Xerox Incremental Parser

Python Connection

6, CH DE MAUPERTUIS
38240 MEYLAN
France

xerox

Printed in France

6, CH DE MAUPERTUIS
38240 MEYLAN
France

Contents

CONTENTS **3**

ADDENDA **5**

XIP PYTHON **6**

Declaring a Python Program **6**
- Important Points 6
- The two declarations 7

The *xips* module **8**
- index=xips.loadgrm(filename) 8
- xips.testdisplay(grm,VALUE) 8
- xips.adddisplay(grm,VALUE) 8
- xips.removedisplay(grm,VALUE) 8
- xips.setdisplay(index,mode) 8
- xips.node(i) 9
- xips.nodeset("POS") 10
- xips.lexicals() 10
- xips.dependency(i) 10
- xips.dependencyset("NAME") 10
- xips.dependencyonfirstnode(node) 10
- xips.dependencyonsecondnode(node) 11
- xips.setexchange(strvar,data) 11
- xips.setint(index,v,i) 11
- xips.setfloat(index,v,f) 12
- xips.setstring(index,v,s) 12
- xips.parsestring(index,str,data) 13
- xips.parsefile(index,filename,data) 13
- xips.parsexmlstring(index,str,depth,data) 14
- xips.parsexmlfile(index,filename,depth,data) 14
- xips.currentrule() 14
- xips.currentruletext(displayheader) 14
- xips.loadkif(filename,arguments,mapping) 14
- xips.kif("functionname",p1,p2…) 14

Python Class: XipNode **15**
- Example 17
- IMPORTANT 17

Python Class: XipDependency **17**

A Real Life Example **18**
- Explanation 19

Calling a python script from within a grammar **19**
- Example 19

PROBABILITIES 21

Instructions 21

- Python API 22

Addenda

This document describes the different functions that are necessary to combine XIP rules with Python instruction (www.python.org).

Important

Only specific versions of XIP are available with Python.
Please ask Xerox for a Python enable version.

XIP PYTHON

XIP provides an interface to transparently execute Python scripts. For more information about Python, please consult any documentation about the language. A tutorial is also available on: http://docs.python.org/index.html.

Declaring a Python Program

The most important point when using a python program is to declare a specific interface between XIP and Python. This is done by declaring a XIP function whose name must match one of the python functions that would be defined in the program. Only one interface can be declared at a time in a file. If more than one interface is to be declared, then use another file. A file should only contain one python interface and nothing else.

Important Points

1. An interface is a simple procedure name with XIP parameters. This name will be used in the rest of the XIP grammar as the way to execute the Python script.
2. The declaration of the interface should always follow the domain keyword *Python*.
3. The possible types of parameters are: #x, int, float, string, vint, vindex, vector, and dictionary.
4. *vint, vindex, vector* and *dictionary* are always considered as global, which means that their values may be modified by the python script.
5. *int, float,* and *string* can be either global or local. A global parameter name should always start with a "_". When a global parameter is declared for one of those types, then the matching python function, should declare them as *global, using exactly the same name as the parameters declared in the XIP interface.*
6. *#x* can be sent as a parameter to a python interface. The python function corresponding to the interface will receive this parameter as a *int* which is the unique index of the XIP node pointed at by #x. XIP provides a specific python class to traverse the XIP chunk tree.
7. The node parameters should always start the list of arguments.

The two declarations

There are two different ways to declare a python interface. These two different implementations lead to two different behaviors which may be difficult to understand without some preliminary explanations on how Python handles namespaces.

In Python, variables and functions are stored in namespaces which are defined by the imported modules.

There is a main namespace, whose name is "__*main*__" in which one can store all her/his code. However, when using this location for storage, there might be some discrepancies, if for example two modules implement the same variable or the same function for different usages. In that case, the last declaration will replace any previous ones, which may lead to some mysterious bugs.

The second method consists in considering each python interface as a *module* in which all variables and functions are local to that module without any risks of interferences with other modules. However, with this second method, one should keep in mind that if more than one file is declared in the grammar, then the others *modules* should import each other in order to use their local variables and functions.

Declaration of type 1

In order to load all the modules in the main memory, the python interface file must start with the keyword "*python:"* followed by the declaration itself.

Python:

entreepoint(….).

Python module starts here…

Declaration of type 2

In order to create a module from each file, the python interface file *should always start with: """, then the declaration as above. The declaration should then be closed with """ again.*
By starting a file with these three double quotes, the embedded *Python* interpreter will parse the file as if this section was the documentation section of the module file (stored in __doc__). The XIP file is then *imported* as a single Python module. The name of the file can then be used as a way to access its inner functions and variables as an actual python module. Exactly as if it was loaded in Python using *import.*

```
"""
Python:

entreepoint(…).

"""
Python module starts here…
```

IMPORTANT

The presence or the absence of these three double-quotes may seems trivial compared to the loading of a python file, however it has a deep influence on how the code is handled. We suggest users to prefer the second declaration as it is closer to the python methodology than the first declaration. However, the first declaration makes it easier to have all modules communicating with the danger of crushing certain declarations with others…

The *xips* module

The *xips* module exports a few interesting functions which can be used to retrieve some specific linguistic information for a syntactic node or a dependency.

- **index=xips.loadgrm(filename)**

This method loads a grm file and returns the index on that grammar, which can then be used by the *parse* methods.

- **xips.testdisplay(grm,VALUE)**

This method returns 1 if the value has been set, 0 otherwise.

- **xips.adddisplay(grm,VALUE)**

This method adds the value VALUE to the display mode

- **xips.removedisplay(grm,VALUE)**

This method remove the display value from the display mode

- **xips.setdisplay(index,mode)**

This method initializes the display mode of XIP. *Mode* is a combination of any following values (exported by XIP):

```
XIP_DISPLAY_LEMMA=1
XIP_DISPLAY_SURFACE=2
XIP_DISPLAY_MARKUP=4
XIP_DISPLAY_ENTREE=8
XIP_DISPLAY_CATEGORY=16
XIP_DISPLAY_REDUCED=32
```

XIP_DISPLAY_FULL=64
XIP_DISPLAY_OFFSET=128
XIP_DISPLAY_WORDNUM=256
XIP_DISPLAY_SENTENCE=512
XIP_DISPLAY_NONE=16384
XIP_DISPLAY_DEPENDENCY_BY_NAME=32768
XIP_DISPLAY_DEPENDENCY_BY_NODE=65536
XIP_DISPLAY_DEPENDENCY_BY_CREATION=131072
XIP_DISPLAY_TREE=262144
XIP_DISPLAY_TREE_PHRASE=524288
XIP_DISPLAY_TREE_COLUMN=1048576
XIP_DISPLAY_MERGE_XML_SUBTREE=2097152
XIP_DISPLAY_CONVERSION_UTF8=4194304
XIP_DISPLAY_EXECUTION_ERROR=8388608
XIP_DISPLAY_MATHEMATICS=16777216
XIP_DISPLAY_DEPENDENCY_NUMBER=33554432
XIP_UTF8_INPUT=67108864
XIP_DISPLAY_EXECUTE_TOKEN=134217728
XIP_DISPLAY_SENTENCE_NUMBER=268435456
XIP_LANGUAGE_GUESSER=536870912
XIP_UTF8_LEXICON=1073741824
XIP_DISPLAY_CHUNK_TREE=2147483648
XIP_DISPLAY_DEPENDENCY_FEATURE_VALUE=4294967296

Example:

xips.setdisplay(index,XIP_DISPLAY_LEMMA|XIP_DISPLAY_CHUNK_TREE)

xips.node(i)

This method retrieves, for a given syntactic node index, a list of strings and integer which has the following structure:

[POS,surface,lemma,features,parent,daughter,last,next,previous,left,right,leftoff,rightoff]

1. **POS** is a string corresponding to the part of speech of the node
2. **surface** is the suface form of the node
3. **lemma** is the lemma form of the node
4. **features** is a string where attributes:values are concatenated in one single string
5. **parent** is the index of the parent node
6. **daughter** is the index of the daughter node
7. **last** is the index of the last node
8. **next**, previous are the indexes of the next or the previous node.
9. **left**,**right** are the left and right number of the tokens
10. **leftoff**, **rightaff** are the left and right offsets of the tokens

Example

xips.node(5) returns [“NOUN”,”The”,”the”,”[fem:+]”,1,2,4,9,8,0,4]

xips.nodeset("POS")

This method retrieves a list of integers, where each value is an index to a node whose part of speech is POS.

Example

xips.nodeset("NOUN") returns [1,4,5]

xips.lexicals()

This method returns the list of lexical nodes indexes.

xips.dependency(i)

This method returns, for a dependency index, a list of strings and integer which has the following structure:

[NAME,features,param1,param2,etc.]

1. NAME is the name of the dependency
2. features is a string where attributes:values are concatenated in one single string
3. param1, param2, etc. is a list of syntactic node indexes

Example

xips.dependency(5) returns ["SUBJ","[passive:+]",3,5]

xips.dependencyset("NAME")

This method returns a list of dependency indexes whose name is NAME.

Example

xips.dependencyset("SUBJ") returns [3,4]

xips.dependencyonfirstnode(node)

This method returns a list of dependency indexes which have *node* as their first parameter. n*ode* is a node index.

Example

```
Python :
displaydependencies(#1) ;

Def displaydependencies (node) :
   depset= xips.dependencyonfirstnode(node)
   for i in depset:
       print XipDependency(i).name
```

xips.dependencyonsecondnode(node)

This method returns a list of dependency indexes which have *node* as their second parameter. n*ode* is a node index.

xips.setexchange(strvar,data)

XIP provides a specific type to handle python object: *python*. You can declare as many python variables as you want. This method is used to associate a XIP variable of type *python* with any python object, which then may be stored and exchanged between different python procedures called from XIP. s*trvar* is the name of the XIP variable. s*trvar* is a string. Python does not have access to XIP variable of that type directly, we must pass the parameter as a string.

Example:

```
//The Python script called from XIP
Python:
getmyvar(string var).

data=[1,2,3]
def getmyvar(var):
  xips.setexchange(var,data)

//The Python script called from XIP
Python:
callwith(string var, python my).

def callwith(var,my):
        … my is the python object that was stored with getmyvar

//In XIP, we declare a variable
Variables:
python myexchange; //we declare a XIP variable with the type python.
Script:
 getmyvar("myexchange"); //now myexchange will point on a python data, it is passed as a string…
 callwith("toto",myexchange); //here we use it directly…
```

N.B XIP already provides a pre-defined python variable: *python_exchange_data*, which is used to store a python object passed as a *data* parameter in *parsestring* and *parsefile…*

xips.setint(index,v,i)

This method sets the variable *v* with the value *i* for the grammar *index*. It returns:

-1, if the grammar does not exist

0 if the variable has not been declared within the grammar,

1 if it worked.

Example:

```
xips.setint(index,”toto”,5)
```

toto should have been declared as a *int* in the grammar.

xips.setfloat(index,v,f)

This method sets the variable *v* with the value *f* for the grammar *index*. It returns:

-1, if the grammar does not exist

0 if the variable has not been declared within the grammar,

1 if it worked.

Example:

```
xips.setfloat(index,”toto”,5)
```

toto should have been declared as a *float* in the grammar.

xips.setstring(index,v,s)

This method sets the string variable *v* with the value *s* for the grammar *index*. It returns:

-1, if the grammar does not exist

0 if the variable has not been declared within the grammar,

1 if it worked.

Example:

```
xips.setstring(index,”toto”,”titi”)
```

toto should have been declared as a *string* in the grammar.

xips.parsestring(index,str,data)

In XIP, more than one grammar can be loaded at time. Each is identified by a unique ID which is the position of the grammar in the loading sequence. In other words, the first grammar to be loaded is identified with the index 0, the second one with the index 1, the third one with the index 2 and so on.

This method applies the grammar whose index is *index* to the string *str.* The index of the grammar should be different from the current grammar, otherwise an error code is returned:

1. *-1* means that the parameters could not be analyzed
2. *-2* means that the index does not correspond to a grammar in memory
3. *-3* means that this grammar is already active and used to parse the current sentence.

If no error occurs, then this procedure returns a string containing the analysis of the string.

The *data* variable has no pre-defined type and may be used by the programmer to exchange data with the XIP grammar. This *data* variable is stored into the XIP variable: *python_exchange_data.*

N.B. XIP is not exactly reentrant which means that the current grammar cannot be used to parse another sentence. It imposes on the user to load more than one grammar at a time, which can then be used to parse a sentence, within a parse.

xips.parsefile(index,filename,data)

This method applies the grammar whose index is *index* to the file *filename.* The index of the grammar should be different from the current grammar, otherwise an error code is returned:

4. *-1* means that the parameters could not be analyzed
5. *-2* means that the index does not correspond to a grammar in memory
6. *-3* means that this grammar is already active and is used to parse the current sentence.

If no error occurs, then this procedure returns a string containing the complete analysis of the file.

The *data* variable has no pre-defined type and may be used by the programmer to exchange data with the XIP grammar. This *data* variable is stored into the XIP variable: *python_exchange_data.*

N.B XIP is not exactly reentrant which means that the current grammar cannot be used to parse another sentence. It imposes on the user to load more than one grammar at a time, which can then be used to parse a file, within the current parse.

- **xips.parsexmlstring(index,str,depth,data)**

This method applies the grammar on an XML string. The *depth* parameter indicates at which XML depth the string should analyzed (see XML parsing for more information)

- **xips.parsexmlfile(index,filename,depth,data)**

This method applies the grammar on an XML file. The *depth* parameter indicates at which XML depth the string should analyzed (see XML parsing for more information)

- **xips.currentrule()**

This method returns the current rule information. It returns a 8 element list: [idrule, numberrule, layer,rule_type,label,pathname, position, line].

- **xips.currentruletext(displayheader)**

This method returns the text of the current rule. It returns a string according to the value of *displayheader (value =0, 1 or 2).*

- **xips.loadkif(filename,arguments,mapping)**

This method loads a KiF program, with some arguments. The arguments are a string in which each argument is separated from the others with a space character. The “mapping” is optional; it defines whether the functions in the KiF program are mapped as Python functions.

It returns a kif handler, which should be used with “xips.kif”

- **xips.kif(kifhandler,“functionname”,p1,p2…)**

This method executes a KiF function, which has been loaded with *loadkif*, with as many parameters as necessary. The implementation ensures that the KiF objects: int, string, long, float Boolean, map and vectors are mapped over corresponding Python objects, for both arguments and the return value. If “mapping” has been set to 1, then

the call to this method can be replaced by a direct call to *functioname(p1,p2..).*

Example:

```
KiF  program: myprogram.kif

automaton au;

au.load('englishlexiconmin.txt');

function check(string ch) {
    return (au.editdistance(ch,2, a_change|a_insert|a_delete));
}


Python Call:
>> xips.loadkif("myprogram.kif","",1)
>> check("embarassed")
>> [{'embarrassed': 1.0, 'embarrasser': 2.0, 'embarrasses': 2.0}]
```

Python Class: XipNode

XIP provides a python class to access the information of a given syntactic node.

```
    import xips

    class XipNode:

     def getnode(self,i):
            #xips.node returns a list of values for a given node
      liste=xips.node(i)
            self.index=i
      self.pos=liste[1]
      self.surface=liste[2]
      self.lemma=liste[3]
      self.features=liste[4]
      self.parent=liste[5]
      self.daughter=liste[6]
      self.last=liste[7]
      self.next=liste[8]
      self.previous=liste[9]
      self.left=liste[10]
      self.right=liste[11]
            self.leftoffset=liste[12]
      self.rightoffset=liste[13]
```

```
        self.leftoffsetchar=liste[14]
    self.rightoffsetchar=liste[15]


#Various methods to send the nodes connected to the current node
def Next(self):
 return XipNode(self.next)

def Previous(self):
 return XipNode(self.previous)

def Last(self):
 return XipNode(self.last)

def Parent(self):
 return XipNode(self.parent)

def Daughter(self):
 return XipNode(self.daughter)

#when creating a new instance, we also call getnode
def __init__(self,i):
   self.surface=""
   self.lemma=""
   self.pos=""
   self.features=""
   self.parent=-1
   self.daugther=-1
   self.last=-1
   self.next=-1
   self.previous=-1
        self.index=-1
        self.left=0
        self.right=0
        self.leftoffset=-1
        self.rightoffset=-1
        self.leftoffsetchar=-1
        self.rightoffsetchar=-1
        self.getnode(i)
```

We use the *xips* module and the *xipnode* method to retrieve a list of linguistic data that we transform into an object. The *XipNode* python class extracts the information from that list and instantiates the different class variables. These variables are the following:

1. *index* is the index of the node itself
2. *surface* is the surface form of the syntactic node

3. *lemma* is the lemma form of the syntactic node
4. *pos* is the part of speech of the syntactic node
5. *features* is a string that contains a list of features with their values
6. *parent* is the index of the parent node
7. *daughter* is the index of the first daughter node
8. *last* is the index of the last daughter node
9. *next* is the index of the next node
10. *previous* is the index of the previous node
11. *left* is the left node number
12. *right* is the right node number
13. *leftoffset* is the left node offset in byte
14. *righoffset* is the right node offset in byte
15. *lefoffsetchar* is the left node offset in character
16. *righoffsetchar* is the right node offset in character

Example

node contains the index of a given syntactic node
#first we create an instance of the class XipNode, to get the syntactic information of node*.*
xnode = XipNode(node)
xnode contains now all the information of node,
xnext = XipNode(xnode.next)
#xnext contains now all the information of the sister node of xnode*.*

IMPORTANT

This class is automatically imported by XIP. It is therefore always available to any python script, executed from within a XIP script.

Python Class: XipDependency

The XIP python interface also provides a dependency class which can be used to retrieve some information about a dependency.

```
class XipDependency:

 def getdependency(self,i):
  if i==-1:
   return 0
       self.index=i
  liste=xips.dependency(i)
  self.name=liste[0]
  self.features=liste[1]
  i=2
  j=0
  while (i<len(liste)):
```

```
        self.parameters.insert(j,liste[i])
        j=j+1
        i=i+1

    def __init__(self,i):
        self.name=""
        self.features=""
        self.parameters=[]
            self.index=-1
            self.getdependency(i)
```

We use as for the previous example, the *xips* module to extract a list of values that we dispatch over our class variables. The definition of each variable is the following:

1. *name* is the name of the dependency
2. *features* is a string that contains a concatenation of *attribute:value* pairs.
3. *index* is the index of that dependency
4. *parameters* is a list of integers, each corresponding to the index of a syntactic node or of a dependency. To obtain the information about a node, uses the XipNode class with one of this value. In the case of a dependency, uses the XipDependency class with this value.

A Real Life Example

The first step consists in creating an interface.

Python: *//domain keyword*

caller(#1,#2,string _str, int i, $1,vector vect). *//The interface*

```
#From now on, all that follows is python
def caller(node1,node2,l,refdep,vectorvalues)
        global _str
        #we get the linguistic information for each of these nodes
        xnode1=XipNode(node1)
        xnode2= XipNode(node2)
        nodenext=node2.Next()
        #we access the linguistic information about the dependency
        xdep = XipDependency(refdep)
        #we store the surface form of node1 in _str
        _str=xnode1.surface
        #we append the lemma of node2 to the vector vectorvalues
        vectorvalues.append(xnode2.lemma)
```

```
#we print some values on screen
print xnode1.lemma, xnode2.surface, _str, xdep.name
#we returns the size of the list
return len(vectorvalues)
```

Explanation

The first important point is the name convention. The name of interface must match the name of one of the python script functions. If there is not such a match, then XIP stops and returns an error message. In this example, both functions are named: *caller.*

The second point is the parameter convention.

1. For each *#1,#2,…,#n* there should be a corresponding parameter in the python function. The matching python parameter is an integer. For each of these two indexes, we create two *XipNode* objects that contain all the linguistic information matching these two node indexes.

   IMPORTANT: *The node parameters should always start the list of arguments*.

2. The global variables, in our example *_str do not appear in the arguments of the python function.* They should be declared with the same name as a *global variable in the body of the python function.*

3. The other parameters are simply declared in the argument list of the python function, with whatever name the programmer would choose. Note that *list variables are never declared as global, even though their values might be modified in the python script.*

Calling a python script from within a grammar

To call a python script from the grammar, the user must use the interface. If the python script returns a value (it can only be a numerical value), then this value will also be returns by the interface itself.

Example

Variables:

```
string s;
int l;
int length;
```

```
        vector dict;

Script:
        s="call";
        l=2;
        |Det#1,?*,Noun#2| if (subject$1(#3,#2) {
                //this function will display some information send to the python script
                 caller(#1,#2,s,l, $1,dict);
        }

        //After the execution, dict contains the list of noun lemmas.
        for (l=0;l<length;l=l+1) {
                print(dict[l]+"\n");
        }

        //After the execution, s contains the surface of the first node
        print(s);
```

Probabilities

XIP provides some mechanisms to handle probabilities with any sorts of rules. Each rule is automatically associated with three numerical (float) values: *weight, threshold* and *value.* Each of these values can be individually instantiated from within or out of the grammar.

By default, any rule whose *value* is superior or equal to the *threshold* is triggered. The default values for each rule are the following:

a) weight=1
b) threshold=0
c) value=1

The user can also trigger a specific mechanism, which automatically computes a new *value* out of the combination of the *weight* with a *random* value. In this case, the *weight should be provided as a value between [0..1], while the threshold should be set to a value between [0..100].* The *value* which is computed by the internal randomizer will be between *[0..100].*

Important: *Value* will be computed only once for a full parse either of a string or of a text.

To trigger this mechanism the user can call XIP with the flag *–random*.

If the user utilizes the XIP API, then the following flags should be set:

Python: XIP_RANDOM_ANALYSIS

Instructions

XIP provides instructions to handle these different values at three different levels: from within, with specific scripts instructions, or from C++ and Python API.
At the end of parse, it is possible to *store* for each rule, the different *weights, thresholds or values that have been computed in a file.* This file can be reloaded anytime as an *addendum* or *parameter* file. For instance, an *Addendum* section in a GRM file can store the path of a probability file.

The instructions for Python are the following:

Python API

Python offers exactly the same possibilities as C++, with of course a different set of instructions:

setruleweight(g,id,w)	set the weight to rule id
setrulethreshold(g,id,w)	set the threshold to rule id
setrulevalue(g,id,w)	set the trigger value to rule id
getruleweight(g,id)	return weight of rule id
getrulethreshold(g,id)	return threshold of rule id
getrulevalue(g,id)	return trigger value of rule id
rulelayer(g,id)	return the layer number of rule ID
ruletype(g,id)	return the type of rule ID
rulecounter(g,id)	return the number of occurrence of rule ID
ruletypestr(g,id)	return the type of rule ID as a string
nbrules(g)	return the number of rules
loadprobabilities(g,filename)	load a probability model stored in filename
saveprobabilities(g,filename)	save a probability model stored in filename

In the case of Python, the randomizer should be set with: XIP_RANDOM_ANALYSIS through a call to setdisplay (or addisplay) instruction.