<table>
<thead>
<tr>
<th></th>
<th>Subject</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Utilities for interfacing with the standard library’s atexit module.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>String &lt;-&gt; bytes encoding/decoding</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Various functions to debug Python internals</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Variants of getattr()</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Metaclasses for Cython extension types</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>Slot wrappers</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>Delete item from PyDict by exact value and hash</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>Indices and Tables</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>27</td>
</tr>
</tbody>
</table>
SageMath has various modules to provide access to low-level Python internals.
CHAPTER ONE

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

```python
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):
    ....:     import sys
    ....:     # see https://github.com/sagemath/sage/issues/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...>
```

(continues on next page)
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})
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
```
CHAPTER THREE

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:

```python
sage: _ = getattr_debug(list, "reverse")  # not tested - broken in python 3.12
getattr_debug(obj=<class 'list'>, name='reverse'):
    type(obj) = <class 'type'>
    object has __dict__ slot (<class 'dict'>)
    did not find 'reverse' in object __dict__
    returning <method 'reverse' of 'list' objects> (<class 'method_descriptor'>)
sage: _ = getattr_debug([], "reverse")
getattr_debug(obj=[], name='reverse'):
    type(obj) = <class 'list'>
    object does not have __dict__ slot
    found 'reverse' in dict of <class 'list'>
    got <method 'reverse' of 'list' objects> (<class 'method_descriptor'>)
    attribute is ordinary descriptor (has __get__)
    calling __get__()
    returning <built-in method reverse of list object at 0x... (<class 'builtin_function_or_method'>)
sage: _ = getattr_debug([], "__doc__")
getattr_debug(obj=[], name='__doc__'):
    type(obj) = <class 'list'>
    object does not have __dict__ slot
    found '__doc__' in dict of <class 'list'>
    got ... 'str'>)
    returning ... 'str'>)
sage: _ = getattr_debug(gp(1), "log")  # needs sage.libs pari
getattr_debug(obj=1, name='log'):
    type(obj) = <class 'sage.interfaces.gp.GpElement'>
    object has __dict__ slot (<class 'dict'>)
```

(continues on next page)
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 (<class 'dict'>)
    found 'value' in dict of <class 'ipywidgets.widgets.widget_int._Int'>
    got <traitlets.traits.CInt object at ... (<class 'traitlets.traits.CInt'>)
    attribute is data descriptor (has __get__ and __set__)
    ignoring __dict__ because we have a data descriptor
    calling __get__()
    returning 0 (<class '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) = <class '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!Hello world!Hello worl~~~

sage.cpython.debug.type_debug(cls)

Print all internals of the type cls

EXAMPLES:

sage: type_debug(object)  # random
<class 'object'> (0x7fc57da7f040)
  ob_refcnt:  9739
  ob_type: <class 'type'>
  tp_name: object
  tp_basicsize: 16
  tp_itemsize: 0
  tp_dctoffset: 0
  tp_weaklistoffset: 0
  tp_base (__base__): NULL

(continues on next page)
<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tp_bases</code></td>
<td><code>_bases__</code>: tuple</td>
</tr>
<tr>
<td><code>tp_mro</code></td>
<td><code>_mro__</code>: tuple</td>
</tr>
<tr>
<td><code>tp_dict</code></td>
<td><code>_dict__</code>: 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&gt;</td>
</tr>
<tr>
<td><code>__reduce__</code></td>
<td>&lt;method '<strong>reduce</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__str__</code></td>
<td>&lt;slot wrapper '<strong>str</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__format__</code></td>
<td>&lt;method '<strong>format</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__class__</code></td>
<td>&lt;attribute '<strong>class</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__delattr__</code></td>
<td>&lt;slot wrapper '<strong>delattr</strong>' of 'object' objects&gt;</td>
</tr>
<tr>
<td><code>__subclasshook__</code></td>
<td>&lt;method '<strong>subclasshook</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>
<tr>
<td><code>tp_alloc</code></td>
<td><code>PyType_GenericAlloc</code></td>
</tr>
<tr>
<td><code>tp_new</code></td>
<td>0x7fc57d7594f0</td>
</tr>
<tr>
<td><code>tp_init</code></td>
<td>0x7fc57d758ee0</td>
</tr>
<tr>
<td><code>tp_dealloc</code></td>
<td>0x7fc57d757010</td>
</tr>
<tr>
<td><code>tp_free</code></td>
<td>PyObject_Del</td>
</tr>
<tr>
<td><code>tp_repr</code></td>
<td>0x7fc57d75b990</td>
</tr>
<tr>
<td><code>tp_print</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_hash</code></td>
<td>_Py_HashPointer</td>
</tr>
<tr>
<td><code>tp_call</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_str</code></td>
<td>0x7fc57d757020</td>
</tr>
<tr>
<td><code>tp_compare</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_richcompare</code></td>
<td><em>richcmp</em>_): NULL</td>
</tr>
<tr>
<td><code>tp_getattr</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_setattr</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_getattro</code></td>
<td><em>getattribute</em>_): PyObject_GenericGetAttr</td>
</tr>
<tr>
<td><code>tp_setattro</code></td>
<td><em>setattribute</em>_): PyObject_GenericSetAttr</td>
</tr>
<tr>
<td><code>tp_iter</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_iternext</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_descr_get</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_descr_set</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_cache</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_weaklist</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_traverse</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_clear</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_is_gc</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_as_number</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_as_sequence</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_as_mapping</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_as_buffer</code></td>
<td>NULL</td>
</tr>
<tr>
<td><code>tp_flags</code></td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>HAVE_GETCHARBUFFER</td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>HAVE_SEQUENCE_IN</td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>HAVE_INPLACEOPS</td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>HAVE_RICHCOMPARE</td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>HAVE_WEAKREFS</td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>HAVE_ITER</td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>HAVE_CLASS</td>
<td><em>flags</em>_): NULL</td>
</tr>
<tr>
<td>BASETYPE</td>
<td><em>flags</em>_): NULL</td>
</tr>
</tbody>
</table>
READY
HAVE_INDEX
HAVE_VERSION_TAG
VALID_VERSION_TAG
tp_version_tag: 2
sage: type_debug(\textbf{None})
Traceback (most recent call last):
...
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'

sage: x = polygen(ZZ, 'x')

sage: QQ[x].gen().bla  # needs sage.libs.flint
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)

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:
Python technicalities, Release 10.3

```
sage: class A():
    ....:  a = 1
    ....:  b = 2
    ....:  c = 3
sage: class B():
    ....:  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']
sage: class C():
    ....:  f = 6
sage: dir_with_other_class(x, B, C)
[..., 'a', 'b', 'c', 'd', 'e', 'f']
```

Check that objects without dicts are well handled:

```
sage: # needs sage.misc.cython
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: from sage.cpython.getattr import getattr_from_other_class
sage: class A():
    ....:  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")
```

`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():
    ....:  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")
```

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 ...
```

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 github issue #8296 for which we do a couple more tests:

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

sage.cpython.getattr.raw_getattr(obj, name)

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 __getattr__ hooks but it does support __getattribute__.

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

EXAMPLES:
The same tests with an inherited new-style class:

```python
sage: class Y(X, object):
.....:    pass

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: 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

META CLASSES 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, and turn off the Cython directive `always_allow_keywords`. This method should return a type to be used as metaclass:

```cython
@cython.always_allow_keywords(False)
def __getmetaclass__(_):
    from foo import MyMetaclass
    return MyMetaclass
```

**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
sage: cython(  # needs sage.misc.cython
    ....: '"
    ....: cimport cython
    ....: cimport sage.cpython.cython_metaclass
    ....: @cython.cdef_class MyCustomType():
    ....:     @cython.always_allow_keywords(False)
    ....:     def __getmetaclass__(_):
    ....:         @cython.cdef_class MyMeta(type):
    ....:             def __init__(*args):
    ....:                 print("Calling MyMeta.__init__(".format(args))
    ....:             return MyMeta
    ....:     cdef class MyDerivedType(MyCustomType):
    ....:         pass
    Calling MyMeta.__init__(<class ...MyCustomType>, None, None, None)
    Calling MyMeta.__init__(<class ...MyDerivedType>, None, None, None)
    sage: class MyPythonType(MyDerivedType):  # needs sage.misc.cython
    ....:     pass
    Calling MyMeta.__init__(<class ...MyPythonType>, 'MyPythonType', (<class ...MyDerivedType'>,), {...})
```

5.2 Implementation

All this is implemented by defining

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

and then implementing the function \texttt{Sage\_PyType\_Ready(t)} which first calls \texttt{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, 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:
....:     def __add__(self, other):
....:         return NotImplemented
sage: X.__add__
<function X.__add__ at ...>
```

```
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)
```

(continues on next page)
54
sage: wrapperdescr_call(Element.__mul__, 7/5, 9)
63/5
sage: from sage.numerical.mip import MixedIntegerLinearProgram
   #... needs sage.numerical.mip
sage: wrapperdescr_call(type.__call__,
   #... needs sage.numerical.mip
.....: 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.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 identical 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.
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