E. Normaliz interactive: PyNormaliz

PyNormaliz serves three purposes:

- It is the bridge from Normaliz to SageMath.
- It provides an interactive access to Normaliz from a Python command line.
- It is a flexible environment for the exploration of Normaliz.

In the following we describe the use of PyNormaliz from a Python command line and document the basic functions that allow the access from SageMath.

For a brief introduction please consult the PyNormaliz tutorial at `https://nbviewer.jupyter.org/github/Normaliz/PyNormaliz/blob/main/doc/PyNormaliz_Tutorial.ipynb`.

You can also open the tutorial for PyNormaliz interactively on `https://mybinder.org` following the link `https://mybinder.org/v2/gh/Normaliz/NormalizJupyter/master`.

E.1. Installation

The PyNormaliz install script assumes that you have executed the

`install_normaliz_with_eantic.sh`

script. (It is however possible to install PyNormaliz with fewer optional packages.) In the following we assume that PyNormaliz resides in the subdirectory `PyNormaliz` of the Normaliz directory. This automatically the case if you have downloaded a Normaliz source package. If you have obtained Normaliz or PyNormaliz in another way, make sure that our assumption is satisfied.

To install PyNormaliz navigate to the Normaliz directory and type

```
./install_pynormaliz.sh --user
```

The script detects your Python3 version, assuming the executable is in the `PATH`. Note that the installation stores the produced files in `~/.local`.

If you want to install PyNormaliz system wide, replace `--user` by `--sudo`. Then you will be asked for your root password. The following additional options are available for `install_pynormaliz.sh`:

- `--python3 <path>`: Path to a python3 executable.
- `--prefix <path>`: Path to the Normaliz install path

Depending on your setup, you might be able to install PyNormaliz via pip, typing

```
pip3 install PyNormaliz
```

at a command prompt.

The installation requires the `setuptools`. If you are missing them install them with `pip3`.

E.2. The high level interface by examples

PyNormaliz has a high level interface which allows a very intuitive use. We load PyNormaliz:

```
winfried@ryzen:~$ python3
Python 3.6.9 (default, Oct  8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import PyNormaliz
>>> from PyNormaliz import *
```

E.2.1. Creating a cone

The only available class in PyNormaliz is `Cone`. As often in this manual, “cone” includes a lattice of reference, unless we are working in an algebraic number field. We come back to this case below. First we have to create a cone (and a lattice). We can use all input types that are allowed in Normaliz input files. They must be given as named parameters as in the following example:

```
>>> C = Cone(cone = [[1,3],[2,1]])
```

This is the example from Section 2.3. There can be several input matrices. The example shows us how Normaliz matrices are represented as Python types: each row is a `list`, and the matrix then is a `list` whose members are the lists representing the rows. Important: This encoding matches exactly the formatted matrices in Normaliz input files.

It is possible to use (decimal) fractions in the input, but they must be encoded as strings. Our cone from above could be defined by

```
>>> C = Cone(cone = [[1,"3.0"],[1,"1/2"]])
```

This creates a `Cone<mpz_class>` on the Normaliz side. One can also create a `Cone<long long>` by

```
>>> C = Cone(cone = [[1,"3.0"],[1,"1/2"]], CreateAsLongLong = True)
```

In the following `Cone` (with a capital C) is a class defined in `PyNormaliz.py`. An instance of this class contains an `NmzCone` which is the Python equivalent of a `Cone<Integer>` defined on the Normaliz side. The `NmzCone` in the `Cone` `C`, is referred to by `C.cone`. This is only important when one wants to access the low level interface.

One can create a cone from a Normaliz input file as follows:

```
C = Cone(file = "example/small")
```

It will read the file `small.in` in the directory `example`/relative to the current directory. `CreateAsLongLong = True` can be used.

For polynomial constraints one uses commands like

```
PolyEq = ["x[1] -x[2]^2", "x[2]*x[3] - 27"]
C.SetPolynomialEquations(PolyEQ)
```

The argument of `SetPolynomialEquations` is a list of strings in which each component represents a polynomial expression. See Sections 4.1.8 and 4.9. The equations are always of type $f(x) = 0$. Similarly, inequalities defined by

```
C.SetPolynomialInequalities(PolyEQ)
```

are vinterpreted as $f(x) \geq 0$.

Selected input types:

- Homogeneous generators: `polytope, subspace, cone, cone_and_lattice, lattice monoid`
- Inhomogeneous generators: `vertices`
- Homogeneous constraints: `inequalities, signs, equations, congruences`
- Inhomogeneous constraints: `inhom_equations, inhom_inequalities, inhom_congruences`
- Linear forms: `grading, dehomogenization`
- Lattice ideals and friends: `lattice_ideal, toric_ideal, normal_toric_ideal`

For explanantions and other input types se the Normaliz manual. The input type `constraints` can't be used in PyNormaliz. Also shortcuts like `nonnegative` or `total_degree` are not available.

E.2.2. Matrices, vectors and numbers

The matrix format of the input is of course also used in PyNormaliz results:

```
>>> C.HilbertBasis()
[[1, 1], [1, 2], [1, 3], [2, 1]]
```

PyNormaliz contains some functions that help reading complicated output. For matrices we can use

```
>>> print_matrix(C.HilbertBasis())
1 1
1 2
1 3
2 1
```

Similarly

```
>>> print_matrix(C.SupportHyperplanes())
-1  2
3 -1
```

Since our input defines an original monoid, we can ask for the module generators over it:

```
>>> print_matrix(C.ModuleGeneratorsOverOriginalMonoid())
0 0
1 1
1 2
2 2
2 3
```

Binomials are retrieved in the same way:

```
>>> print_matrix(C.MarkovBasis())
-1  2 -1 0
-3  1  0 1
```

```
-2 -1  1 1
```

In this connection note that you can set upper and lower bounds for the degrees in the output of Markov and Gröbner bases:

```
C.SetGBDegreeBound(3)
C.SetGBMinDegree(2)
```

If you want to set a monomial order for the Gröbner basis, you must use the `Compute` function:

```
C.Compute("GroebnerBasis", "Lex")
C.GroebnerBasis()
```

Some numerical invariants:

```
>>> C.Rank()
2
>>> C.EmbeddingDim()
2
>>> C.ExternalIndex()
1
>>> C.InternalIndex()
5
```

If we want to know whether a certain cone property has already been computed, we can ask for it:

```
>>> C.IsComputed("HilbertBasis")
True
```

The essential point is that this query does *not* force the computation if the property has not yet been computed. There are several more computation goals that come as matrices, vectors or numbers. We list all of them:

- Matrices: `ExtremeRays, VerticesOfPolyhedron, SupportHyperplanes, HilbertBasis, ModuleGenerators, Deg1Elements, LatticePoints, ModuleGeneratorsOverOriginalMonoid, ExcludedFaces, OriginalMonoidGenerators, MaximalSubspace, Equations, Congruences, GroebnerBasis, Representations`
- Matrices with floating point entries: `ExtremeRaysFloat, SuppHypsFloat, VerticesFloat`
- Vectors: `Grading, Dehomogenization, WitnessNotIntegrallyClosed, GeneratorOfInterior, CoveringFace, AxesScaling`
- Numbers: `TriangulationSize, NumberLatticePoints, RecessionRank, AffineDim, ModuleRank, Rank, EmbeddingDim, ExternalIndex, TriangulationDetSum, GradingDenom, UnitGroupIndex, InternalIndex,`

The numbers have several different representations on the Normaliz side. In Python they are all (long) integers.

E.2.3. Triangulations, automorphisms and face lattice

Some of the raw output is complicated:

```
>>> U = C.UnimodularTriangulation()
>>> U
[[[[1, 2], 1, []], [[2, 3], 1, []], [[0, 3], 1, []]], [[1, 3], [2, 1], [1, 1], [1, 2]]]
```

Taking a close look, we see two members of the outermost `list`. The second is an ordinary matrix, namely the matrix of the rays of the triangulation:

```
>>> print_matrix(U[1])
1 3
2 1
1 1
1 2
```

The first member is not a matrix, but close enough so that we can use `print_matrix`:

```
>>> print_matrix(U[0])
[1, 2] 1 []
[2, 3] 1 []
[0, 3] 1 []
```

In each line we find the information on a simplicial cone, first the list of the rays by their indices relative to the matrix of rays (counting rows from 0). The next is the determinant relative to a lattice basis (in our case the unit vectors). In a unimodular triangulation these determinants must of course be 1. The third component is the list of excluded faces if we have computed a disjoint decomposition. This is explained in Section 7.14.2.

To see an even more complicated data structure we ask for the combinatorial automorphisms:

```
>>> G = C.CombinatorialAutomorphisms()
>>> G
[2, Faase, False,  [[[1, 0]], [[0, 1]]], [[], []], [[[1, 0]], [[0, 1]]]]
```

There are 6 components on the outermost level. The first is the order of the group. The second amswers the question whether the integrality of the automorphisms has been checked. The answer is always "no" for compinatorial automorphisms, and therefore the third give the answer "no" to the question whether the automorphisms are integral.

The next three contain information on the

- extreme rays of the (recession) cone,
- the vertices of the polyhedron,
- he support hyperplane

in this order. In each of them we find

- the action of the group generators on the respective vectors,
- their orbits under the group.

In our case there are no vertices of the polyhedron (only defined for inhomogeneous input). This explains the empty `list`. Fortunately we can print the complicated result nicely with an explanation:

```
>>> print_automs(G)
order  2
```

```
permutations of  extreme rays of (recession) cone
0 :  [1, 0]
orbits of  extreme rays of (recession) cone
0 :  [0, 1]
permutations of  support hyperplanes
0 :  [1, 0]
orbits of  support hyperplanes
0 :  [0, 1]
```

It makes sense to have a look at Section 7.22. (Here we count from 0.)

`AmbientAutomorphisms` and `InputAutomorphisms` yield a slightly different result. The permutations and orbits in the third element of the outer list now refer to the input vectors. The fourth element gives data for thempty set, as does the fifth for `InputAutomorphisms` . For `AmbientAutomorphisms` it lists the permutation and oprbits of the coordinates of the ambient lattice. All this is folloowed by the input vectors for reference. A simple example:

```
>>> C = Cone(cone = [[0,1],[1,0]])
>>> C.AmbientAutomorphisms()
[2, True, True, [[[1, 0]], [[0, 1]]], [[], []], [[[1, 0]], [[0, 1]]], [[0, 1], [1, 0]]]
>>> print_automs(C.AmbientAutomorphisms())
order  2
automorphisms are integral
permutations of  input vectors
0 :  [1, 0]
orbits of  input vectors
0 :  [0, 1]
permutations of  coordinates
0 :  [1, 0]
orbits of  coordinates
0 :  [0, 1]
input vectors
0 1
1 0
```

Of course, we also want to know the face lattice:

```
>>> C.FaceLattice()
[[[0, 0], 0], [[1, 0], 1], [[0, 1], 1], [[1, 1], 2]]
```

Hard to read. Much better:

```
>>> print_matrix(C.FaceLattice())
[0, 0] 0
[1, 0] 1
[0, 1] 1
[1, 1] 2
```

So there are four faces. The `list` contains the support hyperplanes that meet in the face and the number is the codimension. The support hyperplanes are given by their row indices relative to the matrix of support hyperplanes. Also see Section 7.17. The f-vector:

```
>>> C.FVector()
[1, 2, 1]
```

If you want to limit the codimension of the faces computed with `FaceLattice` or `FVector`, set the bound by

```
>>> SetFaceCodimBound(1)
```

Try it and ask for `FaceLattice` once more. If you want to get rid of a previously set bound:

```
>>> SetFaceCodimBound()
```

or take -1 as the argument.

We also have a printer for the Stanley decomposition:

```
>>> print_Stanley_dec(C.StanleyDec())
```

Try it.

The cone properties that fall into the categories discussed in this section are: `Triangulation`, `UnimodularTriangulation`, `LatticePointTriangulation`, `AllGeneratorsTriangulation`, `PlacingTriangulation`, `PullingTriangulation`, `StanleyDec`, `InclusionExclusionData`, `Automorphisms`, `CombinatorialAutomorphisms`, `RationalAutomorphisms`, `EuclideanAutomorphisms`, `AmbientAutomorphisms`, `InputAutomorphisms`, `FaceLattice`, `DualFaceLattice`, `FVector`, `DualFVector`, `Incidence`, `DualIncidence`.

E.2.4. Hilbert and other series

Now we turn to the Hilbert series.

```
>>> C.HilbertSeries()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/home/winfried/../PyNormaliz.py", line 403, in inner
return self._generic_getter(name, **kwargs)
File "/home/winfried/.../PyNormaliz.py", line 393, in _generic_getter
PyNormaliz_cpp.NmzCompute(self.cone, input_list)
PyNormaliz_cpp.NormalizError: Could not compute:
No grading specified and cannot find one. Cannot compute some requested properties!
```

Indeed, we forgot the grading. We could have added it at the time of construction

```
>>> C = Cone(cone = [[1,3],[2,1]], grading = [[1,2]])
```

where it must be given as a matrix with a single row. Or we can add it later:

```
>>> C.SetGrading([1,2])
```

(A similar function is `SetProjectionCoords`.) We check the grading:

```
>>> C.Grading()
```

```
[[1, 2], 1]
```

The number 1 following the vector is the grading denominator.

Now:

```
>>> C.HilbertSeries()
[[1, -1, 0, 1, 0, 0, 0, 1, 0, -1, ..., 0, 0, 0, 0, 1, -1, 1], [1, 28], 0]
```

For space reasons we have omitted some components in the first `list`, the numerator of the Hilbert series. The second gives the denominator, and the last is the shift. Much nicer:

```
>>> print_series(C.HilbertSeries())
(1 -  t +  t^3 +  t^7 -  t^9 +  t^10 +  t^12 -  t^13 +  t^14 +  t^19 +  t^24 -  t^25 +  t^26)
-------------------------------------------------------------------------------------------
(1 - t) (1 - t^28)
```

Options can be added as named parameters:

```
>>> print_series(C.HilbertSeries(HSOP = True))
(1 +  t^3 +  t^5 +  t^6 +  t^8)
-------------------------------
(1 - t^4) (1 - t^7)
```

This representation is much more natural in this case. Perhaps we want so see the Hilbert quasipolynomial:

```
>>> print_quasipol(C.HilbertQuasiPolynomial())
28 5
-5 5
...
10 5
5 5
divide all coefficients by  28
```

In this case it seems better to print the polynomials as vectors of coefficients.

If the quasipolynomial has a large period and high degree, you may want to restrict the information to only a few coefficients from the top:

```
SetNrCoeffQuasiPol(bound)
```

The bound -1 or `SetNrCoeffQuasiPol()`mean “all”, in case you want to get rid of the previously set bound.

Normaliz can compute the values of the coefficients of the Hilbert series for you:

```
>>> C.HilbertSeriesExpansion(10)
[1, 0, 0, 1, 1, 1, 1, 2, 2, 1, 2]
```

For the weighted Ehrhart series we need a polynomial. Let’s add it (can also be done in the constructor with `polynomial = <string>`):

```
>>> C.SetPolynomial("x[1]+x[2]")
```

```
True
```

Then

```
print_series(C.WeightedEhrhartSeries())
```

We don't show the result because it is too long for this manual.

The cone properties of this section: `HilbertSeries, HilbertQuasiPolynomial, EhrhartSeries, EhrhartQuasiPolynomial, WeightedEhrhartSeries, WeightedEhrhartQuasiPolynomial`

E.2.5. Multiplicity, volume and integral

The first time we see a fraction printed as such:

```
>>> C.Multiplicity()
'5/28'
```

Since Python has no built-in type for fractions, we print it as a string.

```
>>> C.EuclideanVolume()
'0.3993'
```

The decimal fractions is rounded to 4 decimals. If you need more precision, you can directly use the low level interface:

```
>>> NmzResult(C.cone,"EuclideanVolume")
0.39929785312496247
```

By default, the low level interface returns raw values. We use it once more:

```
>>> NmzResult(C.cone,"EuclideanIntegral")
0.2638217958147073
```

We have integrated our polynomial from above. In case we have forgotten it:

```
>>> C.Polynomial()
'x[1]+x[2]'
```

For computations with fixed precision one can specify the number of decimal digits:

```
>>> C.setDecimalDigits(50)
```

This function is hardly necessary, since the default value of `100` is almost always satisfactory.

The cone properties of this section: `Multiplicity, Volume, Integral, VirtualMultiplicity, EuclideanVolume, EuclideanIntegral, ReesPrimaryMultiplicity`

E.2.6. Integer hull and other cones as values

Let us define a nonintegral polytope (we vary the format of the numbers on purpose):

```
>>> R = Cone(vertices = [["-3/2", ’7/5’,1], [9,-15,4], ["7.0",8,3]])
>>> R.VerticesOfPolyhedron()
[[-15, 14, 10], [7, 8, 3], [9, -15, 4]]
```

The last component of each vector acts as the denominator of the first two, and we recognize the fractions in the input. Numerical invariants available with inhomogeneous input:

```
>>> R.AffineDim()
2
>>> R.RecessionRank()
0
>>> R.LatticePoints()
[[-1, 1, 1], [0, 0, 1], [0, 1, 1], [1, -2, 1], ...  [2, -1, 1], [2, 0, 1], [2, 1, 1], [2, 2, 1]]
>>> H = R.IntegerHull()
>>> H
<Normaliz Cone>
```

So we have computed a new cone, the cone over the polytope (in this case) spanned by the lattice points in the polytope with rational vertices `[[-15, 14, 10], [7, 8, 3], [9, -15, 4]]`.

```
>>> H.VerticesOfPolyhedron()
[[-1, 1, 1], [1, -2, 1], [1, 2, 1], [2, -3, 1], [2, 2, 1]]
```

The last component is 1 as it must be for lattice points of the polytope.

```
>>> print_matrix(H.SupportHyperplanes())
-1  0 2
 0 -1 2
 1 -2 3
 1  1 1
 3  2 1
```

The other computations that return a cone are `ProjectCone` and `SymmetrizedCone`.

E.2.7. Boolean values

We ask our cone `C` many questions:

```
>>> C.IsGorenstein()
False
>>> C.IsDeg1HilbertBasis()
False
>>> C.IsDeg1ExtremeRays()
False
>>> C.IsPointed()
True
>>> C.IsInhomogeneous()
False
```

```
>>> C.IsEmptySemiOpen()
...
PyNormaliz_cpp.NormalizError: ...: IsEmptySemiOpen can only be computed with excluded faces
>>> C.IsIntegrallyClosed()
False
>>>
>>> C.IsReesPrimary()
...
PyNormaliz_cpp.NormalizError: Could not compute: IsReesPrimary !
```

E.2.8. Algebraic polyhedra

For an algebraic polyhedron we must define the real embedded number field over which the polyhedron is living. This information is given in the cone constructor:

```
>>> A = Cone(number_field=[ "a^2-2", "a", "1.4+/-0.1" ],
             vertices = [["1/2a", "13/3",1], ["-3a^1",-6,2], [-6, "-1/2a-7",1]])
>>> print_matrix(A.VerticesOfPolyhedron())
    -6 -1/2*a-7 1
-3/2*a       -3 1
 1/2*a     13/3 1
>>> print_matrix(A.VerticesFloat())
-6.0000 -7.7071 1.0000
-2.1213 -3.0000 1.0000
 0.7071  4.3333 1.0000
>>> A.RenfVolume()
'-19*a+42'
>>> A.EuclideanVolume()
'7.5650'
>>> print_matrix(A.LatticePoints())
-5 -6 1
...
-1  1 1
0  3 1
>>> A.NumberFieldData()
('a^2 - 2', '[1.4142135623730950488016887242096980785696718753769480731766797 38 +/- 3.57e-64]')
>>> A.GetFieldGeneratorName()
'a'
```

The only point to notice is `RenfVolume` that we must use instead of `Volume` here. The number field data show you to what precision $\sqrt{2}$ had to be computed to make all decisions about positivity for our little polytope.

E.2.9. The collective compute command and algorithmic variants

So far we have asked Normaliz for a single cone property. It is also possible to bundle several computation goals and options in a single compute command:

```
>>> C.Compute("HilbertBasis", "HilbertSeries", "ClassGroup", "DualMode")
True
>>> C.IsComputed("ClassGroup")
True
>>> C.ClassGroup()
[0, 5]
```

which means that the class group is isomorphic to $\mathbb{Z}/(5)$. The first number 0 indicates that the class group has rank 0.

The collective compute command not only allows you to sset several computation goals simultaneously. It allows you to specify algorithmic variants, like `DulaMode`. There is a whole collection of variants explained elsewhere in this manual:

```
DefaultMode, Approximate, BottomDecomposition, NoBottomDec, DualMode,
PrimalMode, Projection, ProjectionFloat, NoProjection, Symmetrize, NoSymmetrization,
NoSubdivision, NoNestedTri, KeepOrder, HSOP, NoPeriodBound, NoLLL, NoRelax, Descent, NoDescent,
NoGradingDenom, GradingIsPositive, ExploitAutomsVectors (not yet implemented), ExploitIsosMult,
StrictIsoTypeCheck, SignedDec, NoSignedDec, FixedPrecision
```

E.2.10. Miscellaneous functions

In order to get some information about what is going on in Normaliz, we can switch on the terminal output:

```
>>> C = Cone(cone = [[1,3],[2,1]], grading = [[1,2]])
>>> C.SetVerbose()
False
>>> C.HilbertBasis(DualMode = True)
Computing support hyperplanes for the dual mode:
************************************************************
starting full cone computation
Generators sorted lexicographically
Starting primal algorithm (only support hyperplanes) ...
Start simplex 1 2
Pointed since graded
Select extreme rays via comparison ... done.
------------------------------------------------------------
transforming data... done.
************************************************************
computing Hilbert basis ...
==================================================
cut with halfspace 1 ...
Final sizes: Pos 1 Neg 1 Neutral 0
```

```
==================================================
cut with halfspace 2 ...
Final sizes: Pos 3 Neg 3 Neutral 1
Hilbert basis 4
Find degree 1 elements
transforming data... done.
[[1, 1], [2, 1], [1, 2], [1, 3]]
```

The return value of `SetVerbose` is the *old value* of *verbose*. We had to redefine `C` to get of the already computed Hilbert basis. The very last line is our Hilbert basis.

If we want to see all data computed for `C`, call

```
>>> C.print_properties()
ExtremeRays:                        NumberLatticePoints:
[[2, 1], [1, 3]]                    0
SupportHyperplanes:                 Rank:
[[-1, 2], [3, -1]]                  2
HilbertBasis:                       EmbeddingDim:
[[1, 1], [2, 1], [1, 2], [1, 3]]    2
Deg1Elements:                       IsPointed:
[]                                  True
OriginalMonoidGenerators:           IsDeg1ExtremeRays:
[[1, 3], [2, 1]]                    False
MaximalSubspace:                    IsDeg1HilbertBasis:
[]                                  False
Grading:                            IsIntegrallyClosed:
[[1, 2], 1]                         False
GradingDenom:                       IsInhomogeneous:
1                                   False
UnitGroupIndex:                     Sublattice:
1                                   [[[1, 0], [0, 1]], [[1, 0], [0, 1]], 1]
InternalIndex:
```

Typeset in two columns. The last property we see is `Sublattice`. It consists of two matrices and a number. See Section D.8.16 for the interpretation.

Finally, we can write a Normaliz output file:

```
>>> C.WriteOutputFile("Wonderful")
True
```

Now you should find a file `Wonderful.out` in the current directory.

One can also write a file for the inout of precomputed data:

```
>>> C.WritePrecompData("Wonderful")
True
```

It creates the file `Wonderful.precomp.in`.

E.3. The low level interface

The low level interface is contained in `NormalizModule.cpp`. Its functions are listed in `PyNormaliz_cppMethods[]`. They allow the construction of an `NmzCone` (accompanied by a lattice), the computation in it, and give access to the computation results. The use of the low level interface is indirectly explained by the examples above. Therefore we keep the discussion short.

E.3.1. The main functions

For the construction one uses

```
NmzCone(**kwargs)
```

The keyword arguments `kwargs` transport Normaliz input types and the corresponding matrices in Python format. In addition we must use `number_field` for algebraic polyhedra. You can use `polynomial` for computations with a polynomial weight. (There is also an extra function for setting the polynomial; see below.) You can also ask for a `Cone<long long>` by adding `CreateAsLongLong = True`.

Once and for all: in the functions listed in the following that apply to a specific `NmzCone`, this `NmzCone` must be the first argument in `*args`.

Computations are started by

```
NmzCompute(*args)
```

The arguments list the computation goals and options as strings.

Access to the computation results is given by

```
NmzResult(*args, **kwargs)
```

There must be exactly two positional arguments. The first is the `NmzCone`, the second names the result to be returned, given as a string.

The `*kwargs` specify handlers, routines that format the raw results of output types that are not existent in Python or should be formatted for another reason. The potential handlers:

`RatHandler` defines the formatting of fractions.
`FloatHandler` defines the formatting of floating point numbers.
`NumberfieldElementHandler` defines the formatting of number field elements.
`VectorHandler` defines the formatting of vectors.
`MatrixHandler` defines the formatting of matrices.

The default handler for vectors and matrices is `list`, and there is not be much point in changing it. If you don't like lists, you can set `VectorHandler=tuple`, for example. But especially `RatHandler` and `NumberfieldElementHandler` are very useful since the raw versions are difficult to read. Examples of handlers can be found in `PyNormaliz.py`.

Note: When `NmzResult` is called, its first action is to reset the handlers to the raw format. Then the `kwargs` are evaluated. In other words: the values of the handlers are only applied to the current result, and not to future ones.

In the same way as the data access functions of Normaliz, `NmzResult` triggers the computation of the required result if it should not have been computed yet. Whether a result has been computed yet can be checked by

```
NmzIsComputed(*args)
```

The second argument of exactly 2 is the result whose computation is to be checked, given as a string.

E.3.2. Additional input and modification of existing cones

These functions allow the input of data that cannot be passed through the cone constructor or modify a cone after construction. For example:

```
NmzSetGrading(cone, grading)
```

The grading is a vector encoded as a Python list. Similarly

```
NmzSetProjectionCoords(cone, coordinates)
```

where `coordinates` is a list with entries 0 or 1.

```
NmzSetPolynomial(cone, polynomial)
```

The polynomial is given as a string.

```
NmzSetNrCoeffQuasiPol(cone, number)
NmzSetFaceCodimBound(cone, number)
```

Do what the names say.

```
NmzModifyCone(cone, type, matrix)
```

This is the PyNormaliz version of the libnormaliz function modifyCone. Please have a look at Section D.6.

E.3.3. Additional data access

Some values cannot be returned as cone properties. For them we have additional access functions.

```
NmzGetPolynomial(cone)
```

returns the polynomial weight if one has been set.

The functions

```
NmzHilbertSeriesExpansion(cone, degree)
NmzEhrhartSeriesExpansion(cone, degree)
NmzWeightedEhrhartSeriesExpansion(cone, degree)
```

return the expansion of the named series up to the given degree as a list of numbers.

```
NmzIntegerHullCone(cone)
NmzProjectCone(cone)
NmzSymmetrizedCone(cone)
```

return `NmzCone`.

```
NmzGetRenfInfo(cone)
NmzFieldGenName(cone)
```

return the data defining the number field.

E.3.4. Miscellaneous functions

```
NmzSetVerbose(cone, value=True)
NmzSetVerboseDefault(value=True)
```

The first sets `verbose` to the specified value for cone, whereas the second sets it for all subsequently defined cones.

```
NmzConeCopy(cone)
```

returns a copy of cone.

```
NmzSetNumberOfNormalizThreads(number)
```

does what its name says. The previous number of threads is returned.

```
NmzWriteOutputFile(cone, project)
NmzWritePrecompData(cone, project)
```

The first writes a Normaliz output file whose name is the string `project` with the suffix `.out`, the second a file whose name is the string `project` with suffix `precomp.in`.
The functions

```
NmzHasEantic(cone)
NmzHasCoCoA(cone)
NmzHasFlint(cone)
NmzHasFlint(cone)
```

return `True` or `False`, depending on whether Normaliz has been built with the corresponding package.

```
NmzListConeProperties()
```

lists all cone properties in case you should have forgotten any of them.

```
error_out(PyObject* m)
```

writes an error message if something bad has happened.

E.3.5. Raw formats of numbers

All Normaliz integers are transformed to Python long integers, and floating point numbers are transformed to Python floats.

Numbers of type `mpq_class` are represented by a `list` with two components on the Python side, namely the numerator and the denominator.

An algebraic number is represented by a `list` whose members are rational numbers each of which is a `list` with two members. They are the coefficients of the polynomial representing the algebraic number.

F. Distributed computation for volume via signed decomposition

Normaliz offers a possibility to compute volumes via signed decomposition by distributing the task to several computers or nodes in a gigh performance cluster that run independently of each other. The principal approach:

(1) The first step is the computation of the hollow triangulation and the generic vector on a single machine. This step can require considerable time and memory.
(2) These data (and some more) are written to "hollow tri" files.
(3) The files are read by Normaliz with the `--Chunk` option that makes it compute the contribution to the volume that comes from a single data file ("chunk") and write this volume to a "mult" file.
(4) A final run of Normaliz with the `--AddChunks` option so that it reads all the "mult" files and adds the partial volumes.

This is certainly a robust and flexible approach to distributed computation. While the main purpose of distributed computation is a massive increase in parallelization, one should consider its use even if the computation is done on a single machine. It limits the loss of data caused by system crashes or similar interruptions to a small amount and allows easy repair. Another advantage is that the most time consuming step (3) needs very little RAM for a single "chunk", compared to step (1).

To make Normaliz write the data files and to stop once they have been written, one uses the cone property

`DistributedComp`

The size of the locks can be set by

`block_size_hollow_tri <n>`

to the input file where `<n>` is the number of simplices of the full triangulation that should go into a single output file. The default value chosen by `DistributedComp` is $500,000$.

The output files are

`<project>.hollow_tri.<n>.gz`

where `<n>` numbers these files consecutively, starting from 0. As usual, `<project>` is the name of the project. These files are gzipped to save disk space.

Moreover, there is a common data file:

`<project>.basic.data`