Table Of Contents

Related Topics

This Page

Quick search

Objects, Fields and Methods

OpenERP Objects

All the ERP's pieces of data are accessible through "objects". As an example, there is a res.partner object to access the data concerning the partners, an account.invoice object for the data concerning the invoices, etc...

Please note that there is an object for every type of resource, and not an object per resource. We have thus a res.partner object to manage all the partners and not a *res.partner* object per partner. If we talk in "object oriented" terms, we could also say that there is an object per level.

The direct consequences is that all the methods of objects have a common parameter: the "ids" parameter. This specifies on which resources (for example, on which partner) the method must be applied. Precisely, this parameter contains a list of resource ids on which the method must be applied.

For example, if we have two partners with the identifiers 1 and 5, and we want to call the res_partner method "send_email", we will write something like:

```
res_partner.send_email(... , [1, 5], ...)
```

We will see the exact syntax of object method calls further in this document.

In the following section, we will see how to define a new object. Then, we will check out the different methods of doing this.

For developers:

- OpenERP "objects" are usually called classes in object oriented programming.
- A OpenERP "resource" is usually called an object in OO programming, instance of a class.

It's a bit confusing when you try to program inside OpenERP, because the language used is Python, and Python is a fully object oriented language, and has objects and instances ...

Luckily, an OpenERP "resource" can be converted magically into a nice Python object using the "browse" class method (OpenERP object method).

The ORM - Object-relational mapping - Models

The ORM, short for Object-Relational Mapping, is a central part of OpenERP.

In OpenERP, the data model is described and manipulated through Python classes and objects. It is the ORM job to bridge the gap – as transparently as possible for the developer – between Python and the underlying relational database (PostgreSQL), which will provide the persistence we need for our objects.

OpenERP Object Attributes

Objects Introduction

To define a new object, you must define a new Python class then instantiate it. This class must inherit from the osv class in the osv module.

Object definition

The first line of the object definition will always be of the form:

```
class name_of_the_object(osv.osv):
        _name = 'name.of.the.object'
        _columns = { ... }
        ...
name_of_the_object()
```

An object is defined by declaring some fields with predefined names in the class. Two of them are required (_name and _columns), the rest are optional. The predefined fields are:

Predefined fields

_auto
: Determines whether a corresponding PostgreSQL table must be generated automatically from the object. Setting _auto to False can be useful in case of OpenERP objects generated from PostgreSQL views. See the "Reporting From PostgreSQL Views" section for more details.

_columns (required)
: The object fields. See the fields section for further details.

_constraints
: The constraints on the object. See the constraints section for details.

_sql_constraints
: The SQL Constraint on the object. See the SQL constraints section for further details.

_defaults
: The default values for some of the object's fields. See the default value section for details.

_inherit

The name of the osv object which the current object inherits from. See the object inheritance section (first form) for further details.

_inherits
: The list of osv objects the object inherits from. This list must be given in a python dictionary of the form: {'name_of_the_parent_object': 'name_of_the_field', ...}. See the object inheritance section (second form) for further details. Default value: {}.

_log_access
: Determines whether or not the write access to the resource must be logged. If true, four fields will be created in the SQL table: create_uid, create_date, write_uid, write_date. Those fields represent respectively the id of the user who created the record, the creation date of record, the id of the user who last modified the record, and the date of that last modification. This data may be obtained by using the perm_read method.

_name (required)
: Name of the object. Default value: None.

_order
: Name of the fields used to sort the results of the search and read methods.

Default value: 'id'.

Examples:

```
_order = "name"
_order = "date_order desc"
```

_rec_name
: Name of the field in which the name of every resource is stored. Default value: 'name'. Note: by default, the name_get method simply returns the content of this field.

_sequence
: Name of the SQL sequence that manages the ids for this object. Default value: None.

_sql
: SQL code executed upon creation of the object (only if _auto is True). It means this code gets executed after the table is created.

_table
: Name of the SQL table. Default value: the value of the _name field above with the dots (.) replaced by underscores (_).

Object Inheritance - _inherit

Introduction

Objects may be inherited in some custom or specific modules. It is better to inherit an object to add/modify some fields.

It is done with:

```
_inherit='object.name'
```

Extension of an object

There are two possible ways to do this kind of inheritance. Both ways result in a new class of data, which holds parent fields and behaviour as well as additional fields and behaviour, but they differ in heavy programatical consequences.

While Example 1 creates a new subclass "custom_material" that may be "seen" or "used" by any view or tree which handles "network.material", this will not be the case for Example 2.

This is due to the table (other.material) the new subclass is operating on, which will never be recognized by previous "network.material" views or trees.

Example 1:

```
class custom_material(osv.osv):
    _name = 'network.material'
    _inherit = 'network.material'
    _columns = {
        'manuf_warranty': fields.boolean('Manufacturer warranty?'),
    }
    _defaults = {
        'manuf_warranty': lambda *a: False,
    }
    custom_material()
```

Tip:

Notice

_name == _inherit

In this example, the 'custom_material' will add a new field 'manuf_warranty' to the object 'network.material'. New instances of this class will be visible by views or trees operating on the superclasses table 'network.material'.

v: latest

This inheritance is usually called "class inheritance" in Object oriented design. The child inherits data (fields) and behavior (functions) of his parent.

Example 2:

```
class other_material(osv.osv):
    _name = 'other.material'
    _inherit = 'network.material'
    _columns = {
        'manuf_warranty': fields.boolean('Manufacturer warranty?'),
    }
    _defaults = {
        'manuf_warranty': lambda *a: False,
    }
    other_material()
```

Tip:

Notice

_name != _inherit

In this example, the 'other_material' will hold all fields specified by 'network.material' and it will additionally hold a new field 'manuf_warranty'. All those fields will be part of the table 'other.material'. New instances of this class will therefore never been seen by views or trees operating on the superclasses table 'network.material'.

This type of inheritancy is known as "inheritance by prototyping" (e.g. Javascript), because the newly created subclass "copies" all fields from the specified superclass (prototype). The child inherits data (fields) and behavior (functions) of his parent.

Inheritance by Delegation - _inherits

Syntax ::

```
class tiny_object(osv.osv)
    _name = 'tiny.object'
    _table = 'tiny_object'
    _inherits = {
        'tiny.object_a': 'object_a_id',
        'tiny.object_b': 'object_b_id',
        ... ,
        'tiny.object_n': 'object_n_id'
    }
    (...)
```

The object 'tiny.object' inherits from all the columns and all the methods from the n objects 'tiny.object_a', ..., 'tiny.object_n'.

To inherit from multiple tables, the technique consists in adding one column to the table tiny_object per inherited object. This column will store a foreign key (an id from another table). The values *'object_a_id' 'object_b_id'* ... *'object_n_id'* are of type string and determine the title of the columns in which the foreign keys from 'tiny.object_a', ..., 'tiny.object_n' are stored.

This inheritance mechanism is usually called " *instance inheritance* " or " *value inheritance* ". A resource (instance) has the VALUES of its parents.

Fields Introduction

Objects may contain different types of fields. Those types can be divided into three categories: simple types, relation types and functional fields. The simple types are integers, floats, booleans, strings, etc ... ; the relation types are used to represent relations between objects (one2one, one2many, many2one). Functional fields are special fields because they are not stored in the database but calculated in real time given other fields of the view.

Here's the header of the initialization method of the class any field defined in OpenERP inherits (as you can see in server/bin/osv/fields.py):

```
def __init__(self, string='unknown', required=False, readonly=False,
             domain=None, context="", states=None, priority=0, change_default=False, size=None,
             ondelete="set null", translate=False, select=False, **args) :
```

There are a common set of optional parameters that are available to most field types:

change_default:	Whether or not the user can define default values on other fields depending on the value of this field. Those default values need to be defined in the ir.values table.
help:	A description of how the field should be used: longer and more descriptive than *string*. It will appear in a tooltip when the mouse hovers over the field.
ondelete:	How to handle deletions in a related record. Allowable values are: 'restrict', 'no action', 'cascade', 'set null', and 'set default'.
priority:	Not used?
readonly:	*True* if the user cannot edit this field, otherwise *False*.
required:	*True* if this field must have a value before the object can be saved, otherwise *False*.
size:	The size of the field in the database: number characters or digits.
states:	Lets you override other parameters for specific states of this object. Accepts a dictionary with the state names as keys and a list of name/value tuples as the values. For example: *states={'posted':[('readonly',True)]}*
string:	The field name as it should appear in a label or column header. Strings containing non-ASCII characters must use python unicode objects. For example: *'tested': fields.boolean(u'Testé')*
translate:	*True* if the *content* of this field should be translated, otherwise *False*.

There are also some optional parameters that are specific to some field types:

v: latest

context:	Define a variable's value visible in the view's context or an on-change function. Used when searching child table of *one2many* relationship?
domain:	Domain restriction on a relational field.

Default value: [].
Example: domain=[(‘field’,’=’,value)])

invisible: Hide the field’s value in forms. For example, a password.

on_change: Default value for the *on_change* attribute in the view. This will launch a function on the server when the field changes in the client. For example, *on_change=”onchange_shop_id(shop_id)”*.

relation: Used when a field is an id reference to another table. This is the name of the table to look in. Most commonly used with related and function field types.

select: Default value for the *select* attribute in the view. 1 means basic search, and 2 means advanced search.

Type of Fields

Basic Types

boolean: A boolean (true, false).
Syntax:

```
fields.boolean('Field Name' [, Optional Parameters]),
```

integer: An integer.
Syntax:

```
fields.integer('Field Name' [, Optional Parameters]),
```

float: A floating point number.
Syntax:

```
fields.float('Field Name' [, Optional Parameters]),
```

Note:
The optional parameter digits defines the precision and scale of the number. The scale being the number of digits after the decimal point whereas the precision is the total number of significant digits in the number (before and after the decimal point). If the parameter digits is not present, the number will be a double precision floating point number. Warning: these floating-point numbers are inexact (not any value can be converted to its binary representation) and this can lead to rounding errors. You should always use the digits parameter for monetary amounts.

Example:

```
'rate': fields.float(
    'Relative Change rate',
    digits=(12,6) [,
    Optional Parameters]),
```

char: A string of limited length. The required size parameter determines its size.
Syntax:

```
fields.char(
        'Field Name',
        size=n [,
        Optional Parameters]), # where ''n'' is an integer.
```

Example:

```
'city' : fields.char('City Name', size=30, required=True),
```

text: A text field with no limit in length.
Syntax:

```
fields.text('Field Name' [, Optional Parameters]),
```

date: A date.
Syntax:

```
fields.date('Field Name' [, Optional Parameters]),
```

datetime: Allows to store a date and the time of day in the same field.
Syntax:

```
fields.datetime('Field Name' [, Optional Parameters]),
```

binary: A binary chain

selection: A field which allows the user to make a selection between various predefined values.
Syntax:

```
fields.selection((('n','Unconfirmed'), ('c','Confirmed')),
                   'Field Name' [, Optional Parameters]),
```

v: latest

Note:
Format of the selection parameter: tuple of tuples of strings of the form:

```
(('key_or_value', 'string_to_display'), ... )
```

Note:
You can specify a function that will return the tuple. Example

```
def _get_selection(self, cursor, user_id, context=None):
    return (
       ('choice1', 'This is the choice 1'),
       ('choice2', 'This is the choice 2'))

_columns = {
   'sel' : fields.selection(
       _get_selection,
       'What do you want ?')
}
```

Example
Using relation fields **many2one** with **selection**. In fields definitions add:

```
...,
'my_field': fields.many2one(
        'mymodule.relation.model',
        'Title',
        selection=_sel_func),
...,
```

And then define the _sel_func like this (but before the fields definitions):

```
def _sel_func(self, cr, uid, context=None):
    obj = self.pool.get('mymodule.relation.model')
    ids = obj.search(cr, uid, [])
    res = obj.read(cr, uid, ids, ['name', 'id'], context)
    res = [(r['id'], r['name']) for r in res]
    return res
```

Relational Types

one2one: A one2one field expresses a one:to:one relation between two objects. It is deprecated. Use many2one instead.
Syntax:

```
fields.one2one('other.object.name', 'Field Name')
```

many2one: Associates this object to a parent object via this Field. For example Department an Employee belongs to would Many to one. i.e Many employees will belong to a Department
Syntax:

```
fields.many2one(
        'other.object.name',
        'Field Name',
        optional parameters)
```

Optional parameters:

- ondelete: What should happen when the resource this field points to is deleted.
    - Predefined value: “cascade”, “set null”, “restrict”, “no action”, “set default”
    - Default value: “set null”
- required: True
- readonly: True
- select: True - (creates an index on the Foreign Key field)

Example

```
'commercial': fields.many2one(
        'res.users',
        'Commercial',
        ondelete='cascade'),
```

one2many: TODO
Syntax:

```
fields.one2many(
        'other.object.name',
        'Field relation id',
        'Fieldname',
        optional parameter)
```

v: latest

Optional parameters:

- invisible: True/False
- states: ?
- readonly: True/False

Example

```
'address': fields.one2many(
        'res.partner.address',
        'partner_id',
        'Contacts'),
```

many2many: TODO
Syntax:

```
fields.many2many('other.object.name',
                 'relation object',
                 'actual.object.id',
                 'other.object.id',
                 'Field Name')
```

Where:

- other.object.name is the other object which belongs to the relation
- relation object is the table that makes the link
- actual.object.id and other.object.id are the fields’ names used in the relation table

Example:

```
'category_ids':
   fields.many2many(
    'res.partner.category',
    'res_partner_category_rel',
    'partner_id',
    'category_id',
    'Categories'),
```

To make it bidirectional (= create a field in the other object):

```
class other_object_name2(osv.osv):
    _inherit = 'other.object.name'
    _columns = {
        'other_fields': fields.many2many(
            'actual.object.name',
            'relation object',
            'actual.object.id',
            'other.object.id',
            'Other Field Name'),
    }
other_object_name2()
```

Example:

```
class res_partner_category2(osv.osv):
    _inherit = 'res.partner.category'
    _columns = {
        'partner_ids': fields.many2many(
            'res.partner',
            'res_partner_category_rel',
            'category_id',
            'partner_id',
            'Partners'),
    }
res_partner_category2()
```

related: Sometimes you need to refer to the relation of a relation. For example, supposing you have objects: City -> State -> Country, and you need to refer to the Country from a City, you can define a field as below in the City object:

```
'country_id': fields.related(
    'state_id',
    'country_id',
    type="many2one",
    relation="res.country",
    string="Country",
    store=False)
```

Where:

- The first set of parameters are the chain of reference fields to follow, with the desired field at the end.
- type is the type of that desired field.
- Use relation if the desired field is still some kind of reference. relation is the table to look up that reference in.

Functional Fields

A functional field is a field whose value is calculated by a function (rather than being stored in the database).

Parameters:

```
fnct, arg=None, fnct_inv=None, fnct_inv_arg=None, type="float",
    fnct_search=None, obj=None, method=False, store=False, multi=False
```

where

- fnct is the function or method that will compute the field value. It must have been declared before declaring the functional field.
- fnct_inv is the function or method that will allow writing values in that field.
- type is the field type name returned by the function. It can be any field type name except function.
- fnct_search allows you to define the searching behaviour on that field.
- method whether the field is computed by a method (of an object) or a global function
- store If you want to store field in database or not. Default is False.
- multi is a group name. All fields with the same *multi* parameter will be calculated in a single function call.

fnct parameter

If *method* is True, the signature of the method must be:

```
def fnct(self, cr, uid, ids, field_name, arg, context):
```

otherwise (if it is a global function), its signature must be:

```
def fnct(cr, table, ids, field_name, arg, context):
```

Either way, it must return a dictionary of values of the form **{id'_1_': value'_1_', id'_2_': value'_2_',...}.**

The values of the returned dictionary must be of the type specified by the type argument in the field declaration.

If *multi* is set, then *field_name* is replaced by *field_names*: a list of the field names that should be calculated. Each value in the returned dictionary is also a dictionary from field name to value. For example, if the fields *'name'*, and *'age'* are both based on the *vital_statistics* function, then the return value of *vital_statistics* might look like this when *ids* is *[1, 2, 5]*:

```
{
    1: {'name': 'Bob', 'age': 23},
    2: {'name': 'Sally', 'age', 19},
    5: {'name': 'Ed', 'age': 62}
}
```

fnct_inv parameter

If *method* is true, the signature of the method must be:

```
def fnct(self, cr, uid, ids, field_name, field_value, arg, context):
```

otherwise (if it is a global function), it should be:

```
def fnct(cr, table, ids, field_name, field_value, arg, context):
```

fnct_search parameter

If method is true, the signature of the method must be:

```
def fnct(self, cr, uid, obj, name, args, context):
```

otherwise (if it is a global function), it should be:

```
def fnct(cr, uid, obj, name, args, context):
```

The return value is a list containing 3-part tuples which are used in search function:

```
return [('id','in',[1,3,5])]
```

obj is the same as *self*, and *name* receives the field name. *args* is a list of 3-part tuples containing search criteria for this field, although the search function may be called separately for each tuple.

Example

Suppose we create a contract object which is :

```
class hr_contract(osv.osv):
    _name = 'hr.contract'
    _description = 'Contract'
    _columns = {
        'name' : fields.char('Contract Name', size=30, required=True),
        'employee_id' : fields.many2one('hr.employee', 'Employee', required=True),
        'function' : fields.many2one('res.partner.function', 'Function'),
```

v: latest

```
    }
hr_contract()
```

If we want to add a field that retrieves the function of an employee by looking its current contract, we use a functional field. The object hr_employee is inherited this way:

```
class hr_employee(osv.osv):
    _name = "hr.employee"
    _description = "Employee"
    _inherit = "hr.employee"
    _columns = {
        'contract_ids' : fields.one2many('hr.contract', 'employee_id', 'Contracts'),
        'function' : fields.function(
            _get_cur_function_id,
            type='many2one',
            obj="res.partner.function",
            method=True,
            string='Contract Function'),
    }
hr_employee()
```

> **Note:**
>
> three points
>
> - type ='many2one' is because the function field must create a many2one field; function is declared as a many2one in hr_contract also.
> - obj ="res.partner.function" is used to specify that the object to use for the many2one field is res.partner.function.
> - We called our method _get_cur_function_id because its role is to return a dictionary whose keys are ids of employees, and whose corresponding values are ids of the function of those employees. The code of this method is:

```
def _get_cur_function_id(self, cr, uid, ids, field_name, arg, context):
    for i in ids:
        #get the id of the current function of the employee of identifier "i"
        sql_req= """
        SELECT f.id AS func_id
        FROM hr_contract c
          LEFT JOIN res_partner_function f ON (f.id = c.function)
        WHERE
          (c.employee_id = %d)
        """ % (i,)

        cr.execute(sql_req)
        sql_res = cr.dictfetchone()

        if sql_res: #The employee has one associated contract
            res[i] = sql_res['func_id']
        else:
            #res[i] must be set to False and not to None because of XML:RPC
            # "cannot marshal None unless allow_none is enabled"
            res[i] = False
    return res
```

The id of the function is retrieved using a SQL query. Note that if the query returns no result, the value of sql_res['func_id'] will be None. We force the False value in this case value because XML:RPC (communication between the server and the client) doesn't allow to transmit this value.

store Parameter

It will calculate the field and store the result in the table. The field will be recalculated when certain fields are changed on other objects. It uses the following syntax:

```
store = {
    'object_name': (
            function_name,
            ['field_name1', 'field_name2'],
            priority)
}
```

It will call function function_name when any changes are written to fields in the list ['field1','field2'] on object 'object_name'. The function should have the following signature:

```
def function_name(self, cr, uid, ids, context=None):
```

Where *ids* will be the ids of records in the other object's table that have changed values in the watched fields. The function should return a list of ids of records in its own table that should have the field recalculated. That list will be sent as a parameter for the main function of the field.

Here's an example from the membership module:

```
'membership_state':
    fields.function(
        _membership_state,
        method=True,
        string='Current membership state',
        type='selection',
```

v: latest

```
        selection=STATE,
        store={
            'account.invoice': (_get_invoice_partner, ['state'], 10),
            'membership.membership_line': (_get_partner_id,['state'], 10),
            'res.partner': (
                lambda self, cr, uid, ids, c={}: ids,
                ['free_member'],
                10)
        }),
```

Property Fields

Declaring a property

A property is a special field: fields.property.

```
class res_partner(osv.osv):
    _name = "res.partner"
    _inherit = "res.partner"
    _columns = {
                'property_product_pricelist':
                                            fields.property(
                                'product.pricelist',
                        type='many2one',
                        relation='product.pricelist',
                        string="Sale Pricelist",
                                method=True,
                                group_name="Pricelists Properties"),
    }
```

Then you have to create the default value in a .XML file for this property:

```
<record model="ir.property" id="property_product_pricelist">
    <field name="name">property_product_pricelist</field>
    <field name="fields_id" search="[('model','=','res.partner'),
      ('name','=','property_product_pricelist')]"/>
    <field name="value" eval="'product.pricelist,'+str(list0)"/>
</record>
```

Tip:

if the default value points to a resource from another module, you can use the ref function like this:

<field name=”value” eval=“‘product.pricelist,’+str(ref(‘module.data_id’))”/>

Putting properties in forms

To add properties in forms, just put the <properties/> tag in your form. This will automatically add all properties fields that are related to this object. The system will add properties depending on your rights. (some people will be able to change a specific property, others won’t).

Properties are displayed by section, depending on the group_name attribute. (It is rendered in the client like a separator tag).

How does this work ?

The fields.property class inherits from fields.function and overrides the read and write method. The type of this field is many2one, so in the form a property is represented like a many2one function.

But the value of a property is stored in the ir.property class/table as a complete record. The stored value is a field of type reference (not many2one) because each property may point to a different object. If you edit properties values (from the administration menu), these are represented like a field of type reference.

When you read a property, the program gives you the property attached to the instance of object you are reading. If this object has no value, the system will give you the default property.

The definition of a property is stored in the ir.model.fields class like any other fields. In the definition of the property, you can add groups that are allowed to change to property.

Using properties or normal fields

When you want to add a new feature, you will have to choose to implement it as a property or as normal field. Use a normal field when you inherit from an object and want to extend this object. Use a property when the new feature is not related to the object but to an external concept.

Here are a few tips to help you choose between a normal field or a property:

Normal fields extend the object, adding more features or data.

A property is a concept that is attached to an object and have special features:

- Different value for the same property depending on the company
- Rights management per field
- It’s a link between resources (many2one)

Example 1: Account Receivable

The default “Account Receivable” for a specific partner is implemented as a property because:

- This is a concept related to the account chart and not to the partner, so it is an account property that is visible on a partner form. Rights have to be managed on this fields for accountants, these are not the same rights that are applied to partner objects. So you have specific rights just for this field of the partner form: only accountants may change the account receivable of a partner.
- This is a multi-company field: the same partner may have different account receivable values depending on the company the user belongs to. In a multi-company system, there is one account chart per company. The account receivable of a partner depends on the company it placed the sale order.
- The default account receivable is the same for all partners and is configured from the general property menu (in administration).

Note:

One interesting thing is that properties avoid “spaghetti” code. The account module depends on the partner (base) module. But you can install the partner (base) module without the accounting module. If you add a field that points to an account in the partner object, both objects will depend on each other. It’s much more difficult to maintain and code (for instance, try to remove a table when both tables are pointing to each others.)

Example 2: Product Times

The product expiry module implements all delays related to products: removal date, product usetime, ... This module is very useful for food industries.

This module inherits from the product.product object and adds new fields to it:

```
class product_product(osv.osv):

    _inherit = 'product.product'
    _name = 'product.product'
    _columns = {

        'life_time': fields.integer('Product lifetime'),
        'use_time': fields.integer('Product usetime'),
        'removal_time': fields.integer('Product removal time'),
        'alert_time': fields.integer('Product alert time'),
        }

product_product()
```

This module adds simple fields to the product.product object. We did not use properties because:

- We extend a product, the life_time field is a concept related to a product, not to another object.
- We do not need a right management per field, the different delays are managed by the same people that manage all products.

ORM methods

Keeping the context in ORM methods

In OpenObject, the context holds very important data such as the language in which a document must be written, whether function field needs updating or not, etc.

When calling an ORM method, you will probably already have a context - for example the framework will provide you with one as a parameter of almost every method. If you do have a context, it is very important that you always pass it through to every single method you call.

This rule also applies to writing ORM methods. You should expect to receive a context as parameter, and always pass it through to every other method you call..