MK.Lab

Основы бизнес-логики

Объект self.env

Вы уже, скорее всего, задумывались, что нужно как-то обращаться к объектам системы.
Разумеется, наш текущий объект — это self.

Вы можете обратиться к любому полю или методу:

self.name=result+B

или

return self.check_my_constraint()

А как обратиться к другим объектам? Как перебирать записи?
Для этого и нужен объект **self.env**. Он содержит:

1. Справочник всех моделей. Можно обратиться к любой модели и любому полю.
2. Объект «контекст». Это хранилище временной информации. В контексте есть служебная часть, которая заполняется системой. Вообще про контекст довольно мало информации, я собрал по крупинкам вот что:

Active_id содержит идентификатор открытой записи.
Active_ids - список активных записей, выбранных галочкой. Работает, если Вы вызываете действие или визард, отметив в списке галочками записи.

default_имя_поля — задаст значение поля по умолчанию. Т.е., если Вы вызовете действие, которое вызовет форму, предварительно заполнив контекст значениями по умолчанию, то форма при открытии уже будет содержать в полях нужные данные.

Контекст даже можно задавать прямо в форме:

<field name="mt_contractid" domain="[('partner_id','=',partner_id)]"
context="{'sale_order_id': active_id}"/>

В данном случае придуманная переменная sale_order_id будет содержать значение идентификатора модели. Эту переменную можно будет считать в другой форме.
Т.е. контекст — это в том числе средство передачи данных от формы к форме на клиенте.

<field name="details_ids" context="{'default_order_id': active_id}">

А вот здесь явно задается значение поля order_id в открываемой новой форме.
Не забывайте, что можно делать скрытые от пользователя поля, которые можно использовать для любых целей.

3. Объект user. Содержит объект типа res.users. Пользователь, из-под которого идет работа.
Нужно для разных задач, начиная от фильтровки данных, заканчивая правами.
Вы можете определить идентификатор пользователя, по нему найти, есть ли он в какой-нибудь группе доступа и дальше уже работать по своему сценарию.

Чтобы создать объект, связанный с определенной моделью:

doctor_obj=self.env['medical.physician']

Но это только объект, он не содержит никаких данных, т.е., не является экземпляром.
Такая конструкция нужна для каких-либо дальнейших действий.

Если Вам нужно получить список всех записей, удовлетворяющих вашему критерию:

doctors=self.env['medical.physician'].search([('fired', '=', False)])

В скобках мы указали домен поиска. Вернитесь к разделу про домены и освежите в памяти.

Если Вам нужно получить доступ к одной записи:
doctors=self.env['medical.physician'].browse(<ID>)

Основные паттерны программирования

Кроме обычной логики, к которой Вы привыкли, при работе с Odoo программисты часто применяют конструкции:

Повсеместная проверка на непустоту списков

doctors=self.env['medical.physician'].search([('fired', '=', False)])

```
if doctors:
  что-то делаем
```

Предположение, что объектов у нас много

```
def check_my()
   for s in self:
     if s.float_field>0:
       и т.д.
```

Это полезная штука, когда мы пишем метод, который потом хотим вызывать не только для одной записи, а для группы записей. Например, массовое проведение счетов в базу.

Проверка поля на пустоту

Во многих местах исходного кода мы видим такие конструкции:
self.tax_id=A or False
Это означает, что в любом поле может оказаться False.
А это означает, что будет ошибка, если мы напишем так:

```
result="Начинаем парад-карнавал "+self.carnaval_name
```

А поле carnaval_name окажется пусто.

Механизмы бизнес-логики

В Odoo предусмотрены следующие механизмы:

1. Onchange
2. Проверка перед сохранением.
3. Произвольный метод класса.
4. Перекрытие метода класса-родителя.
5. Исключения
6. Вычисляемые поля
7. Значения полей по умолчанию

Перед началом работы добавьте в свой .py файл импорт следующих модулей:
```
from odoo import api, fields, models, exceptions
```

Onchange — событие изменения значения поля

Исполняется на сервере. В двух словах, как это работает.
Мы знаем, что наш фронтенд — это Java Script приложение. Оно исполняется в вашем браузере и обменивается с сервером odoo данными в формате JSON RPC.

Когда пользователь меняет значение какого-то поля, веб-приложение отправляет серверу запрос, что значение поля изменилось. Более точный механизм Вы можете

узнать, изучив исходники. Например, известно ли веб-приложению, для каких полей Вы задали обработчики, а для каких нет, чтобы не делать лишних запросов.

Далее сервер ищет обработчик события изменения значения, исполняет его, получает ответ и отправляет в веб-приложение.
Результатом всегда будет набор новых значений полей или сообщение об ошибке.

Как работает?
Мы должны написать декоратор

@api.onchange, который свяжет наш обработчик с общим обработчиком по умолчанию.

Например:

```
    @api.onchange('partner_id')
    def get_contact(self):
        if self.partner_id:
            contr=self.env['partner.contract.customer'].search([('partner_id', '=', self.partner_id.id)])
            if contr:
                sel=contr.sorted(key=lambda r: r.date_start, reverse=True)
                if sel:
                    self.mt_contractid=sel[0]
```

В этом примере **примечательно всё**.
Давайте разберем.
Итак, после декоратора в скобках мы пишем имена полей, при изменении которых будет вызываться данный метод.
Далее в домене Вы видите хитрую конструкцию:

('partner_id', '=', self.partner_id.id)

Спрашивается, почему слева без id, а справа с id? А вот так. Домен сформирует условие Where для SQL запроса, поэтому нужно прописывать простое значение типа integer, float, char, а не res.partner. Напомним, что *self.partner_id это ссылка на res.partner.*

Вы уже обратили внимание на паттерн с проверкой на существование. Всегда нужно предполагать, что чего-то может не быть, что-то может иметь значение False.

На словах можно себе всегда говорить так:
Давайте поищем записи, для которых то-то и то-то.
Если нашли, то делаем так-то.

Идем дальше:
sel=contr.sorted(key=lambda r: r.date_start, reverse=True)

Метод sorted делает тоже самое, что sort() в Python. Про лямбда функции Вы уже, скорее всего, читали. В этом примере лямбда функция просто выдает значение поля date_start.

Сортируем наш список по ключу (ключ в Python это функция, а не скаляр): дата начала, с реверсом. Таким образом самый свежий в данном случае договор будет первым в списке.

В поле self.mt_contractid подставляется первое значение в списке:

self.mt_contractid=sel[0]

Заметьте, что наш метод ничего не возвращает сам. Почему? А потому что на самом деле реальный метод, который мы декорировали, вернет все данные сам и сделает return сам.

Проверка перед сохранением

Такую проверку часто называют проверкой целостности. Аналогично onchange пишем декоратор:

```
@api.constrains('year_of_birth')
def year_of_birth_check(self):
    if self.year_of_birth<1920 or self.year_of_birth>2030:
        raise exceptions.ValidationError('Некорректно заполнен год рождения')
```

Здесь новинка — исключение.
raise exceptions.ValidationError('Некорректно заполнен год рождения')

Если мы выбрасываем исключение, то вся транзакция будет отменена.
Если Вы в цикле что-то обрабатываете, а для какого-то элемента выбросите исключение, то вся большая транзакция откатится.
Не забывайте про это в групповых операциях.

Опытный читатель уже сразу отметил для себя крайне полезным raise exception для отладки.
Чтобы не морочиться с логом, можно выбрасывать исключение с нужным нам значением.
Лог — хорошая идея, консоль — правильная. Но exception быстрая.

Вы можете сами обработать исключение, не выбрасывая пользовательское. Например так:

```
url='https://URL/my/invoices/'+str(inv.id)+'?access_token='+inv.access_token
                try:
                  fetcher = urllib.request.urlopen('https://clck.ru/--?url='+ url)
                  url=fetcher.read().decode('utf-8')
                except:
                  url=url
```

В этом кусочке я делаю сокращение ссылки на быструю оплату счета. Я предположил, что по какой-то причине может не сработать сервис укорачивания ссылки. Если будет ошибка, то выдадим полную ссылку без укорачивания. Это всегда лучше, чем остаться с ошибкой.

MK.Lab

Произвольный метод класса

Вы можете определить любой метод и как обычно его вызывать везде, где надо.

```
def move_debit(self):
  for s in self:
   if s.reconcile_summ<0:
     sum1=-s.reconcile_summ
     move_obj=self.env['account.move']
     move_line_obj=self.env['account.move.line']
     и т. д.
```

Разумеется, этот метод можно вызывать программно, можно добавить в «Действия», а можно повесить на кнопку. Ниже жирным выделено имя кнопки. Оно должно совпадать с именем метода.

<button name="**action_supp_invoice_create**" type="object" string="Начисления подрядчикам"/>

Перекрытие методов

Это довольно интересный метод.
Есть штатные методы write, create, в работу которых можно вмешаться и что-то доделать при записи или при создании. Например, сделать номер для документа.
Сначала было объявление класса:

```
class medical_appointment(models.Model):
  _name = "medical.appointment"
  _inherit = 'mail.thread'
```

потом были поля

```
@api.model
def create(self, vals):
   vals['name'] = self.env['ir.sequence'].next_by_code('medical.appointment') or 'APT'
   msg_body = 'Appointment created'
   self.message_post(body=msg_body)
   result = super(medical_appointment, self).create(vals)
   return result
```

Как всегда, пример примечателен всем.

Сначала мы написали декоратор @api.model. Это означает, что метод будет общим, его нельзя будет вызывать для каждого конкретного экземпляра. Это метод модели.

Затем мы сделали метод create, который называется точно также как штатный.
У метода есть переменная vals. Это словарь, в котором будут все значения, подлежащие записи в базу.

vals['name'] = self.env['ir.sequence'].next_by_code('medical.appointment') or 'APT'

Вмешиваемся и прописываем свое значение для номера. Вызываем штатный нумератор.
Дальше мы хотим в наш лог, который внизу основного экрана, там где появляются сообщения, написать, что объект создан. Для истории.

Именно для этого мы при создании модели сделали наследование:
_inherit = 'mail.thread'

Эта штука добавляет весь функционал модуля «Общение» в нашу модель. Для полного счастья нужно еще в форме после </sheet> добавить   с функционалом. Посмотрите в sale.order, как это сделано.

self.message_post(body=msg_body)

Вызываем метод message_post и наше сообщение остается в логе.
Можно даже отправить что-то по почте этим методом. Достаточно задать нужный параметр.
Например так:

*s.message_post(body='Здравствуйте, Ваша ссылка на предоплату за прием: '+url, subject='Счет на оплату от клиники МНС', **message_type='comment'**)*

Не забываем, что мы изначально хотели перекрыть метод создания, но ничего не сделали для этого, поэтому:

```
result = super(medical_appointment, self).create(vals)
        return result
```

Вызываем super для нашего класса *medical_appointment.*
Дальше вызовется штатный метод create, но уже с поправленными нами значениями vals.

Точно также мы можем забраться в метод write, который записывает значения в базу.

Важный момент. Внутри метода **write** мы можем получить доступ как к новым значениям, так и к старым. И написать что-то типа:
*message_body='Изменение времени приема: '+**self.appointment_date**+ '+3GMT -> '+str(**vals['appointment_date']**)*

В self у нас старые значения, в vals новые.

Вычисляемые поля

Я отнес вычисляемые поля в бизнес-логику, а не в создание моделей, потому что это и есть бизнес-логика. Что-то как-то считается и показывается.
Python работает довольно шустро, поэтому можно делать вычисляемые поля, которые могут целиком содержать, например, весь текст для sms, отправляемому клиенту.

MK.Lab

Вычисляемые поля вычисляются только тогда, когда к ним есть обращение. Т.е., если у Вас поля нет на форме, нет в списке, то алгоритм вычисления не будет вызываться каждый раз для каждой записи, вызывая торможение.

Итак, пишем:

```
class appointmentcancel(models.Model):
   _inherit='medical.appointment'
   need_prepay=fields.Boolean(string='Нужна предоплата', compute='get_prepay')
```

И потом в этом классе должны написать:

```
   def get_prepay(self):
      for s in self:
         if s.room and s.consultations_id:
            s.need_prepay=s.room.need_prepay or s.consultations_id.to_weight
```

Оцените логическое выражение, выделенное жирным. Теперь можно писать такое в значениях.
Результатом работы метода, который вычисляет вычисляемое поле, является не возврат значения для присвоения, а явным образом задание этого значения в результате работы алгоритма:

```
 s.need_prepay=s.room.need_prepay or s.consultations_id.to_weight
```

Значения полей по умолчанию

Иногда бывает полезно подставить что-то в поле, например, текущего пользователя, или текущую дату.

Пишем:
appointment_date = fields.Datetime('Appointment Date',required=True, **default = fields.Datetime.now**)

Т.е,. мы видим, что вызвана функция.
Мы можем вызывать и метод своего класса, только нужно определить его выше нашего поля.

```
class Contract(models.Model):
   _name = 'contract.contract'
   _inherit = ['mail.thread', 'mail.activity.mixin']

   def get_dateend(self):
      if self.date_start:
         six_months = fields.Datetime.from_string(self.date_start) + relativedelta(months=+11)
      else:
         six_months = datetime.today() + relativedelta(months=+11)
      return fields.Datetime.to_string(six_months)

   name = fields.Char(string='Номер')
```

date_start = fields.Date(string='Дата договора', required=True, default=fields.Datetime.now())

date_end = fields.Date(string='Дата окончания', required=True, ***default=get_dateend****)*

Очевидно, что в этом кусочке мы пытаемся прибавить 11 месяцев к дате договора. В отличие от вычисляемых полей, где метод может ничего и не возвращать, в случае значения по умолчанию метод обязан что-то вернуть.

Задание

Сделайте для своего модуля или для «СНТ Рассвет» обработки:
1. Изменение значения поля, onchange.
2. Проверку целостности, например, чтобы название собрания больше не повторялось бы.
3. Значение поля по умолчанию.
4. Вычисляемое поле, например, сколько было участников.
5. Кнопку, вызывающую метод. Например, если кол-во участников больше 10, то пусть выводится фраза «Кворум».
6. Добавите контекст для вопросов собрания.
7. Добавьте домен в поле «протокол»