LA-UR- 02-0393

Approved for public release; distribution is unlimited.

Title: PYTHON AND COMPUTER VISION

Author(s): DOAK, JUSTIN/112259/NIS-7
PRASAD, LAKSHMAN/117082/NIS-7

Submitted to: TENTH INTERNATIONAL PYTHON CONFERENCE
ALEXANDRIA, VA
FEBRUARY 3-7, 2002

Los Alamos
NATIONAL LABORATORY

Python and Computer Vision

Justin Doak and Lakshman Prasad

Mr. Doak and Dr. Prasad are both technical staff members at Los Alamos National Laboratory in the Safeguards Systems Group of the Nonproliferation and International Security Division.

Abstract

This paper discusses the use of Python in a computer vision (CV) project. We begin by providing background information on the specific approach to CV employed by the project. This includes a brief discussion of Constrained Delaunay Triangulation (CDT), the Chordal Axis Transform (CAT), shape feature extraction and syntactic characterization, and normalization of strings representing objects. (The terms "object" and "blob" are used interchangeably, both referring to an entity extracted from an image.) The rest of the paper focuses on the use of Python in three critical areas: 1) interactions with a MySQL database, 2) rapid prototyping of algorithms, and 3) gluing together all components of the project including existing C and C++ modules. For 1), we provide a schema definition and discuss how the various tables interact to represent objects in the database as tree structures. 2) focuses on an algorithm to create a hierarchical representation of an object, given its string representation, and an algorithm to match unknown objects against objects in a database. And finally, 3) discusses the use of Boost Python to interact with the pre-existing C and C++ code that creates the CDTs and CATs, performs shape feature extraction and syntactic characterization, and normalizes object strings. The paper concludes with a vision of the future use of Python for the CV project.

Keywords: Image Understanding, Computer Vision, Python, MySQL, MySQLdb, Boost Python, Constrained Delaunay Triangulation, Chordal Axis Transform, Shape Feature Extraction and Syntactic Characterization, Normalization, and Rapid Prototyping.

Introduction

The goal of the Computer Vision project is to accurately and rapidly recognize objects in imagery by comparing them to known objects in a database. The near-term focus is on creating and implementing novel techniques applicable to image understanding applications. A longer-term focus is on developing an underlying framework that will facilitate deployment of the technology to new applications and enable algorithm interchange. As of May, 2001, however, only a portion of the software needed to perform the near-term goal was complete. Moreover, those portions of the project that had been completed were isolated entities incapable of communicating directly with each other. In addition, a database, where the known objects reside, had not yet been integrated into the project. The need for 1) setting up a database, 2) rapidly prototyping algorithms, and 3) integrating existing software with newly created modules became clear.

To this end, an experienced software engineer was hired who had worked with Python for over a year in private industry. After much discussion, the rest of the team was convinced that Python would be an acceptable prototyping and integration environment for the project, while speed-critical portions of the project could be developed using C++. Demonstrated use of Boost Python to create C++ extension modules was critical to obtaining support for Python on the project. Also influential was the successful use of MySQLdb to access a MySQL database and the rapid implementation of several key algorithms.

The Computer Vision Project

Before we can process objects computationally, we must first recast contours, representing the exterior boundary and interior holes of an object, into parts (i.e., limbs, stems, and handles) with associated metrical information (e.g., area, length, and width). The CV Project performs this recasting through a variety of geometric transformations. From an input image, objects in the image are segmented out as sets of point sequences representing the discretized boundaries of the shapes. These sets of point sequences define polygons for the contours of objects in the image. For each polygon, we generate its Constrained Delauney Triangulation (CDT), which decomposes the interior of the shape into sleeve (S), junction (J), and terminal (T) triangles localizing the global structural properties of the shape. (See Figure 1.) The Chordal Axis Transform (CAT), a list of sequences of ordered pairs where each sequence characterizes a limb or a torso; is then induced from chains of adjacent S, J, and T triangles. (Note that torsos are further classified into either stems or handles.) Next, the CAT extracts, labels, and calculates metrical information for the principal features (i.e., limbs, stems, and handles) of a shape. The limbs (L), stems ('(' and ')'), and handles (H) make up the feature primitives in a context-free grammar that generates strings representing valid contours; shape exteriors and holes are represented as strings in the context-free language derived from the grammar. (See Figure 2.) Finally, the string representation of the object is normalized thus allowing identical or similar objects to obtain the same representation regardless of original position. This is currently accomplished by selecting a part to begin the string (the ordering of parts is unchanged) that results in a balancing of the areas for the major sections of an object. (The reader is refered to References 1 and 2 for more information on geometric processing in the CV Project.)

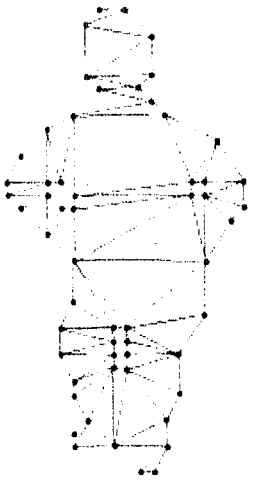


Fig. 1: The CDT of a Human Shape

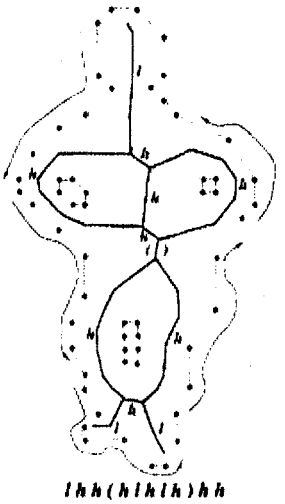


Fig. 2: Feature Labeling and String Representation of a Shape Exterior

The Use of Python

Historically, the CV Project has employed a handful of researchers working on distinct and separate aspects of the problem. Each researcher would write code to prototype and test certain ideas or algorithms that were fundamental to the overall project. (The researchers program in C and C++ and show little desire to switch development environments.) As the project expands, the establishment of software processes, configuration management practices, and a development environment has become critical. These practices should assist, for example, the use of multiple developers, code integration, documentation, and testing and shorten the start-up time for new researchers and developers.

Many factors were considered when choosing a development environment. However, the determining factor in the choice of development environment was the small amount of software engineering support relative to researchers on the CV Project. An environment needed to be chosen that optimized the use of developer time while meeting the overall needs of the project. With its high-level syntax, dynamic typing, and interpreted nature (to name just a few), Python can clearly boost developer productivity. However, how would it do as a language for supporting the other needs of the project?

Database Interactions

The stated goal of the CV Project is to perform object recognition by comparing an unknown object to objects in a database. A database implementation, a schema, and a python database interface are all needed to support this goal.

Database Implementation

For our initial prototyping, we selected MySQL as the database implementation. It is free, fast, and has proven very robust during our past experiences with it. One of the knocks against MySQL has been that it couldn't provide the same level of integrity as beefier databases such as Oracle or Sybase (See Why Not MySQL?.). Recently, support for transactions, with two-phase commits and rollbacks, has been added so earlier concerns about the integrity of MySQL may be moot at this point. Nonetheless, for

real-time, embedded applications that we envision for the CV technology, MySQL may not be the most appropriate solution. Other databases that are faster with a smaller footprint, such as the Empress RDBMS, are being evaluated.

Schema

Below, we give both a textual and a graphical description of the database schema.

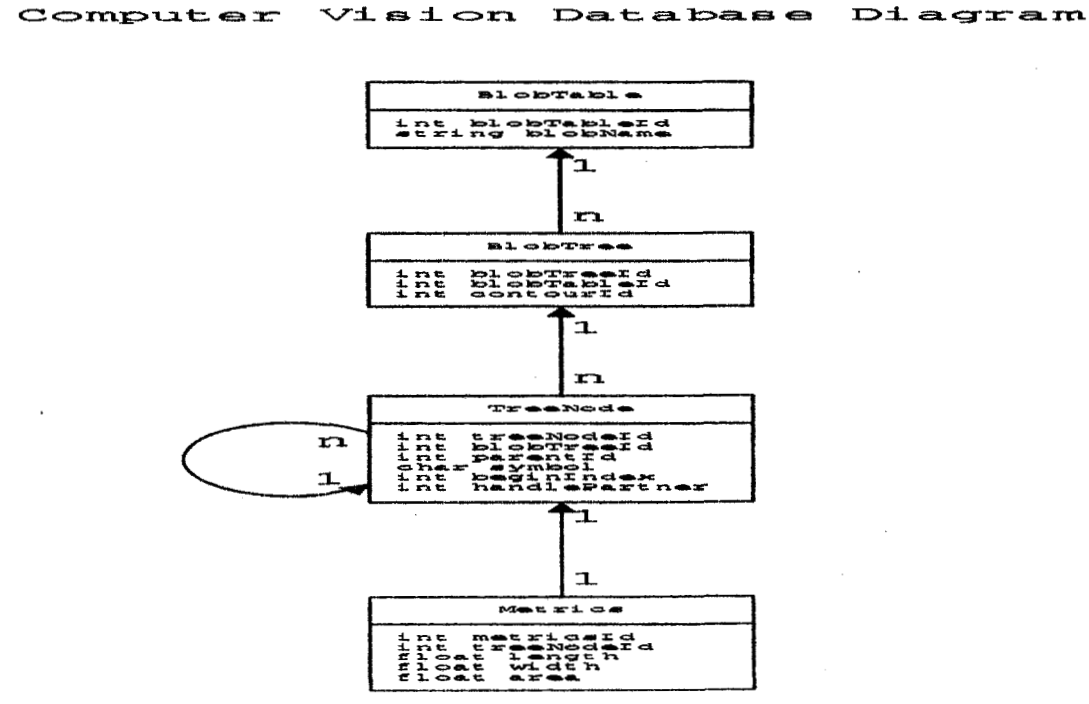
```
BlobTable
    blobTableId
    BlobName

BlobTree
    blobTreeId
    blobTableId
    contourId

TreeNode
    treeNodeId
    blobTreeId
    parentId
    symbol
    beginIndex
    handlePartner

Metrics
    metricsId
    treeNodeId
    length
    width
    area
```

Fig. 3: Database Schema Diagram (Textual)



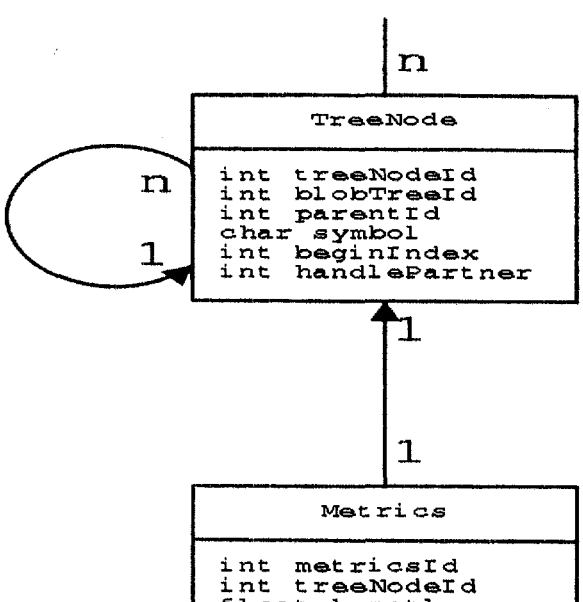


Fig. 4: Database Schema Diagram (Graphical)

Note in the diagram that arrows signify a reference to another table. For example, the n-to-one pointer from BlobTree to BlobTable indicates that each BlobTree instance has an id, or a reference, to a BlobTable and that there may be multiple BlobTrees that refer to a single BlobTable.

The schema for the CV Project must support a hierarchical or tree-like representation for the individual contours of an object. (An object is comprised of a single contour, representing the exterior of the object, and zero or more contours representing holes inside the object.) This tree-like representation is

generated from a contour's parts. A BlobTable entry names a single, possibly composite, object. The trees represented by a set of BlobTree records, all corresponding to the same object, describe the contours of that object. The nodes directly below the root of a tree represent the most abstract form of the contour while sub-levels correspond to finer-grained descriptions. The TreeNode table represents the nodes in a BlobTree and the Metrics table describes metrical information (e.g., area) for each node. BlobTree.beginIndex serves to order sibling nodes. We describe the tables more fully below.

The topmost table for an object is BlobTable. (Blob could not be used as the name of the table since that is a reserved word in the MySQL database.) BlobTable's main purpose is to bind contours to an object. It also provides a field for the name of the object.

The BlobTree table represents shapes (i.e., each instance of this table is a single contour). This table contains the id of the BlobTable instance to which it belongs (i.e., this is how you link a group of contours with an object). The purpose of this table is to associate a set of tree nodes with a particular contour.

The most complex of the database tables is TreeNode. It points at the BlobTree instance to which it belongs. In addition, there is a field containing a pointer to the node's parent. (The root's parent pointer is null.) There is also a field containing a symbol representing either a limb, torso, or handle. Finally, the table contains a field with the beginning index of the symbol (or meta-symbol) in the original string; this information is used to determine the ordering of the children of a node.

Each TreeNode table is associated with a Metrics table. The Metrics table contains information describing the size of the limb, handle, or torso in question. The table currently contains length, area, and width fields but could contain other metrical information as well.

Python Database Interface

The MySQLdb module implements a Python Database API Specification 2.0 compliant interface to MySQL. The interface requires that one first connect to the database and then create a cursor object. The cursor object is then able to execute queries and fetch results. In an attempt to provide an abstraction layer above this interface, a DbSession class was created. Application classes instantiate DbSession in __init__ and save the resulting object as a private member variable. When methods in the class need to perform queries against the database, the query method of the DbSession instance is called to perform the request. Although application classes must instantiate their own version of DbSession, there is in fact only one connection to the database in the form of a static member variable. We may refactor this into module-level functions so that each class does not require its own instance of DbSession.

We acknowledge that there are cleaner methods of abstracting database specifics. In particular, this implementation does not factor SQL variations into its design. Nonetheless, the DbSession idiom has proven effective at saving developer effort, again one of the more important considerations for this project.

Rapid Algorithm Prototyping

The two essential components of the CV Project are algorithm design and software development. In particular, the ability to rapidly prototype algorithms to get feedback on their effectiveness and efficiency is extremely valuable. This is, of course, where Python excels. We have found that Python's

dynamic typing, high-level syntax, interpreted nature, interactivity, lists and dictionaries, and powerful built-in libraries provide the most assistance when rapidly prototyping algorithms. Futhermore, our experience has been that it may take several weeks to generate an effective algorithm through a creative process of brainstorming, whiteboarding, and running several examples through by hand. But it may only take a few days to implement that algorithm in Python. (This is not a linear process but rather an iterative one where algorithms are envisioned, implemented, modified as a result of what was learned by the implementation, and then re-implemented.)

Currently, the project has not implemented any type of algorithm framework to facilitate algorithm interchange. This is due mainly to the initial focus on developing an end-to-end solution, from raw image to object recognition, rather than an extensible framework. As new ideas for solving various aspects of the problem emerge (and the project moves from a prototyping to a production phase), we will investigate certain design patterns (e.g., "Strategy" and "Template Method") to determine if they are appropriate. We now discuss two algorithms from the CV Project that were rapidly implemented in Python.

Creating String Hierarchies

After executing the processes of CDT, CAT, shape feature extraction and syntactic characterization, and normalization, we are left with what is often an extremely long string representing both major features and minor nuances along the exterior contour of an object. If this is what was stored in the database, matching an unknown object to objects in the database would consist of matching one very long string (representing the unknown object) to many very long strings (representing the objects in the database). It is also possible that two strings corresponding to similar objects could differ in minor parts hindering a fair comparison of the objects. Clearly, this will not suffice for real-world applications where real-time and robust recognition is required. Thus, we have created an algorithm that constructs a hierarchical or tree-like structure from a string representing a shape contour. This improves the efficiency of the matching algorithm (discussed below): shape features along contours are first compared at the most abstract level; only if contours match at this level are they compared at lower levels of detail in terms of finer shape features occuring along them.

As discussed earlier in this paper, we have created a context-free grammar for generating strings that are members of a language representing contours of objects. This grammar has the following rules, which are used in the hierarchicalization process.

1. L -> l
2. H -> h
3. L -> LL
4. L -> (LL)
5. L -> (H)
6. H -> HH
7. H -> HLH

The symbols 'l' , 'h', and "()" are the terminal (i.e., leaf) symbols for limbs, handles, and stems. 'H' and 'L' respresent non-terminal symbols (i.e., interior nodes) for handles and limbs. Rules 1 and 2 allow the generation of 'l's and 'h's from their non-terminal counterparts. Rules 3 through 7 allow the generation of more complex shapes from simple limbs or handles. Note that '(' and ')' can only be terminals (i.e., they have no non-terminal couterparts). They may nonetheless "disappear" as you go to more abstract

representations by reverse application of Rules 4 and 5.

The abstraction algorithm begins by creating the leaf nodes of the tree, applying Rules 1 and 2 in reverse to any 'l' and 'h' symbols. These nodes correspond directly to the symbols in the input string (except for the conversion of 'l's and 'h's to their equivalent non-terminals). Next, we walk through the leaf nodes from left-to-right (corresponding to their order in the original string) applying Rules 3 through 7 in reverse. For example, if the symbols "HLH" are encountered, they are replaced by the 'H' symbol. For each rule application, we create a node corresponding to the symbol on the LHS of the rule that will become the parent node for the nodes corresponding to the symbols on the RHS of the rule. We apply as many of the rules as possible to the elements in this list as we traverse from left-to-right. (In the future, metrical information will be used in conjunction with symbols to help determine when, and when not, to apply rules.) When we have made an entire pass through the list of nodes corresponding to the highest-level in the tree without applying a single rule, we are finished and we create a root node that becomes the parent of the nodes at the highest-level.

The following is an example of a printed version of an exterior contour. The various levels correspond to the different abstractions for the contour and are not meant to correspond directly to rule applications. (Of course, this printed version is derived from the hierarchical representation created by the rule applications.) Note that leaf symbols are lowercase and that some symbols are repeated from level-to-level to provide a complete representation of the object at each level in the tree. The most abstract representation of the object (a Chinese character) is given by "hLHLHl".

```
h L      H         L         H     l
h(ll)  H  l H  ( L   L  )  H    H  l
h(ll) H H lH lh((ll)(ll)) H  H H lhl
h(ll)hlhhhlhhlh((ll)(ll))hlhhlhhhlhl
```

A Matching Algorithm

To begin, the algorithm filters objects in the database to reduce the number of times the matching algorithm must be executed. Next, the unknown object is compared against each of the remaining objects (after filtering) in the database. For each comparison, the algorithm extracts the strings for the most abstract representations of the objects and sends them to an edit distance (EDF) function. The value returned by the EDF is used to initialize the overall matching score for the objects, lower scores indicating a closer match. The algorithm continues by comparing the children of nodes that matched at the most abstract level, again using the EDF to penalize mismatches. Weighting factors for tree level and subtrees are used to avoid excessively penalizing mismatches in branches (i.e., you're not comparing the whole trees) and in less abstract representations. Matching scores range from 0.0 to infinity with 0.0 indicating an identical match.

There are four components to the matching algorithm:

- Filtering. Objects in the database that do not exactly match the input object at the most abstract level are not further considered. Objects that matched exactly are further analyzed to determine if their metrical information (i.e., areas) is within a certain tolerance factor of the input object's metrical information. If so, a complete matching score (using the components described below) is calculated.

- An edit distance function. This calculates a score based upon the number of insertions, deletions,

and substitutions necessary to make two strings identical. In addition, even if two symbols in the strings being compared match, they may make a non-zero contribution to the overall edit distance score if there is a difference in metrical values (i.e., area). Metrical information may also contribute to the weights given to the aforementioned insertions, deletions, and substitutions. Another responsibility of this function is to return a list of matching nodes to be expanded and explored by the matching algorithm.

- A weighting factor for tree level. Obviously, edit distance scores at higher levels in the trees (corresponding to more abstract representations of the objects) should result in a greater penalty than edit distance scores obtained by comparisons at lower levels. This weighting factor is 1 / (2**(x-1)) where x is the level in the tree. So, for example, the edit distance score at Level 1 in the tree, the level directly below the root, is multiplied by 1 / (2**0) or 1. At level 6 in the tree, the edit distance score is multipled by 1 / (2**5) or 1/32 to obtain the penalty.

- A weighting factor for subtrees. Like the weighting factor for tree level, this value is multiplied by the edit distance score. So, wherever there is an edit distance score, we multiply by *both* weighting factors to obtain the penalty. We use metrical information from the nodes that matched at Level 1 (i.e., the level directly below the root) to compute the weighting factor for subtrees. In the following example, we are comparing two trees, X and Y, and determining the weighting factor for the subtrees represented by two nodes who matched at Level 1. Tree X has three nodes at Level 1 with areas of A1, A2, and A3. Tree Y has four nodes at Level 1 with areas of B1, B2, B3, and B4. A1 and B1 correspond to the areas for the matching nodes. Note that if A1 is small relative to A2 and A3 and B1 is small relative to B2, B3, and B4, the value will be close to 0.0. If A1 is large relative to A2 and A3 and B1 is large relative to B2, B3, and B4, the value will be close to 1.0. Note: we divide by 2.0 to keep the value between 0.0 and 1.0.

  ((A1 / (A1 + A2 + A3)) + (B1 / (B1 + B2 + B3 + B4))) / 2.0

  Let us briefly point out the importance of weighting the subtrees. Take as an example a situation where there are many matching nodes at Level 1 in the objects being compared. If we do *not* weight the subtrees, we will obtain an unacceptably high cumulative penalty for subsequent mismatches that in reality only corresponds to a comparison of branches from the two trees. This is mitigated by multiplying by a normalizing factor for the various subtrees.

A Glue Language

One of Python's strengths is its ability to use extension modules written in other languages. C and C++ extension modules are the most common and are hence very well supported by Python. Perhaps the reason for this popularity is that C and C++ offer something that Python lacks, speed. A common idiom in a Python application is that everything is written in Python *except* for those modules requiring speed; these are written as C or C++ extension modules.

Numerous tools are available to support the creation of C and C++ extension modules in Python. We wanted a tool that made it easy to export a C++ library into Python and didn't require alterations to the C++ code. Based on recommendations from former co-workers, Boost Python was tested and found to be adequate for our needs. We have used it to expose, within our Python environment, the interfaces for existing C++ software. Specifically, Boost Python requires the following steps:

1. Create a C++ API.
2. Write C++ code for a Python module that exposes the API. (This sounds strange, but you are actually writing C++ code that defines a Python module.)
3. Build a shared library using the .o files from the C++ API, the C++ code for a Python module that exposes the API, and some Boost utilities.
4. Put this .so file off your PYTHONPATH.
5. Run python, import the module that exposes the C++ API, and use the C++ functionality from within python.

Currently, we have created a Python extension module from the C++ normalization code. Next, we will create a Python extension module for the geometric processing code (that creates the string representations of objects) after refactoring and conversion to C++. Additionally, the CV application's prime bottleneck is anticipated to be matching where an unknown input object may be compared to a large number of objects in the database; it is therefore a prime candidate to become a C++ extension module.

Conclusions

The CV Project is progressing from a stage where papers and proof-of-concept prototypes are being augmented by software processes (e.g., version control), object-oriented software engineering, and a dynamic development environment. In addition, funding for the project is progressing from seed money to substantially higher levels to support various applications of the technology. The increase in funding will require the addition of new personnel heightening the need for seamless interactions with regards to presentations, publications, documentation, and, of course, software development. Python is proving itself a key player in this transition. There is a high-quality interface to MySQL (MySQLdb); algorithms can be rapidly prototyped; and Boost Python allows us to interface to existing C/C++ modules and write new modules in C++ where speed is critical. Realistically, real-time applications may eventually require that the entire system be written in C++. Nonetheless, Python will still have a role to play in the rapid prototyping of algorithms for proof-of-concept before implementation in C++.

Future Directions

As mentioned in Conclusions, the project is moving towards applying the technolgy to various applications. Thus, a shift is expected towards environments and tools that will perform for real-time, embedded applications. To work towards this goal, we are in the process of performing an evaluation of the Empress RDBMS, touted as extremely fast and lightweight. We also anticipate the eventual need to rewrite certain cpu-intensive Python modules in C++ (e.g., matching); it remains to be seen whether the entire system will need to be in C++ for speed.

Acknowledgements

This work has been supported by research and development funds from the Nonproliferation and International Security Division at Los Alamos National Laboratory.

References

1. L. Prasad, R. L. Rao, "A Geometric Transform for Shape Feature Extraction", Proceedings of

SPIE's 45th Annual Meeting, 2000, San Diego, CA, Vol. **4117** (2000), p 222.

2. L. Prasad, A. N. Skourikhine, B. R. Schlei, "Feature-based Syntactic and Metric Shape Recognition", Proceedings of SPIE's 45th Annual Meeting, 2000, San Diego, CA, Vol. **4117** (2000), p 234.

3. Ben Adida, "Why Not MySQL?"

4. David Abrahams, "The Boost Python Library (Boost.Python)"

5. Empress Software Inc., Empress RDBMS

6. Justin Doak, Lakshman Prasad, and Alexei Skourikhine, The Computer Vision Project

7. MySQL

8. MySQLdb

9. Python

10. Python Database API Specification 2.0