odoo Technical Memento v9.0 (Rev 1.0)

STOP / START SERVER

Sebelum lebih dalam mempalajari odoo, sebaiknya kita mengetahui cara untuk **stop**, **start** atau **restart** Server odoo. Hal ini dibutuhkan karena setiap kali kita melakukan perubahan code (.py) membutuhkan restart server odoo

- Windows
  - Windows Service > odoo > restart
- Linux
  - service odoo restart
- Mac OS
  -
- IDE : Stop/Run atau Rerun

Update Apps List , Install, Upgrade & Uninstall

Update Apps List, dilakukan apabila kita menambahkan folder baru pada addons (menambahkan modul baru)
Install, untuk install module.
Upgrade, untuk melakkan upgrade module yang telah terinstall, Upgrade dibutuhkan apabila ada perubahan pada module.
Uninstall, untuk uninstall suatu module.

Membuat modul odoo : idea

idea : contoh dalam materi ini adalah membuat module yang diambil dari kasus sederhana tetang “IDE” (gagasan/pemikiran). Setiap IDE dari suatu member akan disimpan, meliputi beberapa data : **Judul, deskripsi, tanggal dll**, setiap IDE harus mendapat **persetujuan** terlebih dahulu agar bisa dilakukan **pemungutan suara** (vote), setiap IDE memiliki **skor** yang didapat dari vote.

Note: **Modular development/Technology**
Odoo mengunakan modul sebagai kontainer dari suatu aplikasi atau fitur untuk memudahkan maintain code dan pengembangan aplikasi. Di dalam sebuah modul diterapkan pola MVC (Model View Controller pattern) yang sangat kuat dan handal.

Apa saja yang ada di dalam sebuah MODUL

Sebuah modul dapat terdiri dari beberapa elemen, seperti dibawah ini :

- **business objects** : Dibuat dengan Python classes yang diturunkan dari kelas **models.Model, models.TransientModel, models.AbstractModel** atau

lainnya, semua bisnis objek yang terbentuk akan menjadi sumber-daya bagi Odoo yang akan dikelola secara teritegrasi, lengkap dan handal.

- **Data files** : berupa file XML, CSV, atau YML sebagai meta-data untuk :
  - menus
  - views
  - intial data
  - configuration data (modules parametrization)
  - workflows declaration, and
  - demo data (optional but recommended for testing, e.g. sample ide)
- **wizards** : stateful interactive forms used to assist users, often available as contextual actions on resources ;
- **reports** : QWEB, RML (XML format), MAKO or OpenOffice report templates, to be merged with any kind of business data, and generate HTML, ODT or PDF reports.
- **Script / static** : Javascript, CSS, Less, sass, img, dll
- **Web controllers**

  Handle requests from web browsers

Struktur MODUL

Setiap modul memiliki satu direktori yang diletakan pada direktori /..../addons atau direktori addons yang didefinisikan sendiri melalui konfigurasi file (openerp-server.conf)

Note: You can declare your own addons directory in the configuration file of Odoo (passed to the server with the -c option) using the addons_path option.

Petunjuk penamaan directory, file, variable :
https://www.odoo.com/documentation/9.0/reference/guidelines.html

```
addons/
        |- idea/                # Folder dari modul (required)
            |- models/          # Models , python code
            |- views/           # Views (forms,lists), menus and actions
            |- static/          # Static definitions
            |- data/            # Data definitions
            |- demo/            # Demo and unit test population data
            |- i18n/            # Translation files
            |- report/          # Report definitions
            |- security/        # Declaration of groups and access rights
            |- wizard/          # Wizards definitions
            |- workflow/        # Workflow definitions
        |- __init__.py          # Python package initialization (required)
        |- __openerp__.py       # module declaration / manifest file (required)
```

file **__init__.py** merupakan Python code untuk constructor/inisiator suatu modul.

__init__.py :

```
# Import semua file dan direktori yang mengandung python code
import idea
```

file **__openerp__.py** (__manifest__.py) merupakan manifest file untuk suatu modul, file ini terbuat dari sebuah Python dictionary untuk mendefinisikan isi dari suatu modul, seperti : name, dependencies, description, dan composition.

__openerp__.py (__manifest__.py):

```
{
   'name' : 'Idea',
   'version' : '1.0',
   'author' : 'Edaptec',
    'summar' : 'Edaptec',
   'description' : 'Ideas management module',
   'category': 'Latihan',
   'website': 'https://www.edaptec.com',
   'depends' : ['base'],   # list of dependencies, conditioning startup order
   'data' :[                       # data files to load at module install
     'security/groups.xml',      # always load groups first!
```

```
    'security/ir.model.access.csv', # load access rights after groups
    'view/views.xml',
    'wizard/wizard.xml',
    'report/report.xml',
  ],
  'demo': ['demo/demo.xml'],     # demo data (for unit tests)
 'installable': True,
 'auto_install': False,          # indikasi install, saat buat database baru
}
```

Silahkan membuat suatu direktori untuk modul IDE :
- Buat direktori : idea
- Buat direktori-direktori didalam idea : models, views, data, security
- Buat Inisiator file & manifest file, berserta code/isi nya.

Membuat MODEL

Model dideklarasi sebagai kelas Python yang diturunkan dari Kelas models.Model yang merupakan salah satu kelas yang dikembangkan dengan konsep ORM (thanks psycopg) yang sangat handal dan bersahabat bagi programer sehinga hampir tidak memerlukan penulisan SQL.

idea.py :

```
from odoo import models, fields, api, _
class idea(models.Model):
   _name = 'idea.idea'
   name = fields.Char('Title', size=64, required=True, translate=True)
   date = fields.Date('Date Release')
   state = fields.Selection([('draft', 'Draft'),
                       ( 'confirmed', 'Confirmed')], 'State', required=True, readonly=True,
                      default='draft')
   # Description is read-only when not draft!
   description = fields.Text('Description', states={'draft': [('readonly', False)]})
   active = fields.Boolean('Active', default=True)
   confirm_date = fields.Date('Confirm date')
   # by convention, many2one fields end with '_id'
   confirm_partner_id = fields.Many2one('res.partner', 'Confirm By')
   sponsor_ids = fields.Many2many('res.partner', 'idea_sponsor_rel', 'idea_id', 'sponsor_id',
'Sponsors')
   score = fields.Integer('Score', default=0, readonly=True)
   owner = fields.Many2one('res.partner', 'Owner', index=True)
   _sql_constraints = [('name_unik', 'unique(name)', _('Ideas must be unique!'))]
```

Silahkan membuat satu file python di dalam folder /models/idea.py.

Membuat VIEW & MENU

Buat sebuah file **'.../views/idea_view.xml'** untuk mendefinisikan sebuah VIEW, seperti contoh dibawah ini :

```
<odoo>
```

FORM VIEW

```
    <record id="idea_view_form" model="ir.ui.view">
        <field name="name">idea.view.form</field>
        <field name="model">idea.idea</field>
        <field name="arch" type="xml">
            <form string="Idea form">
                <sheet>
                    <group>
                        <group>
                            <field name="name"/>
                            <field name="date"/>
                            <field name="score"/>
                        </group>
                        <group>
                            <field name="active"/>
                        </group>
                    </group>
                </sheet>
            </form>
        </field>
    </record>
```

LIST VIEW

```
<record id="idea_view_list" model="ir.ui.view">
    <field name="name">idea.view.list</field>
    <field name="model">idea.idea</field>
    <field name="arch" type="xml">
        <tree>
            <field name="name"/>
            <field name="date"/>
            <field name="score"/>
        </tree>
    </field>
</record>
```

ACTION WINDOW

```
<record id="idea_action" model="ir.actions.act_window">
   <field name="name">Idea</field>
   <field name="res_model">idea.idea</field>
   <field name="view_id" ref="idea_view_list"/>
   <field name="view_type">form</field>
   <field name="view_mode">form,tree</field>
</record>
```

MENU

```
<menuitem id="exercise_menu" name="Exercise"
        sequence="10"/>

<menuitem id="idea_mmenu" parent="idea.exercise_menu" name="Idea"
        action="idea_action" sequence="10"/>


</odoo>
```

Tambahkan informasi view pada manifest file, seperti :

```
'data':['views/idea_view.xml'],
```

Restart odoo server
Update Apps List
Install / Update model “Idea”

MODEL - ATTRIBUTES

Berikut ini adalah Predefined **attributes** yang digunakan dalam kelas Python untuk membuat sebuah bisnis objek :

Predefined models.Model attributes for business objects	
_name (required)	Nama bisnis objek, notasi dot (in module namespace)
_rec_name	Alternative field untuk digunakan sebagai “name”, digunakan pada fungsi name_get() (default: 'name')
_inherit	_name dari induk (parent) business object (for inheritance) • If _name is set, names of parent models to inherit from. Can be a str if inheriting from a single parent

	• If _name is unset, name of a single model to extend in-place See Inheritance and extension.
_order	Nama field yang digunakan untuk diurutkan pada List View (default: 'id')
_auto	jika *True* (default) maka ORM akan membuat tabel database, set to *False* untuk membuat table sesuai keinginan Anda dengan cara mendefinisikanya pada init() method
_table	Nama table yang akan dibuat (default: _name with dots '.' replaced by underscores '_') Name of the table backing the model created when _auto, automatically generated by default.

_inherits	for decoration inheritance: dictionary mapping the _name of the parent business object(s) to the names of the corresponding
	foreign key fields to use
	`_inherits = {`
	`  'a.model': 'a_field_id',`
	`  'b.model': 'b_field_id'`
	`}`

~~_constraints~~	~~list of (constraint_function, message, fields) defining Python constraints. The fields list is indicative~~ ~~Deprecated since version 8.0: use **constrains()**~~ ~~(func_name, message, fields) (→70)~~
_sql_constraints	list of tuples defining the SQL constraints to execute when generating the backing table, in the form (name, sql_def, message) (→55)
_parent_store (boolean)	Alongside parent_left and parent_right, sets up a nested set to enable fast hierarchical queries on the records of the current model (default: False)
_parent_name (boolean)	Nama field yang memiliki relasi ke model parent. Pada model berhierarcy, attribute ini HARUS diisi, biasanya diisi “parent_id”
_parent_order	Nama field yang digunakan untuk mengurutkan hierarcy.
_log_access	Jika True (default=True), 4 fields (create_uid, create_date, write_uid, write_date) akan dibuat dan digunakan untuk log record-level operations, made accessible via the perm_read() function
_sql	SQL code to create the table/view for this object (if _auto is False) – can be replaced by SQL execution in the init() method

~~_columns (required)~~	Not use.
~~_defaults~~	Not use.

ORM field types

Objects may contain 3 types of fields: *simple, relational,* and *functional*.
Simple types are integers, floats, booleans, strings, data, datetime etc.
Relational fields represent the relationships between objects (one2many, many2one, many2many).
Functional fields are not stored in the database but calculated on-the-fly as Python functions. Relevant examples in the idea class above are indicated with the corresponding line numbers (→XX,XX)

ORM fields types	
Common attributes supported by all fields (optional unless specified)	
• string : field label (required) • required : True if mandatory • readonly : True if not editable • help : help tooltip • select : True to create a database index on this column • context : dictionary with contextual/parameters (for relational fields) • change_default: True if field should be usablevas condition for default values in clients • states : dynamic changes to this field's common attributes based on the state field. possible attributes are: 'readonly', 'required', 'invisible' (→40) • index : whether the field is indexed in database (boolean, by default False) • default : the default value for the field; this is either a static value, or a function taking a recordset and returning a value • groups : comma-separated list of group xml ids (string of external-ID); this restricts the field access to the users of the given groups only • copy(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) • oldname : the previous name of this field, so that ORM can rename it automatically at migration • track_visibility : choose : always, onchange. Ability to track on view, Your object need to inherit from mail.thread. for traking the visiblity which produces the messages •change_default : jika change_dedault field berubah valuenya, maka semua field yang memiliki default akan di-trigger shg valuenya menjadi default.	
Simple fields	
Pada New API odoo 8, Semua type diawali dengan huruf besar	
Char(string,size,translate=False,..) *Text-based fields*	• translate: True if field values can be translated by users, for char fields • size: optional max size for char fields

	(→41,45)
Text(string, translate=False,...) *Text-based fields*	• translate: True if field values can be translated by users, for text fields

Boolean(...) Integer(...)	`'active': fields.boolean('Active'),`
	`'priority': fields.integer('Priority'),`
	`'start_date': fields.date('Start Date'),`

Float(string, digits=None, ...) *Decimal value*	• digits: tuple (precision, scale) (→58) a pair (total, decimal), or a function taking a database cursor and returning a pair (total, decimal)
Selection(selection, string, ...) Selection(selection_add, string, ...) Field allowing selection among a set of predefined values	• selection: specifies the possible values for this field. It is given as either a list of tuple (value, string), or a model method, or a method name or function returning such a list **(required)** (→42) • selection_add: provides an extension of the selection in the case of an overridden field. It is a list of pairs (value, string).
Html(string,....)	Bases: openerp.fields._String
Date(string, **kwargs)	Fields.Date menyediakan beberapa fungsi seperti : • static context_today(record, timestamp=None) 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 , 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 from_string(value) Convert an ORM value into a date value • static to_string(value) Convert a date value into the format expected by the ORM. • static today (value) Return the current day in the format expected by the ORM. This function may be used to compute default values.
Datetime(string, **kwargs)	Fields.Datetime menyediakan beberapa fungsi seperti : • static context_timestamp(record, timestamp=None) Returns the given timestamp converted to the client's timezone. This method is not meant for use as a _defaults initializer, because datetime fields are automatically converted upon display on client side. For _defaults you fields.datetime.now() should be used instead. Parameters , timestamp (datetime) --naive datetime value (expressed in UTC) to be converted to the client timezone Return type datetime

	• static from_string(value) Convert an ORM value into a datetime value • static to_string(value) Convert a datetime value into the format expected by the ORM. • static today (value) Return the current datetime in the format expected by the ORM. This function may be used to compute default values.
binary(string, filters=None, ...) Field for storing a file or binary content.	• filters: optional filename filters for selection `'picture': fields.binary('Picture',filters='*.png,*.gif')`
	Relational fields
Common attributes supported by **relational** fields	• comodel: name of the target model (string) • domain: optional filter in the form of arguments for search (see search()) an optional domain to set on candidate values on the client side (domain or string) • context: an optional context to use on the client side when handling that field (dictionary) • ondelete: what to do when the referred record is deleted; possible values are: 'set null', 'restrict', 'cascade' • auto_join: whether JOINs are generated upon search through that field (boolean, by default False) • delegate: set it to True to make fields of the target model accessible from the current model (corresponds to _inherits)
many2one(comodel_name=None, string=None, ondelete='set null', ...) (→50) Relationship towards a parent object (using a foreign key)	The value of such a field is a recordset of size 0 (no record) or 1 (a single record). • comodel_name: name of the target model (string) • domain: optional filter in the form of arguments for search (see search()) an optional domain to set on candidate values on the client side (domain or string) • context: an optional context to use on the client side when handling that field (dictionary) • ondelete: what to do when the referred record is deleted; possible values are: 'set null', 'restrict', 'cascade', see PostgreSQL documentatio
one2many(comodel_name=None, inverse_name=None, string=None,, ...) (→55) Virtual relationship towards multiple objects (inverse of many2one)	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. • comodel_name: name of the target model (string) (**required**) • inverse_name: field name of the inverse many2one field in comodel_name (string) or corresponding foreign key (**required**)

	• domain: optional filter in the form of arguments for search (see search()) an optional domain to set on candidate values on the client side (domain or string) • context: an optional context to use on the client side when handling that field (dictionary) • auto_join: whether JOINs are generated upon search through that field (boolean, by default False) • limit: optional limit to use upon read (integer)
many2many(comodel_name=None, relation=None, column1=None, column2=None, string=None, ...) (→56) Bidirectional multiple relationship between objects	• comodel_name: name of the target model (string) (**required**) • relation: optional name of the table that stores the relation in the database (string) • column1: optional name of the column referring to "these" records in the table **relation** (string) • column1: optional name of the column referring to "those" records in the table relation (string) 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! • domain: optional filter in the form of arguments for search (see search()) an optional domain to set on candidate values on the client side (domain or string) • context: an optional context to use on the client side when handling that field (dictionary) • limit: optional limit to use upon read (integer)
~~reference(string, selection, size,...)~~ ~~Field with dynamic relationship to any other object, associated with an assistant widget~~	~~• selection: model_name of allowed objects types and corresponding label (same format as values for selection fields) **(required)**~~ ~~• size: size of text column used to store it (storage format is 'model_name,object_id')~~ ~~`'contact': fields.reference('Contact',[ ('res.partner','Partner'), ('res.partner.contact','Contact')])`~~

Tip: relational fields symmetry
• one2many ↔ many2one are symmetric
• many2many ↔ many2many are symmetric when inversed (swap field1 and field2 if explicit)
• one2many ↔ many2one + many2one ↔ one2many = many2many

Computed Fields

One can define a field whose value is computed instead of simply being read from the database. The attributes that are specific to computed fields are given below. To define such a field, simply provide a value for the attribute `compute`.

Parameters :

- `compute` : name of a method that computes the field
- `inverse` : name of a method that inverses the field (optional)
- `search` : name of a method that implement search on the field (optional)
- `store` : whether the field is stored in database (boolean, by default `False` on computed fields)
- `compute_sudo` : whether the field should be recomputed as superuser to bypass access rights (boolean, by default `False`)

The methods given for `compute`, `inverse` and `search` are model methods. Their signature is shown in the following example:

```
document = fields.Char(compute='_get_document', inverse='_set_document')
upper = fields.Char(compute='_compute_upper',
                    inverse='_inverse_upper',
                    search='_search_upper')


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)
```

The compute method has to assign the field on all records of the invoked recordset. The decorator `openerp.api.depends()` must be applied on the compute method to specify the field dependencies; those dependencies are used to determine when to recompute the field; recomputation is automatic and guarantees cache/database consistency. Note that the same method can be used for several fields, you simply have to assign all the given fields in the method; the method will be invoked once for all those fields.

By default, a computed field is not stored to the database, and is computed on-the-fly. Adding the attribute `store=True` will store the field's values in the database. The advantage of a stored field is that searching on that field is done by the database itself. The disadvantage is that it requires database updates when the field must be recomputed.

The inverse method, as its name says, does the inverse of the compute method: the invoked records have a value for the field, and you must apply the necessary changes on the field dependencies such that the computation gives the expected value. Note that a computed field without an inverse method is readonly by default.

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.

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')

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

Related fields

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 attribute

Parameters :
related : sequence of field

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.

```
number = fields.Char(related='move_id.name', store=True, readonly=True,
copy=False)
atau

commercial_partner_id = fields.Many2one('res.partner', string='Commercial Entity',
```

```
        related='partner_id.commercial_partner_id', store=True, readonly=True,
        help="The commercial entity that will be used on Journal Entries for this invoice")
```

onchange: updating UI on the fly

When a user changes a field's value in a form (but hasn't saved the form yet), it can be useful to automatically update other fields based on that value e.g. updating a final total when the tax is changed or a new invoice line is added.

- computed fields are automatically checked and recomputed, they do not need an **onchange**
- for non-computed fields, the **onchange()** decorator is used to provide new field values:

```
@api.onchange('field1', 'field2') # if these fields are changed, call method
def check_change(self):
    if self.field1 < self.field2:
        self.field3 = True
```

the changes performed during the method are then sent to the client program and become visible to the user

- Both computed fields and new-API onchanges are automatically called by the client without having to add them in views
- It is possible to suppress the trigger from a specific field by adding **on_change="0"** in a view:

```
<field name="name" on_change="0"/>
```

will not trigger any interface update when the field is edited by the user, even if there are function fields or explicit onchange depending on that field.

Note

onchange methods work on virtual records assignment on these records is not written to the database, just used to know which value to send back to the client

Company-dependent fields

Formerly known as 'property' fields, the value of those fields depends on the company. In other words, users that belong to different companies may see different values for the field on a given record.

Parameters :
company_dependent : whether the field is company-dependent (boolean)

Incremental definition

A field is defined as class attribute on a model class. If the model is extended (see Model), 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")
```

Special / Reserved field names

A few field names are reserved for pre-defined behavior in Odoo. Some of them are created automatically by the system, and in that case any field wih that name will be ignored.

id	unique system identifier for the object
name	field whose value is used to display the record in lists, etc. if missing, set _rec_name to specify another field to use
active	toggle visibility: records with active set to False are hidden by default
sequence	defines order and allows drag&drop reordering if visible in list views
state	lifecycle stages for the object, used by the states attribute
parent_id	defines tree structure on records, and enables child_of operator
parent_left, parent_right	used in conjunction with _parent_store flag on object, allows faster access to tree structures (see also Performance Optimization section)
create_date, create_uid, write_date, write_uid	used to log creator, last updater, date of creation and last update date of the record. disabled if _log_access flag is set to False (created by by ORM, do not add them)

Working with the ORM

Inheriting from the modesl.Model class makes all the ORM methods available on business objects. These methods may be invoked on the self object within the Python class itself (see examples in the table below), or from outside the class by first obtaining an instance via the ORM pool system.

ORM Methods

Common parameters, used by multiple methods	• self: database connection (cursor) • context: optional dictionary of contextual parameters, e.g. `{ 'lang': 'en_US', ... }`
New Methods	
new(values={})	Create new record on memory Return a new record instance attached to the current environment and initialized with the provided ``value``. The record is *not* created in database, it only exists in memory.

<table>
<tr><td>update</td><td>???</td></tr>
<tr><td colspan="2">Common Methods</td></tr>
<tr><td>search(args[, offset=0][, limit=None][, order=None][, count=False])

Returns: recordset</td><td>• args: A search domain. Use an empty list to match all records.
• offset: optional number of records to skip
• limit: optional max number of records to return
• order: optional columns to sort by (default: self._order)
• count: if True, returns only the number of records matching the criteria, not their ids

<pre>#Operators: =, !=, >, >=, <, <=, like, ilike,
#in, not in, child_of, parent_left, parent_right
#Prefix operators: '&' (default), '|', '!'
#Fetch non-spam partner shops + partner 34
ids = self.search(
['|', ('partner_id', '!=', 34),
'!', ('name', 'ilike', 'spam'),],
order='partner_id')</pre></td></tr>
<tr><td>search_count(args)

Returns: int</td><td>Returns the number of records in the current model matching the provided domain.
• args: A search domain. Use an empty list to match all records.</td></tr>
<tr><td>name_search(name='', args=None, operator='ilike', limit=100)

Returns: list of pairs (id, text_repr) for all matching records.</td><td>• name: the name pattern to match. Display_name yang dicari
• args: A search domain. Use an empty list to match all records.
• operator: domain operator for matching name, such as 'like' or '='.
• limit: optional max number of records to return.

<pre># Countries can be searched by code or name
def name_search(self,name='',domain=[],
operator='ilike',context=None,limit=80):
   ids = []
   if name and len(name) == 2:
      ids = self.search(cr, user,
         [('code', '=', name)] + args,
         limit=limit, context=context)
   if not ids:
      ids = self.search(cr, user,
         [('name', operator, name)] + args,
         limit=limit, context=context)
   return self.name_get(cr,uid,ids)</pre></td></tr>
<tr><td colspan="2">CRUD</td></tr>
<tr><td>create(vals)

Creates a new record with the specified value
Returns a recordset containing the record created</td><td>• vals: dictionary of field values

<pre>idea_id = self.create(
  { 'name': 'Spam recipe',
    'description' : 'spam & eggs',
    'inventor_id': 45,
  })</pre></td></tr>
</table>

browse([ids])
 	
Returns a recordset for the ids provided as parameter in the current environment requested
	`idea = self.browse([42,43])`
	`print 'Idea description:', idea.description`
	`print 'Inventor country code:',`
	`  idea.inventor_id.address[0].country_id.code`
	`for vote in idea.vote_ids:`
	`  print 'Vote %2.2f' % vote.vote`

unlink() Deletes the records of the current set Returns: True	`record.unlink()`

write(vals)
 	
Updates all records in the current set with the provided values
Returns: True
	• vals: dictionary of field values to update
	
	`record.write({ 'name': 'spam & eggs',`
	`'partner_id': 24,`
	`})`

Danger

for historical and compatibility reasons, Date and Datetime fields use strings as values (written and read) rather than date or datetime. These date strings are UTC-only and formatted according to openerp.tools.misc.DEFAULT_SERVER_DATE_FORMAT and openerp.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, _, 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, _)

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

(3, id, _)

removes the record of id **id** from the set, but does not delete it. Can not be used on **One2many**. Can not be used in**create()**.

(4, id, _)

adds an existing record of id **id** to the set. Can not be used on **One2many**.

(5, _, _)

removes all records from the set, equivalent to using the command **3** on every record explicitly. Can not be used on **One2many**. Can not be used in **create()**.

(6, _, 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**. Can not be used on **One2many**.

> **Note**
>
> Values marked as _ in the list above are ignored and can be anything, generally **0** or **False**.

read([fields]) Reads the requested fields for the records in self, low-level/RPC method. In Python code, prefer browse(). Returns : a list of dictionaries mapping field names to their values, with one dictionary per record	• fields: optional list of field names to return (default: all fields) `results = records.read(['name', 'inventor_id'])`
Other recordset operations	
exists() Returns a new recordset containing only the records which exist in the database. Can be used to check whether a record (e.g. obtained externally) still exists:	`if not record.exists():` `  raise Exception("The record has been deleted")`
ref() Environment method returning the record matching a provided external id	`env.ref('base.group_public')`

ensure_one()
 	
Verifies that the current recorset holds a single record. Raises an exception otherwise.
	`records.ensure_one()`
	`# is equivalent to but clearer than:`
	`assert len(records) == 1, "Expected singleton"`

filtered()
 	
returns a recordset containing only records satisfying the provided predicate function. The predicate can also be a string to filter by a field being true or
	`# 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")`

false	
sorted() returns a recordset sorted by the provided key function. If no key is provided, use the model's default sort order	`# sort records by name` `records.sorted(key=lambda r: r.name)`
mapped() returns a recordset sorted by the provided key function. If no key is provided, use the model's default sort order	`# sort records by name` `records.sorted(key=lambda r: r.name)`
Berikut dibawah ini fungsi-fungsi versi 7, apakah masih ada ???	

read_group(cr, uid, domain, fields,
groupby, offset=0, limit=None, orderby=None, context=None)
 	
Returns: list of dictionaries with requested field values, grouped by given groupby field(s).
	• domain: search filter (see search())
	• fields: list of field names to read
	• groupby: field or list of fields to group by
	• offset, limit: see search()
	• orderby: optional ordering for the results
	
	`> print self.read_group(cr,uid,[], ['score'],`
	`['inventor_id'])`
	`[{'inventor_id': (1, 'Administrator'),`
	`'score': 23, # aggregated score`
	`'inventor_id_count': 12, # group count`
	`},`
	`{'inventor_id': (3, 'Demo'),`
	`'score': 13,`
	`'inventor_id_count': 7,`
	`}]`

copy(cr, uid, id, defaults,context=None) Duplicates record with given id updating it with defaults values. Returns: True	• defaults: dictionary of field values to modify in the copied values when creating the duplicated object

default_get(cr, uid, fields, context=None)
 	
Returns: a dictionary of the default values for fields (set on the object class, by the user preferences, or via the context)
	• fields: list of field names
	
	`defs = self.default_get(cr,uid, ['name','active'])`
	`# active should be True by default`
	`assert defs['active']`

perm_read(cr, uid, ids, details=True) Returns: a list of ownership dictionaries for each requested record	• details: if True, *_uid fields values are replaced with pairs (id, name_of_user) • returned dictionaries contain: object id (id), creator user id (create_uid), creation date (create_date), updater user id (write_uid), update date (write_date)

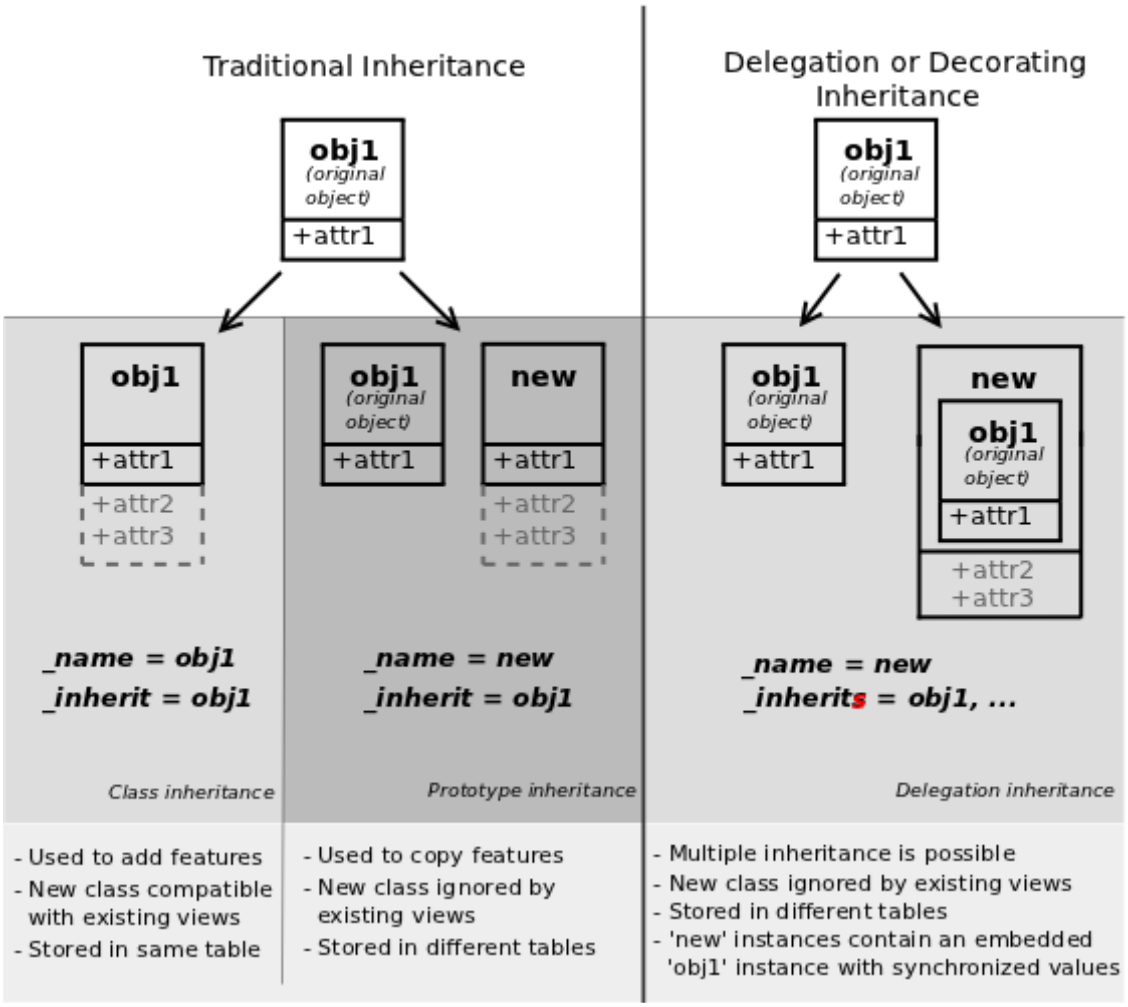
	perms = self.perm_read(cr,uid,[42,43]) print 'creator:', perms[0].get('create_uid', 'n/a')

fields_get(cr, uid, fields=None, context=None)
 	
Returns a dictionary of field dictionaries, each one describing
a field of the business objec
	• fields: list of field names
	class idea(osv.osv):
	(...)
	_columns = {
	'name' : fields.char('Name',size=64)
	(...)
	
	def test_fields_get(*self*,cr,uid):
	assert(self.fields_get('name')['size'] == 64)

fields_view_get(cr, uid, view_id=None, view_type='form', context=None, toolbar=False) Returns a dictionary describing the composition of the requested view (including inherited views)	• view_id: id of the view or None • view_type: type of view to return if view_id is None ('form','tree', ...) • toolbar: True to also return context actions def test_fields_view_get(*self*,cr,uid): idea_obj = *self*.pool.get('idea.idea') form_view = idea_obj.fields_view_get(cr,uid)
name_get(cr, uid, ids, context=None) Returns tuples with the text representation of requested objects for to-many relationships	# Ideas should be shown with invention date def name_get(*self*,cr,uid,ids): res = [] for r in self.read(cr,uid,ids['name','create_date']) res.append((r['id'], '%s (%s)' (r['name'],year)) return res
export_data(cr, uid, ids, fields, context=None) Exports fields for selected objects, returning a dictionary with a datas matrix. Used when exporting data via client menu.	• fields: list of field names • context may contain import_comp (default: False) to make exported data compatible with import_data() (may prevent exporting some fields)
import_data(cr, uid, fields, data, mode='init', current_module='', noupdate=False, context=None, filename=None) Imports given data in the given module Used when exporting data via client menu	• fields: list of field names • data: data to import (see export_data()) • mode: 'init' or 'update' for record creation • current_module: module name • noupdate: flag for record creation • filename: optional file to store partial import state for recovery

Tip: use read() through webservice calls, but prefer browse() internally

Inheritance mechanisms




```
37.   class idea2(models.Model):
38.       _inherit = 'idea.idea'
39.       def _score_calc(self,cr,uid,ids,field,arg,context=None):
40.           res = {}
41.           # This loop generates only 2 queries thanks to browse()!
42.           for idea in self.browse(cr,uid,ids,context=context):
43.              sum_vote = sum([v.vote for v in idea.vote_ids])
44.              avg_vote = sum_vote/len(idea.vote_ids)
45.           res[idea.id] = avg_vote
46.           return res
47.
48.       Score = fields.Float(compute=_score_calc,string='Score')
```

Domains

Domain merupakan sebuah list dari kriteria, setiap kriteria merupakan sebuah tuple yang terdiri dari tiga bagian (field_name, operator, value)
Operator yang dapat diguankan =, !=, >, >=, <, <=, =? , like, ilike

=? 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.

Takes the semantics of the model into account (i.e following the relationship field named by **_parent_name**).

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.

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)
```

Recordset

Lihat odoo_api_8.pdf

Envirotment

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**) 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)
```

Altering the environment

The environment can be customized from a recordset. This returns a new version of the recordset using the altered environment.

sudo()

creates a new environment with the provided user set, uses the administrator if none is provided (to bypass access rights/rules in safe contexts), returns a copy of the recordset it is called on using the new environment:

```
# create partner object as administrator
env['res.partner'].sudo().create({'name': "A Partner"})

# list partners visible by the "public" user
public = env.ref('base.public_user')
env['res.partner'].sudo(public).search([])
```

with_context()

1. can take a single positional parameter, which replaces the current environment's context
2. can take any number of parameters by keyword, which are added to either the current environment's context or the context set during step 1

```
# look for partner, or create one with specified timezone if none is
# found
env['res.partner'].with_context(tz=a_tz).find_or_create(email_address)
```

with_env()

replaces the existing environment entirely

Building the module interface

To construct a module, the main mechanism is to insert data records declaringthe module interface components. Each module element is a regular data record: menus, views, actions, reports, roles, access rights, etc.

Common XML structure

XML files declared in a module's data section contain record declarations in the following form:

```
<?xml version="1.0" encoding="utf-8"?>
<odoo>

     <record model="object_model_name" id="object_xml_id">
        <field name="field1">value1</field>
        <field name="field2">value2</field>
     </record>
```

```
93.
94.      <record model="object_model_name2" id="object_xml_id2">
95.         <field name="field1" ref="module.object_xml_id"/>
96.         <field name="field2" eval="ref('module.object_xml_id')"/>
97.      </record>

98. </odoo>
```

Each type of record (view, menu, action) supports a specific set of child entities and attributes, but all share the following special attributes:

- id the unique (per module) external identifier of this record (xml_id)
- ref may be used instead of normal element content to reference another record (works cross-module by prepending the module name)
- eval used instead of element content to provide value as a Python expression, that
  can use the ref() method to find the database id for a given xml_id

Tip: XML RelaxNG validation

Odoo validates the syntax and structure of XML files, according to a RelaxNG grammar, found in server/bin/import_xml.rng.

For manual check use xmllint: xmllint –relaxng /path/to/import_xml.rng <file>

Common CSV syntax

CSV files can also be added in the data section and the records will be inserted by the OSV's import_data() method, using the CSV filename to determine the target object model. The ORM automatically reconnects relationships based on the following special column names :

id (xml_id)	column containing identifiers for relationships
many2one_field	reconnect many2one using name_search()
many2one_field:id	reconnect many2one based on object's xml_id
many2one_field.id	reconnect many2one based on object's database id
many2many_field	reconnect via name_search(), multiple values w/ commas
many2many_field:id	reconnect w/ object's xml_id, multiple values w/ commas
many2many_field.id	reconnect w/ object's database id, multiple values w/ commas
one2many_field/field	creates one2many destination record and sets field value

IR.MODEL.ACCESS.CSV

```
"id","name","model_id:id","group_id:id","perm_read","perm_write","perm_create","perm_unlink"
"access_idea_idea","idea.idea","model_idea_idea","base.group_user",1,0,0,0
"access_idea_vote","idea.vote","model_idea_vote","base.group_user",1,0,0,0
```

Menus and actions

Actions are declared as regular records and can be triggered in 3 ways:

- by clicking on menu items linked to a specific action
- by clicking on buttons in views, if these are connected to actions
- as contextual actions on an object (visible in the side bar)

Action declaration

```
<record model="ir.actions.act_window" id="action_id">
    <field name="name">action.name</field>
    <field name="view_id" ref="view_id"/>
    <field name="domain">[list of 3-tuples (max 250 characters)]</field>
    <field name="context">{context dictionary (max 250 characters)}</field>
    <field name="res_model">object.model.name</field>
    <field name="view_type">form|tree</field>
    <field name="view_mode">form,tree,calendar,graph</field>
    <field name="target">new</field>
    <field name="search_view_id" ref="search_view_id"/>
</record>
```

id	identifier of the action in table ir.actions.act_window, must be unique
name	action name (required)
view_id	specific view to open (if missing, highest priority view of given type is used)
domain	tuple (see search() arguments) for filtering the content of the view

context context dictionary to pass to the view
res_model object model on which the view to open is defined
view_type set to form to open records in edit mode, set to tree for a hierarchy view only
view_mode if view_type is form, list allowed modes for viewing records (form, tree, ...)
target set to new to open the view in a new window/popup
search_view_id identifier of the search view to replace default search form (new in version 6.0)

Menu declaration

The menuitem element is a shortcut for declaring an ir.ui.menu record and connect it with a corresponding action via an ir.model.data record.

```
115.      <menuitem id="menu_id" parent="parent_menu_id" name="label"
116.      action="action_id" groups="groupname1,groupname2" sequence="10"/>
```

id identifier of the menuitem, must be unique
parent external ID (xml_id) of the parent menu in the hierarchy
name optional menu label (default: action name)
action identifier of action to execute, if any
groups list of groups that can see this menu item (if missing, all groups can see it)
sequence integer index for ordering sibling menuitems (10,20,30..)

Views and inheritance

Views form a hierarchy. Several views of the same type can be declared on the same object, and will be used depending on their priorities. By declaring an inherited view it is possible to add/remove features in a view.

Generic view declaration

```
117.  <record model="ir.ui.view" id="view_id">
118.      <field name="name">view.name</field>
119.      <field name="model">object_name</field>
120.      <!-- types: tree,form,calendar,search,graph,gantt,kanban -->
121.      <field name="type">form</field>
122.      <field name="priority" eval="16"/>
123.            <field name="arch" type="xml">
124.            <!-- view content: <form>, <tree>, <graph>, … -->
125.            </field>
126.  </record>
```

id unique view identifier
name view name
model object model on which the view is defined (same as *res_model* in actions)
type view type: *form*, *tree*, *graph*, *calendar*, *search, gantt, kanban*
priority view priority, smaller is higher (default: 16)
arch architecture of the view, see various view types below

Forms (to view/edit records)

Forms allow creation/edition or resources, and correspond to <form> elements.
Allowed elements all (see form elements below)

```
<form string="Idea form">
    <group col="6" colspan="4">
        <group colspan="5" col="6">
            <field name="name" colspan="6"/>
            <field name="inventor_id"/>
            <field name="inventor_country_id" />
            <field name="score"/>
        </group>
        <group colspan="1" col="2">
            <field name="active"/><field name="invent_date"/>
        </group>
    </group>
    <notebook colspan="4">
        <page string="General">
            <separator string="Description"/>
            <field colspan="4" name="description" nolabel="1"/>
        </page>
        <page string="Votes">
            <field colspan="4" name="vote_ids" nolabel="1">
               <tree>
                   <field name="partner_id"/>
                   <field name="vote"/>
               </tree>
            </field>
        </page>
        <page string="Sponsors">
            <field colspan="4" name="sponsor_ids" nolabel="1"/>
        </page>
    </notebook>
    <field name="state"/>
    <button name="do_confirm" string="Confirm" type="object"/>
</form>
```

Form Elements

Common attributes for all elements:

- string: label of the element
- nolabel: 1 to hide the field label
- colspan: number of column on which the field must span
- rowspan: number of rows on which the field must span
- col: number of column this element must allocate to its child elements
- invisible: *1* to hide this element completely
- eval: evaluate this Python code as element content (content is string by default)
- attrs: Python map defining dynamic conditions on these attributes: readonly,invisible, required based on search tuples on other field values

field automatic widgets depending on the corresponding field type. Attributes:

- string: label of the field for this particular view
- nolabel: *1* to hide the field label
- required: override required field attribute from Model for this view

- readonly: override readonly field attribute from Model for this view
- password: *True* to hide characters typed in this field
- context: Python code declaring a context dictionary
- domain: Python code declaring list of tuples for restricting values
- on_change: Python method to call when field value is changed
- groups: comma-separated list of group (id) allowed to see this field
- widget: select alternative field widget (*url, email, image, float_time, reference, html, progressbar, statusbar, handle, etc.*)

properties dynamic widget showing all available properties (no attribute)

button clickable widget associated with actions. Specific attributes:
- type: type of button: *workflow* (default), *object*, or *action*
- name: workflow signal, function name (without parentheses) or action to call (depending on type)
- confirm: text of confirmation message when clicked
- states: comma-separated list of states in which this button is shown

separator horizontal separator line for structuring views, with optional label

newline place-holder for completing the current line of the view

label free-text caption or legend in the form

group used to organise fields in groups with optional label (adds frame)

notebook, page *notebook* elements are tab containers for *page* elements. Attributes:
- name: label for the tab/page
- position: tabs position in notebook (*inside, top, bottom, left, right*)

Dynamic views

In addition to what can be done with states and attrs attributes, functions may be called by view elements (via buttons of type object, or on_change triggers on fields) to obtain dynamic behavior. These functions may alter the view interface by returning a Python map with the following entries:

value	a dictionary of field names and their new values
domain	a dictionary of field names and their updated domains of value
warning	a dictionary with a title and message to show a warning dialog

Lists and Hierarchical Tree Lists

List views include *field* elements, are created with type *tree*, and have a <tree> parent element. They are used to define flat lists (editable or not) and hierarchical lists.

Attributes
- colors: list of colors or HTML color codes mapped to Python conditions
- editable: *top* or *bottom* to allow in-place edit
- toolbar: set to *True* to display the top level of object hierarchies as a side toolbar (only for hierarchical lists, i.e.

opened with actions that set the view_type to “tree” instead of “mode”)

Allowed elements *field, group, separator, tree, button, filter, newline*

```
<tree string="Idea Categories" toolbar="1" colors="blue:state==draft">
<field name="name"/>
<field name="state"/>
</tree>
```