This page introduces the Odoo Coding Guidelines. Those aim to improve the quality of Odoo Apps code. Indeed proper code improves readability, eases maintenance, helps debugging, lowers complexity and promotes reliability. These guidelines should be applied to every new module and to all new development.

> **⚠ Warning**
> When modifying existing files in **stable version** the original file style strictly supersedes any other style guidelines. In other words please never modify existing files in order to apply these guidelines. It avoids disrupting the revision history of code lines. Diff should be kept minimal. For more details, see our **pull request guide (https://odoo.com/submit-pr)**.

> **⚠ Warning**
> When modifying existing files in **master (development) version** apply those guidelines to existing code only for modified code or if most of the file is under revision. In other words modify existing files structure only if it is going under major changes. In that case first do a **move** commit then apply the changes related to the feature.

Module structure

Directories

A module is organized in important directories. Those contain the business logic; having a look at them should make you understand the purpose of the module.

data/ : demo and data xml

models/ : models definition

controllers/ : contains controllers (HTTP routes)

views/ : contains the views and templates

static/ : contains the web assets, separated into *css/, js/, img/, lib/, ...*

Other optional directories compose the module.

wizard/ : regroups the transient models (`models.TransientModel`) and their views

report/ : contains the printable reports and models based on SQL views. Python objects and XML views are included in this directory

tests/ : contains the Python tests

File naming

File naming is important to quickly find information through all odoo addons. This section explains how to name files in a standard odoo module. As an example we use a plant nursery (https://github.com/tivisse/odoodays-2018/tree/master/plant_nursery) application. It holds two main models *plant.nursery* and *plant.order*.
Concerning *models*, split the business logic by sets of models belonging to a same main model. Each set lies in a given file named based on its main model. If there is only one model, its name is the same as the module name. Each inherited model should be in its own file to help understanding of impacted models.

```
addons/plant_nursery/
|-- models/
|   |-- plant_nursery.py (first main model)
|   |-- plant_order.py (another main model)
|   |-- res_partner.py (inherited Odoo model)
```

Concerning *security* and access rights and rules two main files should be used. First one is the definition of access rights done in a `ir.model.access.csv` file. User groups are defined in `<module>_groups.xml`. Access rules are defined in `<model>_security.xml`.

```
addons/plant_nursery/
|-- security/
|   |-- ir.model.access.csv
|   |-- plant_nursery_groups.xml
|   |-- plant_nursery_security.xml
|   |-- plant_order_security.xml
```

Concerning *views*, backend views should be split like models and suffixed by `_views.xml`. Backend views are list, form, kanban, activity, graph, pivot, .. views. To ease split by model in views main menus not linked to specific actions may be extracted into an optional `<module>_menus.xml` file. Templates (QWeb pages used notably for portal / website display) and bundles (import of JS and CSS assets) are put in separate files. Those are respectively `<model>_templates.xml` and `assets.xml` files.

```
addons/plant_nursery/
|-- views/
|   | -- assets.xml (import of JS / CSS)
|   | -- plant_nursery_menus.xml (optional definition of main menus)
|   | -- plant_nursery_views.xml (backend views)
|   | -- plant_nursery_templates.xml (portal templates)
|   | -- plant_order_views.xml
|   | -- plant_order_templates.xml
|   | -- res_partner_views.xml
```

Concerning *data*, split them by purpose (demo or data) and main model. Filenames will be the main_model name suffixed by `_demo.xml` or `_data.xml`. For instance for an application having demo and data for its main model as well as subtypes, activities and mail templates all related to mail module:

```
addons/plant_nursery/
|-- data/
|   |-- plant_nursery_data.xml
|   |-- plant_nursery_demo.xml
|   |-- mail_data.xml
```

Concerning *controllers*, generally all controllers belong to a single controller contained in a file named **<module_name>.py** . An old convention in Odoo is to name this file **main.py** but it is considered as outdated. If you need to inherit an existing controller from another module do it in **<inherited_module_name>.py** . For example adding portal controller in an application is done in **portal.py** .

```
addons/plant_nursery/
|-- controllers/
|   |-- plant_nursery.py
|   |-- portal.py (inheriting portal/controllers/portal.py)
|   |-- main.py (deprecated, replaced by plant_nursery.py)
```

Concerning *static files*, Javascript files follow globally the same logic as python models. Each component should be in its own file with a meaningful name. For instance, the activity widgets are located in **activity.js** of mail module. Subdirectories can also be created to structure the 'package' (see web module for more details). The same logic should be applied for the templates of JS widgets (static XML files) and for their styles (scss files). Don't link data (image, libraries) outside Odoo: do not use an URL to an image but copy it in the codebase instead.
Concerning *wizards*, naming convention is the same of for python models: **<transient>.py** and **<transient>_views.xml** . Both are put in the wizard directory. This naming comes from old odoo applications using the wizard keyword for transient models.

```
addons/plant_nursery/
|-- wizard/
|   |-- make_plant_order.py
|   |-- make_plant_order_views.xml
```

Concerning *statistics reports* done with python / SQL views and classic views naming is the following :

```
addons/plant_nursery/
|-- report/
|   |-- plant_order_report.py
|   |-- plant_order_report_views.xml
```

Concerning *printable reports* which contain mainly data preparation and Qweb templates naming is the following :

```
addons/plant_nursery/
|-- report/
|   |-- plant_order_reports.xml (report actions, paperformat, ...)
|   |-- plant_order_templates.xml (xml report templates)
```

The complete tree of our Odoo module therefore looks like

```
addons/plant_nursery/
|-- __init__.py
|-- __manifest__.py
|-- controllers/
|   |-- __init__.py
|   |-- plant_nursery.py
|   |-- portal.py
|-- data/
|   |-- plant_nursery_data.xml
|   |-- plant_nursery_demo.xml
|   |-- mail_data.xml
|-- models/
|   |-- __init__.py
|   |-- plant_nursery.py
|   |-- plant_order.py
|   |-- res_partner.py
|-- report/
|   |-- __init__.py
|   |-- plant_order_report.py
|   |-- plant_order_report_views.xml
|   |-- plant_order_reports.xml (report actions, paperformat, ...)
|   |-- plant_order_templates.xml (xml report templates)
|-- security/
|   |-- ir.model.access.csv
|   |-- plant_nursery_groups.xml
|   |-- plant_nursery_security.xml
|   |-- plant_order_security.xml
|-- static/
|   |-- img/
|   |   |-- my_little_kitten.png
|   |   |-- troll.jpg
|   |-- lib/
|   |   |-- external_lib/
|   |-- src/
|   |   |-- js/
|   |   |   |-- widget_a.js
|   |   |   |-- widget_b.js
|   |   |-- scss/
|   |   |   |-- widget_a.scss
|   |   |   |-- widget_b.scss
|   |   |-- xml/
|   |   |   |-- widget_a.xml
|   |   |   |-- widget_a.xml
|-- views/
|   |-- assets.xml
|   |-- plant_nursery_menus.xml
|   |-- plant_nursery_views.xml
|   |-- plant_nursery_templates.xml
|   |-- plant_order_views.xml
|   |-- plant_order_templates.xml
|   |-- res_partner_views.xml
|-- wizard/
|   |--make_plant_order.py
|   |--make_plant_order_views.xml
```

> File names should only contain `[a-z0-9_]` (lowercase alphanumerics and `_`)

> **⚠ Warning**
> Use correct file permissions : folder 755 and file 644.

XML files

Format

To declare a record in XML, the **record** notation (using *<record>*) is recommended:

- Place `id` attribute before `model`
- For field declaration, `name` attribute is first. Then place the *value* either in the `field` tag, either in the `eval` attribute, and finally other attributes (widget, options, ...) ordered by importance.
- Try to group the record by model. In case of dependencies between action/menu/views, this convention may not be applicable.
- Use naming convention defined at the next point
- The tag *<data>* is only used to set not-updatable data with `noupdate=1` . If there is only not-updatable data in the file, the `noupdate=1` can be set on the `<odoo>` tag and do not set a `<data>` tag.

```
<record id="view_id" model="ir.ui.view">
    <field name="name">view.name</field>
    <field name="model">object_name</field>
    <field name="priority" eval="16"/>
    <field name="arch" type="xml">
        <tree>
            <field name="my_field_1"/>
            <field name="my_field_2" string="My Label" widget="statusbar" statusbar_visi
        </tree>
    </field>
</record>
```

Odoo supports custom tags acting as syntactic sugar:

- menuitem: use it as a shortcut to declare a `ir.ui.menu`
- template: use it to declare a QWeb View requiring only the `arch` section of the view.
- report: use to declare a report action (actions.html#reference-actions-report)
- act_window: use it if the record notation can't do what you want

The 4 first tags are preferred over the *record* notation.

XML IDs and naming

Security, View and Action

Use the following pattern :

- For a menu: **`<model_name>_menu`** , or **`<model_name>_menu_do_stuff`** for submenus.
- For a view: **`<model_name>_view_<view_type>`** , where *view_type* is **`kanban`** , **`form`** , **`tree`** , **`search`** , ...
- For an action: the main action respects **`<model_name>_action`** . Others are suffixed with **`_<detail>`** , where *detail* is a lowercase string briefly explaining the action. This is used only if multiple actions are declared for the model.
- For window actions: suffix the action name by the specific view information like **`<model_name>_action_view_<view_type>`** .
- For a group: **`<module_name>_group_<group_name>`** where *group_name* is the name of the group, generally 'user', 'manager', ...
- For a rule: **`<model_name>_rule_<concerned_group>`** where *concerned_group* is the short name of the concerned group ('user' for the 'model_name_group_user', 'public' for public user, 'company' for multi-company rules, ...).

Name should be identical to xml id with dots replacing underscores. Actions should have a real naming as it is used as display name.

```
<!-- views  -->
<record id="model_name_view_form" model="ir.ui.view">
    <field name="name">model.name.view.form</field>
    ...
</record>

<record id="model_name_view_kanban" model="ir.ui.view">
    <field name="name">model.name.view.kanban</field>
    ...
</record>

<!-- actions -->
<record id="model_name_action" model="ir.act.window">
    <field name="name">Model Main Action</field>
    ...
</record>

<record id="model_name_action_child_list" model="ir.actions.act_window">
    <field name="name">Model Access Childs</field>
</record>

<!-- menus and sub-menus -->
<menuitem
    id="model_name_menu_root"
    name="Main Menu"
    sequence="5"
/>
<menuitem
    id="model_name_menu_action"
    name="Sub Menu 1"
    parent="module_name.module_name_menu_root"
    action="model_name_action"
    sequence="10"
/>

<!-- security -->
<record id="module_name_group_user" model="res.groups">
    ...
</record>

<record id="model_name_rule_public" model="ir.rule">
    ...
</record>

<record id="model_name_rule_company" model="ir.rule">
    ...
</record>
```

Inheriting XML

Xml Ids of inheriting views should use the same ID as the original record. It helps finding all inheritance at a glance. As final Xml Ids are prefixed by the module that creates them there is no overlap.

Naming should contain an `.inherit.{details}` suffix to ease understanding the override purpose when looking at its name.

```
<record id="model_view_form" model="ir.ui.view">
    <field name="name">model.view.form.inherit.module2</field>
    <field name="inherit_id" ref="module1.model_view_form"/>
    ...
</record>
```

New primary views do not require the inherit suffix as those are new records based upon the first one.

```
<record id="module2.model_view_form" model="ir.ui.view">
    <field name="name">model.view.form.module2</field>
    <field name="inherit_id" ref="module1.model_view_form"/>
    <field name="mode">primary</field>
    ...
</record>
```

Python

> **⚠ Warning**
> Do not forget to read the **Security Pitfalls (security.html#reference-security-pitfalls)** section as well to write secure code.

PEP8 options

Using a linter can help show syntax and semantic warnings or errors. Odoo source code tries to respect Python standard, but some of them can be ignored.

E501: line too long

E301: expected 1 blank line, found 0

E302: expected 2 blank lines, found 1

Imports

The imports are ordered as

1. External libraries (one per line sorted and split in python stdlib)
2. Imports of `odoo`
3. Imports from Odoo modules (rarely, and only if necessary)

Inside these 3 groups, the imported lines are alphabetically sorted.

```
# 1 : imports of python lib
import base64
import re
import time
from datetime import datetime
# 2 : imports of odoo
import odoo
from odoo import api, fields, models, _ # alphabetically ordered
from odoo.tools.safe_eval import safe_eval as eval
# 3 : imports from odoo addons
from odoo.addons.website.models.website import slug
from odoo.addons.web.controllers.main import login_redirect
```

Idiomatics of Programming (Python)

Each python file should have `# -*- coding: utf-8 -*-` as first line.

Always favor *readability* over *conciseness* or using the language features or idioms.

Don't use `.clone()`

```
# bad
new_dict = my_dict.clone()
new_list = old_list.clone()
# good
new_dict = dict(my_dict)
new_list = list(old_list)
```

Python dictionary : creation and update

```
# -- creation empty dict
my_dict = {}
my_dict2 = dict()

# -- creation with values
# bad
my_dict = {}
my_dict['foo'] = 3
my_dict['bar'] = 4
# good
my_dict = {'foo': 3, 'bar': 4}

# -- update dict
# bad
my_dict['foo'] = 3
my_dict['bar'] = 4
my_dict['baz'] = 5
# good
my_dict.update(foo=3, bar=4, baz=5)
my_dict = dict(my_dict, **my_dict2)
```

Use meaningful variable/class/method names

Useless variable : Temporary variables can make the code clearer by giving names to objects, but that doesn't mean you should create temporary variables all the time:

```
# pointless
schema = kw['schema']
params = {'schema': schema}
# simpler
params = {'schema': kw['schema']}
```

Multiple return points are OK, when they're simpler

```
# a bit complex and with a redundant temp variable
def axes(self, axis):
        axes = []
        if type(axis) == type([]):
                axes.extend(axis)
        else:
                axes.append(axis)
        return axes

 # clearer
def axes(self, axis):
        if type(axis) == type([]):
                return list(axis) # clone the axis
        else:
                return [axis] # single-element list
```

Know your builtins : You should at least have a basic understanding of all the Python builtins (http://docs.python.org/library/functions.html (http://docs.python.org/library/functions.html))

```
value = my_dict.get('key', None) # very very redundant
value = my_dict.get('key') # good
```

Also, `if 'key' in my_dict` and `if my_dict.get('key')` have very different meaning, be sure that you're using the right one.

Learn list comprehensions : Use list comprehension, dict comprehension, and basic manipulation using `map`, `filter`, `sum`, ... They make the code easier to read.

```
# not very good
cube = []
for i in res:
        cube.append((i['id'],i['name']))
# better
cube = [(i['id'], i['name']) for i in res]
```

Collections are booleans too : In python, many objects have "boolean-ish" value when evaluated in a boolean context (such as an if). Among these are collections (lists, dicts, sets, ...) which are "falsy" when empty and "truthy" when containing items:

```
bool([]) is False
bool([1]) is True
bool([False]) is True
```

So, you can write `if some_collection:` instead of `if len(some_collection):`.

Iterate on iterables

```
# creates a temporary list and looks bar
for key in my_dict.keys():
        "do something..."
# better
for key in my_dict:
        "do something..."
# accessing the key,value pair
for key, value in my_dict.items():
        "do something..."
```

Use dict.setdefault

```
# longer.. harder to read
values = {}
for element in iterable:
    if element not in values:
        values[element] = []
    values[element].append(other_value)

# better.. use dict.setdefault method
values = {}
for element in iterable:
    values.setdefault(element, []).append(other_value)
```

As a good developer, document your code (docstring on methods, simple comments for tricky part of code)

In additions to these guidelines, you may also find the following link interesting: http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html (http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html) (a little bit outdated, but quite relevant)

Programming in Odoo

Avoid to create generators and decorators: only use the ones provided by the Odoo API.

As in python, use `filtered`, `mapped`, `sorted`, ... methods to ease code reading and performance.

Make your method work in batch

When adding a function, make sure it can process multiple records by iterating on self to treat each record.

```
def my_method(self)
    for record in self:
        record.do_cool_stuff()
```

For performance issue, when developing a 'stat button' (for instance), do not perform a `search` or a `search_count` in a loop. It is recommended to use `read_group` method, to compute all value in only one request.

```
def _compute_equipment_count(self):
""" Count the number of equipment per category """
    equipment_data = self.env['hr.equipment'].read_group([('category_id', 'in', self.ids
    mapped_data = dict([(m['category_id'][0], m['category_id_count']) for m in equipment
    for category in self:
        category.equipment_count = mapped_data.get(category.id, 0)
```

Propagate the context

The context is a `frozendict` that cannot be modified. To call a method with a different context, the `with_context` method should be used :

```
records.with_context(new_context).do_stuff() # all the context is replaced
records.with_context(**additionnal_context).do_other_stuff() # additionnal_context value
```

> **⚠ Warning**
>
> Passing parameter in context can have dangerous side-effects.
>
> Since the values are propagated automatically, some unexpected behavior may appear. Calling `create()` method of a model with *default_my_field* key in context will set the default value of *my_field* for the concerned model. But if during this creation, other objects (such as sale.order.line, on sale.order creation) having a field name *my_field* are created, their default value will be set too.

If you need to create a key context influencing the behavior of some object, choice a good name, and eventually prefix it by the name of the module to isolate its impact. A good example are the keys of `mail` module : *mail_create_nosubscribe*, *mail_notrack*, *mail_notify_user_signature*, ...

Think extendable

Functions and methods should not contain too much logic: having a lot of small and simple methods is more advisable than having few large and complex methods. A good rule of thumb is to split a method as soon as it has more than one responsibility (see http://en.wikipedia.org/wiki/Single_responsibility_principle (http://en.wikipedia.org/wiki/Single_responsibility_principle)).

Hardcoding a business logic in a method should be avoided as it prevents to be easily extended by a submodule.

```
# do not do this
# modifying the domain or criteria implies overriding whole method
def action(self):
    ...  # long method
    partners = self.env['res.partner'].search(complex_domain)
    emails = partners.filtered(lambda r: arbitrary_criteria).mapped('email')

# better but do not do this either
# modifying the logic forces to duplicate some parts of the code
def action(self):
    ...
    partners = self._get_partners()
    emails = partners._get_emails()

# better
# minimum override
def action(self):
    ...
    partners = self.env['res.partner'].search(self._get_partner_domain())
    emails = partners.filtered(lambda r: r._filter_partners()).mapped('email')
```

The above code is over extendable for the sake of example but the readability must be taken into account and a tradeoff must be made.
Also, name your functions accordingly: small and properly named functions are the starting point of readable/maintainable code and tighter documentation.
This recommendation is also relevant for classes, files, modules and packages. (See also http://en.wikipedia.org/wiki/Cyclomatic_complexity (http://en.wikipedia.org/wiki/Cyclomatic_complexity))

Never commit the transaction

The Odoo framework is in charge of providing the transactional context for all RPC calls. The principle is that a new database cursor is opened at the beginning of each RPC call, and committed when the call has returned, just before transmitting the answer to the RPC client, approximately like this:

```
def execute(self, db_name, uid, obj, method, *args, **kw):
    db, pool = pooler.get_db_and_pool(db_name)
    # create transaction cursor
    cr = db.cursor()
    try:
        res = pool.execute_cr(cr, uid, obj, method, *args, **kw)
        cr.commit() # all good, we commit
    except Exception:
        cr.rollback() # error, rollback everything atomically
        raise
    finally:
        cr.close() # always close cursor opened manually
    return res
```

If any error occurs during the execution of the RPC call, the transaction is rolled back atomically, preserving the state of the system.
Similarly, the system also provides a dedicated transaction during the execution of tests suites, so it can be rolled back or not depending on the server startup options.
The consequence is that if you manually call `cr.commit()` anywhere there is a very high chance that you will break the system in various ways, because you will cause partial commits, and thus partial and unclean rollbacks, causing among others:

1. inconsistent business data, usually data loss
2. workflow desynchronization, documents stuck permanently
3. tests that can't be rolled back cleanly, and will start polluting the database, and triggering error (this is true even if no error occurs during the transaction)

Here is the very simple rule:

> You should **NEVER** call `cr.commit()` yourself, **UNLESS** you have created your own database cursor explicitly! And the situations where you need to do that are exceptional!
>
> And by the way if you did create your own cursor, then you need to handle error cases and proper rollback, as well as properly close the cursor when you're done with it.

And contrary to popular belief, you do not even need to call `cr.commit()` in the following situations: - in the `_auto_init()` method of an *models.Model* object: this is taken care of by the addons initialization method, or by the ORM transaction when creating custom models - in reports: the `commit()` is handled by the framework too, so you can update the database even from within a report - within *models.Transient* methods: these methods are called exactly like regular *models.Model* ones, within a transaction and with the corresponding `cr.commit()/rollback()` at the end - etc. (see general rule above if you have in doubt!)
All `cr.commit()` calls outside of the server framework from now on must have an **explicit comment** explaining why they are absolutely necessary, why they are indeed correct, and why they do not break the transactions. Otherwise they can and will be removed !

Use translation method correctly

Odoo uses a GetText-like method named “underscore” `_( )` to indicate that a static string used in the code needs to be translated at runtime using the language of the context. This pseudo-method is accessed within your code by importing as follows:

```
from odoo import _
```

A few very important rules must be followed when using it, in order for it to work and to avoid filling the translations with useless junk.
Basically, this method should only be used for static strings written manually in the code, it will not work to translate field values, such as Product names, etc. This must be done instead using the translate flag on the corresponding field.
The method accepts optional positional or named parameter The rule is very simple: calls to the underscore method should always be in the form `_('literal string')` and nothing else:

```
# good: plain strings
error = _('This record is locked!')

# good: strings with formatting patterns included
error = _('Record %s cannot be modified!', record)

# ok too: multi-line literal strings
error = _("""This is a bad multiline example
             about record %s!""", record)
error = _('Record %s cannot be modified' \
          'after being validated!', record)

# bad: tries to translate after string formatting
#      (pay attention to brackets!)
# This does NOT work and messes up the translations!
error = _('Record %s cannot be modified!' % record)

# bad: formatting outside of translation
# This won't benefit from fallback mechanism in case of bad translation
error = _('Record %s cannot be modified!') % record

# bad: dynamic string, string concatenation, etc are forbidden!
# This does NOT work and messes up the translations!
error = _("'" + que_rec['question'] + "' \n")

# bad: field values are automatically translated by the framework
# This is useless and will not work the way you think:
error = _("Product %s is out of stock!") % _(product.name)
# and the following will of course not work as already explained:
error = _("Product %s is out of stock!" % product.name)

# Instead you can do the following and everything will be translated,
# including the product name if its field definition has the
# translate flag properly set:
error = _("Product %s is not available!", product.name)
```

Also, keep in mind that translators will have to work with the literal values that are passed to the underscore function, so please try to make them easy to understand and keep spurious characters and formatting to a minimum. Translators must be aware that formatting patterns such as `%s` or `%d`, newlines, etc. need to be preserved, but it's important to use these in a sensible and obvious manner:

```
# Bad: makes the translations hard to work with
error = "'" + question + _("' \nPlease enter an integer value ")

# Ok (pay attention to position of the brackets too!)
error = _("Answer to question %s is not valid.\n" \
          "Please enter an integer value.", question)

# Better
error = _("Answer to question %(title)s is not valid.\n" \
          "Please enter an integer value.", title=question)
```

In general in Odoo, when manipulating strings, prefer `%` over `.format()` (when only one variable to replace in a string), and prefer `%(varname)` instead of position (when multiple variables have to be replaced). This makes the translation easier for the community translators.

Symbols and Conventions

Model name (using the dot notation, prefix by the module name) :

When defining an Odoo Model : use singular form of the name (*res.partner* and *sale.order* instead of *res.partnerS* and *saleS.orderS*)

When defining an Odoo Transient (wizard) : use `<related_base_model>.<action>` where *related_base_model* is the base model (defined in *models/*) related to the transient, and *action* is the short name of what the transient do. Avoid the *wizard* word. For instance : `account.invoice.make`, `project.task.delegate.batch`, ...

When defining *report* model (SQL views e.i.) : use `<related_base_model>.report.<action>`, based on the Transient convention.

Odoo Python Class : use camelcase (Object-oriented style).

```
class AccountInvoice(models.Model):
    ...
```

Variable name :

use camelcase for model variable

use underscore lowercase notation for common variable.

suffix your variable name with *_id* or *_ids* if it contains a record id or list of id. Don't use `partner_id` to contain a record of res.partner

```
Partner = self.env['res.partner']
partners = Partner.browse(ids)
partner_id = partners[0].id
```

`One2Many` and `Many2Many` fields should always have *_ids* as suffix (example: sale_order_line_ids)

`Many2One` fields should have *_id* as suffix (example : partner_id, user_id, ...)

Method conventions

Compute Field : the compute method pattern is *_compute_<field_name>*

Search method : the search method pattern is *_search_<field_name>*

Default method : the default method pattern is *_default_<field_name>*

Selection method: the selection method pattern is *_selection_<field_name>*

Onchange method : the onchange method pattern is *_onchange_<field_name>*

Constraint method : the constraint method pattern is *_check_<constraint_name>*

Action method : an object action method is prefix with *action_*. Since it uses only one record, add `self.ensure_one()` at the beginning of the method.

In a Model attribute order should be

1. Private attributes (`_name` , `_description` , `_inherit` , ...)
2. Default method and `_default_get`
3. Field declarations
4. Compute, inverse and search methods in the same order as field declaration
5. Selection method (methods used to return computed values for selection fields)
6. Constrains methods (`@api.constrains`) and onchange methods (`@api.onchange`)
7. CRUD methods (ORM overrides)
8. Action methods
9. And finally, other business methods.

```
class Event(models.Model):
    # Private attributes
    _name = 'event.event'
    _description = 'Event'

    # Default methods
    def _default_name(self):
        ...

    # Fields declaration
    name = fields.Char(string='Name', default=_default_name)
    seats_reserved = fields.Integer(oldname='register_current', string='Reserved Seats',
        store=True, readonly=True, compute='_compute_seats')
    seats_available = fields.Integer(oldname='register_avail', string='Available Seats',
        store=True, readonly=True, compute='_compute_seats')
    price = fields.Integer(string='Price')
    event_type = fields.Selection(string="Type", selection='_selection_type')

    # compute and search fields, in the same order of fields declaration
    @api.depends('seats_max', 'registration_ids.state', 'registration_ids.nb_register')
    def _compute_seats(self):
        ...

    @api.model
    def _selection_type(self):
        return []

    # Constraints and onchanges
    @api.constrains('seats_max', 'seats_available')
    def _check_seats_limit(self):
        ...

    @api.onchange('date_begin')
    def _onchange_date_begin(self):
        ...

    # CRUD methods (and name_get, name_search, ...) overrides
    def create(self, values):
        ...

    # Action methods
    def action_validate(self):
        self.ensure_one()
        ...

    # Business methods
    def mail_user_confirm(self):
        ...
```

Javascript and CSS

Static files organization

Odoo addons have some conventions on how to structure various files. We explain here in more details how web assets are supposed to be organized.

The first thing to know is that the Odoo server will serve (statically) all files located in a *static/* folder, but prefixed with the addon name. So, for example, if a file is located in *addons/web/static/src/js/some_file.js*, then it will be statically available at the url *your-odoo-server.com/web/static/src/js/some_file.js*

The convention is to organize the code according to the following structure:

- *static*: all static files in general
  - *static/lib*: this is the place where js libs should be located, in a sub folder. So, for example, all files from the *jquery* library are in *addons/web/static/lib/jquery*
  - *static/src*: the generic static source code folder
    - *static/src/css*: all css files
    - *static/src/fonts*
    - *static/src/img*
    - *static/src/js*
      - *static/src/js/tours*: end user tour files (tutorials, not tests)
    - *static/src/scss*: scss files
    - *static/src/xml*: all qweb templates that will be rendered in JS
  - *static/tests*: this is where we put all test related files.
    - *static/tests/tours*: this is where we put all tour test files (not tutorials).

Javascript coding guidelines

- `use strict;` is recommended for all javascript files
- Use a linter (jshint, ...)
- Never add minified Javascript Libraries
- Use camelcase for class declaration

More precise JS guidelines are detailed in the github wiki (https://github.com/odoo/odoo/wiki/Javascript-coding-guidelines). You may also have a look at existing API in Javascript by looking Javascript References.

CSS coding guidelines

- Prefix all your classes with *o_<module_name>* where *module_name* is the technical name of the module ('sale', 'im_chat', ...) or the main route reserved by the module (for website module mainly, i.e. : 'o_forum' for *website_forum* module). The only exception for this rule is the webclient: it simply uses *o_* prefix.

- Avoid using *id* tag
- Use Bootstrap native classes
- Use underscore lowercase notation to name class

Git

Configure your git

Based on ancestral experience and oral tradition, the following things go a long way towards making your commits more helpful:

- Be sure to define both the user.email and user.name in your local git config

```
git config --global <var> <value>
```

- Be sure to add your full name to your Github profile here. Please feel fancy and add your team, avatar, your favorite quote, and whatnot ;-)

Commit message structure

Commit message has four parts: tag, module, short description and full description. Try to follow the preferred structure for your commit messages

```
[TAG] module: describe your change in a short sentence (ideally < 50 chars)

Long version of the change description, including the rationale for the change,
or a summary of the feature being introduced.

Please spend a lot more time describing WHY the change is being done rather
than WHAT is being changed. This is usually easy to grasp by actually reading
the diff. WHAT should be explained only if there are technical choices
or decision involved. In that case explain WHY this decision was taken.

End the message with references, such as task or bug numbers, PR numbers, and
OPW tickets, following the suggested format:
task-123 (related to task)
Fixes #123  (close related issue on Github)
Closes #123  (close related PR on Github)
opw-123 (related to ticket)
```

Tag and module name

Tags are used to prefix your commit. They should be one of the following

- **[FIX]** for bug fixes: mostly used in stable version but also valid if you are fixing a recent bug in development version;

[REF] for refactoring: when a feature is heavily rewritten;

[ADD] for adding new modules;

[REM] for removing resources: removing dead code, removing views, removing modules, ...;

[REV] for reverting commits: if a commit causes issues or is not wanted reverting it is done using this tag;

[MOV] for moving files: use git move and do not change content of moved file otherwise Git may loose track and history of the file; also used when moving code from one file to another;

[REL] for release commits: new major or minor stable versions;

[IMP] for improvements: most of the changes done in development version are incremental improvements not related to another tag;

[MERGE] for merge commits: used in forward port of bug fixes but also as main commit for feature involving several separated commits;

[CLA] for signing the Odoo Individual Contributor License;

[I18N] for changes in translation files;

After tag comes the modified module name. Use the technical name as functional name may change with time. If several modules are modified, list them or use various to tell it is cross-modules. Unless really required or easier avoid modifying code across several modules in the same commit. Understanding module history may become difficult.

Commit message header

After tag and module name comes a meaningful commit message header. It should be self explanatory and include the reason behind the change. Do not use single words like “bugfix” or “improvements”. Try to limit the header length to about 50 characters for readability.
Commit message header should make a valid sentence once concatenated with **`if applied, this commit will <header>`**. For example **`[IMP] base: prevent to archive users linked to active partners`** is correct as it makes a valid sentence **`if applied, this commit will prevent users to archive...`**.

Commit message full description

In the message description specify the part of the code impacted by your changes (module name, lib, transversal object, ...) and a description of the changes.
First explain WHY you are modifying code. What is important if someone goes back to your commit in about 4 decades (or 3 days) is why you did it. It is the purpose of the change.

What you did can be found in the commit itself. If there was some technical choices involved it is a good idea to explain it also in the commit message after the why. For Odoo R&D developers “PO team asked me to do it” is not a valid why, by the way.

Please avoid commits which simultaneously impact multiple modules. Try to split into different commits where impacted modules are different. It will be helpful if we need to revert changes in a given module separately.

Don’t hesitate to be a bit verbose. Most people will only see your commit message and judge everything you did in your life just based on those few sentences. No pressure at all.

You spend several hours, days or weeks working on meaningful features. Take some time to calm down and write clear and understandable commit messages.

If you are an Odoo R&D developer the WHY should be the purpose of the task you are working on. Full specifications make the core of the commit message. **If you are working on a task that lacks purpose and specifications please consider making them clear before continuing.**

Finally here are some examples of correct commit messages :

```
[REF] models: use `parent_path` to implement parent_store

 This replaces the former modified preorder tree traversal (MPTT) with the
 fields `parent_left`/`parent_right`[...]

[FIX] account: remove frenglish

 [...]

 Closes #22793
 Fixes #22769

[FIX] website: remove unused alert div, fixes look of input-group-btn

 Bootstrap's CSS depends on the input-group-btn
 element being the first/last child of its parent.
 This was not the case because of the invisible
 and useless alert.
```

> Use the long description to explain the *why* not the *what*, the *what* can be seen in the diff