Dragonfly Documentation

Release 0.6.6b1

Christo Butcher

2017-04-02

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 the following speech recognition engines:

- *Dragon NaturallySpeaking* (DNS), a product of *Nuance*
- *Windows Speech Recognition* (WSR), included with Microsoft Windows Vista, Windows 7, and freely available for Windows XP

Dragonfly's documentation is available online at Read the Docs. Dragonfly's FAQ is available at Stackoverflow. Dragonfly's mailing list/discussion group is available at Google Groups.

CHAPTER 1

Documentation

Besides this page, the following documentation is also available:

Introduction

Contents:

Features and target audience

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

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.

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.

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.

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.

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.

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.

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.

Related resources

Demonstrations

The following demonstrations show how various people use Dragonfly and speech recognition:

- 2013-03-20, Tavis Rudd: Using Python to Code by Voice — A demonstration video showing off his use of Dragonfly at PyCon 2013
- 2010-02-02, Tim Harper: Dragonfly Tutorial — Instruction video showing how to install Dragonfly and how to develop command modules for it
- 2009-11-08, John Graves: Python No Hands with Dragonfly — Demonstration video presented at Kiwi PyCon 2009 showing how to program without touching a keyboard or mouse
- 2009-04-09: Dragonfly Mini-Demo of Continuous Command Recognition — A demonstration video showing the use of continuous command recognition

Community

The following resources are used by the Dragonfly community and speech recognition users/developers:

- Dragonfly Speech Google Group
- Dragonfly FAQ at Stackoverflow
- Hands-Free Coding Blog — James Stout's blog on his experiences with Dragonfly and other tools for doing hands-free technical computer work
- Speech Computing Forum on Dragonfly — Archive.org snapshot from 2013-12-07, before this forum was taken down

Applications

The following applications integrate Dragonfly for speech recognition:

- Damselfly — Tristen Hayfield's system for using Dragonfly voice commands to control Linux apps
- Dictation Toolbox — A collection of tools for speech recognition, including:
  - Aenea — A client-server library for using voice commands from a Windows system running Dragonfly to one or more other systems, e.g. running Linux
- Pyrson — Len Boyette's Digital Life Assistant (DLA) linking several Python libraries, including Dragonfly for speech recognition
- SublimeSpeech — A Dragonfly-based speech recognition plug-in for Sublime Text 2
- Speechcoder — A plug-in for Notepad++ to facilitate writing code

Command modules

The following sources offer a wide variety of command modules for use with Dragonfly:

- Barry Sims's Voice Coding Grammars
- Cesar Crusius' command modules
- Davitenio's command modules
- Dictation Toolbox' Dragonfly Scripts
- Dragonfly::Sorcery — David Conway's Dragonfly resources
- Simian Hacker's Code-by-Voice — Support files for Chris Cowan's "code by voice" setup using Dragon NaturallySpeaking and Dragonfly
- WhIzz2000's command modules
- Designing Dragonfly grammars — A blog post by James Stout discussing techniques to design command grammars

Forks

The following repositories contain forks of the Dragonfly source code, made by various people for them to improve it or experiment with it:

- Hawkeye Parker's repo
- Tylercal's repo

Unrelated

The following speech recognition resources are unrelated to Dragonfly, but may be interesting to its users nevertheless:

- PySpeech — A small Python module for interfacing with WSR
- Pastebin document containing explanations of the Natlink API
- Unimacro
- Vocola
- DragonControl — Nicholas Riley's scripts for using Dragon Medical under Windows 7 in VMware Fusion as a dictation buffer for OS X

Object model

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

Grammar sub-package

Dragonfly's core is a language object model containing the following objects:

- *Grammars* – these represent collections of *rules*.
- *Rules* – these implement complete or partial voice commands, and contain a hierarchy of *elements*.

- *Elements* – these form the language building blocks of voice commands, and represent literal words, element sequences, references to other rules, etc.

To illustrate this language model, we discuss an example grammar which contains 2 voice commands: **"command one"** and **"(second command | command two) [test]"**.

- ***Grammar*: container for the two voice commands**
  - ***Rule*: first voice command rule "command one"**
    - *Literal element*: element for the literal words **"command one"**. This element is the root-element of the first command rule
  - ***Rule*: second voice command rule "(second command | command two) [test]"**
    - ***Sequence element*: root-element of the second command rule**
      - ***Alternative element*: first child element of the sequence**

        Literal element: element for the literal words **"second command"**

        Literal element: element for the literal words **"command two"**
      - ***Optional element*: second child element of the sequence**

        Literal element: element for the literal words **"test"**

All of these different objects are described below and in subsections.

Grammar classes

Recognition callbacks

The speech recognition engine processes the audio it receives and calls the following methods of grammar classes to notify them of the results:

- `Grammar.process_begin()`: Called when the engine detects the start of a phrase, e.g. when the user starts to speak. This method checks the grammar's context and activates or deactivates its rules depending on whether the context matches.
- `Grammar._process_begin()`: Called by `Grammar.process_begin()` allowing derived classes to easily implement custom functionality without losing the context matching implemented in `Grammar.process_begin()`.
- `Grammar.process_recognition()`: Called when recognition has completed successfully and results are meant for this grammar.
- `Grammar.process_recognition_other()`: Called when recognition has completed successfully, but the results are not meant for this grammar.
- `Grammar.process_recognition_failure()`: Called when recognition was not successful, e.g. the microphone picked up background noise.

The last three methods are not defined for the base Grammar class. They are only called if they are defined for derived classes.

Grammar class

class **`Grammar`**(*name*, *description=None*, *context=None*, *engine=None*)

Grammar class for managing a set of rules.

This base grammar class takes care of the communication between Dragonfly's object model and the backend speech recognition engine. This includes compiling rules and elements, loading them, activating and deactivating them, and unloading them. It may, depending on the engine, also include receiving recognition results and dispatching them to the appropriate rule.

- *name* – name of this grammar
- *description* (str, default: None) – description for this grammar
- *context* (Context, default: None) – context within which to be active. If *None*, the grammar will always be active.

_process_begin(*executable*, *title*, *handle*)
Start of phrase callback.

This usually is the method which should be overridden to give derived grammar classes custom behavior.

This method is called when the speech recognition engine detects that the user has begun to speak a phrase. This method is called by the `Grammar.process_begin` method only if this grammar's context matches positively.

Arguments:

- *executable* – the full path to the module whose window is currently in the foreground.
- *title* – window title of the foreground window.
- *handle* – window handle to the foreground window.

disable()
Disable this grammar so that it is not active to receive recognitions.

enable()
Enable this grammar so that it is active to receive recognitions.

enabled
Whether a grammar is active to receive recognitions or not.

engine
A grammar's SR engine.

enter_context()
Enter context callback.

This method is called when a phrase-start has been detected. It is only called if this grammar's context previously did not match but now does match positively.

exit_context()
Exit context callback.

This method is called when a phrase-start has been detected. It is only called if this grammar's context previously did match but now doesn't match positively anymore.

lists
List of a grammar's lists.

load()
Load this grammar into its SR engine.

loaded
Whether a grammar is loaded into its SR engine or not.

name
A grammar's name.

process_begin(*executable*, *title*, *handle*)
Start of phrase callback.

Usually derived grammar classes override ''Grammar._process_begin'' instead of this method, because this method merely wraps that method adding context matching.

This method is called when the speech recognition engine detects that the user has begun to speak a phrase.

Arguments:

- *executable* – the full path to the module whose window is currently in the foreground.
- *title* – window title of the foreground window.
- *handle* – window handle to the foreground window.

rules
List of a grammar's rules.

unload()
Unload this grammar from its SR engine.

ConnectionGrammar class

class ConnectionGrammar(*name*, *description=None*, *context=None*, *app_name=None*)
Grammar class for maintaining a COM connection well within a given context. This is useful for controlling applications through COM while they are in the foreground. This grammar class will take care of dispatching the correct COM interface when the application comes to the foreground, and releasing it when the application is no longer there.

- `name` – name of this grammar.
- `description` – description for this grammar.
- `context` – context within which to maintain the COM connection.
- `app_name` – COM name to dispatch.

application
COM handle to the application.

connection_down()
Method called immediately after exiting this instance's context and disconnecting from the application.

By default this method doesn't do anything. This method should be overridden by derived classes if they need to clean up after disconnection.

connection_up()
Method called immediately after entering this instance's context and successfully setting up its connection.

By default this method doesn't do anything. This method should be overridden by derived classes if they need to synchronize some internal state with the application. The COM connection is available through the `self.application` attribute.

Rules

This section describes the following classes:

- *`dragonfly.grammar.rule_base.Rule`* – the base rule class
- *`dragonfly.grammar.rule_compound.CompoundRule`* – a rule class of which the root element is a `dragonfly.grammar.element_compound.Compound` element.
- *`dragonfly.grammar.rule_mapping.MappingRule`* – a rule class for creating multiple spoken-form -> semantic value voice-commands.

Rule class

class **Rule**(*name=None*, *element=None*, *context=None*, *imported=False*, *exported=False*)

Rule class for implementing complete or partial voice-commands.

This rule class represents a voice-command or part of a voice- command. It contains a root element, which defines the language construct of this rule.

Constructor arguments:

- *name* (*str*) – name of this rule. If *None*, a unique name will automatically be generated.
- *element* (*Element*) – root element for this rule
- *context* (*Context*, default: *None*) – context within which to be active. If *None*, the rule will always be active when its grammar is active.
- *imported* (boolean, default: *False*) – if true, this rule is imported from outside its grammar
- *exported* (boolean, default: *False*) – if true, this rule is a complete top-level rule which can be spoken by the user. This should be *True* for voice-commands that the user can speak.

The *self._log* logger objects should be used in methods of derived classes for logging purposes. It is a standard logger object from the *logger* module in the Python standard library.

active

This rule's active state. (Read-only)

disable()

Disable this grammar so that it is never active to receive recognitions, regardless of whether its context matches or not.

element

This rule's root element. (Read-only)

enable()

Enable this grammar so that it is active to receive recognitions when its context matches.

enabled

This rule's enabled state. An enabled rule is active when its context matches, a disabled rule is never active regardless of context. (Read-only)

exported

This rule's exported status. See *Exported rules* for more info. (Read-only)

grammar

This rule's grammar object. (Set once)

imported

This rule's imported status. See *Imported rules* for more info. (Read-only)

name

This rule's name. (Read-only)

process_begin(*executable*, *title*, *handle*)

Start of phrase callback.

This method is called when the speech recognition engine detects that the user has begun to speak a phrase. It is called by the rule's containing grammar if the grammar and this rule are active.

The default implementation of this method checks whether this rule's context matches, and if it does this method calls `_process_begin()`.

Arguments:

- *executable* – the full path to the module whose window is currently in the foreground
- *title* – window title of the foreground window
- *handle* – window handle to the foreground window

process_recognition(*node*)
Rule recognition callback.

This method is called when the user has spoken words matching this rule's contents. This method is called only once for each recognition, and only for the matching top-level rule.

The default implementation of this method does nothing.

Note: This is generally the method which developers should override in derived rule classes to give them custom functionality when a top-level rule is recognized.

---

value(*node*)
Start of phrase callback.

This method is called to obtain the semantic value associated with a particular recognition. It could be called from another rule's *value()* if that rule references this rule. If also be called from this rule's *process_recognition()* if that method has been overridden to do so in a derived class.

The default implementation of this method returns the value of this rule's root element.

Note: This is generally the method which developers should override in derived rule classes to change the default semantic value of a recognized rule.

CompoundRule class

The CompoundRule class is designed to make it very easy to create a rule based on a single compound spec.

This rule class has the following parameters to customize its behavior:

- *spec* – compound specification for the rule's root element
- *extras* – extras elements referenced from the compound spec
- *defaults* – default values for the extras
- *exported* – whether the rule is exported
- *context* – context in which the rule will be active

Each of these parameters can be passed as a (keyword) arguments to the constructor, or defined as a class attribute in a derived class.

Example usage

The CompoundRule class can be used to define a voice-command as follows:

```
class ExampleRule(CompoundRule):

    spec = "I want to eat <food>"
    extras = [Choice("food", {
                              "(an | a juicy) apple": "good",
                              "a [greasy] hamburger": "bad",
                             }
                    )
              ]
```

```
    def _process_recognition(self, node, extras):
        good_or_bad = extras["food"]
        print "That is a %s idea!" % good_or_bad

rule = ExampleRule()
grammar.add_rule(rule)
```

Class reference

class **CompoundRule**(*name=None*, *spec=None*, *extras=None*, *defaults=None*, *exported=None*, *context=None*)

Rule class based on the compound element.

Constructor arguments:

- *name* (*str*) – the rule's name
- *spec* (*str*) – compound specification for the rule's root element
- *extras* (sequence) – extras elements referenced from the compound spec
- *defaults* (*dict*) – default values for the extras
- *exported* (boolean) – whether the rule is exported
- *context* (*Context*) – context in which the rule will be active

process_recognition(*node*)

Process a recognition of this rule.

This method is called by the containing Grammar when this rule is recognized. This method collects information about the recognition and then calls *self._process_recognition*.

•*node* – The root node of the recognition parse tree.

MappingRule class

The MappingRule class is designed to make it very easy to create a rule based on a mapping of spoken-forms to semantic values.

This class has the following parameters to customize its behavior:

- *mapping* – mapping of spoken-forms to semantic values
- *extras* – extras elements referenced from the compound spec
- *defaults* – default values for the extras
- *exported* – whether the rule is exported
- *context* – context in which the rule will be active

Each of these parameters can be passed as a (keyword) arguments to the constructor, or defined as a class attribute in a derived class.

Example usage

The MappingRule class can be used to define a voice-command as follows:

```
class ExampleRule(MappingRule):

    mapping  = {
                "[feed] address [bar]":                 Key("a-d"),
                "subscribe [[to] [this] feed]":         Key("a-u"),
```

```
                "paste [feed] address":                 Key("a-d, c-v, enter"),
                "feeds | feed (list | window | win)":   Key("a-d, tab:2, s-tab"),
                "down [<n>] (feed | feeds)":            Key("a-d, tab:2, s-tab, down:
↪%(n)d"),
                "up [<n>] (feed | feeds)":              Key("a-d, tab:2, s-tab, up:
↪%(n)d"),
                "open [item]":                          Key("a-d, tab:2, c-s"),
                "newer [<n>]":                          Key("a-d, tab:2, up:%(n)d"),
                "older [<n>]":                          Key("a-d, tab:2, down:%(n)d"),
                "mark all [as] read":                   Key("cs-r"),
                "mark all [as] unread":                 Key("cs-u"),
                "search [bar]":                         Key("a-s"),
                "search [for] <text>":                  Key("a-s") + Text("%(text)s\n
↪"),
               }
    extras   = [
                Integer("n", 1, 20),
                Dictation("text"),
               ]
    defaults = {
                "n": 1,
               }

    rule = ExampleRule()
    grammar.add_rule(rule)
```

Class reference

class **MappingRule**(*name=None*, *mapping=None*, *extras=None*, *defaults=None*, *exported=None*, *context=None*)

Rule class based on a mapping of spoken-forms to semantic values.

Constructor arguments:

- *name* (*str*) – the rule's name
- *mapping* (*dict*) – mapping of spoken-forms to semantic values
- *extras* (sequence) – extras elements referenced from the spoken-forms in *mapping*
- *defaults* (*dict*) – default values for the extras
- *exported* (boolean) – whether the rule is exported
- *context* (*Context*) – context in which the rule will be active

process_recognition(*node*)

Process a recognition of this rule.

This method is called by the containing Grammar when this rule is recognized. This method collects information about the recognition and then calls MappingRule._process_recognition.

•*node* – The root node of the recognition parse tree.

Element classes

Fundamental element classes

Dragonfly grammars are built up out of a small set of fundamental building blocks. These building blocks are implemented by the following *element* classes:

- *ElementBase* – the base class from which all other element classes are derived
- *Sequence* – sequence of child elements which must all match in the order given

- *Alternative* – list of possibilities of which only one will be matched
- *Optional* – wrapper around a child element which makes the child element optional
- *Repetition* – repetition of a child element
- *Literal* – literal word which must be said exactly by the speaker as given
- *RuleRef* – reference to a *dragonfly.grammar.rule_base.Rule* object; this element allows a rule to include (i.e. reference) another rule
- *ListRef* – reference to a `dragonfly.grammar.list.List` object

The following *element* classes are built up out of the fundamental classes listed above:

- *Dictation* – free-form dictation; this element matches any words the speaker says, and includes facilities for formatting the spoken words with correct spacing and capitalization
- *DictListRef* – reference to a `dragonfly.all.DictList` object; this element is similar to the `dragonfly.all.ListRef` element, except that it returns the value associated with the spoken words instead of the spoken words themselves

ElementBase class

class ElementBase(*name=None*, *default=None*)

Base class for all other element classes.

Constructor argument:

- *name* (*str*, default: *None*) – the name of this element; can be used when interpreting complex recognition for retrieving elements by name.

_copy_sequence(*sequence*, *name*, *item_types=None*)

Utility function for derived classes that checks that a given object is a sequence, copies its contents into a new tuple, and checks that each item is of a given type.

_get_children()

Returns an iterable of this element's children.

This method is used by the `children()` property, and should be overloaded by any derived classes to give the correct children element.

By default, this method returns an empty tuple.

children

Iterable of child elements. (Read-only)

decode(*state*)

Attempt to decode the recognition stored in the given *state*.

dependencies(*memo*)

Returns an iterable containing the dependencies of this element and of this element's children.

The dependencies are the objects that are necessary for this element. These include lists and other rules.

element_tree_string()

Returns a formatted multi-line string representing this element and its children.

gstring()

Returns a formatted grammar string of the contents of this element and its children.

The grammar string is of a format similar to that used by Natlink to define its grammars.

value(*node*)

Determine the semantic value of this element given the recognition results stored in the *node*.

Argument:

- *node* – a `dragonfly.grammar.state.Node` instance representing this element within the recognition parse tree

The default behavior of this method is to return an iterable containing the recognized words matched by this element (i.e. *node.words()*).

Sequence class

class **`Sequence`**(*children=()*, *name=None*, *default=None*)

Element class representing a sequence of child elements which must all match a recognition in the correct order.
Constructor arguments:

- *children* (iterable, default: *()*) – the child elements of this element
- *name* (*str*, default: *None*) – the name of this element

For a recognition to match, all child elements must match the recognition in the order that they were given in the *children* constructor argument.

Example usage: >>> from dragonfly.test import ElementTester >>> seq = Sequence([Literal("hello"), Literal("world")]) >>> test_seq = ElementTester(seq) >>> test_seq.recognize("hello world") ['hello', 'world'] >>> test_seq.recognize("hello universe") RecognitionFailure

`_get_children`()

Returns the child elements contained within the sequence.

`children`

Iterable of child elements. (Read-only)

`value`(*node*)

The *value* of a `Sequence` is a list containing the values of each of its children.

Alternative class

class **`Alternative`**(*children=()*, *name=None*, *default=None*)

Element class representing several child elements of which only one will match.
Constructor arguments:

- *children* (iterable, default: *()*) – the child elements of this element
- *name* (*str*, default: *None*) – the name of this element

For a recognition to match, at least one of the child elements must match the recognition. The first matching child is used. Child elements are searched in the order they are given in the *children* constructor argument.

`_get_children`()

Returns the alternative child elements.

`children`

Iterable of child elements. (Read-only)

`value`(*node*)

The *value* of an `Alternative` is the value of its child that matched the recognition.

Optional class

class **`Optional`**(*child*, *name=None*, *default=None*)

Element class representing an optional child element.
Constructor arguments:

- *child* (*ElementBase*) – the child element of this element
- *name* (*str*, default: *None*) – the name of this element

Recognitions always match this element. If the child element does match the recognition, then that result is used. Otherwise, this element itself does match but the child is not processed.

_get_children()
Returns the optional child element.

children
Iterable of child elements. (Read-only)

value(*node*)
The *value* of a `Optional` is the value of its child, if the child did match the recognition. Otherwise the *value* is *None*.

Repetition class

class Repetition(*child*, *min=1*, *max=None*, *name=None*, *default=None*)
Element class representing a repetition of one child element.
Constructor arguments:
- *child* (*ElementBase*) – the child element of this element
- *min* (*int*, default: *1*) – the minimum number of times that the child element must be recognized; may be 0
- *max* (*int*, default: *None*) – the maximum number of times that the child element must be recognized; if *None*, the child element must be recognized exactly *min* times (i.e. *max = min + 1*)
- *name* (*str*, default: *None*) – the name of this element

For a recognition to match, at least one of the child elements must match the recognition. The first matching child is used. Child elements are searched in the order they are given in the *children* constructor argument.

children
Iterable of child elements. (Read-only)

get_repetitions(*node*)
Returns a list containing the nodes associated with each repetition of this element's child element.
Argument:
- *node* (*Node*) – the parse tree node associated with this repetition element; necessary for searching for child elements within the parse tree

value(*node*)
The *value* of a `Repetition` is a list containing the values of its child.

The length of this list is equal to the number of times that the child element was recognized.

Literal class

class Literal(*text*, *name=None*, *value=None*, *default=None*)

children
Iterable of child elements. (Read-only)

dependencies(*memo*)
Returns an iterable containing the dependencies of this element and of this element's children.

The dependencies are the objects that are necessary for this element. These include lists and other rules.

RuleRef class

class **RuleRef**(*rule*, *name=None*, *default=None*)

children
Iterable of child elements. (Read-only)

ListRef class

class **ListRef**(*name*, *list*, *key=None*, *default=None*)

children
Iterable of child elements. (Read-only)

DictListRef class

class **DictListRef**(*name*, *dict*, *key=None*, *default=None*)

children
Iterable of child elements. (Read-only)

Dictation class

class **Dictation**(*name=None*, *format=True*, *default=None*)

children
Iterable of child elements. (Read-only)

dependencies(*memo*)
Returns an iterable containing the dependencies of this element and of this element's children.

The dependencies are the objects that are necessary for this element. These include lists and other rules.

Compound class

class **Compound**(*spec*, *extras=None*, *actions=None*, *name=None*, *value=None*, *value_func=None*, *elements=None*, *default=None*)

Choice class

class **Choice**(*name*, *choices*, *extras=None*, *default=None*)

Context classes

Dragonfly uses context classes to define when grammars and rules should be active. A context is an object with a `Context.matches()` method which returns *True* if the system is currently within that context, and *False* if it is not.

The following context classes are available:

- *Context* – the base class from which all other context classes are derived
- *AppContext* – class which based on the application context, i.e. foreground window executable, title, and handle

Logical operations

It is possible to modify and combine the behavior of contexts using the Python's standard logical operators:

logical AND `context1 & context2` – *all* contexts must match

logical OR `context1 | context2` – *one or more* of the contexts must match

logical NOT `~context1` – *inversion* of when the context matches

For example, to create a context which will match when Firefox is in the foreground, but only if Google Reader is *not* being viewed:

```
firefox_context = AppContext(executable="firefox")
reader_context = AppContext(executable="firefox", title="Google Reader")
firefox_but_not_reader_context = firefox_context & ~reader_context
```

Class reference

class AppContext (*executable=None*, *title=None*, *exclude=False*)

Context class using foreground application details.

This class determines whether the foreground window meets certain requirements. Which requirements must be met for this context to match are determined by the constructor arguments.

Constructor arguments:

- *executable* (*str*) – (part of) the path name of the foreground application's executable; case insensitive
- *title* (*str*) – (part of) the title of the foreground window; case insensitive

class Context

Base class for other context classes.

This base class implements some basic infrastructure, including what's required for logical operations on context objects. Derived classes should at least do the following things:

•During initialization, set *self._str* to some descriptive, human readable value. This attribute is used by the `__str__()` method.

•Overload the *Context.matches()* method to implement the logic to determine when to be active.

The *self._log* logger objects should be used in methods of derived classes for logging purposes. It is a standard logger object from the *logger* module in the Python standard library.

matches (*executable*, *title*, *handle*)

Indicate whether the system is currently within this context.

Arguments:

- *executable* (*str*) – path name to the executable of the foreground application
- *title* (*str*) – title of the foreground window
- *handle* (*int*) – window handle to the foreground window

The default implementation of this method simply returns *True*.

Note: This is generally the method which developers should overload to give derived context classes custom functionality.

Grammars

A *grammar* is a collection of rules. It manages the rules, loading and unloading them, activating and deactivating them, and it takes care of all communications with the speech recognition engine. When a recognition occurs, the associated grammar receives 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 the *Grammar classes* section.

Rules

Rules represent voice commands or parts of voice commands. Each rule has a single root *element*, the basis of a tree structure of elements defining how the rule is built up out of speakable parts. The element tree determines what a user must say to cause this rule to be recognized.

Exported rules

Rules can be exported or not exported. Whether a rule is exported or not is defined when the rule is created.

Only exported rules can be spoken directly by the user. In other words, they form the entry points into a grammar, causing things to happen (callbacks to be called) when appropriate words are recognized.

WSR distinguishes between top-level rules, which can be recognized directly, and exported rules, which can be referenced from rules in other grammars. NatLink doesn't allow inter-grammar rule referencing and uses exported to refer to directly recognizable rules. Dragonfly follows NatLink in functionality and terminology on this topic.

Properties of exported rules:

- Exported rules are known as a top-level rules for WSR (`SRATopLevel`).
- Exported rules can be spoken by the user directly.
- Exported rules can be referenced from other rules within the same grammar.
- Exported rules can be referenced from rules in other grammars (only possible for WSR).
- Exported rules can be enabled and disabled to receive recognitions or not (`enable()`, `disable()`).
- Exported rules have callbacks which are called when recognition occurs (`process_begin()`, `process_recognition()`).

Non-exported rules cannot be recognized directly but only as parts of other rules that reference them.

Properties of non-exported rules:

- Non-exported rules can't be spoken by the user directly.
- Non-exported rules can be referenced from other rules within the same grammar.
- Non-exported rules can't be referenced from rules in other grammars (never possible for DNS).

Imported rules

Rules can be imported, i.e. defined outside the grammar referencing them, or not imported, i.e. defined within the grammar. Whether a rule is imported or not is defined when the rule is created.

NatLink in general doesn't allow rules from one grammar to be imported into another grammar, i.e. inter-grammar rule referencing. However, it does provide the following three built-in rules which can be imported:

- `dgnletters` – All the letters of the alphabet for spelling
- `dgnwords` – All words active during dictation
- `dgndictation` – A special rule which corresponds to free-form dictation; imported by Dragonfly for its *`Dictation`* element

Elements

Elements are the basic building blocks of the language model. They define exactly what can be said and thereby form 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.

Engines sub-package

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

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.

class `EngineBase`
Base class for engine-specific back-ends.

`connect``()`
Connect to back-end SR engine.

`connection``()`
Context manager for a connection to the back-end SR engine.

`disconnect``()`
Disconnect from back-end SR engine.

`language`
Current user language of the SR engine.

mimic(*words*)
Mimic a recognition of the given *words*.

name
The human-readable name of this engine.

speak(*text*)
Speak the given *text* using text-to-speech.

Engine backends

SR back-end package for DNS and Natlink

get_engine()
Retrieve the Natlink back-end engine object.

is_engine_available()
Check whether Natlink is available.

SR back-end package for SAPI 5

get_engine()
Retrieve the Sapi5 back-end engine object.

is_engine_available()
Check whether SAPI is available.

Dictation container classes

Dictation container base class

This class is used to store the recognized results of dictation elements within voice-commands. It offers access to both the raw spoken-form words and be formatted written-form text.

The formatted text can be retrieved using *format()* or simply by calling `str(...)` on a dictation container object. A tuple of the raw spoken words can be retrieved using *words*.

class DictationContainerBase(*words*)
Container class for dictated words as recognized by the `Dictation` element.

This base class implements the general functionality of dictation container classes. Each supported engine should have a derived dictation container class which performs the actual engine- specific formatting of dictated text.

A dictation container is created by passing it a sequence of words as recognized by the backend SR engine. Each word must be a Unicode string.
Parameters **words** (*sequence-of-unicode*) – A sequence of Unicode strings.

format()
Format and return this dictation as a Unicode object.

words
Sequence of the words forming this dictation.

Dictation container class for Natlink

This class is derived from `DictationContainerBase` and implements dictation formatting for the Natlink and Dragon NaturallySpeaking engine.

class NatlinkDictationContainer(*words*)
Container class for dictated words as recognized by the `Dictation` element for the Natlink and DNS engine.

format()
Format and return this dictation.

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.execute()` 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.

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.
```

More examples

For more examples on how to use and manipulate Dragonfly action objects, please see the doctests for the *dragonfly.actions.action_base.ActionBase* here: *Action doctests*.

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.

Action class reference

ActionBase base class

class ActionBase
Base class for Dragonfly's action classes.

class Repeat(*count=None*, *extra=None*)
Action repeat factor.

Integer Repeat factors ignore any supply data:

```
>>> integer = Repeat(3)
>>> integer.factor()
3
>>> integer.factor({"foo": 4})  # Non-related data is ignored.
3
```

Named Repeat factors retrieved their factor-value from the supplied data:

```
>>> named = Repeat(extra="foo")
>>> named.factor()
Traceback (most recent call last):
  ...
ActionError: No extra repeat factor found for name 'foo' ('NoneType' object is␣
↪unsubscriptable)
>>> named.factor({"foo": 4})
4
```

Repeat factors with both integer count and named extra values set combined (add) these together to determine their factor-value:

```
>>> combined = Repeat(count=3, extra="foo")
>>> combined.factor()
Traceback (most recent call last):
  ...
ActionError: No extra repeat factor found for name 'foo' ('NoneType' object is␣
↪unsubscriptable)
>>> combined.factor({"foo": 4}) # Combined factors 3 + 4 = 7.
7
```

Key action

This section describes the *Key* action object. This type of action is used for sending keystrokes to the foreground application. Examples of how to use this class are given in *Example key actions*.

Keystroke specification format

The *spec* argument passed to the *Key* constructor specifies which keystroke events will be emulated. It is a string consisting of one or more comma-separated keystroke elements. Each of these elements has one of the following two possible formats:

Normal press-release key action, optionally repeated several times: [*modifiers* −] *keyname* [/ *innerpause*] [: *repeat*] [/ *outerpause*]

Press-and-hold a key, or release a held-down key: [*modifiers* −] *keyname* : *direction* [/ *outerpause*]

The different parts of the keystroke specification are as follows. Note that only *keyname* is required; the other fields are optional.

- *modifiers* – Modifiers for this keystroke. These keys are held down while pressing the main keystroke. Can be zero or more of the following:
  - `a` – alt key
  - `c` – control key
  - `s` – shift key
  - `w` – Windows key
- *keyname* – Name of the keystroke. Valid names are listed in *Key names*.
- *innerpause* – The time to pause between repetitions of this keystroke.
- *repeat* – The number of times this keystroke should be repeated. If not specified, the key will be pressed and released once.
- *outerpause* – The time to pause after this keystroke.
- *direction* – Whether to press-and-hold or release the key. Must be one of the following:
  - `down` – press and hold the key
  - `up` – release the key

  Note that releasing a key which is *not* being held down does *not* cause an error. It harmlessly does nothing.

Key names

- Lowercase letter keys: `a` or `alpha`, `b` or `bravo`, `c` or `charlie`, `d` or `delta`, `e` or `echo`, `f` or `foxtrot`, `g` or `golf`, `h` or `hotel`, `i` or `india`, `j` or `juliet`, `k` or `kilo`, `l` or `lima`, `m` or `mike`, `n` or `november`, `o` or `oscar`, `p` or `papa`, `q` or `quebec`, `r` or `romeo`, `s` or `sierra`, `t` or `tango`, `u` or `uniform`, `v` or `victor`, `w` or `whisky`, `x` or `xray`, `y` or `yankee`, `z` or `zulu`
- Uppercase letter keys: `A` or `Alpha`, `B` or `Bravo`, `C` or `Charlie`, `D` or `Delta`, `E` or `Echo`, `F` or `Foxtrot`, `G` or `Golf`, `H` or `Hotel`, `I` or `India`, `J` or `Juliet`, `K` or `Kilo`, `L` or `Lima`, `M` or `Mike`, `N` or `November`, `O` or `Oscar`, `P` or `Papa`, `Q` or `Quebec`, `R` or `Romeo`, `S` or `Sierra`, `T` or `Tango`, `U` or `Uniform`, `V` or `Victor`, `W` or `Whisky`, `X` or `Xray`, `Y` or `Yankee`, `Z` or `Zulu`
- Number keys: `0` or `zero`, `1` or `one`, `2` or `two`, `3` or `three`, `4` or `four`, `5` or `five`, `6` or `six`, `7` or `seven`, `8` or `eight`, `9` or `nine`
- Symbol keys: `bang` or `exclamation`, `at`, `hash`, `dollar`, `percent`, `caret`, `and` or `ampersand`, `star` or `asterisk`, `leftparen` or `lparen`, `rightparen` or `rparen`, `minus` or `hyphen`, `underscore`, `plus`, `backtick`, `tilde`, `leftbracket` or `lbracket`, `rightbracket` or `rbracket`, `leftbrace` or `lbrace`, `rightbrace` or `rbrace`, `backslash`, `bar`, `colon`, `semicolon`, `apostrophe` or `singlequote` or `squote`, `quote` or `doublequote` or `dquote`, `comma`, `dot`, `slash`, `lessthan` or `leftangle` or `langle`, `greaterthan` or `rightangle` or `rangle`, `question`, `equal` or `equals`
- Whitespace and editing keys: `enter`, `tab`, `space`, `backspace`, `delete` or `del`
- Modifier keys: `shift`, `control` or `ctrl`, `alt`
- Special keys: `escape`, `insert`, `pause`, `win`, `apps` or `popup`
- Navigation keys: `up`, `down`, `left`, `right`, `pageup` or `pgup`, `pagedown` or `pgdown`, `home`, `end`
- Number pad keys: `npmul`, `npadd`, `npsep`, `npsub`, `npdec`, `npdiv`, `numpad0` or `np0`, `numpad1` or `np1`, `numpad2` or `np2`, `numpad3` or `np3`, `numpad4` or `np4`, `numpad5` or `np5`, `numpad6` or `np6`, `numpad7` or `np7`, `numpad8` or `np8`, `numpad9` or `np9`
- Function keys: `f1`, `f2`, `f3`, `f4`, `f5`, `f6`, `f7`, `f8`, `f9`, `f10`, `f11`, `f12`, `f13`, `f14`, `f15`, `f16`, `f17`, `f18`, `f19`, `f20`, `f21`, `f22`, `f23`, `f24`
- Multimedia keys: `volumeup` or `volup`, `volumedown` or `voldown`, `volumemute` or `volmute`, `tracknext`, `trackprev`, `playpause`, `browserback`, `browserforward`

Example key actions

The following code types the text "Hello world!" into the foreground application:

```
Key("H, e, l, l, o, space, w, o, r, l, d, exclamation").execute()
```

The following code is a bit more useful, as it saves the current file with the name "dragonfly.txt" (this works for many English-language applications):

```
action = Key("a-f, a/50") + Text("dragonfly.txt") + Key("enter")
action.execute()
```

The following code selects the next four lines by holding down the *shift* key, slowly moving down 4 lines, and then releasing the *shift* key:

```
Key("shift:down, down/25:4, shift:up").execute()
```

The following code locks the screen by pressing the *Windows* key together with the *l*:

```
Key("w-l").execute()
```

Key class reference

class Key (*spec=None*, *static=False*)

Keystroke emulation action.

Constructor arguments:

- *spec* (*str*) – keystroke specification
- *static* (boolean) – flag indicating whether the specification contains dynamic elements

The format of the keystroke specification *spec* is described in *Keystroke specification format*.

This class emulates keyboard activity by sending keystrokes to the foreground application. It does this using Dragonfly's keyboard interface implemented in the `keyboard` and `sendinput` modules. These use the `sendinput()` function of the Win32 API.

Text action

This section describes the *Text* action object. This type of action is used for typing text into the foreground application.

It differs from the `Key` action in that *Text* is used for typing literal text, while *dragonfly.actions.action_key.Key* emulates pressing keys on the keyboard. An example of this is that the arrow-keys are not part of a text and so cannot be typed using the *Text* action, but can be sent by the *dragonfly.actions.action_key.Key* action.

class Text (*spec=None*, *static=False*, *pause=0.02*, *autofmt=False*)

Action that sends keyboard events to type text.

Arguments:

- *spec* (*str*) – the text to type
- *static* (boolean) – if *True*, do not dynamically interpret *spec* when executing this action
- *pause* (*float*) – the time to pause between each keystroke, given in seconds
- *autofmt* (boolean) – if *True*, attempt to format the text with correct spacing and capitalization. This is done by first mimicking a word recognition and then analyzing its spacing and capitalization and applying the same formatting to the text.

Paste action

class Paste (*contents*, *format=None*, *paste=None*, *static=False*)

Paste-from-clipboard action.

Constructor arguments:

- *contents* (*str*) – contents to paste
- *format* (*int*, Win32 clipboard format) – clipboard format
- *paste* (instance derived from *ActionBase*) – paste action
- *static* (boolean) – flag indicating whether the specification contains dynamic elements

This action inserts the given *contents* into the Windows system clipboard, and then performs the *paste* action to paste it into the foreground application. By default, the *paste* action is the `Control-v` keystroke. The default clipboard format to use is the *Unicode* text format.

Mouse action

This section describes the *Mouse* action object. This type of action is used for controlling the mouse cursor and clicking mouse button.

Below you'll find some simple examples of *Mouse* usage, followed by a detailed description of the available mouse events.

Example mouse actions

The following code moves the mouse cursor to the center of the foreground window (`(0.5, 0.5)`) and then clicks the left mouse button once (`left`):

```
# Parentheses ("(...)") give foreground-window-relative locations.
# Fractional locations ("0.5", "0.9") denote a location relative to
#  the window or desktop, where "0.0, 0.0" is the top-left corner
#  and "1.0, 1.0" is the bottom-right corner.
action = Mouse("(0.5, 0.5), left")
action.execute()
```

The line below moves the mouse cursor to 100 pixels left of the desktop's right edge and 250 pixels down from its top edge (`[-100, 250]`), and then double clicks the right mouse button (`right:2`):

```
# Square brackets ("[...]") give desktop-relative locations.
# Integer locations ("1", "100", etc.) denote numbers of pixels.
# Negative numbers ("-100") are counted from the right-edge or the
#  bottom-edge of the desktop or window.
Mouse("[-100, 250], right:2").execute()
```

The following command drags the mouse from the top right corner of the foreground window (`(0.9, 10), left:down`) to the bottom left corner (`(25, -0.1), left:up`):

```
Mouse("(0.9, 10), left:down, (25, -0.1), left:up").execute()
```

The code below moves the mouse cursor 25 pixels right and 25 pixels up (`<25, -25>`):

```
# Angle brackets ("<...>") move the cursor from its current position
#  by the given number of pixels.
Mouse("<25, -25>").execute()
```

Mouse specification format

The *spec* argument passed to the *Mouse* constructor specifies which mouse events will be emulated. It is a string consisting of one or more comma-separated elements. Each of these elements has one of the following possible formats:

Mouse movement actions:

- location is absolute on the entire desktop: `[` *number* `,` *number* `]`
- location is relative to the foreground window: `(` *number* `,` *number* `)`
- move the cursor relative to its current position: `<` *pixels* `,` *pixels* `>`

In the above specifications, the *number* and *pixels* have the following meanings:

- *number* – can specify a number of pixels or a fraction of the reference window or desktop. For example:
  - `(10, 10)` – 10 pixels to the right and down from the foreground window's left-top corner
  - `(0.5, 0.5)` – center of the foreground window
- *pixels* – specifies the number of pixels

Mouse button-press action:

keyname [`:` *repeat*] [`/` *pause*]

- *keyname* – Specifies which mouse button to click:
  - `left` – left mouse button key
  - `middle` – middle mouse button key
  - `right` – right mouse button key
- *repeat* – Specifies how many times the button should be clicked:
  - `0` – don't click the button, this is a no-op
  - `1` – normal button click
  - `2` – double-click
  - `3` – triple-click
- *pause* – Specifies how long to pause *after* clicking the button. The value should be an integer giving in hundredths of a second. For example, `/100` would mean one second, and `/50` half a second.

Mouse button-hold or button-release action:

keyname `:` *hold-or-release* [`/` *pause*]

- *keyname* – Specifies which mouse button to click; same as above.
- *hold-or-release* – Specified whether the button will be held down or released:
  - `down` – hold the button down
  - `up` – release the button
- *pause* – Specifies how long to pause *after* clicking the button; same as above.

Mouse class reference

class **Mouse**(*spec=None*, *static=False*)

Action that sends mouse events.

Arguments:

- *spec* (*str*) – the mouse actions to execute
- *static* (boolean) – if *True*, do not dynamically interpret *spec* when executing this action

Function action

The *Function* action wraps a callable, optionally with some default keyword argument values. On execution, the execution data (commonly containing the recognition extras) are combined with the default argument values (if present) to form the arguments with which the callable will be called.

Simple usage:

```
>>> def func(count):
...     print "count:", count
...
>>> action = Function(func)
>>> action.execute({"count": 2})
count: 2
True
```

```
>>> # Additional keyword arguments are ignored:
>>> action.execute({"count": 2, "flavor": "vanilla"})
count: 2
True
```

Usage with default arguments:

```
>>> def func(count, flavor):
...     print "count:", count
...     print "flavor:", flavor
...
>>> # The Function object can be given default argument values:
>>> action = Function(func, flavor="spearmint")
>>> action.execute({"count": 2})
count: 2
flavor: spearmint
True
>>> # Arguments given at the execution-time to override default values:
>>> action.execute({"count": 2, "flavor": "vanilla"})
count: 2
flavor: vanilla
True
```

Class reference

class `Function` (*function*, ***defaults*)

Call a function with extra keyword arguments.

Constructor arguments:

- *function* (callable) – the function to call when this action is executed
- defaults – default keyword-values for the arguments with which the function will be called

Mimic action

class `Mimic` (**words*, ***kwargs*)

Mimic recognition action.

The constructor arguments are the words which will be mimicked. These should be passed as a variable argument list. For example:

```
action = Mimic("hello", "world", r"!\exclamation-mark")
action.execute()
```

If an error occurs during mimicking the given recognition, then an *ActionError* is raised. A common error is that the engine does not know the given words and can therefore not recognize them. For example, the following attempts to mimic recognition of *one single word* including a space and an exclamation-mark; this will almost certainly fail:

```
Mimic("hello world!").execute()    # Will raise ActionError.
```

The constructor accepts the optional *extra* keyword argument, and uses this to retrieve dynamic data from the extras associated with the recognition. For example, this can be used as follows to implement dynamic mimicking:

```
class ExampleRule(MappingRule):
    mapping  = {
                "mimic recognition <text> [<n> times]":
                    Mimic(extra="text") * Repeat(extra="n"),
               }
    extras   = [
                IntegerRef("n", 1, 10),
                Dictation("text"),
               ]
    defaults = {
                "n": 1,
               }
```

The example above will allow the user to speak **"mimic recognition hello world! 3 times"**, which would result in the exact same output as if the user had spoken **"hello world!"** three times in a row.

Playback action

The *Playback* action mimics a sequence of recognitions. This is for example useful for repeating a series of prerecorded or predefined voice-commands.

This class could for example be used to reload with one single action:

```
action = Playback([
                   (["focus", "Natlink"], 1.0),
                   (["File"], 0.5),
                   (["Reload"], 0.0),
                  ])
action.execute()
```

Class reference

class **Playback**(*series*, *speed=1*)

Playback a series of recognitions.

Constructor arguments:

- *series* (sequence of 2-tuples) – the recognitions to playback. Each element must be a 2-tuple of the form *(["words", "two", "mimic"], interval)*, where *interval* is a float giving the number of seconds to pause after the given words are mimicked.
- *speed* (*float*) – the factor by which to speed up playback. The intervals after each mimic are divided by this number.

speed

Factor to speed up playback.

WaitWindow action

class **WaitWindow**(*title=None*, *executable=None*, *timeout=15*)

Wait for a specific window context action.

Constructor arguments:

- *title* (*str*) – part of the window title: not case sensitive
- *executable* (*str*) – part of the file name of the executable; not case sensitive
- *timeout* (*int* or *float*) – the maximum number of seconds to wait for the correct context, after which an `ActionError` will be raised.

When this action is executed, it waits until the correct window context is present. This window context is specified by the desired window title of the foreground window and/or the executable name of the foreground application. These are specified using the constructor arguments listed above. The substring search used is *not* case sensitive.

If the correct window context is not found within *timeout* seconds, then this action will raise an `ActionError` to indicate the timeout.

FocusWindow action

class `FocusWindow` (*executable=None*, *title=None*)

Bring a window to the foreground action.

Constructor arguments:

- *executable* (*str*) – part of the filename of the application's executable to which the target window belongs; not case sensitive.
- *title* (*str*) – part of the title of the target window; not case sensitive.

This action searches all visible windows for a window which matches the given parameters.

BringApp and StartApp actions

The *`StartApp`* and *`BringApp`* action classes are used to start an application and bring it to the foreground. *`StartApp`* starts an application by running an executable file, while *`BringApp`* first checks whether the application is already running and if so brings it to the foreground, otherwise starts it by running the executable file.

Example usage

The following example brings Notepad to the foreground if it is already open, otherwise it starts Notepad:

```
BringApp(r"C:\Windows\system32\notepad.exe").execute()
```

Note that the path to *notepad.exe* given above might not be correct for your computer, since it depends on the operating system and its configuration.

In some cases an application might be accessible simply through the file name of its executable, without specifying the directory. This depends on the operating system's path configuration. For example, on the author's computer the following command successfully starts Notepad:

```
BringApp("notepad").execute()
```

Class reference

class `BringApp` (**args*, ***kwargs*)

Bring an application to the foreground, starting it if it is not yet running.

When this action is executed, it looks for an existing window of the application specified in the constructor arguments. If an existing window is found, that window is brought to the foreground. On the other hand, if no window is found the application is started.

Note that the constructor arguments are identical to those used by the *`StartApp`* action class.

Constructor arguments:

- *args* (variable argument list of *str*'s) – these strings are passed to `subprocess.Popen()` to start the application as a child process
- *cwd* (*str*, default *None*) – if not *None*, then start the application in this directory

class StartApp (**args*, ***kwargs*)
Start an application.

When this action is executed, it runs a file (executable), optionally with commandline arguments.
Constructor arguments:
- *args* (variable argument list of *str*'s) – these strings are passed to subprocess.Popen() to start the application as a child process
- *cwd* (*str*, default *None*) – if not *None*, then start the application in this directory

Pause action

class Pause (*spec=None*, *static=False*)
Pause for the given amount of time.

The *spec* constructor argument should be a *string* giving the time to wait. It should be given in hundredths of a second. For example, the following code will pause for 20/100s = 0.2 seconds:

```
Pause("20").execute()
```

The reason the *spec* must be given as a *string* is because it can then be used in dynamic value evaluation. For example, the following code determines the time to pause at execution time:

```
action = Pause("%(time)d")
data = {"time": 37}
action.execute(data)
```

Miscellaneous topics

Contents:

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:

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.

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)
```

Clipboard class

class **Clipboard**(*contents=None*, *text=None*, *from_system=False*)

copy_from_system(*formats=None*, *clear=False*)

Copy the Windows system clipboard contents into this instance.

Arguments:

- *formats* (iterable, default: None) – if not None, only the given content formats will be retrieved. If None, all available formats will be retrieved.
- *clear* (boolean, default: False) – if true, the Windows system clipboard will be cleared after its contents have been retrieved.

copy_to_system(*clear=True*)

Copy the contents of this instance into the Windows system clipboard.

Arguments:

- *clear* (boolean, default: True) – if true, the Windows system clipboard will be cleared before this instance's contents are transferred.

get_format (*format*)
Retrieved this instance's content for the given *format*.
Arguments:
- *format* (int) – the clipboard format to retrieve.

If the given *format* is not available, a *ValueError* is raised.

get_text()
Retrieve this instance's text content. If no text content is available, this method returns *None*.

has_format (*format*)
Determine whether this instance has content for the given *format*.
Arguments:
- *format* (int) – the clipboard format to look for.

has_text()
Determine whether this instance has text content.

Configuration toolkit

Configuration toolkit

Dragonfly's configuration toolkit makes it very easy to store program data in a separate file from the main program logic. It uses a three-phase *setup* – *load* – *use* system:

- *setup* – a Config object is created and its structure and default contents are defined.
- *load* – a separate file containing the user's configuration settings is looked for and, if found, its values are loaded into the Config object.
- *use* – the program directly accesses the configuration through easy Config object attributes.

This configuration toolkit uses the following three classes:

- *Config* – a collection of configuration settings, grouped within one or more sections
- *Section* – a group of items and/or subsections
- *Item* – a single configuration setting

Usage example

The main program using Dragonfly's configuration toolkit would normally look something like this:

```
from dragonfly.all import Config, Section, Item

# *Setup* phase.
# This defines a configuration object with the name "Example
#  configuration".  It contains one section with the title
#  "Test section", which has two configuration items.  Both
#  these items have a default value and a docstring.
config                 = Config("Example configuration")
config.test            = Section("Test section")
config.test.fruit      = Item("apple", doc="Must eat fruit.")
config.test.color      = Item("blue", doc="The color of life.")

# *Load* phase.
# This searches for a file with the same name as the main program,
#  but with the extension ".py" replaced by ".txt".  It is also
```

```
#  possible to explicitly specify the configuration file's path.
#  See Config.load() for more details.
config.load()

# *Use* phase.
# The configuration values can now be accessed through the
#  configuration object as follows.
print "The color of life is", config.test.color
print "You must eat an %s every day" % config.test.fruit
```

The configuration defined above is basically complete. Every configuration item has a default value and can be accessed by the program. But if the user would like to modify some or all of these settings, he can do so in an external configuration file without modifying the main program code.

This external configuration file is interpreted as Python code. This gives its author powerful tools for determining the desired configuration settings. However, it will usually consist merely of variable assignments. The configuration file for the program above might look something like this:

```
# Test section
test.fruit = "banana"    # Bananas have more potassium.
test.color = "white"     # I like light colors.
```

Implementation details

This configuration toolkit makes use of Python's special methods for setting and retrieving object attributes. This makes it much easier to use, as there is no need to use functions such as *value = get_config_value("item_name")*; instead the configuration values are immediately accessible as Python objects. It also allows for more extensive error checking; it is for example trivial to implement custom *Item* classes which only allow specific values or value types, such as integers, boolean values, etc.

Configuration class reference

class **Config**(*name*)

Configuration class for storing program settings.

Constructor argument:

- *name* (*str*) – the name of this configuration object.

This class can contain zero or more *Section* instances, each of which can contain zero or more *Item* instances. It is these items which store the actual configuration settings. The sections merely divide the items up into groups, so that different configuration topics can be split for easy readability.

generate_config_file(*path=None*)

Create a configuration file containing this configuration object's current settings.

- *path* (*str*, default: *None*) – path to the configuration file to load. If *None*, then a path is generated from the calling module's file name by replacing its extension with ".txt".

load(*path=None*)

Load the configuration file at the given *path*, or look for a configuration file associated with the calling module.

- *path* (*str*, default: *None*) – path to the configuration file to load. If *None*, then a path is generated from the calling module's file name by replacing its extension with ".txt".

If the *path* is a file, it is loaded. On the other hand, if it does not exist or is not a file, nothing is loaded and this configuration's defaults remain in place.

class `Section`(*doc*)

Section of a configuration for grouping items.

Constructor argument:

- *doc* (*str*) – the name of this configuration section.

A section can contain zero or more subsections and zero or more configuration items.

class `Item`(*default*, *doc=None*, *namespace=None*)

Configuration item for storing configuration settings.

Constructor arguments:

- *default* – the default value for this item
- *doc* (*str*, default: *None*) – an optional description of this item
- *namespace* (*dict*, default: *None*) – an optional namespace dictionary which will be made available to the Python code in the external configuration file during loading

A configuration item is the object that stores the actual configuration settings. Each item has a default value, a current value, an optional description, and an optional namespace.

This class performs the checking of configuration values assigned to it during loading of the configuration file. The default behavior of this class is to only accept values of the same Python type as the item's default value. So, if the default value is a string, then the value assigned in the configuration file must also be a string. Otherwise an exception will be raised and loading will fail.

Developers who want other kinds of value checking should override the `Item.validate()` method of this class.

`validate`(*value*)

Determine whether the given *value* is valid.

This method performs validity checking of the configuration value assigned to this item during loading of the external configuration file. If the default behavior is to raise a *TypeError* if the type of the assigned value is not the same as the type of the default value.

Project

Contents:

Code style

Commit message format

Commit messages should be written according to the guidelines described here. These guidelines are meant to ensure high-quality and easily readable content of commit messages. They also result in a pleasant viewing experience using standard Git tools.

Purpose of a commit message

Every commit message should provide the reader with the following information:

- Why this change is necessary
- How this change addresses the issue
- What other effects this change has

Structure of a commit message

First line

The first line of a commit message represents the title or summary of the change. Standard Git tools often display it differently than the rest of the message, for example in bold font, or show only this line, for example when summarizing multiple commits.

A commit message's first line should be formatted as follows:

- The first line should be no longer than 50 characters
- The first line should start with a capital letter
- The first line should not end with a full-stop
- The first line should be followed by a blank line, if and only if the commit message contains more text below
- The first line should use the imperative present tense, e.g. "Add cool new feature" or "Fix some little bug"

Issue references

Issues can be referenced anywhere within a commit message via their numbered tag, e.g. "#7".

Commits that change the status of an issue, for example fixing a bug or implementing a feature, should make the relationship and change explicit on the last line of the commit message using the following format: `Resolve #X`. GitHub will automatically update the issue accordingly.

Please see GitHub's help on commit message keywords for more information.

Example

```
Commit title summarizing the change (50 characters or less)

Subsequent text providing further information about the change, if
necessary. Lines are wrapped at 72 characters.

- May contain bullet points, prefixed by "- " at the beginning of the
  first line for a bullet point and "  " for subsequent lines
- Non-breakable text, such as long URLs, may extend past 72 characters;
  doesn't look nice, but at least they still work
```

Background and inspiration

The commit message format described here is based on common views, such as those expressed here:

- http://git-scm.com/book/ch5-2.html
- http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
- http://who-t.blogspot.nl/2009/12/on-commit-messages.html
- http://dieter.plaetinck.be/why-rewriting-git-history-and-why-commits-imperative-present-tense.html

Release versioning

The versions of Dragonfly releases are strongly inspired by the semantic versioning concept, as promoted by Tom Preston-Werner and documented at *http://semver.org/*.

Each version string has the format "major.minor.patch", where each part has the following meaning:

- *Major* –
- *Minor* –
- *Patch* –

Version incrementation

Release process

Preparation

1. Version number and release branch
   (a) Determine the appropriate version number for this release, according to *Release versioning*.
   (b) Create the new release branch, named `release-X.Y.Z`
   (c) Update the `version.txt` file to contain the new version number.
2. Tickets
   (a) Update ticket status for this release, where relevant: https://github.com/t4ngo/dragonfly/issues
3. Release files
   (a) Verify that `CHANGES.txt` includes the change log for this release.
   (b) Verify that `AUTHORS.txt` is up to date with recent contributors.
   (c) Verify that `setup.py` specifies all required dependencies, including their versions, e.g. with the `install_requires` and `test_requires` parameters.
   (d) Verify that `MANIFEST.in` includes all necessary data files.
4. Draft announcement
   (a) Write a draft announcement text to send to the mailing list after the release process has been completed.

Build and test

1. Test building of documentation
2. Build distributions
3. Test installation of distributions
4. Test on PyPI test server
   (a) Upload distributions to PyPI test server
   (b) Test installation from PyPI test server
   (c) Verify package is displayed correctly on PyPI test server
5. Tag release

(a) Tag git revision

(b) Push to GitHub

Release

1. Upload to GitHub
   (a) Upload distributions to GitHub: https://github.com/t4ngo/dragonfly/releases
2. Trigger building of documentation on Read the Docs
   (a) Check whether documentation was built automatically, and if not trigger it: https://readthedocs.org/builds/dragonfly/
3. Upload to PyPI server
   (a) Upload distributions to PyPI server
   (b) Test installation from PyPI server
   (c) Verify package is displayed correctly on PyPI server

Post-release

1. Announce release
   (a) Website
   (b) Mailing list

Test suite

The Dragonfly library contains tests to verify its functioning and assure its quality. These tests come in two distinct types:

- Tests based on unittest.
- Tests based on doctest; these also serve as documentation by providing usage examples.

See the links below for tests of both types.

Tests in doctests format:

Doctests for the fundamental element classes

Sequence element class

Test fixture initialization:

```
>>> from dragonfly import *
>>> from dragonfly.test import ElementTester
```

Sequence

Basic usage:

```
>>> seq = Sequence([Literal("hello"), Literal("world")])
>>> test_seq = ElementTester(seq)
>>> test_seq.recognize("hello world")
[u'hello', u'world']
>>> test_seq.recognize("hello universe")
RecognitionFailure
```

Constructor arguments:

```
>>> c1, c2 = Literal("hello"), Literal("world")
>>> len(Sequence(children=[c1, c2]).children)
2
>>> Sequence(children=[c1, c2], name="sequence_test").name
'sequence_test'
>>> Sequence([c1, c2], "sequence_test").name
'sequence_test'
>>> Sequence("invalid_children_type")
Traceback (most recent call last):
  ...
TypeError: children object must contain only <class 'dragonfly.grammar.elements_basic.
↪ElementBase'> types.  (Received ('i', 'n', 'v', 'a', 'l', 'i', 'd', '_', 'c', 'h',
↪'i', 'l', 'd', 'r', 'e', 'n', '_', 't', 'y', 'p', 'e'))
```

Doctests for the Compound element class

Basic usage

Test fixture initialization:

```
>>> from dragonfly import *
>>> from dragonfly.test import ElementTester
```

"Hello world"

The spec of the compound element below is parsed into a single literal "hello world". The semantic value of the compound element will therefore be the same as for that literal element, namely "hello world".

```
>>> element = Compound("hello world")
>>> tester = ElementTester(element)

>>> tester.recognize("hello world")
u'hello world'
>>> tester.recognize("hello universe")
RecognitionFailure
```

"Hello [there] (world | universe)"

The spec of the compound element below is parsed into a sequence with three elements: the word "hello", an optional "there", and an alternative of "world" or "universe". The semantic value of the compound element will therefore have

three elements, even when "there" is not spoken.

```
>>> element = Compound("hello [there] (world | universe)")
>>> tester = ElementTester(element)

>>> tester.recognize("hello world")
[u'hello', None, u'world']
>>> tester.recognize("hello there world")
[u'hello', u'there', u'world']
>>> tester.recognize("hello universe")
[u'hello', None, u'universe']
>>> tester.recognize("hello galaxy")
RecognitionFailure
```

Doctests for the List class

List and ListRef element classes

Basic usage

Setup test tooling:

```
>>> from dragonfly import *
>>> from dragonfly.test import ElementTester
>>> list_fruit = List("list_fruit")
>>> element = Sequence([Literal("item"), ListRef("list_fruit_ref", list_fruit)])
>>> tester_fruit = ElementTester(element)
>>> # Explicitly load tester grammar because lists can only be updated
>>> # for loaded grammars.
>>> tester_fruit.load()
```

Empty lists cannot be recognized:

```
>>> tester_fruit.recognize("item")
RecognitionFailure
>>> tester_fruit.recognize("item apple")
RecognitionFailure
```

A list update is automatically available for recognition without reloading the grammar:

```
>>> tester_fruit.recognize("item apple")
RecognitionFailure

>>> list_fruit.append("apple")
>>> list_fruit
['apple']
>>> tester_fruit.recognize("item apple")
[u'item', u'apple']
>>> tester_fruit.recognize("item banana")
RecognitionFailure

>>> list_fruit.append("banana")
>>> list_fruit
['apple', 'banana']
>>> tester_fruit.recognize("item apple")
[u'item', u'apple']
```

```
>>> tester_fruit.recognize("item banana")
[u'item', u'banana']
>>> tester_fruit.recognize("item apple banana")
RecognitionFailure

>>> list_fruit.remove("apple")
>>> list_fruit
['banana']
>>> tester_fruit.recognize("item apple")
RecognitionFailure
>>> tester_fruit.recognize("item banana")
[u'item', u'banana']
```

Lists can contain the same value multiple times, although that does not affect recognition:

```
>>> list_fruit.append("banana")
>>> list_fruit
['banana', 'banana']
>>> tester_fruit.recognize("item banana")
[u'item', u'banana']
>>> tester_fruit.recognize("item banana banana")
RecognitionFailure
```

Tear down test tooling:

```
>>> # Explicitly unload tester grammar.
>>> tester_fruit.unload()
```

Multiple lists

Setup test tooling:

```
>>> list_meat = List("list_meat")
>>> list_veg = List("list_veg")
>>> element = Sequence([Literal("food"),
...                     ListRef("list_meat_ref", list_meat),
...                     ListRef("list_veg_ref", list_veg)])
>>> tester_meat_veg = ElementTester(element)
>>> # Explicitly load tester grammar because lists can only be updated
>>> # for loaded grammars.
>>> tester_meat_veg.load()
```

Multiple lists can be combined within a single rule:

```
>>> list_meat.append("steak")
>>> tester_meat_veg.recognize("food steak")
RecognitionFailure
>>> list_veg.append("carrot")
>>> tester_meat_veg.recognize("food steak carrot")
[u'food', u'steak', u'carrot']
>>> list_meat.append("hamburger")
>>> tester_meat_veg.recognize("food hamburger carrot")
[u'food', u'hamburger', u'carrot']
```

Tear down test tooling:

```
>>> # Explicitly unload tester grammar.
>>> tester_meat_veg.unload()
```

A single list can be present multiple times within a rule:

```
>>> element = Sequence([Literal("carnivore"),
...                     ListRef("list_meat_ref1", list_meat),
...                     ListRef("list_meat_ref2", list_meat)])
>>> tester_carnivore = ElementTester(element)
>>> # Explicitly load tester grammar because lists can only be updated
>>> # for loaded grammars.
>>> tester_carnivore.load()

>>> tester_carnivore.recognize("carnivore steak")
RecognitionFailure
>>> tester_carnivore.recognize("carnivore hamburger steak")
[u'carnivore', u'hamburger', u'steak']
>>> tester_carnivore.recognize("carnivore steak hamburger")
[u'carnivore', u'steak', u'hamburger']
>>> tester_carnivore.recognize("carnivore steak steak")
[u'carnivore', u'steak', u'steak']

>>> list_meat.remove("steak")
>>> tester_carnivore.recognize("carnivore steak hamburger")
RecognitionFailure
>>> tester_carnivore.recognize("carnivore hamburger hamburger")
[u'carnivore', u'hamburger', u'hamburger']
```

Tear down test tooling:

```
>>> # Explicitly unload tester grammar.
>>> tester_carnivore.unload()
```

Unique list names

The names of lists must be unique within a grammar:

```
>>> list_fruit1 = List("list_fruit")
>>> list_fruit2 = List("list_fruit")
>>> element = Sequence([Literal("fruit"),
...                     ListRef("list_fruit1_ref", list_fruit1),
...                     ListRef("list_fruit2_ref", list_fruit2)])
>>> tester_fruit = ElementTester(element)
>>> # Explicitly load tester grammar because lists can only be updated
>>> # for loaded grammars.
>>> tester_fruit.load()
Traceback (most recent call last):
    ...
GrammarError: Two lists with the same name 'list_fruit' not allowed.
```

ListRef construction

ListRef objects must be created referencing the correct type of list object:

```
>>> print ListRef("list_fruit_ref", [])  # Fails.
Traceback (most recent call last):
    ...
TypeError: List argument to ListRef constructor must be a Dragonfly list.
>>> print ListRef("list_fruit_ref", List("list_fruit"))  # Succeeds.
ListRef('list_fruit')
```

RecognitionObserver base class

Note: `RecognitionObserver` instances can be used for both the DNS and the WSR backend engines. However, WSR does not offer access to the words recognized by a different context, and therefore the `RecognitionObservers.on_recognition()` will always be called with `words = False`.

Test fixture initialization:

```
>>> from dragonfly import *
>>> from dragonfly.test import ElementTester
```

Trivial demonstration of RecognitionObserver class

The following class is derived from RecognitionObserver and prints when its callback methods are called:

```
>>> class RecognitionObserverDemo(RecognitionObserver):
...     def on_begin(self):
...         print "on_begin()"
...     def on_recognition(self, words):
...         print "on_recognition(): %s" % (words,)
...     def on_failure(self):
...         print "on_failure()"
...
>>> recobs_demo = RecognitionObserverDemo()
>>> recobs_demo.register()
>>> test_lit = ElementTester(Literal("hello world"))
>>> test_lit.recognize("hello world")
on_begin()
on_recognition(): (u'hello', u'world')
u'hello world'
>>> test_lit.recognize("hello universe")
on_begin()
on_failure()
RecognitionFailure
>>> recobs_demo.unregister()
```

Tests for RecognitionObserver class

A class derived from RecognitionObserver which will be used here for testing it:

```
>>> class RecognitionObserverTester(RecognitionObserver):
...     def __init__(self):
...         RecognitionObserver.__init__(self)
...         self.waiting = False
```

```
...             self.words = None
...         def on_begin(self):
...             self.waiting = True
...         def on_recognition(self, words):
...             self.waiting = False
...             self.words = words
...         def on_failure(self):
...             self.waiting = False
...             self.words = False
...
>>> test_recobs = RecognitionObserverTester()
>>> test_recobs.register()
>>> test_recobs.waiting, test_recobs.words
(False, None)
```

Simple literal element recognitions:

```
>>> test_lit = ElementTester(Literal("hello world"))
>>> test_lit.recognize("hello world")
u'hello world'
>>> test_recobs.waiting, test_recobs.words
(False, (u'hello', u'world'))
>>> test_lit.recognize("hello universe")
RecognitionFailure
>>> test_recobs.waiting, test_recobs.words
(False, False)
```

Integer element recognitions:

```
>>> test_int = ElementTester(Integer(min=1, max=100))
>>> test_int.recognize("seven")
7
>>> test_recobs.waiting, test_recobs.words
(False, (u'seven',))
>>> test_int.recognize("forty seven")
47
>>> test_recobs.waiting, test_recobs.words
(False, (u'forty', u'seven'))
>>> test_int.recognize("one hundred")
RecognitionFailure
>>> test_recobs.waiting, test_recobs.words
(False, False)
>>> test_lit.recognize("hello world")
u'hello world'
```

RecognitionHistory class

Basic usage of the RecognitionHistory class:

```
>>> history = RecognitionHistory()
>>> test_lit.recognize("hello world")
u'hello world'
>>> # Not yet registered, so didn't receive previous recognition.
>>> history
[]
>>> history.register()
```

```
>>> test_lit.recognize("hello world")
u'hello world'
>>> # Now registered, so should have received previous recognition.
>>> history
[(u'hello', u'world')]
>>> test_lit.recognize("hello universe")
RecognitionFailure
>>> # Failed recognitions are ignored, so history is unchanged.
>>> history
[(u'hello', u'world')]
>>> test_int.recognize("eighty six")
86
>>> history
[(u'hello', u'world'), (u'eighty', u'six')]
```

The RecognitionHistory class allows its maximum length to be set:

```
>>> history = RecognitionHistory(3)
>>> history.register()
>>> history
[]
>>> for i, word in enumerate(["one", "two", "three", "four", "five"]):
...     assert test_int.recognize(word) == i + 1
>>> history
[(u'three',), (u'four',), (u'five',)]
```

The length must be a positive integer. A length of 0 is not allowed:

```
>>> history = RecognitionHistory(0)
Traceback (most recent call last):
  ...
ValueError: length must be a positive int or None, received 0.
```

Minimum length is 1:

```
>>> history = RecognitionHistory(1)
>>> history.register()
>>> history
[]
>>> for i, word in enumerate(["one", "two", "three", "four", "five"]):
...     assert test_int.recognize(word) == i + 1
>>> history
[(u'five',)]
```

Action doctests

ActionBase test suite

The `ActionBase` class implements various basing behaviors of action objects.

Test tool

The following PrintAction is used in this test suite:

```
>>> from dragonfly import ActionBase, Repeat
>>> class PrintAction(ActionBase):
...     def __init__(self, name):
...         ActionBase.__init__(self)
...         self._name = name
...     def execute(self, data=None):
...         if data:  print "executing %r %r" % (self._name, data)
...         else:     print "executing %r" % (self._name,)
...
>>> a = PrintAction("a")
>>> a.execute()
executing 'a'
>>> a.execute({"foo": 2})
executing 'a' {'foo': 2}
>>>
```

Concatenating actions

Concatenation of multiple actions:

```
>>> b = PrintAction("b")
>>> (a + b).execute()            # Simple concatenation.
executing 'a'
executing 'b'
>>> (a + b).execute({"foo": 2}) # Simple concatenation.
executing 'a' {'foo': 2}
executing 'b' {'foo': 2}

>>> c = a
>>> c += b                       # In place concatenation.
>>> c.execute({"foo": 2})
executing 'a' {'foo': 2}
executing 'b' {'foo': 2}
>>> c += a                       # In place concatenation.
>>> c.execute()
executing 'a'
executing 'b'
executing 'a'

>>> (c + c).execute()            # Same object concatenation.
executing 'a'
executing 'b'
executing 'a'
executing 'a'
executing 'b'
executing 'a'
```

Repeating actions

Actions can be repeated by multiplying them with a factor:

```
>>> (a * 3).execute()
executing 'a'
executing 'a'
```

```
executing 'a'
>>> ((a + b) * 2).execute({"foo": 2})
executing 'a' {'foo': 2}
executing 'b' {'foo': 2}
executing 'a' {'foo': 2}
executing 'b' {'foo': 2}

>>> factor = Repeat(3)              # Integer-factor repetition.
>>> (a * factor).execute()
executing 'a'
executing 'a'
executing 'a'
>>> factor = Repeat(extra="foo")    # Named-factor repetition.
>>> ((a + b) * factor).execute({"foo": 2})
executing 'a' {'foo': 2}
executing 'b' {'foo': 2}
executing 'a' {'foo': 2}
executing 'b' {'foo': 2}
>>> ((a + b) * factor).execute({"bar": 2})
Traceback (most recent call last):
  ...
ActionError: No extra repeat factor found for name 'foo'

>>> c = a
>>> c.execute({"foo": 2})
executing 'a' {'foo': 2}
>>> c *= Repeat(extra="foo")
>>> c.execute({"foo": 2})
executing 'a' {'foo': 2}
executing 'a' {'foo': 2}
>>> c += b
>>> c *= 2
>>> c.execute({"foo": 1})
executing 'a' {'foo': 1}
executing 'b' {'foo': 1}
executing 'a' {'foo': 1}
executing 'b' {'foo': 1}
>>> c *= 2
>>> c.execute({"foo": 0})
executing 'b' {'foo': 0}
executing 'b' {'foo': 0}
executing 'b' {'foo': 0}
executing 'b' {'foo': 0}
>>> c *= 0
>>> c.execute({"foo": 1})
```

Binding data to actions

Binding of data to actions:

```
>>> a_bound = a.bind({"foo": 2})
>>> a_bound.execute()
executing 'a' {'foo': 2}

>>> b_bound = b.bind({"bar": 3})
>>> b_bound.execute()
```

```
executing 'b' {'bar': 3}
```

Earliest bound data is used during execution:

```
>>> ab_bound = a_bound + b_bound
>>> ab_bound.execute({"bar": "later"})
executing 'a' {'foo': 2, 'bar': 'later'}
executing 'b' {'bar': 3}

>>> ab_bound = (a_bound + b_bound).bind({"bar": "later"})
>>> ab_bound.execute()
executing 'a' {'foo': 2, 'bar': 'later'}
executing 'b' {'bar': 3}
```

Tests based on the unittest framework reside in the `dragonfly.test` package.

Running the test suite

Using DNS

Follow the following steps to run the test suite for the DNS backend Aland

1. Start DNS. (And ensure that NatLink is also automatically started.)
2. Extract the Dragonfly source code in a directory `<dir>`.
3. Run the tests with the following commands:
   - `cd <dir>`
   - `C:\Python26\python.exe <dir>\setup.py test`

Using WSR

Follow the following steps to run the test suite for the DNS backend Aland

1. Start WSR.
2. Wake WSR up, so that it is *not* in sleeping state, and then turn the microphone *off*. (It is important to wake the microphone up first, because otherwise it'll be off and sleeping at the same time. This causes all recognitions to fail. Weird, huh?)
3. Extract the Dragonfly source code in a directory `<dir>`.
4. Run the tests with the following commands:
   - `cd <dir>`
   - `C:\Python26\python.exe <dir>\setup.py test --test-suite=dragonfly.test.suites.sapi5_suite`

Direct links within this documentation to help you get started:

- *Features and target audience*
- *Installation*

CHAPTER 2

Usage example

A very simple example of Dragonfly usage is to create a static voice command with a callback that will be called when the command is spoken. This is done as follows:

```
from dragonfly.all import Grammar, CompoundRule

# Voice command rule combining spoken form and recognition processing.
class ExampleRule(CompoundRule):
    spec = "do something computer"                  # Spoken form of command.
    def _process_recognition(self, node, extras):   # Callback when command is spoken.
        print "Voice command spoken."

# Create a grammar which contains and loads the command rule.
grammar = Grammar("example grammar")                # Create a grammar to contain the␣
↪command rule.
grammar.add_rule(ExampleRule())                     # Add the command rule to the␣
↪grammar.
grammar.load()                                      # Load the grammar.
```

The example above is very basic and doesn't show any of Dragonfly's exciting features, such as dynamic speech elements. To learn more about these, please take a look at the project's documentation here.

CHAPTER 3

Rationale behind Dragonfly

Dragonfly offers a powerful and unified interface to developers who want to use speech recognition in their software. It is used for both speech-enabling applications and for automating computer activities.

In the field of scripting and automation, there are other alternatives available that add speech-commands to increase efficiency. Dragonfly differs from them in that it is a powerful development platform. The open source alternatives currently available for use with DNS are compared to Dragonfly as follows:

- Vocola uses its own easy-to-use scripting language, whereas Dragonfly uses Python and gives the macro-writer all the power available.
- Unimacro offers a set of macros for common activities, whereas Dragonfly is a platform on which macro-writers can easily build new commands.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W