1 Utilities for interfacing with the standard library’s atexit module. 3
2 String <-> bytes encoding/decoding 5
3 Various functions to debug Python internals 7
4 Variants of getattr() 11
5 Metaclasses for Cython extension types 17
   5.1 How to use ...................................................... 17
   5.2 Implementation ................................................... 18
6 Slot wrappers 19
7 Delete item from PyDict by exact value and hash 21
8 Indices and Tables 23
Python Module Index 25
Index 27
SageMath has various modules to provide access to low-level Python internals.
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:

    ```python
    sage: import atexit
    sage: from sage.cpython.atexit import restore_atexit
    sage: def handler(*args, **kwargs):
    ....:     import sys
    # see https://trac.sagemath.org/ticket/25270#comment:56
    ....:     sys.stdout.write(str(args, kwargs))
    ....:     sys.stdout.write('
')
    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...>
    ```
We test the run option:

```python
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})
```

```python
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:

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

TWO

STRING <-> BYTES ENCODING/DECODING

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

Convert bytes to str.

This decodes the given bytes to a Python 3 unicode str using the specified encoding. It is a no-op on str input.

EXAMPLES:

```python
sage: from sage.cpython.string import bytes_to_str
sage: s = bytes_to_str(b'\xcf\x80')
sage: 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.

It encodes the given str to a Python 3 bytes using the specified encoding. It is a no-op on bytes input.

EXAMPLES:

```python
sage: from sage.cpython.string import str_to_bytes
sage: 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
```
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 __getattr__. 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")
ggetattr_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")
ggetattr_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 objects at 0x... (<type 'builtin_method_or_method'>)

sage: _ = getattr_debug([], "__doc__")
ggetattr_debug(obj=[], name='__doc__'):
    type(obj) = <type 'list'>
    object does not have __dict__ slot
    found '__doc__' in dict of <type 'list'>
    got ... 'str'>)
    returning ... 'str'>)

sage: _ = getattr_debug(gp(1), "log")
ggetattr_debug(obj=1, name='log'):

(continues on next page)
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 ... (type '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')

ggetattr_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 world!Hello world~~~

sage.cpython.debug.type_debug(cls)

Print all internals of the type cls

EXAMPLES:

sage: type_debug(object)  # random
<type 'object'> (0x7f57d7f040)
ob_refcnt: 9739
ob_type: <type 'type'>
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tp_name</code></td>
<td>object</td>
</tr>
<tr>
<td><code>tp_basicsize</code></td>
<td>16</td>
</tr>
<tr>
<td><code>tp_itemsize</code></td>
<td>0</td>
</tr>
<tr>
<td><code>tp_dictoffset</code></td>
<td>0</td>
</tr>
<tr>
<td><code>tp_weaklistoffset</code></td>
<td>0</td>
</tr>
<tr>
<td><code>tp_base</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_bases</code></td>
<td>tuple:</td>
</tr>
<tr>
<td><code>tp_mro</code></td>
<td>tuple:</td>
</tr>
<tr>
<td><code>tp_dict</code></td>
<td>dict:</td>
</tr>
<tr>
<td><code>__setattr__</code></td>
<td>&lt;slot wrapper '<strong>setattr</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__reduce_ex__</code></td>
<td>&lt;method '<strong>reduce_ex</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__new__</code></td>
<td>&lt;built-in method '<strong>new</strong>' of type object at 0x7fc57d7594f0&gt;</td>
</tr>
<tr>
<td><code>__reduce__</code></td>
<td>&lt;method '<strong>reduce</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__repr__</code></td>
<td>&lt;slot wrapper '<strong>repr</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__hash__</code></td>
<td>&lt;slot wrapper '<strong>hash</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__sizeof__</code></td>
<td>&lt;method '<strong>sizeof</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__doc__</code></td>
<td>'The most base type'</td>
</tr>
<tr>
<td><code>__init__</code></td>
<td>&lt;slot wrapper '<strong>init</strong>' of 'object' objects&gt;</td>
</tr>
</tbody>
</table>

`tp_alloc`: PyType_GenericAlloc

`tp_new`: 0x7fc57d7594f0

`tp_init`: 0x7fc57d758ee0

`tp_dealloc`: 0x7fc57d757010

`tp_free`: PyObject_Del

`tp_repr`: 0x7fc57d757b90

`tp_print`: NULL

`tp_hash`: _Py_HashPointer

`tp_call`: NULL

`tp_str`: 0x7fc57d757020

`tp_compare`: NULL

`tp_richcompare`: NULL

`tp_getattr`: NULL

`tp_setattr`: NULL

`tp_getattro`: PyObject_GenericGetAttr

`tp_setattro`: PyObject_GenericSetAttr

`tp_iter`: NULL

`tp_iternext`: NULL

`tp_descr_get`: NULL

`tp_descr_set`: NULL

`tp_cache`: NULL

`tp_weaklist`: NULL

`tp_traverse`: NULL

`tp_clear`: NULL

`tp_is_gc`: NULL

`tp_as_number`: NULL

`tp_as_sequence`: NULL
tp_as_mapping: NULL
tp_as_buffer: NULL

```python
tp_flags:
    HAVE_GETCHARBUFFER
    HAVE_SEQUENCE_IN
    HAVE_INPLACEOPS
    HAVE_RICHCOMPARE
    HAVE_WEAKREFS
    HAVE_ITER
    HAVE_CLASS
    BASETYPE
    READY
    HAVE_INDEX
    HAVE_VERSION_TAG
    VALID_VERSION_TAG

```

```python
tp_version_tag: 2

```

```python
sage: type_debug(None)
```

Traceback (most recent call last):
...

```python
TypeError: None is not a type
```
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:**

```python
sage: 1.bla  #indirect doctest
Traceback (most recent call last):
  ...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'bla'
```

```python
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'
```

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

**AUTHOR:**
- Simon King (2011-05-21)

```python
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
```

(continues on next page)
Check that objects without dicts are well handled:

```python
sage: cython("cdef class A:
    cdef public int a")
sage: cython("cdef class B:
    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:**

```python
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)
```
Static methods work:

```python
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:

```python
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
```

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

```python
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:

```python
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:

```python
sage: "__weakref__" in dir(A)
True
sage: "__weakref__" in dir(1) # py2
```

(continues on next page)
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__'

Like getattr(obj, name) but without invoking the binding behavior of descriptors under normal attribute access. This can be used to easily get unbound methods or other descriptors.

This ignores __getattribute__ hooks but it does support __getattr__.

Note: For Cython classes, __getattr__ is actually implemented as __getattribute__, which means that it is not supported by raw_getattr.

EXAMPLES:

sage: class X:
    ....:    @property
    ....:    def prop(self):
    ....:        return 42
    ....:    def method(self):
    ....:        pass
    ....:    def __getattr__(self, name):
    ....:        return "magic " + name

sage: raw_getattr(X, "prop")
<property object at ...>
sage: raw_getattr(X, "method")
<function ...method at ...>
sage: raw_getattr(X, "attr")
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'attr'

sage: x = X()
sage: raw_getattr(x, "prop")
<property object at ...>
sage: raw_getattr(x, "method")
<function ...method at ...>
The same tests with an inherited new-style class:

```python
sage: class Y(X, object):
    ...:     pass
sage: raw_getattr(Y, "prop")
<property object at ...>
sage: raw_getattr(Y, "method")
<function ...method at ...>
sage: raw_getattr(Y, "attr")
Traceback (most recent call last):
  ...
AttributeError: '...' object has no attribute 'attr'
sage: y = Y()
sage: raw_getattr(y, "prop")
<property object at ...>
sage: raw_getattr(y, "method")
<function ...method at ...>
sage: raw_getattr(y, "attr")
'magic attr'
sage: y.__dict__["prop"] = 'no'
sage: y.__dict__["method"] = 'yes'
sage: y.__dict__["attr"] = 'ok'
sage: raw_getattr(y, "prop")
<property object at ...>
sage: raw_getattr(y, "method")
'yes'
sage: raw_getattr(y, "attr")
'ok'
```
CHAPTER 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:

```cython
cimport sage.cpython.cython_metaclass
cdef class MyCustomType(object):
    def __getmetaclass__(self):
        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:

```python
cython('''
....: cimport sage.cpython.cython_metaclass
....: cdef class MyCustomType(object):
....:     def __getmetaclass__(self):
....:         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

```c
#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.
A slot wrapper is installed in the dict of an extension type to access a special method implemented in C. For example, \texttt{object.__init__} or \texttt{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__
# py2
<unbound method X.__add__>
sage: X.__add__
# py3
<function X.__add__ at ...>
```

\begin{Verbatim}
\texttt{sage\textunderscore cpython\textunderscore wrapperdescr\textunderscore wrapperdescr\textunderscore call(slotwrapper, self, *args, **kwds)}
\end{Verbatim}

Call a slot wrapper without any type checks.

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

**INPUT:**

- \texttt{slotwrapper} – a slot wrapper (for example \texttt{int.__add__}).
- \texttt{self} – the first positional argument. Normally, this should be of the correct type (an \texttt{int} when calling \texttt{int.__add__}). However, this check is skipped: you can pass an arbitrary object.
- \texttt{*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)
```

(continues on next page)
```python
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(MixedIntegerLinearProgram.__call__, [5, 6], 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.

AUTHORS:

• Nils Bruin (2017-05)

sage.cpython.dict_del_by_value.init_lookdict()
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 cannot be found, the function does nothing and silently returns.
CHAPTER
EIGHT

INDICES AND TABLES

• Index
• Module Index
• Search Page
C
sage.cpython.atexit, 3
sage.cpython.cython_metaclass, 17
sage.cpython.debug, 7
sage.cpython.dict_del_by_value, 21
sage.cpython.getattr, 11
sage.cpython.string, 5
sage.cpython.wrapperdescr, 19
INDEX

A
AttributeErrorMessage (class in sage.cpython.getattr), 11

B
bytes_to_str() (in module sage.cpython.string), 5

C
cls (sage.cpython.getattr.AttributeErrorMessage attribute), 11

dir_with_other_class() (in module sage.cpython.getattr), 11

G
getattr_debug() (in module sage.cpython.debug), 7
getattr_from_other_class() (in module sage.cpython.getattr), 12

I
init_lookdict() (in module sage.cpython.dict_del_by_value), 21

M
module
sage.cpython.atexit, 3
sage.cpython.cython_metaclass, 17
sage.cpython.debug, 7
sage.cpython.dict_del_by_value, 21
sage.cpython.getattr, 11
sage.cpython.string, 5
sage.cpython.wrapperdescr, 19

N
name (sage.cpython.getattr.AttributeErrorMessage attribute), 11

R
raw_getattr() (in module sage.cpython.getattr), 14
restore_atexit (class in sage.cpython.atexit), 3

S
sage.cpython.atexit module, 3
sage.cpython.cython_metaclass module, 17
sage.cpython.debug module, 7
sage.cpython.dict_del_by_value module, 21
sage.cpython.getattr module, 11
sage.cpython.string module, 5
sage.cpython.wrapperdescr module, 19
shortrepr() (in module sage.cpython.debug), 8
str_to_bytes() (in module sage.cpython.string), 5

T
test_del_dictitem_by_exact_value() (in module sage.cpython.dict_del_by_value), 21
type_debug() (in module sage.cpython.debug), 8

W
wrapperdescr_call() (in module sage.cpython.wrapperdescr), 19