

MATISSE[®] Python Programmer's Guide

August 2009



MATISSE Python Programmer's Guide

Copyright ©1992–2009 Matisse Software Inc. All Rights Reserved.

Matisse Software Inc.
930 San Marcos Circle
Mountain View, CA 94043
USA

Printed in USA.

This manual is copyrighted. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without prior written consent of Matisse Software Inc. This manual and the software described in it are provided under the terms of a license between Matisse Software Inc. and the recipient, and their use is subject to the terms of that license.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. and international patents.

TRADEMARKS: MATISSE and the MATISSE logo are registered trademarks of Matisse Software Inc. All other trademarks belong to their respective owners.

PDF generated 21 August 2009

Contents

Introduction	6
Intended Audience	6
Conventions	6
1 Overview	7
2 Using the Interface	8
2.1 Dynamic Object Interface	8
2.2 Accessing the Database	8
2.3 Creating Objects	11
2.4 Manipulating Attribute Values	11
2.5 Manipulating Relationship Values	13
2.6 Retrieving Objects	14
2.7 Removing Objects	16
2.8 Class Discovery	16
2.9 Extending the Database Schema Dynamically	17
3 Python Interface Class Hierarchy	18
4 Python Class Reference	19
Database.....	19
MT	23
MtObject.....	25
MtObjectCursor	28
SQLStatement.....	29
SQLCursor.....	31
VersionCursor	31
5 Meta-Schema Class Descriptions	33
MtAttribute.....	33
MtClass.....	34
MtIndex.....	35
MtEntryPointDictionary.....	36
MtRelationship	36
6 MATISSE Exception Reference	38
Index	41

Tables

Table 4.1	MATISSE and Python Data Types	25
Table 4.2	Default Python to MATISSE Mapping	28

Figures

Figure 4.1	MtObjectCursor Inheritance	28
Figure 4.2	SQLCursor Inheritance	31
Figure 4.3	VersionCursor Inheritance	32
Figure 5.1	MtAttribute Inheritance	33
Figure 5.2	MtClass Inheritance	34
Figure 5.3	MtIndex Inheritance	35
Figure 5.4	MtRelationship Inheritance	37

Introduction

Intended Audience

This manual should be read by anyone developing Python scripts or applications interacting with the MATISSE Object database.

You may also read:

- ◆ [Getting Started with Matisse®](#)
- ◆ [Matisse® SQL Programmer's Guide](#)
- ◆ [Matisse® ODL Programmer's Guide](#)

Conventions

This document uses the following conventions:

Text

The running text is written in characters like these.

Code

All computer variables, code, commands, and interactions are shown in this font. Also, any code and commands that the user must enter are shown in this font on a gray background.

variable

In a program example, or in an interaction, a variable, which means anything that is dependent on the user environment, is written in italics.

References

References to another part of the MATISSE documentation are made as shown here.

```
class
```

In the drawings, classes are shown in a black rectangle.



In the drawings, inheritance is shown with the OMT convention. A black triangle is for inheritance.



1 Overview

This manual documents the interface between the MATISSE object database and the Python scripting language.

The MATISSE-Python interface consists of a set of classes that provide access to MATISSE objects. MATISSE objects are stored in a language- and platform-independent format so that objects created using the C, C++, and Java interfaces can be used by the MATISSE-Python binding without conversions. Also, objects created or modified by the MATISSE-Python interface are accessible from any other platform or language transparently.

The MATISSE-Python interface is made of two layers:

- ◆ The low-level binding is a Python extension, written in C, that maps every MATISSE C function to a Python function. This layer fits in the shared object “`_matisse.so`” (or dynamic library “`_matisse.dll`” on Windows platforms) that will be dynamically loaded by Python when “`import _matisse`” is executed.
- ◆ The second layer is a 100% Python module named `MATISSE.py`. This module is a set of Python classes that includes all the functions from the low-level `_matisse` modules. This layer provides a coherent and easy way to access MATISSE objects from Python. `MATISSE.py` also takes advantage of many Python language features to accelerate your development with MATISSE.

NOTE: This manual only documents the high-level binding. For more information about the low-level binding, you can refer to the [MATISSE C API Reference](#), and look at the MATISSE-Python source code that is freely available.

You can create your schema (class, attribute, relationship, etc.) with the MATISSE-Python binding, but generally, you would prefer to write an ODL file or SQL statements to do so. Currently the `mt_odl` utility does not generate stubs for Python. The ODL file must be directly loaded into MATISSE with the `mt_odl` utility.

A complete description of the MATISSE ODL syntax is provided in the [Matisse® ODL Programmer's Guide](#).

2 Using the Interface

2.1 Dynamic Object Interface

Why a Dynamic Object Interface?

Applications accessing objects from different databases may need to discover the object description on the fly. Also, applications providing dynamic application schema extensions may need to manipulate dynamically new object descriptions.

For these applications, MATISSE provides the Dynamic Object Interface (DOI).

What is DOI?

DOI is a way to handle an object without any predefined information about its description such as its class, super-classes, and properties.

DOI provides an interface to discover the class description of an object and to manipulate its contents.

DOI provides an interface to browse the application schema, to define new classes, and to create new objects.

This interface allows you to retrieve the class of the object. Then from the class, you can discover all the properties (superclasses, attributes, and relationships), and you can also update the properties.

The interface allows you to identify the content (data type and value) of each attribute or relationship of an object and to update it.

By using DOI, an application can handle dynamic changes without the need to upgrade the application itself.

2.2 Accessing the Database

Managing Connections

The Database class allows you to manage the connection to the database. The usual way to establish a connection to the database is as follows:

```
from MATISSE import *

db = Database("aHostName" , "aDatabaseName")
# establish the connection
db.open()
# select a current database
db.select()
#
```

```

# management of your data
#
# unselect the currently selected database
db.unselect()
# close the connection
db.close()

```

Whenever access control is enabled, you may want to connect to the database using a specific username/password. The *username* and *password* can be specified using the `open` method.

The `SetConnectionOption` method allows you to set an option of the database descriptor.

```

from MATISSE import *

db = Database("aHostName", "aDatabaseName")
# Set the dynamic schema modification on
db.setConnectionOption
(Database.DATA_ACCESS_MODE,
 MT.DATA_DEFINITION)
# establish the connection
db.open("aUsername", "aPasswd")

```

Multiple databases can easily be managed by opening several connections and switching from one connection to another. You can always retrieve the currently selected connection with the `databaseGetCurrent()` function.

```

from MATISSE import *

db1 = Database("aHostName", "aDatabaseName")
db2 = Database("aHostName_2", "aDatabaseName_2")
# establish the connections
db1.open()
db2.open()
# select a current database
db2.select()
# ...
# unselect the current database and
# select a new one
curentDB = databaseGetCurrent()
curentDB.unselect()
db1.select()

```

Handling Transactions

To update the database, a transaction has to be started. A transaction can be opened only when a connection is selected. Starting the transaction must be done explicitly; there is no implicit transaction when accessing the database.

Only one transaction or version access at a time can be opened on a connection.

```

from MATISSE import *

```

```

db = Database("host", "database")
db.open()
db.select()
# open a transaction context
db.startTransaction()
#
# update your data
#
# Commit the changes
db.commit()
db.unselect()
db.close()

```

A database version corresponding to a committed transaction can be created. You may define the prefix of the database version name when committing the transaction. The `commit` method will return a unique name for the database version, avoiding name conflict.

```

# open a transaction context
db.startTransaction()
#
# update your data
#
# Commit the changes
dbVersionName = db.commit("Monday")

```

When a transaction is started, it can be cancelled at any time by using the `rollback` method.

Accessing a Database Version

A database version can only be accessed when a connection is selected. Accessing a database version must be done explicitly. Only one transaction or version access at a time can be opened on a connection.

The argument of the `startVersionAccess` method, when it is specified, must be the name of a database version. If it is not specified, the current database version is accessed.

```

from MATISSE import *

db = Database("host", "database")
db.open()
db.select()
# open a database version context
db.startVersionAccess("Monday00000561")
#
# access data
#
# end the database version context
db.endVersionAccess()
db.unselect()
db.close()

```

Catching Exceptions

The MATISSE Python interface manages exceptions. All the exceptions specific to MATISSE inherit from the `MtException` class.

```
try:
    db = Database("host", "database")
    db.open()
#
# ...
#
except ConnectError
    # error management
```

2.3 Creating Objects

Using the Dynamic Invocation Interface

To create a new object, you may use the constructor defined on the `MtObject` class:

```
cl = db.getMtClass("Person")
obj = MtObject(cl)
```

Persistent like Python Class

For instance, to define a persistent Python class for `Person`, you may write the following source code:

```
class Person(MtObject):
    def __init__(self, firstname, lastname):
        MtObject.__init__(self,
            databaseGetCurrent().getMtClass("Person"))
        self["FirstName"] = firstname
        self["LastName"] = lastname
    ...
```

Then to create a new instance of `Person` in the database:

```
p = Person("John", "Smith")
```

Note that object creation/manipulation must be performed in a database transaction.

2.4 Manipulating Attribute Values

You can manipulate attribute values using the “[]” operator. The operator argument is the property name given as a character string or as an object representing the property.

Updating an Attribute Value

To set the `FirstName` and `LastName` values of an instance of `Person`:

```
# Getting the Schema descriptors
att = db.getMtAttribute("LastName")
cl = db.getMClass("Person")
# Creating an object
aPerson = MtObject(cl)
# updating the FirstName attribute
aPerson["FirstName"] = "John"
# updating the LastName attribute
aPerson[att] = "Smith"
```

The `setValue` method allows you to set an attribute value with a specific MATISSE datatype.

To set the `Age` of an instance of `Person`:

```
# Getting the Schema descriptors
att = db.getMtAttribute("Age")
cl = db.getMtClass("Person")
# Creating an object
aPerson = MtObject(cl)
# setting the age
aPerson.setValue(att, 21, MT.SHORT)
```

The `clearValue` method allows you to remove the current value of an attribute. A subsequent access to the attribute will return the default value.

To remove the `FirstName` value of an instance of `Person`:

```
att = db.getMtAttribute("FirstName")
# clearing the FirstName value
aPerson.clearValue(att)
```

or:

```
del aPerson[att]
```

or:

```
del aPerson["FirstName"]
```

Accessing an Attribute Value

To access the `FirstName` and `LastName` values of an instance of `Person`:

```
att = db.getMtAttribute("LastName")
aPerson = db.lookup("Smith", att)
# accessing the FirstName
firstName = aPerson["FirstName"]
# accessing the LastName using
# the attribute descriptor
LastName = aPerson[att]
```

The `getValue` method returns a full description of the value. This method returns an instance of `MtValue` containing:

- ◆ `value`: The value itself
- ◆ `defaultP`: The boolean indicating whether it is a default value
- ◆ `dimensions`: The sequence of dimensions for a list or an array
- ◆ `mtType`: MATISSE data type returned as integer. You can use the constants defined in the `MT` class to compare with a specific type.

To access the `LastName` value descriptor of an instance of `Person`:

```
att = db.getMtAttribute("LastName")
aPerson = db.lookup("Smith", att)
# accessing the LastName Descriptor
LastNameVal = aPerson.getValue(att)
type = LastNameVal.mtType
if type == MT.STRING:
    lastName = LastNameVal.value
```

Streaming a Large Attribute Value

You may want to stream by chunks media asset data that is stored in the database. The MATISSE Python interface provides methods to read and update parts of an attribute value. These methods support only the LIST-type values such as `MT.BYTE`, `MT.SHORT`, `MT.INTEGER`, `MT.FLOAT` or `MT.DOUBLE`.

`getListElements` returns an array of elements from the attribute value, starting at `offset`. The end of the list is reached when the returned length is less than the requested length `num`.

The declaration is:

```
MtObject.getListElements(attr, type, num,
offset=MT.CURRENT_OFFSET)
```

`setListElements` updates the list attribute value with the content of `elements`, starting at `offset`. When `discardAfter` is true, the last element in `elements` will become the last element of the list attribute value; otherwise the elements after the updated chunk remain unchanged.

The declaration is:

```
MtObject.setListElements(attr, type, elements,
offset=MT.CURRENT_OFFSET, discardAfter=1 )
```

2.5 Manipulating Relationship Values

You manipulate relationship values using the “[]” operator. The operator argument is the property name given as a character string or an object representing the property. The returned value is a Python list of instances of `MtObject`.

Updating a Relationship Value

To update the `Siblings` successors of an instance of `Person`:

```
rel = db.getMRelationship("Siblings")
# updating the Siblings relationship
aPerson[rel] = (brother,)
# add successors to the Siblings relationship
aPerson[rel] = aPerson[rel] + (brother, sister)
# clearing the relationship content
aPerson[rel] = ()
```

or:

```
aPerson["Siblings"] = ()
```

or:

```
del aPerson[rel]
```

Accessing a Relationship Value

To access the `Siblings` successors of an instance of `Person`:

```
rel = db.getMRelationship("Siblings")
# get the successors of some instance of Person
s = aPerson[rel]
```

or:

```
s = aPerson["Siblings"]
```

2.6 Retrieving Objects

Cursors

To visit objects from the database, you will usually call a method that returns an instance cursor. A cursor represents a group of objects that are accessible one after another using the `next` method.

```
...
# the cursor is already opened
cur = xxx.openXXXCursor(...);
obj = cur.next()
while obj:
    ...
    obj = cur.next()
cur.close()
...
```

When all elements of a cursor have been visited, the `next` method then returns `null`. The cursor then must be closed using the `close` method.

Entry Point

If an entry-point dictionary has been declared for a given attribute, you can open a cursor on the entry-point with the `openEntryPointCursor` method .

```
Cursor MtObject.openEntryPointCursor(String key,
MtEntryPointDictionary dict, MtClass cl=None)
```

The `cl` argument is a filter on the class for the objects that are returned through the cursor. For example, objects can be selected regarding a particular sub-class. The `cl` argument is optional.

For example, retrieving all the persons named Smith:

```
dict = db.getMtEntryPointDictionary("LastNameDict")
cl = db.getMtClass("Person")
cur = db.openEntryPointCursor
        ("Smith", dict, cl)
...

```

When the entry-point is a primary key, only one object is returned, and a cursor is not appropriate. The `lookup` method returns the object accessible through the entry-point. If there is no object, the `lookup` method returns `null`. If there is more than one object, an `InvalidOperation` exception is raised.

For example, retrieving a person by his Social Security Number (SSN):

```
dict = db.getMtEntryPointDictionary("SSNDict")
aPerson = db.lookup("000-00-00", dict);

```

Class Extent

For each class, MATISSE maintains a set of the instances of the class. The `openInstanceCursor` method returns a cursor that enables visiting all instances of the class.

Visiting all the instances of the Person class:

```
cl = db.getMtClass("Person")
cur = db.openInstanceCursor(cl)
p = cur.next()
while p:
    ...
    p = cur.next()
cur.close()

```

Index

Let's suppose the class `Person` is indexed by last name and age. The index name is `Persons`. To retrieve all the persons named Smith who are between 20 and 30 years old, you may write:

```
start = ("Smith", 20)
end = ("Smith", 30)
cl = db.getMtClass("Person")
cur = db.openIndexCursor("Persons",
                        cl,
                        MT.DIRECT,
                        start,
                        end)

p = cur.next()

```

```

while p:
    ...
    p = cur.next()
cur.close()

```

SQL Query

To retrieve objects using a SQL SELECT statement, you can define a projection with the `REF()` column. For example, to retrieve all instances of the class `Person` older than 20:

```

stmt = SQLStatement()
stmt.execDirect("SELECT REF(p) FROM Person p WHERE p.age >
20")
if stmt.getType() == MT.SQL_SELECT:
    cur = stmt.openStream()
    p = cur.nextRow()
    while p:
        print "%-20s" % (p[0]["FirstName"])
        p = cur.nextRow()
    cur.close()

```

2.7 Removing Objects

To remove an object from the database, you must use the `remove` method.

```

att = db.getMtAttribute("LastName")
aPerson = db.lookup("Smith", att)
# remove the object
aPerson.remove()

```

2.8 Class Discovery

DOI provides an interface to discover the class description of an object and to manipulate its contents.

To access the value of each attribute, you may browse the object attributes through a cursor and apply the `getValue` method for each attribute:

```

dict = db.getMtEntryPointDictionary("LastNameDict")
aPerson = db.lookup("Smith", dict)
# discover the attributes
attcur = aPerson.openAttributeCursor();
att = attcur.next()
while att:
    # get the attribute name
    attName = att["MtName"]
    # get the attribute value
    val = aPerson.getValue(att)

```

```

# get the attribute type
t = val.mtType
# get the value rank
rank = len(val.dimensions)
att = attcur.next()
attcur.close()

```

2.9 Extending the Database Schema Dynamically

The MATISSE Python interface allows you to define new classes on the fly. For instance, creating the class `GIFFile` as a subclass of the existing class `File`:

```

# Retrieve "File" class from its name
fileClass = db.getMtClass("File")
# Create new "GIFFile" class
newClass = MtClass("GIFFile")
# make it sub-class of "File"
newClass["MtSuperclasses"] = (fileClass,)
# create an attribute
newAtt = MtAttribute("Version")
# link the attribute with the new class
newClass["MtAttributes"] = (newAttribute,)

```

A simpler alternative is to execute a SQL DDL (Data Definition Language) statement:

```

stmt = SQLStatement()
stmt.execDirect("CREATE CLASS GIFFile UNDER File (Version
STRING)")

```

NOTE: To extend the application schema dynamically, the database connection has to be opened in the `MT.DATA_DEFINITION` mode. See the `Database.setOption` method for more details.

3 Python Interface Class Hierarchy

Inheritance Graph

This section presents the class hierarchy of the MATISSE Python interface.

The binding classes are defined in the `MATISSE` module.

MATISSE Objects

```
MtObject  
  SchemaObject (UNDOCUMENTED)  
    MtAttribute  
    MtClass  
    MtRelationship  
    MtIndex  
    MtEntryPointDictionary
```

Cursors

```
Cursor (UNDOCUMENTED)  
  MtObjectCursor  
    MtObjectSetCursor  
      SQLCursor  
  MtPropertyCursor  
  VersionCursor  
SQLStatement
```

Database

```
Database  
MT
```

4 Python Class Reference

Database

Synopsis `module MATISSE`
 `class Database`

Description This class handles database mechanisms such as connection, transaction, and locking. It also allows retrieval of objects through entry points or indexes.

Example `db = Database("myHost", "myDatabase")`
 `db.open("myUsername", "myPasswd")`
 `db.select()`
 `db.startTransaction()`
 `cur = db.openEntryPointCursor("Mary",`
 `firstNameAtt, personCl)`
 `...`
 `db.commit()`
 `db.unselect()`
 `db.close()`
 `del db`

Fields `static SERVER_EXECUTION_PRIORITY`
 `static LOCK_WAIT_TIME`
 `static DATA_ACCESS_MODE`

These class variables define the connection options of the `setOption` and `getOption` methods.

Constructors `Database(String hostName, String databaseName)`

This constructor defines a database descriptor.

Methods

Close `close()` throws `MtException`

This method disconnects the application from the database. Note that to call this method, there must be no transaction or version access still opened, and no instance of `Database` should be selected.

Commit `String commit(String prefix=None)` throws `MtException`

These methods validate a transaction. When a transaction commit is performed:

- ◆ Validation and check functions are triggered on updated objects.

- ◆ The transaction is committed. When a prefix argument is specified, an access version is saved in the database. The access version name is the concatenation of the `prefix` and of the eight-character string added by MATISSE to avoid name conflicts.

If a validation error occurs during the transaction, it is neither validated nor cancelled. The transaction is still active and all corrective measures can be undertaken.

End Read-Only Transaction `endVersionAccess()` throws `MtException`
This method ends a transaction in read only mode.

Get Connection Information `String getHost()`
 `String getName()`
 `bool connectionIsOpen()`
 `int getOption(int optionDsc)`

These methods return information from the database descriptor. The following options are available:

- ◆ `SERVER_EXECUTION_PRIORITY`: defines the execution priority of the connection on server side.
- ◆ `LOCK_WAIT_TIME`: defines the maximum waiting time to acquire a lock on an object.
- ◆ `DATA_ACCESS_MODE`: defines the data access mode of the connection.

Access to Schema Information `MtClass getMtClass(String mtName)`
 throws `MtException`
 `MtAttribute getMtAttribute(String mtName, MtClass`
 `aClass=None) throws MtException`
 `MtRelationship getMtRelationship(String mtName,`
 `MtClass aClass=None)) throws MtException`
 `MtIndex getMtIndex(String idxName) throws MtException`
 `MtEntryPointDictionary getMtEntryPointDictionary(String`
 `dictName) throws MtException`

These methods provide access to MATISSE schema descriptors for classes, attributes, and relationships.

Lock Objects `lockObjectsFromEntryPoint`
 `(int lock,`
 `String keyword,`
 `MtEntryPointDictionary dict,`
 `MtClass cl=None)`
 throws `MtException`

These methods lock all the objects that are accessible by an entry point. The lock value is `MT.WRITE` or `MT.READ`.

Get Single Object from Entry-Point `MtObject lookup`
 `(String keyword,`
 `MtEntryPointDictionary dict,`

```
MtClass cl=None)  
throws MtException
```

This method returns the object accessible through the entry-point. If there is no object, `None` is returned. If there are more than one object, an `InvalidOperation` exception is raised.

The `cl` argument is a filter on the class for the objects that are returned through the cursor. For example, objects can be selected regarding a particular subclass.

```
Open Connection    open (username=None,  
                    password=None)  
                    throws MtException
```

These methods open a connection with the database described in the `Database` object. In order to be able to call it, no database must be selected.

This method uses the system user id as `username` whenever the parameters `username` and `password` are omitted or are set to `None`.

```
Entry-Point Cursor  MtObjectCursor openEntryPointCursor  
                    (String keyword,  
                    MtEntryPointDictionary dict,  
                    MtClass cl=None)  
                    throws MtException
```

This method returns a cursor that enables you to visit all the objects accessible by an entry point.

The `cl` argument is a filter on the class for the objects that are returned through the cursor. For example, objects can be selected regarding a particular subclass.

```
Index Cursor        MtObjectCursor openIndexCursor  
                    (String index,  
                    MtClass cl,  
                    int direction,  
                    List start=None,  
                    List end=None)  
                    throws MtException
```

This method returns a cursor that contains the result of index range queries.

The `cl` argument is a filter on the class for the objects that are returned through the cursor. For example, objects can be selected regarding a particular subclass. The value of `cl` can be set to `None`.

The `start` and `end` parameters define the boundaries for the range query.

`direction` defines the direction of the result contained in the cursor. The possible values for `direction` are `MT.DIRECT` or `MT.REVERSE`.

```
Get Object from    MtObject getObjectFromMtOid  
OID                (int oid)
```

This method returns the object accessible through an OID.

Get Instance Count	<pre>int getInstanceNumber(MtClass aClass) throws MtException</pre> <p>This class method returns the number of instances of the argument class.</p>
Visit Instances	<pre>MtObjectCursor openInstanceCursor(MtClass aClass) throws MtException</pre> <p>This class method returns a cursor that enables you to go through all the instances of a class.</p>
Version Cursor	<pre>VersionCursor openVersionCursor() throws MtException</pre> <p>This method returns a cursor that provides the names of all the versions that have been saved into the database.</p>
Rollback	<pre>rollback() throws MtException</pre> <p>This method ends the transaction in progress and cancels all the changes.</p>
Select Database	<pre>select() throws MtException</pre> <p>This method selects the database as the current selected database.</p> <p>The <code>select</code> method can be called at any time, even if a transaction is in progress in the currently selected database. This mechanism enables switching contexts back and forth.</p>
Set Option	<pre>setOption (int optionDsc, int optionValue) throws MtException</pre> <p>This method sets an option for the database connection. The following options are available:</p> <ul style="list-style-type: none"> ◆ <code>SERVER_EXECUTION_PRIORITY</code>: defines the execution priority of the connection on the server side. The MT class defines the possible values. <code>MT.MIN_SERVER_EXECUTION_PRIORITY</code> is the default value. ◆ <code>LOCK_WAIT_TIME</code>: defines the maximum waiting time to acquire a lock on an object. The possible values are <code>MT.WAIT_FOREVER</code>, <code>MT.NO_WAIT</code>, or the number of seconds the user is willing to wait. <code>MT.NO_WAIT</code> prevents deadlocks. The <code>LOCK_WAIT_TIME</code> option can be updated at any time. <code>MT.WAIT_FOREVER</code> is the default value. ◆ <code>DATA_ACCESS_MODE</code>: defines the data access mode of the connection. <code>MT.DATA_MODIFICATION</code> enables updating of the data. <code>MT.DATA_DEFINITION</code> enables updating the application schema as well as the data itself. <code>MT.DATA_READONLY</code> allows only reading of the data. <code>MT.DATA_MODIFICATION</code> is the default value.
Start Transaction	<pre>startTransaction(int priority=MT.MIN_TRAN_PRIORITY) throws MtException</pre> <p>This method starts a transaction in order to update objects into the database.</p>

The `priority` parameter assigns a priority to the transaction in order to choose a victim in case of a deadlock. The victim among all locked transactions is the one with the lowest priority and the least amount of updates.

The `priority` value ranges between 0 and 9.

Start Read-Only Transaction `startVersionAccess(
String versionName=None)
throws MtException`

This method starts a transaction in read-only mode. This feature allows access to a consistent view of the database without any locking.

The `versionName` parameter can contain a saved version provided by an earlier commit operation. When the parameter is omitted the most recent database version (or “current version”) is accessed.

Database State `boolean transactionIsInProgress()
throws MtException
boolean versionAccessIsInProgress()
throws MtException`

These methods indicate whether a transaction or a version access is in progress.

Print `__repr__()`

This method returns a string such as "Database('database', 'host')".

Unselect Database `unselect()
throws MtException`

This method deselects the currently selected database.

The `unselect` method can be called at any time, even if a transaction is in progress. The transaction context is preserved until the `select` method is called back.

Get Current Database `Database databaseGetCurrent()`

This function returns the currently selected database.

MT

Synopsis `module MATISSE
class MT`

Description This class handles constants.

Variables All these class variables define the MATISSE Python Interface constants.

Execution Priority `MIN_SERVER_EXECUTION_PRIORITY
NORMAL_SERVER_EXECUTION_PRIORITY`

	ABOVE_NORMAL_SERVER_EXECUTION_PRIORITY
	MAX_SERVER_EXECUTION_PRIORITY
Wait Time	NO_WAIT
	WAIT_FOREVER
Access Mode	DATA_DEFINITION
	DATA_MODIFICATION
	DATA_READONLY
Transaction Priority	MIN_TRAN_PRIORITY
	MAX_TRAN_PRIORITY
Index Direction	DIRECT
	REVERSE
Lock Types	READ
	WRITE
Successor Position	AFTER
	APPEND
	FIRST
Value Types	BOOLEAN
	CHAR
	DATE
	DOUBLE
	DOUBLE_LIST
	FLOAT
	FLOAT_LIST
	LONG
	LONG_LIST
	NIL
	NUMERIC
	SHORT
	SHORT_LIST
	INTEGER
	INTEGER_LIST
	STRING
	STRING_LIST
	TIME_INTERVAL
	TIMESTAMP
	BYTE
	BYTES
Index	MAX_CRITERIA
	ASCEND
	DESCEND

MtObject

Synopsis `module MATISSE`
 `class MtObject`

Description This class implements all the mechanisms useful for accessing objects in the Python binding.

Constructor `MtObject(MtClass c1)` throws `MtException`
This constructor creates a new instance of `c1` in the database.

Methods

Adding and Setting Successors `addSuccessor(MtRelationship rel, MtObject obj, int where=MT_APPEND, MtObject after=None)`
 `addSuccessors(MtRelationship rel, List<MtObject> objs)`
 `setSuccessors(MtRelationship rel, List<MtObject> objs)`
 `__setitem__(MtRelationship rel, List<MtObject> objs)`
 `...`

Comparison `int __cmp__(MtObject other)`
This method returns 0 if the MATISSE object identifiers are the same.

Free `free()`
Free the MATISSE object from the client cache memory.

[] Operator any `__getitem__(property prop)` throws `MtException`
This method returns the value of a property. The `prop` argument can either be an instance or a name of an `MtAttribute` or `MtRelationship`. If `prop` is a relationship, `__getitem__` will return a list of `MtObject`. If `prop` is an attribute, `__getitem__` will return the MATISSE value converted into a Python type.

[Table 4.1](#) lists the mapping from MATISSE datatypes to Python data types.

Table 4.1 MATISSE and Python Data Types

MATISSE Data Type	Python Type	Comment
<code>MT_BOOLEAN</code>	<code>Int</code>	0 or 1
<code>MT_TIMESTAMP</code>	<code>List</code>	(Y,M,D,H,M,S,m)
<code>MT_DATE</code>	<code>List</code>	(Y,M,D,0,0,0,0)
<code>MT_INTERVAL</code>	<code>List</code>	(s,D,H,M,S,m)
<code>MT_BYTE</code>	<code>Int</code>	
<code>MT_SHORT</code>	<code>Int</code>	

Table 4.1 MATISSE and Python Data Types

MATISSE Data Type	Python Type	Comment
MT_INTEGER	Int	
MT_LONG	Long	Possible data loss
MT_NUMERIC	String	
MT_CHAR	String	
MT_DOUBLE	Float	
MT_NIL	None	
MT_STRING	String	
MT_FLOAT	Float	
MT_BYTES	String	
MT_XXX_LIST	List of xxx	

Fetch `fetch()` throws `MtException`

This method loads the attributes and relationships parts of an object at the same time. In MATISSE, attributes and relationships parts are managed separately to improve parallelism and to enhance performance.

Note that the `Database` class provides a `fetch` method to load several objects at the same time.

Stream Value Read `List getListElements(MtAttribute att, int type, int num, int first=MT.CURRENT_OFFSET)`
`...`

Get Class `MtClass getMtClass()`
throws `MtException`

This method returns an instance of `MtClass` corresponding to the class of the `MtObject`.

Get OID `int getMtOid()`
`int __int__()`

This method returns the object ID (`MtOid`) of this object.

Get Successors and Predecessors `List<MtObject> getSuccessors(MtRelationship)`
`List<MtObject> getPredecessors(MtRelationship)`
`List<MtObject> __getitem__(MtRelationship)`
`...`

Get Attribute Value `MtValue getValue(MtAttribute att)`
throws `MtException`

This method returns a copy of the attribute value. The returned value is an instance of `MtValue` containing:

- ◆ `value`: The value itself

- ◆ defaultP: The boolean indicating whether it is a default value or not
- ◆ dimension: The dimension for a list
- ◆ type: MATISSE datatype

Instance of	<code>bool isInstanceOf()</code>	This method indicates whether this object is an instance of the class <code>cl</code> or one of its subclasses.
Predefined Object	<code>bool isPredefinedObject()</code>	This function indicates whether this object is part of the initial meta-schema.
Lock	<code>lock(int lock) throws MtException</code>	This method allows acquiring a database lock for this object. The value is <code>MT.READ</code> or <code>MT.WRITE</code> .
Visit the Object Attributes	<code>MtPropertyCursor openMtAttributeCursor() throws MtException</code>	This method returns a cursor allowing you to iterate over all object's attributes.
Visit the Object Inverse Relationships	<code>MtObjectCursor openMtIRelationshipCursor() throws MtException</code>	This method returns a cursor allowing you to iterate over the inverse relationships for the object.
Visit the Object Relationships	<code>MtObjectCursor openMtRelationshipCursor() throws MtException</code>	This method returns a cursor allowing you to iterate over the relationships for the object.
Successors and Predecessors Streams	<code>MtObjectCursor openSuccessorsCursor(MtRelationship rel) MtObjectCursor openPredecessorsCursor(MtRelationship rel)</code>	
Remove Attribute Value	<code>removeValue(MtAttribute att) throws MtException __delitem__(MtAttribute att) throws MtException</code>	This method removes the current attribute value, therefore, the next time the value will be read, the attribute default value will be returned.
Removing Successors	<code>removeSuccessors(MtRelationship rel, List<MtObject> objs) __delitem__(MtRelationship rel)</code>	
Set Attribute Value	<code>setValue(MtAttribute att, Object value, int type=None, int rank=None, List dimensions) throws MtException __setitem__(MtAttribute att, Object value) throws MtException</code>	

This method updates the attribute value. It can be any kind of value type. [Table 4.2](#) lists the default Python to MATISSE mapping. These mappings can be overridden by specifying a type in `setValue()`.

Table 4.2 Default Python to MATISSE Mapping

Python Type	MATISSE Type
String	MT.STRING
Int	MT.INTEGER
Long	MT.INTEGER
Float	MT.DOUBLE

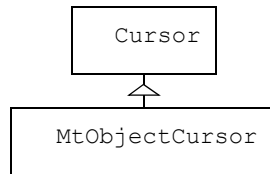
Stream Value Write	<pre>setListElements(MtAttribute att, int type, List value, int first=MT_CURRENT_OFFSET, bool discardAfter=1) ... </pre>
Remove an Object from the Database	<pre>remove() throws MtException</pre> <p>This method removes the object from the database.</p>
Print	<pre>__repr__() throws MtException</pre> <p>This method returns a string that describes a particular object (<class name>(<oid>)).</p>
Object Size	<pre>int size()</pre>

MtObjectCursor

Synopsis `module MATISSE`
 `class MtObjectCursor`

Inheritance This class inherits the Cursor interface.

Figure 4.1 MtObjectCursor Inheritance



Description The `MtObjectCursor` class allows you to visit a collection of MATISSE objects through a cursor mechanism.

The `openIndexCursor` and `openInstanceCursor` method defined on the `Database` class return such an object cursor.

Example To access all the instances of the class `Person`:

```
cl = db.getMtClass("Person")
cur = db.openInstanceCursor(cl)

aPerson = cur.next()
while aPerson:
    ....
    aPerson = cur.next()
cur.close()
```

Methods `close()` throws `MtException`

This method closes the cursor. The closed cursor cannot be used anymore.

CAUTION: The `close` method must be called when you do not intend to use the cursor anymore, or when you have visited all the items available through the cursor.

`MtObject next()` throws `MtException`

This method returns the next object available through the cursor. The `None` value is returned when the end of the cursor has been reached.

`skip(int num)` throws `MtException`

This method skips the `num` next objects available through the cursor.

SQLStatement

Synopsis `module MATISSE`
`class SQLStatement`

Description `SQLStatement` let you execute a SQL query and access the results.

Example Select all the columns for all the instances of `Person`.

```
stmt = SQLStatment()
stmt.execDirect("SELECT * FROM Person")
if stmt.getType() == MT.SQL_SELECT:
    cur = stmt.openStream()
    row = cur.nextRow()
    while row:
        for col in range(1, stmt.numResultCols() + 1):
            print "%-20s - %s" % (stmt.getColumnInfo(col)[1],
row[col - 1])
        row = cur.nextRow()
    cur.close()
```

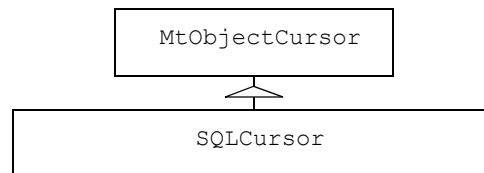
Free	<code>free()</code> <code>__del__()</code>	Frees the statement
Execute	<code>execDirect(string query)</code>	Executes query.
Open a Cursor	<code>SQLCursor open()</code>	Opens a cursor on the selection
Statement Info	<code>int getType()</code>	Returns either <code>SQL_ERROR</code> or the type of statement that has been successfully executed. Possible values are: <code>SQL_SELECT, SQL_SET_TRANSACTION, SQL_SET_OPTION, SQL_DROP_SELECTION, SQL_COMMIT, SQL_ROLLBACK, SQL_UPDATE, SQL_DELETE, SQL_INSERT, SQL_ALTER_ADD, SQL_ALTER_DROP, SQL_ALTER_ALTER, SQL_DROP, SQL_CREATE, SQL_METHOD, SQL_PROCEDURE, SQL_COMPILE, SQL_ERROR.</code>
	<code>string getInfo(int stmtAttribute)</code>	Returns additional information about the statement. You may refer to the MATISSE C API Reference for a detailed description.
Column Info	<code>int numResultCols()</code>	Returns the number of columns in the current result set.
	<code>list<(type,name)> getColumnInfo(int colnum)</code>	Returns a list of column description. A column description is a two-element list containing the type and the name of the column.
	example: <code>type, name = sel.getColumnInfo(3)</code>	Retrieves the type and the name of the third column.
Out Parameters	<code>any getParamValue(int param)</code>	Returns the resulting value after successful execution of a SQL method. Currently, only <code>SQL_RETVALUE</code> is allowed for <code>param</code> .
	<code>list getParamListElements(int param, int type, int num)</code> <code>list getParamDimensions(int param)</code>	Returns the resulting list value after successful execution of a SQL method.

SQLCursor

Synopsis `module MATISSE`
 `class SQLCursor`

Inheritance This class inherits from the class `MtObjectCursor`.

Figure 4.2 **SQLCursor Inheritance**



Description There are two ways to fetch elements from a `SQLCursor`: the tradition Object by Object with the `Next()` method from `MtObjectCursor`, or row by row where each row is a list of attribute values.

Methods `List nextRow()`

Initially sets the cursor to the first row, on subsequent calls advances to the next row in the result set. Returns a list of values that corresponds to the projection defined in the select list for the query.

```
list<(type,any)> getRowValue(int colnum)
```

Returns the type and value for the column `colnum` for the row at the current cursor position.

```
list getRowListElements(int colnum, int type, int numelts)
```

Returns a list value within a stream.

```
List setCurrent(int pos)
```

Sets the cursor to the row at position `pos`.

```
SQLStatement getSQLStatement()
```

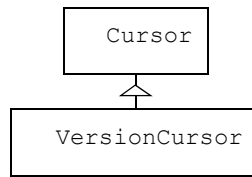
Returns the statement from which the cursor was opened.

VersionCursor

Synopsis `module MATISSE`
 `class VersionCursor`

Inheritance This class inherits from the abstract class `Database`.

Figure 4.3 VersionCursor Inheritance



Description The `VersionCursor` class allows you to iterate through the name of all the saved versions of the database.

The `openVersionCursor` method defined on `Database` class returns such a version cursor.

Methods `close()` throws `MtException`

This method closes the cursor. The closed cursor cannot be used anymore.

CAUTION: The `close` method must be called when you do not intend to use the cursor anymore, or when you have visited all the items available through the cursor.

`String next()` throws `MtException`

This method returns the next version available through the cursor. The `None` value is returned when the end of the cursor has been reached.

`skip(int num)` throws `MtException`

This method skips the `num` next names of version available through the cursor.

5 Meta-Schema Class Descriptions

These MATISSE meta-schema classes are defined in the `MATISSE` module.

MATISSE Objects

```
MtObject  
SchemaObject (UNDOCUMENTED)  
MtAttribute  
MtClass  
MtRelationship  
MtIndex  
MtEntryPointDictionary
```

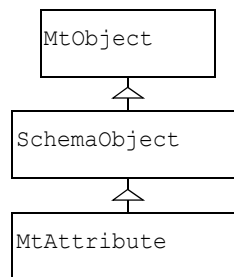
MtAttribute

Synopsis

```
module MATISSE  
class MtAttribute  
extends SchemaObject
```

Inheritance `MtAttribute` inherits from the abstract class `SchemaObject`.

Figure 5.1 MtAttribute Inheritance



Description This class handles descriptors for a MATISSE attribute and allows manipulation of the attribute descriptors. It also allows you to dynamically define or update attributes.

Example To define a new attribute descriptor:

```
# create an attribute  
newAtt = MtAttribute("Title")  
# link the attribute with the new class  
newClass["MtAttributes"] = (newAtt,)
```

Constructor `MtAttribute(String mtName)`

This constructor creates a new attribute descriptor into the database.

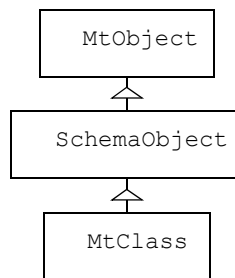
MtClass

Synopsis

```
module MATISSE
public class MtClass
extends SchemaObject
```

Inheritance MtClass inherits from the abstract class SchemaObject.

Figure 5.2 MtClass Inheritance



Description This class handles descriptors for a MATISSE class and allows manipulation of class descriptors. It also enables you to dynamically define or update classes.

Example To define a new class descriptor:

```
# Create new "GIFFile" class
newClass = MtClass("GIFFile")
```

Constructor

```
MtClass (String mtName)
throws MtException
```

This constructor creates a class descriptor in the database.

Class Description Methods

These methods provide access to the description of a class in the class hierarchy.

Get All Attribute Descriptors `List<MtAttribute> getAllMtAttributes() throws MtException`

This method returns all the attribute descriptors that are inherited and defined for the class.

Get All Relationship Descriptors `List<MtRelationship> getAllMtRelationships() throws MtException`

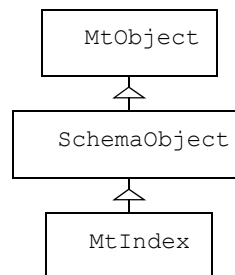
This method returns all the relationship descriptors that are inherited and defined for the class.

Get All Inverse Relationship Descriptors	<pre>List<MtRelationship> getAllMtInverseRelationships() throws MtException</pre> <p>This method returns all the inverse relationship descriptors that are inherited and defined for the class.</p>
Get All Super Classes	<pre>MtObjectVector getAllMtSuperclasses() throws MtException</pre> <p>This method returns all the superclasses of a class, including those inherited. The result is sorted from the bottom to the top of the class hierarchy.</p>
Get All Subclasses	<pre>MtObjectVector getAllMtSubclasses() throws MtException</pre> <p>This method returns all the subclasses of a class.</p>
Instances Number	<pre>int getInstanceNumber()</pre> <p>Returns the number of instances of this class and all its subclasses.</p>
Instance Cursors	<pre>MtObjectCursor openInstanceCursor() MtObjectCursor openOwnInstanceCursor()</pre>

MtIndex

Synopsis	<pre>module MATISSE public class MtIndex extends SchemaObject</pre>
Inheritance	MtIndex inherits the abstract class SchemaObject.

Figure 5.3 MtIndex Inheritance



Description This class handles a descriptor for a MATISSE index and allows manipulation of index descriptors. It also allows you to dynamically define or update indexes.

Example To define a new index descriptor:

```
# create an index
newIdx = MtIndex("PersonIdx")
```

```
# link the index with the class Person
newIdx["MtClasses"] = (personClass)
```

Constructor `MtIndex(String mtName)`

This constructor creates a new index descriptor in the database.

MtEntryPointDictionary

Synopsis

```
module MATISSE
public class MtEntryPointDictionary
extends SchemaObject
```

Inheritance `MtEntryPointDictionary` inherits the abstract class `SchemaObject`.

Description This class handles a descriptor for a MATISSE entry-point dictionary and allows manipulation of dictionary descriptors. It also allows you to dynamically define or update dictionaries.

Example To define a new dictionary descriptor:

```
# create an entry-point dictionary
newDict = MtEntryPointDictionary("PersonDict")
# link the dictionary with the class Person
newDict["MtClasses"] = (personClass)
```

Constructor `MtEntryPointDictionary(String mtName)`

This constructor creates a new entry-point dictionary descriptor in the database.

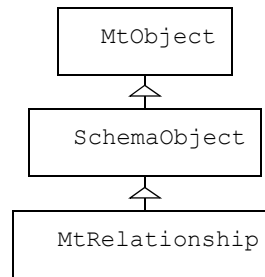
MtRelationship

Synopsis

```
module MATISSE
public class MtRelationship
extends SchemaObject
```

Inheritance `MtRelationship` inherits the abstract class `SchemaObject`.

Figure 5.4 MtRelationship Inheritance



Description This class handles descriptors for a MATISSE relationship and allows manipulation of relationship descriptors. It also allows you to dynamically define or update relationships.

Example To define a new relationship descriptor:

```
# create a relationship
newRel = MtRelationship("Siblings")
# link the relationship with the new class
newClass["MtRelationships"] = (newRel)
```

Constructor `MtRelationship(String mtName)`

This constructor creates an instance of `MtRelationship`. The relationship is created on the database.

6 MATISSE Exception Reference

All the exceptions specific to MATISSE inherit from the `MtException` exception.

`ConnectError`

This exception may be raised when you attempt to connect to a database and this operation is impossible, either because the name or the host of the database is not correct, or because the server and client version are incompatible, or because the user/password specification is not correct, etc.

`ConnectionLost`

This exception can occur at any time if the connection to the database has been lost.

`CommitLockNotObtained`

This exception may be raised only in a specific context at commit time. The database is in a state where all the objects have been checked, and most of the changes have been written. Some specific actions do not get the lock they need. There are only two possible actions when this exception is raised: either abort the transaction, or commit again.

`DataDefinitionException`

This exception may be raised when you have modified a schema object in the transaction which does not allow `DATA_DEFINITION` (or while you are modifying it).

`ForbiddenSchemaOperation`

This exception is raised when you attempt to modify a schema object.

`InexistentObject`

The object which is modified or read no longer exists.

`InvalidArgument`

One of the arguments of the function is incorrect in this context.

InvalidObject

This exception may be raised at commit time, when an object does not fulfill its constraints. Note that this exception is raised when a C function returns a `MATISSE_USERERROR` status, even if it is not at commit time. This will not occur if you have no C service function.

InvalidOperation

This error may occur when you attempt an operation that is impossible in this context.

InvalidValue

This exception is raised when a value is invalid in this context. It may be that you try to add a successor which is already a successor from the given relationship, or that you have specified the name of a version that does not exist, etc.

LockNotObtained

This error will be raised when you try to get an implicit or an explicit lock, and you were not able to get it.

MemoryFault

This error may occur when MATISSE has tried to allocate memory for its own use and did not succeed.

MtException

The superclass of all the MATISSE exceptions. This class has a method:

```
long getStatus()
```

This method returns the MATISSE status. You can compare it with the statuses specified in the `MtClass`. Note that the status can be 0, when the exception has been thrown by the binding rather than by MATISSE itself.

StreamClosedByDBA

This exception occurs during a cursor operation if the database administrator has closed the corresponding stream in the database.

TransactionAborted

This exception will be raised when the transaction is aborted, either due to a deadlock, or because the database administrator has explicitly aborted.

TransactionAbortedInTrigger

This exception may be raised if the C function for a trigger returns an error status.

TransactionError

This exception is raised when you attempt to read an object outside the scope of a transaction or a version access, or when you attempt to modify an object outside the scope of a transaction, or when you try to start a transaction or a version access when another one has already started.

UnexpectedException

An error status has been returned by a MATISSE function, but should not in this context. Call your favorite MATISSE support.

Index

Symbols

>rollback
 Database 22
__cmp__ 25

C

close
 Database 19
 MtObjectCursor 29
 VersionCursor 32
commit
 Database 19
CommitLockNotObtained 38
ConnectError 38
ConnectionLost 38
constructor
 Database 19
 MtAttribute 33
 MtClass 34
 MtEntryPointDictionary 36
 MtIndex 36
 MtRelationship 37

D

Database 19–23
 close 19
 commit 19
 constructor 19
 endVersionAccess 20
 getHost 20
 getInstanceNumber 22
 getMtAttribute 20
 getMtClass 20
 getMtEntryPointDictionary 20
 getMtRelationship 20
 getName 20
 getPriority 20

lockObjectsFromEP 20
lookup 20
open 21
openEntryPointCursor 21
openIndexCursor 21
openInstanceCursor 22
openVersionCursor 22
repr 23
rollback 22
select 22
setWaitTime 22
startTransaction 22
transactionIsInProgress 23
unselect 23
versionAccessIsInProgress 23
DataDefinitionException 38

E

endVersionAccess
 Database 20

F

fetch
 MtObject 26
ForbiddenSchemaOperation 38

G

getHost
 Database 20
getInstanceNumber
 Database 22
getMtAttribute
 Database 20
getMtClass
 Database 20
MtObject 26

- getMtKey 26
- getMtRelationship
 - Database 20
- getName
 - Database 20
- getObjectFromMtKey 21
- getPriorityDatabase 20
- getValue
 - MtObject 26

I

- InexistentObject 38
- InvalidArgument 38
- InvalidObject 39
- InvalidOperation 39
- InvalidValue 39

L

- lock
 - MtObject 27
- LockNotObtained 39
- lockObjectsFromEP
 - Database 20
- lookup
 - Database 20

M

- MemoryFault 39
- meta-schema class 33**
 - MtAttribute 33
 - MtClass 34
 - MtIndex 35, 36
- MT 23–24
- MtAttribute
 - constructor 33**
- MtClass
 - constructor 34**
- MtEntryPointDictionary
 - constructor 36**

- MtException 39
- MtIndex
 - constructor 36**
- MtObject 25–28
 - `__cmp__` 25
 - fetch 26
 - getMtClass 26
 - getValue 26
 - lock 27
 - openAttributeCursor 27
 - openIRelationshipCursor 27
 - openRelationshipCursor 27
 - remove 28
 - repr 28
- MtObjectCursor 28–29
 - close 29
 - next 29
 - skip 29
- MtRelationship
 - constructor 37**

N

- next
 - MtObjectCursor 29
 - VersionCursor 32

O

- open
 - Database 21
- openAttributeCursor
 - MtObject 27
- openIndexCursor
 - Database 21
- openInstanceCursor
 - Database 22
- openIRelationshipCursor
 - MtObject 27
- openRelationshipCursor
 - MtObject 27

operator 25

skip 32

R

remove

 MtObject 28

remove attribute value 27

repr

 Database 23

 MtObject 28

S

setWaitTime

 Database 22

skip

 MtObjectCursor 29

 VersionCursor 32

SQLCursor 31

SQLSelection 29–??

startTransaction

 Database 22

StreamClosedByDBA 39

T

TransactionAborted 39

TransactionAbortedInTrigger 40

TransactionError 40

transactionIsInProgress

 Database 23

U

UnexpectedException 40

unselect

 Database 23

V

versionAccessIsInProgress

 Database 23

VersionCursor 31–32

 close 32

 next 32