




10/11/2015

Odoo Development Coding Convention

For Version 8.0



Matiar Rahman
GENWEB2

Table of Contents

1	Python Style Guide	4
1.1	Magic Methods	4
1.2	.clone().....	4
1.3	The "clone and update"	5
1.4	"manual update"	5
1.5	Java dictionary creation.....	6
1.6	"temporary kwargs"	6
1.7	Deprecated methods (formally or informally).....	7
1.8	Useless intermediate variables	7
1.9	3 strikes, and the code's out	8
1.10	Multiple return points are OK, when they're simpler	10
1.11	Try to avoid type-testing	10
1.12	Don't use type if you already know what the type you want is	10
1.13	But really, if you need type testing just use the tools python provides.....	11
1.14	Don't create functions just to call callable	11
1.15	Know your builtins	12
1.16	Learn list comprehensions.....	12
1.17	Learn your standard library	13
1.18	Collections are Booleans too	13
1.19	You can append a single object to a list, it's ok	14
1.20	Add lists into bigger lists.....	14
1.21	Learn your standard library (2)	14
1.22	Iterate on iterables.....	15
1.23	Chaining calls is ok, as long as you don't abuse it (too much).....	15
1.24	Use dict.setdefault	16
1.25	Use constants and avoid magic numbers.....	16
2	Conventions.....	17
2.1	Snake_casing or CamelCasing.....	17
2.2	Imports.....	17
2.3	Model.....	17
2.4	Fields	17

2.5	Translation	17
2.6	API	17
2.7	Exceptions.....	17
2.8	A typical module import would be:	17
2.9	Classes.....	18
2.10	New Exceptions classes	18
2.11	RedirectWarning.....	18
2.12	AccessDenied.....	18
2.13	AccessError	18
2.14	Class MissingError:	18
2.15	DeferredException:.....	19
2.16	Compatibility	19
2.17	Fields	19
2.18	Default or compute	19
2.19	Modifying self in method	19
2.20	Doing thing in dry run	20
2.21	Using Cursor	20
2.22	Displayed Name	20
2.23	Constraints.....	20
2.24	Qweb view or not Qweb view	20
3	Odoo Specific Guidelines	21
3.1	Bazaar is your historian	21
3.2	Call your fish a fish	21
3.3	Do not bypass the ORM.....	21
3.4	No SQL injections, please!	22
3.5	Factor out the code	23
3.6	The infamous context.....	24
3.7	There is better than lambda, sometimes	25
3.8	Keep your methods short/simple when possible	26
3.9	Never commit the transaction.....	26
3.10	Use the gettext method correctly.....	28
4	Automated YAML Tests Guideline	30

4.1	Syntax	30
4.2	Tests are run on the server side.....	30
4.3	Be precise about the goal of the test	31
4.4	Avoid relying on data that can be changed by the user before launching the test	32
4.5	Write things in test that can be easily tested by the YAML system.....	32
4.6	Avoid relying on existing demo data if the user can change it.	33
4.7	Don't check the full text of an exception	34
4.8	Be more functional, explain what the user means to do, not where she clicks	34
4.9	You can use "onchange" calls in your tests, to simulate the client interface.....	35
5	Python.....	36
6	Imports.....	37
7	Idioms	37
8	Symbols.....	38
8.1	Variable name :.....	38
8.2	Method conventions	38
8.3	In a Model attribute order should be.....	39
9	Git	40
9.1	Commit message.....	40

1 Python Style Guide

1.1 Magic Methods

Magic methods (starting *and* ending with two underscores) should *not* have to be called directly unless you're overriding a method of the same name.

Magic methods are used to implement specific protocols and are called for you, either due to operator access or due to some special operation using them:

```
# bad
levels.__contains__(name)
# good
name in levels
# very bad
kw.__setitem__('nodes', nodes)
# much better
kw['nodes'] = nodes
```

1.2 .clone()

Rarely necessary (unless you really have no idea what the type of the variable you're trying to clone is), never necessary for built-in collections: just call the constructor with your existing collection:

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

And don't clone manually, please:

```
# surely you jest!
values = []
for val in view_items:
```

```
values += [val]

# sane
values = list(view_items)
```

1.3 The "clone and update"

The `dict` constructor takes both a single (optional) positional argument (either a dictionary-like object or an iterable of 2-tuples) and an unlimited number of keyword arguments. Thus, you can "merge" two different dictionaries into a third, new, dictionary:

```
# bad
dictionary3 = dictionary.clone()
dictionary3.update(dictionary2)

# worse
dictionary3 = {}
dictionary3.update(d1)
dictionary3.update(d2)

# good
dictionary3 = dict(dictionary1, **dictionary2)
```

You can use those properties for simpler operations, such as cloning an existing dict and (re) setting a key:

```
# no
context = kw.clone()
context['foo'] = 'bar'

# yes
context = dict(kw, foo='bar')
```

1.4 "manual update"

The signature of `dict.update` is the same as `dict()`: a single, optional, positional argument and an unlimited number of keyword arguments.

Thus the following are possible:

merging a dict from another one:

```
# bad
for key, value in other_dict.items():
    my_dict[key] = value

# good
my_dict.update(other_dict)
```

Setting a bunch of keys at the same time:

```
# bad
my_dict['foo'] = 3
my_dict['bar'] = 4
my_dict['baz'] = 5

# good
my_dict.update(
    foo=3,
    bar=4,
    baz=5)
```

1.5 Java dictionary creation

Python isn't java, it has literals:

```
# very very very bad
my_dict = {}
my_dict['foo'] = 3
my_dict['bar'] = 4
my_dict['baz'] = 5

# good
my_dict = {
    'foo': 3,
    'bar': 4,
    'baz': 5
}
```

1.6 "temporary kwargs"

Keyword arguments are a good way to get a bunch of unspecified supplementary arguments if e.g. you just want to forward them:

```
def foo(**kwargs):
    logging.debug('Calling foo with arguments %s', kwargs)
    return bar(**kwargs)
```

Or if you retrieved a ready-made dict (from another function) and want to pass its content to another function or method:

```
sessions = some_function_returning_a_dict_of_sessions()
some_other_function(**sessions)
```

But there is no point whatsoever in creating a dict for the sake of passing it as `**kwargs`, just provide the damn keyword arguments:

```
# waste of time and space
my_dict = {
    'foo': 3,
    'bar': 4,
    'baz': 5
}
some_func(**my_dict)

# good
some_func(foo=3, bar=4, baz=5)
```

1.7 Deprecated methods (formally or informally)

`dict.has_key(key)` is deprecated, please use the `in` operator:

```
# bad
kw.has_key('cross_on_pages')

# good
'cross_on_pages' in kw
```

1.8 Useless intermediate variables

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']}
```

1.9 3 strikes, and the code's out

A bit of redundancy can be accepted: maybe you have to get the content of an axis:

```
col_axes = []
if kw.has_key('col_axis'):
    col_axes = self.axes(kw['col_axis'])
```

and a second one:

```
col_axes = []
if kw.has_key('col_axis'):
    col_axes = self.axes(kw['col_axis'])
page_axes = []
if kw.has_key('page_axis'):
    page_axes = self.axes(kw['page_axis'])
```

But at the third strike, chances are you're going to need the thing again and again. Time to refactor:

```
def get_axis(self, name, kw):
    if name not in kw:
        return []
    return self.axes(kw[name])

#[...]
col_axes = self.get_axis('col_axis', kw)
page_axes = self.get_axis('page_axis', kw)
```

The refactoring could also be an improvement of a method already called (be sure to check where the method is called in order not to break other pieces of code. Or write tests):

```
# from
def axes(self, axis):
    axes = []
    if type(axis) == type([]):
        axes.extend(axis)
    else:
        axes.append(axis)
    return axes

def output(self, **kw):
    col_axes = []
    if kw.has_key('col_axis'):
        col_axes = self.axes(kw['col_axis'])
    page_axes = []
    if kw.has_key('page_axis'):
        page_axes = self.axes(kw['page_axis'])
    cross_on_rows = []
    if kw.has_key('cross_on_rows'):
        cross_on_rows = self.axes(kw['cross_on_rows'])

# to:
def axes(self, axis):
    if axis is None: return []
    axes = []
    if type(axis) == type([]):
        axes.extend(axis)
    else:
        axes.append(axis)
    return axes

def output(self, **kw):
```

```
col_axes = self.axes(kw.get('col_axis'))
page_axes = self.axes(kw.get('page_axis'))
cross_on_rows = self.axes(kw.get('cross_on_rows'))
```

1.10 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
```

1.11 Try to avoid type-testing

Python is a dynamically typed languages, if you don't absolutely need to receive a list, then don't test for a list, just do your stuff (e.g. iterating on it, then caller can provide any kind of iterator or container)

1.12 Don't use type if you already know what the type you want is

The type of a list is `list`, the type of a dict is `dict`:

```
# bad
def axes(self, axis):
```

```

        if type(axis) == type([]): # we already know what the type of [] is
            return list(axis)
        else:
            return [axis]
# good
def axes(self, axis):
    if type(axis) == list:
        return list(axis)
    else:
        return [axis]

```

plus Python types are singletons, so you can just test for identity, it reads better:

```

# better
def axes(self, axis):
    if type(axis) is list:
        return list(axis)
    else:
        return [axis]

```

1.13 But really, if you need type testing just use the tools python provides

The previous piece of code will fail if the caller provided a *subclass* of `list` (which is possible and allowed), because `==` and `is` don't check for subtypes. `isinstance` does:

```

# shiny
def axes(self, axis):
    if isinstance(axis, list):
        return list(axis) # clone the axis
    else:
        return [axis] # single-element list

```

1.14 Don't create functions just to call callable

```
# dumb, ``str`` is already callable
parent_id = map(lambda x: str(x), parent_id.split(','))

# better
parent_id = map(str, parent_id.split(','))
```

1.15 Know your builtins

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

For example, `isinstance` can take more than one type as its second argument, so you could write:

```
def axes(self, axis):
    if isinstance(axis, (list, set, dict)):
        return list(axis)
    else:
        return [axis]
```

Another one is `dict.get`, whose second argument defaults to `None`:

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

# good
value= my_dict.get('key')
```

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.

1.16 Learn list comprehensions

When used correctly, list comprehensions can greatly enhance the quality of a piece of code when mapping and/or filtering collections:

```
# not very good
cube = []

for i in res:
```

```
cube.append((i['id'],i['name']))  
  
# better  
cube = [(i['id'], i['name']) for i in res]
```

But beware: with great power comes great responsibility, and list comprehensions can become overly complex. In that case, either revert back to "normal" `for` loops, extract functions or do your transformation in multiple steps

1.17 Learn your standard library

Python is provided "with batteries included", but these batteries are often criminally underused. Some standard modules to know are `itertools`, `operator` and `collections`, among others (though be careful to note the python version at which functions and objects were introduced, don't break compatibility with the officially supported versions for your tool):

```
# no  
cube = map(lambda i: (i['id'], i['name']), res)  
  
# still no  
cube = [(i['id'], i['name']) for i in res]  
  
# yes, with operator.itemgetter  
cube = map(itemgetter('id', 'name'), res)
```

1.18 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
```

Therefore, no need to call `len`:

```
# redundant  
if len(some_collection):  
    "do something..."  
  
# simpler
```

```
if some_collection:
    "do something..."
```

1.19 You can append a single object to a list, it's ok

```
# no
some_list += [some_item]
# yes
some_list.append(some_item)
# very no
view += [(code, values)]
# yes
view.append((code, values))
```

1.20 Add lists into bigger lists

```
# obscure
my_list = []
my_list += list1
my_list += list2
# simpler
my_list = list1 + list2
```

1.21 Learn your standard library (2)

Itertools is your friend for all things iterable:

```
# ugly
my_list = []
my_list += list1
my_list += list2
for element in my_list:
    "do something..."
# unclear, creates a pointless temporary list
```

```
for element in list1 + list2:
    "do something..."
# says what I mean
for element in itertools.chain(list1, list2):
    "do something..."
```

1.22 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..."
# creates a temporary list
for key, value in my_dict.items():
    "do something..."
# only iterates
for key, value in my_dict.iteritems():
    "do something..."
```

1.23 Chaining calls is ok, as long as you don't abuse it (too much)

```
# what's the point of the ``graph`` temporary variable?
# plus it shadows the ``graph`` module, bad move
graph = graph.Graph(kw)
mdx = ustr(graph.display())
# just as readable
mdx = ustr(grah.Graph(kw).display())
```

NOTE:

yes, here the temporary variable *graph* is redundant but sometimes using such temporary variables simplify code debugging when you want to inspect the variable and you put breakpoint on the single line expression it's difficult to know when to do step-in and step-out.

1.24 Use dict.setdefault

If you need to modify a nested container for example:

```
# 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)
```

1.25 Use constants and avoid magic numbers

```
# bad
limit = 20

# bad
search(cr, uid, ids, domain, limit=20, context=context)
```

You should use a constant, name it correctly, and perhaps add a comment on it explaining where the value comes from. And of course it's cleaner, easier to read and there's only one place to modify. Oh and that is true not just for numbers, but for any literal value that is semantically a constant!

```
# better
DEFAULT_SEARCH_LIMIT = 20 # limit to 20 results due to screen size

search(cr, uid, ids, domain, limit=DEFAULT_LIMIT, context=context)
```

2 Conventions

2.1 Snake_casing or CamelCasing

That was not clear. But it seems that OpenERP SA will continue to use snake case.

2.2 Imports

As discussed with Raphaël Collet. This convention should be the one to use after RC1.

2.3 Model

```
from openerp import models
```

2.4 Fields

```
from openerp import fields
```

2.5 Translation

```
from openerp import _
```

2.6 API

```
from openerp import api
```

2.7 Exceptions

```
from openerp import exceptions
```

2.8 A typical module import would be:

```
from openerp import models, fields, api, _
```

2.9 Classes

Class should be initialized like this:

```
class Toto(models.Model):  
    pass
```

```
class Titi(models.TransientModel):  
    pass
```

2.10 New Exceptions classes

`except_orm` exception is deprecated. We should use `openerp.exceptions.Warning` and subclasses instances

2.11 RedirectWarning

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

Should receive as parameters:

param int action_id:

id of the action where to perform the redirection

param string button_text:

text to put on the button that will trigger the redirection.

2.12 AccessDenied

Login/password error. No message, no traceback.

2.13 AccessError

Access rights error.

2.14 Class MissingError:

Missing record(s)

2.15 DeferredException:

Exception object holding a traceback for asynchronous reporting.

Some RPC calls (database creation and report generation) happen with an initial request followed by multiple, polling requests. This class is used to store the possible exception occurring in the thread serving the first request, and is then sent to a polling request.

2.16 Compatibility

When catching orm exception we should catch both types of exceptions:

```
try:
    pass
except (Warning, except_orm) as exc:
    pass
```

2.17 Fields

Fields should be declared using new fields API. Putting string key is better than using a long property name:

```
class AClass(models.Model):

    name = fields.Char(string="This is a really long long name") # ok
    really_long_long_name = fields.Char()
```

That said the property name must be meaningful. Avoid name like 'nb' etc.

2.18 Default or compute

`compute` option should not be used as a workaround to set default. Default should only be used to provide property initialisation.

That said they may share the same function.

2.19 Modifying self in method

We should never alter self in a Model function. It will break the correlation with current Environment caches.

2.20 Doing thing in dry run

If you use the `do_in_draft` context manager of Environment it will not be committed but only be done in cache.

2.21 Using Cursor

When using cursor you should use current environment cursor:

```
self.env.cr
```

except if you need to use threads:

```
with Environment.manage(): # class function
    env = Environment(cr, uid, context)
```

2.22 Displayed Name

`__name_get` is deprecated.

You should define the `display_name` field with options:

- `compute`
- `inverse`

2.23 Constraints

Should be done using `@api.constrains` decorator in conjunction with the `@api.one` if performance allows it.

2.24 Qweb view or not Qweb view

If no advance behavior is needed on Model view, standard view (non Qweb) should be the preferred choice.

3 Odoo Specific Guidelines

3.1 Bazaar is your historian

Do not comment out code if you want to disable it. The code is versioned in Bazaar, and regardless of your opinion about Bazaar, it does not lose the file history.

Leaving comments just makes the code messier and harder to read. And don't worry about making your changes obvious, that's the purpose of `diff`:

```
# no example for this one, code was really removed ;-)
```

3.2 Call your fish a fish

Always give your variables a meaningful name. You may know what it is referring to now, but you won't in 2 months, and others don't either. One-letter variables are acceptable only in lambda expressions and loop indices, or perhaps in pure maths expressions (and even there it doesn't hurt to use a real name):

```
# unclear and misleading

a = {}
ffields = {}

# better

results = {}
selected_fields = {}
```

3.3 Do not bypass the ORM

You should never use the database cursor directly when the ORM can do the same thing! By doing so you are bypassing all the ORM features, possibly the transactions, access rights and so on.

And chances are that you are also making the code harder to read and probably less secure (see also next guideline):

```
# very very wrong
```

```

cr.execute('select id from auction_lots where auction_id in (' +
          ','.join(map(str,ids))+') and state=%s and obj_price>0',
          ('draft',))
auction_lots_ids = [x[0] for x in cr.fetchall()]

# no injection, but still wrong
cr.execute('select id from auction_lots where auction_id in %s \'
          'and state=%s and obj_price>0',
          (tuple(ids),'draft',))
auction_lots_ids = [x[0] for x in cr.fetchall()]

# better
auction_lots_ids = self.search(cr,uid,
                               [('auction_id','in',ids),
                                ('state','=','draft'),
                                ('obj_price','>',0)])

```

3.4 No SQL injections, please!

Care must be taken not to introduce SQL injections vulnerabilities when using manual SQL queries. The vulnerability is present when user input is either incorrectly filtered or badly quoted, allowing an attacker to introduce undesirable clauses to a SQL query (such as circumventing filters or executing UPDATE or DELETE commands).

The best way to be safe is to never, NEVER use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string.

The second reason, which is almost as important, is that it is the job of the database abstraction layer (psycopg2) to decide how to format query parameters, not your job! For example psycopg2 knows that when you pass a list of values it needs to format them as a comma-separated list, enclosed in parentheses!

```

# the following is very bad:
# - it's a SQL injection vulnerability
# - it's unreadable
# - it's not your job to format the list of ids
cr.execute('select distinct child_id from account_account_consol_rel ' +

```

```

        'where parent_id in ('+', '.join(map(str, ids))+')')

# better
cr.execute('SELECT DISTINCT child_id '\
          'FROM account_account_consol_rel '\
          'WHERE parent_id IN %s',
          (tuple(ids),))

```

This is very important, so please be careful also when refactoring, and most importantly do not copy these patterns!

Here is a [memorable example](#) to help you remember what the issue is about (but do not copy the code there).

Before continuing, please be sure to read the online documentation of pycopg2 to learn of to use it properly:

- [The problem with query parameters](#)
- [How to pass parameters with pycopg2](#)
- [Advanced parameter types](#)

3.5 Factor out the code

If you are writing the same code over and over and it is more than one line, then you must factor it out into a reusable function or method:

```

# after writing this multiple times:
terp = get_module_resource(module, '__openerp__.py')
if not os.path.isfile(terp):
    terp = addons.get_module_resource(module, '__terp__.py')
    info = eval(tools.file_open(terp).read())

# make a function out of it
def _load_information_from_description_file(module):
    for filename in ['__openerp__.py', '__terp__.py']:
        description_file = addons.get_module_resource(module, filename)
        if os.path.isfile(description_file):
            return eval(tools.file_open(description_file).read())

```



```
raise Exception('The module %s does not contain a description file!')
```

3.6 The infamous context

Do not use mutable objects as default values for functions, because they are created as constants (evaluated only once), so you will have possible side-effects if you modify them. The usual example of this is the `context` argument to all ORM methods:

```
# bad, this could have side-effects
def spam(eggs, context={}):
    setting = context.get('foo')
    #...

# this is better if your need to use the context
def spam(eggs, context=None):
    if context is None:
        context = {}
    setting = context.get('foo')
    #...
```

Also be careful with boolean tests on lists and maps, because an empty dict, list or tuple will evaluate as `False`:

```
# bad, you shadow the original context if it's empty
def spam(eggs, context=None):
    if not context:
        context = {}
    setting = context.get('foo')
    #...
```

And it's okay if you only need to forward it, you can pass `None` and let the downstream code handle it:

```
# fine
def spam(eggs, context=None):
    setting = get_setting(True, context=context)
```

3.7 There is better than lambda, sometimes

Instead of writing trivial lambda expression to extract items or attributes from a list of data structures, learn to use list comprehension or `operator.itemgetter` and `operator.attrgetter` instead, which are often more readable and faster:

```
# not very readable
partner_tuples = map(lambda x: (x['id'], x['name']), partners)

# better with list comprehension for just one item/attribute
partner_ids = [partner['id'] for partner in partners]

# better with operator for many items/attributes
from operator import itemgetter
# ...
partner_tuples = map(itemgetter('id', 'name'), partners)
```

As of version 6.0 you can also use literal values as defaults for your ORM columns, which means that you can stop writing these:

```
# lots of trivial one-liners in 5.0
_defaults = {
    'active': lambda *x: True,
    'state': lambda *x: 'draft',
}

# much simpler as of 6.0
_defaults = {
```

```
'active': True,  
'state': 'draft',  
}
```

Warning

Be careful with this, because non-callable defaults are only evaluated once! If you want to generate new default values for each record you really need to keep the `lambda` or make it a callable.

The most frequent error is with timestamps, as in the following example:

```
# This will always give the server start time!  
_defaults = {  
    'timestamp': time.strftime('%Y-%m-%d %H:%M:%S'),  
}  
  
# You need to keep it callable, e.g:  
_defaults = {  
    'timestamp': lambda *x: time.strftime('%Y-%m-%d %H:%M:%S'),  
}
```

3.8 Keep your methods short/simple when possible

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
- it is too big to fit on one screen.

Also, name your functions accordingly: small and properly named functions are the starting point of readable/maintainable code and tighter documentation.

3.9 Never commit the transaction

The OpenERP/OpenObject 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:

- inconsistent business data, usually data loss ;
- workflow desynchronization, documents stuck permanently ;
- 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:

Warning

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 `osv.osv` 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 `osv.osv_memory` methods: these methods are called exactly like regular `osv.osv` ones, within a transaction and with the corresponding `cr.commit()/rollback()` at the end ;
- etc. (see general rule above if you have in doubt!)

And another very simple rule:

Warning

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!

3.10 Use the gettext method correctly

OpenERP 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 tools.translate 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 `translateflag` on the corresponding field.

The rule is very simple: calls to the underscore method should *always* be in the form `_('literalstring')` 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: 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)

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

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

4 Automated YAML Tests Guideline

This is a guideline for project managers and developers on automated tests.

4.1 Syntax

Write scenarios using the YAML syntax, not the "cucumber" syntax, so that we can directly integrate these tests in OpenERP modules (and implement them later).

4.2 Tests are run on the server side

As tests are performed server side, you don't need to write things like

```
# wrong
* Given I am logged in as admin user with proper password
* module pos is installed
```

The user is admin by default, so no need to write this in each test file. Instead of saying 'I check that the module pos is installed', just write the name of the module where you will put this test. The first line would then be "module: pos".

4.3 Be precise about the goal of the test

Try to use functional terms instead of object

- This is not good enough:

- I will test the manufacturing order

- An example of a good introduction:

- In order to test the CRM in OpenERP, I will do a customer qualification
- process that begins with the first customer contact (a lead),
- which will be converted to a business opportunity and a partner.

wrong

*In caldav testing of creation of calendar as Admin

better

*In order to test the calendar synchronisation with mobile phones and outlook, I will test the caldav interface for meetings.

wrong:

In order to test the manufacturing order process and modules as an administrator I want to see if the features work correctly.

better:

In order to test the manufacturing of products, I will test the features of the manufacturing order document: consume raw materials, produce finished goods, scrap some products and split in production lots.

wrong:

Testing of Document for ftp server is running and being connected or

not

better:

In order to test the document management system, I will try different operations on the FTP interface, and check their impacts on OpenERP's documents.

4.4 Avoid relying on data that can be changed by the user before launching the test

wrong (mrp):

Then my manufacturing order turned into 'Waiting for goods' state.

And one picking was also generated for my manufacturing order:

'INT/00001'.

This is not a good test, who knows if it will be 'INT/0001' or another sequence 'INT/0002' ? Because the user may have created internal pickings before they installed the mrp module.

wrong (pos):

I press new record from toolbar.

Some default values are filled automatically like:

Company>Tinysprl,Journal>x Sales Journal

You can not test if the company is 'Tiny SPRL'. The user may have configured it to his own company name. It's better to test: "The company is set by default to the company of the admin user."

Take care of these data dependencies, because it may crash the tests simply because the user changed some demo data before launching the tests.

You can rely on demo data defined by the module where you put your test file, but if the demo data are defined by another module, try to either create your own data or find a way to work with ids instead of names.

4.5 Write things in test that can be easily tested by the YAML system

For example, the mail gateway. You can not write this:

```
First I have made one fetchmail rc script using the proper syntax
with specific email server and address information and used crm.lead
as model Then I run the fetchmail command like: fetchmail -f <created
rc file name> The script should exit successfully
```

Because you can not set-up an email pop account to test this.

I would rather do:

```
I have a list of different emails with different encoding and
different kind of attachments stored in the directory test/emails.
I test to pass all these documents through the mailgateway script:
something like:
    for each email file:
        call the script with stdin<this email file
```

If possible call in python directly, not using os.system.

For the FTP, it can be tested by the YAML as you simply have to use the Python FTP client in your yaml code:

```
import ftplib
```

4.6 Avoid relying on existing demo data if the user can change it.

Bad example:

When I pressed '*Confirm Production*' button. Then I could see the Finished Products into Products to Consume with quantity 10.00.

Then my manufacturing order turned into 'Waiting for goods' state. And one picking was also generated for my manufacturing order: 'INT/00001'.

And the following values appeared in the Products to Consume

product_id	product_qty	product_uom	location_id		
[CPU_GEN] Regular processor config	10.00	PCE	Stock		
[HDD1] HDD Seagate 7200.8 80GB	10.00	PCE	Stock		
[TOW1] ATX Mid-size Tower	10.00	PCE	Stock		
[MOU] Mouse	10.00	PCE	Stock		
[KEYA] Keyboard -AZERTY	10.00	PCE	Stock		

For such an example, I would have created a few products and a bom in the test scenario. And test the manufacturing order on these test data.

4.7 Don't check the full text of an exception

Then I got the following error message:

xmlrpclib.Fault: <Fault warning -- Error:

```
Couldn't find bill of material for product: 'Traceback (most recent call last):
File in dispatch
result = ExportService.getService(service_name).dispatch(method, auth, params)
File "/home/uco/workspace/Trunk/openobject-server/bin/service/web_services.py",
line 587, in dispatch
res = fn(db, uid, *params)
File "/home/uco/workspace/Trunk/openobject-server/bin/osv/osv.py", line 64, in w
rapper
self.abortResponse(1, inst.name, inst.exc_type, inst.value)
File "/home/uco/workspace/Trunk/openobject-server/bin/netsvc.py", line 66, in ab
ortResponse
raise Exception("%s -- %s\\n\\n%s"%(origin, description, details))
Exception: warning -- Error

Couldn't find bill of material for product\n'>
```

Simply do::

And it should generate an exception to say that it cannot find a BoM defined for this product.

4.8 Be more functional, explain what the user means to do, not where she clicks

wrong:

I press new record from toolbar of lead's view

Some default values are filled automatically like: priority>Normal,user_id>Administra
tor, state>Draft

```
Then I give some values for lead:
|name|section_id|partner_name|phone|mobile|
|Carrie Helle|Sales Department|Stonage IT|(855) 924-4364|(333) 715-1450|
Then I press the save button from toolbar
The lead is created successfully
```

No need to write the all the data of the form in the English text (phone, mobile, ...). These data will be written in the final YAML, when you implement the test. A better final YAML for the above example should look like this:

```
-
  As I met a new customer in a fair, I create a new lead "Stonage IT"
  to record his data.
-
  !record {model:rcrm.lead, id:partner_carrie}
    name: Stonage IT
    contact_name: Carrie Helle
    phone: (855) 924-4364
    mobile: (333) 715-1450
-
  I check that the state field is set automatically by default.
-
  !assert {model:crm.lead, id:partner_carrie} state
```

4.9 You can use "onchange" calls in your tests, to simulate the client interface

- I create a new sale order by filling the partner. I want addresses to be filled up by the onchange call but I still need to provide dummy addresses (required fields) to allow the record to be created.
- **!record {model: sale.order, id: my_order}:**
partner_id: base.res_partner_asus pricelist_id: product.list0
partner_order_id: base.main_address partner_invoice_id:
base.main_address partner_shipping_id: base.main_address
- I then call the onchange method and update the record with the returned value.
- **!python {model: sale.order}: |**

```
my_order = self.browse(cr, uid, ref('my_order')) value =  
my_order.onchange_partner_id(my_order['partner_id']).get('value', {})  
my_order.write(value)
```

5 Python

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
- E126: continuation line over-indented for hanging indent
- E123: closing bracket does not match indentation of opening bracket's line
- E127: continuation line over-indented for visual indent
- E128: continuation line under-indented for visual indent
- E265: block comment should start with '# '

6 Imports

The imports are ordered as

1. External libraries (one per line sorted and split in python stdlib)
2. Imports of `openerp`
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
# 2 : imports of openerp
import openerp
from openerp import api, fields, models # alphabetically ordered
from openerp.tools.safe_eval import safe_eval as eval
from openerp.tools.translate import _
# 3 : imports from odoo modules
from openerp.addons.website.models.website import slug
from openerp.addons.web.controllers.main import login_redirect
```

7 Idioms

- Prefer % over `.format()`, prefer `%(varname)` instead of position (This is better for translation)
- Try to avoid generators and decorators
- Always favor *Readability* over *conciseness* or using the language features or idioms.
- Use list comprehension, dict comprehension, and basic manipulation using `map`, `filter`, `sum`, ... They make the code easier to read.
- The same applies for recordset methods : use `filtered`, `mapped`, `sorted`, ...
- Each python file should have `# -*- coding: utf-8 -*-` as first line
- Use the `UserError` defined in `openerp.exceptions` instead of overriding `Warning`, or find a more appropriate exception in `exceptions.py`

- Document your code (docstring on methods, simple comments for the tricky part of the code)
- Use meaningful variable/class/method names

8 Symbols

Odoo Python Class : use camelcase for code in api v8, underscore lowercase notation for old api.

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

class account_invoice(osv.osv):
    ...
```

8.1 Variable name :

- use camelcase for model variable
- use underscore lowercase notation for common variable.
- since new API works with record or recordset instead of id list, don't suffix variable name with `_id` or `_ids` if they not contain id or list of id.

```
ResPartner = self.env['res.partner']
partners = ResPartner.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`, ...)

8.2 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>`
- 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_`. Its decorator is `@api.multi`, but since it use only one record, add `self.ensure_one()` at the beginning of the method.

8.3 In a Model attribute order should be

7. Private attributes (`_name`, `_description`, `_inherit`, ...)
8. Default method and `_default_get`
9. Field declarations
10. Compute and search methods in the same order as field declaration
11. Constrains methods (`@api.constrains`) and onchange methods (`@api.onchange`)
12. CRUD methods (ORM overrides)
13. Action methods
14. 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')

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

    # 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
def create(self):
    ...

# Action methods
@api.multi
def action_validate(self):
    self.ensure_one()
    ...

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

9 Git

9.1 Commit message

Prefix your commit with

- **[IMP]** for improvements
- **[FIX]** for bug fixes
- **[REF]** for refactoring
- **[ADD]** for adding new resources
- **[REM]** for removing of resources
- **[MERGE]** for merge commits (only for forward/back-port)
- **[CLA]** for signing the Odoo Individual Contributor License

Then, in the message itself, specify the part of the code impacted by your changes (module name, lib, transversal object, ...) and a description of the changes.

- Always include a meaningful commit message: it should be self explanatory (long enough) including the name of the module that has been changed and the reason behind the change. Do not use single words like "bugfix" or "improvements".
- 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 a module separately).

[FIX] website, website_mail: 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.

[IMP] fields: reduce memory footprint of list/set field attributes

[REF] web: add module system to the web client

This commit introduces a new module system for the javascript code. Instead of using global ...

Note

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