Dragonfly Documentation

Release 0.6.4

Christo Butcher

October 24, 2014

Contents

Dragonfly is a speech recognition framework. It is a Python package which offers a high-level object model and allows its users to easily write scripts, macros, and programs which use speech recognition.

It currently supports Dragon NaturallySpeaking (DNS) and Window Speech Recognition (WSR) as backend engines.

Contents:

CHAPTER 1

Introduction

Contents:

1.1 Features and target audience

This section gives a brief introduction into Dragonfly, its features, and the audience it's targeted at.

1.1.1 Features

Dragonfly was written to make it very easy for Python macros, scripts, and applications to interface with speech recognition engines. Its design allows speech commands and grammar objects to be treated as first-class Python objects. This allows easy and intuitive definition of complex command grammars and greatly simplifies processing recognition results.

Language object model The core of Dragonfly is based on a flexible object model for handling speech elements and command grammars. This makes it easy to define complex language constructs, but also greatly simplifies retrieving the semantic values associated with a speech recognition.

Support for multiple speech recognition engines Dragonfly's modular nature lets it use different speech recognition engines at the back end, while still providing a single front end interface to its users. This means that a program that uses Dragonfly can be run on any of the supported back end engines without any modification. Currently Dragonfly supports Dragon NaturallySpeaking and Windows Speech Recognition (included with Windows Vista).

Built-in action framework Dragonfly contains its own powerful framework for defining and executing actions. It includes actions for text input and key-stroke simulation.

1.1.2 Target audience

Dragonfly is a Python package. It is a library which can be used by people writing software that interfaces with speech recognition. Its main target audience therefore consists of programmers.

On the other hand, Dragonfly's high-level object model is very easy and intuitive to use. It is very rewarding for people without any prior programming experience to see their first small attempts to be rewarded so quickly by making their computer listen to them and speak to them. This is exactly how some of Dragonfly's users were introduced to writing software.

Dragonfly also offers a robust and unified platform for people using speech recognition to increase their productivity and efficiency. An entire repository of Dragonfly command-modules is available which contains command grammars for controlling common applications and automating frequent desktop activities.

1.2 Installation

This section describes how to install Dragonfly. The installation procedure of Dragonfly itself is straightforward. Its dependencies, however, differ depending on which speech recognition engine is used.

1.2.1 Prerequisites

To be able to use the dragonfly, you will need the following:

- **Python**, *v2.5 or later* – for example available from ActiveState.
- **Win32 extensions for Python** – already included in ActiveState's Python distribution or available from Mark Hammond's page.
- **Natlink** *(only for Dragon NaturallySpeaking users)* – for example available from Daniel Rocco.

1.2.2 Installation of Dragonfly

Dragonfly is a Python package. A simple installer of the *next, next, finish* type is available from the project's download page. This installer was created with Python's standard distutils and has been tested on Microsoft Windows XP and Vista.

Dragonfly's installer will install the library in your Python's local site-packages directory under the `dragonfly` subdirectory.

1.2.3 Installation for Dragon NaturallySpeaking

Dragonfly uses Natlink to communicate with DNS. Natlink is available in various forms, including Daniel Rocco's efficient and tidy pure-Python package. It is available here.

Once Natlink is up and running, Dragonfly command-modules can be treated as any other Natlink macro files. Natlink automatically loads macro files from a predefined directory. Common locations are:

- `C:\Program Files\NatLink\MacroSystem`
- `My Documents\Natlink`

At least one of these should be present after installing Natlink. That is the place where you should put Dragonfly command-modules so that Natlink will load them. Don't forget to turn the microphone off and on again after placing a new command-modules in the Natlink directory, because otherwise Natlink does not immediately see the new file.

1.2.4 Installation for Windows Speech Recognition

If WSR is available, then no extra installation needs to be done. Dragonfly can find and communicate with WSR using standard COM communication channels.

If you would like to use Dragonfly command-modules with WSR, then you must run a *loader* program which will load and manage the command-modules. A simple *loader* is available in the `dragonfly/examples/dragonfly-main.py` file. When run, it will scan the directory it's in for other `*.py` files and try to load them as command-modules.

1.3 Object model

The core of Dragonfly is a language object model revolving around three types: grammars, rules, and elements. This section describes that object model.

1.3.1 Grammars

A grammar is a collection of rules. It manages the rules, loading and unloading them, activating and deactivating them, and taking care of all communications with the speech recognition engine. When they recognition occurs, the associated grammar received the recognition event and dispatches it to the appropriate rule.

Normally a grammar is associated with a particular context or functionality. Normally the rules within a grammar are somehow related to each other. However, neither of these is strictly necessary, they are just common use patterns.

The `Grammar` class and derived classes are described in section *RefGrammarClasses*.

1.3.2 Rules

A rule represents a voice command or part of one. Its elements define exactly what its contents are, i.e. what can be said to activate this rule. It has several attributes which determine, for example, whether it is a top-level "standalone" command which can be spoken directly, or whether it can only be referenced from a different top level rule.

Each rule has one root element which defines the language content of the rule. Top-level rules have callback methods which are called when the rule is recognized.

1.3.3 Elements

Elements are the basic building blocks of the language model. The define exactly what can be said, inform the content of rules. The most common elements are:

- `Literal(...)` – one or more literal words.
- `Sequence(...)` – a series of other elements.
- `Alternative(...)` – a choice of other elements, only one of which can be said within a single recognition.
- `Optional(...)` – an element container which makes its single child element optional.
- `RuleRef(...)` – a reference to another rule.
- `ListRef(...)` – a reference to a list, which is a dynamic language element which can be updated and modified without reloading the grammar.
- `Dictation(...)` – a free-form dictation element which allows the speaker to say one or more natural language words.

The above mentioned element types are at the heart of Dragonfly's object model. But of course using them all the time to specify every grammar would be quite tedious. There is therefore also a special element which constructs these basic element types from a string specification:

- `Compound(...)` – a special element which parses a string spec to create a hierarchy of basic elements. The format of this spec is described *RefCompound*.

CHAPTER 2

Grammar sub-package

CHAPTER 3

Engines sub-package

Dragonfly supports multiple speech recognition engines as its backend. The *engines* sub-package implements the interface code for each supported engine.

3.1 EngineBase class

The `dragonfly.engines.engine_base.EngineBase` class forms the base class for this specific speech recognition engine classes. It defines the stubs required and performs some of the logic necessary for Dragonfly to be able to interact with a speech recognition engine.

To be continued: compiler classes, natlink and wsr implementation classes.

CHAPTER 4

Actions sub-package

The Dragonfly library contains an action framework which offers easy and flexible interfaces to common actions, such as sending keystrokes and emulating speech recognition. Dragonfly's actions sub-package has various types of these actions, each consisting of a Python class. There is for example a `dragonfly.actions.action_key.Key` class for sending keystrokes and a `dragonfly.actions.action_mimic.Mimic` class for emulating speech recognition.

Each of these actions is implemented as a Python class and this makes it easy to work with them. An action can be created (*defined what it will do*) at one point and executed (*do what it was defined to do*) later. Actions can be added together with the + operator to attend them together, thereby creating series of actions.

Perhaps the most important method of Dragonfly's actions is their `dragonfly.actions.action_base.ActionBase.execut` method, which performs the actual event associated with its action.

Dragonfly's action types are derived from the `dragonfly.actions.action_base.ActionBase` class. This base class implements standard action behavior, such as the ability to concatenate multiple actions and to duplicate an action.

4.1 Basic examples

The code below shows the basic usage of Dragonfly action objects. They can be created, combined, executed, etc.

```
from dragonfly.all import Key, Text

a1 = Key("up, left, down, right")    # Define action a1.
a1.execute()                         # Send the keystrokes.

a2 = Text("Hello world!")            # Define action a2, which
                                     #  will type the text.
a2.execute()                         # Send the keystrokes.

a4 = a1 + a2                         # a4 is now the concatenation
                                     #  of a1 and a2.
a4.execute()                         # Send the keystrokes.

a3 = Key("a-f, down/25:4")           # Press alt-f and then down 4 times
                                     #  with 25/100 s pause in between.
a4 += a3                             # a4 is now the concatenation
                                     #  of a1, a2, and a3.
a4.execute()                         # Send the keystrokes.

Key("w-b, right/25:5").execute()     # Define and execute together.
```

4.2 Combining voice commands and actions

A common use of Dragonfly is to control other applications by voice and to automate common desktop activities. To do this, voice commands can be associated with actions. When the command is spoken, the action is executed. Dragonfly's action framework allows for easy definition of things to do, such as text input and sending keystrokes. It also allows these things to be dynamically coupled to voice commands, so as to enable the actions to contain dynamic elements from the recognized command.

An example would be a voice command to find some bit of text:

- Command specification: `please find <text>`
- Associated action: `Key("c-f") + Text("%(text)s")`
- Special element: `Dictation("text")`

This triplet would allow the user to say *"please find some words"*, which would result in *control-f* being pressed to open the *Find* dialogue followed by *"some words"* being typed into the dialog. The *special element* is necessary to define what the dynamic element *"text"* is.

4.3 Action class reference

CHAPTER 5

Windows sub-package

Dragonfly includes several toolkits for non-speech user interface in. These include for example dialog GUIs, generic window control, and access to the Windows system clipboard.

Contents of Dragonfly's windows sub-package:

5.1 Clipboard toolkit

Dragonfly's clipboard toolkit offers easy access to and manipulation of the Windows system clipboard. The `dragonfly.windows.clipboard.Clipboard` class forms the core of this toolkit. Each instance of this class is a container with a structure similar to the Windows system clipboard, mapping content formats to content data.

5.1.1 Usage examples

An instance of something contains clipboard data. The data stored within an instance can be transferred to and from the Windows system clipboard as follows: (before running this example, the text "asdf" was copied into the Windows system clipboard)

```
>>> from dragonfly.windows.clipboard import Clipboard
>>> instance = Clipboard()         # Create empty instance.
>>> print instance
Clipboard()

>>> instance.copy_from_system()    # Retrieve from system clipboard.
>>> print instance
Clipboard(unicode=u'asdf', text, oemtext, locale)
>>> # The line above shows that *instance* now contains content for
>>> #  4 different clipboard formats: unicode, text, oemtext, locale.
>>> #  The unicode format content is also displayed.

>>> instance.copy_to_system()      # Transfer back to system clipboard.
```

The situation frequently occurs that a developer would like to use the Windows system clipboard to perform some task without the data currently stored in it being lost. This backing up and restoring can easily be achieved as follows:

```
>>> from dragonfly.windows.clipboard import Clipboard
>>> # Initialize instance with system clipboard content.
... original = Clipboard(from_system=True)
>>> print original
Clipboard(unicode=u'asdf', text, oemtext, locale)
```

```
>>> # Use the system clipboard to do something.
... temporary = Clipboard({Clipboard.format_unicode: u"custom content"})
>>> print temporary
Clipboard(unicode=u'custom content')
>>> temporary.copy_to_system()
>>> from dragonfly.all import Key
>>> Key("c-v").execute()

>>> # Restore original system clipboard content.
... print Clipboard(from_system=True) # Show system clipboard contents.
Clipboard(unicode=u'custom content', text, oemtext, locale)
>>> original.copy_to_system()
>>> print Clipboard(from_system=True) # Show system clipboard contents.
Clipboard(unicode=u'asdf', text, oemtext, locale)
```

5.1.2 Clipboard class

CHAPTER 6

Miscellaneous

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*