Sage Reference Manual: Python technicalities

Release 8.2

The Sage Development Team

May 07, 2018

CONTENTS

SageMath has various modules to provide access to low-level Python internals.

UTILITIES FOR INTERFACING WITH THE STANDARD LIBRARY'S ATEXIT MODULE.

class sage.cpython.atexit.**restore_atexit**

Bases: object

Context manager that restores the state of the atexit module to its previous state when exiting the context.

INPUT:

- run (bool, default: False) – if True, when exiting the context (but before restoring the old exit functions), run all atexit functions which were added inside the context.

- clear (bool, default: equal to run) – if True, clear already registered atexit handlers upon entering the context.

> **Warning:** The combination run=True and clear=False will cause already-registered exit functions to be run twice: once when exiting the context and again when exiting Python.

EXAMPLES:

For this example we will wrap the entire example with restore_atexit(clear=True) so as to start with a fresh atexit module state for the sake of the example.

Note that the function atexit._run_exitfuncs() runs all registered handlers, and then clears the list of handlers, so we can use it to test manipulation of the atexit state:

```
sage: import atexit
sage: from sage.cpython.atexit import restore_atexit
sage: def handler(*args, **kwargs):
....:     print((args, kwargs))
sage: atexit.register(handler, 1, 2, c=3)
<function handler at 0x...>
sage: atexit.register(handler, 4, 5, d=6)
<function handler at 0x...>
sage: with restore_atexit(clear=True):
....:     atexit._run_exitfuncs()  # Should be none registered
....:     atexit.register(handler, 1, 2, c=3)
....:     with restore_atexit():
....:         atexit._run_exitfuncs()  # Run just registered handler
....:     atexit._run_exitfuncs()  # Handler should be run again
<function handler at 0x...>
((1, 2), {'c': 3})
((1, 2), {'c': 3})
```

We test the `run` option:

```
sage: with restore_atexit(run=True):
....:       # this handler is run when exiting the context
....:       _ = atexit.register(handler, 7, 8, e=9)
((7, 8), {'e': 9})
sage: with restore_atexit(clear=False, run=True):
....:       # original handlers are run when exiting the context
....:       pass
((4, 5), {'d': 6})
((1, 2), {'c': 3})
```

The original handlers are still in place:

```
sage: atexit._run_exitfuncs()
((4, 5), {'d': 6})
((1, 2), {'c': 3})
```

STRING <-> BYTES ENCODING/DECODING

sage.cpython.string.**bytes_to_str**(*b*, *encoding=None*, *errors=None*)

Convert `bytes` to `str`.

On Python 2 this is a no-op since `bytes is str`. On Python 3 this decodes the given `bytes` to a Python 3 unicode `str` using the specified encoding.

EXAMPLES:

```
sage: import six
sage: from sage.cpython.string import bytes_to_str
sage: s = bytes_to_str(b'\xcf\x80')
sage: if six.PY2:
....:      s == b'\xcf\x80'
....: else:
....:      s == u'π'
True
sage: bytes_to_str([])
Traceback (most recent call last):
...
TypeError: expected bytes, list found
```

sage.cpython.string.**str_to_bytes**(*s*, *encoding=None*, *errors=None*)

Convert `str` or `unicode` to `bytes`.

On Python 3 this encodes the given `str` to a Python 3 `bytes` using the specified encoding.

On Python 2 this is a no-op on `str` input since `str is bytes`. However, this function also accepts Python 2 `unicode` objects and treats them the same as Python 3 unicode `str` objects.

EXAMPLES:

```
sage: import six
sage: from sage.cpython.string import str_to_bytes
sage: if six.PY2:
....:      bs = [str_to_bytes('\xcf\x80'), str_to_bytes(u'π')]
....: else:
....:      bs = [str_to_bytes(u'π')]
sage: all(b == b'\xcf\x80' for b in bs)
True
sage: str_to_bytes([])
Traceback (most recent call last):
...
TypeError: expected str ... list found
```

VARIOUS FUNCTIONS TO DEBUG PYTHON INTERNALS

sage.cpython.debug.**getattr_debug**(*obj*, *name*, *default='_no_default'*)
A re-implementation of `getattr()` with lots of debugging info.

This will correctly use `__getattr__` if needed. On the other hand, it assumes a generic (not overridden) implementation of `__getattribute__`. Note that Cython implements `__getattr__` for a cdef class using `__getattribute__`, so this will not detect a `__getattr__` in that case.

INPUT:

- `obj` – the object whose attribute is requested

- `name` – (string) the name of the attribute

- `default` – default value to return if attribute was not found

EXAMPLES:

```
sage: _ = getattr_debug(list, "reverse")
getattr_debug(obj=<type 'list'>, name='reverse'):
  type(obj) = <type 'type'>
  object has __dict__ slot (<type 'dict'>)
  did not find 'reverse' in MRO classes
  found 'reverse' in object __dict__
  returning <method 'reverse' of 'list' objects> (<type 'method_descriptor'>)
sage: _ = getattr_debug([], "reverse")
getattr_debug(obj=[], name='reverse'):
  type(obj) = <type 'list'>
  object does not have __dict__ slot
  found 'reverse' in dict of <type 'list'>
  got <method 'reverse' of 'list' objects> (<type 'method_descriptor'>)
  attribute is ordinary descriptor (has __get__)
  calling __get__()
  returning <built-in method reverse of list object at 0x... (<type 'builtin_
↪function_or_method'>)
sage: _ = getattr_debug([], "__doc__")
getattr_debug(obj=[], name='__doc__'):
  type(obj) = <type 'list'>
  object does not have __dict__ slot
  found '__doc__' in dict of <type 'list'>
  got "list() -> new empty list\nlist(iterable) -> ne~~~ (<type 'str'>)
  returning "list() -> new empty list\nlist(iterable) -> ne~~~ (<type 'str'>)
sage: _ = getattr_debug(gp(1), "log")
getattr_debug(obj=1, name='log'):
  type(obj) = <class 'sage.interfaces.gp.GpElement'>
  object has __dict__ slot (<type 'dict'>)
  did not find 'log' in MRO classes
```

```
  object __dict__ does not have 'log'
  calling __getattr__()
  returning log (<class 'sage.interfaces.expect.FunctionElement'>)
sage: from ipywidgets import IntSlider
sage: _ = getattr_debug(IntSlider(), "value")
getattr_debug(obj=IntSlider(value=0), name='value'):
  type(obj) = <class 'ipywidgets.widgets.widget_int.IntSlider'>
  object has __dict__ slot (<type 'dict'>)
  found 'value' in dict of <class 'ipywidgets.widgets.widget_int._Int'>
  got <traitlets.traitlets.CInt object at ... (<class 'traitlets.traitlets.CInt'>)
  attribute is data descriptor (has __get__ and __set__)
  ignoring __dict__ because we have a data descriptor
  calling __get__()
  returning 0 (<type 'int'>)
sage: _ = getattr_debug(1, "foo")
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'foo'
sage: _ = getattr_debug(1, "foo", "xyz")
getattr_debug(obj=1, name='foo'):
  type(obj) = <type 'sage.rings.integer.Integer'>
  object does not have __dict__ slot
  did not find 'foo' in MRO classes
  class does not have __getattr__
  attribute not found
  returning default 'xyz'
```

sage.cpython.debug.**shortrepr**(*obj*, *max=50*)

Return repr(obj) bounded to max characters. If the string is too long, it is truncated and ~~~ is added to the end.

EXAMPLES:

```
sage: from sage.cpython.debug import shortrepr
sage: print(shortrepr("Hello world!"))
'Hello world!'
sage: print(shortrepr("Hello world!" * 4))
'Hello world!Hello world!Hello world!Hello world!'
sage: print(shortrepr("Hello world!" * 5))
'Hello world!Hello world!Hello world!Hello worl~~~
```

VARIANTS OF GETATTR()

class sage.cpython.getattr.**AttributeErrorMessage**

Bases: object

Tries to emulate the standard Python AttributeError message.

Note: The typical fate of an attribute error is being caught. Hence, under normal circumstances, nobody will ever see the error message. The idea for this class is to provide an object that is fast to create and whose string representation is an attribute error's message. That string representation is only created if someone wants to see it.

EXAMPLES:

```
sage: 1.bla  #indirect doctest
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'bla'
sage: QQ[x].gen().bla
Traceback (most recent call last):
...
AttributeError: 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_
↪rational_flint' object has no attribute 'bla'
```

```
sage: from sage.cpython.getattr import AttributeErrorMessage
sage: AttributeErrorMessage(int(1), 'bla')
'int' object has no attribute 'bla'
```

AUTHOR:

- Simon King (2011-05-21)

cls

name

sage.cpython.getattr.**dir_with_other_class**(*self*, *cls*)

Emulates dir(self), as if self was also an instance cls, right after caller_class in the method resolution order (self.__class__.mro())

EXAMPLES:

```
sage: class A(object):
....:     a = 1
....:     b = 2
....:     c = 3
```

```
sage: class B(object):
....:     b = 2
....:     c = 3
....:     d = 4
sage: x = A()
sage: x.c = 1; x.e = 1
sage: from sage.cpython.getattr import dir_with_other_class
sage: dir_with_other_class(x, B)
[..., 'a', 'b', 'c', 'd', 'e']
```

Check that objects without dicts are well handled:

```
sage: cython("cdef class A:\n    cdef public int a")
sage: cython("cdef class B:\n    cdef public int b")
sage: x = A()
sage: x.a = 1
sage: hasattr(x,'__dict__')
False
sage: dir_with_other_class(x, B)
[..., 'a', 'b']
```

sage.cpython.getattr.**getattr_from_other_class**(*self*, *cls*, *name*)

Emulate getattr(self, name), as if self was an instance of cls.

INPUT:

- self – some object

- cls – a new-style class

- name – a string

If self is an instance of cls, raises an AttributeError, to avoid a double lookup. This function is intended to be called from __getattr__, and so should not be called if name is an attribute of self.

EXAMPLES:

```
sage: from sage.cpython.getattr import getattr_from_other_class
sage: class A(object):
....:     def inc(self):
....:         return self + 1
....:
....:     @staticmethod
....:     def greeting():
....:         print("Hello World!")
....:
....:     @lazy_attribute
....:     def lazy_attribute(self):
....:         return repr(self)
sage: getattr_from_other_class(1, A, "inc")
<bound method A.inc of 1>
sage: getattr_from_other_class(1, A, "inc")()
2
```

Static methods work:

```
sage: getattr_from_other_class(1, A, "greeting")()
Hello World!
```

Caveat: lazy attributes work with extension types only if they allow attribute assignment or have a public attribute `__cached_methods` of type `<dict>`. This condition is satisfied, e.g., by any class that is derived from `Parent`:

```
sage: getattr_from_other_class(1, A, "lazy_attribute")
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'lazy_
→attribute'
```

The integer ring is a parent, so, lazy attributes work:

```
sage: getattr_from_other_class(ZZ, A, "lazy_attribute")
'Integer Ring'
sage: getattr_from_other_class(PolynomialRing(QQ, name='x', sparse=True).one(), A,
→ "lazy_attribute")
'1'
sage: getattr_from_other_class(17, A, "lazy_attribute")
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'lazy_
→attribute'
```

In general, descriptors are not yet well supported, because they often do not accept to be cheated with the type of their instance:

```
sage: A.__weakref__.__get__(1)
Traceback (most recent call last):
...
TypeError: descriptor '__weakref__' for 'A' objects doesn't apply
to 'sage.rings.integer.Integer' object
```

When this occurs, an `AttributeError` is raised:

```
sage: getattr_from_other_class(1, A, "__weakref__")
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute '__weakref__'
```

This was caught by trac ticket #8296 for which we do a couple more tests:

```
sage: "__weakref__" in dir(A)
True
sage: "__weakref__" in dir(1)
False
sage: 1.__weakref__
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute '__weakref__'

sage: n = 1
sage: ip = get_ipython()                     # not tested: only works in interactive
→shell
sage: ip.magic_psearch('n.N')                # not tested: only works in interactive
→shell
n.N
sage: ip.magic_psearch('n.__weakref__')  # not tested: only works in interactive
→shell
```

Caveat: When __call__ is not defined for instances, using `A.__call__` yields the method `__call__` of the class. We use a workaround but there is no guarantee for robustness.

> sage: getattr_from_other_class(1, A, "__call__") Traceback (most recent call last): ... AttributeError: 'sage.rings.integer.Integer' object has no attribute '__call__'

FIVE

METACLASSES FOR CYTHON EXTENSION TYPES

Cython does not support metaclasses, but this module can be used to implement metaclasses for extension types.

> **Warning:** This module has many caveats and you can easily get segfaults if you make a mistake. It relies on undocumented Python and Cython behaviour, so things might break in future versions.

5.1 How to use

To enable this metaclass mechanism, you need to put `cimport sage.cpython.cython_metaclass` in your module (in the `.pxd` file if you are using one).

In the extension type (a.k.a. `cdef class`) for which you want to define a metaclass, define a method `__getmetaclass__` with a single unused argument. This method should return a type to be used as metaclass:

```
cimport sage.cpython.cython_metaclass
cdef class MyCustomType(object):
    def __getmetaclass__(_):
        from foo import MyMetaclass
        return MyMetaclass
```

The above `__getmetaclass__` method is analogous to `__metaclass__ = MyMetaclass` in Python 2.

> **Warning:** `__getmetaclass__` must be defined as an ordinary method taking a single argument, but this argument should not be used in the method (it will be `None`).

When a type `cls` is being constructed with metaclass `meta`, then `meta.__init__(cls, None, None, None)` is called from Cython. In Python, this would be `meta.__init__(cls, name, bases, dict)`.

> **Warning:** The `__getmetaclass__` method is called while the type is being created during the import of the module. Therefore, `__getmetaclass__` should not refer to any global objects, including the type being created or other types defined or imported in the module (unless you are very careful). Note that non-imported `cdef` functions are not Python objects, so those are safe to call.
>
> The same warning applies to the `__init__` method of the metaclass.

> **Warning:** The __new__ method of the metaclass (including the __cinit__ method for Cython extension types) is never called if you're using this from Cython. In particular, the metaclass cannot have any attributes or virtual methods.

EXAMPLES:

```
sage: cython('''
....: cimport sage.cpython.cython_metaclass
....: cdef class MyCustomType(object):
....:     def __getmetaclass__(_):
....:         class MyMetaclass(type):
....:             def __init__(*args):
....:                 print("Calling MyMetaclass.__init__{}".format(args))
....:         return MyMetaclass
....:
....: cdef class MyDerivedType(MyCustomType):
....:     pass
....: ''')
Calling MyMetaclass.__init__(<type '...MyCustomType'>, None, None, None)
Calling MyMetaclass.__init__(<type '...MyDerivedType'>, None, None, None)
sage: MyCustomType.__class__
<class '...MyMetaclass'>
sage: class MyPythonType(MyDerivedType):
....:     pass
Calling MyMetaclass.__init__(<class '...MyPythonType'>, 'MyPythonType', (<type '...
→MyDerivedType'>,), {...})
```

5.2 Implementation

All this is implemented by defining

```
#define PyTypeReady(t)   Sage_PyType_Ready(t)
```

and then implementing the function `Sage_PyType_Ready(t)` which first calls `PyType_Ready(t)` and then handles the metaclass stuff.

SLOT WRAPPERS

A slot wrapper is installed in the dict of an extension type to access a special method implemented in C. For example, `object.__init__` or `Integer.__lt__`. Note that slot wrappers are always unbound (there is a bound variant called method-wrapper).

EXAMPLES:

```
sage: int.__add__
<slot wrapper '__add__' of 'int' objects>
```

Pure Python classes have normal methods, not slot wrappers:

```
sage: class X(object):
....:     def __add__(self, other):
....:         return NotImplemented
sage: X.__add__
<unbound method X.__add__>
```

sage.cpython.wrapperdescr.**wrapperdescr_call**(*slotwrapper*, *self*, **args*, ***kwds*)
    Call a slot wrapper without any type checks.

    The main reason to use this is to call arithmetic slots like `__mul__` without having to worry about whether to call `T.__mul__(a, b)` or `T.__rmul__(b, a)`.

    INPUT:

    - `slotwrapper` – a slot wrapper (for example `int.__add__`).

    - `self` – the first positional argument. Normally, this should be of the correct type (an `int` when calling `int.__add__`). However, this check is skipped: you can pass an arbitrary object.

    - `*args, **kwds` – further arguments.

    > **Warning:** Since this skips type checks, it can easily crash Python if used incorrectly.

    EXAMPLES:

```
sage: from sage.cpython.wrapperdescr import wrapperdescr_call
sage: wrapperdescr_call(Integer.__mul__, 6, 9)
54
sage: wrapperdescr_call(Integer.__mul__, 7/5, 9)
63/5
sage: from sage.structure.element import Element
sage: wrapperdescr_call(Element.__mul__, 6, 9)
54
```

```
sage: wrapperdescr_call(Element.__mul__, 7/5, 9)
63/5
sage: from sage.numerical.mip import MixedIntegerLinearProgram
sage: wrapperdescr_call(type.__call__, MixedIntegerLinearProgram,
↪maximization=False)
Mixed Integer Program (no objective, 0 variables, 0 constraints)
```

DELETE ITEM FROM PYDICT BY EXACT VALUE AND HASH

Beware that the implementation of the routine here relies on implementation details of CPython's dict that go beyond the published API. This file depends on python version when cythonized. It expects PY_VERSION_HEX to be available in the cythonization and the result depends on it (and needs to match the python version the C-compiler compiles it against). Usage should do something along the lines of

> **cythonize("dict_del_by_value.pyx",** compile_time_env({"PY_VERSION_HEX": sys.hexversion}))

AUTHORS:

• Nils Bruin (2017-05)

sage.cpython.dict_del_by_value.**test_del_dictitem_by_exact_value**(*D*, *value*, *h*)
    This function helps testing some cdef function used to delete dictionary items.

    INPUT:

        • D – a Python <dict>.

        • value – an object that is value D.

        • h – the hash of the key under which to find value in D.

    The underlying cdef function deletes an item from D that is in the hash bucket determined by h and whose value is identic with value. Of course, this only makes sense if the pairs (h, value) corresponding to items in D are pair-wise distinct.

    If a matching item can not be found, the function does nothing and silently returns.

    See trac ticket #13394 for a discussion.

```
sage: from sage.cpython.dict_del_by_value import test_del_dictitem_by_exact_value
sage: B=1000
sage: L=list(range(B))
sage: D1=dict()
sage: D2=dict()
sage: for i in range(100000):        # long time
....:     ki=L[floor(random()*B)]
....:     vi=L[floor(random()*B)]
....:     D1[ki]=vi
....:     D2[ki]=vi
....:     ko=L[floor(random()*B)]
....:     if ko in D1:
....:         vo=D1[ko]
....:         del D1[ko]
....:         test_del_dictitem_by_exact_value(D2,vo,hash(ko))
....:     assert D1 == D2
```

No action is taken if the item prescribed by key hash and value does not exist in the dictionary:

```
sage: D = {1: ZZ}
sage: test_del_dictitem_by_exact_value(D, ZZ, 2)
sage: D
{1: Integer Ring}
sage: test_del_dictitem_by_exact_value(D, QQ, 1)
sage: D
{1: Integer Ring}
```

EIGHT

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

C

INDEX