• Installation: Installation | Configuration | Setup a database | Start the server
• Tutorials: Create a module
• Models: Model definition | Field types | Domain syntax | Access rights | Triggers
• Wizards: Wizards definition
CHAPTER THREE

THE VIEW LAYER

- **Views**: View types | Extension
- **Reports**: Report definition
- **Actions**: Actions
• **Modules** *Module definition | Create a module*
5.1 Using trytond

Introduction key parts of trytond:

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 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.

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:

```ini
[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**
  - listen
  - hostname
  - root
  - num_proxies
  - cache_timeout
  - cors
  - avatar_base
  - avatar_timeout
- **database**
  - uri
  - path
  - list
  - retry
  - language
  - avatar_filestore
  - avatar_prefix
  - default_name
  - timeout
  - minconn
  - maxconn
- **request**
  - max_size
  - max_size_authenticated
- **cache**
  - transaction
  - model
  - record
  - field
  - clean_timeout
  - count_timeout
- count_clear
  • queue
    - worker
    - clean_days
    - batch_size
  • error
    - clean_days
  • table
  • ssl
    - privatekey
    - certificate
  • email
    - uri
    - from
    - retry
  • session
    - authentications
    - authentication_ip_network
    - max_age
    - timeout
    - max_attempt
    - max_attempt_ip_network
    - ip_network_4
    - ip_network_6
  • password
    - length
    - forbidden
    - reset_timeout
    - passlib
  • attachment
    - filestore
    - store_prefix
  • bus
    - allow_subscribe
    - url_host
- `long_polling_timeout`
- `cache_timeout`
- `select_timeout`

- `html`
  - `src`
  - `plugins`
  - `css`
  - `class`

- `wsgi middleware`

**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 `trytond`.

**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. For example:

```python
cors =
    http://example.com
    https://example.com
```

**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 TRYTOND_DATABASE_URI or sqlite:/// if not set.
The available databases are:
**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
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

batch_size

The default number of the instances to process in a batch.
Default: 20

e_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.

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 5322) 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:

```
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:

```
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

5.1. Using trytond
**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}
```

Note: The options can be set using environment variables with names like: TRYTOND_WSGI_<MIDDLEWARE>__<NAME>.

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:

```
$ trytond-admin -c <config file> -d <database name> -u <module name> --activate-...
```

### 5.1.4 Logging configuration

Without any configuration, *trytond* writes 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 messages from INFO level 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)
```

(continues on next page)
5.1.5 How to start the server

Web service

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

```shell
$ 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:

```
$ tryond-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.

Worker service

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

```
$ tryond-worker -c <config file> -d <database>
```

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 --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 Model.
- Fields are defined as model attributes.
- Tryton generates the table definitions
- Tryton provides an API following the active record pattern to access the records.
Example

This example defines a Party model which has a name and a code fields:

```python
from trytond.model import ModelView, ModelSQL, fields

class Party(ModelSQL, ModelView):
    __name__ = "party.party"
    name = fields.Char('Name')
    code = fields.Char('Code')
```

The class must be registered in the Pool by the register() method of the 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 give 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</td>
<td>int</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float/Decimal</td>
<td>float</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>decimal.Decimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>str</td>
<td>str</td>
</tr>
<tr>
<td>VARCHAR(length)</td>
<td>str</td>
<td>str</td>
</tr>
<tr>
<td>TEXT</td>
<td>str</td>
<td>str</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>datetime.datetime(^2)</td>
<td>datetime.datetime(^2)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>datetime.datetime without microseconds(^2)</td>
<td>datetime.datetime without microseconds(^2)</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>
<tr>
<td>JSON</td>
<td>dict</td>
<td>dict</td>
</tr>
</tbody>
</table>

\(^1\) Corresponding to the SQL 92 standard or to a PostgreSQL type.
\(^2\) datetime objects are in UTC but without timezone.

5.1. Using trytond
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:

```
import datetime
from trytond.model import ModelView, ModelSQL, fields

class Item(ModelSQL, ModelView):
    "Item"
    __name__ = 'item'
    since = fields.Date('since')

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

See also method `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 been changed by the user. The instance method has the following name:

```
on_change_<field name>
```

An instance of Model is created by using the values from the client’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 the value of 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:

```
on_change_with_<field name>
```

Just like for the classic `on_change`, an instance of Model is created by using the values from the client’s fields specified by the `on_change_with` attribute. The returned value of the method is pushed back to the client-side record as the new value of the field.
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:

\[
\text{domain} = \left[ (\text{<field name>}, \text{<operator>}, \text{<operand>}) \right]
\]

**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:

\[
\text{domain} = \left[ (\text{country.name'}, '=' , \text{Japan'}) \right]
\]

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:

\[
\text{domain} = \left[ (\text{origin.party.name'}, '=' , \text{John Doe'}, \text{sale.sale'}) \right]
\]

A field of type Dict can be searched by key also by using one dot. For example:

\[
\text{domain} = \left[ (\text{attributes.color'}, '=' , \text{yellow'}) \right]
\]

**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:

\[
\text{domain} = \left[ \right]
\]

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:

\[
\text{domain} = [
\left( \text{field name1'}, \text{operator1'}, \text{operand1'} \right), \\
\left( \text{field name2'}, \text{operator2'}, \text{operand2'} \right), \\
\left( \text{field name3'}, \text{operator3'}, \text{operand3'} \right), \\]
\]

5.1. Using trytond
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'),
],
]
```

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'),
    ],
]
```

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:

```python
(<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>. 

5.1. Using trytond
<=

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 trytond.model.fields.One2Many / trytond.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 trytond.model.fields.One2Many / trytond.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(arg1[, arg2[, ...]])
```

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:

```
'states' 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):
    "My Model"
    __name__ = 'my_model'

    def process(self):
        pool = Pool()
        Warning = pool.get('res.user.warning')
        warning_name = Warning.format('mywarning', [self])
```

(continues on next page)
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 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:

<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.

Wizard

URL

5.1.16 Views

The views are used to display records of an 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:

```
<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-->
</record>
```
There are different types of views:

- **Form**
  - form
  - label
  - field
  - image
  - separator
  - newline
  - button
  - link
  - notebook
  - page
  - group
  - hpaned, vpaned
  - child

- **Tree**
  - tree
  - field
  - prefix, suffix
  - button

- **List-Form**

- **Graph**
  - graph
  - x, y
  - field

- **Board**
  - board
  - image
  - separator
  - label
  - newline
  - notebook
Some attributes are 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) remove the constraint on the number of columns. The default value is 4.

- **states**
  A string of **PYSON statement** that is evaluated with the values of the current record.
  It must return a dictionary where keys can be:

  - **invisible**
    If true, the widget is hidden.

  - **required**
    If true, the field is required.

  - **readonly**
    If true, the field is 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 is 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

A form view is used to display one record.

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.

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"
```
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`.

### form

Each form view must start with this tag with those attributes:

**on_write**

The name of a method on the Model of the view that is 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:

```python
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`.

**cursor**

The name of the field that must have the cursor by default.

**col.**

**label**

Display static string with those attributes:

**string**

The string that is displayed in the label.

**name**

The name of the field whose description is used for string. Except if `string` is set, it uses this value and the value of the field if `string` is empty.
id, yexpand, yfill, yalign, xexpand, xfill, xalign, colspan, states, help.

It requires that either id or name is defined.

**field**

Display a field of the object with the value of the current record with those attributes:

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

- **string**  
The string that is displayed for the widget.

- **widget**  
The widget that must be used instead of the default one.

- **help**  
The string that is 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**  
It is a comma separated list, that specifies the order of the view used to display the relation. (Example: tree, form) Only for One2Many fields.

- **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 is not displayed, but it fills 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 True.

- **spell**  
Only for Text widgets, a PYSON statement that is evaluated to the language code for which spell checking must be done.

yexpand, yfill, xexpand, xfill, colspan, help, pre_validate, completion, factor, symbol, help_field.
image

Display an image with those attributes:

**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.

**size**

The size of the image in pixels. The default value is `48`.

`yexpand`, `yfill`, `colspan`, `states`, `help`.

separator

Display a horizontal separator with those attributes:

**string**

The string that is displayed above the separator.

**name**

The name of the field from which the description is used for string.

`id`, `yexpand`, `yfill`, `colspan`, `states`, `help`.

It requires that either `id` or `name` is defined.

newline

Force to use a new row.

button

Display a button with those attributes:

**name**

The name of the function that is called on click. 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
The name of an icon to display in the button.

confirm
A text that is displayed in a confirmation pop-up when the button is clicked.

keyword
Specify where the button is displayed in the client toolbar. The valid values are the keywords starting with form_
from Actions without the form_ part.

colspan, states, help.

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

name
The XML id of ir.action.act_window.

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.

colspan, states.
notebook

Display a notebook which can contain page tags with the attributes:

colspan, states.

page

Define a tab inside a notebook with the attributes:

string
   The string that is displayed in the tab.

angle
   The angle in degrees between the baseline of the label and the horizontal, measured counterclockwise.

col, id, states.

It requires that either id or name is defined.

group

Group widgets inside a sub-form with the attributes:

string
   If set a frame is drawn around the field with a label containing the string. Otherwise, the frame is invisible.

rowspan
   The number of rows the group spans in the table.

expandable
   If this attribute is present the content of the group is expandable by the user to reveal its content. A value of 1 means that the group starts expanded, a value of 0 means that the group starts unexpanded. There is no default value.

homogeneous
   If True all the tables cells are the same size.

col, id, yexpand, yfill, yalign, xexpand, xfill, xalign, colspan, states.

It 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, colspan (the default value is 4).
child

Define the two children of a hpaned or vpaned.

Tree

A 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 has the drag and drop activated if the field_childs and the parent field are defined in the ir.ui.view record.

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

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>
```

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

Each tree view must start with this tag with those attributes:

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 is 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 is evaluated as string muted, success, warning or danger with the context of the record to provide a visual context to the row.

on_write.

5.1. Using trytond
field

name
The name of the field.

readonly
A 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 is 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 is 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 is added on the bottom of list with the sum of all the fields in the column.

width
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 attribute set. Resize doesn’t work if this option is enabled.

pre_validate, completion, factor, symbol, help_field.

prefix, suffix

A field could contain one or many prefix or suffix that is displayed in the same column with the attributes:

string
The text that is displayed.

name
The name of the field whose value is 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 and 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 form-button.

List-Form

A List-forms view displays records as a list of editable forms. It uses the same schema as the form views.

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

Graph

A graph view is used to display records in graph.

Example:

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

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.

graph

Each graph view must start with this tag with those attributes:

- **type**
  - The type of graph: vbar, hbar, line, pie.
- **background**
  - An hexadecimal 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 record to use.

string
The string to use as label for the field.

key
Used to distinguish fields with the same name but with different domain.

domain
A PySON statement which is evaluated with the record value as context. If the result is true the field value is added to the graph.

fill
Define if the graph is filled.

empty
Define if the line graph must put a point for missing dates.

color
The color of the field.

interpolation
Define 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

Board

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: ..

highlight:: xml

```xml
<field name="domain"
        pyson="1"
        eval="[('field', '=', Eval('actions', {}).get('module.action_id', {}).get('active_id', None))]"/>
```

(continues on next page)
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**

Each board view must start with this tag with the attribute:

`col`.

**image**

Same as `form-image`.

**separator**

Same as `form-separator`.

**label**

Same as `form-label`.

**newline**

Same as `form-newline`.

**notebook**

Same as `form-notebook`.

**page**

Same as `form-page`.

**group**

Same as `form-group`.

5.1. Using trytond
hpaned, vpaned

Same as form-paned.

calendar

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

Example:

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

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

calendar

Each calendar view must start with this tag with those attributes:

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 mode that is 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 lightblue.

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.

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` is not set or evaluated to `True`, the inheritance will be proceeded.

Example:

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

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.

5.1.18 Wizard

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

The basics:

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

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'),
            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 = pool.get('ir.translation')
        self.result.file = Translation.translation_export(
            self.start.language.code, self.start.module.name)
        return 'result'

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

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.

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.

model
   The __name__ of the Model the report is based. Report that is not for a specific model, needs to leave this empty.

report
   The path to the template file starting with the module directory.

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

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 Template Language](#).

**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 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 record 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 > Models > Models. Relation fields can be accessed to any depth.

**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 record 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. 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:

```
<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:

```
<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:

```
<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 record is set into context. Now the invoice report will be able to access the employee record.

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

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

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 Model. This include the context, the user and the arguments. The first argument of the method must be an instance or a list of instances of Model. This other arguments must be JSON-ifiable.

A task is pushed into the queue by calling the desired method on the __queue__. This stores in the queue all the current parameters of the task and it will be execute by a worker or at the end of the transaction if no worker is configured. The following 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.

queue_batch
An integer to divide the instances by batch of this size. If the value is true then the size is the value defined by the configuration queue of batch_size. Default is None which means no division.

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:

```python
from trytond.model import Model
from trytond.pool import PoolMeta

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

    @classmethod
    def __setup__(cls):
```

(continues on next page)
```
    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.wsgi.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.wsgi 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**

Take the first parameter as database name and replace it by the corresponding instance of the Pool.

**with_transaction([readonly])**

Start 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.

**user_application(name[, json])**

Set the 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 `notify()` function. Sending the notifications is done in a transactional way and will occur at the 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()
        if compute_stock(self.product) < 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`. 

5.1. Using trytond
Module Structure

A module is a directory in `tryond/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 keys:

- `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:

```
[tryton]
version=0.0.1
depends:
  ir
  res
  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 are inserted into the database on activation.

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**
  Prevent the framework to update the records,

- **depends**
  Import data only if all modules in the comma separated module list value are activated,

- **grouped**
  Create records at the end with a grouped call.

- **language**
  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.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 is loaded into the database on each 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**
  - The field type of ir.translation.

- **name**
  - The field name of ir.translation.

- **module**
  - The field module ir.translation.

- **xml_id**
  - The XML id that is stored in ir.model.data as fs_id. It 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 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"
```

(continues on next page)
module = 'my_module'

@with_transaction()

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

del ModuleTestCase

Note: The ModuleTestCase must be deleted to not be discovered by unittest as it fails to run without module declaration.

Running trytond's tests

You can run a specific test file using unittest command line like:

$ python -m unittest trytond.tests.test_tools

To run all trytond's tests using discover of unittest with:

$ python -m unittest discover -s trytond.tests

To run all modules tests:

$ python -m unittest discover -s trytond.modules

Running your module's tests

You just need to replace the directory path with the one of your module:

$ python -m unittest discover -s trytond.modules.my_module.tests

Extending trytond's tests

Python modules extending trytond core can define additional classes to register in tests module. Those modules must create an entry point trytond.tests which defines a register function to be called with the module name.

Testing options

Tryton runs tests against the configured database backend. You can specify the name of the database to use via the environment variable DB_NAME. Otherwise it generates a random name.

A configuration file can be used by setting its path to the environment variable TRYTOND_CONFIG.

The tests recreate frequently the database. You can accelerate the creation by setting a cache directory in DB_CACHE environment which will be used to dump and restore initial databases backups.
5.2 API Reference

Reference of the trytond API:

5.2.1 Models

- Model
- ModelView
- ModelStorage
- ModelSQL
- Workflow
- ModelSingleton
- DictSchemaMixin
- MatchMixin
- UnionMixin
- SymbolMixin
- DigitsMixin
- sequence_ordered
- MultiValueMixin
- ValueMixin
- DeactivableMixin
- tree
- avatar_mixin

Model

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

The base class that every kind of model inherits.

Class attributes are:

    Model.__name__
    The a unique name to reference the model throughout the platform.

    Model.__access__
    A set that contains the names of relation field for which the access rights are also checked for this model.

    Model.__rpc__
    A dictionary with method name as key and an instance of RPC as value.

    Model._rec_name
    The name of the field used as name of records. The default value is name.
Model.id
The definition of the field id of records.

Model.__queue__
Return a queue caller for the model. The called method will be pushed into the queue.

Model._fields
A dictionary with the field name as key and its Field instance as value.

Model._record
The record class to store internally the values of the instances.

Model._defaults
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 Pool. See trytond.pool.PoolBase.__setup__().

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

classmethod Model.__register__(module_name)
Register the model in ir.model and ir.model.field. See trytond.pool.PoolBase.__register__().

classmethod Model.default_get(fields_names[, with_rec_name])
Return 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 _rec_name field.

classmethod Model.fields_get([fields_names[, level]])
Return the definition of each field on the model.

level defines the number of relations to include in the relation field definition.

classmethod Model.__names__([field])
Return 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)
Return the list of changes by calling on_change method of each field.

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

Model.on_change_notify(fieldnames)
Returns a list of type and message couple to display on the client side. Available types are info, warning and error.

Note: To be called by the client, this method must be decorated by depends() with the fields needed.

Model.pre_validate()
Validate the instance before being stored. This method is called by the client to validate the instance.
ModelView

class trytond.model.ModelView

Add the requirements to display the record in a view.

Class attributes:

ModelView._buttons
    A dictionary with button name as key and the states dictionary for the button. The 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 states depend. This is used as default attributes of the buttons for the views that show them.

Static methods:

static ModelView.button()
    Decorate button’s method to check group access and rule.

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

    Note: 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:

```py
{
    '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.

eexports
   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 this method is called.

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

Add storage capability of record.

Class attributes are:

ModelStorage.create_uid
   The definition of the Many2One field that points to the user who created the record.

ModelStorage.create_date
   The definition of the Timestamp field that stores the creation time of the record.

ModelStorage.write_uid
   The definition of the Many2One field that points to the last user who modified the record.

ModelStorage.write_date
   The definition of the Timestamp field that stored the last modification time of the record.

ModelStorage.rec_name
   The name of the Field used as record name.

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 calls actions defined in ir.trigger with on_create 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 virtual fields _write and _delete can be used the read the writeable and deleteable state of the records. Regarding the _timestamp virtual fields it contains a timestamp that is used in the context to make a soft lock preventing update collisions.

    Note: The order of the returned list is not guaranteed.

classmethod ModelStorage.index_get_field(name)
    Return the index to order of the calls to field get.

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 with on_write set and condition was false before write() and true after.

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

classmethod ModelStorage.delete(records)
    Delete records.

classmethod ModelStorage.trigger_delete(records)
    Trigger delete actions. It will call actions defined in ir.trigger with on_delete 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 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:
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 Many2One, it is also possible to use the dotted notation to sort on a specific field from the target record. Or for a 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.

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

classmethod ModelStorage.search_rec_name(name, clause)
searcher for the 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 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.
class method ModelStorage.validate(records)

Validate the integrity of records after creation and modification.

This method must be overridden to add validation and must raise a `ValidationError` if validation fails.

class method ModelStorage.validate_fields(records, field_names)

Validate the integrity of records after modification of the fields. This method must be overridden to add validation for the field names set and must raise an exception if validation fails.

Dual methods:

class method 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 Function field `rec_name`.

**ModelSQL**

class trytond.model.ModelSQL

Implement `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 `__name__` is used with dots converted to underscores.

ModelSQL._order

The default `order` parameter of `search()` method.

ModelSQL._order_name

The name of the field on which the records must be sorted when sorting on a field referring to the model.

If not set, `_rec_name` is used.

ModelSQL._history

If true, all changes on records are stored in an history table.

ModelSQL._sql_constraints

A list of SQL constraints that are added on the table:

```
[ ([<constraint name>, <constraint>, <xml id>]), ... ]
```

constraint name

The name of the SQL constraint in the database.
constraint
An instance of Constraint

xml id
The message id for 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 raises 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 still generates 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 still generates 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:
Dual methods:

```python
classmethod ModelSQL.lock(records)
```

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

**Constraint**

```python
class trytond.model.Constraint(table)
```

Represent a SQL constraint for the table.

Instance attributes:

- `Constraint.table`
  The SQL Table on which the constraint is defined.

**Check**

```python
class trytond.model.Check(table, expression)
```

Represent a check `Constraint` which enforce the validity of the `expression`.

Instance attributes:

- `Check.expression`
  The SQL expression to check.

**Unique**

```python
class trytond.model.Unique(table, *columns)
```

Represent a unique `Constraint` which enforce the uniqueness of the group of columns.

Instance attributes:

- `Unique.columns`
  The tuple of SQL Column instances.

- `Unique.operators`
  The tuple of Equal operators.
**Exclude**

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

Represent 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**

```python
class trytond.model.Workflow
```

A `mixin` to handle transition check.

Class attribute:

- `Workflow._transition_state`
  The name of the field that will be used to check state transition. The default value is `state`.

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

Static methods:

```python
static Workflow.transition(state)
```

Decorate method to filter records for which the transition is valid and finally to update the state of the filtered record.

**ModelSingleton**

```python
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:

```python
classmethod ModelSingleton.get_singleton()
```

Return the instance of the unique record if there is one.
DictSchemaMixin

class trytond.model.DictSchemaMixin

A mixin for the schema of Dict field.

Class attributes are:

DictSchemaMixin.name
   A Char field for the name of the key.

DictSchemaMixin.string
   A Char field for the string of the key.

DictSchemaMixin.help
   The Char field used as the help text for the key.

DictSchemaMixin.type_
   The Selection field for the type of the key.
   The available types are:
   • boolean
   • integer
   • char
   • float
   • numeric
   • date
   • datetime
   • selection

DictSchemaMixin.digits
   The 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 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 Function field to return the JSON version of the selection.

Static methods:

static DictSchemaMixin.default_digits()
   Return the default value for digits.

Class methods:
classmethod DictSchemaMixin.get_keys(records)
    Return the definition of the keys for the records.

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 fields.Float.digits and fields.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]]])

Return a mixin class which defines the order of a ModelSQL with an 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 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)

getter 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 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 is considered as soft deletion.

**tree**

```
trytond.model.tree([\_parent[, \_name[, \_separator]]])
```

Return 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 get\.__rec_name__() constructs the name by concatenating each parent names using it as separator and 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().

5.2. API Reference
trytond, Release latest

**avatar mixins**

```python
tryond.model.avatar_mixin([size[, default ]])
```

Return 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.

```python
class tryond.model.AvatarMixin
```

AvatarMixin.**avatars**

The `One2Many` field used to store the `ir.avatar` records.

AvatarMixin.**avatar**

The `Binary` field that contains the avatar.

AvatarMixin.**avatar_url**

The `Char` field that contains the URL for the avatar.

AvatarMixin.**has_avatar**

Indicate whether the record has an avatar.

```python
classmethod AvatarMixin.generate_avatar(records, field)
```

Generate a default avatar for each record using the field.

### 5.2.2 Fields

- **Field**
- **Default value**
- **Searching**
- **Ordering**
- **Depends**
- **Field types**
  - `Boolean`
  - `Integer`
  - `BigInteger`
  - `Char`
  - `Text`
  - `FullText`
  - `Float`
  - `Numeric`
  - `Date`
  - `DateTime`
  - `Timestamp`
- `Time`
- `TimeDelta`
- `Binary`
- `Selection`
- `MultiSelection`
- `Reference`
- `Many2One`
- `One2Many`
- `Many2Many`
- `One2One`
- `Function`
- `MultiValue`
- `Dict`

**Field**

```python
class trytond.model.fields.Field
```

Fields define the behavior of the data on model’s record.
The following attributes are available to all field types. All are optional except `string`.

**Field.string**

A string for the label of the field.

**Field.help**

A multi-line help string for the field.

**Field.required**

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

**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.

**Field.domain**

A domain constraint that is applied on the field value.

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

**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 is evaluated with the values of the record.

Field.select
If True, the content of the field is indexed.

Field.on_change
A set of field names.
If this attribute is set, the client calls 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 readonly transaction.

The set of field names is filled by using the decorator depends().

Field.on_change_with
A set of field names.
Same as 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 readonly transaction.

The set of field names is filled by using the decorator depends().

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

Field.display_depends
A computed set of field names on which the field depends when being displayed in a read only view.

Field.edition Depends
A computed set of field names on which the field depends when being displayed in a writable view.

Field.validation depends
A computed set of field names on which the field depends when being validated.
Field.context

A dictionary which updates the current context for relation field.

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

Field.loading

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

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 definition 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)

Add to model the default RPC instances needed by the field.

Field.definition(model, language)

Return a dictionary with the definition of the field.

Field.definition_translations(model, language)

Return 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:

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

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

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:

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

Where `tables` is a nested dictionary, see `tables`.

Depends

```
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

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

A boolean field.

Integer

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

An integer field.

BigInteger

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

A long integer field.
Char

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 arguments:

Char.size
    The maximum length (in characters) of the field. The size is enforced at the storage level and in the client input.
    The value can be a PYSON statement.

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.

Charautocomplete
    A set of field names.
    
    If this attribute is set, the client calls the method autocomplete_<field name> of the model when the user changes one of those field values. The method signature is:
    
    autocomplete_<field name>()

    This method must return a list of string that is used by the client to make autocompletion proposal. 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 is 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

class trytond.model.fields.Text(string[, size[, translatable[, **options]]]])
A multi line string field.

Text has some extra 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

class trytond.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

class trytond.model.fields.Float(string[, digits[, **options]])
A floating-point number field. It is represented in Python by a float instance.

Float has some extra 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 is done. The tuple can be replaced by a string containing the name of a Many2One pointing to a DigitsMixin.

Numeric

class trytond.model.fields.Numeric(string[, digits[, **options]])
A fixed-point number field.

Numeric has some extra arguments:

Numeric.digits
Same as Float.digits.
Date

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

Instance methods:

- `Date.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.DateTimeField(string[, format, **options])
A date and time field.
It is stored in UTC while displayed in the user timezone.
```

`DateTime` has some extra arguments:

- `DateTime.format`
  - A string format as used by `strftime()`.
  - This format is 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 field.
```

Time

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

`Time` has some extra arguments:

- `Time.format`
  - Same as `DateTime.format`.

TimeDelta

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

`TimeDelta` has some extra arguments:
trytond, Release latest

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

class trytond.model.fields.Binary(string, \**options)

A binary field.

**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 some extra 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

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 some extra arguments:

Selection.selection

A list of 2-tuples that looks like this:

```
[('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 returns 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.sort
- If True, the choices is sorted by human-readable value.
- Default value is True.

**Note:** If it is False, search results ordered by the field uses 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 calls the selection method of the model when the user changes on of the fields defined in the list and gives the values of each fields in the list.
- The selection method should be an instance method.
- The set of field names is 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 uses 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 tuple field with limited values to choose from. **MultiSelection** has some extra arguments:

MultiSelection.selection
- Same as `Selection.selection`. 

5.2. API Reference
trytond, Release latest

MultiSelection.sort
   Same as Selection.sort.

MultiSelection.translate_selection
   Same as Selection.translate_selection.

MultiSelection.help_selection
   Same as Selection.help_selection.

Instance methods:

MultiSelection.translated([name])
   Same as Selection.translated() but returns a list of translated values.

Reference

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

A string field that refers to a record of a model.
   ‘<model name>,<record id>’

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

Reference has some extra arguments:

Reference.selection
   Same as Selection.selection but only for model name.

Reference.sort
   Same as Selection.sort.

Reference.selection_change_with
   Same as Selection.selection_change_with.

Reference.translate_selection
   Same as Selection.translate_selection.

Reference.help_selection
   Same as Selection.help_selection.

Reference.datetime_field
   Same as Many2One.datetime_field.

Reference.search_order
   A dictionary that contains a PYSON expression defining the default order used to display search results in the clients per model name.

Reference.search_context
   Same as Many2One.search_context.

Instance methods:

Reference.translated([name])
   Same as translated() but for the translated name of the target model.
**Many2One**

```python
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 some extra arguments:

**Many2One.model_name**

The name of the target model.

**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**: tries to delete the record.
- **RESTRICT**: prevents the deletion of the target record.
- **SET NULL**: clears the relation field.

SET NULL is the default setting.

**Note:** SET NULL is 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 `_history` to request a value for a given date and time on a historicized model.
**Many2One**

The target_search

Define the kind of SQL query to use when searching on related target.

Allowed values are:

- **subquery**: uses a subquery based on the ids.
- **join**: adds 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.

The search_order

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

The search_context

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

**Note**: search_context overrides the values from the client context.

### One2Many

A one-to-many relation field.

It requires to have the opposite Many2One field or a Reference field defined on the target model.

**One2Many** accepts as written value a list of tuples like this:

- ('create', {[<field name>: value, ...], ...}): create new target records and link them to this one.
- ('write', ids, {[<field name>: value, ...], ...}): write values to target ids.
- ('delete', ids): delete the target ids.
- ('add', ids): link the target ids to this record.
- ('remove', ids): unlink the target ids from this record.
- ('copy', ids, {[<field name>: value, ...], ...}): 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 arguments:

- **model_name**: The name of the target model.
- **field**: The name of the field that handles the opposite Many2One or Reference.

**One2Many** has some extra 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 `trytond.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 trytond.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.

`Many2Many` accepts as written value a list of tuples like the `One2Many`.

`Many2Many` has some extra 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`.

5.2. API Reference
Many2Many has some extra 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.

Many2Many.filter
Same as One2Many.filter.

Many2Many.search_order
Same as Many2One.search_order.

Many2Many.search_context
Same as Many2One.search_context.

Instance methods:

Many2Many.get_relation()
Return the relation Model.

Many2Many.get_target()
Return the target Model.

Many2Many.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.

One2One has some extra arguments:

One2One.datetime_field
Same as Many2One.datetime_field.

One2One.filter
Same as One2Many.filter.

Instance methods:

One2One.get_relation()
Return the relation Model.

One2One.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 some extra arguments:

Function.getter

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

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.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 are not 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`.

**MultiValue**

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 prefered over the descriptors of the field.

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

**Dict**

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 some extra arguments:

*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_]]])`

Return 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`.

5.2.3 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 uses.

Class attributes are:

`Wizard.__name__`

The unique name to reference the wizard throughout the platform.

`Wizard.start_state`

The name of the starting state.

`Wizard.end_state`

The name of the ending state.

If an instance method with this name exists on the wizard, it is 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`

A dictionary with state name as key and `State` as value.

`trytond.wizard.model`

The `Model` class on which the wizard is executed.

`trytond.wizard.record`

The `Model` instance on which the wizard is executed.

`trytond.wizard.records`

The list of `Model` instances on which the wizard is executed.

Class methods are:

`classmethod Wizard.__setup__()`

Setup the class before adding into the `Pool`.

`classmethod Wizard.__post_setup__()`

Setup the class after added into the `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.

### State

```python
class trytond.wizard.State
    The base for any wizard state.
```

Instance attributes are:

- `State.name`
    - The name of the state.

### StateView

```python
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)`
    - Return 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)
Return button definitions of the wizard.

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

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_.

StateAction
class trytond.wizard.StateAction(action_id)
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()
Return the ir.action definition.

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

Button
class trytond.wizard.Button(string, state[, icon[, default[, validate ]]])
Define 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.
trytond, Release latest

Button\texttt{.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.

5.2.4 PYSON

PYSON is the Python Statement and Object Notation.

There is also a more \textit{practical introduction into PYSON statements}.

\texttt{class \ trytond.pyson.PYSON}

Base class of any PYSON statement. It is never used directly.

Instance methods:

\texttt{PYSON.pylon()}

Return the internal dictionary representation of the statement.

\texttt{PYSON.types()}

Return a set of all possible types which the statement can become when evaluated.

Class methods:

\texttt{classmethod PYSON.eval(dct, context)}

Return the evaluation of the statement given in \texttt{dct} within the \texttt{context}.

\texttt{dct} contains a dictionary which is the internal representation of a PYSON statement.

\texttt{context} contains a dictionary with contextual values.

\textbf{Encoder and Decoder}

\texttt{class \ trytond.pyson.PYSONEncoder}

Encoder for PYSON statements into string representations.

Instance methods:

\texttt{PYSONEncoder.encode(object)}

Return a string representation of a given PYSON statement.

\texttt{object} contains a PYSON statement.

\texttt{class \ trytond.pyson.PYSONDecoder([context[, noeval ]])}

Decoder for string into the evaluated or not PYSON statement.

Instance methods:

\texttt{PYSONDecoder.decode(object)}

Return a PYSON statement evaluated or not of a given string.

\texttt{object} contains a string.
Statements

The following classes can be used as \textit{PYSON} statement:

- \texttt{Eval}
- \texttt{Not}
- \texttt{Bool}
- \texttt{And}
- \texttt{Or}
- \texttt{Equal}
- \texttt{Greater}
- \texttt{Less}
- \texttt{If}
- \texttt{Get}
- \texttt{In}
- \texttt{Date}
- \texttt{DateTime}
- \texttt{Len}
- \texttt{Id}

\textbf{Eval}

\begin{verbatim}
class trytond.pyson.Eval(value[, default])
\end{verbatim}

Represent the PYSON statement for evaluations.

When evaluated, it returns the value of the statement named by \texttt{value}, if defined in the evaluation context, otherwise the \texttt{default} value (empty string by default).

\textbf{Note:} The default value determines the type of the statement.

\textbf{Note:} If the value includes dots the value will be dereferenced. For example:

\begin{verbatim}
Eval('_parent_sale.number')
\end{verbatim}

The \texttt{number} value of the \texttt{_parent_sale} key of the evaluation context will be returned.
Not

class trytond.pyson.Not(value)
    Represent the PYSON 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.

Bool

class trytond.pyson.Bool(value)
    Represent the PYSON statement for boolean evaluations.
    Returns the boolean representation of the value of the statement named by value.

And

class trytond.pyson.And(*statements)
    Represent the PYSON 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.

Or

class trytond.pyson.Or(*statements)
    Represent the PYSON 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.

Equal

class trytond.pyson.Equal(statement1, statement2)
    Represent the PYSON 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.

Greater

class trytond.pyson.Greater(statement1, statement2[, equal])
    Represent the PYSON 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.

**Less**

```python
class trytond.pyson.Less(statement1, statement2[, equal])
```

Represent the PYSON 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. If 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.

**If**

```python
class trytond.pyson.If(condition, then_statement, else_statement)
```

Represent the PYSON 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`.

**Get**

```python
class trytond.pyson.Get(obj, key[, default])
```

Represent the PYSON 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`.

**In**

```python
class trytond.pyson.In(key, obj)
```

Represent the PYSON 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.
Date

class trytond.pyson.Date(year, month, day, delta_years, delta_month, delta_days, start)
Represent the PYSON 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:

year
A PYSON statement of type int or long.

month
A PYSON statement of type int or long.

day
A PYSON statement of type int or long.

delta_years
A PYSON statement of type int or long.

delta_month
A PYSON statement of type int or long.

delta_days
A PYSON statement of type int or long.

start
A PYSON statement of type date.

DateTime

class trytond.pyson.DateTime(year, month, day, hour, minute, second, microsecond, delta_years, delta_month, delta_days, delta_hours, delta_minutes, delta_seconds, delta_microseconds, start)
Represent the PYSON 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
A PYSON statement of type int or long.

month
A PYSON statement of type int or long.

day
A PYSON statement of type int or long.

hour
A PYSON statement of type int or long.
minute

A PYSON statement of type int or long.

second

A PYSON statement of type int or long.

microsecond

A PYSON statement of type int or long.

delta_years

A PYSON statement of type int or long.

delta_month

A PYSON statement of type int or long.

delta_days

A PYSON statement of type int or long.

delta_hours

A PYSON statement of type int or long.

delta_minutes

A PYSON statement of type int or long.

delta_seconds

A PYSON statement of type int or long.

delta_microseconds

A PYSON statement of type int or long.

start

A PYSON statement of type datetime.

Len

class trytond.pyson.Len(value)

Represent the PYSON statement for length of a dictionary, list or string.

Returns the number of items in value.

Id

class trytond.pyson.Id(module, fs_id)

Represent the PYSON statement for filesystem id evaluations.

When converted into the internal dictionary, it returns the database id stored in `ir.model.data`.

5.2.5 Transaction

class trytond.transaction.Transaction

Represents a Tryton transaction that contains thread-local parameters of a database connection. The Transaction instances are context manager that commits or rollbacks 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.check_warnings

The set of warnings already checked.

Transaction.timestamp

Transaction.started_at

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.

static Transaction.monotonic_time()

Return a monotonic time used to populate :attr:`~Transaction.started_at`.

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.

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 start() should be used instead of calling this method.

Transaction.set_context(context, **kwargs)

Update the transaction context and return a context manager. The context is restored when exiting the with statement.

Transaction.reset_context()

Clear the transaction context and return a context manager. The context is restored when exiting the with statement.

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.

Transaction.set_current_transaction(transaction)

Add a specific transaction on the top of the transaction stack. A transaction is commited or rollbacked only when its last reference is popped from the stack.
Transaction.new_transaction([autocommit, 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.commit()

Commit the transaction and all data managers associated.

Transaction.rollback()

Rollback the transaction and all data managers associated.

Transaction.join(datamanager)

Register in the transaction a data manager conforming to the Two-Phase Commit protocol.

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.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.

5.2.6 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(missing)

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.

5.2.7 Tools

Tools API reference.

Miscellaneous

trytond.tools.file_open(name[, mode[, subdir[, encoding ]]])
    Open the named file in subdir from the root directory.

trytond.tools.find_path(name[, subdir ])
    Return the path of the named file in subdir from root directory.

trytond.tools.find_dir(name[, subdir ])
    Return the path of the named directory in subdir from root directory.

trytond.tools.resolve(name)
    Resolve a dotted name to a global object.

trytond.tools.unescapeWildcard(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 metaclass to create a singleton object.

ImmutableDict

class trytond.tools.immutabledict.ImmutableDict
    Implement an immutable dictionary.

5.2.8 Pool

class trytond.pool.Pool([database_name])
    Store the instances of Model, Wizard and Report 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 are 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 databases.

Instance methods:
Pool.get(name[, type])
    Return the named object of type from the pool.
Pool.iterobject(type)
    Return an iterator over objects of type.

Pool.fill(module, modules)
    Fill the pool with the registered classes 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.

PoolMeta
class trytond.pool.PoolMeta
    A metaclass helper to setup __name__ on class to be registered in the Pool.

PoolBase
class trytond.pool.PoolBase
    The base class of registered classes.

    Class methods:
    classmethod PoolBase.__setup__()  # Setup the class.
    classmethod PoolBase.__post_setup__()  # Post setup the class.
    classmethod PoolBase.__register__()  # Register the class.

5.2.9 RPC
class trytond.rpc.RPC([readonly[, instantiate[, result[, check_access[, unique[, fresh_session[, cache ]]]]]]])
    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.

RPCCache

class trytond.rpc.RPCCache(days[, seconds])
   Define cache duration of RPC result.

Instance attributes are:
RPC.duration
   A datetime.timedelta instance.

Instance methods are:
RPC.headers()
   Return a dictionary of the headers.

5.2.10 Internationalization

trytond.i18n.gettext(message_id[, language[, **variables]])
   Return 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.

trytond.i18n.lazy_gettext(message_id[, language[, **variables]])
   Return a LazyString that will be translated with gettext later when actually used.

5.2.11 Sendmail

trytond.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 5322 from-address string, a list of RFC 5322 to-address strings (a bare string is treated as a list with 1 address), and an email message. The caller may pass a Transaction instance to join otherwise the current one is joined. A specific data manager can be specified otherwise the default SMTPDataManager is used for sending email. The strict value is passed to instantiate the default SMTPDataManager.

   Warning: An SMTP failure is only logged without raising any exception.
trytond.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.

trytond.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 trytond.sendmail.SMTPDataManager([uri[, strict]])

Implement 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.

5.2.12 FileStore

class trytond.filestore.FileStore

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.

FileStore.get(id[, prefix])

Retrieve the content of the file referred by the id in the prefixed directory.

FileStore.getmany(ids[, prefix])

Retrieve a list of contents for the sequence of ids.

FileStore.size(id[, prefix])

Return the size of the file referred by the id in the prefixed directory.

FileStore.sizemany(ids[, prefix])

Return a list of sizes for the sequence of ids.

FileStore.set(data[, prefix])

Store the data in the prefixed directory and return the identifiers.

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.
### 5.2.13 Cache

**class** `trytond.cache.Cache(name[, size_limit[, duration[, context ]]])`

Use to cache values between server requests.

The name should be unique and it’s used to identify the cache. Usually `<class_name>.<content_name>` is used 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.

---

**Warning:** As there is no deepcopy of the values cached, they must never be mutated after being set in or retrieved from the cache.

---

**Cache.hit**

Count the number of times the cache returned a cached value.

**Cache.miss**

Count the number of times the cache did not contain the key.

**classmethod** `Cache.stats()`

Yield statistics for each instance.

**Cache.get(key[, default])**

Retrieve the value of the key in the cache.

If a `default` is specified it is returned when the key is missing otherwise it returns `None`.

**Cache.set(key, value)**

Set the value of the key in the cache.

**Cache.clear()**

Clear all the keys in the cache.

**classmethod** `Cache.clear_all()`

Clear all cache instances.

**classmethod** `Cache.sync(transaction)`

Synchronize caches between servers using `transaction` instance.

**Cache.sync_since(value)**

Return `True` if the last synchronization was done before `value`.

**classmethod** `Cache.commit(transaction)`

Apply cache changes from transaction.

**classmethod** `Cache.rollback(transaction)`

Remove cache changes from transaction.
trytond, Release latest

**classmethod**  `Cache.drop(dbname)`

Drop all caches for named database.

**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.

### 5.2.14 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 requests 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.

```
class trytond.bus.Bus
    Expose two methods that are used by the framework: publish and subscribe.
```

**classmethod**  `Bus.publish(channel, message)`

Send a message to a specific channel.

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)`

Create 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.

**Note:** 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` function.

```python
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 receives the message. If `client` and `user` are not set, the system send the notification to the current user client. Otherwise the notification is sent to the client whose id matches `client`.

**Notification message**

Notification messages are composed of four parts:

- **kind**
  - The string `notification`.

- **title**
  - A string containing a one-line summary of the message.

- **body**
  - A string containing a 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.

**5.2.15 Tests**

```python
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.

```python
trytond.tests.test_tryton.USER
```

The user id used to test the transactions

```python
trytond.tests.test_tryton.CONTEXT
```

The context used to test the transactions

```python
trytond.tests.test_tryton.activate_module(name)
```

Activate the named module for the tested database.

In case database does not exist and the `DB_CACHE` environment variable is set then Tryton restores a backup found in the directory pointed by `DB_CACHE`. Otherwise it proceeds to the creation of the database and the activation of the module.
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.

It creates a temporary database with the module activated in setUpClass and drops it in the tearDownClass method.

ModuleTestCase.module

Name of the tested module.

Helpers

trytond.tests.test_tryton.with_transaction(user=1, context=None)

Return a decorator to run a test case inside a Transaction. It is rolled back and the cache cleared at the end of the test.

doctest helpers

trytond.tests.test_tryton.doctest_setup()

Prepare 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 4.2: 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()

Clean 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 compatibility.

trytond.tests.test_tryton.load_doc_tests(name, path, loader, tests, pattern)

An helper that follows the load_tests protocol to load as DoctTest all *.rst files in directory, with the module name and the path to the module file from which the doc tests are registered.

trytond.tests.test_tryton.suite()

A function returning a subclass of unittest.TestSuite that drops the database if it does not exist prior to the run of the tests.

Tools

trytond.tests.tools.activate_modules(modules)

Activate a list of modules for scenario based on proteus doctests.

trytond.tests.tools.set_user(user, config)

Set the user of the config proteus connection to user.
5.3 Tutorials

5.3.1 Module Tutorial

A step by step tutorial to create a first module. In this tutorial we will create a simple Tryton module to manage business opportunities.

We will use Tryton and SQLite as database, so the installation should be pretty straightforward.

We will call our module opportunity. This simple module will do the following things:

**Setup**

**Create virtual environment**

This step will cover the installation of tryton from a developer perspective. We will assume that you are already fluent with venv and pip.

Let’s create a virtual environment inside your working directory:

```
$ python -m venv .venv
$ source .venv/bin/activate
```

**Install cookiecutter and mercurial**

To bootstrap the module, we provide a cookiecutter template. First we install cookiecutter and mercurial with:

```
$ python -m pip install cookiecutter mercurial
```

**Setup module**

The Tryton template can be rendered into a module with:

```
$ cookiecutter hg+https://hg.tryton.org/cookiecutter
```

**Note:** The version number must use the same two first numbers as the Tryton series wanted.
Install module

Now we can install the new module in editable mode:

```
$ python -m pip install --editable opportunity
```

Continue with *initializing the database*

Initialize the database

By default Tryton, use an SQLite database stored in the folder `db` of your home directory. This can be changed in the `database` section of the `configuration`.

Now creating a Tryton database is only a matter of executing the following commands:

```
$ mkdir ~/db
$ touch ~/db/test.sqlite
$ trytond-admin -d test --all
```

You will be prompted to set the administrator email and password.

Once the database is initialized you can run the Tryton server:

```
$ trytond
```

Connecting to the database using a Tryton client you will be greeted by the module configuration wizard.

We will continue with *the anatomy of the module*.

Anatomy of a module

A Tryton module is a python module thus it is a directory that contains a `__init__.py` file.

A Tryton module must also contain a `tryton.cfg` file which is used to define the dependencies between modules and also lists the XML files that must be loaded by Tryton.

Usually a module will define views used in the user interface, those views are described by XML files stored in the view directory.

Translations are handled with po files that sit in the locale directory, one file per language.

Let’s continue with *creating the models*.

Define model

The models are the base objects of a module to store and display data. The `ModelSQL` is the base class that implements the persistence in the SQL database. The `ModelView` is the base class that implements the view layer. And of course, a model would be useless without its fields.

Let’s start with a simple model to store the opportunities with a description, a start and end date, a link to a party and an optional comment. Our model in `opportunity.py` file currently looks like this:

```python
from trytond.model import ModelSQL, fields

class Opportunity(ModelSQL):
```

(continues on next page)
"Opportunity"
__name__ = 'training.opportunity'
_rec_name = 'description'

description = fields.Char("Description", required=True)
start_date = fields.Date("Start Date", required=True)
end_date = fields.Date("End Date")
party = fields.Many2One('party.party', "Party", required=True)
comment = fields.Text("Comment")

As you can see a Model must have a __name__ attribute. This name is used to make reference to this object. It is also used to build the name of the SQL table to store the opportunity records in the database.

The _rec_name attribute defines the field that will be used to compute the name of the record. The name of the record is its textual representation.

The party field is a relation field (Many2One) to another Model of Tryton named party.party. This model is defined by the party module.

Register the model in the Pool

Once a Tryton model is defined, you need to register it in the Pool. This is done in the __init__.py file of your module with the following code:

```python
from trytond.pool import Pool
from . import opportunity

def register():
    Pool.register(
        opportunity.Opportunity,
        module='opportunity', type_='model')
```

Models in the pool are inspected by Tryton when activating or updating a module in order to create or update the schema of the table in the database.

Activate the opportunity module

Now that we have a basic module, we will use it to create the related table into the database created.

First we must edit the tryton.cfg file to specify that this module depends on the party and ir module. We need to do this because the Opportunity model contains the party field which refers to the Party model. And we always need the ir module which is always included in Tryton server.

Here is the content of our tryton.cfg file:

```plaintext
[tryton]
version=x.y.0
depends:
   ir
   party
```

As we defined a new dependency, we must refresh the installation with:
Now we can activate the opportunity module and its dependencies:

```bash
$ trytond-admin -d test -u opportunity --activate-dependencies
```

This step has created the tables into your database. You can check it with the `sqlite3` command line:

```bash
$ sqlite3 ~/db/test.sqlite

CREATE TABLE "training_opportunity" (  
id INTEGER PRIMARY KEY AUTOINCREMENT,  
"comment" TEXT,  
"create_uid" INTEGER,  
"create_date" TIMESTAMP,  
"description" VARCHAR,  
"end_date" DATE,  
"start_date" DATE,  
"write_date" TIMESTAMP,  
"party" INTEGER,  
"write_uid" INTEGER);
```

The next step will be displaying record.

**Display records**

Having records in the database is nice but we want the user to manage this records through the user interface.

In order to denote that a model can be displayed in the interface, you have to inherit from `ModelView`:

```python
from trytond.model import ModelSQL, ModelView
...

class Opportunity(ModelSQL, ModelView):
    ...
```

When you inherit from `ModelView`, your model gains the methods required to display the data on Tryton clients. Those methods allow to retrieve the fields and the definition of the views used by a model, to apply attributes on view elements and they also provide all the machinery for `on_change` and `on_change_with`.

**Tryton Views**

In Tryton data can be displayed using different kind of views. The available view types and it’s attributes are listed on the `Views` topic.

Tryton views are usual Tryton records that are persisted into the database. This design choice means that views are extendable and that you can use the traditional Tryton concepts when interacting with them.
Define views

Views are defined in XML files and they contain one XML tag for each element displayed in the view. The root tag of the view defines the view type. An example view for our opportunity module will be as follows:

Here is the content of the form view of opportunity in `view/opportunity_form.xml`:

```
<form>
    <label name="party"/>
    <field name="party"/>
    <label name="description"/>
    <field name="description"/>
    <label name="start_date"/>
    <field name="start_date"/>
    <label name="end_date"/>
    <field name="end_date"/>
    <separator name="comment" colspan="4"/>
    <field name="comment" colspan="4"/>
</form>
```

And here is the content of the list view in `view/opportunity_list.xml`:

```
<tree>
    <field name="party"/>
    <field name="description"/>
    <field name="start_date"/>
    <field name="end_date"/>
</tree>
```

The value of the `name` attribute for `field` and `label` tags is the name of the field attribute of the model. Each XML tag can contain different attributes to customize how the widgets are displayed in the views. The full reference can be found on the Views section.

Once a view is defined it must be registered on the Tryton database in order to make the server know about them. In order to do so with should register it on a XML file specifying the following information:

- **model**
  The name of the model of the view

- **type**
  Possible values are: tree, form, calendar, graph, board

- **name**
  The name of the XML file (without extension) in the view folder which contains the view definition

Here is the content of the `opportunity.xml` file:

```
<tryton>
    <data>
        <record model="ir.ui.view" id="opportunity_view_form">
            <field name="model">training.opportunity</field>
            <field name="type">form</field>
            <field name="name">opportunity_form</field>
        </record>
        <record model="ir.ui.view" id="opportunity_view_list">
            <field name="model">training.opportunity</field>
            <field name="type">list</field>
            <field name="name">opportunity_list</field>
        </record>
    </data>
</tryton>
```
Now we have to declare the XML data file in the tryton.cfg file:

```
[tryton]
...
xml:
  opportunity.xml
```

Create menu entry

In order to show our models on the user menu we need an ir.action.act_window and a menu entry.

An action window is used to relate one or more views, usually a list and a form view.

Here is the definition of the opportunities action to append into opportunity.xml:

```
<tryton>
  <data>
    ...
    <record model="ir.action.act_window" id="act_opportunity_form">
      <field name="name">Opportunities</field>
      <field name="res_model">training.opportunity</field>
    </record>
    <record model="ir.action.act_window.view" id="act_opportunity_form_view1">
      <field name="sequence" eval="10"/>
      <field name="view" ref="opportunity_view_list"/>
      <field name="act_window" ref="act_opportunity_form"/>
    </record>
    <record model="ir.action.act_window.view" id="act_opportunity_form_view2">
      <field name="sequence" eval="20"/>
      <field name="view" ref="opportunity_view_form"/>
      <field name="act_window" ref="act_opportunity_form"/>
    </record>
  </data>
</tryton>
```

A menu entry is created using the special menuitem XML tag which accepts the following values:

**id**
Required XML identifier to refer this menu_item from other records.

**sequence**
Used to define the order of the menus.

**action**
The action to execute when clicking the menu.
name
The string that will be shown on the menu. If no name is entered and an action is set, the action name will be used.

parent
The parent menu when creating a sub-menu.

Let's add a menu entry to the `opportunity.xml` file with:

```xml
<tryton>
  <data>
    ...  
    <menuitem name="Opportunities" sequence="50" id="menu_opportunity"/>
    <menuitem parent="menu_opportunity" action="act_opportunity_form" sequence="10" id="menu_opportunity_form"/>
  </data>
</tryton>
```

**Update database**

As we have defined new XML records, we need to update the database with:

```bash
$ trytond-admin -d test --all
```

And restart the server and reconnect with the client to see the new menu entries:

```bash
$ trytond
```

Let's continue with *setting default values*.

**Set default values**

Default values are useful to save time for users when entering data. On Tryton `default values` are computed on server side and they will be set by the client when creating a new record if the field is shown on the view. If the field is not shown on the view, the server will set this values when storing the new records in the database.

In order to define a default value for a field you should define a class method named `default_<field_name>` that returns the default value. For example to add today as the default date of our `Opportunity` model the following class method is added in `opportunity.py` file:

```python
from trytond.pool import Pool
...

class Opportunity(ModelSQL, ModelView):
    ...
    @classmethod
    def default_start_date(cls):
        pool = Pool()
```

(continues on next page)
Date = pool.get('ir.date')
return Date.today()

Call other model methods

In the previous example we called the today method of the ir.date model from the Pool instance. The __name__ value is used to get the class. It is very important to get the class from the pool instead of using a normal Python import, because the pool ensures that all of the extensions are applied depending on the activated modules. For example, if we have the company module also activated the correct timezone for the user company will be used for computing the today value.

Great, you have learned how to define default values, and how to call methods defined on other classes in the pool. Let’s continue with reacting on user input.

React to user input

Tryton provides a way to change the value of a field depending on other fields. This computation is done on the server and the values are sent back to the client. The value is not stored on the server until the user saves the record. This is a great way to react to user inputs.

For example, in order to set the end date of our opportunity depending on the start date, we can add the following instance method to Opportunity class in opportunity.py file:

```python
import datetime as dt
...
class Opportunity(ModelSQL, ModelView):
    ...
    @fields.depends('start_date')
    def on_change_with_end_date(self):
        if self.start_date:
            return self.start_date + dt.timedelta(days=3)
```

In this case the depends() decorator indicates the names of the fields which will trigger the computation when their values are changed. You should take care to set all the fields used to make the computation because the server will have only access to those fields. This ensures that the client reacts to each field the computation depends on.

We can also compute the values of other fields when some field change. In this case we use the on_change_<field_name> function instead of on_change_with_<field_name>. The depends() decorator indicates the fields that will be available to compute the new values. In order to set the other fields value, we must assign them to the instance and the changes will be propagated to the client.

So for example we can compute the description and the comment of our opportunity model depending on the party by adding this method to the Opportunity class in opportunity.py file:

```python
class Opportunity(ModelSQL, ModelView):
    ...
    @fields.depends('party', 'description', 'comment')
    def on_change_party(self):
        if self.party:
            if not self.description:
                self.description = self.party.rec_name
            if not self.comment:
```

(continues on next page)
Great, you have learned how to compute values depending on other fields values. Let's continue with adding computed fields.

### Add computed fields

Computed fields can also be defined to avoid storing duplicated data in the database. For example, as we have the start date and the end date of our opportunity we can always compute the duration the opportunity lasts. This is done with a `Function` field, which can be used to represent any kind of field.

Let's see how this can be done in `opportunity.py` file:

```python
class Opportunity(ModelSQL, ModelView):
    ...
    duration = fields.Function(
        fields.TimeDelta("Duration"), 'compute_duration')
    ...
    def compute_duration(self, name):
        if self.start_date and self.end_date:
            return self.end_date - self.start_date
        return None
```

The first parameter of the `Function` field is another `Field` instance which defined the type of the field to mimic and on the second parameter, the `getter`, we must specify the name of the method used to compute the value.

`Function` fields are read-only by default, but we can make them writable by defining a `setter` attribute, which is a method to call to store the value. Similarly we can also provide a method to search or order on them. All the `Function` fields possibilities are explained on `Function` fields reference.

**Warning:** If you change the start date or the end date of the opportunity, you will notice that the days value is not updated until the record is saved. That's because function fields are computed only on server side.

**Note:** We let you add the new field to the views.
Combine Function fields and on_change_with

On previous steps we learned how on_change and Function fields work. One interesting feature is to combine them to compute and update the value. So we can have a computed field that changes every time the user modifies one of the values of the form.

It’s a common pattern to use an on_change_with method as getter of a Function field, so the value is correctly computed on client side and then it reacts to the user input.

In order to achieve it the following changes must be done in opportunity.py file:

```python
class Opportunity(ModelSQL, ModelView):
    ...
    duration = fields.Function(
        fields.TimeDelta("Duration"), 'on_change_with_duration')
    ...
@fields.depends('start_date', 'end_date')
def on_change_with_duration(self, name=None):
    if self.start_date and self.end_date:
        return self.end_date - self.start_date
    return None
```

The important facts are the following:

- Add depends() decorator to react on user input
- Change the name of the method to on_change_with_<field_name>
- Add a default None value for the name argument as it won’t be supplied when the client updates the values reacting to user input.

Great, you designed a Function fields which reacts to the user input. Let’s go to the next step to add domain restrictions.

Add domain restriction to fields

One common requirement is to add restrictions to the possible value of a field. For example we can define the following restrictions:

- The value of a numeric field must be greater that zero.
- The value of another field must be greater than the value of other.
- Related record must have fields with specific values. For example allow to select only products of kind service.

This is represented using a domain clause. The domain clause syntax is explained on domain reference.

A very interesting thing of the domain, is that the client evaluates them, so:

- If we set a value that invalidate the domain of some fields, they are marked. A notification is displayed before saving.
- When searching for a related record, only the records that satisfy the domain are available. So it is not possible to select invalid records.
- When creating a new related record, the client automatically enforces only valid values. Fields that can have only one value are filled and set read only.

For example, it may be interesting to add the address of the party on our Opportunity model. In this case we are interested on selecting only the addresses related to the party.

Let’s see how to do it:
from trytond.pyson import Eval
...

class Opportunity(ModelSQL, ModelView):
    ...
    address = fields.Many2One(  
        'party.address', 'Address',
        domain=[
            ('party', '!=', Eval('party', -1)),
        ])

The domain uses the value of the party field with the Eval object. This defines a relation between party and address field.

Note: It is up to you to add the new field to the views and update the database.

Using conditional domains

Sometimes it is interesting to apply a domain only if another field is set. For example we want to ensure the start date is before the end date but both fields are optionals, so we don’t want to apply any domain if they are empty. This can be solved by using a conditional domain.

Let’s see how we can achieve it:

from trytond.pyson import If, Bool, Eval
...

class Opportunity(ModelSQL, ModelView):
    ...
    start_date = fields.Date(  
        "Start Date", required=True,  
        domain=[
            If(Bool(Eval('end_date')),  
                ('start_date', '<=', Eval('end_date')),  
                ()]
        )

    end_date = fields.Date(  
        "End Date",  
        domain=[
            If(Bool(Eval('end_date')),  
                ('end_date', '>=', Eval('start_date')),  
                ()]
        )

In this case we used the following statements:

* If which expects three values: (condition, true-statement, false-statement) In this case we use to return a domain on when the condition is True and return an empty domain on False.
* Bool used to convert the field value into boolean.

All of the domains are PSYON statements.

Great, you have learned to add constraint on the fields value. Let’s continue with adding a workflow.

5.3. Tutorials
Define workflow

Often records follow a workflow to change their state. For example the opportunity can be converted or lost. Tryton has a `Workflow` class that provides the tooling to follow a workflow based on the field defined in `_transition_state` which is by default `state`.

First we need to inherit from `Workflow` and add a `Selection` field to store the state of the record:

```python
from trytond.model import Workflow
...

class Opportunity(Workflow, ModelSQL, ModelView):
...

    state = fields.Selection(
        [('draft', 'Draft'),
         ('converted', 'Converted'),
         ('lost', 'Lost'),
          ],
       'State',
       required=True, readonly=True, sort=False)

@classmethod
def default_state(cls):
    return 'draft'
```

We must define the allowed transitions between states by filling the `_transitions` set with tuples using the `__setup__()` method:

```python
class Opportunity(Workflow, ModelSQL, ModelView):
...

    @classmethod
    def __setup__(cls):
        super().__setup__()
        cls._transitions.update({
            ('draft', 'converted'),
            ('draft', 'lost'),
        })
```

For each target state, we must define a `transition()` method. For example when the opportunity is converted we fill the `end_date` field with today:

```python
class Opportunity(Workflow, ModelSQL, ModelView):
...

    @Workflow.transition('converted')
    def convert(cls, opportunities):
        pool = Pool()
        Date = pool.get('ir.date')
        cls.write(opportunities, {
            'end_date': Date.today(),
        })
```

**Note:** We let you define the transition method for lost.

Now we need to add a button for each transition so the user can trigger them.
We must declare the button in the `_buttons` dictionary and decorate the transition method with the `button()` to be callable from the client:

```python
class Opportunity(Workflow, ModelSQL, ModelView):
    ...
    @classmethod
    def __setup__(cls):
        ...
        cls._buttons.update({
            'convert': {},
            'lost': {},
        })

    @classmethod
    @ModelView.button
    @Workflow.transition('converted')
    def convert(cls, opportunities):
        ...

    @classmethod
    @ModelView.button
    @Workflow.transition('lost')
    def lost(cls, opportunities):
        ...
```

Every button must also be recorded in `ir.model.button` to define its label (and also the access right). We must add to the `opportunity.xml` file:

```xml
<tryton>
    <data>
        ...<record model="ir.model.button" id="opportunity_convert_button">
            <field name="name">convert</field>
            <field name="string">Convert</field>
            <field name="model" search="[
                ('model', '=', 'training.opportunity')]">
        </record>
        ...
        <record model="ir.model.button" id="opportunity_lost_button">
            <field name="name">lost</field>
            <field name="string">Lost</field>
            <field name="model" search="[
                ('model', '=', 'training.opportunity')]">
        </record>
    </data>
</tryton>
```

Now we can add the `state` field and the buttons in the form view. The buttons can be grouped under a `group` tag. This is how the `view/opportunity_form.xml` must be adapted:

```xml
<form>
    ...
    <label name="state"/>
    <field name="state"/>
    <group col="2" colspan="2" id="button">
        <button name="lost" icon="tryton-cancel"/>
    </group>
</form>
```

(continues on next page)
trytond, Release latest

Note: We let you add the state field on the list view.

Update database

As we have defined new fields and XML records, we need to update the database with:

```bash
$ trytond-admin -d test --all
```

And restart the server and reconnect with the client to test the workflow:

```bash
$ trytond
```

Exercise

As exercise we let you add a transition between lost and draft which will clear the end_date.

Let's continue with adding more reaction with dynamic state.

Add dynamic state to fields

Sometimes you want to make fields read-only, invisible or required under certain conditions. This can be achieved using the states attribute of the Field. It is a dictionary with the keys readonly, invisible and required. The values are PYSON statements that are evaluated with the values of the record.

In our example we make some fields read-only when the record is not in the state opportunity, the “End Date” required for the converted and lost state and make the comment invisible if empty:

```python
class Opportunity(...):
    ...
    description = fields.Char("Description", required=True,
        states={
            'readonly': Eval('state') != 'draft',
        })
    start_date = fields.Date("Start Date", required=True,
        states={
            'readonly': Eval('state') != 'draft',
        })
    end_date = fields.Date("End Date",
        states={
            'readonly': Eval('state') != 'draft',
            'required': Eval('state').in_(['converted', 'lost'])),
```

(continues on next page)
party = fields.Many2One('party.party', 'Party', required=True,
    states={
        'readonly': Eval('state') != 'draft',
    })
address = fields.Many2One('party.address', 'Address',
    domain=[
        ('party', '=', Eval('party')),
    ],
    states={
        'readonly': Eval('state') != 'draft',
    })
comment = fields.Text('Comment',
    states={
        'readonly': Eval('state') != 'draft',
        'invisible': (Eval('state') != 'draft') & ~Eval('comment'),
    })

It is also possible to set the readonly, invisible and icon states on the `_buttons`. So we can make invisible each button for the state in which the transition is not available:

class Opportunity(ModelSQL, ModelView):
    ...
    @classmethod
    def __setup__(cls):
        ...
        cls._buttons.update({
            'convert': {
                'invisible': Eval('state') != 'draft',
                'depends': ['state'],
            },
            'lost': {
                'invisible': Eval('state') != 'draft',
                'depends': ['state'],
            },
        })

Note: The fields in `Eval` statement must be added to the depends attribute to register the field on which the states depend.
**Exercise**

As exercise we let you define the state for the button that reset to draft state.
Let's *extend the party model*.

**Extend model**

Sometimes we want to extend an existing *Model* to add *Field* or methods. This can be done using the extension mechanism of Tryton which can combine classes with the same __name__ that are registered in the *Pool*.

**Extend the Party model**

Let’s add an opportunities field on the party.party model. The model in party.py file of our module looks like this:

```python
from trytond.model import fields
from trytond.pool import PoolMeta

class Party(metaclass=PoolMeta):
    __name__ = 'party.party'
    opportunities = fields.One2Many(
        'training.opportunity', 'party', 'Opportunities')
```

This new class must be register in the *Pool*. So in __init__.py we add:

```python
from . import party
def register():
    Pool.register(..., party.Party,
                   module='opportunity', type_='model')
```

**Extend the Party view**

Now that we added a new field to the party.party *Model*, we can also add it the form view. This is done by adding a ir.ui.view record that inherit the party form view of the party module. Here is the content of the party.xml file:

```xml
<tryton>
    <data>
        <record model="ir.ui.view" id="party_view_form">
            <field name="model">party.party</field>
            <field name="inherit" ref="party.party_view_form"/>
            <field name="name">party_form</field>
        </record>
    </data>
</tryton>
```

The type is replaced by:
inherit

A reference to the XML id of the view extended prefixed by the name of the module where the view is declared.

The content of the inheriting view must contain an XPath expression to define the position from which to include the partial view XML. Here is the content of the form view in view/party_form.xml:

```xml
<data>
  <xpath expr="/form/notebook/page[@name='identifiers']" position="after">
    <page name="opportunities" col="1">
      <field name="opportunities"/>
    </page>
  </xpath>
</data>
```

And finally we must declare the new XML data in the tryton.cfg file:

```plaintext
[tryton]
...
xml:
  ...
  party.xml
```

**Update database**

As we have defined new field and XML record, we need to update the database with:

```bash
$ trytond-admin -d test --all
```

And restart the server and reconnect with the client to see the new field on the party:

```bash
$ trytond
```

Let’s use a wizard to convert the opportunity.

**Create wizard**

Sometime you want to add functionalities to a model that do not suite the use of a button. For this kind of use case the wizard is the preferred solution. A wizard is a kind of state machine where states can be a form view, an action or transition.

Let’s create a wizard that converts the opportunities by asking for the end date.

First we define a ModelView class in opportunity.py:

```python
class ConvertStart(ModelView):
    "Convert Opportunities"
    __name__ = 'training.opportunity.convert.start'
    end_date = fields.Date("End Date", required=True)
```

And we register it in the Pool in __init__.py:
def register():
    Pool.register(
       ..., opportunity.ConvertStart,
       module='opportunity', type_='model')

Then the form view record in opportunity.xml:

<tryton>
<data>
...
<record model="ir.ui.view" id="opportunity_convert_start_view_form">
    <field name="model">training.opportunity.convert.start</field>
    <field name="type">form</field>
    <field name="name">opportunity_convert_start_form</field>
</record>
</data>
</tryton>

And the view in view/opportunity_convert_start_form.xml:

<form col="2">
    <label string="Convert Opportunities?" id="convert_opportunities" colspan="2" xalign="0"/>
    <label name="end_date"/>
    <field name="end_date"/>
</form>

Now we can define the Wizard with a start StateView for the form and a convert StateTransition in opportunity.py:

from trytond.wizard import Wizard, StateView, StateTransition, Button
...

class Opportunity(...):
    ...
    @classmethod
    @Workflow.transition('converted')
    def convert(cls, opportunities, end_date=None):
       pool = Pool()
       Date = pool.get('ir.date')
       cls.write(opportunities, {
           'end_date': end_date or Date.today(),
        })
...

class Convert(Wizard):
    "Convert Opportunities"
    __name__ = 'training.opportunity.convert'

    start = StateView(
        'training.opportunity.convert.start',
        'opportunity.opportunity_convert_start_view_form', [Button("Cancel", 'end', 'tryton-cancel'),

(continues on next page)
Button("Convert", 'convert', 'tryton-ok', default=True),
]
convert = StateTransition()

def transition_convert(self):
    self.model.convert(self.records, self.start.end_date)
    return 'end'

Note: We added an optional end_date to the convert method.

And we register it in the Pool as type wizard in __init__.py:

def register():
    ...
    Pool.register(
        opportunity.Convert,
        module='opportunity', type_='wizard')

Finally we just need to create a ir.action.wizard and ir.action.keyword in opportunity.xml:

<tryton>
    <data>
        ...
        <record model="ir.action.wizard" id="act_convert_opportunities">
            <field name="name">Convert Opportunities</field>
            <field name="wiz_name">training.opportunity.convert</field>
            <field name="model">training.opportunity</field>
        </record>
        <record model="ir.action.keyword" id="act_convert_opportunities_keyword">
            <field name="keyword">form_action</field>
            <field name="model">training.opportunity,-1</field>
            <field name="action" ref="act_convert_opportunities"/>
        </record>
    </data>
</tryton>

The ir.action.wizard links the Wizard with the Model.

name
    The string that is shown on the menu.

wiz_name
    The name of the Wizard.

model
    The name of the Model.

And the ir.action.keyword makes the Wizard available as action to any training.opportunity.

keyword
    The type of keyword.

model
    The model or record for which the action must be displayed. Use -1 as id for any record.
action

The link to the action.

Update database

As we have defined new fields and XML records, we need to update the database with:

```bash
$ trytond-admin -d test --all
```

And restart the server and reconnect with the client to test the wizard:

```bash
$ trytond
```

Let’s create a report to print opportunities.

Create report

A frequent requirement is to generate a printable document for a record. For that we use `trytond.report.Report` which provides the tooling to render OpenDocument based on relatorio template.

First we create a `trytond.report.Report` class in `opportunity.py`:

```python
from trytond.report import Report
...
class OpportunityReport(Report):
    __name__ = 'training.opportunity.report'
```

And we register it in the `Pool` as type `report` in `__init__.py`:

```python
def register():
    ...
    Pool.register(
        opportunity.OpportunityReport,
        module='opportunity', type_='report'
    )
```

Now we have to create a `ir.action.report` and `ir.action.keyword` in `opportunity.xml`:

```xml
<tryton>
    <data>
        ...
        <record model="ir.action.report" id="report_opportunity">
            <field name="name">Opportunity</field>
            <field name="report_name">training.opportunity.report</field>
            <field name="model">training.opportunity</field>
            <field name="report">opportunity/opportunity.fodt</field>
            <field name="template_extension">odt</field>
        </record>
        <record model="ir.action.keyword" id="report_opportunity_keyword">
            <field name="keyword">form_print</field>
            <field name="model">training.opportunity,-1</field>
        </record>
    </data>
</tryton>
```
The `ir.action.report` links the `trytond.report.Report` with the `Model`.

**name**
- The string that is shown on the menu.

**report_name**
- The name of the `trytond.report.Report`.

**model**
- The name of the `Model`.

**report**
- The path to the template file starting with the module directory.

**template_extension**
- The template format.

And like for the `wizard`, the `ir.action.keyword` makes the `trytond.report.Report` available as action to any `training.opportunity`.

Finally we create the OpenDocument template as `opportunity.fodt` using LibreOffice. We use the Genshi XML Template Language implemented by relatorio using Placeholder Text. The rendering context contains the variable `records` which is a list of selected record instances.

Here is an example of the directives to insert in the document:

```xml
<for each="opportunity in records">
    Opportunity: <opportunity.rec_name>
    Party: <opportunity.party.rec_name>
    Start Date: <format_date(opportunity.start_date) if opportunity.start_date else ''>
    End Date: <format_date(opportunity.end_date) if opportunity.end_date else ''>
    Comment:
    <for each="line in (opportunity.comment or '').splitlines()">
        <line/>
    </for>
</for>
```

**Note:** We must render text field line by line because OpenDocument does not consider simple breakline.

### Update database

As we have registered new report and XML records, we need to update the database with:

```
$ trytond-admin -d test --all
```

And restart the server and reconnect with the client to test rendering the report:

```
$ trytond
```

Next we create a reporting model using SQL query.
Define aggregated model

Aggregated data are useful to analyze business. Tryton can provide such data using `ModelSQL` class which are not based on an existing table in the database but using a SQL query. This is done by defining a `table_query()` method which returns a SQL `FromItem`.

Let's create a `ModelSQL` which aggregate the number of opportunity converted or lost per month.

First we create a `ModelSQL` class which defines a `table_query()` in `opportunity.py`:

```python
from sql import Literal
from sql.aggregate import Count, Min
from sql.functions import CurrentTimestamp, DateTrunc

class OpportunityMonthly(ModelSQL, ModelView):
    __name__ = 'training.opportunity.monthly'
    month = fields.Date("Month")
    converted = fields.Integer("Converted")
    lost = fields.Integer("Lost")

    @classmethod
def table_query(cls):
        pool = Pool()
        Opportunity = pool.get('training.opportunity')
        opportunity = Opportunity.__table__()
        month = cls.month.sql_cast(  
            DateTrunc('month', opportunity.end_date))
        query = opportunity.select(  
            Literal(0).as_('create_uid'),  
            CurrentTimestamp().as_('create_date'),  
            Literal(None).as_('write_uid'),  
            Literal(None).as_('write_date'),  
            Min(opportunity.id).as_('id'),  
            month.as_('month'),  
            Count(  
                Literal(''),  
                filter_=opportunity.state == 'converted').as_('converted'),  
            Count(  
                Literal(''),  
                filter_=opportunity.state == 'lost').as_('lost'),  
            where=opportunity.state.in_(['converted', 'lost']),
            group_by=[month])
        return query
```

**Note:** The table query must return a value for all the fields of the model but also a unique `id` and a value for the create and write fields.

**Note:** We get the SQL table from the `__table__()` method.
Note: We use `sql_cast()` to convert the timestamp returned by `date_trunc` into a `date`.

Then as usual we register the `ModelSQL` class in the in the `Pool` as type `model` in `__init__.py`:

```python
def register():
    ...
    Pool.register(
        opportunity.OpportunityMonthly,
        module='opportunity', type_='model')
```

And to display we create a list view and the menu entry in `opportunity.xml`:

```xml
<tryton>
    <data>
        ...
        <record model="ir.ui.view" id="opportunity_monthly_view_list">
            <field name="model">training.opportunity.monthly</field>
            <field name="type">tree</field>
            <field name="name">opportunity_monthly_list</field>
        </record>
        <record model="ir.action.act_window" id="act_opportunity_monthly_form">
            <field name="name">Monthly Opportunities</field>
            <field name="res_model">training.opportunity.monthly</field>
        </record>
        <record model="ir.action.act_window.view" id="act_opportunity_monthly_form_view">
            <field name="view" ref="opportunity_monthly_view_list"/>
            <field name="act_window" ref="act_opportunity_monthly_form"/>
            <field name="sequence" eval="10"/>
        </record>
        <menuitem parent="menu_opportunity" action="act_opportunity_monthly_form" sequence="50" id="menu_opportunity_monthly_form"/>
    </data>
</tryton>
```

And now the view in `view/opportunity_monthly_list.xml`:

```xml
<tree>
    <field name="month"/>
    <field name="converted"/>
    <field name="lost"/>
</tree>
```
Update database

As we have registered new model and XML records, we need to update the database with:

```
$ trytond-admin -d test --all
```

And restart the server and reconnect with the client to test computing aggregate:

```
$ trytond
```

**Note:** As you can see the model behaves like the other models, except that you can not create, delete nor write on them.

This is all for your first module. If you want to learn more about Tryton, you can continue on specific topics.

Let’s start with *installing Tryton for developers.*
INDICES, GLOSSARY AND TABLES

• genindex
• modindex
• search
trytond.bus, 116
trytond.cache, 114
trytond.exceptions, 109
trytond.filestore, 114
trytond.i18n, 113
trytond.model, 68
trytond.model.exceptions, 109
trytond.model.fields, 82
trytond.pool, 111
trytond.pyson, 102
trytond.rpc, 112
trytond.sendmail, 113
trytond.tests.test_tryton, 117
trytond.tests.tools, 118
trytond.tools, 110
trytond.tools.email_, 111
trytond.tools.immutabledict, 111
trytond.tools.singleton, 111
trytond.transaction, 107
trytond.wizard, 98
Symbols
__access__ (trytond.model.Model attribute), 68
__name__ (trytond.model.Model attribute), 68
__name__ (trytond.wizard.Wizard attribute), 99
__names__() (trytond.model.Model class method), 69
__post_setup__() (trytond.model.Model class method), 69
__post_setup__() (trytond.pool.PoolBase class method), 112
__post_setup__() (trytond.wizard.Wizard class method), 99
__queue__ (trytond.model.Model attribute), 69
__register__() (trytond.model.Model class method), 69
__register__() (trytond.pool.PoolBase class method), 112
__register__() (trytond.wizard.Wizard class method), 99
__rpc__ (trytond.model.Model attribute), 68
__rpc__ (trytond.wizard.Wizard attribute), 99
__setup__() (trytond.model.Model class method), 69
__setup__() (trytond.pool.PoolBase class method), 112
__setup__() (trytond.wizard.Wizard class method), 99
__table__() (trytond.model.ModelSQL class method), 75
__table_handler__() (trytond.model.ModelSQL class method), 75
__table_history__() (trytond.model.ModelSQL class method), 75
__buttons (trytond.model.ModelView attribute), 70
__defaults (trytond.model.Model attribute), 69
__fields (trytond.model.Model attribute), 69
__history (trytond.model.ModelSQL attribute), 74
__order (trytond.model.ModelSQL attribute), 74
__order_name (trytond.model.ModelSQL attribute), 74
__rec_name (trytond.model.Model attribute), 68
__record (trytond.model.Model attribute), 69
__sql_constraints (trytond.model.ModelSQL attribute), 74
__table (trytond.model.ModelSQL attribute), 74
__transition_state (trytond.model.Workflow attribute), 77
__transitions (trytond.model.Workflow attribute), 77

A
AccessButtonError, 109
AccessError, 109
action_id (trytond.wizard.StateAction attribute), 101
activate_module() (in module trytond.tests.test_tryton), 117
activate_modules() (in module trytond.tests.tools), 118
active (trytond.model.DictSchemaMixin attribute), 87
add_remove (trytond.model.fields.Many2Many attribute), 96
add_remove (trytond.model.fields.One2Many attribute), 94
And (class in trytond.pyson), 104
atexit() (trytond.transaction.Transaction method), 109
autocomplete (trytond.model.fields.Char attribute), 87
avatar (trytond.model.AvatarMixin attribute), 82
avatar_mixin() (in module trytond.model), 82
avatar_url (trytond.model.AvatarMixin attribute), 82
AvatarMixin (class in trytond.model), 82
avatars (trytond.model.AvatarMixin attribute), 82

B
BigInteger (class in trytond.model.fields), 86
Binary (class in trytond.model.fields), 90
Bool (class in trytond.pyson), 104
Boolean (class in trytond.model.fields), 86
browse() (trytond.model.ModelStorage class method), 73
Bus (class in trytond.bus), 116
Button (class in trytond.wizard), 101
button() (trytond.model.ModelView static method), 70
button_action() (trytond.model.ModelView static method), 70
button_change() (trytond.model.ModelView static method), 70
buttons (trytond.wizard.StateView attribute), 100

C
Cache (class in trytond.cache), 115
| **Eval** (class in tryton.pyson) | 103 |
| **eval()** (tryton.pyson.PYSON class method) | 102 |
| **Exclude** (class in tryton.model) | 77 |
| **excludes** (tryton.model.Exclude attribute) | 77 |
| **execute()** (tryton.wizard.Wizard class method) | 99 |
| **export_data()** (tryton.model.ModelStorage class method) | 73 |
| **export_data_domain()** (tryton.model.ModelStorage class method) | 73 |
| **expression** (tryton.model.Check attribute) | 76 |

**F**

| **Field** (class in tryton.model.fields) | 83 |
| **field** (tryton.model.fields.One2Many attribute) | 94 |
| **fields_get()** (tryton.model.Model class method) | 69 |
| **fields_view_get()** (tryton.model.ModelView class method) | 70 |
| **file_id** (tryton.model.fields.Binary attribute) | 90 |
| **file_open()** (in module tryton.tools) | 110 |
| **filename** (tryton.model.fields.Binary attribute) | 90 |
| **FileStore** (class in tryton.filestore) | 114 |
| **fill()** (tryton.pool.Pool method) | 112 |
| **filter()** (tryton.model.fields.Many2Many attribute) | 96 |
| **filter()** (tryton.model.fields.Many2One attribute) | 95 |
| **find_dir()** (in module tryton.tools) | 110 |
| **find_path()** (in module tryton.tools) | 110 |
| **firstline()** (in module tryton.tools) | 110 |
| **Float** (class in tryton.model.fields) | 88 |
| **ForeignKeyError** | 110 |
| **format** (tryton.model.fields.DateTime attribute) | 89 |
| **format** (tryton.model.fields.Time attribute) | 89 |
| **format()** (tryton.model.DictSchemaMixin method) | 79 |
| **fresh_session** (tryton.rpc.RPC attribute) | 113 |
| **FullText** (class in tryton.model.fields) | 88 |
| **Function** (class in tryton.model.fields) | 97 |

**G**

| **generate_avatar()** (tryton.model.AvatarMixin class method) | 82 |
| **Get** (class in tryton.pyson) | 105 |
| **get()** (tryton.cache.Cache method) | 115 |
| **get()** (tryton.filestore.FileStore method) | 114 |
| **get()** (tryton.model.fields.Function method) | 97 |
| **get()** (tryton.pool.Pool method) | 111 |
| **get_action()** (tryton.wizard.StateAction method) | 101 |
| **get_buttons()** (tryton.wizard.StateView method) | 100 |
| **get_defaults()** (tryton.wizard.StateView method) | 100 |
| **get_digits()** (tryton.model.DigitsMixin method) | 80 |
| **get_keys()** (tryton.model.DictSchemaMixin class method) | 78 |
| **get_multivalue()** (tryton.model.MultiValueMixin method) | 81 |
| **get_rec_name()** (tryton.model.ModelStorage method) | 74 |
| **get_relation()** (tryton.model.fields.Many2Many method) | 96 |
| **get_relation()** (tryton.model.fields.One2One method) | 96 |
| **get_relation_fields()** (tryton.model.DictSchemaMixin class method) | 79 |
| **get_selection_json()** (tryton.model.DictSchemaMixin method) | 79 |
| **get_singleton()** (tryton.model.ModelSingleton class method) | 79 |
| **get_smtp_server()** (in module tryton.sendmail) | 114 |
| **get_symbol()** (tryton.model.SymbolMixin method) | 80 |
| **get_target()** (tryton.model.fields.Many2Many method) | 96 |
| **get_target()** (tryton.model.fields.One2One method) | 96 |
| **get_view()** (tryton.wizard.StateView method) | 100 |
| **getmany()** (tryton.filestore.FileStore method) | 114 |
| **getter()** (tryton.model.fields.Function attribute) | 97 |
| **getter_with_context** (tryton.model.fields.Function attribute) | 97 |
| **gettext()** (in module tryton.i18n) | 113 |
| **Greater** (class in tryton.pyson) | 104 |

**H**

| **has_avatar** (tryton.model.AvatarMixin attribute) | 82 |
| **headers()** (tryton.rpc.RPC method) | 113 |
| **help()** (tryton.model.DictSchemaMixin attribute) | 78 |
| **help()** (tryton.model.fields.Field attribute) | 83 |
| **help_selection** (tryton.model.fields.MultiSelection attribute) | 92 |
| **help_selection** (tryton.model.fields.Reference attribute) | 92 |
| **help_selection** (tryton.model.fields.Selection attribute) | 91 |
| **history_revisions()** (tryton.model.ModelSQL class method) | 75 |
| **hit** (tryton.cache.Cache attribute) | 115 |

**I**

| **icon** (tryton.wizard.Button attribute) | 101 |
| **Id** (class in tryton.pyson) | 107 |
| **id** (tryton.model.Model attribute) | 68 |
| **If** (class in tryton.pyson) | 105 |
| **ImmutableDict** (class in tryton.tools.immutabledict) | 111 |
| **import_data()** (tryton.model.ModelStorage class method) | 73 |
| **ImportDataError** | 109 |
| **In** (class in tryton.pyson) | 105 |
translated() (trytond.model.fields.Dict method), 98
translated() (trytond.model.fields.MultiSelection method), 92
translated() (trytond.model.fields.Reference method), 92
translated() (trytond.model.fields.Selection method), 91
tree() (in module trytond.model), 81
TreeMixin (class in trytond.model), 81
trigger_create() (trytond.model.ModelStorage class method), 71
trigger_delete() (trytond.model.ModelStorage class method), 71
trigger_records (trytond.transaction.Transaction attribute), 108
trigger_write() (trytond.model.ModelStorage class method), 72
trigger_write_get_eligibles() (trytond.model.ModelStorage class method), 72
trytond.bus module, 116
trytond.cache module, 114
trytond.exceptions module, 109
trytond.filestore module, 114
trytond.i18n module, 113
trytond.model module, 68
trytond.model.exceptions module, 109
trytond.model.fields module, 82
trytond.pool module, 111
trytond.pyson module, 102
trytond.rpc module, 112
trytond.sendmail module, 113
trytond.tests.test_tryton module, 117
trytond.tests.tools module, 118
trytond.tools module, 110
trytond.tools.email_ module, 111
trytond.tools.immutabledict module, 111
trytond.tools.singleton module, 111
trytond.transaction module, 107
trytond.wizard module, 98
TrytonException, 109
types() (trytond.pyson.PYSON method), 102
U
unescape_wildcard() (in module trytond.tools), 110
union_column() (trytond.model.UnionMixin class method), 79
union_columns() (trytond.model.UnionMixin class method), 79
union_models() (trytond.model.UnionMixin static method), 79
union_shard() (trytond.model.UnionMixin class method), 79
union_unshard() (trytond.model.UnionMixin class method), 79
UnionMixin (class in trytond.model), 79
Unique (class in trytond.model), 76
unique (trytond.rpc.RPC attribute), 112
USER (in module trytond.tests.test_tryton), 117
user (trytond.transaction.Transaction attribute), 108
UserError, 109
UserWarning, 109
V
validate (trytond.wizard.Button attribute), 101
validate() (trytond.model.ModelStorage class method), 74
validate_fields() (trytond.model.ModelStorage class method), 74
validation_depends (trytond.model.fields.Field attribute), 84
ValidationError, 109
ValueMixin (class in trytond.model), 81
view (trytond.wizard.StateView attribute), 100
view_attributes() (trytond.model.ModelView class method), 71
view_toolbar_get() (trytond.model.ModelView class method), 70
W
where (trytond.model.Exclude attribute), 77
with_transaction() (in module trytond.tests.test_tryton), 118
Wizard (class in trytond.wizard), 99
Workflow (class in trytond.model), 77
write() (trytond.model.ModelStorage class method), 72
write_date (trytond.model.ModelStorage attribute), 71
write_uid (trytond.model.ModelStorage attribute), 71