

UD06. DESARROLLO DE MÓDULOS. MODELOS

Sistemas de Gestión Empresarial

2 Curso // CFGS DAM // Informática y Comunicaciones

Alfredo Oltra

**Cicles
Formatius**

ÍNDIX

1 INTRODUCCIÓN	4
1.1 Atributos tipo field simples	6
1.2 Atributos <i>fields</i> relacionales	7
1.3 Atributos <i>fields</i> calculados (<i>computed</i>)	12
1.4 Valores por defecto	13
1.5 Restricciones (constraints)	14
2 BIBLIOGRAFIA	15
3 AUTORES	15

Versión: 231210.1713

Licencia




Reconocimiento – NoComercial – CompartirIgual (by-nc-sa). No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 **Atención.** Importante prestar atención a esta información.

 **Interesante.** Ofrece información sobre algún detalle a tener en cuenta.

1 INTRODUCCIÓN

Los modelos son una abstracción propia de muchos frameworks y relacionada con el *ORM*. Un modelo se define como una clase Python que hereda de la clase `models.Model`. Al heredar de esta clase, adquiere unas propiedades de forma transparente para el programador. A partir de este momento, las clases del lenguaje de programación quedan por debajo de un nivel más de abstracción.

Una clase heredada de `models.Model` se comporta de la siguiente manera:

- Puede ser accedida como modelo, como *recordset* (conjunto de registros) o como *singleton* (un único registro). Si es accedida como modelo, tiene métodos de modelo para, por ejemplo, crear *recordsets*. Si es accedida como *recordset*, se puede acceder a los datos que guarda.
- Puede tener atributos internos de la clase, ya que sigue siendo Python. Pero los atributos que se guardan en la base de datos se han de definir cómo *fields*. Un *field* es una instancia de la clase `fields.Field`, y tiene sus propios atributos y funciones.
Odoo analizará el modelo, buscará los atributos tipo *field* y sus propiedades y mapeará automáticamente todo esto en el *ORM*.
- Los métodos definidos para los *recordset* reciben un argumento llamado *self* que es un *recordset* con una colección de registros. Por tanto, deben iterar en el *self* para hacer su función en cada uno de los registros.
- Los modelos tienen sus propias funciones para no tener que acceder a la base de datos para modificar o crear registros. Además, incorporan restricciones de integridad.

Este es el ejemplo de un modelo con solamente un *field*:

```
class AModel(models.Model):
    _name = 'a.model'
    _description = 'descripción opcional'
    name = fields.Char(
        string="Name",          # El nombre en el label (Opcional)
        compute="_compute_name_custom", # En caso de ser computado, el nombre
        # de la función
        store=True,             # En caso de ser computado, si se guarda o no
        select=True,            # Forzar que esté indexado
        default='Nombre',       # Valor por defecto, puede ser una función
        readonly=True,          # El usuario no puede escribir directamente
        inverse="_write_name"    # En caso de ser computada y se modifique
        required=True,          # Field obligatorio
        translate=True,         # Si se puede traducir
        help='blabla',          # Ayuda al usuario
        company_dependent=True, # Transforma columna a ir.property
        search='_search_function', # En caso de ser computado, cómo buscar en él.
        copy =True              # Si se puede copiar con copy()
    )
```

Sobre el código anterior, veamos en detalle todo lo que pasa:

- Se define una clase de Python que hereda de `models.Model`
- Se definen dos atributos `_name` y `_description`. El `_name` es obligatorio en los modelos y es el nombre del modelo. Aquí se observa la abstracción, ya no se accederá a la clase `Amodel`, sino al modelo `a.model`.
- Luego está la definición de otro atributo tipo *field* que será mapeado por el ORM en la base de datos. Como se puede observar, llama al constructor de la clase `fields.CharField` con unos argumentos. Todos los argumentos son opcionales en el caso de *Char*. Hay constructores para todos los tipos de datos.



Es muy probable que a estas alturas no entiendas el porqué de la mayoría del código anterior. Los frameworks requieren entender muchas cosas antes de poder empezar. No obstante, con ese fragmento de código ya tenemos solucionado el almacenamiento en la base de datos, la integridad de los datos y parte de la interacción con el usuario. .



Odoo está pensado para que sea fácilmente modificable por la web. Sin necesidad de entrar al código. Esto es muy útil, por ejemplo, para prototipar las vistas.

Una de las funcionalidades es el modo desarrollador, que permite, entre otras muchas cosas, explorar los modelos que tiene en este momento el servidor.

Los modelos tienen algunos atributos del modelo, como `_name` o `_description`. Otro atributo de modelo importante es `_rec_name` que indica de que atributo toma nombre el registro y que por defecto apunta al atributo `name` (no confundir con `_name`).

- En las vistas (que veremos más adelante), en algunos campos se basa en el atributo marcado por `_rec_name`, que por defecto es `name`. Si no tenemos un atributo `name` o queremos que sea otro el que de nombre, podemos modificarlo con `_rec_name='nombreatributo'`

1.1 Atributos tipo field simples

Los modelos tienen otros atributos tipo field, que se mapean en la base de datos y a los que el usuario tiene acceso y métodos que conforman el controlador. A continuación vamos a detallar todos los tipos de field que hay y sus posibilidades.

En primer lugar, definimos los *fields* de datos más habituales:

- Integer
- Char
- Text
- Date
- Datetime
- Float
- Boolean
- Html
- Binary: archivos binarios que guarda en el formato *base64*. Pueden guardarse imágenes u otros elementos. Antes de Odoo 13 en este tipo de *fields* se guardaban las imágenes.

Dentro de los *fields* de datos, hay algunos un poco más complejos:

- Image: a partir de la versión 13 de Odoo se pueden guardar imágenes en este *field*. Hay que definir el *max_width* o *max_height* y se redimensionará al guardar.
- Selection: guarda un dato, pero hay que decirle con una lista de tuplas las opciones que tiene.

```
type = fields.Selection([('1', 'Basic'), ('2', 'Intermediate'), ('3', 'Completed')])
aselection = fields.Selection(selection='a_function_name') # se puede definir
su contenido en una función.
```

Todos los *fields* mencionados tienen un constructor que funciona de la misma manera que en el ejemplo anterior. Pueden tener un nombre, un valor por defecto, o incluso puede definirse su contenido mediante una función.

A lo largo de este texto se verán ejemplos de cómo se han definido *fields* según las necesidades.

1.2 Atributos *fields* relacionales

A continuación, vamos a observar los *fields* relacionales. Dado que el ORM evita que tengamos que crear las tablas y sus relaciones en la base de datos, cuando existen relaciones entre modelos se necesitan unos campos que definan esas relaciones.

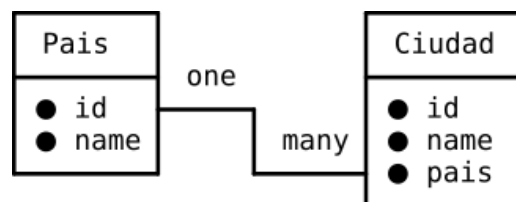
Por ejemplo, un pedido de venta tiene un cliente y un cliente puede hacer muchos pedidos de venta. A su vez, ese pedido tiene muchas líneas de pedido, que son solo de ese pedido y tienen un producto, que puede estar en muchas líneas de venta.

En situaciones como la del ejemplo, estas relaciones acaban estando en la base de datos con claves ajenas. Pero con los frameworks que implementan ORM, todo esto es mucho más sencillo.

Para ello utilizaremos los *fields* relacionales de *Odoo*:

- **Many2one**: es el más simple. Indica que el modelo en el que está tiene una relación muchos a uno con otro modelo. Esto significa que un registro tiene relación con un único registro del otro modelo, mientras que el otro registro puede tener relación con muchos registros del modelo que tiene el *Many2one*. En la tabla de la base de datos, esto se traducirá en una clave ajena a la otra tabla.

Ejemplo donde se pretende que cada ciudad almacene su país.



```

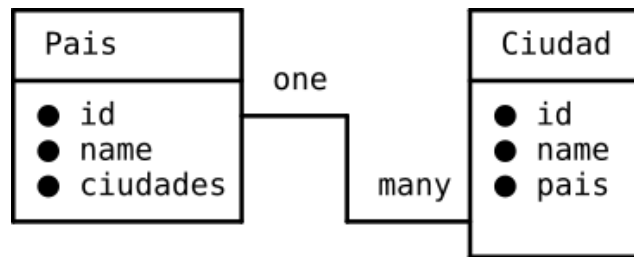
pais_id = fields.Many2one('modulo.pais') # La forma más común, en el Modelo Ciudad
pais_id = fields.Many2one(comodel_name='modulo.pais') # Otra forma, con argumento
  
```



Al principio puede parecer contra intuitivo el nombre de *Many2one* con el tipo de relación. Reflexiona sobre este diagrama y haz otras pruebas para acostumbrarte a este tipo de relación.

- **One2many**: La inversa del *Many2one*. De hecho, necesita que exista un *Many2one* en el otro modelo relacionado. Este *field* no supone ningún cambio en la base de datos, ya que es el equivalente a hacer un 'SELECT' sobre las claves ajenas de la otra tabla. El *One2many* se comporta como un campo calculado cada vez que se va a ver.

Ejemplo donde se pretende que cada país pueda acceder a sus ciudades.

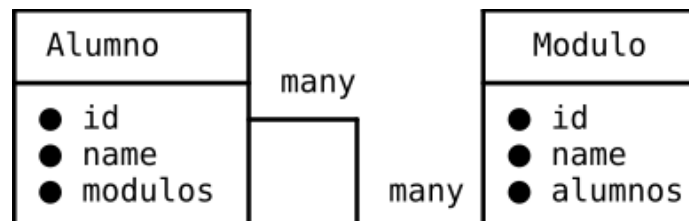


```

pais_id = fields.Many2one('modulo.pais') # Esto en el modelo Ciudad, indica un
many2one
ciudades_ids = field.One2many('modulo.ciudad', 'pais_id') # En el modelo Pais.
# El nombre del modelo y el field que tiene el Many2one necesario para que
funcione.
  
```

- **Many2many:** Se trata de una relación muchos a muchos. Esto se acaba mapeando como una tabla intermedia con claves ajenas a las dos tablas. Hacer los *Many2many* simplifica mucho la gestión de estas tablas intermedias y evita redundancias o errores. La mayoría de los *Many2many* son muy fáciles de gestionar, pero algunos necesitan conocer realmente qué ha pasado en el ORM.

Ejemplo que indica que un alumno puede tener muchos módulos y un módulo puede tener muchos alumnos.

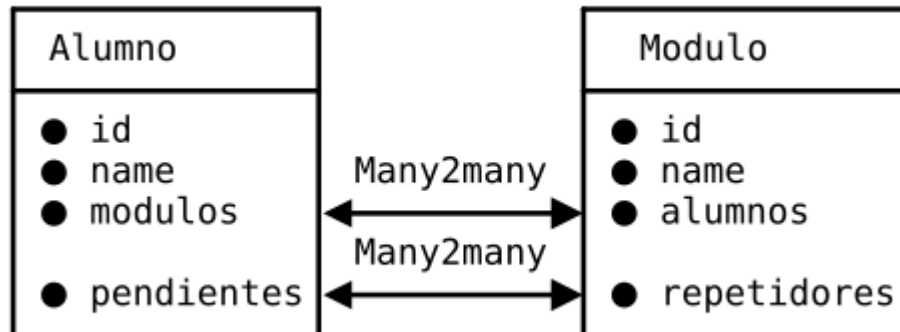


```

modulos_ids = fields.Many2many('modulo.modulo') # Esto en el modelo Alumno
alumnos_ids = field.Many2many('modulo.alumno') # Esto en el modelo Modulo.
  
```

En el ejemplo anterior, Odoon interpretará que estos dos *Many2many* corresponden a la misma relación y creará una tabla intermedia con un nombre generado a partir del nombre de los dos modelos. No obstante, no tenemos control sobre la tabla intermedia.

Puede que en otros contextos, necesitemos tener dos relaciones *Many2many* independientes sobre dos mismos modelos. Observemos este diagrama:



Existen dos relaciones *Many2many*:

- Las de *alumnos con módulos*, descrita en el ejemplo anterior.
- Una nueva relación, donde se relacionan alumnos repetidores con módulos pendientes.

No deben coincidir, pero si no se especifica una tabla intermedia diferente, *Odoo* considerará que es la misma relación. En estos casos hay que especificar la tabla intermedia con la sintaxis completa para evitar errores:

```

alumnos_ids = fields.Many2many(comodel_name='modulo.alumno',
    relation='modulos_alumnos', # El nombre de la tabla intermedia
    column1='modulo_id', # El nombre en la tabla intermedia de la clave a este modelo
    column2='alumno_id') # El nombre de la clave al otro modelo.
repetidores_ids = fields.Many2many(comodel_name='modulo.alumno',
    relation='modulos_alumnos_repetidores', # El nombre de la tabla intermedia
    column1='modulo_id', # El nombre en la tabla intermedia de la clave a este modelo
    column2='alumno_id') # El nombre de la clave al otro modelo.
modulos_ids = field.Many2many(comodel_name='modulo.modulo',
    relation='modulos_alumnos', # El nombre de la tabla intermedia
    column1='alumno_id', # El nombre en la tabla intermedia de la clave a este modelo
    column2='modulo_id') # El nombre de la clave al otro modelo.
pendientes_ids = field.Many2many(comodel_name='modulo.modulo',
    relation='modulos_alumnos_repetidores', # El nombre de la tabla intermedia
    column1='alumno_id', # El nombre en la tabla intermedia de la clave a este modelo
    column2='modulo_id') # El nombre de la clave al otro modelo.
  
```

Las relaciones *Many2one*, *One2many* y *Many2many* suponen la mayoría de las relaciones necesarias en cualquier programa. Hay otro tipo de *fields* relacionales especiales que facilitan la programación:

- **related**: En realidad no es un tipo de *field*, sino una posible propiedad de cualquiera de los tipos. Lo que hace un *field related* es mostrar un dato que está en un registro de otro modelo con el cual se tiene una relación *Many2one*.

Si tomamos como ejemplo el anterior de las ciudades y países, imaginemos que queremos mostrar la bandera del país en el que está la ciudad. La bandera será un campo *Image* que estará en el modelo país, pero lo queremos mostrar también en el modelo *ciudad*. Para ello tenemos dos posibles soluciones:

- La solución mala sería guardar la bandera en cada ciudad.
- La buena solución es usar un *field related* para acceder a la bandera.

```
pais_id = fields.Many2one('modulo.pais') # Esto en el modelo Ciudad
bandera = fields.Image(related='pais_id.bandera') # Suponiendo que existe el
field bandera y es de tipo Image.
```



El *field related* puede incluir también el valor `store=True` en caso de que queramos que se guarde en la base de datos. En la mayoría de casos es redundante y no sirve, pero, puede que por razones de rendimiento, o para poder buscar sobre él, se deba guardar. Esto no respeta la tercera forma normal. En ese caso, *Odoo* se encarga de mantener la coherencia de los datos.



Otro uso posible de los *field related* puede ser hacer referencia a *fields* del propio modelo para tener los datos repetidos. Esto es muy útil en las imágenes para, por ejemplo, almacenar versiones con distintas resoluciones. También puede ser útil para mostrar los mismos *fields* con diferentes representaciones gráficas (diferentes *widgets*).

- **Reference:** es una referencia a un campo arbitrario de un modelo. En realidad no provoca una relación en la base de datos. Lo que guarda es el nombre del modelo y del campo en un *field char*.
- **Many2oneReference:** es un *Many2one* pero en el que también hay que indicar el modelo al que hace referencia. No son muy utilizados.



En algunas ocasiones, influidos por el pensamiento de las bases de datos relacionales, podemos decidir que necesitamos una relación *One2one*. *Odoo* dejó de usarlas por motivos de rendimiento y recomienda en su lugar unir los dos modelos en uno. No obstante, se puede imitar con dos *Many2many* computados o un *One2many* limitado a un solo registro. En los dos casos, será tarea del programador garantizar el buen funcionamiento de esa relación.

Una vez estudiado el concepto de modelo y de los *fields*, detengámonos un momento a analizar este código que define 2 modelos:

```
# -*- coding: utf-8 -*-
from odoo import models, fields, api
from openerp.exceptions import ValidationError
#####
class net(models.Model):
    _name = 'networks.net'
    _description = 'Networks Model'
    name = fields.Char()
    net_ip = fields.Char()
    mask = fields.Integer()
    net_map = fields.image()
    net_class = fields.Selection([('a', 'A'), ('b', 'B'), ('c', 'C')])
    pcs = fields.One2many('networks.pc', 'net')
    servers = fields.Many2many('networks.pc', relation='net_servers')

class pc(models.Model):
    _name = 'networks.pc'
    _description = 'PCs Model'
    name = fields.Char(default="PC")
    number = fields.Integer()
    ip = fields.Char()
    ping = fields.Float()
    registered = fields.Date()
    uptime = fields.Datetime()
    net = fields.Many2one('networks.net')
    user = fields.Many2one('res.partner')
    servers = fields.Many2many('networks.net', relation='net_servers')
```

Como se puede ver, están casi todos los tipos básicos de *field*. También podemos ver *fields* relacionales. Prestemos atención al *Many2one* llamado *net* de los *PC* que permite que funcione el *One2many* llamado *pcs* del modelo *networks.net*. También son interesantes los *Many2many* en los que declaramos el nombre de la relación para controlar el nombre de la tabla intermedia.

Una vez repasados los tipos de *fields* y visto un ejemplo, ya podríamos hacer un módulo con datos estáticos y relaciones entre los modelos. Nos faltaría la vista para poder ver estos modelos en el cliente web. Puedes pasar directamente a la parte 3 de la documentación, centrada sobre las vistas si quieres tener un módulo funcional mínimo, pero en el modelo quedan algunas cosas importantes que explicar.

1.3 Atributos *fields* calculados (*computed*)

Hasta ahora, los *fields* que hemos visto almacenaban algo en la base de datos. No obstante, puede que no queramos que algunos datos estén guardados en la base de datos, sino que se recalculen cada vez que vamos a verlos. En ese caso, hay que utilizar campos computados.

Un *field computed* se define igual que uno normal, pero entre sus argumentos hay que indicar el nombre de la función que lo computa:

```
taken_seats = fields.Float(string="Taken seats", compute='_taken_seats')

# El decorador @api.depends() indica que se llama a
# la función cada vez que cambie el valor de los fields seats y attendee_ids.
@api.depends('seats', 'attendee_ids')
def _taken_seats(self):
    for r in self: # El for recorre self, que es un recordset con los registros activos
        if not r.seats: # r es un singleton. Se puede acceder a sus atributos como un
            objeto
            r.taken_seats = 0.0 # esta asignación ya hace que se vea el resultado.
        else:
            r.taken_seats = 100.0 * len(r.attendee_ids) / r.seats
```

Los *field computed* no se guardan en la base de datos, pero en algunas ocasiones puede que necesitemos que se guarde (por ejemplo, para buscar sobre ellos). En ese caso podemos usar `store=True`. Esto es peligroso, ya que puede que no recalcula más ese campo. El decorador `@api.depends()` soluciona ese problema si el *computed* depende de otro *field*.

En caso de no querer guardar en la base de datos, pero si querer buscar en el campo, *Odoo* proporciona la función *search*. Esta función se ejecuta cuando se intenta buscar por ese campo. Esta función retornará un dominio de búsqueda (esto se explicará más adelante). El problema es que tampoco puede ser un dominio muy complejo y limita la búsqueda.



Hay un truco para poder tener *fields computed* con `store=True` y a la vez poder buscar u ordenar. Lo que se puede hacer es otro *field* del mismo tipo que no sea *computed*, pero que se sobrescriba cuando se ejecuta el método del que sí es *computed*. De esta manera, se guarda en la base de datos, aunque se recalcula cada vez. El problema es que deben estar los dos *fields* en la vista. Esto se soluciona poniendo `invisible='1'` en el *field computed*. El usuario no lo observa, sin embargo, *Odoo* lo recalcula

En pocas ocasiones necesitamos escribir directamente en un *field computed*. Si es *computed* será porque su valor depende de otros factores. Si se permitiera escribir en un *field computed*, no sería coherente con los *fields* de los que depende. No obstante sí que podemos permitir que se escriba directamente si hacemos uso de la función *inverse*, la cual ha de sobrescribir los *fields* de los que depende el *computed* para que el cálculo sea el que introduce el usuario.

1.4 Valores por defecto

En muchas ocasiones, para facilitar el trabajo a los usuarios, algunos *fields* pueden tener un valor por defecto. Este valor por defecto puede ser siempre el mismo o ser computado en el momento en que se inicia el formulario. A diferencia de los *fields computed*, se permite por defecto que el usuario lo modifique y no depende de lo que el usuario va introduciendo en otros *fields*.

Para que un *field* tenga el valor por defecto, hay que poner en su constructor el argumento *default=*. En caso de ser un valor estático, solo hay que poner el valor. En caso de ser un valor por defecto calculado, se puede poner la función que lo calcula.

```
name = fields.Char(default="Unknown")
user_id = fields.Many2one('res.users', default=lambda self: self.env.user)
start_date = fields.Date(default=lambda self: fields.Date.today())
active = fields.Boolean(default=True)

def compute_default_value(self):
    return self.get_value()

a_field = fields.Char(default=compute_default_value)
```

En este ejemplo:

- En la primera línea se ve la manera más fácil de asignar un valor por defecto, una simple cadena de caracteres para el *field Char*.
- En la segunda línea se usa una función *lambda* que obtiene el usuario.
- En la tercera una que obtiene la fecha.
- En la cuarta va un *default* simple que asigna un valor booleano.
- Por último, tenemos una función que luego es referenciada en el *default* del último *field*.

Recordamos que las funciones *lambda* son funciones anónimas definidas en el lugar donde se van a invocar. No pueden tener más de una línea.

La segunda cosa interesante es la definición de la función antes de ser invocada. *Python* es un lenguaje interpretado y no puede invocar funciones que no se han definido previamente de ser invocadas. En el caso de los *fields computed*, esto no tiene importancia porque se invocan cuando el usuario necesita ver el *field*. Pero los *default* se ejecutan cuando se reinicia el servidor *Odoo* e interpreta todos los ficheros *Python*.

Este efecto se puede ver de forma clara en la función *lambda* que calcula la hora. Si en vez de lo que hay en el ejemplo, pusiéramos directamente la función *fields.Date.today()*, pondría la fecha de reinicio del servidor y no la fecha de creación del registro. En cambio, al referenciar a una función *lambda*, esta se ejecuta cada vez que el programa entra en esa referencia. Al igual que al referenciar a una función normal con nombre.



No hay mejor manera de aprender que probar las cosas. Crea un modelo nuevo con valores por defecto y prueba a poner la fecha solo con la función de fecha y, luego, dentro de la función *lambda* para comprobar que pasa .

1.5 Restricciones (*constraints*)

No en todos los *fields* los usuarios pueden poner de todo. Por ejemplo, podemos necesitar limitar el precio de un producto en función de unos límites preestablecidos. Si el usuario crea un nuevo producto y se pasa al poner el precio, no debe dejarle guardar. Las restricciones se consiguen con un decorador llamado `@api.constraint()` el cual ejecutará una función que revisará si el usuario ha introducido correctamente los datos. Veamos un ejemplo:

```
from odoo.exceptions import ValidationError
...
@api.constrains('age')
def _check_age(self):
    for record in self:
        if record.age > 20:
            raise ValidationError("Your record is too old: %s" % record.age)
```

Se supone que hay un *field* llamado *age*. Cuando es modificado y se intenta guardar se llama a la función que tiene el decorador. Esta recorre el *recordset* (recordemos que es importante poner siempre el `for record in self`). Para cada *singleton* compara la edad y si alguna es mayor de 20, lanza un error de validación. Este lanzamiento impide que se guarde en la base de datos y avisa al usuario de su error.

En ocasiones es más cómodo poner una restricción SQL que hacer el algoritmo que lo comprueba. Además de más eficiente en términos de computación. Para ello podemos usar una `_sql_constraints`:

```
_sql_constraints = [
    ('name_uniq', 'unique(name)', 'Custom Warning Message'),
    ('contact_uniq', 'unique(contact)', 'Custom Warning Message')
]
```

Las SQL constraints son una lista de tuplas en las que está el nombre de la restricción, la restricción SQL y el mensaje en caso de fallo.

2 BIBLIOGRAFIA

1. Documentación de Odoo

3 AUTORES

A continuación ofrecemos en orden alfabético (por apellido) el listado de autores que han hecho aportaciones a este documento.

- Jose Castillo Aliaga
- Sergi García Barea
- Alfredo Oltra