Dobby Documentation

Release 0.1

Antoine Bertin

April 09, 2014

Dobby is a voice commanded bot that will serve you the best he can by executing commands that you can customize in the database. Dobby-Qt (not developed yet) will provide a nice interface to build your scenarios and help you configuring Dobby

CHAPTER 1

Usage

To display Dobby.py usage, you can type `./Dobby.py --help`:

```
usage: Dobby.py [-h] [-d] [-p PID_FILE] [-c CONFIG_FILE] [--list-devices]
                [--data-dir DATA_DIR] [-q | -v] [--version]

Your servant

optional arguments:
  -h, --help            show this help message and exit
  -d, --daemon          run as daemon
  -p PID_FILE, --pid-file PID_FILE
                        create pid file
  -c CONFIG_FILE, --config-file CONFIG_FILE
                        config file to use
  --list-devices        list available devices and exit
  --data-dir DATA_DIR   data directory to store cache, config, logs and
                        database
  -q, --quiet           disable console output
  -v, --verbose         verbose console output
  --version             show program's version number and exit
```

You can run Dobby once to have default `config.ini` and `dobby.db` generated Dobby requires you to have a well-configured speech-dispatcher and a running julius module server

CHAPTER 2

API Documentation

2.1 Triggers

Triggers are used to trigger events when an audio input is received.

2.1.1 Base

Trigger

class dobby.triggers.**Trigger**(*event_queue*)

Threaded Trigger base class. A trigger will `raise` a `RecognitionEvent` or an `CommandEvent` when the detection is successful

Parameters **event_queue** (*Queue.Queue*) – queue where to put the events

raise_event(*event*)

Raise an event in the `event_queue`

stop()

Stop the thread

Events

Events are raised by triggers to communicate

class dobby.triggers.**RecognitionEvent**

A RecognitionEvent indicates that the recognition can be launched to analyze the next voice command

2.1.2 Clapper

The Clapper is used to match a `Sequence` of `Blocks`. The `Pattern` is composed of `PatternItems` that will validates or not, depending on its parameters, a specific kind of `Block`.

Sequence

Pattern

Clapper

2.1.3 Julius

class dobby.triggers.julius.**Julius**(*event_queue*, *command*, *recognizer*, *action*)
Bases: dobby.triggers.Trigger

Analyze an audio source and put an event in the queue if voice command is spoken

Parameters

- **command** (*string*) – voice command to match
- **recognizer** (*Recognizer*) – Julius Recognizer instance
- **action** (*boolean*) – whether to fire CommandEvents or not

2.2 Recognizers

Recognizers are simple interfaces to speech recognition softwares

2.2.1 Base

class dobby.recognizers.**Recognizer**
Bases: threading.Thread

Threaded Recognizer base class. A queue can be subscribed to the Recognizer and hence, receive recognized pyjulius.Sentence objects

subscribers
List of subscribers

publish(*sentence*)
Publish a recognized sentence to all subscribers

Parameters **sentence** (*pyjulius.Sentence*) – the recognized sentence

---

Note: The type of the *sentence* parameter may change in the near future when a new Recognizer will be added

---

stop()
Stop the thread

subscribe(*subscriber*)
Add a queue to the Recognizer's subscribers list. A subscriber will receive published pyjulius.Sentence objects

Parameters **subscriber** (*Queue.Queue*) – subscriber to append

unsubscribe(*subscriber*)
Remove a queue from the subscribers list

Parameters **subscriber** (*Queue.Queue*) – subscriber to remove

2.2.2 Julius

class dobby.recognizers.julius.**Julius**(*host*, *port*, *encoding*, *min_score*)
Bases: dobby.recognizers.Recognizer

Julius Recognizer is based on Julius speech recognition engine. It uses pyjulius to connect to julius instance running in module mode

Parameters

- **host** (*string*) – host of the server
- **port** (*integer*) – port of the server
- **encoding** (*string*) – encoding used to decode server's output
- **min_score** (*float*) – minimum score under which the recognition result will be ignored

run()
Run the recognition and publish() the recognized pyjulius.Sentence objects

2.3 Models

Models are connected to the database with SQLAlchemy but can also contain their own logic

2.3.1 Sentence

2.3.2 Association

class dobby.models.association.**Association**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Association model that represents an association between a Scenario and a Action with a special order

Parameters ****kwargs** – can set all attributes

scenario_id
Foreing key to Scenario.id

action_id
Foreing key to Action.id

order
Order in which the Action objects of a Scenario should be executed

scenario
Direct access to the Scenario object

action
Direct access to the Action object

dobby.models.association.**relationship**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign_keys=None*, *uselist=None*, *order_by=False*, *backref=None*, *back_populates=None*, *post_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection_class=None*, *passive_deletes=False*, *passive_updates=True*, *remote_side=None*, *enable_typechecks=True*, *join_depth=None*, *comparator_factory=None*, *single_parent=False*, *innerjoin=False*, *distinct_target_key=None*, *doc=None*, *active_history=False*, *cascade_backrefs=True*, *load_on_pending=False*, *strategy_class=None*, *_local_remote_pairs=None*, *query_class=None*, *info=None*)

Provide a relationship between two mapped classes.

This corresponds to a parent-child or associative table relationship. The constructed class is an instance of `RelationshipProperty`.

A typical `relationship()`, used in a classical mapping:

```
mapper(Parent, properties={
  'children': relationship(Child)
})
```

Some arguments accepted by `relationship()` optionally accept a callable function, which when called produces the desired value. The callable is invoked by the parent `Mapper` at "mapper initialization" time, which happens only when mappers are first used, and is assumed to be after all mappings have been constructed. This can be used to resolve order-of-declaration and other dependency issues, such as if `Child` is declared below `Parent` in the same file:

```
mapper(Parent, properties={
    "children":relationship(lambda: Child,
                        order_by=lambda: Child.id)
})
```

When using the *declarative_toplevel* extension, the Declarative initializer allows string arguments to be passed to `relationship()`. These string arguments are converted into callables that evaluate the string as Python code, using the Declarative class-registry as a namespace. This allows the lookup of related classes to be automatic via their string name, and removes the need to import related classes at all into the local module space:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", order_by="Child.id")
```

See also:

relationship_config_toplevel - Full introductory and reference documentation for `relationship()`.

orm_tutorial_relationship - ORM tutorial introduction.

Parameters

- **argument** – a mapped class, or actual `Mapper` instance, representing the target of the relationship.

  :paramref:'~.relationship.argument' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  declarative_configuring_relationships - further detail on relationship configuration when using Declarative.

- **secondary** – for a many-to-many relationship, specifies the intermediary table, and is typically an instance of `Table`. In less common circumstances, the argument may also be specified as an `Alias` construct, or even a `Join` construct.

  :paramref:'~.relationship.secondary' may also be passed as a callable function which is evaluated at mapper initialization time. When using Declarative, it may also be a string argument noting the name of a `Table` that is present in the `MetaData` collection associated with the parent-mapped `Table`.

  The **:paramref:'~.relationship.secondary'** keyword argument is typically applied in the case where the intermediary `Table` is not otherwise exprssed in any direct class mapping. If the "secondary" table is also explicitly mapped elsewhere (e.g. as in *association_pattern*), one should consider applying the **:paramref:'~.relationship.viewonly'** flag so that this `relationship()` is not used for persistence operations which may conflict with those of the association object pattern.

  See also:

  relationships_many_to_many - Reference example of "many to many".

  orm_tutorial_many_to_many - ORM tutorial introduction to many-to-many relationships.

  self_referential_many_to_many - Specifics on using many-to-many in a self-referential case.

  declarative_many_to_many - Additional options when using Declarative.

  association_pattern - an alternative to **:paramref:'~.relationship.secondary'** when composing association table relationships, allowing additional attributes to be specified on the association table.

  composite_secondary_join - a lesser-used pattern which in some cases can enable complex `relationship()` SQL conditions to be used.

  New in version 0.9.2: **:paramref:'~.relationship.secondary'** works more effectively when referring to a `Join` instance.

- **active_history=False** – When `True`, indicates that the "previous" value for a many-to-one reference should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple many-to-ones only needs to be aware of the "new" value in order to perform a flush. This flag is available for applications that make use of `attributes.get_history()` which also need to know the "previous" value of the attribute.

- **backref** – indicates the string name of a property to be placed on the related mapper's class that will handle this relationship in the other direction. The other property will be created automatically when the mappers are configured. Can also be passed as a `backref()` object to control the configuration of the new relationship.

  See also:

  relationships_backref - Introductory documentation and examples.

:paramref:'~.relationship.back_populates' - alternative form of backref specification.

`backref()` - allows control over `relationship()` configuration when using **:paramref:'~.relationship.backref'**.

- **back_populates** – Takes a string name and has the same meaning as **:paramref:'~.relationship.backref'**, except the complementing property is **not** created automatically, and instead must be configured explicitly on the other mapper. The complementing property should also indicate **:paramref:'~.relationship.back_populates'** to this relationship to ensure proper functioning.

  See also:

  relationships_backref - Introductory documentation and examples.

  :paramref:'~.relationship.backref' - alternative form of backref specification.

- **cascade** – a comma-separated list of cascade rules which determines how Session operations should be "cascaded" from parent to child. This defaults to `False`, which means the default cascade should be used - this default cascade is `"save-update, merge"`.

  The available cascades are `save-update`, `merge`, `expunge`, `delete`, `delete-orphan`, and `refresh-expire`. An additional option, `all` indicates shorthand for `"save-update, merge, refresh-expire, expunge, delete"`, and is often used as in `"all, delete-orphan"` to indicate that related objects should follow along with the parent object in all cases, and be deleted when de-associated.

  See also:

  unitofwork_cascades - Full detail on each of the available cascade options.

  tutorial_delete_cascade - Tutorial example describing a delete cascade.

- **cascade_backrefs=True** – a boolean value indicating if the `save-update` cascade should operate along an assignment event intercepted by a backref. When set to `False`, the attribute managed by this relationship will not cascade an incoming transient object into the session of a persistent parent, if the event is received via backref.

  See also:

  backref_cascade - Full discussion and examples on how the **:paramref:'~.relationship.cascade_backrefs'** option is used.

- **collection_class** – a class or callable that returns a new list-holding object. will be used in place of a plain list for storing elements.

  See also:

  custom_collections - Introductory documentation and examples.

- **comparator_factory** – a class which extends `RelationshipProperty.Comparator` which provides custom SQL clause generation for comparison operations.

  See also:

  `PropComparator` - some detail on redefining comparators at this level.

  custom_comparators - Brief intro to this feature.

- **distinct_target_key=None** – Indicate if a "subquery" eager load should apply the DISTINCT keyword to the innermost SELECT statement. When left as `None`, the DISTINCT keyword will be applied in those cases when the target columns do not comprise the full primary key of the target table. When set to `True`, the DISTINCT keyword is applied to the innermost SELECT unconditionally.

It may be desirable to set this flag to False when the DISTINCT is reducing performance of the innermost subquery beyond that of what duplicate innermost rows may be causing.

New in version 0.8.3: - **:paramref:'~.relationship.distinct_target_key'** allows the subquery eager loader to apply a DISTINCT modifier to the innermost SELECT.

Changed in version 0.9.0: - **:paramref:'~.relationship.distinct_target_key'** now defaults to `None`, so that the feature enables itself automatically for those cases where the innermost query targets a non-unique key.

See also:

loading_toplevel - includes an introduction to subquery eager loading.

- **doc** – docstring which will be applied to the resulting descriptor.
- **extension** – an `AttributeExtension` instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class.

  Deprecated since version 0.7: Please see `AttributeEvents`.

- **foreign_keys** – a list of columns which are to be used as "foreign key" columns, or columns which refer to the value in a remote column, within the context of this `relationship()` object's **:paramref:'~.relationship.primaryjoin'** condition. That is, if the **:paramref:'~.relationship.primaryjoin'** condition of this `relationship()` is `a.id == b.a_id`, and the values in `b.a_id` are required to be present in `a.id`, then the "foreign key" column of this `relationship()` is `b.a_id`.

  In normal cases, the **:paramref:'~.relationship.foreign_keys'** parameter is **not required.** `relationship()` will automatically determine which columns in the **:paramref:'~.relationship.primaryjoin'** conditition are to be considered "foreign key" columns based on those `Column` objects that specify `ForeignKey`, or are otherwise listed as referencing columns in a `ForeignKeyConstraint` construct. **:paramref:'~.relationship.foreign_keys'** is only needed when:

  1. There is more than one way to construct a join from the local table to the remote table, as there are multiple foreign key references present. Setting `foreign_keys` will limit the `relationship()` to consider just those columns specified here as "foreign".

     Changed in version 0.8: A multiple-foreign key join ambiguity can be resolved by setting the **:paramref:'~.relationship.foreign_keys'** parameter alone, without the need to explicitly set **:paramref:'~.relationship.primaryjoin'** as well.

  2. The `Table` being mapped does not actually have `ForeignKey` or `ForeignKeyConstraint` constructs present, often because the table was reflected from a database that does not support foreign key reflection (MySQL MyISAM).

  3. The **:paramref:'~.relationship.primaryjoin'** argument is used to construct a non-standard join condition, which makes use of columns or expressions that do not normally refer to their "parent" column, such as a join condition expressed by a complex comparison using a SQL function.

  The `relationship()` construct will raise informative error messages that suggest the use of the **:paramref:'~.relationship.foreign_keys'** parameter when presented with an ambiguous condition. In typical cases, if `relationship()` doesn't raise any exceptions, the **:paramref:'~.relationship.foreign_keys'** parameter is usually not needed.

  :paramref:'~.relationship.foreign_keys' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

relationship_foreign_keys

relationship_custom_foreign

`foreign()` - allows direct annotation of the "foreign" columns within a **:paramref:'~.relationship.primaryjoin'** condition.

New in version 0.8: The `foreign()` annotation can also be applied directly to the **:paramref:'~.relationship.primaryjoin'** expression, which is an alternate, more specific system of describing which columns in a particular **:paramref:'~.relationship.primaryjoin'** should be considered "foreign".

- **info** – Optional data dictionary which will be populated into the `MapperProperty.info` attribute of this object.

  New in version 0.8.

- **innerjoin=False** – when `True`, joined eager loads will use an inner join to join against related tables instead of an outer join. The purpose of this option is generally one of performance, as inner joins generally perform better than outer joins.

  This flag can be set to `True` when the relationship references an object via many-to-one using local foreign keys that are not nullable, or when the reference is one-to-one or a collection that is guaranteed to have one or at least one entry.

  If the joined-eager load is chained onto an existing LEFT OUTER JOIN, `innerjoin=True` will be bypassed and the join will continue to chain as LEFT OUTER JOIN so that the results don't change. As an alternative, specify the value `"nested"`. This will instead nest the join on the right side, e.g. using the form "a LEFT OUTER JOIN (b JOIN c)".

  New in version 0.9.4: Added `innerjoin="nested"` option to support nesting of eager "inner" joins.

  See also:

  what_kind_of_loading - Discussion of some details of various loader options.

  :paramref:'.joinedload.innerjoin' - loader option version

- **join_depth** – when non-`None`, an integer value indicating how many levels deep "eager" loaders should join on a self-referring or cyclical relationship. The number counts how many times the same Mapper shall be present in the loading condition along a particular join branch. When left at its default of `None`, eager loaders will stop chaining when they encounter a the same target mapper which is already higher up in the chain. This option applies both to joined- and subquery- eager loaders.

  See also:

  self_referential_eager_loading - Introductory documentation and examples.

- **lazy='select'** – specifies how the related items should be loaded. Default value is `select`. Values include:

  - `select` - items should be loaded lazily when the property is first accessed, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
  - `immediate` - items should be loaded as the parents are loaded, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
  - `joined` - items should be loaded "eagerly" in the same query as that of the parent, using a JOIN or LEFT OUTER JOIN. Whether the join is "outer" or not is determined by the **:paramref:'~.relationship.innerjoin'** parameter.

- `subquery` - items should be loaded "eagerly" as the parents are loaded, using one additional SQL statement, which issues a JOIN to a subquery of the original statement, for each collection requested.
- `noload` - no loading should occur at any time. This is to support "write-only" attributes, or attributes which are populated in some manner specific to the application.
- `dynamic` - the attribute will return a pre-configured `Query` object for all read operations, onto which further filtering operations can be applied before iterating the results. See the section *dynamic_relationship* for more details.
- True - a synonym for 'select'
- False - a synonym for 'joined'
- None - a synonym for 'noload'

See also:

`/orm/loading` - Full documentation on relationship loader configuration.

dynamic_relationship - detail on the `dynamic` option.

- **load_on_pending=False** – Indicates loading behavior for transient or pending parent objects.

  When set to `True`, causes the lazy-loader to issue a query for a parent object that is not persistent, meaning it has never been flushed. This may take effect for a pending object when autoflush is disabled, or for a transient object that has been "attached" to a `Session` but is not part of its pending collection.

  The **:paramref:'~.relationship.load_on_pending'** flag does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before a flush proceeds. This flag is not not intended for general use.

  See also:

  `Session.enable_relationship_loading()` - this method establishes "load on pending" behavior for the whole object, and also allows loading on objects that remain transient or detached.

- **order_by** – indicates the ordering that should be applied when loading these items. **:paramref:'~.relationship.order_by'** is expected to refer to one of the `Column` objects to which the target class is mapped, or the attribute itself bound to the target class which refers to the column.

  :paramref:'~.relationship.order_by' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

- **passive_deletes=False** – Indicates loading behavior during delete operations.

  A value of True indicates that unloaded child items should not be loaded during a delete operation on the parent. Normally, when a parent item is deleted, all child items are loaded so that they can either be marked as deleted, or have their foreign key to the parent set to NULL. Marking this flag as True usually implies an ON DELETE <CASCADE|SET NULL> rule is in place which will handle updating/deleting child rows on the database side.

  Additionally, setting the flag to the string value 'all' will disable the "nulling out" of the child foreign keys, when there is no delete or delete-orphan cascade enabled. This is typically used when a triggering or error raise scenario is in place on the database side. Note that the

foreign key attributes on in-session child objects will not be changed after a flush occurs so this is a very special use-case setting.

See also:

passive_deletes - Introductory documentation and examples.

- **passive_updates=True** – Indicates loading and INSERT/UPDATE/DELETE behavior when the source of a foreign key value changes (i.e. an "on update" cascade), which are typically the primary key columns of the source row.

  When True, it is assumed that ON UPDATE CASCADE is configured on the foreign key in the database, and that the database will handle propagation of an UPDATE from a source column to dependent rows. Note that with databases which enforce referential integrity (i.e. PostgreSQL, MySQL with InnoDB tables), ON UPDATE CASCADE is required for this operation. The relationship() will update the value of the attribute on related items which are locally present in the session during a flush.

  When False, it is assumed that the database does not enforce referential integrity and will not be issuing its own CASCADE operation for an update. The relationship() will issue the appropriate UPDATE statements to the database in response to the change of a referenced key, and items locally present in the session during a flush will also be refreshed.

  This flag should probably be set to False if primary key changes are expected and the database in use doesn't support CASCADE (i.e. SQLite, MySQL MyISAM tables).

  See also:

  passive_updates - Introductory documentation and examples.

  :paramref:'.mapper.passive_updates' - a similar flag which takes effect for joined-table inheritance mappings.

- **post_update** – this indicates that the relationship should be handled by a second UPDATE statement after an INSERT or before a DELETE. Currently, it also will issue an UPDATE after the instance was UPDATEd as well, although this technically should be improved. This flag is used to handle saving bi-directional dependencies between two individual rows (i.e. each row references the other), where it would otherwise be impossible to INSERT or DELETE both rows fully since one row exists before the other. Use this flag when a particular mapping arrangement will incur two rows that are dependent on each other, such as a table that has a one-to-many relationship to a set of child rows, and also has a column that references a single child row within that list (i.e. both tables contain a foreign key to each other). If a flush operation returns an error that a "cyclical dependency" was detected, this is a cue that you might want to use **:paramref:'~.relationship.post_update'** to "break" the cycle.

  See also:

  post_update - Introductory documentation and examples.

- **primaryjoin** – a SQL expression that will be used as the primary join of this child object against the parent object, or in a many-to-many relationship the join of the primary object to the association table. By default, this value is computed based on the foreign key relationships of the parent and child tables (or association table).

  :paramref:'~.relationship.primaryjoin' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  relationship_primaryjoin

- **remote_side** – used for self-referential relationships, indicates the column or list of columns that form the "remote side" of the relationship.

  :paramref:'.relationship.remote_side' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  Changed in version 0.8: The `remote()` annotation can also be applied directly to the `primaryjoin` expression, which is an alternate, more specific system of describing which columns in a particular `primaryjoin` should be considered "remote".

  See also:

  self_referential - in-depth explaination of how **:paramref:'~.relationship.remote_side'** is used to configure self-referential relationships.

  `remote()` - an annotation function that accomplishes the same purpose as **:paramref:'~.relationship.remote_side'**, typically when a custom **:paramref:'~.relationship.primaryjoin'** condition is used.

- **query_class** – a `Query` subclass that will be used as the base of the "appender query" returned by a "dynamic" relationship, that is, a relationship that specifies `lazy="dynamic"` or was otherwise constructed using the `orm.dynamic_loader()` function.

  See also:

  dynamic_relationship - Introduction to "dynamic" relationship loaders.

- **secondaryjoin** – a SQL expression that will be used as the join of an association table to the child object. By default, this value is computed based on the foreign key relationships of the association and child tables.

  :paramref:'~.relationship.secondaryjoin' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  relationship_primaryjoin

- **single_parent** – when True, installs a validator which will prevent objects from being associated with more than one parent at a time. This is used for many-to-one or many-to-many relationships that should be treated either as one-to-one or one-to-many. Its usage is optional, except for `relationship()` constructs which are many-to-one or many-to-many and also specify the `delete-orphan` cascade option. The `relationship()` construct itself will raise an error instructing when this option is required.

  See also:

  unitofwork_cascades - includes detail on when the **:paramref:'~.relationship.single_parent'** flag may be appropriate.

- **uselist** – a boolean that indicates if this property should be loaded as a list or a scalar. In most cases, this value is determined automatically by `relationship()` at mapper configuration time, based on the type and direction of the relationship - one to many forms a list, many to one forms a scalar, many to many is a list. If a scalar is desired where normally a list would be present, such as a bi-directional one-to-one relationship, set **:paramref:'~.relationship.uselist'** to False.

  The **:paramref:'~.relationship.uselist'** flag is also available on an existing `relationship()` construct as a read-only attribute, which can be used to determine if this `relationship()` deals with collections or scalar attributes:

```
>>> User.addresses.property.uselist
True
```

See also:

relationships_one_to_one - Introduction to the "one to one" relationship pattern, which is typically when the **:paramref:'~.relationship.uselist'** flag is needed.

- **viewonly=False** – when set to True, the relationship is used only for loading objects, and not for any persistence operation. A `relationship()` which specifies **:paramref:'~.relationship.viewonly'** can work with a wider range of SQL operations within the **:paramref:'~.relationship.primaryjoin'** condition, including operations that feature the use of a variety of comparison operators as well as SQL functions such as `cast()`. The **:paramref:'~.relationship.viewonly'** flag is also of general use when defining any kind of `relationship()` that doesn't represent the full set of related objects, to prevent modifications of the collection from resulting in persistence operations.

2.3.3 Actions

Base

class dobby.models.actions.**Action**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Action base model that holds the text-to-speech

Parameters ****kwargs** – can set all attributes

id
Action id

tts
To-be-formatted or formatted text-to-speech

associations
Link to a list of `Association` objects by following the database relationship

format_tts()
Format the `tts` into a valid text-to-speech for `TTS`

dobby.models.actions.**relationship**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign_keys=None*, *uselist=None*, *order_by=False*, *backref=None*, *back_populates=None*, *post_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection_class=None*, *passive_deletes=False*, *passive_updates=True*, *remote_side=None*, *enable_typechecks=True*, *join_depth=None*, *comparator_factory=None*, *single_parent=False*, *innerjoin=False*, *distinct_target_key=None*, *doc=None*, *active_history=False*, *cascade_backrefs=True*, *load_on_pending=False*, *strategy_class=None*, *_local_remote_pairs=None*, *query_class=None*, *info=None*)
Provide a relationship between two mapped classes.

This corresponds to a parent-child or associative table relationship. The constructed class is an instance of `RelationshipProperty`.

A typical `relationship()`, used in a classical mapping:

```
mapper(Parent, properties={
  'children': relationship(Child)
})
```

Some arguments accepted by `relationship()` optionally accept a callable function, which when called produces the desired value. The callable is invoked by the parent `Mapper` at "mapper initialization" time, which happens only when mappers are first used, and is assumed to be after all mappings have been constructed. This can be used to resolve order-of-declaration and other dependency issues, such as if `Child` is declared below `Parent` in the same file:

```
mapper(Parent, properties={
    "children":relationship(lambda: Child,
                        order_by=lambda: Child.id)
})
```

When using the *declarative_toplevel* extension, the Declarative initializer allows string arguments to be passed to `relationship()`. These string arguments are converted into callables that evaluate the string as Python code, using the Declarative class-registry as a namespace. This allows the lookup of related classes to be automatic via their string name, and removes the need to import related classes at all into the local module space:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", order_by="Child.id")
```

See also:

relationship_config_toplevel - Full introductory and reference documentation for `relationship()`.

orm_tutorial_relationship - ORM tutorial introduction.

Parameters

- **argument** – a mapped class, or actual `Mapper` instance, representing the target of the relationship.

  :paramref:'~.relationship.argument' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  declarative_configuring_relationships - further detail on relationship configuration when using Declarative.

- **secondary** – for a many-to-many relationship, specifies the intermediary table, and is typically an instance of `Table`. In less common circumstances, the argument may also be specified as an `Alias` construct, or even a `Join` construct.

  :paramref:'~.relationship.secondary' may also be passed as a callable function which is evaluated at mapper initialization time. When using Declarative, it may also be a string argument noting the name of a `Table` that is present in the `MetaData` collection associated with the parent-mapped `Table`.

The **:paramref:‘~.relationship.secondary‘** keyword argument is typically applied in the case where the intermediary `Table` is not otherwise exprssed in any direct class mapping. If the “secondary” table is also explicitly mapped elsewhere (e.g. as in *association_pattern*), one should consider applying the **:paramref:‘~.relationship.viewonly‘** flag so that this `relationship()` is not used for persistence operations which may conflict with those of the association object pattern.

See also:

relationships_many_to_many - Reference example of “many to many”.

orm_tutorial_many_to_many - ORM tutorial introduction to many-to-many relationships.

self_referential_many_to_many - Specifics on using many-to-many in a self-referential case.

declarative_many_to_many - Additional options when using Declarative.

association_pattern - an alternative to **:paramref:‘~.relationship.secondary‘** when composing association table relationships, allowing additional attributes to be specified on the association table.

composite_secondary_join - a lesser-used pattern which in some cases can enable complex `relationship()` SQL conditions to be used.

New in version 0.9.2: **:paramref:‘~.relationship.secondary‘** works more effectively when referring to a `Join` instance.

- **active_history=False** – When `True`, indicates that the “previous” value for a many-to-one reference should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple many-to-ones only needs to be aware of the “new” value in order to perform a flush. This flag is available for applications that make use of `attributes.get_history()` which also need to know the “previous” value of the attribute.
- **backref** – indicates the string name of a property to be placed on the related mapper’s class that will handle this relationship in the other direction. The other property will be created automatically when the mappers are configured. Can also be passed as a `backref()` object to control the configuration of the new relationship.

  See also:

  relationships_backref - Introductory documentation and examples.

  :paramref:‘~.relationship.back_populates‘ - alternative form of backref specification.

  `backref()` - allows control over `relationship()` configuration when using **:paramref:‘~.relationship.backref‘**.

- **back_populates** – Takes a string name and has the same meaning as **:paramref:‘~.relationship.backref‘**, except the complementing property is **not** created automatically, and instead must be configured explicitly on the other mapper. The complementing property should also indicate **:paramref:‘~.relationship.back_populates‘** to this relationship to ensure proper functioning.

  See also:

  relationships_backref - Introductory documentation and examples.

  :paramref:‘~.relationship.backref‘ - alternative form of backref specification.

- **cascade** – a comma-separated list of cascade rules which determines how Session operations should be “cascaded” from parent to child. This defaults to `False`, which means the default cascade should be used - this default cascade is `"save-update, merge"`.

The available cascades are `save-update`, `merge`, `expunge`, `delete`, `delete-orphan`, and `refresh-expire`. An additional option, `all` indicates shorthand for `"save-update, merge, refresh-expire, expunge, delete"`, and is often used as in `"all, delete-orphan"` to indicate that related objects should follow along with the parent object in all cases, and be deleted when de-associated.

See also:

unitofwork_cascades - Full detail on each of the available cascade options.

tutorial_delete_cascade - Tutorial example describing a delete cascade.

- **cascade_backrefs=True** – a boolean value indicating if the `save-update` cascade should operate along an assignment event intercepted by a backref. When set to `False`, the attribute managed by this relationship will not cascade an incoming transient object into the session of a persistent parent, if the event is received via backref.

  See also:

  backref_cascade - Full discussion and examples on how the **:paramref:'~.relationship.cascade_backrefs'** option is used.

- **collection_class** – a class or callable that returns a new list-holding object. will be used in place of a plain list for storing elements.

  See also:

  custom_collections - Introductory documentation and examples.

- **comparator_factory** – a class which extends `RelationshipProperty.Comparator` which provides custom SQL clause generation for comparison operations.

  See also:

  `PropComparator` - some detail on redefining comparators at this level.

  custom_comparators - Brief intro to this feature.

- **distinct_target_key=None** – Indicate if a "subquery" eager load should apply the DISTINCT keyword to the innermost SELECT statement. When left as `None`, the DISTINCT keyword will be applied in those cases when the target columns do not comprise the full primary key of the target table. When set to `True`, the DISTINCT keyword is applied to the innermost SELECT unconditionally.

  It may be desirable to set this flag to False when the DISTINCT is reducing performance of the innermost subquery beyond that of what duplicate innermost rows may be causing.

  New in version 0.8.3: - **:paramref:'~.relationship.distinct_target_key'** allows the subquery eager loader to apply a DISTINCT modifier to the innermost SELECT.

  Changed in version 0.9.0: - **:paramref:'~.relationship.distinct_target_key'** now defaults to `None`, so that the feature enables itself automatically for those cases where the innermost query targets a non-unique key.

  See also:

  loading_toplevel - includes an introduction to subquery eager loading.

- **doc** – docstring which will be applied to the resulting descriptor.

- **extension** – an `AttributeExtension` instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class.

  Deprecated since version 0.7: Please see `AttributeEvents`.

- **foreign_keys** – a list of columns which are to be used as "foreign key" columns, or columns which refer to the value in a remote column, within the context of this `relationship()` object's **:paramref:'~.relationship.primaryjoin'** condition. That is, if the **:paramref:'~.relationship.primaryjoin'** condition of this `relationship()` is `a.id == b.a_id`, and the values in `b.a_id` are required to be present in `a.id`, then the "foreign key" column of this `relationship()` is `b.a_id`.

  In normal cases, the **:paramref:'~.relationship.foreign_keys'** parameter is **not required.** `relationship()` will automatically determine which columns in the **:paramref:'~.relationship.primaryjoin'** conditition are to be considered "foreign key" columns based on those `Column` objects that specify `ForeignKey`, or are otherwise listed as referencing columns in a `ForeignKeyConstraint` construct. **:paramref:'~.relationship.foreign_keys'** is only needed when:

  1. There is more than one way to construct a join from the local table to the remote table, as there are multiple foreign key references present. Setting `foreign_keys` will limit the `relationship()` to consider just those columns specified here as "foreign".

     Changed in version 0.8: A multiple-foreign key join ambiguity can be resolved by setting the **:paramref:'~.relationship.foreign_keys'** parameter alone, without the need to explicitly set **:paramref:'~.relationship.primaryjoin'** as well.

  2. The `Table` being mapped does not actually have `ForeignKey` or `ForeignKeyConstraint` constructs present, often because the table was reflected from a database that does not support foreign key reflection (MySQL MyISAM).

  3. The **:paramref:'~.relationship.primaryjoin'** argument is used to construct a non-standard join condition, which makes use of columns or expressions that do not normally refer to their "parent" column, such as a join condition expressed by a complex comparison using a SQL function.

  The `relationship()` construct will raise informative error messages that suggest the use of the **:paramref:'~.relationship.foreign_keys'** parameter when presented with an ambiguous condition. In typical cases, if `relationship()` doesn't raise any exceptions, the **:paramref:'~.relationship.foreign_keys'** parameter is usually not needed.

  :paramref:'~.relationship.foreign_keys' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  relationship_foreign_keys

  relationship_custom_foreign

  `foreign()` - allows direct annotation of the "foreign" columns within a **:paramref:'~.relationship.primaryjoin'** condition.

  New in version 0.8: The `foreign()` annotation can also be applied directly to the **:paramref:'~.relationship.primaryjoin'** expression, which is an alternate, more specific system of describing which columns in a particular **:paramref:'~.relationship.primaryjoin'** should be considered "foreign".

- **info** – Optional data dictionary which will be populated into the `MapperProperty.info` attribute of this object.

  New in version 0.8.

- **innerjoin=False** – when `True`, joined eager loads will use an inner join to join against

related tables instead of an outer join. The purpose of this option is generally one of performance, as inner joins generally perform better than outer joins.

This flag can be set to `True` when the relationship references an object via many-to-one using local foreign keys that are not nullable, or when the reference is one-to-one or a collection that is guaranteed to have one or at least one entry.

If the joined-eager load is chained onto an existing LEFT OUTER JOIN, `innerjoin=True` will be bypassed and the join will continue to chain as LEFT OUTER JOIN so that the results don't change. As an alternative, specify the value `"nested"`. This will instead nest the join on the right side, e.g. using the form "a LEFT OUTER JOIN (b JOIN c)".

New in version 0.9.4: Added `innerjoin="nested"` option to support nesting of eager "inner" joins.

See also:

what_kind_of_loading - Discussion of some details of various loader options.

:paramref:'.joinedload.innerjoin' - loader option version

- **join_depth** – when non-`None`, an integer value indicating how many levels deep "eager" loaders should join on a self-referring or cyclical relationship. The number counts how many times the same Mapper shall be present in the loading condition along a particular join branch. When left at its default of `None`, eager loaders will stop chaining when they encounter a the same target mapper which is already higher up in the chain. This option applies both to joined- and subquery- eager loaders.

  See also:

  self_referential_eager_loading - Introductory documentation and examples.

- **lazy='select'** – specifies how the related items should be loaded. Default value is `select`. Values include:

  - `select` - items should be loaded lazily when the property is first accessed, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
  - `immediate` - items should be loaded as the parents are loaded, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
  - `joined` - items should be loaded "eagerly" in the same query as that of the parent, using a JOIN or LEFT OUTER JOIN. Whether the join is "outer" or not is determined by the **:paramref:'~.relationship.innerjoin'** parameter.
  - `subquery` - items should be loaded "eagerly" as the parents are loaded, using one additional SQL statement, which issues a JOIN to a subquery of the original statement, for each collection requested.
  - `noload` - no loading should occur at any time. This is to support "write-only" attributes, or attributes which are populated in some manner specific to the application.
  - `dynamic` - the attribute will return a pre-configured `Query` object for all read operations, onto which further filtering operations can be applied before iterating the results. See the section *dynamic_relationship* for more details.
  - True - a synonym for 'select'
  - False - a synonym for 'joined'
  - None - a synonym for 'noload'

See also:

`/orm/loading` - Full documentation on relationship loader configuration.

dynamic_relationship - detail on the `dynamic` option.

- **load_on_pending=False** – Indicates loading behavior for transient or pending parent objects.

  When set to `True`, causes the lazy-loader to issue a query for a parent object that is not persistent, meaning it has never been flushed. This may take effect for a pending object when autoflush is disabled, or for a transient object that has been "attached" to a `Session` but is not part of its pending collection.

  The **:paramref:'~.relationship.load_on_pending'** flag does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before a flush proceeds. This flag is not not intended for general use.

  See also:

  `Session.enable_relationship_loading()` - this method establishes "load on pending" behavior for the whole object, and also allows loading on objects that remain transient or detached.

- **order_by** – indicates the ordering that should be applied when loading these items. **:paramref:'~.relationship.order_by'** is expected to refer to one of the `Column` objects to which the target class is mapped, or the attribute itself bound to the target class which refers to the column.

  :paramref:'~.relationship.order_by' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

- **passive_deletes=False** – Indicates loading behavior during delete operations.

  A value of True indicates that unloaded child items should not be loaded during a delete operation on the parent. Normally, when a parent item is deleted, all child items are loaded so that they can either be marked as deleted, or have their foreign key to the parent set to NULL. Marking this flag as True usually implies an ON DELETE <CASCADE|SET NULL> rule is in place which will handle updating/deleting child rows on the database side.

  Additionally, setting the flag to the string value 'all' will disable the "nulling out" of the child foreign keys, when there is no delete or delete-orphan cascade enabled. This is typically used when a triggering or error raise scenario is in place on the database side. Note that the foreign key attributes on in-session child objects will not be changed after a flush occurs so this is a very special use-case setting.

  See also:

  passive_deletes - Introductory documentation and examples.

- **passive_updates=True** – Indicates loading and INSERT/UPDATE/DELETE behavior when the source of a foreign key value changes (i.e. an "on update" cascade), which are typically the primary key columns of the source row.

  When True, it is assumed that ON UPDATE CASCADE is configured on the foreign key in the database, and that the database will handle propagation of an UPDATE from a source column to dependent rows. Note that with databases which enforce referential integrity (i.e. PostgreSQL, MySQL with InnoDB tables), ON UPDATE CASCADE is required for this

operation. The relationship() will update the value of the attribute on related items which are locally present in the session during a flush.

When False, it is assumed that the database does not enforce referential integrity and will not be issuing its own CASCADE operation for an update. The relationship() will issue the appropriate UPDATE statements to the database in response to the change of a referenced key, and items locally present in the session during a flush will also be refreshed.

This flag should probably be set to False if primary key changes are expected and the database in use doesn't support CASCADE (i.e. SQLite, MySQL MyISAM tables).

See also:

passive_updates - Introductory documentation and examples.

:paramref:'.mapper.passive_updates' - a similar flag which takes effect for joined-table inheritance mappings.

- **post_update** – this indicates that the relationship should be handled by a second UPDATE statement after an INSERT or before a DELETE. Currently, it also will issue an UPDATE after the instance was UPDATEd as well, although this technically should be improved. This flag is used to handle saving bi-directional dependencies between two individual rows (i.e. each row references the other), where it would otherwise be impossible to INSERT or DELETE both rows fully since one row exists before the other. Use this flag when a particular mapping arrangement will incur two rows that are dependent on each other, such as a table that has a one-to-many relationship to a set of child rows, and also has a column that references a single child row within that list (i.e. both tables contain a foreign key to each other). If a flush operation returns an error that a "cyclical dependency" was detected, this is a cue that you might want to use **:paramref:'~.relationship.post_update'** to "break" the cycle.

  See also:

  post_update - Introductory documentation and examples.

- **primaryjoin** – a SQL expression that will be used as the primary join of this child object against the parent object, or in a many-to-many relationship the join of the primary object to the association table. By default, this value is computed based on the foreign key relationships of the parent and child tables (or association table).

  :paramref:'~.relationship.primaryjoin' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  relationship_primaryjoin

- **remote_side** – used for self-referential relationships, indicates the column or list of columns that form the "remote side" of the relationship.

  :paramref:'.relationship.remote_side' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  Changed in version 0.8: The `remote()` annotation can also be applied directly to the `primaryjoin` expression, which is an alternate, more specific system of describing which columns in a particular `primaryjoin` should be considered "remote".

  See also:

self_referential - in-depth explaination of how **:paramref:'~.relationship.remote_side'** is used to configure self-referential relationships.

`remote()` - an annotation function that accomplishes the same purpose as **:paramref:'~.relationship.remote_side'**, typically when a custom **:paramref:'~.relationship.primaryjoin'** condition is used.

- **query_class** – a `Query` subclass that will be used as the base of the "appender query" returned by a "dynamic" relationship, that is, a relationship that specifies `lazy="dynamic"` or was otherwise constructed using the `orm.dynamic_loader()` function.

  See also:

  dynamic_relationship - Introduction to "dynamic" relationship loaders.

- **secondaryjoin** – a SQL expression that will be used as the join of an association table to the child object. By default, this value is computed based on the foreign key relationships of the association and child tables.

  :paramref:'~.relationship.secondaryjoin' may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  relationship_primaryjoin

- **single_parent** – when True, installs a validator which will prevent objects from being associated with more than one parent at a time. This is used for many-to-one or many-to-many relationships that should be treated either as one-to-one or one-to-many. Its usage is optional, except for `relationship()` constructs which are many-to-one or many-to-many and also specify the `delete-orphan` cascade option. The `relationship()` construct itself will raise an error instructing when this option is required.

  See also:

  unitofwork_cascades - includes detail on when the **:paramref:'~.relationship.single_parent'** flag may be appropriate.

- **uselist** – a boolean that indicates if this property should be loaded as a list or a scalar. In most cases, this value is determined automatically by `relationship()` at mapper configuration time, based on the type and direction of the relationship - one to many forms a list, many to one forms a scalar, many to many is a list. If a scalar is desired where normally a list would be present, such as a bi-directional one-to-one relationship, set **:paramref:'~.relationship.uselist'** to False.

  The **:paramref:'~.relationship.uselist'** flag is also available on an existing `relationship()` construct as a read-only attribute, which can be used to determine if this `relationship()` deals with collections or scalar attributes:

  ```
  >>> User.addresses.property.uselist
  True
  ```

  See also:

  relationships_one_to_one - Introduction to the "one to one" relationship pattern, which is typically when the **:paramref:'~.relationship.uselist'** flag is needed.

- **viewonly=False** – when set to True, the relationship is used only for loading objects, and not for any persistence operation. A `relationship()` which specifies **:paramref:'~.relationship.viewonly'** can work with a wider range of SQL operations within the **:paramref:'~.relationship.primaryjoin'** condition, including operations that feature the

use of a variety of comparison operators as well as SQL functions such as `cast()`. The **:paramref:'~.relationship.viewonly'** flag is also of general use when defining any kind of `relationship()` that doesn't represent the full set of related objects, to prevent modifications of the collection from resulting in persistence operations.

Weather

Datetime

class `dobby.models.actions.datetime.`**`Datetime`**(***kwargs*)

Bases: `dobby.models.actions.Action`

Datetime Action uses current time

`format_tts()` uses `time.strftime()` to format `tts`

2.4 Controller

The controller is responsible for handling most of the logic of Dobby and mixing everything together

class `dobby.controller.`**`Controller`**(*event_queue*, *tts_queue*, *session*, *recognizer*, *recognition_timeout*, *failed_message*, *confirmation_messages*)

Bases: `threading.Thread`

Threaded controller that holds the main logic of Dobby. It grabs events as they come and put corresponding (according to the database) processed actions in the queue. Error message and confirmation messages are customizable

Parameters

- **event_queue** (*Queue.Queue*) – where to listen for events
- **tts_queue** (*Queue.Queue*) – where to put the tts from processed actions
- **session** (*Session*) – Dobby database session
- **recognition_timeout** (*integer*) – time to wait for a `Command` to be recognized once a `RecognitionEvent` is received
- **failed_message** (*string*) – error message to say when the recognized `Command` does not match anything in the database
- **confirmation_messages** (*list*) – a random message to say is picked and sent to the action queue whenever a `RecognitionEvent` is caught

`stop`()

Stop the thread

2.5 Speakers

Speakers handle the voice synthesis part of Dobby.

2.5.1 Base

class dobby.speakers.**Speaker**(*tts_queue*)
Bases: threading.Thread

Threaded Speaker base. Its task is to speak each actions it gets in a row

Parameters **tts_queue** (*Queue.Queue*) – where to pick text-to-speech

run()
Wait for events in the tts_queue and speak the received TTS. Once the thread is told to stop, terminate() is called

speak(*text*)
Speak the text and block until it's said

Parameters **text** (*string*) – text to speech

stop()
Stop the thread

terminate()
Terminate the thread

2.5.2 Speechd

d