Coupling a Linguistic Formalism and a Script Language

Claude Roux
Xerox Research Centre Europe/ 6,
chemin de Maupertuis, 38240 Meylan,
France
Claude.roux@xrce.xerox.com

Abstract

This article presents a novel syntactic parser architecture, in which a linguistic formalism can be enriched with all sorts of constraints, included extra-linguistic ones, thanks to the seamless coupling of the formalism with a programming language.

1 Introduction

The utilization of constraints in natural language parsers (see Blache and Balfourier,2001 or Tapanainen and Järvinen , 1994) is central to most systems today. However, these constraints are often limited to purely linguistic features, such as linearity or dependency relations between categories within a given syntactic tree. Most linguistic formalisms have been created with the sole purpose of extracting linguistic information from bits and pieces of text. They usually use a launch and forget strategy, where a text is analyzed according to local constraints, displayed and then discarded to make room for the next block of text. These parsers take each sentence as an independent input, on which grammar rules are applied together with constraints. However, no sentence is independent of a text, and no text is really independent of extra-linguistic information. In order to assess correctly the phrase *President Bush*, we need to know that *Bush* is a proper name, whose function is "President". *Washington* can be a town, a state, the name of a famous president, but also the name of an actor. Moreover, the analysis of a sentence is never an independent process, if *President Bush* is found in a text, the reference to *president*, later in the document will be related to this phrase.

These problems are certainly not new and a dense literature has been written about how to better deal with these issues. However, most solutions rely on formalism enrichments with solutions "engraved in stone", that makes it difficult to adapt a grammar to new domains (see Declerck, 2002, or Roux, 2004), even though they use XML representation or database to store huge amounts of extra-linguistic information. The interpretation of these data is intertwined into the very fabric of the parser and requires deep modifications to use new sources with a complete different DTD.

Of course, there are many other ways to solve these problems. For instance, in the case of languages such as Prolog or Lisp, the grammar formalism is often indistinguishable from the programming language itself. For these parsers, the querying of external information is easily solved as grammar rules can be naturally augmented with non-linguistic procedures that are written in the same language. In other cases, when the parser is independent from any specific programming languages, the problem can prove difficult to solve. The formalism can of course be augmented with new instructions to tackle the querying of external information. However the time required to enrich the parser language may not be worth the effort, as the development of a complete new instruction set is a heavy and complex task that is loosely related to linguistic parser programming.

We propose in this article a new way of building natural language parsers, with the coupling of a script language with an already rich linguistic formalism.

2 Scripting

The system we describe in this article mix a natural language parser, namely Xerox Incremental Parser (XIP hereafter, see Aït-Mohktar et al., 2002, Roux, 1999) with a scripting language, in our case Python. The interaction of a grammar with scripting instructions is almost as old as

Proceedings of the 3rd Workshop on Constraints and Language Processing (CSLP-06), pages 33–40,
Sydney, July 2006.

computational linguistics. Pereira and Shieber for instance, in their book: Prolog and Natural Language Processing (see Pereira and Shieber, 1987) already suggested mixing grammar rules with extra-linguistic information. This mélange was made possible thanks to the homogeneity between grammar rules and the external code, written in both cases in the same programming language. However, these programming languages are not exactly tuned to do linguistic analyses; they are often slow and cumbersome. Moreover, they blur the boundary between program and grammar rules, as the programming language is both the algorithm language and the rule language. Allen (see Allen, 1994) proposes a different approach in his TRAINS Parsing system. The grammar formalism is independent to a certain extent from the implementation language, which is LISP in this case. However, since the grammar is translated into a LISP program, it is easy for a linguist to specialize the generated rules with external LISP procedures. Nevertheless, the grammar formalism remains very close to LISP data description, which makes the grammar rules somewhat difficult to read.

The other solution, which is usually favored by computational linguists, is to store the external information in databases, which are accessed with some pre-defined instructions and translated into linguistic features. For instance (see Declerk, 2002 or Roux, 2004), the external information is presented as an XML document whose DTD is defined once and for all. This DTD is then enriched with extra-linguistic information that a parser can exploit to guide rule application. This method alleviates the necessity of a complex interaction mechanism between the parser and its external data sources. The XPath language is used to query this document in order to retrieve salient information at parsing time, which is then translated into local linguistic features. However, only static information can be exploited, as these XML databases must be built beforehand.

Similar mechanisms have also been proposed in other architectures to help heterogeneous linguistic modules to communicate through a common XML interface (see Cunningham et al.,2002, Blache and Guénot , 2003). These architectures are very powerful as they connect together tools that only need to comply with a common input/output DTD. Specialized Java modules can then be written which are applied to intermediate representations to add their own touch of extra-linguistic data. Since, the intermediate representation is an XML document, the number of possible enrichments is almost limitless, as each module will only extract from this document the XML markup tags that it is designed to handle. However, since XML is by nature furiously verbose, the overall system might be very slow as it might spend a large amount of time translating external XML representation into internal representations.

Furthermore, applications that require natural language processing also have different purposes, different needs. They may require a shallow output, such as a simple tokenization with a whiff of tagging, or a much deeper analysis. Syntactic parsing is usually integrated as a black box into these architectures, with little control left over the grammar execution, control which nevertheless might prove very important in many cases. An XML document, for instance, often contains some specific markup tags to identify a title, a section or author name. If the parser is given some indications about the input, it could be guided through the grammar maze to favor these rules that are better suited to analyze a title for example.

Finally, syntactic parsing, when it is limited to lexical information, often fails to assess correctly some ambiguous relations. Thus, the only way to deal with PP-attachment or anaphoric pronoun antecedents is to use both previous analyses and external information. However, most syntactic parsers are often ill geared to link with external modules. The formalism is engraved into a C program as in Link Grammar (see Grinberg et al.,1995) or as in Sylex (see Constant, 1995) which offers little or no opening to the rest of the world, as it is mainly designed to accomplish one unique task. We will show how the seamless integration of a script language into the very fabric of the formalism simplifies the task of keeping track of previous analyses together with the use of external sources of data.

3 Xerox Incremental Parser (XIP)

The XIP engine has been developed by a research team in computational linguistics at the Xerox Research Centre Europe (see Aït-Mokhtar et al., 2001). It has been designed from the beginning to follow a strictly incremental strategy, where rules apply one after the other. There is only one analysis path that is followed for a given linguistic unit (phrase, sentence or even paragraph): the failure of a rule does not prevent the whole analysis from continuing to comple-

tion. Since the system never backtracks on any rules, XIP cannot propel itself into a combinatorial explosion.

XIP can be divided into two main components:

- A component that builds a chunk tree on the basis of lexical nodes.
- A component that creates functions or dependencies that connect together distant nodes from the chunk tree.

The central goal of this parser is the extraction of dependencies. A dependency is a function that connects together distant nodes within a chunk tree. The system constructs a dependency between two nodes, if these two nodes are in a specific configuration within the chunk tree or if a specific set of dependencies has already been extracted for some of these nodes (see Hagege and Roux, 2002). The notion of constraint embedded in XIP is both configurational and Boolean. The configuration part is based on tree regular rules which express constraints over node configuration, while the Boolean constraints are expressed over dependencies.

3.1 Three Level of Analysis

The parsing is done in three different stages:

- Part-of-speech disambiguation and chunking.
- Dependency Extraction between words on the basis of sub-tree patterns over the chunk sequence.
- Combination of those dependencies with Boolean operators to generate new dependencies, or to modify or delete existing dependencies.

3.2 The Different Steps of Analysis

Below is an example of how a sentence is parsed. We present a little grammar, written in the XIP formalism, together with the output yielded by these rules.

Example
The chunking rules produce a chunk tree.

In a first stage, chunking rules are applied and the following chunk tree is built for this sentence.
Below is a small XIP grammar that can analyze the above example:

```
1> AP = Adj.
2> NP @= Det,(AP),(Noun),Noun.
3> FV= verb.
4> SC= NP,FV.
```

Each rule is associated with a layer number, which defines the order in which the rules must be executed.
If this grammar is applied to the above sentence, the result is the following:

```
TOP{SC{NP{The AP{chunking} rules}
      FV{produce}}
   NP{a chunk tree}
   .}
```

TOP is a node that is automatically created, once all chunking rules have applied, to transform this sequence of chunks into a tree.
(The "@" denotes a longest match strategy. The rule is then applied to the longest sequence of categories found in the linguistic unit)

The next step consists of extracting some basic dependencies from this tree. These dependencies are obtained with some very basic rules that only connect nodes that occur in a specific sub-tree configuration.

```
SUBJ(produce,rules)
OBJ(produce,tree)
```

SUBJ is a subject relation, which has been extracted with the following rule:

```
| NP{?*, noun#1}, FV{?*,verb#2}|
  SUBJ(#2,#1).
```

This rule links together the noun and the verb respectively the sub-nodes of a NP and a VP that are next to each other. The "{…}" denotes a pattern over sub-nodes.

Other rules may then be applied to this output, to add or modify existing dependencies.

```
if (SUBJ(#1,#2) & OBJ(#1,#3))
 TRIPLET(#2,#1,#3).
```

For instance, the above rule will generate a three slot dependency *TRIPLET* with the nodes extracted from the subject and object dependencies. If we apply this rule to our previous example, we will create: TRIPLET(rules,produce,tree).

3.3 Script Language

The utilization of a script language deeply ingrained into the parser fabric might sound like a pure technical gadget with very little influence on parsing theories. However, the development of a parser often poses some very trivial problems, which we can sum up in the three questions below:

- How can we use previous analyses?
- How do we access external information?
- How do we control the grammar from an embedding application?

Usually, the answer for each of these questions leads to three different implementations, as none of these problems seem to have any connections whatsoever. Their only common point seems to be some extra-programming into the parser engine. If a grammar and a parser are both written in the same programming language, the problem is relatively simple to solve. However, if the grammar is written in a formalism specifically designed for linguistic analysis interpreted with a linguistic compiler (as it is the case for XIP), then any new features that would implement some of these instructions translate into a modification of the parsing engine itself. However, one cannot expand the parser engine forever. The solution that has been chosen in XIP is to develop a script language, which linguists can use to enrich the original grammatical formalism with new instructions.

3.4 First attempts

The first attempts to add scripting instructions to XIP consisted in enriching the grammar with numerical and string variables together with some instructions to handle these values. For instance, it is possible in XIP to declare a string variable, to instantiate it with the lemma value of a syntactic node and to apply some string modifications upon it. However, the development of such a script language, however useful it proved, became closer and closer to a general-purpose programming language, which XIP was not designed to be. The task of developing a full-fledged programming language with a large instruction set is a complex ongoing process, which has little connection with parsing theories. Nevertheless, there was a need for such an addendum, which led the development team to link XIP with Python, whose own ongoing development is backed up by thousands of dedicated computer scientists.

3.5 Python

Scripting languages have been around for a very long time. Thus Perl and Awk have been part of the Unix OS for at least twenty years. Python is already an old language, in computational time scale. It has been central to the Linux environment for more than ten years. Most of the basic installation procedures are written in that language. It has also been ported to a variety of platforms such as Windows or Mac OS. The language syntax is close to C, but lacks type verification. However, the language is thoroughly documented and a large quantity of specialized libraries is available. Python has also been chosen because of the simplicity of its API, which allows programmers to link easily a Python engine to their own application or to enlarge the language with new libraries. The other reason of this choice, over for instance a more conventional language such as C or Java is the fact that it is an interpreted language. A XIP grammar is a set of text files, which are all compiled on the fly in memory every time the parser is run. It stems from this choice that any addenda to this grammar should be written in a language that is also compiled on the fly. In this way, the new instructions can be developed in parallel with the grammar and immediately put in test. It also simplifies the non-trivial task of debugging a complete grammar as any modifications on any parts of the grammar can be immediately experimented together with the python script.
We have produced two different versions of the XIP-python parsing engine.

3.6 Python Embedded within XIP

We have linked the python engine to XIP, which allows us to call and execute python scripts from within the parsing engine. In this case, a grammar rule can call a python script to verify specific conditions. The python scripts are then appended to the grammar itself. These scripts have full access to all linguistic objects constructed so far. XIP is the master program with python scripts being triggered by grammar rules.

3.7 XIP as a Python Library

We have created a specific XIP library which can be freely imported in python. In this case, the XIP library exports a basic API, compliant with the python programming interface, which allows python developers to benefit from the XIP en-

gine. The XIP results are then returned as python objects. Since the purpose in this article is to show how a grammar formalism can be enriched with new instructions, we will mainly concentrate on the first point.

3.8 Interfacing Python and a XIP grammar

A XIP grammar mainly handles syntactic nodes, features, categories, and dependencies. In order to be efficient, a Python script, called from a XIP grammar, should have access to all this information in a simple and natural way. The notion of procedure has already been added to the XIP formalism. They can be used in any sort of rule.

Example

```
if (subject(#1,#2) & TestNode(#1))
   ambiguous(#1).
```

The above rule tests the existence of a subject dependency and will use the TestNode procedure to check some properties of the #1 node. If all these conditions are true, then a new dependency: *ambiguous* is created with #1 as parameter.

3.9 Interface

The *TestNode* procedure is declared in a XIP grammar in the following way:

Example

```
Python: //XIP field name
   TestNode(#1). //the XIP procedure name, with
                 XIP parameter style.

   //All that follows is in Python
   def TestNode(node):
   …
```

The only constraint is that the XIP procedure name (*TestNode*) should also have been implemented as a Python procedure. If this Python procedure is missing, then the grammar compilation fails.

The system works as a very simple linker, where the code integrity is verified to the presence of common names in XIP and Python.

However, the next step, which consists in translating XIP data into Python data, is done at runtime.

XIP recognizes many different sorts of data, which can all be transmitted to a Python script, such as syntactic nodes, dependencies, integer variables, string variables, or even vector variables. Each of these data is then translated into simple Python variables. However, the syntactic nodes and the dependencies are not directly transformed into Python objects; we simply propagate them into the Python code as integers. Each node and each dependency has a unique index, which simplifies the task of sharing parameters between XIP and Python.

3.10 XIP API

Python procedures have access to all internal parsing data through a specific API. This API consists of a dozen instructions, which can be called anywhere in the Python code. For instance, XIP provides Python instructions to return a node or a dependency object on the basis of its index. We have implemented the Python XipNode class, with the following fields:

```
class XipNode
    index          #the unique index of the node
    POS            #the part of speech
    Lemma          #a vector of possible lemmas
                   for the node
    Surface        #the surface form as it ap
                   pears in the sentence
    features       #a vector of attribute-value
                   features
    leftoffset,rightoffset  #the text offsets
    next,previous,parent,child    # indexes
```

A XipNode object is automatically created when the object creator is called with the node index as parameter. We can also travel through the syntactic tree, thanks to the *next, previous, parent, child* indexes that are provided by this class.

There is a big difference between using this API and exploiting the regular output of a syntactic parser. Since the Python procedures are called at runtime from the grammar, they have full access to the on-going linguistic data. Second, the selection of syntactic nodes on which to apply Python procedures is done at *the grammar level*, which means that the access of specific nodes is done through the parsing engine itself, without any need to duplicate any sorts of tree operators, which would be mandatory in the case of a Java, XML or C++ object. Finally, the memory footprint is only limited to the nodes that are requested by the application, there is no need to reduplicate the whole linguistic data structure. The memory footprint reduction also has the effect of speeding up the execution.

3.11 Other Basic Instructions

XIP provides the following Python instructions:

- **XipDependency(index)** builds a XipDependency object.
- **nodeset(POS)** returns a vector of node indices corresponding to a POS: *nodeset("noun")*
- **dependencyset(POS)** returns a vector of dependency indices corresponding to a dependency name: *dependencyset("SUBJECT")*
- **dependencyonfirstnode(n)** returns a vector of dependency indices, whose first parameter is the node index *n: dependencyonfirstnode(12)*

These basic instructions make it possible for a Python script to access all internal XIP data at any stages.

3.12 An Example

Let us define the Python code of *TestNbSenses*, which checks whether a verbal node is highly ambiguous according to WordNet. As a demonstration, a verb will be said to be highly ambiguous if the number of its senses is larger than 10.

```
def TestNbSenses(i):
   n=XipNode(i)
   senses=N[n.lemma].getSenses()
   if len(senses)>=10:
     return 1
   return 0
```

We can now use this procedure in a syntactic rule to test the ambiguity of a verb in order to guide the grammar:

```
if (subject(#1,#2) & TestNbSenses(#1))
        ambiguous(#1).
```

The dependency *ambiguous* will be created for a verbal node, if this verb is highly ambiguous.

4 Back to the Initial Questions

The questions we wish to answer are the following:

- How can we use previous analyses?
- How do we access external information?
- How do we control the grammar from an embedding application?

We have shown in the previous section how new instructions could be easily defined and thus become part of the XIP formalism. These instructions are mapped to a Python program which offers all we need to answer the above questions.

4.1 How can we use previous analyses?

Since, we have a full access to the internal linguistic representation of XIP, we can store whatever data we might find useful for a given task. For instance, we could decide to count the number of time a word has been detected in the course of parsing. This could be implemented with a Python dictionary variable.

```
Python:
   countword(#1).
   getcount(#1).
   …
```

The first procedure *countword* receives a node index as input. It translates it into a XipNode, and it uses the lemma as an entry for the Python dictionary *wordcounter*. At the end of the process, *wordcounter* contains a list of words with their number of occurrences. The second procedure implements a simple test which returns the number of time a word has been found. It returns 0, if it is an unknown word.
The grammar rule below is used to count words:

```
|Noun#1| {
        countword(#1);
}
```

The instruction |*noun#1*| automatically loops between all *noun* nodes.
The rule below is used to test if a word has already been found:

```
if (subject(#1,#2) & getcount(#2)) …
```

4.2 How do we access external information?

We have already given an example with WordNet. Thanks to the large number of libraries available, a Python script can benefit from WordNet information. It can also connect to a variety of databases such as MySQL, which also allows a grammar to query a database for specific data.
For instance, we could store in a database verb-noun couples that have been extracted from a

large corpus. Then, at runtime, a grammar could check whether a certain verb and a certain noun have already been found together in another document.

Example

Python:
TestCouple(#1,#2).

```
def TestCouple(v,n):
   noun=XipNode(n)
   verb=XipNode(v)
   cmd=”select * from couples where ”
   cmd+=”verb=”+verb.lemma+"
   cmd+=” and noun=”+noun.lemma+”;”
   nb=mysql.execute(cmd)
   return nb
```

In the XIP grammar:

```
|FV{verb#1},PP{prep,NP{noun#2}}|
  if (TestCouple(#1,#2))
     Complement(#1,#2).
```

If we have a *verb* followed by a *PP*, then if we have already found in a previous analysis a link between the *verb* and the *noun* embedded in the *PP*, we create a dependency *Complement* over the *verb* and the *noun*.

4.3 How do we control the grammar from an embedding application?

Since a Python script can exploit any sort of input, from text files to databases; it becomes relatively simple to implement a simple Python procedure that blocks the execution of certain grammar rules. If we examine the above example, we can see how the grammar execution can be modified by an external calling program. For instance, the selection of a different database will have a strong influence on how dependencies are constructed.

5 Expression Power

The main goal of this article is to describe a way to articulate no-linguistic constraints with a dedicated linguistic formalism. The notion of constraint in this perspective does not only apply to purely linguistic properties such as category order or dependency building constraints; it is enlarged to encompass properties that are rarely taken into account in syntactic theories. It should be noted, however, that if most theories are designed to apply to a single sentence, nothing prevents these formalisms to benefit from extra-linguistic data through a complex feature system that would encode the sentence context. How these features are instantiated is nevertheless out the realm of these theories. The originality of our system lies in the fact that we intertwine from the beginning these constraints into the fabric of the formalism. Since any rules can be governed by a Boolean expression, which in turn can accept any Boolean python functions, it becomes feasible to define a formalism in which a constraint is no longer reduced to only linguistic data, but to any properties that a full-fledged programming language can allow. Thus, any rule can be constrained during its application with complex constraints which are implemented as a python script.

Example

pythontest is a generic Boolean python function, which any XIP rules can embed within its own set of constraints.

Below are some examples of XIP rules, which are constrained with this generic python function. A constraint in XIP is introduced with the keyword “*if*”.

- A chunking rule:

PP = prep, NP#1, if (*pythontest*(#1)).

- A dependency rule:

if (subject(#1,#2) & *pythontest*(#1)) …

However, since any rule might be constrained with an external process it should be noted that this system can no longer be described as a pure linguistic parser. Its expression power largely exceeds what is usually expected from a syntactic formalism.

6 Implementation Examples

We have successfully used Python in our grammars in two different applications so far. The first implementation consists of a script that is called at the end of any sentence analysis to store the results in a MySQL database. Since the saving is done with a Python program, it is very simple to modify this script to store only information that is salient to a particular application. In this respect, the maintenance of such a script is much simpler and much flexible than its C++ or Java counterpart. The storage is also done at runtime which limits the amount of data kept in memory.

The second example is the implementation of a co-reference system (Salah Aït-Mohktar to appear), which uses Python as a backup language to keep a specific representation of linguistic information that is used at the end of the analysis to link together pronouns and their antecedents. Once again, this program could have been created in C++ or Java, using the C++ or the Java XIP API, however, the development of such a system in python benefits from the simplicity of the language itself and its direct bridge to internal XIP representation.

7 Conclusion

The integration of a linguistic parser into an application has always posed some tricky problems. First, the grammar, whether it has been compiled into an external library or run through an interpreter, often works as a black box, which allows little or no possibility of interfering with the internal execution. Second, the output is usually frozen into one single object which forces the calling applications to perform format translation afterward. In many systems (Cunningham et al.,2002, Grinberg et al., 1995), the output is often a large, complex object, or a large XML document. This has an impact on both memory footprint (these objects might be very large) and the analysis speed as the system must re-implement some tree operators to traverse these objects. Thereby, the automatic extraction of all nodes that share a common property on the basis of these objects requires some cumbersome programming, when this could be more elegantly handled through the linguistic formalism. Third, the use of extra-linguistic information often imposes a modification of the parsing engine itself, which prevents developers from switching quickly between heterogeneous data sources. For a long time, linguistic formalisms have been conceived as specialized theoretical languages with little if no algorithmic possibilities. However, today, the use of syntactic parsers in large applications triggers the need for more than just pure linguistic description. For all these reasons, the integration of a script language as part of the formalism seems a reasonable solution, as it will transform dedicated linguistic formalisms to linguistically driven programming languages.

Reference

Gazdar G., Klein E., Pullum G., Sag A. I., 1985. *Generalized Phrase Structure Grammar*, Blackwell, Cambridge Mass., Harvard University Press.

Pereira F. and S. Shieber, 1987. *Prolog and Natural Language Analysis*, CSLI, Chicago University Press.

Allen J. F, 1994. *TRAINS Parsing System*, Natural Language Understanding, Second Ed., chapters 3,4,5.

Tapanainen P., Järvinen T. 1994. *Syntactic analysis of natural language using linguistic rules and corpus-based patterns*, Proceedings of the 15th conference on Computational linguistics, Kyoto, Japan, pages: 629-634.

Constant P. 1995. *L'analyseur Linguistique SYLEX*, 5 ème école d'été du CNET.

Grinberg D., Lafferty John, Sleator D., 1995. *A robust parsing algorithm for link grammars*, Carnegie Mellon University Computer Science technical report CMU-CS-95-125, also Proceedings of the Fourth International Workshop on Parsing Technologies, Prague, September, 1995.

Fellbaum C., 1998. *WordNet: An Electronic Lexical Database*, Rider University and Princeton University, Cambridge, MA: The MIT Press (Language, speech, and communication series), 1998, xxii+423 pp; hardbound, ISBN 0-262-06197-X.

Roux C. 1999. *Phrase-Driven Parser*,Proceedings of VEXTALL 99, Venezia, San Servolo, V.I.U. - 22-24.

Blache P., Balfourier J.-M., 2001. *Property Grammars: a Flexible Constraint-Based Approach to Parsing*, in proceedings of *IWPT-2001*.

Aït-Mokhtar S., Chanod J-P., Roux C., 2002. *Robustness beyond shallowness incremental dependency parsing*, NLE Journal, 2002.

Hagège C., Roux C.,2002. *A Robust And Flexible Platform for Dependency Extraction*, in proceedings of LREC 2002.

Declerck T. 2002, *A set of tools for integrating linguistic and non-linguistic information*, Proceedings of SAAKM.

H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan.,2002. *GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications*, Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02), Philadelphia, July 2002.

Blache P., Guénot M-L. 2003. *Flexible Corpus Annotation with Property Grammars*, BulTreeBank Project

Roux C., 2004. *Une Grammaire XML*, TALN Conference, Fez, Morocco, April, 19-22, 2004.

[Python] http://www.python.org/