# dragonfluid Documentation

*Release 0.9.0.a5*

Charles J. Daniels

September 25, 2015

# Contents

**1 Welcome to dragonfluid! 3**
1.1 About . . . 3
1.2 It's Not For Everyone . . . 3
1.3 Quick Start . . . 3
1.4 How To Speak . . . 4

**2 Concepts 5**
2.1 Overview . . . 5
2.2 Registration . . . 5
2.3 Intros . . . 5
2.4 Literalization . . . 6
2.5 Translation . . . 7

**3 Publicly Supported Objects 9**

**4 Grammars 11**

**5 Elements 13**

**6 Rules 17**

**7 Decorators 19**

**8 Glossary 21**

**9 Indices and tables 23**

**Python Module Index 25**

Contents:

CHAPTER 1

# Welcome to dragonfluid!

## 1.1 About

The dragonfluid library is a simple extension to dragonfly, a library to create *rules or macros* that work with Dragon NaturallySpeaking or Windows Speech Recognition. dragonfluid adds “out of the box” support for *chaining* multiple commands in a row without pausing during speech. You are assumed to be familiar with dragonfly and its use.

## 1.2 It’s Not For Everyone

If you have existing voice commands whose words you absolutely do not want to alter, dragonfluid might not be for you, especially if those commands consist of hoots, made up words, or novel syllables. If you are willing to alter your commands then little should get in your way. I recommend trying in any case rather than assuming the worst if you’re interested in easy to add on chaining support, but thought you should know in advance it’s not equally great with all scenarios.

You may want to read further details regarding `The Effect Of Fluidity` on command recognition, especially if you have strange commands or commands you cannot or will not alter.

## 1.3 Quick Start

The easiest way to give it a whirl is to:

- use `pip install dragonfluid` from the command line to install dragonfluid, if Python pip is on your path, otherwise it can be located in the Scripts folder of your Python installation or downloaded. For help installing with pip, look here.
- or download and unzip a release from here and from the command line run the setup script using `python setup.py install` assuming python.exe is on your path. For help installing, look here.
- once installed, import the following objects into your code,

```
from dragonfluid import GlobalRegistry, FluidRule, QuickFluidRules
```

- replace any `Grammar` class with *`GlobalRegistry`*,
- replace any `CompoundRule` class with *`FluidRule`*,
- replace any `MappingRule` class with *`QuickFluidRules`*,
- reload your macro files!

You don't have to change all your files at once, but *chaining* will generally occur only between dragonfluid rule types added to dragonfluid grammars.

## 1.4 How To Speak

Just speak naturally. Don't worry if pauses are needed, speak as if you trust they are not, and only then address the situations where the intended functionality does not result.

However, now that you can speak multiple commands in a row, there is an additional need to say a literal tag before anything that looks like a command but is not meant to be one. The default *literal tag* options are "literal", "english", and "English". For additional details see the *literalization concept section*.

CHAPTER 2

# Concepts

## 2.1 Overview

dragonfluid is primarily focused on one task – recognizing the occurrence of commands in the middle an *utterance* to allow multiple commands to be spoken in a row without pauses.

When a *rule* is meant to allow *chaining* to other rules, it looks for *registered commands* embedded in the utterance that triggered it, and when it encounters one, the whole utterance from that first command on is put aside. Once the rule finishes processing, the put aside command portion is then mimic'ed. To the speech recognition system, the mimic seems like you just spoke the command right then. And since what was mimic'ed might contain several chained commands, each rule simply cuts off the part meant for it, and forwards the rest.

## 2.2 Registration

Registration is the recording of *commands* that are to be noticed from within the middle of an *utterance*. A `Registry` holds this information and is consulted by rules that perform *chaining* when checking to see if an utterance has embedded commands.

The most common use of a `Registry` is through the `GlobalRegistry`, which is a type of dragonfly `Grammar`. It can be used across files and the rules will all see each other. It's a good default choice. If you have a need to isolate some rules, you can use a `RegistryGrammar` to hold those rules. A `RegistryGrammar` or the `Registry` it holds can be used locally within a single file, or potentially used across a subset of files, but it has no awareness of what is registered in the `GlobalRegistry`.

## 2.3 Intros

When a rule is *registered*, the initial fixed literal text of the command spec is determined and remembered to act as a trigger that the command occurred. These triggers are referred to as the *intros*. This process is largely automatic, but can be guided.

If a *spec* has only words and no extras elements, such as:

```
spec = "next page"
# intros = ["next page"]
```

then the entire spec counts as the intro. If a spec has any extra elements in it, the intros stop at the first extra they encounter. For instance:

```
spec = "go to page <page_number>"
# intros = ["go to page"]
```

This means that any commands whose spec begins with an extra will have an empty string as its intro, and therefore will not be *chained* to from other commands.

Intros is plural, because there can be many:

```
spec = "(close|quit)"
# intros = ["close", "quit"]
```

And it can get arbitrarily complex:

```
spec = "(go [to]|at) next line"
# intros = ["go next line", "go to next line", "at next line"]
```

Each intro will be as long as possible until an extra is encountered:

```
spec = "(insert <part>|delete) below this line"
# intros = ["insert", "delete below this line"]
```

Lastly, consider the following scenario:

```
spec = "copy <direction> word"
extras = (Choice("direction", {"left":"left", "right":"right"}), )
# intros = ["copy"]
```

The automatic generation of intros stops at the `direction` extra, but we can tell that all cases can be determined in advance. The following intros would result in less need for *literal tags*:

```
intros = ["copy left word", "copy right word"]
```

Rules that undergo registration allow you to supply the intros directly to override the automatically generated ones, supplied either to the __init__ or as a class attribute, similar to the spec. So we could supply these improved upon intros. There is a short cut option called **intro_spec** that, instead of supplying individual intros, lets you give a new spec from which to derive them. Our original scenario would then look like:

```
spec = "copy <direction> word"
intros_spec = "copy (left|right) word"
extras = (Choice("direction", {"left":"left", "right":"right"}), )
# intros = ["copy left word", "copy right word"]
```

When supplying intros, directly or through intros_spec, you must supply appropriate values, for if you have no “zixo” command but you place that in a list of intros, if “zixo” occurs in the middle of an utterance, it will get mimic‘ed along with all that follows, the mimic will match no commands, and depending on your setup, that whole rest of the *utterance* will be lost and must then be repeated.

## 2.4 Literalization

Literalization in the context of dragonfluid is an indication that something said, even though it may look like a *registered* command, is actually intended as *free speech dictation*. This is accomplished by preceded these command impostors with a spoken *literal tag*. The default options are “literal”, “english”, and “English”, and they are configurable. It is `Registry`‘s that maintain and work with literal tags.

You don’t necessarily need to literalize every word that begins a command. If you have a command “drop previous element <words>” in your arsenal but no other commands begin with the word drop, then you would not need to literalize the word drop unless it was followed by the words “previous element”. So “drop me a line” could be said plainly. Commands are recognized only by any one of their *registered intros*, avoiding any need for literalization when possible.

You can further minimize the need for literal tags by crafting your commands to not sound like things you tend to dictate. Simple strategies include using rarer words or making commands sound more like headlines or Tarzan speak.

If you actually want to use a literal tag in free speech, just precede it by any literal tag, including itself. “English English” and “literal English” both just *translate* to “English”.

When a literal tag has been literalized to serve as *free speech dictation*, it does not serve as a literal tag for what follows.

## 2.5 Translation

Translation in the context of dragonfluid is taking exact words spoken by the user that may or may not contain *literal tags*, and producing the intended free speech that results from removing any literal tags whenever they are serving the role of literal tags. This is the most common desired form when grabbing free speech dictation for use in the processing of your rules, such as when outputting text to an entry field or document.

Translation happens behind the scenes in the Dictation elements of `FluidRule`‘s. More advanced usage requires a choice of translated versus non-traslated results, and `SplitDictation` objects can return either.

CHAPTER 3

# Publicly Supported Objects

Below is an exhaustive list of the objects that will be imported upon calling:

```
from dragonfluid import *
```

They are mainly ordered by their likelihood of use. You should be familiar with at least the first four. Beyond that is considered more advanced, but certainly can still be common place.

*GlobalRegistry*

*FluidRule*

*QuickFluidRules*

*ActiveGrammarRule*

*SplitDictation*

*SplitForcedDictation*

*RegisteredRule*

*ContinuingRule*

*QuickFluidRule*

*RegistryGrammar*

*Registry*

CHAPTER 4

# Grammars

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

Bases: *`dragonfluid._grammars.RegistryGrammar`*

The GlobalRegistry is a *`RegistryGrammar`* with a single globally shared *`Registry`*. It can be used as the Grammar object across many files, allowing the rules to know about each other for chaining.

**`__init__`**(*name*, *description=None*, *context=None*, *engine=None*, ***kwargs*)

**Parameters**

- **`name`** – Passed to dragonfly Grammar
- **`description`** – Passed to dragonfly Grammar
- **`context`** – Passed to dragonfly Grammar
- **`engine`** – Passed to dragonfly Grammar
- **`**kwargs`** – Passed to *`RegistryGrammar`*

**class** **`RegistryGrammar`**(*name*, *registry=None*, ***kwargs*)

Bases: `dragonfly.grammar.grammar_base.Grammar`

A RegistryGrammar is like a normal Grammar object, except it registers and unregisters *`RegisteredRule`*‘s as they are activated and deactivated, maintaining a registry of those that are currently active.

*`ContinuingRule`*‘s that are added to this grammar will automatically use this object’s registry when seeking out commands embedded in utterances.

**`__init__`**(*name*, *registry=None*, ***kwargs*)

**Parameters**

- **`name`** – Passed to dragonfly Grammar
- **`registry`** (Registry) – The Registry object that serves as the active *Registration* list. It may be shared across RegistryGrammar instances. If None, a local Registry object is created.
- **`**kwargs`** – Passed safely to dragonfly Grammar

**class** **`Registry`**(*literal_tags=['English', 'english', 'literal']*, *override_tags=False*)

A registry maintains information about a set of known active rules and the *literal tags* that must precede their *intros* when their commands are meant as free speech dictation.

Working directly with a Registry object is an advanced use case.

A registry exposes services regarding inspection and parsing of utterances as it relates to its literal tags and currently actively registered commands.

**`__init__`**(*literal_tags=['English', 'english', 'literal'], override_tags=False*)

**Parameters**

- **`literal_tags`** (*string list*) – These words will function as *literalization* markers to indicate that what follows is not a command, but rather free speech dictation.
- **`override_tags`** (*bool*) – If False, the literal_tags supplied to __init__ will be added to the defaults, otherwise they will replace them.

**`has_partial`**(*partial_command*)
Returns True if the string supplied is an initial substring of a registered intro, assuming only full words are supplied.

**`is_registered`**(*intro*)

**Parameters** **`command_intro`** (*string*) – A command *intro* to test for *registration*.

**Returns** True if registered, False otherwise

**Return type** bool

**`register_rule`**(*rule*)
Adds the rule to a list of known active rules. Not generally called directly by users. For more information see the *registration* concept section.

**`starts_with_registered`**(*words_iterable*)
Returns True if the iterable of strings begins with the words of a registered command.

**`translate_literals`**(*words_iterable*)
Returns a list of words, stripped of *literal tags* in a semantically meaningful way. Final isolated literal_tag's are stripped.

When a literal_tag precedes a literal_tag, the second occurrence only is retained.

In a string of all literal_tag's, exactly the odd indexed ones (in a 0-indexed sense) would be returned.

**`unregister_rule`**(*rule*)
Removes the rule from the list of known active rules. Not generally called directly by users.

CHAPTER 5

# Elements

class **SplitDictation** (*name*, *registry=None*, *forced_dictation=False*, ***kwargs*)

A rule element used to split recognized dictation into an initial free dictation part, and a following command part. Either part is optional, unless the element is initialized with the forced_dictation element to True.

The following example shows this element being used and retrieved in the standard expected way.

```
from dragonfluid import RegistryRule, SplitDictation

class SplitterRule(RegistryRule):
    spec = "set name <name_split>"
    extras = (SplitDictation("name_split"), )
    def _process_recognition(self, node, extras):
        name_split = extras["name_split"]
        name = name_split.dictation
```

The result is a type of container from which parts of the result may be retrieved. The full list of attributes are individually documented below, but a simple naming scheme is in place. The first part of the attribute name indicates the part desired:

- **full** - The entire utterance
- **dictation** - The utterance only up to the first accepted command, may be the empty string if the utterance began with an accepted command
- **command** - The rest of the utterance starting with the first accepted command, through the end of the utterance

The second part indicates the return type desired:

- **_words** - A string list of the words
- **_container** - The same type of dictation container that a Dictation element would yield, some derived class of BaseDictationContainer as appropriate for the speech recognition system in use.
- *default* - If neither of the above are indicated, the default result type is a string.

The third part indicates whether literal tags should be retained or translated out:

- **_notrans** - Retain the literal tags
- **_trans** - Strip literal tags and return only the intended content
- *default* - If neither of the above are indicated, the result will have the default behavior most common when using the part requested. **full** and **command** parts will retain literal tags, while **dictation** parts will strip them so as to only return the intended free speech. Default translation of the parts applies to all return types.

There is also an issue of formatting. The various dictation containers have a formatting option. For Windows Speech Recognition there is no real formatting provided beyond separating words with spaces. Dragon NaturallySpeaking provides more sophisticated formatting. All return types except for the **_container** values have formatting applied to the result returned. If you absolutely do not want the formatting applied, you must request the containers directly, from which you can choose to apply formatting or not. If you choose a **_trans** container, it will have had literal tags stripped, but otherwise be unmodified.

**`__init__`**(*name*, *registry=None*, *forced_dictation=False*, ***kwargs*)

**Parameters**

- **`name`** (*string*) – The name of this element, used as the keyname in the extras dictionary passed back to _process_recognition
- **`registry`** (Registry) – The *`Registry`* instance that determines what words form a command and what literal tags are in effect. If None, the *`ActiveGrammarRule`* decorator will set the registry of any *`RegistryGrammar`* derived instance the containing rule is added to.
- **`forced_dictation`** (*bool*) – When True, refuses to recognize utterance-initial commands, so as to ensure this element returns non-empty free dictation.
- **`kwargs`** – Passed safely to Dictation.__init__

**`command`**
Alias for *`command_notrans`*

**`command_container`**
Alias for *`command_container_notrans`*

**`command_container_notrans`**
Returns any and all content starting from first full command intro, if any. Content is returned as a BaseDictationContainer of the appropriate type given the speech recognition system in use, without any alterations of any sort applied to the container contents.

**`command_container_trans`**
Returns any and all content starting from first full command intro, if any. Content is returned as a BaseDictationContainer of the appropriate type given the speech recognition system in use, with no formatting applied yet with literal tags translated to their intended result.

**`command_index`**
Returns the 0-based word index at which the first accepted full command intro occurs, or the index beyond last if no such intro occurs. If forced_dictation was set True during initialization, any utterance-initial command will be skipped to ensure dictation content is non-empty.

**`command_notrans`**
Returns any and all content starting from first full command intro, if any. Content is returned as a string with formatting and with literal tags retained.

**`command_trans`**
Returns any and all content starting from first full command intro, if any. Content is returned as a string with formatting and with literal tags retained.

**`command_words`**
Alias for *`command_words_notrans`*

**`command_words_notrans`**
Returns any and all content starting from first full command intro, if any. Content is returned as a word list with formatting and with literal tags retained.

**`command_words_trans`**
Returns any and all content starting from first full command intro, if any. Content is returned as a word list with formatting and with literal tags translated to their intended result.

**`dictation`**
Alias for *`dictation_trans`*.

**`dictation_container`**
Alias for *`dictation_container_trans`*.

**`dictation_container_notrans`**
Returns any and all content up to the first full command intro, if any. Content is returned as a BaseDictationContainer of the appropriate type given the speech recognition system in use, without any alterations of any sort applied to the container contents.

**`dictation_container_trans`**
Returns any and all content up to the first full command intro, if any. Content is returned as a BaseDictationContainer of the appropriate type given the speech recognition system in use, with no formatting applied yet with literal tags translated to their intended result.

**`dictation_notrans`**
Returns any and all content up to the first full command intro, if any. Content is returned as a string with formatting and with literal tags retained.

**`dictation_trans`**
Returns any and all content up to the first full command intro, if any. Content is returned as a string with formatting and with literal tags translated to their intended result.

**`dictation_words`**
Alias for *`dictation_words_trans`*.

**`dictation_words_notrans`**
Returns any and all content up to the first full command intro, if any. Content is returned as a word list with formatting and with literal tags retained.

**`dictation_words_trans`**
Returns any and all content up to the first full command intro, if any. Content is returned as a word list with formatting and with literal tags translated to their intended result.

**`full`**
Alias for *`full_notrans`*.

**`full_container`**
Alias for *`full_container_notrans`*.

**`full_container_notrans`**
Returns the full content, as a BaseDictationContainer of the appropriate type given the speech recognition system in use, without any alterations of any sort applied to the container contents.

**`full_container_trans`**
Returns the full content, as a BaseDictationContainer of the appropriate type given the speech recognition system in use, with no formatting applied yet with literal tags translated to their intended result.

**`full_notrans`**
Returns the full content, as a string, with formatting applied and with literal tags retained.

**`full_trans`**
Returns the full content, as a string, with formatting applied and with literal tags translated to their intended result.

**`full_words`**
Alias for *`full_words_notrans`*.

**`full_words_notrans`**
Returns the full content, as a word list, with formatting applied and with literal tags retained.

**`full_words_trans`**
Returns the full content, as a word list, with formatting applied and with literal tags translated to their intended result.

**`translate`**(*words_iterable*)
Returns a word list, as translated.

**class `SplitForcedDictation`**(*name*, *registry=None*, ***kwargs*)
A SplitDictation with forced_dictation set to True, guaranteed to return a value for dictation, even if it must ignore an utterance-initial command from which to provide it.

**`__init__`**(*name*, *registry=None*, ***kwargs*)

**Parameters**

- **`name`** (*string*) – The name of this element, used as the keyname in the extras dictionary passed back to _process_recognition
- **`registry`** (*_Registry*) – The _Registry instance that determines what words form a command
- **`kwargs`** – Passed safely to *`SplitDictation`*

CHAPTER 6

# Rules

**class** **`FluidRule`**(***kwargs*)

Bases: *`dragonfluid._rules.RegisteredRule`*, *`dragonfluid._rules.ContinuingRule`*

A FluidRule is both a *`RegisteredRule`* and a *`ContinuingRule`*, meaning it can be *chained* to from other commands, and then chain off to further commands. This is the most common case, for general use unless you have specific needs. These always attempt to chain automatically.

It must be added to a *`RegistryGrammar`*, such as the *`GlobalRegistry`*, to enabled all features.

**`__init__`**(***kwargs*)

**Parameters** **`**kwargs`** – passed to *`ContinuingRule`* and *`RegisteredRule`*

**class** **`QuickFluidRules`**(*grammar*)

Used like a MappingRule but results in *`FluidRule`*'s rather than simple CompoundRule's.

The `mapping` attribute is extended. In addition to the normal key/value pairs of spec/action, a value may also be a list or tuple whose first element is the usual action, and whose second element is a dict of parameters to be passed as **kwargs to *`QuickFluidRule`*.

**`__init__`**(*grammar*)

Not usually called directly, but rather via *`ActiveGrammarRule`*.

**Parameters** **`grammar`** – The Grammar to add rules to, generally a *`RegistryGrammar`* such as the *`GlobalRegistry`*.

**class** **`RegisteredRule`**(*intros=None*, *intros_spec=None*, ***kwargs*)

A rule that can undergo *registration* to allow its command to be noticed in the middle of an utterance, allowing other commands to pass off to this rule. It must be added to a *`RegistryGrammar`*, such as the *`GlobalRegistry`*, for the registration to actually be performed. Otherwise, it acts like a normal CompoundRule.

**`__init__`**(*intros=None*, *intros_spec=None*, ***kwargs*)

For information regarding `intros` and `intros_spec`, refer to the *intros documentation*.

**Parameters**

- **`intros`** (*string, string list, or None*) – If None, the command *intros* will be automatically determined from the spec, otherwise any string provided, by itself or in a list, will be registered as an intro of the command. If supplied, overrides any provided `intros_spec`.
- **`intros_spec`** (*string*) – If supplied, will be parsed to obtained the intros for the command, similar in manner to how spec is parsed.
- **`**kwargs`** – passed safely to CompoundRule

**class `ContinuingRule`**(***kwargs*)

A rule that automatically looks for *embedded commands* and *chains* to them. It must be added to a `RegistryGrammar`, such as the `GlobalRegistry` to enable all features.

**`__init__`**(***kwargs*)

**Parameters** **`**kwargs`** – passed safely to CompoundRule

**class `QuickFluidRule`**(*spec, action, args={}, **kwargs*)

Bases: `dragonfluid._rules.FluidRule`

A shortcut to assign an action to a spec.

Example:

```
rule = QuickFluidRule("press home key", Key("home"))
```

**`__init__`**(*spec, action, args={}, **kwargs*)

**Parameters**

- **`spec`** (*string*) – The spec for this command, from which *intros* will be determined.
- **`action`** (a dragonfly action) – The action to be executed when this command is said.
- **`args`** (*dict*) – Provides a way to add to or modify the extras dictionary. The args dictionary has keys of name strings, items of function callbacks. The callbacks are supplied a single parameter of a dictionary of extras, and their return value is assigned to the extra named by the key. When the `action` is executed, it will then have these final values available to it.
- **`**kwargs`** – Passed to `FluidRule`, except `"name"` and `"spec"` ignored.

# Decorators

**ActiveGrammarRule**(*grammar*)

A rule class decorator to automatically instantiate and add the rule to the grammar specified.

Example:

```
from dragonfly import Grammar, CompoundRule, MappingRule
from dragonfluid import ActiveGrammarRule, FluidRule, QuickFluidRules

my_grammar_instance = Grammar("my_grammar")

@ActiveGrammarRule(my_grammar_instance)
class MyRule(CompoundRule):
    pass

@ActiveGrammarRule(my_grammar_instance)
class MyRules(MappingRule):
    pass

@ActiveGrammarRule(my_grammar_instance)
class MyFluidRule(FluidRule):
    pass

@ActiveGrammarRule(my_grammar_instance)
class MyQuickRules(QuickFluidRules):
    pass
```

CHAPTER 8

# Glossary

**chaining** The ability to invoke multiple recognition elements in a row by speaking them as a single *utterance*, i.e. without pausing between them. In dragonfluid, chains may be of any length. Depending on the scenario, two neighboring chained elements may be comprised of free speech dictation and a command, in either order, or a pair of commands.

Only rules derived from `ContinuingRule` will pass off, or chain, to successive commands.

**command action** The action executed when a *rule* is triggered by its *command*.

**command** Spoken content within an *utterance* that is meant to trigger the execution of a *command action*. Often specified in the form of a *spec*. In contrast with *dictation*.

**dictation, free speech dictation** Spoken content within an *utterance* that is meant to be captured as its textual representation, generally as a means to supply content to a *command action*, such as for printing to the screen. In contrast with a *command*.

**dictation container** A type of value produced by a `Dictation` element, derived from `DictationContainerBase`, and specific to the speech recognition system in use. The elements provided by dragonfluid, such a `SplitDictation` can also return these containers upon request.

**embedded command** A *command* within an *utterance* that does not occur at the beginning of the utterance.

**extras** Broadly speaking, an extra is a part of a command that hears and results in a certain type of content.

An extra uses a named element, derived from `ElementBase`, and provides a value, such as text or a *dictation container*. Dragonfly documentation provides a list of elements.

It is often used here as a term for the dictionary of extras passed to the _process_recognition callback of a rule. You are generally expected to know how to access the various extras from this dictionary, and when documentation states that extras are passed to or returned from a function, the form implied is this dictionary. Note that this dictionary generally does not container the underlying element that generates a value, so extras are distinct from elements, with extras using elements and producing values under the same name.

**intro, command intro** The initial part of a command's *spec*, consisting only of static literal words, up to the first encountered *extra* reference in angle brackets. For further details see the *intros* concept section.

**literal tag** A word spoken within an *utterance* to specific that what follows is *free speech dictation* even when it looks a *command* or literal tag. For further details see the *literalization* concept section.

**registered command** A command that can be triggered from within the middle of an *utterance*. For further details see the *registration* concept section.

**rule, macro** A triggerable event. The trigger is the *command* and the event triggered is the *command action*.

**spec** A common dragonfly attribute that determines which spoken words will trigger a *rule*. It may be a fixed literal command spec, such as "show desktop", or it may include references to *extras* in angle brackets, such as "delete left <characterCount> characters".

**utterance** The contiguous stream of spoken content captured by your speech recognition system starting from the moment your it determines you have begun speaking through until the moment it encounters enough silence to qualify as a pause given its configuration.

CHAPTER 9

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## d

dragonfluid, 9
dragonfluid._decorators, 19
dragonfluid._elements, 13

## Symbols

__init__() (ContinuingRule method), 18
__init__() (FluidRule method), 17
__init__() (GlobalRegistry method), 11
__init__() (QuickFluidRule method), 18
__init__() (QuickFluidRules method), 17
__init__() (RegisteredRule method), 17
__init__() (Registry method), 11
__init__() (RegistryGrammar method), 11
__init__() (SplitDictation method), 14
__init__() (SplitForcedDictation method), 16

## A

ActiveGrammarRule() (in module dragonfluid._decorators), 19

## C

chaining, **21**
command, **21**
command (SplitDictation attribute), 14
command action, **21**
command intro, **21**
command_container (SplitDictation attribute), 14
command_container_notrans (SplitDictation attribute), 14
command_container_trans (SplitDictation attribute), 14
command_index (SplitDictation attribute), 14
command_notrans (SplitDictation attribute), 14
command_trans (SplitDictation attribute), 14
command_words (SplitDictation attribute), 14
command_words_notrans (SplitDictation attribute), 14
command_words_trans (SplitDictation attribute), 14
ContinuingRule (class in dragonfluid._rules), 17

## D

dictation, **21**
dictation (SplitDictation attribute), 15
dictation container, **21**
dictation_container (SplitDictation attribute), 15
dictation_container_notrans (SplitDictation attribute), 15
dictation_container_trans (SplitDictation attribute), 15
dictation_notrans (SplitDictation attribute), 15
dictation_trans (SplitDictation attribute), 15
dictation_words (SplitDictation attribute), 15
dictation_words_notrans (SplitDictation attribute), 15
dictation_words_trans (SplitDictation attribute), 15
dragonfluid (module), 9
dragonfluid._decorators (module), 19
dragonfluid._elements (module), 13

## E

embedded command, **21**
extras, **21**

## F

FluidRule (class in dragonfluid._rules), 17
free speech dictation, **21**
full (SplitDictation attribute), 15
full_container (SplitDictation attribute), 15
full_container_notrans (SplitDictation attribute), 15
full_container_trans (SplitDictation attribute), 15
full_notrans (SplitDictation attribute), 15
full_trans (SplitDictation attribute), 15
full_words (SplitDictation attribute), 15
full_words_notrans (SplitDictation attribute), 15
full_words_trans (SplitDictation attribute), 16

## G

GlobalRegistry (class in dragonfluid._grammars), 11

## H

has_partial() (Registry method), 12

## I

intro, **21**
is_registered() (Registry method), 12

## L

literal tag, **21**

M

macro, **21**

Q

QuickFluidRule (class in dragonfluid._rules), 18
QuickFluidRules (class in dragonfluid._rules), 17

R

register_rule() (Registry method), 12
registered command, **21**
RegisteredRule (class in dragonfluid._rules), 17
Registry (class in dragonfluid._grammars), 11
RegistryGrammar (class in dragonfluid._grammars), 11
rule, **21**

S

spec, **22**
SplitDictation (class in dragonfluid._elements), 13
SplitForcedDictation (class in dragonfluid._elements), 16
starts_with_registered() (Registry method), 12

T

translate() (SplitDictation method), 16
translate_literals() (Registry method), 12

U

unregister_rule() (Registry method), 12
utterance, **22**