NatLink:
A Python Macro System for Dragon NaturallySpeaking

Joel Gould
Director of Emerging Technologies
Dragon Systems

Copyright Information

- This is version 1.1 of this presentation
  - Changes: look in corner of slides for V 1.1 indication
- This version of the presentation was given to the Voice Coder's group on June 25, 2000
-
-
- Contact Joel Gould for more information
joelg@alum.mit.edu

Outline of Today's Talk

- **Introduction**
- Getting started with NatLink
- Basics of Python programming
- Specifying Grammars
- Handling Recognition Results
- Controlling Active Grammars
- Examples of advanced projects
- Where to go for more help

What is NaturallySpeaking?

- World's first and best large vocabulary continuous speech recognition system
- Primarily designed for dictation by voice
- Also contains fully functional continuous command recognition (based on SAPI 4)
- Professional Edition includes simple basic-like language for writing simple macros

What is Python?

- Interpreted, object-oriented pgm. language
- Often compared to Perl, but more powerful
- Free and open-source, runs on multiple OSs
- Ideal as a macro language since it is interpreted and interfaces easily with C
- Also used for web programming, numeric programming, rapid prototyping, etc.

What is NatLink?

- A compatibility module (like NatText):
  - NatLink allows you to write NatSpeak command macros in Python
- A Python language extension:
  - NatLink allows you to control NatSpeak from Python
- Works with all versions of NatSpeak
- Free and open-source, freely distributable*

*Licensing Restrictions

- NatLink requires that you have a legally licensed copy of Dragon NaturallySpeaking

- To use NatLink you must also agree to the license agreement for the NatSpeak toolkit
  - Soon Natlink will require the NatSpeak toolkit
  - The NatSpeak toolkit is a free download from http://www.dragonsys.com

V 1.1

NatLink is Better than Prof. Ed.

- Grammars can include alternates, optionals, repeats and nested rules
- Can restrict recognition to one grammar
- Can change grammars at start of any recog.
- Can have multiple macro files
- Changes to macro files load immediately
- Macros have access to all features of Python

NatLink is Harder to Use

- NatLink is not a supported product
  Do not call Tech Support with questions
- NatLink *may* not work with NatSpeak > 5
  - It will work fine with NatSpeak 5.0 **V 1.1**
- Documentation is not complete
- No GUI or fancy user interface
- Requires some knowledge of Python
- More like real programming

Outline of Today's Talk

- Introduction
- **Getting started with NatLink**
- Basics of Python programming
- Specifying Grammars
- Handling Recognition Results
- Controlling Active Grammars
- Examples of advanced projects
- Where to go for more help

What you Need to Install

- Dragon NaturallySpeaking
  - Any edition, version 3.0 or better
- Python 1.5.2 for Windows:
  py152.exe from http://www.python.org/
  - You do not need to install Tcl/Tk
- NatLink: natlink.zip from
  http://www.synapseadaptive.com/joel/default.htm
- Win32 extensions are optional:
  win32all.exe from http://www.python.org/

Setting up NatLink

- Install NatSpeak and Python
- Unzip natlink.zip into c:\NatLink
- Run \NatLink\MacroSystem\EnableNL.exe
  - This sets the necessary registry variables
  - This also turns NatLink on or off
- To run sample macros, copy macro files
  - From: \NatLink\SampleMacros
  - To: \NatLink\MacroSystem

How to Create Macro Files

- Macro files are Python source files
- Use Wordpad or any other text editor
  - save files as text with .py extension
- Global files should be named _xxx.py
- App-specific files should be named with the application name (ex: wordpad_xxx.py)
- Copy files to \NatLink\MacroSystem
  - Or to \NatSpeak\Users\username\Current

Sample Example 1

- File _sample1.py contains one command
- Say “demo sample one” and it types:
  Heard macro “sample one”

Source Code for _sample1.py

```
import natlink
from natlinkutils import *

class ThisGrammar(GrammarBase):

    gramSpec = """
        <start> exported = demo sample one;
    """

    def gotResults_start(self,words,fullResults):
        natlink.playString('Heard macro "sample one"{enter}')

    def initialize(self):
        self.load(self.gramSpec)
        self.activateAll()

thisGrammar = ThisGrammar()
thisGrammar.initialize()

def unload():
    global thisGrammar
    if thisGrammar: thisGrammar.unload()
    thisGrammar = None
```

This is the grammar. You can say: "demo sample one"

This is the action. We type text into the active window.

Most of the rest of this file is boiler plate.

Sample Example 2

- Add a second command with alternatives
- Type (into application) the command and alternative which was recognized
- NatLink will tell you which rule was recognized by calling a named function
  - gotResults_firstRule for <firstRule>
  - gotResults_secondRule for <secondRule>

Extract from _sample2.py

```
# ...
class ThisGrammar(GrammarBase):

    gramSpec = """
        <firstRule> exported = demo sample two [ help ];
        <secondRule> exported = demo sample two
            ( red | blue | green | purple | black | white | yellow |
              orange | magenta | cyan | gray );
    """

    def gotResults_firstRule(self,words,fullResults):
        natlink.playString('Say "demo sample two {ctrl+i}color{ctrl+i}"{enter}')

    def gotResults_secondRule(self,words,fullResults):
        natlink.playString('The color is "%s" {enter}'%words[3])

    def initialize(self):
        self.load(self.gramSpec)
        self.activateAll()
# ...
```

This is the grammar. It has two rules.

What we do when "firstRule" is heard.

What we do when "secondRule" is heard. Words[3] is the 4th word in the result.

Outline of Today's Talk

- Introduction
- Getting started with NatLink
- **Basics of Python programming**
- Specifying Grammars
- Handling Recognition Results
- Controlling Active Grammars
- Examples of advanced projects
- Where to go for more help

Strings and Things

- String constants can use either single quote or double quotes

```
'This is a string'
"This string has a single quote (') inside"
```

- Use triple quotes for multiple line strings

```
"""line 1 of string
line 2 of string"""
```

- Plus will concatenate two strings

```
'one'+'two'='onetwo'
```

- Percent sign allows sprintf-like functions

```
'I heard %d' % 13 = 'I heard 13'
'the %s costs $%1.2f' % ('book',5) = 'the book costs $5.00'
```

Comments and Blocks

- Comments begin with pound sign

```
# Comment from here until end of line
print 'hello' # comment starts at pound sign
```

- Blocks are delimited by indentation, the line which introduces a block ends in a colon

```
if a==1 and b==2:
    print 'a is one'
    print 'b is two'
else:
    print 'either a is not one or b is not two'
x = 0
while x < 10:
    print x
    x = x + 1
print 'all done'
```

Lists and Loops

- Lists are like arrays; they are sets of things
- Uses brackets when defining a list

```
myList = [1,2,3]
another = ['one',2,myList]
```

- Use brackets to get or change a list element

```
print myList[1]    # prints 2
print another[2]   # prints [1,2,3]
```

- The “for” statement can iterate over a list

```
total = 0
for x in myList:
    total = total + x
print x      # prints 6 (1+2+3)
```

Defining and Calling Functions

- Use the “def” statement to define a function
- List the arguments in parens after the name

```
def globalFunction(x,y):
    total = x + y
    print 'the total is',total
```

- Example of a function call

```
globalFunction(4,7)  # this prints "the total is 11"
```

- Return statement is optional

```
def addNumbers(x,y)
    return x + y
print addNumbers(4,7)    # this prints "11"
```

Modules and Classes

- Call functions inside other modules by using the module name before the function

```
import string
print string.upper('word')
```

- Define classes with “class” statement and class functions with “def” statement

```
class MyClass:
    def localFunction(self,x):
            print 'value is x'
object = MyClass          # create instance of MyClass
object.localFunction(10)  # prints "value is 10"
```

Self and Class Inheritance

- "Self" param passed to class functions points back to that instance

```
class ParentClass:
    def sampleFunc(self,value):
            self.variable = value
    def parentFunc(self):
            self.sampleFunc(10)
            return self.variable      # returns 10
```

- You can also use "self" to reference functions in parent classes (inherence)

```
class ChildClass(ParentClass):
    def childFunc(self):
            print self.parentFunc()     # prints "10"
            print self.variable         # also prints "10"
```

Outline of Today's Talk

- Introduction
- Getting started with NatLink
- Basics of Python programming
- **Specifying Grammars**
- Handling Recognition Results
- Controlling Active Grammars
- Examples of advanced projects
- Where to go for more help

Introduction to Grammars

- NatLink grammars are based on SAPI
- Grammars include: rules, lists and words
  - distinguished by how they are spelled
  - <rule>, {list}, word, "word with space"
- Grammar specification is a set of rules
- A rule is combination of references to words, lists and other rules

```
<myRule> = one <subRule> and {number} ;
<subRule> = hundred | thousand ;
```

Specifying Rules

- NatLink compiles a set of rules when a grammar is loaded

```
def initialize(self):
    self.load(self.gramSpec)        # this compiles and load rules
    self.activateAll()
```

- Rules should be defined in a Python string

```
gramSpec = "<myRule> = one two three;"
gramSpec2 = """
      <ruleOne> = go to sleep;
      <ruleTwo> = wake up;
"""
```

- Define rules as rule-name, equal-sign, expression; end rule with a semicolon

Basic Rule Expressions

- Words in a sequence must spoken in order
  - <rule> = one two three;
  - Must say “one two three”
- Use brackets for options expressions
  - <rule> = one [two] three;
  - Can say “one two three” or “one three”
- Vertical bar for alternatives, parens to group
  - <rule> = one (two | three four) five;
  - Can say “one two five” or “one three four five”

Nested Rules and Repeats

- Rules can refer to other rules
  - <rule> = one <subRule> four;
  - <subRule> = two | three;
  - Can say “one two four” or “one three four”
- Use plus sign for repeats, one or more times
  - <rule> = one (two)+ three
  - Can say “one two three”, “one two two three”, “one two two two three”, etc.

Exported and Imported Rules

- You can only activate “exported” rules
  - <myRule> exported = one two three;
- Exported rules can also be used by other grammars; define external rule as imported
  - <myRule> imported;
  - <rule> = number <myRule>;
- NatSpeak defines three importable rules:
  - <dgnwords> = set of all dictation words
  - <dgndictation> = repeated dictation words
  - <dgnletters> = repeated spelling letters

Dealing with (Grammar) Lists

- Lists are sets of words defined later
- Referencing a list causes it to be created
  - <rule> = number {myList};
- Fill list with words using setList function

```
def initialize(self):
    self.load(self.gramSpec)
    self.setList('myList',['one','two','three'])      # fill the list
    self.activateAll()
```

  - You can now say “number one”, “number two” or “number three”

What is a Word?

- Words in NatSpeak and NatLink are strings
  - Words can have embedded spaces
  - “hello”, “New York”, “:-)”
- In NatLink grammars, use quotes around words if the word is not just text or numbers
- Grammar lists are lists of words
- For recognition, words from lists are returned just like words in rules

Special Word Spellings

- Words with separate spoken form are spelled with backslash: "written\spoken"
- Punctuation is most common example
  - ".\period"
  - "{\open brace"
- Letters are spelled with two backslashes
  - "a\\l", "b\\l", "c\\l", etc.

Grammar Syntax

- NatSpeak requires rules in binary format
  - Binary format is defined by SAPI and is documented in SAPI documentation
- Gramparser.py converts text to binary
- Rule syntax is described in gramparser.py
- NatSpeak also supports dictation grammars and “Select XYZ” grammars. These are covered in another talk.

Outline of Today's Talk

- Introduction
- Getting started with NatLink
- Basics of Python programming
- Specifying Grammars
- **Handling Recognition Results**
- Controlling Active Grammars
- Examples of advanced projects
- Where to go for more help

Getting Results

- When a rule is recognized, NatLink calls your function named “gotResults_xxx”
  - where “xxx” is the name of the rule
- You get passed the sequential words recognized in that rule
  - gotResults(self,**words**,fullResults)
- Function called for innermost rule only
  - consider the following example

Extract from _sample3.py

```
# ...
class ThisGrammar(GrammarBase):

    gramSpec = """
        <mainRule> exported = <ruleOne>;
        <ruleOne> = demo <ruleTwo> now please;
        <ruleTwo> = sample three;
    """

    def gotResults_mainRule(self,words,fullResults):
        natlink.playString('Saw <mainRule> = %s{enter}' % repr(words))

    def gotResults_ruleOne(self,words,fullResults):
        natlink.playString('Saw <ruleOne> = %s{enter}' % repr(words))

    def gotResults_ruleTwo(self,words,fullResults):
        natlink.playString('Saw <ruleTwo> = %s{enter}' % repr(words))

    def initialize(self):
# ...
```

“repr(x)” formats “x” into a printable string.

Running Demo Sample 3

- When you say “demo sample 3 now please”, resulting text sent to application is:

```
Saw <ruleOne> = ['demo']
Saw <ruleTwo> = ['sample', 'three']
Saw <ruleOne> = ['now','please']
```

- Rule “mainRule” has no words so gotResults_mainRule is never called
- gotResults_ruleOne is called twice, before and after gotResults_ruleTwo is called
- Each function only sees relevant words

Other gotResults Callbacks

- If defined, “gotResultsInit” is called first
- If defined, “gotResults” is called last
  - Both get passed all the words recognized
- Called functions from previous example:

```
gotResultsInit( ['demo','sample','three','now','please'] )
gotResults_ruleOne( ['demo'] )
gotResults_ruleTwo( ['sample','three'] )
gotResults_ruleOne( ['now','please'] )
gotResults( ['demo','sample','three','now','please'] )
```

Common Functions

- natlink.playString(keys) sends keystrokes
  - works just like “SendKeys” in NatSpeak Pro.
  - include special keystrokes in braces: “{enter}”
- natlink.setMicState(state) controls mic
  - where state is 'on', 'off' or 'sleeping'
  - natlink.getMicState() returns current state
- natlink.execScript(command) runs any built-in NatSpeak scripting command
  - natlink.execScript('SendKeys "{enter}"')

More Common Functions

- natlink.recognitionMimic(words) behaves as if passed words were “heard”

```
natlink.recognitionMimic(['Select','hello','there'])
```

  - works just like “HeardWord” in NatSpeak Pro.
- natlink.playEvents(list) to control mouse
  - pass in a list of windows input events
  - natlinkutils.py has constants and buttonClick()
- natlink.getClipboard() returns clipboard text
  - use this to get text from application

Mouse Movement _sample4.py

```
# ...
class ThisGrammar(GrammarBase):

    gramSpec = """
        <start> exported = demo sample four;
    """

    def gotResults_start(self,words,fullResults):
        # execute a control-left drag down 30 pixels
        x,y = natlink.getCursorPos()
        natlink.playEvents( [ (wm_keydown,vk_control,1),
                              (wm_lbuttondown,x,y),
                              (wm_mousemove,x,y+30),
                              (wm_lbuttonup,x,y+30),
                              (wm_keyup,vk_control,1) ] )

    def initialize(self):
        self.load(self.gramSpec)
        self.activateAll()
# ...
```

Press control key

Press left button

Move mouse

Get current mouse position

Release left button (at new position)

Release control key

Clipboard Example _sample5.py

```
# ...
class ThisGrammar(GrammarBase):

    gramSpec = """
        <start> exported = demo sample five
            [ (1 | 2 | 3 | 4) words ];
    """

    def gotResults_start(self,words,fullResults):
        # figure out how many words
        if len(words) > 3:
            count = int(words[3])
        else:
            count = 1
        # select that many words
        natlink.playString('{ctrl+right}{left}')
        natlink.playString('{ctrl+shift+left %d}'%count)
        natlink.playString('{ctrl+c}')
        text = natlink.getClipboard()
        # reverse the text
        newText = reverse(text)
        natlink.playString(newText)
# ...
```

If more than 3 words recognized, 4th word will be word count.

This selects previous "count" words

Copy selected text to clipboard, then fetch it

Reverse function defined later in file

Debugging and using Print

- If file is changed on disk, it is automatically reloads at start of utterance
- Turning on mic also looks for new files
- Python output is shown in popup window
  - Window automatically appears when necessary
- Python errors cause tracebacks in window
  - Correct file, toggle microphone to reload
- Use “print” statement to display debug info

Outline of Today's Talk

- Introduction
- Getting started with NatLink
- Basics of Python programming
- Specifying Grammars
- Handling Recognition Results
- **Controlling Active Grammars**
- Examples of advanced projects
- Where to go for more help

Global vs App-Specific

- Files whose name begins with underscore are always loaded; ex: _mouse.py
- Files whose name begins with a module name only load when that module is active
  - Ex: wordpad.py, excel_sample.py
- Once a file is loaded it is always active
- To restrict grammars:
  - test for active application at start of utterance
  - or, activate grammar for one specific window

Activating Rules

- Any exported rule can be activated
- GrammarBase has functions to activate and deactivate rules or sets of rules
  - self.activate(rule) - makes name rule active
  - self.activateAll() - activates all exported rules
- By default, activated rule is global
  - self.activate(rule,window=N) - activates a rule only when window N is active
- You can (de)activate rules at any time

Start of Utterance Callback

- If defined, “gotBegin” function is called at the start of every recognition
  - it gets passed the module information: module filename, window caption, window id
- The “window id” can be passed to activate()
- Use matchWindow() to test window title

```
if matchWindow(moduleInfo,'wordpad','font'):
    self.activate('fontRule',noError=1)
else:
    self.deactivate('fontRule',noError=1)
```

Prevents errors if rule is already (not) active.

Using Exclusive Grammars

- If any grammar is “exclusive” then only exclusive grammars will be active
- Allows you to restrict recognition
  - But you can not turn off dictation without also turning off all built-in command and control
- Use self.setExclusive(state), state is 0 or 1
  - Can also call self.activate(rule,exclusive=1)
- Any number of rules from any number of grammars can all be exclusive together

Activation Example _sample6.py

```
class ThisGrammar(GrammarBase):

    gramSpec = """
        <mainRule> exported = demo sample six [ main ];
        <fontRule> exported = demo sample six font;
    """

    def initialize(self):
        self.load(self.gramSpec)

    def gotBegin(self,moduleInfo):
        windowId = matchWindow(moduleInfo,'natspeak','Dragon')
        if windowId:
            self.activate('mainRule',window=windowId,noError=1
        windowId = matchWindow(moduleInfo,'natspeak','Font')
        if windowId:
            self.activate('fontRule',exclusive=1,noError=1)
        else:
            self.deactivate('fontRule',noError=1)
            self.setExclusive(0)
```

No activateAll() in initialize function !

Link <mainRule> to main window (has "Dragon" in title).

Turn on <fontRule> exclusively when window title contains "Font"

Otherwise, turn off <fontRule> and exclusiveness.

Activating Rules from a Table

- This is from my own Lotus Notes macros:

```
def gotBegin(self, moduleInfo):
    self.deactivateAll()
    captions = [
        ( 'New Memo -', 'newMemo' ),
        ( 'New Reply -', 'newReply' ),
        ( 'Inbox -', 'inbox' ),
        ( '- Lotus Notes', 'readMemo' ),
        ]
    for caption,rule_name in captions:
        winHandle = matchWindow(moduleInfo, 'nlnotes', caption)
        if winHandle:
            self.activate(rule_name, window=winHandle)
            return
```

Activate nothing by default

This table maps caption substring to rule-name to activate

Loop over table to find first window caption which matches

V 1.1

Outline of Today's Talk

- Introduction
- Getting started with NatLink
- Basics of Python programming
- Specifying Grammars
- Handling Recognition Results
- Controlling Active Grammars
- **Examples of advanced projects**
- Where to go for more help

Using OLE Automation

- You can use OLE Automation from Python with the Python Win32 extensions
- Using excel_sample7.py:
  - say “demo sample seven”
- Any cells which contain the name of colors will change to match that color

Extract from excel_sample7.py

```
class ThisGrammar(GrammarBase):

    gramSpec = """
        <start> exported = demo sample seven;
    """

    def initialize(self):
        self.load(self.gramSpec)

    def gotBegin(self,moduleInfo):
        winHandle=matchWindow(moduleInfo,'excel','Microsoft Excel')
        if winHandle:
            self.activateAll(window=winHandle)

    def gotResults_start(self,words,fullResults):
        application=win32com.client.Dispatch('Excel.Application')
        worksheet=application.Workbooks(1).Worksheets(1)
        for row in range(1,50):
            for col in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
                cell=worksheet.Range(col+str(row))
                if colorMap.has_key(cell.Value):
                    cell.Font.Color=colorMap[cell.Value]
                    cell.Borders.Weight = consts.xlThick
# ...
```

Activate grammar when we know window handle

OLE Automation code just like using Visual Basic

“colorMap” maps name of color to value (defined earlier)

Mouse Control in Python

- _mouse.py included in NatLink download
- Control mouse and caret like in DDWin:
  - "mouse down … slower … left … button click"
  - "move down … faster … stop"
- Uses exclusive mode to limit commands
- Uses timer callback to move the mouse

Implementing “Repeat That” 1

```
# ...
lastResult = None

class CatchAllGrammar(GrammarBase):

    gramSpec = """
        <start> exported = {emptyList};
    """

    def initialize(self):
        self.load(self.gramSpec,allResults=1)
        self.activateAll()

    def gotResultsObject(self,recogType,resObj):
        global lastResult
        if recogType == 'reject':
            lastResult = None
        else:
            lastResult = resObj.getWords(0)
# ...
```

This grammar is never recognized because list is empty

But, allResults flag means that gotResultsObject is called for every recognition

After every recognition, we remember what words were just recognized

V 1.1

Implementing “Repeat That” 2

```
class RepeatGrammar(GrammarBase):

    gramSpec = """
        <start> exported = repeat that
          [ ( 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
              10 | 20 | 30 | 40 | 50 | 100 ) times ];
    """

    def initialize(self):
        self.load(self.gramSpec)
        self.activateAll()

    def gotResults_start(self,words,fullResults):
        global lastResult
        if len(words) > 2: count = int(words[2])
        else: count = 1
        if lastResult:
            for i in range(count):
                natlink.recognitionMimic(lastResult)
# ...
```

Notice that the count is optional

The 3rd word in the result is the count

Use recognitionMimic to simulate the recognition of the same words; NatSpeak will test against active grammars or dictation as it the words were spoken.

V 1.1

Grammars with Dictation

```
class ThisGrammar(GrammarBase):

    gramSpec = """
        <dgndictation> imported;
        <ruleOne> exported = demo sample eight <dgndictation> [ stop ];
        <dgnletters> imported;
        <ruleTwo> exported = demo sample eight spell <dgnletters> [ stop ];
    """

    def gotResults_dgndictation(self,words,fullResults):
        words.reverse()
        natlink.playString(' ' + string.join(words))

    def gotResults_dgnletters(self,words,fullResults):
        words = map(lambda x: x[:1], words)
        natlink.playString(' ' + string.join(words, ''))

    def initialize(self):
        self.load(self.gramSpec)
        self.activateAll()

# ...
```

V 1.1

<dgndictation> is built-in rule for dictation.
Optional word ”stop” is never recognized.

<dgnletters> is built-in rule for spelling.
I had to add word “spell” or the spelling was confused with dictation in <ruleOne>

Outline of Today's Talk

- Introduction
- Getting started with NatLink
- Basics of Python programming
- Specifying Grammars
- Handling Recognition Results
- Controlling Active Grammars
- Examples of advanced projects
- **Where to go for more help**

NatLink Documentation

- \NatLink\NatLinkSource\NatLink.txt contains the documentation for calling the natlink module from Python
- Example macro files are all heavily documented; in \NatLink\SampleMacros
- Grammar syntax defined in gramparser.py
- GrammarBase defined in natlinkutils.py
  - also defines utility functions and constants

Where to Get More Help

- Joel's NatSpeak web site:
  http://www.synapseadaptive.com/joel/default.htm
- Python language web site:
  http://www.python.org/
- Books on Python
  - See Joel's NatSpeak site for recommendations
- NatPython mailing list:
  http://harvee.billerica.ma.us/mailman/listinfo/natpython
- Using COM from Python:
  Python Programming on Win32 by Mark Hammond

Looking at the Source Code

- NatLink source code included in download
- Source code is well documented
- Written in Microsoft Visual C++ 6.0
- Some features from Microsoft SAPI
  - get SAPI documentation from Microsoft
- Dragon-specific extensions not documented

All Done

"Microphone Off"