pyjsgf Documentation

Release 1.9.0

Dane Finlay

Jan 04, 2023

Contents:

Release v1.9.0

JSpeech Grammar Format (JSGF) compiler, matcher and parser package for Python.

JSGF is a format used to textually represent grammars for speech recognition engines. You can read the JSGF specification here.

CHAPTER 1

Introduction

pyjsgf can be used to construct JSGF rules and grammars, compile them into strings or files, and find grammar rules that match speech hypothesis strings. Matching speech strings to tags is also supported. There are also parsers for grammars, rules and rule expansions.

There are some usage examples in pyjsgf/examples which may help you get started.

1.1 Installation

To install pyjsgf, run the following:

```
$ pip install pyjsgf
```

If you are installing in order to *develop* pyjsgf, clone/download the repository, move to the root directory and run:

```
$ pip install -e .
```

1.2 Supported Python Versions

pyjsgf has been written and tested for Python 2.7 and 3.5.

Please file an issue if you notice a problem specific to the version of Python you are using.

1.3 Unit Testing

There are extensive unit tests in pyjsgf/test. There is also a Travis CI project here. The test coverage is not 100%, but most classes, methods and functions are covered pretty well.

1.4 Multilingual Support

Due to Python's Unicode support, pyjsgf can be used with Unicode characters for grammar, import and rule names, as well as rule literals. If you need this, it is better to use Python 3 or above where all strings are Unicode strings by default.

If you must use Python 2.x, you'll need to define Unicode strings as either `u"text"` or `unicode(text, encoding)`, which is a little cumbersome. If you want to define Unicode strings in a source code file, you'll need to define the source code file encoding.

1.5 Documentation

The documentation for this project is written in reStructuredText and built using Sphinx. Run the following to build it locally:

```
$ cd docs
$ make html
```

CHAPTER 2

`jsgf` — JSpeech Grammar Format (JSGF) package

This package contains classes and functions for compiling, matching and parsing JSGF grammars using rules, imports and rule expansions, such as sequences, repeats, optional and required groupings.

2.1 `errors` — Error classes module

This module contains pyjsgf's exception classes.

2.1.1 Classes

class `jsgf.errors.`**`CompilationError`**

Error raised when compiling an invalid grammar.

This error is currently only raised if a `Literal` expansion is compiled with the empty string (`''`) as its `text` value.

class `jsgf.errors.`**`GrammarError`**

Error raised when invalid grammar operations occur.

This error is raised under the following circumstances:

- When matching or resolving referenced rules that are out-of-scope.
- Attempting to enable, disable or retrieve a rule that isn't in a grammar.
- Attempting to remove rules referenced by other rules in the grammar.
- Attempting to add a rule to a grammar using an already taken name.
- Using an invalid name (such as *NULL* or *VOID*) for a grammar name, rule name or rule reference.
- Passing a grammar string with an illegal expansion to a parser function, such as a tagged repeat (e.g. `blah+ {tag}`).

class `jsgf.errors.`**`ExpansionError`**

This error class has been **deprecated** and is no longer used.

class `jsgf.errors.`**`MatchError`**

This error class has been **deprecated** and is no longer used.

2.2 `expansions` — Expansion classes and functions module

This module contains classes for compiling and matching JSpeech Grammar Format rule expansions.

2.2.1 Classes

class `jsgf.expansions.`**`AlternativeSet`**(**expansions*)

Class for a set of expansions, one of which can be spoken.

`generate`()

Generate a matching string for this alternative set.

Each alternative has an equal chance of being chosen for string generation.

If weights are set, then the probability of an alternative is its weight over the sum of all weights:

```
p = w / sum(weights)
```

`set_weight`(*child*, *weight*)

Set the weight of a child.

The weight determines how likely it is that an alternative was spoken.

Higher values are more likely, lower values are less likely. A value of 0 means that the alternative will never be matched. Negative weights are not allowed.

Note: weights are compiled as floating-point numbers accurate to 4 decimal places, e.g. 5.0001.

Parameters

- **`child`** (*`Expansion | int | str`*) – child/list index/compiled child to set the weight for.
- **`weight`** (*`float | int`*) – weight value - must be >= 0

`weights`

The dictionary of alternatives to their weights.

Return type dict

class `jsgf.expansions.`**`ChildList`**(*expansion*, *seq=()*)

List wrapper class for expansion child lists.

The `parent` attribute of each child will be set appropriately when they added or removed from lists.

`clear`()

Remove all expansions from this list and unset their parent attributes.

`orphan_children`()

Set each child's parent to None.

class `jsgf.expansions.`**`Expansion`**(*children*)

Expansion base class.

`_make_matcher_element`()

Method used by the matcher_element property to create ParserElements.

Subclasses should implement this method for speech matching functionality.

children

List of children.

Returns ChildList

collect_leaves(*order=0*, *shallow=False*)

Collect all descendants of an expansion that have no children. This can include self if it has no children. RuleRefs are also counted as leaves.

Parameters

- **order** – tree traversal order (default 0: pre-order)
- **shallow** – whether to not collect leaves from trees of referenced rules

Returns list

compiled_tag

Get the compiled tag for this expansion if it has one. The empty string is returned if there is no tag set.

Returns str

copy(*shallow=False*)

Make a copy of this expansion. This returns a deep copy by default. Neither referenced rules or their expansions will be deep copied.

Parameters **shallow** – whether to create a shallow copy (default: False)

Returns Expansion

current_match

Currently matched speech value for this expansion.

If the expansion hasn't been matched, this will be None (if required) or '' (if optional).

Returns str | None

generate()

Generate a string matching this expansion.

had_match

Whether this expansion has a `current_match` value that is not '' or None. This will also check if this expansion was part of a complete repetition if it has a Repeat or KleeneStar ancestor.

Returns bool

invalidate_calculations()

Invalidate calculations stored in the lookup tables that involve this expansion. This only effects `mutually_exclusive_of` and `is_descendant_of`, neither of which are used in compiling or matching rules.

This should be called if a child is added to an expansion or if an expansion's parent is changed outside of what `JointTreeContext` does.

Some changes may also require invalidating descendants, the `map_expansion` function can be used with this method to accomplish that:

```
map_expansion(self, Expansion.invalidate_calculations)
```

invalidate_matcher()

Method to invalidate the parser element used for matching this expansion. This is method is called automatically when a parent is set or a ChildList is modified. The parser element will be recreated again when required.

This only needs to be called manually if modifying an expansion tree *after* matching with a Dictation expansion.

is_alternative

Whether or not this expansion has an AlternativeSet ancestor with more than one child.

Returns bool

is_descendant_of(*other*)

Whether this expansion is a descendant of another expansion.

Parameters **other** – Expansion

Returns bool

is_optional

Whether or not this expansion has an optional ancestor.

Returns bool

leaves

Collect all descendants of an expansion that have no children. This can include self if it has no children. RuleRefs are also counted as leaves.

Parameters

- **order** – tree traversal order (default 0: pre-order)
- **shallow** – whether to not collect leaves from trees of referenced rules

Returns list

leaves_after

Generator function for leaves after this one (if any).

Returns generator

static make_expansion(*e*)

Take an object, turn it into an Expansion if it isn't one and return it.

Parameters **e** – str | Expansion

Returns Expansion

matchable_leaves_after

Generator function yielding all leaves after self that are not mutually exclusive of it.

Returns generator

matcher_element

Lazily initialised *pyparsing* `ParserElement` used to match speech to expansions. It will also set `current_match` values.

Returns pyparsing.ParserElement

matches(*speech*)

Match speech with this expansion, set `current_match` to the first matched substring and return the remainder of the string.

Matching ambiguous rule expansions is **not supported** because it not worth the performance hit. Ambiguous rule expansions are defined as some optional literal x followed by a required literal x. For example, successfully matching `'test'` for the following rule is not supported:

```
<rule> = [test] test;
```

Parameters **`speech`** – str

Returns str

`matching_slice`

Slice of the last speech string matched. This will be `None` initially.

Return type slice

`mutually_exclusive_of`(*other*)

Whether this expansion cannot be spoken with another expansion.

Parameters **`other`** – Expansion

Returns bool

`parent`

This expansion's parent, if it has one.

Setting the parent will call `Expansion.invalidate_matcher` as necessary on the new and old parents.

Returns Expansion | None

`repetition_ancestor`

This expansion's closest Repeat or KleeneStar ancestor, if it has one.

Returns Expansion

`reset_for_new_match`()

Call `reset_match_data` for this expansion and all of its descendants.

`reset_match_data`()

Reset any members or properties this expansion uses for matching speech, i.e. `current_match` values.

This does **not** invalidate `matcher_element`.

`root_expansion`

Traverse to the root expansion r and return it.

Returns Expansion

`tag`

JSGF tag for this expansion.

Set to `None` for no tag and the empty string for no tag text.

Returns str

`validate_compilable`()

Check that the expansion is compilable. If it isn't, this method should raise a `CompilationError`.

Raises CompilationError

class `jsgf.expansions.`**`JointTreeContext`**(*root_expansion*)

Class that temporarily joins an expansion tree with the expansion trees of all referenced rules by setting the parent relationships.

This is useful when it is necessary to view an expansion tree and the expansion trees of referenced rules as one larger tree. E.g. when determining mutual exclusivity of two expansions, if an expansion is optional or used for repetition in the context of other trees, etc.

Note: this class will reduce the matching performance if used, but will only be noticeable with larger grammars.

On `__exit__`, the trees will be detached recursively.

This class can be used with Python's `with` statement.

static detach_tree(*x*)

If x is a NamedRuleRef, detach its referenced rule's expansion from this tree.

Parameters **x** – Expansion

static join_tree(*x*)

If x is a NamedRuleRef, join its referenced rule's expansion to this tree.

Parameters **x** – Expansion

class `jsgf.expansions.`**KleeneStar**(*expansion*)

JSGF Kleene star operator for allowing zero or more repeats of an expansion.

For example:

```
<kleene> = (please)* don't crash;
```

generate()

Generate a string matching this expansion.

This method can generate zero or more repetitions of the child expansion, zero repetitions meaning the empty string ("") will be returned.

Return type str

class `jsgf.expansions.`**Literal**(*text*, *case_sensitive=False*)

Expansion class for literals.

case_sensitive

Case sensitivity used when matching and compiling *`Literal`* rule expansions.

This property can be `True` or `False`. Matching and compilation will be *case-sensitive* if `True` and *case-insensitive* if `False`. The default value is `False`.

Return type bool

Returns literal case sensitivity

generate()

Generate a string matching this expansion's text.

This will just return the value of `text`.

Return type str

matching_regex_pattern

A regex pattern for matching this expansion.

This property has been left in for backwards compatibility. The `Expansion.matches` method now uses the `matcher_element` property instead.

Returns regex pattern object

text

Text to match/compile.

This will return lowercase text if *`case_sensitive`* is not `True`.

Return type str

Returns text

validate_compilable()

Check that the expansion is compilable. If it isn't, this method should raise a `CompilationError`.

Raises CompilationError

class `jsgf.expansions.`**`NamedRuleRef`**`(`*name*`)`

Class used to reference rules by name.

`generate``()`

Generate a string matching the referenced rule's expansion.

Return type str

`referenced_rule`

Find and return the rule this expansion references in the grammar.

This raises an error if the referenced rule cannot be found using `self.rule.grammar` or if there is no link to a grammar.

Raises GrammarError

Returns Rule

class `jsgf.expansions.`**`NullRef`**

Reference expansion for the special *NULL* rule.

The *NULL* rule always matches speech. If this reference is used by a rule, that part of the rule expansion requires no speech substring to match.

class `jsgf.expansions.`**`OptionalGrouping`**`(`*expansion*`)`

Class for expansions that can be optionally spoken in a rule.

`generate``()`

Generate a string matching this expansion.

class `jsgf.expansions.`**`Repeat`**`(`*expansion*`)`

JSGF plus operator for allowing one or more repeats of an expansion.

For example:

```
<repeat> = (please)+ don't crash;
```

`generate``()`

Generate a string matching this expansion.

This method can generate one or more repetitions of the child expansion.

Return type str

`get_expansion_matches``(`*e*`)`

Get a list of an expansion's `current_match` values for each repetition.

Returns list

`get_expansion_slices``(`*e*`)`

Get a list of an expansion's `matching_slice` values for each repetition.

Returns list

`repetitions_matched`

The number of repetitions last matched.

Returns int

`reset_match_data``()`

Reset any members or properties this expansion uses for matching speech, i.e. `current_match` values.

This does **not** invalidate `matcher_element`.

class `jsgf.expansions.`**RequiredGrouping**(**expansions*)

Subclass of `Sequence` for wrapping multiple expansions in parenthesises.

class `jsgf.expansions.`**RuleRef**(*referenced_rule*)

Subclass of `NamedRuleRef` for referencing another rule with a Rule object.

Parameters **referenced_rule** –

class `jsgf.expansions.`**Sequence**(**expansions*)

Class for expansions to be spoken in sequence.

class `jsgf.expansions.`**VoidRef**

Reference expansion for the special *VOID* rule.

The *VOID* rule can never be spoken. If this reference is used by a rule, then it will not match unless the reference it is optional.

2.2.2 Functions

`jsgf.expansions.`**filter_expansion**(*e*, *func=<function <lambda>>*, *order=0*, *shallow=False*)

Find all expansions in an expansion tree for which func(x) == True.

Parameters

- **e** – Expansion
- **func** – callable (default: the identity function, f(x)->x)
- **order** – int
- **shallow** – whether to not process trees of referenced rules (default False)

Returns list

`jsgf.expansions.`**find_expansion**(*e*, *func=<function <lambda>>*, *order=0*, *shallow=False*)

Find the first expansion in an expansion tree for which func(x) is True and return it. Otherwise return None.

This function will stop searching once a matching expansion is found, unlike the other top-level functions in this module.

Parameters

- **e** – Expansion
- **func** – callable (default: the identity function, f(x)->x)
- **order** – int
- **shallow** – whether to not process trees of referenced rules (default False)

Returns Expansion | None

`jsgf.expansions.`**flat_map_expansion**(*e*, *func=<function <lambda>>*, *order=0*, *shallow=False*)

Call map_expansion with the arguments and return a single flat list.

Parameters

- **e** – Expansion
- **func** – callable (default: the identity function, f(x)->x)
- **order** – int
- **shallow** – whether to not process trees of referenced rules (default False)

Returns list

`jsgf.expansions.`**`map_expansion`**(*e*, *func=<function <lambda>>*, *order=0*, *shallow=False*)

Traverse an expansion tree and call func on each expansion returning a tuple structure with the results.

Parameters

- **`e`** – Expansion
- **`func`** – callable (default: the identity function, f(x)->x)
- **`order`** – int
- **`shallow`** – whether to not process trees of referenced rules (default False)

Returns tuple

`jsgf.expansions.`**`matches_overlap`**(*m1*, *m2*)

Check whether two regex matches overlap.

Returns bool

`jsgf.expansions.`**`restore_current_matches`**(*e*, *values*, *override_none=True*)

Traverse an expansion tree and restore matched data using the values dictionary.

Parameters

- **`e`** – Expansion
- **`values`** – dict
- **`override_none`** – bool

`jsgf.expansions.`**`save_current_matches`**(*e*)

Traverse an expansion tree and return a dictionary populated with each descendant `Expansion` and its match data.

This will also include `e`.

Parameters **`e`** – Expansion

Returns dict

2.3 `jsgf.ext` — JSGF extensions sub-package

This sub-package contains extensions to JSGF, notably the `Dictation`, `SequenceRule` and `DictationGrammar` classes.

2.3.1 `expansions` — Extension expansion classes and functions module

This module contains extension rule expansion classes and functions.

Classes

`class` `jsgf.ext.expansions.`**`Dictation`**

Class representing dictation input matching any spoken words.

This is largely based on the `Dictation` element class in the dragonfly Python library.

`Dictation` expansions compile to a special reference (`<DICTATION>`), similar to `NullRef` and `VoidRef`. See the `DictationGrammar` class if you want to use this expansion type with CMU Pocket Sphinx.

The matching implementation for `Dictation` expansions will look ahead for possible next literals to avoid matching them and making the rule fail to match. It will also look backwards for literals in possible future repetitions.

It will **not** however look at referencing rules for next possible literals. If you have match failures because of this, only use `Dictation` expansions in public rules *or* use the `JointTreeContext` class before matching if you don't mind reducing the matching performance.

`matching_regex_pattern`
A regex pattern for matching this expansion.

This property has been left in for backwards compatibility. The `Expansion.matches` method now uses the `matcher_element` property instead.

Returns regex pattern object

`use_current_match`
Whether to match the `current_match` value next time rather than matching one or more words.

This is used by the `SequenceRule.graft_sequence_matches` method.

Returns bool

`validate_compilable``()`
Check that the expansion is compilable. If it isn't, this method should raise a `CompilationError`.

Raises CompilationError

Functions

`jsgf.ext.expansions.`**`calculate_expansion_sequence`**`(`*expansion*, *should_deepcopy=True*`)`
Split an expansion into *2*n* expansions where *n* is the number of `Dictation` expansions in the expansion tree.

If there aren't any `Dictation` expansions, the result will be the original expansion.

Parameters

- **`expansion`** – Expansion
- **`should_deepcopy`** – whether to deepcopy the expansion before using it

Returns list

`jsgf.ext.expansions.`**`expand_dictation_expansion`**`(`*expansion*`)`
Take an expansion and expand any `AlternativeSet` with alternatives containing `Dictation` expansions. This function returns a list of all expanded expansions.

Parameters **`expansion`** – Expansion

Returns list

2.3.2 `rules` — Extension rule classes module

This module contains extension rule classes.

Classes

`class` `jsgf.ext.rules.`**`SequenceRule`**`(`*name*, *visible*, *expansion*, *case_sensitive=False*`)`
Class representing a list of regular expansions and `Dictation` expansions that must be spoken in a sequence.

can_repeat
Whether the entire SequenceRule can be repeated multiple times.

Note that if the rule can be repeated, data from a repetition of the rule, such as `current_match` values of each sequence expansion, should be stored before `restart_sequence` is called for a further repetition.

compile()
Compile this rule's expansion tree and return the result.

Returns str

current_is_dictation_only
Whether the current expansion in the sequence contains only `Dictation` expansions.

Returns bool

entire_match
If the entire sequence is matched by successive calls to the matches method, this returns all strings that matched joined together by spaces.

Returns str

expansion_sequence
The expansion sequence used by the rule.

Returns tuple

static graft_sequence_matches(*sequence_rule*, *expansion*)
Take a `SequenceRule` and an expansion and attempt to graft the matches of all expansions in the sequence onto the given expansion in-place.

Not all expansions in the sequence need to have been matched.

Parameters

- **sequence_rule** – SequenceRule
- **expansion** – Expansion

has_next_expansion
Whether there is another sequence expansion after the current one.

Returns bool

matches(*speech*)
Return whether or not speech matches the current expansion in the sequence.

This also sets `current_match` values for the original expansion used to create this rule.

This method will only match once and return False on calls afterward until `refuse_matches` is False.

Parameters **speech** – str

Returns bool

refuse_matches
Whether or not matches on this rule can succeed.

This is set to False if `set_next` is called and there is a next expansion or if `restart_sequence` is called.

This can also be manually set with the setter for problematic situations where, for example, the current expansion is a `Repeat` expansion with a `Dictation` descendant.

Returns bool

restart_sequence()

Resets the current sequence expansion to the first one in the sequence and clears the match data of each sequence expansion.

set_next()

Moves to the next expansion in the sequence if there is one.

tags

The set of JSGF tags in this rule's expansion. This does not include tags in referenced rules.

Returns set

class `jsgf.ext.rules.`**PublicSequenceRule**(*name*, *expansion*, *case_sensitive=False*)

SequenceRule subclass with `visible` set to True.

`jsgf.ext.rules.`**HiddenSequenceRule**

alias of `jsgf.ext.rules.PrivateSequenceRule`

2.3.3 `grammars` — Extension grammar classes module

This module contains extension grammar classes.

Classes

class `jsgf.ext.grammars.`**DictationGrammar**(*rules=None*, *name='default'*, *case_sensitive=False*)

Grammar subclass that processes rules using `Dictation` expansions so they can be compiled, matched and used with normal JSGF rules with utterance breaks.

Parameters

- **rules** – list
- **name** – str
- **case_sensitive** – bool

add_rule(*rule*)

Add a rule to the grammar.

This method will override the new rule's `case_sensitive` value with the grammar's `case_sensitive` value.

Parameters **rule** – Rule

Raises GrammarError

compile()

Compile this grammar's header, imports and rules into a string that can be recognised by a JSGF parser.

Returns str

compile_as_root_grammar()

Compile this grammar with one public "root" rule containing rule references in an alternative set to every other rule as such:

```
public <root> = <rule1>|<rule2>|..|<ruleN>;
<rule1> = ...;
<rule2> = ...;
.
```

(continues on next page)

(continued from previous page)

```
.
.
<ruleN> = ...;
```

This is useful if you are using JSGF grammars with CMU Pocket Sphinx.

Returns str

find_matching_rules(*speech*, *advance_sequence_rules=True*)

Find each visible rule passed to the grammar that matches the *speech* string. Also set matches for the original rule.

Parameters

- **speech** – str
- **advance_sequence_rules** – whether to call `set_next()` for successful sequence rule matches.

Returns list

get_generated_rules(*rule*)

Get the rules generated from a rule added to this grammar.

Parameters **rule** – Rule

Returns generator

get_original_rule(*rule*)

Get the original rule from a generated rule.

Parameters **rule** – Rule

Returns Rule

match_rules

The rules that the `find_matching_rules` method will match against.

Returns list

rearrange_rules()

Move each `SequenceRule` in this grammar between the dictation rules list and the internal grammar used for JSGF only rules depending on whether a `SequenceRule`'s current expansion is dictation-only or not.

remove_rule(*rule*, *ignore_dependent=False*)

Remove a rule from this grammar.

Parameters

- **rule** – Rule object or the name of a rule in this grammar
- **ignore_dependent** – whether to check if the rule has dependent rules

Raises GrammarError

reset_sequence_rules()

Reset each `SequenceRule` in this grammar so that they can accept matches again.

rules

The rules in this grammar.

This includes internal generated rules as well as original rules.

Returns list

2.4 `grammars` — Grammar classes module

This module contains classes for compiling, importing from and matching JSpeech Grammar Format grammars.

2.4.1 Classes

class `jsgf.grammars.`**`Import`**(*name*)

Import objects used in grammar compilation and import resolution.

Import names must be fully qualified. This means they must be in the reverse domain name format that Java packages use. Wildcards may be used to import all public rules in a grammar.

The following are valid rule import names:

- com.example.grammar.rule_name
- grammar.rule_name
- com.example.grammar.*
- grammar.*

There are two reserved rule names: *NULL* and *VOID*. These reserved names cannot be used as import names. You can however change the case to 'null' or 'void' to use them, as names are case-sensitive.

`grammar_name`

The full name of the grammar to import from.

Returns grammar name

Return type str

`resolve`(*memo=None*, *file_exts=None*)

Resolve this import statement and return the imported `Rule` object(s).

This method attempts to parse grammar files in the working directory and its sub-directories. If a dictionary was passed for the *memo* argument, then that dictionary will be updated with the parsed grammar and rules.

Errors will be raised if the grammar could not be found and parsed, or if the import statement could not be resolved.

Parameters

- **`memo`** (*`dict`*) – dictionary of import names to grammar rules
- **`file_exts`** (*`list | tuple`*) – list of grammar file extensions to check against (default: `(".jsgf", ".jgram")`)

Returns imported Rule or list of Rules

Return type Rule | list

Raises GrammarError | JSGFImportError

`rule_name`

The name of the rule to import from the grammar.

Returns rule name

Return type str

`wildcard_import`

Whether this import statement imports every grammar rule.

Returns bool

Return type bool

class `jsgf.grammars.`**`Grammar`**(*name='default'*, *case_sensitive=False*)

Base class for JSGF grammars.

Grammar names can be either a qualified name with dots or a single name. A name is defined as a single word containing one or more alphanumeric Unicode characters and/or any of the following special characters: +-:;,=|/()[]@#%!^&~$

For example, the following are valid grammar names: com.example.grammar grammar

There are two reserved rule names: *NULL* and *VOID*. These reserved names cannot be used as grammar names. You can however change the case to 'null' or 'void' to use them, as names are case-sensitive.

`add_import`(*_import*)

Add an import statement to the grammar.

Parameters **`_import`** – Import

`add_imports`(**imports*)

Add multiple imports to the grammar.

Parameters **`imports`** – imports

`add_rule`(*rule*)

Add a rule to the grammar.

This method will override the new rule's `case_sensitive` value with the grammar's *`case_sensitive`* value.

Parameters **`rule`** – Rule

Raises GrammarError

`add_rules`(**rules*)

Add multiple rules to the grammar.

This method will override each new rule's `case_sensitive` value with the grammar's *`case_sensitive`* value.

Parameters **`rules`** – rules

Raises GrammarError

`case_sensitive`

Case sensitivity used when matching and compiling `Literal` rule expansions.

Setting this property will override the `case_sensitive` values for each `Rule` and `Literal` expansion in the grammar or in any newly added grammar rules.

Return type bool

Returns case sensitivity

`compile`()

Compile this grammar's header, imports and rules into a string that can be recognised by a JSGF parser.

Returns str

`compile_as_root_grammar`()

Compile this grammar with one public "root" rule containing rule references in an alternative set to every other rule as such:

```
public <root> = <rule1>|<rule2>|..|<ruleN>;
<rule1> = ...;
<rule2> = ...;
.
.
.
<ruleN> = ...;
```

This is useful if you are using JSGF grammars with CMU Pocket Sphinx.

Returns str

`compile_grammar`(*charset_name='UTF-8'*, *language_name='en'*, *jsgf_version='1.0'*)

Compile this grammar's header, imports and rules into a string that can be recognised by a JSGF parser.

This method is **deprecated**, use `compile` instead.

Parameters

- **`charset_name`** –
- **`language_name`** –
- **`jsgf_version`** –

Returns str

`compile_to_file`(*file_path*, *compile_as_root_grammar=False*)

Compile this grammar by calling `compile` and write the result to the specified file.

Parameters

- **`file_path`** – str
- **`compile_as_root_grammar`** – bool

`disable_rule`(*rule*)

Disable a rule in this grammar, preventing it from appearing in the compile method output or being matched with the find_matching_rules method.

Parameters **`rule`** – Rule object or the name of a rule in this grammar

Raises GrammarError

`enable_rule`(*rule*)

Enable a rule in this grammar, allowing it to appear in the compile method output and to be matched with the find_matching_rules method.

Rules are enabled by default.

Parameters **`rule`** – Rule object or the name of a rule in this grammar

Raises GrammarError

`find_matching_rules`(*speech*)

Find each visible rule in this grammar that matches the *speech* string.

Parameters **`speech`** – str

Returns list

`find_tagged_rules`(*tag*, *include_hidden=False*)

Find each rule in this grammar that has the specified JSGF tag.

Parameters

- **`tag`** – str
- **`include_hidden`** – whether to include hidden rules (default False).

Returns list

`get_rule`(*name*)

Get a rule object with the specified name if one exists in the grammar or its imported rules.

If `name` is a fully-qualified rule name, then this method will attempt to import it.

Parameters **`name`** – str

Returns Rule

Raises GrammarError | TypeError | JSGFImportError

`get_rule_from_name`(*name*)

Get a rule object with the specified name if one exists in the grammar or its imported rules.

If `name` is a fully-qualified rule name, then this method will attempt to import it.

Parameters **`name`** – str

Returns Rule

Raises GrammarError | TypeError | JSGFImportError

`get_rules`(**names*)

Get rule objects with the specified names, if they exist in the grammar.

Parameters **`names`** – str

Returns list

Raises GrammarError

`get_rules_from_names`(**names*)

Get rule objects with the specified names, if they exist in the grammar.

Parameters **`names`** – str

Returns list

Raises GrammarError

`import_environment`

A dictionary of imported rules and their grammars that functions as the import environment of this grammar.

The import environment dictionary is updated internally by the *`resolve_imports()`* method.

Return type dict

Returns dictionary of import names to grammar rules

`import_names`

The import names associated with this grammar.

Returns list

`imports`

Get the imports for this grammar.

Returns list

`jsgf_header`

The JSGF header string for this grammar. By default this is:

```
#JSGF V1.0;
```

Returns str

match_rules
The rules that the `find_matching_rules` method will match against.

Returns list

remove_import(*_import*)
Remove an Import from the grammar.

Parameters **_import** – Import

remove_imports(**imports*)
Remove multiple imports from the grammar.

Parameters **imports** – imports

remove_rule(*rule*, *ignore_dependent=False*)
Remove a rule from this grammar.

Parameters

- **rule** – Rule object or the name of a rule in this grammar
- **ignore_dependent** – whether to check if the rule has dependent rules

Raises GrammarError

resolve_imports(*memo=None*, *file_exts=None*)
Resolve each import statement in the grammar and make the imported `Rule` object(s) available for referencing and matching.

This method attempts to parse grammar files in the working directory and its sub-directories. If a dictionary was passed for the *memo* argument, then that dictionary will be updated with the parsed grammars and rules.

Errors will be raised if a grammar could not be found and parsed, or if an import statement could not be resolved.

Parameters

- **memo** (*`dict`*) – dictionary of import names to grammar rules
- **file_exts** (*`list | tuple`*) – list of grammar file extensions to check against (default: `(".jsgf", ".jgram")`)

Returns dictionary of import names to grammar rules

Return type dict

Raises GrammarError | JSGFImportError

rule_names
The rule names of each rule in this grammar.

Returns list

rules
Get the rules added to this grammar.

Returns list

visible_rules

The rules in this grammar which have the visible attribute set to True.

Returns list

class `jsgf.grammars.`**RootGrammar**(*rules=None*, *name='root'*, *case_sensitive=False*)

A grammar with one public "root" rule containing rule references in an alternative set to every other rule as such:

```
public <root> = <rule1>|<rule2>|..|<ruleN>;
<rule1> = ...;
<rule2> = ...;
.
.
.
<ruleN> = ...;
```

This is useful if you are using JSGF grammars with CMU Pocket Sphinx.

compile()

Compile this grammar's header, imports and rules into a string that can be recognised by a JSGF parser.

This method will compile the grammar using `compile_as_root_grammar`.

Returns str

2.5 `parser` — Parser module

This module contains functions that parse strings into `Grammar`, `Import`, `Rule` and `Expansion` objects.

2.5.1 Supported functionality

The parser functions support the following:

- Alternative sets, e.g. `a|b|c`.
- Alternative set weights (e.g. `/10/ a | /20/ b | /30/ c`).
- C++ style single/in-line and multi-line comments (`// ...` and `/* ... */` respectively).
- Import statements.
- Optional groupings, e.g. `[this is optional]`.
- Public and private/hidden rules.
- Required groupings, e.g. `(a b c) | (e f g)`.
- Rule references, e.g. `<command>`.
- Sequences, e.g. `run <command> [now] [please]`.
- Single or multiple JSGF tags, e.g. `text {tag1} {tag2} {tag3}`.
- Special JSGF rules `<NULL>` and `<VOID>`.
- Unary kleene star and repeat operators (`*` and `+`).
- Using Unicode alphanumeric characters for names, references and literals.
- Using semicolons or newlines interchangeably as line delimiters.

2.5.2 Limitations

This parser will fail to parse long alternative sets due to recursion depth limits. The simplest workaround for this limitation is to split long alternatives into groups. For example:

```
// Raises an error.
<n> = (0|...|100);

// Will not raise an error.
// As a side note, this will be parsed to '(0|...|100)'.
<n> = (0|...|50)|(51|...|100);
```

This workaround could be done automatically in a future release.

This limitation also applies to long sequences, but it is much more difficult to reach the limit.

2.5.3 Extended Backus–Naur form

Extended Backus–Naur form (EBNF) is a notation for defining context-free grammars. The following is the EBNF used by pyjsgf's parsers:

```
alphanumeric = ? any alphanumeric Unicode character ? ;
weight = '/' , ? any non-negative number ? , '/' ;
atom = [ weight ] , ( literal | '<' , reference name , '>' |
       '(' , exp , ')' | '[' , exp , ']' ) ;
exp = atom , [ { tag | '+' | '*' | exp | '|' , [ weight ] , exp } ] ;
grammar = grammar header , grammar declaration ,
          [ { import statement } ] , { rule definition } ;
grammar declaration = 'grammar' , reference name , line end ;
grammar header = '#JSGF', ( 'v' | 'V' ) , version , word ,
                 word , line end ;
identifier = { alphanumeric | special } ;
import name = qualified name , [ '.*' ] | identifier , '.*' ;
import statement = 'import' , '<' , import name  , '>' , line end ;
line end = ';' | '\n' ;
literal = { word } ;
qualified name = identifier , { '.' , identifier }  ;
version = ? an integer or floating-point number ? ;
reference name = identifier | qualified name ;
rule definition = [ 'public' ] , '<' , reference name , '>' , '=' ,
                  exp , line end ;
special = '+' | '-' | ':' | ';' | ',' | '=' | '|' | '/' | '$' |
          '(' | ')' | '[' | ']' | '@' | '#' | '%' | '!' | '^' |
          '&' | '~' | '\' ;
tag = '{' , { tag literal } , '}' ;
tag literal = { word character | '\{' | '\}' } ;
word = { word character } ;
word character = alphanumeric | "'" | '-' ;
```

I've not included comments for simplicity; they can be used pretty much anywhere. pyparsing handles that for us.

2.5.4 Functions

`jsgf.parser.`**`parse_expansion_string`**`(s)`

Parse a string containing a JSGF expansion and return an `Expansion` object.

Parameters **`s`** – str

Returns Expansion

Raises ParseException, GrammarError

`jsgf.parser.`**`parse_grammar_file`**`(`*path*`)`

Parse a JSGF grammar file and a return a `Grammar` object with the defined attributes, name, imports and rules.

This method will not attempt to import rules or grammars defined in other files, that should be done by an import resolver, not a parser.

Parameters **`path`** – str

Returns Grammar

Raises ParseException, GrammarError

`jsgf.parser.`**`parse_grammar_string`**`(`*s*`)`

Parse a JSGF grammar string and return a `Grammar` object with the defined attributes, name, imports and rules.

Parameters **`s`** – str

Returns Grammar

Raises ParseException, GrammarError

`jsgf.parser.`**`parse_rule_string`**`(`*s*`)`

Parse a string containing a JSGF rule definition and return a `Rule` object.

Parameters **`s`** – str

Returns Rule

Raises ParseException, GrammarError

`jsgf.parser.`**`valid_grammar`**`(`*s*`)`

Whether a string is a valid JSGF grammar string.

Note that this method will not return False for grammars that are otherwise valid, but have out-of-scope imports.

Parameters **`s`** – str

Returns bool

2.6 `references` — References module

This module contains the base class for referencing rules and grammars by name.

2.6.1 Classes

`class` `jsgf.references.`**`BaseRef`**`(`*name*`)`

Base class for JSGF rule and grammar references.

`name`

The referenced name.

Returns str

`static valid``(`*name*`)`

Static method for checking if a reference name is valid.

This should be overwritten appropriately in subclasses.

Parameters **`name`** – str

Returns bool

2.7 `rules` — Rule classes module

This module contains classes for compiling and matching JSpeech Grammar Format rules.

2.7.1 Classes

`class` `jsgf.rules.`**`Rule`**(*name*, *visible*, *expansion*, *case_sensitive=False*)

Base class for JSGF rules.

Rule names can be a single word containing one or more alphanumeric Unicode characters and/or any of the following special characters: +-:;,=|/()[]@#%!^&~$

For example, the following are valid rule names:

- hello
- Zürich
- user_test
- $100
- 1+2=3

There are two reserved rule names: NULL and VOID. These reserved names cannot be used as rule names. You can however change the case to 'null' or 'void' to use them, as names are case-sensitive.

Parameters

- **`name`** – str
- **`visible`** – bool
- **`expansion`** – a string or Expansion object
- **`case_sensitive`** – whether rule literals should be case sensitive (default False).

`active`

Whether this rule is enabled or not. If it is, the rule can be matched and compiled, otherwise the `compile` and `matches` methods will return "" and False respectively.

Returns bool

`case_sensitive`

Case sensitivity used when matching and compiling `Literal` rule expansions.

This property can be `True` or `False`. Matching and compilation will be *case-sensitive* if `True` and *case-insensitive* if `False`. The default value is `False`.

Setting this property will override the `case_sensitive` value for each `Literal` in the rule and in referenced rules.

Return type bool

Returns literal case sensitivity

`compile`()

Compile this rule's expansion tree and return the result.

Returns str

dependencies

The set of rules which this rule directly and indirectly references.

Returns set

dependent_rules

The set of rules in this rule's grammar that reference this rule. Returns an empty set if this rule is not in a grammar.

Returns set

disable()

Stop this rule from producing compile output or from matching speech strings.

enable()

Allow this rule to produce compile output and to match speech strings.

expansion

This rule's expansion.

Returns Expansion

find_matching_part(*speech*)

Searches for a part of speech that matches this rule and returns it.

If no part matches or the rule is disabled, return None.

Parameters **speech** – str

Returns str | None

fully_qualified_name

This rule's fully qualified name.

Fully-qualified rule names are the grammar name plus the rule name. For example, if `"com.example.grammar"` is the grammar name and `"rule"` is the rule name, then `"com.example.grammar.rule"` is the fully-qualified name.

generate()

Generate a string matching this rule.

Return type str

get_tags_matching(*speech*)

Match a speech string and return a list of any matching tags in this rule and in any referenced rules.

Parameters **speech** – str

Returns list

has_tag(*tag*)

Check whether there are expansions in this rule or referenced rules that use a given JSGF tag.

Parameters **tag** – str

Returns bool

matched_tags

A list of JSGF tags whose expansions have been matched. The returned list will be in the order in which tags appear in the compiled rule.

This includes matching tags in referenced rules.

Returns list

`matches`(*speech*)

Whether speech matches this rule.

Matching ambiguous rule expansions is **not supported** because it not worth the performance hit. Ambiguous rule expansions are defined as some optional literal x followed by a required literal x. For example, successfully matching `'test'` for the following rule is not supported:

```
<rule> = [test] test;
```

Parameters **`speech`** – str

Returns bool

`qualified_name`

This rule's qualified name.

Qualified rule names are the last part of the grammar name plus the rule name. For example, if `"com.example.grammar"` is the full grammar name and `"rule"` is the rule name, then `"grammar.rule"` is the qualified name.

`reference_count`

The number of dependent rules.

Returns int

`tags`

A list of JSGF tags used by this rule and any referenced rules. The returned list will be in the order in which tags appear in the compiled rule.

Returns list

`was_matched`

Whether this rule matched last time the `matches` method was called.

Returns bool

class `jsgf.rules.`**`PublicRule`**(*name*, *expansion*, *case_sensitive=False*)

Rule subclass with `visible` set to True.

`jsgf.rules.`**`HiddenRule`**

alias of `jsgf.rules.PrivateRule`

CHAPTER 3

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 starting with version 1.1.1.

3.1 1.9.0 – 2020-04-07

3.1.1 Added

- Add JSGF import resolution.
- Add new Grammar methods for getting Rule objects by name.
- Add ‘grammar_name’, ‘rule_name’ and ‘wildcard_import’ Import class properties.
- Add ‘qualified_name’ and ‘fully_qualified_name’ Rule class properties.

3.1.2 Changed

- Add missing Grammar ‘import_names’ property.
- Add missing Grammar ‘remove_imports()’ method.
- Change Grammar class to silently reject duplicate Import objects.
- Rename HiddenRule rule classes to PrivateRule instead and leave in HiddenRule aliases.

3.1.3 Fixed

- Change Grammar.get_rule_from_name() method to validate the ‘name’ parameter.
- Change Grammar.remove_import() to only accept Import objects.

- Fix bug in Grammar.add_rule() that could cause RecursionErrors.

3.2 1.8.0 – 2020-01-31

3.2.1 Added

- Add 'case_sensitive' properties to Literal, Rule & Grammar classes.

3.2.2 Changed

- Change ChildList into a list wrapper class instead of a sub-class.

3.2.3 Fixed

- Fix a pyparsing-related bug with the Repeat expansion class.
- Fix issues preventing serialization of expansions, rules and grammars.

3.3 1.7.1 – 2019-07-10

3.3.1 Added

- Add 'matching benchmark.py' script.

3.3.2 Changed

- Add a classifier for Python 3.4 in setup.py as it is a supported version.
- Rewrite Expansion.repetition_ancestor property.
- Use the setuptools.find_packages() function in setup.py instead of defining packages manually.

3.3.3 Fixed

- Fix missing call to reset_for_new_match() in Rule.find_matching_part(). Some tests have also been added.

3.4 1.7.0 – 2019-06-19

3.4.1 Added

- Add Expansion and Rule *generate()* methods for generating matching strings. Thanks @embie27.
- Add unit tests for *generate()* methods into a new *test/test_generators.py* file. Thanks @embie27.

3.4.2 Changed

- Include the *mock* package in *requirements.txt* (test requirement).

3.4.3 Fixed

- Fix two rule expansion parser bugs related to unary operators (+ or *).
- Keep required groupings during parsing to avoid unexpected consequences. Thanks @synesthesiam.

3.5 1.6.0 – 2019-03-17

3.5.1 Added

- Add support for JSGF alternative weights to the AlternativeSet class and the rule expansion parser.
- Add 'Expansion.matching_slice' property for getting the slice of the last speech string that matched an Expansion.
- Add 'Repeat.get_expansion_slices' method for getting matching slices of repeated rule expansions.

3.5.2 Changed

- Change AlternativeSet 'weights' list to use a dictionary instead.
- Change grammar class and parser to allow for optional grammar header values.
- Change input and output of the 'save_current_matches' and 'restore_current_matches' expansion functions.
- Change jsgf.ext.Dictation expansions to compile as "<DICTATION>" instead of "".
- Simplify the parser code and improve its performance.
- Use '<NULL>' instead of '<VOID>' to compile expansions that should have children but don't.

3.5.3 Fixed

- Fix parser bug where sequences can lose tags. Thanks @synesthesiam.
- Fix parser bug with sequences and alternative sets.

3.6 1.5.1 – 2018-10-28

3.6.1 Added

- Add section in parser documentation with EBNF for the grammar parser.

3.6.2 Changed

- Change install instructions to use pip instead.

3.6.3 Fixed

- Fix a few problems with the README.
- Fix missing newlines from Grammar.compile_to_file(). Thanks @daanzu.

3.7 1.5.0 – 2018-09-11

3.7.1 Added

- Add Expansion.matcher_element property.
- Add Expansion.invalidate_matcher method.
- Add Rule.find_matching_part method. Thanks @embie27.
- Add docstrings to undocumented classes and methods.
- Add Sphinx documentation project files in *docs/* and use autodoc for automatic module, class, class member and function documentation.
- Add *CHANGELOG.rst* file and include it in the documentation.

3.7.2 Changed

- Make speech string matching scale to large rules/grammars.
- Make jsgf.ext.Dictation expansions match correctly in most circumstances.
- Allow rules to use optional only rule expansions.
- Update docstrings in all Python modules.
- Change internal matching method to implement for subclasses from _matches_internal to _make_matcher_element.

3.7.3 Deprecated

- Add deprecation note for the Grammar.compile_grammar method.
- Deprecate the ExpansionError and MatchError classes.

3.7.4 Fixed

- Fix issue #12 and probably some other bugs where speech wouldn't match rules properly.
- Fix __hash__ methods for the Dictation and AlternativeSet classes.

3.7.5 Removed

- Remove support for matching ambiguous rule expansion because it is not worth the performance hit.

3.8 1.4.1 – 2018-08-20

3.8.1 Added

- Add ChildList list subclass for storing rule expansion children and updating parent-child relationships appropriately on list operations.

3.8.2 Changed

- Change Literal.text attribute into a property with some validation.

3.8.3 Fixed

- Fix AlternativeSet bug with parser (issue #9). Thanks @embie27.

3.9 1.4.0 – 2018-08-09

3.9.1 Added

- Implement grammar, rule and expansion parsers.
- Add setters for the BaseRef name property and Expansion children property.

3.9.2 Changed

- Allow imported rule names to be used by NamedRuleRefs.

3.9.3 Fixed

- Fix NamedRuleRefs for rule expansion functions and the Rule.dependencies property.

3.10 1.3.0 – 2018-07-14

3.10.1 Added

- Add methods/properties to the Rule and Grammar classes for JSGF tag support.
- Add rule resolution for NamedRuleRef class.
- Add method and property for checking expansion match values for each repetition.

3.10.2 Fixed

- Fix various bugs with JSGF rule expansions.

3.11 1.2.3 – 2018-06-02

3.11.1 Added

- Add 'six' as a required package to support Python versions 2.x and 3.x.

3.11.2 Changed

- Change add_rule methods of grammar classes to silently fail when adding rules that are already in grammars.

3.11.3 Fixed

- Fix hash implementations and __str__ methods for rule classes.
- Other minor fixes.

3.12 1.2.2 – 2018-04-28

3.12.1 Added

- Add Expansion.collect_leaves method.

3.12.2 Changed

- Reset match data for unmatched branches of expansion trees.
- Change Expansion leaf properties to also return RuleRefs.
- Move some Literal class properties to the Expansion superclass.

3.13 1.2.1 – 2018-04-27

3.13.1 Added

- Add calculation caching to improve matching performance.
- Add optional shallow parameter to Expansion functions like map_expansion.

3.13.2 Fixed

- Fix bug with BaseRef/RuleRef comparison.
- Fix bug in expand_dictation_expansion function.

3.14 1.2.0 – 2018-04-09

3.14.1 Added

- Add a few methods and properties to Expansion classes.
- Add JointTreeContext class and find_expansion function.
- Add __rep__ methods to base classes for convenience.

3.14.2 Fixed

- Fix a bug where rules with mutiple RuleRefs wouldn't match.

3.15 1.1.1 – 2018-03-26

First tagged release and start of proper versioning. Too many changes to list here, see the changes by following the link above.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index