OpenERP 8.0 Python API

Raphael Collet rco@openerp.com

Final Motivations

- More object-oriented style
  - Support for Pythonic style
- Keep request parameters apart
- Different API for specific features
  - Function fields
  - Default values
  - Constraints
- Keep backwards compatibility

Object-oriented style

```
def write(self, cr, uid, ids, values, context=None):
    super(C, self).write(cr, uid, ids, values, context=context)
    for record in self.browse(cr, uid, ids, context=context):
        if record.increment:
            self.write(cr, uid, [record.id], {
                'count': record.count + 1,
            }, context=context)
```

```
@multi
def write(self, values):
    super(C, self).write(values)
    for record in self:
        if record.increment:
            record.count += 1
```

Every class instance is a *recordset*

A recordset is a collection of records; it encapsulates a list of record ids.

```
records = model.browse(ids)             # return a recordset
assert isinstance(records, Model)

print bool(records)                     # test for emptyness
print len(records)                      # size of collection

records1 = records[:3]                  # slicing
records2 = records[-3]                  # selection
records = records1 + records2           # concatenation

for record in records:                  # iteration returns
    assert isinstance(record, Model)    # recordsets, too
    assert len(record) == 1
```

Recordsets to access data

Get/set a field on a recordset: do it on its *first* record.

```
records = model.browse(ids)

for record in records:
    value = record.name
    record.name = (value or '') + '!'

for record in records:
    value = record['name']
    record['name'] = (value or '') + '!'

assert records.name == records[0].name

# relational fields return recordsets
assert isinstance(records.parent_id, Model)
```

Compatible with old browse records.

Recordsets to access data

An empty recordset behaves as a *null instance*.

```
records = model.browse()
assert len(records) == 0

# data fields
assert records.name == False

# relational fields return empty recordsets
assert isinstance(records.parent_id, Model)
assert not records.parent_id

# no effect
records.name = 'Foo'
```

Model methods are called on recordsets

```
model = registry['res.partner']
assert isinstance(model, Model)

# get a recordset
records = model.create(values)
records = model.search(domain)

# perform operation on records
records.write({'name': 'Foo'})
result = records.read()

# call model method
defaults = records.default_get(['name'])
```

Method decorators: **`@model`**

`self` represents the *model*, its contents is irrelevant.

```
@model
def default_get(self, field_names):
    ...

def default_get(self, cr, uid, field_names, context=None):
    ...
```

```
stuff = model.browse(ids)

stuff.default_get(['name'])
model.default_get(cr, uid, ['name'], context=context)
```

Backwards compatible!

Method decorators: **`@multi`**

`self` is the recordset on which the operation applies.

```
@multi
def augment(self, arg):
    for rec in self:
        rec.value += arg

def augment(self, cr, uid, ids, arg, context=context):
    ...
```

```
stuff = model.browse(ids)

stuff.augment(42)
model.augment(cr, uid, ids, 42, context=context)
```

Backwards compatible!

Method decorators: **@one**

Automatic loop; `self` is a *singleton*.

```
@one
def name_get(self):
    return (self.id, self.name)

def name_get(self, cr, uid, ids, context=None):
    recs = self.browse(cr, uid, ids, context=context)
    return [(rec.id, rec.name) for rec in recs]
```

```
stuff = model.browse(ids)

stuff.name_get()
model.name_get(cr, uid, ids, context=context)
```

Would become the default.

Method decorators: **@returns**

Automatic browse/unbrowse result.

```
@returns('res.partner')
def current_partner(self):
    return registry['res.partner'].browse(42)

@returns('self')
def search(self, cr, uid, domain, ...):
    return range(42)
```

```
# old-style calls return record ids
ids = model.current_partner(cr, uid, context=context)
ids = model.search(cr, uid, domain, context=context)

# new-style calls return a recordset
records = model.current_partner()
records = model.search(domain)
```

Encapsulation: **`scope(cr, uid, context)`**

Scopes are nestable with statement `with`.

```
# the scope object is a proxy to the current scope
from openerp import scope

with scope(cr, uid, context):
    records.write({'name': 'Fool'})

    with scope(lang='fr'):
        # modifies French translation
        records.write({'name': 'Fool'})

        with scope.SUDO():
            # write as superuser (with lang 'fr')
            company.write({'name': 'Ma Compagnie'})

            # retrieve parameters by deconstruction
            cr, uid, context = scope
```

Every recordset is attached to a scope

Getting/setting a field is done in the attached scope.

```
with scope(lang='en'):

    with scope(lang='fr'):
        record1 = model.browse(id)  # attached to French scope
        print record1.name          # French translation

    print record1.name              # still French translation!
    record1.name = 'Jean-Pierre'    # modify French translation

    print record1.partner.name      # still French translation

    record2 = record1.scoped()      # attached to English scope
    print record2.name              # English translation
```

Each scope carries a *cache* for records.

Automatic cache invalidation

No need to "rebrowse" records after an update.

```
record = model.browse(42)
assert record.name == 'Foo'

# simple update
record.write({'name': 'Bar'})
assert record.name == 'Bar'

# update of many2one field -> one2many invalidated
parent = record.parent_id
record.parent_id = another_record
assert record not in parent.child_ids

# explicit invalidation, in case you need it
record.invalidate_cache(['name', 'parent_id'], ids)
record.invalidate_cache(['name', 'parent_id'])
record.invalidate_cache()
```

Declaring fields

```
class res_partner(Model):
    name = fields.Char(required=True)
    customer = fields.Boolean(help="Check this box if ...")

    parent_id = fields.Many2one('res.partner',
                    string='Related Company')
    child_ids = fields.One2many('res.partner', 'parent_id',
                    string='Contacts')

    display_name = fields.Char('Name', compute='_display_name',
                        store=False)

    @one
    @depends('name', 'email')
    def _display_name(self):
        self.display_name = self.name
        if self.email:
            self.display_name += " <%s>" % self.email
```

Function fields

Compute method must assign the field(s) on records.

```
class sale_order(Model):
    amount = fields.Float(compute='_amounts')
    taxes  = fields.Float(compute='_amounts')
    total  = fields.Float(compute='_amounts')

    @multi
    @depends('lines.amount', 'lines.taxes')
    def _amounts(self):
        for order in self:
            order.amount = sum(line.amount for line in order.lines)
            order.taxes = sum(line.taxes for line in order.lines)
            order.total = order.amount + order.taxes
```

Method decorators: **@depends**

Required for automating

- cache invalidation
- recomputation of stored fields

```
@one
@depends('product_id.price', 'quantity')
def _amounts(self):
    self.amount = self.product_id.price * self.quantity
```

No need for complex "store" parameter anymore!

Function fields with inverse

```
foo = fields.Integer()
bar = fields.Integer(compute='_get_bar', inverse='_set_bar',
                     search='_search_bar')

@one
@depends('foo')
def _get_bar(self):
    self.bar = self.foo + 1

@one
def _set_bar(self):
    self.foo = self.bar - 1

@model
def _search_bar(self, operator, value):
    ...     # return some domain on 'foo'
```

Currently under development.

Default values

Same mechanism as function fields

compute method has no dependency

```
from openerp import scope

class my_stuff(Model):
    company_id = fields.Boolean(compute='_default_company')

    @one
    def _default_company(self):
        self.company_id = scope.user.company_id
```

For function fields, compute method serves *both* purposes.

Onchange methods

Generic method

- take all form values as input
- automatically handle function fields
- overridden to implement specific triggers

```
@one
def onchange(self, field_name):
    super(C, self).onchange(field_name)
    if field_name == 'partner':
        self.delivery_address = self.partner
```

Currently under development.

Method decorators: **@constraint**

```
@one
@constraint("Name and description must be different")
@depends('name', 'description')
def _check_name_desc(self):
    return self.name != self.description
```

Currently under development.

New domain syntax(es)

AST to represent the logic expression.

```
domain = AND(NE('name', 'Foo'), LT('age', 18))
```

Alternative construction with criteria (helper).

```
c = model.criteria()
domain = (c.name != 'Foo') & (c.age < 18)
```

Textual representation.

```
domain = "name != 'Foo' and age < 18"
```

This is speculative: maybe not for OpenERP 8.0.

It's a team work:

Raphael Collet, Vo Minh Thu, Xavier Morel

Work in progress

Server and addons branches on launchpad:

lp:~openerp-dev/openobject-server/trunk-apiculture

lp:~openerp-dev/openobject-addons/trunk-apiculture

Feedback: rco@openerp.com