## CONTENTS

1 First steps 1
2 The model layer 3
3 The view layer 5
4 The development process 7
5 Contents 9
  5.1 Using tryton 9
  5.2 API Reference 64
6 Indices, glossary and tables 111

Python Module Index 113
Index 115
FIRST STEPS

• Installation: Installation | Configuration | Setup a database | Start the server
• **Models**: Model syntax | Field types | Domain syntax | Access rights | Triggers
• Views: View types | Extension
• Reports: Report definition
THE DEVELOPMENT PROCESS

- Modules Module definition
5.1 Using tryton

Introduction to all the key parts of tryton:

5.1.1 How to install Tryton

Install Tryton

There are three options to install Tryton:

- Install the version provided by your operating system distribution. This is the quickest and recommended option for those who has operating system that distributes Tryton.

- Install the published package. You first need to have `pip` (<https://pip.pypa.io/>) installed. Then to install `trytond` run:

  ```
  python -m pip install trytond
  ```

  You can also install for example the `sale` module with:

  ```
  python -m pip install trytond_sale
  ```

- Without installation, you need to make sure you have all the dependencies installed and then run:

  ```
  python bin/trytond
  ```

  You can register modules by linking them into the `trytond/modules` folder.

**Warning:** Note that you may need administrator/root privileges for this step, as this command will by default attempt to install trytond to the Python site-packages directory on your system.
5.1.2 Configuration file for Tryton

The configuration file controls some aspects of the behavior of Tryton. The file uses a simple ini-file format. It consists of sections, led by a [section] header and followed by name = value entries:

```
[database]
uri = postgresql://user:password@localhost/
path = /var/lib/trytond
```

For more information see ConfigParser.

The default value of any option can be changed using environment variables with names using this syntax: TRYTOND_<SECTION>_<NAME>.

Sections

This section describes the different main sections that may appear in a Tryton configuration file, the purpose of each section, its possible keys, and their possible values. Some modules could request the usage of other sections for which the guideline asks them to be named like their module.

web

Defines the behavior of the web interface.

listen

Defines the couple of host (or IP address) and port number separated by a colon to listen on.

Default localhost:8000

Note: To listen on all IPv4 interfaces use the value 0.0.0.0:8000 and for all IPv6 interfaces use ::8000.

hostname

Defines the hostname to use when generating a URL when there is no request context available, for example during a cron job.

root

Defines the root path served by GET requests.

Default: Under the www directory of user's home running tryond.
num_proxies

The number of proxy servers in front of trytond.
Default: 0

cache_timeout

The cache timeout in seconds.
Default: 12h

cors

The list (one per line) of origins allowed for Cross-Origin Resource sharing.

avatar_base

The base URL without a path for avatar URL.
Default: ''

Note: It can be used to setup a CDN.

avatar_timeout

The time in seconds that the avatar can be stored in cache.
Default: 7 days

database

Defines how the database is managed.

uri

Contains the URI to connect to the SQL database. The URI follows the RFC-3986. The typical form is:

database://username:password@host:port/?param1=value1&param2=value2

The parameters are database dependent, check the database documentation for a list of valid parameters.
Default: The value of the environment variable TRYOND_DATABASE_URI or sqlite:/// if not set.
The available databases are:

5.1. Using trytond
PostgreSQL

psycopg2 supports two type of connections:

- TCP/IP connection: postgresql://user:password@localhost:5432/
- Unix domain connection: postgresql://username:password@

Please refer to psycopg2 for the complete specification of the URI.
A list of parameters supported by PostgreSQL can be found in the documentation.

SQLite

The URI is defined as sqlite://
If the name of the database is :memory:, the parameter mode will be set to memory thus using a pure in-memory database.

The recognized query parameters can be found in SQLite’s documentation.

path

The directory where Tryton stores files and so the user running trytond must have write access on this directory.
Default: The db folder under the user home directory running trytond.

list

A boolean value to list available databases.
Default: True

retry

The number of retries when a database operational error occurs during a request.
Default: 5

language

The main language of the database that will be used for storage in the main table for translations.
Default: en
**avatar_filestore**

This configuration value indicates whether the avatars should be stored in the `tryond.filestore` (True) or the database (False).

Default: False

**avatar_prefix**

The prefix to use with the FileStore to store avatars.

Default: None

**default_name**

The name of the database to use for operations without a database name. Default: `template1` for PostgreSQL, `:memory:` for SQLite.

**timeout**

The timeout duration in seconds after which the connections to unused databases are closed. Default: 1800 (30 minutes)

**minconn**

The minimum number of connections to keep in the pool (if the backend supports pool) per process. Default: 1

**maxconn**

The maximum number of simultaneous connections to the database per process. Default: 64

**request**

**max_size**

The maximum size in bytes of unauthenticated request (zero means no limit).

Default: 2MB

**max_size_authenticated**

The maximum size in bytes of an authenticated request (zero means no limit).

Default: 2GB

---

5.1. Using tryond
**cache**

Defines size of various cache.

**transaction**

The number of contextual caches kept per transaction.
Default: 10

**model**

The number of different model kept in the cache per transaction.
Default: 200

**record**

The number of record loaded kept in the cache of the list. It can be changed locally using the `_record_cache_size` key in `Transaction.context`.
Default: 2000

**field**

The number of field to load with an eager `Field.loading`.
Default: 100

**clean_timeout**

The minimum number of seconds between two cleanings of the cache. If the value is 0, the notification between processes will be done using channels if the back-end supports them.
Default: 300

**count_timeout**

The cache timeout duration in seconds of the estimation of records.
Default: 86400 (1 day)
count_clear

The number of operations after which the counting estimation of records is cleared.
Default: 1000

queue

worker

Activate asynchronous processing of the tasks. Otherwise they are performed at the end of the requests.
Default: False

clean_days

The number of days after which processed tasks are removed.
Default: 30

error

clean_days

The number of days after which reported errors are removed.
Default: 90

table

This section allows to override the default generated table name for a ModelSQL. The main goal is to bypass limitation on the name length of the database backend. For example:

```
[table]
account.invoice.line = acc_inv_line
account.invoice.tax = acc_inv_tax
```

ssl

Activates SSL on the web interface.

Note: It is recommended to delegate the SSL support to a proxy.

5.1. Using trytond
privatekey

The path to the private key.

certificate

The path to the certificate.

**Tip:** Set only one of `privatekey` or `certificate` to `true` if the SSL is delegated.

e-mail

**Note:** Email settings can be tested with the `trytond-admin` command

uri

The **SMTP-URL** to connect to the SMTP server which is extended to support **SSL** and **STARTTLS**. The available protocols are:

- `smtp`: simple SMTP
- `smtp+tls`: SMTP with STARTTLS
- `smtps`: SMTP with SSL

The `uri` accepts the following additional parameters:

- `local_hostname`: used as FQDN of the local host in the HELO/EHLO commands, if omitted it will use the value of `socket.getfqdn()`.
- `timeout`: A number of seconds used as timeout for blocking operations. A `socket.timeout` will be raised when exceeded. If omitted the default timeout will be used.

Default: `smtp://localhost:25`

from

Defines the default From address (using RFC-822) for emails sent by Tryton.

For example:

```
from: "Company Inc" <info@example.com>
```
retry

The number of retries when the SMTP server returns a temporary error.
Default: 5

session

 authentications

A comma separated list of the authentication methods to try when attempting to verify a user's identity. Each method is tried in turn, following the order of the list, until one succeeds. In order to allow multi-factor authentication, individual methods can be combined together using a plus (+) symbol.

Example:

```plaintext
authentications = password+sms,ldap
```

Each combined method can have options to skip them if they are met except for the first method. They are defined by appending their name to the method name after a question mark (?) and separated by colons (:).

Example:

```plaintext
authentications = password+sms?ip_address:device_cookie
```

By default, Tryton only supports the password method. This method compares the password entered by the user against a stored hash of the user's password. By default, Tryton supports the ip_address and device_cookie options. The ip_address compares the client IP address with the known network list defined in authentication_ip_network. The device_cookie checks the client device is a known device of the user. Other modules can define additional authentication methods and options, please refer to their documentation for more information.

Default: password

 authentication_ip_network

A comma separated list of known IP networks used to check for ip_address authentication method option.
Default: ''

max_age

The time in seconds that a session stay valid.
Default: 2592000 (30 days)
timeout

The time in seconds without activity before the session is no more fresh.
Default: 300 (5 minutes)

max_attempt

The maximum authentication attempt before the server answers unconditionally Too Many Requests for any other attempts. The counting is done on all attempts over a period of timeout.
Default: 5

max_attempt_ip_network

The maximum authentication attempt from the same network before the server answers unconditionally Too Many Requests for any other attempts. The counting is done on all attempts over a period of timeout.
Default: 300

ip_network_4

The network prefix to apply on IPv4 address for counting the authentication attempts.
Default: 32

ip_network_6

The network prefix to apply on IPv6 address for counting the authentication attempts.
Default: 56

password

length

The minimal length required for the user password.
Default: 8

forbidden

The path to a file containing one forbidden password per line.
**reset_timeout**

The time in seconds until the reset password expires.
Default: 86400 (24h)

**passlib**

The path to the INI file to load as CryptContext. If no path is set, Tryton will use the schemes argon2, scrypt, bcrypt or pbkdf2_sha512.
Default: None

**attachment**

Defines how to store the attachments

**filestore**

A boolean value to store attachment in the FileStore.
Default: True

**store_prefix**

The prefix to use with the FileStore.
Default: None

**bus**

**allow_subscribe**

A boolean value to allow clients to subscribe to bus channels.
Default: False

**url_host**

If set redirects bus requests to the host URL.
**long_polling_timeout**

The time in seconds to keep the connection to the client opened when using long polling for bus messages.

Default: 300

**cache_timeout**

The number of seconds a message should be kept by the queue before being discarded.

Default: 300

**select_timeout**

The timeout duration of the select call when listening on a channel.

Default: 5

**html**

**src**

The URL pointing to TinyMCE editor.

Default: `https://cloud.tinymce.com/stable/tinymce.min.js`

**plugins**

The space separated list of TinyMCE plugins to load. It can be overridden for specific models and fields using the names: `plugins-<model>-<field>` or `plugins-<model>`.

Default: ``

**css**

The JSON list of CSS files to load. It can be overridden for specific models and fields using the names: `css-<model>-<field>` or `css-<model>`.

Default: []

**class**

The class to add on the body. It can be overridden for specific models and fields using the names: `class-<model>-<field>` or `class-<model>`.

Default: ''
wsgi middleware

The section lists the WSGI middleware class to load. Each middleware can be configured with a section named wsgi <middleware> containing args and kwargs options.

Example:

```
[wsgi middleware]
ie = werkzeug.contrib.fixers.InternetExplorerFix

[wsgi ie]
kwargs={'fix_attach': False}
```

5.1.3 How to setup a database

The database section of the configuration must be set before starting.

Create a database

Depending of the database backend choosen, you must create a database (see the documentation of the choosen backend). The user running trytond must be granted the privilege to create tables. For backend that has the option, the encoding of the database must be set to UTF-8.

Initialize a database

A database can be initialized using this command line:

```
trytond-admin -c <config file> -d <database name> --all
```

At the end of the process, trytond-admin will ask to set the password for the admin user.

Update a database

To upgrade to a new series, the command line is:

```
trytond-admin -c <config file> -d <database name> --all
```

**Warning:** Prior to upgrade see if there is no manual action to take on the migration topic.

To activate a new language on an existing database, the command line is:

```
trytond-admin -c <config file> -d <database name> --all -l <language code>
```

Once activated, the language appears in the user preferences.

When installing new modules, the list of modules must be updated with:

```
trytond-admin -c <config file> -d <database name> --update-modules-list
```

Once updated, the new modules can be activated from the client or activated with:
5.1.4 Logging configuration

Without any configuration, trytond writes ERROR messages to standard output. For each verbose flag set, the log level decreases.

Logs can be configured using a configparser-format file. The filename can be specified using trytond logconf parameter.

Example

This example allows to write INFO messages on standard output and on a disk log file rotated every day.

```ini
[formatters]
keys=simple

[handlers]
keys=rotate,console

[loggers]
keys=root

[formatter_simple]
format=[%(asctime)s] %(levelname)s:%(name)s:%(message)s
datefmt=%a %b %d %H:%M:%S %Y

[handler_rotate]
class=handlers.TimedRotatingFileHandler
args=('/tmp/tryton.log', 'D', 1, 30)
formatter=simple

[handler_console]
class=StreamHandler
formatter=simple
args=(sys.stdout,)

[logger_root]
level=INFO
handlers=rotate,console
```
5.1.5 How to start the server

Web service

You can start the default web server bundled in Tryton with this command line:

```
trytond -c <config file>
```

The server will wait for client connections on the interface defined in the `web` section of the `configuration`.

**Note:** When using multiple config files the order is important as last entered files will override the items of first files.

**Warning:** This runs the of Werkzeug development server which should not be used on production systems.

WSGI server

If you prefer to run Tryton inside your own WSGI server instead of the simple server of Werkzeug, you can use the application `trytond.application.app`. Following environment variables can be set:

- `TRYTOND_CONFIG`: Point to `configuration` file.
- `TRYTOND_LOGGING_CONFIG`: Point to `logging` file.
- `TRYTOND_LOGGING_LEVEL`: An integer to set the default `logging level` (default: `ERROR`).
- `TRYTOND_COROUTINE`: Use coroutine for concurrency.
- `TRYTOND_DATABASE_NAMES`: A list of database names in CSV format, using python default dialect.

**Warning:** You must manage to serve the static files from the web root.

Coroutine server

The Werkzeug server uses thread for concurrency. This is not optimal for the long-polling request on the `bus` as each client consumes permanently one thread. You can start the server with coroutine using the option `--coroutine`.

**Note:** This will use the pure-Python, gevent-friendly WSGI server.

Cron service

If you want to run some `scheduled actions`, you must also run the cron server with this command line:

```
trytond-cron -c <config file> -d <database>
```

The server will wake up every minutes and perform the scheduled actions defined in the `database`. You can also launch the command every few minutes from a scheduler with the option `--once`.

5.1. Using trytond
Worker service

If you want to use a pool of workers to run \textit{asynchronously some tasks}, you must activate the worker in the queue section of the \textit{configuration} and run the worker manager with this command line:

\begin{verbatim}
trytond-worker -c <config file> -d <database>
\end{verbatim}

The manager will dispatch tasks from the queue to a pool of worker processes.

Services options

You will find more options for those services by using \texttt{--help} arguments.

5.1.6 Models

A model represents a single business logic or concept. It contains fields and defines the behaviors of the record. Most of the time, each model stores records in a single database table.

The basics:

- Each model is a Python class that subclasses one of \texttt{trytond.model.model.Model}.
- \textit{Fields} are defined as model attributes.
- Tryton generates the table definitions
- Tryton provides an API following the \textit{active record} pattern to access the records.

Example

This example defines a \texttt{Party} model which has a name and a code fields:

\begin{verbatim}
from trytond.model import ModelView, ModelSQL, fields
class Party(ModelSQL, ModelView):
    __name__ = "party.party"
    name = fields.Char('Name')
    code = fields.Char('Code')
\end{verbatim}

The class must be registered in the \texttt{Pool} by the \texttt{register()} method of the \texttt{module}. Model classes are essentially data mappers to records and Model instances are records.

Model attributes define meta-information of the model. They are class attributes starting with an underscore. Some model properties are instance attributes allowing to update them at other places in the framework.
5.1.7 Backend Types supported

This table gives a comprehensive list of the SQL Types that are expected to be supported by the database backends. If the type is not supported then the backend will have to emulate the behavior described here.

The columns are in the following order:

- The SQL type\(^1\) representing the field
- The python type expected on input
- The python type received on output

None will represent the NULL value and vice versa, it can be used as input or output for any SQL type.

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>Python input type</th>
<th>Python output type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>int / long(^2)</td>
<td>int / long(^2)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float / int / long(^7)</td>
<td>float</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>decimal.Decimal</td>
<td>decimal.Decimal</td>
</tr>
<tr>
<td>VARCHAR / VARCHAR(length)</td>
<td>str / unicode(^3)</td>
<td>str / unicode(^7)</td>
</tr>
<tr>
<td>TEXT</td>
<td>str / unicode(^3)</td>
<td>str / unicode(^7)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>datetime.datetime</td>
<td>datetime.datetime</td>
</tr>
<tr>
<td>DATETIME</td>
<td>datetime.datetime without microseconds(^4)</td>
<td>datetime.datetime without microseconds(^4)</td>
</tr>
<tr>
<td>DATE</td>
<td>datetime.date</td>
<td>datetime.date</td>
</tr>
<tr>
<td>TIME</td>
<td>datetime.time</td>
<td>datetime.time</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>datetime.timedelta</td>
<td>datetime.timedelta</td>
</tr>
<tr>
<td>BLOB</td>
<td>bytes</td>
<td>bytes</td>
</tr>
</tbody>
</table>

5.1.8 Default value of fields

When a record is created, each field, which doesn’t have a value specified, is set with the default value if exists.

The following class method:

```
Model.default_<field name>()
```

Return the default value for field name.

This example defines an Item model which has a default since:

```python
import datetime
from trytond.model import ModelView, ModelSQL, fields
class Item(ModelSQL, ModelView):
    "Item"
    __name__ = 'item'
```

\(^1\) Corresponding to the SQL 92 standard or to a PostgreSQL type.
\(^2\) In Python 2 integers over `sys.maxint` are represented by the `long` type.
\(^3\) Str when using python 3; unicode when using python 2.
\(^4\) Datetime objects are not localized to any timezone.
since = fields.Date('since')

@classmethod
def default_since(cls):
    return datetime.date.today()

See also method Model.default_get: default_get

5.1.9 on_change of fields

Tryton allows developers to define methods that can be called once a field’s value has changed by the user this is the on_change method. The method has the following name:

```
Model.on_change_<field name>
```

This is an instance method, an instance of Model will be created by using the values from the form’s fields specified by the on_change list defined on the field. Any change made on the instance will be pushed back to the client-side record.

There is also a way to define a method that must update a field whenever any field from a predefined list is modified. This list is defined by the on_change_with attribute of the field. The method that will be called has the following name:

```
Model.on_change_with_<field_name>
```

Just like for the classic on_change, an instance of Model will be created by using the values entered in the form’s fields specified by the on_change_with attribute. The method must return the new value of the field to push back to the client-side record.

5.1.10 Domain

Domains represent a set of records. A domain is a list of none or more clauses. A clause is a condition, which returns true or false. A record belongs to a domain, when the final result of the list of clauses returns true.

Syntax

The definition of a simple domain with one clause is represented by this pattern:

```
domain = [(<field name>, <operator>, <operand>)]
```

<field name> Is the name of a fields or a pyson statement, that evaluates to a string.

A field of type Many2One or Many2Many or One2Many or One2One or Reference can be dereferenced to related models. This is illustrated by the following example:

```
domain = [['country.name', '=', 'Japan']]
```

The number of dots in a clause is not limited.

Warning: For trytond.model.fields.Reference, an extra ending clause is needed to define the target model to join, for example:

```
domain = [['origin.party.name', '=', 'John Doe', 'sale.sale']]
```
A field of type `Dict` can be searched by key also by using one dot. For example:

```python
domain = [('attributes.color', '=', 'yellow')]
```

**Warning:** Order comparison of `date` and `datetime` types is not supported.

**operator** Is an operator out of `Domain Operators` or a `pyson` statement, that evaluates to a domain operator string.

**operand** Is an operand or a `pyson` statement. The type of operand depends on the kind of `<field name>`.

The definition of an empty domain is:

```python
domain = []
```

An empty domain without clauses will always return all `active` records. A record is active, when its appropriate `Model` contains a `Boolean` field with name `active`, and set to true. When the appropriate `Model` does not contain a `Boolean` field with name `active` all records are returned.

A domain can be setup as a combination of clauses, like shown in this pattern:

```python
domain = [
    ('field name1', 'operator1', 'operand1'),
    ('field name2', 'operator2', 'operand2'),
    ('field name3', 'operator3', 'operand3'),
]
```

The single clauses are implicitly combined with a logical `AND` operation.

In the domain syntax it is possible to provide explicitly the combination operation of the clauses. These operations can be `AND` or `OR`. This is illustrated by the following pattern:

```python
domain = [ 'OR', [
    ('field name1', 'operator1', 'operand1'),
    ('field name2', 'operator2', 'operand2'),
    ('field name3', 'operator3', 'operand3'),
], [
    ('field name1', 'operator1', 'operand1'),
    ('field name2', 'operator2', 'operand2'),
    ('field name3', 'operator3', 'operand3'),
],]
```

Here the domain is evaluated like this: `((clause1 AND clause2) OR clause3)`. Please note that the `AND` operation is implicit assumed when no operator is given. While the `OR` operation must be given explicitly. The former pattern is equivalent to the following completely explicit domain definition:

```python
domain = [ 'OR', [
    'AND', [
        ('field name1', 'operator1', 'operand1'),
        ('field name2', 'operator2', 'operand2'),
    ], [
        ('field name3', 'operator3', 'operand3'),
    ],
], [
    'AND', [
        ('field name1', 'operator1', 'operand1'),
        ('field name2', 'operator2', 'operand2'),
    ], [
        ('field name3', 'operator3', 'operand3'),
    ],
],]
```

Obviously the use of the implicit `AND` operation makes the code more readable.
Domain Operators

The following operators are allowed in the domain syntax. `<field name>`, `<operator>` and `<operand>` are dereferenced to their values. The description of each operator follows this pattern, unless otherwise noted:

```
(<field name>, <operator>, <operand>)
```

`=`

Is a parity operator. Returns true when `<field name>` equals to `<operand>`.

`!=`

Is an imparity operator. It is the negation of the `=` operator.

`like`

Is a pattern matching operator. Returns true when `<field name>` is contained in the pattern represented by `<operand>`.

In `<operand>` an underscore `_` matches any single character, a percent sign `%` matches any string with zero or more characters. To use `_` or `%` as literal, use the backslash `\` to escape them. All matching is case sensitive.

`not like`

Is a pattern matching operator. It is the negation of the `like` operator.

`ilike`

Is a pattern matching operator. The same use as `like` operator, but matching is case insensitive.

`not ilike`

Is a pattern matching operator. The negation of the `ilike` operator.

`in`

Is a list member operator. Returns true when `<field name>` is in `<operand>` list.
not in

Is a list non-member operator. The negation of the in operator.

<

Is a less than operator. Returns true for type string of <field name> when <field name> is alphabetically sorted before <operand>.

Returns true for type number of <field name> when <field name> is less than <operand>.

>

Is a greater than operator. Returns true for type string of <field name> when <field name> is alphabetically sorted after <operand>.

Returns true for type number of <field name> when <field name> is greater <operand>.

<=

Is a less than or equal operator. Returns the same as using the < operator, but also returns true when <field name> is equal to <operand>.

>=

Is a greater than or equal operator. Returns the same as using the > operator, but also returns true when <field name> is equal to <operand>.

child_of

Is a parent child comparison operator. Returns true for records that are a child of <operand>. <operand> is a list of ids and <field name> must be a many2one or a many2many. In case <field name> is not linked to itself, the clause pattern extends to:

```<field name>, ['child_of'|'not_child_of'], <operand>, <parent field>)```

Where <parent field> is the name of the field constituting the many2one on the target model.

not child_of

Is a parent child comparison operator. It is the negation of the child_of operator.
parent_of

Is a parent child comparison operator. It is the same as child_of operator but if `<field name>` is a parent of `<operand>`.

not parent_of

Is a parent child comparison operator. It is the negation of this parent_of operator.

where

Is a `tryond.model.fields.One2Many / tryond.model.fields.Many2Many` domain operator. It returns true for every row of the target model that match the domain specified as `<operand>`.

not where

Is a `tryond.model.fields.One2Many / tryond.model.fields.Many2Many` domain operator. It returns true for every row of the target model that does not match the domain specified as `<operand>`.

5.1.11 PYSON

PYSON is the PYthon Statement and Object Notation. It is a lightweight domain specific language for the general representation of statements. PYSON is used to encode statements which can be evaluated in different programming languages, serving for the communication between tryond and any third party software. A PYSON parser can easily be implemented in other programming languages. So third party softwares do not need to depend on Python to be able to fully communicate with the Tryton server.

PYSON is a deterministic algorithm which will always succeed to evaluate statements. There is a default behavior for unknown values. It is statically typed and checked on instantiation.

There is also a reference documentation of the API.

Syntax

The syntax of a PYSON statement follows this pattern:

```
Statement(argument1[, argument2[, ...]])
```

where arguments can be another statement or a value. The evaluation direction is inside out, deepest first.
## PYSON Examples

Given the PYSON statement:

```
Eval('active_id', -1)
```

Eval() checks the evaluation context for the variable `active_id` and returns its value or -1 if not defined. A similar expression in Python looks like this:

```
'active_id' in locals() and active_id or -1
```

Given the PYSON statement:

```
Not(Bool(Eval('active')))
```

Eval() checks the evaluation context for a variable `active` and returns its value to Bool() or '' if not defined. Bool() returns the corresponding boolean value of the former result to Not(). Not() returns the boolean negation of the previous result. A similar expression in Python looks like this:

```
'active' in locals() and active == False
```

Given the PYSON statement:

```
Or(Not(Equal(Eval('state'), 'draft')), Bool(Eval('lines')))
```

In this example are the results of two partial expressions Not(Equal(Eval('state'), 'draft')) and Bool(Eval('lines')) evaluated by a logical OR operator. The first expression part is evaluated as follow: When the value of Eval('state') is equal to the string 'draft' then return true, else false. Not() negates the former result. A similar expression in Python looks like this:

```
'state' in locals() and 'lines' in locals() \ 
    and state != 'draft' or bool(lines)
```

Given the PYSON statement:

```
If(In('company', Eval('context', {})), '=', '!=')
```

In this example the result is determined by an if-then-else condition. In('company', Eval('context', {})) is evaluated like this: When the key 'company' is in the dictionary context, returns true, otherwise false. If() evaluates the former result and returns the string '=' if the result is true, otherwise returns the string '!='. A similar expression in Python looks like this:

```
'context' in locals() and isinstance(context, dict) \ 
    and 'company' in context and '=' or '!='
```

Given the PYSON statement:

```
Get(Eval('context', {}), 'company', 0)
```

Eval() checks the evaluation context for a variable `context` if defined, return the variable `context`, otherwise return an empty dictionary {}. Get() checks the former resulting dictionary and returns the value of the key 'company', otherwise it returns the number 0. A similar expression in Python looks like this:

```
'context' in locals() and context.get('company', 0) or 0
```
5.1.12 Access Rights

There are 5 levels of access rights: Model, Actions, Field, Button and Record Rule. They are based on the user’s group membership. If any of those levels are violated, an error is raised.

The access rights are checked if the Transaction.context has the key _check_access set to True (set by default by RPC.check_access) and if the user is not root.

**Model**

They are defined by records of ir.model.access which define for each couple of model and group, the read, write, create and delete permission. The permissions are related to the ModelStorage methods with the same name and on search() using the read permission.

If any group the user belongs to has the checked permission activated, then the user is granted this permission.

If there is no record for the model, then access is granted to all users.

---

**Note:** Relation fields for which the user has no read access are automatically removed from the views.

**Actions**

Each ir.action has a groups field which contains a list of user groups that are allowed to see and launch it.

There is a special case for wizard for which the read access on the model is also checked and also the write access if there is no groups linked.

**Field**

They are defined by records of ir.model.field.access and work like those for Model but are applied to fields.

---

**Note:** Fields for which the user has no read access are automatically removed from the views.

**Button**

For each button of a model the records of ir.model.button define the list of groups that are allowed to call it. The user only needs to belong to one of the groups to be granted the permission to use it.

If no group is defined for a button, the write permission to the model is checked instead.

The read permission to the model is always enforced.

---

**Note:** Buttons for which the user has no access are marked readonly.
Button Rule

The `ir.model.button` can contain a list of rules which define how many different users must click on the button. Each rule, for which the condition is met, must be passed to actually trigger the action. The counter can be reset when another defined button is clicked.

Record Rule

The record rules are conditions that records must meet for the user to be granted permission to use them. They are defined by records of `ir.rule.group` which contains:

- a model on which it applies
- the permissions granted
- a set of user groups to which the rule applies
- a global flag to always enforce
- a default flag to add to all users
- a list of `ir.rule` with a `domain` to select the records to which the rule applies.

A rule group matches a record if the record is validated by at least one of the domains. The access is granted to a record:

- if the user belongs to a group which has at least one matching rule group that has the permission,
- or if there is a default matching rule group with the permission,
- or if there is a global matching rule group with the permission.

Otherwise the access is denied if there is any matching rule group.

**Note:** Records for which the user has no read access are filtered out from the `search()` result.

5.1.13 User Errors and Warnings

When processing a request, you can stop the flow by raising an exception that will be displayed to the user as an error message or a warning. They are respectively `UserError` and `UserWarning`.

User Errors

An error displays a message and optionally a description to the user.

Example:

```python
from trytond.exceptions import UserError
from trytond.model import Model

class MyModel(Model):
    "My Model"
    __name__ = 'my_model'

    def process(self):
        if check_failed:
            raise UserError("You cannot process.", "because...")
```
Note: They are often used in combination with `gettext()` to translate the messages.

**User Warnings**

A warning displays a confirmation message with optionally a description to the user. The user can decide to continue so the request is processed again without stopping at the warning. Otherwise the user can cancel its request. The warning instance is identified by a name which allows to skip it the next time it is checked, that’s why they often build using the `format` method which uses record instances to generate a unique name based on ids.

Example:

```python
from trytond.exceptions import UserWarning
from trytond.model import Model
from trytond.pool import Pool

class MyModel(Model):
    __name__ = 'my_model'

def process(self):
    Warning = Pool().get('res.user.warning')
    warning_name = Warning.format('mywarning', [self])
    if Warning.check(warning_name):
        raise UserWarning(warning_name, "Process cannot be canceled.")
```

Note: If there is no user interaction the warnings can be skipped by setting the `_skip_warnings` key of the context to True.

### 5.1.14 Triggers

Triggers allow to define methods of `trytond.model.Model` that are called at the end of the transaction when one of those events happen to a record:

- On Creation
- On Modification
- On Deletions
- On Time: When a condition changes over time.

The method signature is:

```python
<method name>(cls, records, trigger)
```

Where `records` is the list of records that triggered the event and `trigger` is the `ir.trigger` instance which is triggered.

Triggers are defined by records of `ir.trigger`. Each record must define a pyson condition which will be evaluated when the event occurs. Only those records for which the condition is evaluated to true will be processed by the trigger with the exception of modification triggers which will only process the records for which the condition is evaluated to false before and evaluated to true after the modification.
### 5.1.15 Actions

Actions are used to describe specific behaviors in the client.

There are four types of actions:

- Report
- Window
- Wizard
- URL

**Keyword**

Keywords define where to display the action in the client.

There are five places:

- Open tree (`tree_open`)
- Print form (`form_print`)
- Action form (`form_action`)
- Form relate (`form_relate`)
- Open Graph (`graph_open`)

**Report**

**Window**

The window action describe how to create a new tab in the client.

**View**

**Domain**

The window action could have a list of domains which could be activated on the view. The boolean field count indicates if the client must display the number of records for this domain.

**Warning:** The counting option must be activated only on domains which have not too much records otherwise it may overload the database.
5.1.16 Views

The views are used to display records of a ModelView to the user.

In Tryton, ModelView can have several views. An action opens a window and defines which view to show.

The views are built from XML that is stored in the view directory of the module or in the databases thanks to the model ir.ui.view.

So generally, they are defined in xml files with this kind of xml where name is the name of the XML file in the view directory:

```xml
<record model="ir.ui.view" id="view_id">
    <field name="model">model name</field>
    <field name="type">type name</field>
    <!--field name="inherit" ref="inherit_view_id"-->
    <!--field name="field_childs">field name</field-->
    <field name="name">view_name</field>
</record>
```

There are different types of views:

- Form
- Tree
- List-Forms
- Graph
- Board
- Calendar

Form view

The RNG describing the xml of a form view is stored in trytond/ir/ui/form.rng. There is also a RNC in trytond/ir/ui/form.rnc.

A form view is used to display one record of an object.

Elements of the view are put on the screen following the rules:

- Elements are placed on the screen from left to right, from top to bottom, according to the order of the xml.
- The screen composed of a table with a fixed number of columns and enough rows to handle all elements.
- Elements take one or more columns when they are put in the table. If there are not enough free columns on the current row, the elements are put at the beginning of the next row.
XML description

List of attributes shared by many form elements:

- **id**: A unique identifier for the tag if there is no name attribute.
- **yexpand**: A boolean to specify if the label should expand to take up any extra vertical space.
- **yfill**: A boolean to specify if the label should fill the vertical space allocated to it in the table cell.
- **yalign**: The vertical alignment, from 0.0 to 1.0.
- **xexpand**: The same as yexpand but for horizontal space.
- **xfill**: The same as yfill but for horizontal space.
- **xalign**: The horizontal alignment, from 0.0 to 1.0.
- **colspan**: The number of columns the widget must take in the table.
- **col**: The number of columns the container must have.
  
  A negative value (or zero) will remove the constraint on the number of columns.
  
  The default value is 4.
- **states**: A string of **PYSON statement** that will be evaluated with the values of the current record.
  
  It must return a dictionary where keys can be:
  
  - **invisible**: If true, the widget will be hidden.
  - **required**: If true, the field will be required.
  - **readonly**: If true, the field will be readonly.
  - **icon**: Only for button, it must return the icon name to use or False.
  - **pre_validate**: Only for button, it contains a domain to apply on the record before calling the button.
  - **depends**: Only for button, it must return the list of field on which the button depends.
- **help**: The string that will be displayed when the cursor hovers over the widget.
- **pre_validate**: A boolean only for fields `trytond.model.fields.One2Many` to specify if the client must pre-validate the records using `trytond.model.Model.pre_validate()`.
- **completion**: A boolean only for fields `trytond.model.fields.Many2One`, `trytond.model.fields.Many2Many` and `trytond.model.fields.One2Many` to specify if the client must auto-complete the field. The default value is True.
- **create**: A boolean to specify if the user can create targets from the widget. The default value is True.
- **delete**: A boolean to specify if the user can delete targets from the widget. The default value is True.
- **factor**: A factor to apply on fields `trytond.model.fields.Integer`, `trytond.model.fields.Float` and `trytond.model.fields.Numeric` to display on the widget. The default value is 1.
- **symbol**: Only on numerical fields, the name of field which provides the symbol to display.
- **grouping**: A boolean only on numerical fields to specify if the client must use grouping separators to display on the widget. The default value is True.
- **help_field**: The name of Dict field mapping the Selection value with its help string.
form

Each form view must start with this tag.

- on_write: The name of a method on the Model of the view that will be called when a record is saved. The method must return a list of record ids that the client must reload if they are already loaded. The function must have this syntax:
  
  on_write(self, ids)

  **Note:** The method must be registered in `trytond.model.Model.__rpc__`.

- creatable: A boolean to specify if the form can be used to create new record. The default value is true.
- col: see in `common-attributes-col`.
- cursor: The name of the field that must have the cursor by default.

label

Display static string.

- string: The string that will be displayed in the label.
- name: The name of the field whose description will be used for string. Except if `string` is set, it will use this value and the value of the field if `string` is empty.
- id: see `common-attributes-id`.
- yexpand: see in `common-attributes-yexpand`.
- yfill: see in `common-attributes-yfill`.
- yalign: see in `common-attributes-yalign`.
- xexpand: see in `common-attributes-xexpand`.
- xfill: see in `common-attributes-xfill`.
- xalign: see in `common-attributes-xalign`.
- colspan: see in `common-attributes-colspan`.
- states: see in `common-attributes-states`.
- help: see in `common-attributes-help`.
- Requires that either id or name is defined.
field

Display a field of the object with the value of the current record.

- **name**: The name of the field.
- **string**: The string that will be displayed for the widget.
- **widget**: The widget that must be used instead of the default one.
- **help**: The string that will be displayed when the cursor stays over the widget.
- **width**: The minimum width the widget should request, or -1 to unset.
- **height**: The minimum height the widget should request, or -1 to unset.
- **readonly**: Boolean to set the field readonly.
- **mode**: Only for One2Many fields: it is a comma separated list, that specifies the order of the view used to display the relation. (Example: tree,form)
- **view_ids**: A comma separated list that specifies the view ids used to display the relation. For Many2One and Many2Many, the order should always be tree then form.
- **product**: Only for One2Many fields, a comma separated list of target field name used to create records from the cartesian product.
- **completion**: Only for Many2One fields, it is a boolean to set the completion of the field.
- **invisible**: The field will not be displayed, but it will fill cells in the table.
- **filename_visible**: Only for Binary fields, boolean that enables the display of the filename.
- **toolbar**: Only for Rich Text widget, boolean that enables the display of the Rich Text toolbar. The default value is 1.
- **yexpand**: see in common-attributes-yexpand.
- **yfill**: see in common-attributes-yfill.
- **xexpand**: see in common-attributes-xexpand.
- **xfill**: see in common-attributes-xfill.
- **colspan**: see in common-attributes-colspan.
- **help**: see in common-attributes-help.
- **pre_validate**: see in common-attributes-pre_validate.
- **completion**: see in common-attributes-completion.
- **factor**: see in common-attributes-factor.
- **spell**: Only for Text widgets, a PYSON statement that will be evaluated to the language code for which spell checking must be done.
- **symbol**: see in common-attributes-symbol.
- **help_field**: see in common-attributes-help_field.
image

Display an image.

- **type**: the type of image source. Available values are **icon** or **url**. The default value is **icon**.
- **name**: The image name or the field name which contains the image name. For the **icon** type it must be the name of a record of `ir.ui.icon`. For the **url** type it must be the URL. It can be relative to the server.
- **url_size**: the name of the size parameter to add to the URL.
- **yexpand**: see in `common-attributes-yexpand`.
- **yfill**: see in `common-attributes-yfill`.
- **colspan**: see in `common-attributes-colspan`.
- **size**: the size of the image in pixels. The default value is 48.
- **states**: see in `common-attributes-states`.
- **help**: see in `common-attributes-help`.

separator

Display a horizontal separator.

- **string**: The string that will be displayed above the separator.
- **name**: The name of the field from which the description will be used for string.
- **id**: see in `common-attributes-id`.
- **yexpand**: see in `common-attributes-yexpand`.
- **yfill**: see in `common-attributes-yfill`.
- **colspan**: see in `common-attributes-colspan`.
- **states**: see in `common-attributes-states`.
- **help**: see in `common-attributes-help`.

Requires that either **id** or **name** is defined.

newline

Force to use a new row.

button

Display a button.

- **name**: The name of the function that will be called. The function must have this syntax:
  ```python
  button(cls, records)
  ```

  The function may return an **ir.action** id or one of those client side action keywords:

  - **new**: to create a new record
  - **delete**: to delete the selected records
• remove: to remove the record if it has a parent
• copy: to copy the selected records
• next: to go to the next record
• previous: to go to the previous record
• close: to close the current tab
• switch <view type> [view id]: to switch the view
• reload: to reload the current tab
• reload context: to reload user context
• reload menu: to reload menu
• icon
• confirm: A text that will be displayed in a confirmation popup when the button is clicked.
• colspan: see in common-attributes-colspan.
• states: see in common-attributes-states.
• help: see in common-attributes-help.
• keyword: specify where will the button be displayed in the client toolbar. The valid values are the keywords starting with form_ from Actions without the form_ part.

Warning: The button should be registered on ir.model.button where the default value of the string, confirm and help attributes can be can be defined.

link

Display an ir.action.act_window as a button with a counter or one counter per tab. When clicked it opens the window.

• name: The XML id of ir.action.act_window.
• colspan: see in common-attributes-colspan.
• states: see in common-attributes-states.
• icon: The name of the icon to display.
• empty: If set to hide the button is not displayed if the counter is zero. The default is show.

notebook

It adds a notebook widget which can contain page tags.

• colspan: see in common-attributes-colspan.
• states: see in common-attributes-states.
Define a new tab inside a notebook.

- **string**: The string that will be displayed in the tab.
- **angle**: The angle in degrees between the baseline of the label and the horizontal, measured counterclockwise.
- **col**: see in *common-attributes-col*.
- **id**: see in *common-attributes-id*.
- **states**: see in *common-attributes-states*.
- Requires that either **id** or **name** is defined.

Create a sub-table in a cell.

- **string**: If set a frame will be drawn around the field with a label containing the string. Otherwise, the frame will be invisible.
- **rowspan**: The number of rows the group spans in the table.
- **col**: see in *common-attributes-col*.
- **expandable**: If this attribute is present the content of the group will be expandable by the user to reveal its content. A value of “1” means that the group will start expanded, a value of “0” means that the group will start unexpanded. There is no default value.
- **homogeneous**: If True all the tables cells are the same size.
- **id**: see in *common-attributes-id*.
- **yexpand**: see in *common-attributes-yexpand*.
- **yfill**: see in *common-attributes-yfill*.
- **yalign**: see in *common-attributes-yalign*.
- **xexpand**: see in *common-attributes-xexpand*.
- **xfill**: see in *common-attributes-xfill*.
- **xalign**: see in *common-attributes-xalign*.
- **colspan**: see in *common-attributes-colspan*.
- **states**: see in *common-attributes-states*.
- Requires that either **id** or **name** is defined.
hpaned, vpaned

- **position**: The pixel position of divider, a negative value means that the position is unset.
- **id**: see in `common-attributes-id`.
- **colspan**: see in `common-attributes-colspan`. The default for panes is 4 columns.

child

Contains the children of a hpaned or vpaned.

**Example**

```xml
<form col="6">
    <label name="name"/>
    <field name="name" xexpand="1"/>
    <label name="code"/>
    <field name="code"/>
    <label name="active"/>
    <field name="active" xexpand="0" width="100"/>
    <notebook colspan="6">
        <page string="General">
            <field name="addresses" mode="form,tree" colspan="4" view_ids="party.address_view_form,party.address_view_tree_sequence"/>
            <label name="type"/>
            <field name="type" widget="selection"/>
            <label name="lang"/>
            <field name="lang" widget="selection"/>
            <label name="website"/>
            <field name="website" widget="url"/>
            <separator string="Categories" colspan="4"/>
            <field name="categories" colspan="4"/>
        </page>
        <page string="Accounting">
            <label name="vat_country"/>
            <field name="vat_country"/>
            <label name="vat_number"/>
            <field name="vat_number"/>
        </page>
    </notebook>
</form>
```
Tree view

The RNG that describes the xml for a tree view is stored in trytond/ir/ui/tree.rng. There is also a RNC in trytond/ir/ui/tree.rnc.

Tree view is used to display records inside a list or a tree.

It is a tree if there is a field_childs defined and this tree will have drag and drop activated if the field_childs and the parent field are defined in the view.

The columns of the view are put on the screen from left to right.

XML description

tree

Each tree view must start with this tag.

- on_write: see form-attributes-on_write.
- editable: A boolean to specify if the list is editable.
- creatable: A boolean to specify if the editable list can be used to create new record. The default value is true.
- sequence: The name of the field that is used for sorting. This field must be an integer and it will be updated to match the new sort order when the user uses Drag and Drop on list rows.
- keyword_open: A boolean to specify if the client should look for a tree_open action on double click instead of switching view.
- tree_state: A boolean to specify if the client should save the state of the tree.
- visual: A PYSON statement that will be evaluated as string muted, success, warning or danger with the context of the record to provide a visual context to the row.

field

- name: The name of the field.
- readonly: Boolean to set the field readonly.
- widget: The widget that must be used instead of the default one.
- tree_invisible: A string of PYSON statement that will be evaluated as boolean with the context of the view to display or not the column.
- optional: A boolean to define if the column is hidden or not. Defining the optional attribute allows each user to show/hide the column. The attribute value is used as default when the user has no custom setting for it.
- visual: A PYSON statement that will be evaluated as string muted, success, warning or danger with the context of the record to provide a visual context to the field.
- icon: The name of the field that contains the name of the icon to display in the column.
- sum: A text for the sum widget that will be added on the bottom of list with the sum of all the fields in the column.
- width: Set the width of the column.
- expand: An integer that specifies if the column should be expanded to take available extra space in the view. This space is shared proportionally among all columns that have their “expand” property set. Resize doesn’t work if this option is enabled.
• **pre_validate**: see in *common-attributes-pre_validate*.
• **completion**: see in *common-attributes-completion*.
• **factor**: see in *common-attributes-factor*.
• **symbol**: see in *common-attributes-symbol*.
• **help_field**: see in *common-attributes-help_field*.

**prefix or suffix**

A field could contain one or many prefix or suffix that will be displayed in the same column.

• **string**: The text that will be displayed.
• **name**: The name of the field whose value will be displayed.
• **icon**: The image name or the field name which contains the image name. For the icon type it must be the name of a record of *ir.ui.icon*. For the url type it must be the URL. It can be relative to the server.
• **icon_type**: the type of icon source. Available values are icon or url. The default value is icon.
• **url_size**: the name of the size parameter to add to the URL.

**button**

Same as in *form-button*.

**Example**

```xml
<tree sequence="sequence">
  <field name="name"/>
  <field name="percentage">
    <suffix name="percentage" string="%"/>
  </field>
  <field name="group"/>
  <field name="type"/>
  <field name="active"/>
  <field name="sequence" tree_invisible="1"/>
</tree>
```

**button**

Display a button.

• **string**: The string that will be displayed inside the button.
• **name**: The name of the function that will be called. The function must have this syntax:
  
  button(cls, records)

• **confirm**: A text that will be displayed in a confirmation popup when the button is clicked.
• **help**: see in *common-attributes-help*
List-Form view

The list-form views use the same schema as the form views.
List-forms display records as a list of editable forms.

Note: The performance of the list-form does not allow to scale well for large number of records

Graph view

The RNG that describes the xml for a graph view is stored in trytond/ir/ui/graph.rng. There is also a RNC in trytond/ir/ui/graph.rnc.

XML description

graph

Each graph view must start with this tag.

- type: vbar, hbar, line, pie
- background: an hexaecimal value for the color of the background.
- color: the main color.
- legend: a boolean to specify if the legend must be displayed.

x, y

Describe the field that must be used for axis. x must contain only one tag field and y must at least one but may contain many.

field

- name: the name of the field on the object to use.
- string: allow to override the string that comes from the object.
- key: can be used to distinguish fields with the same name but with different domain.
- domain: a PySON string which is evaluated with the object value as context. If the result is true the field value is added to the graph.
- fill: defined if the graph shall be filled.
- empty: defined if the line graph must put a point for missing dates.
- color: the color of the field.
- interpolation: defined how the line graph must interpolate points. The default is linear.
  - constant-center: use the value of the nearest point, see Nearest-neighbor interpolation
  - constant-left: use the value of the nearest left point.
  - constant-right: use the value of the nearest right point.
– **linear**: see linear interpolation

**Example**

```xml
<graph string="Invoice by date" type="vbar">
  <x>
    <field name="invoice_date"/>
  </x>
  <y>
    <field name="total_amount"/>
  </y>
</graph>
```

**Board view**

The RNG that describes the xml for a board view is stored in trytond/ir/ui/board.rng. There is also a RNC in trytond/ir/ui/graph.rnc.

Board view is used to display multiple views at once.

Elements are put on the screen following the same rules as for Form view.

The views can be updated by the selection of records on an other view inside the same board by using in the domain the `active_id` or `active_ids` from the `_actions` dictionary with the action id of the other view as key. For example: ..

```xml
<field name="domain" pyson="1" eval="[('field', '=', Eval('_actions', {}).get('module.action_id', {})).get('module.action_id', '{}').get('active_id'))]"/>
```

**XML description**

**board**

Each board view must start with this tag.

- **col**: see in *common-attributes-col*.

**image**

Same as in Form view.

**separator**

Same as in Form view.
label

Same as in Form view.

newline

Same as in Form view.

notebook

Same as in Form view.

group

Same as in Form view.

hpaned, vpaned

Same as in Form view.

child

Same as in Form view.

action

- **name**: The id of the action window.
- **colspan**: see in common-attributes-colspan.

Calendar view

The RNG that describes the xml for a calendar view is stored in trytond/ir/ui/calendar.rng. There is also a RNC in trytond/ir/ui/calendar.rnc.

Calendar view is use to display records as events on a calendar based on a dtstart and optionally a dtend.
XML description

calendar

Each calendar view must start with this tag.

- **dtstart**: The name of the field that contains the start date.
- **dtend**: The name of the field that contains the end date.
- **mode**: An optional name for the view that will be used first. Available views are: day, week and month. The default value is month.
- **editable**: A boolean to specify if the calendar is editable. The default value is True.
- **color**: An optional field name that contains the text color for the event. The default value is black.
- **background_color**: An optional field name that contains the background color for the event. The default value is light blue.
- **width**: The minimum width the calendar should request, use -1 to unset.
- **height**: The minimum height the calendar should request, use -1 to unset.

field

- **name**: The name of the field.

Example

```
<calendar dtstart="planned_date">
  <field name="code"/>
  <field name="product"/>
  <field name="reference"/>
</calendar>
```

5.1.17 Extending View

Extending a view means, that the original view will be modified by a set of rules which are defined with XML.

For this purpose, the extension engine uses xpath expressions.

The view is defined with the field **inherit** of the ir.ui.view.

If the field domain (a Python string representation of a **domain**) is not set or evaluated to True, the inheritance will be proceeded.
XML Description

data

Each view must start with this tag.

xpath

• expr: the xpath expression to find the nodes in the inherited view.
• position: Define the position in relation to the nodes found. It can be before, after, replace, inside or replace_attributes which will change the attributes.

Example

```
<data>
  <xpath
    expr="/form/notebook/page/separato[@name="signature"]"
    position="before">
    <label name="company"/>
    <field name="company"/>
    <label name="employee"/>
    <field name="employee"/>
  </xpath>
</data>
```

5.1.18 Wizard

A wizard describes a series of steps defined as `trytond.wizard.State`. The wizard stores data in `ir.session`. wizard between states.

The basics:

• Each wizard is a Python class that subclasses `trytond.wizard.Wizard`.
• The states of the wizard are attributes that are instances of `trytond.wizard.State`.

Example

This example defines a wizard which export translations

```
from trytond.wizard import Wizard, StateView, StateTransition, Button
from trytond.pool import Pool

class TranslationExport(Wizard):
    "Export translation"
    __name__ = "ir.translation.export"

    start = StateView('ir.translation.export.start',
                      'ir.translation_export_start_view_form', [
                      Button('Cancel', 'end', 'tryton-cancel'),
```
(continues on next page)
Button('Export', 'export', 'tryton-ok', default=True),
]
export = StateTransition()
result = StateView('ir.translation.export.result',
    'ir.translation_export_result_view_form', [
        Button('Close', 'end', 'tryton-close'),
    ])

def transition_export(self):
    pool = Pool()
    translation_obj = pool.get('ir.translation')
    file_data = translation_obj.translation_export(
        self.start.language.code, self.start.module.name)
    self.result.file = buffer(file_data)
    return 'result'

def default_result(self, fields):
    return {'file': self.result.file,}

Pool.register(TranslationExport, type_='wizard')

The class must be registered in the Pool.

5.1.19 Reports

Tryton can generate dynamic reports in many formats from templates. The reports are generated in one step as follows: a report template in a special file format, explained later, is interpolated with dynamic data and placed into a document of the same file format. Tryton’s ability to generate documents in this way allows documents to be generated for any editor that supports the Open Document Format which can be converted to third party formats, such as PDF. LibreOffice must be installed on the server host for format conversion.

Report Templates

Report templates are files with a format supported by relatorio, that contain snippets of the Genshi templating language. Here is an example of the text that would be placed in an open document text document, *.odt, that displays the full name and the address lines of the first address of each party. The genshi code is placed in the template using Functions->Placeholder->Text Fields. These are specific to ODT files.

Report API

Python API

XML Description

When defining an ir.action.report the following attributes are available:

- name: The name of the report.
• **report_name**: The name of the report model, for example `my_module.my_report`. This is the name you would use with `Pool().get`

• **model**: If this report is of an existing model this is its name. For example `my_module.my_model`. Custom reports that aren’t of a specific model will need to leave this blank.

• **report**: The path to the template file starting with the module, for example `my_module/my_report.odt`.

• **template_extension**: The template format.

• **single**: `True` if the template works only for one record. If such report is called with more than one record, a zip file containing all the reports will be generated.

• **record_name**: A Genshi Expression to compute the filename for each record.

**Report Usage**

**Using genshi and open office reports**

**Setting up an ODT file**

If you are creating a report from scratch you should perform the following steps:

• Remove user data
  – “File > Properties…”
  – Uncheck “Apply user data”
  – Click on “Reset”

• Select Style and Formatting
  – Press F11 or “Format > Style and Formatting”
  – Click on the drop down at the right top
  – Select “Load Styles”
  – Click on “From File…”
  – Select a existing report (`company/header_A4.odt`)

• Set some parameters
  – Set the zoom to 100% (View>Zoom)
  – Set the document in read-only mode (File>Properties>Security) (Decreases the time it takes to open the document.)

• Usage
  – Use Liberation fonts (Only necessary if being officially included in Tryton)
  – Try to use styles in report templates so that they can be extended.
Using Genshi in an ODT file

The genshi code is placed in the template using Functions->Placeholder->Text Fields. These are specific to *.odt files and can be found in the open office menu at Insert -> Fields -> Other and then Functions -> Placeholder -> Text. Type genshi code into the Placeholder field. There are alternatives for embedding genshi that are supported by relatorio but their use is not encouraged within Tryton.

Also note that relatorio only supports a subset of genshi. The directives that are supported by relatorio can be found here: Quick Example.

See genshi’s documentation for more information: Genshi XML Templates

Examples

The modules company, account_invoice and stock all contain helpful examples.

Also see relatorio’s site for some examples:

- Quick Example
- In Depth Introduction
- Example Documents

Accessing models from within the report

By default instances of the models the report is for are passed in to the report via a list of objects called records (or record if single is True). These records behave just as they would within trytond itself. You can access any of the models relations as well. For example within the invoice report each object is an invoice and you can access the name of the party of the invoice via invoice.party.name. Additional objects can be passed to a report. This is discussed below in Passing custom data to a report.

Within Tryton the underlying model the report can be found by following the Menu to Administration > UI > Actions > Report. Furthermore in tryton the fields for that model can be found by following the menu to Administration > Model > Model. Model relation fields can be accessed to any depth, for example, one could access invoice.party.addresses to get a list of addresses for the party of an invoice.

Creating a simple report template for a model from client

Once you have created a report template it has to be uploaded to trytond. This can be done by creating a new record in the Administration > UI > Actions > Report menu. Just make sure to include the template file in the content field.

In order to make the report printable from a model create a “Print form” keyword related to the model where the report should be available.
Creating a simple report template for a model in XML

Less work has to be done if you just want a simple report representation of a model. There are just 2 steps. First, create a report template file in a format supported by relatorio. Second, describe your report in XML making sure to define the correct report_name and model.

Replacing existing Tryton reports

To replace an existing report you must deactivate the old report and activate the new report.

For example to deactivate the sale report:

```xml
<record model="ir.action.report" id="sale.report_sale">
    <field name="active" eval="False"/>
</record>
```

Then you must activate the new sale report that exists in your new module:

```xml
<record model="ir.action.report" id="report_sale">
    <field name="name">Sale</field>
    <field name="report_name">sale.sale</field>
    <field name="model">sale.sale</field>
    <field name="report">my_module/sale.odt</field>
    <field name="template_extension">odt</field>
</record>
```

And create the keyword for the new report:

```xml
<record model="ir.action.keyword" id="report_sale_keyword">
    <field name="keyword">form_print</field>
    <field name="model">sale.sale,-1</field>
    <field name="action" ref="report_sale"/>
</record>
```

Passing custom data to a report

In this example Report.get_context is overridden and an employee object is set into context. Now the invoice report will be able to access the employee object.

```python
from trytond.report import Report
from tryton.pool import Pool

class InvoiceReport(Report):
    __name__ = 'account.invoice'

    @classmethod
    def get_context(cls, records, header, data):
        pool = Pool()
        Employee = pool.get('company.employee')
        context = super().get_context(records, header, data)
        employee_id = Transaction().context.get('employee')
```

(continues on next page)
employee = Employee(employee_id) if employee_id else None
context['employee'] = employee

return context

Pool.register(InvoiceReport, type_='report')

5.1.20 Remote Procedure Call

There are two protocols supported by trytond: JSON-RPC (Version 1.0) and XML-RPC. The URL of the calls must end with the database name with a trailing ‘/’.

The available methods are:

common.db.login

It takes as parameters: the user name and a dictionary of login parameters. It returns in case of success the user ID and the session. If the parameters are not valid to authenticate the user, it returns nothing. Otherwise if it misses a key in the parameters, it raises a LoginException exception with the missing key name, type and the message to ask to the user.

common.db.logout

It takes no parameters and it invalidate the current session.

Authorization

Most of the calls require authorization, there are two methods:

Basic

It follows the Basic access authentication.

Session

The authorization field is constructed by the username, the user ID and the session combined with a single colon and encoded in Base64. The session is retrieved by calling the method common.db.login.
5.1.21 Task Queue

Tryton provides a way to run asynchronously some tasks. You must activate the worker in the queue section of the configuration and run the worker manager otherwise the tasks will be run at the end of the transaction.

A task is the parameters that defines how to call a method from a trytond.model.Model. This include the trytond.transaction.Transaction.context, the trytond.transaction.Transaction.user and the arguments. The first argument of the method must be an instance or a list of instances of trytond.model.Model. This other arguments must be JSON-ifiable.

A task is pushed into the queue by calling the desired method on the trytond.model.Model.__queue__. This stores in the queue all the current parameters of the call and it will be execute by a worker or at the end of the transaction if no worker is configured. The following trytond.transaction.Transaction.context keys are used as parameters for the queue:

- queue_name: The name of the queue. Default value is default.
- queue_scheduled_at: A datetime.timedelta to add to current time to define when the task should be started. Default value is None which means directly.
- queue_expected_at: A datetime.timedelta to add to current time to define when the task should be finished. Default value is None which means as soon as possible.

Warning: There is no access right verification during the execution of the task.

Example:

```python
from trytond.model import Model

class MyModel(Model):
    "My Model"
    __name__ = 'my_model'

    @classmethod
    def launch(cls, records):
        for record in records:
            cls.__queue__.process(record, 42)

    def process(self, value):
        self.value = value
```

5.1.22 Scheduled Actions

Tryton provides a scheduler (aka cron) which can execute methods of models periodically at set intervals.

The planning is managed by ir.cron records which store the method to call and the interval of time between calls. The method must be a class method of a Model which can be called without any parameters.

To register a new method with the scheduler, you must extend the ir.cron model and append the new method to the selection attribute of the method field in __setup__(). The name of the selection must be the model name and the method name joined together with a | between them.

Example:
class Cron(metaclass=PoolMeta):
    __name__ = 'ir.cron'

    @classmethod
    def __setup__(cls):
        super().__setup__()
        cls.method.selection.append((
            'my_model|my_method', 'Run my method'),
        )

class MyModel(Model):
    "My Model"
    __name__ = 'my_model'

    @classmethod
    def my_method(cls):
        pass

5.1.23 User Application

Tryton provides a way to connect URL rules to a callable endpoint using the decorator method route of the trytond.application.app instance. This allows you to define a custom API based on HTTP that can be used to create a specific user application.

The decorator takes as first parameter a string which follow the Rule Format of Werkzeug and as second parameter sequence of HTTP methods.

Example:

```python
from trytond.application import app

@app.route('/hello', methods=['GET'])
def hello(request):
    return 'Hello world'
```

The following converter is added by Tryton:

- **base64**: This converter accepts any Base64 string and transforms it into its corresponding bytes value.

Tryton also provides some wrappers in trytond.protocols.wrappers to ease the creation of such route.

- **set_max_request_size(size)**: change the default limit of the request to the size in bytes.
- **allow_null_origin**: allow requests which have their Origin set to null.
- **with_pool**: which takes the first parameter as database name and replace it by the corresponding instance of the Pool.
- **with_transaction([readonly])**: which starts a Transaction using the Pool from with_pool. If readonly is not set, the transaction will not be readonly for POST, PUT, DELETE and PATCH methods and readonly for all others.

5.1. Using trytond
• `user_application(name[, json])`: which set the `Transaction.user` from the `Authorization` header using the type `bearer` and a valid key for the named user application.

### User Application Key

Tryton also provides an easy way to manage access to user application using keys per named application. A key is created with a `POST` request on the URL `/<database_name>/user/application/` which returns the key. The request must contain as data a json object with the keys:

- `user`: the user login
- `application`: the name of the application

After the creation, the key must be validated by the user from the preferences of a Tryton client.

A key can be deleted with a `DELETE` request on the same URL. The request must contain as data a json object with the keys:

- `user`: the user login
- `key`: the key to delete
- `application`: the name of the application of the key

### 5.1.24 Sending notifications

Tryton embeds a bus system allowing the system to send text messages to clients logged in the system.

It allows the server to warn quickly the client user about some events using the `trytond.bus.notify()` function. Sending the notifications is done in a transactional way and will occur at then end of the transaction.

For example, we warn the user of low stock level when selecting a product:

```python
from trytond.bus import notify
class SaleLine:
    __name__ = 'sale.line'

    def on_change_product(self):
        super().on_change_product()

        # compute the product current stock
        stock = ...

        if stock < 0:
            notify('Not enough stock', priority=3)
```
5.1.25 Modules

The modules of Tryton extend the functionality of the platform. The server comes by default with only a basic functionality included in these modules: ir, res.

Module Structure

A module is a directory in trytond/modules which contains at least two files:

- __init__.py: a Tryton module must be a Python module.
- tryton.cfg: a Configuration file that describes the Tryton module.

__init__.py file

It is the Python __init__.py to define a module. It must contains a method named register() that must register to the pool all the objects of the module.

tryton.cfg file

It is a configuration file using the format of ConfigParser that must contain tryton section with this following name:

- version: The version number of the module.
- depends: A one per line list of modules on which this module depends.
- extras_depend: A one per line list of modules on which this module may depend.
- xml: The one per line list of the XML files of the module. They will be loaded in the given order when the module is activated or updated.

Here is an example:

```python
[tryton]
version=0.0.1
depends:
  ir
tres
country
xml:
  party.xml
category.xml
address.xml
contact_mechanism.xml
```
Python Files

The Python files define the models for the modules.

XML Files

The XML files define data that will be inserted into the database.

There is an rnc for those files stored in trytond/tryton.rnc.

The following snippet gives a first idea of what an xml file looks:

```xml
<?xml version="1.0"?>
<tryton>
  <data>
    <record model="res.group" id="group_party_admin">
      <field name="name">Party Administration</field>
    </record>
    <record model="res.user-res.group"
            id="user_admin_group_party_admin">
      <field name="user" ref="res.user_admin"/>
      <field name="group" ref="group_party_admin"/>
    </record>
    <menuitem name="Party Management" sequence="0" id="menu_party"
              icon="tryton-users"/>
    <record model="ir.ui.view" id="party_view_tree">
      <field name="model">party.party</field>
      <field name="type">tree</field>
      <field name="arch">
        <![CDATA[
        <tree string="Parties">
          <field name="code"/>
          <field name="name"/>
          <field name="lang"/>
          <field name="vat_code"/>
          <field name="active" tree_invisible="1"/>
          <field name="vat_country" tree_invisible="1"/>
          <field name="vat_number" tree_invisible="1"/>
        </tree>]]>
      </field>
    </record>
  </data>
</tryton>
```

Here is the list of the tags:

- `tryton`: The main tag of the xml
- `data`: Define a set of data inside the file. It can have the attributes:
  - `noupdate` to prevent the framework to update the records,
  - `depends` to import data only if all modules in the comma separated module list value are activated,
- grouped to create records at the end with a grouped call.
- language to import data only if the language is translatable.

- **record**: Create a record of the model defined by the attribute `model` in the database. The `id` attribute can be used to refer to the record later in any xml file.

- **field**: Set the value of the field with the name defined by the attribute `name`.

Here is the list of attributes:

- **search**: Only for relation field. It contains a domain which is used to search for the value to use. The first value found will be used.
- **ref**: Only for relation field. It contains an xml id of the relation to use as value. It must be prefixed by the module name with an ending dot, if the record is defined in an other module.
- **eval**: Python code to evaluate and use result as value. The following expressions are available:
  - `time`: The python `time` module
  - `version`: The current Tryton version
  - `ref`: A function that converts an XML id into a database id.
  - `Decimal`: The python `Decimal` object
  - `datetime`: The python `datetime` module
- **pyson**: convert the evaluated value into `PYSON` string.
- **depends**: set value only if all modules in the comma separated module list value are activated.

---

**Note**: Field content is considered as a string. So for fields that require other types, it is required to use the `eval` attribute.

- **menuitem**: Shortcut to create `ir.ui.menu` records.

Here is the list of attributes:

- **id**: The id of the menu.
- **name**: The name of the menu.
- **icon**: The icon of the menu.
- **sequence**: The sequence value used to order the menu entries.
- **parent**: The xml id of the parent menu.
- **action**: The xml id of the action linked to the menu.
- **groups**: A list of xml id of group, that have access to the menu, separated by commas.
- **active**: A boolean telling if the menu is active or not.
5.1.26 Translation

The translation of the user interface is provided module-wise. Translations are stored in the `locale/` directory of a module, each language in a PO-file. The official language files are named after the POSIX locale standard, e.g. `de_DE.po`, `es.po`, `es_AR.po`, `es_EC.po`...

The names of custom language files must match the code of the language in the Model `ir.lang`.

If a language is set `translatable`, the translations will be loaded into the database on each trytond module update. Tryton supports derivative translations. This means that if the translation of a term is missing in one language, it will search on the parent languages. Also when activate a children language, you must also activate all parents.

Translation Wizards

Set Translations

The wizard adds new translations to the base language `en`.

Clean Translations

The wizard deletes obsolete translations from the database.

Synchronize Translations

The wizard updates the translations of the selected language based on the translations of the base language `en`. It will also remove duplicate translations with its direct parent.

Export Translations

The wizard requires to select a language and a module and will export the translations for this selection into a PO-file.

Override translations

Translations of a module can be overridden by another module. This can be done by putting a PO file into the `locale/override` directory of the module that shall contain the translations to override.

To override the translation of another module the `msgctxt` string must have the following content:

```
type:name:module.xml_id
```

- `type`: Value of the field type of `ir.translation`.
- `name`: Value of the field name of `ir.translation`.
- `module`: Value of the field module `ir.translation`.
- `xml_id`: The XML id that is stored in `ir.model.data` as `fs_id`

The `xml_id` part is optional and can be omitted if it is `None`. 
5.1.27 Testing

Tryton supports both functional and unit tests.

Testing your module

Functional tests

Functional tests are written as doctests using proteus.

Unit tests

Tryton provides the `ModuleTestCase` class that bundles a set of tests that are useful for every module.

Unit tests in `ModuleTestCase` can be decorated with `trytond.tests.test_tryton.with_transaction()` to run the test in a transaction.

To use it in your own module you just have to inherit from `ModuleTestCase` and set the class attribute `module` to the name of your module.

```python
from trytond.tests.test_tryton import ModuleTestCase, with_transaction

class MyModuleTestCase(ModuleTestCase):
    "My Module Test Case"
    module = 'my_module'

    @with_transaction()
    def test_method(self):
        "Test method"
        self.assertTrue(True)
```

Tests from this modules are found by the function `trytond.modules.my_module.tests.suite` which must return a `unittest.TestSuite` containing all the module’s tests. This function is called by the Tryton test runner script to gather all the tests.

A typical `suite()` function thus looks like this:

```python
def suite():
    suite = trytond.tests.test_tryton.suite()
    suite.addTests(unittest.TestLoader().loadTestsFromTestCase(MyModuleTestCase))
    suite.addTests(doctest.DocFileSuite('scenario_my_module.rst',
        tearDown=doctest_teardown, encoding='utf-8',
        optionflags=doctest.REPORT_ONLY_FIRST_FAILURE,
        checker=doctest_checker))
    return suite
```
Running your module’s tests

Tryton provides the script `tryond/tests/run-tests.py`, just invoke it like that:

```
run-tests.py -m my_module
```

Testing tryond

Extending tests

Python modules extending tryton core can define additional tests that should be added to the existing ones. Those modules must create an entry point `tryond.tests`. Any file in the module path specified by this entry point starting with `test_` and ending by `.py` will be imported. Each of those file must define a `suite()` function that returns a `unittest.TestSuite` that will be included in the trytond test suite. If the module from the entry point defines a `register` function it will be called when registering the test-specific models in the `tryond.pool.Pool`.

Running tryond tests

You should use the script `tryond/tests/run-tests.py` by invoking it like that:

```
run-tests.py [-c configuration]
```

You can use a different configuration file to check tryond against different backend.

5.2 API Reference

5.2.1 Models

Model API reference.

Model

```
class tryond.model.Model([id, **kwargs])
```

This is the base class that every kind of `model` inherits. It defines common attributes of all models.

Class attributes are:

- `__name__`: It contains the a unique name to reference the model throughout the platform.
- `__access__`: A set that contains the names of relation field for which the access rights are also checked for this model.
- `__rpc__`: It contains a dictionary with method name as key and an instance of `tryond.rpc.RPC` as value.
- `__rec_name__`: It contains the name of the field used as name of records. The default value is ‘name’.
- `id`: The definition of the field `id` of records.
Model.__queue__
It returns a queue caller for the model. The called method will be pushed into the queue.

Model.__fields__
It contains a dictionary with the field name as key and its field instance as value.

Model.__record__
It stores the record class to store internally the values of the instances.

Model.__defaults__
It contains a dictionary with the field name as key and its default method as value.

Class methods:

classmethod Model.__setup__()  
Setup the class before adding into the trytond.pool.Pool.

classmethod Model.__post_setup__()  
Setup the class after added into the trytond.pool.Pool.

classmethod Model.__register__(module_name)  
Registers the model in ir.model and ir.model.field.

classmethod Model.default_get(fields_names[, with_rec_name])  
Returns a dictionary with the default values for each field in fields_names. Default values are defined by the returned value of each instance method with the pattern default_`field_name`() . with_rec_name allow to add rec_name value for each many2one field. The default_rec_name key in the context can be used to define the value of the Model._rec_name field.

classmethod Model.fields_get([fields_names[, level]])  
Return the definition of each field on the model. The level defines the number of relations to include in the relation field definition.

classmethod Model.__names__(field)  
Returns a dictionary with the name of the model and the field. It is a convenience-method used to format messages which should include those names.

Instance methods:

Model.on_change(fieldnames)  
Returns the list of changes by calling on_change method of each field.

Model.on_change_with(fieldnames)  
Returns the new values of all fields by calling on_change_with method of each field.

Model.pre_validate()  
This method is called by the client to validate the instance.

ModelView

class trytond.model.ModelView  
It adds requirements to display a view of the model in the client.

Class attributes:

ModelView._buttons
It contains a dictionary with button name as key and the states dictionary for the button. The states possible keys are invisible, readonly and icon which have a PYSON statement as value and depends which has a list of field names on which the button states depend. The keys will be set as default attributes on the buttons for the views that show them.
Static methods:

**static** ModelView.button()

Decorate button method to check group access and rule.

**static** ModelView.button_action(action)

Same as ModelView.button() but return the action id of the XML id action or the action value updated by the returned value of the method.

**static** ModelView.button_change([\*fields, methods])

Same as ModelView.button() but for button that change values of the fields on client side (similar to on_change). The methods argument can be used to duplicate the field names from other decorated methods. This is useful if the decorated method calls another method.

**Warning:** Only on instance methods.

Class methods:

**classmethod** ModelView.fields_view_get([view_id[, view_type[, level]]])

Return a view definition used by the client. The definition is:

```python
{
    'model': model name,
    'type': view type,
    'view_id': view id,
    'arch': XML description,
    'fields': {
        field name: {
            ...,
        },
    },
    'field_childs': field for tree,
}
```

**classmethod** ModelView.view_toolbar_get()

Returns the model specific actions and exports in a dictionary with keys:

- print: a list of available reports
- action: a list of available actions
- relate: a list of available relations
- exports: a list of available exports

**classmethod** ModelView.view_attributes()

Returns a list of XPath, attribute, value and an optional depends list. Each element from the XPath will get the attribute set with the JSON encoded value. If the depends list is set its fields are added to the view if the xpath matches at least one element.

**Note:** The `view_id` is set to the context when calling this method.

**classmethod** ModelView.parse_view(tree, type, view_id[, field_children[, level[, view_depends]]])

Return the sanitized XML and the corresponding fields definition.
Note: This method is public mainly to allow modification the existing XML of the view by code.

ModelStorage

class trytond.model.ModelStorage

It adds storage capability.

Class attributes are:

ModelStorage.create_uid

The definition of the trytond.model.fields.Many2One field create_uid of records. It contains the id of the user who creates the record.

ModelStorage.create_date

The definition of the trytond.model.fields.DateTime field create_date of records. It contains the date-time of the creation of the record.

ModelStorage.write_uid

The definition of the trytond.model.fields.Many2One field write_uid of the records. It contains the id of the last user who writes on the record.

ModelStorage.write_date

The definition of the trytond.model.fields.DateTime field write_date of the records. It contains the datetime of the last write on the record.

ModelStorage.rec_name

The definition of the trytond.model.fields.Function field rec_name. It is used in the client to display the records with a single string.

Static methods:

static ModelStorage.default_create_uid()

Return the default value for create_uid.

static ModelStorage.default_create_date()

Return the default value for create_date.

Class methods:

classmethod ModelStorage.create(vlist)

Create records. vlist is list of dictionaries with fields names as key and created values as value and return the list of new instances.

classmethod ModelStorage.trigger_create(records)

Trigger create actions. It will call actions defined in ir.trigger if on_create is set and condition is true.

classmethod ModelStorage.read(ids, fields_names)

Return a list of dictionary for the record ids. The dictionary is composed of the fields as key and their values. fields_names can contain dereferenced fields from related models. Their values will be returned under the referencing field suffixed by a .. The number of dots in the name is not limited. The order of the returned list is not guaranteed.

classmethod ModelStorage.index_get_field(name)

Return the index sort order of the field get calls.

classmethod ModelStorage.write(records, values[
[records, values], ...])

Write values on the list of records. values is a dictionary with fields names as key and written values as value.
classmethod ModelStorage.trigger_write_get_eligibles(records)
    Return eligible records for write actions by triggers. This dictionary is to pass to trigger_write().

classmethod ModelStorage.trigger_write(eligibles)
    Trigger write actions. It will call actions defined in ir.trigger if on_write is set and condition was false before write() and true after.

classmethod ModelStorage.index_set_field(name)
    Return the index sort order of the field set calls.

classmethod ModelStorage.delete(records)
    Delete records.

classmethod ModelStorage.trigger_delete(records)
    Trigger delete actions. It will call actions defined in ir.trigger if on_delete is set and condition is true.

classmethod ModelStorage.copy(records, default)
    Duplicate the records. default is a dictionary of default value per field name for the created records.
    The values of default may be also callable that take a dictionary containing the fields and values of the record copied and return of the value.
    The keys of default may use the dotted notation for the fields.One2Many to define the default to pass to its copy operation.
    New records are returned following the input order.

classmethod ModelStorage.search(domain, offset, limit, order, count)
    Return a list of records that match the domain.
    If offset or limit are set, the result starts at the offset and has the length of the limit.
    The order is a list of tuples defining the order of the result:
    [ ('field name', 'ASC'), ('other field name', 'DESC'), ... ]
    The first element of the tuple is a field name of the model and the second is the sort ordering as ASC for ascending, DESC for descending or empty for a default order. This second element may contain 'NULLS FIRST' or 'NULLS LAST' to sort null values before or after non-null values. If neither is specified the default behavior of the backend is used.
    In case the field used is a fields.Many2One, it is also possible to use the dotted notation to sort on a specific field from the target record. Or for a fields.Dict field, the dotted notation is used to sort on the key’s value.
    If count is set to True, then the result is the number of records. The count result is limited upto the value of limit if set.

classmethod ModelStorage.search_count(domain, offset, limit)
    Return the number of records that match the domain. The result is limited upto the value of limit if set and reduced by offset.

classmethod ModelStorage.search_read(domain, offset, limit, order, fields_names)
    Call search() and read() at once. Usefull for the client to reduce the number of calls.

classmethod ModelStorage.search_rec_name(name, clause)
    Searcher for the trytond.model.fields.Function field rec_name.

classmethod ModelStorage.search_global(cls, text)
    Yield tuples (record, name, icon) for records matching text. It is used for the global search.

classmethod ModelStorage.count()
    Return an estimation of the number of records stored.
classmethod ModelStorage.browse(ids)
    Return a list of record instance for the ids.

classmethod ModelStorage.export_data(records, fields_names)
    Return a list of list of values for each records. The list of values follows fields_names. Relational fields are
defined with / at any depth. Descriptor on fields are available by appending . and the name of the method on
the field that returns the descriptor.

classmethod ModelStorage.export_data_domain(domain, fields_names[, offset[, limit[, order ]]])
    Call search() and `meth'export_data` together. Useful for the client to reduce the number of calls and the data
transfered.

classmethod ModelStorage.import_data(fields_names, data)
    Create or update records for all values in data. The field names of values must be defined in fields_names. It
returns the number of imported records.

classmethod ModelStorage.check_xml_record(records, values)
    Verify if the records are originating from XML data. It is used to prevent modification of data coming from XML
files. This method must be overiden to change this behavior.

classmethod ModelStorage.validate(records)
    Validate the integrity of records after creation and modification. This method must be overridden to add valida-
tion and must raise an exception if validation fails.

Dual methods:

classmethod ModelStorage.save(records)
    Save the modification made on the records.

Instance methods:

ModelStorage.resources()
    Return a dictionary with the number of attachments (attachment_count), notes (note_count) and unread
note (note_unread).

ModelStorage.get_rec_name(name)
    Getter for the `trytond.model.fields.Function` field rec_name.

ModelSQL

class trytond.model.ModelSQL
    It implements ModelStorage for an SQL database.

Class attributes are:

ModelSQL._table
    The name of the database table which is mapped to the class. If not set, the value of Model._name is used with
dots converted to underscores.

ModelSQL._order
    The default order parameter of ModelStorage.search() method.

ModelSQL._order_name
    The name of the field (or an SQL statement) on which the records must be sorted when sorting on a field refering
to the model. If not set, ModelStorage._rec_name will be used.

ModelSQL._history
    If true, all changes on records will be stored in a history table.

ModelSQL._sql_constraints
    A list of SQL constraints that are added on the table:
constraint name is the name of the SQL constraint in the database
• constraint is an instance of Constraint
• message id for trytond.i18n.gettext()

Class methods:

classmethod ModelSQL.__table__()  
Return a SQL Table instance for the Model.

classmethod ModelSQL.__table_history__()  
Return a SQL Table instance for the history of Model.

classmethod ModelSQL.__table_handler__([module_name[, history ]])  
Return a TableHandler for the Model.

classmethod ModelSQL.table_query()  
Could be defined to use a custom SQL query instead of a table of the database. It should return a SQL FromItem.

Warning: By default all CRUD operation will raise an error on models implementing this method so the create, write and delete methods may also been overriden if needed.

classmethod ModelSQL.history_revisions(ids)  
Return a sorted list of all revisions for ids. The list is composed of the date, id and username of the revision.

classmethod ModelSQL.restore_history(ids, datetime)  
Restore the record ids from history at the specified date time. Restoring a record will still generate an entry in the history table.

Warning: No access rights are verified and the records are not validated.

classmethod ModelSQL.restore_history_before(ids, datetime)  
Restore the record ids from history before the specified date time. Restoring a record will still generate an entry in the history table.

Warning: No access rights are verified and the records are not validated.

classmethod ModelSQL.search(domain[, offset[, limit[, order[, count[, query ]]]]])  
Same as ModelStorage.search() with the additional query argument. If query is set to True, the the result is the SQL query.

classmethod ModelSQL.search_domain(domain[, active_test[, tables ]])  
Convert a domain into a SQL expression by returning the updated tables dictionary and a SQL expression. Where tables is a nested dictionary containing the existing joins:

```python
{
    'None': (<Table invoice>, None),
    'party': {
        'None': (<Table party>, <join_on sql expression>),
        'addresses': {
```

(continues on next page)
None: (<Table address>, <join_on sql expression>),
      
Dual methods:

**classmethod** ModelSQL.lock(records)

Take a lock for update on the records or take a lock on the whole table.

## Constraint

**class** trytond.model.Constraint(table)

It represents a SQL constraint on a table of the database and it follows the API of the python-sql expression.

Instance attributes:

**Constraint.table**

The SQL Table on which the constraint is defined.

## Check

**class** trytond.model.Check(table, expression)

It represents a check Constraint which enforce the validity of the expression.

Instance attributes:

**Check.expression**

The SQL expression to check.

## Unique

**class** trytond.model.Unique(table, \*columns)

It represents a unique Constraint which enforce the uniqueness of the group of columns with respect to all the rows in the table.

Instance attributes:

**Unique.columns**

The tuple of SQL Column instances.

**Unique/operators**

The tuple of Equal operators.
Exclude

class trytond.model.Exclude(table[, (expression, operator), ...[, where ]])

It represents an exclude Constraint which guarantees that if any two rows are compared on the specified expression using the specified operator not all of these comparisons will return TRUE.

Instance attributes:

Exclude.excludes
The tuple of expression and operator.

Exclude.columns
The tuple of expressions.

Exclude.operators
The tuple of operators.

Exclude.where
The clause for which the exclusion applies.

Workflow

class trytond.model.Workflow

A Mix-in class to handle transition check.

Class attribute:

Workflow._transition_state
The name of the field that will be used to check state transition.

Workflow._transitions
A set containing tuples of from and to state.

Static methods:

static Workflow.transition(state)
Decorate method to filter ids for which the transition is valid and finally to update the state of the filtered ids.

ModelSingleton

class trytond.model.ModelSingleton

Modify ModelStorage into a singleton. This means that there will be only one record of this model. It is commonly used to store configuration value.

Class methods:

classmethod ModelSingleton.get_singleton()
Return the instance of the unique record if there is one.
**DictSchemaMixin**

```python
class trytond.model.DictSchemaMixin
```

A mixin for the schema of `trytond.model.fields.Dict` field.

Class attributes are:

**DictSchemaMixin.name**

The definition of the `trytond.model.fields.Char` field for the name of the key.

**DictSchemaMixin.string**

The definition of the `trytond.model.fields.Char` field for the string of the key.

**DictSchemaMixin.help**

The definition of the `trytond.model.fields.Char` field used as the help text for the key.

**DictSchemaMixin.type**

The definition of the `trytond.model.fields.Selection` field for the type of the key. The available types are:

- boolean
- integer
- char
- float
- numeric
- date
- datetime
- selection

**DictSchemaMixin.digits**

The definition of the `trytond.model.fields.Integer` field for the digits number when the type is float or numeric.

**DictSchemaMixin.domain**

A `domain` constraint on the dictionary key that will be enforced only on the client side.

The key must be referenced by its name in the left operator of the domain. The `PYSON` evaluation context used to compute the domain is the dictionary value. Likewise the domain is tested using the dictionary value.

**DictSchemaMixin.selection**

The definition of the `trytond.model.fields.Text` field to store the couple of key and label when the type is selection. The format is a key/label separated by `:` per line.

**DictSchemaMixin.selection_sorted**

If the `selection` must be sorted on label by the client.

**DictSchemaMixin.selection_json**

The definition of the `trytond.model.fields.Function` field to return the JSON version of the `selection`.

Static methods:

```python
static DictSchemaMixin.default_digits()
```

Return the default value for `digits`.

Class methods:

```python
classmethod DictSchemaMixin.get_keys(records)
```

Return the definition of the keys for the records.

---

**5.2. API Reference** 73
classmethod DictSchemaMixin.get_relation_fields()
    Return a dictionary with the field definition of all the keys like the result of Model.fields_get().
    It is possible to disable this method (returns an empty dictionary) by setting in the dict section of the configuration, the Model.__name__ to False.

Instance methods:

DictSchemaMixin.get_selection_json(name)
    Getter for the selection_json.

DictSchemaMixin.format(value[, lang])
    Format the value using the key definition and the language.

MatchMixin

class trytond.model.MatchMixin
    A mixin to add to a Model a match method on pattern. The pattern is a dictionary with field name as key and the value to compare. The record matches the pattern if for all dictionary entries, the value of the record is equal or not defined.

Instance methods:

MatchMixin.match(pattern[, match_none])
    Return if the instance match the pattern. If match_none is set None value of the instance will be compared.

UnionMixin

class trytond.model.UnionMixin
    A mixin to create a ModelSQL which is the UNION of some ModelSQL’s. The ids of each models are sharded to be unique.

Static methods:

static UnionMixin.union_models()
    Return the list of ModelSQL’s names

Class methods:

classmethod UnionMixin.union_shard(column, model)
    Return a SQL expression that shards the column containing record id of model name.

classmethod UnionMixin.union_unshard(record_id)
    Return the original instance of the record for the sharded id.

classmethod UnionMixin.union_column(name, field, table, Model)
    Return the SQL column that corresponds to the field on the union model.

classmethod UnionMixin.union_columns(model)
    Return the SQL table and columns to use for the UNION for the model name.
SymbolMixin

class trytond.model.SymbolMixin

A mixin to manage the display of symbols on the client side.

Instance methods:

SymbolMixin.get_symbol(sign[, symbol])

Return a symbol and its position. The position indicates whether the symbol should appear before (0) or after (1) the value. If no symbol parameter is supplied then the mixin uses the value of attribute named symbol.

DigitsMixin

class trytond.model.DigitsMixin

A mixin to manage the digits of Float.digits and Numeric.digits from a Model.

Instance methods:

DigitsMixin.get_digits()

Return a tuple of two integers to use a digits attribute.

sequence_ordered

trytond.model.sequence_ordered([field_name[, field_label[, order ]]])

Returns a mixin class which defines the order of a ModelSQL with an trytond.model.fields.Integer field. field_name indicates the name of the field to be created and its default values is sequence. field_label defines the label which will be used by the field and defaults to Sequence. Order specifies the order direction and defaults to ASC NULLS FIRST.

MultiValueMixin

class trytond.model.MultiValueMixin

A mixin for Model to help having trytond.model.fields.MultiValue fields with multi-values on a ValueMixin. The values are stored by creating one record per pattern. The patterns are the same as those on MatchMixin.

Class methods:

classmethod MultiValueMixin.multivalue_model(field)

Return the ValueMixin on which the values are stored for the field name. The default is class name suffixed by the field name.

classmethod MultiValueMixin.setter_multivalue(records, name, value, \*\*pattern)

The setter method for the trytond.model.fields.Function fields.

Instance methods:

MultiValueMixin.multivalue_records(field)

Return the list of all ValueMixin records linked to the instance. By default, it returns the value of the first found trytond.model.fields.One2Many linked to the multivalue model or all the records of this one.

MultiValueMixin.multivalue_record(field, \*\*pattern)

Return a new record of ValueMixin linked to the instance.

MultiValueMixin.get_multivalue(name, \*\*pattern)

Return the value of the field name for the pattern.
MultiValueMixin.set_multivalue(name, value[, save], \*pattern)

Store the value of the field name for the pattern. If save is true, it will be stored in the database, otherwise the modified ValueMixin records are returned unsaved. save is true by default.

Warning: To customize the pattern, both methods must be override the same way.

ValueMixin

class trytond.model.ValueMixin

A mixin to store the values of MultiValueMixin.

DeactivableMixin

class trytond.model.DeactivableMixin

A mixin to add soft deletion to the model. It renders all the fields as read-only when the record is inactive.

Class attributes are:

DictSchemaMixin.active

The definition of the trytond.model.fields.Boolean field to store soft deletion state. False values will be considered as soft deletion.

tree

trytond.model.tree([parent[, name[, separator ]]])

Returns a mixin class TreeMixin. parent indicates the name of the field that defines the parent of the tree and its default value is parent. name indicates the name of the field that defines the name of the record and its default value is name. If separator is set, the ModelStorage.get_rec_name() constructs the name by concatenating each parent names using it as separator and ModelStorage.search_rec_name() is adapted to search across the tree.

class trytond.model.TreeMixin

classmethod TreeMixin.check_recursion(records)

 Helper method that checks if there is no recursion in the tree defined by tree().

avatar.mixin

trytond.model.avatar.mixin([size[, default ]])

Returns a mixin AvatarMixin. size defines the size of the avatar image and its default value is 64. default indicates the name of the field to use for generating a default avatar, if it's not set then no default avatar is generated.

class trytond.model.AvatarMixin

AvatarMixin.avatar

The trytond.model.fields.One2Many field used to store the ir.avatar records.

AvatarMixin.avatar

The trytond.model.fields.Binary field that contains the avatar.

AvatarMixin.avatar_url

The trytond.model.fields.Char field that contains the URL for the avatar.
**AvatarMixin**. **has_avatar**
Indicates whether the record has an avatar.

**classmethod AvatarMixin.generate_avatar(records, field)**
Generate a default avatar for each record using the field.

**Fields**

Fields define the behavior of the data on model’s record.

**Field options**

The following arguments are available to all field types. All are optional except Field.string.

**string**

Field.string
A string for the label of the field.

**help**

Field.help
A multi-line help string for the field.

**required**

Field.required
If True, the field is not allowed to be empty. Default is False.

**readonly**

Field.readonly
If True, the field is not editable in the client. Default is False.

**Warning:** For relational fields, it means only the new, delete, add and remove buttons are inactivated. The editable state of the target record must be managed at the target model level.
**domain**

**Field.** `domain`

A `domain` constraint that will be applied on the field value.

**Note:** For `Reference` field it is a dictionary that contains the domain per model name.

**states**

**Field.** `states`

A dictionary that defines dynamic states of the field and overrides the static one. Possible keys are `required`, `readonly` and `invisible`. The values are `PYSON` statements that will be evaluated with the values of the record.

**select**

**Field.** `select`

If true, the content of the field will be indexed.

**on_change**

**Field.** `on_change`

A set of field names. If this attribute is set, the client will call the method `on_change_<field name>` of the model when the user changes the current field value and will give the values of each fields in this list. The method signature is:

```
on_change_<field name>()
```

This method must change the value of the fields to be updated.

**Note:** The `on_change_<field name>` methods are running in a rollbacked transaction.

The set of field names could be filled by using the decorator `depends()`.

**on_change_with**

**Field.** `on_change_with`

A set of field names. Same like `on_change`, but defined the other way around. If this attribute is set, the client will call the method `on_change_with_<field name>` of the model when the user changes one of the fields defined in the list and will give the values of each fields in this list. The method signature is:

```
on_change_with_<field name>()
```
This method must return the new value of the field.

**Note:** The on_change_with_<field name> methods are running in a rollbacked transaction.

The set of field names could be filled by using the decorator `depends()`.

### depends

**Field.**`depends`

A list of field names on which the current one depends. This means that the client will also read these fields even if they are not defined on the view. `Field.depends` is used per example to ensure that `PYSON` statement could be evaluated.

### context

**Field.**`context`

A dictionary which will update the current context for relation field.

**Warning:** The context could only depend on direct field of the record and without context.

### loading

**Field.**`loading`

Define how the field must be loaded: lazy or eager.

### name

**Field.**`name`

The name of the field.

Instance methods:

**Field.**`convert_domain`(domain, tables, Model)

Convert the simple `domain` clause into a SQL expression or a new domain. `tables` could be updated to add new joins.

**Field.**`sql_format`(value)

Convert the value to use as parameter of SQL queries.

**Field.**`sql_type`

Return the namedtuple('SQLType', 'base type') which defines the SQL type to use for creation and casting. Or `None` if the field is not stored in the database.

`sql_type` is using the `_sql_type` attribute to compute its return value. The backend is responsible for the computation.

For the list of supported types by Tryton see `backend types`.

**Field.**`sql_cast`(expression)

Return the SQL expression with cast with the type of the field.
Field `sql_column(table)`
Return the Column instance based on table.

Field `set_rpc(model)`
Adds to model the default RPC instances required by the field.

Field `definition(model, language)`
Returns a dictionary with the definition of the field.

Field `definition_translations(model, language)`
Returns a list of translation sources used by `definition()`.

**Default value**

See `default value`

**Searching**

A class method could be defined for each field which must return a SQL expression for the given domain instead of the default one. The method signature is:

```python
domain_<field name>(domain, tables)
```

Where `domain` is the simple `domain` clause and `tables` is a nested dictionary, see `convert_domain()`.

**Ordering**

A class method could be defined for each field which must return a list of SQL expression on which to order instead of the field. The method signature is:

```python
order_<field name>(tables)
```

Where `tables` is a nested dictionary, see `convert_domain()`.

**Depends**

```python
trytond.model.fields.depends([\*fields, \*methods])
```

A decorator to define the field names on which the decorated method depends. The `methods` argument can be used to duplicate the field names from other decorated methods. This is useful if the decorated method calls another method.

**Field types**

**Boolean**

```python
class trytond.model.fields.Boolean(string[, \**options])
```
A true/false field.
**Integer**

```python
class trytond.model.fields.Integer(string[, **options])
```

An integer field.

**BigInteger**

```python
class trytond.model.fields.BigInteger(string[, **options])
```

A long integer field.

**Char**

```python
class trytond.model.fields.Char(string[, size[, translate[, **options]]])
```

A single line string field.

Search by similarity is used for the `ilike` operator and `is_full_text()` value if the backend supports it and a threshold is set. The similarity threshold is defined for the context key `<model name>.<field name>.search_similarity` or `search_similarity`.

The field is ordered using the similarity with the context value from the key `<model name>.<field name>.order` if it is set.

*Char* has some extra optional arguments:

- **Char.size**
  - The maximum length (in characters) of the field. The size is enforced at the storage level and in the client input.

- **Char.translate**
  - If true, the value of the field is translatable. The value readed and stored will depend on the language defined in the context.

- **Char.autocomplete**
  - A set of field names. If this attribute is set, the client will call the method `autocomplete_<field name>` of the model when the user changes one of those field value. The method signature is:

```python
autocomplete_<field name>()
```

This method must return a list of string that will populate the ComboboxEntry in the client. The set of field names could be filled by using the decorator `depends()`.

- **Char.search_unaccented**
  - If this attribute is set to True, `ilike` searches will be performed on unaccented strings. The default value is True.

  **Warning:** The database backend must supports unaccented search.

- **Char.search_full_text**
  - If this attribute is set to True, `ilike` searches with an `is_full_text()` value use the full text search of the backend. The default value is False.

  The context can be used to force the full text search behaviour. This is done using the key `<model name>.<field name>.search_full_text`. If True, the full text search is used no matter what the value. If False, no full text search is performed.
The full text ranking value is added to the similarity if the `search_full_text` is True.

**Note:** The database backend must support full text search otherwise `ilike` is always used.

---

**Text**

```python
class tryond.model.fields.Text(string, size, translatable, \**options)
```

A multi line string field. `Text` has four extra optional arguments:

- **Text.size**
  - Same as `Char.size`
- **Text.translate**
  - Same as `Char.translate`
- **Text.search_unaccented**
  - Same as `Char.search_unaccented`
- **Text.search_full_text**
  - Same as `Char.search_full_text` The default value is True.

---

**FullText**

```python
class tryond.model.fields.FullText(\**options)
```

An internal field to store a list of parsed strings ordered by weights. The field is ordered using the full text ranking with the context value from the key `<model name>.<field name>.order` if it is set.

---

**Float**

```python
class tryond.model.fields.Float(string, digits, \**options)
```

A floating-point number field. It will be represented in Python by a `float` instance.

**Float** has one extra optional arguments:

- **Float.digits**
  - A tuple of two integers. The first integer defines the total of numbers in the integer part. The second integer defines the total of numbers in the decimal part. Integers can be replaced by a `PYSON` statement. If digits is None or any values of the tuple is None, no validation on the numbers will be done. The tuple can be replaced by a string containing the name of a `Many2One` pointing to a `DigitsMixin`. 

---
**Numeric**

```python
class trytond.model.fields.Numeric(string[, digits[,**options ]])
```

A fixed-point number field. It will be represented in Python by a `decimal.Decimal` instance.

*Numeric* has one extra optional arguments:

Numeric.digits

Same as *Float.digits*

**Date**

```python
class trytond.model.fields.Date(string[,**options ])
```

A date, represented in Python by a `datetime.date` instance.

Instance methods:

Field.sql_cast(expression[,[timezone ]])

Return the SQL expression cast as date. If timezone is set the expression is first converted to this timezone.

**DateTime**

```python
class trytond.model.fields.DateTime(string,[,format,**options ])
```

A date and time, represented in Python by a `datetime.datetime` instance. It is stored in UTC while displayed in the user timezone.

DateTime.format

A string format as used by strftime. This format will be used to display the time part of the field. The default value is `%H:%M:%S`. The value can be replaced by a *PYSON* statement.

**Timestamp**

```python
class trytond.model.fields.Timestamp(string[,**options ])
```

A timestamp, represented in Python by a `datetime.datetime` instance.

**Time**

```python
class trytond.model.fields.Time(string[,[format,**options ]])
```

A time, represented in Python by a `datetime.time` instance.

Time.format

Same as *DateTime.format*
**TimeDelta**

```python
class trytond.model.fields.TimeDelta(string[, converter[, **options]]])
```

An interval, represented in Python by a `datetime.timedelta` instance.

**TimeDelta.converter**

The name of the context key containing the time converter. A time converter is a dictionary with the keys: \(s\) (second), \(m\) (minute), \(h\) (hour), \(d\) (day), \(w\) (week), \(M\) (month), \(Y\) (year) and the value in second.

**Binary**

```python
class trytond.model.fields.Binary(string[, **options])
```

A binary field. It will be represented in Python by a `bytes` instance.

**Warning:** If the context contains a key composed of the model name and field name separated by a dot and its value is the string `size` then the read value is the size instead of the content.

**Binary** has three extra optional arguments:

**Binary.filename**

Name of the field that holds the data’s filename. Default value is an empty string, which means the data has no filename (in this case, the filename is hidden, and the “Open” button is hidden when the widget is set to “image”).

**Binary.file_id**

Name of the field that holds the `FileStore` identifier. Default value is `None` which means the data is stored in the database. The field must be on the same table and accept `char` values.

**Warning:** Switching from database to file-store is supported transparently. But switching from file-store to database is not supported without manually upload to the database all the files.

**Binary.store_prefix**

The prefix to use with the `FileStore`. Default value is `None` which means the database name is used.

**Selection**

```python
class trytond.model.fields.Selection(selection, string[, sort[, selection_change_with[, translate[, help_selection[, **options]]]]]])
```

A string field with limited values to choose from.

**Selection** has one extra required argument:

**Selection.selection**

A list of 2-tuples that looks like this:

```python
[['M', 'Male'], ['F', 'Female']]
```
The first element in each tuple is the actual value stored. The second element is the human-readable name. It can also be the name of a class or instance method on the model, that will return an appropriate list. The signature of the method is:

```python
selection()
```

**Note:** The method is automatically added to `trytond.model.Model._rpc` if not manually set.

**Selection** has two extra optional arguments:

**Selection.sort**

If true, the choices will be sorted by human-readable value. Default value is `True`.

**Note:** If it is `False`, search results ordered by the field will use the index of the selection instead of the human-readable name.

**Selection.selection_change_with**

A set of field names. If this attribute is set, the client will call the `selection` method of the model when the user changes one of the fields defined in the list and will give the values of each fields in the list. The `selection` method should be an instance method. The set of field names could be filled by using the decorator `depends()`.

**Selection.translate_selection**

If true, the human-readable values will be translated. Default value is `True`.

**Selection.help_selection**

A dictionary mapping the selection value with its help string.

Instance methods:

**Selection.translated([name])**

Returns a descriptor for the translated value of the field. The descriptor must be used on the same class as the field. It will use the language defined in the context of the instance accessed.

**MultiSelection**

```python
class trytond.model.fields.MultiSelection(selection, string, sort, translate, help_selection, **options)
```

A list field with limited values to choose from.

**MultiSelection** has one extra required argument:

**MultiSelection.selection**

Same as `Selection.selection`

**MultiSelection** has two extra optional arguments:

**MultiSelection.sort**

Same as `Selection.sort`

**MultiSelection.translate_selection**

Same as `Selection.translate_selection`
MultiSelection.\texttt{help\_selection}
   Same as \texttt{Selection.help\_selection}

Instance methods:

\texttt{MultiSelection.\texttt{translated}(\texttt{name})}
   Same as \texttt{Selection.translated()} but returns a list of translated values.

\textbf{Reference}

\texttt{class trytond.model.fields.Reference(string, selection[, sort[, selection_change_with[, translate[, help_selection[, search_order[, search_context[, \*\*options]]]]]]]])}

A field that refers to a record of a model. It will be represented in Python by a \texttt{str} instance like this:

\begin{verbatim}
'\textless model name\textgreater ,\textless record id\textgreater '
\end{verbatim}

But a tuple can be used to search or set value.

\textit{Reference} has three extra optional arguments:

Reference.\texttt{selection}   
   Same as \texttt{Selection.selection} but only for model name.

Reference.\texttt{sort}        
   Same as \texttt{Selection.sort}.

Reference.\texttt{selection\_change\_with}   
   Same as \texttt{Selection.selection\_change\_with}.

Reference.\texttt{translate\_selection}   
   Same as \texttt{Selection.translate\_selection}.

Reference.\texttt{help\_selection}   
   Same as \texttt{Selection.help\_selection}.

Reference.\texttt{datetime\_field}   
   Same as \texttt{Many2One.datetime\_field}.

Reference.\texttt{search\_order}   
   Same as \texttt{Many2One.search\_order}.

Reference.\texttt{search\_context}   
   Same as \texttt{Many2One.search\_context}.

Instance methods:

Reference.\texttt{translated}(\texttt{name})   
   Same as \texttt{translated()} but for the translated name of the target model.
Many2One

class trytond.model.fields.Many2One(model_name, string[, left[, right[, path[, ondelete[, datetime_field[, target_search[, search_order[, search_context[, **options]]]]]]]])]

A many-to-one relation field.

Many2One has one extra required argument:

Many2One.model_name
The name of the target model.

Many2One has some extra optional arguments:

Many2One.left
The name of the field that stores the left value for the Modified Preorder Tree Traversal. It only works if the model_name is the same then the model.

**Warning:** The MPTT Tree will be rebuild on database update if one record is found having left or right field value equals to the default or NULL.

Many2One.right
The name of the field that stores the right value. See left.

Many2One.path
The name of the Char field that stores the path. It only works if the model_name is the same as the model.

**Note:** The path is used to optimize searches using the child_of or parent_of operators.

**Warning:** The paths in the tree will be rebuilt during the database update if any of the records are found to have a path field equal to the default, or NULL.

Many2One.ondelete
Define the behavior of the record when the target record is deleted. Allowed values are:

- CASCADE: it will try to delete the record.
- RESTRICT: it will prevent the deletion of the target record.
- SET NULL: it will empty the relation field.

SET NULL is the default setting.

**Note:** SET NULL will be override into RESTRICT if required is true.

Many2One.datetime_field
If set, the target record will be read at the date defined by the datetime field name of the record. It is usually used in combination with trytond.model.ModelSQL._history to request a value for a given date and time on a historicized model.

Many2One.target_search
Define the kind of SQL query to use when searching on related target. Allowed values are:
• subquery: it will use a subquery based on the ids.
• join: it will add a join on the main query.

join is the default value.

Note: join could improve the performance if the target has a huge amount of records.

**Many2One**. `search_order`

A **PYSON** expression defining the default order used to display search results in the clients.

**Many2One**. `search_context`

A dictionary defining the default context used when searching from the client.

Beware that `search_context` will override the values from the client `context`.

**One2Many**

class `trytond.model.fields.One2Many(model_name, field, string[, add_remove[, order[, datetime_field[, size[, search_order[, search_context[, **options]]]]]]]])`

A one-to-many relation field. It requires to have the opposite **Many2One** field or a **Reference** field defined on the target model.

This field accepts as written value a list of tuples like this:

• ("create", [[<field name>: value, ...], ...]): it will create new target records and link them to this one.
• ("write", [[, ids, ...], {<field name>: value, ...}, ...]): it will write values to target ids.
• ("delete", [, ids, ...]): it will delete the target ids.
• ("add", [, ids, ...]): it will link the target ids to this record.
• ("remove", [, ids, ...]): it will unlink the target ids from this record.
• ("copy", ids[, {<field name>: value, ...}]): it will copy the target ids to this record. Optional field names and values may be added to override some of the fields of the copied records.

Note: **PYSON** statement or **Field.depends** of target records can access value of the parent record fields by prepending _parent_ to the opposite field name and followed by the dotted notation.

**One2Many** has some extra required arguments:

**One2Many**. `model_name`

The name of the target model.

**One2Many**. `field`

The name of the field that handles the opposite **Many2One** or **Reference**.

**One2Many** has some extra optional arguments:

**One2Many**. `add_remove`

A **domain** to select records to add. If set, the client will allow to add/remove existing records instead of only create/delete.

**One2Many**. `filter`

A **domain** that is not a constraint but only a filter on the records.
**Warning:** Only a static domain is allowed, it cannot contain any PYSON statements.

One2Many.

*order*

A list of tuple defining the default order of the records like for `tryond.model.ModelSQL._order`.

One2Many.

*datetime_field*

Same as `Many2One.datetime_field`

One2Many.

*size*

An integer or a PYSON expression denoting the maximum number of records allowed in the relation.

One2Many.

*search_order*

Same as `Many2One.search_order`

One2Many.

*search_context*

Same as `Many2One.search_context`

**Instance methods:**

One2Many.

*remove*(instance, records)*

Remove the target records from the instance instead of deleting them.

Many2Many

```python
class tryond.model.fields.Many2Many(relation_name, origin, target, string[, order[, datetime_field[, size[, search_order[, search_context[, \*\*options]]]]]]
```

A many-to-many relation field. It requires to have the opposite origin `Many2One` field or a `Reference` field defined on the relation model and a `Many2One` field pointing to the target.

This field accepts as written value a list of tuples like the `One2Many`.

*Many2Many* has some extra required arguments:

Many2Many.

*relation_name*

The name of the relation model.

Many2Many.

*origin*

The name of the field that has the `Many2One` or `Reference` to the record.

Many2Many.

*target*

The name of the field that has the `Many2One` to the target record.

**Note:** A `Many2Many` field can be used on a simple `ModelView`, like in a `Wizard`. For this, *relation_name* is set to the target model and *origin* and *target* are set to `None`.

*Many2Many* has some extra optional arguments:

Many2Many.

*order*

Same as `One2Many.order`

Many2Many.

*datetime_field*

Same as `Many2One.datetime_field`

Many2Many.

*size*

An integer or a PYSON expression denoting the maximum number of records allowed in the relation.
**Many2Many**

- **add_remove**
  An alias to the domain for compatibility with the `One2Many`.

- **filter**
  Same as `Many2Many.filter`

- **search_order**
  Same as `Many2One.search_order`

- **search_context**
  Same as `Many2One.search_context`

**Instance methods:**

- **get_relation()**
  Return the relation `Model`.

- **get_target()**
  Return the target `Model`.

- **delete(instance, records):**
  Delete the target records from the instance instead of removing them.

**One2One**

class trytond.model.fields.One2One(relation_name, origin, target, string[, datetime_field[, **options]]

A one-to-one relation field.

**Warning**: It is on the relation_name `Model` that the unicity of the couple (origin, target) must be checked.

- **datetime_field**
  Same as `Many2One.datetime_field`

- **filter**
  Same as `One2Many.filter`

**Instance methods:**

- **get_relation()**
  Return the relation `Model`.

- **get_target()**
  Return the target `Model`.

**Function**

class trytond.model.fields.Function(field, getter[, setter[, searcher[, getter_with_context]]]])

A function field can emulate any other given field.

*Function* has a required argument:

**Function.getter**

The name of the classmethod or instance of the `Model` for getting values. The signature of the classmethod is:

```python
getter(instances, name)
```
where name is the name of the field, and it must return a dictionary with a value for each instance.

Or the signature of the classmethod is:

```
getter(instances, names)
```

where names is a list of name fields, and it must return a dictionary containing for each names a dictionary with a value for each instance.

The signature of the instancemethod is:

```
getter(name)
```

where name is the name of the field, and it must return the value.

**Function** has some extra optional arguments:

**Function.setter**

The name of the classmethod of the `Model` to set the value. The signature of the method is:

```
setter(instances, name, value)
```

where name is the name of the field and value the value to set.

**Warning:** The modifications made to instances will not be saved automatically.

**Function.searcher**

The name of the classmethod of the `Model` to search on the field. The signature of the method is:

```
searcher(name, clause)
```

where name is the name of the field and clause is a *domain clause*. It must return a list of *domain* clauses but the operand can be a SQL query.

**Function.getter_with_context**

A boolean telling if the getter result depends on the context. If it does not depend, the getter is called without context and the result is stored in the transaction cache when readonly. The default value is True.

Instance methods:

**Function.get(ids, model, name[, values])**

Call the `getter` classmethod where model is the `Model` instance of the field, name is the name of the field.

**Function.set(ids, model, name, value)**

Call the `setter` classmethod where model is the `Model` instance of the field, name is the name of the field, value is the value to set.

**Function.search(model, name, clause)**

Call the `searcher` classmethod where model is the `Model` instance of the field, name is the name of the field, clause is a clause of *domain*. 

5.2. API Reference
### MultiValue

```python
class trytond.model.fields.MultiValue(field)
```

A multivalue field that is like a `Function` field but with predefined `getter` and `setter` that use the `MultiValueMixin` for stored values.

**Warning:** The `get_multivalue()` and `set_multivalue()` should be preferred over the descriptors of the field.

**Warning:** The `default` method of the field must accept pattern as keyword argument.

### Dict

```python
class trytond.model.fields.Dict(schema_model, **options)
```

A dictionary field with predefined keys.

**Note:** It is possible to store the dict as JSON in the database if the backend supports by manually altering the column type to JSON on the database.

**Dict** has one extra required argument:

- **Dict.schema_model**
  The name of the `DictSchemaMixin` model that stores the definition of keys.

- **Dict.search_unaccented**
  Same as `Char.search_unaccented` but when searching on key's value.

Instance methods:

- **Dict.translated([name, type_])**
  Returns a descriptor for the translated values or keys of the field following `type_`. The descriptor must be used on the same class as the field. Default `type_` is `values`.

### 5.2.2 Wizard

A wizard is a finite state machine.

There is also a more practical introduction into wizards.

```python
class trytond.wizard.Wizard(session_id)
```

This is the base for any wizard. It contains the engine for the finite state machine. A wizard must have some `State` instance attributes that the engine will use.

Class attributes are:

- **Wizard.__name__**
  It contains the unique name to reference the wizard throughout the platform.

- **Wizard.start_state**
  It contains the name of the starting state.
Wizard.end_state
It contains the name of the ending state. If an instance method with this name exists on the wizard, it will be called on deletion of the wizard and it may return one of the client side action keywords.

Wizard.__rpc__
Same as trytond.model.Model.__rpc__.

Wizard.states
It contains a dictionary with state name as key and State as value.

trytond.wizard.model
It contains the Model class on which the wizard is executed.

trytond.wizard.record
It contains the Model instance on which the wizard is executed.

trytond.wizard.records
It contains the list of Model instances on which the wizard is executed.

Class methods are:

classmethod Wizard.__setup__()
Setup the class before adding into the trytond.pool.Pool.

classmethod Wizard.__post_setup__()
Setup the class after added into the trytond.pool.Pool.

classmethod Wizard.__register__(module_name)
Register the wizard.

classmethod Wizard.create()
Create a session for the wizard and returns a tuple containing the session id, the starting and ending state.

classmethod Wizard.delete(session_id)
Delete the session.

classmethod Wizard.execute(session_id, data, state_name)
Execute the wizard for the state. session_id is a session id. data is a dictionary with the session data to update. active_id, active_ids, active_model and action_id must be set in the context according to the records on which the wizard is run.

5.2.3 State

class trytond.wizard.State
This is the base for any wizard state.

Instance attributes are:
State.name
The name of the state.
5.2.4 StateView

class trytond.wizard.StateView(model_name, view, buttons)
A StateView is a state that will display a form in the client. The form is defined by the ModelView with the
name model_name, the XML id in view and the buttons. The default value of the view can be set with a method
on wizard having the same name as the state but starting with default_.

Instance attributes are:

StateView.model_name
The name of the ModelView.

StateView.view
The XML id of the form view.

StateView.buttons
The list of Button instances to display on the form.

Instance methods are:

StateView.get_view(wizard, state_name)
Returns the view definition like fields_view_get().

• wizard is a Wizard instance
• state_name is the name of the StateView instance

StateView.get_defaults(wizard, state_name, fields)
Return default values for the fields.

• wizard is a Wizard instance
• state_name is the name of the State
• fields is the list of field names

StateView.get_buttons(wizard, state_name)
Returns button definitions of the wizard.

• wizard is a Wizard instance
• state_name is the name of the StateView instance

5.2.5 StateTransition

class trytond.wizard.StateTransition
A StateTransition brings the wizard to the state returned by the method having the same name as the state
but starting with transition_.

5.2.6 StateAction

class trytond.wizard.StateAction(action_id)
A StateAction is a StateTransition which let the client launch an ir.action. This action definition can
be customized with a method on wizard having the same name as the state but starting with do_.

Instance attributes are:

StateAction.action_id
The XML id of the ir.action.

Instance methods are:
StateAction.get_action()  
Returns the ir.action definition.

5.2.7 StateReport

class trytond.wizard.StateReport(report_name)
   A StateReport is a StateAction which find the report action by name instead of XML id.

5.2.8 Button

class trytond.wizard.Button(string, state=[], icon=[], default=[], validate=)
   A Button is a single object containing the definition of a wizard button.

Instance attributes are:

Button.string  
The label display on the button.

Button.state  
The next state to reach if button is clicked.

Button.icon  
The name of the icon to display on the button.

Button.default  
A boolean to set it as default on the form.

Button.validate  
A boolean or None. If True, validation of the form will occur, if False it won’t. If the value is None the validation will occur only if the state of the button is not the wizard ending state.

5.2.9 PYSON

PYSON is the PYthon Statement and Object Notation.

There is also a more practical introduction into PYSON statements.

class trytond.pyson.PYSON
Base class of any PYSON statement. It is never used directly.

Instance methods:

PYSON.pyson()  
Method that returns the internal dictionary representation of the statement.

PYSON.types()  
Method that returns a set of all possible types which the statement can become when evaluated.

classmethod PYSON.eval(dct, context)
Method which returns the evaluation of the statement given in dct within the context. dct contains a dictionary which is the internal representation of a PYSON statement. context contains a dictionary with contextual values.
Encoder and Decoder

class trytond.pyson.PYSONEncoder
Encoder for PYSON statements into string representations.

Instance method:
PYSONEncoder.encode(object)
Returns a string representation of a given PYSON statement. object contains a PYSON statement.

class trytond.pyson.PYSONDecoder([context[, noeval]])
Decoder for string into the evaluated or not PYSON statement.

Instance method:
PYSONDecoder.decode(object)
Returns a PYSON statement evaluated or not of a given string. object contains a string.

Statements

The following statements can be used in PYSON.

class trytond.pyson.Eval(value[, default])
An Eval() object represents the PYSON Eval() statement for evaluations. When evaluated, it returns the value of the statement named by value, if defined in the evaluation context, otherwise the default value (empty string by default).

Note: The default value determines the type of the statement.

Note: If the value includes dots the value will be dereferenced. For example:

```
Eval('_parent_sale.number')
```

The number value of the _parent_sale key of the evaluation context will be returned.

class trytond.pyson.Not(value)
A Not object represents the PYSON Not() statement for logical negations. When evaluated, returns the boolean negation of the value of the statement named by value, if defined in the evaluation context. Returns an instance of itself.

class trytond.pyson.Bool(value)
A Bool object represents the PYSON Bool() statement for boolean evaluations. Returns the boolean representation of the value of the statement named by value.

class trytond.pyson.And(*statements)
An And object represents the PYSON And() statement for logical and operations. Returns the result of the logical conjunction of two or more values named by the statements in the statements tuple.

class trytond.pyson.Or(*statements)
An Or object represents the PYSON Or() statement for logical or operations. Returns the result of the logical disjunction of two or more values named by the statements in the statements tuple.
class trytond.pyson.Equal(statement1, statement2)

An Equal object represents the PYSON Equal() statement for equation comparisons. Returns true when a value of a statement named by statement1 and the value of a statement named by statement2 are equal, otherwise returns false.

class trytond.pyson.Greater(statement1, statement2[, equal])

A Greater object represents the PYSON Greater() statement for greater-than comparisons. Returns true when the value of the statement named by statement1 is strictly greater than the value of the statement named by statement2, otherwise returns false. Is the value of the variable named by equal is true, then returns also true when both values of statements named by statement1 and statement2 are equal. In this case Greater works as a greater-than or equal operator.

Note: None value is replaced by 0 for the comparison.

class trytond.pyson.Less(statement1, statement2[, equal])

A Less object represents the PYSON Less() statement for less-than comparisons. Returns true when the value of the statement named by statement1 is strictly less than the value of the statement named by statement2, otherwise returns false. Is the value of the variable named equal is true, then returns also true when both values of the statements named by statement1 and statement2 are equal. In this case Less works as a less-than or equal operator.

Note: None value is replaced by 0 for the comparison.

class trytond.pyson.If(condition, then_statement, else_statement)

An If object represents the PYSON If() statement for conditional flow control operations. Returns the value of the statement named by then_statement when the value of the statement named by condition evaluates true. Otherwise returns the value of the statement named by else_statement.

class trytond.pyson.Get(obj, key[, default])

A Get object represents the PYSON Get() statement for dictionary look-up operations and evaluation. Look up and returns the value of a key named by key in an object named by obj if defined. Otherwise returns the value of the variable named by default.

class trytond.pyson.In(key, obj)

An In object represents the PYSON In() statement for look-up dictionary or integer objects. Returns true when a list (or dictionary) object named by obj contains the value of the variable (or key) named by key. Otherwise returns false.

class trytond.pyson.Date([year[, month[, day[, delta_years[, delta_month[, delta_days[, start]]]]]]])

A Date object represents the PYSON Date() statement for date related conversions and basic calculations. Returns a date object which represents the values of arguments named by the variables explained below. Missing values of arguments named by year or month or day take their defaults from start or the actual date. When values of arguments named by delta_* are given, they are added to the values of the appropriate arguments in a date and time preserving manner.

Arguments:

delta_years Contains a PYSON statement of type int or long.

delta_month Contains a PYSON statement of type int or long.

delta_days Contains a PYSON statement of type int or long.

5.2. API Reference
**trytond Documentation, Release latest**

**delta_month**  Contains a PYSON statement of type int or long.

**delta_days**  Contains a PYSON statement of type int or long.

**start**  Contains a PYSON statement of type date.

```python
class trytond.pyson.DateTime([year[, month[, day[, hour[, minute[, second[, microsecond[, delta_years[, delta_months[, delta_days[, delta_hours[, delta_minutes[, delta_seconds[, delta_microseconds[, start]]]]]]]]]]]]]]]])
```

A **DateTime** object represents the PYSON `Date()` statement for date and time related conversions and calculations. Returns a date time object which represents the values of variables named by the **arguments** explained below. Missing values of arguments named by `year`, `month`, `day`, `hour`, `minute`, `second`, `microseconds` take their defaults from `start` or the actual date and time in UTC. When values of arguments named by `delta_` are given, these are added to the appropriate attributes in a date and time preserving manner.

Arguments:

- **year**  Contains a PYSON statement of type int or long.
- **month**  Contains a PYSON statement of type int or long.
- **day**  Contains a PYSON statement of type int or long.
- **hour**  Contains a PYSON statement of type int or long.
- **minute**  Contains a PYSON statement of type int or long.
- **second**  Contains a PYSON statement of type int or long.
- **microsecond**  Contains a PYSON statement of type int or long.
- **delta_years**  Contains a PYSON statement of type int or long.
- **delta_month**  Contains a PYSON statement of type int or long.
- **delta_days**  Contains a PYSON statement of type int or long.
- **delta_hours**  Contains a PYSON statement of type int or long.
- **delta_minutes**  Contains a PYSON statement of type int or long.
- **delta_seconds**  Contains a PYSON statement of type int or long.
- **delta_microseconds**  Contains a PYSON statement of type int or long.
- **start**  Contains a PYSON statement of type datetime.

**class trytond.pyson.Len(value)**

A **Len** object represents the PYSON `Len()` statement for length of a dictionary, list or string. Returns the number of items in `value`.

**class trytond.pyson.Id(module, fs_id)**

An **Id** object represents the PYSON `Id()` statement for filesystem id evaluations. When converted into the internal dictionary, it returns the database id stored in `ir.model.data`. 
5.2.10 Transaction

```python
class trytond.transaction.Transaction
```

This class represents a Tryton transaction that contains thread-local parameters of a database connection. The Transaction instances are context manager that will commit or rollback the database transaction. In the event of an exception the transaction is rolled back, otherwise it is committed.

- **Transaction.database**
  The database.
- **Transaction.readonly**
- **Transaction.connection**
  The database connection as defined by the PEP-0249.
- **Transaction.user**
  The id of the user.
- **Transaction.context**
- **Transaction.create_records**
- **Transaction.delete_records**
- **Transaction.trigger_records**
- **Transaction.timestamp**
  The monotonic timestamp when the transaction started.
- **Transaction.language**
  The language code defines in the context.
- **Transaction.counter**
  Count the number of modification made in this transaction.

```python
static Transaction.monotonic_time()
```

Return a monotonic time used to populate :attr:`Transaction.started_at`.

```python
Transaction.start(database_name, user[, readonly[, context[, close[, autocommit ]]]])
```

Start a new transaction and return a context manager. The non-readonly transaction will be committed when exiting the with statement without exception. The other cases will be rollbacked.

```python
Transaction.stop([commit])
```

Stop the transaction. If commit is True, the transaction will be committed otherwise it will be rollbacked. The context manager returned by `Transaction.start()` should be used instead of calling this method.

```python
Transaction.set_context(context, **kwargs)
```

Update the transaction context and return a context manager. The context will be restored when exiting the with statement.

```python
Transaction.reset_context()
```

Clear the transaction context and return a context manager. The context will be restored when exiting the with statement.

```python
Transaction.set_user(user[, set_context])
```

Modify the user of the transaction and return a context manager. `set_context` will put the previous user id in the context to simulate the record rules. The user will be restored when exiting the with statement.

```python
Transaction.set_current_transaction(transaction)
```

Add a specific transaction on the top of the transaction stack. A transaction is committed or rollbacked only when its last reference is popped from the stack.
Transaction\texttt{.new\_transaction}([\texttt{autocommit}, \texttt{readonly}])

Create a new transaction with the same database, user and context as the original transaction and adds it to the stack of transactions.

Transaction\texttt{.commit}()

Commit the transaction and all data managers associated.

Transaction\texttt{.rollback}()

Rollback the transaction and all data managers associated.

Transaction\texttt{.join(datamanager)}

Register in the transaction a data manager conforming to the Two-Phase Commit protocol. More information on how to implement such data manager is available at the Zope documentation.

This method returns the registered datamanager. It could be a different yet equivalent (in term of python equality) datamanager than the one passed to the method.

Transaction\texttt{.atexit(func, \*args, \*\*kwargs)}

Register a function to be executed upon normal transaction termination. The function can not use the current transaction because it will be already committed or rollbacked.

5.2.11 Exceptions

exception trytond.exceptions.TrytonException

The base class for all Tryton exceptions.

exception trytond.exceptions.UserError(message[, description[, domain]])

The base class for exceptions used to display an error message to users. The domain may be a 2-tuple containing a domain and a dictionary of field definitions used to format the domain and append to the description.

exception trytond.exceptions.UserWarning(name, message[, description])

The base class for exceptions used to display a warning message to users.

exception trytond.exceptions>LoginExceptions(name, message[, type])

The exception used to request name parameter for the login process.

exception trytond.exceptions.ConcurrencyException(message)

The exception raised on concurrent modification.

exception trytond.exceptions.RateLimitException

The exception raised when user has sent too many login requests.

exception trytond.exceptions.MissingDependenciesException(missings)

The exception raised when modules are missing.

exception trytond.model.exceptions.AccessError

The exception raised when trying to access a record without the rights.

exception trytond.model.exceptions.AccessButtonError

The exception raised when trying to execute a button without the rights.

exception trytond.model.exceptions.ImportDataError

The exception raises when importing data fails.

exception trytond.model.exceptions.ValidationError

The base class for all record validation error.

exception trytond.model.exceptions.DomainValidationError

The exception raised when the domain of a field is not valid.
exception trytond.model.exceptions.RequiredValidationError
   The exception raised when a required field is empty.

exception trytond.model.exceptions.SizeValidationError
   The exception raised when the size of a field is too big.

exception trytond.model.exceptions.DigitsValidationError
   The exception raised when the value of a field does not respect its digits.

exception trytond.model.exceptions.SelectionValidationError
   The exception raised when the value is not in the selection.

exception trytond.model.exceptions.TimeFormatValidationError
   The exception raised when the time format of a field is not respected.

exception trytond.model.exceptions.ForeignKeyError
   The exception raised when a foreign key is not respected.

exception trytond.model.exceptions.SQLConstraintError
   The exception raised when a _sql_constraints is not respected.

exception trytond.model.exceptions.RecursionError
   The exception raised by check_recursion.

5.2.12 Tools

Tools API reference.

**Miscellaneous**

trytond.tools.resolve(name)
   Resolve a dotted name to a global object.

trytond.tools.unescape_wildcard(string[, wildcards[, escape]])
   Return the string without the wild card escapes.

trytond.tools.is_full_text(value[, escape])
   Determine if the value can be used as full text search. This is the case when the value starts and ends with a % or does not contain any wild cards.

trytond.tools.sql_pairing(x, y)
   Return an SQL expression that pairs SQL integers x and y.

trytond.tools.firstline(text)
   Return first non-empty line of a text field.

trytond.tools.remove_forbidden_chars(value)
   Return a copy of the string with forbidden char from Char replaced by space.
Email

trytond.tools.email_.set_from_header(message, sender, from_)

Fill email headers to appear at best from the address.

Singleton

class trytond.tools.singleton.Singleton

A class to use as metaclass to create a singleton object.

ImmutableDict

class trytond.tools.immutabledict.ImmutableDict

A class that implements an immutable dictionary.

5.2.13 Pool

class trytond.pool.Pool([database_name])

The Pool store the instances of models, wizards and reports per database.

Static methods:

static Pool.register(klass, ..., type_, module[, depends])

Register the classes of type (model, report or wizard) for the module. If depends is set, the registration happens only if all the modules are activated.

static Pool.register_mixin(mixin, classinfo, module)

Register the mixin for the module. The mixin will be included to all subclasses of classinfo.

Class methods:

classmethod Pool.start()

Start the pool by registering all Tryton modules found.

classmethod Pool.stop(database_name)

Stop the pool by removing instances for the database.

classmethod Pool.database_list()

List all started database.

Instance methods:

Pool.get(name[, type])

Return the named instance of type from the pool.

Pool.iterobject([type])

Return an interator over instances names.

Pool.fill(module, modules)

Fill the pool with the registered class from the module and for the activated modules and return a list of classes for each type in a dictionary.

Pool.setup([classes])

Call all setup methods of the classes provided or for all the registered classes.
Pool.setup_mixin(modules)
Include all the mixin registered for the modules to the corresponding registered classes.

### 5.2.14 PoolMeta

class trytond.pool.PoolMeta
The PoolMeta is a metaclass helper to setup __name__ on class to be registered in the Pool.

### 5.2.15 PoolBase

class trytond.pool.PoolBase
The base class of registered class that will be setup.

### 5.2.16 RPC

class trytond.rpc.RPC([readonly[, instantiate[, result[, check_access[, unique[, fresh_session[, cache ]]]]]]])
RPC is an object to define the behavior of Remote Procedure Call.
Instance attributes are:
RPC.readonly
The transaction mode
RPC.instantiate
The position or the slice of the argument to be instanciated
RPC.result
The function to transform the result
RPC.check_access
Set _check_access in the context to activate the access right on model and field. Default is True.
RPC.unique
If set, it ensures the instantiated records are unique. Default is True.
RPC.fresh_session
If set, it requires a fresh session. Default is False.
RPC.cache
A RPCCache instance to compute the cache duration for the answer.

### 5.2.17 RPCCache

class trytond.rpc.RPCCache([days[, seconds]])
Instance attributes are:
RPC.duration
A timedelta instance.
Instance methods are:
RCP.headers()
   Returns a dictionary of the headers.

5.2.18 Internationalization

tryond.i18n.gettext(message_id[, language[,**variables]])
   Returns the message translated into the language. The message_id is the XML id for the ir.message that is to
   be translated, and the variables keyword arguments are used as a mapping to format the string. If language
   is not set, then the Transaction.language is used.

tryond.i18n.lazy_gettext(message_id[, language[,**variables]])
   Return a LazyString that will be translated with gettext later when actually used.

5.2.19 Sendmail

tryond.sendmail.sendmail_transactional(from_addr, to_addrs[, msg[, transaction[, datamanager[, strict ]]]])
   Send email message only if the current transaction is successfully committed. The required arguments are an RFC 822
   from-address string, a list of RFC 822 to-address strings (a bare string will be treated as a list with 1 address), and
   an email message. The caller may pass a Transaction instance to join otherwise the current one will be joined. A
   specific data manager can be specified otherwise the default SMTPDataManager will be used for sending email. The
   strict value is passed to instantiate the default SMTPDataManager.

   Warning: An SMTP failure will be only logged without raising any exception.

tryond.sendmail.sendmail(from_addr, to_addrs[, msg[, server[, strict ]]])
   Send email message like sendmail_transactional() but directly without caring about the transaction and return
   the server. The caller may pass a server instance from smtplib. It may return a new server instance if a reconnection
   was needed and if the instance comes from get_smtp_server(). If strict is True, an exception is raised if it is not
   possible to connect to the server.

tryond.sendmail.get_smtp_server([uri[, strict]])
   Return a SMTP instance from smtplib using the uri or the one defined in the email section of the configuration. If
   strict is True, an exception is raised if it is not possible to connect to the server.

class tryond.sendmail.SMTPDataManager([uri[, strict]])
   A SMTPDataManager implements a data manager which send queued email at commit. An option optional uri can
   be passed to configure the SMTP connection. If strict is True, the data manager prevents the transaction if it fails to
   send the emails.

   SMTPDataManager.put(from_addr, to_addrs, msg)
      Queue the email message to send.
5.2.20 FileStore

```python
class trytond.filestore.FileStore
```

The class is used to store and retrieve files from the directory defined in the configuration path of database section. It uses a two levels of directory composed of the 2 chars of the file hash. It is an append only storage.

```
trytond.filestore.get(id[, prefix])
```

Retrieve the content of the file referred by the id in the prefixed directory.

```
trytond.filestore.getmany(ids[, prefix])
```

Retrieve a list of contents for the sequence of ids.

```
trytond.filestore.size(id[, prefix])
```

Return the size of the file referred by the id in the prefixed directory.

```
trytond.filestore.sizemany(ids[, prefix])
```

Return a list of sizes for the sequence of ids.

```
trytond.filestore.set(data[, prefix])
```

Store the data in the prefixed directory and return the identifiers.

```
trytond.filestore.setmany(data[, prefix])
```

Store the sequence of data and return a list of identifiers.

**Note:** The class can be overridden by setting a fully qualified name of a alternative class defined in the configuration class of the database section.

5.2.21 Cache

```python
class trytond.cache.Cache(name[, size_limit[, duration[, context]]])
```

The class is used to cache values between server requests. The `name` should be unique and it's used to identify the cache. We usually use `<class_name>.<content_name>` to make it unique. The `size_limit` parameter can be used to limit the number of values cached and it has 1024 as the default value. The `duration` parameter defines how long a cached value stays valid but if it is not set the value remains valid until it is cleared. And the `context` parameter is used to indicate if the cache depends on the user context and is true by default. The cache is cleaned on Transaction starts and resets on Transaction commit or rollback.

```
trytond.cache.hit
```

Count the number of times the cache returned a cached value.

```
trytond.cache.miss
```

Count the number of times the cache did not contain the key.

```python
classmethod trytond.cache.stats()
```

Yield statistics for each instance.

**Warning:** As there is no deep copy of the values cached, they must never be mutated after being set in or retrieved from the cache.
trytond.cache.get(key[, default])
Retrieve the value of the key in the cache. If a default is specified it will be returned when the key is missing otherwise it will return None.

trytond.cache.set(key, value)
Sets the value of the key in the cache.

trytond.cache.clear()
Clears all the keys in the cache.

classmethod trytond.cache.clear_all()
Clears all cache instances.

classmethod trytond.cache.sync(transaction)
Synchronize cache instances using transaction.

trytond.cache.sync_since(value)
Return True if the last synchronization was done before value.

classmethod trytond.cache.commit(transaction)
Apply cache changes from transaction.

classmethod trytond.cache.rollback(transaction)
Remove cache changes from transaction.

classmethod trytond.cache.drop(dbname)
Drops all the caches for database dbname.

**Note:** By default Tryton uses a MemoryCache, but this behaviour can be overridden by setting a fully qualified name of an alternative class defined in the configuration class of the cache section.

---

### 5.2.22 Bus

**class** trytond.bus.Bus

The Tryton server listens on POST requests on the routes matching /<database_name>/bus and replies with JSON dictionary containing:

channel The channel on which this message has been received.

message A dictionary that is the message the client must handle. The specification of the message depends of its type.

   All messages should at least content a unique identifier in the key message_id and their type in the key of the same name.

Client sending their request on the route must be authenticated. The request must submit a JSON dictionary containing:

last_message A value identifying the last message received by the client. This value can be null.

channels A list of strings denoting the channels the client is listening to.

The :class:Bus exposes two methods that are used by the framework: publish and subscribe.
classmethod Bus.publish(channel, message)
    Send a message to a specific channel.
    Currently implemented messages are:
    • Notifications

classmethod Bus.subscribe(database, channels[, last_message])
    Subscribe a user client to some channels of messages.
    The message_id parameter defines the last message ID received by the client. It defaults to None when not provided.

The default implementation provides an helper method to construct the response:

classmethod Bus.create_response(channel, message)
    Creates a dictionary suitable as a response from a message and a timestamp.
    channel is the channel on which the message has been received.
    message is the content of the message sent to the client.

Please note that the implementation relies on the fact that the order of the messages received is consistent across different trytond instances allowing to dispatch the request to any trytond server running.

Notification

Tryton provides a shortcut to send a notification with the notify method.

trytond.bus.notify(title[, body[, priority[, user[, client]]]])
    Send a text message to a user’s client to be displayed using a notification popup. The meaning of title, body and priority is defined in Notification message.
    If user is not set, the current user is used. Otherwise user is the user ID to notify.
    If client is not set then every client of the user will receive the message. If client and user are not set, the system will send the notification to the current user client. Otherwise the notification will be sent to the client whose ID matches client.

Notification message

Notification messages are composed of four parts:

kind  The string ‘notification’.


title  A string. A one-line summary of the message.

body  A string. The short informative message for the user. It can span multiple lines but no markup is allowed.

priority  An integer between 0 (low priority) to 3 (urgent). The notification priority on the platform supporting it.
## 5.2.23 Test

**trytond.tests.test_tryton.DB_NAME**

The name of the database to use for testing. Its value is taken from the environment variable of the same name.

**trytond.tests.test_tryton.USER**

The user id used to test the transactions

**trytond.tests.test_tryton.CONTEXT**

The context used to test the transactions

**trytond.tests.test_tryton.activate_module(name)**

Activates the module name for the tested database.

In case database does not exist and the DB_CACHE environment variable is set then Tryton will restore a backup found in the directory pointed by DB_CACHE. Otherwise it will proceed to the creation of the database and the activation of the module.

### 5.2.24 ModuleTestCase

**class trytond.tests.test_tryton.ModuleTestCase**

A subclass of unittest.TestCase that tests a Tryton module. Some tests are included to ensure that the module works properly.

This class creates a temporary database with the module activated in setUpClass and drops it in the tearDownClass method.

**ModuleTestCase.module**

Name of the tested module.

### 5.2.25 Helpers

**trytond.tests.test_tryton.with_transaction(user=1, context=None)**

This function returns a decorator to run a test case inside a transaction. The transaction is rolled back and the cache cleared at the end of the test.

### doctest helpers

**trytond.tests.test_tryton.doctest_setup()**

A function that prepares the run of the doctest by creating a database and dropping it beforehand if necessary. This function should be used as the setUp parameter.

Deprecated since version The: doctest_setup function should not be used anymore to set up DocFileSuite. New modules should use activate_modules() instead.

**trytond.tests.test_tryton.doctest_teardown()**

A function that cleans up after the run of the doctest by dropping the database. It should be used as tearDown parameter when creating a DocFileSuite.

**trytond.tests.test_tryton.doctest_checker**

A specialized doctest checker to ensure the Python 2/3 compatibility

**trytond.tests.test_tryton.suite()**
A function returning a subclass of `unittest.TestSuite` that will drop the database if it does not exist prior to the run of the tests.

### 5.2.26 Tests tools

#### `trytond.tests.tools.activate_modules(modules)`

This function is used in proteus doctests to activate a list of `modules` in the scenario.

#### `trytond.tests.tools.set_user(user, config)`

This function will set the user of the `config` proteus connection to `user`.
INDICES, GLOSSARY AND TABLES

- genindex
- modindex
- search
trytond.bus, 106
trytond.cache, 105
trytond.exceptions, 100
trytond.filestore, 104
trytond.i18n, 104
trytond.model, 64
trytond.model.exceptions, 100
trytond.model.fields, 77
trytond.pool, 102
trytond.pyson, 95
trytond.rpc, 103
trytond.sendmail, 104
trytond.tests.test_tryton, 107
trytond.tests.tools, 109
trytond.tools, 101
trytond.tools.email_, 101
trytond.tools.immutabledict, 102
trytond.tools.singleton, 102
trytond.transaction, 98
trytond.wizard, 92
Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>access</strong></td>
<td>(trytond.model.Model attribute)</td>
<td>64</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>(trytond.model.Model attribute)</td>
<td>64</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>(trytond.wizard.Wizard attribute)</td>
<td>92</td>
</tr>
<tr>
<td><strong>names</strong></td>
<td>(trytond.model.Model class method)</td>
<td>65</td>
</tr>
<tr>
<td><strong>post_setup</strong></td>
<td>(trytond.model.Model class method)</td>
<td>65</td>
</tr>
<tr>
<td><strong>post_setup</strong></td>
<td>(trytond.wizard.Wizard class method)</td>
<td>93</td>
</tr>
<tr>
<td><strong>queue</strong></td>
<td>(trytond.model.Model attribute)</td>
<td>64</td>
</tr>
<tr>
<td><strong>register</strong></td>
<td>(trytond.model.Model class method)</td>
<td>65</td>
</tr>
<tr>
<td><strong>register</strong></td>
<td>(trytond.wizard.Wizard class method)</td>
<td>93</td>
</tr>
<tr>
<td><strong>rpc</strong></td>
<td>(trytond.model.Model attribute)</td>
<td>64</td>
</tr>
<tr>
<td><strong>rpc</strong></td>
<td>(trytond.wizard.Wizard attribute)</td>
<td>93</td>
</tr>
<tr>
<td><strong>setup</strong></td>
<td>(trytond.model.Model class method)</td>
<td>65</td>
</tr>
<tr>
<td><strong>setup</strong></td>
<td>(trytond.wizard.Wizard class method)</td>
<td>93</td>
</tr>
<tr>
<td><strong>table</strong></td>
<td>(trytond.model.ModelSQL class method)</td>
<td>70</td>
</tr>
<tr>
<td><strong>table_handler</strong></td>
<td>(trytond.model.ModelSQL class method)</td>
<td>70</td>
</tr>
<tr>
<td><strong>table_history</strong></td>
<td>(trytond.model.ModelSQL class method)</td>
<td>70</td>
</tr>
<tr>
<td>_buttons</td>
<td>(trytond.model.ModelView attribute)</td>
<td>65</td>
</tr>
<tr>
<td>_defaults</td>
<td>(trytond.model.Model attribute)</td>
<td>65</td>
</tr>
<tr>
<td>_fields</td>
<td>(trytond.model.Model attribute)</td>
<td>65</td>
</tr>
<tr>
<td>_history</td>
<td>(trytond.model.ModelSQL attribute)</td>
<td>69</td>
</tr>
<tr>
<td>_order</td>
<td>(trytond.model.ModelSQL attribute)</td>
<td>69</td>
</tr>
<tr>
<td>_order_name</td>
<td>(trytond.model.ModelSQL attribute)</td>
<td>69</td>
</tr>
<tr>
<td>_rec_name</td>
<td>(trytond.model.Model attribute)</td>
<td>64</td>
</tr>
<tr>
<td>_record</td>
<td>(trytond.model.Model attribute)</td>
<td>65</td>
</tr>
<tr>
<td>_sql_constraints</td>
<td>(trytond.model.ModelSQL attribute)</td>
<td>69</td>
</tr>
<tr>
<td>_table</td>
<td>(trytond.model.ModelSQL attribute)</td>
<td>69</td>
</tr>
<tr>
<td>_transition_state</td>
<td>(trytond.model.Workflow attribute)</td>
<td>72</td>
</tr>
<tr>
<td>_transitions</td>
<td>(trytond.model.Workflow attribute)</td>
<td>72</td>
</tr>
</tbody>
</table>

A

AccessButtonError, 100
AccessError, 100

B

BigInteger (class in trytond.model.fields), 81
Binary (class in trytond.model.fields), 84
Bool (class in trytond.pyson), 96
Boolean (class in trytond.model.fields), 80
browse() (trytond.model.ModelStorage class method), 68
Bus (class in trytond.bus), 106
Button (class in trytond.wizard), 95
button() (trytond.model.ModelView static method), 66
button_action() (trytond.model.ModelView static method), 66
button_change() (trytond.model.ModelView static method), 66
buttons (trytond.wizard.StateView attribute), 94

C

Cache (class in trytond.cache), 105
cache (trytond.rpc.RPC attribute), 103
Char (class in trytond.model.fields), 81
Check (class in trytond.model), 71
check_access (trytond.rpc.RPC attribute), 103
left (trytond.model.fields.Many2One attribute), 87
Len (class in trytond.pyson), 98
Less (class in trytond.pyson), 97
loading (trytond.model.fields.Field attribute), 79
lock() (trytond.model.ModelSQL class method), 71
LoginExceptions, 100

M

Many2Many (class in trytond.model.fields), 89
Many2One (class in trytond.model.fields), 87
match() (trytond.model.MatchMixin method), 74
MatchMixin (class in trytond.model), 74
miss (in module trytond.cache), 105
MissingDependenciesException, 100
Model (class in trytond.model), 64
model (in module trytond.wizard), 93
model_name (trytond.model.fields.Many2One attribute), 87
model_name (trytond.model.fields.One2Many attribute), 88
model_name (trytond.wizard.StateView attribute), 94
ModelSingleton (class in trytond.model), 72
ModelSQL (class in trytond.model), 69
ModelStorage (class in trytond.model), 67
ModelView (class in trytond.model), 65
module
trytond.bus, 106
trytond.cache, 105
trytond.exceptions, 100
trytond.filestore, 104
trytond.i18n, 104
trytond.model, 64
trytond.model.exceptions, 100
trytond.model.fields, 77
trytond.pool, 102
trytond.pyson, 95
trytond.rpc, 103
trytond.sendmail, 104
trytond.tests.test_tryton, 107
trytond.tests.tools, 109
trytond.tools, 101
trytond.tools.email, 101
trytond.tools.immutabledict, 102
trytond.tools.singleton, 102
trytond.transaction, 98
trytond.wizard, 92
module (trytond.tests.test_tryton.ModuleTestCase attribute), 108
ModuleTestCase (class in trytond.tests.test_tryton), 108
monotonic_time() (trytond.transaction.Transaction static method), 99
MultiSelection (class in trytond.model.fields), 85
MultiValue (class in trytond.model.fields), 92
multivalue_model() (trytond.model.MultiValueMixin class method), 75
multivalue_record() (trytond.model.MultiValueMixin method), 75
multivalue_records() (trytond.model.MultiValueMixin method), 75
MultiValueMixin (class in trytond.model), 75

N

name (trytond.model.DictSchemaMixin attribute), 73
name (trytond.model.fields.Field attribute), 79
new_transaction() (trytond.transaction.Transaction method), 99
Not (class in trytond.pyson), 96
notify() (in module trytond.bus), 107
Numeric (class in trytond.model.fields), 83

O

on_change (trytond.model.fields.Field attribute), 78
on_change() (trytond.model.Model method), 65
on_change_with (trytond.model.fields.Field attribute), 78
on_change_with() (trytond.model.Model method), 65
ondelete (trytond.model.fields.Many2One attribute), 87
One2Many (class in trytond.model.fields), 88
One2One (class in trytond.model.fields), 90
operators (trytond.model.Exclude attribute), 72
operators (trytond.model.Unique attribute), 71
Or (class in trytond.pyson), 96
order (trytond.model.fields.Many2Many attribute), 89
order (trytond.model.fields.One2Many attribute), 89
origin (trytond.model.fields.Many2Many attribute), 89

P

parse_view() (trytond.model.ModelView class method), 66
path (trytond.model.fields.Many2One attribute), 87
Pool (class in trytond.pool), 102
PoolBase (class in trytond.pool), 103
PoolMeta (class in trytond.pool), 103
pre_validate() (trytond.model.Model method), 65
publish() (trytond.bus.Bus class method), 106
put() (trytond.sendmail.SMTPDataManager method), 104
PYSON (class in trytond.pyson), 95
pyson() (trytond.pyson.PYSON method), 95
PYSONDecoder (class in trytond.pyson), 96
PYSONEncoder (class in trytond.pyson), 96

R

RateLimitException, 100
read() (trytond.model.ModelStorage class method), 67
module, 105
trytond.exceptions module, 100
trytond.filestore module, 104
trytond.i18n module, 104
trytond.model module, 64
trytond.model.exceptions module, 100
trytond.model.fields module, 77
trytond.pool module, 102
trytond.pyson module, 95
trytond.rpc module, 103
trytond.sendmail module, 104
trytond.tests.test_tryton module, 107
trytond.tests.tools module, 109
trytond.tools module, 101
trytond.tools.email module, 101
trytond.tools.immutabledict module, 102
trytond.tools.singleton module, 102
trytond.transaction module, 98
trytond.wizard module, 92
TrytonException, 100
types() (trytond.pyson.PYSON method), 95

unique (trytond.rpc.RPC attribute), 103
USER (in module trytond.tests.test_tryton), 108
user (trytond.transaction.Transaction attribute), 99
UserError, 100
UserWarning, 100

V
validate (trytond.wizard.Button attribute), 95
validate() (trytond.model.ModelStorage class method), 69
ValidationWarning, 100
ValueMixin (class in trytond.model), 76
view (trytond.wizard.StateView attribute), 94
view_attributes() (trytond.model.ModelView class method), 66
view_toolbar_get() (trytond.model.ModelView class method), 66

W
where (trytond.model.Exclude attribute), 72
with_transaction() (in module trytond.tests.test_tryton), 108
Wizard (class in trytond.wizard), 92
Workflow (class in trytond.model), 72
write() (trytond.model.ModelStorage class method), 67
write_date (trytond.model.ModelStorage attribute), 67
write_uid (trytond.model.ModelStorage attribute), 67

U
unescape_wildcard() (in module trytond.tools), 101
union_column() (trytond.model.UnionMixin class method), 74
union_columns() (trytond.model.UnionMixin class method), 74
union_models() (trytond.model.UnionMixin static method), 74
union_shard() (trytond.model.UnionMixin class method), 74
union_unshard() (trytond.model.UnionMixin class method), 74
UnionMixin (class in trytond.model), 74
Unique (class in trytond.model), 71