PyLucene Features

Table of contents

Warning:

Before calling any PyLucene API that requires the Java VM, start it by calling `initVM(classpath, ...)`. More about this function in here.

1. Installing PyLucene

PyLucene is a Python extension built with JCC.

To build PyLucene, JCC needs to be built first. Sources for JCC are included with the PyLucene sources. Instructions for building and installing JCC are here.

Instruction for building PyLucene are here.

2. API documentation

PyLucene is closely tracking Java Lucene releases. It intends to supports the entire Lucene API.

PyLucene also includes a number of Lucene contrib packages: the Snowball analyzer and stemmers, the highlighter package, analyzers for other languages than english, regular expression queries, specialized queries such as 'more like this' and more.

This document only covers the pythonic extensions to Lucene offered by PyLucene as well as some differences between the Java and Python APIs. For the documentation on Java Lucene APIs, see here.

To help with debugging and to support some Lucene APIs, PyLucene also exposes some Java runtime APIs.

2.1. Samples

The best way to learn PyLucene is to look at the many samples included with the PyLucene source release or on the web at:

- http://svn.apache.org/viewcvs.cgi/lucene/pylucene/trunk/samples
- http://svn.apache.org/viewcvs.cgi/lucene/pylucene/trunk/samples/LuceneInAction

A large number of samples are shipped with PyLucene. Most notably, all the samples published in the *Lucene in Action* book that did not depend on a third party Java library for which there was no obvious Python equivalent were ported to Python and PyLucene.

Lucene in Action is a great companion to learning Lucene. Having all the samples available in Python should make it even easier for Python developers.

Lucene in Action was written by Erik Hatcher and Otis Gospodnetic, both part of the Java Lucene development team, and is available from Manning Publications.

2.2. Threading support with attachCurrentThread

Before PyLucene APIs can be used from a thread other than the main thread that was not created by the Java Runtime, the `attachCurrentThread()` method must be called on the `JCCEnv` object returned by the `initVM()` or `getVMEnv()` functions.

2.3. Exception handling with lucene.JavaError

Java exceptions are caught at the language barrier and reported to Python by raising a JavaError instance whose args tuple contains the actual Java Exception instance.

2.4. Handling Java arrays

Java arrays are returned to Python in a `JArray` wrapper instance that implements the Python sequence protocol. It is possible to change array elements but not to change the array size.

A few Lucene APIs take array arguments and expect values to be returned in them. To call such an API and be able to retrieve the array values after the call, a Java array needs to instantiated first.
For example, accessing termDocs:

```
termDocs = reader.termDocs(Term("isbn", isbn))
docs = JArray('int')(1)   # allocate an int[1] array
freq = JArray('int')(1)   # allocate an int[1] array
if termDocs.read(docs, freq) == 1:
    bits.set(docs[0])     # access the array's first element
```

In addition to `'int'`, the `'JArray'` function accepts `'object'`, `'string'`, `'bool'`, `'byte'`, `'char'`, `'double'`, `'float'`, `'long'` and `'short'` to create an array of the corresponding type. The `JArray('object')` constructor takes a second argument denoting the class of the object elements. This argument is optional and defaults to Object.

To convert a char or byte array to a Python string use a `''.join(array)` construct.

Instead of an integer denoting the size of the desired Java array, a sequence of objects of the expected element type may be passed in to the array constructor.
For example:

```
# creating a Java array of double from the [1.5, 2.5] list
JArray('double')([1.5, 2.5])
```

All methods that expect an array also accept a sequence of Python objects of the expected element type. If no values are expected from the array arguments after the call, it is hence not necessary to instantiate a Java array to make such calls.

See JCC for more information about handling arrays.

2.5. Differences between the Java Lucene and PyLucene APIs

- The PyLucene API exposes all Java Lucene classes in a flat namespace in the PyLucene module. For example, the Java import statement `import org.apache.lucene.index.IndexReader;` corresponds to the Python import statement `from lucene import IndexReader`
- Downcasting is a common operation in Java but not a concept in Python. Because the wrapper objects implementing exactly the APIs of the declared type of the wrapped object, all classes implement two class methods called instance_ and cast_ that verify and cast an instance respectively.

2.6. Pythonic extensions to the Java Lucene APIs

Java is a very verbose language. Python, on the other hand, offers many syntactically attractive constructs for iteration, property access, etc... As the Java Lucene samples from the *Lucene in Action* book were ported to Python, PyLucene received a number of pythonic extensions listed here:

- Iterating search hits is a very common operation. Hits instances are iterable in Python. Two values are returned for each iteration, the zero-based number of the document in the Hits instance and the document instance itself.
  The Java loop:

  ```
                for (int i = 0; i < hits.length(); i++) {
                    Document doc = hits.doc(i);
                    System.out.println(hits.score(i) + " : " +
  doc.get("title"));
                }
  ```

  can be written in Python:

  ```
               for hit in hits:
                   hit = Hit.cast_(hit)
                   print hit.getScore(), ':', hit.getDocument['title']
  ```

  if hit.iterator()'s next() method were declared to return `Hit` instead of `Object`, the above cast_() call would not be unnecessary.
  The same java loop can also be written:

```
for i xrange(len(hits)):
    print hits.score(i), ':', hits[i]['title']
```

- Hits instances partially implement the Python 'sequence' protocol.
  The Java expressions:

  ```
  hits.length()
  doc = hits.get(i)
  ```

  are better written in Python:

  ```
  len(hits)
  doc = hits[i]
  ```

- Document instances have fields whose values can be accessed through the mapping protocol.
  The Java expression:

  ```
  doc.get("title")
  ```

  is better written in Python:

  ```
  doc['title']
  ```

- Document instances can be iterated over for their fields.
  The Java loop:

  ```
  Enumeration fields = doc.fields();
  while (fields.hasMoreElements()) {
      Field field = (Field) fields.nextElement();
      ...
  }
  ```

  is better written in Python:

  ```
  for field in doc.getFields():
      field = Field.cast_(field)
      ...
  ```

  Once JCC heeds Java 1.5 annotations and once Java Lucene makes use of them, such casting should become unncessary.

2.7. Extending Java Lucene classes from Python

Many areas of the Lucene API expect the programmer to provide their own implementation or specialization of a feature where the default is inappropriate. For example, text analyzers and tokenizers are an area where many parameters and environmental or cultural factors are calling for customization.

PyLucene enables this by providing Java extension points listed below that serve as proxies for Java to call back into the Python implementations of these customizations.

These extension points are simple Java classes that JCC generates the native C++ implementations for. It is easy to add more such extensions classes into the 'java' directory of the PyLucene source tree.

To learn more about this topic, please refer to the JCC documentation.

Please refer to the classes in the 'java' tree for currently available extension points. Examples of uses of these extension points are to be found in PyLucene's unit tests and *Lucene in Action* samples.