

Matisse[®] Python Programmer's Guide

February 2012



MATISSE Python Programmer's Guide

Copyright ©1992–2012 Matisse Software Inc. All Rights Reserved.

This manual and the software described in it are 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 13 February 2012

| | | |
|----------|--|-----------|
| 1 | Introduction | 5 |
| | Scope of This Document | 5 |
| | Before Reading This Document | 5 |
| | Before Running the Examples | 5 |
| 2 | Connection and Transaction | 7 |
| | Building the Examples | 7 |
| | Read Write Transaction | 7 |
| | Read-Only Access | 8 |
| | Version Access | 8 |
| | Specific Options | 9 |
| | More about MtDatabase | 11 |
| 3 | Working with Objects and Values | 12 |
| | Running the Examples on Objects | 12 |
| | Creating Objects | 12 |
| | Listing Objects | 14 |
| | Deleting Objects | 15 |
| | Comparing Objects | 16 |
| | Running the Examples on Values | 16 |
| | Setting and Getting Values | 16 |
| | Removing Values | 18 |
| | Streaming Values | 18 |
| | Retrieving an Object from its Oid | 19 |
| 4 | Working with Relationships | 20 |
| | Running the Examples on Relationships | 20 |
| | Setting and Getting Relationship Elements | 20 |
| | Adding and Removing Relationship Elements | 21 |
| | Listing Relationship Elements | 21 |
| | Counting Relationship Elements | 22 |
| 5 | Working with Indexes | 23 |
| | Running the Examples on Indexes | 23 |
| | Index Lookup | 23 |
| | Index Lookup Count | 24 |
| | Index Entries Count | 24 |
| 6 | Working with Entry-Point Dictionaries | 25 |
| | Running the Examples on Dictionaries | 25 |
| | Entry-Point Dictionary Lookup | 25 |
| | Entry-Point Dictionary Lookup Count | 26 |
| 7 | Working with SQL | 27 |
| | Running the Examples on SQL | 27 |
| | Executing a SQL Statement | 27 |
| | Creating Objects | 28 |

| | |
|--|-----------|
| Updating Objects | 28 |
| Retrieving Values | 29 |
| Retrieving Objects from a SELECT statement | 30 |
| Retrieving Objects from a Block Statement | 31 |
| Executing DDL Statements | 32 |
| Executing SQL Methods | 33 |
| Deleting Objects | 36 |
| 8 Working with Class Reflection | 37 |
| Running the Examples on Reflection | 37 |
| Creating Objects | 37 |
| Listing Objects | 38 |
| Working with Indexes | 39 |
| Working with Entry Point Dictionaries | 40 |
| Discovering Object Properties | 41 |
| Adding Classes | 42 |
| Deleting Objects | 42 |
| Removing Classes | 43 |
| 9 Working with Database Events | 44 |
| Running the Events Example | 44 |
| Events Subscription | 44 |
| Events Notification | 45 |
| More about MtEvent | 46 |
| 10 Handling Object Factories | 47 |
| Connection with Factory | 47 |
| Creating your Object Factory | 47 |
| 11 Building your Application | 50 |
| Discovering the Matisse Python Classes | 50 |
| Generating Stub Classes | 50 |
| Extending the generated Stub Classes | 50 |
| Generated Public Methods | 51 |

1 Introduction

Scope of This Document

This document is intended to help Python programmers learn the aspects of Matisse design and programming that are unique to the Matisse Python binding.

Aspects of Matisse programming that the Python binding shares with other interfaces, such as basic concepts and schema design, are covered in *Getting Started with Matisse*.

Future releases of this document will add more advanced topics. If there is anything you would like to see added, or if you have any questions about or corrections to this document, please send e-mail to support@matisse.com.

Before Reading This Document

Throughout this document, we presume that you already know the basics of Python programming and either relational or object-oriented database design, and that you have read the relevant sections of *Getting Started with Matisse*.

Before Running the Examples

Before running the following examples, you must do the following:

- Install Matisse 9.0.0 or later.
- Install the Python version 2.7 or later for your operating system (a free download from www.python.org).
- Download and extract the Matisse Python binding source code and sample code from the Matisse Web site:

<http://www.matisse.com/developers/documentation/>

The sample code files are grouped in subdirectories by chapter number. For example, the code snippets from the following chapter are in the `chap_2` directory.

- Build the Matisse Python binding from the source code. Follow the building instructions as detailed in the `BUILD` file.
- Create and initialize a database. You can simply start the Matisse Enterprise Manager, select the database 'example' and right click on 'Re-Initialize'.
- From a Unix shell prompt or on MS Windows from a 'Command Prompt' window, change to the `chap_x` subdirectory in the directory where you installed the examples.

- If applicable, load the ODL file into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema'. For example you may import chap_3/objects.odl for the Chapter 3 demo.

- Generate Python class files:

```
mt_sdl stubgen -lang python objects.odl
```

- Run the application. For instance in chap_3:

```
python -i createObjects.py
```

2 Connection and Transaction

All interaction between client Python applications and Matisse databases takes place within the context of transactions (either explicit or implicit) established by database connections, which are transient instances of the `MtDatabase` class. Once the connection is established, your Python application may interact with the database using the schema-specific methods generated by `mt_sdl`. The following sample code shows a variety of ways of connecting with a Matisse database.

Note that in this chapter there is no ODL file as you do not need to create an application schema.

Building the Examples

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_2` directory in your installation (under `examples`).
3. Launch the application:

```
Windows:
python -i connect.py

UNIX:
python -i connect.py
```

Read Write Transaction

The following code connects to a database, starts a transaction, commits the transaction, and closes the connection:

```
try:
    db = MtDatabase(dbhost, dbname, MtDynamicObjectFactory())
    db.open()
    db.startTransaction()
    print "Connection and read-write access to {}".format(str(db))
    # db.commit()
    db.rollback()
    db.close()
except MtException as mtex:
    print "MtException: " + str(mtex)
    code, msg = mtex
    print "MtException.code: " + str(code)
    print "MtException.message: " + msg
```

1. Launch the application:

```
Windows:
python -i connect.py

UNIX:
python -i connect.py
```

Read-Only Access

The following code connects to a database in read-only mode, suitable for reports:

```
try:
    db = MtDatabase(dbhost, dbname, MtDynamicObjectFactory())
    db.open()
    db.startVersionAccess()
    print "Connection and read-only access to {}".format(str(db))
    db.endVersionAccess()
    db.close()
except MtException as mtex:
    print "MtException error:"
    print mtex
```

1. Launch the application:

```
Windows:
python -i versionConnect.py
```

```
UNIX:
python -i versionConnect.py
```

Version Access

The following code illustrates methods of accessing various versions of a database.

```
def listVersions(db):
    for ver in db.versionIterator():
        vertime = db.getVersionFromName(ver)
        print "- {} ({}).format(ver, vertime)

def removeVersions(db):
    itr = db.versionIterator()
    for ver in itr:
        vertime = db.getVersionFromName(ver)
        print "- {} ({}).format(ver, vertime)
        db.removeVersion(ver)

def versionNavigation(dbhost, dbname):
    print "Matisse Python Version {}.{}.{}.{}\n".format(MtDatabase.getMajorVersion(),
                                                         MtDatabase.getMinorVersion(),
                                                         MtDatabase.getReleaseVersion(),
                                                         MtDatabase.getPatchVersion())

    try:
        db = MtDatabase(dbhost, dbname, MtDynamicObjectFactory())
        db.open()
        print "Current version: {}\n".format(db.getCurrentVersion())

        db.startTransaction()
        print "Version list before regular commit:"
        listVersions(db)
        db.commit()
```

```

db.startTransaction()
print "Version list after regular commit:"
listVersions(db)
versionName = db.commit("mysnapshot-")

db.startVersionAccess()
print "Version list after named commit:"
listVersions(db)
db.endVersionAccess()

db.startVersionAccess(versionName)
print "Successful access to version: {}".format(versionName)
db.endVersionAccess()

db.startTransaction()
print "Remove all versions:"
removeVersions(db)
db.commit()

db.close()
except MtException as mtex:
    print "MtException error:"
    print mtex

```

1. Launch the application:

```

Windows:
python -i versionNavigation.py

```

```

UNIX:
python -i versionNavigation.py

```

Specific Options

This example shows how to enable the local client-server memory transport and to set or read various connection options and states.

```

def startAccess(db, readonly):
    if readonly:
        db.startVersionAccess()
        print "read-only access to {}".format(str(db))
    else:
        db.startTransaction()
        print "read-write access to {}".format(str(db))

def endAccess(db):
    if db.isVersionAccessInProgress():
        db.endVersionAccess()
        print "version access to {} ended".format(str(db))
    elif db.isTransactionInProgress():
        db.commit()
        print "transaction on {} committed".format(str(db))
    else:
        print "No transaction nor version access in progress for {}".format(str(db))

```

```

def isReadOnlyAccess(db):
    return (db.getOption(MtDatabase.DATA_ACCESS_MODE) == MtDatabase.DATA_READONLY)

def setAccessMode(db,mode):
    if mode == "T":
        db.setOption(MtDatabase.DATA_ACCESS_MODE, MtDatabase.DATA_MODIFICATION)
        print "DATA_MODIFICATION (read-write transaction) mode"
    elif mode == "v":
        db.setOption(MtDatabase.DATA_ACCESS_MODE, MtDatabase.DATA_READONLY)
        print "DATA_READONLY (version) mode"
    elif mode == "S":
        db.setOption(MtDatabase.DATA_ACCESS_MODE, MtDatabase.DATA_DEFINITION)
        print "DATA_DEFINITION (schema definition) mode"
    else:
        print "unknown mode"

def advancedConnect(dbhost,dbname,mode):
    print "Matisse Python Version {}.{}.{}.{}\n".format(MtDatabase.getMajorVersion(),
                                                       MtDatabase.getMinorVersion(),
                                                       MtDatabase.getReleaseVersion(),
                                                       MtDatabase.getPatchVersion())

    try:
        db = MtDatabase(dbhost,dbname,MtDynamicObjectFactory())
        db.open()

        if db.isConnectionOpen():
            setAccessMode(db, mode)

            startAccess(db, isReadOnlyAccess(db))

            print "\ndo something...\n"

            endAccess(db)

        print "Test Completed"

        db.close()
    except MtException as mtex:
        print "MtException error:"
        print mtex

```

1. Launch the application:

Windows:
python -i advancedConnect.py

UNIX:
python -i advancedConnect.py

More about MtDatabase

As illustrated by the previous sections, the `MtDatabase` class provides all the methods for database connections and transactions. The reference documentation for the `MtDatabase` class is included in the Matisse Python Binding API documentation located from the Matisse Python binding installation root directory in `docs/python/api/matisse.html`.

3 Working with Objects and Values

This chapter explains how to manipulate object with the object interface of the Matisse Python binding. The object interface allows you to directly retrieve objects from the Matisse database without Object-Relational mapping, navigate from one object to another through the relationship defined between them, and update properties of objects without writing SQL statements.

The object interface can be used with Matisse Python SQL interface as well. For example, you can retrieve objects with SQL, then use the object interface to navigate to other objects from these objects, or update properties of these objects using the accessor methods defined on these classes.

Running the Examples on Objects

This sample program creates objects from 2 classes (`Person` and `Employee`), lists all `Person` objects (which includes both objects, since `Employee` is a subclass of `Person`), deletes objects, then lists all `Person` objects again to show the deletion. Note that because `FirstName` and `LastName` are not nullable, they *must* be set when creating an object.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_3` directory in your installation (under `examples`).
3. Load `objects.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_3/objects.odl` for this demo.
4. Generate Python class files:

```
mt_sdl stubgen -lang python objects.odl
```

Creating Objects

This section illustrates the creation of objects. The stubclass provides a default constructor which is the base factory for creating persistent objects.

```
@staticmethod
def createPerson(db):
    """
    Default constructor provided as an example.
    You may delete this constructor or modify it to suit your needs. If you
    modify it, please revise this comment accordingly.
    @param MtDatabase db a database
    @return Person a new instance of Person
    """
    return Person(Person.getClass(db))
```

You can also use the default constructor defined on the `MtObject` class.

```
def __init__(self, entry, mtdb = None):
    """
    With a class object parameter, creates a new persistent Matisse instance.
    This constructor is generally used only by generated stubs.
```

With entry as an int, constructs a `MtObject` bound to an existing object. This constructor is generally used for internal purposes only.

```
@param MtClass cls the class to instantiate
@param int mtOid an existing object ID; no check is performed on the OID's validity
@param MtDatabase db the object's database
"""
if isinstance(entry, MtClass):
    self.mtoid = _matisse_py.create_object(entry.getMtDatabase().getHandle(),
entry.getMtOid())
    self.mtodb = entry.getMtDatabase()
else:
    self.mtoid = int(entry)
    self.mtodb = mtodb

# Create a new Person object (instance of class Person)
# use the dynamic object factory
p = Person.createPerson(db)
p.setFirstName("John")
p.setLastName("Smith")
p.setAge(42)

a = PostalAddress(PostalAddress.getClass(db))
a.setCity("Portland")
a.setPostalCode("97201")
p.setAddress(a)
print "Person John Smith created.\n"

# Create a new Employee object
e = Employee(Employee.getClass(db))
e.setFirstName("Jane")
e.setLastName("Jones")
# Age is nullable we can leave it unset
e.setHireDate(date(2009,11,8))
# numeric datatype
e.setSalary(str(85000.00))
print "Employee Jane Jones created.\n"
```

1. Launch the application:

Windows:
python -i createObjects.py

UNIX:
python -i createObjects.py

If your application need to create a large number of objects all at once, we recommend that you use the `preallocate()` method defined on `MtDatabase` which provide a substantial performance optimization.

```
db.startTransaction()

# Optimize the objects loading
# Preallocate OIDs so objects can be created in the client workspace
# without requesting any further information from the server
db.preallocate(DEFAULT_ALLOCATOR_CNT)

for i in range(SAMPLE_OBJECT_CNT):
```

```

# Create a new Employee object
e = Employee(Employee.getClass(db))
fname = fNameSample[random.randint(0,12345) % MAX_SAMPLES]
lname = lNameSample[random.randint(0,12345) % MAX_SAMPLES]
e.setFirstName(fname)
e.setLastName(lname)
hyear = (2000+(random.randint(0,12345) % MAX_SAMPLES))
e.setHireDate(date(hyear,6,1))
salary = salarySample[random.randint(0,12345) % MAX_SAMPLES]
e.setSalary(salary)

a = PostalAddress(PostalAddress.getClass(db))
addrIdx = random.randint(0,12345) % MAX_SAMPLES
a.setCity(addressSample[addrIdx][0])
a.setPostalCode(addressSample[addrIdx][1])
e.setAddress(a)

print "Employee {} {} {} salary={ } hdate={ } created.".format((i+1),fname,lname,
                                                                e.getSalary(),
                                                                e.getHireDate())

if (i % OBJECT_PER_TRAN_CNT == 0):
    db.commit()
    db.startTransaction()

# check the remaining number of preallocated objects.
if (db.numPreallocated() < 2):
    db.preallocate(DEFAULT_ALLOCATOR_CNT)

# createion completed - commit last transaction
if (db.isTransactionInProgress()):
    db.commit()

```

1. Launch the application:

Windows:
python -i loadObjects.py

UNIX:
python -i loadObjects.py

Listing Objects

This section illustrates the enumeration of objects from a class. The `instanceIterator()` static method defined on a generated subclass allows you to enumerate the instances of this class and its subclasses. The `getInstanceNumber()` method returns the number of instances of this class.

```

# List all Person objects
print str(Person.getInstanceNumber(db)) + " Person(s) in the database."
print str(PostalAddress.getInstanceNumber(db)) + " Address(s) in the database."
itr = Person.instanceIterator(db)
for x in itr:
    location = "???"
    if (x.getAddress() != None): location = x.getAddress().getCity()
    print "- {} {} from {} is a {}".format(x.getFirstName(),
                                          x.getLastName(),
                                          location,

```

```
x.getMtClass().getMtName()
```

1. Launch the application:

```
Windows:
python -i listObjects.py
```

```
UNIX:
python -i listObjects.py
```

The `ownInstanceIterator()` static method allows you to enumerate the own instances of a class (excluding its subclasses). The `getOwnInstanceNumber()` method returns the number of instances of a class (excluding its subclasses).

```
# List all Person objects
print str(Person.getOwnInstanceNumber(db)) + " Person(s) (excluding subclasses)
in the database."

itr = Person.ownInstanceIterator(db)
for x in itr:
    location = "???"
    if (x.getAddress() != None): location = x.getAddress().getCity()
    print "- {} {} from {} is a {}".format(x.getFirstName(),
                                          x.getLastName(),
                                          location,
                                          x.getMtClass().getMtName())
```

1. Launch the application:

```
Windows:
python -i listOwnInstances.py
```

```
UNIX:
python -i listOwnInstances.py
```

Deleting Objects

This section illustrates the removal of objects. The `remove()` method delete an object.

```
# Remove created objects
...
# NOTE: does not remove the object sub-parts
p.remove()
```

To remove an object and its sub-parts, you need to override the `deepRemove()` method in the subclass to meet your application needs. For example the implementation of `deepRemove()` in the `Person` class that contains a reference to a `PostalAddress` object is as follows:

```
def deepRemove(self):
    """
    Overrides MtObject.deepRemove() to remove the Address object if any.
    """
    pAddr = self.getAddress()
    if (pAddr != None):
        pAddr.deepRemove()
```

```

    super(Person, self).deepRemove()

    ...
    p.deepRemove()

```

1. Launch the application:

```

Windows:
python -i deleteObjects.py

UNIX:
python -i deleteObjects.py

```

The `removeAllInstances()` method defined on `MtClass` delete all the instances of a class.

```

Person.getClass(db).removeAllInstances()

```

1. Launch the application:

```

Windows:
python -i deleteAllObjects.py

UNIX:
python -i deleteAllObjects.py

```

Comparing Objects

This section illustrates how to compare objects. Persistent objects must be compared with the `==` method. You can't compare persistent object with the `'is'` operator.

```

...
if (p1 == p2):
    print "Same objects"

```

Running the Examples on Values

This example shows how to get and set values for various Matisse data types including Null values, and how to check if a property of an object is a Null value or not.

This example uses the database created for `Objects Example`. It creates objects, then manipulates its values in various ways.

Setting and Getting Values

This section illustrates the set, update and read object property values. The stubclass provides a set and a get method for each property defined in the class.

```

# Create a new Employee object
e = Employee.createEmployee(db)
e.setComment("FirstName, LastName, Age, HireDate & Salary Set")
e.setFirstName("John")

```

```

e.setLastName("Jones")

# Setting numbers
# Age is nullable we can leave it unset
e.setAge(42)

# Setting Date (use date and datetime for Date and Timestamp)
e.setHireDate(date(2009,11,8))

# Setting Numeric (int, double or string)
# numeric datatype is managed as string
# since not since not native to python
e.setSalary("85000.00")
e.setSalary(85000.00)
e.setSalary(85000)

print "Setting Age to null..."
e.setNull(Employee.getAgeAttribute(db))

```

1. Launch the application:

Windows:

```
python -i setObjectValues.py
```

UNIX:

```
python -i setObjectValues.py
```

```

# Getting String values
print "Comment: " + str(e.getComment())
print "- {} {} is a {}".format(e.getFirstName(),
                               e.getLastName(),
                               e.getMtClass().getMtName())
# suppresses output if no value set
if ( not e.isAgeNull() ):
    print " {} years old".format(e.getAge())
# Getting number values
print " Number of dependents: " + str(e.getDependents())
# Getting numeric values - returned as a string
print " Salary: " + str(e.getSalary())
# Getting date values - returned as a date
print " Hiring Date: " + str(e.getHireDate())

```

1. Launch the application:

Windows:

```
python -i getObjectValues.py
```

UNIX:

```
python -i getObjectValues.py
```

Removing Values

This section illustrates the removal of object property values. Removing the value of an attribute will return the attribute to its default value.

```
# Removing value returns attribute to default
e.removeAge()

# suppresses output if no value set
if ( not e.isAgeNull() ):
    print " {} years old".format(e.getAge())
else:
    if ( e.isAgeDefaultValue() ):
        print " Age: null (default value)"
    else:
        print " Age: null"
```

1. Launch the application:

```
Windows:
python -i removeObjectValues.py

UNIX:
python -i removeObjectValues.py
```

Streaming Values

This section illustrates the streaming of blob-type values (MT_BYTES, MT_AUDIO, MT_IMAGE, MT_VIDEO). The subclass provides streaming methods (setPhotoElements(), getPhotoElements()) for each blob-type property defined in the class. It also provides a method (getPhotoSize()) to retrieve the blob size without reading it.

```
# Store Image using a buffer stream
print "Storing an image from a stream of fixed size buffer."
buf2 = ""
e.setPhotoElements(buf2, MtType.BEGIN_OFFSET, 0, True)

f = open('matisse.gif', "rb")
buf2 = f.read(512)
while buf2 != "":
    buflen = len(buf2)
    e.setPhotoElements(buf2, MtType.CURRENT_OFFSET, buflen, False)
    buf2 = f.read(512)

f.close()

print "Image of {} bytes stored.".format(e.getPhotoSize())

print "Streaming an image out"
# Getting blobs (save value of e.Photo as out.gif in the
# program directory)
```

```

fo = open('imageout2.gif', "wb+")
buf2out = e.getPhotoElements(MtType.BEGIN_OFFSET, 512)
while buf2out != "":
    fo.write(buf2out)
    if (len(buf2out) < 512): break
    buf2out = e.getPhotoElements(MtType.CURRENT_OFFSET, 512)

fo.close()

```

1. Launch the application:

Windows:
python -i readWriteStreamingValues.py

UNIX:
python -i readWriteStreamingValues.py

Retrieving an Object from its Oid

This section illustrates a very commonly used feature in the binding. Using the Object Identifier (OID) is very efficient for retrieving one object from the database. The example below illustrates how to view an image stored into the database using the object Identifier to quickly retrieve the object.

```

db = MtDatabase(dbhost, dbname, MtDynamicObjectFactory())
db.open()
db.startVersionAccess()
print "Retrieve a Person object from its oid {}".format(photoid)
p = db.upcast(photoid)
db.endVersionAccess()
db.close()

```

4 Working with Relationships

One of the major advantages of the object interface of the Matisse Python binding is the ability to navigate from one object to another through a relationship defined between them. Relationship navigation is as easy as accessing an object property.

Running the Examples on Relationships

This example creates several objects, then manipulates the relationships among them in various ways.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_4` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_4/examples.odl` for this demo.
4. Generate Python class files:

```
mt_sdl stubgen -lang python examples.odl
```

Setting and Getting Relationship Elements

This section illustrates the set, update and get object relationship values. The stubclass provides a set and a get method for each relationship defined in the class.

```
m1 = Manager.createManager(db)
...
# Set a relationship
# Need to report to someone since the relationship
# cardinality minimum is set to 1
m1.setReportsTo(m1)
...
m2 = Manager.createManager(db)
...
# Set a relationship
m2.setReportsTo(m1)
...

e = Employee.createEmployee(db)
...
# Set a relationship
e.setReportsTo(m2)

# Set a relationship
m1.setAssistant(e)
# Set a relationship
m2.setAssistant(e)
...
c1 = Person.createPerson(db)
...
c2 = Person.createPerson(db)
```

```

...
# Set successors
children = (c1, c2)
m2.setChildren(children)

# Get all successors
children = m2.getChildren()

```

1. Launch the application:

```

Windows:
python -i setRelationships.py

```

```

UNIX:
python -i setRelationships.py

```

Adding and Removing Relationship Elements

This section illustrates the adding and removing of relationship elements. The stubclass provides a `append`, a `remove` and a `clear` method for each relationship defined in the class.

```

c2 = Person.createPerson(db)
c3 = Person.createPerson(db)
...
# add one successor
m2.appendChildren(c2)
# add multiple successors
m2.appendChildren( (c3, ) )
...
# removing successors (this only breaks links, it does not
# remove objects)
m2.removeChildren( (c2, ) )
# removing one successor
m2.removeChildren(c3)

# clearing all successors (this only breaks links, it does
# not remove objects)
m2.clearChildren()

```

1. Launch the application:

```

Windows:
python -i addToRelationship.py
python -i removeFromRelationship.py

```

```

UNIX:
python -i addToRelationship.py
python -i removeFromRelationship.py

```

Listing Relationship Elements

This section illustrates the listing of relationship elements for one-to-many relationships. The stubclass provides an iterator method for each one-to-many relationship defined in the class.

```
# Iterate when the relationship is large is always more efficient
for p in m2.childrenIterator():
    print "    " + p.getFirstName()
```

1. Launch the application:

```
Windows:
python -i iterateRelationship.py

UNIX:
python -i iterateRelationship.py
```

Counting Relationship Elements

This section illustrates the counting of relationship elements for one-to-many relationships. The stubclass provides an get size method for each one-to-many relationship defined in the class.

```
# Get the relationship size without loading the Python objects
# which is the fast way to get the size
childrenCnt = m2.getChildrenSize()

print "Relationship size without loading:"
print "    {} has {} kid(s).\n".format(m2.getFirstName(), childrenCnt)

# an alternative to get the relationship size
# but the Python objects are loaded before you can get the count
childrenCnt = len(m2.getChildren())

print "Relationship size from the loaded array:"
print "    {} has {} kid(s).\n".format(m2.getFirstName(), childrenCnt)
```

1. Launch the application:

```
Windows:
python -i getRelationshipSize.py

UNIX:
python -i getRelationshipSize.py
```

5 Working with Indexes

While indexes are used mostly by the SQL query optimizer to speed up queries, the Matisse Python binding also provides the index query APIs to look up objects based on a key value(s). The stubclass defines both lookup methods and iterator methods for each index defined on the class.

Running the Examples on Indexes

Using the `PersonName` index, it checks whether the database contains an entry for a person matching the specified name. The application will list the names in the database, indicate whether the specified name was found, and return results within a sample range (defined in the source) using an iterator.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_5` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_5/examples.odl` for this demo.
4. Generate Python class files:

```
mt_sdl stubgen -lang python examples.odl
```

Index Lookup

This section illustrates retrieving objects from an index. The stubclass provides a lookup and a iterator method for each index defined on the class.

```
// the lookup function returns null to represent no match
found = Person.lookupPersonName(db, lastName, firstName)
```

1. Launch the application:

```
Windows:
python -i lookupObjects.py
```

```
UNIX:
python -i lookupObjects.py
```

```
# open an iterator for a specific range
fromFirstName = "Fred"
toFirstName = "John"
fromLastName = "Jones"
toLastName = "Murray"
print "\nLookup from \"{} {}\" to \"{}
{}\"".format(fromFirstName, fromLastName, toFirstName, toLastName)
```

```
itr = Person.personNameIterator(db, fromLastName, fromFirstName, toLastName,
toFirstName)
```

```
print "\nFound with no class filter:"
for p in itr:
    print "  {} {}".format(p.getFirstName(),p.getLastName())
```

1. Launch the application:

```
Windows:
python -i iterateIndex.py

UNIX:
python -i iterateIndex.py
```

Index Lookup Count

This section illustrates retrieving the object count for a matching index key. The `getObjectNumber()` method is defined on the `MtIndex` class.

```
$key = array( $lastName, $firstName ) ;
$count = Person::getPersonNameIndex($db)->getObjectNumber($key);
print "{$count} objects retrieved\n";
```

1. Launch the application:

```
Windows:
python -i lookupObjectsCount.py

UNIX:
python -i lookupObjectsCount.py
```

Index Entries Count

This section illustrates retrieving the number of entries in an index. The `getIndexEntriesNumber()` method is defined on the `MtIndex` class.

```
key = ( lastName, firstName )
count = Person.getPersonNameIndex(db).getObjectNumber(key)
print str(count) + " objects retrieved\n"
```

1. Launch the application:

```
Windows:
python -i countIndexEntries.py

UNIX:
python -i countIndexEntries.py
```

6 Working with Entry-Point Dictionaries

An entry-point dictionary is an indexing structure containing keywords derived from a value, which is especially useful for full-text indexing. While the entry-point dictionary can be used with SQL query using `ENTRY_POINT` keyword, the object interface of the Matisse Python binding also provides APIs to directly retrieve objects using the entry-point dictionaries.

Running the Examples on Dictionaries

Using the `commentDict` entry-point dictionary, the example retrieves the `Person` objects in the database with `Comments` fields containing a specified character string.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_6` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_6/examples.odl` for this demo.
4. Generate Python class files:

```
mt_sdl stubgen -lang python examples.odl
```

Entry-Point Dictionary Lookup

This section illustrates retrieving objects from an entry-point dictionary. The stubclass provides access to lookup methods and iterator methods for each entry-point dictionary defined on the class.

```
# the lookup function returns null to represent no match
# if more than one match an exception is raised
found = Person.getCommentDictDictionary(db).lookup(searchstring)
```

1. Launch the application:

```
Windows:
python -i lookupObjects.py
```

```
UNIX:
python -i lookupObjects.py
```

```
hits = 0

itr = Person.commentDictIterator(db, searchstring)
for p in itr:
    print " {} {}".format(p.getFirstName(), p.getLastName())
    hits += 1
print "{} Person(s) with 'comment' containing '{}'\n".format(hits, searchstring)
```

1. Launch the application:

```
Windows:  
python -i iterateEpDict.py
```

```
UNIX:  
python -i iterateEpDict.py
```

Entry-Point Dictionary Lookup Count

This section illustrates retrieving the object count for a matching entry-point key. The `getObjectNumber()` method is defined on the `MtEntryPointDictionary` class.

```
count = Person.getCommentDictDictionary(db).getObjectNumber(searchstring)  
print str(count) + " matching object(s) retrieved\n"
```

1. Launch the application:

```
Windows:  
python -i lookupObjectsCount.py
```

```
UNIX:  
python -i lookupObjectsCount.py
```

7 Working with SQL

Running the Examples on SQL

This sample program demonstrates how to manipulate objects via the Matisse Python SQL interface. It creates objects (`Person`, `Employee` and `Manager`) and it executes `SELECT` statements to retrieve objects. It also shows how to create SQL methods and execute them.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `SQL` directory in your installation (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `sql/examples.odl` for this demo.
4. Generate Python class files:

```
mt_sdl stubgen -lang python examples.odl
```

Executing a SQL Statement

After you open a connection to a Matisse database, you can execute statements (i.e., SQL statements or SQL methods) using a `MtStatement` object. You can create a statement object for a specific `MtDatabase` object using the `createStatement` method.

You can create more specific `Statement` objects for different purposes:

- `MtStatement` - It is specifically used for the SQL statements where you don't need to pass any value as a parameter
- `MtPreparedStatement` - It is a subclass of the statement class. The main difference is that, unlike the statement class, prepared statement is compiled and optimized once and can be used multiple times by setting different parameter values.
- `MtCallableStatement` - It provides a way to call a stored procedure on the server from a Python program. Callable statements also need to be prepared first, and then their parameters are set using the `set` methods.
- `MtResultSet` - It represents a table of data, which is usually generated by executing a statement that queries the database. A `ResultSet` object maintains a cursor pointing to its current row of data.

NOTE: With the Matisse Python SQL interface you usually don't need to use the Python stub classes unless you want to retrieve objects from a SQL statement or from the execution of a SQL method.

Creating Objects

You can also create objects into the database without the Python stub classes. The following code demonstrates how to create multiple objects of the same class using a prepared statement.

```

db.startTransaction()

# Create an instance of PreparedStatement
commandText = "INSERT INTO Person (FirstName, LastName, Age) VALUES (?, ?, ?)"
pstmt = db.prepareStatement(commandText)

# Set parameters
pstmt.setString(1, "James")
pstmt.setString(2, "Watson")
pstmt.setInt(3, 75)

print "Executing: " + pstmt.getStmtText()

# Execute the INSERT statement
inserted = pstmt.executeUpdate()

print "Inserted: " + str(inserted)

# Set parameters for the next execution
pstmt.setString(1, "Elizabeth")
pstmt.setString(2, "Watson")
pstmt.setNull(3)

print "Executing: " + pstmt.getStmtText()

# Execute the INSERT statement with new parameters
inserted = pstmt.executeUpdate()

print "Inserted: " + str(inserted)

# Clean up
pstmt.close()

db.commit()

```

1. Launch the application:

```

Windows:
python -i insertObjects.py

```

```

UNIX:
python -i insertObjects.py

```

Updating Objects

You can also create objects into the database without the Python stub classes. The following code demonstrates how to create multiple objects of the same class using a prepared statement.

```

db.startTransaction()

# Create an instance of Statement
stmt = db.createStatement()
# Set the relationship 'Spouse' between these two Person objects
commandText = "SELECT REF(p) FROM Person p WHERE FirstName = 'James' AND
LastName = 'Watson' INTO p1"
stmt.execute(commandText)
commandText = "UPDATE Person SET Spouse = p1 WHERE FirstName = 'Elizabeth' AND
LastName = 'Watson'"
inserted = stmt.executeUpdate(commandText)

# Clean up
pstmt.close()

db.commit()

```

1. Launch the application:

```

Windows:
python -i insertObjects.py

UNIX:
python -i insertObjects.py

```

Retrieving Values

You use the `ResultSet` object, which is returned by the `executeQuery` method, to retrieve values or objects from the database. Use the `next` method combined with the appropriate `getString`, `getInt`, etc. methods to access each row in the result.

The following code demonstrates how to retrieve string and integer values from a `ResultSet` object after executing a `SELECT` statement.

```

# Create an instance of PreparedStatement
commandText = "SELECT FirstName, LastName, Spouse.FirstName AS Spouse, Age FROM
Person WHERE LastName = ? LIMIT 10"
pstmt = db.prepareStatement(commandText)

# Set parameters
pstmt.setString(1, "Watson")

print "Executing: " + pstmt.getStmtText()

# Execute the SELECT statement and get a ResultSet
rset = pstmt.executeQuery()

print "Total selected: " + str(rset.getTotalNumObjects())
print "Total qualified: " + str(rset.getTotalNumQualified())

# Print column names
numberOfColumns = rset.getColumnCount()

# get the column names column indexes start from 1
for i in range(numberOfColumns):

```

```

        sys.stdout.write("{:16s} ".format(rset.getColumnname(i+1)))
print("")
for i in range(numberOfColumns):
    sys.stdout.write("----- ")
print("")

# Read rows one by one
while rset.next():
    # Get values for the first and second column
    fname = rset.getString(1)
    lname = rset.getString(2)
    sfname = rset.getString(3)
    age = rset.getInt(4)
    # The third column 'Age' can be null. Check if it is null or not first.
    if rset.wasNull():
        age = "NULL"
    # Print the current row
    print "{:16s} {:16s} {:16s} {}".format(fname, lname, sfname, age)

# Clean up and close the database connection
rset.close()
pstmt.close()

```

1. Launch the application:

```

Windows:
python -i selectValues.py

```

```

UNIX:
python -i selectValues.py

```

Retrieving Objects from a SELECT statement

You can retrieve Python objects directly from the database without using the Object-Relational mapping technique. This method eliminates the unnecessary complexity in your application, i.e., O/R mapping layer, and improves your application performance and maintenance.

To retrieve objects, use `REF` in the select-list of the query statement and the `getObject` method returns an object. The following code example shows how to retrieve `Person` objects from a `ResultSet` object.

```

# Create an instance of PreparedStatement
commandText = "SELECT REF(p) FROM Person p WHERE LastName = 'Watson';"
pstmt = db.createStatement()

print "Executing: " + commandText

# Execute the SELECT statement and get a ResultSet
rset = pstmt.executeQuery(commandText)

print "Total selected: " + str(rset.getTotalNumObjects())
print "Total qualified: " + str(rset.getTotalNumQualified())
print "Total columns: " + str(rset.getColumnCount())

print " Object Class:      FirstName:      LastName: Spouse FirstName: Age:"

```

```

# Read rows one by one
while rset.next():
    # Get the Person object
    p = rset.getObject(1)
    # Get object property values
    clsnam = p.getMtClass().getMtName()
    fname = p.getFirstName()
    lname = p.getLastName()
    sfname = p.getSpouse().getFirstName()
    # The third column 'Age' can be null. Check if it is null or not first.
    if p.isAgeNull():
        age = "NULL"
    else:
        age = p.getAge()
    # Print the current row
    print "{:16s} {:16s} {:16s} {:17s} {}".format(clsnam, fname, lname, sfname,
age)

# Clean up and close the database connection
rset.close()
pstmt.close()

```

1. Launch the application:

```

Windows:
python -i selectObjects.py

```

```

UNIX:
python -i selectObjects.py

```

Retrieving Objects from a Block Statement

You can also retrieve a collection of Python objects directly from the database by executing a SQL block statement.

The `getObject` method defined on a `MtCallableStatement` is used to return one object as well as an object collection. The following code example shows how to retrieve a collection of `Person` objects from a `MtCallableStatement`.

```

# Create an instance of CallStatement
commandText = "BEGIN\n"
commandText += " DECLARE res SELECTION(Employee);\n"
commandText += " DECLARE emp_sel SELECTION(Employee);\n"
commandText += " DECLARE mgr_sel SELECTION(Manager);\n"
commandText += " SELECT REF(p) FROM ONLY Employee p WHERE p.ReportsTo IS NULL
INTO emp_sel;\n"
commandText += " SELECT REF(p) FROM Manager p WHERE COUNT(p.Team) > 1 INTO
mgr_sel;\n"
commandText += " SET res = SELECTION(emp_sel UNION mgr_sel);\n"
commandText += " RETURN res;\n"
commandText += "END";

stmt = db.prepareCall(commandText)

```

```

print "Executing: " + stmt.getStmtText()

# Execute a block statement, and get the returned object selection
isRset = stmt.execute()

# Get Result Type
resultType = stmt.getResultType()
if ((resultType == MtStatement.METHOD) or
    (resultType == MtStatement.PROCEDURE)):
    # CALL statement with a return value
    returnType = stmt.getParamType(0)

    if ( MtType.SELECTION == returnType or
        MtType.OID == returnType ):
        print "result Type: {} - return Type: {} - result Class Type:
{}".format(MtStatement.stmtTypeToStr(resultType),
            MtType.toString(returnType),
            stmt.getStmtInfo(MtStatement.STMT_SELCLASS))
            sel = stmt.getObject(0)

        print "result Cnt: " + str(len(sel))

        for e in sel:
            print "{}: {} {} - Hiring Date: {}".format(e.getMtClass().getMtName(),
                e.getFirstName(),
                e.getLastName(),
                e.getHireDate() )

stmt.close()

```

1. Launch the application:

Windows:
python -i insertObjects.py

UNIX:
python -i insertObjects.py

Executing DDL Statements

You can also create schema objects from a Python application via SQL.

Creating a Class

You can create schema objects using the `executeUpdate` Method as long as the transaction is started in the `DATA DEFINITION` mode.

```

db.open()
db.setOption(MtDatabase.DATA_ACCESS_MODE, MtDatabase.DATA_DEFINITION)
db.startTransaction()
# Execute the DDL statement
stmt = db.createStatement()
stmt.executeUpdate ("CREATE CLASS Manager UNDER Employee (bonus INTEGER)")
stmt.close()

```

```
db.commit()
db.close()
```

Creating a SQL Method

Creating a schema object using the `execute` Method does not require to start a transaction. A transaction will be automatically started in the `DATA DEFINITION` mode.

```
db.open()
# NOTE: no transaction or version access mode started
# the query will auto-start the appropriate mode

stmt = db.createStatement()

# The first method returns the number of Person objects which have a specified last
name
commandText = \
    "CREATE STATIC METHOD CountByLName(lname STRING)\n" + \
    "RETURNS INTEGER\n" + \
    "FOR Person\n" + \
    "BEGIN\n" + \
    "  DECLARE cnt INTEGER;\n" + \
    "  SELECT COUNT(*) INTO cnt FROM Person WHERE LastName = lname;\n" + \
    "  RETURN cnt;\n" + \
    "END;"

print "\ncreating...\n" + commandText
stmt.execute(commandText)
stmt.close()
db.commit()
db.close()
```

1. Launch the application:

```
Windows:
python -i createSqlMethod.py
```

```
UNIX:
python -i createSqlMethod.py
```

Executing SQL Methods

You can call a SQL method using the `CALL` syntax, i.e., simply passing the SQL method name followed by its arguments as an SQL statement. You can also use the Callable Statement object, which allows you to explicitly specify the method's parameters.

Executing a Method returning a Value

The following program code shows how to call the SQL method `CountByLName` of the `Person` class.

```
# Specify the stored method. we call a static method,
# the name is consisted of class name and method name.
# Use CALL syntax to call the method
commandText = "CALL Person::CountByLName(?);"
```

```

# Create an instance of CallableStatement
stmt = db.prepareCall(commandText)

# Set parameters
stmt.setString(1, "Watson")

print "Executing:"
print stmt.getStmtText()

# Execute the stored method
stmt.execute()

# Get the returned value
count = stmt.getInt(0)

# Print it
print str(count) + " objects found\n"

# Clean up
stmt.close()

```

1. Launch the application:

Windows:
python -i callSqlMethod1.py

UNIX:
python -i callSqlMethod1.py

Executing a Method returning an Object

The following program code shows how to call the SQL method `FindByName` of the `Person` class.

```

# Specify the stored method. we call a static method,
# the name is consisted of class name and method name.
# Use CALL syntax to call the method
commandText = "CALL Person::FindByName('Watson', 'James');"

# Create an instance of CallableStatement
stmt = db.prepareCall(commandText)

print "Executing:"
print stmt.getStmtText()

# Execute the stored method
stmt.execute()

# Get the returned value
p = stmt.getObject(0)

# Print it
if (p != None):
    print "Found: {} {}\n".format(p.getLastName(), p.getFirstName())
else:
    print "no matching object found\n"

```

```
# Clean up
stmt.close()
```

1. Launch the application:

```
Windows:
python -i callSqlMethod2.py

UNIX:
python -i callSqlMethod2.py
```

Catching a Method Execution Error

The following program code shows how to retrieve the execution stack trace of a SQL method when an error occurs.

```
try:
    db = MtDatabase(dbhost, dbname, MtDynamicObjectFactory())
    db.open()

    db.startVersionAccess()

    # Specify the SQL method. Since we call a static method,
    # the name is consisted of class name and method name.
    # Use CALL syntax to call the method
    commandText = "CALL Employee::GetAnEmployee();"

    # Create an instance of CallableStatement
    stmt = db.prepareCall(commandText)

    print "The test executes a method that produces a runtime error"
    print "and shows how to get the error stack trace"

    print "Executing:"
    print stmt.getStmtText()

    # Execute the stored method
    stmt.execute()

    # Get the returned value
    p = stmt.getObject(0)

    # Print it
    if (p != None):
        print "Found: {} {}\n".format(p.getLastName(), p.getFirstName())
    else:
        print "no matching object found\n"

    # Clean up
    stmt.close()

    print "Done\n"

    db.endVersionAccess()

    db.close()
```

```

except MtException as mtex:
    stackTrace = stmt.getStmtInfo(MtStatement.STMT_ERRSTACK)
    print "Execution Error in:\n" + stmt.getStmtText()
    print "Execution Stack Trace:\n" + stackTrace
    code, msg = mtex
    print "MtException.message:\n" + msg

```

1. Launch the application:

Windows:
python -i callSqlMethod3.py

UNIX:
python -i callSqlMethod3.py

Deleting Objects

You can delete objects from the database with a DELETE statement as follows:

```

db.startTransaction()

stmt = db.createStatement()

# Delete all the instances of the Person Class
# Execute the DELETE statement
result = stmt.executeUpdate("DELETE FROM Person")

stype = MtStatement.stmtTypeToString(stmt.getResultType())
print "{} statement executed affecting {} objects in the
database.\n".format(stype,result)

# Clean up
stmt.close()

db.commit()

```

1. Launch the application:

Windows:
python -i clearPersonObjects.py

UNIX:
python -i clearPersonObjects.py

8 Working with Class Reflection

This section illustrates Matisse Reflection mechanism. This example shows how to manipulate persistent objects without having to create the corresponding Python subclass. It also presents how to discover all the object properties.

Running the Examples on Reflection

This example creates several objects, then manipulates them to illustrate Matisse Reflection mechanism.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `reflection` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `reflection/examples.odl` for this demo.

Creating Objects

This example shows how to create persistent objects without the corresponding Python subclass. The static method `get()` defined on all Matisse Meta-Schema classes (i.e. `MtClass`, `MtAttribute`, etc.) allows you to access to the schema descriptor necessary to create objects. Each object is an instance of the `MtObject` base class. The `MtObject` class holds all the methods to update the object properties (attribute and relationships (i.e. `setString()`, `setSuccessors()`, etc.).

```
# the MtCoreObjectFactory class provides a minimal object factory
# well suited for applications using reflection to manipulate objects
db = MtDatabase(dbhost, dbname, MtCoreObjectFactory())
db.open()
db.startTransaction()

print "Creating one Person...\n"
pCls = MtClass.get(db, "Person")
fnAtt = MtAttribute.get(db, "FirstName", pCls)
lnAtt = MtAttribute.get(db, "LastName", pCls)
cgAtt = MtAttribute.get(db, "collegeGrad", pCls)

p = MtObject(pCls)
p.setString(fnAtt, "John")
p.setString(lnAtt, "Smith")
p.setBoolean(cgAtt, False)

print "Creating one Employee...\n"
eCls = MtClass.get(db, "Employee")
hdAtt = MtAttribute.get(db, "hireDate", eCls)
slAtt = MtAttribute.get(db, "salary", eCls)
e = MtObject(eCls)
e.setString(fnAtt, "James")
e.setString(lnAtt, "Roberts")
e.setDate(hdAtt, date(2010, 1, 6))
```

```

e.setNumeric(slAtt, "5123.25")
e.setBoolean(cgAtt, True)

print "Creating one Manager...\n"
mCls = MtClass.get(db, "Manager")
tmRshp = MtRelationship.get(db, "team", mCls)
m = MtObject(mCls)
m.setString(fnAtt, "Andy")
m.setString(lnAtt, "Brown")
m.setDate(hdAtt, date(2009,11,8))
m.setNumeric(slAtt, "7421.25")
m.setSuccessors(tmRshp, (m, e))
m.setBoolean(cgAtt, True)

db.commit()
db.close()

```

1. Launch the application:

```

Windows:
python -i createObjects.py

UNIX:
python -i createObjects.py

```

Listing Objects

This example shows how to list persistent objects without the corresponding Python subclass. The `instanceIterator()` method defined on the `MtClass` object allows you to access all instances defined on the class.

```

db.startVersionAccess()

pCls = MtClass.get(db, "Person")
fnAtt = MtAttribute.get(db, "FirstName", pCls)
lnAtt = MtAttribute.get(db, "LastName", pCls)
cgAtt = MtAttribute.get(db, "collegeGrad", pCls)
print "\n {} Person(s) in the database.\n".format(pCls.getInstanceNumber())

itr = pCls.instanceIterator()
for p in itr:
    print "- cls={} oid={} - {} {} - collegeGrad={}\n".format(
        p.getMtClass().getMtName(),
        p.getMtOid(),
        p.getString(fnAtt),
        p.getString(lnAtt),
        p.getBoolean(cgAtt))

db.endVersionAccess()

```

1. Launch the application:

```

Windows:
python -i listObjects.py

```

```
UNIX:
python -i listObjects.py
```

Working with Indexes

This example shows how to retrieve persistent objects from an index. The `MtIndex` class holds all the methods retrieves objects from an index key.

```
db.startVersionAccess()

pCls = MtClass.get(db, "Person")
fnAtt = MtAttribute.get(db, "FirstName", pCls)
lnAtt = MtAttribute.get(db, "LastName", pCls)

pIdx = MtIndex.get(db, "personName")
print "\n {} entries in the index.\n".format(pIdx.getIndexEntriesNumber())

firstName = "Andy"
lastName = "Brown"
print "Looking for: {} {}\n".format(firstName, lastName)

# lookup for the number of objects matching the key
key = ( lastName, firstName )
count = pIdx.getObjectNumber(key)
print str(count) + " matching objects to be retrieved."

if (count > 1):
    # More than one matching object
    # Retrieve them with an iterator
    for p in pIdx.iterator(key, key):
        print " found {} {} OID={}".format(p.getString(fnAtt),
                                           p.getString(lnAtt),
                                           p.getMtOid())
else:
    # At most 1 object
    # Retrieve the matching object with the lookup method
    p = pIdx.lookup(key)
    if (p != None):
        print " found {} {}".format(p.getString(fnAtt), p.getString(lnAtt))
    else:
        print " Nobody found"

db.endVersionAccess()
```

1. Launch the application:

```
Windows:
python -i indexLookup.py
```

```
UNIX:
python -i indexLookup.py
```

Working with Entry Point Dictionaries

This example shows how to retrieve persistent objects from an Entry Point Dictionary. The `MtEntryPointDictionary` class holds the methods to retrieve objects from a string key.

```

pCls = MtClass.get(db, "Person")
mCls = MtClass.get(db, "Manager")
fnAtt = MtAttribute.get(db, "FirstName", pCls)
lnAtt = MtAttribute.get(db, "LastName", pCls)
cgAtt = MtAttribute.get(db, "collegeGrad", pCls)

# Get the EntryPointDictionary Descriptor object
cgEpd = MtEntryPointDictionary.get(db, "collegeGradDict")

collegeGrad = "true"

print "Looking for Persons with CollegeGrad={}".format(collegeGrad)

cnt = cgEpd.getObjectNumber(collegeGrad)
print "{} matching objects to be retrieved.\n".format(cnt)

if cnt > 1:
    # More than one matching object
    # Retrieve them with an iterator
    print "Looking up from an iterator:"
    cgItr = cgEpd.iterator(collegeGrad)
    for p in cgItr:
        print "  found OID={} {} {} collegeGrad={}\n".format(p.getMtOid(),
                                                            p.getString(fnAtt),
                                                            p.getString(lnAtt),
                                                            p.getBoolean(cgAtt))

    print "Retrieving them all at once filtered by class:"
    plst = cgEpd.lookupObjects(collegeGrad, mCls)
    for p in plst:
        print "  found OID={} {} {} collegeGrad={}\n".format(p.getMtOid(),
                                                            p.getString(fnAtt),
                                                            p.getString(lnAtt),
                                                            p.getBoolean(cgAtt))

else:
    # At most 1 object
    # Retrieve the matching object with the lookup method
    p = cgEpd.lookup(collegeGrad)
    if (p != None):
        print "  found OID={} {} {} collegeGrad={}\n".format(p.getMtOid(),
                                                            p.getString(fnAtt),
                                                            p.getString(lnAtt),
                                                            p.getBoolean(cgAtt))

    else:
        print "  Nobody found\n"

```

1. Launch the application:

Windows:
python -i entryPointLookup.py

UNIX:

```
python -i entryPointLookup.py
```

Discovering Object Properties

This example shows how to list the properties directly from an object. The `MtObject` class holds the `attributesIterator()` method, `relationshipsIterator()` method and `inverseRelationshipsIterator()` method which enumerate the object properties.

```
pCls = MtClass.get(db, "Person")

print "\n {} Person(s) in the database.\n".format(pCls.getInstancesNumber())

itr = pCls.instancesIterator()
for p in itr:
    print "- cls={} oid={}".format(p.getMtClass().getMtName(), p.getMtOid())
    print "  Attributes:"

    for att in p.attributesIterator():
        propType = att.getMtType()
        valType = p.getType(att)

        fmtVal = ""
        if valType == MtType.NULL:
            fmtVal = "(null)"
        elif valType == MtType.DATE:
            fmtVal = p.getDate(att)
        elif valType == MtType.NUMERIC:
            fmtVal = p.getNumeric(att)
        else:
            fmtVal = p.getValue(att).value
        print "  - att={} att-type={} - val-type={} value={}".format(
            att.getMtName(),
            MtType.toString(propType),
            MtType.toString(valType),
            fmtVal)

    print "  Relationships:"
    for rel in p.relationshipsIterator():
        print "    - rel={}: {} element(s)".format(rel.getMtName(),
p.getSuccessorSize(rel))
    print "  Inverse Relationships:"
    for rel in p.inverseRelationshipsIterator():
        print "    - rel={}: {} element(s)".format(rel.getMtName(),
p.getSuccessorSize(rel))
    print "Done\n"
```

1. \Launch the application:

```
Windows:
python -i listObjectProperties.py
```

```
UNIX:
python -i listObjectProperties.py
```

Adding Classes

This example shows how to add a new class to the database schema. The connection needs to be open in the DDL (`MtDatabase.DATA_DEFINITION`) mode. Then you need to create instances of `MtClass`, `MtAttribute` and `MtRelationship` and connect them together.

```
$db = new \matisse\MtDatabase($hostname, $dbname, new MtCoreObjectFactory());

// open connection in DDL mode
$db->setOption(\matisse\MtDatabase::DATA_ACCESS_MODE,
\matisse\MtDatabase::DATA_DEFINITION);
$db->open();

$db->startTransaction();

print "Creating 'PostalAddress' class and linking it to 'Person'...\n";

$cAtt = \matisse\reflect\MtAttribute::createAttribute($db, "City",
\matisse\reflect\MtType::STRING);
$pcAtt = \matisse\reflect\MtAttribute::createAttribute($db, "PostalCode",
\matisse\reflect\MtType::STRING);

$paClass = \matisse\reflect\MtClass::createClass($db, "PostalAddress", array( $cAtt,
$pcAtt ), null);

$pClass = \matisse\reflect\MtClass::get($db, "Person");

$adRshp = \matisse\reflect\MtRelationship::createRelationship($db, "Address",
$paClass, array ( 0, 1 ) );
$pClass->addMtRelationship($adRshp);

$db->commit();
$db->close();
```

1. Launch the application:

```
Windows:
python -i addClass.py

UNIX:
python -i addClass.py
```

Deleting Objects

This example shows how to delete persistent objects. The `MtObject` class holds `remove()` and `deepRemove()`. Note that on `MtObject` `deepRemove()` does not execute any cascading delete but only calls `remove()`.

```
db.startTransaction()

pCls = MtClass.get(db, "Person")
print "\n {} Person(s) in the database.\n".format(pCls.getInstancesNumber())

itr = pCls.instancesIterator()
for o in itr:
```

```

    o.deepRemove()

db.commit()

```

1. Launch the application:

```

Windows:
python -i deleteObjects.py

UNIX:
python -i deleteObjects.py

```

Removing Classes

This example shows how to remove a class for the database schema. The `deepRemove()` method defined on `MtClass` will delete the class and its properties and indexes. The connection needs to be open in `MtDatabase.DATA_DEFINITION` mode.

```

db = MtDatabase(dbhost, dbname, MtCoreObjectFactory())

# open connection in DDL mode
db.setOption(MtDatabase.DATA_ACCESS_MODE, MtDatabase.DATA_DEFINITION)

db.open()
db.startTransaction()

paClass = MtClass.get(db, "PostalAddress")

paClass.deepRemove()

db.commit()

```

1. Launch the application:

```

Windows:
python -i removeClass.py

UNIX:
python -i removeClass.py

```

9 Working with Database Events

This section illustrates Matisse Event Notification mechanism. The sample application is divided in two sections. The first section is event selection and notification. The second section is event registration and event handling.

Running the Events Example

This example creates several events, then manipulates them to illustrate the Event Notification mechanism.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `events` directory (under `examples`).
3. Launch the application:
To run the example, you need to open 2 command line windows and run one command in each windows.

```
Windows:
python -i subscribeEvents.py
python -i notifyEvents.py
```

```
UNIX:
python -i subscribeEvents.py
python -i notifyEvents.py
```

Events Subscription

This section illustrates event registration and event handling. Matisse provides the `MtEvent` class to manage database events. You can subscribe up to 32 events (`MtEvent.EVENT1` to `MtEvent.EVENT32`) and then wait for the events to be triggered.

```
TEMPERATURE_CHANGES_EVT = MtEvent.EVENT1
RAINFALL_CHANGES_EVT = MtEvent.EVENT2
HIMIDITY_CHANGES_EVT = MtEvent.EVENT3
WINDSPEED_CHANGES_EVT = MtEvent.EVENT4

db = MtDatabase(dbhost, dbname, MtDynamicObjectFactory())
db.open()

# Create a subscriber Event
subscriber = MtEvent(db)

# Subscribe to all 4 events
eventSet = TEMPERATURE_CHANGES_EVT | RAINFALL_CHANGES_EVT | HIMIDITY_CHANGES_EVT |
WINDSPEED_CHANGES_EVT

# Subscribe
subscriber.subscribe(eventSet)
```

```

for i in range(20):
    # Wait 1000 ms for events to be triggered
    # return 0 if not event is triggered until the timeout is reached
    print "wait 1 sec for event..."
    triggeredEvents = subscriber.wait(1000)
    if (triggeredEvents != 0):
        print "Events ({:d}) triggered with value
{:X}:\n".format(i,triggeredEvents)

        if ((triggeredEvents & TEMPERATURE_CHANGES_EVT) == 0):
            print "No "
            print "Change in temperature\n"

        if ((triggeredEvents & RAINFALL_CHANGES_EVT) == 0):
            print "No "
            print "Change in rain fall\n"

        if ((triggeredEvents & HIMIDITY_CHANGES_EVT) == 0):
            print "No "
            print "Change in humidity\n"

        if ((triggeredEvents & WINDSPEED_CHANGES_EVT) == 0):
            print "No "
            print "Change in wind speed\n"
        else:
            print "No Event received after 1 sec\n"

print "Unsubscribe to 4 Events\n"
# Unsubscribe to all 4 events
subscriber.unsubscribe()

```

Events Notification

This section illustrates event selection and notification.

```

TEMPERATURE_CHANGES_EVT = MtEvent.EVENT1
RAINFALL_CHANGES_EVT = MtEvent.EVENT2
HIMIDITY_CHANGES_EVT = MtEvent.EVENT3
WINDSPEED_CHANGES_EVT = MtEvent.EVENT4

db = MtDatabase(dbhost,dbname,MtDynamicObjectFactory())
db.open()
# Create a notifier Event
notifier = MtEvent(db)
# wait a little ...
print "waiting 3 seconds one or more subscribers can be started ... \n"

sleep(3)

for i in range(MAX_MEASURES):
    eventSet = 0
    if (changeInTemperature(i)): eventSet |= TEMPERATURE_CHANGES_EVT
    if (changeInRainfall(i)): eventSet |= RAINFALL_CHANGES_EVT
    if (i % 2 == 0): eventSet |= HIMIDITY_CHANGES_EVT
    if (i % 2 == 1): eventSet |= WINDSPEED_CHANGES_EVT

```

```

print "events {:X}".format(eventSet)
# Notify of some events
notifier.notify(eventSet)

print "Events (#{}) posted:\n".format(i+1)
if ((eventSet & TEMPERATURE_CHANGES_EVT) > 0): print "Change in temperature\n"
if ((eventSet & RAINFALL_CHANGES_EVT) > 0): print "Change in rain fall\n"
if ((eventSet & HUMIDITY_CHANGES_EVT) > 0): print "Change in humidity\n"
if ((eventSet & WINDSPEED_CHANGES_EVT) > 0): print "Change in wind speed\n"

# wait a little ...
print "waiting 1 sec ... \n"
sleep(1)

db.close()

```

More about MtEvent

As illustrated by the previous sections, the `MtEvent` class provides all the methods for managing database events. The reference documentation for the `MtEvent` class is included in the Matisse Python Binding API documentation located from the Matisse installation root directory in `docs/python/api/matisse.html`.

10 Handling Object Factories

The persistent objects retrieved from the database are automatically mapped to the most appropriate Python class. The object factories are responsible for creating the Python objects with the expected class.

Connection with Factory

Using MtDynamicObjectFactory

The `MtDynamicObjectFactory` is the default objects which can create a Python from an object retrieved from the database. In most cases, you use a `MtDynamicObjectFactory` object.

```
db = MtDatabase(dbhost, dbname, MtDynamicObjectFactory())
```

Using MtCoreObjectFactory

This factory is the basic `MtObject`-based object factory. This factory is the most appropriate for application which does use generated stubs. This factory is faster than the `MtDynamicObjectFactory` Object Factory since it doesn't use reflection to build objects.

```
db = MtDatabase(dbhost, dbname, MtCoreObjectFactory())
```

Creating your Object Factory

Implementing the MtObjectFactory interface

The `MtObjectFactory` interface describes the mechanism used by `MtDatabase` to create the appropriate Python object for each Matisse object. Implementing the `MtObjectFactory` interface requires to define the `getPythonClass()` method which return Python class corresponding to a Matisse Class Name, the `getDatabaseClass()` method which return the database class name from the Python class name and the `getObjectInstance()` method which return a Python object based on an oid.

```
class MyAppObjectFactory(MtObjectFactory):
    """
    A minimal object factory.
    """

    def getObjectInstance(self, db, mtOid):
        """
        Implements <b>MtObjectFactory.getObjectInstance</b>.

        @param MtDatabase db a database
        @param int mtOid the OID of the Python object to create
        @return Object the Python object represented by <b>mtOid</b>
        """
        if mtOid == 0:
            return None
        else:
            return MtObject(mtOid, db)
```

```

def getPythonClass(self, mtClsName):
    """
    Implements <b>MtObjectFactory.getPythonClass</b>.

    @param string mtClsName    a Matisse Class Name
    @return string the Python MtObject full classname
    """
    return "MtObject"

def getDatabaseClass(self, pyClsName):
    """
    Implements <b>MtObjectFactory.getDatabaseClass</b>.

    @param string pyClsName    a Python Class Name
    @return string             the Matisse full class name
    """
    return pyClsName

```

Implementing a Sub-Class of MtCoreObjectFactory

This MtCoreObjectFactory is a basic MtObject-based object factory which can be extended to implement your own Object Factory.

```

class MyOwnObjectFactory(MtCoreObjectFactory):
    """
    My Own Application object factory.
    """

    def getObjectInstance(self, db, mtOid):
        """
        Implements <b>MtObjectFactory.getObjectInstance</b>.

        @param MtDatabase db a database
        @param int mtOid the OID of the Python object to create
        @return Object the Python object represented by <b>mtOid</b>
        """
        if self.isSchemaObject(db, mtOid):
            return super(MyOwnObjectFactory, self).getObjectInstance(db, mtOid)
        else:
            # create you onw object
            myobject = ...
            return myobject

    def getPythonClass(self, mtClsName):
        """
        Implements <b>MtObjectFactory.getPythonClass</b>.

        @param string mtClsName    a Matisse Class Name
        @return string the Python MtObject full classname
        """
        return mtClsName

    def getDatabaseClass(self, pyClsName):
        """
        Returns the Matisse class name associated to the Python class stub

```

```
@param string pyClsName    a Python Class Name
@return string             the Matisse full class name
"""
return pyClsName
```

11 Building your Application

This section describes the process for building an application from scratch with the Matisse Python binding.

Discovering the Matisse Python Classes

The Matisse Python library is comprised of in 3 Python files:

1. **matisse.py** contains all the core classes defined in the **matisse** module. These classes manages the database connection.
1. **matisse_reflect.py** contains all the core classes defined in the **matisse_reflect** module. These classes manages the object factories. It also includes the Matisse meta-schema classes.
2. **matisse_sql.py** contains all the SQL-related classes defined in the **matisse_sql** module. These classes manages the execution of al types of SQL statements.

The Matisse Python API documentation included in the delivery provides a detailed description of all the classes and methods.

Generating Stub Classes

The Python binding relies on object-to-object mapping to access objects from the database. Matisse `mt_sdl` utility allows you to generate the stub classes mapping your database schema classes. Generating Python stub classes is a 2 steps process:

1. Design a database schema using ODL (Object Definition Language).
2. Generate the Python code from the ODL file:

```
mt_sdl stubgen -lang python myschema.odl
```

A `.py` file will be created for each class defined in the database. When you update your database schema later, load the updated schema into the database. Then, execute the `mt_sdl` utility in the directory where you first generated the class files, to update the files. Your own program codes added to these stub class files will be preserved.

Extending the generated Stub Classes

You can add your own source code outside of the `BEGIN` and `END` markers produced in the generated stub class.

```
// BEGIN Matisse SDL Generated Code
// DO NOT MODIFY UNTIL THE 'END of Matisse SDL Generated Code' MARK BELOW
...
// END of Matisse SDL Generated Code
```

Appendix A: Generated Public Methods

The following methods are generated automatically in the `.py` class files generated by `mt_sdl`.

For schema classes

The following methods are created for each schema class. These are class methods (also called static methods): that is, they apply to the class as a whole, not to individual instances of the class. These examples are taken from `Person`.

Count instances `getInstanceNumber(db)`
 `getOwnInstanceNumber(db)`

Open an iterator `instanceIterator(db)`
 `ownInstanceIterator(db)`

Sample constructor `createPerson(db)`

Sample toString `__str__()`

Get descriptor `getClass(db)`

Returns an `MtClass` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

Factory constructor `__init__(self, cls_or_oid, db=None)`

This constructor is called by `MtObjectFactory`. Actually, the constructor is inherited from `MtObject`

For all attributes

The following methods are created for each attribute. For example, if the ODL definition for class `Check` contains the attributes `Date` and `Amount`, the `Check.py` Python file will contain the methods `getDate` and `getAmount`. These examples are taken from `Person.firstName`.

Get value `getFirstName(self)`

Set value `setFirstName(self, val)`

Remove value `removeFirstName(self)`

Check Null value `isFirstNameNull(self)`

Check Default value `isFirstNameDefault(self)`

Get descriptor `getFirstNameAttribute(db)`

Returns an `MtAttribute` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

For list-type attributes only

The following methods are created for each list-type attribute. These examples are from `Person.photo`.

```

Get elements   getPhotoElements(self, offset, len)

Set elements   setPhotoElements(self, value, offset, len, discardAfter)

Count elements getPhotoSize(self)

```

For all relationships

The following methods are created for each relationship. These examples are from `Person.spouse`.

```

Clear successors clearSpouse(self)

Get descriptor   getSpouseRelationship(db)

                    Returns an MtRelationship object. This method supports advanced Matisse
                    programming techniques such as dynamically modifying the schema.

```

For relationships where the maximum cardinality is 1

The following methods are created for each relationship with a maximum cardinality of 1. These examples are from `Manager.assistant`.

```

Get successor   getAssistant(self)

Set successor   setAssistant(self, succ)

```

For relationships where the maximum cardinality is greater than 1

The following methods are created for each relationship with a maximum cardinality greater than 1. These examples are from `Manager.team`.

```

Get successors   getTeam(self)

Open an iterator teamIterator(self)

Count successors getTeamSize(self)

Set successors   setTeam(self, succs)

Add successors  Insert one successor before any existing successors:
                    prependTeam(self, succ)

                    Add one successor after any existing successors:
                    appendTeam(self, succ)

                    Add multiple successors after any existing successors:
                    appendTeam(self, succs)

Remove           removeTeam(self, succ)
successors

```

```
removeTeam(self, succs)
```

Remove specified successors.

For indexes

The following methods are created for every index defined for a database. These examples are for the only index defined in the example, `Person.personName`.

Lookup `lookupPersonName(db, lastName, firstName)`

Open an iterator `personNameIterator(db, fromLastName, fromFirstName, toLastName, toFirstName)`
`personNameIterator(db, fromLastName, fromFirstName, toLastName, toFirstName, filterClass, direction, numObjPerBuffer)`

Get descriptor `getPersonNameIndex(db)`

Returns an `MtIndex` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

For entry-point dictionaries

The following methods are created for every entry-point dictionary defined for a database. These examples are for the only dictionary defined in the example, `Person.commentDict`.

Lookup `lookupCommentDict(db, value)`

Open an iterator `commentDictIterator(db, value)`
`commentDictIterator(db, value, filterClass, numObjPerBuffer)`

Get descriptor `getCommentDictDictionary(db)`

Returns an `MtEntryPointDictionary` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.