, 5 + 5 modulo 7 == 3, because we only consider remainders once all the 7s have been subtracted away. This kind of arithmetic is essential to these branches of mathematics (group and number theory), but doesn't get much focus in K-12 – just some stuff about "clock arithmetic" (time-keeping systems tend to be modulo various cosmic outer or atomic inner frequencies).

With operator overloading, it's easy to write code to generate a table of all members of a set multiplied by all other members, in a finite case. Integers modulo N provide such finite cases. If the members consist of modulus totatives, then group properties are displayed. If the modulus is prime, then all positives less than it are its totatives.[10] These generalizations connect with theorems from Euler and Fermat. RSA makes use of these theorems.

Complex Numbers

Given Python's excellent complex number support, which might be supplemented with complex number Decimals, we should count on doing complex number arithmetic with Python in K-12. Given the Argand Diagram, complex numbers have many associations with vector arithmetic. Complex number multiplication (closed within the complex numbers field) begets rotation in a plane. Rotation in a sphere is accomplished using another set closed (but not commutative) under multiplication: that of the quaternions.

Whether Hamilton's quaternions should be introduced in K-12 I don't know. I've done some experimental curriculum writing in this domain.[11] The question is: where are we going with Clifford Algebra, and should we be making quaternions more a part of linear algebra as a way of making Clifford Algebra the target? I don't claim to have the answer to this question, but suspect the answer might be "yes."

The Calculus

Regarding the calculus. I think Python is well-suited to introducing a concept of the derivative because its functions are top-level and may be passed as arguments to other functions. The derivative is very amenable to a functional programming approach. Define D(f) such that it returns a new function, f '. In Python, you may do this with a predefined small h:

```
def d(f, h = 1e-8):
    def deriv(x):
         return (f(x+h) – f(x))/h
    return deriv
```

The above may be improved upon, but here follows the traditional text book formulation, and is elegantly correct in defining functions as the inputs and outputs, not specific values at any point. Of course given finite h, the above is not a true derivative as calculus conceives of it. In those terms, this is a discrete math approximation of some deeper analog reality.

When I talk about a 2nd math track in K-12, I'm speaking of the discrete math track. However, I haven't been talking about this 2nd track replacing the first, the more traditional pre-calculus and calculus track. My expectation is the two will co-exist, and students will have the option of going with either or both. The fact that doors are opening in new directions doesn't mean they're closing in others, only that the flow is becoming more equal. It's now more possible to enjoy a technical career, without having a lot of calculus background. You spent your time in discrete topics instead, and that choice is now increasingly respected.

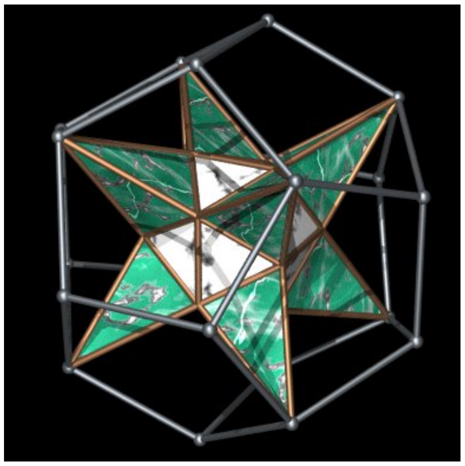
Computer Graphics

The conventional wisdom around computer graphics has been to follow Logo's model, and exploit the powers of a graphical cursor, with optional pen colors, over a canvas. An (x,y) grid is supplied, along with commands for angular rotation and translation (forward and back). This regime easily translates to a robotic context, where the graphical cursor becomes the robot.

I have no problem with the conventional wisdom or Lego/Logo robotics. However, I think the rapid evolution of ray tracing, and the accessibility of free power tools in this domain, suggests a more austere approach could have just as visually stimulating results. Instead of flying a turtle, which in the 3D case has tended to invoke such concepts as roll, pitch and yaw (the turtle as airplane metaphor), we simply "connect the stars" in various polyvertexial constellations, in either closed polyhedral, or open molecular networks.[12]

Most of my experiments in this realm have involved POV-Ray, with some forays into VRML and other applications, including some in the still-obscure elastic interval geometry genre. These applications take care of perspective, so the computations on the Python side use straight K-12 text book algorithms, no CS-specific "homogenous coordinates" and such. It's the straightforwardness of the APIs, adapted and made more uniform through Python, that make this project doable, and the results may have high production values, given all the textures and sky mappings a ray-tracer may supply.

This Python + ray-tracer approach dovetails nicely with the earlier thread about sequences, since now the polyhedral numbers, sequences of growing agglomerations (such as the tetrahedral or cuboctahedral), may be rendered graphically. The pictures and models encountered up to this point now turn out to have been generated using Python in many cases. This provides students with some satisfying closure, as their own graphical results (including animations) start to look more and more like those in the teaching materials.



A stellated icosahedron
generated using Python + POV-Ray

Geometry

In this realm of computer graphics, we must discuss topics in geometry. I favor starting with the spatial shapes, such as in Montessori, with emphasis on familiar solids or blocks (cylinder, cone, wedge, brick, arch, ball), as well as on the Greek pantheon: the Platonic and Archimedean polyhedra, with related stock.

In this regard, I'm influenced by the New England Transcendentalist R. Buckminster Fuller, pioneer of geodesic architectures, as he integrated polyhedral and lattice-based imagery in a single synergetic construct he called the *isotropic vector matrix*, and which turns out to match the CCP and FCC (already well-understood lattices).[13] This tetrahedron / octahedron truss-work has tetrahedra and octahedra in relative abundance of 2:1, but in terms of volume, the octahedron outweighs the tetrahedron 4:1. Every vertex is identically at the center of a cuboctahedron of twelve surrounding neighboring vertices. The voronoi cell around each vertex is the rhombic dodecahedron, of relative volume 6, given the tetra:octa ratio of 1:4. The aforementioned cuboctahedra have a volume of 20.[14]

My expectation is that Fuller's innovations will continue to make inroads, as a result of work that only partially overlaps any Python-based curriculum reform initiatives. These two intellectual currents may be different aspects of the digital versus analog re-evaluation. However that may be, I see my role as one of connecting the dots for the purpose of discussion, in some cases helping to catalyze positive developments that might have happened anyway, but more slowly.

Conclusions

The gist of my position is that K-12 is changing shape, in response to changes in industry. The emergence of a strong IT sector, evident in places like Singapore, is coincident with a rise in the biological sciences, and an accompanying hunger for computing power. Discrete mathematics, at the basis of digital computing, is therefore enjoying something of a Renaissance, to the point where it's now possible to envision a 2nd math track emerging pre-college, one which emphasizes more CS and discrete math topics, such as we find in the group and number theory behind such algorithms as RSA.

Python is well-positioned to facilitate and smooth these changes, where earlier systems languages have proved ill at ease. With dynamic typing, we don't lose the sense of a class hierarchy, and therefore preserve the OO context of dot-notation. However, we do so without requiring the strict compile-time auditing associated with static typing. In removing this overhead, without sacrificing the elegance of the OO model, Python stands out. This makes it a good precursor to static typing (the concepts make sense in retrospect), while allowing us to transmit the core of OO to non-CS majors, who may not stick around for the systems language courses to follow.

The tools which Python brings to bear, as an OO language in particular, will allow us to better unify some important concepts in the traditional curriculum. We have ways of building sets into a larger presentation about data structures, and "math objects" as type extensions (by composition, by inheritance) within an always-changing hierarchy. Math is a living tree, not a dead one.

I don't expect any one uniform curriculum to evolve from these discussions and proposals. Rather, we will continue to experiment with a wide variety of pedagogical initiatives, learning from one another's failures and successes. Mistakes remain an important source of feedback, as we refine and define through a process of trial and error. In particular, I don't expect that my particular blend of Fuller-influenced geometry, ray tracing, and hypertoons with VPython, will be an overnight success.[15] Instead, I will continue fine tuning and collaborating with my peers, many of whom will share my admiration for Python, as a tool for "programming to learn," not just for "learning to program."

NOTES

[1] Guido van Rossum, *Computer Programming for Everybody (Revised Proposal)*, CNRI Proposal #90120-1a, July 1999. http://www.python.org/doc/essays/cp4e.html

[2] "Because students tend to make choices about their educational pathways while they are still in high school, I would suggest that what happens in high schools is probably more relevant than the curriculum choices colleges are making." Chris Stephenson, Executive Director of the Computer Science Teachers Association (USA), discussion board 05-24-05 14:38, "Re: The Empty Pipeline in Computer Science," *The Chronicle of Higher Education* http://chronicle.com/forums/colloquy/read.php?f=1&i=5109&t=5092

[3] The convergence of OO, dot-notation, and conventional math notations, is a long-running thread in my writings. For example, see: *Trends in Early Mathematics Learning: Looking Beyond Y2K* at my Oregon Curriculum Network web site: http://www.4dsolutions.net/ocn/trends2000.html

[4] SMSG = School Mathematics Study Group, headed by Edward Begle, funded by the USA's National Science Foundation (NSF) in response to Sputnik. See Ralph Raimi's *Whatever Happened to the New Math* for some more background: http://www.math.rochester.edu/people/faculty/rarm/smsg.html

[5] For my essays on J, with links to Jsoftware, see the J section of my CP4E page: http://www.4dsolutions.net/ocn/cp4e.html (also where I link to many of my Python-based essays). Kenneth Iverson himself gave me some assistance on *Jiving in J*.

[6] Note the space between 1 and .__add__ -- otherwise the parser will confuse the dot with a decimal point.

[7] For example, the number of balls in successive shells of a growing cuboctahedron (= CCP) is given here: http://www.research.att.com/projects/OEIS?Anum=A005901 (1, 12, 42, 92, 162…)

[8] I go into some detail on these topics, minus any use of generators, in *Getting Serious about Seri es*, the first section of a 4-part essay on Python and mathematics: http://www.4dsolutions.net/ocn/numeracy0.html

[9] *Beyond the Gene* (blog entry by K. Urner), http://worldgame.blogspot.com/2005/01/beyond-gene.html

---

[10] http://www.4dsolutions.net/ocn/flash/group.html (a Flash animation about group theory, with background information for teachers, including about totatives and totients – K. Urner, Oregon Curriculum Network)

[11] See my *Getting Inventive with Vectors* for some more about quaternions in Python: http://www.4dsolutions.net/ocn/numeracy1.html

[12] My use of *polyhedral* vs. *molecular* to characterize open and closed networks is intended to be suggestive, more than technically precise; some molecules are polyhedral after all, such as buckminsterfullerene.

[13] http://www.grunch.net/synergetics/octet.html (octet truss = IVM = fcc = ccp) – note link to studies by Alexander Graham Bell.

[14] See my essay on volumes in synergetics: http://www.grunch.net/synergetics/volumes.html

[15] For more on hypertoons, see: http://wiki.python.org/moin/HyperToons (also, watch for an article about them in an upcoming issue of *PyZine*).