Object Relational Mapping module:

- Hierarchical structure
- Constraints consistency and validation
- Object metadata depends on its status
- Optimised processing by complex query (multiple actions at once)
- Default field values
- Permissions optimisation
- Persistent object: DB postgresql
- Data conversion
- Multi-level caching system
- Two different inheritance mechanisms
- **Rich set of field types:**
  - classical (varchar, integer, boolean, …)
  - relational (one2many, many2one, many2many)
  - functional

Models

Model fields are defined as attributes on the model itself:

```
from odoo import models, fields
class AModel(models.Model):
    _name = 'a.model.name'

    field1 = fields.Char()
```

> ⚠ **Warning**
> this means you cannot define a field and a method with the same name, the last one will silently overwrite the former ones.

By default, the field’s label (user-visible name) is a capitalized version of the field name, this can be overridden with the `string` parameter.

```
field2 = fields.Integer(string="Field Label")
```

For the list of field types and parameters, see the fields reference.

Default values are defined as parameters on fields, either as a value:

```
name = fields.Char(default="a value")
```

or as a function called to compute the default value, which should return that value:

```
def _default_name(self):
    return self.get_value()

name = fields.Char(default=lambda self: self._default_name())
```

API

class **odoo.models.BaseModel** (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L254)

Base class for Odoo models.

Odoo models are created by inheriting one of the following:

`Model` for regular database-persisted models

`TransientModel` for temporary data, stored in the database but automatically vacuumed every so often

`AbstractModel` for abstract super classes meant to be shared by multiple inheriting models

The system automatically instantiates every model once per database. Those instances represent the available models on each database, and depend on which modules are installed on that database. The actual class of each instance is built from the Python classes that create and inherit from the corresponding model.

Every model instance is a "recordset", i.e., an ordered collection of records of the model. Recordsets are returned by methods like `browse()`, `search()`, or field accesses. Records have no explicit representation: a record is represented as a recordset of one record.

To create a class that should not be instantiated, the `_register` attribute may be set to False.

_auto = *False*

Whether a database table should be created. If set to `False`, override `init()` to create the database table.

Automatically defaults to `True` for `Model` and `TransientModel`, `False` for `AbstractModel`.

> To create a model without any table, inherit from `AbstractModel`.

_log_access

Whether the ORM should automatically generate and update the Access Log fields.

Defaults to whatever value was set for `_auto`.

`_table` `=` ***`None`***

SQL table name used by model if `_auto`

`_sequence` `=` ***`None`***

SQL sequence to use for ID field

`_sql_constraints` `=` ***`[]`***

SQL constraints [(name, sql_def, message)]

`_register` `=` ***`True`***

registry visibility

`_abstract` `=` ***`True`***

Whether the model is *abstract*.

> ➧ **See also**
>
> `AbstractModel`

`_transient` `=` ***`False`***

Whether the model is *transient*.

> ➧ **See also**
>
> `TransientModel`

`_name` `=` ***`None`***

the model name (in dot-notation, module namespace)

`_description` `=` ***`None`***

the model's informal name

`_inherit` `=` ***`None`***

Python-inherited models:

Type:

str (https://docs.python.org/3/library/stdtypes.html#str) or list (https://docs.python.org/3/library/stdtypes.html#list)(str (https://docs.python.org/3/library/stdtypes.html#str))

> If `_name` is set, name(s) of parent models to inherit from
>
> If `_name` is unset, name of a single model to extend in-place

`_inherits = {}`

dictionary {'parent_model': 'm2o_field'} mapping the _name of the parent business objects to the names of the corresponding foreign key fields to use:

```
_inherits = {
    'a.model': 'a_field_id',
    'b.model': 'b_field_id'
}
```

implements composition-based inheritance: the new model exposes all the fields of the inherited models but stores none of them: the values themselves remain stored on the linked record.

> **⚠ Warning**
>
> if multiple fields with the same name are defined in the `_inherits` -ed models, the inherited field will correspond to the last one (in the inherits list order).

`_rec_name = None`

field to use for labeling records, default: `name`

`_order = 'id'`

default order field for searching results

`_check_company_auto = False`

On write and create, call `_check_company` to ensure companies consistency on the relational fields having `check_company=True` as attribute.

`_parent_name = 'parent_id'`

the many2one field used as parent field

`_parent_store = False`

set to True to compute parent_path field.

Alongside a `parent_path` field, sets up an indexed storage of the tree structure of records, to enable faster hierarchical queries on the records of the current model using the `child_of` and `parent_of` domain operators.

`_date_name = 'date'`

field to use for default calendar view

`_fold_name = 'fold'`

field to determine folded groups in kanban views

AbstractModel

odoo.models.AbstractModel [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L254)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L254)

alias of `odoo.models.BaseModel`

Model

class **odoo.models.Model** [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L6400)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L6400)

Main super-class for regular database-persisted Odoo models.

Odoo models are created by inheriting from this class:

```
class user(Model):
    ...
```

The system will later instantiate the class once per database (on which the class' module is installed).

_auto = *True*

Whether a database table should be created. If set to `False`, override `init()` to create the database table.

Automatically defaults to `True` for `Model` and `TransientModel`, `False` for `AbstractModel`.

> To create a model without any table, inherit from `AbstractModel`.

_abstract = *False*

Whether the model is *abstract*.

> ➡ **See also**
> `AbstractModel`

_transient = *False*

Whether the model is *transient*.

> ➡ **See also**
> `TransientModel`

TransientModel

class **odoo.models.TransientModel** [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L6416)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L6416)

Model super-class for transient records, meant to be temporarily persistent, and regularly vacuum-cleaned.

A TransientModel has a simplified access rights management, all users can create new records, and may only access the records they created. The superuser has unrestricted access to all TransientModel records.

_auto = *True*

Whether a database table should be created. If set to `False` , override `init()` to create the database table.

Automatically defaults to `True` for `Model` and `TransientModel` , `False` for `AbstractModel` .

> To create a model without any table, inherit from `AbstractModel` .

_abstract = *False*

Whether the model is *abstract*.

> ➡ **See also**
> `AbstractModel`

_transient = *True*

Whether the model is *transient*.

> ➡ **See also**
> `TransientModel`

Fields

class **odoo.fields.Field** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L92)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L92)

The field descriptor contains the field definition, and manages accesses and assignments of the corresponding field on records. The following attributes may be provided when instanciating a field:

Parameters:

string (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – the label of the field seen by users; if not set, the ORM takes the field name in the class (capitalized).

help (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – the tooltip of the field seen by users

invisible – whether the field is invisible (boolean, by default `False`)

readonly (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the field is readonly (default: **False**)
This only has an impact on the UI. Any field assignation in code will work (if the field is a stored field or an inversable one).

required (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the value of the field is required (default: **False**)

index (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the field is indexed in database. Note: no effect on non-stored and virtual fields. (default: **False**)

default (**value or callable**) – the default value for the field; this is either a static value, or a function taking a recordset and returning a value; use **default=None** to discard default values for the field

states (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – a dictionary mapping state values to lists of UI attribute-value pairs; possible attributes are: **readonly** , **required** , **invisible** .

> ⚠ **Warning**
>
> Any state-based condition requires the **state** field value to be available on the client-side UI. This is typically done by including it in the relevant views, possibly made invisible if not relevant for the end-user.

groups (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – comma-separated list of group xml ids (string); this restricts the field access to the users of the given groups only

company_dependent (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the field value is dependent of the current company;
The value isn't stored on the model table. It is registered as **ir.property** . When the value of the company_dependent field is needed, an **ir.property** is searched, linked to the current company (and current record if one property exists).
If the value is changed on the record, it either modifies the existing property for the current record (if one exists), or creates a new one for the current company and res_id.
If the value is changed on the company side, it will impact all records on which the value hasn't been changed.

copy (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the field value should be copied when the record is duplicated (default: **True** for normal fields, **False** for **one2many** and computed fields, including property fields and related fields)

store (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the field is stored in database (default: **True** , **False** for computed fields)

group_operator (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – aggregate function used by **read_group()** when grouping on this field.
Supported aggregate functions are:

array_agg : values, including nulls, concatenated into an array

count : number of rows

count_distinct : number of distinct rows

bool_and : true if all values are true, otherwise false

bool_or : true if at least one value is true, otherwise false

max : maximum value of all values

min : minimum value of all values

avg : the average (arithmetic mean) of all values

sum : sum of all values

group_expand (**str** (https://docs.python.org/3/library/stdtypes.html#str)) –

function used to expand read_group results when grouping on the current field.

```
@api.model
def _read_group_selection_field(self, values, domain, order):
    return ['choice1', 'choice2', ...] # available selection choices.

@api.model
def _read_group_many2one_field(self, records, domain, order):
    return records + self.search([custom_domain])
```

Computed Fields

Parameters:

compute (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – name of a method that computes the field

> 🡆 **See also**
> **Advanced Fields/Compute fields**

compute_sudo (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the field should be recomputed as superuser to bypass access rights (by default **True** for stored fields, **False** for non stored fields)

inverse (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – name of a method that inverses the field (optional)

search (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – name of a method that implement search on the field (optional)

related (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – sequence of field names

> 🡆 **See also**
> **Advanced fields/Related fields**

Basic Fields

***class* odoo.fields.Boolean (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1164)**

Encapsulates a **bool** (https://docs.python.org/3/library/functions.html#bool).

***class* odoo.fields.Char (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1551)**

Basic string field, can be length-limited, usually displayed as a single-line string in clients.

Parameters:

size (**int** (https://docs.python.org/3/library/functions.html#int)) – the maximum size of values stored for that field

trim (**bool** (https://docs.python.org/3/library/functions.html#bool)) – states whether the value is trimmed or not (by default, **True**). Note that the trim operation is applied only by the web client.

translate (**bool** (https://docs.python.org/3/library/functions.html#bool) **or** **callable**) – enable the translation of the field’s values; use `translate=True` to translate field values as a whole; `translate` may also be a callable such that `translate(callback, value)` translates `value` by using `callback(term)` to retrieve the translation of terms.

class odoo.fields.Float (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1217)

Encapsulates a `float` (https://docs.python.org/3/library/functions.html#float).

The precision digits are given by the (optional) `digits` attribute.

Parameters:

digits (**tuple** (https://docs.python.org/3/library/stdtypes.html#tuple) **(** **int** (https://docs.python.org/3/library/functions.html#int) **,** **int** (https://docs.python.org/3/library/functions.html#int) **) or** **str** (https://docs.python.org/3/library/stdtypes.html#str)) – a pair (total, decimal) or a string referencing a `DecimalPrecision` record name.

When a float is a quantity associated with an unit of measure, it is important to use the right tool to compare or round values with the correct precision.

The Float class provides some static methods for this purpose:

`round()` to round a float with the given precision. `is_zero()` to check if a float equals zero at the given precision. `compare()` to compare two floats at the given precision.

ⓘ Example

To round a quantity with the precision of the unit of mesure:

```
fields.Float.round(self.product_uom_qty, precision_rounding=self.product_uom_id.r
```

To check if the quantity is zero with the precision of the unit of mesure:

```
fields.Float.is_zero(self.product_uom_qty, precision_rounding=self.product_uom_id
```

To compare two quantities:

```
field.Float.compare(self.product_uom_qty, self.qty_done, precision_rounding=self.p
```

The compare helper uses the __cmp__ semantics for historic purposes, therefore the proper, idiomatic way to use this helper is like so:

> ***if result == 0, the first and second floats are equal if result < 0, the first float is lower than the second if result > 0, the first float is greater than the second***

class **odoo.fields.Integer** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1179)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1179)

Encapsulates an `int` (https://docs.python.org/3/library/functions.html#int).

Advanced Fields

class **odoo.fields.Binary** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1913)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1913)

Encapsulates a binary content (e.g. a file).

Parameters:

attachment (`bool` (https://docs.python.org/3/library/functions.html#bool)) – whether the field should be stored as `ir_attachment` or in a column of the model's table (default: `True`).

class **odoo.fields.Html** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1630)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1630)

Encapsulates an html code content.

Parameters:

- **sanitize** (`bool` (https://docs.python.org/3/library/functions.html#bool)) – whether value must be sanitized (default: `True`)
- **sanitize_tags** (`bool` (https://docs.python.org/3/library/functions.html#bool)) – whether to sanitize tags (only a white list of attributes is accepted, default: `True`)
- **sanitize_attributes** (`bool` (https://docs.python.org/3/library/functions.html#bool)) – whether to sanitize attributes (only a white list of attributes is accepted, default: `True`)
- **sanitize_style** (`bool` (https://docs.python.org/3/library/functions.html#bool)) – whether to sanitize style attributes (default: `False`)
- **strip_style** (`bool` (https://docs.python.org/3/library/functions.html#bool)) – whether to strip style attributes (removed and therefore not sanitized, default: `False`)
- **strip_classes** (`bool` (https://docs.python.org/3/library/functions.html#bool)) – whether to strip classes attributes (default: `False`)

class **odoo.fields.Image** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L2110)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L2110)

Encapsulates an image, extending `Binary` .

If image size is greater than the `max_width` / `max_height` limit of pixels, the image will be resized to the limit by keeping aspect ratio.

Parameters:

- **max_width** (`int` (https://docs.python.org/3/library/functions.html#int)) – the maximum width of the image (default: `0` , no limit)
- **max_height** (`int` (https://docs.python.org/3/library/functions.html#int)) – the maximum height of the image (default: `0` , no limit)

verify_resolution (**bool** (https://docs.python.org/3/library/functions.html#bool)) – whether the image resolution should be verified to ensure it doesn't go over the maximum image resolution (default: `True`). See `odoo.tools.image.ImageProcess` for maximum image resolution (default: `45e6`).

> If no `max_width` / `max_height` is specified (or is set to 0) and `verify_resolution` is False, the field content won't be verified at all and a **Binary** field should be used.

***class* odoo.fields.Monetary (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1317)**

Encapsulates a **float** (https://docs.python.org/3/library/functions.html#float) expressed in a given `res_currency` .

The decimal precision and currency symbol are taken from the `currency_field` attribute.

Parameters:

currency_field (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – name of the **Many2one** field holding the `res_currency` this monetary field is expressed in (default: `'currency_id'`)

***class* odoo.fields.Selection (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L2175)**

Encapsulates an exclusive choice between different values.

Parameters:

selection (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **tuple** (https://docs.python.org/3/library/stdtypes.html#tuple) **(** **str** (https://docs.python.org/3/library/stdtypes.html#str) **,** **str** (https://docs.python.org/3/library/stdtypes.html#str) **)) or callable or** **str** (https://docs.python.org/3/library/stdtypes.html#str)) – specifies the possible values for this field. It is given as either a list of pairs `(value, label)` , or a model method, or a method name.

selection_add (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **tuple** (https://docs.python.org/3/library/stdtypes.html#tuple) **(** **str** (https://docs.python.org/3/library/stdtypes.html#str) **,** **str** (https://docs.python.org/3/library/stdtypes.html#str) **))**) –
provides an extension of the selection in the case of an overridden field. It is a list of pairs `(value, label)` or singletons `(value,)` , where singleton values must appear in the overridden selection. The new values are inserted in an order that is consistent with the overridden selection and this list:

```
selection = [('a', 'A'), ('b', 'B')]
selection_add = [('c', 'C'), ('b',)]
> result = [('a', 'A'), ('c', 'C'), ('b', 'B')]
```

ondelete –
provides a fallback mechanism for any overridden field with a selection_add. It is a dict that maps every option from the selection_add to a fallback action.
This fallback action will be applied to all records whose selection_add option maps to it.
The actions can be any of the following:

'set null' – the default, all records with this option will have their selection value set to False.

'cascade' – all records with this option will be deleted along with the option itself.

'set default' – all records with this option will be set to the default of the field definition

<callable> – a callable whose first and only argument will be the set of records containing the specified Selection option, for custom processing

The attribute `selection` is mandatory except in the case of `related` or extended fields.

class odoo.fields.Text (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1609)

Very similar to `Char` but used for longer contents, does not have a size and usually displayed as a multiline text box.

Parameters:

translate (`bool` (https://docs.python.org/3/library/functions.html#bool) `or` `callable`) – enable the translation of the field's values; use `translate=True` to translate field values as a whole; `translate` may also be a callable such that `translate(callback, value)` translates `value` by using `callback(term)` to retrieve the translation of terms.

Date(time) Fields

`Dates` and `Datetimes` are very important fields in any kind of business application. Their misuse can create invisible yet painful bugs, this section aims to provide Odoo developers with the knowledge required to avoid misusing these fields.

When assigning a value to a Date/Datetime field, the following options are valid:

- A `date` or `datetime` object.
- A string in the proper server format:
  - `YYYY-MM-DD` for `Date` fields,
  - `YYYY-MM-DD HH:MM:SS` for `Datetime` fields.
- `False` or `None`.

The Date and Datetime fields class have helper methods to attempt conversion into a compatible type:

- `to_date()` will convert to a `datetime.date` (https://docs.python.org/3/library/datetime.html#datetime.date)
- `to_datetime()` will convert to a `datetime.datetime` (https://docs.python.org/3/library/datetime.html#datetime.datetime).

> ⓘ **Example**
>
> To parse date/datetimes coming from external sources:
>
> ```
> fields.Date.to_date(self._context.get('date_from'))
> ```

Date / Datetime comparison best practices:

Date fields can **only** be compared to date objects.

Datetime fields can **only** be compared to datetime objects.

> **⚠ Warning**
>
> Strings representing dates and datetimes can be compared between each other, however the result may not be the expected result, as a datetime string will always be greater than a date string, therefore this practice is **heavily** discouraged.

Common operations with dates and datetimes such as addition, substraction or fetching the start/end of a period are exposed through both `Date` and `Datetime` . These helpers are also available by importing `odoo.tools.date_utils` .

> Timezones
>
> Datetime fields are stored as `timestamp without timezone` columns in the database and are stored in the UTC timezone. This is by design, as it makes the Odoo database independent from the timezone of the hosting server system. Timezone conversion is managed entirely by the client side.

class **`odoo.fields.Date`** **(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1705)**

Encapsulates a python `date` (https://docs.python.org/3/library/datetime.html#datetime.date) object.

static **`add(`*`value, *args, **kwargs`*`)`** **(https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L179)**

Return the sum of `value` and a `relativedelta` .

Parameters:

- `value` – initial date or datetime.
- `args` – positional args to pass directly to `relativedelta` .
- `kwargs` – keyword args to pass directly to `relativedelta` .

Returns:

the resulting date/datetime.

static **`context_today(`*`record, timestamp=None`*`)`** **(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1724)**

Return the current date as seen in the client's timezone in a format fit for date fields.

> This method may be used to compute default values.

Parameters:

- `record` – recordset from which the timezone will be obtained.
- `timestamp` (`datetime`) – optional datetime value to use instead of the current date and time (must be a datetime, regular dates can't be converted between timezones).

Return type:

date

static **end_of(*value*, *granularity*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L140)](https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L140)

Get end of a time period from a date or a datetime.

Parameters:

`value` – initial date or datetime.

`granularity` – Type of period in string, can be year, quarter, month, week, day or hour.

Returns:

A date/datetime object corresponding to the start of the specified period.

static **start_of(*value*, *granularity*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L101)](https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L101)

Get start of a time period from a date or a datetime.

Parameters:

`value` – initial date or datetime.

`granularity` – type of period in string, can be year, quarter, month, week, day or hour.

Returns:

a date/datetime object corresponding to the start of the specified period.

static **subtract(*value*, **args*, ***kwargs*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L191)](https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L191)

Return the difference between `value` and a `relativedelta`.

Parameters:

`value` – initial date or datetime.

`args` – positional args to pass directly to `relativedelta`.

`kwargs` – keyword args to pass directly to `relativedelta`.

Returns:

the resulting date/datetime.

static **to_date(*value*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1749)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1749)

Attempt to convert `value` to a `date` object.

> ⚠ **Warning**
>
> If a datetime object is given as value, it will be converted to a date object and all datetime-specific information will be lost (HMS, TZ, …).

Parameters:

value (`str` (https://docs.python.org/3/library/stdtypes.html#str) `or` `date` `or` `datetime`) – value to convert.

Returns:

an object representing `value` .

Return type:

date or None (https://docs.python.org/3/library/constants.html#None)

static `to_string(value)` (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1777)

Convert a `date` or `datetime` object to a string.

Parameters:

value – value to convert.

Returns:

a string representing `value` in the server's date format, if `value` is of type `datetime` , the hours, minute, seconds, tzinfo will be truncated.

Return type:

str (https://docs.python.org/3/library/stdtypes.html#str)

static `today(*args)` (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1716)

Return the current day in the format expected by the ORM.

> This function may be used to compute default values.

class `odoo.fields.Datetime` (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1804)

Encapsulates a python `datetime` (https://docs.python.org/3/library/datetime.html#datetime.datetime) object.

static `add(value, *args, **kwargs)` (https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L179)

Return the sum of `value` and a `relativedelta` .

Parameters:

`value` – initial date or datetime.

`args` – positional args to pass directly to `relativedelta` .

`kwargs` – keyword args to pass directly to `relativedelta` .

Returns:

the resulting date/datetime.

static **context_timestamp(*record*, *timestamp*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1829)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1829)

Return the given timestamp converted to the client's timezone.

> This method is *not* meant for use as a default initializer, because datetime fields are automatically converted upon display on client side. For default values, `now()` should be used instead.

Parameters:

`record` – recordset from which the timezone will be obtained.

`timestamp` (**`datetime`**) – naive datetime value (expressed in UTC) to be converted to the client timezone.

Returns:

timestamp converted to timezone-aware datetime in context timezone.

Return type:

datetime

static **end_of(*value*, *granularity*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L140)](https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L140)

Get end of a time period from a date or a datetime.

Parameters:

`value` – initial date or datetime.

`granularity` – Type of period in string, can be year, quarter, month, week, day or hour.

Returns:

A date/datetime object corresponding to the start of the specified period.

static **now(**args*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1815)](https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1815)

Return the current day and time in the format expected by the ORM.

> This function may be used to compute default values.

static **start_of(*value*, *granularity*)** [(https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L101)](https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L101)

Get start of a time period from a date or a datetime.

Parameters:

`value` – initial date or datetime.

`granularity` – type of period in string, can be year, quarter, month, week, day or hour.

Returns:

a date/datetime object corresponding to the start of the specified period.

***static* subtract(*value*, **args*, ***kwargs*) (https://github.com/odoo/odoo/blob/14.0/odoo/tools/date_utils.py#L191)**

Return the difference between `value` and a `relativedelta`.

Parameters:

`value` – initial date or datetime.

`args` – positional args to pass directly to `relativedelta`.

`kwargs` – keyword args to pass directly to `relativedelta`.

Returns:

the resulting date/datetime.

***static* to_datetime(*value*) (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1857)**

Convert an ORM `value` into a `datetime` value.

Parameters:

value (`str` (https://docs.python.org/3/library/stdtypes.html#str) `or` `date` `or` `datetime`) – value to convert.

Returns:

an object representing `value`.

Return type:

datetime or None (https://docs.python.org/3/library/constants.html#None)

***static* to_string(*value*) (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1882)**

Convert a `datetime` or `date` object to a string.

Parameters:

value (`datetime` `or` `date`) – value to convert.

Returns:

a string representing `value` in the server’s datetime format, if `value` is of type `date`, the time portion will be midnight (00:00:00).

Return type:

str (https://docs.python.org/3/library/stdtypes.html#str)

***static* today(**args*) (https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L1824)**

Return the current day, at midnight (00:00:00).

Relational Fields

class **`odoo.fields.Many2one`** **(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L2499)**

The value of such a field is a recordset of size 0 (no record) or 1 (a single record).

Parameters:

- **`comodel_name`** (**`str`** (https://docs.python.org/3/library/stdtypes.html#str)) – name of the target model **`Mandatory`** except for related or extended fields.
- **`domain`** – an optional domain to set on candidate values on the client side (domain or string)
- **`context`** (**`dict`** (https://docs.python.org/3/library/stdtypes.html#dict)) – an optional context to use on the client side when handling that field
- **`ondelete`** (**`str`** (https://docs.python.org/3/library/stdtypes.html#str)) – what to do when the referred record is deleted; possible values are: **`'set null'`** , **`'restrict'`** , **`'cascade'`**
- **`auto_join`** (**`bool`** (https://docs.python.org/3/library/functions.html#bool)) – whether JOINs are generated upon search through that field (default: **`False`**)
- **`delegate`** (**`bool`** (https://docs.python.org/3/library/functions.html#bool)) – set it to **`True`** to make fields of the target model accessible from the current model (corresponds to **`_inherits`**)
- **`check_company`** (**`bool`** (https://docs.python.org/3/library/functions.html#bool)) – Mark the field to be verified in **`_check_company()`** (../howtos/company.html#odoo.models.Model._check_company). Add a default company domain depending on the field attributes.

class **`odoo.fields.One2many`** **(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L3029)**

One2many field; the value of such a field is the recordset of all the records in **`comodel_name`** such that the field **`inverse_name`** is equal to the current record.

Parameters:

- **`comodel_name`** (**`str`** (https://docs.python.org/3/library/stdtypes.html#str)) – name of the target model
- **`inverse_name`** (**`str`** (https://docs.python.org/3/library/stdtypes.html#str)) – name of the inverse **`Many2one`** field in **`comodel_name`**
- **`domain`** – an optional domain to set on candidate values on the client side (domain or string)
- **`context`** (**`dict`** (https://docs.python.org/3/library/stdtypes.html#dict)) – an optional context to use on the client side when handling that field
- **`auto_join`** (**`bool`** (https://docs.python.org/3/library/functions.html#bool)) – whether JOINs are generated upon search through that field (default: **`False`**)
- **`limit`** (**`int`** (https://docs.python.org/3/library/functions.html#int)) – optional limit to use upon read

The attributes **`comodel_name`** and **`inverse_name`** are mandatory except in the case of related fields or field extensions.

class **`odoo.fields.Many2many`** **(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L3305)**

Many2many field; the value of such a field is the recordset.

Parameters:

comodel_name – name of the target model (string) mandatory except in the case of related or extended fields

relation (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – optional name of the table that stores the relation in the database

column1 (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – optional name of the column referring to "these" records in the table **relation**

column2 (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – optional name of the column referring to "those" records in the table **relation**

The attributes `relation`, `column1` and `column2` are optional. If not given, names are automatically generated from model names, provided `model_name` and `comodel_name` are different!

Note that having several fields with implicit relation parameters on a given model with the same comodel is not accepted by the ORM, since those field would use the same table. The ORM prevents two many2many fields to use the same relation parameters, except if

both fields use the same model, comodel, and relation parameters are explicit; or

at least one field belongs to a model with `_auto = False`.

Parameters:

domain – an optional domain to set on candidate values on the client side (domain or string)

context (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – an optional context to use on the client side when handling that field

check_company (**bool** (https://docs.python.org/3/library/functions.html#bool)) – Mark the field to be verified in **_check_company()** (../howtos/company.html#odoo.models.Model._check_company). Add a default company domain depending on the field attributes.

limit (**int** (https://docs.python.org/3/library/functions.html#int)) – optional limit to use upon read

Pseudo-relational fields

class **odoo.fields.Reference** **(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L2388)**

Pseudo-relational field (no FK in database).

The field value is stored as a **string** (https://docs.python.org/3/library/stdtypes.html#str) following the pattern `"res_model.res_id"` in database.

class **odoo.fields.Many2oneReference** **(https://github.com/odoo/odoo/blob/14.0/odoo/fields.py#L2742)**

Pseudo-relational field (no FK in database).

The field value is stored as an **integer** (https://docs.python.org/3/library/functions.html#int) id in database.

Contrary to `Reference` fields, the model has to be specified in a `Char` field, whose name has to be specified in the `model_field` attribute for the current `Many2oneReference` field.

Parameters:

model_field (`str` (https://docs.python.org/3/library/stdtypes.html#str)) – name of the `Char` where the model name is stored.

Computed Fields

Fields can be computed (instead of read straight from the database) using the `compute` parameter. **It must assign the computed value to the field**. If it uses the values of other *fields*, it should specify those fields using `depends()` .

```
from odoo import api
total = fields.Float(compute='_compute_total')

@api.depends('value', 'tax')
def _compute_total(self):
    for record in self:
        record.total = record.value + record.value * record.tax
```

dependencies can be dotted paths when using sub-fields:

```
@api.depends('line_ids.value')
def _compute_total(self):
    for record in self:
        record.total = sum(line.value for line in record.line_ids)
```

computed fields are not stored by default, they are computed and returned when requested. Setting `store=True` will store them in the database and automatically enable searching.

searching on a computed field can also be enabled by setting the `search` parameter. The value is a method name returning a Search domains.

```
upper_name = field.Char(compute='_compute_upper', search='_search_upper')

def _search_upper(self, operator, value):
    if operator == 'like':
        operator = 'ilike'
    return [('name', operator, value)]
```

The search method is invoked when processing domains before doing an actual search on the model. It must return a domain equivalent to the condition: `field operator value` .

Computed fields are readonly by default. To allow *setting* values on a computed field, use the `inverse` parameter. It is the name of a function reversing the computation and setting the relevant fields:

```
document = fields.Char(compute='_get_document', inverse='_set_document')

def _get_document(self):
    for record in self:
        with open(record.get_document_path) as f:
            record.document = f.read()
def _set_document(self):
    for record in self:
        if not record.document: continue
        with open(record.get_document_path()) as f:
            f.write(record.document)
```

multiple fields can be computed at the same time by the same method, just use the same method on all fields and set all of them:

```
discount_value = fields.Float(compute='_apply_discount')
total = fields.Float(compute='_apply_discount')

@api.depends('value', 'discount')
def _apply_discount(self):
    for record in self:
        # compute actual discount from discount percentage
        discount = record.value * record.discount
        record.discount_value = discount
        record.total = record.value - discount
```

> ⚠ **Warning**
>
> While it is possible to use the same compute method for multiple fields, it is not recommended to do the same for the inverse method.
>
> During the computation of the inverse, **all** fields that use said inverse are protected, meaning that they can't be computed, even if their value is not in the cache.
>
> If any of those fields is accessed and its value is not in cache, the ORM will simply return a default value of `False` for these fields. This means that the value of the inverse fields (other than the one triggering the inverse method) may not give their correct value and this will probably break the expected behavior of the inverse method.

Related fields

A special case of computed fields are *related* (proxy) fields, which provide the value of a sub-field on the current record. They are defined by setting the `related` parameter and like regular computed fields they can be stored:

```
nickname = fields.Char(related='user_id.partner_id.name', store=True)
```

The value of a related field is given by following a sequence of relational fields and reading a field on the reached model. The complete sequence of fields to traverse is specified by the `related` attribute.

Some field attributes are automatically copied from the source field if they are not redefined: **`string`**, **`help`**, **`readonly`**, **`required`** (only if all fields in the sequence are required), **`groups`**, **`digits`**, **`size`**, **`translate`**, **`sanitize`**, **`selection`**, **`comodel_name`**, **`domain`**, **`context`**. All semantic-free attributes are copied from the source field.

By default, the values of related fields are not stored to the database. Add the attribute **`store=True`** to make it stored, just like computed fields. Related fields are automatically recomputed when their dependencies are modified.

> The related fields are computed in sudo mode.

> ### ⚠ Warning
>
> You cannot chain [`Many2many`] or [`One2many`] fields in **`related`** fields dependencies.
>
> **`related`** can be used to refer to a [`One2many`] or [`Many2many`] field on another model on the condition that it's done through a **`Many2one`** relation on the current model. **`One2many`** and **`Many2many`** are not supported and the results will not be aggregated correctly:
>
> ```
> m2o_id = fields.Many2one()
> m2m_ids = fields.Many2many()
> o2m_ids = fields.One2many()
>
> # Supported
> d_ids = fields.Many2many(related="m2o_id.m2m_ids")
> e_ids = fields.One2many(related="m2o_id.o2m_ids")
>
> # Won't work: use a custom Many2many computed field instead
> f_ids = fields.Many2many(related="m2m_ids.m2m_ids")
> g_ids = fields.One2many(related="o2m_ids.o2m_ids")
> ```

Automatic fields

`odoo.fields.id`

Identifier **`field`**

If length of current recordset is 1, return id of unique record in it.

Raise an Error otherwise.

Access Log fields

These fields are automatically set and updated if **`_log_access`** is enabled. It can be disabled to avoid creating or updating those fields on tables for which they are not useful.

By default, **`_log_access`** is set to the same value as **`_auto`**

`odoo.fields.create_date`

Stores when the record was created, **`Datetime`**

`odoo.fields.create_uid`

Stores *who* created the record, `Many2one` to a `res.users`.

`odoo.fields.write_date`

Stores when the record was last updated, `Datetime`

`odoo.fields.write_uid`

Stores who last updated the record, `Many2one` to a `res.users`.

> ⚠ **Warning**
>
> `_log_access` *must* be enabled on `TransientModel`.

Reserved Field names

A few field names are reserved for pre-defined behaviors beyond that of automated fields. They should be defined on a model when the related behavior is desired:

`odoo.fields.name`

default value for `_rec_name`, used to display records in context where a representative “naming” is necessary.

`Char`

`odoo.fields.active`

toggles the global visibility of the record, if `active` is set to `False` the record is invisible in most searches and listing.

`Boolean`

`odoo.fields.state`

lifecycle stages of the object, used by the `states` attribute on `fields`.

`Selection`

`odoo.fields.parent_id`

default_value of `_parent_name`, used to organize records in a tree structure and enables the `child_of` and `parent_of` operators in domains.

`Many2one`

`odoo.fields.parent_path`

When `_parent_store` is set to True, used to store a value reflecting the tree structure of `_parent_name`, and to optimize the operators `child_of` and `parent_of` in search domains. It must be declared with `index=True` for proper operation.

`Char`

`odoo.fields.company_id`

Main field name used for Odoo multi-company behavior.

Used by `:meth:~odoo.models._check_company` to check multi company consistency. Defines whether a record is shared between companies (no value) or only accessible by the users of a given company.

`Many2one` :type: `res_company`

Recordsets

Interactions with models and records are performed through recordsets, an ordered collection of records of the same model.

> **⚠ Warning**
>
> Contrary to what the name implies, it is currently possible for recordsets to contain duplicates. This may change in the future.

Methods defined on a model are executed on a recordset, and their `self` is a recordset:

```
class AModel(models.Model):
    _name = 'a.model'
    def a_method(self):
        # self can be anything between 0 records and all records in the
        # database
        self.do_operation()
```

Iterating on a recordset will yield new sets of *a single record* ("singletons"), much like iterating on a Python string yields strings of a single characters:

```
def do_operation(self):
    print(self) # => a.model(1, 2, 3, 4, 5)
    for record in self:
        print(record) # => a.model(1), then a.model(2), then a.model(3), ...
```

Field access

Recordsets provide an "Active Record" interface: model fields can be read and written directly from the record as attributes.

> When accessing non-relational fields on a recordset of potentially multiple records, use `mapped()`:

```
total_qty = sum(self.mapped('qty'))
```

Field values can also be accessed like dict items, which is more elegant and safer than `getattr()` for dynamic field names. Setting a field's value triggers an update to the database:

```
>>> record.name
Example Name
>>> record.company_id.name
Company Name
>>> record.name = "Bob"
>>> field = "name"
>>> record[field]
Bob
```

⚠ Warning

Trying to read a field on multiple records will raise an error for non relational fields.

Accessing a relational field (`Many2one` , `One2many` , `Many2many`) *always* returns a recordset, empty if the field is not set.

Record cache and prefetching

Odoo maintains a cache for the fields of the records, so that not every field access issues a database request, which would be terrible for performance. The following example queries the database only for the first statement:

```
record.name             # first access reads value from database
record.name             # second access gets value from cache
```

To avoid reading one field on one record at a time, Odoo *prefetches* records and fields following some heuristics to get good performance. Once a field must be read on a given record, the ORM actually reads that field on a larger recordset, and stores the returned values in cache for later use. The prefetched recordset is usually the recordset from which the record comes by iteration. Moreover, all simple stored fields (boolean, integer, float, char, text, date, datetime, selection, many2one) are fetched altogether; they correspond to the columns of the model's table, and are fetched efficiently in the same query.

Consider the following example, where `partners` is a recordset of 1000 records. Without prefetching, the loop would make 2000 queries to the database. With prefetching, only one query is made:

```
for partner in partners:
    print partner.name          # first pass prefetches 'name' and 'lang'
                                # (and other fields) on all 'partners'
    print partner.lang
```

The prefetching also works on *secondary records*: when relational fields are read, their values (which are records) are subscribed for future prefetching. Accessing one of those secondary records prefetches all secondary records from the same model. This makes the following example generate only two queries, one for partners and one for countries:

```
countries = set()
for partner in partners:
    country = partner.country_id        # first pass prefetches all partners
    countries.add(country.name)         # first pass prefetches all countries
```

Method decorators

The Odoo API module defines Odoo Environments and method decorators.

odoo.api.autovacuum(*method*) (https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L291)

Decorate a method so that it is called by the daily vacuum cron job (model `ir.autovacuum`). This is typically used for garbage-collection-like tasks that do not deserve a specific cron job.

odoo.api.constrains(args*)** (https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L100)

Decorate a constraint checker.

Each argument must be a field name used in the check:

```
@api.constrains('name', 'description')
def _check_description(self):
    for record in self:
        if record.name == record.description:
            raise ValidationError("Fields name and description must be different")
```

Invoked on the records on which one of the named fields has been modified.

Should raise `ValidationError` if the validation failed.

> ⚠ **Warning**
>
> `@constrains` only supports simple field names, dotted names (fields of relational fields e.g. `partner_id.customer`) are not supported and will be ignored.
>
> `@constrains` will be triggered only if the declared fields in the decorated method are included in the `create` or `write` call. It implies that fields not present in a view will not trigger a call during a record creation. A override of `create` is necessary to make sure a constraint will always be triggered (e.g. to test the absence of value).

odoo.api.depends(args*)** (https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L182)

Return a decorator that specifies the field dependencies of a “compute” method (for new-style function fields). Each argument must be a string that consists in a dot-separated sequence of field names:

```
pname = fields.Char(compute='_compute_pname')

@api.depends('partner_id.name', 'partner_id.is_company')
def _compute_pname(self):
    for record in self:
        if record.partner_id.is_company:
            record.pname = (record.partner_id.name or "").upper()
        else:
            record.pname = record.partner_id.name
```

One may also pass a single function as argument. In that case, the dependencies are given by calling the function with the field’s model.

odoo.api.depends_context(*args) (https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L207)

Return a decorator that specifies the context dependencies of a non-stored “compute” method. Each argument is a key in the context’s dictionary:

```
price = fields.Float(compute='_compute_product_price')

@api.depends_context('pricelist')
def _compute_product_price(self):
    for product in self:
        if product.env.context.get('pricelist'):
            pricelist = self.env['product.pricelist'].browse(product.env.context['pric
        else:
            pricelist = self.env['product.pricelist'].get_default_pricelist()
        product.price = pricelist.get_products_price(product).get(product.id, 0.0)
```

All dependencies must be hashable. The following keys have special support:

`company` (value in context or current company id),

`uid` (current user id and superuser flag),

`active_test` (value in env.context or value in field.context).

odoo.api.model(method) (https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L302)

Decorate a record-style method where `self` is a recordset, but its contents is not relevant, only the model is. Such a method:

```
@api.model
def method(self, args):
    ...
```

odoo.api.model_create_multi(*method*) [(https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L348)](https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L348)

Decorate a method that takes a list of dictionaries and creates multiple records. The method may be called with either a single dict or a list of dicts:

```
record = model.create(vals)
records = model.create([vals, ...])
```

odoo.api.onchange(args*) [(https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L133)](https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L133)**

Return a decorator to decorate an onchange method for given fields.

In the form views where the field appears, the method will be called when one of the given fields is modified. The method is invoked on a pseudo-record that contains the values present in the form. Field assignments on that record are automatically sent back to the client.

Each argument must be a field name:

```
@api.onchange('partner_id')
def _onchange_partner(self):
    self.message = "Dear %s" % (self.partner_id.name or "")
```

```
return {
    'warning': {'title': "Warning", 'message': "What is this?", 'type': 'notification'
}
```

If the type is set to notification, the warning will be displayed in a notification. Otherwise it will be displayed in a dialog as default.

> **⚠ Warning**
>
> `@onchange` only supports simple field names, dotted names (fields of relational fields e.g. `partner_id.tz`) are not supported and will be ignored

> **⚠ Danger**
>
> Since `@onchange` returns a recordset of pseudo-records, calling any one of the CRUD methods (`create()` , `read()` , `write()` , `unlink()`) on the aforementioned recordset is undefined behaviour, as they potentially do not exist in the database yet.
>
> Instead, simply set the record’s field like shown in the example above or call the `update()` method.

> **⚠ Warning**
>
> It is not possible for a `one2many` or `many2many` field to modify itself via onchange. This is a webclient limitation - see **[#2693 (https://github.com/odoo/odoo/issues/2693)](https://github.com/odoo/odoo/issues/2693)**.

odoo.api.returns(*model*, *downgrade=None*, *upgrade=None*) [(https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L233)](https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L233)

Return a decorator for methods that return instances of **`model`**.

Parameters:

- **`model`** – a model name, or **`'self'`** for the current model
- **`downgrade`** – a function **`downgrade(self, value, *args, **kwargs)`** to convert the record-style **`value`** to a traditional-style output
- **`upgrade`** – a function **`upgrade(self, value, *args, **kwargs)`** to convert the traditional-style **`value`** to a record-style output

The arguments **`self`**, **`*args`** and **`**kwargs`** are the ones passed to the method in the record-style.

The decorator adapts the method output to the api style: **`id`**, **`ids`** or **`False`** for the traditional style, and recordset for the record style:

```
@model
@returns('res.partner')
def find_partner(self, arg):
    ...     # return some record

# output depends on call style: traditional vs record style
partner_id = model.find_partner(cr, uid, arg, context=context)

# recs = model.browse(cr, uid, ids, context)
partner_record = recs.find_partner(arg)
```

Note that the decorated method must satisfy that convention.

Those decorators are automatically *inherited*: a method that overrides a decorated existing method will be decorated with the same **`@returns(model)`**.

Environment

The **`Environment`** stores various contextual data used by the ORM: the database cursor (for database queries), the current user (for access rights checking) and the current context (storing arbitrary metadata). The environment also stores caches.
All recordsets have an environment, which is immutable, can be accessed using **`env`** and gives access to:

- the current user (**`user`**)
- the cursor (**`cr`**)
- the superuser flag (**`su`**)
- or the context (**`context`**)

```
>>> records.env
<Environment object ...>
>>> records.env.user
res.user(3)
>>> records.env.cr
<Cursor object ...)
```

When creating a recordset from an other recordset, the environment is inherited. The environment can be used to get an empty recordset in an other model, and query that model:

```
>>> self.env['res.partner']
res.partner()
>>> self.env['res.partner'].search([['is_company', '=', True], ['customer', '=', True]])
res.partner(7, 18, 12, 14, 17, 19, 8, 31, 26, 16, 13, 20, 30, 22, 29, 15, 23, 28, 74)
```

`Environment.ref(`*`xml_id`*`, `*`raise_if_not_found=True`*`)` [(https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L509)](https://github.com/odoo/odoo/blob/14.0/odoo/api.py#L509)

Return the record corresponding to the given `xml_id` .

`Environment.lang`

Return the current language code.

Return type:

str (https://docs.python.org/3/library/stdtypes.html#str)

`Environment.user`

Return the current user (as an instance).

Return type:

`res_users`

`Environment.company`

Return the current company (as an instance).

If not specified in the context (`allowed_company_ids`), fallback on current user main company.

Raises:

AccessError – invalid or unauthorized `allowed_company_ids` context key content.

Returns:

current company (default=`self.user.company_id`)

Return type:

res.company

⚠ Warning

> No sanity checks applied in sudo mode ! When in sudo mode, a user can access any company, even if not in his allowed companies.
>
> This allows to trigger inter-company modifications, even if the current user doesn’t have access to the targeted company.

`Environment.companies`

Return a recordset of the enabled companies by the user.

If not specified in the context(**`allowed_company_ids`**), fallback on current user companies.

Raises:

AccessError – invalid or unauthorized **`allowed_company_ids`** context key content.

Returns:

current companies (default=`self.user.company_ids`)

Return type:

res.company

> ⚠ **Warning**
>
> No sanity checks applied in sudo mode ! When in sudo mode, a user can access any company, even if not in his allowed companies.
>
> This allows to trigger inter-company modifications, even if the current user doesn’t have access to the targeted company.

Altering the environment

`Model.with_context([context][, **overrides]) → records` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5077)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5077)

Returns a new version of this recordset attached to an extended context.

The extended context is either the provided **`context`** in which **`overrides`** are merged or the *current* context in which **`overrides`** are merged e.g.:

```
# current context is {'key1': True}
r2 = records.with_context({}, key2=True)
# -> r2._context is {'key2': True}
r2 = records.with_context(key2=True)
# -> r2._context is {'key1': True, 'key2': True}
```

`Model.with_user(user)` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5032)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5032)

Return a new version of this recordset attached to the given user, in non-superuser mode, unless **`user`** is the superuser (by convention, the superuser is always in superuser mode.)

`Model.with_company(company)` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5043)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5043)

Return a new version of this recordset with a modified context, such that:

```
result.env.company = company
result.env.companies = self.env.companies | company
```

Parameters:

company (**res_company** or int) – main company of the new environment.

> **⚠ Warning**
> When using an unauthorized company for current user, accessing the company(ies) on the environment may trigger an AccessError if not done in a sudoed environment.

Model.with_env(*env*) [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4986)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4986)

Return a new version of this recordset attached to the provided environment.

Parameters:

env (**Environment**) –

> **⚠ Warning**
> The new environment will not benefit from the current environment's data cache, so later data access may incur extra delays while re-fetching from the database. The returned recordset has the same prefetch object as `self` .

Model.sudo([*flag=True*]) [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5000)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5000)

Returns a new version of this recordset with superuser mode enabled or disabled, depending on `flag` . The superuser mode does not change the current user, and simply bypasses access rights checks.

> **⚠ Warning**
> Using `sudo` could cause data access to cross the boundaries of record rules, possibly mixing records that are meant to be isolated (e.g. records from different companies in multi-company environments).
>
> It may lead to un-intuitive results in methods which select one record among many - for example getting the default company, or selecting a Bill of Materials.

> Because the record rules and access control will have to be re-evaluated, the new recordset will not benefit from the current environment's data cache, so later data access may incur extra delays while re-fetching from the database. The returned recordset has the same prefetch object as `self` .

SQL Execution

The `cr` attribute on environments is the cursor for the current database transaction and allows executing SQL directly, either for queries which are difficult to express using the ORM (e.g. complex joins) or for performance reasons:

```
self.env.cr.execute("some_sql", param1, param2, param3)
```

Because models use the same cursor and the `Environment` holds various caches, these caches must be invalidated when *altering* the database in raw SQL, or further uses of models may become incoherent. It is necessary to clear caches when using `CREATE`, `UPDATE` or `DELETE` in SQL, but not `SELECT` (which simply reads the database).

> Clearing caches can be performed using the `invalidate_cache()` method.

`Model.invalidate_cache(`*`fnames=None, ids=None`*`)` (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5704)

Invalidate the record caches after some records have been modified. If both `fnames` and `ids` are `None`, the whole cache is cleared.

Parameters:

- `fnames` – the list of modified fields, or `None` for all fields
- `ids` – the list of modified record ids, or `None` for all

⚠ **Warning**

Executing raw SQL bypasses the ORM, and by consequent, Odoo security rules. Please make sure your queries are sanitized when using user input and prefer using ORM utilities if you don't really need to use SQL queries.

Common ORM methods

Create/update

`Model.create(`*`vals_list`*`) → records` (https://github.com/odoo/odoo/blob/14.0/<decorator-gen-128>#L3760)

Creates new records for the model.

The new records are initialized using the values from the list of dicts `vals_list`, and if necessary those from `default_get()`.

Parameters:

vals_list (`list` (https://docs.python.org/3/library/stdtypes.html#list)) –

values for the model's fields, as a list of dictionaries:

```
[{'field_name': field_value, ...}, ...]
```

For backward compatibility, `vals_list` may be a dictionary. It is treated as a singleton list `[vals]`, and a single record is returned.

see `write()` for details

Returns:

the created records

Raises:

`AccessError` –

if user has no create rights on the requested object

if user tries to bypass access rules for create on the requested object

`ValidationError` – if user tries to enter invalid value for a field that is not in selection

`UserError` – if a loop would be created in a hierarchy of objects a result of the operation (such as setting an object as its own parent)

`Model.copy(`*`default=None`*`)` **(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4639)**

Duplicate record `self` updating it with default values

Parameters:

default (`dict` (https://docs.python.org/3/library/stdtypes.html#dict)) – dictionary of field values to override in the original values of the copied record, e.g: `{'field_name': overridden_value, ...}`

Returns:

new record

`Model.default_get(`*`fields_list`*`) → default_values` **(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1253)**

Return default values for the fields in `fields_list` . Default values are determined by the context, user defaults, and the model itself.

Parameters:

fields_list (`list` (https://docs.python.org/3/library/stdtypes.html#list)) – names of field whose default is requested

Returns:

a dictionary mapping field names to their corresponding default values, if they have a default value.

Return type:

dict (https://docs.python.org/3/library/stdtypes.html#dict)

> Unrequested defaults won’t be considered, there is no need to return a value for fields whose names are not in `fields_list` .

`Model.name_create(`*`name`*`) → record` **(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1744)**

Create a new record by calling `create()` with only one value provided: the display name of the new record.

The new record will be initialized with any default values applicable to this model, or provided through the context. The usual behavior of `create()` applies.

Parameters:

name – display name of the record to create

Return type:

tuple (https://docs.python.org/3/library/stdtypes.html#tuple)

Returns:

the `name_get()` pair value of the created record

`Model.write(vals)` (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L3493)

Updates all records in the current set with the provided values.

Parameters:

vals (`dict` (https://docs.python.org/3/library/stdtypes.html#dict)) –

fields to update and the value to set on them e.g:

```
{'foo': 1, 'bar': "Qux"}
```

will set the field `foo` to `1` and the field `bar` to `"Qux"` if those are valid (otherwise it will trigger an error).

Raises:

`AccessError` –
if user has no write rights on the requested object

if user tries to bypass access rules for write on the requested object

`ValidationError` – if user tries to enter invalid value for a field that is not in selection

`UserError` – if a loop would be created in a hierarchy of objects a result of the operation (such as setting an object as its own parent)

For numeric fields (`Integer` , `Float`) the value should be of the corresponding type

For `Boolean` , the value should be a `bool` (https://docs.python.org/3/library/functions.html#bool)

For `Selection` , the value should match the selection values (generally `str` (https://docs.python.org/3/library/stdtypes.html#str), sometimes `int` (https://docs.python.org/3/library/functions.html#int))

For `Many2one` , the value should be the database identifier of the record to set

Other non-relational fields use a string for value

> **⚠ Danger**
>
> for historical and compatibility reasons, `Date` and `Datetime` fields use strings as values (written and read) rather than `date` **(https://docs.python.org/3/library/datetime.html#datetime.date)** or `datetime` **(https://docs.python.org/3/library/datetime.html#datetime.datetime)**. These date strings are UTC-only and formatted according to `odoo.tools.misc.DEFAULT_SERVER_DATE_FORMAT` and `odoo.tools.misc.DEFAULT_SERVER_DATETIME_FORMAT`

`One2many` and **`Many2many`** use a special "commands" format to manipulate the set of records stored in/associated with the field.

This format is a list of triplets executed sequentially, where each triplet is a command to execute on the set of records. Not all commands apply in all situations. Possible commands are:

`(0, 0, values)`

adds a new record created from the provided `value` dict.

`(1, id, values)`

updates an existing record of id `id` with the values in `values` . Can not be used in `create()` .

`(2, id, 0)`

removes the record of id `id` from the set, then deletes it (from the database). Can not be used in `create()` .

`(3, id, 0)`

removes the record of id `id` from the set, but does not delete it. Can not be used in `create()` .

`(4, id, 0)`

adds an existing record of id `id` to the set.

`(5, 0, 0)`

removes all records from the set, equivalent to using the command `3` on every record explicitly. Can not be used in `create()` .

`(6, 0, ids)`

replaces all existing records in the set by the `ids` list, equivalent to using the command `5` followed by a command `4` for each `id` in `ids` .

`Model.flush(`*`fnames=None`*`,`*`records=None`*`)`
(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5404)

Process all the pending computations (on all models), and flush all the pending updates to the database.

Parameters:

- **`(list<str>)`** (**`fnames`**) – list of field names to flush. If given, limit the processing to the given fields of the current model.
- **`(Model)`** (**`records`**) – if given (together with **`fnames`**), limit the processing to the given records.

Search/Read

`Model.browse([`*`ids`*`]) → records`
(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4932)

Returns a recordset for the ids provided as parameter in the current environment.

```
self.browse([7, 18, 12])
res.partner(7, 18, 12)
```

Parameters:

ids (**int** (https://docs.python.org/3/library/functions.html#int) **or** **list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **int** (https://docs.python.org/3/library/functions.html#int) **)** **or** **None** (https://docs.python.org/3/library/constants.html#None)) – id(s)

Returns:

recordset

Model.search(*args[, offset=0][, limit=None][, order=None][, count=False]*) (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1679)

Searches for records based on the `args` search domain.

Parameters:

args – A search domain. Use an empty list to match all records.

offset (**int** (https://docs.python.org/3/library/functions.html#int)) – number of results to ignore (default: none)

limit (**int** (https://docs.python.org/3/library/functions.html#int)) – maximum number of records to return (default: all)

order (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – sort string

count (**bool** (https://docs.python.org/3/library/functions.html#bool)) – if True, only counts and returns the number of matching records (default: False)

Returns:

at most `limit` records matching the search criteria

Raises:

AccessError –

if user tries to bypass access rules for read on the requested object.

Model.search_count(*args*) → int (https://docs.python.org/3/library/functions.html#int) (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1669)

Returns the number of records in the current model matching the provided domain.

Model.name_search(*name=''*, *args=None*, *operator='ilike'*, *limit=100*) → records (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1766)

Search for records that have a display name matching the given `name` pattern when compared with the given `operator`, while also matching the optional search domain (`args`).

This is used for example to provide suggestions based on a partial value for a relational field. Sometimes be seen as the inverse function of `name_get()`, but it is not guaranteed to be.

This method is equivalent to calling `search()` with a search domain based on `display_name` and then `name_get()` on the result of the search.

Parameters:

- **`name`** (`str` (https://docs.python.org/3/library/stdtypes.html#str)) – the name pattern to match
- **`args`** (`list` (https://docs.python.org/3/library/stdtypes.html#list)) – optional search domain (see `search()` for syntax), specifying further restrictions
- **`operator`** (`str` (https://docs.python.org/3/library/stdtypes.html#str)) – domain operator for matching `name`, such as `'like'` or `'='`.
- **`limit`** (`int` (https://docs.python.org/3/library/functions.html#int)) – optional max number of records to return

Return type:

list (https://docs.python.org/3/library/stdtypes.html#list)

Returns:

list of pairs `(id, text_repr)` for all matching records.

`Model.read([`*`fields`*`])` (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L2988)

Reads the requested fields for the records in `self`, low-level/RPC method. In Python code, prefer `browse()`.

Parameters:

fields – list of field names to return (default is all fields)

Returns:

a list of dictionaries mapping field names to their values, with one dictionary per record

Raises:

AccessError – if user has no read rights on some of the given records

`Model.read_group(`*`domain, fields, groupby, offset=0, limit=None, orderby=False, lazy=True`*`)` (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L2209)

Get the list of records in list view grouped by the given `groupby` fields.

Parameters:

- **`domain`** (`list` (https://docs.python.org/3/library/stdtypes.html#list)) – A search domain. Use an empty list to match all records.
- **`fields`** (`list` (https://docs.python.org/3/library/stdtypes.html#list)) – list of fields present in the list view specified on the object. Each element is either ‘field’ (field name, using the default aggregation), or ‘field:agg’ (aggregate field with aggregation function ‘agg’), or ‘name:agg(field)’ (aggregate field with ‘agg’ and return it as ‘name’). The possible aggregation functions are the ones provided by PostgreSQL (https://www.postgresql.org/docs/current/static/functions-aggregate.html) and ‘count_distinct’, with the expected meaning.

groupby (**list** (https://docs.python.org/3/library/stdtypes.html#list)) – list of groupby descriptions by which the records will be grouped. A groupby description is either a field (then it will be grouped by that field) or a string 'field:groupby_function'. Right now, the only functions supported are 'day', 'week', 'month', 'quarter' or 'year', and they only make sense for date/datetime fields.

offset (**int** (https://docs.python.org/3/library/functions.html#int)) – optional number of records to skip

limit (**int** (https://docs.python.org/3/library/functions.html#int)) – optional max number of records to return

orderby (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – optional `order by` specification, for overriding the natural sort ordering of the groups, see also `search()` (supported only for many2one fields currently)

lazy (**bool** (https://docs.python.org/3/library/functions.html#bool)) – if true, the results are only grouped by the first groupby and the remaining groupbys are put in the __context key. If false, all the groupbys are done in one call.

Returns:

list of dictionaries(one dictionary for each record) containing:

the values of fields grouped by the fields in `groupby` argument

__domain: list of tuples specifying the search criteria

__context: dictionary with argument like `groupby`

Return type:

[{'field_name_1': value, ..]

Raises:

AccessError –

if user has no read rights on the requested object

if user tries to bypass access rules for read on the requested object

Fields/Views

`Model.fields_get([fields][, attributes])` (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L2863)

Return the definition of each field.

The returned value is a dictionary (indexed by field name) of dictionaries. The _inherits'd fields are included. The string, help, and selection (if present) attributes are translated.

Parameters:

allfields – list of fields to document, all if empty or not provided

attributes – list of description attributes to return for each field, all if empty or not provided

`Model.fields_view_get([view_id | view_type='form'])` (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1581)

Get the detailed composition of the requested view like fields, model, view architecture

Parameters:

- **`view_id`** (**`int`** (https://docs.python.org/3/library/functions.html#int)) – id of the view or None
- **`view_type`** (**`str`** (https://docs.python.org/3/library/stdtypes.html#str)) – type of the view to return if view_id is None ('form', 'tree', …)
- **`toolbar`** (**`bool`** (https://docs.python.org/3/library/functions.html#bool)) – true to include contextual actions
- **`submenu`** – deprecated

Returns:

composition of the requested view (including inherited views and extensions)

Return type:

dict (https://docs.python.org/3/library/stdtypes.html#dict)

Raises:

`AttributeError` (https://docs.python.org/3/library/exceptions.html#AttributeError) –

if the inherited view has unknown position to work with other than 'before', 'after', 'inside', 'replace'

if some tag other than 'position' is found in parent view

`Invalid ArchitectureError` – if there is view type other than form, tree, calendar, search etc defined on the structure

Search domains

A domain is a list of criteria, each criterion being a triple (either a **`list`** or a **`tuple`**) of **`(field_name, operator, value)`** where:

`field_name` (**`str`**)

a field name of the current model, or a relationship traversal through a **`Many2one`** using dot-notation e.g. **`'street'`** or **`'partner_id.country'`**

`operator` (**`str`**)

an operator used to compare the **`field_name`** with the **`value`** . Valid operators are:

`=`

equals to

`!=`

not equals to

`>`

greater than

`>=`

greater than or equal to

`<`

less than

`<=`

less than or equal to

`=?`
unset or equals to (returns true if `value` is either `None` or `False`, otherwise behaves like `=`)

`=like`
matches `field_name` against the `value` pattern. An underscore `_` in the pattern stands for (matches) any single character; a percent sign `%` matches any string of zero or more characters.

`like`
matches `field_name` against the `%value%` pattern. Similar to `=like` but wraps `value` with '%' before matching

`not like`
doesn't match against the `%value%` pattern

`ilike`
case insensitive `like`

`not ilike`
case insensitive `not like`

`=ilike`
case insensitive `=like`

`in`
is equal to any of the items from `value`, `value` should be a list of items

`not in`
is unequal to all of the items from `value`

`child_of`
is a child (descendant) of a `value` record (value can be either one item or a list of items).
Takes the semantics of the model into account (i.e following the relationship field named by `_parent_name`).

`parent_of`
is a parent (ascendant) of a `value` record (value can be either one item or a list of items).
Takes the semantics of the model into account (i.e following the relationship field named by `_parent_name`).

`value`
variable type, must be comparable (through `operator`) to the named field.

Domain criteria can be combined using logical operators in *prefix* form:

'&'

logical *AND*, default operation to combine criteria following one another. Arity 2 (uses the next 2 criteria or combinations).

'|'

logical *OR*, arity 2.

'!'

logical *NOT*, arity 1.

> Mostly to negate combinations of criteria Individual criterion generally have a negative form (e.g. `=` -> `!=`, `<` -> `>=`) which is simpler than negating the positive.

> ### ⓘ Example
>
> To search for partners named *ABC*, from belgium or germany, whose language is not english:
>
> ```
> [('name','=','ABC'),
>  ('language.code','!=','en_US'),
>  '|',('country_id.code','=','be'),
>      ('country_id.code','=','de')]
> ```
>
> This domain is interpreted as:
>
> ```
>     (name is 'ABC')
> AND (language is NOT english)
> AND (country is Belgium OR Germany)
> ```

Unlink

`Model.unlink()` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L3401)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L3401)

Deletes the records of the current set

Raises:

- **`AccessError`** –
  if user has no unlink rights on the requested object

  if user tries to bypass access rules for unlink on the requested object
- **`UserError`** – if the record is default property for other records

Record(set) information

`Model.ids`

Return the list of actual record ids corresponding to `self`.

`odoo.models.env`

Returns the environment of the given recordset.

Type:

`Environment`

`Model.exists() → records` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4657)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4657)

Returns the subset of records in `self` that exist, and marks deleted records as such in cache. It can be used as a test on records:

```
if record.exists():
    ...
```

By convention, new records are returned as existing.

`Model.ensure_one()` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4973)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L4973)

Verify that the current recorset holds a single record.

Raises:

odoo.exceptions.ValueError – `len(self) != 1`

`Model.name_get() → [id, name, …]` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1723)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L1723)

Returns a textual representation for the records in `self`. By default this is the value of the `display_name` field.

Returns:

list of pairs `(id, text_repr)` for each records

Return type:

[list (https://docs.python.org/3/library/stdtypes.html#list)](https://docs.python.org/3/library/stdtypes.html#list)([tuple (https://docs.python.org/3/library/stdtypes.html#tuple)](https://docs.python.org/3/library/stdtypes.html#tuple))

`Model.get_metadata()` [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L3180)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L3180)

Return some metadata about the given records.

Returns:

list of ownership dictionaries for each requested record

Return type:

list of dictionaries with the following keys:

id: object id

create_uid: user who created the record

create_date: date when the record was created write_uid: last user who changed the record

write_date: date of the last change to the record

xmlid: XML ID to use to refer to this record (if there is one), in format **`module.name`**

noupdate: A boolean telling if the record will be updated or not

Operations

Recordsets are immutable, but sets of the same model can be combined using various set operations, returning new recordsets.

- **`record in set`** returns whether **`record`** (which must be a 1-element recordset) is present in **`set`**. **`record not in set`** is the inverse operation
- **`set1 <= set2`** and **`set1 < set2`** return whether **`set1`** is a subset of **`set2`** (resp. strict)
- **`set1 >= set2`** and **`set1 > set2`** return whether **`set1`** is a superset of **`set2`** (resp. strict)
- **`set1 | set2`** returns the union of the two recordsets, a new recordset containing all records present in either source
- **`set1 & set2`** returns the intersection of two recordsets, a new recordset containing only records present in both sources
- **`set1 - set2`** returns a new recordset containing only records of **`set1`** which are *not* in **`set2`**

Recordsets are iterable so the usual Python tools are available for transformation (**`map()`** (https://docs.python.org/3/library/functions.html#map), **`sorted()`** (https://docs.python.org/3/library/functions.html#sorted), **`itertools.ifilter`** , ...) however these return either a **`list`** (https://docs.python.org/3/library/stdtypes.html#list) or an iterator (https://docs.python.org/3/glossary.html#term-iterator), removing the ability to call methods on their result, or to use set operations.

Recordsets therefore provide the following operations returning recordsets themselves (when possible):

Filter

`Model.filtered(`*`func`*`)` **(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5254)**

Return the records in **`self`** satisfying **`func`** .

Parameters:

func (**`callable`** **`or`** **`str`** (https://docs.python.org/3/library/stdtypes.html#str)) – a function or a dot-separated sequence of field names

Returns:

recordset of records satisfying func, may be empty.

```
# only keep records whose company is the current user's
records.filtered(lambda r: r.company_id == user.company_id)

# only keep records whose partner is a company
records.filtered("partner_id.is_company")
```

Model.filtered_domain(*domain*) [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5276)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5276)

Map

Model.mapped(*func*) [(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5202)](https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5202)

Apply `func` on all records in `self`, and return the result as a list or a recordset (if `func` return recordsets). In the latter case, the order of the returned recordset is arbitrary.

Parameters:

func (`callable` or `str` [(https://docs.python.org/3/library/stdtypes.html#str)](https://docs.python.org/3/library/stdtypes.html#str)) – a function or a dot-separated sequence of field names

Returns:

self if func is falsy, result of func applied to all `self` records.

Return type:

[list (https://docs.python.org/3/library/stdtypes.html#list)](https://docs.python.org/3/library/stdtypes.html#list) or recordset

```
# returns a list of summing two fields for each record in the set
records.mapped(lambda r: r.field1 + r.field2)
```

The provided function can be a string to get field values:

```
# returns a list of names
records.mapped('name')

# returns a recordset of partners
records.mapped('partner_id')

# returns the union of all partner banks, with duplicates removed
records.mapped('partner_id.bank_ids')
```

> Since V13, multi-relational field access is supported and works like a mapped call:
>
> ```
> records.partner_id  # == records.mapped('partner_id')
> records.partner_id.bank_ids  # == records.mapped('partner_id.bank_ids')
> records.partner_id.mapped('name')  # == records.mapped('partner_id.name')
> ```

Sort

`Model.sorted(`*`key=None`*`, `*`reverse=False`*`)` **(https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L5377)**

Return the recordset **`self`** ordered by **`key`** .

Parameters:

key (**`callable`** **`or`** **`str`** (https://docs.python.org/3/library/stdtypes.html#str) **`or`** **`None`** (https://docs.python.org/3/library/constants.html#None)) – either a function of one argument that returns a comparison key for each record, or a field name, or **`None`** , in which case records are ordered according the default model's order

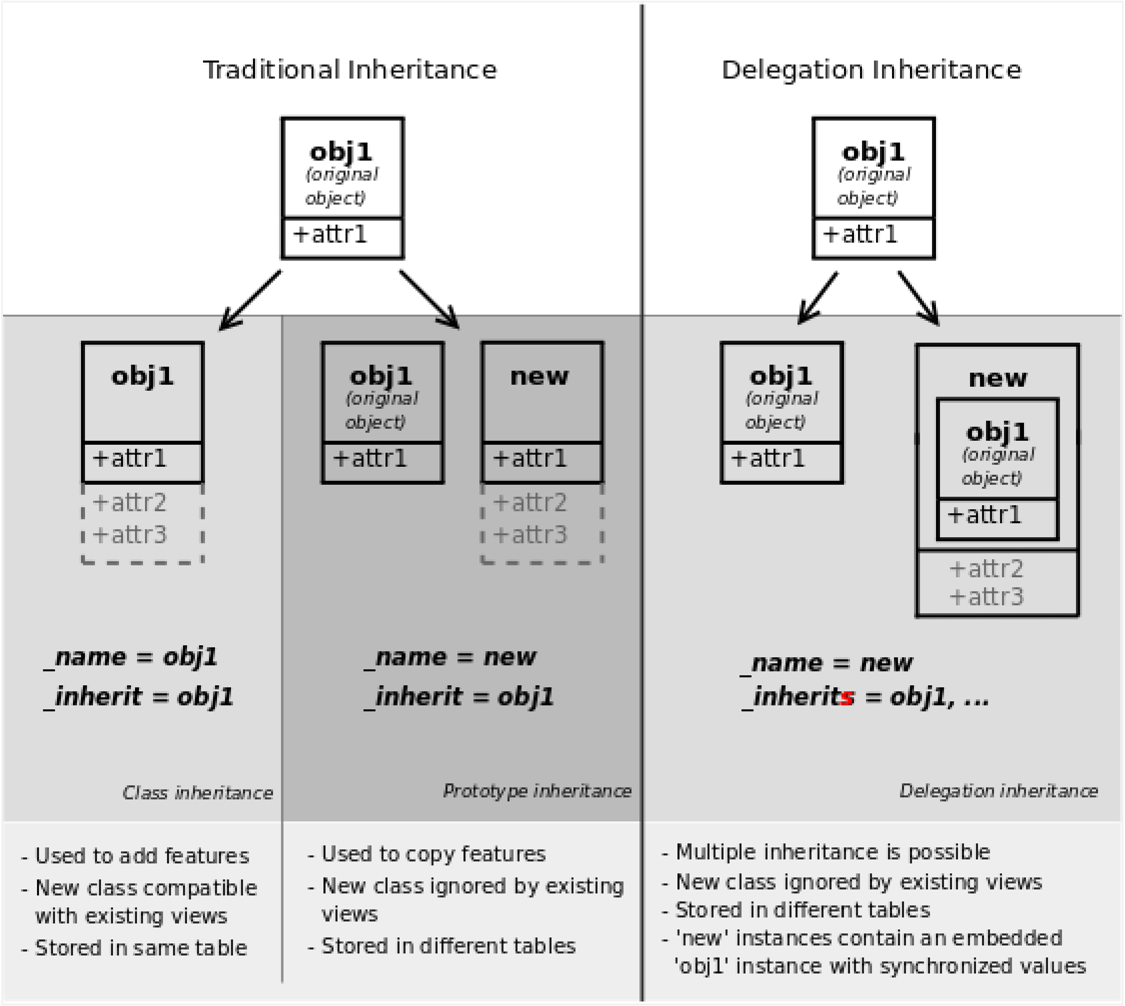
reverse (**`bool`** (https://docs.python.org/3/library/functions.html#bool)) – if **`True`** , return the result in reverse order

```
# sort records by name
records.sorted(key=lambda r: r.name)
```

Inheritance and extension

Odoo provides three different mechanisms to extend models in a modular way:

- creating a new model from an existing one, adding new information to the copy but leaving the original module as-is
- extending models defined in other modules in-place, replacing the previous version
- delegating some of the model's fields to records it contains



Classical inheritance

When using the `_inherit` and `_name` attributes together, Odoo creates a new model using the existing one (provided via `_inherit`) as a base. The new model gets all the fields, methods and meta-information (defaults & al) from its base.

```
class Inheritance0(models.Model):
    _name = 'inheritance.0'
    _description = 'Inheritance Zero'

    name = fields.Char()

    def call(self):
        return self.check("model 0")

    def check(self, s):
        return "This is {} record {}".format(s, self.name)

class Inheritance1(models.Model):
    _name = 'inheritance.1'
    _inherit = 'inheritance.0'
    _description = 'Inheritance One'

    def call(self):
        return self.check("model 1")
```

and using them:

```
a = env['inheritance.0'].create({'name': 'A'})
b = env['inheritance.1'].create({'name': 'B'})
    a.call()
    b.call()
```

will yield:

```
"This is model 0 record A"
"This is model 1 record B"
```

the second model has inherited from the first model’s `check` method and its `name` field, but overridden the `call` method, as when using standard Python inheritance (https://docs.python.org/3/tutorial/classes.html#tut-inheritance).

Extension

When using `_inherit` but leaving out `_name`, the new model replaces the existing one, essentially extending it in-place. This is useful to add new fields or methods to existing models (created in other modules), or to customize or reconfigure them (e.g. to change their default sort order):

```
class Extension0(models.Model):
    _name = 'extension.0'
    _description = 'Extension zero'

    name = fields.Char(default="A")

class Extension1(models.Model):
    _inherit = 'extension.0'

    description = fields.Char(default="Extended")
```

```
record = env['extension.0'].create({})
record.read()[0]
```

will yield:

```
{'name': "A", 'description': "Extended"}
```

> It will also yield the various **automatic fields** unless they’ve been disabled

Delegation

The third inheritance mechanism provides more flexibility (it can be altered at runtime) but less power: using the `_inherits` a model *delegates* the lookup of any field not found on the current model to “children” models. The delegation is performed via **`Reference`** fields automatically set up on the parent model.

The main difference is in the meaning. When using Delegation, the model **has one** instead of **is one**, turning the relationship in a composition instead of inheritance:

```
class Screen(models.Model):
    _name = 'delegation.screen'
    _description = 'Screen'

    size = fields.Float(string='Screen Size in inches')

class Keyboard(models.Model):
    _name = 'delegation.keyboard'
    _description = 'Keyboard'

    layout = fields.Char(string='Layout')

class Laptop(models.Model):
    _name = 'delegation.laptop'
    _description = 'Laptop'

    _inherits = {
        'delegation.screen': 'screen_id',
        'delegation.keyboard': 'keyboard_id',
    }

    name = fields.Char(string='Name')
    maker = fields.Char(string='Maker')

    # a Laptop has a screen
    screen_id = fields.Many2one('delegation.screen', required=True, ondelete="cascade")
    # a Laptop has a keyboard
    keyboard_id = fields.Many2one('delegation.keyboard', required=True, ondelete="cascad
```

```
record = env['delegation.laptop'].create({
    'screen_id': env['delegation.screen'].create({'size': 13.0}).id,
    'keyboard_id': env['delegation.keyboard'].create({'layout': 'QWERTY'}).id,
})
    record.size
    record.layout
```

will result in:

```
13.0
'QWERTY'
```

and it's possible to write directly on the delegated field:

```
record.write({'size': 14.0})
```

⚠ Warning

when using delegation inheritance, methods are *not* inherited, only fields

⚠ Warning

> `_inherits` is more or less implemented, avoid it if you can;
>
> chained `_inherits` is essentially not implemented, we cannot guarantee anything on the final behavior.

Fields Incremental Definition

A field is defined as class attribute on a model class. If the model is extended, one can also extend the field definition by redefining a field with the same name and same type on the subclass. In that case, the attributes of the field are taken from the parent class and overridden by the ones given in subclasses.

For instance, the second class below only adds a tooltip on the field `state`:

```
class First(models.Model):
    _name = 'foo'
    state = fields.Selection([...], required=True)

class Second(models.Model):
    _inherit = 'foo'
    state = fields.Selection(help="Blah blah blah")
```

Error management

The Odoo Exceptions module defines a few core exception types.

Those types are understood by the RPC layer. Any other exception type bubbling until the RPC layer will be treated as a 'Server error'.

> If you consider introducing new exceptions, check out the `odoo.addons.test_exceptions` module.

exception **odoo.exceptions.AccessDenied(*message='Access Denied'*)[source] (https://github.com/odoo/odoo/blob/14.0/odoo/exceptions.py#L65)**

Login/password error.

> No traceback.
>
> **ⓘ Example**
>
> When you try to log with a wrong password.

exception **odoo.exceptions.AccessError(*message*)[source] (https://github.com/odoo/odoo/blob/14.0/odoo/exceptions.py#L84)**

Access rights error.

> **ⓘ Example**
>
> When you try to read a record that you are not allowed to.

exception **odoo.exceptions.CacheMiss(*record*, *field*)[source] (https://github.com/odoo/odoo/blob/14.0/odoo/exceptions.py#L93)**

Missing value(s) in cache.

> **ⓘ Example**
>
> When you try to read a value in a flushed cache.

exception **odoo.exceptions.MissingError(*message*)[source] (https://github.com/odoo/odoo/blob/14.0/odoo/exceptions.py#L105)**

Missing record(s).

> **ⓘ Example**
>
> When you try to write on a deleted record.

exception **odoo.exceptions.RedirectWarning(*message*, *action*, *button_text*, *additional_context=None*)[source] (https://github.com/odoo/odoo/blob/14.0/odoo/exceptions.py#L42)**

Warning with a possibility to redirect the user instead of simply displaying the warning message.

Parameters:

- **message** (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – exception message and frontend modal content
- **action_id** (**int** (https://docs.python.org/3/library/functions.html#int)) – id of the action where to perform the redirection
- **button_text** (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – text to put on the button that will trigger the redirection.
- **additional_context** (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – parameter passed to action_id. Can be used to limit a view to active_ids for example.

exception **odoo.exceptions.UserError(*message*)[source] (https://github.com/odoo/odoo/blob/14.0/odoo/exceptions.py#L21)**

Generic error managed by the client.

Typically when the user tries to do something that has no sense given the current state of a record. Semantically comparable to the generic 400 HTTP status codes.

exception **odoo.exceptions.ValidationError(*message*)[source] (https://github.com/odoo/odoo/blob/14.0/odoo/exceptions.py#L114)**

Violation of python constraints.

> **ⓘ Example**
>
> When you try to create a new user with a login which already exist in the db.