CONCRETE ABSTRACT ALGEBRA IN PYTHON

John Kerl
University of Arizona
Department of Mathematics
Software Interest Group
April 5, 2006

Overview

- History
- Python basics
- Concrete groups
- Algorithms for abstract groups

History

SACK: simple algebra calculator.

Previous version was written in C during a senior-level undergraduate course in abstract algebra. Naive algorithms; short feature list.

Imitates Unix `bc` program: If the computer can multiply 10981765243 and 76452183109, then it ought to be able to compose the permutations (10 9 8)(1 7)(6 5 2)(4 3) and (7 6)(4 5)(2 1 8 3)(10 9).

Joubert: "To teach is to learn twice." I wrote SACK from scratch and learned many things in the process, which I would not had I stuck solely with programs such as GAP.

Python . . .

. . . is an **object-oriented** (illustrated by example in this paper) **scripting** (not compiled as a separate step, though Perl and Python do just-in-time compilation) **language**.

We have many mathematical software tools: Mathematica, Maple, MATLAB, etc. Python on the other hand is a **general-purpose language**. However, it is intutive and math-friendly (e.g. lists and complex floating-point numbers).

Sometimes there is no pre-existing tool or library for a particular problem and we need to program. Python is good as a first programming language.

More information

These slides are a condensation of:

`http://math.arizona.edu/~kerl/doc/kerl-pyaa.pdf`

For a thorough introduction, see Lutz & Ascher's *Learning Python* (O'Reilly 2004), or `www.python.org`.

Basics

Python may be run either **interactively** (command prompt) or **scripted** (put commands in one or more files).

```
kerl@gila:pyaa% python
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> 1+2
3
```

Script example: have a file `factorial.py`.

```
def fact(n): # Here is how you write a comment in Python.
    if (n < 0):
        return 0
    elif (n <= 1):
        return 1
    else:
        return n * fact(n-1)
```

Then:

```
kerl@gila:pyaa% python -i factorial.py
>>> fact(10)
3628800
>>>
```

Main points about python

- Variables don't need to be declared before use.
- **Control structures**, e.g. if-statements, while-loops, for-loops, etc are indicated purely by **indentation**, rather than by curly braces or end-statements as in most other languages.
- Due largely to these two facts, Python reads like **pseudocode**: Python programs often do just what they look like they do, with little syntactical overhead.

There are four language features which make Python ideal for abstract algebra: **lists**, **operator overloading**, **run-time binding**, and **modules**.

Lists

Python has a flexible **list** type with the following features:

- **Indexed**, e.g. they can be treated as arrays (note: zero-based).
- **Nested**, e.g. they can be used for matrices, or higher-dimensional arrays. Useful for representing cosets, direct products, etc.
- **Heterogeneous**, e.g. the first element of a list can be a number, the second a string, the third a sublist, etc. Ee.g. matrices: each list element is an array, all of the same length; list of conjugacy classes: not all the same length.

Zero-based indexing and bounds checking:

```
>>> mylist=[1,2,3]
>>> mylist[0]
1
>>> mylist[1]
2
>>> mylist[2]
3
>>> mylist[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

List heterogeneity:

```
>>> mylist=["hello", 3, [2,3,4]]
>>>
>>> mylist[0]
'hello'
>>> mylist[1]
3
>>> mylist[2]
[2, 3, 4]
```

Iteration:

Often we want to loop over the elements of a list:

```
>>> mylist=[0,2,4,6]
>>> for element in mylist:
...     print element

0
2
4
6

>>> for i in range(0, 4):
...     print i**2,
...
0 1 4 9
```

Operator overloading

Example: modular addition (representation of cyclic group on n elements.)

We want to be able to use this in the following way:

```
a = modadd_t(5, 11)
b = modadd_t(8, 11)
c = a * b
print c
```

Implementation:

```
class modadd_t:
    def __init__(self, residue, modulus):
        self.residue = residue % modulus
        self.modulus = modulus
    # Use "*" for addition.  Seems weird, but groups
    # are abstracted multiplicatively in SACK.
    def __mul__(a,b):
        if (a.modulus != b.modulus):
            print "Mixed moduli %d, %d" % (a, b)
            sys.exit(1)
        c = modadd_t(a.residue + b.residue, a.modulus)
        return c
    ...
```

Run-time binding

Above we saw a `modadd_t` data type. Suppose there are others, e.g. `dih_t` for elements of the dihedral group $\mathcal{D}_n$, etc.

```
a = modadd_t(2, 5);       d=modadd_t(3, 5)
b = dih_t(2, 0, 8);       e=dih_t(3, 1, 8)
c = pmt_t([0 3 1 2], 4); f=pm_t([3 2 0 1], 4)
X = [a,b,c];              Y = [d,e,f]
Z = [X[0]*Y[0], X[1]*Y[1], X[2]*Y[2]]
```

More generally, we can make functions which operate on lists of objects, and the functions we write don't need to know anything ahead of time about the data types of those objects. As long as the objects can be multiplied using the * operator, our code will work just fine. This is the essence of abstraction.

Modules

See the paper for info; not presented in the slides.

Concrete groups

We already saw `modadd_t`. Here we'll look at `dih_t`.

Dihedral groups (symmetries of plane n-gons) could have been implemented as subgroups of $\mathcal{S}_n$. In fact, *all* these small finite groups could be done this way. The point is that by implementing specific data types, we obtain more user-friendly input-output representations.

There are two levels of software needed for each group: (1) routines to deal with individual group elements, and (2) routines to construct a list of all the elements of a group.

Dihedral groups

The dihedral group $\mathcal{D}_n$ is here taken to be the symmetry group on a plane n-gon. It has order $2n$ and is given by the following presentation:

$$\mathcal{D}_n = \langle \rho, \phi \mid \rho^n = \phi^2 = 1, \phi\rho = \rho^{n-1}\phi \rangle.$$

From the presentation, the element ρ (*rho* for *rotate*) has order n; the element ϕ (*phi* for *flip*) has order 2. Furthermore, repeated use of the final relation enables any element of $\mathcal{D}_n$ to be put into the form $\rho^i\phi^j$ for $i = 0, 1, 2, \ldots, n-1$ and $j = 0, 1$. Thus, given two elements $\rho^i\phi^j$ and $\rho^k\phi^\ell$ of $\mathcal{D}_n$, we obtain the product

$$\begin{aligned} j = 0: \qquad \rho^i\rho^k\phi^\ell &= \rho^{i+k}\phi^\ell \\ j = 1: \qquad \rho^i\phi\rho^k\phi^\ell &= \rho^{i-k}\phi^{\ell+1} \end{aligned}$$

As with the cyclic group, the modulus n is carried around in each element of the group. The group operation may be implemented as follows.

As with the modular-addition data type, we reduce the exponent of ρ mod n at construction time, and likewise we reduce the exponent of ϕ mod 2, to obtain unique representatives. This permits the `__eq__` and `__ne__` methods to do exact comparisons.

```
class dih_t:
    def __init__(self, argrot, argflip, argn):
        self.n    = argn
        self.rot  = argrot  % self.n
        self.flip = argflip & 1
```

```
def __eq__(a,b):
    return ((a.rot == b.rot) and (a.flip == b.flip))
def __ne__(a,b):
    return not (a == b)
def __mul__(a,b):
    if (a.n != b.n):
        raise RuntimeError
    if (a.flip):
        crot = a.rot - b.rot
    else:
        crot = a.rot + b.rot
    c = dih_t(crot, a.flip ^ b.flip, a.n)
    return c
...
```

Fill routines

Loop over exponents on ρ, namely 0 to $n-1$, and flip exponents 0 to 1:

```
def get_elements(params_string):
    n = dih_tm.params_from_string(params_string)
    elts = []
    for i in range(0, n):
        for j in range(0, 2):
            elt = dih_tm.dih_t(i, j, n)
            elts.append(elt)
    return elts
```

Algorithms for abstract groups

Now that we can do arithmetic on group elements, and now that we can obtain groups, we can do **abstract** computations on various **concrete** groups. Thanks to Python's heterogeneous lists, operator overloading, and run-time binding, we can write some very straightforward code to do this.

Data structures

Groups are represented simply as lists of elements.

Set routines

A few self-explanatory set-related routines:

```
def element_of(x, S):
    for a in S:
        if (a == x):
            return 1
    return 0
def subset_of(T, S):
    for t in T:
        if (not element_of(t, S)):
            return 0
    return 1
def set_append_unique(S, x):
    if (not element_of(x, S)):
        S.append(x)
```

Group axioms

It is easy to check whether an array of elements is in fact a group:

```
def is_group(G):
    if (not is_closed(G)):
        return 0
    if (not is_associative(G)):
        return 0
    if (not has_unique_id(G)):
        return 0
    if (not has_inverses(G)):
        return 0
    return 1
```

Here is the associativity-checking routine; the others are similar. As well, the is-abelian routine follows the same pattern.

```
def is_associative(G):
    for a in G:
        for b in G:
            ab = a * b
            for c in G:
                bc = b * c
                ab_c = ab * c
                a_bc = a * bc
                if (ab_c != a_bc):
                    return 0
    return 1
```

Orders

Order of a group: `len(G)`. Order of an element:

```
def get_order(x):
    xp = x * x
    k = 2
    while (1):
        if (xp == x):
            return k-1
        xp = xp * x
        k = k + 1
    return 0
```

There is a trick here: We could require that the group's identity e be passed in as a separate argument, then find the minimal positive exponent j such that $x^j = e$. Instead, we can find the minimal positive exponent k such that $x^k = x$, then return $k - 1$.

This routine permits another pair of concepts to be implemented easily:

- Maximal element order: For each element x of the group G, compute the element order of x. Take the maximum of all these.

- Cyclicity: Compute the maximal element order. Test it for equality against the group order.

Example:

```
kerl@gila:pyaa% sack s:6 order .
720
kerl@gila:pyaa% sack a:6 order .
360
```

Example:

```
kerl@gila:pyaa% sack a:4 orders .
0,1,2,3 1
0,2,3,1 3
0,3,1,2 3
1,0,3,2 2
1,2,0,3 3
1,3,2,0 3
2,0,1,3 3
2,1,3,0 3
2,3,0,1 2
3,0,2,1 3
3,1,0,2 3
3,2,1,0 2
```

Cayley table

Printing a Cayley table uses the fact that Python's `print` statement writes a carriage return unless its arguments are followed by a comma. The key point here is that due to run-time binding, Python invokes the `__str__` method appropriate to each object in the list.

```
def print_cayley_table(G):
    for a in G:
        for b in G:
            c = a*b
            print c,
        print
```

Example:

```
kerl@gila:pyaa% sack q8 caytbl .
 1 -1  i -i  j -j  k -k
-1  1 -i  i -j  j -k  k
 i -i -1  1  k -k -j  j
-i  i  1 -1 -k  k  j -j
 j -j -k  k -1  1  i -i
-j  j  k -k  1 -1 -i  i
 k -k  j -j -i  i -1  1
-k  k -j  j  i -i  1 -1
```

Cosets

A routine to compute left cosets:

```
def left_cosets(G, H):
    oG = len(G)
    oH = len(H)
    iGH = oG / oH
    ... (error-checking here: does |H| divide |G|?)
    GH = []
    for g in G:
        gHe = range(0, oH) # Make a list of length |H|
        for j in range(0, oH):
            gHe[j] = g * H[j]
        gH = coset(gHe)
        set_append_unique(GH, gH)
    return GH
```

Direct products

A SACK tuple is just an object containing a list of elements. Multiplication (overloaded multiplication operator!) of two such objects is done elementwise.

```
class tuple:
    def __init__(self, slots):
        self.slots = copy.copy(slots)
    def __mul__(a,b):
        n = len(a.slots)
        c = tuple(a.slots)
        for i in range(0, n):
            c.slots[i] = a.slots[i] * b.slots[i]
        return c
```

Given that, it is straightforward to construct the direct product of two given groups:

```
def direct_product(G1, G2):
    n1 = len(G1)
    n2 = len(G2)
    n3 = n1 * n2
    G3 = []
    for i in range(0, n1):
        for j in range(0, n2):
            G3.append(tuple([G1[i], G2[j]]))
    return G3
```

Nilpotency and solvability

The basic ingredient is the **commutator**:

$$[x, y] = xyx^{-1}y^{-1}.$$

```
def commutator(x, y):
    return x * y * x.inv() * y.inv()
```

Closing a finite group, given generators: since our groups are finite, we need only to compute closure with respect to addition, not inversion. (In a finite group, the inverse of any element is obtained by raising it to a sufficiently high postiive exponent.) Idea: (1) Remember the size of the set. (2) Add all pairwise products to the set, if they are not already there. (3) Repeat steps 1 and 2 until no more elements are added.

```
def close_group(G):
    while (1):
        n = len(G)
        for i in range(0, n):
            x = G[i]
            for j in range(0, n):
                y = G[j]
                xy = x * y
                yx = y * x
                set_append_unique(G, xy)
                set_append_unique(G, yx)
        if (len(G) == n):
            return
```

Usual characterization of nilpotency is via ascending central series. Grove's *Algebra* offers a more convenient characterization of nilpotency: Define

$$[G, H] = \langle [g, h] : g \in G, h \in H \rangle.$$

Define

$$L_1 = [G, G] \quad \text{and} \quad L_{k+1} = [G, L_k], k > 1.$$

Then G is nilpotent iff $L_n(G) = \{1\}$ for some n.

```
def nilbracket(G, Gi):
    G2 = []
    for a in G:
        for b in Gi:
            set_append_unique(G2, commutator(a, b))
    close_group(G2)
    return G2
```

Compute L_k for successively higher k, looping until $|L_{k+1}| = |L_k|$; G is nilpotent iff that stable size is 1.

```
def is_nilpotent(G):
    Gp = copy.copy(G)
    while (1):
        Gpp = nilbracket(G,Gp)
        oGp  = len(Gp)
        oGpp = len(Gpp)
        if (oGpp == 1):
            return 1
        if (oGp == oGpp):
            return 0
        Gp = Gpp
```

The routines for derived subgroup and solvability follow the same pattern.

Examples:

```
kerl@gila:pyaa% sack d:3 solvable .
  solvable
kerl@gila:pyaa% sack d:4 solvable .
  solvable
kerl@gila:pyaa% sack d:2 nilpotent .
  nilpotent
kerl@gila:pyaa% sack d:3 nilpotent .
  non-nilpotent
kerl@gila:pyaa% sack d:4 nilpotent .
  nilpotent
kerl@gila:pyaa% sack d:6 nilpotent .
  non-nilpotent
kerl@gila:pyaa% sack d:16 nilpotent .
  nilpotent
```

Further directions

The software sketched here involves groups written multiplicatively. However, in Python one may overload any arithmetic operators, so there is no reason one cannot write similar software to deal with rings, fields, linear algebra, tensor products, and so on.