Dragonfly Documentation

Release 0.35.0

Christo Butcher

2022-10-19

Contents

Dragonfly is a speech recognition framework for Python that makes it convenient to create custom commands to use with speech recognition software. It 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.

Dragonfly can be used for general programming by voice. It is flexible enough to allow programming in any language, not just Python. It can also be used for speech-enabling applications, automating computer activities and dictating prose.

Dragonfly contains its own powerful framework for defining and executing actions. It includes actions for text input and key-stroke simulation. This framework is cross-platform, working on Windows, macOS and Linux (X11 only). See the *actions sub-package documentation* for more information, including code examples.

This project is a fork of the original t4ngo/dragonfly project.

Dragonfly currently supports the following speech recognition engines:

- *Dragon*, a product of *Nuance*. All versions up to 15 (the latest) are supported. *Home*, *Professional Individual* and previous similar editions of *Dragon* are supported. Other editions may work too
- *Windows Speech Recognition* (WSR), included with Microsoft Windows Vista, Windows 7+, and freely available for Windows XP
- *Kaldi*, open source (AGPL) and multi-platform.
- *CMU Pocket Sphinx*, open source and multi- platform.

Dragonfly's documentation is available online at Read the Docs. Dragonfly's FAQ is available in the documentation *here*. There are also a number of Dragonfly-related questions on Stackoverflow, although many of them are related to issues resolved in the latest version of Dragonfly.

Dragonfly's mailing list/discussion group is available at Google Groups.

CHAPTER 1

CompoundRule 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 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.
```

To use this example, save it in a command module in your module loader directory or Natlink user directory, load it and then say *do something computer*. If the speech recognition engine recognized the command, then `Voice command spoken.` will be printed in the Natlink messages window. If you're not using Dragon, then it will be printed into the console window.

CHAPTER 2

MappingRule usage example

A more common use of Dragonfly is the `MappingRule` class, which allows defining multiple voice commands. The following example is a simple grammar to be used when Notepad is the foreground window:

```
from dragonfly import (Grammar, AppContext, MappingRule, Dictation,
                       Key, Text)

# Voice command rule combining spoken forms and action execution.
class NotepadRule(MappingRule):
    # Define the commands and the actions they execute.
    mapping = {
        "save [file]":            Key("c-s"),
        "save [file] as":         Key("a-f, a/20"),
        "save [file] as <text>":  Key("a-f, a/20") + Text("%(text)s"),
        "find <text>":            Key("c-f/20") + Text("%(text)s\n"),
    }

    # Define the extras list of Dragonfly elements which are available
    # to be used in mapping specs and actions.
    extras = [
        Dictation("text")
    ]


# Create the grammar and the context under which it'll be active.
context = AppContext(executable="notepad")
grammar = Grammar("Notepad example", context=context)

# Add the command rule to the grammar and load it.
grammar.add_rule(NotepadRule())
grammar.load()
```

To use this example, save it in a command module in your module loader directory or Natlink user directory, load it, open a Notepad window and then say one of mapping commands. For example, saying *save* or *save file* will cause the control and S keys to be pressed.

CHAPTER 3

Documentation

Besides this page, the following documentation is also available:

3.1 Introduction

Contents:

3.1.1 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 & above), Kaldi (cross-platform), and CMU Pocket Sphinx (cross-platform).

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. This framework is cross-platform, working on Windows, macOS and Linux (X11 only). See the *actions sub-package documentation* for more information, including code examples.

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.

3.1.2 Installation

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

Prerequisites

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

- **Python** – available from the Python dowloads page. **Version 2.7 (32-bit) is required if using Natlink**.
- **Win32 extensions for Python** *(only for Windows users)* – available from the pywin32 releases page.
- **Natlink** *(only for Dragon users)* – latest versions available from SourceForge.
- wmctrl, xdotool and xsel programs *(only for Linux/X11 users)* – usually available from your system's package manager.

Note on Python 2.7: Python version 2.7 (32-bit) is required if using the Natlink engine back-end, at least for the moment. Support for this version is not maintained for the other engine back-ends and will be **dropped completely** in the first *MAJOR* release following stable Natlink support for Python 3.

Note for Linux users: Dragonfly is only fully functional in an X11 session. You may also need to manually set the `DISPLAY` environment variable. Input action classes, application contexts and the `Window` class will **not** be functional under Wayland. It is recommended that Wayland users switch to X11.

Installation of Dragonfly

Dragonfly is a Python package. It can be installed as *dragonfly2* using pip:

```
pip install dragonfly2
```

The distribution name has been changed to *dragonfly2* in order to upload releases to PyPI.org, but everything can still be imported using *dragonfly*. If you use any grammar modules that include something like `pkg_resources.require("dragonfly >= 0.6.5")`, you will need to either replace `dragonfly` with `dragonfly2` or remove lines like this altogether.

If you have dragonfly installed under the original *dragonfly* distribution name, you'll need to remove the old version using:

```
pip uninstall dragonfly
```

Dragonfly can also be installed by cloning this repository or downloading it from the releases page and running the following (or similar) command in the project's root directory:

```
python setup.py install
```

If pip fails to install *dragonfly2* or any of its required or extra dependencies, then you may need to upgrade pip with the following command:

```
pip install --upgrade pip
```

Installation for specific SR engine back-ends

Each Dragonfly speech recognition engine back-end and its requirements are documented separately:

- *Natlink and DNS engine back-end*
- *SAPI 5 and WSR engine back-end*
- *Kaldi engine back-end*
- *CMU Pocket Sphinx engine back-end*
- *Text-input engine back-end*

3.1.3 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's FAQ*
- Dragonfly-related Stackoverflow Questions
- 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
  - Caster — A collection of Dragonfly-based tools aimed at enabling programming entirely by voice
- Dragonfluid — A Dragonfly extension to allow voice commands to be spoken together without pausing. Supports Dragon NaturallySpeaking and Windows Speech Recognition
- 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
- Christo Butcher's command modules
- WhIzz2000's command modules
- Hawkeye Parker's command modules
- Simian Hacker's Code-by-Voice — Support files for Chris Cowan's "code by voice" setup using Dragon NaturallySpeaking and Dragonfly
- 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:

- Tylercal's repo

Unrelated

The following 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
- Getting Started with Eye Tracking — James Stout's hands-free coding blog post on using eye tracking with dragonfly
- EyeXMouse — Eric Paulson's fork of EyeXMouse for cursor control using the Tobii EyeX eye tracker

3.2 Frequently Asked Questions (FAQ)

The following is a list of frequently asked questions related to the Dragonfly speech recognition framework.

Table of Contents

3.2.1 General Questions

What is Dragonfly?

Dragonfly is a speech recognition framework for Python that makes it convenient to create custom commands to use with speech recognition software. It 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.

Dragonfly can be used for general programming by voice. It is flexible enough to allow programming in any language, not just Python. It can also be used for speech-enabling applications, automating computer activities and dictating prose.

Which speech recognition software and operating systems are supported?

Dragonfly supports the following speech recognition (SR) engines:

- *Dragon*, a product of *Nuance*. All versions up to 15 (the latest) are supported. *Home*, *Professional Individual* and previous similar editions of *Dragon* are supported. Other editions may work too
- *Windows Speech Recognition* (WSR), included with Microsoft Windows Vista, Windows 7+, and freely available for Windows XP
- *Kaldi*
- *CMU Pocket Sphinx*

Dragonfly has cross platform support for Windows, macOS and Linux (using X11). The following table shows which engines are available on which platforms:

Operating system	Available SR engines
Windows	DNS, WSR, Kaldi, Sphinx
Linux	Kaldi, Sphinx
macOS	Kaldi, Sphinx

Windows-only speech recognition software, i.e. DNS and WSR, can be used to control Linux or macOS machines via Aenea, a client-server library for using Dragonfly voice macros on remote hosts.

Dragonfly's X11 support should work just fine on non-Linux unices, such as FreeBSD. If you are planning to use the Kaldi SR engine backend on a platform like FreeBSD, you will need to compile the Kaldi engine dependencies manually.

Where can I find examples Dragonfly command modules?

There is a list of repositories and other projects containing Dragonfly command modules under the *Related resources* -> *Command modules* section of the documentation. There are also example command modules in dragonfly/examples.

What is the difference between dragonfly and dragonfly2?

Dragonfly is the original project written by Christo Butcher (t4ngo). It is no longer actively maintained. *Dragonfly2* is a fork of dragonfly that uses a different *distribution* name in order to upload releases to the Python Package Index, so that the package can be installed by running:

```
pip install dragonfly2
```

It is important to note that the import name is still "dragonfly":

```
from dragonfly import Grammar, MappingRule, Key, Text, Mouse, Dictation
```

Dragonfly2 is intended to be backwards-compatible continuation of the original project. Many bugs and other issues are fixed in this version. It supports using additional speech recognition engine backends (e.g. the *Kaldi engine*). It also works with Python 3 and has cross-platform support for Windows, GNU/Linux and macOS. Dragonfly2 also has many other new features not found in the old version.

See the *changelog* for the full list of changes between the two versions.

How can I use older Dragonfly scripts with Dragonfly2?

Older dragonfly scripts are mostly written with Python 2.x in mind. Python version 2.7 has reached the end of its life as of January 2020 (see Python 2.7 EOL). For complicated reasons, Dragonfly's Python 3.x support has come a bit later than most other active projects. You will need to convert older Python 2.x code, to use it with Python 3.x. There are a few ways to convert older code:

- 2to3 - command-line program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code.
- python-modernize - a command-line program that uses *2to3* to make Python 2 code compatible with Python 3.

You may be interested in the Python 2-3 code porting guide if you prefer to do things manually.

A number of older dragonfly command modules also include the following code:

```
try:
    import pkg_resources
    pkg_resources.require("dragonfly >= 0.6.5")
except ImportError:
    pass
```

Since the distribution name has been changed to *dragonfly2*, you will need to either replace `dragonfly` with `dragonfly2` or remove code like this altogether.

Where are some good resources on learning Python?

If you just want to use Dragonfly for flexible computer control or for programming in other languages and you don't have much background in Python, then the following resources from the Python Software Foundation might be useful to you:

- Beginner's Guide for non-programmers
- Beginner's Guide for programmers
- The Python Tutorial
- Latest Python documentation

3.2.2 API Questions

How do I use an "extra" in a Dragonfly spec multiple times?

Sometimes it is desirable to use the same "extra" multiple times in a Dragonfly `Compound`, `CompoundRule` or `MappingRule` specification (or "spec"). You **cannot** use the same reference name in the same spec. However, there is always an efficient solution available using multiple names. Solutions to two common problems are listed below using the generic compound spec `"<X1> and <X2>"`.

```
from dragonfly import IntegerRef, Choice, RuleRef, RuleWrap

# For saying and processing two numbers, e.g. "one and ten".
int_extras = [
    IntegerRef("X1", 1, 20),
    IntegerRef("X2", 1, 20)
]

# For saying and processing a Choice element two times,
# e.g. "alpha and bravo".
my_choice = Choice("", {
    "alpha": "a",
    "bravo": "b",
    "charlie": "c"
})
# Use RuleWrap to wrap the Choice element into a private rule only
# pronounceable via references (i.e. with RuleRef elements).
# This is more efficient than using two identical Choice elements.
my_choice_rule = RuleWrap("", my_choice).rule
alpha_extras = [
    RuleRef(my_choice_rule, "X1"),
    RuleRef(my_choice_rule, "X2")
]
```

All of these example *extras* lists and their elements can be used with `Compound` or `Choice` elements or `CompoundRule` or `MappingRule` grammar rules.

Is there a way to re-use a function with different "extra" names?

Dragonfly's `Function` action class is normally used to call a Python function when a spoken command is recognized. `Function` actions pass recognized "extra" values via key word arguments, rather than positional arguments.

Below are two methods to re-use a Python function without redefining it:

```
from dragonfly import Function

# Define a function to be used by two Function actions.
def add_and_print(x, y):
    print("%d" % (x + y))

# --- Method one ---
# Use a lambda function.
Function(lambda x, z: add_and_print(x, z))

# --- Method two ---
# Use the optional 'remap_data' argument to pass the 'z' argument
# as 'y' internally.
Function(add_and_print, dict(z='y'))
```

See the *Function action's documentation* for more information and code examples.

Is there a way to recognize negative integers with Dragonfly?

Yes. The simplest way of recognizing negative integers is to use `IntegerRef` and `Modifier` elements together in a command with an appropriate prefix word such as "negative" or "minus":

```
from dragonfly import IntegerRef, Modifier, Text

# Define a MappingRule command for typing a negative integer.
mapping = {
    "(minus|negative) <n>": Text("%(n)d"),
}

# The special Modifier element lets us modify the value of an element.
# Here we use it to specify the "n" extra as a negated integer between 1
# and 50.
extras = [
    Modifier(IntegerRef("n", 1, 50), lambda n: n*-1)
]
```

Is there a way to construct Dragonfly grammars manually with elements?

Yes. The *dragonfly.grammar.rule_basic.BasicRule* is the rule class to use for constructing Dragonfly rules and grammars manually with elements instead of with compound specs and extras.

The following is an example of how to use `BasicRule` and common Dragonfly element and action classes together:

```
from dragonfly import (BasicRule, Repetition, Alternative, Literal, Text,
                       Grammar)

class ExampleRule(BasicRule):
    # Define a rule element that accepts 1 to 5 (exclusive) repetitions
    # of either 'test one', 'test two' or 'test three'. These commands
    # type their respective numbers in succession using the Text action.
    element = Repetition(
        Alternative((
            Literal("test one", value=Text("1")),
            Literal("test two", value=Text("2")),
            Literal("test three", value=Text("3")),
        )),
        1, 5
    )

# Create a grammar with the example rule and load it.
rule = ExampleRule()
grammar = Grammar("BasicRule Example")
grammar.add_rule(rule)
grammar.load()
```

Please note that extras in action specification strings (e.g. *n* in `Key("left:%(n)d")`) will **not** work for the `BasicRule` class. For this functionality, you should use `CompoundRule` or `MappingRule` instead. You can also override the `_process_recognition()` method and use the `node` object to retrieve the desired extra / element and its value.

Does Dragonfly support using Windows Speech Recognition with the GUI?

Yes. To use WSR with the GUI, you need to initialize the SAPI5 shared process engine in the module loader script file:

```
from dragonfly import get_engine
get_engine("sapi5shared")
```

If you are using Dragonfly's command-line interface, then you need to pass "sapi5shared" as the engine name:

```
python -m dragonfly load -e sapi5shared _\*.py
```

There are significant issues with using WSR's shared recognizer for command-based speech recognition. This is because of the built-in commands and dictation output. Dragonfly defaults to the *in-process* SAPI5 engine because it doesn't have these defaults.

Is there an easy way to check which speech recognition engine is in use?

Yes. The current engine can be checked using the *dragonfly.engines.get_current_engine()* function. The following code prints the name of the current engine if one has been initialized:

```
from dragonfly import get_current_engine
engine = get_current_engine()
if engine:
    print("Engine name: %r" % engine.name)
else:
    print("No engine has been initialized.")
```

3.2.3 Troubleshooting Questions

Why are my command modules are not being loaded/detected?

If you have placed Python files into the *MacroSystem* / user directory (using DNS/Natlink) or the directory where your module loader script is (using another engine) and there is no indication that the files were loaded, then there can be a few reasons why:

1. Your Python files don't start with an underscore _ and end with *.py*.
2. You've put the files in the wrong directory. If you're using Natlink, then try running the Natlink configuration-program to double check where Natlink loads files from.

In the case that your command modules are being loaded and you're getting error messages not mentioned in the FAQ, then see the *Unanswered Questions* section.

How do I fix "No handlers could be found for logger X" error messages?

This error is specific to Python 2.x. It isn't a Dragonfly error, but as many users still use Python 2.7, it is listed here. This is the most common example of the error:

```
No handlers could be found for logger "action"
```

There are two easy methods for to solving this problem:

```
# --- Method one ---
# Set up a basic logging handler for console output using the 'logging'
# module.
import logging
logging.basicConfig()

# --- Method two ---
# Set up Dragonfly's logging handler from the 'dragonfly.log' module.
# This sets up a logging handler for console output, appends log messages
# to a log file (~/.dragonfly.log) and sets sane defaults for Dragonfly's
# internal loggers.
from dragonfly.log import setup_log
setup_log()
```

For either method, add the two lines of code near the top of one of your command modules or command module loader script, if you use one.

Cannot load compatibility module support error when starting Dragon

This is a known issue with Natlink. Please see this Natlink troubleshooting page for solutions on how to solve this and other issues that occur before the Natlink messages window appears.

Why isn't GUI code working properly when using Natlink?

Common GUI libraries won't work in modules loaded by Natlink because the Python interpreter used by Natlink is embedded. This also prevents use of Python's multiprocessing module.

Running GUI-related code in modules loaded by Natlink can cause Dragon or Python to crash. For example, Python will crash if the pywin32 `win32ui` module is imported when using Natlink out of process via the *command-line interface* or an external module loader script (see Dragonfly issue #205).

Dragonfly's *Remote Procedure Call (RPC) sub-package* can be used to write external GUI programs/components with Dragonfly speech recognition engine integration without the above concerns.

Why isn't multi-threaded code working when using Natlink?

Python threads should work normally with the Natlink backend in *dragonfly2* versions 0.23.0 and above. Python threads will start in previous versions, but will hang and only run very occasionally. If Python threads are still hanging when using the Natlink engine backend, then try adding the following code in one of your command modules:

```
from dragonfly import get_engine
get_engine().apply_threading_fix()
```

The `apply_threading_fix()` method is called automatically when a grammar is loaded for the first time. Calling it manually is only required if you need Python threads to work *before* any Dragonfly grammar is loaded. The method only needs to be called once.

If threads still aren't working properly for you, please see the *Unanswered Questions* section.

How do I fix "failed to decode recognition" errors?

"Failed to decode recognition" is the error message displayed when Dragonfly is unable to match what was said to a grammar rule. This can occur when saying command words to match a dictation part of a rule.

One way around this to add a top-level grammar rule for dictating other words in your rules:

```
from dragonfly import Dictation, Text

mapping = {
    "reserved (word|words) <text>": Text("%(text)s")
}

extras = [
    Dictation("text")
]
```

Another way around the problem is to have an "extra" for reserved words:

```
from dragonfly import Choice, Text

mapping = {
    "type <reserved>": Text("%(reserved)s")
}

extras = [
    Choice("reserved", {
        "alpha": "alpha",
        "bravo": "bravo",
        "charlie": "charlie",
    })
]
```

How can I increase the speech recognition accuracy?

Low recognition accuracy is usually caused by either bad-quality audio input or a speech model that isn't trained to your voice or use case. You might try the following:

- Re-positioning your microphone.
- Using a different microphone.
- Training words or phrases.
- Change the speech recognition engine settings (e.g. adjust Dragon's accuracy/speed slider).
- Using a different engine back-end if possible, e.g. the Kaldi back-end is typically more accurate than CMU Pocket Sphinx and WSR back-ends.

Dragonfly also has programmatic methods for increasing recognition accuracy. They can be used to fine tune accuracy for specific commands or parts of commands:

1. *Kaldi Grammar/Rule/Element Weights* (Kaldi-only)
2. Quoted words in *dragonfly.grammar.elements_basic.Literal* elements (only applies to Dragon)

Why isn't Dragonfly code aware of DPI scaling settings on Windows?

There can be problems with Dragonfly's monitor-related functionality on Windows Vista and above if the system is set up to use one or more monitors with a high number of dots per inch (DPI). For this reason, Dragonfly *attempts* to set the DPI awareness for the process when it is imported. The `SetProcessDpiAwareness()` function is used to do this on Windows 8.1 and above.

If you need to set the DPI awareness manually using a different DPI awareness value, do so before importing dragonfly. The following is equivalent to what dragonfly does internally:

```
import ctypes
ctypes.windll.shcore.SetProcessDpiAwareness(2)   # PROCESS_PER_MONITOR_DPI_AWARE
```

The `SetProcessDpiAware()` function can be used instead on older Windows versions (e.g. Vista and 7). The `SetProcessDpiAwarenessContext()` function can be used on Windows 10 (version 1703) and above. For more information on this topic, please see the following Microsoft documentation pages:

- High DPI Desktop Application Development on Windows
- Setting the default DPI awareness for a process

Why aren't Dragonfly actions working with Windows admin applications?

Since Windows Vista, Windows has built-in security features to isolate processes with different integrity levels. This is done to prevent user-level processes from performing tasks that require administrative authorization via User Account Control. To this end, Windows also prevents user-level processes from sending keystrokes, mouse events or otherwise controlling processes or windows running as the administrator. Since our Python code normally runs as the user, Windows prevents us from interacting with administrative windows.

Windows has a way for accessibility software to run with special privileges. This involves building a signed executable with a special manifest `uiAccess="true"` attribute set and installing it under either the *Program Files* or *Windows/System32* directories. For more on this, see Microsoft's Security Considerations for Assistive Technologies documentation page.

Unfortunately, this is not easily achievable with Python programs. Developing a secure UI Access solution for Dragonfly would be quite complicated and, given that it is a small project with only a few developers, present significant security risks. For these reasons, Dragonfly will **not** be implementing UI Automation support. The following are a few alternative solutions:

1. Use Dragon's UI Automation capability

   Dragon runs a UI Automation service in the background and, clearly, uses it to allow users to interact with administrative applications. This may be good enough if you don't need to perform complex tasks.

2. Run Python as the administrator

   Running the Python process that loads, recognizes and processes your commands as the administrator should work around the limitations. **Be careful if you do this**; Windows won't go as far to stop bugs in your code from doing damage!

3. Use AutoHotkey's *Run with UI Access* feature

   AutoHotkey (AHK) is an automation scripting language for Windows. One of its many features allows running AHK code with UI Access. See the relevant AutoHotkey FAQ on UAC for how to set this up and use it.

   Using this, you can define hotkeys for interacting with administrative windows. This can help if you are able to use a keyboard, even if only for a short time. Unfortunately, Dragonfly's actions won't be able to trigger these hotkeys since the AHK code will be running in elevated mode. One way around this is to run a local web server in your AHK script, perhaps using AHKhttp or something similar. Then you can send HTTP requests to the server to run your AHK code.

   If you do use the local web server approach mentioned, it is very important to ensure that requests received by the server are properly authorized so that only your Python code has (indirect) UI access (see Web API security).

Why aren't Dragonfly's input actions working on my Linux system?

Dragonfly's `Key`, `Text` and `Mouse` action classes use the xdotool program on Linux. These actions will not work if it isn't installed. It can normally be installed through your system's package manager. On Debian-based or Ubuntu-based systems, this is done by running the following console command:

```
sudo apt install xdotool
```

The `Window` class also requires the wmctrl program:

```
sudo apt install wmctrl
```

The keyboard/mouse input classes will only work in an X11 session. You will get the following error if you are using Wayland or something else:

```
NotImplementedError: Keyboard support is not implemented for this platform!
```

If you see this message, then you will probably need to switch to X11 or use something like ydotool to have keyboard/mouse input work properly.

If you are using X11 and still see this message, then it means that the `DISPLAY` environment variable checked by Dragonfly internally is not set.

3.2.4 Unanswered Questions

If your question isn't listed above, then there are a few ways to get in touch:

- Open a new issue on GitHub.
- Join one of Dragonfly's chat channels:
  - Gitter channel
  - Matrix channel
- Ask your question on the Dragonfly mailing list.
- Send an email to Dane Finlay, the project maintainer, at Danesprite@posteo.net.

3.3 Object model (Grammar sub-package)

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

3.3.1 Grammars

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. If defined, this method should return whether to continue rule processing afterwards (`True` or `False`).
- `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.

Example Grammar using recognition callbacks

```
from dragonfly import Grammar

class CallbackGrammar(Grammar):

    def process_recognition(self, words, results):
        print("process_recognition()")
```

(continues on next page)

(continued from previous page)

```
        print(words)
        print(results)

        # Grammar rule processing should continue after this method.
        return True

    def process_recognition_other(self, words, results):
        print("process_recognition_other()")
        print(words)
        print(results)

    def process_recognition_failure(self, results):
        print("process_recognition_failure()")
        print(results)
```

Recognition callbacks with results objects

The last three methods mentioned above can define an optional `results` parameter, the value of which differs between each SR engine back-end:

SR engine back-end	Type of `results` objects
Dragon/Natlink	`ResObj`*[0]
Kaldi	`Recognition`†[0]
CMU Pocket Sphinx	`None`
WSR/SAPI 5	`ISpeechRecoResultDispatch`‡[0159]
Text input ("text")	`None`

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.

[0] See the natlink.txt file for info on `ResObj`.
[0] See *Kaldi documentation section*.
[0] See the SAPI 5 documentation on ISpeechRecoResultDispatch for how to use it.
[159] SAPI 5 does not yield results objects for other grammars, so `process_recognition_other()` callbacks will return `None` instead.

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.

activate_rule(*rule*)
Activate a rule loaded in this grammar.

Internal: this method is normally *not* called directly by the user, but instead automatically when the rule itself is activated by the user.

active_rules
List of a grammar's active rules.

add_all_dependencies()
Iterate through the grammar's rules and add all the necessary dependencies.

Internal This method is called when the grammar is loaded.

add_dependency(*dep*)
Add a rule or list dependency to this grammar.

Internal: this method is normally *not* called by the user, but instead automatically during grammar compilation.

add_list(*lst*)
Add a list to this grammar.

Lists **cannot** be added to grammars that are currently loaded.

Parameters **lst** (ListBase) – Dragonfly list

add_rule(*rule*)
Add a rule to this grammar.

The following rules apply when adding rules into grammars:

1. Rules **cannot** be added to grammars that are currently loaded.
2. Two or more rules with the same name are **not** allowed.

> **Warning:** Note that while adding the same Rule object to more than one grammar is allowed, it is **not** recommended! This is because the context and active/enabled states of these rules will not function correctly if used. It is better to use *separate* Rule instances for each grammar instead.

Parameters **rule** (Rule) – Dragonfly rule

context
A grammar's context, under which it and its rules will be active and receive recognitions if it is also enabled.

deactivate_rule(*rule*)
Deactivate a rule loaded in this grammar.

Internal: this method is normally *not* called directly by the user, but instead automatically when the rule itself is deactivated by the user.

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.

get_complexity_string()
Build and return a human-readable text giving insight into the complexity of this grammar.

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.

remove_list(*lst*)
Remove a list from this grammar.

Lists **cannot** be removed from grammars that are currently loaded.
Parameters **lst** (ListBase) – Dragonfly list

remove_rule(*rule*)
Remove a rule from this grammar.

Rules **cannot** be removed from grammars that are currently loaded.
Parameters **rule** (Rule) – Dragonfly rule

rule_names
List of grammar's rule names.

rules
List of a grammar's rules.

set_context(*context*)
Set the context for this grammar, under which it and its rules will be active and receive recognitions if it is also enabled.

Use of this method overwrites any previous context.

Contexts can be modified at any time, but will only be checked when *process_begin()* is called.

Parameters **context** (*Context* | *None*) – context within which to be active. If *None*, the grammar will always be active.

set_exclusive(*exclusive*)
Alias of *set_exclusiveness()*.

set_exclusiveness(*exclusive*)
Set the exclusiveness of this grammar.

unload()
Unload this grammar from its SR engine.

update_list(*lst*)
Update a list's content loaded in this grammar.

Internal: this method is normally *not* called directly by the user, but instead automatically when the list itself is modified by the user.

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.

3.3.2 Rules

This section describes the following classes:

- *dragonfly.grammar.rule_base.Rule* – the base rule class

- *dragonfly.grammar.rule_basic.BasicRule* – a rule class for defining voice commands using elements directly.
- *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 ImportedRule(*name*)

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

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: *True*) – 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)

context
This rule's context, under which it will be active and receive recognitions if it is also enabled and its grammar is active.

disable()
Disable this rule 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 rule 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.

---

set_context(*context*)
Set the context for this rule, under which it will be active and receive recognitions if it is also enabled and its grammar is active.

Use of this method overwrites any previous context.

Contexts can be modified at any time, but will only be checked when *process_begin()* is called.
Parameters **context** (*Context* | *None*) – context within which to be active. If *None*, the rule will be active when its grammar is.

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.

BasicRule class

The BasicRule class is designed to make it easy to create a rule from an element tree, rather than building one indirectly via compound element specs.

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

- *name* (*str*) – the rule's name

- *element* (*Element*) – root element for this rule
- *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.

Note: The BasicRule class has only limited support for the "extras" and "defaults" functionality of the *dragonfly.grammar.rule_compound.CompoundRule* and *dragonfly.grammar.rule_mapping.MappingRule* classes. By default, the *extras* dictionary passed to `_process_recognition()` will only contain an entry for the root element of the rule.

Example usage

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

```
from dragonfly import (BasicRule, Repetition, Alternative, Literal, Text,
                       Grammar)

class ExampleRule(BasicRule):
    # Define a rule element that accepts 1 to 5 (exclusive) repetitions
    # of either 'test one', 'test two' or 'test three'. These commands
    # type their respective numbers in succession using the Text action.
    element = Repetition(
        Alternative((
            Literal("test one", value=Text("1")),
            Literal("test two", value=Text("2")),
            Literal("test three", value=Text("3")),
        )),
        1, 5
    )

# Create a grammar with the example rule and load it.
rule = ExampleRule()
grammar = Grammar("BasicRule Example")
grammar.add_rule(rule)
grammar.load()
```

The above *BasicRule* example can be defined without sub-classing:

```
rule = BasicRule(
    element=Repetition(
        Alternative((
            Literal("test one", value=Text("1")),
            Literal("test two", value=Text("2")),
            Literal("test three", value=Text("3")),
        )),
        1, 5)
)
```

Class reference

class BasicRule(*name=None*, *element=None*, *exported=None*, *context=None*)

Rule class for implementing complete or partial voice-commands defined using an element.

Constructor arguments:

- *name* (*str*) – the rule's name
- *element* (*Element*) – root element for this rule
- *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.

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",
```

(continues on next page)

(continued from previous page)

```
                                "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.

specs

Each spoken-form in the rule. :rtype: list

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.

3.3.3 Lists

This section describes the following classes:

- *dragonfly.grammar.list.ListBase* – the base list class
- *dragonfly.grammar.list.List* – sub-class of Python's built-in `list` type. It can be updated and modified without reloading a grammar.
- *dragonfly.grammar.list.DictList* – sub-class of Python's built-in `dict` type. It can be updated and modified without reloading a grammar.

The *List Updates* section discusses possible performance issues with modifying Dragonfly lists and ways to avoid these issues altogether.

List classes

class ListBase(*name*)
Base class for dragonfly list objects.

valid_types
The types of object at a Dragonfly list can contain.

name
Read-only access to a list's name.

grammar
Set-once access to a list's grammar object.

class List(*name*, **args*, ***kwargs*)
Wrapper for Python's built-in list that supports automatic engine notification of changes.

Use *ListRef* elements in a grammar rule to allow matching speech to list items.

set(*other*)
Set the contents of this list to the contents of another.

append(**args*, ***kwargs*)
Append object to the end of the list.

extend(**args*, ***kwargs*)
Extend list by appending elements from the iterable.

insert(**args*, ***kwargs*)
Insert object before index.

pop(**args*, ***kwargs*)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(**args*, ***kwargs*)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse(**args*, ***kwargs*)
Reverse *IN PLACE*.

sort(**args*, ***kwargs*)
Stable sort *IN PLACE*.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

grammar
Set-once access to a list's grammar object.

index()
Return first index of value.

Raises ValueError if the value is not present.

name
Read-only access to a list's name.

valid_types
The types of object at a Dragonfly list can contain.

class DictList(*name*, **args*, ***kwargs*)
Wrapper for Python's built-in dict that supports automatic engine notification of changes. The object's keys are used as the elements of the engine list, while use of the associated values is left to the user.

Use *DictListRef* elements in a grammar rule to allow matching speech to dictionary keys.

set(*other*)
Set the contents of this dict to the contents of another.

clear() → None. Remove all items from D.

fromkeys(**args*, ***kwargs*)
Create a new dictionary with keys from iterable and values set to value.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(**args*, ***kwargs*)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

copy() → a shallow copy of D

get()
Return the value for key if key is in the dictionary, else default.

grammar
Set-once access to a list's grammar object.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

name
Read-only access to a list's name.

valid_types
The types of object at a Dragonfly list can contain.

values() → an object providing a view on D's values

List Updates

Lists are updated after each modifying operation, e.g. `list.append()`, `list.remove()`, `dict[key] = value`, `dict.pop()`, etc. This is fine for a few list modifications here and there, but is inefficient for adding / removing many items at once.

The simplest solution is to use the *ListBase* context manager:

```
# Do list modification inside a 'with' block to only do one list update
# at the end.
my_list = List("my_list")
with my_list:
    for x in range(50):
        my_list.append(str(x))
```

Some methods like `list.extend()` or `dict.update()` will also only update the list once afterwards:

```
# Add multiple list items using extend().
my_list = List("my_list")
my_list.extend([str(x) for x in range(50)])

# Add multiple dictionary keys using update().
dictionary = DictList("dictionary")
dictionary.update({str(x):x for x in range(50)})
```

3.3.4 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
- *Impossible* – a special element that cannot be recognized
- *Empty* – a special element that is always recognized

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
- *Modifier* – modifies the output of another element by applying a function to it following recognition
- *DictListRef* – reference to a `dragonfly.DictList` object; this element is similar to the `dragonfly.ListRef` element, except that it returns the value associated with the spoken words instead of the spoken words themselves
- *RuleWrap* – an element class used to wrap a Dragonfly element into a new private rule to be referenced by the same element or other *RuleRef* elements

See the following documentation sections for additional information and usage examples:

- *Elements (Object model section)*
- *Doctests for the fundamental element classes*

Compound element classes

The following special *element* classes exist as convenient ways of constructing basic element types from string specifications:

- *Compound* – a special element which parses a string spec to create a hierarchy of basic elements.
- *Choice* – a special element taking a `choice` dictionary argument, interpreting keys as *Compound* string specifications and values for what to return when compound specs are successfully decoded during the recognition process.

  The `choice` argument may also be a list or tuple of strings, in which case the strings are also interpreted as *Compound* strings specifications. However, the values returned when compound specs are successfully decoded during the recognition process are the recognized words. **Note**: these values will be matching part(s) of the compound specs.

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.
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

_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
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

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)

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.

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*)
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
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

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)

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.

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*)
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

- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

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)

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.

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*)
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*, *optimize=True*)
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
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken
- *optimize* (*bool*, default: *True*) – whether the engine's compiler should compile the element optimally

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.

If the *optimize* argument is set to *True*, the compiler will ignore the *min* and *max* limits to reduce grammar complexity. If the number of repetitions recognized is more than the *max* value, the rule will fail to match.

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.

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

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*)

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*, *quote_start_str='"'*, *quote_end_str='"'*, *strip_quote_strs=True*)

Element class representing one or more literal words which must be said exactly by the speaker as given.

Quoted words can be used to potentially improve accuracy. This currently only has an effect if using the Natlink SR engine back-end.

Constructor arguments:

- *text* (*str*) – the words to be said by the speaker
- *name* (*str*, default: *None*) – the name of this element
- *value* (*object*, default: *the recognized words*) – value returned when this element is successfully decoded
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken
- *quote_start_str* (*str*, default: '') – the string used for specifying the start of quoted words.
- *quote_end_str* (*str*, default: '') – the string used for specifying the end of quoted words.
- *strip_quote_strs* (*bool*, default: *True*) – whether the start and end quote strings should be stripped from this element's word lists.

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.

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()*).

words

The words to be said by the speaker.

words_ext
The words to be said by the speaker, with any quoted words as single items. This is extends the *words* property.

RuleRef class

class RuleRef(*rule*, *name=None*, *default=None*)
Element class representing a reference to another Dragonfly rule.
Constructor arguments:
- *rule* (*Rule*) – the Dragonfly rule to reference
- *name* (*str*, default: *None*) – the name of this element
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

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.

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()*).

ListRef class

class ListRef(*name*, *list*, *key=None*, *default=None*)
Element class representing a reference to a Dragonfly List.
Constructor arguments:
- *name* (*str*, default: *None*) – the name of this element
- *list* (*ListBase*) – the Dragonfly List to reference
- *key* (*object*, default: *None*) – key to differentiate between list references at runtime
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

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.

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()*).

DictListRef class

class DictListRef(*name*, *dict*, *key=None*, *default=None*)
Element class representing a reference to a Dragonfly DictList.
Constructor arguments:
- *name* (*str*, default: *None*) – the name of this element
- *dict* (*DictList*) – the Dragonfly DictList to reference
- *key* (*object*, default: *None*) – key to differentiate between list references at runtime
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

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.

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()*).

Impossible class

class Impossible(*name=None*)
Element class representing speech that cannot be recognized.
Constructor arguments:
- *name* (*str*, default: *None*) – the name of this element

Using an *Impossible* element in a Dragonfly rule makes it impossible to recognize via speech or mimicry.

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.

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()*).

Empty class

class Empty(*name=None*, *value=True*, *default=None*)
Element class representing something that is always recognized.
Constructor arguments:
- *name* (*str*, default: *None*) – the name of this element
- *value* (*object*, default: *True*) – value returned when this element is successfully decoded (always)

Empty elements are equivalent to children of *Optional* elements.

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.

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()*).

Dictation class

class Dictation(*name=None*, *format=True*, *default=None*)
Element class representing free dictation.
Constructor arguments:

- *name* (*str*, default: *None*) – the name of this element
- *format* (*bool*, default: *True*) – whether the value returned should be a `DictationContainerBase` object. If *False*, then the returned value is a list of the recognized words
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

Returns a string-like `DictationContainerBase` object containing the recognised words. By default this is formatted as a lowercase sentence. Alternative formatting can be applied by calling string methods like *replace* or *upper* on a *Dictation* object, or by passing an arbitrary formatting function (taking and returning a string) to the *apply* method.

Camel case text can be produced using the *camel* method. For example:

```
str_func = lambda s: s.upper()
Dictation("formattedtext").apply(str_func)
Dictation("snake_text").lower().replace(" ", "_")
Dictation("camelText").camel()
```

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.

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()*).

Modifier class

class Modifier(*element*, *modifier=None*)
Element allowing direct modification of the output of another element at recognition time.
Constructor arguments:
- *element* (*Element*) – The element to be recognised, e.g. *Dictation* or *Repetition*, with appropriate arguments passed.
- *modifier* (*function*) – A function to be applied to the value of this element when it is recognised.

Examples:

```
# Recognises an integer, returns the integer plus one
Modifier(IntegerRef("plus1", 1, 20), lambda n: n+1)

# Recognises a series of integers, returns them separated by
# commas as a string
int_rep = Repetition(IntegerRef("", 0, 10), min=1, max=5,
```

(continues on next page)

(continued from previous page)

```
                      name="num_seq")
Modifier(int_rep, lambda r: ", ".join(map(str, r)))
```

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.

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

RuleWrap class

class RuleWrap(*name*, *element*, *default=None*)
Element class to wrap a Dragonfly element into a new private rule to be referenced by this element or others.

RuleWrap is a sub-class of *RuleRef*, so *RuleWrap* elements can be used in the same way as *RuleRef* elements.
Constructor arguments:

- *name* (*str*, default: *None*) – the name of this element
- *element* (*Element*) – the Dragonfly element to be wrapped
- *default* (*object*, default: *None*) – the default value used if this element is optional and wasn't spoken

Examples:

```
# For saying and processing a Choice element two times.
letter = RuleWrap("letter1", Choice("", {
    "alpha": "a",
    "bravo": "b",
    "charlie": "c"
}))
letter_extras = [
    letter,
    RuleRef(letter.rule, "letter2"),
    RuleRef(letter.rule, "letter3")
]
```

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.

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()*).

Compound class

class Compound(*spec*, *extras=None*, *actions=None*, *name=None*, *value=None*, *value_func=None*, *elements=None*, *default=None*)
Element which parses a string spec to create a hierarchy of basic elements.
Constructor arguments:
- *spec* (*str*) – compound specification
- *extras* (sequence) – extras elements referenced from the compound spec
- *actions* (*dict*) – this argument is currently unused
- *name* (*str*) – the name of this element
- *value* (*object*, default: *None*) – value to be returned when this element is successfully decoded If *None*, then the value(s) of child nodes are used instead
- *value_func* (*callable*, default: *None*) – function to be called for the node value when this element is successfully decoded. If *None*, then the value(s) of child nodes are used. This argument takes precedence over the *value* argument if it is present
- *elements* (sequence) – same as the extras argument
- *default* (default: *None*) – the default value of this element

Example:

```
# Define a command to type the sum of two spoken integers between
# 1 and 50 using a Compound element.
mapping = {
    "type <XAndY>": Text("%(XAndY)d"),
}
extras = [
    Compound(
        # Pass the compound spec and element name.
        spec="<x> and <y>",
        name="XAndY",

        # Pass the internal IntegerRef extras.
        extras=[IntegerRef("x", 1, 50), IntegerRef("y", 1, 50)],

        # Pass a value function that adds the two spoken integers
        # together.
        value_func=lambda node, extras: extras["x"] + extras["y"])
]
```

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

Choice class

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

Element allowing a dictionary of phrases (compound specs) to be recognized to be mapped to objects to be used in an action.

A list or tuple of phrases to be recognized may also be used. In this case the strings are also interpreted as *Compound* string specifications. However, the values returned when compound specs are successfully decoded during the recognition process are the recognized words. **Note**: these values will be matching part(s) of the compound specs.

Constructor arguments:

- *name* (*str*) – the name of this element
- *choices* (*dict*, *list* or *tuple*) – dictionary mapping recognized phrases to returned values **or** a list/tuple of recognized phrases
- *extras* (*list*, default: *None*) – a list of included extras
- *default* (default: *None*) – the default value of this element

Example using a dictionary:

```
# Tab switching command, e.g. 'third tab'.
mapping = {
    "<nth> tab": Key("c-%(nth)s"),
}
extras = [
    Choice("nth", {
        "first"          : "1",
        "second"         : "2",
        "third"          : "3",
        "fourth"         : "4",
        "fifth"          : "5",
        "sixth"          : "6",
        "seventh"        : "7",
        "eighth"         : "8",
        "(last | ninth)": "9",
        "next"           : "pgdown",
        "previous"       : "pgup",
    }),
]
```

Example using a list:

```
# Command for recognizing and typing nth words, e.g.
# 'type third'.
mapping = {
    "type <nth>": Text("%(nth)s"),
}
extras = [
    Choice("nth", [
        "first",
        "second",
        "third",
        "fourth",
        "fifth",
        "sixth",
        "seventh",
        "eighth",

        # Note that the decoded value for a compound spec like
```

(continues on next page)

(continued from previous page)

```
        #  this one, when used in a list/tuple of choices,
        #  rather than a dictionary, is the recognized word:
        #  "last" or "ninth".
        "(last | ninth)",

        "next",
        "previous",
    ]),
]
```

3.3.5 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 is based on the application context, i.e. foreground window executable, title, and handle
- *FuncContext* – class that evaluates a given function/lambda/callable, whose return value is interpreted as a *bool*, determining whether the context is active

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
```

Matching other window attributes

The *AppContext* class can be used to match window attributes and properties other than the title and executable. To do this, pass extra keyword arguments to the constructor:

```
# Context for a maximized Firefox window.
maximized_firefox = AppContext(executable="firefox", is_maximized=True)

# Context for a browser in fullscreen mode.
# 'role' and 'is_fullscreen' are X11 only.
fullscreen_browser = AppContext(role="browser", is_fullscreen=True)
```

(continues on next page)

(continued from previous page)

```
# Context for Android Studio or PyCharm using the X11 'cls' property.
AppContext(cls=["jetbrains-studio", "jetbrains-pycharm-ce"])
```

Class reference

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

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.

If multiple strings are passed in a list, True will be returned if the foreground window matches one or more of them. This applies to the *executable* and *title* arguments and key word arguments for most window attributes.

Constructor arguments:

- *executable* (*str* or *list*) – (part of) the path name of the foreground application's executable; case insensitive
- *title* (*str* or *list*) – (part of) the title of the foreground window; case insensitive
- *key word arguments* – optional window attributes/properties and expected values; case insensitive

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.

---

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.

---

class FuncContext(*function*, ***defaults*)

Context class that evaluates a given function, whose return value is interpreted as a *bool*, determining whether the context is active.

The foreground application details are optionally passed to the function as arguments named *executable*, *title*, and/or *handle*, if any/each matches a so-named keyword argument of the function. Default arguments may also be passed to the function, through this class's constructor.

Constructor arguments:

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

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.

---

class LogicAndContext(**children*)

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.

---

class LogicNotContext(*child*)

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.

---

class LogicOrContext(**children*)

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.

3.3.6 Recognition observers

This section describes classes and functions for observing Dragonfly's recognition state events:

- `on_begin()` – called when speech start is detected.
- `on_recognition()` – called when speech successfully decoded to a grammar rule or to dictation. This is called *before* grammar rule processing (i.e. `Rule.process_recognition()`).
- `on_failure()` – called when speech failed to decode to a grammar rule or to dictation.
- `on_end()` – called when speech ends, either with a successful recognition or in failure.
- `on_post_recognition()` – called *after* all rule processing has completed after a successful recognition.

Recognition observer classes

class PlaybackHistory (*length=10*)
Storage class for playing back recent recognitions via the `Playback` action.

Instances of this class monitor recognitions and store them internally. This class derives from the built in *list* type and can be accessed as if it were a normal *list* containing `Playback` actions for recent recognitions. Note that an instance's contents are updated automatically as recognitions are received.

class RecognitionHistory (*length=10*)
Storage class for recent recognitions.

Instances of this class monitor recognitions and store them internally. This class derives from the built in *list* type and can be accessed as if it were a normal *list* containing recent recognitions. Note that an instance's contents are updated automatically as recognitions are received.

complete
False if phrase-start detected but no recognition yet.

class RecognitionObserver
Recognition observer base class.

Sub-classes should override one or more of the event methods.

on_begin()
Method called when the observer is registered and speech start is detected.

on_end(*results*)
Method called when speech ends, either with a successful recognition (after `on_recognition`) or in failure (after `on_failure`).
Parameters **results** (*engine-specific type*) – *optional* engine recognition results object

on_failure(*results*)
Method called when speech failed to decode to a grammar rule or to dictation.
Parameters **results** (*engine-specific type*) – *optional* engine recognition results object

on_post_recognition(*words*, *rule*, *node*, *results*)
Method called when speech successfully decoded to a grammar rule or to dictation.

This is called *after* grammar rule processing (i.e. `Rule.process_recognition()`).
Parameters

- **words** (*tuple*) – recognized words
- **rule** (*Rule*) – *optional* recognized rule
- **node** (*Node*) – *optional* parse tree node
- **results** (*engine-specific type*) – *optional* engine recognition results object

on_recognition(*words*, *rule*, *node*, *results*)
Method called when speech successfully decoded to a grammar rule or to dictation.

This is called *before* grammar rule processing (i.e. `Rule.process_recognition()`).
Parameters

- **words** (*tuple*) – recognized words
- **rule** (*Rule*) – *optional* recognized rule
- **node** (*Node*) – *optional* parse tree node
- **results** (*engine-specific type*) – *optional* engine recognition results object

register()
Register the observer for recognition state events.

unregister()
Unregister the observer for recognition state events.

Recognition state change callbacks

class CallbackRecognitionObserver(*event*, *function*)
Observer class for calling recognition state change callbacks.

This class is used by the `register_*_callback` functions and is registered with the current engine on initialization.
Constructor arguments:
- *event* (*str*) – the name of the recognition event to register for (e.g. `"on_begin"`).
- *function* (*callable*) – function to call on the specified recognition event.

register_beginning_callback(*function*)
Register a callback function to be called when speech starts.

The *CallbackRecognitionObserver* object returned from this function can be used to unregister the callback function.
Parameters **function** (*callable*) – callback function
Returns recognition observer
Return type *CallbackRecognitionObserver*

register_ending_callback(*function*)
Register a callback function to be called when speech ends, either successfully (after calling the recognition callback) or in failure (after calling the failure callback).

The *CallbackRecognitionObserver* object returned from this function can be used to unregister the callback function.

Parameters **function** (*callable*) – callback function
Returns recognition observer
Return type *CallbackRecognitionObserver*

register_failure_callback (*function*)
Register a callback function to be called on recognition failures.

The *CallbackRecognitionObserver* object returned from this function can be used to unregister the callback function.

Parameters **function** (*callable*) – callback function
Returns recognition observer
Return type *CallbackRecognitionObserver*

register_post_recognition_callback (*function*)
Register a callback function to be called after all rule processing has completed after recognition success.

The *CallbackRecognitionObserver* object returned from this function can be used to unregister the callback function.

Parameters **function** (*callable*) – callback function
Returns recognition observer
Return type *CallbackRecognitionObserver*

register_recognition_callback (*function*)
Register a callback function to be called on recognition success.

The *CallbackRecognitionObserver* object returned from this function can be used to unregister the callback function.

Parameters **function** (*callable*) – callback function
Returns recognition observer
Return type *CallbackRecognitionObserver*

Doctest usage examples

See Dragonfly's *doctests for recognition observers* for some usage examples.

3.3.7 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.

3.3.8 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.

The *Rule* class and derived classes are described in the *Rules* section.

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

3.3.9 Lists

Lists are dynamic language elements which can be updated and modified without reloading a grammar. There are two list types:

- *`List`* – sub-class of Python's built-in `list` type
- *`DictList`* – sub-class of Python's built-in `dict` type

Lists of either type can be instantiated and used like normal lists / dictionaries. They are meant to be used in Dragonfly rules via the special `ListRef` and `DictListRef` elements. The engine is automatically notified when lists are modified.

The `ListBase` class and derived classes are described in the *Lists* section. See the *Doctests for the List class* for usage examples.

3.3.10 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
- `Repetition` – an element class representing a repetition of one child element
- `RuleRef` – a reference to another rule
- `RuleWrap` – an element class used to wrap a Dragonfly element into a new private rule to be referenced by the same element or other `RuleRef` elements
- `ListRef` – a reference to a Dragonfly `List`
- `DictListRef` – a reference to a Dragonfly `DictList`
- `Dictation` – a free-form dictation element which allows the speaker to say one or more natural language words
- `Modifier` – a special element that allows direct modification of the output of another element at recognition time
- `Impossible` – a special element that cannot be recognized
- `Empty` – a special element that is always recognized, similar to children of `Optional` elements

See the *Doctests for the fundamental element classes* for element usage examples.

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 are therefore also special elements which construct these basic element types from string specifications:

- `Compound` – a special element which parses a string spec to create a hierarchy of basic elements.
- `Choice` – a special element taking a `choice` dictionary argument, interpreting keys as `Compound` string specifications and values for what to return when compound specs are successfully decoded during the recognition process.

  The `choice` argument may also be a list or tuple of strings, in which case the strings are also interpreted as `Compound` string specifications. However, the values returned when compound specs are successfully decoded during the recognition process are the recognized words. **Note**: these values will be matching part(s) of the compound specs.

See the *Doctests for the Compound element class* for `Compound` usage examples and see the *Element classes* section for class references and further documentation on each element class.

3.4 Engines sub-package

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

Also contained within this sub-package are a number of text-to-speech implementations. These can be used independently of the speech recognition engines via the `get_speaker()` function.

3.4.1 Main SR engine back-end interface

get_current_engine()
Get the currently initialized SR engine object.

If an SR engine has not been initialized yet, `None` will be returned instead.

Return type EngineBase | None

Returns engine object or None

Usage example:

```
# Print the name of the current engine if one has been
# initialized.
from dragonfly import get_current_engine
engine = get_current_engine()
if engine:
    print("Engine name: %r" % engine.name)
else:
    print("No engine has been initialized.")
```

get_engine(*name=None*, ***kwargs*)
Get the engine implementation.

This function will initialize an engine instance using the `get_engine` and `is_engine_available` functions in the engine packages and return an instance of the first available engine. If one has already been initialized, it will be returned instead.

If no specific engine is requested and no engine has already been initialized, this function will initialize and return an instance of the first available engine in the following order:

SR engine back-end	Engine name string(s)
1. Dragon/Natlink	`"natlink"`
2. Kaldi	`"kaldi"`
3. WSR/SAPI 5	`"sapi5", "sapi5inproc", "sapi5shared"`
4. CMU Pocket Sphinx	`"sphinx"`

The *Text-input engine* can be initialized by specifying `"text"` as the engine name. This back-end will **not** be initialized if no specific engine is requested because the back-end is not a real SR engine and is used mostly for testing.

Arguments:

Parameters

- **name** (*str*) – optional human-readable name of the engine to return.
- ****kwargs** – optional keyword arguments passed through to the engine for engine-specific configuration.

Return type *EngineBase*
Returns engine instance
Raises EngineError

get_speaker(*name=None*)
Get the text-to-speech (speaker) implementation.

This function will initialize and return a speaker instance instance of the available speaker back-end. If one has already been initialized, it will be returned instead.

If no specific speaker back-end is requested and no speaker has already been initialized, this function will initialize and return an instance of the first available back-end in the following order:

TTS speaker back-end	Speaker name string(s)
1. SAPI 5	`"sapi5"`
2. Dragon/Natlink	`"natlink"`
3. eSpeak	`"espeak"`
4. CMU Flite	`"flite"`
5. Text (stdout)	`"text"`

The first two speaker back-ends are only available on Microsoft Windows. The second requires that Dragon NaturallySpeaking and Natlink are installed on the system.

The third and fourth back-ends, eSpeak and CMU Flite, may be used on most platforms. These require that the appropriate command-line programs are installed on the system.

The last back-end (text) is used as a fallback when no real speaker implementation is available. This back-end writes input text to stdout, i.e., prints text to the console.

Arguments:
Parameters **name** (*str*) – optional human-readable name of the speaker to return.
Return type SpeakerBase
Returns speaker instance
Raises EngineError

register_engine_init(*engine*)
Register initialization of an engine.

This function sets the default engine to the first engine initialized.

register_speaker_init(*speaker*)
Register initialization of a speaker.

This function sets the default speaker to the first speaker initialized.

3.4.2 Engine back-ends

Base engine classes

Contents

EngineBase class

The `dragonfly.engines.base.EngineBase` class forms the base class for the 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.

create_timer(*callback*, *interval*, *repeating=True*)
Create and return a timer using the specified callback and repeat interval.

disconnect()
Disconnect from back-end SR engine.

do_recognition(*begin_callback=None*, *recognition_callback=None*, *failure_callback=None*, *end_callback=None*, *post_recognition_callback=None*, **args*, ***kwargs*)
Recognize speech in a loop until interrupted or `disconnect()` is called.

Recognition callback functions can optionally be registered.

Extra positional and key word arguments are passed to `_do_recognition()`.

Parameters

- **begin_callback** (*callable | None*) – optional function to be called when speech starts.
- **recognition_callback** (*callable | None*) – optional function to be called on recognition success.
- **failure_callback** (*callable | None*) – optional function to be called on recognition failure.
- **end_callback** (*callable | None*) – optional function to be called when speech ends, either successfully (after calling the recognition callback) or in failure (after calling the failure callback).

- **post_recognition_callback** (*callable | None*) – optional function to be called after all rule processing has completed.

grammars
Grammars loaded into this engine.

language
Current user language of the SR engine. (Read-only)
Return type str

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

name
The human-readable name of this engine.

process_grammars_context(*window=None*)
Enable/disable grammars & rules based on their current contexts.

This must be done preemptively for some SR engine back-ends, such as WSR, that don't apply context changes upon/after the utterance start has been detected. The WSR engine should call this automatically whenever the foreground application (or its title) changes. The user may want to call this manually to update when custom contexts.

The *window* parameter is optional window information, which can be passed in as an optimization if it has already been gathered.

quoted_words_support
Whether this engine can compile and recognize quoted words.
Return type bool

recognise_forever(*begin_callback=None*, *recognition_callback=None*, *failure_callback=None*, *end_callback=None*, *post_recognition_callback=None*, **args*, ***kwargs*)
Alias of *do_recognition()* left in for backwards-compatibility

recognize_forever(*begin_callback=None*, *recognition_callback=None*, *failure_callback=None*, *end_callback=None*, *post_recognition_callback=None*, **args*, ***kwargs*)
Alias of *do_recognition()* left in for backwards-compatibility

set_exclusive(*grammar*, *exclusive*)
Alias of *set_exclusiveness()*.

set_exclusiveness(*grammar*, *exclusive*)
Set the exclusiveness of a grammar.

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

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 object can be expected to behave like a string, responding as you would expect to string methods like `replace()`. The formatted text can be retrieved using *format()* or simply by calling `str(...)` on a dictation container object. By default, formatting returns the words joined with spaces, but custom formatting can be applied by calling string methods on the `Dictation` object. A tuple of the raw spoken words can be retrieved using *words*.

String Formatting Examples

The following examples demonstrate dictation input can be formatted by calling string methods on `Dictation` elements.

Python example:

```
mapping = {
    # Define commands for writing Python methods, functions and classes.
    "method [<under>] <snaketext>":
        Text("def %(under)s%(snaketext)s(self):") + Key("left:2"),
    "function <snaketext>":
        Text("def %(snaketext)s():") + Key("left:2"),
    "classy [<classtext>]":
        Text("class %(classtext)s:") + Key("left"),

    # Define a command for accessing object members.
    "selfie [<under>] [<snaketext>]":
        Text("self.%(under)s%(snaketext)s"),
}

extras = [
    # Define a Dictation element that produces snake case text,
    # e.g. hello_world.
    Dictation("snaketext", default="").lower().replace(" ", "_"),

    # Define a Dictation element that produces text matching Python's
    # class casing, e.g. DictationContainer.
    Dictation("classtext", default="").title().replace(" ", ""),

    # Allow adding underscores before cased text.
    Choice("under", {"under": "_"}, default=""),
]

rule = MappingRule(name="PythonExample", mapping=mapping, extras=extras)
```

Markdown example:

```
mapping = {
    # Define a command for typing Markdown headings 1 to 7 with optional
    # capitalized text.
    "heading [<num>] [<capitalised_text>]":
        Text("#")*Repeat("num") + Text(" %(capitalised_text)s"),
}

extras = [
    Dictation("capitalised_text", default="").capitalize(),
    IntegerRef("num", 1, 7, 1),
]

rule = MappingRule(name="MdExample", mapping=mapping, extras=extras)
```

Camel-case example using the `Dictation.camel()` method:

```
mapping = {
    # Define a command for typing camel-case text, e.g. helloWorld.
    "camel <camel_text>": Text(" %(camel_text)s"),
}
```

(continues on next page)

(continued from previous page)

```
extras = [
    Dictation("camel_text", default="").camel(),
]

rule = MappingRule(name="CamelExample", mapping=mapping, extras=extras)
```

Example using the `Dictation.apply()` method for random casing:

```
from random import random

def random_text(text):
    # Randomize the case of each character.
    result = ""
    for c in text:
        r = random()
        if r < 0.5:
            result += c.lower()
        else:
            result += c.upper()
    return result

mapping = {
    "random <random_text>": Text("%(random_text)s"),
}

extras = [
    Dictation("random_text", default="").apply(random_text),
]

rule = MappingRule(name="RandomExample", mapping=mapping, extras=extras)
```

Class reference

class DictationContainerBase(*words*, *methods=None*)

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.
- **methods** (*list-of-triples*) – Tuples describing string methods to call on the output.

apply_methods(*joined_words*)

Apply any string methods called on the `Dictation` object to a given string.

Called during *format()*.

format()

Format and return this dictation as a Unicode object.

words
Sequence of the words forming this dictation.

Engine timer classes

Multiplexing interface to a timer

class Timer(*function*, *interval*, *manager*, *repeating=True*)
Timer class for calling a function every N seconds.

Constructor arguments:
- *function* (*callable*) – the function to call every N seconds. Must have no required arguments.
- *interval* (*float*) – number of seconds between calls to the function. Note that this is on a best-effort basis only.
- *manager* (*TimerManagerBase*) – engine timer manager instance.
- *repeating* (*bool*) – whether to call the function every N seconds or just once (default: True).

Instances of this class are normally initialised from `engine.create_timer()`.

call()
Call the timer's function.

This method is normally called by the timer manager.

start()
Start calling the timer's function on an interval.

This method is called on initialisation.

stop()
Stop calling the timer's function on an interval.

class TimerManagerBase(*interval*, *engine*)
Base timer manager class.

_activate_main_callback(*callback*, *msec*)
Virtual method to implement to start calling *main_callback()* on an interval.

_deactivate_main_callback()
Virtual method to implement to stop calling *main_callback()* on an interval.

add_timer(*timer*)
Add a timer and activate the main callback if required.

disable()
Method to disable execution of the main timer callback.

This method is used for testing timer-related functionality without race conditions.

enable()
Method to re-enable the main timer callback.

The main timer callback is enabled by default. This method is only useful if *disable()* is called.

main_callback()
Method to call each timer's function when required.

remove_timer(*timer*)
Remove a timer and deactivate the main callback if required.

class ThreadedTimerManager(*interval*, *engine*)
Timer manager class using a daemon thread.

This class is used by the "text" engine. It is only suitable for engine backends with no recognition loop to execute timer functions on.

> **Warning:** The timer interface is **not** thread-safe. Use the `enable()` and `disable()` methods if you need finer control over timer function execution.

class DelegateTimerManager(*interval*, *engine*)
Timer manager class that calls `main_callback()` through an engine-specific callback function.

Engines using this class should implement the methods in `DelegateManagerInterface`.

This class is used by the SAPI 5 engine.

class DelegateTimerManagerInterface
DelegateTimerManager interface.

set_timer_callback(*callback*, *sec*)
Method to set the timer manager's callback.

Parameters

- **callback** (*callable | None*) – function to call every N seconds
- **sec** (*float | int*) – number of seconds between calls to the callback function

Natlink and DNS engine back-end

Dragonfly uses Natlink to communicate with Dragon. The latest versions of Natlink are available from SourceForge. All versions in the 4.X range should work with Dragonfly. Please see the Natlink install instructions on *qh.antenna.nl* for how to install Natlink on your machine and configure it.

Dragonfly and Natlink support Dragon versions up to 15 (latest). The *Individual* editions of *Dragon* are recommended, although other editions such as *Home* will also work.

Python version 2.7 (32-bit) is required to use this engine back-end, at least for the moment. Support for this version is not maintained for the other engine back-ends and will be **dropped completely** in the first *MAJOR* release following stable Natlink support for Python 3.

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 **or** from the optional user directory. Common locations are:

- `C:\NatLink\NatLink\MacroSystem`
- `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.

Engine Configuration

This engine can be configured by passing (optional) keyword arguments to the `get_engine()` function, which passes them to the engine constructor (documented below). For example:

```
engine = get_engine("natlink",
  retain_dir="natlink_recordings",
)
```

The engine can also be configured via the *command-line interface*:

```
# Initialize the Natlink engine back-end with custom arguments, then load
# command modules and recognize speech.
python -m dragonfly load _*.py --engine natlink --engine-options \
    retain_dir="natlink_recordings"
```

Engine API

class NatlinkEngine(*retain_dir=None*)

Speech recognition engine back-end for Natlink and DNS.

Parameters **retain_dir** (*str|None*) – directory to save audio data: A `.wav` file for each utterance, and `retain.tsv` file with each row listing (wav filename, wav length in seconds, grammar name, rule name, recognized text) as tab separated values.

If this parameter is used in a module loaded by `natlinkmain`, then the directory will be relative to the Natlink user directory (e.g. `MacroSystem`).

DictationContainer

alias of *dragonfly.engines.backend_natlink.dictation.NatlinkDictationContainer*

apply_threading_fix()

Start a thread and engine timer internally to allow Python threads to work properly while connected to natlink. The fix is only applied once, successive calls have no effect.

This method is called automatically when *connect()* is called or when a grammar is loaded for the first time.

connect()

Connect to natlink with Python threading support enabled.

disconnect()

Disconnect from natlink.

mimic(*words*)

Mimic a recognition of the given *words*.

Note: This method has a few quirks to be aware of:

1. Mimic is not limited to one element per word as seen with proper nouns from DNS. For example, "Buffalo Bills" can be passed as one word.
2. Mimic can handle by the extra formatting by DNS built-in commands.
3. Mimic is case sensitive.

---

set_exclusiveness(*grammar*, *exclusive*)

Set the exclusiveness of a grammar.

set_retain_directory(*retain_dir*)

Set the directory where audio data is saved.

Retaining audio data may be useful for acoustic model training. This is disabled by default.

If a relative path is used and the code is running via natspeak.exe, then the path will be made relative to the Natlink user directory or base directory (e.g. `MacroSystem`).

Parameters **retain_dir** (*string|None*) – retain directory path

speak(*text*)

Speak the given *text* using text-to-speech.

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*, *methods*)

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

format()

Format and return this dictation.

Multiplexing interface to the Natlink timer

class NatlinkTimerManager(*interval*, *engine*)

Timer manager for the Natlink engine.

This class utilises `natlink.setTimerCallback()` to ensure that timer functions are called on-time regardless of Dragon's current status.

Python code run outside of timer functions will be blocked when `natlink` functions are executing. This is a limitation with Python threads.

`engine.connect()` must be called before using timers with this manager.

Note: long-running timers will block dragonfly from processing what was said, so be careful with how you use them!

SAPI 5 and WSR engine back-end

Dragonfly can use the built-in speech recognition included with Microsoft Windows Vista and above: Windows Speech Recognition (WSR). If WSR is available on the machine, 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/dfly-loader-wsr.py` file. When run, it will scan the directory it's in for files beginning with _ and ending with `.py`, then try to load them as command-modules.

A *more full-featured loader* is available in the `dragonfly/examples/wsr_module_loader_plus.py` file. It includes a basic sleep/wake grammar to control recognition (simply say "start listening" or "halt listening"), along with a rudimentary user interface via sound effects and console text (easily modified in the file). It otherwise operates like the above loader.

You can download Dragonfly's module loaders and other example files in `dragonfly/examples` from the source code repository.

Dragonfly interfaces with this speech recognition engine using Microsoft's Speech API version 5. This is why it is referred to in many places by "SAPI" or "SAPI 5" instead of WSR.

Shared vs in-process recognizers

The WSR / SAPI 5 back-end has two engine classes:

- *sapi5inproc* - engine class for SAPI 5 in process recognizer. This is the default implementation and has no GUI (yet). `get_engine()` will return an instance of this class if the `name` parameter is `None` (default) or `"sapi5inproc"`. It is recommended that you run this from command-line.
- *sapi5shared* - engine class for SAPI 5 shared recognizer. This implementation uses the Windows Speech Recognition GUI. This implementation's behaviour can be inconsistent and a little buggy at times, which is why it is no longer the default. To use it anyway pass `"sapi5"` or `"sapi5shared"` to `get_engine()`.

The engine class can be selected by passing one of the above-mentioned names names via the *command-line interface* instead of using `get_engine()` directly:

```
# Initialize the SAPI 5 engine using the shared recognizer class.
python -m dragonfly load _*.py --engine sapi5shared
```

Engine Configuration

This engine can be configured by passing (optional) keyword arguments to the `get_engine()` function, which passes them to the engine constructor (documented below). For example:

```
engine = get_engine("sapi5inproc",
  retain_dir="C:/sapi5_recordings",
)
```

The engine can also be configured via the *command-line interface*:

```
# Initialize the SAPI 5 engine back-end with custom arguments, then load
# command modules and recognize speech.
python -m dragonfly load _*.py --engine sapi5inproc --engine-options \
    retain_dir="C:/sapi5_recordings"
```

Engine API

class Sapi5SharedEngine(*retain_dir=None*)

Speech recognition engine back-end for SAPI 5 shared recognizer.

Parameters **retain_dir** (*str|None*) – Retains recognized audio and/or metadata in the given directory, saving audio to `retain_[timestamp].wav` file and metadata to `retain.tsv`.

Disabled by default (`None`).

DictationContainer

alias of *dragonfly.engines.base.dictation.DictationContainerBase*

activate_grammar(*grammar*)

Activate the given *grammar*.

activate_rule(*rule*, *grammar*)

Activate the given *rule*.

connect()

Connect to back-end SR engine.

deactivate_grammar(*grammar*)
Deactivate the given *grammar*.

deactivate_rule(*rule*, *grammar*)
Deactivate the given *rule*.

disconnect()
Disconnect from back-end SR engine.

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

Note: This method has a few quirks to be aware of:

1. Mimic can fail to recognize a command if the relevant grammar is not yet active.
2. Mimic does not work reliably with the shared recognizer unless there are one or more exclusive grammars active.
3. Mimic can **crash the process** in some circumstances, e.g. when mimicking non-ASCII characters.

---

set_exclusiveness(*grammar*, *exclusive*)
Set the exclusiveness of a grammar.

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

class Sapi5InProcEngine(*retain_dir=None*)
Speech recognition engine back-end for SAPI 5 in process recognizer.

Parameters **retain_dir** (*str|None*) – Retains recognized audio and/or metadata in the given directory, saving audio to `retain_[timestamp].wav` file and metadata to `retain.tsv`.

Disabled by default (`None`).

connect(*audio_source=0*)
Connect to the speech recognition backend.

The audio source to use for speech recognition can be specified using the *audio_source* argument. If it is not given, it defaults to the first audio source found.

get_audio_sources()
Get the available audio sources.

This method returns a list of audio sources, each represented by a 3-element tuple: the index, the description, and the COM handle for the audio source.

select_audio_source(*audio_source*)
Configure the speech recognition engine to use the given audio source.
The audio source may be specified as follows:

- As an *int* specifying the index of the audio source to use
- As a *str* containing the description of the audio source to use, or a substring thereof

The *get_audio_sources()* method can be used to retrieve the available sources together with their indices and descriptions.

Kaldi engine back-end

This version of dragonfly contains an engine implementation using the free, open source, cross-platform Kaldi speech recognition toolkit. You can read more about the Kaldi project on the Kaldi project site.

This backend relies greatly on the kaldi-active-grammar library, which extends Kaldi's standard decoding for use in a dragonfly-style environment, allowing combining many dynamic grammars that can be set active/inactive based on

contexts in real-time. It also provides basic infrastructure for compiling, recognizing, and parsing grammars with Kaldi, plus a compatible model. For more information, see its page.

Both this backend and kaldi-active-grammar are under **active development** by @daanzu. Kaldi-backend-specific issues, suggestions, and feature requests are welcome & encouraged, but are probably better sent to the kaldi-active-grammar repository. If you value this work and want to encourage development of a free, open source, cross-platform engine for dragonfly as a competitive alternative to commercial offerings, kaldi-active-grammar accepts donations (not affiliated with the dragonfly project itself).

Please note that Kaldi Active Grammar is licensed under the GNU Affero General Public License v3 (AGPL-3.0-or-later). This has an effect on what Dragonfly can be used to do when used in conjunction with the Kaldi engine back-end.

Sections:

Setup

Want to get started **quickly & easily on Windows**? A self-contained, portable, batteries-included (python & libraries & model) distribution of kaldi-active-grammar + dragonfly2 is available at the kaldi-active-grammar project releases page. Otherwise...

Requirements:

- Python 3.6+; *64-bit required!*
- OS: Windows/Linux/MacOS all supported (see *Cross-platform*)
- Only supports Kaldi left-biphone models, specifically *nnet3 chain* models, with specific modifications
- ~1GB+ disk space for model plus temporary storage and cache, depending on your grammar complexity
- ~500MB+ RAM for model and grammars, depending on your model and grammar complexity
- Python package dependencies (which should be installed automatically by following the instructions below): * kaldi-active-grammar * sounddevice * webrtcvad

Note for Linux: You may need the `portaudio` headers to be installed in order to be able to install/compile the `sounddevice` Python package. Under `apt`-based distributions, you can get them by running `sudo apt install portaudio19-dev`. You may also need to make your user account a member of the `audio` group to be able to access your microphone. Do this by running `usermod -a -G audio account_name_here`.

Installing the correct versions of the Python dependencies can be most easily done by installing the `kaldi` sub-package of `dragonfly2` using:

```
pip install 'dragonfly2[kaldi]'
```

If you are installing to *develop* dragonfly2, use the following instead (from your dragonfly2 git repository):

```
pip install -e '.[kaldi]'
```

Note: If you have errors installing the kaldi-active-grammar package, make sure you're using a 64-bit Python, and update your `pip` by executing `pip install --upgrade pip`.

You will also need a **model** to use. You can download a compatible general English Kaldi nnet3 chain model from kaldi-active-grammar. Unzip it into a directory within the directory containing your grammar modules.

Note for Linux: Before proceeding, you'll need to install the `wmctrl`, `xdotool` and `xsel` programs. Under `apt`-based distributions, you can get them by running:

```
sudo apt install wmctrl xdotool xsel
```

You may also need to manually set the `DISPLAY` environment variable.

Once the dependencies and model are installed, you're ready to go!

Getting Started

A simple, single-file, standalone demo/example can be found in the dragonfly/examples/kaldi_demo.py script. Simply run it from the directory containing the above model (or modify the configuration paths in the file) using:

```
python path/to/kaldi_demo.py
```

For more structured and long-term use, you'll want to use a **module loader**. Copy the dragonfly/examples/kaldi_module_loader_plus.py script into the folder with your grammar modules and run it using:

```
python kaldi_module_loader_plus.py
```

This file is the equivalent to the 'core' directory that NatLink uses to load grammar modules. When run, it will scan the directory it's in for files beginning with _ and ending with `.py`, then try to load them as command-modules.

This file also includes a basic sleep/wake grammar to control recognition (simply say "start listening" or "halt listening").

A more basic loader is in dragonfly/examples/kaldi_module_loader.py.

Updating To A New Version

When updating to a new version of dragonfly, you should always rerun `pip install 'dragonfly2[kaldi]'` (or `pip install '.[kaldi]'`, etc.) to make sure you get the required version of kaldi_active_grammar.

Engine Configuration

This engine can be configured by passing (optional) keyword arguments to the `get_engine()` function, which passes them to the `KaldiEngine` constructor (documented below). For example:

```
engine = get_engine("kaldi",
  model_dir='kaldi_model',
  tmp_dir=None,
  audio_input_device=None,
  audio_self_threaded=True,
  audio_auto_reconnect=True,
  audio_reconnect_callback=None,
```

(continues on next page)

(continued from previous page)

```
  retain_dir=None,
  retain_audio=None,
  retain_metadata=None,
  retain_approval_func=None,
  vad_aggressiveness=3,
  vad_padding_start_ms=150,
  vad_padding_end_ms=200,
  vad_complex_padding_end_ms=600,
  auto_add_to_user_lexicon=True,
  allow_online_pronunciations=False,
  lazy_compilation=True,
  invalidate_cache=False,
  expected_error_rate_threshold=None,
  alternative_dictation=None,
)
```

The engine can also be configured via the *command-line interface*:

```
# Initialize the Kaldi engine backend with custom arguments, then load
# command modules and recognize speech.
python -m dragonfly load _*.py --engine kaldi --engine-options " \
    model_dir=kaldi_model_daanzu \
    vad_padding_end_ms=300"
```

KaldiEngine(*model_dir=None*, *tmp_dir=None*, *input_device_index=None*, *audio_input_device=None*, *audio_self_threaded=True*, *audio_auto_reconnect=True*, *audio_reconnect_callback=None*, *retain_dir=None*, *retain_audio=None*, *retain_metadata=None*, *retain_approval_func=None*, *vad_aggressiveness=3*, *vad_padding_start_ms=150*, *vad_padding_end_ms=200*, *vad_complex_padding_end_ms=600*, *auto_add_to_user_lexicon=True*, *allow_online_pronunciations=False*, *lazy_compilation=True*, *invalidate_cache=False*, *expected_error_rate_threshold=None*, *alternative_dictation=None*, *compiler_init_config=None*, *decoder_init_config=None*)

Speech recognition engine back-end for Kaldi recognizer.

Arguments (all optional):

- `model_dir` (`str|None`) – Directory containing model.
- `tmp_dir` (`str|None`) – Directory to use for temporary storage and cache (used for caching during and between executions but safe to delete).
- `audio_input_device` (`int|str|None|False`) – Microphone PortAudio input device: the default of `None` chooses the default input device, or `False` disables microphone input. To see a list of available input devices and their corresponding indexes an names, call `get_engine('kaldi').print_mic_list()`. To select a specific device, pass an `int` representing the index number of the device, or pass a `str` representing (part of) the name of the device. If a string is given, the device is selected which contains all space-separated parts in the right order. Each device string contains the name of the corresponding host API in the end. The string comparison is case-insensitive. The string match must be unique.
- `audio_auto_reconnect` (`bool`) – Whether to automatically reconnect the audio device if it appears to have stopped (by not returning any audio data for some period of time).
- `audio_reconnect_callback` (`callable|None`) – Callable to be called every time the audio system attempts to reconnect (automatically or manually). It must take exactly one positional argument, which is the `MicAudio` object.
- `retain_dir` (`str|None`) – Retains recognized audio and/or metadata in the given directory, saving audio to `retain_[timestamp].wav` file and metadata to `retain.tsv`. What is automatically retained it is

determined by `retain_audio` and `retain_metadata`. If both are `False` but this is set, you can actively choose to retain a given recognition. See below for more information.

- `retain_audio` (`bool|None`) – Whether to retain audio data for all recognitions. If `True`, then requires `retain_dir` to be set. If `None`, then defaults to `True` if `retain_dir` is set to `True`. See below for more information.
- `retain_metadata` (`bool|None`) – Whether to retain metadata for all recognitions. If `True`, then requires `retain_dir` to be set. If `None`, then defaults to `True` if `retain_dir` is set to `True`. See below for more information.
- `retain_approval_func` (`Callable`) – If retaining is enabled, this is called upon each recognition, to determine whether or not to retain it. It must accept as a parameter the *`dragonfly.engines.backend_kaldi.audio.AudioStoreEntry`* under consideration, and return a `bool` (`True` to retain). This is useful for ignoring recognitions that tend to be noise, perhaps contain sensitive content, etc.
- `vad_aggressiveness` (`int`) – Aggressiveness of the Voice Activity Detector: an integer between `0` and `3`, where `0` is the least aggressive about filtering out non-speech, and `3` is the most aggressive.
- `vad_padding_start_ms` (`int`) – Approximate length of padding/debouncing (in milliseconds) at beginning of each utterance for the Voice Activity Detector. Smaller values result in lower latency recognition (faster reactions), but possibly higher likelihood of false positives at beginning of utterances, and more importantly higher possibility of not capturing the entire beginning of utterances.
- `vad_padding_end_ms` (`int`) – Approximate length of silence (in milliseconds) at ending of each utterance for the Voice Activity Detector. Smaller values result in lower latency recognition (faster reactions), but possibly higher likelihood of false negatives at ending of utterances.
- `vad_complex_padding_end_ms` (`int|None`) – If not None, the Voice Activity Detector behaves differently for utterances that are complex (usually meaning inside dictation), using this value instead of `vad_padding_end_ms`, so you can attain longer utterances to take advantage of context to improve recognition quality.
- `auto_add_to_user_lexicon` (`bool`) – Enables automatically adding unknown words to the *User Lexicon*. This will only work if you have additional required packages installed. This will only work locally, unless you also enable `allow_online_pronunciations`.
- `allow_online_pronunciations` (`bool`) – Enables online pronunciation generation for unknown words, if you have also enabled `auto_add_to_user_lexicon`, and you have the required packages installed.
- `lazy_compilation` (`bool`) – Enables deferred grammar/rule compilation, which then allows parallel compilation up to your number of cores, for a large speed up loading uncached.
- `invalidate_cache` (`bool`) – Enables invalidating the engine's cache prior to initialization, possibly for debugging.
- `expected_error_rate_threshold` (`float|None`) – Threshold of "confidence" in the recognition, as measured in estimated error rate (between 0 and ~1 where 0 is perfect), above which the recognition is ignored. Setting this may be helpful for ignoring "bad" recognitions, possibly around `0.1` depending on personal preference.
- `alternative_dictation` (`callable|None`) – Enables alternative an dictation model/engine and chooses the provider. Possible values:
  - `None` – Disabled
  - a Python `callable` – See *Alternative Dictation* section below

Cross-platform

Although Kaldi & this dragonfly engine implementation can run on multiple platforms, including on architectures other than x86, not all other dragonfly components are currently fully cross-platform. This is an area ongoing work.

User Lexicon

Kaldi uses pronunciation dictionaries to lookup phonetic representations for words in grammars & language models in order to recognize them. The default model comes with a large dictionary, but obviously cannot include all possible words. There are multiple ways of handling this.

Ignoring unknown words: If you use words in your grammars that are *not* in the dictionary, a message similar to the following will be printed:

```
Word not in lexicon (will not be recognized): 'notaword'
```

These messages are only warnings, and the engine will continue to load your grammars and run. However, the unknown words will effectively be impossible to be recognized, so the rules using them will not function as intended. To fix this, try changing the words in your grammars by splitting up the words or using to similar words, e.g. changing "natlink" to "nat link".

Automatically adding words to User Lexicon: Set the engine parameter `auto_add_to_user_lexicon=True` to enable. If an unknown word is encountered while loading a grammar, its pronunciation is predicted based on its spelling. This uses either a local library, or a free cloud service if the library is not installed.

The local library (*g2p_en*) can be installed by running the following on the command line:

```
pip install g2p_en==2.0.0
```

Note that the dependencies for this library can be difficult to install, in which case it is recommended to use the cloud service instead. Set the engine parameter `allow_online_pronunciations=True` to enable it.

Manually editing User Lexicon: You can add a word without specifying a pronunciation, and let it be predicted as above, by running at the command line:

```
python -m kaldi_active_grammar add_word cromulent
```

Or you can add a word with a specified pronunciation:

```
python -m kaldi_active_grammar add_word cromulent "K R OW M Y UW L AH N T"
```

You can also directly edit your `user_lexicon.txt` file, which is located in the model directory. You may add words (with pronunciation!) or modify or remove words that you have already added. The format is simple and whitespace-based:

```
cromulent k r A m j V l V n t
embiggen I m b I g V n
```

Note on Phones: Currently, adding words only accepts pronunciations using the "CMU"/"ARPABET" phone set (with or without stress), but the model and `user_lexicon.txt` file store pronunciations using "X-SAMPA" phone set.

When hand-crafting pronunciations, you can look online for examples. Also, for X-SAMPA pronunciations, you can look in the model's `lexicon.txt` file, which lists all of its words and their pronunciations (in X-SAMPA). Look for words with similar sounds to what you are speaking.

To empty your user lexicon, you can simply delete `user_lexicon.txt`, or run:

```
python -m kaldi_active_grammar reset_user_lexicon
```

Preserving Your User Lexicon: When changing models, you can (and probably should) copy your `user_lexicon.txt` file from your old model directory to the new one. This will let you keep your additions.

Also, if there is a `user_lexicon.txt` file in the current working directory of your initial loader script, its contents will be automatically added to the `user_lexicon.txt` in the active model when it is loaded.

User Lexicon and Dictation

New pronunciations for existing words that are already in the dictation language model will not be recognized during dictation elements specifically until the dictation model is recompiled. Recompilation is quite time consuming (on the order of 15 minutes), but can be performed by running:

```
python -m kaldi_active_grammar compile_dictation_graph -m kaldi_model
```

Entirely new words added to the user lexicon will not be recognized during dictation elements specifically at all currently.

However, you can achieve a similar result for both of these weaknesses with the following: create a rule that recognizes a repetition of alternates between normal dictation and a special rule that recognizes all of your special terminology. An example of this can be seen in this dictation grammar. This technique can also help mitigate dictation **recognizing the wrong of similar sounding words** by emphasizing the word you want to be recognized, possibly with the addition of a weight parameter.

Experimental: You can avoid the above issues by using this engine's "user dictation" feature. This also allows you to have separate "spoken" and "written forms" of terms in dictation. Do so by adding any words you want added/modified to the user dictation list (identical spoken and written form) or dictlist (different spoken and written forms), and using the `UserDictation` element in your grammars (in place of the standard dragonfly `Dictation` element):

```
from dragonfly import get_engine, MappingRule, Function
from dragonfly.engines.backend_kaldi.dictation import UserDictation as Dictation
get_engine().add_word_list_to_user_dictation(['kaldi'])
get_engine().add_word_dict_to_user_dictation({'open F S T': 'openFST'})
class TestUserDictationRule(MappingRule):
    mapping = { "dictate <text>": Function(lambda text: print("text: %s" % text)), }
    extras = [ Dictation("text"), ]
```

Grammar/Rule/Element Weights

Grammars, rules, and/or elements can have a weight specified, where those with higher weight value are more likely to be recognized, compared to their peers, for an ambiguous recognition. This can be used to adjust the probability of them be recognized.

The default weight value for everything is `1.0`. The exact meaning of the weight number is somewhat inscrutable, but you can treat larger values as more likely to be recognized, and smaller values as less likely. **Note:** you may need to use much larger or smaller numbers than you might expect to achieve your desired results, possibly orders of magnitude (base 10).

An example:

```
class WeightExample1Rule(MappingRule):
    mapping = { "kiss this guy": ActionBase() }
class WeightExample2Rule(MappingRule):
```

(continues on next page)

(continued from previous page)

```
    mapping = { "kiss the sky": ActionBase() }
    weight = 2
class WeightExample3Rule(MappingRule):
    mapping = {
      "start listening {weight=0.01}": ActionBase(),  # Be less eager to wake up!
      "halt listening": ActionBase(),
      "go (north | nowhere {w=0.01} | south)": ActionBase(),
    }
```

The weight of a grammar is effectively propagated equally to its child rules, on top of their own weights. Similarly for rules propagating weights to child elements.

Retaining Audio and/or Recognition Metadata

You can optionally enable retention of the audio and metadata about the recognition, using the `retain_dir` engine parameter.

Note: This feature is completely optional and disabled by default!

The metadata is saved TSV format, with fields in the following order:

- `audio_data`: file name of the audio file for the recognition
- `grammar_name`: name of the recognized grammar
- `rule_name`: name of the recognized rule
- `text`: text of the recognition
- `likelihood`: the engine's estimated confidence of the recognition (not very reliable)
- `tag`: a single text tag, described below
- `has_dictation`: whether the recognition contained (in part) a dictation element

Tag: You can mark the previous recognition with a single text tag to be stored in the metadata. For example, mark it as incorrect with a rule containing:

```
"action whoops": Function(lambda: engines.get_engine().audio_store[0].set('tag',
↪'misrecognition'))
```

Or, you can mark it specifically to be saved, even if `retain_audio` is `False` and recognitions are not normally saved, as long as `retain_dir` is set. This also demonstrates that `.set()` can be chained to tag it at the same time:

```
"action corrected": Function(lambda: engines.get_engine().audio_store[0].set('tag',
↪'corrected').set('force_save', True))
```

This is useful for retaining only known-correct data for later training.

Alternative Dictation

This backend supports optionally using an alternative method of recognizing (some or all) dictation, rather than the default Kaldi model, which is always used for command recognition. You may want to do this for higher dictation accuracy (at the possible cost of higher latency or what would otherwise cause lower command accuracy), dictating in another language, or some other reason. You can use one of:

- an alternative Kaldi model

- an alternative local speech recognition engine
- a cloud speech recognition engine

Note: This feature is completely optional and disabled by default!

You can enable this by setting the `alternative_dictation` engine option. Valid options:

- A `callable` object: Any external engine. The callable must accept at least one argument (for the audio data) and any keyword arguments. The audio data is passed in standard Linear16 (`int`) PCM encoding. The callable should return the recognized text.

Using Alternative Dictation

To use alternative dictation, you *must both* pass the `alternative_dictation` option *and* use a specialized `Dictation` element. The standard dragonfly `Dictation` does not support alternative dictation. Instead, this backend provides two subclasses of it: `AlternativeDictation` and `DefaultDictation`. These two subclasses both support alternative dictation; they differ only in whether they do alternative dictation by default.

`AlternativeDictation` and `DefaultDictation` can be used as follows. Assume we are defining a variable `element` that is used by the code:

```
class TestDictationRule(MappingRule):
  mapping = { "dictate <text>": Text("%(text)s") }
  extras = [ element ]
```

Examples:

```
element = AlternativeDictation("text")                    # alternative dictation
element = DefaultDictation("text")                        # no alternative dictation
element = AlternativeDictation("text", alternative=False) # no alternative dictation
element = DefaultDictation("text", alternative=True)      # alternative dictation

# all AlternativeDictation instances instantiated after this (in any file!) will␣
↪default to alternative=False
AlternativeDictation.alternative_default = False
element = AlternativeDictation("text")                    # no alternative dictation
element = AlternativeDictation("text", alternative=True)  # alternative dictation

# all DefaultDictation instances instantiated after this (in any file!) will default␣
↪to alternative=True
DefaultDictation.alternative_default = True
element = DefaultDictation("text")                        # alternative dictation
element = DefaultDictation("text", alternative=False)     # no alternative dictation

AlternativeDictation.alternative_default = True
DefaultDictation.alternative_default = False
# all AlternativeDictation and DefaultDictation instances instantiated after this are␣
↪back to normal
```

If you want to replace all uses of standard `Dictation` in a file:

```
from dragonfly.engines.backend_kaldi.dictation import AlternativeDictation as␣
↪Dictation
# OR
from dragonfly.engines.backend_kaldi.dictation import DefaultDictation as Dictation
```

Limitations & Future Work

Please let me know if anything is a significant problem for you.

Known Issues

- Entirely new words added to the user lexicon will not be recognized during dictation elements specifically at all currently. You can get around this by constructing a rule that alternates between a dictation element and a mapping rule containing your new words, as demonstrated here.
- Dragonfly `Lists` and `DictLists` function as normal. Upon updating a dragonfly list or dictionary, the rules they are part of will be recompiled & reloaded. This will add some delay, which I hope to optimize.

Dictation Formatting & Punctuation

The native dictation only provides recognitions as unformatted lowercase text without punctuation. Improving this generally is multifaceted and complex. However, the *alternative dictation* feature can avoid this problem by using the formatting & punctuation applied by a cloud provider.

Models: Other Languages, Other Sizes, & Training

The `kaldi-active-grammar` library currently only supplies a single general English model. Many standard Kaldi models (of varying quality) are available online for various languages. Although such standard Kaldi models must be first modified to work with this framework, the process is not difficult and could be automated (future work).

There are also various sizes of Kaldi model, with a trade-off between size/speed and accuracy. Generally, the smaller and faster the model, the lower the accuracy. The included model is relatively large. Let me know if you need a smaller one.

Training (personalizing) Kaldi models is possible but complicated. In addition to requiring many steps using a specialized software environment, training these models currently requires using a GPU for an extended period. This may be a case where providing a service for training is more feasible.

Text-to-speech

This isn't a limitation of Kaldi; text-to-speech is not a project goal for them, although as the natlink and WSR engines both support text-to-speech, there might as well be some suggestions if this functionality is desired, perhaps utilized by a custom dragonfly action. The Jasper project contains a number of Python interface classes to popular open source text-to-speech software such as *eSpeak*, *Festival* and *CMU Flite*.

Engine API

class KaldiEngine(*model_dir=None*, *tmp_dir=None*, *input_device_index=None*, *audio_input_device=None*, *audio_self_threaded=True*, *audio_auto_reconnect=True*, *audio_reconnect_callback=None*, *retain_dir=None*, *retain_audio=None*, *retain_metadata=None*, *retain_approval_func=None*, *vad_aggressiveness=3*, *vad_padding_start_ms=150*, *vad_padding_end_ms=200*, *vad_complex_padding_end_ms=600*, *auto_add_to_user_lexicon=True*, *allow_online_pronunciations=False*, *lazy_compilation=True*, *invalidate_cache=False*, *expected_error_rate_threshold=None*, *alternative_dictation=None*, *compiler_init_config=None*, *decoder_init_config=None*)

Speech recognition engine back-end for Kaldi recognizer.

DictationContainer
alias of *dragonfly.engines.base.dictation.DictationContainerBase*

activate_grammar(*grammar*)
Activate the given *grammar*.

activate_rule(*rule*, *grammar*)
Activate the given *rule*.

add_word_dict_to_user_dictation(*word_dict*)
Make UserDictation elements able to recognize each item of given dict of strings *word_dict*. The key is the "spoken form" (which is recognized), and the value is the "written form" (which is returned as the text in the UserDictation element). Note: all characters in the keys will be converted to lowercase, but the values are returned as text verbatim.

add_word_list_to_user_dictation(*word_list*)
Make UserDictation elements able to recognize each item of given list of strings *word_list*. Note: all characters will be converted to lowercase, and recognized as such.

connect()
Connect to back-end SR engine.

deactivate_grammar(*grammar*)
Deactivate the given *grammar*.

deactivate_rule(*rule*, *grammar*)
Deactivate the given *rule*.

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

ignore_current_phrase()
Marks the current phrase's recognition to be ignored when it completes, or does nothing if there is none. Returns *bool* indicating whether or not there was a current phrase being heard.

in_phrase
Whether or not the engine is currently in the middle of hearing a phrase from the user.

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

prepare_for_recognition()
Can be called optionally before `do_recognition()` to speed up its starting of active recognition.

recognize_wave_file(*filename*, *realtime=False*, ***kwargs*)
Does recognition on given wave file, treating it as a single utterance (without VAD), then returns.

recognize_wave_file_as_stream(*filename*, *realtime=False*, ***kwargs*)
Does recognition on given wave file, treating it as a stream and processing it with VAD to break it into multiple utterances (as with normal microphone audio input), then returns.

saving_adaptation_state
Whether or not the engine is currently automatically saving adaptation state between utterances.

set_exclusiveness(*grammar*, *exclusive*)
Set the exclusiveness of a grammar.

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

start_saving_adaptation_state()
Enable automatic saving of adaptation state between utterances, which may improve recognition accuracy in the short term, but is not stored between runs.

stop_saving_adaptation_state()
Disables automatic saving of adaptation state between utterances, which you might want to do when you expect there to be noise and don't want it to pollute your current adaptation state.

class UserDictation(*name=None*, *format=True*, *default=None*)
Imitates the standard Dictation element class, using individual chunks of Dictation or the user's added terminology.

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()*).

class AlternativeDictation(**args*, ***kwargs*)

class DefaultDictation(**args*, ***kwargs*)

Kaldi Recognition Results Class

class Recognition(*engine*, *kaldi_rule*, *words*, *words_are_dictation_mask=None*)
Kaldi recognition results class.

Kaldi Audio

class AudioStoreEntry(*audio_data*, *grammar_name*, *rule_name*, *text*, *likelihood*, *tag*, *has_dictation*, *force_save=False*)

set(*key*, *value*)
Sets given key (as *str*) to value, returning the AudioStoreEntry for chaining; usable in lambda functions.

CMU Pocket Sphinx engine back-end

This version of dragonfly contains an engine implementation using the open source, cross-platform CMU Pocket Sphinx speech recognition engine. You can read more about the CMU Sphinx speech recognition projects on the CMU Sphinx wiki.

Setup

There are three Pocket Sphinx engine dependencies:

- pyaudio
- pyjsgf
- sphinxwrapper

You can install these by running the following command:

```
pip install 'dragonfly2[sphinx]'
```

If you are installing to *develop* dragonfly, use the following instead:

```
pip install -e '.[sphinx]'
```

Once the dependencies are installed, you'll need to copy the dragonfly/examples/sphinx_module_loader.py script into the folder with your grammar modules and run it using:

```
python sphinx_module_loader.py
```

This file is the equivalent to the 'core' directory that NatLink uses to load grammar modules. When run, it will scan the directory it's in for files beginning with _ and ending with `.py`, then try to load them as command-modules.

Cross-platform Engine

Pocket Sphinx runs on most platforms, including on architectures other than x86, so it only makes sense that the Pocket Sphinx dragonfly engine implementation should work on non-Windows platforms like macOS as well as on Linux distributions. To this effect, I've made an effort to mock Windows-only functionality for non-Windows platforms for the time being to allow the engine components to work correctly regardless of the platform.

Using dragonfly with a non-Windows operating system can already be done with Aenea using the existing *NatLink* engine. Aenea communicates with a separate Windows system running *NatLink* and *DNS* over a network connection and has server support for Linux (using X11), macOS, and Windows.

Engine configuration

This engine can be configured by changing the engine configuration.

You can make changes to the `engine.config` object directly in your *sphinx_engine_loader.py* file before `connect()` is called or create a *config.py* module in the same directory using .

The `LANGUAGE` option specifies the engine's user language. This is English (`"en"`) by default.

Audio configuration

Audio configuration options used to record from the microphone, validate input wave files in and write wave files if the training data directory is set.

These options must match the requirements for the acoustic model being used. The default values match the requirements for the 16kHz CMU US English models.

- `CHANNELS` – number of audio input channels (default: `1`).

- `SAMPLE_WIDTH` – sample width for audio input in bytes (default: `2`).
- `RATE` – sample rate for audio input in Hz (default: `16000`).
- `FRAMES_PER_BUFFER` – frames per recorded audio buffer (default: `2048`).

Keyphrase configuration

The following configuration options control the engine's built-in keyphrases:

- `WAKE_PHRASE` – the keyphrase to listen for when in sleep mode (default: `"wake up"`).
- `WAKE_PHRASE_THRESHOLD` – threshold value* for the wake keyphrase (default: `1e-20`).
- `SLEEP PHRASE` – the keyphrase to listen for to enter sleep mode (default: `"go to sleep"`).
- `SLEEP_PHRASE_THRESHOLD` – threshold value* for the sleep keyphrase (default: `1e-40`).
- `START_ASLEEP` – boolean value for whether the engine should start in a sleep state (default: `True`).
- `START_TRAINING_PHRASE` – keyphrase to listen for to start a training session where no processing occurs (default: `"start training session"`).
- `START_TRAINING_PHRASE_THRESHOLD` – threshold value* for the start training keyphrase (default: `1e-48`).
- `END_TRAINING_PHRASE` – keyphrase to listen for to end a training session if one is in progress (default: `"end training session"`).
- `END_TRAINING_PHRASE_THRESHOLD` – threshold value* for the end training keyphrase (default: `1e-45`).

* Threshold values need to be set for each keyphrase. The CMU Sphinx LM tutorial has some advice on keyphrase threshold values.

If your language isn't set to English, all built-in keyphrases will be disabled by default if they are not specified in your configuration.

Any keyphrase can be disabled by setting the phrase and threshold values to `""` and `0` respectively.

Decoder configuration

The `DECODER_CONFIG` object initialised in the engine config module can be used to set various Pocket Sphinx decoder options.

The following is the default decoder configuration:

```
import os

from sphinxwrapper import DefaultConfig

# Configuration for the Pocket Sphinx decoder.
DECODER_CONFIG = DefaultConfig()

# Silence the decoder output by default.
DECODER_CONFIG.set_string("-logfn", os.devnull)

# Set voice activity detection configuration options for the decoder.
# You may wish to experiment with these if noise in the background
# triggers speech start and/or false recognitions (e.g. of short words)
# frequently.
```

(continues on next page)

(continued from previous page)

```
# Descriptions for VAD configuration options were retrieved from:
# https://cmusphinx.github.io/doc/sphinxbase/fe_8h_source.html

# Number of silence frames to keep after from speech to silence.
DECODER_CONFIG.set_int("-vad_postspeech", 30)

# Number of speech frames to keep before silence to speech.
DECODER_CONFIG.set_int("-vad_prespeech", 20)

# Number of speech frames to trigger vad from silence to speech.
DECODER_CONFIG.set_int("-vad_startspeech", 10)

# Threshold for decision between noise and silence frames.
# Log-ratio between signal level and noise level.
DECODER_CONFIG.set_float("-vad_threshold", 3.0)
```

There does not appear to be much documentation on these options outside of the pocketsphinx/cmdln_macro.h and sphinxbase/fe.h header files. If this is incorrect or has changed, feel free to suggest an edit.

The easiest way of seeing the available decoder options as well as their default values is to run the `pocketsphinx_continuous` command with no arguments.

Changing Models and Dictionaries

The `DECODER_CONFIG` object can be used to configure the pronunciation dictionary as well as the acoustic and language models. You can do this with something like:

```
DECODER_CONFIG.set_string('-hmm', '/path/to/acoustic-model-folder')
DECODER_CONFIG.set_string('-lm', '/path/to/lm-file.lm')
DECODER_CONFIG.set_string('-dict', '/path/to/dictionary-file.dict')
```

The language model, acoustic model and pronunciation dictionary should all use the same language or language variant. See the CMU Sphinx wiki for a more detailed explanation of these components.

Training configuration

The engine can save *.wav* and *.txt* training files into a directory for later use. The following are the configuration options associated with this functionality:

- `TRAINING_DATA_DIR` – directory to save training files into (default: `""`).
- `TRANSCRIPT_NAME` – common name of files saved into the training data directory (default: `"training"`).

Set `TRAINING_DATA_DIR` to a valid directory path to enable recording of *.txt* and *.wav* files. If the path is a relative path, it will be interpreted as relative to the module loader's directory.

The engine will **not** attempt to make the directory for you as it did in previous versions of *dragonfly*.

Engine API

class SphinxEngine

Speech recognition engine back-end for CMU Pocket Sphinx.

DictationContainer

alias of *dragonfly.engines.base.dictation.DictationContainerBase*

cancel_recognition()
If a recognition was in progress, cancel it before processing the next audio buffer.

check_valid_word(*word*)
Check if a word is in the current Sphinx pronunciation dictionary.
Return type bool

config
Python module/object containing engine configuration.

You will need to restart the engine with *disconnect()* and *connect()* if the configuration has been changed after *connect()* has been called.
Returns config module/object

connect()
Set up the CMU Pocket Sphinx decoder.

This method does nothing if the engine is already connected.

create_timer(*callback*, *interval*, *repeating=True*)
Create and return a timer using the specified callback and repeat interval.

Note: Timers will not run unless the engine is recognising audio. Normal threads can be used instead with no downsides.

default_search_result
The last hypothesis object of the default search.

This does not currently reach recognition observers because it is intended to be used for dictation results, which are currently disabled. Nevertheless this object can be useful sometimes.
Returns Sphinx Hypothesis object | None

disconnect()
Deallocate the CMU Sphinx decoder and any other resources used by it.

This method effectively unloads all loaded grammars and key phrases.

end_training_session()
End the training if one is in progress. This will allow recognition processing once again.

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

mimic_phrases(**phrases*)
Mimic a recognition of the given *phrases*.

This method accepts variable phrases instead of a list of words.

pause_recognition()
Pause recognition and wait for *resume_recognition()* to be called or for the wake keyphrase to be spoken.

process_buffer(*buf*)
Recognise speech from an audio buffer.

This method is meant to be called in sequence for multiple audio buffers. It will do nothing if *connect()* hasn't been called.
Parameters **buf** (*str*) – audio buffer

process_wave_file(*path*)
Recognise speech from a wave file and return the recognition results.

This method checks that the wave file is valid. It raises an error if the file doesn't exist, if it can't be read or if the WAV header values do not match those in the engine configuration.

If recognition is paused (sleep mode), this method will call *resume_recognition()*.

The wave file must use the same sample width, sample rate and number of channels that the acoustic model uses.

If the file is valid, *process_buffer()* is then used to process the audio.

Multiple utterances are supported.

Parameters **path** – wave file path
Raises IOError | OSError | ValueError
Returns recognition results
Return type generator

recognising
Whether the engine is currently recognising speech.

To stop recognition, use *disconnect()*.

Return type bool

recognition_paused
Whether the engine is waiting for the wake phrase to be heard or for *resume_recognition()* to be called.

Return type bool

resume_recognition(*notify=True*)
Resume listening for grammar rules and key phrases.

set_exclusiveness(*grammar*, *exclusive*)
Set the exclusiveness of a grammar.

set_keyphrase(*keyphrase*, *threshold*, *func*)
Add a keyphrase to listen for.

Key phrases take precedence over grammars as they are processed first. They cannot be set for specific contexts (yet).

Parameters

- **keyphrase** (*str*) – keyphrase to add.
- **threshold** (*float*) – keyphrase threshold value to use.
- **func** (*callable*) – function or method to call when the keyphrase is heard.

Raises UnknownWordError

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

start_training_session()
Start the training session. This will stop recognition processing until either *end_training_session()* is called or the end training keyphrase is heard.

training_session_active
Whether a training session is in progress.

Return type bool

unset_keyphrase(*keyphrase*)
Remove a set keyphrase so that the engine no longer listens for it.

Parameters **keyphrase** (*str*) – keyphrase to remove.

write_transcript_files(*fileids_path*, *transcription_path*)
Write .fileids and .transcription files for files in the training data directory and write them to the specified file paths.

This method will raise an error if the `TRAINING_DATA_DIR` configuration option is not set to an existing directory.

Parameters

- **fileids_path** (*str*) – path to .fileids file to create.
- **transcription_path** (*str*) – path to .transcription file to create.

Raises IOError | OSError

Multiplexing interface for the CMU Pocket Sphinx engine

class SphinxTimerManager(*interval*, *engine*)

Timer manager for the CMU Pocket Sphinx engine.

This class allows running timer functions if the engine is currently processing audio via one of three engine processing methods:

- `process_buffer()`
- `process_wave_file()`
- `recognise_forever()`

Timer functions will run whether or not recognition is paused (i.e. in sleep mode).

Note: long-running timers will block dragonfly from processing what was said, so be careful with how you use them! Audio frames will not normally be dropped because of timers, long-running or otherwise.

Normal threads can be used instead of timers if desirable. This is because the main recognition loop is done in Python rather than in C/C++ code, so there are no unusual multi-threading limitations.

Improving Speech Recognition Accuracy

CMU Pocket Sphinx can have some trouble recognising what was said accurately. To remedy this, you may need to adapt the acoustic model that Pocket Sphinx is using. This is similar to how Dragon sometimes requires training. The CMU Sphinx adaption tutorial covers this topic. There is also a YouTube video on model adaption.

Adapting your model may not be necessary; there might be other issues with your setup. There is more information on tuning the recognition accuracy in the CMU Sphinx tuning tutorial.

The engine can record what you say into *.wav* and *.txt* files if the `TRAINING_DATA_DIR` configuration option mentioned above is set to an existing directory. To get files compatible with the Sphinx accoustic model adaption process, you can use the `write_transcript_files()` engine method.

Mismatched words may use the engine decoder's default search, typically a language model search.

There are built-in key phrases for starting and ending training sessions where no grammar rule processing will occur. Key phrases will still be processed. See the `START_TRAINING_PHRASE` and `END_TRAINING_PHRASE` engine configuration options. One use case for the training mode is training potentially destructive commands or commands that take a long time to execute their actions.

To use the training files, you will need to correct any incorrect phrases in the *.transcription* or *.txt* files. You can then use the SphinxTrainingHelper bash script to adapt your model. This script makes the process considerably easier, although you may still encounter problems. You should be able to play the wave files using most media players (e.g. VLC, Windows Media Player, aplay) if you need to.

You will want to remove the training files after a successful adaption. This must be done manually for the moment.

Limitations

This engine has a few limitations, most notably with spoken language support and dragonfly's `Dictation` functionality.

Dictation

Mixing free-form dictation with grammar rules is difficult with the CMU Sphinx decoders. It is either dictation or grammar rules, not both. For this reason, Dragonfly's CMU Pocket Sphinx SR engine supports speaking free-form dictation, but only on its own.

Parts of rules that have required combinations with `Dictation` and other basic Dragonfly elements such as `Literal`, `RuleRef` and `ListRef` will not be recognised properly using this SR engine via speaking. They can, however, be recognised via the `engine.mimic()` method, the `Mimic` action or the `Playback` action.

Note: This engine's previous `Dictation` element support using utterance breaks has been removed because it didn't really work very well.

Unknown words

CMU Pocket Sphinx uses pronunciation dictionaries to lookup phonetic representations for words in grammars, language models and key phrases in order to recognise them. If you use words in your grammars and/or key phrases that are *not* in the dictionary, a message similar to the following will be printed:

> *grammar 'name' used words not found in the pronunciation dictionary: notaword*

If you get a message like this, try changing the words in your grammars/key phrases by splitting up the words or using to similar words, e.g. changing "natlink" to "nat link".

I hope to eventually have words and phoneme strings dynamically added to the current dictionary and language model using the Pocket Sphinx ps_add_word function (from Python of course).

Spoken Language Support

There are a only handful of languages with models and dictionaries available from source forge, although it is possible to build your own language model using lmtool or pronunciation dictionary using lextool. There is also a CMU Sphinx tutorial on building language models.

If the language you want to use requires non-ascii characters (e.g. a Cyrillic language), you will need to use Python version 3.4 or higher because of Unicode issues.

Dragonfly Lists and DictLists

Dragonfly `Lists` and `DictLists` function as normal, private rules for the Pocket Sphinx engine. On updating a dragonfly list or dictionary, the grammar they are part of will be reloaded. This is because there is unfortunately no JSGF equivalent for lists.

Text-to-speech

This isn't a limitation of CMU Pocket Sphinx; text-to-speech is not a project goal for them, although as the natlink and WSR engines both support text-to-speech, there might as well be some suggestions if this functionality is desired, perhaps utilised by a custom dragonfly action.

The Jasper project contains a number of Python interface classes to popular open source text-to-speech software such as eSpeak, Festival and CMU Flite.

Text-input engine back-end

The text-input engine is a convenient, always available implementation designed to be used via the `engine.mimic()` method.

To initialise the text-input engine, do the following:

```
get_engine("text")
```

Note that *dragonfly.engines.get_engine()* called without `"text"` will **never** initialise the text-input engine. This is because real speech recognition backends should be returned from the function by default.

All dragonfly elements and rule classes should be supported. *executable*, *title*, and *handle* keyword arguments may optionally be passed to `engine.mimic()` to simulate a particular foreground window.

Engine Configuration

Dragonfly's *command-line interface* can be used to test command modules with the text-input engine. Below are a few use cases for this engine:

```
# Example 1:
# Initialize the text-input engine back-end, load a command module and
# recognize from stdin.
echo "hello world" | python -m dragonfly test _module1.py

# Example 2:
# Initialize the text-input engine back-end and load a command module
# for browser commands and recognize from stdin with a processing delay
# of 2 seconds. This allows testing context-specific browser commands.
python -m dragonfly test _browser.py --delay 2

# Example 3:
# Initialize the text-input engine back-end using German (de) as the
# language, then load a command module and recognize from stdin.
python -m dragonfly test _german_commands.py --language de
```

Engine API

class TextInputEngine

Text-input Engine class.

DictationContainer

alias of *dragonfly.engines.base.dictation.DictationContainerBase*

connect()

Connect to back-end SR engine.

create_timer(*callback*, *interval*, *repeating=True*)
Create and return a timer using the specified callback and repeat interval.

Timers created using this engine will be run in a separate daemon thread, meaning that their callbacks will **not** be thread safe. `threading.Timer()` may be used instead with no blocking issues.

disconnect()
Disconnect from back-end SR engine.

language
Current user language of the SR engine.
Return type str

mimic(*words*, ***kwargs*)
Mimic a recognition of the given *words*.
Parameters **words** (*str|iter*) – words to mimic
Keyword Arguments optional *executable*, *title* and/or *handle* keyword arguments may be used to simulate a specific foreground window context. The current foreground window attributes will be used instead for any keyword arguments not present.

set_exclusiveness(*grammar*, *exclusive*)
Set the exclusiveness of a grammar.

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

3.4.3 Text-to-speech (speaker) back-ends

For more information on the available text-to-speech implementations, see the following sections:

3.5 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.

Contents

3.5.1 Basic examples

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

```
from dragonfly 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.
```

3.5.2 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*.

3.5.3 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.

3.5.4 Action class reference

ActionBase base class

class ActionBase
Base class for Dragonfly's action classes.

exception ActionError

class ActionRepetition(*action*, *factor*)

class ActionSeries(**actions*)

stop_on_failures = True
Whether to stop executing if an action in the series fails.

class BoundAction(*action*, *data*)

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

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

Integer Repeat factors ignore any supply data:

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

Integer Repeat factors can be specified with the * operator:

```
>>> from dragonfly import Function
>>> def func():
...     print("executing 'func'")
...
>>> action = Function(func) * 3
>>> action.execute()
executing 'func'
executing 'func'
executing 'func'
```

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

```
>>> named = Repeat("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(extra="foo", count=3)
>>> 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
```

class UnsafeActionSeries(**actions*)

Key action

This section describes the *Key* action object. This type of action is used for sending keystrokes to the foreground application. This works on Windows, Mac OS and with X11 (e.g. on Linux). Examples of how to use this class are given in *Example key actions*.

To use this class on X11/Linux, the xdotool program must be installed and the `DISPLAY` environment variable set. This class does **not** support typing keys in Wayland sessions.

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. It should be given in hundredths of a second. For example, "20" will pause for 20/100s = 0.2 seconds.
- *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. It should be given in hundredths of a second. For example, "20" will pause for 20/100s = 0.2 seconds.
- *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: `!` or `bang` or `exclamation`, `@` or `at`, `#` or `hash`, `$` or `dollar`, `%` or `percent`, `^` or `caret`, `&` or `and` or `ampersand`, `*` or `star` or `asterisk`, `(` or `leftparen` or `lparen`, `)` or `rightparen` or `rparen`, `minus` or `hyphen`, `_` or `underscore`, `+` or `plus`, `` ` `` or `backtick`, `~` or `tilde`, `[` or `leftbracket` or `lbracket`, `]` or `rightbracket` or `rbracket`, `{` or `leftbrace` or `lbrace`, `}` or `rightbrace` or `rbrace`, `\` or `backslash`, `|` or `bar`, `colon`, `;` or `semicolon`, `'` or `apostrophe` or `singlequote` or `squote`, `"` or `quote` or `doublequote` or `dquote`, `comma`, `.` or `dot`, `slash`, `<` or `lessthan` or `leftangle` or `langle`, `>` or `greaterthan` or `rightangle` or `rangle`, `?` or `question`, `=` or `equal` or `equals`
- Whitespace and editing keys: `enter`, `tab`, `space`, `backspace`, `delete` or `del`
- Main modifier keys: `shift`, `control` or `ctrl`, `alt`
- Right modifier keys: `rshift`, `rcontrol` or `rctrl`, `ralt`

- Special keys: `escape`, `insert`, `pause`, `win`, `rwin`, `apps` or `popup`, `snapshot` or `printscreen`
- Lock keys: `scrolllock`, `numlock`, `capslock`
- 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:

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

Windows key support

Keyboard events sent by *Key* actions on Windows are calculated using the current foreground window's keyboard layout. The class will fallback on Unicode events for keys not typeable with the current layout.

The *Key* action can be used to type arbitrary Unicode characters on Windows using the relevant Windows API. This is disabled by default because it ignores the up/down status of modifier keys (e.g. ctrl).

It can be enabled by changing the `unicode_keyboard` setting in *~/.dragonfly2-speech/settings.cfg* to `True`:

```
unicode_keyboard = True
```

The `use_hardware` parameter can be set to `True` if you need to selectively require hardware events for a *Key* action:

```
# Passing use_hardware=True will guarantee that Ctrl+C is always
# pressed, regardless of the layout. See below.
Key("c-c", use_hardware=True).execute()
```

If the Unicode keyboard is not enabled or the `use_hardware` parameter is `True`, then no keys will be typed and an error will be logged for untypeable keys:

```
action.exec (ERROR): Execution failed: Keyboard interface cannot type this character:
↪'μ'
```

Keys in ranges 0-9, a-z and A-Z are always typeable. If keys in these ranges cannot be typed using the current keyboard layout, then the equivalent key will be used instead. For example, the following code will result in the ‘’ key being pressed when using the main Cyrillic keyboard layout:

```
# This is equivalent to Key(u", , c-").
Key("z, Z, c-z", use_hardware=True).execute()
```

Unlike the `Text` action, individual *Key* actions can send both hardware *and* Unicode events. So the following example will work if the Unicode keyboard is enabled:

```
# Type 'σμ' and then press Ctrl+Z.
Key(u"σ, μ, c-z").execute()
```

X11 key support

The *Key* action can be used to type arbitrary keys and Unicode characters on X11/Linux. It is not limited to the key names listed above, although all of them will work too.

Unicode characters are supported on X11 by passing their Unicode code point to the keyboard implementation. For example, the character `'€'` is converted to `'U20AC'`. The Unicode code point can also be passed directly, e.g. with `Key('U20AC')`.

Unlike on Windows, the *Key* action is able to use modifiers with Unicode characters on X11.

This class does **not** support typing keys in Wayland sessions.

Example X11 key actions

In addition to the examples in the previous section, the following example will work on X11/Linux.

The following code will type ‘σμ’ into the foreground application and then press ctrl+z:

```
Key("σ,μ,c-z").execute()
```

The following code will press ‘μ’ while holding control and alt:

```
Key("ca-μ").execute()
```

The following code will press the browser refresh multimedia key:

```
Key("XF86Refresh").execute()
```

Although this key is not defined in dragonfly’s typeables list, it still works because it is passed directly to xdotool. X11 (Xorg) multimedia keys can be found online: XF86 keyboard symbols.

Key class reference

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

Keystroke emulation action.

Constructor arguments:

- *spec* (*str*) – keystroke specification
- *static* (boolean) – flag indicating whether the specification contains dynamic elements
- *use_hardware* (boolean) – if *True*, send keyboard events using hardware emulation instead of as Unicode text. This will respect the up/down status of modifier keys.

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 for the current platform. The implementation for Windows uses the `sendinput()` Win32 API function. The implementation for Mac OS uses pynput. The implementation for X11/Linux uses xdotool.

class EventData(*keyname*, *direction*, *modifiers*, *inner_pause*, *repeat*, *outer_pause*)

Create new instance of EventData(keyname, direction, modifiers, inner_pause, repeat, outer_pause)

direction
Alias for field number 1

inner_pause
Alias for field number 3

keyname
Alias for field number 0

modifiers
Alias for field number 2

outer_pause
Alias for field number 5

repeat
Alias for field number 4

Text action

This section describes the *Text* action object. This type of action is used for typing text into the foreground application. This works on Windows, Mac OS and with X11 (e.g. on Linux).

To use this class on X11/Linux, the xdotool program must be installed and the `DISPLAY` environment variable set. This class does **not** support typing text in Wayland sessions.

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.

Windows Unicode Keyboard Support

The *Text* action can be used to type arbitrary Unicode characters using the relevant Windows API. This is disabled by default because it ignores the up/down status of modifier keys (e.g. ctrl).

It can be enabled by changing the `unicode_keyboard` setting in *~/.dragonfly2-speech/settings.cfg* to `True`:

```
unicode_keyboard = True
```

If you need to simulate typing arbitrary Unicode characters *and* have *individual* *Text* actions respect modifier keys normally for normal characters, set the configuration as above and use the `use_hardware` parameter for *Text* as follows:

```
action = Text("σμ") + Key("ctrl:down") + Text("]", use_hardware=True) + Key("ctrl:up")
action.execute()
```

Some applications require hardware emulation versus Unicode keyboard emulation. If you use such applications, add their executable names to the `hardware_apps` list in the configuration file mentioned above to make dragonfly always use hardware emulation for them.

If hardware emulation is required, then the action will use the keyboard layout of the foreground window when calculating keyboard events. If any of the specified characters are not typeable using the current window's keyboard layout, then an error will be logged and no keys will be typed:

```
action.exec (ERROR): Execution failed: Keyboard interface cannot type this character:
↪'μ'
```

Keys in ranges 0-9, a-z and A-Z are always typeable. If keys in these ranges cannot be typed using the current keyboard layout, then the equivalent key will be used instead. For example, the following code will result in the '' key being pressed when using the main Cyrillic keyboard layout:

```
# This is equivalent to Text(u"").
Text("zZ").execute()
```

These settings and parameters have no effect on other platforms.

X11/Linux Unicode Keyboard Support

The *Text* action can also type arbitrary Unicode characters on X11. This works regardless of the `use_hardware` parameter or `unicode_keyboard` setting.

Unlike on Windows, modifier keys will be respected by *Text* actions on X11. As such, the previous Windows example will work and can even be simplified a little:

```
action = Text("σμ") + Key("ctrl:down") + Text("]") + Key("ctrl:up")
action.execute()
```

It can also be done with one `Key` action:

```
Key("σ,μ,c-]").execute()
```

Text class reference

class Text(*spec=None*, *static=False*, *pause=None*, *autofmt=False*, *use_hardware=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.
- *use_hardware* (boolean) – if *True*, send keyboard events using hardware emulation instead of as Unicode text. This will respect the up/down status of modifier keys.

Paste action

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

Paste-from-clipboard action.

Constructor arguments:

- *contents* (*str* | *dict*) – contents to paste. This may be a simple string to paste, a dynamic action *spec* or a dictionary of clipboard format ints to contents (typically strings).
- *format* (*int*, clipboard format integer) – clipboard format. This argument is ignored if *contents* is a dictionary.
- *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 system clipboard, and then performs the *paste* action to paste it into the foreground application. By default, the *paste* action is the `Ctrl-v` keystroke or `Super-v` on a mac. The default clipboard format used by this action 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 primary monitor's left edge (if possible) 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 left-edge of the
#  primary monitor. They are used to access monitors above or to the
#  left of the primary monitor.
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:

- move the cursor relative to the top-left corner of the desktop monitor containing coordinates `[0, 0]` (i.e. the primary monitor): `[` *number* `,` *number* `]`
- move the cursor 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
  - `four` – fourth mouse button key
  - `five` – fifth mouse button key
  - `wheelup` – mouse wheel up
  - `stepup` – mouse wheel up 1/3
  - `wheeldown` – mouse wheel down
  - `stepdown` – mouse wheel down 1/3
  - `wheelright` – mouse wheel right
  - `stepright` – mouse wheel right 1/3
  - `wheelleft` – mouse wheel left
  - `stepleft` – mouse wheel left 1/3
- *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 across platforms

Please note that there are some platforms which do not support emulating every mouse button listed above. If an unsupported mouse button (*keyname*) is specified and the *Mouse* action executed, an error is raised. For instance, scrolling the mouse wheel horizontally (e.g. *wheelleft*) is not, by default, a supported operation on X11:

```
ValueError: Unsupported scroll event: wheelleft
```

Fortunately, this particular problem can be fixed by installing the *pynput* library:

```
pip install pynput
```

On MacOS, however, Dragonfly cannot be used to scroll horizontally.

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: %d" % 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: %d" % count)
...     print("flavor: %s" % 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
```

Usage with the `remap_data` argument:

```
>>> def func(x, y, z):
...     print("x: %d" % x)
...     print("y: %d" % y)
...     print("z: %d" % z)
...
>>> # The Function object can optionally be given a second dictionary
>>> # argument to use extras with different names. It should be
>>> # compatible with the 'defaults' parameter:
>>> action = Function(func, dict(n="x", m="y"), z=4)
>>> action.execute({"n": 2, "m": 3})
x: 2
y: 3
z: 4
True
```

Class reference

class Function (*function*, *remap_data=None*, ***defaults*)

Call a function with extra keyword arguments.

Constructor arguments:

- *function* (callable) – the function to call when this action is executed
- *remap_data* (dict, default: None) – optional dict of data keys to function keyword arguments
- defaults – default keyword-values for the arguments with which the function will be called

Mimic action

The *Mimic* action mimics a single recognition. This is useful for repeating a single prerecorded or predefined voice-command.

This class could for example be used to open a new Windows Explorer window:

```
action = Mimic("open", "windows", "explorer")
action.execute()
```

A more in-depth example is given below in the class reference.

Mimic quirks

Some SR engine back-ends have confusing `engine.mimic()` method behavior. See the engine-specific mimic method documentation in sections under *Engines sub-package* for more information.

Class reference

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),
```

(continues on next page)

(continued from previous page)

```
                  ])
action.execute()
```

Mimic quirks

Some SR engine back-ends have confusing `engine.mimic()` method behavior. See the engine-specific mimic method documentation in sections under *Engines sub-package* for more information.

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*, *index=None*, *filter_func=None*, *focus_only=False*)

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.
- *index* (*str* or *int*) – zero-based index of the target window, for multiple matching windows; can be a string (for substitution) but must be convertible to an integer.

- *filter_func* (*callable*) – called with a single argument (the window object), and should return `True` for your target windows; example: `lambda window: window.get_position().dy > 100`.
- *focus_only* (*bool*, default *False*) – if *True*, then attempt to focus the window without raising it by using the *Window.set_focus()* method instead of *set_foreground()*. This argument may do nothing depending on the platform.

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()
```

Applications on MacOS are started and switched to using the application name:

```
BringApp("System Preferences").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
- *title* (*str*, default *None*) – if not *None*, then match existing windows using this title.
- *index* (*str* or *int*) – zero-based index of the target window, for multiple matching windows; can be a string (for substitution) but must be convertible to an integer.
- *filter_func* (*callable*) – called with a single argument (the window object), and should return `True` for your target windows; example: `lambda window: window.get_position().dy > 100`.

- *focus_after_start* (*bool*, default *False*) – if *True*, then attempt to bring the window to the foreground after starting the application. Does nothing if the application is already running.
- *focus_only* (*bool*, default *False*) – if *True*, then attempt to focus a matching window without raising it by using the *set_focus()* method instead of *set_foreground()*. This argument may do nothing depending on the platform.

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
- *focus_after_start* (*bool*, default *False*) – if *True*, then attempt to bring the window to the foreground after starting the application.

A single *list* or *tuple* argument can be used instead of variable arguments.

RunCommand action

The *RunCommand* action takes a command-line program to run including any required arguments. On execution, the program will be started as a subprocess.

Processing will occur asynchronously by default. Commands running asynchronously should not normally prevent the Python process from exiting.

It may sometimes be necessary to use a list for the action's *command* argument instead of a string. This is because some command-line shells may not work 100% correctly with Python's built-in `shlex.split()` function.

This action should work on Windows and other platforms.

Example using the ping command:

```
from dragonfly import RunCommand

# Ping localhost for 4 seconds.
RunCommand('ping -w 4 localhost').execute()
```

Example using a command list instead of a string:

```
from dragonfly import RunCommand

# Ping localhost for 4 seconds.
RunCommand(['ping', '-w', '4', 'localhost']).execute()
```

Example using the optional function parameter:

```
from __future__ import print_function
from locale import getpreferredencoding
from six import binary_type
from dragonfly import RunCommand

def func(proc):
    # Read lines from the process.
    encoding = getpreferredencoding()
    for line in iter(proc.stdout.readline, b''):
        if isinstance(line, binary_type):
            line = line.decode(encoding)
```

(continues on next page)

(continued from previous page)

```
        print(line, end='')

RunCommand('ping -w 4 localhost', func).execute()
```

Example using the optional synchronous parameter:

```
from dragonfly import RunCommand

RunCommand('ping -w 4 localhost', synchronous=True).execute()
```

Example using the optional hide_window parameter:

```
from dragonfly import RunCommand

# Use hide_window=False for running GUI applications via RunCommand.
RunCommand('notepad.exe', hide_window=False).execute()
```

Example using the subprocess's `Popen` object:

```
from dragonfly import RunCommand

# Initialise and execute a command asynchronously.
cmd = RunCommand('ping -w 4 localhost')
cmd.execute()

# Wait until the subprocess finishes.
cmd.process.wait()
```

Example using a subclass:

```
from __future__ import print_function
from locale import getpreferredencoding
from six import binary_type
from dragonfly import RunCommand

class Ping(RunCommand):
    command = "ping -w 4 localhost"
    synchronous = True
    def process_command(self, proc):
        # Read lines from the process.
        encoding = getpreferredencoding()
        for line in iter(proc.stdout.readline, b''):
            if isinstance(line, binary_type):
                line = line.decode(encoding)
            print(line, end='')

Ping().execute()
```

Class reference

class RunCommand(*command=None*, *process_command=None*, *synchronous=False*, *hide_window=True*)

Start an application from the command-line.

This class is similar to the `StartApp` class, but is designed for running command-line applications and optionally processing subprocesses.

Constructor arguments:

- *command* (str or list) – the command to run when this action is executed. It will be parsed by `shlex.split()` if it is a string and passed directly to `subprocess.Popen` if it is a list. Command arguments can be included.
- *process_command* (callable) – optional callable to invoke with the `Popen` object after successfully starting the subprocess. Using this argument overrides the *process_command()* method.
- *synchronous* (bool, default *False*) – whether to wait until *process_command()* has finished executing before continuing.
- *hide_window* (bool, default *True*) – whether to hide the application window. Set to *False* if using this action with GUI programs. This argument only applies to Windows. It has no effect on other platforms.

process

The `Popen` object for the current subprocess if one has been started, otherwise `None`.

process_command(*proc*)

Method to override for custom handling of the command's `Popen` object.

By default this method prints lines from the subprocess until it exits.

ContextAction

class ContextAction(*default=None*, *actions=None*)

Action class to execute a different action depending on which context is currently active.

This is especially useful for allowing the same commands to work in multiple applications without redefining them in other grammars. An example of this is the redo shortcut. Some applications use *Ctrl+Shift+Z*, while others might use *Ctrl+Y* instead. `ContextAction` could be used to define *Ctrl+Y* as the default and use *Ctrl+Shift+Z* for specific contexts:

```
redo = ContextAction(default=Key('c-y'), actions=[
    # Use cs-z for rstudio
    (AppContext(executable="rstudio"), Key('cs-z')),
])
```

This class was originally written for the Aenea project by Alex Roper and has been modified to work without Aenea's functionality.

Constructor arguments:

- *default* (action object, default *do nothing*) – the default action to execute if there was no matching context in *actions*.
- *actions* (iterable, default *empty list*) – an iterable object containing context-action pairs. The action of the first matching context will be executed.

add_context(*context*, *action*)

Add a context-action pair to the *actions* list.

Parameters

- **context** (`Context`) – dragonfly context
- **action** (`ActionBase`) – dragonfly action

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)
```

PlaySound action

The *PlaySound* action class is used to play wave files.

Example usage

The following example shows how to play a wave file using the *PlaySound* action class:

```
PlaySound(file="tada.wav").execute()
```

Windows

On Windows, *PlaySound* uses the PlaySound Windows API function.

The action can be used to play Windows system sounds. For example:

```
# Play the system shutdown sound.
PlaySound("SystemExit").execute()

# Play the logout sound.
PlaySound("WindowsLogout").execute()
```

System sound names are matched against registry keys.

Invalid file paths or unknown system sounds will result in the default error sound being played. `RuntimeErrors` will be raised if Windows fails to play a known system sound.

Other platforms

On other platforms, the *PlaySound* class will use PyAudio to play specified wave files.

Invalid file paths will result in errors on other platforms.

Class reference

class PlaySound(*name=''*, *file=None*)

Start playing a wave file or system sound.

When this action is executed, the specified wave file or named system sound is played.

Playing named system sounds is only supported on Windows.
Constructor arguments:

- *name* (*str*, default *empty string*) – name of the Windows system sound to play. This argument is effectively an alias for *file* on other platforms.
- *file* (*str*, default *None*) – path of wave file to play when the action is executed.

If *name* and *file* are both *None*, then waveform playback will be silenced on Windows when the action is executed. Nothing will happen on other platforms.

3.6 Spoken Language Support

This section documents Dragonfly's support for spoken languages.

3.6.1 Languages with speech recognition engine support

Speech recognition engines supported by Dragonfly have a set spoken language. This language can be checked via the `engine.language` property, which returns an ISO 639-1 code (e.g. "en"):

```
from dragonfly import get_engine
engine = get_engine()

# Print the engine language.
print("Engine language: {}".format(engine.language))
```

Each speech recognition engine supported by Dragonfly supports many languages. These are listed below with citations.

It is worth noting that Dragonfly's use of ISO 639-1 language codes means that no distinction is made between variants of languages. For example, U.S. English and U.K. English will both yield `"en"` and be treated as the same language, even though there are some differences.

Languages supported by Dragon

The following languages are supported by *Dragon Professional Individual* version 15[1]:

- English (multiple variants)
- Dutch
- French
- German
- Italian
- Spanish

Please check the linked Nuance knowledgebase page for the languages supported by other versions and editions of Dragon.

Languages supported by Windows Speech Recognition

The following languages are supported by Windows Speech Recognition (WSR) as of 2016[2]:

- English (U.S.) (*)

[1] https://nuance.custhelp.com/app/answers/detail/a_id/6280/kw/Dragon%20NaturallySpeaking%20languages%20supported/related/1
[2] https://web.archive.org/web/20160501101405/http://www.microsoft.com:80/enable/products/windowsvista/speech.aspx

- English (U.K.)
- Chinese (Simplified) (*)
- Chinese (Traditional)
- French (France)
- German (Germany)
- Japanese
- Spanish (Spain)

* Successfully tested.

Microsoft does not appear to be documenting the languages available for WSR any more, which is why the provided citation for this section is an archive.org link. Currently, the best way to find out if your language is supported is to look for available speech models in the Windows language settings: **Settings > Time & Language > Language**.

Languages supported by CMU Pocket Sphinx

The CMU Pocket Sphinx engine documentation page has a section on *spoken language support*. There are CMU Pocket Sphinx models and dictionaries available from Source Forge for the following languages[3]:

- English (U.S.) (*)
- English (Indian)
- Catalan
- Chinese (Mandarin) (*)
- Dutch
- French
- German
- Greek
- Hindi
- Italian
- Kazakh
- Portuguese
- Russian (*)
- Spanish

* Successfully tested.

English (U.S.) is the default language used by the CMU Pocket Sphinx engine.

Languages supported by Kaldi

The following languages are supported by the *Kaldi engine back-end*:

- English (U.S.)

[3] https://sourceforge.net/projects/cmusphinx/files/Acoustic%20and%20Language%20Models/

It is possible for Kaldi to support other languages in the future. This requires finding decent models for other languages and making minor modifications to enable their use by the Kaldi Active Grammar library.

You can request to have your language supported by opening a new issue or by contacting David Zurow (@daanzu) directly.

3.6.2 Languages with built-in grammar support

Dragonfly's `Integer`, `IntegerRef` and `Digits` classes have support for multiple spoken languages. Each supported language has a sub-package under `dragonfly.language`. The current engine language will be used to load the language-specific content classes in these sub-packages.

This functionality is **optional**. Languages other than those listed below can still be used if the speech recognition supports them.

The following languages are supported:

- Arabic - "ar"
- Dutch - "nl"
- English - "en"
- German - "de"
- Indonesian - "id"
- Malaysian - "ms"

English has additional time, date and character related classes.

Language classes reference

ShortIntegerRef

`ShortIntegerRef` is a modified version of `IntegerRef` which allows for greater flexibility in the way that numbers may be pronounced, allowing for words like "hundred" to be dropped. This may be particularly useful when navigating files by line or page number.

Some examples of allowed pronunciations:

Pronunciation	Result
one	1
ten	10
twenty three	23
two three	23
seventy	70
seven zero	70
hundred	100
one oh three	103
hundred three	103
one twenty seven	127
one two seven	127
one hundred twenty seven	127
seven hundred	700
thousand	1000
seventeen hundred	1700
seventeen hundred fifty three	1753
seventeen fifty three	1753
one seven five three	1753
seventeen five three	1753
four thousand	4000

The class works in the same way as `IntegerRef`, by adding the following as an extra.

```
ShortIntegerRef("name", 0, 1000),
```

References

3.7 Windows sub-package

Dragonfly includes several toolkits for non-speech user interfaces. These include for example, cross-platform generic window control, access to monitor information and access to the system clipboard.

Contents of Dragonfly's windows sub-package:

3.7.1 Clipboard toolkit

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

Dragonfly chooses the clipboard class to alias as `Clipboard` based on the platform:

- *dragonfly.windows.win32_clipboard.Win32Clipboard* is used on Windows.
- *dragonfly.windows.x11_clipboard.XselClipboard* is used on X11/Linux. This class requires the `xclip` program.
- *dragonfly.windows.pyperclip_clipboard.PyperclipClipboard* is used on other platforms, such as macOS or Linux.

API differences

All platform clipboard classes have the same API, plus any platform-specific formats. The following two basic formats are always available:

1. `format_text` – ANSI text format (format int: 1)
2. `format_unicode` – Unicode text format (format int: 13)

The basic clipboard format constants used should match those used on Windows, e.g. `format_unicode` is represented by 13 (`CF_UNICODETEXT`). To be safe, use the `format_*` attributes defined by the clipboard class.

Usage examples

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

Base Clipboard class

class BaseClipboard(*contents=None*, *text=None*, *from_system=False*)
Base clipboard class.

classmethod clear_clipboard()
Clear the system clipboard.

classmethod convert_format_content(*format*, *content*)
Convert content for the given *format*, if necessary, and return it.
This method operates on the following formats:
- *text* – encodes content to a binary string, if necessary and if possible.
- *unicode* – decodes content to a text string, if necessary and if possible.
- *hdrop* – converts content into a tuple of file paths, if necessary and if possible.

  String content must be a list of existing, absolute file paths separated by new lines and/or null characters. All specified file paths must be absolute paths referring to existing files on the system.

If the content cannot be converted for the given *format*, an error is raised.
Arguments:
- *format* (int) – the clipboard format to convert.
- *content* (string) – the clipboard contents to convert.

copy_from_system(*formats=None*, *clear=False*)
Copy the 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 system clipboard will be cleared after its contents have been retrieved.

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

get_available_formats()
Retrieve a list of this instance's available formats.

The preferred text format, if available, will always be the first on the list followed by any remaining formats in numerical order.

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.

classmethod get_system_text()
Retrieve the system clipboard text.

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.

set_format(*format*, *content*)
Set this instance's content for the given *format*.
Arguments:
- *format* (int) – the clipboard format to set.
- *content* (string) – the clipboard contents to set.

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

If *None* is given as the *content*, any content stored for the given *format* will be cleared.

classmethod set_system_text(*content*)
Set the system clipboard text.
Arguments:
- *content* (string) – the clipboard contents to set.

If *None* is given as the *content*, text on the system clipboard will be cleared.

set_text(*content*)
Set the text content for this instance.
Arguments:
- *content* (string) – the text content to set.

If *None* is given as the *content*, any text content stored in this instance will be cleared.

classmethod synchronized_changes(*timeout*, *step=0.001*, *formats=None*, *initial_clipboard=None*)
Context manager for synchronizing local and system clipboard changes. This takes the same arguments as the *wait_for_change()* method.
Arguments:
- *timeout* (float) – timeout in seconds.
- *step* (float, default: 0.001) – number of seconds between each check.
- *formats* (iterable, default: None) – if not None, only changes to the given content formats will register. If None, all formats will be observed.
- *initial_clipboard* (Clipboard, default: None) – if a clipboard is given, the method will wait until the system clipboard differs from the instance's contents.

Use with a Python 'with' block:

```
from dragonfly import Clipboard, Key

# Copy the selected text with Ctrl+C and wait until a system
#  clipboard change is detected.
timeout = 3
with Clipboard.synchronized_changes(timeout):
    Key("c-c", use_hardware=True).execute()

# Retrieve the system text.
text = Clipboard.get_system_text()
```

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

classmethod wait_for_change(*timeout*, *step=0.001*, *formats=None*, *initial_clipboard=None*)
Wait (poll) for the system clipboard to change.

This is a blocking method which returns whether or not the system clipboard changed within a specified timeout period.
Arguments:
- *timeout* (float) – timeout in seconds.
- *step* (float, default: 0.001) – number of seconds between each check.

- *formats* (iterable, default: None) – if not None, only changes to the given content formats will register. If None, all formats will be observed.
- *initial_clipboard* (Clipboard, default: None) – if a clipboard is given, the method will wait until the system clipboard differs from the instance's contents.

Windows Clipboard class

class Win32Clipboard(*contents=None*, *text=None*, *from_system=False*)

Class for interacting with the Windows system clipboard.

This is Dragonfly's default clipboard class on Windows.

classmethod clear_clipboard()

Clear the system clipboard.

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

Copy the 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 system clipboard will be cleared after its contents have been retrieved.

copy_to_system(*clear=True*)

Copy the contents of this instance into the system clipboard.

Arguments:

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

classmethod get_system_text()

Retrieve the system clipboard text.

classmethod set_system_text(*content*)

Set the system clipboard text.

Arguments:

- *content* (string) – the clipboard contents to set.

If *None* is given as the *content*, text on the system clipboard will be cleared.

classmethod synchronized_changes(*timeout*, *step=0.001*, *formats=None*, *initial_clipboard=None*)

Context manager for synchronizing local and system clipboard changes. This takes the same arguments as the *wait_for_change()* method.

Arguments:

- *timeout* (float) – timeout in seconds.
- *step* (float, default: 0.001) – number of seconds between each check.
- *formats* (iterable, default: None) – if not None, only changes to the given content formats will register. If None, all formats will be observed.
- *initial_clipboard* (Clipboard, default: None) – if a clipboard is given, the method will wait until the system clipboard differs from the instance's contents.

Use with a Python 'with' block:

```
from dragonfly import Clipboard, Key

# Copy the selected text with Ctrl+C and wait until a system
#  clipboard change is detected.
timeout = 3
with Clipboard.synchronized_changes(timeout):
```

(continues on next page)

(continued from previous page)

```
    Key("c-c", use_hardware=True).execute()

# Retrieve the system text.
text = Clipboard.get_system_text()
```

classmethod wait_for_change(*timeout*, *step=0.001*, *formats=None*, *initial_clipboard=None*)
Wait (poll) for the system clipboard to change.

This is a blocking method which returns whether or not the system clipboard changed within a specified timeout period.

Arguments:

- *timeout* (float) – timeout in seconds.
- *step* (float, default: 0.001) – number of seconds between each check.
- *formats* (iterable, default: None) – if not None, only changes to the given content formats will register. If None, all formats will be observed.
- *initial_clipboard* (Clipboard, default: None) – if a clipboard is given, the method will wait until the system clipboard differs from the instance's contents.

Windows Clipboard context manager

class win32_clipboard_ctx(*timeout=0.5*, *step=0.001*)
Python context manager for safely opening the Windows clipboard by polling for access, timing out after the specified number of seconds.

Arguments:

- *timeout* (float, default: 0.5) – timeout in seconds.
- *step* (float, default: 0.001) – number of seconds between each attempt to open the clipboard.

Notes:

- The polling is necessary because the clipboard is a shared resource which may be in use by another process.
- Nested usage will not close the clipboard early.

Use with a Python 'with' block:

```
with win32_clipboard_ctx():
    # Do clipboard operation(s).
    win32clipboard.EmptyClipboard()
```

X11 Clipboard classes

class BaseX11Clipboard(*contents=None*, *text=None*, *from_system=False*)
Base X11 clipboard class.

classmethod clear_clipboard()
Clear the system clipboard.

copy_from_system(*formats=None*, *clear=False*)
Copy the 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 system clipboard will be cleared after its contents have been retrieved.

copy_to_system(*clear=True*)
Copy the contents of this instance into the system clipboard.

Arguments:

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

format_x_clipboard = 13
Format for the clipboard X selection (alias of format_unicode).

format_x_primary = 65536
Format for the primary X selection.

format_x_secondary = 65537
Format for the secondary X selection.

classmethod get_system_text()
Retrieve the system clipboard text.

classmethod set_system_text(*content*)
Set the system clipboard text.
Arguments:

- *content* (string) – the clipboard contents to set.

If *None* is given as the *content*, text on the system clipboard will be cleared.

class XselClipboard(*contents=None*, *text=None*, *from_system=False*)
Class for interacting with X selections (clipboards) using xsel.

This is Dragonfly's default clipboard class on X11/Linux.

Pyperclip Clipboard class

class PyperclipClipboard(*contents=None*, *text=None*, *from_system=False*)
Class for interacting with the system clipboard via the pyperclip Python package.

This is Dragonfly's default clipboard class on platforms other than Windows.

Note: This class does work on Windows, however the Windows *dragonfly.windows.win32_clipboard.Win32Clipboard* class should be used instead because this class doesn't support as many clipboard formats.

---

classmethod clear_clipboard()
Clear the system clipboard.

copy_from_system(*formats=None*, *clear=False*)
Copy the 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 system clipboard will be cleared after its contents have been retrieved.

copy_to_system(*clear=True*)
Copy the contents of this instance into the system clipboard.
Arguments:

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

classmethod get_system_text()
Retrieve the system clipboard text.

classmethod set_system_text(*content*)
Set the system clipboard text.
Arguments:
- *content* (string) – the clipboard contents to set.

If *None* is given as the *content*, text on the system clipboard will be cleared.

3.7.2 Monitor classes

The monitor classes are how Dragonfly gets information on the available monitors (a.k.a. screens, displays) for the current platform. Currently the Windows, macOS and X11 (Linux) platforms are supported.

class BaseMonitor(*handle*, *rectangle*)
The base monitor class.

classmethod get_all_monitors()
Get a list containing each connected monitor.
Return type list
Returns monitors

classmethod get_monitor(*handle*, *rectangle*)
Get a `Monitor` object given a monitor handle.

Given the same handle, this method will return the same object.
Parameters
- **handle** (*int*) – monitor handle
- **rectangle** (*Rectangle*) – monitor rectangle

Return type Monitor
Returns monitor

handle
Protected access to handle attribute.

is_primary
Whether this is the primary display monitor.
Return type bool
Returns true or false

name
The name of this monitor.
Return type str
Returns monitor name

rectangle
Protected access to rectangle attribute.

class FakeMonitor(*handle*, *rectangle*)
The monitor class used on unsupported platforms.

classmethod get_all_monitors()
Get a list containing each connected monitor.
Return type list
Returns monitors

name
The name of this monitor.
Return type str
Returns monitor name

class Win32Monitor(*handle*, *rectangle*)
The monitor class used on Win32.

classmethod get_all_monitors()
Get a list containing each connected monitor.
Return type list
Returns monitors

is_primary
Whether this is the primary display monitor.
Return type bool
Returns true or false

name
The device name of this monitor.
Return type str
Returns monitor name

class X11Monitor(*name*, *rectangle*)
The monitor class used on X11 (Linux).

This implementation parses monitor information from `xrandr`.

classmethod get_all_monitors()
Get a list containing each connected monitor.
Return type list
Returns monitors

is_primary
Whether this is the primary display monitor.

name
Protected access to name attribute.

class DarwinMonitor(*handle*, *rectangle*)
The monitor class used on Mac OS (Darwin).

Monitors for Mac OS are **not** automatically updated because `AppKit` doesn't update monitor information after it is retrieved. A simple workaround for this is to restart the Python interpreter after a monitor change.

classmethod get_all_monitors()
Get a list containing each connected monitor.
Return type list
Returns monitors

name
The name of this monitor.

This module initializes the monitor interface for the current platform.

class MonitorList
Special read-only, self-updating monitors list class.

Supports indexing and iteration.

monitors = <dragonfly.windows.monitor.MonitorList object>
`MonitorsList` instance

3.7.3 Window classes

Dragonfly's Window classes are interfaces to the window control and placement APIs for the current platform. Currently the Windows, macOS and X11 (Linux) platforms are supported.

The `FakeWindow` class will be used on unsupported platforms.

Base Window class

class BaseWindow(*id*)

The base Window class for controlling and placing windows.

classname

Read-only access to the window's class name.

close()

Close the window (if possible).

cls_name

Alias of *classname*.

executable

Read-only access to the window's executable.

classmethod get_all_windows()

Get a list of all windows.

get_containing_monitor()

Method to get the `Monitor` containing the window.

This checks which monitor contains the center of the window.

Returns containing monitor
Return type Monitor

classmethod get_foreground()

Get the foreground window.

classmethod get_matching_windows(*executable=None*, *title=None*)

Find windows with a matching executable or title.

Window searches are case-insensitive.

If neither parameter is be specified, then it is effectively the same as calling *get_all_windows()*.

Parameters

- **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.

Return type list

get_normalized_position()

Method to get the window's normalized position.

This is useful when working with multiple monitors.

Returns normalized position
Return type Rectangle

get_position()

Method to get the window's position as a `Rectangle` object.

Returns window position
Return type Rectangle

classmethod get_window(*id*)
Get a `Window` object given a window id.

Given the same id, this method will return the same object.

handle
Protected access to handle attribute.

id
Protected access to id attribute.

is_maximized
Whether the window is currently maximized.

is_minimized
Whether the window is currently minimized.

is_visible
Whether the window is currently visible.

This may be indeterminable for some windows.

matches(*context*)
Whether the window matches the given context.

maximize()
Maximize the window (if possible).

minimize()
Minimize the window (if possible).

move(*rectangle*, *animate=None*)
Move the window, optionally animating its movement.
Parameters

- **rectangle** (*Rectangle*) – new window position and size
- **animate** (*str*) – name of window mover

name
Protected access to name attribute.

pid
Read-only access to the window's process ID.

This will be the PID of the window's process, not any subprocess.

If the window has no associated process id, this will return `None`.
Returns pid
Return type int | None

restore()
Restore the window if it is minimized or maximized.

set_focus()
Set the window as the active window without raising it.

Note: this method will behave like *set_foreground()* in environments where this isn't possible.

set_foreground()
Set the window as the foreground (active) window.

set_normalized_position(*rectangle*, *monitor=None*)
Method to get the window's normalized position.

This is useful when working with multiple monitors.

Parameters

- **rectangle** (*Rectangle*) – window position
- **monitor** (*Monitor*) – monitor to normalize to (default: the first one).

set_position(*rectangle*)
Method to set the window's position using a `Rectangle` object.

Parameters **rectangle** (*Rectangle*) – window position

title
Read-only access to the window's title.

class FakeWindow(*id*)
Fake Window class used when no implementation is available.

close()
Close the window (if possible).

classmethod get_all_windows()
Get a list of all windows.

classmethod get_foreground()
Get the foreground window.

get_position()
Method to get the window's position as a `Rectangle` object.

Returns window position
Return type Rectangle

is_maximized
Whether the window is currently maximized.

is_minimized
Whether the window is currently minimized.

is_visible
Whether the window is currently visible.

This may be indeterminable for some windows.

maximize()
Maximize the window (if possible).

minimize()
Minimize the window (if possible).

restore()
Restore the window if it is minimized or maximized.

set_focus()
Set the window as the active window without raising it.

Note: this method will behave like *set_foreground()* in environments where this isn't possible.

set_foreground()
Set the window as the foreground (active) window.

set_position(*rectangle*)
Method to set the window's position using a `Rectangle` object.

Parameters **rectangle** (*Rectangle*) – window position

Window class for Windows

class Win32Window(*handle*)
The Window class is an interface to the Win32 window control and placement.

classmethod get_all_windows()
Get a list of all windows.

classmethod get_foreground()
Get the foreground window.

classmethod get_matching_windows(*executable=None*, *title=None*)
Find windows with a matching executable or title.

Window searches are case-insensitive.

If neither parameter is be specified, then it is effectively the same as calling *get_all_windows()*.

Parameters

- **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.

Return type list

get_position()
Method to get the window's position as a `Rectangle` object.

Returns window position
Return type Rectangle

is_enabled
Shortcut to win32gui.IsWindowEnabled() function.

is_maximized
Whether the window is currently maximized.

is_minimized
Shortcut to win32gui.IsIconic() function.

is_valid
Shortcut to win32gui.IsWindow() function.

is_visible
Shortcut to win32gui.IsWindowVisible() function.

set_focus()
Set the window as the active window without raising it.

Note: this method will behave like *set_foreground()* in environments where this isn't possible.

set_foreground()
Set the window as the foreground (active) window.

set_position(*rectangle*)
Method to set the window's position using a `Rectangle` object.

Parameters **rectangle** (*Rectangle*) – window position

Window class for X11

class X11Window(*id*)
The Window class is an interface to the window control and placement APIs for X11.

Window control methods such as *close()* will return `True` if successful.

This class requires the following external programs:

- `wmctrl`
- `xdotool`
- `xprop`

close()
Close the window (if possible).

cls
Read-only access to the window's class.

classmethod get_all_windows()
Get a list of all windows.

classmethod get_foreground()
Get the foreground window.

classmethod get_matching_windows(*executable=None*, *title=None*)
Find windows with a matching executable or title.

Window searches are case-insensitive.

If neither parameter is be specified, then it is effectively the same as calling *get_all_windows()*.

Parameters

- **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.

Return type list

get_position()
Method to get the window's position as a `Rectangle` object.

Returns window position
Return type Rectangle

is_focused
Whether the window has input focus.

This does not work for all window types (e.g. pop up menus).

Return type bool

is_fullscreen
Whether the window is in fullscreen mode.

This does not work for all window types (e.g. pop up menus).

Return type bool

is_maximized
Whether the window is currently maximized.

is_minimized
Whether the window is currently minimized.

is_visible
Whether the window is currently visible.

This may be indeterminable for some windows.

maximize()
Maximize the window (if possible).

minimize()
Minimize the window (if possible).

restore()
Restore the window if it is minimized or maximized.

role
Read-only access to the window's X11 role attribute.
Returns role
Return type str

set_focus()
Set the input focus to this window.

This method will set the input focus, but will not necessarily bring the window to the front.

set_foreground()
Set the window as the foreground (active) window.

set_position(*rectangle*)
Method to set the window's position using a `Rectangle` object.
Parameters **rectangle** (*Rectangle*) – window position

state
Read-only access to the X window state.

Windows can have multiple states, so this returns a tuple.

This property invokes a (relatively) long-running function, so store the result locally instead of using it multiple times.

If the window does not have the _NET_WM_STATE property, then `None` will be returned.
Returns window state (if any)
Return type tuple | None

type
Read-only access to the window's X11 type property, if it is set.
Returns type
Return type str

Window class for macOS

class DarwinWindow(*id*)
The Window class is an interface to the macOS window control and placement.

close()
Close the window (if possible).

full_screen()
Enable full screen mode for this window.

Note: this doesn't allow transitioning out of full screen mode.

classmethod get_all_windows()
Get a list of all windows.

get_attribute(*attribute*)
Method to get an attribute of a macOS window.

Note: this method doesn't distinguish between multiple instances of the same application.
Parameters **attribute** (*string*) – attribute name
Returns attribute value

classmethod get_foreground()
Get the foreground window.

classmethod get_matching_windows(*executable=None*, *title=None*)
Find windows with a matching executable or title.

Window searches are case-insensitive.

If neither parameter is be specified, then it is effectively the same as calling *get_all_windows()*.

Parameters

- **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.

Return type list

get_position()
Method to get the window's position as a `Rectangle` object.

Returns window position
Return type Rectangle

get_properties()
Method to get the properties of a macOS window.

Return type dict
Returns window properties

is_maximized
Whether the window is currently maximized.

is_minimized
Whether the window is currently minimized.

is_visible
Whether the window is currently visible.

This may be indeterminable for some windows.

maximize()
Maximize the window (if possible).

minimize()
Minimize the window (if possible).

restore()
Restore the window if it is minimized or maximized.

set_focus()
Set the window as the active window without raising it.

Note: this method will behave like *set_foreground()* in environments where this isn't possible.

set_foreground()
Set the window as the foreground (active) window.

set_position(*rectangle*)
Method to set the window's position using a `Rectangle` object.

Parameters **rectangle** (*Rectangle*) – window position

3.8 Accessibility API

The accessibility API enables text selection and editing more powerful than what Dragon provides natively, to a wider range of applications (e.g. Google Chrome on Windows and Mozilla Firefox on Windows and X11). It is currently in Beta and the API may change at any time.

3.8.1 Windows

To use this API on Windows, install pyia2. To use this with Chrome, you may also need to register an additional 64-bit IAccessible2 DLL which can be obtained here.

3.8.2 X11 (Linux)

To use this on X11, you will need to install the Python library pyatspi. You can typically get this from your distribution's package manager. See this stack overflow question for examples.

Next, add the following to your ~/.profile file:

```
export GTK_MODULES=gail:atk-bridge
export OOO_FORCE_DESKTOP=gnome
export GNOME_ACCESSIBILITY=1
export QT_ACCESSIBILITY=1
export QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
```

Some applications will not support assistive technologies unless these settings are enabled. You may also need to enable GNOME accessibility with `gsettings`:

```
gsettings set org.gnome.desktop.interface toolkit-accessibility true
```

This page has some useful information on testing / troubleshooting accessibility support. The `troubleshoot` script from check-a11y can find problems with the accessibility stack:

```
git clone https://git.debian.org/git/pkg-a11y/check-a11y
cd check-a11y
./troubleshoot
```

Most functionality works properly on X11, except for a few known issues:

- Text selection occasionally requires use of the Mouse action due to limitations of the text selection API.
- LibreOffice treats each paragraph as a separate focusable element, so movement between paragraphs is not yet supported.

3.8.3 Entry points

This module initializes the accessibility controller for the current platform.

get_accessibility_controller()
: Get the OS-independent accessibility controller which is the gateway to all accessibility functionality. Returns None if OS is not supported.

get_stopping_accessibility_controller()
: Same as *get_accessibility_controller()*, but automatically stops when used in a *with* context.

3.8.4 Controller class

class AccessibilityController(*os_controller*)
OS-independent controller for accessing accessibility functionality.

is_editable_focused()
True if an editable text field is focused.

move_cursor(*text_query*, *position*)
Moves the cursor before or after text that matches the provided query.

replace_text(*text_query*, *replacement*)
Replaces text which matches the provided query.

select_text(*text_query*)
Selects text which matches the provided query.

stop()
Stops the controller (otherwise process exit may be blocked).

3.8.5 TextQuery class

class TextQuery(*start_phrase=''*, *start_relative_position=None*, *start_relative_phrase=''*, *through=False*, *end_phrase=''*, *end_relative_position=None*, *end_relative_phrase=''*)
A query to match a range of text.

end_phrase = None
The phrase at the end of the match (or the sole phrase).

end_relative_phrase = None
The phrase to match before or after at the end.

end_relative_position = None
Whether to match before or after the *end_relative_phrase*.

start_phrase = None
The phrase at the start of the match.

start_relative_phrase = None
The phrase to match before or after at the start.

start_relative_position = None
Whether to match before or after the *start_relative_phrase*.

through = None
True if matching from a start point to the end phrase.

3.8.6 CursorPosition class

class CursorPosition
The cursor position relative to a range of text.

AFTER = 2
The position after the text.

BEFORE = 1
The position before the text.

3.9 Command-line Interface (CLI)

Command-line interface to the Dragonfly speech recognition framework

```
usage: python -m dragonfly [-h] {test,load,load-directory} ...
```

3.9.1 Positional Arguments

command Possible choices: test, load, load-directory

3.9.2 Sub-commands

test

Load grammars from Python files for testing with a dragonfly engine. By default input from stdin is passed to engine.mimic() after command modules are loaded.

```
python -m dragonfly test [-h] [-e ENGINE]
                         [-o ENGINE_OPTIONS [ENGINE_OPTIONS ...]]
                         [--language LANGUAGE] [-n] [-d DELAY]
                         [-l {DEBUG,INFO,WARNING,ERROR,CRITICAL}] [-q]
                         [file [file ...]]
```

Positional Arguments

file Command module file(s).

Named Arguments

-e, --engine Name of the engine to use for testing.

Default: "text"

-o, --engine-options One or more engine options to be passed to *get_engine()*. Each option should specify a key word argument and value. Multiple options should be separated by spaces or commas. Values are treated as Python literals if possible, otherwise as strings.

Default: []

--language Speaker language to use. Only applies if using an engine backend that supports changing the language (e.g. the "text" engine).

Default: "en"

-n, --no-input Whether to load command modules and then exit without reading input from stdin or recognizing speech.

Default: False

-d, --delay Time in seconds to delay before mimicking each command. This is useful for testing contexts.

Default: 0

-l, --log-level Possible choices: DEBUG, INFO, WARNING, ERROR, CRITICAL

Log level to use.

Default: "INFO"

-q, --quiet Equivalent to '-l WARNING' – suppresses INFO and DEBUG logging.

Default: False

load

Load and recognize from command module files.

```
python -m dragonfly load [-h] [-e ENGINE]
                         [-o ENGINE_OPTIONS [ENGINE_OPTIONS ...]]
                         [--language LANGUAGE] [-n] [--no-recobs-messages]
                         [-l {DEBUG,INFO,WARNING,ERROR,CRITICAL}] [-q]
                         [file [file ...]]
```

Positional Arguments

file Command module file(s).

Named Arguments

-e, --engine Name of the engine to use for loading and recognizing from command modules. By default, this is the first available engine backend.

-o, --engine-options One or more engine options to be passed to *get_engine()*. Each option should specify a key word argument and value. Multiple options should be separated by spaces or commas. Values are treated as Python literals if possible, otherwise as strings.

Default: []

--language Speaker language to use. Only applies if using an engine backend that supports changing the language (e.g. the "text" engine).

Default: "en"

-n, --no-input Whether to load command modules and then exit without reading input from stdin or recognizing speech.

Default: False

--no-recobs-messages Disable recognition state messages for each spoken phrase.

Default: False

-l, --log-level Possible choices: DEBUG, INFO, WARNING, ERROR, CRITICAL

Log level to use.

Default: "INFO"

-q, --quiet Equivalent to '-l WARNING' – suppresses INFO and DEBUG logging.

Default: False

load-directory

Load and recognize from command module files in one or more directories. Only module files starting with an underscore (_*.py) are loaded by this command.

```
python -m dragonfly load-directory [-h] [-r] [-e ENGINE]
                                   [-o ENGINE_OPTIONS [ENGINE_OPTIONS ...]]
                                   [--language LANGUAGE] [-n]
                                   [--no-recobs-messages]
                                   [-l {DEBUG,INFO,WARNING,ERROR,CRITICAL}]
                                   [-q]
                                   dir [dir ...]
```

Positional Arguments

dir One or more command module directories.

Named Arguments

-r, --recursive Whether to recursively load command modules in sub-directories.

Default: False

-e, --engine Name of the engine to use for loading and recognizing from command modules. By default, this is the first available engine backend.

-o, --engine-options One or more engine options to be passed to *get_engine()*. Each option should specify a key word argument and value. Multiple options should be separated by spaces or commas. Values are treated as Python literals if possible, otherwise as strings.

Default: []

--language Speaker language to use. Only applies if using an engine backend that supports changing the language (e.g. the "text" engine).

Default: "en"

-n, --no-input Whether to load command modules and then exit without reading input from stdin or recognizing speech.

Default: False

--no-recobs-messages Disable recognition state messages for each spoken phrase.

Default: False

-l, --log-level Possible choices: DEBUG, INFO, WARNING, ERROR, CRITICAL

Log level to use.

Default: "INFO"

-q, --quiet Equivalent to '-l WARNING' – suppresses INFO and DEBUG logging.

Default: False

3.9.3 Usage Examples

Below are some examples using each of the available sub-commands. All examples should work in Bash, PowerShell, cmd.exe, etc.

`test` examples

```
# Load a command module and mimic two commands separately.
echo "command one\n command two" | python -m dragonfly test module.py

# Same test without the pipe.
python -m dragonfly test module.py
command one
command two

# Same test with quieter output.
echo "command one\n command two" | python -m dragonfly test -q module.py

# Test loading two modules with the sphinx engine and exit without
# checking input.
python -m dragonfly test -e sphinx --no-input module1.py module2.py

# Load a command module with the text engine's language set to German.
python -m dragonfly test --language de module.py

# Use the --delay command to test context-dependent commands.
echo "save file" | python -m dragonfly test --delay 1 _notepad_example.py
```

`load` examples

```
# Load command modules and recognize speech in a loop using the default
# engine.
python -m dragonfly load _*.py

# Load command modules and exit afterwards.
python -m dragonfly load --no-input _*.py

# Load command modules and disable recognition state messages such as
# "Speech start detected".
python -m dragonfly load --no-recobs-messages _*.py

# Load one command module with the Sphinx engine.
python -m dragonfly load --engine sphinx sphinx_commands.py

# Initialize the Kaldi engine backend with custom arguments, then load
# command modules and recognize speech.
python -m dragonfly load _*.py --engine kaldi --engine-options " \
    model_dir=kaldi_model_zamia \
    vad_padding_end_ms=300"
```

`load-directory` examples

```
# Load command modules in the "command-modules" directory and recognize
# speech in a loop using the default engine.
python -m dragonfly load-directory command-modules

# Load command modules in the "command-modules" directory and any sub-
# directories, then recognize speech in a loop using the default engine.
python -m dragonfly load-directory -r command-modules
python -m dragonfly load-directory --recursive command-modules

# Load command modules in the "command-modules" directory and recognize
# speech without printing recognition state messages.
python -m dragonfly load-directory --no-recobs-messages command-modules

# Load command modules in the "wsr-modules" directory and recognize
# speech using the WSR/SAPI5 in-process engine backend.
python -m dragonfly load-directory -e sapi5inproc wsr-modules

# Initialize the Kaldi engine backend with some custom arguments, then
# load command modules in the current directory and recognize speech.
python -m dragonfly load-directory . --engine kaldi --engine-options " \
    model_dir=kaldi_model_zamia \
    vad_padding_end_ms=300"
```

3.10 Logging infrastructure

Dragonfly's logging infrastructure is defined in the `dragonfly.log` module. It defines sane defaults for the various loggers used in the library as well as functions for setting up logging and tracing.

3.10.1 Adjusting logger levels

Dragonfly's logger levels can be adjusted much like Python logger levels:

```
import logging
logging.getLogger("engine").setLevel(logging.DEBUG)
```

The one caveat is that this must be done *after* the *setup_log()* function is called, otherwise the levels you set will be overridden. By default, the function is only called near the top of the module loader scripts (e.g. *dragonfly/examples/dfly-loader-wsr.py*), not within dragonfly itself.

It is not necessary to call *setup_log()* at all. Standard Python logging functions such as `basicConfig()` can be used at the top of module loader scripts instead.

If you are not using dragonfly with a module loader, you will need to set up a logging handler to avoid messages like tho following:

```
No handlers could be found for logger "typeables"
```

3.10.2 Functions

setup_log(*use_stderr=True*, *use_file=True*, *use_stdout=False*)
Setup Dragonfly's logging infrastructure with sane defaults.

Parameters

- **use_stderr** (*bool*) – whether to output log messages to stderr (default: True).
- **use_file** (*bool*) – whether to output log messages to the *~/.dragonfly.log* log file (default: True).
- **use_stdout** (*bool*) – this parameter does nothing andhas been left in for backwards-compatibility (default: False).

setup_tracing(*output*, *limit=None*)

Setup call tracing for low-level debugging.

Parameters

- **output** (*file*) – the file to write tracing messages to.
- **limit** (*int|None*) – the recursive depth limit for tracing (default: None).

3.11 Remote Procedure Call (RPC) sub-package

Dragonfly's remote procedure call (RPC) sub-package allows for interaction with a dragonfly speech recognition engine running in a remote process. This is useful for building responsive GUI applications without having to integrate them into a loaded grammar or grammar rule.

Some use cases for this framework include:

- Listing available commands and grammars in the current context.
- Integrating and displaying documentation.
- Guiding the user through complex commands.
- A GUI for building dragonfly/third-party grammars.

3.11.1 RPC server

Dragonfly's RPC server handles requests by processing each method through the current engine's *multiplexing timer interface*. This allows engines to handle requests safely and keeps engine-specific implementation details out of the *dragonfly.rpc* sub-package.

Security tokens

The RPC server uses mandatory security tokens to authenticate client requests. This avoids some security issues where, for example, malicious web pages could send POST requests to the server, even if running on localhost.

If sending requests over an open network, please ensure that the connection is secure by using TLS or SSH port forwarding.

If the server's `security_token` constructor parameter is not specified, a new token will be generated and printed to the console. Clients must specify the security token either as the last positional argument or as the `security_token` keyword argument.

Errors will be raised if clients send no security token or a token that doesn't match the server's.

Class reference

exception PermissionDeniedError

Error raised if clients send security tokens that don't match the server's.

class RPCServer(*address='127.0.0.1'*, *port=50051*, *ssl_context=None*, *threaded=True*, *security_token=None*)

RPC server class.

This class will run a local web server on port 50051 by default. The server expects requests using JSON-RPC 2.0.

Constructor arguments:

- *address* – address to use (*str*, default: "127.0.0.1")
- *port* – port to use (*int*, default: 50051)
- *ssl_context* – SSL context object to pass to *werkzeug.serving.run_simple* (*SSLContext*, default: None).
- *threaded* – whether to use a separate thread to process each request (*bool*, default: True).
- *security_token* – security token for authenticating clients (*str*, default: None). A new token will be generated and printed if this parameter is unspecified.

The *ssl_context* parameter is explained in more detail in Werkzeug's SSL serving documentation.

Secure connections can also be set up using OpenSSH port forwarding with a command such as:

```
$ ssh -NTf -L 50051:127.0.0.1:50051 <system-with-rpc-server>
```

Minor note: using an IP address instead of a hostname for the *address* parameter should increase performance somewhat, e.g. "127.0.0.1" instead of "localhost".

Warning: Do **not** send requests to the server from the main engine thread; thread deadlocks *will* occur if you do this because the main thread cannot call timer functions and wait for a response at the same time. The RPC framework was designed to be used from *remote* processes.

Requests will not be processed if the engine is not connected and processing speech.

add_method(*method*, *name=None*)

Add an RPC method to the server.

Restarting the server is *not* required for the new method to be available.

This can be used to override method implementations if that is desirable.

This method can also be used as a decorator.

Parameters

- **method** (*callable*) – the implementation of the RPC method to add.
- **name** (*str*) – optional name of the RPC method to add. If this is None, then `method.__name__` will be used instead.

remove_method(*name*)

Remove an RPC method from the server. This will not raise an error if the method does not exist.

Restarting the server is *not* required for the change to take effect.

Parameters **name** (*str*) – the name of the RPC method to remove.

send_request(*method*, *params*, *id=0*)

Utility method to send a JSON-RPC request to the server. This will block the current thread until a response is received.

This method is mostly used for testing. If called from the engine's main thread, a deadlock *will* occur.

This will raise an error if the request fails with an error or if the server is unreachable.

The server's security token will automatically be added to the `params` list/dictionary.

Parameters

- **method** (*str*) – name of the RPC method to call.
- **params** (*list | dict*) – parameters of the RPC method to call.
- **id** (*int*) – ID of the JSON-RPC request (default: 0).

Returns JSON-RPC response
Return type dict
Raises RuntimeError

start()
Start the server.

This method is non-blocking, the RPC server will run on a separate daemon thread. This way it is not necessary to call *stop()* before the main thread terminates.

stop()
Stop the server if it is running.

url
The URL to send JSON-RPC requests to.

3.11.2 RPC methods

For RPC methods to work they must be added to the server with `add_method()`. For example:

```
from dragonfly.engines import get_engine
from dragonfly.rpc import RPCServer

# Initialise and start the server.
server = RPCServer()
server.start()

# Add the RPC method via decoration.
@server.add_method
def get_engine_language():
    return get_engine().language

# add_method() can also be used normally.
server.add_method(get_engine_language)
```

Sending requests

Requests can be sent to the server using, the `send_rpc_request()` function from Python:

```
send_rpc_request(
    server.url, method="get_engine_language",
    params=[server.security_token], id=0
)
```

Other tools such as curl can also be used.

```
Using positional arguments:
$ curl --data-binary '{"jsonrpc":"2.0","id": "0","method": "speak","params": ["hello␣
↪world", "<security-token>"]}' -H 'content-type:text/json;' http://127.0.0.1:50051

Using key word arguments:
$ curl --data-binary '{"jsonrpc":"2.0","id": "0","method": "speak","params": {"text":
↪"hello world", "security_token": "<security-token>"}}' -H 'content-type:text/json;'
↪http://127.0.0.1:50051
```

(continues on next page)

(continued from previous page)

Built-in RPC methods

get_engine_language()
Get the current engine's language.
Returns language code
Return type str

get_recognition_history()
Get the recognition history if an observer is registered.

The *register_history()* method **must** be called to register the observer first.
Returns history
Return type list

is_in_speech()
Whether the user is currently speaking.

The *register_history()* method **must** be called to register the observer first.
Return type bool

list_grammars()
Get a list of grammars loaded into the current engine.

This includes grammar rules and attributes.

mimic(*words*)
Mimic the given *words*.
Parameters **words** – string or list of words to mimic
Returns whether the mimic was a success
Return type bool

register_history(*length=10*, *record_failures=False*)
Register an internal recognition observer.
Parameters
- **length** (*int*) – length to initialize the `RecognitionHistory` instance with (default `10`).
- **record_failures** (*bool*) – whether to record recognition failures (default `False`).

speak(*text*)
Speak the given *text* using text-to-speech using `engine.speak()`.
Parameters **text** (*str*) – text to speak using text-to-speech

unregister_history()
Unregister the internal recognition observer.

3.11.3 RPC utility functions

send_rpc_request(*url*, *method*, *params*, *id=0*)
Utility function to send a JSON-RPC request to a server.

This will raise an error if the request fails with an error or if the server is unreachable.
Parameters
- **url** (*str*) – the URL to send the JSON-RPC request to.
- **method** (*str*) – name of the RPC method to call.

- **params** (*list*) – parameters of the RPC method to call.
- **id** (*int*) – ID of the JSON-RPC request (default: 0).

Returns JSON-RPC response
Return type dict
Raises RuntimeError

3.12 Configuration toolkit

3.12.1 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 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.
```

Example command modules

The configuration toolkit is utilized in a number of command modules in the *t4ngo/dragonfly-modules* repository, available on GitHub. See *Related Resources: Command modules*.

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.

3.13 Continous Command Recognition (CCR)

One of dragonfly's most powerful features is continuous command recognition (CCR), that is commands that can be spoken together without pausing. This is done through use of a `dragonfly.grammar.element_basic.Repetition` rule element. There is a mini-demo of continuous command recognition on YouTube. There are also a few projects using dragonfly which make writing CCR rules easier:

- Caster (documentation) — Caster has out-of-the-box support for CCR, with commands for typing alphanumeric and common punctuation characters as well as some useful navigation commands. You can also add custom CCR rules using the `MergeRule` class. There is more information on how to do that in the documentation.
- dragonfluid (documentation) — dragonfluid has drop-in replacements for dragonfly's element, grammar and rule classes that support CCR. The documentation has more information on this. You should be able to use *dragonfluid* with *dragonfly2* by installing it without dependencies:

```
pip install --no-deps dragonfluid
```

- dragonfly-scripts — The dragonfly-scripts project uses a `SeriesMappingRule` class in many grammar files to allow commands in a `MappingRule` to be recognised continuously without pauses.

3.14 Project

Contents:

3.14.1 Contributor Code of Conduct

As contributors and maintainers of the Dragonfly project, we pledge to respect everyone who contributes by posting issues, updating documentation, submitting pull requests, providing feedback in comments, and any other activities.

Communication through any of Dragonfly's channels (GitHub, Discord, Gitter, mailing lists, etc.) must be constructive and never resort to personal attacks, trolling, public or private harassment, insults, or other unprofessional conduct.

We promise to extend courtesy and respect to everyone involved in this project regardless of gender, gender identity, sexual orientation, disability, age, race, ethnicity, religion, or level of experience. We expect anyone contributing to the Dragonfly project to do the same.

If any member of the community violates this code of conduct, the maintainers of the Dragonfly project may take action, removing issues, comments, and PRs or blocking accounts as deemed appropriate.

If you are subject to or witness unacceptable behavior, or have any other concerns, please email the project maintainer at Danesprite@posteo.net.

This *Contributor Code of Conduct* has been adapted from Angular's Contributor Code of Conduct (Version 0.3b-angular).

3.14.2 Code style

Dragonfly's code base generally follows the PEP 8 Style Guide for Python Code. There are some parts of Dragonfly that don't follow that guide precisely, which is okay; it is only a guideline and blind adherence to it can actually be bad sometimes (see PEP 8 A Foolish Consistency is the Hobgoblin of Little Minds).

When making code submissions for Dragonfly, try to generally stick to the PEP 8 style guide.

Dragonfly's source files have historically had a maximum line length of 76 characters, although up to 85 characters is fine (PEP 8). The exception to this rule is where a long URL is used in a comment or string. Long URLs should usually be on separate lines.

3.14.3 Documentation style

Most docstrings in Dragonfly generally follow the Sphinx docstring format. A different style is used in some places, mostly for constructor arguments:

```
class Grammar(object):
  """
      Grammar class for managing a set of rules.

      Constructor arguments:
       - *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.

  """
```

The Sphinx docstring format is preferred, although the above format is also acceptable. The important thing is that the documentation is readable in the source file and in the various generated formats (e.g. HTML, PDF, man page).

Notice that the docstring's content is also indented. This is done in a few places and is also acceptable.

3.14.4 Contributing

There are many ways to contribute to dragonfly and the project would certainly benefit from more contributions.

Reporting bugs/other issues

If you come across bugs or other issues with dragonfly, you can open a new issue in the issue tracker.

If the issue is a bug, make sure you describe what you expected to happen as well as what actually happened. Include a full traceback if there was an exception.

Submitting patches/pull requests

If you have changes that can resolve an issue in the issue tracker, you can create a pull request to merge your changes with the master branch.

Documentation changes

There are parts of dragonfly that are undocumented as well as, undoubtedly, documented functionality which is poorly explained, out of date or incorrect. If you notice problems with the documentation, you can open a new issue in the issue tracker.

Dragonfly's documentation is written in the reStructuredText format. ReStructuredText is similar to the Markdown format. If you are unfamiliar with the format, the reStructuredText primer might be a good starting point.

The Sphinx documentation engine and Read the Docs are used to generate documentation from the *.txt* files in the *documentation/* folder. Docstrings in the source code are included in a semi-automatic way through use of the sphinx.ext.autodoc extension.

To build the documentation locally, install Sphinx and any other documentation requirements:

```
$ cd documentation
$ pip install -r requirements.txt
```

Then run the following command on Windows to build the documentation:

```
$ make.bat html
```

Or use the Makefile on other systems:

```
$ make html
```

If there were no errors during the build process, open the *build/html/index.html* file in a web browser. Make changes, rebuild the documentation and reload the doc page(s) in your browser as you go.

Improving spoken language support

Dragonfly supports using various languages with speech recognition engines. Language-specific code is located in sub-packages under *dragonfly.language* and loaded automatically when *dragonfly.language* is imported.

English is fully supported with mappings from English words to characters, integers (e.g. for `IntegerRef`) and time/date formats.

Other languages such as German and Dutch only have mappings for using `IntegerRef` (and similar) elements.

Additional SR engine backends

Dragonfly supports using various speech recognition engines: Dragon NaturallySpeaking (DNS), Windows Speech Recognition (WSR) and CMU Pocket Sphinx.

Adding an additional engine backend is a significant undertaking. Engine implementations should be placed in a sub-package of *dragonfly.engines*, e.g. *backend_natlink*. The engine's sub-package should define an engine class implementing the `EngineBase` class and a `get_engine` function to be used in the *dragonfly/engines/__init__.py* file.

Examining the source code under *dragonfly/engines/backend_text/* may be a good place to start, as it is currently the simplest engine implementation.

Implementations should define and use sub-classes of the `DictationContainer`, `RecObsManagerBase` and `TimerManagerBase` base classes from *dragonfly.engines.base*. These are for dictation result containers, managing recognition observers and managing multiplexing timers respectively.

A compiler class to translate dragonfly grammars into a format that the SR engine accepts will probably also be required. Engine backends other than the "text" engine have compiler classes to look at as examples.

If there are additional engine dependencies, they should be placed into the *setup.py* file in the `extras_require` dictionary. For example:

```
extras_require={
    ...,
    "EngineX": ["engine_dependency >= X.Y.Z"],
},
```

This allows the engine's dependencies to be installed using something like:

```
$ pip install dragonfly2[EngineX]
```

In addition to the engine implementation, each engine should define its own test suite in the *dragonfly/test/suites.py* file. For example:

```
# Define test files to run for the new engine, including common ones.
x_names = [
    # Assume that "test_engine_x.py" exists. Including the '.py' file
    # extension is not necessary.
    "test_engine_x",

    # Include the tests for English integer content.
    "test_language_en_number",
] + common_names

# Exclude one or more common names if the new engine doesn't [yet]
# support certain dragonfly functionality. Also display a warning.
x_names.remove("test_timer")
_log.warning("Excluding 'test_timer' for engine X because multiplexing "
             "timers are not supported (yet).")

# Build a test suite for the engine.
x_suite = build_suite(EngineTestSuite("<engine_name>"), x_names)
```

The test suite should be runnable using the following (or similar) command:

```
$ python setup.py test --test-suite=dragonfly.test.suites.x_suite
```

The *suites.py* file should be able to build each engine's test suite **without** engine-specific dependencies being available, such as *natlink*. You should be able to test this by running the default test suite in a virtual environment:

```
$ python setup.py test
```

The above command should run successfully for Python versions 2.7 and 3.x.

The new engine and its tests should be documented in the engines and test suites documentation pages respectively. If the engine implementation doesn't work with some of dragonfly's functionality, such as `Dictation` elements, it should be mentioned somewhere in the engine's documentation.

3.14.5 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

3.14.6 Release versioning

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

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

- *Major* – version when you make incompatible API changes.
- *Minor* – version when you add functionality in a backwards-compatible manner.
- *Patch* – version when you make backwards-compatible bug fixes.

Please see semver.org for guidelines on version incrementation and other topics.

3.14.7 Release process

Preparation

1. Version number and release branch
    1. Determine the appropriate version number for this release, according to *Release versioning*.
    2. Update the `version.txt` file to contain the new version number.
2. Tickets
    1. Update ticket status for this release, where relevant: https://github.com/dictation-toolbox/dragonfly/issues
3. Release files
    1. Verify that `CHANGELOG.rst` includes the change log for this release.
    2. Verify that `AUTHORS.txt` is up to date with recent contributors.

3. Verify that `setup.py` specifies all required dependencies, including their versions, e.g. with the `install_requires` and `test_requires` parameters.
4. Verify that `documentation/requirements.txt` specifies all required dependencies for building the documentation.
5. Verify that `MANIFEST.in` includes all necessary data files.

4. Draft announcement
    1. 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 (test.pypi.org)
    1. Upload distributions to PyPI test server
    2. Test installation from PyPI test server
    3. Verify package is displayed correctly on PyPI test server
5. Tag release
    1. Tag git revision as `X.Y.Z`
    2. Push to GitHub

Release

1. Upload to GitHub
    1. Upload distributions to GitHub: https://github.com/dictation-toolbox/dragonfly/releases
2. Trigger building of documentation on Read the Docs
    1. Check whether documentation was built automatically, and if not trigger it: https://readthedocs.org/builds/dragonfly2/
3. Upload to PyPI server
    1. Upload distributions to PyPI server
    2. Test installation from PyPI server
    3. Verify package is displayed correctly on PyPI server

Post-release

1. Announce release
    1. Website
    2. Mailing list
    3. Gitter channel

3.15 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:

3.15.1 Doctests for the fundamental element classes

Test fixture initialization:

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

Alternative element class

Usage:

```
>>> alt = Alternative([Literal("hello"), Literal("goodbye")])
>>> test_alt = ElementTester(alt)
>>> test_alt.recognize("hello")
u'hello'
>>> test_alt.recognize("goodbye")
u'goodbye'
>>> test_alt.recognize("hi")
RecognitionFailure
```

Literal element class

Usage:

```
>>> literal = Literal("hello")
>>> test_literal = ElementTester(literal)
>>> test_literal.recognize("hello")
u'hello'
>>> test_literal.recognize("goodbye")
RecognitionFailure
```

Quoted words usage:

```
>>> # Quoted words are not joined in the 'words' property.
>>> # Also, double quotes are not present.
>>> literal = Literal('this is a "quoted string" example')
>>> literal.words
['this', 'is', 'a', 'quoted', 'string', 'example']
>>> # The 'words_ext' property shows quoted words as single items.
>>> # Double quotes are also not present.
>>> literal.words_ext
```

(continues on next page)

(continued from previous page)

```
['this', 'is', 'a', 'quoted string', 'example']
>>> # Rules with quoted words can be recognized.
>>> test_literal = ElementTester(literal)
>>> test_literal.recognize('this is a quoted string example')
u'this is a quoted string example'
```

Incomplete quotes:

```
>>> # Incomplete quotes are left in.
>>> literal = Literal('"quote')
>>> literal.words
['"quote']
>>> literal.words_ext
['"quote']
>>> literal = Literal('a "quoted string" example plus "quote')
>>> literal.words
['a', 'quoted', 'string', 'example', 'plus', '"quote']
>>> literal.words_ext
['a', 'quoted string', 'example', 'plus', '"quote']
```

Non-default quote constructor arguments:

```
>>> literal = Literal(u'this is a «quoted string» example',
...                   quote_start_str=u'«', quote_end_str=u'»')
>>> literal.words
['this', 'is', 'a', 'quoted', 'string', 'example']
>>> literal.words_ext
['this', 'is', 'a', 'quoted string', 'example']
>>> test_literal = ElementTester(literal)
>>> test_literal.recognize('this is a quoted string example')
u'this is a quoted string example'

>>> # Quotes are left in if specified.
>>> literal = Literal('"quoted string"', strip_quote_strs=False)
>>> literal.words
['"quoted', 'string"']
>>> literal.words_ext
['"quoted string"']
```

Optional element class

Usage:

```
>>> seq = Sequence([Literal("hello"), Optional(Literal("there"))])
>>> test_seq = ElementTester(seq)
>>> # Optional parts of the sequence can be left out.
>>> test_seq.recognize("hello")
[u'hello', None]
>>> test_seq.recognize("hello there")
[u'hello', u'there']
>>> test_seq.recognize("goodbye")
RecognitionFailure
```

Sequence element class

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'))
```

Repetition element class

Basic usage:

```
>>> # Repetition is given a dragonfly element, in this case a Sequence.
>>> seq = Sequence([Literal("hello"), Literal("world")])
>>> # Specify min and max values to allow more than one repetition.
>>> rep = Repetition(seq, min=1, max=16, optimize=False)
>>> test_rep = ElementTester(rep)
>>> test_rep.recognize("hello world")
[[u'hello', u'world']]
>>> test_rep.recognize("hello world hello world")
[[u'hello', u'world'], [u'hello', u'world']]
>>> # Incomplete recognitions result in recognition failure.
>>> test_rep.recognize("hello universe")
RecognitionFailure
>>> test_rep.recognize("hello world hello universe")
RecognitionFailure
>>> # Too many recognitions also result in recognition failure.
>>> test_rep.recognize(" ".join(["hello world"] * 17))
RecognitionFailure
>>> # Using the 'optimize' argument:
rep = Repetition(seq, min=1, max=16, optimize=True)
>>> test_rep = ElementTester(rep)
>>> test_rep.recognize("hello world")
[[u'hello', u'world']]
>>> test_rep.recognize("hello world hello world")
[[u'hello', u'world'], [u'hello', u'world']]
```

Exact number of repetitions:

```
>>> seq = Sequence([Literal("hello"), Literal("world")])
>>> rep = Repetition(seq, min=3, max=None, optimize=False)
>>> test_rep = ElementTester(rep)
>>> test_rep.recognize("hello world")
RecognitionFailure
>>> test_rep.recognize("hello world hello world")
RecognitionFailure
>>> test_rep.recognize("hello world hello world hello world")
[[u'hello', u'world'], [u'hello', u'world'], [u'hello', u'world']]
>>> test_rep.recognize("hello world hello world hello world hello world")
RecognitionFailure
```

min must be less than max:

```
>>> rep = Repetition(Literal("hello"), min=3, max=3, optimize=False)
Traceback (most recent call last):
  ...
AssertionError: min must be less than max
```

Modifier element class

Basic usage:

```
>>> # Repetition is given a dragonfly element, in this case an IntegerRef.
>>> i = IntegerRef("n", 1, 10)
>>> # Specify min and max values to allow more than one repetition.
>>> rep = Repetition(i, min=1, max=16, optimize=False)
>>> # Modify the element to format the output
>>> mod = Modifier(rep, lambda rep: ", ".join(map(str, rep)))
>>> test_rep = ElementTester(mod)
>>> test_rep.recognize("one two three four")
'1, 2, 3, 4'
```

RuleRef element class

Basic usage:

```
>>> # Define a simple private CompoundRule and reference it.
>>> greet = CompoundRule(name="greet", spec="greetings", exported=False)
>>> ref = RuleRef(rule=greet, name="greet")
>>> test_ref = ElementTester(ref)
>>> test_ref.recognize("greetings")
u'greetings'
>>> test_ref.recognize("hello")
RecognitionFailure
```

Empty element class

Usage:

```
>>> empty = Empty()
>>> test_empty = ElementTester(empty)
>>> test_empty.recognize("hello")
```

(continues on next page)

(continued from previous page)

```
RecognitionFailure
>>> empty_seq = Sequence([Literal("hello"), Empty(),
...                       Literal("goodbye")])
>>> test_empty = ElementTester(empty_seq)
>>> test_empty.recognize("hello goodbye")
[u'hello', True, u'goodbye']
>>> test_empty.recognize("hello empty goodbye")
RecognitionFailure
```

Impossible element class

Usage:

```
>>> impossible = Impossible()
>>> test_impossible = ElementTester(impossible)
>>> test_impossible.recognize("hello")
RecognitionFailure
>>> test_impossible.recognize("")
RecognitionFailure
>>> impossible_seq = Sequence([Literal("hello"), Impossible(),
...                       Literal("goodbye")])
>>> test_impossible = ElementTester(impossible_seq)
>>> test_impossible.recognize("hello goodbye")
RecognitionFailure
>>> test_impossible.recognize("hello empty goodbye")
RecognitionFailure
```

3.15.2 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
```

3.15.3 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
```

(continues on next page)

(continued from previous page)

```
>>> 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']
```

(continues on next page)

(continued from previous page)

```
>>> 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.
```

List add/remove restrictions

Setup test tooling:

```
>>> list_fruit = List("list_fruit", ["apple", "banana"])
>>> 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()
```

Lists cannot be added while the grammar is loaded:

```
>>> list_other = List("list_other", ["other"])
>>> tester_fruit.add_list(list_other)  # Fails.
Traceback (most recent call last):
    ...
GrammarError: Cannot add list while loaded.
```

Lists cannot be removed while the grammar is loaded:

```
>>> tester_fruit.remove_list(list_fruit)  # Fails.
Traceback (most recent call last):
    ...
GrammarError: Cannot remove list while loaded.
```

Tear down test tooling:

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

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')
```

List context manager

List operations can be performed optimally via Python's `with` statement:

```
>>> list_fruit = List("list_fruit")
>>> element = Sequence([Literal("item"), ListRef("list_fruit_ref", list_fruit)])
>>> tester_fruit = ElementTester(element)
>>> tester_fruit.load()
>>> with list_fruit:
...     list_fruit.append("apple")
...     list_fruit.append("banana")
...
```

(continues on next page)

(continued from previous page)

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

Tear down test tooling:

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

3.15.4 RecognitionObserver base class

If you're looking for the class and function reference for recognition observers, you want *this page*.

Note: `RecognitionObserver` instances can be used for both the DNS and the WSR backend engines. However, WSR will sometimes call observer methods multiple times, so be careful using observers with it.

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()")
...     def on_end(self):
...         print("on_end()")
...     def on_post_recognition(self, words, rule):
...         print("on_post_recognition(): %r from %r" % (words, rule))
...
>>> 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')
on_end()
on_post_recognition(): (u'hello', u'world') from _ElementTestRule(rule)
u'hello world'
>>> test_lit.recognize("hello universe")
on_begin()
on_failure()
```

(continues on next page)

(continued from previous page)

```
on_end()
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)
```

(continues on next page)

(continued from previous page)

```
>>> test_lit.recognize("hello world")
u'hello world'
```

3.15.5 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
```

(continues on next page)

(continued from previous page)

```
>>> history
[(u'five',)]
```

3.15.6 Recognition callback functions

Register recognition callback functions:

```
>>> # Define and register functions for each recognition state event.
>>> def on_begin():
...     print("on_begin()")
...
>>> def on_recognition(words):
...     print("on_recognition(): %s" % (words,))
...
>>> def on_failure():
...     print("on_failure()")
...
>>> def on_end():
...     print("on_end()")
...
>>> def on_post_recognition(words, rule):
...     print("on_post_recognition(): %r from %r" % (words, rule))
...
>>> on_begin_obs = register_beginning_callback(on_begin)
>>> on_success_obs = register_recognition_callback(on_recognition)
>>> on_failure_obs = register_failure_callback(on_failure)
>>> on_end_obs = register_ending_callback(on_end)
>>> on_post_recognition_obs = register_post_recognition_callback(on_post_recognition)
```

Callback functions are called during recognitions:

```
>>> test_lit = ElementTester(Literal("hello world"))
>>> test_lit.recognize("hello world")
on_begin()
on_recognition(): (u'hello', u'world')
on_end()
on_post_recognition(): (u'hello', u'world') from _ElementTestRule(rule)
u'hello world'
>>> test_lit.recognize("hello universe")
on_begin()
on_failure()
on_end()
RecognitionFailure
```

Callback functions are unregistered using the observer objects returned by each function:

```
>>> on_begin_obs.unregister()
>>> on_success_obs.unregister()
>>> on_failure_obs.unregister()
>>> on_end_obs.unregister()
>>> on_post_recognition_obs.unregister()
```

3.15.7 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, Function
>>> class PrintAction(ActionBase):
...     def __init__(self, name):
...         ActionBase.__init__(self)
...         self._name = name
...     def execute(self, data=None):
...         if data:
...             # Print a sorted representation of the data dict.
...             sorted_data = "{%s}" % ", ".join([
...                 "%r: %r" % (key, value)
...                 for key, value in sorted(data.items())
...             ])
...             print("executing %r %s" % (self._name, sorted_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'
```

(continues on next page)

(continued from previous page)

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

Concatenating failing actions

Series execution normally stops if an action in the series fails:

```
>>> bad_function = Function(lambda: 1/0)
>>> # This will produce log messages about a ZeroDivisionError.
>>> failing_series = (a + bad_function + b)
>>> failing_series.execute()
executing 'a'
```

Series execution will continue if 'stop_on_failures' is False:

```
>>> failing_series.stop_on_failures = False
>>> failing_series.execute()
executing 'a'
executing 'b'
```

Or if using the '|' or '|=' operators:

```
>>> (a | bad_function | b).execute()
executing 'a'
executing 'b'
>>> unsafe_action = a | b
>>> unsafe_action |= bad_function
>>> unsafe_action |= a
>>> unsafe_action.execute()
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.
```

(continues on next page)

(continued from previous page)

```
>>> (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' {'bar': 'later', 'foo': 2}
```

(continues on next page)

(continued from previous page)

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

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

Function action test suite

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.

Using the Function action

Simple usage::

```
>>> from dragonfly import Function
>>> def func(count):
...     print("count: %d" % 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: %d" % count)
...     print("flavor: %s" % 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
```

Usage with the `remap_data` argument:

```
>>> def func(x, y, z):
...     print("x: %d" % x)
...     print("y: %d" % y)
```

(continues on next page)

(continued from previous page)

```
...     print("z: %d" % z)
...
>>> # The Function object can optionally be given a second dictionary
>>> # argument to use extras with different names. It should be
>>> # compatible with the 'defaults' parameter:
>>> action = Function(func, dict(n="x", m="y"), z=4)
>>> action.execute({"n": 2, "m": 3})
x: 2
y: 3
z: 4
True
```

3.15.8 Word formatting for DNS v10 and lower

The `dragonfly.engines.backend_natlink.dictation_format.WordParserDns10` class converts DNS v10 (and lower) recognition results to `dragonfly.engines.backend_natlink.dictation_format.Word` objects which contain written and spoken forms together with formatting information. For example:

```
>>> from dragonfly.engines.backend_natlink.dictation_format import WordParserDns10
>>> # Make sure the engine is connected before using WordParserDns10.
>>> from dragonfly import get_engine
>>> engine = get_engine("natlink")
>>> engine.connect()
>>> parser_dns10 = WordParserDns10()
>>> print(parser_dns10.parse_input("hello"))
Word('hello')
>>> print(parser_dns10.parse_input(".\\full-stop"))
Word('.', 'full-stop', no_space_before, two_spaces_after, cap_next, not_after_period)
```

Nonexistent words can be parsed, but don't have any formatting info:

```
>>> print(parser_dns10.parse_input("nonexistent-word"))
Word('nonexistent-word')
```

This helper function allows for a compact notation of input string tests:

```
>>> from dragonfly.engines.backend_natlink.dictation_format import WordFormatter
>>> def format_dictation_dns10(input):
...     if isinstance(input, basestring):
...         input = input.split()
...     return WordFormatter(parser=WordParserDns10()).format_dictation(input)
>>> format_dictation_dns10("hello world")
u'hello world'
```

The following tests cover in-line capitalization:

```
>>> format_dictation_dns10("\\All-Caps hello world")
u'HELLO world'
>>> format_dictation_dns10("\\Caps-On hello world")
u'Hello World'
>>> format_dictation_dns10("\\Caps-On hello of the world")
u'Hello of the World'
>>> format_dictation_dns10("hello \\Caps-On of the world")
```

(continues on next page)

(continued from previous page)

```
u'hello Of the World'
>>> format_dictation_dns10("\\Caps-On hello world \\Caps-Off goodbye universe")
u'Hello World goodbye universe'
>>> format_dictation_dns10("hello \\All-Caps-On world goodbye \\All-Caps-Off universe
↪")
u'hello WORLD GOODBYE universe'
>>> format_dictation_dns10("hello \\All-Caps-On world \\Caps-On goodbye universe")
u'hello WORLD Goodbye Universe'
>>> format_dictation_dns10("hello \\All-Caps-On world goodbye \\Caps-Off universe")
u'hello WORLD GOODBYE universe'
```

The following tests cover in-line spacing:

```
>>> format_dictation_dns10("\\No-Space hello world")
u'hello world'
>>> format_dictation_dns10("hello \\No-Space world")
u'helloworld'
>>> format_dictation_dns10("\\No-Space-On hello world")
u'helloworld'
>>> format_dictation_dns10("\\No-Space-On hello world goodbye \\No-Space-Off universe
↪")
u'helloworldgoodbye universe'
>>> format_dictation_dns10("\\No-Space-On hello world \\No-Space-Off goodbye universe
↪")
u'helloworld goodbye universe'
>>> format_dictation_dns10("\\No-Space-On hello world \\space-bar goodbye universe")
u'helloworld goodbyeuniverse'

>>> # The following are different from some DNS installations!
>>> format_dictation_dns10("hello \\No-Space-On world goodbye universe")
u'helloworldgoodbyeuniverse'
>>> format_dictation_dns10("hello \\No-Space-On world \\space-bar goodbye universe")
u'helloworld goodbyeuniverse'
```

The following tests cover punctuation and other symbols that influence spacing and capitalization of surrounding words:

```
>>> format_dictation_dns10("hello \\New-Line world")
u'hello\nworld'
>>> format_dictation_dns10("hello \\New-Paragraph world")
u'hello\n\nWorld'
>>> format_dictation_dns10("hello .\\full-stop world")
u'hello. World'
>>> format_dictation_dns10("hello ,\\comma world")
u'hello, world'
>>> format_dictation_dns10("hello .\\full-stop \\New-Line world")
u'hello.\nWorld'
>>> format_dictation_dns10("hello -\\hyphen world")
u'hello-world'
>>> format_dictation_dns10("hello (\\left-paren world")
u'hello (world'
>>> format_dictation_dns10("hello )\\right-paren world")
u'hello) world'
>>> format_dictation_dns10("hello \\\\backslash world")
u'hello\\world'
```

The "." character at the end of certain words is "swallowed" by following words that start with that same character:

```
>>> format_dictation_dns10(["hello", "etc.\\et cetera", "world"])
u'hello etc. world'
>>> format_dictation_dns10(["hello", "etc.\\et cetera", ".\\full-stop", "world"])
u'hello etc. World'
>>> format_dictation_dns10(["hello", "etc.\\et cetera", "...\\ellipsis", "world"])
u'hello etc... world'
>>> # The following are different from some DNS installations!
>>> format_dictation_dns10("hello .\\full-stop ...\\ellipsis world")
u'hello... world'
>>> format_dictation_dns10("hello ...\\ellipsis .\\full-stop world")
u'hello... World'
```

Letters and numbers spoken in line within dictation allow efficient spelling of for example words not present in the dictionary:

```
>>> format_dictation_dns10(["a\\alpha", "b\\bravo",
...                         "c\\charlie", "d\\delta",
...                         "e\\echo",
...                         "x\\xray", "z\\zulu",
...                         "y\\yankee"])
u'abcdexzy'
```

Words may contain spaces in their written and/or spoken forms. For example a custom word added by the user might have the following form with a space in both spoken and written forms:

```
>>> format_dictation_dns10(["custom written\\custom spoken"])
u'custom written'
>>> format_dictation_dns10(["custom written\\custom spoken",
...                         "\\All-Caps",
...                         "custom written\\custom spoken",
...                         "\\Cap",
...                         "custom written\\custom spoken"])
u'custom written CUSTOM written Custom written'
>>> format_dictation_dns10(["custom written\\custom spoken",
...                         "\\Caps-On",
...                         "custom written\\custom spoken",
...                         "\\All-Caps-On",
...                         "custom written\\custom spoken",
...                         "\\All-Caps-Off",
...                         "custom written\\custom spoken"])
u'custom written Custom Written CUSTOM WRITTEN custom written'
```

Certain words, such as numbers, are not formatted according to the same rules as "normal" words, i.e. those which specified written and spoken forms and formatting info:

```
#    >>> format_dictation_dns10("one two three")
#    u'123'
```

3.15.9 Word formatting for DNS v11 and higher

The `dragonfly.engines.backend_natlink.dictation_format.WordParserDns11` class converts DNS v11 (and higher) recognition results to `dragonfly.engines.backend_natlink.dictation_format.Word` objects which contain written and spoken forms together with formatting information. For example:

```
>>> from dragonfly.engines.backend_natlink.dictation_format import WordParserDns11
>>> parser_dns11 = WordParserDns11()
>>> print(parser_dns11.parse_input("hello"))
Word('hello')
>>> print(parser_dns11.parse_input(".\\period\\full-stop"))
Word('.', 'full-stop', no_space_before, two_spaces_after, cap_next, not_after_period)
```

Nonexistent words can be parsed, but don't have any formatting info:

```
>>> print(parser_dns11.parse_input("nonexistent-word"))
Word('nonexistent-word')
```

This helper function allows for a compact notation of input string tests:

```
>>> from dragonfly.engines.backend_natlink.dictation_format import WordFormatter
>>> from six import string_types
>>> def format_dictation_dns11(input):
...     if isinstance(input, string_types):
...         input = input.split()
...     return WordFormatter(parser=WordParserDns11()).format_dictation(input)
>>> format_dictation_dns11("hello world")
u'hello world'
```

The following tests cover in-line capitalization:

```
>>> format_dictation_dns11("\\all-caps\\All-Caps hello world")
u'HELLO world'
>>> format_dictation_dns11("\\caps-on\\Caps-On hello world")
u'Hello World'
>>> format_dictation_dns11("\\caps-on\\Caps-On hello of the world")
u'Hello Of The World'
>>> # Note: output above should probably be u'Hello of the World'
>>> format_dictation_dns11("hello \\caps-on\\Caps-On of the world")
u'hello Of The World'
>>> # Note: output above should probably be u'hello Of the World'
>>> format_dictation_dns11("\\caps-on\\Caps-On hello world \\caps-off\\Caps-Off␣
↪goodbye universe")
u'Hello World goodbye universe'
>>> format_dictation_dns11("hello \\all-caps-on\\All-Caps-On world goodbye \\all-caps-
↪off\\All-Caps-Off universe")
u'hello WORLD GOODBYE universe'
>>> format_dictation_dns11("hello \\all-caps-on\\All-Caps-On world \\caps-on\\Caps-On␣
↪goodbye universe")
u'hello WORLD Goodbye Universe'
>>> format_dictation_dns11("hello \\all-caps-on\\All-Caps-On world goodbye \\caps-
↪off\\Caps-Off universe")
u'hello WORLD GOODBYE universe'
```

The following tests cover in-line spacing:

```
>>> format_dictation_dns11("\\no-space\\No-Space hello world")
u'hello world'
>>> format_dictation_dns11("hello \\no-space\\No-Space world")
u'helloworld'
>>> format_dictation_dns11("\\no-space-on\\No-Space-On hello world")
u'helloworld'
>>> format_dictation_dns11("\\no-space-on\\No-Space-On hello world goodbye \\no-space-
↪off\\No-Space-Off universe")
```

(continues on next page)

(continued from previous page)

```
u'helloworldgoodbye universe'
>>> format_dictation_dns11("\\no-space-on\\No-Space-On hello world \\no-space-off\\No-
↪Space-Off goodbye universe")
u'helloworld goodbye universe'
>>> format_dictation_dns11("\\no-space-on\\No-Space-On hello world \\space-bar\\space-
↪bar goodbye universe")
u'helloworld goodbyeuniverse'

>>> # The following are different from some DNS installations!
>>> format_dictation_dns11("hello \\no-space-on\\No-Space-On world goodbye universe")
u'helloworldgoodbyeuniverse'
>>> format_dictation_dns11("hello \\no-space-on\\No-Space-On world \\space-bar\\space-
↪bar goodbye universe")
u'helloworld goodbyeuniverse'
```

The following tests cover punctuation and other symbols that influence spacing and capitalization of surrounding words:

```
>>> format_dictation_dns11("hello \\new-line\\New-Line world")
u'hello\nworld'
>>> format_dictation_dns11("hello \\new-paragraph\\New-Paragraph world")
u'hello\n\nWorld'
>>> format_dictation_dns11("hello .\\period\\full-stop world")
u'hello. World'
>>> format_dictation_dns11("hello ,\\comma\\comma world")
u'hello, world'
>>> format_dictation_dns11("hello .\\period\\full-stop \\new-line\\New-Line world")
u'hello.\nWorld'
>>> format_dictation_dns11("hello -\\hyphen\\hyphen world")
u'hello-world'
>>> format_dictation_dns11("hello (\\left-paren\\left-paren world")
u'hello (world'
>>> format_dictation_dns11("hello )\\right-paren\\right-paren world")
u'hello) world'
>>> format_dictation_dns11("hello \\\\hyphen\\backslash world")
u'hello\\world'
```

The "." character at the end of certain words is "swallowed" by following words that start with that same character:

```
>>> format_dictation_dns11(["hello", "etc.\\\\et cetera", "world"])
u'hello etc. world'
>>> format_dictation_dns11(["hello", "etc.\\\\et cetera", ".\\period\\full-stop",
↪"world"])
u'hello etc. World'
>>> format_dictation_dns11(["hello", "etc.\\\\et cetera", "...\\ellipsis\\ellipsis",
↪"world"])
u'hello etc... world'
>>> # The following are different from some DNS installations!
>>> format_dictation_dns11("hello .\\period\\full-stop ...\\ellipsis\\ellipsis world")
u'hello... world'
>>> format_dictation_dns11("hello ...\\ellipsis\\ellipsis .\\period\\full-stop world")
u'hello... World'
```

Letters and numbers spoken in line within dictation allow efficient spelling of for example words not present in the dictionary:

```
>>> format_dictation_dns11(["A\\letter\\alpha", "B\\letter\\bravo",
...                         "C\\letter\\Charlie", "D\\letter\\delta",
...                         "E\\letter\\echo", "F\\letter\\foxtrot",
...                         "X\\letter\\X ray", "Z\\letter\\Zulu",
...                         "Y\\letter\\Yankee"])
u'ABCDEFXZY'
>>> format_dictation_dns11("D\\letter E\\letter F\\letter")
u'DEF'
>>> format_dictation_dns11(["J\\uppercase-letter\\capital J",
...                         "O\\letter", "H\\letter", "N\\letter"])
u'JOHN'
>>> format_dictation_dns11("J\\letter\\Juliett O\\letter\\Oscar"
...                        " H\\letter\\hotel N\\letter\\November")
u'JOHN'
```

Letters spoken in line within dictation have spaces separating adjacent words:

```
>>> format_dictation_dns11(["hello", "A\\letter\\alpha",
...                         "B\\letter\\bravo", "C\\letter\\Charlie",
...                         "world"])
u'hello ABC world'
>>> format_dictation_dns11(["hello", "A\\uppercase-letter\\capital A",
...                         "B\\uppercase-letter\\capital B",
...                         "C\\uppercase-letter\\capital C", "world"])
u'hello ABC world'
```

Words may contain spaces in their written and/or spoken forms. For example a custom word added by the user might have the following form with a space in both spoken and written forms:

```
>>> format_dictation_dns11(["custom written\\\\custom spoken"])
u'custom written'
>>> format_dictation_dns11(["custom written\\\\custom spoken",
...                         "\\all-caps\\all caps",
...                         "custom written\\\\custom spoken",
...                         "\\cap\\cap",
...                         "custom written\\\\custom spoken"])
u'custom written CUSTOM written Custom written'
>>> format_dictation_dns11(["custom written\\\\custom spoken",
...                         "\\caps-on\\caps on",
...                         "custom written\\\\custom spoken",
...                         "\\all-caps-on\\all caps on",
...                         "custom written\\\\custom spoken",
...                         "\\all-caps-off\\all caps off",
...                         "custom written\\\\custom spoken"])
u'custom written Custom Written CUSTOM WRITTEN custom written'
```

Certain words, such as numbers, are not formatted according to the same rules as "normal" words, i.e. those which specified written and spoken forms and formatting info:

```
#    >>> format_dictation_dns11("one two three")
#    u'123'
#    >>> format_dictation_dns11("one\\number two three four")
#    u'1234'
#    >>> format_dictation_dns11("thirty four")
#    u'34'
```

Spoken words with multiple, context-dependent written forms, such as "point" and ".", are formatted correctly:

```
>>> format_dictation_dns11("\cap\cap what is the point of that "
...                        "?\question-mark\question-mark")
u'What is the point of that?'
>>> format_dictation_dns11(".\point\point")
u'.'
```

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

3.15.10 Running the test suite

This section contains instructions on how to run dragonfly's test suite with the SR engine backends.

Most tests were written with an English model andvocabulary in mind, so they will not pass if English words are not in the SR engine's vocabulary. The only exception to this is the text-input ("text") backend.

Using DNS

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

1. Start DNS. (And ensure that NatLink is also automatically started.)
2. Extract the Dragonfly source code in a directory `<dir>`.
3. Run the follow commands to install test requirements:

   ```
   cd <dir>
   pip install -r test_requirements.txt
   ```

4. Run the tests with the following command:

   ```
   python setup.py test --test-suite=natlink
   ```

Different tests need to be run for different DNS versions. The natlink test suite will attempt to run the correct tests for the version of DNS you are using.

These tests may fail with certain DNS versions, models and/or vocabularies. It is possible for the test suite to fail even though everything still works in command modules loaded by `natlinkmain`. This is a natlink issue.

It is not feasible to test dragonfly with all DNS configurations. Please get in touch if the tests fail for you **and** specific dragonfly functionality doesn't work in natlink command modules (i.e. `.py` files in the `MacroSystem` folder).

Using WSR

Follow the following steps to run the test suite for the WSR backend:

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 follow commands to install test requirements:

   ```
   cd <dir>
   pip install -r test_requirements.txt
   ```

5. Run the tests with the following command:

```
python setup.py test --test-suite=sapi5
```

You can also run the test suite for the in process engine class. It has no GUI for the moment. Run the tests with the following command:

```
python setup.py test --test-suite=sapi5inproc
```

Some SAPI5 engine tests will fail intermittently, particularly with the shared process engine class (`"sapi5"` / `"sapi5shared"`). There is not much we can do about this.

Using the Kaldi engine

Follow the following steps to run the test suite for the Kaldi backend:

1. Extract the Dragonfly source code in a directory `<dir>`.
2. Follow the set up and install instructions for the Kaldi engine *on this page*.
3. Run the following commands to install the Kaldi engine and test dependencies:

```
cd <dir>
pip install -e '.[kaldi]'
pip install -r test_requirements.txt
```

4. Run the tests with the following command:

```
python setup.py test --test-suite=kaldi
```

Using the CMU Pocket Sphinx engine

Follow the following steps to run the test suite for the Sphinx backend:

1. Extract the Dragonfly source code in a directory `<dir>`.
2. Run the following commands to install the Sphinx engine and test dependencies:

```
cd <dir>
pip install -e '.[sphinx]'
pip install -r test_requirements.txt
```

3. Run the tests with the following command:

```
python setup.py test --test-suite=sphinx
```

Using the text-input engine

Follow the following steps to run the test suite for the "text" backend:

1. Extract the Dragonfly source code in a directory `<dir>`.
2. Run the follow commands to install test requirements:

```
cd <dir>
pip install -r test_requirements.txt
```

3. Run the tests with the following commands:

```
cd <dir>
python setup.py test
```

This is the default test suite for dragonfly and has no additional dependencies. The `--test-suite` argument does not need to be specified in this case.

3.16 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog using the reStructuredText format instead of Markdown. This project adheres to Semantic Versioning as of version 0.7.0.

Note: this project had no release versions between 0.6.6b1 and 0.7.0. Notable changes made between these versions are documented in the commit history and will be placed under headings in this file over time.

3.16.1 0.35.0 - 2022-03-19

Added

- Add get_speaker() function for retrieving the specified or default text-to-speech (speaker) implementation.
- Add eSpeak and CMU Flite text-to-speech implementations.

Changed

- Detach text-to-speech functionality from the Natlink and SAPI 5 engine back-ends and make it available via the get_speaker() function.
- Change Kaldi, CMU Pocket Sphinx and text-input engine back-ends to use the default or specified text-to-speech implementation for engine.speak().
- Use xdotool for mouse functions on X11 if pynput is not installed.
- Remove the pynput requirement on Linux.
- Remove the requirement to upper-case mimicked dictated words with some engine back-ends.

Fixed

- Fix incorrect handling of inactive grammars by CMU Pocket Sphinx back-end.
- Remove dictation element limitation effecting most engine back-ends.

3.16.2 0.34.1 - 2022-02-24

Fixed

- Fix a problem with the behaviour of the X11Window.close() method.

3.16.3 0.34.0 - 2021-12-09

Changed

- Change Kaldi back-end to enable local-only pronunciation generation by default.
- Change accessibility sub-package to use AT-SPI if X11 is detected.
- Give X11 precedence over other platforms in initialization modules.
- Remove deprecation notices for the dragonfly.timer interface (since it works fine).

Fixed

- Fix Kaldi marking of whether retained utterance has dictation.
- Fix a word pronunciation generation bug with the Kaldi back-end (thanks @daanzu).
- Fix problems caused by relative import statements in the library by using the absolute form instead.
- Improve X11 detection by checking the DISPLAY environment variable.

3.16.4 0.33.0 - 2021-10-01

Changed

- Change command-line interface (CLI) to allow using multiple engine options arguments.
- Change load-directory CLI command to accept more than one directory.
- Remove English-language alternatives for the integer two.
- Set the Kaldi engine back-end as unavailable to old Python versions.
- Set the Kaldi engine option *auto_add_to_user_lexicon* to False by default.

Fixed

- Fix a bug that occurs when using non-matching glob patterns as command module files (CLI).
- Fix a few bugs with the CLI engine options argument.

3.16.5 0.32.1 - 2021-09-06

Fixed

- Fix a bug with Text action default pause values.
- Fix various bugs with keyboard input actions on Windows.

3.16.6 0.32.0 - 2021-06-26

Added

- Add Clipboard.synchronized_changes() context manager.
- Add Clipboard.wait_for_change() class method.

Changed

- Disable Kaldi debug logging workaround for Windows Key action bug.
- Make Clipboard class instances comparable based on content difference.

Fixed

- Add missing logger names and default levels into log.py.
- Fix bug caused by storing an internal lark grammar in a text file.
- Fix bug caused by storing the required KaldiAG version in a text file.
- Fix bug with mouse button emulation on Windows.
- Fix bug with the open/close mechanism of the Windows Clipboard class.
- Fix incorrect and messy Key action code for handling unknown keys.
- Fix incorrect handling of extended scan codes on Windows (keyboard input).

3.16.7 0.31.2 - 2021-05-15

Fixed

- Fix a number of bugs and other issues with keyboard input functionality on Windows.

3.16.8 0.31.1 - 2021-05-09

Fixed

- Fix AttributeError bug in natlink engine.py file.

3.16.9 0.31.0 - 2021-05-04

Changed

- Adjust order of preferred SR engine implementations in get_engine() to prefer Kaldi over SAPI 5 and Sphinx.
- Change clipboard toolkit to support retrieving copied files (format_hdrop) on X11.
- Change clipboard toolkit to support setting and retrieving the three X selections using the xsel program (X11).
- Change get_engine() to log an info message for initialized engines.
- Change the Choice element class to allow using list/tuple choices.

- Make various improvements to the Kaldi engine back-end and bump required Kaldi-Active-Grammar version to 2.1.0. (thanks @daanzu).

Fixed

- Fix a number of Kaldi engine back-end bugs (thanks @daanzu).
- Fix leak of file open() without close in dragonfly/config.py (thanks @wolfmanstout).
- Fix problem with the default action used by the Paste action class.
- Fix the Windows keyboard code so that letter and number keys can be pressed using the Key and Text actions, regardless of the active keyboard layout.

Removed

- Remove Google cloud speech-to-text functionality from the Kaldi engine back-end (unneeded dependency).

3.16.10 0.30.1 - 2021-03-30

Fixed

- Fix incorrect dictation formatting for DNS letters (thanks @wolfmanstout).
- Fix incorrect handling of input strings in DNS dictation formatting classes.

3.16.11 0.30.0 - 2021-03-20

Added

- Add Clipboard class methods: get_available_formats() and convert_format_content().

Changed

- Change the Clipboard classes so they have a consistent API across platforms.
- Change the German integer element content to support compound words in range 20-99, including when used as part of larger numbers.
- Change the Paste action class to make use of the Clipboard class changes.
- Change the X11Window class to filter out unhelpful xdotool BadWindow error messages.
- Change the internal integer builder classes to support replacing or adding alternative recognition words for numbers.

Fixed

- Fix a bug with the pyperclip Clipboard.copy_to_system() method.
- Fix diction.formatter logger name to match log.py (thanks @wolfmanstout).
- Fix incorrect handling of None in the Clipboard classes (thanks @MarkRx).
- Fix issues with German integers in the millions and with the words “ein” and “eins”.

- Fix smart casting for command-line interface (CLI) to handle other possible exceptions for some inputs (thanks @daanzu).

3.16.12 0.29.0 - 2020-12-31

Changed

- Add _functions_example.py example command module (thanks @LexiconCode).
- Add smart type casting to CLI loader engine parameter options (thanks @daanzu).
- Improve error message when X11 can't be detected (thanks @dasnessie)

Fixed

- Fix Kaldi grammar loading/unloading while in phrase (thanks @daanzu).
- Fix recognition observer bug with the KaldiEngine.mimic() method.

3.16.13 0.28.1 - 2020-11-15

Fixed

- Fix DNS dictation bug where formatting can be applied incorrectly.

3.16.14 0.28.0 - 2020-10-24

Changed

- Add methods for setting Rule and Grammar contexts after instantiation (thanks @Timoses).
- Add warning messages on adding the same exported rule to multiple grammars.
- Change default values for Kaldi vad_padding_end_ms and vad_complex_padding_end_ms engine config arguments (thanks @daanzu).
- Set DPI awareness automatically when dragonfly is imported (Windows only) (thanks @wolfmanstout).

Fixed

- Add missing Grammar.remove_list() method.
- Add missing validation and type checks into the ContextAction class.
- Fix SAPI5 engine processing issues with window title context changes.
- Fix various Kaldi engine bugs and other issues (thanks @daanzu).

3.16.15 0.27.1 - 2020-09-18

Changed

- Add missing debug logging for the FuncContext class (thanks @Timoses).

Fixed

- Fix two bugs with the Monitor class for X11 (Linux).

3.16.16 0.27.0 - 2020-09-09

Added

- Add Kaldi support for special user-modifiable Dictation elements (thanks @daanzu).

Changed

- Bump required KaldiAG version to 1.8.0 for various improvements. See KaldiAG release notes (thanks @daanzu).
- Change X11Window.maximize() method to use Extended Window Manager Hints instead of a hardcoded shortcut.
- Add the *wmctrl* command-line program as a **new required X11 session dependency** for properly maximizing X11 windows via maximization hints.
- Improve Kaldi engine error messages for failed compilation (thanks @daanzu).
- Reorganize, fix and improve various parts of the documentation.

Fixed

- Fix Repetition element so the *max* argument is an exclusive bound (thanks @starwarswii).
- Fix Windows bug where the printscreen key cannot be pressed.

3.16.17 0.26.0 - 2020-08-08

Added

- Add useful audio-related initialization arguments to Kaldi engine back-end (thanks @daanzu).

Changed

- Bump required Kaldi-Active-Grammar version to 1.7.0 fix bugs.
- Change the get_engine() function to stop overriding the default (first) engine if another engine is initialized.

Fixed

- Add missing high numbers to short integer elements (thanks @mrob95).
- Fix Python 2.7 bug where StartApp action args may only use ASCII characters.
- Fix bug with BasicRule where it can only be used as a derived class.
- Fix various Kaldi engine bugs and documentation issues (thanks @daanzu).

3.16.18 0.25.1 - 2020-07-29

Fixed

- Fix Kaldi & Sphinx recognition delays on macOS by optimizing window attribute checks in those engine backends.

3.16.19 0.25.0 - 2020-07-20

Added

- Add BasicRule class for defining commands using elements directly.
- Add Kaldi engine support for recognition confidence information, plus various new engine arguments (thanks @daanzu).
- Add convenience method Window.matches(context) (thanks @daanzu).

Changed

- Move SAPI 5 engine process_grammars_context() method into the EngineBase class.
- Update required versions for lark-parser and KaldiAG version.

Fixed

- Add missing WordParserDns11 entry for the special em dash word.
- Add partial fix for dictation-related DNS recognition failures.
- Change the Windows Clipboard class to poll for clipboard access like the cross-platform class does.
- Fix CLI error reporting and file pattern matching issues.
- Fix bug causing Win32Window.set_foreground() to fail sometimes.
- Fix bug in the NatlinkEngine.disconnect() method.
- Fix some Kaldi engine bugs.

Removed

- Remove old and unused compile() method from ElementBase (thanks @kb100).

3.16.20 0.24.0 - 2020-05-21

Added

- Add optional 'results' arguments to recognition and grammar callbacks that expose internal engine results objects for Natlink and SAPI 5 SR engines.
- Add support for quoted words in rules, which can potentially fix certain recognition issues with Dragon.

Changed

- Change the setup_log() function to output log messages to stderr.
- Make Dictation-only rules work with the Sphinx engine again.
- Make keyboard input faster on X11 by passing '–delay 0' as an argument to Xdotool.
- Update, fix and improve various parts of the documentation.
- Use the old Win32 Clipboard class on Windows instead of the cross-platform class.

Fixed

- Fix sdist package installs by including missing files like version.txt (thanks @thatch).
- Fix the Win32 Clipboard class handling of empty clipboard errors and the CF_TEXT format.
- Raise an error if args were passed to get_engine() but silently ignored (thanks @shervinemami).

3.16.21 0.23.2 - 2020-04-11

Fixed

- Add missing __str__ visualization method for UnsafeActionSeries.
- Add missing catch for IOErrors in the Function.__str__() method.
- Fix __str__ visualization methods that break Unicode support.
- Fix some bugs with how Dragonfly command modules are loaded.

3.16.22 0.23.1 - 2020-04-09

Fixed

- Add temporary mitigation for Windows keyboard action processing bug specific to the Kaldi engine (thanks @daanzu).

3.16.23 0.23.0 - 2020-04-06

Changed

- Add get_current_engine() function that doesn't initialize an engine.
- Add is_primary and name properties to all Monitor classes.
- Change SAPI5 engine backend to use the recognizer language selected in the options window instead of "en".
- Reword confusing Natlink warning message shown when Dragon isn't running.
- Update and fix various parts of the documentation.

Fixed

- Add automatic fix for the NatlinkEngine class that allows threads to work properly after the first grammar is loaded.
- Change Dragonfly monitor lists to always have the primary monitor with coordinates (0, 0) first on the list.
- Fix Mouse action bug with negative absolute screen coordinates that made monitors tricky to access sometimes.
- Fix bug where X11Window.executable may return None in certain circumstances.
- Support AppContext edge cases where window executables or titles aren't valid (thanks @shervinemami).

3.16.24 0.22.0 - 2020-03-20

Changed

- Add __str__ method to essential action classes for visualization (thanks @dmakarov).
- Change the Dictation element's value to be a list of recognized words instead of a DictationContainer object if the 'format' constructor argument is False. Previously, the 'format' argument did nothing.
- Make various improvements to Dragonfly's documentation.
- Make various improvements to the Kaldi engine's audio code (thanks @daanzu).

Fixed

- Add code to verify that natlink is on the Python path before initializing the engine (thanks @LexiconCode).
- Fix Python 2.7 console output encoding errors in on_recognition() callbacks in CLI and module loaders.
- Fix a minor bug in DictListRef's constructor.
- Fix bugs where X11 Keyboard and Window class sub-processes can exit early.
- Fix encoding bug with the string representation of BoundAction.
- Fix some Python 3.x bugs with the Natlink engine and its tests (thanks @mrob95).
- Make DarwinWindow get_window_module/pid methods error safe (thanks @dmakarov).

3.16.25 0.21.1 - 2020-02-24

Fixed

- Add set_exclusive() alias methods to Grammar & EngineBase classes to make some older grammars work again.
- Fix a few issues related to the Impossible and Empty elements (thanks @caspark and @daanzu).
- Fix Win32 modifier bug where the control key could be released if held down when Window.set_foreground() is called.
- Make all engine mimic() methods fail properly when given empty input.

3.16.26 0.21.0 - 2020-02-15

Added

- Add optional recursive mode to CommandModuleDirectory class.
- Add new load and load-directory CLI commands as alternatives to module loader scripts.
- Add new on_end() and on_post_recognition() recognition observers with optional parameters (thanks @daanzu).
- Add Window.set_focus() method for focusing windows without raising them (only supported on X11).
- Add 'focus_only' argument to BringApp and FocusWindow actions to support focusing windows without raising them (only supported on X11).

Changed

- Add context manager to ListBase class for optimized list updates.
- Add missing CommandModule properties and methods to CommandModuleDirectory class.
- Change ActionBase class to catch all exceptions raised during execution, not just ActionErrors (thanks @daanzu).
- Change ActionSeries class to stop execution if errors occur. The ActionSeries.stop_on_failures attribute, UnsafeActionSeries class and the '|' and '|=' operators can be used to have the previous behaviour.
- Change Kaldi retain support to allow retaining only specifically chosen recognitions (thanks @daanzu).
- Change on_recognition() recognition observer to allow optional rule and node parameters on functions (thanks @daanzu).
- Change setup.py test command to support running the test suites with different pytest options (thanks @daanzu).
- Change the StartApp action to use the macOS 'open' program if applicable.
- Clean up and enhance log messages and dependency checks done in the is_engine_available() and get_engine() functions (thanks @LexiconCode).
- Use application IDs instead of application names to differentiate between different application processes on macOS (thanks @dmakarov).

Fixed

- Fix Dragonfly's CLI so glob patterns are expanded where necessary (i.e. if using cmd.exe on Windows).
- Fix Kaldi version number checking (thanks @daanzu).
- Fix Python 2/3 bool incompatibility with dictation containers (thanks @daanzu).
- Fix bug with CommandModuleDirectory 'excludes' constructor parameter.
- Fix bug with the command-line interface where the 'command' argument wasn't required.
- Fix Function action deprecation warning in Python 3.

3.16.27 0.20.0 - 2020-01-03

Added

- Add DarwinWindow class for macOS using 'py-applescript' (thanks to various Aenea contributors).
- Add Kaldi engine support for defining your own, external engine to use for dictation elements (thanks @daanzu).
- Add Kaldi engine support for weights on individual rule elements (thanks @daanzu).
- Add support for special specifiers in Compound specs (thanks @daanzu).

Changed

- Change Kaldi default model directory to 'kaldi_model' (thanks @daanzu).
- Change dragonfly's CLI test command to accept zero file arguments.
- Clean up code in grammar, actions and windows sub-packages.
- Improve overall Kaldi engine recognition accuracy (thanks @daanzu).
- Make a few minor Windows-related speed optimizations (thanks @Versatilus).

Fixed

- Add missing DNS parser entry for the special "numeral" word.
- Fix a Windows bug where the wrong mouse buttons will be pressed if the primary/secondary buttons are inverted.
- Fix a bug with dragonfly's CLI 'test' command where grammars weren't properly unloaded.
- Fix on_recognition() observer callback for the natlink engine.
- Fix various Kaldi engine bugs (thanks @daanzu).
- Fix wsr_module_loader_plus.py for newer Python versions.

Removed

- Remove basic Kaldi module loader 'kaldi_module_loader.py'.

3.16.28 0.19.1 - 2019-11-28

Fixed

- Change the Key action to accept all escaped or encoded characters as key names on Windows.
- Fix a bug where the Key/Text 'use_hardware' argument is ignored.

3.16.29 0.19.0 - 2019-11-26

Added

- Add FocusWindow constructor arguments to select by index or filter by passed function (thanks @daanzu).
- Add extra FocusWindow arguments to BringApp action to use for window matching.

- Add Natlink engine support for retaining recognition data (thanks @daanzu).
- Add RunCommand 'hide_window' argument for using the action class with GUI applications.
- Add StartApp and BringApp 'focus_after_start' argument for raising started applications.
- Add unified 'engine.do_recognition()' method for recognising in a loop from any engine.

Changed

- Add much faster Lark-based parser for compound specs (thanks @mrob95).
- Allow retaining Kaldi engine recognition metadata without audio data (thanks @daanzu).
- Change Key action to allow typing Unicode on Windows.
- Change StartApp and BringApp to allow a single list/tuple constructor argument.
- Change dragonfly's test suite to use *pytest* instead.
- Change engine recognition loops to exit on engine.disconnect().
- Change the base Rule class's default 'exported' value to True (thanks @daanzu).
- Implement the PlaySound action for other platforms using pyaudio.
- Make other various optimisations and changes (thanks @mrob95).
- Various improvements to the Kaldi engine (thanks @daanzu).

Fixed

- Change Key and Text actions to handle multiple keyboard layouts on Windows.
- Change NatlinkEngine.mimic() to handle string arguments.
- Change X11Window class to handle xdotool/xprop errors gracefully instead of panicking.
- Fix Win32Window.get_matching_windows() and the FocusWindow action for recent Dragon versions.
- Fix a few bugs with the RunCommand, StartApp and BringApp actions.
- Fix bug with Kaldi retain audio support where the last dictation wasn't retained (thanks @comodoro).
- Fix engine bugs where grammars could not be loaded/unloaded during Grammar.process_begin() (thanks @mrob95).
- Fix various bugs related to grammar exclusivity.

Removed

- Remove no longer used EngineTestSuite class.
- Remove unfinished command family app sub-package (dragonfly.apps.family).
- Remove unused Win32 dialog and control classes.

3.16.30 0.18.0 - 2019-10-13

Added

- Add grammar/rule weights support for the Kaldi backend (thanks @daanzu).
- Add new functions for recognition state change callbacks.
- Add optional –delay argument to Dragonfly's test command (CLI).
- Allow the passing of window attributes to text engine mimic (thanks @mrob95).

Changed

- Add magic repr methods for debugging (thanks @mrob95).
- Add pyobjc as a required package on Mac OS (for AppKit).
- Improve Kaldi backend performance by parsing directly on the FST instead of with pyparsing (thanks @daanzu).
- Make Kaldi backend work with Python 3 (thanks @daanzu).
- Make other various improvements to the Kaldi backend (thanks @daanzu).
- Make the Monitor class and list work on X11 (Linux) & Mac OS.
- Make the Mouse action work on X11 (Linux) & Mac OS.
- Move 3 monitor-related methods from Win32Window to BaseWindow.

Fixed

- Change Sphinx and text engines to not accept mimicking of non-exported rules (expected behaviour).
- Fix CompoundRule bug where the 'exported' parameter was effectively ignored.
- Fix Natlink engine bug where Canadian English isn't recognised (thanks @dusty-phillips).
- Fix Natlink engine for all variants of supported languages.
- Fix case sensitivity bug with AppContext keyword arguments.
- Fix quite a few bugs with the Kaldi backend (thanks @daanzu).
- Fix two bugs with the text engine's mimic method (thanks @mrob95).

3.16.31 0.17.0 - 2019-09-12

Added

- Add alpha support for the accessibility API on Linux (thanks @wolfmanstout).
- Add keywords argument handling to AppContext class for matching window attributes other than titles and executables.
- Add the ability to set formatting flags for natlink dictation containers (thanks @alexboche).

Changed

- Add Python 3 compatible natlink compiler test (thanks @mrob95).
- Add a note about installing the *xdotool* program in the Kaldi engine documentation (thanks @JasoonS).
- Change the Sphinx engine to allow grammars with the same name (again).
- Move dependency adding code from engine classes into Grammar methods (thanks @mrob95).
- Remove extraneous trailing whitespace from 116 files (thanks @mrob95).
- Remove redundant 'grammar.engine = self' lines from engine classes (thanks @mrob95).
- Lots of Kaldi engine backend improvements & bug fixes (thanks @daanzu).
- Remove keyboard-related messages sometimes printed at import time because similar messages are printed later anyway.
- Update documentation sections on running dragonfly's test suite.
- Update documentation section on logging and logging handlers.

Fixed

- Add check to avoid preparing expensive debug logs when they will be discarded (thanks @wolfmanstout).
- Add missing is_maximized property for Win32Window class.
- Fix Python 3 support in a few places.
- Fix a few problems with the Sphinx engine.
- Fix case sensitivity bug with Window.get_matching_windows().
- Fix minor bug with Win32.get_all_windows().
- Fix various character encoding issues with dragonfly and its unit tests.
- Log 'Is X installed?' messages in X11Window if xprop or xdotool are missing.
- Re-raise errors due to missing xprop or xdotool programs instead of suppressing them.

3.16.32 0.16.1 - 2019-08-04

Added

- Add Dictation string formatting examples into documentation.
- Add Kaldi informational messages during grammar loading pauses.

Changed

- Clean up code style in engines/base/dictation.py.
- Bump required kaldi-active-grammar version to 0.6.0.
- Update Kaldi engine documentation (thanks @daanzu and @LexiconCode).

Fixed

- Fix Win32Window.set_foreground() failures by forcing the interpreter's main thread to "receive" the last input event (press & release control).
- Fix quite a few bugs with the Kaldi engine. (thanks @daanzu).
- Make the Sphinx engine ignore unknown words in grammars instead of raising errors.

3.16.33 0.16.0 - 2019-07-21

Added

- Add FakeWindow class imported as 'Window' on unsupported platforms.
- Add RPC methods for getting speech state & recognition history.
- Add Window.get_matching_windows() and Window.get_window class methods.
- Add X11Window class for interacting with windows on X11 (adapted from Aenea).
- Add alternative dragonfly module loader for natlink.
- Add documentation for X11 keyboard and window support.
- Add enhancements to Dictation and DictationContainer objects (thanks @mrob95).
- Add missing Integer Repeat factor example into documentation.
- Add optional '–language' argument to dragonfly's 'test' command (CLI).
- Add xdotool & libxdo keyboard implementations to replace pynput on X11 (adapted from Aenea).

Changed

- Change the dragonfly.windows.window module to import the current platform's Window class.
- Improve Kaldi documentation and add an example demo script (thanks @daanzu).
- Make test_actions.py and test_window.py files run with all test suites and on all platforms.
- Move some code from FocusWindow into Window classes.
- Rename dragonfly's Window class to Win32Window and move it into win32_window.py.
- Swap Repeat class's constructor arguments so that 'extra' is first (backwards-compatible) (thanks @mrob95).
- Unmock the Window, WaitWindow, FocusWindow, BringApp and StartApp classes for all platforms.
- Update Kaldi engine backend with user lexicon support, microphone listing, other improvements and bug fixes (thanks @daanzu).

Fixed

- Fix DragonflyError raised if importing ShortIntegerContent whilst using a speaker language that isn't English.
- Fix Thread.isAlive() deprecation warnings in Python 3.7.
- Fix import error in SAPI5 engine file (specific to Python 3).
- Fix incorrect file names in the 'plus' module loaders.

- Fix problem with building documentation when kaldi_active_grammar is installed.
- Fix spec string decoding in the Text action class.

3.16.34 0.15.0 - 2019-06-24

Added

- Add new Kaldi engine backend for Linux & Windows, including documentation and module loaders (thanks @daanzu).
- Add more featureful loader for WSR with sleep/wake functionality (thanks @daanzu).
- Add FuncContext class that determines context activity by callable argument (thanks @daanzu).
- Allow all timer manager callbacks to be manually disabled (used in tests).

Changed

- Change RunCommand action to use a member for the process_command argument.
- Change how Sapi5Compiler compiles Impossible elements (more impossible now).
- Change sphinx engine install instructions and required dependency versions.
- Change the dragonfly.timer._Timer class so that it works correctly for all supported engines and platforms via engine.create_timer().
- Make local development documentation use read_the_docs theme (thanks @daanzu).
- Move timer-related engine code into DelegateTimerManagerInterface so it is re-used by multiple engines.

Deprecated

- Deprecate the old dragonfly.timer._Timer class.

Fixed

- Fix SAPI5 engine setting grammars as not exclusive (thanks @daanzu).
- Fix SAPI5 window change detection and allow manually processing (thanks @daanzu).
- Fix slow RPC response times for WSR and natlink by adjusting engine timer intervals.
- Preserve Dragon mic state in the NatlinkEngine.speak() method (thanks @lexxish).

Removed

- Remove sphinxwrapper Git sub-module from project.

3.16.35 0.14.1 - 2019-05-31

Changed

- Change English integers to include “too” and “to” as equivalents for “two” (thanks @lexxish).

3.16.36 0.14.0 - 2019-05-21

Added

- Add documentation on dragonfly's logging infrastructure.
- Add dragonfly.rpc sub-package and usage example.
- Add enable() and disable() methods to ThreadedTimerManager class.
- Add optional "repeating" parameter to the multiplexing Timer class and engine.create_timer() method.
- Add recognize_forever() method to WSR engine class.

Changed

- Change AppContext class to allow lists of titles and executables (thanks @mrob95).
- Change WSR engine to call timer functions on the main thread.
- Change dragonfly stdout logging formatter to include the level name.
- Make dragonfly's multiplexing timer classes more thread safe.
- Replace WSR module loader's PumpWaitingMessages loop with engine.recognize_forever().
- Simplify sphinx engine availability checks.

Fixed

- Fix WSR engine context bug with a hook for foreground window changes (thanks @tylercal).
- Fix a bug with Monitor objects caused by incorrect coordinate calculations (thanks @tylercal).
- Fix some example files that break if used with Python 3.
- Stop calling setup_log() in a few dragonfly modules to avoid side effects.
- Stop encoding to windows-1252 in a few places if using Python 3 (thanks @tylercal).
- Stop erasing dragonfly's logging file now that setup_log() isn't always used.

3.16.37 0.13.0 - 2019-04-24

Added

- Add and document optional "remap_data" parameter to Function action to allow using extras with different names than the function argument names.
- Add Key, Text and Paste action support for X11 and Mac OS using pynput.
- Add modified ContextAction class from Aenea (thanks @calmofthestorm).
- Add more flexible ShortIntegerRef class (thanks @mrob95).

Changed

- Allow saying "oh" as well as "zero" for IntegerRefs.
- Change the Sphinx engine to disallow multiple grammars with the same name.
- Change the Text action's default pause value to 0.005 seconds & make it configurable.
- Rename *Language Support* doc page to *Language Support & Sub-package*.
- Rename 3 example command modules to start with underscores.
- Stop mocking Windows-only sendinput classes & functions on other platforms.
- Update some documentation to mention that dragonfly's module loaders will load from files matching "_*.py" rather than "*.py".

Fixed

- Allow Text sub-classes to override the '_pause_default' attribute.
- Fix Sphinx engine bug where grammar searches could be overridden.
- Fix some issues with dragonfly's mocked actions.

3.16.38 0.12.0 - 2019-04-04

Added

- Add *CONTRIBUTING.rst* file.
- Add Repetition 'optimize' parameter that should reduce grammar complexity.
- Add SphinxEngine.default_search_result property.
- Add SphinxEngine.write_transcript_files method.
- Add WSR/SAPI5 retain audio support for saving recognition data (thanks @daanzu).
- Add example *sphinx_wave_transcriber.py* script into *dragonfly/examples*.
- Allow passing keyword arguments to get_engine() functions (thanks @daanzu).

Changed

- Change Sphinx and text engines to call notify_recognition() before rule processing.
- Change Sphinx engine to allow specifying default decoder search options other than "-lm".
- Change SphinxEngine.process_wave_file() method to yield recognised words.
- Change the format of the Sphinx engine's saved training data.
- Disable the Sphinx engine's built-in key phrases if the engine language isn't English.
- Disable writing Sphinx engine training data to files by default.
- Erase dragonfly's log file when creating the logging handler to avoid large files.
- Make all Sphinx engine configuration optional.
- Replace Sphinx engine's *PYAUDIO_STREAM_KEYWORD_ARGS* config option with 4 new options.

- Simplify Sphinx engine backend code and improve its performance.
- Update Sphinx engine documentation to reflect the other changes.

Fixed

- Add rule processing error handling to the Sphinx and text engines.
- Fix lots of bugs with the Sphinx engine backend.
- Fix Sphinx engine's support for exclusive grammars and multiplexing timers.
- Minimise dropped audio frames when recording with the Sphinx engine.

Removed

- Remove Sphinx engine's *config.py* file.
- Remove the Sphinx engine's support for Dictation elements for now.
- Remove/hide some unnecessary public SphinxEngine methods and properties.

3.16.39 0.11.1 - 2019-02-22

Changed

- Change the RunCommand action to allow the *command* argument to be a list to pass directly to *subprocess.Popen* instead of through *shlex.split()*.

Fixed

- Fix the RunCommand action so it properly parses command strings using non-POSIX/Windows paths.
- Fix minor issues with RunCommand's string representation and error logging.

3.16.40 0.11.0 - 2019-01-30

Added

- Add additional tests to dragonfly's test suites.
- Add documentation for dragonfly's timer classes.
- Add new synchronous and process properties and error handling to the RunCommand action.
- Add timer manager class for the text input and SAPI 5 engines.

Changed

- Change default engine class for SAPI 5 engine backend to Sapi5InProcEngine.
- Change logging framework to use *~/.dragonfly.log* as the log file to make logging work on Windows and on other operating systems.
- Change the Natlink test suite to run different tests for different DNS versions.

- Change the default test suite to the "text" engine's test suite and add it to the CI build.
- Change typeables.py so that all symbols can be referred to by their printable representation (thanks @wolfmanstout).
- Make several changes to the SAPI 5 engine backend so it passes the relevant dragonfly tests.
- Update how _generate_typeables.py generates code used in typeables.py.
- Update several documentation pages.
- Use a RecognitionObserver in dfly-loader-wsr.py for user feedback when using Sapi5InProcEngine.

Fixed

- Add default implementation for the RunCommand.process_command method so that most commands don't hang without an implementation.
- Fix bug where the Text action intermittently ignores the hardware_apps override (thanks @wolfmanstout).
- Fix some encoding bugs with the text input engine.
- Fix various issues with dragonfly's tests and test framework.

Removed

- Remove old test files.

3.16.41 0.10.1 - 2019-01-06

Fixed

- Disable **backwards-incompatible** Unicode keyboard functionality by default for the Text action. Restoring the old behaviour requires deleting/modifying the *~/.dragonfly2-speech/settings.cfg* file.

3.16.42 0.10.0 - 2018-12-28

Added

- Add configurable Windows Unicode keyboard support to the Text action (thanks @Versatilus).
- Add Windows accessibility API support to Dragonfly (thanks @wolfmanstout).
- Add a command-line interface for Dragonfly with a "test" command.
- Add multi-platform RunCommand action.
- Add text input engine backend.

Changed

- Change default paste key for the Paste action to Shift+insert.
- Change typeables.py to log errors for untypeable characters.
- Make **backwards-incompatible** change to the Text class where it no longer respects modifier keys being held down by default.

- Move TestContext class from Pocket Sphinx engine tests into test/infrastructure.py.
- Move command module classes from loader scripts into dragonfly/loader.py.

Fixed

- Fix various Unicode and encoding issues (thanks @Versatilus).

3.16.43 0.9.1 - 2018-11-22

Changed

- Various changes to documentation.
- Make Arabic, Indonesian and Malaysian languages automatically load if required.

Fixed

- Fix a bug with dragonfly's MagnitudeIntBuilder class specific to Python 3.x.
- Replace all imports using 'dragonfly.all' with just 'dragonfly'.
- Fix a bug where mouse wheel scrolling fails with high repeat values (thanks @wolfmanstout).
- Fix a few minor problems with the Pocket Sphinx engine.
- Fix error handling and logging when initialising the WSR/SAPI5 engine.

3.16.44 0.9.0 - 2018-10-28

Added

- Add default VAD decoder config options to Pocket Sphinx engine config module.
- Add documentation page on dragonfly's supported languages.
- Add repository core.autorclf settings for consistent file line endings.
- Add scrolling and extra button support for dragonfly's Mouse action (thanks @Versatilus).

Changed

- Adjust pyperclip version requirements now that a bug is fixed.
- Change error types raised in a few Rule class methods.
- Change NatlinkEngine.speak() to turn on the mic after speech playback for consistency between Dragon versions.
- Normalise all file line endings to Unix-style line feeds.

Fixed

- Make Read the Docs generate documentation from Python modules again.

3.16.45 0.8.0 - 2018-09-27

Added

- Add EngineBase.grammars property for retrieving loaded grammars.
- Add MappingRule.specs property to allow retrieval of specs after initialisation.
- Add checks in Sphinx engine for using unknown words in grammars and keyphrases.
- Add configurable speech and hypothesis recording to Sphinx engine for model training.
- Add Sphinx engine documentation page.

Changed

- Change Sphinx engine module loader to use local engine config if it exists.
- Change README to reference the new documentation page on the Sphinx engine.
- Change documentation/conf.py to allow the docs to be built locally.
- Change package distribution name to *dragonfly2* in order to upload releases to PyPI.org.
- Update README and documentation/installation.txt with instructions to install via pip.
- Replace README.md with README.rst because PyPI doesn't easily support markdown any more.

Fixed

- Fix a bug with CompoundRule.spec.
- Fix translation of RuleRef without explicit name in dragonfly2jsgf (thanks @daanzu).
- Update virtual keyboard extended key support (thanks @Versatilus).
- Add missing methods for WSR and Sphinx engines in test/element_tester.
- Fix a few minor problems with the Sphinx engine.
- Fix bug where newly-constructed rules were not inactivated (thanks @wolfmanstout).

Removed

- Remove pyjsgf submodule as it can be installed via pip now.
- Remove Sphinx engine's README now that there is a documentation page.
- Remove ez_setup.py and stop using it in setup.py.

3.16.46 0.7.0 - 2018-07-10

Added

- Add multi-platform Clipboard class that works on Windows, Linux, Mac OS X.
- Support Unicode grammar specs and window titles.
- Support alternate keyboard layouts.

- Add additional speech recognition backend using CMU Pocket Sphinx.
- Add optional Sphinx dependencies as pyjsgf and sphinxwrapper Git sub-modules.
- Add additional unit tests for enhancements.
- Add additional six and pyperclip dependencies in setup.py.

Changed

- Mock Windows-specific functionality for other platforms to allow importing.
- Make pywin32 only required on Windows.
- Made natlink optional in dragonfly/timer.py.
- Clean up code styling and semantic issues.
- Convert code base to support Python 3.x as well as Python 2.7.
- Update natlink links in documentation.

Fixed

- Make the Paste action work with the Unicode clipboard format (thanks @comodoro).
- Fix issues with dragonfly's monitor list and class.

3.16.47 2016

TODO

3.16.48 2015

TODO

3.16.49 2014

TODO

3.16.50 0.6.6b1 - 2009-04-13

TODO

3.16.51 0.6.5 - 2009-04-08

TODO

3.16.52 0.6.4 - 2009-02-01

TODO

3.16.53 0.6.4-rc3 - 2008-12-06

TODO

3.16.54 0.6.4-rc2 - 2008-12-02

TODO

3.16.55 0.6.4-rc1 - 2008-11-12

TODO

3.16.56 0.6.1 - 2008-10-18

This release is the first in the Git version control system.

Direct links within this documentation to help you get started:

- *Features and target audience*
- *Installation*

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

Q

R

S

T

U

V

W

X