PYGILDAS: Interleaving Python and GILDAS

End-user and developer documentation

Sébastien Bardeau, Emmanuel Reynier,
Jérôme Pety and Stéphane Guilloteau

24-apr-2018

Version 0.8

1 Context

Goal: to create an inter-operable environment in which a SIC driven program can also use all Python facilities and vice-versa: a Python code could import a SIC-based program.
Method: first create a common space in which SIC and Python communicates by their respective "objects": SIC variables and Python `NumPy` arrays.
Content: description of the SIC variable to Python object mapping.

2 End-user documentation

2.1 Prerequisites *(what you have to know)*

If you wonder how to compile Gildas with the Python binding activated, please refer to section 3.1.1.

2.1.1 Python basics

Identifiers To the opposite to SIC, Python (like C) is case sensitive. SIC variables will be imported in Python variables with names arbitraryly lowercased. There are also a few forbidden characters in Python identifiers, especially '%' and '$'. The dot '.' is reserved for a special purpose. See subsection 2.3.2 for more details.

Multidimensional-array arrangement in memory Multidimensional arrays in C (and thus Python) are stored in row-major order; in Fortran they are in column-major order. For a 2-dimensional array (matrix), this means that C stores each row contiguously in memory, while Fortran stores each column contiguously. More generally, for an N-dimensional array, in C the last dimension is contiguous in memory, while in Fortran the first dimension is contiguous. This means that for the same area allocated in memory, Fortran and C indices are transposed:

$$a_{Fortran}(d_1, ..., d_N) \leftrightarrow a_C[d_N, ..., d_1]$$

where N is the number of dimensions and d_i the i^{th} dimension.
Remember also that first element through one dimension has index 1 in Fortran, and index 0 in C. Finally, we can write:

$$a_{Fortran}(i, j, ..., k) \equiv a_C[k-1, ..., j-1, i-1]$$

This element is the same but accessed from Fortran or from C respectively.

The `vars()` built-in function With no arguments, displays the dictionary[1] of the current name area. Thus `vars().keys()`[2] is a list of all variables defined in the current name area. With a variable as argument, displays the attributes list of this variable.

The `dir()` built-in function With no arguments, displays the whole list of variables, functions or other objects defined in the current name area. With a variable as argument, displays the list of attributes but also methods and other objects associated to this variable.

The `__doc__` attribute Each Python object (functions, classes, variables,...) provides (if programmer has filled it) a short documentation which describes its features. You can access it with commands like `print myobject.__doc__`. You can provide a documentation for your own objects (functions for example) in the body of their definition as a string surrounded by three double-quotes:

```
>>> def myfunc():
...     """'myfunc' documentation."""
...     pass
...
>>> print myfunc.__doc__
'myfunc' documentation.
```

Global/local variables and module name spaces Python has the usual notion of global and local variables. A variable may be either visible in all the main code (it is *global*) or only in the function which defines it (it is *local*). When a variable is called in a function, Python searches it in the local name space and if not found, it searches it in the global name space.

When a module is imported, it has its own global and locals name spaces, which it does not share with the main ones. Thus, if a variable is defined as global in a module, it can only be accessed as an element of the module. Let us assume we want to import a module named `mymodule` which provides an `execute()` function. This function executes command lines in the module global name space[3].

```
>>> import mymodule
>>> mymodule.execute('a = 0')
>>> mymodule.a
0

>>> a # is not defined (or visible) in the main name space.
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
```

Note that importing all module objects in the main name space does not give a solution:

[1] a Python dictionary is an *associative array*: a set of couples keys + items.
[2] the `.keys()` (resp. `.values()`) method returns the keys (resp. the values) list of any dictionary.
[3] it may be coded this way: `def execute(string): exec(string) in globals()`

```
>>> from mymodule import execute
>>> # 'execute()' is now a member of the main name space
>>> execute('a = 0')
>>> a # is not defined...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
>>> execute('print a') # but seems defined 'somewhere'...
0
>>> print __name__ # Prints the current module name
__main__
>>> execute('print __name__') # Prints the module name the
...                           # 'execute()' function works in
mymodule
```

`execute()` function still works in `mymodule` (and defines variables as members of it) although there is no '`mymodule`' module visible in the main name space.

Finally you will have to import the brand new created variable if you want to make it visible in the main name space:

```
>>> from mymodule import a
>>> a
0
```

Note that the two objects `a` (imported from `mymodule`) and `mymodule.a` are the same object (not a copy):

```
>>> execute('b = [0, 0, 0]')
>>> from mymodule import b
>>> b
[0, 0, 0]
>>> b[0] = 1
>>> b # the one imported into 'main' from 'mymodule'
[1, 0, 0]
>>> execute('print b') # the one in 'mymodule'
[1, 0, 0]
```

2.1.2 `NumPy` basics

Array handling is provided to Python by the optional package `NumPy`. `NumPy` derives from the older module `Numeric` which documentation is available at `http://numeric.scipy.org/numpydoc/numdoc.htm`. You will have to pay for the `NumPy` documentation[4], but the `Numeric` one may be sufficient to begin handling arrays with Python. You can also find an extensive list of the `NumPy` functions, methods and attributes at `http://www.scipy.org/Numpy_Example_List_With_Doc` .
The type `ndarray` provided by `NumPy` module has a large set of attributes and methods which can help to deal with these objects. This module also brings many useful functions.

[4] see `http://www.tramy.us`

```
>>> from numpy import array, reshape
>>> a = reshape(array(range(1,25)),(2,3,4))
>>>     # integers from 1 to 24 rearranged in a 2x3x4 data cube
>>> a
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]],
       [[13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]]])
>>> type(a)
<type 'numpy.ndarray'>
```

Basic array elements access may be summarized as follows:

```
>>> a[0] # First subarray through first dimension
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> a[0][1] # Second subarray through first dimension of subarray 'a[0]'
array([5, 6, 7, 8])
>>> a[0][1][2]
7
>>> a[0,1,2] # Same as above
7
>>> j = (0,1,2)
>>> a[j] # Tuples are valid indices
7

>>> a[:] # All subarrays through first dimension, thus 'a' itself
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]],
       [[13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]]])

>>> a[0:1] # lower limit is included, upper limit excluded,
           # thus only a[0]
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]]])
>>> a[:,1:,:2] # All elements through 1st dimension,
               # all elements except 1st one (0) through 2nd dimension,
               # 1st two elements (0 and 1) through 3rd dimension:
array([[[ 5,  6],
        [ 9, 10]],
       [[17, 18],
        [21, 22]]])
```

Attributes Attributes come with any `numpy.ndarray`. Among others we can mention:

- The `.shape` built-in attribute: is a Python tuple storing the dimensions of the input array.

- The `.dtype` built-in attribute: a `numpy.ndarray` brings its datatype in a special object called `dtype`, which can be retrieved with the `.dtype` attribute. Default type is `dtype('<i4')` (*standard integer*) for integers, and `'<f8'` (*double precision*) for floats. We will import different kind of numbers from SIC (different memory size allocated per element), so the dtypes will be one of `'<f8'` (*double precision float*), `'<f4'` (*single precision float*), `'<i4'` (*integers*), or `'|S1'` (*single characters*).
- The `.itemsize` built-in attribute: is the size allocated in memory (in bytes) for each array element (see dtypes above).

Methods Methods come with any `numpy.ndarray`. Among others:

- The `.flatten()` built-in method: returns a *flat* (rank 1) vector compound with a copy of all input array elements.
- The `.tostring()` built-in method: returns a string representation of the data portion of the array it is applied to. It can be used to concatenate elements of a character (`'|S1'`) array.

Functions Functions must be imported from `NumPy` module. Among others:

- The `array()` built-in function: returns a `numpy.ndarray` from a list of elements (or list of lists,...), from a tuple (or tuple of tuples,...), or even from any valid `numpy.ndarray`. The array constructor also takes an optional `dtype` (see above), an optional `savespace` argument, and an optional `copy` argument (see below).
  Try also `ones()` or `zeros()` with a tuple containing dimensions as argument to create new arrays.
- The `size()` built-in function: displays the number of elements of the input array.
- The `rank()` built-in function: displays the number of dimensions of the input array (0 = scalar, 1 = vector, etc).
- The `len()` built-in function: displays the number of elements of the input array along the first dimension.
- See also: `reshape()`.

Subarrays and derivates An important feature is that arrays that derive from `numpy.ndarray` are not copies: they still point to the memory area of the initial array.

```
>>> from numpy import zeros
>>> a = zeros((2,3))
>>> a
array([[0, 0, 0],
       [0, 0, 0]])
>>> b = a[0] # Subarray
>>> b
array([0, 0, 0])
```

```
>>> c = reshape(a,(6,)) # 1D version of 'a'
>>> c
array([0, 0, 0, 0, 0, 0])
>>> a[0,0] = 1
>>> b[1] = 2
>>> c[2] = 3
>>> a
array([[1, 2, 3],
       [0, 0, 0]])
>>> b
array([1, 2, 3])
>>> c
array([1, 2, 3, 0, 0, 0])
```

The array constructor (built-in function) `array()` can take an optional boolean argument `copy` which indicates if the above feature applies or not to the derived array:

```
>>> a = array((0,0,0))
>>> a
array([0, 0, 0])
>>> b = array(a,copy=True)  # A copy: does not share its data with 'a'
>>> c = array(a,copy=False) # Not a copy: shares its data with 'a'

>>> a[0] = 1
>>> b[1] = 2
>>> c[2] = 3
>>> a
array([1, 0, 3])
>>> b
array([0, 2, 0])
>>> c
array([1, 0, 3])
```

2.2 Using Python from SIC

2.2.1 The SIC command PYTHON

If Python and the NumPy module was detected, SIC provides the command PYTHON with various behaviors depending on what follows (or not) on command line:

- **Starting Python**: in all cases, the first call to the command starts the Python interpreter in the background if it was not already launched. This is done only once:

```
SIC> PYTHON
Python 2.5 (r25:51908, Nov 14 2006, 22:44:28)
[GCC 4.1.1 20061011 (Red Hat 4.1.1-30)] on linux2
Entering interactive session. Type 'Sic()' to go back to SIC.
>>>
```

  The SIC variables are not yet available at this stage, assuming the user may not need them. You have to invoke the .get() method (with no arguments) to import them (see section 2.4 for details):

```
>>> pysic.get()
Importing all SIC variables into Python...
... done.
>>>
```

- **Switching to Python**: by default, if no argument follows, an interactive session is launched (see above). It has the same behavior as the standard interactive Python mode: multiline definition for functions and classes, and command history. To exit both SIC and Python, type *CTRL-D* as usual. The exit() function is also available. To go back to SIC, call the Sic object with Sic().

- **Executing a Python script**: if the first argument which follows the command PYTHON is a string ending with the three characters '.py', SIC assumes that you gave it a Python filename. SIC will look in all the MACRO#DIR: directories it usually looks in for its own procedures. SIC also solves the logical names if any. If the file is found, it will be executed by the Python interpreter in the current name space:

```
SIC> TYPE test.py
print "Hello world!"
a = 1
a
print "a value is", a

# This line is a comment
for i in xrange(1,4):
   print i

if True:
   print "True"
else:
   print "False"
```

```
SIC> PYTHON test.py
Hello world!
a value is 1
1
2
3
True
SIC>
```

Note that the command `a = 1` does not print any output. You have to explicitely request a print to see any value. After the execution you fall back to the SIC prompt. If the `SIC%VERIFY` flag is set to `ON`, commands are printed before execution:

```
SIC> SIC VERIFY ON
I-SIC,  VERIFY       is ON
SIC> PYTHON test.py
>>> print "Hello world!"
Hello world!
>>> a = 1
>>> a
>>> print "a value is", a
a value is 1
>>> for i in xrange(1,4):
...    print i
1
2
3
>>> if True:
...    print "True"
... else:
...    print "False"
True
SIC>
```

- **Passing arguments to Python scripts**: You can pass arguments to the Python script after its name. They will be available in the `sys.argv` list as usually when you launch Python scripts from shell:

```
SIC> TYPE test2.py
import sys
for i in sys.argv:
  print i, type(i)
SIC> PYTHON test2.py 1 qwerty "ABCD"
test2.py <type 'str'>
1 <type 'str'>
qwerty <type 'str'>
"ABCD" <type 'str'>
SIC>
```

Arguments are also parsed by SIC before being sent to Python:

```
SIC> DEFINE DOUBLE A
SIC> LET A 1.234
SIC> py test2.py A "A" 'A'
test2.py <type 'str'>
A <type 'str'>
"A" <type 'str'>
1.234 <type 'str'>
SIC>
```

- **Parsing Python script arguments**: Sic does NOT interpret arguments starting by a slash as options of the command. This can be useful to simulate a Sic-like calling sequence, but implemented in Python. The Gildas-Python module `sicparse.py` can help you to parse such options in Python:

```
SIC> type test3.py
import sicparse
parser = sicparse.OptionParser()

# Declare 1 known option:
parser.add_option(
  "-m",           # Short name
  "--myopt",      # Full name
  dest="optval",  # Variable name where option value will be stored
  nargs=1,        # Number of arguments expected
  type="float",   # Kind of arguments
  default=1.23)   # Default value if option is absent

# Call again add_options to declare more options

# Parse:
try:
  (options, args) = parser.parse_args()
except:
  raise StandardError,"Invalid option"

print "Success"
print "Command arguments: ",args
print "Options: ",options

SIC> python test3.py ABCD
Success
Command arguments:  ['ABCD']
Options:  {'optval': 1.23}
SIC> python test3.py ABCD /myopt 4.56
Success
Command arguments:  ['ABCD']
Options:  {'optval': 4.56}
SIC>
```

But of course, as in Sic, arguments starting with a slash must be then double-quoted:

```
SIC> python test3.py /home/me
Usage: test3.py [options]

test3.py: error: no such option: /home/me
Traceback (most recent call last):
  File "./test3.py", line 19, in <module>
    raise StandardError,"Invalid option"
StandardError: Invalid option
SIC> python test3.py "/home/me"
Success
Command arguments:  ['"/home/me"']
Options:  {'optval': 1.23}
```

- **Raising errors in Python scripts**: There are typically 3 ways to raise errors in Python scripts:

  1. let Python raise its own errors, e.g.

     ```
     SIC> type test4.py
     print 1/0
     SIC> python test4.py
     Traceback (most recent call last):
       File "./test4.py", line 1, in <module>
         print 1/0
     ZeroDivisionError: integer division or modulo by zero
     SIC>
     ```

  2. `raise` your own Python error, e.g.

     ```
     SIC> type test5.py
     arg = 0
     if (arg==0):
       raise StandardError, "Argument must not be null"
     else:
       print 1/arg
     SIC> python test5.py
     Traceback (most recent call last):
       File "./test5.py", line 3, in <module>
         raise StandardError, "Argument must not be null"
     StandardError: Argument must not be null
     SIC>
     ```

  3. turn ON the error status of the `SIC\PYTHON` command, such as it behaves correctly when hand is given back to Sic. For example:

     ```
     SIC> type test6.py
     def main():
       try:
         print 1/0
       except:
         print "E-SIC, Some error when printing 1/0"
         pysic.sicerror()  # Set the SIC\PYTHON command error
     ```

```
    return            # Leave the function and the script

if __name__ == "__main__":
    main()
```

```
SIC> py test6.py
E-SIC, Some error when printing 1/0
SIC>
```

Note that these 3 methods raise an error in Sic, i.e. the flag `SIC%ERROR` is true on return, and that procedures will stop in such a case (if `ON ERROR` is `PAUSE`).
The last syntax gives the best integration in Sic, because in particular it hides the Python error traceback. However, this requires 1) to catch the error with a `try/except` clause, 2) to tell Sic an occured thanks to the method `sicerror()`, and 3) to leave the script right after. This last point can only be achieved by putting the whole code in a (top) function, from which you can `return` when required. `exit` is not satisfying because it performs too much cleaning, nor `raise` since a traceback would be back.

- **Sending Sic-formatted messages to Sic**: the method `message()` can be used to send messages to Sic. They will be formatted and filtered by the standard Sic messaging mechanism, e.g.:

```
>>> pysic.message(pysic.seve.i,"FOO","Hello world!")
I-FOO,  Hello world!
```

The `seve` instance contains the usual elements (`f, e, w, r, i, d, t, c`) which have to be used to indicate the message severity.

- **Executing a single-line Python command**: if any other character string follows the `PYTHON` command, it will be sent to the Python interpreter as a single line command. Multiline feature can not be used in this case:

```
SIC> PYTHON print "Hello world!"
Hello world!
SIC>
```

The command line sent after the `PYTHON` command is executed by the Python interpreter. It is also printed after a `>>>` prompt if the `SIC%VERIFY` flag is `ON`:

```
SIC> SIC VERIFY ON
I-SIC,  VERIFY       is ON
SIC> PYTHON print "Hello world!"
>>> print "Hello world!"
Hello world!
SIC>
```

Take care that what you type at SIC prompt after `PYTHON` command is now case sensitive! There will be no reformatting, except for single-quoted strings:

```
SIC> PYTHON print 'Hello world?'
>>> print 'Helloworld?'
Helloworld?
SIC>
```

Please use double-quotes instead for commands sent by SIC to Python.

2.2.2 Calling Python functions

The Python community has developped a large set of additional modules for many purposes. Some functions that SIC does not provide can be imported from an external Python module. User can also define its own functions written with Python.

The numerical functions can be called from SIC formulas. When SIC does not find a function in its own builtin or user-defined ones, it will have a look in the Python main namespace area (if Python interpreter is launched!). If one object name matches and is callable, it will call this Python function with the arguments you provided:

```
SIC> PYTHON from scipy.special import jn
>>> from scipy.special import jn
SIC> PYTHON print jn.__doc__
>>> print jn.__doc__
y = jn(n,x) returns the Bessel function of integer order n at x.
SIC> DEF REAL A
SIC> A = JN(0,1)
SIC> EXA A
A               =   0.7651977               ! Real    GLOBAL
SIC>
```

`jn`[5] is the Bessel function, called here at order 0 with `x = 1`. Always take care to import (or create) your Python functions in the main namespace. Note that function names are fully lowercased before trying to match them with Python names.

[5] `scipy` and its subpackages provide a large set of mathematical functions for scientific computing. See project webpage for more informations: `http://www.scipy.org`

2.3 Using SIC from Python

2.3.1 PyGildas modules

Most of the Gildas packages can be directly imported as Python modules:

- `import pysic`
- `import pygreg`
- `import pyastro`
- `import pyclass`[6]
- `import pyclic`
- `import pymapping`

During the import, 3 steps are executed:

- SIC (resp. GreG) is initialized as a background process,
- the `Sic` object is instantiated into Python `__main__` (see subsection 2.3.4),
- all SIC variables are imported into Python `__main__` (see subsection 2.4).

From this point, you can use the (shared) variables, jump to SIC prompt, and use any of the methods the module defines (see hereafter).

2.3.2 Special commands

A set of special functions has been written for user convenience. They are provided as methods of the modules `pysic, pygreg, ...` if SIC is launched from Python, and also in every case as methods of an instance called `Sic` in the Python `__main__` (see subsection 2.3.4):

- `comm(string)`: takes a string as argument and sends it to the SIC interpreter, which will execute it. If SIC fails to interpret the command, or if the command itself returns an error, `comm` will raise an error in Python[7]. As usual in Python, it can be catched with the `try` and `except` couple of directives.
- `setgdict(dictionary)`: define input dictionary as the area where Gildas variables will be imported, and automatically import these into the input dictionary. If a dictionary was already defined, it is first cleaned before filling the new one given in input.
  A call to this function is required if user wants to access the imported variables. This avoids polluting name areas because of uncontroled default. In interactive mode, the most common usage is to use the current name area, e.g. `setgdict(globals()); print pi`. In script mode, prefer using a custom dictionary, e.g. `d = dict(); setgdict(d); print d['pi']`, or even better the `gdict` container (see subsection 2.3.3).

[6] the current Class was called Class90 before Feb '08

[7] Please note that the `@` command in SIC will not raise any error even if a command fails in the procedure executed. User is expected to correct it on-the-fly, or to tune the error handling with the `ON ERROR` command in SIC.

- `get([string[,integer[,boolean]]])`: Takes a string as argument, which must be a valid SIC name (in SIC format, thus not case sensitive). Variable can be a scalar, an array, a structure, an header, a table, or an image. The second argument may be the level of the variable to be imported (0 = global, 1 = 1-local, and so on). It defaults to the current execution level. A third argument indicates if the function should be verbose (`True`, default), or not (`False`).
  '`get()`' creates a `SicVar` or a `SicStructure` instance sharing its data with the corresponding variable in SIC. Its name is the original SIC name, but lowercased, and converted in respect with the Python name conventions (see table 1). '`$`' can not appear in Python names: it is converted to '`_`'. '`%`' (structures in SIC) has a special treatment and is converted to '`.`' (see subsection 2.4.3). All structures elements are linked to the corresponding `SicStructure` as `SicVar` (or `SicStructure`) attributes, and all header components of the images are linked to the corresponding `SicVar` as `SicVar` attributes.
  '`get()`' fails to import variable into Python when its converted name is a Python reserved keyword (such as '`def`', '`del`', '`import`', '`as`', ...), or a built-in function name (such as '`type`', '`range`', '`min`', '`max`', ...).
  With no argument, '`get()`' imports all SIC variables by iteratively calling '`get()`' itself with their names as argument.

Table 1: SIC to Python names conversion

SIC	Python
`"ABC"`	`"abc"`
`"ABC$DEF"`	`"abc_def"`
`"ABC%DEF"`	`"abc.def"`

- `exa([var1[,var2[,...]]])`: with one or more `SicVar` or `SicStructure` instances as argument, displays one line per instance with the corresponding SIC variable name, details on its type, dimensions,... With no argument, displays this line for the full list of the SIC variables. Example:

```
>>> exa(pi)
PI              is a        REAL*8, 0D            (GLOBAL,RO) -> pi
```

- `define(string[,boolean])`: takes one string as argument and defines one or more variables in both SIC and Python. First keyword must be the type (one of `real`, `double`, `integer`, `character` or `structure`, or a non-ambiguous truncated form of them), followed by one or more valid variable-creator SIC name (e.g. '`A`', '`B*8`', '`C[2]`', '`D%E`' or even '`F%G*8[2,3]`'). All variables created this way are read-and-write. By default they are local to the current execution level, but you can provide a second optional argument set to `True` to make the variables global. For images, give the command line you would give to the SIC interpreter, but without the `define` keyword. `exa()` is finally called on the newly created instance(s). Examples:

```
>>> define('real a')
 A               is a        REAL*4, 0D            (GLOBAL,RW) -> a
>>> define('double b c[2] d[2,3]')
 B               is a        REAL*8, 0D            (GLOBAL,RW) -> b
 C               is a        REAL*8, 1D (2x0x0x0)  (GLOBAL,RW) -> c
 D               is a        REAL*8, 2D (2x3x0x0)  (GLOBAL,RW) -> d
>>> define('structure e')
 E               is a   <structure>, 0D            (GLOBAL,RW) -> e
>>> define('character e%f e%g*6[2,3]')
 E%F             is a   CHARACTER*1, 0D            (GLOBAL,RW) -> e.f
 E%G             is a   CHARACTER*6, 2D (2x3x0x0)  (GLOBAL,RW) -> e.g
>>> define('image h centaurus.gdf read')
 H               is a (image)REAL*4, 2D (512x512x1x1) (GLOBAL,RO) -> h
```

- `delete(var1[,var2[,...]])`: takes one or more `SicVar` or `SicStructure` instances as arguments. Tries to delete the corresponding variable in SIC. On success, also deletes the input instance.
- `getlogical(string)`: takes one Python string as argument. The method translates a SIC logical and returns it as a Python string. Input name is case insensitive. An empty string is returned if no such SIC logical exists (no error is raised).

2.3.3 The `gdict` container

Modules have a special attribute named `gdict` (actually, a class instance) from which all Gildas variables can be easily retrieved and defined:

```
>>> import pysic
>>> pysic.get()
Importing all SIC variables into Python...
... done.
>>> g = pysic.gdict
>>> g
Container for imported Gildas variables. Variables are accessed as
   attributes of this object.
>>> dir(g)         # List of all imported variables
['__builtins__', 'debug_recursive', 'gildas_node', 'linedb', 'no', 'pi',
'run_window', 'sic', 'type_dble', 'type_inte', 'type_logi', 'type_long',
'type_real', 'yes']
>>> g.pi           # Get Gildas variable PI
3.14159265359
>>> g.a = (1,2,3)  # Set a new Gildas variable A
A               is a      INTEGER*4, 1D (3x0x0x0)   (GLOBAL,RW) -> a
>>> pysic.comm('exa a')
A               is an integer Array      of dimensions      3
            1            2            3
```

Whatever the name area has been defined by user with `setgdict`, any `gdict` instance will allways find the Gildas dictionary[8]. The variable handling can be found easier than the dictionary equivalent `d['pi']`.

[8]Technically, `gdict` instances have no attributes. They only have the methods `__getattr__` and `__setattr__` which dynamically retrieve or define the `SicVar` instances.

2.3.4 The `Sic` object in Python `__main__`

This object is obsolescent. Whatever is the way you launched SIC and Python through the Pygildas features, an object named `Sic` will be created at initialization time in the Python `__main__`. It is used for these purposes:

- It stores (as attributes) aliases of commands documented at subsection 2.3.2. If user called Python from SIC, he can only access them from the `Sic` object. If user called Gildas from Python (e.g. by importing `pysic` or `pygreg`), he can either call these commands with the `pysic`/`pygreg` or `Sic` prefix. Calling `Sic()` switches to the SIC prompt.
- It stores as an attribute the array of 10 instances named `localspaces` and used to save local variables (see subsection 2.5).
- It stores a logical flag named `warnings` which activate/deactivate warnings (printed when importing a variable failed for example). It is set to `True` by default, but user can switch it to `False`.

2.4 Importing SIC objects

Importing SIC objects (more precisely *making them visible from Python*) is quite easy and all the trick is based on the `NumPy` package and its `ndarray` type object. All SIC variables are imported through instances of two special classes: `SicVar` and `SicStructure` which implements `numpy.ndarray` features.

2.4.1 SIC arrays

SIC arrays are imported as `SicVar` instances. Let us define[9] a 1D array `A` in SIC:

```
>>> Sic.setgdict(globals())
Importing all SIC variables into Python...
... done.
>>> Sic.comm('DEFINE INTEGER A[3]')
```

Defining a variable in SIC automatically imports it in Python `__main__`. Here `A` is imported into a `SicVar` instance with name '`a`':

```
>>> a
[0 0 0]
>>> print a
<SicVar instance>
array([0, 0, 0])
>>> type(a)
<type 'instance'>
>>> print a.__doc__
A SicVar instance.
```

All of the `numpy.ndarray` attributes, methods or functions should apply to the `SicVar` instances, because almost all these instances behaviors are redirected to their `numpy.ndarray` component. Their elements can be accessed with standard `NumPy` indexing syntax:

```
>>> a[0]
0
>>> a[1:] # Elements 1 and subsequents
[0 0]
>>> a += 1 # Adds 1 to all elements
>>> a
[1 1 1]

>>> a[:] = 0 # Sets all elements to 0
>>> a
[0 0 0]
>>> len(a) # Length
3
>>> a.shape
(3,)
```

Remember that data pointed to by `SicVar` instances is not a copy: if you modify it in Python it will be modified in SIC:

[9] the `comm()` method of the `Sic` object or the `pysic, pygreg, ...` modules (depending of the calling method) sends its string argument as a command line to SIC. See subsection 2.3.2 for all the usefull methods.

```
>>> a[0] = 1
>>> a
[1 0 0]
>>> Sic.comm('EXA A')
A               is an integer Array      of dimensions      3
            1            0            0
```

Remember also that derived arrays share their data with the initial array:

```
>>> b = a[0:2] # First two elements (upper limit is excluded)
>>> print b.__doc__
A SicVar instance.
>>> b
[1 0]
>>> b += 1 # Adds 1 to all elements
>>> b
[2 1]
>>> a
[2 1 0]
```

Note that `b`, as a derived array, **is** itself a `SicVar` instance, but is only visible in Python. And remember the different memory arrangement between SIC (Fortran) and C (Python) for multidimensionnal arrays (see subsection 2.1.1):

```
>>> Sic.comm('DEFINE INTEGER C[2,3,4]')
>>> c.shape
(4, 3, 2)
>>> c[0,0,0] = 1  # First element
>>> c[3,2,1] = 2  # Last element
>>> Sic.comm('EXAMINE C')
C     is an integer Array    of dimensions      2     3     4
     1          0          0          0          0          0
     0          0          0          0          0          0
     0          0          0          0          0          0
     0          0          0          0          0          2
```

2.4.2 SIC scalars

Particular `numpy.ndarray`'s which have no (0) dimension have a special treatment: they still have `numpy.ndarray` type, but their representation and handling is similar to standard Python scalars. SIC scalars are imported in Python as `numpy.ndarray` scalars through `SicVar` instances:

```
>>> Sic.comm('DEF REAL D')
>>> d
0.0
>>> print d.__doc__
A SicVar instance.
```

You have to take care that you cannot modify scalars (and, actually, any `SicVar` or `SicStructure` instance) with commands such as `d = 1`, because this unbounds `d` to its corresponding `SicVar` instance (which is lost and deleted if no other variable uses it), and bounds it to integer 1. If it happens, you can `get('D')` again: `SicVar` instance is recreated and all the previous changes will not be lost because `D` have not been deleted in SIC.

```
>>> Sic.comm('LET D 1.')
>>> d
1.0
>>> d = 2.            # ooops... 'd' instance is lost!
>>> type(d)
<type 'float'>        # 'd' is bound to a standard float,
>>> Sic.comm('EXA D') # but 'D' remains unchanged in SIC.
D          =   1.000000              ! Real    GLOBAL
>>> Sic.get('D')      # Reincarnates d.
>>> d
1.0
```

Thus, to modify scalars from Python, you have to access its data through its unique element:

```
>>> d[0] = 2.
>>> d
2.0
>>> print d.__doc__
A SicVar instance.
```

2.4.3 SIC structures

SIC structures are imported in Python into instances of the special class named `SicStructure`:

```
>>> Sic.comm('DEFINE STRUCTURE E')
>>> e
<pgutils.SicStructure instance at 0xb7f00bcc>
```

SIC structure elements are stored in attributes of `SicStructure` instances, and these attributes are themselves `SicVar` instances. Call the `print` statement to print all (and only) `SicVar` (or `SicStructure`) attributes, and some details on them.

```
>>> Sic.comm('DEFINE INTEGER E%A')
>>> Sic.comm('DEFINE REAL E%B[2]')
>>> Sic.comm('DEFINE DOUBLE E%C[2,3]')
>>> e
<sicutils.SicStructure instance at 0xb7f009cc>
>>> print e # Detailed printing
<SicStructure instance>
a = 0                                  INTEGER*4 0D
c = [[ 0.  0.] [ 0.  0.] [ 0.  0.]]      REAL*8 2D (2x3x0x0)
b = [ 0.  0.]                            REAL*4 1D (2x0x0x0)
>>> print e.c # 'c' attribute is a SicVar instance
<SicVar instance>
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

Nested structures can be recursively imported:

```
>>> Sic.comm('DEFINE STRUCTURE E%D')
>>> Sic.comm('DEFINE INTEGER E%D%A')
>>> print e
<SicStructure instance>
a = 0                                   INTEGER*4 0D
c = [[ 0.  0.] [ 0.  0.] [ 0.  0.]]       REAL*8 2D (2x3x0x0)
b = [ 0.  0.]                             REAL*4 1D (2x0x0x0)
d = <SicStructure>
>>> print e.d
<SicStructure instance>
a = 0                                   INTEGER*4 0D
```

`SicStructure` instances can be added new attributes and these attributes are automatically instantiated in the SIC structure[10]:

```
>>> e.e = 1
E%E             is a      INTEGER*4, 0D             (GLOBAL,RW) -> e.e
>>> Sic.comm('EXA E%E')
E%E             =            1               ! Integer GLOBAL
>>>
>>> e.f = (1,2,3)
E%F             is a      INTEGER*4, 1D (3x0x0x0)   (GLOBAL,RW) -> e.f
>>> Sic.comm('EXA E%F')
E%F             is an integer Array      of dimensions      3
            1            2            3
>>>
>>> import numpy
>>> e.g = numpy.zeros((3,3))  # Create a 3x3 real numpy array
E%G             is a         REAL*8, 2D (3x3x0x0)   (GLOBAL,RW) -> e.g
>>> Sic.comm('EXA E%G')
E%G             is a double precision Array      of dimensions      3     3
   0.00000000000000000       0.00000000000000000       0.00000000000000000
   0.00000000000000000       0.00000000000000000       0.00000000000000000
   0.00000000000000000       0.00000000000000000       0.00000000000000000
```

2.4.4 SIC images

Standalone headers or tables are imported like any other structure or array respectively. SIC images, which combines a header with a table, are also imported in `SicVar` instances. Thus, image elements are accessed through standard indexing to the instance itself, and header components through attributes of this instance:

```
>>> Sic.comm('DEFINE IMAGE F centaurus2.gdf WRITE')
PygildasWarning: 'max' is a ndarray attribute, could not overwrite it.
PygildasWarning: 'min' is a ndarray attribute, could not overwrite it.
PygildasWarning: 'ndim' is a ndarray attribute, could not overwrite it.
```

You have to take care that `numpy.ndarray`'s provide a large set of attributes and methods, and some of them have name identicals to headers components: they are not imported and only a warning is printed.

[10] if possible. The destination value must be caste-able by the `numpy.array` method into a `numpy.ndarray`

```
>>> f
[[ 0.01676085  0.0141144   0.0194073   0.0141144   0.0105858 ]
 [ 0.02911095  0.01852515  0.0141144   0.01323225  0.0123501 ]
 [ 0.09968295  0.0264645   0.          0.0141144   0.01499655]
 [ 0.035286    0.0141144   0.01676085  0.01499655  0.01323225]
 [ 0.0229359   0.017643    0.0123501   0.0158787   0.0158787 ]]
>>> f[0,0]
0.0167608503252
>>> print f
<SicVar instance>
array([[ 0.01676085,  0.0141144 ,  0.0194073 ,  0.0141144 ,  0.0105858 ],
       [ 0.02911095,  0.01852515,  0.0141144 ,  0.01323225,  0.0123501 ],
       [ 0.09968295,  0.0264645 ,  0.        ,  0.0141144 ,  0.01499655],
       [ 0.035286  ,  0.0141144 ,  0.01676085,  0.01499655,  0.01323225],
       [ 0.0229359 ,  0.017643  ,  0.0123501 ,  0.0158787 ,  0.0158787 ]], dtype=float32)
rms       = 0.0                                        REAL*4 0D
major     = 0.0                                        REAL*4 0D
blan      = 8                                       INTEGER*4 0D
ptype     = 3                                       INTEGER*4 0D
system    = 'EQUATORIAL  '                       CHARACTER*12 0D
... (and so on for all header elements)
>>> print f.rms # 'rms' is an attribute of 'f', and a SicVar itself.
<SicVar instance>
array(0.0, dtype=float32)
```

2.4.5 Handling character strings

Character strings are imported in a `SicVar` instance as any other SIC variable:

```
>>> Sic.comm('DEFINE CHARACTER G*8')
>>> Sic.comm('LET G "ABCDEFGH"')
>>> g
'ABCDEFGH'
>>> print g
<SicVar instance>
array('ABCDEFGH',
      dtype='|S8')
>>> len(g)
8
```

Remember that `numpy.ndarray`'s provide the `.tostring()` method which returns a Python string resulting of the concatenation of elements of a character array ('`|S*`' dtypes). This can be useful to handle SIC strings in Python. Nevertheless string concatenation and multiplication is already implemented in `SicVar` instances:

```
>>> h = 'xyz' + g + 'ijk'
>>> h
'xyzABCDEFGHijk'
>>> type(h)
<type 'str'> # A standard Python string
>>> h = 2*g
>>> h
'ABCDEFGHABCDEFGH'
```

These `SicVar` strings can also be easily modified[11]:

```
>>> g[0] = "qwerty"
>>> g
'qwerty  ' # note 'g' has been automatically blank filled
>>> Sic.comm('EXA G')
G     = qwerty             ! Character* 8 GLOBAL
```

[11] whereas real Python strings are *immutable* objects: you can not modify them inplace and have to make copies for such handlings.

2.5 Object status in SIC:

Read-only or read-and-write? Read or write status is preserved when variables are imported in Python, and trying to modify their values will raise an error:

```
>>> pi
3.14159265359
>>> print pi.__doc__ # pi was imported from SIC
A SicVar instance.
>>> pi[0] = 0
Traceback (most recent call last):
  File "<STDIN>", line 1, in <module>
  File "/home/bardeau/gildas/gildas-src-dev/pc-fedora6-ifort/kernel/
python/pgutils.py", line 204, in __setitem__
    raise PygildasValueError, "Variable is read-only in SIC."
pgutils.PygildasValueError: Variable is read-only in SIC.
```

The protection of `SicVar` intances against deletion or re-definition (ie using `pi = 1` instead of `pi[0] = 1`) is not implemented. See considerations in subsection 2.4.2.

Global or local variable (in the SIC sense)? `SicVar` and `SicStructure` instances have a `__siclevel__` attribute which is an integer set to the level of the corresponding variable in SIC (0 for global, 1 for first local level, and so on).

When SIC defines any variable, it automatically imports it in the Python `__main__` name space[12]. When this variable is SIC-local (when executing procedures for example), importing it in Python may overwrite a lower level instance which has the same name. To prevent this, the `Sic` object in Python `__main__` provides an array of 10 instances (one per level) which are used to temporarily store object which would have been erased. This array is named `localspaces` and objects are saved as attributes of its 10 elements. Consider these SIC procedures:

```
SIC> type localtest1.sic
define integer a
let a 123
pause
@localtest2.sic
SIC> type localtest2.sic
define real a[3]
pause
```

They will define local variables named '`A`' at different local levels. Let's define a global '`A`' variable and execute these procedures:

[12]This is subject to changes as long as the Gildas-Python binding is in a beta development status.

```
SIC> define char a*8
SIC> let a "qwertyui"
SIC> @localtest1.sic
SIC_2> define integer a
SIC_2> let a 123
SIC_2> pause
SIC_3> python
Entering interactive session. Type 'Sic()' to go back to SIC.
>>> a
123
>>> a.__siclevel__
1
>>> Sic.localspaces[0].a
'qwertyui'
>>> Sic.localspaces[0].a.__siclevel__
0
```

'a' object currently in Python __main__ is the level-1 SIC object. The SIC-global 'A' variable has been saved as an attribute of the `Sic.localspaces[0]` element.
Let's go deeper:

```
SIC_3> continue
SIC_2> @localtest2.sic
SIC_3> define real a[3]
SIC_3> pause
SIC_4> python
Entering interactive session. Type 'Sic()' to go back to SIC.
>>> a
[ 0.  0.  0.]
>>> a.__siclevel__
2
>>> vars(Sic.localspaces[0])
{'a': 'qwertyui'}
>>> vars(Sic.localspaces[1])
{'a': 123}
```

Now we find the level-2 object in the Python __main__, and the 2 lower-level variables with the same name are saved in the `Sic.localspaces` array. At any stage, user can access its current-level variables in the Python __main__, but also any lower level ones through the `Sic.localspaces` array.
Let's end procedures:

```
SIC_4> continue
SIC> python
Entering interactive session. Type 'Sic()' to go back to SIC.
>>> a
'qwertyui'
>>> vars(Sic.localspaces[0])
{}
>>> vars(Sic.localspaces[1])
{}
```

In an automatic cascading mechanism, the lower-level variables are moved back to the Python __main__ when the current level one is deleted. The `Sic.localspaces` array is cleaned at the same time so their is no double reference for any object. Thus at the end of the procedures, the

SIC-global variables are back in the Python `__main__`, and the `Sic.localspaces` array has no more attributes.

Remember that an object is saved in the `Sic.localspaces` array if and only if an upper level variable is defined with the same name. Thus, at any time, the Python `__main__` may contain objects from different levels. User does not have to care about the saving and unsaving mechanism: all is automatic and goes back as it was after procedure execution. He only has to know that he can access any lower level variable in the `Sic.localspaces` array.

3 Programmer documentation

3.1 Installing PyGILDAS

3.1.1 Prerequisites

Before compiling, Gildas administrative scripts[13] try to detect your Python installation. It retrieves the version number of the python executable visible from your shell (`python -V`) and searches for associated `libpython`*version*`.*` and `Python.h`. The latter header file is not always shipped with default system installations, but it is required to compile the Python binding for Gildas. If not found, you should consider to install the Python development package for your installation.

WARNING: if your Python is the one installed on your system, try only to install the development package for it. NEVER try to upgrade it! Python is deeply used in modern Linux systems and you might break it definitely.

Additionally, Gildas-Python binding rely on the extended array-support module `NumPy`. It must be importable in Python in order to activate the compilation of the binding.

Versions: PyGILDAS is known to work with Python from versions 2.6.* up to 3.4.*, and with `NumPy` version 1.4.* to 1.7.*.

3.1.2 How to build your own Python version

You can easily build your own Python binaries and libraries by following the steps below:

1. Retrieve the sources from the official website: `http://www.python.org/download/`
2. Unpack, compile and install Python (any version, here with Python 2.7):

```
cd <compilation-directory>
tar jvzf Python-2.7.tar.bz2
cd Python-2.7
./configure --enable-shared [--prefix=/your/custom/installation/path]
make
make test
make install
```

   The `--enable-shared` option ensures to build both static and dynamic Python libraries. This option is *mandatory* for a correct behaviour of the Gildas-Python binding. The `--prefix` option allows you to install Python in a custom location (instead of `/usr/local`). This is useful in particular if you do not have administrative priviledges. Finally you should refer to section 3.1.3 if you want to enable the command line history in the Gildas-Python binding.

3. Make your new Python available. Fill the binary and library location in the corresponding environment variables:

```
export PATH=/your/custom/installation/path/bin:$PATH
export LD_LIBRARY_PATH=/your/custom/installation/path/lib:$LD_LIBRARY_PATH
```

[13] i.e. when executing `source admin/gildas-env.sh`

You can make your custom Python the permanent default for yourself (i.e. overriding the system Python), or you can make it a transient default to be used only for Gildas compilation. This depends on your needs, see post-compilation instructions in section 3.1.6.

4. Check your installation:

```
which python
python -V
python -c "import sqlite3"
```

`sqlite3` Python module is optional, needed only for the extension named Weeds for Class. If `import sqlite3` fails, install (or ask your system administrator) sqlite3 headers on your system (system package providing `sqlite3.h` i.e. sqlite3-devel), and restart from step 2.

Then you are ready to install `NumPy` (see below).

3.1.3 Python module `readline` for command history

The Python interpreter launched from SIC provides the command line history if and only if the Python module `readline` is imported. This is done automatically when the interpreter is launched. If this module can not be imported, an `ImportError` will be raised and you will not have the command history at the Python prompt.

The `readline` module is a Python builtin module. In most case it has been compiled and installed during your Python installation. Nevertheless, in case of a new compilation and installation of Python, it may not be available. You can check this by trying to import it in a standard Python session:

```
>>> import readline
```

If an `ImportError` raises, here is what you (or your system administrator) have to do. The basic idea is that the `readline` Python module is compiled from a `readline.c` in Python source. Your system must provide the libraries `libreadline.a` and `libtermcap.a` to compile it successfully.

1. Check the `config.log` file (the output of the `./configure` command) in the directory used to compile Python. Search for `readline` occurences and look for errors that did not allow to find `readline` on your system.

2. It may appear that some symbols where undefined. They are provided by the two system libraries `libreadline.a` and `libtermcap.a`. Check if you have them, install them if not. Launch `./configure` command again.

3. If the `readline` Python module still not compiles, try fo force `./configure` to link against one or two of the above system libraries:

```
./configure --with-libs='-ltermcap'
```

The module should now compile. `make` and `make install` your Python with its new module.

3.1.4 Install NumPy module for Python

NumPy is mandatory for the Gildas-Python binding. Building and installing it requires some specific options and setting a correct environment. You can follow these steps:

1. Download NumPy from: http://www.scipy.org/Download

2. Unpack and compile NumPy. You can build NumPy either for the Python installed on your system or for a custom one: the build will be done for the Python which is seen from your shell (which python).

```
cd <compilation-directory>
tar xvzf numpy-1.3.0.tar.gz
cd numpy-1.3.0
python setup.py build --fcompiler=<compiler>
```

   Usually gnu95 is the correct argument for the --fcompiler option. Else (or if your want to know more), you should consider the following point:

```
  The two most popular open source fortran compilers are g77 and gfortran.
Unfortunately, they are not ABI compatible, which means that concretely
you should avoid mixing libraries built with one with another. In
particular, if your blas/lapack/atlas is built with g77, you *must* use
g77 when building numpy and scipy; on the contrary, if your atlas is
built with gfortran, you *must* build numpy/scipy with gfortran.

* Choosing the fortran compiler
  - To build with g77:
    python setup.py build --fcompiler=gnu
  - To build with gfortran:
    python setup.py build --fcompiler=gnu95

* How to check the ABI of blas/lapack/atlas?
  One relatively simple and reliable way to check for the compiler used
to build a library is to use ldd on the library. If libg2c.so is a
dependency, this means that g77 has been used. If libgfortran.so is a
dependency, gfortran has been used. If both are dependencies, this means
both have been used, which is almost always a very bad idea.
```

   In other words, for standard users, NumPy has a linear algebra module (linalg) which is linked with the system's blas/lapack/atlas libraries. This implies that the Fortran sources of NumPy must be compiled with the same compiler which was used for the libraries. Check e.g. with ldd /usr/lib(64)/lapack.so which compiler you will have to use.

3. Install NumPy:

```
python setup.py install [--prefix=/some/custom/installation/prefix]
```

   The installation prefix (--prefix) is required *only* if you want to install NumPy in a location different from your Python installation directory (e.g. you are using the default Python

executable provided by your system but you do not have root permissions). In this case (only), you will have to fill the `$PYTHONPATH` environment variable to indicate where Python can find `NumPy`.
In the other cases, Python will install the module into its own installation directory, which is fine in particular when you are using a custom installation of Python. In this case you will also not have to augment the `$PYTHONPATH` since the module is installed in a standard location known by Python.

4. Check the installation. Launch Python and type:

```
import numpy
numpy
numpy.__version__
```

   You should recognize the installation path and the correct `NumPy` version.

3.1.5 Install `scipy` module for Python (optional)

The module named `scipy` (*Scientific Python*) is not necessary for the Gildas-Python binding, but it provides useful functionalities you may want. The build and installation process is exactly the same as for `NumPy`:

1. Download `scipy` from: `http://www.scipy.org/Download`
2. Unpack and compile `scipy`:

```
cd <compilation-directory>
tar xvzf scipy-0.7.1.tar.gz
cd scipy-0.7.1
python setup.py build --fcompiler=<compiler>
```

3. Install:

```
python setup.py install [--prefix=/some/custom/installation/prefix]
```

   If you provide a custom installation path (`--prefix`), use the same as for `NumPy`: you will not have to augment again the `$PYTHONPATH` for this second module.

4. Check the installation:

```
import scipy
scipy
scipy.__version__
from scipy.special import jn
```

   (the last line tries to import the Bessel functions named `jn`).

3.1.6 Compiling Gildas with your custom Python (if any)

If you compiled and installed a custom version of Python and the needed modules, then you can run the usual Gildas administrative scripts:

```
cd <installation-directory>
tar xvzf gildas-src-jul14.tgz
cd gildas-src-jul14
source admin/gildas-env.sh [-c <compiler>]
# read carefully the messages (in particular those for Python)
make
make install
# read the last instructions
```

If you followed correctly the installation of your custom Python and the compilation of Gildas, Gildas is now binded to your custom Python libraries. You have then 2 possibilities:

- if you use Python inside Gildas (i.e. SIC\PYTHON commands), you can safely revert the `python` command to its original state. Gildas will use your custom Python libraries, including its specific syntax and rules.
- if you use the Gildas-Python modules (e.g. pygreg, pyclass) in Python, then you must import them from your custom Python executable. Either you keep it permanently in your $PATH, either you call it explictly by a custom name (e.g. `python34`) before importing the modules. Here also this means you have to use the correct syntax and rules.

3.2 How SIC variables are imported

3.2.1 Comments on the strategy

Concerning variables handling from both SIC and Python, the first priority was to manage the same data in memory from the two processes, e.g. a modification of a variable in one process should be instantaneously visible in the other with no particular resynchronization.

Another consideration is that Python does not natively handle multi-dimensionnal arrays. This feature can be easily added, but we need the `NumPy` package.

It appeared quite naturally that the solution was to use the `PyArray_SimpleNewFromData()` function of the `NumPy` C-API: it can construct a `numpy.ndarray` from the memory address of an array, its dimensions and its datatype.

With this feature the strategy can be summarized as follow:

- `NumPy` freezes its `array` definition (attributes, size in memory and so on) at compilation time. We can not add to it more interesting features, especially new attributes or write protection for read-only variables.
- an overlay had to be created with one purpose: defining a set of features like
  - the ability to define attributes to variables,
  - the protection of read-only variables,
  - a special treatment for character string variables (concatenation),

– ...

and all this must keep unmodified all the `numpy.ndarray` features (indexing, array handling, special functions and attributes, and so on)

- this overlay was created as a Python class named `SicVar`. It has three attributes: a string named `__sicname__`, which is the SIC name of the variable, a `numpy.ndarray` named `__sicdata__`, which shares its data with the SIC variable, and its SIC-level (global or local) stored as an integer named `__siclevel__`. All `numpy.ndarray`-like requests on a `SicVar` are redirected to the `__sicdata__` attribute: with this, `SicVar` instances behaves like any other `numpy.ndarray`.

3.2.2 The `SicVar` class

`SicVar` instances should be automatically initialized with the `get()` function. When importing a variable, `get()` calls the instance creator `SicVar()` with three arguments: SIC variable name (as a Python string), its level (as an integer), and memory address of its descriptor (Python integer). It is the own responsibility of the `get()` function to assign this instance to the correct Python variable name.
During the initialization, `SicVar()` defines three attributes to the instance: `__sicname__`, `__siclevel__`, and `__sicdata__`. By convention the '__' delimiters denote objects that are hidden to the user, but still accessible. `__sicname__` and `__siclevel__` are a Python string and an integer, and are used to keep a trace of what the original SIC variable is. `__sicdata__` is a `numpy.ndarray` initialized with the informations provided by the descriptor (data address, dimensions, type,...).

```
>>> Sic.define('real a')
 A               is a        REAL*4, 0D            (RW) -> a
>>> vars(a)    # vars() displays attributes names and values
{'__sicdata__': array(0.0, dtype=float32), '__sicname__': 'A', '__siclevel__': 0}
>>> type(a.__sicdata__)
<type 'numpy.ndarray'> # A NumPy array
>>> type(a.__sicname__)
<type 'str'>   # A Python string
>>> type(a.__siclevel__)
<type 'int'>   # A Python integer
```

`SicVar` instances are used to import SIC standard variables (scalars or arrays). Due to its class type, it can also accept new attributes, and thus be used to import images (for example in an instance named `ima`). In this case, the image (the array component) is imported through the `__sicdata__` attribute (`ima.__sicdata__`), and can be directly accessed from `ima` (`ima[0,0]`). Header variables are imported into other attributes (`ima.naxis1`, `ima.naxis2`,...), provided they do not attempt to overwrite a `numpy.ndarray` attribute or method. As any other imported SIC variables, these header attributes are themselves `SicVar` instances. See subsection 2.4.4 for details and example on importing SIC images.

All attributes of a `SicVar` instance are write-protected against deletion or overwriting. This is a first importance for `__sicdata__`, because deleting it would be dramatic. Remember that the command `a.__sicdata__ = 0` do not modify its content but bounds it to the 0 integer, loosing data and all information about the corresponding SIC variable. But end-user should never have to deal with the `__sicdata__` attribute, and have to work on the instance itself instead (for

example `a[0] = 0`).
These warnings also apply to images attributes: do not try `ima.naxis1 = 0` but `ima.naxis1[0] = 0` instead.

`SicVar` instances come with a set of methods and attributes, for one part inherited from the `numpy.ndarray` type, and for the other part (re)defined at instance initialization. For example the addition (`__add__` and `__radd__`) is redefined to add numbers or concatenate character strings.

```
>>> dir(a) # All attributes and methods.
['__add__', '__copy__', '__deepcopy__', '__delattr__', '__div__',
'__doc__', '__eq__', '__ge__', '__getattr__', '__getitem__',
'__gt__', '__iadd__', '__idiv__', '__imul__', '__init__', '__int__',
'__isub__', '__le__', '__len__', '__lt__', '__module__', '__mul__',
'__ne__', '__nonzero__', '__radd__', '__rdiv__', '__repr__',
'__rmul__', '__rsub__', '__setattr__', '__setitem__', '__sicdata__',
'__sicname__', '__str__', '__sub__', 'astype', 'byteswapped',
'copy', 'iscontiguous', 'itemsize', 'resize', 'savespace',
'spacesaver', 'tolist', 'toscalar', 'tostring', 'typecode']
```

3.2.3 The `SicStructure` class

The only purpose of the `SicStructure` instances is to be objects that accept user-defined attributes (as for SIC structures actually). The definition of the `SicStructure` class is rather simple. It is instantiated by the `SicStructure()` creator with the SIC structure name and level as only arguments, which are stored in its `__sicname__` and `__siclevel__` attributes.

This empty instance in then filled by the `get()` function: it loops through all SIC variable dictionary and searches for all names which begin with the structure name. For each valid variable it `get()`'s it as a `SicVar` attribute of the `SicStructure` instance. `SicStructure`s support nested structures, as user or program can define any kind of attributes (`SicVar`, `SicStructure`, or even standard Python objects).

For user convenience, the `__str__()` method (called by `print myinstance` or `str(myinstance)`) of the `SicStructure` class is redefined to print all (and only) `SicVar` and `SicStructure` attributes, with some details on them.

3.3 Array elements types

As mentionned earlier (see subsection 2.1.2), `numpy.ndarray` elements can be of different kind, precision, and thus memory size. This is reflected through the `.dtype` attribute of any `numpy.ndarray`. By default, integer elements are created with the '`<i4`' datatype (standard integer), and floats with the '`<f8`' one (double precision float). Nevertheless, not all SIC elements follows these types. Attention have been paid to import SIC data into the correct type (see table 2).

Table 2: OBSOLETE TABLE. SEE NUMPY DOCUMENTATION. SIC to `Numeric` type conversion (for a 32-bit architecture)

SIC type	`Numeric` typecode	# of bytes
`INTEGER*4`	`'i'`	`sizeof(int) = 4`
`REAL*4`	`'f'`	`sizeof(float) = 4`
`REAL*8`	`'d'`	`sizeof(double) = 8`
`LOGICAL*4`	`'i'`	`sizeof(int) = 4`
`CHARACTER`	`'c'`	`sizeof(char) = 1`

Mixing arrays in Python formulas with element size different from the default Python behavior is not problematic. `NumPy` deals with all these types and applies coercion to the adequate type, and all is completely transparent for the user.

4 Python 3K

As of mid-october 2014, the Gildas-Python binding supports Python 3. This is implies more or less transparent changes depending on how far you are involved in this topic.

4.1 References

Before switching to Python 3K, you might want to have a closer look to what this implies. You may find the following documents helpful:

- Python 2 or Python 3? `https://wiki.python.org/moin/Python2orPython3`
- What's new in Python 3K: `http://docs.python.org/py3k/whatsnew/3.0.html`
- Porting to Python 3: `https://wiki.python.org/moin/PortingPythonToPy3k`
- Strings are now Unicode in Python 3K: `https://docs.python.org/3.0/howto/unicode.html`

4.2 Standard end-user

As a standard user, you are of course exposed to all the changes implied by Python 3. You can use the `2to3` tool shipped with Python 3 to translate your Python scripts. Note also that some Python 3 syntaxes and behaviors are backward compatible with Python 2.6 and 2.7, you should prefer them when possible.

Regarding the Gildas-Python binding itself, you should consider that:

- the support for Python 2.5 and lower has been removed (this, because Python offers some Python 3 pre-compatibility of our Gildas-Python internal code starting from Python 2.6),
- the default Python strings are now Unicode strings, while Sic characters strings (and their representation as a `SicVariable` in Python) remain ASCII strings (1 byte per character). We tried to deal with this as transparently as possible for you, so that you can still perform the usual actions on the `SicVariable` strings. Unfortunately, this implies many hidden conversions between ASCII and Unicode.

4.3 Developers using the Python binding

If you are in charge of a Python module or package integrated in Gildas, you should be aware of the following:

- as Gildas supports both Python 2 and 3, you are asked to provide your scripts with Python 2 syntax only. If needed, the scripts will be processed by the `2to3` Python tool at compilation time. Try also to use Python 3 syntax backported to Python 2 as much as possible in order to limit the differences between both syntaxes.
- if your scripts are part of a Python module, they will be implicitly Python-compiled[14] at compilation time. This pre-compilation is always done sooner or later by Python at first use of the module. Our intent is to offer this compilation for users who do not have write-access to the module installation path (e.g. shared installations of Gildas). See link for details.

[14] see `https://docs.python.org/3/library/py_compile.html`

- designing the hierarchy of your Python package and (sub)modules is less trivial than it seems. Check carefully the *relative imports* guidelines as described e.g. in `https://docs.python.org/2/faq/programming.html#what-are-the-best-practices-for-using-import-in-a-module`

4.4 Developers of Fortran GILDAS packages

As a pure Gildas developer (Fortran), the Gildas-Python binding offers to you the possibility to make your library importable from Python. This is done with the rule `IMPORT_FROM_PYTHON = yes` in the Makefile. Say, if your package is named *foo*, you should be aware that:

- the Python binary module *pyfoo.so* is not anymore a symbolic link to *libfoo.so*. Since symbolic links are not portable, the module *pyfoo.so* has been made a real binary file.
- the file *foo-pyimport.c* (which provides the Python entry points) should not be compiled anymore into the Gildas library *libfoo.so*. It is now the responsibility of Python to compile it implicitly and to put it in the binary module *pyfoo.so*.

A What's new?

- 2018-apr-24
  - Removed support for `Numeric` obsolete python package.
- 2018-apr-09
  - The `__dir__` method of the `gdict` container (see section 2.3.3) is now redefined to return the list of Gildas-Python shared variables.
- 2014-oct-13
  - Added support for Python 3 (see details in section 4). Removed support for Python 2.5 and lower. Python 2.6 and 2.7 still supported.
- 2013-may-31
  - Added argument `verbose=True|False` to the method `get()` (indicate if the method should be verbose or not when importing Sic variables).
- 2013-may-30
  - Document how to build Gildas with Python 2, when Python 3 is the default on the system. Waiting for Python 3 support in the Gildas-Python binding, currently under development.
- 2012-may-14
  - The method `.sicerror()` is added to the Gildas-Python modules (e.g. pysic, pygreg, etc). It is intended to set the error status ON in Sic when executing a Python script, in order to raise custom errors from Python.
  - The method `.message()` is added to the Gildas-Python modules, in conjuction to the object `seve`. They should be used to print messages like Sic does, including on-screen and/or to-file redirection, and application of filtering rules.
- 2012-mar-06
  - Double quotes surrounding character strings arguments in SIC are now preserved when they are passed to the sys.argv list in Python. This is a change of behavior as they were implicitely removed up to now, but could lead to problems on some cases, e.g. for Python commands using a leading "/" for options as Sic does.
- 2010-sep-07
  - SIC aliases created from the SIC side are now automatically imported to the Python side, like any other variables. SIC aliases were already supported, but not imported by default.
- 2010-jul-08
  - Python 2.7 and `NumPy` 1.4.1 are supported.
- 2010-mar-03

  - Added automatic creation of booleans (SIC logicals) when they are attached to a `SicStructure`.

- 2010-jan-11
  - When executing a Python script from Gildas, 'sys.argv' is now a list (and not a tuple anymore). This conforms the standard Python behavior.

- 2010-jan-07
  - Improved retrieving of the arguments passed to a Python script through the SIC PYTHON command. Make sure that double quoted strings with arbitrary number of blanks are not split nor modified.

- 2009-nov-18
  - Nested structures could not be correctly imported into Python in some cases. Fixed.

- 2009-oct-06
  - Added method 'getlogical' which translates a SIC logical and returns it in a Python string.

- 2009-apr-27
  - Fixed a recursive call at initialization time of SicVar instances.
  - Ensure support for Python 2.6.2

- 2008-aug-01
  - Ensure support for Python 2.5.2

- 2008-jun-02
  - Gildas packages are available with a `py*` prefix: `pysic`, `pygreg`, `pyastro`, `pyclass` and `pymapping`.
  - Enhanced support of string (scalar) variables: they are imported as a single element, i.e. characters are not splitted anymore in a subarray.
  - Enhanced support of `SicStructure`: adding an attribute to a `SicStructure` in Python automatically augments the corresponding structure in SIC.

- 2008-mar-06
  - New beta release, still under development. As of the current date, `pysic` and `pygreg` modules are obsolescent. They are replaced by a direct import of Gildas libraries which have been made Python-importable. This can be done by import commands such as `import libgreg`. Please consider that the prefix `lib` will soon disappear, and `pysic` and `pygreg` modules will not be available anymore at this point.
  - Thanks to the point above, some of the Gildas programs are now also importable from Python: `libastro`, `libclass`. More will come soon.

- 2006-mar-13

  - Python modules `pysic` and `pygreg` are now provided to users. Importing one of them into a Python script launches the SIC (resp. GreG) process and imports its variables into the Python `__main__`.
  - `Sic.localspaces`: to prevent overwriting a variable when a deeper-level one is created in SIC with the same name, a saving and unsaving automatic mechanism of the `SicVar` and `SicStructure` instances into a 10-dimensional array (one per level) is now available. To achieve this, the `SicVar` and `SicStructure` instances now have a `__siclevel__` attribute.
  - `Sic.warnings` logical flag activates/deactivates `PygildasWarning`'s.
  - Improvement of the `Numeric` 24.2 retrocompatibility.
  - Enforced variable write-protection (if any) by setting the 'writeable' flag of the `numpy.ndarray`'s. For compatibility with `Numeric`, read-and-write or read-only status in SIC is still checked each time array elements are attempted to be modified.
  - `define()` command can now be passed a second optional boolean argument which describes if the variable will be global or not. By default it is not, e.g. it is local to the current execution level.
  - Images are now automatically imported into Python at creation time. The `get()` command is now also able to import images, thus the `getimage()` command is not available anymore for users.
  - `pexecfile()` function is provided to user. It does the same as the `execfile()` Python function (execute a Python script in current name space), but also prints the commands during execution.
  - `SIC\PYTHON PythonCommandLine` and `SIC\PYTHON PythonProcedureName.py` functionalities now print or not the command lines depending on the `SIC%VERIFY` flag value.
  - `SIC\PYTHON PythonProcedureName.py` functionality now allows to be passed arguments. They can be retrieved in the '`sys.argv`' list from Python. Arguments are parsed by SIC before calling the script. SIC now also searches in the `MACRO#DIR:` directories to find the Python script. Logical names are parsed if any.
  - Exit behavior is now the same whatever is the master process. To quit both SIC and Python, use *CTRL-D* (or `exit()`) from Python, or `EXIT` from SIC. As a consequence, use `PYTHON` (from SIC) and `Sic()` (from Python) to switch to the other command-line interpreter.
  - 64 bits support.

- 2006-oct-12
  - First beta release.

- 2006-sep-13
  - Python embedding into SIC. Python interpreter is started with new SIC command `'PYTHON'`, and can execute Python command-lines, Python scripts, or even jump to interactive mode.
  - Python (numerical) functions can now be called in SIC formulas, provided Python interpreter is started and function exists in Python `__main__`.

  - Support for both `Numeric` and `NumPy` modules: `__sicdata__` attribute is a `numpy.ndarray` if `NumPy` is installed, or a `Numeric.array` if not.

- 2006-jul-07
  - Loss of the `sicarray` type. `SicVar` instances have now two elements: `__sicname__` (the SIC variable name) and `__sicdata__` (a `numpy.ndarray`). Previously `__sicdata__` was a `sicarray` with itself a `Numeric.array` attribute.
  - `NumPy` support: SIC variables (arrays and scalars) are now imported as `numpy.ndarray`, using the same C-API function from `Numeric` that `NumPy` still supports.
  - Adaptation of the adequates methods of the `SicVar` class to handle the unique element of the `numpy.ndarray` scalars through `[0]` indexing (`NumPy` only provides indexing for its scalars through `[()]` index).
  - Attempting to define an image attribute which name is already used for a native method or attribute of the `numpy.ndarray`'s will not raise an error anymore but will only warn the user.
  - `SicVar` strings now support concatenation (`+`) and duplication (`*`).
  - Automatic blank filling when assigning a Python string to a `SicVar` string (`Numeric` provides this feature but `NumPy` does not: it requires the two strings to have the same length).
  - A `SicVar` subarray is now also a `SicVar` with the same `__sicname__` attribute and with a `__sicdata__` sharing its data with the original array, and thus with the SIC variable (previously a `SicVar` subarray was a `Numeric.array`). Such subarrays inherit the `SicVar` specific methods, such as string handling or write protection if any.
  - Usefull functions such as `get()`, `comm()` or `exa()` are now methods of the `Sicfunctions` class, and are visible through the `sic` instance created during the process initialization.
  - Creation of the `sic()`, `greg()`, `define()` and `delete()` methods of the `Sic functions` class.

- 2006-jun-21
  - First unofficial release.

Contents

1 Context **1**

2 End-user documentation **1**
2.1 Prerequisites *(what you have to know)* . . . 1
2.1.1 Python basics . . . 1
Identifiers . . . 1
Multidimensional-array arrangement in memory . . . 1
`vars()` . . . 2
`dir()` . . . 2
`__doc__` . . . 2
2.1.2 `NumPy` basics . . . 3
Attributes . . . 4
Methods . . . 5
Functions . . . 5
Subarrays and derivates . . . 5
2.2 Using Python from SIC . . . 7
2.2.1 The SIC command `PYTHON` . . . 7
2.2.2 Calling Python functions . . . 12
2.3 Using SIC from Python . . . 13
2.3.1 PyGildas modules . . . 13
2.3.2 Special commands . . . 13
`comm()` . . . 13
`setgdict()` . . . 13
`get()` . . . 14
`exa()` . . . 14
`define()` . . . 14
`delete()` . . . 15
`getlogical()` . . . 15
2.3.3 The `gdict` container . . . 15
2.3.4 The `Sic` object in Python `__main__` . . . 16
2.4 Importing SIC objects . . . 17
2.4.1 SIC arrays . . . 17
2.4.2 SIC scalars . . . 18
2.4.3 SIC structures . . . 19
2.4.4 SIC images . . . 20
2.4.5 Handling character strings . . . 21
2.5 Object status in SIC: . . . 23
Read-only or read-and-write? . . . 23
Global or local variable? . . . 23

3 Programmer documentation **26**
3.1 Installing PyGILDAS . . . 26
3.1.1 Prerequisites . . . 26
3.1.2 How to build your own Python version . . . 26
3.1.3 Python module `readline` for command history . . . 27
3.1.4 Install `NumPy` module for Python . . . 28

3.1.5 Install `scipy` module for Python (optional) . . . 29
3.1.6 Compiling Gildas with your custom Python (if any) . . . 30
3.2 How SIC variables are imported . . . 30
3.2.1 Comments on the strategy . . . 30
3.2.2 The `SicVar` class . . . 31
3.2.3 The `SicStructure` class . . . 32
3.3 Array elements types . . . 33

4 Python 3K **34**
4.1 References . . . 34
4.2 Standard end-user . . . 34
4.3 Developers using the Python binding . . . 34
4.4 Developers of Fortran GILDAS packages . . . 35

A What's new? **36**

Index

NumPy, 3